Contents

1 Foreword

by Zach Lieberman[1]

Openframeworks began around 2004 / 2005. I was teaching at Parsons School of Design, and at the same time, making a living as an artist creating interactive projects with code.

I had graduated a few years earlier from grad school in 2002, and we were using computational tools like Director / Lingo and Flash when I was there – it was the heyday of flash experimentation. In 2002, my professor, Golan Levin, invited me to collaborate with him on some projects after I graduated from school and he introduced me to ACU, a C++ library that had been developed at MIT under the Aesthetics and Computation Group, the group that John Maeda ran which had students like Elise Co, Peter Cho, Ben Fry, Casey Reas and Golan himself.

ACU as a library was great, but quite oriented to late 90s tech such as SGI machines. It was also not open source and not actively maintained. Folks like Ben Fry had fortuitously moved on to developing Processing and it seemed like it wasn't going to last us a long time for our work. It seemed like we would need an alternative codebase moving forward.

In addition, I really wanted to share the code I was writing in my professional practice with my students. I remember vividly having conversations with the administration of the department at Parsons where they said, "Art students don't want to learn C++." I started teaching classes with OF in 2005, and I had students like Theo Watson, Christine Sugrue, and Evan Roth who created remarkably wild, experimental work with OF and that argument crumbled rapidly. C++ provided low level access to the machine for things like computer vision, sound processing, access to hardware as well as access to a wide array of librarys that other folks had written. It opened up doors to expressive uses of computation.

Theo Watson, who was my student in 2005, joined the OF team to help produce an OSX version. I'm not totally sure when our first official release was, but I remember vividly presenting openframeworks to the public in a sort of beta state at the 2006 OFFF festival where we had an advanced processing workshop held at Hangar. One of the participants of that workshop, Arturo Castro, joined the OF team to help produce a linux version. Theo, Arturo and I have been joined by a huge group of people who use OF and help contribute to it.

[1] http://thesystemis.com

In 2008, we won a prize at Ars Electronica and built a laboratory called OF lab which brought many of us who had been working remotely, face-to-face, often for the first time. It was the first of many such world-wide meetings, labs, workshops, and developer conferences that have helped grow the community. That year we also held an OF workshop in Japan at YCAM and discovered that there was a whole community of people across the world using this tool. It was way more global then we had thought. It was simultaneously heartening and also a little bit frightening, the realization that there were people around the world who were using this tool to make a living.

We've been lucky in these events to be able to work with great institutions such as The Studio for Creative Inquiry, V&A museum, YCAM, Imal, Ars Electronica and Eyebeam, which have helped sponsor events to bring the OF community together.

In recent years, we've tried to help grow and expand the community – folks like Kyle McDonald have helped organize the diverse interests of developers and keep everything focused while Caitlin Morris has produced detailed surveys of the community. Greg Borenstein and James George helped launch ofxAddons.com, an online repository which helps organize the impressive quantity of addons that the community is creating on a daily basis. In addition, there are now section leaders for the development of OF, helping grow different parts of the codebase and helping imagine what modern OF looks like. Finally, there are countless contributors to OF who help with the codebase, documentation, examples, addons and answering questions on the forum.

More than anything, we've tried as hard as we can to create a friendly atmosphere around programming, which can be an unfriendly activity and in some ways isolating. We preach the idea of art-making as a laboratory, as R&D for humanity, and OF is one attempt to make a large lab where we can grow and share experiments together. Somehow, luckily, we've attracted some of the most amazing, helpful, warm-hearted, lovely people to come be a part of this, and if you're not already, we'd like to say **welcome**.

1.1 about this book

This book, much in the spirit of openframeworks, is a community driven affair and it's **very much a work in progress**. It was a suggestion on the openframeworks developers mailing list which kicked this off and for a the past months we've been hacking away on it.

A couple of notes,

- Feedback is very much appreciated and we'd like to know what's missing, or what you'd like to have in here. Likewise, if you find something helpful, we'd love to hear it, too! Our github repo is active and we take issues and pull requests.

- Please note that there are likely gaps here. We've tried to roughly lay out chapters in order of skill level, but since it's a collectively written book, it can feel a bit disorienting, with some chapters being on the long side, while some are short. Think of it not as a book you read front to back, but more like a collection of short tutorials from the community.

1.2 credits

Countless thanks to Rachel Uwa for organizing this project, Tega Brain for helping edit it, Ishac Bertran and Andy Clymer for directing the design, and Arturo Castro, Christoph Buchner and Michael Hadley for developing the codebase for generating the book. **MORE MORE MORE MORE**

Editors: **MORE MORE MORE MORE**

Chapter Authors: **MORE MORE MORE MORE**

[2] http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

2 philosophy

by Zach Lieberman[1]

openFrameworks is guided by a number of goals: it should be collaborative, usable and simple, consistent and intuitive, cross-platform, powerful, and extensible. openFrameworks is also driven by a "do it with others" (DIWO) philosophy.

2.0.1 Collaborative

openFrameworks development is collaborative. It thrives on the contributions of many people who engage in frequent discussion, and collaborate on addons and projects. We encourage people to make openFrameworks their own and contribute to the ecosystem.

openFrameworks is developed in a collaborative way. We use git, a distributed versioning system, which means that people can branch, experiment, and make suggestions. If you look at the network diagram on GitHub, it looks like some alien diagram, full of weaving branches, code pulling apart and coming together. There's a huge community all over the world working on the core code: fixing bugs, submitting pull requests, and shaping the tool the way they want to see it. It's a world wide project and it's common to wake up in the USA to an inbox full of pull requests and issues emails from coders in Asia and Europe. Over 70 people have contributed to the openFrameworks core directly, and hundreds of people have forked the code or contributed in other ways.

2.0.2 Simplicity

openFrameworks tries to balance usability and simplicity. The earliest versions of openFrameworks used the core as a tool for teaching C++ and OpenGL, but over time the examples have become the best way to learn while the core takes advantage of more advanced features. In exchange, we've created many more examples that come with openFrameworks, with the goal of trying to make simple, hackable starting points for experimentation.

We want openFrameworks to be as simple as possible, especially for folks coming from other languages and environments. C++ is a "large" language, large in the sense that

[1] http://thesystemis.com

you can write very different types of C++ code. If you go to the bookstore, you'll see hundreds of C++ books. We want to create a library where you don't need to be an expert, where at most you might need a book or two, but that the patterns, approaches and style of the code is simple and intuitive. We were especially interested in achieving a sort of parity with Processing, where many of the functions are similar, allowing easier movement from one framework to another.

2.0.3 Consistent and Intuitive

openFrameworks is consistent and intuitive: it should operate on the principle of least surprise, so that what you learn about one part of openFrameworks it can be applied to other parts of it. Beginners can use openFrameworks to learn about common programming patterns, and advanced users will be able to apply their experience from other languages and toolkits.

Student first is the motto. A lot of the thinking guiding openFrameworks is: what would I would have liked in a tool 5 or 10 years ago? We want the patterns of coding to be simple and to make it as easy as possible to type. This means having self-explanatory function names like "play" and "stop" for video players, and variable names that are intuitive. We have lots of discussions about intuition, driven by a desire to make the code as straightforward as possible. You should learn by typing, autocomplete should be helpful, etc.

2.0.4 Cross-platform

openFrameworks is a cross-platform toolkit. openFrameworks supports as many development environments and operating systems as possible. When you download openFrameworks, you can pick your platform and development environment of choice, and have projects and examples ready to learn from and play with. Difficult to port code is kept out of the core, and kept in addons instead.

openFrameworks is designed to work on a multitude of platforms: OS X, Windows, Linux, iOS, Android, embedded ARM Linux systems, as well as experimental platforms such as BlackBerry PlayBook. openFrameworks developers have devised some clever ways of interfacing with other languages, such such as Java in the case of Android, or Objective-C in the case of iOS.

The joy of a cross-platform library is that it's easy to port your ideas from platform to platform. You can sketch on your laptop then immediately run it on your phone. It allows your ideas to come first, without worrying about the grunt work in making something work across platforms.

2.0.5 Powerful

openFrameworks is powerful: it allows you to leverage advanced libraries like OpenCV, use hardware like your graphics card efficiently, and connect peripherals like cameras and other devices.

We chose C++ because it's a fairly low level language but can still be programmed in a high level way. Because C++ is an extension of the older C programming language, it's possible to write very low level, oldschool C code or higher level C++ code. In openFrameworks, we try to harness both approaches and present simple, clear, yet powerful ways of working with code. Using C++ also makes it easier to interface to the many libraries that have been written in C and C++ without needing to rely on a wrapper for another language.

openFrameworks essentially wraps other libraries such as OpenGL, Cairo, FreeType, FreeImage, and OpenCV. You can think of openFrameworks as a layer of code between user code (the code you will write) and these libraries. The libraries have different styles, idioms, approaches, etc. and our job is to wrap them in a way which makes them more consistent and intuitive.

2.0.6 Extensible

openFrameworks is extensible. When you find something missing from openFrameworks it's easy to create addons that extend it. The core addons for openFrameworks generally wrap libraries rather than solving problems in novel ways. When openFrameworks wraps libraries, the libraries are left exposed for further hacking.

One mental image of openFrameworks is a scaffolding, or shoulders to stand on, while building what you want. One thing that helps keep the core light is that rather than try to include everything we can, openFrameworks has an "addon" system that allows for additional code, libraries, and approaches to be shared between users and woven into projects as necessary.

An openFrameworks addon can be a snippet of code, or it might wrap much more complex libraries such as OpenNI, Tesseract, or Box2d. Addon names usually begin with the prefix "ofx", allowing you to easily see the difference between "core" code and non core code. In addition, we include "core addons", addons that we think people will probably want to use, such as ofxOpenCv, but aren't essential for every project.

We try to organize and support the community developing addons through the http://ofxaddons.com site, which automatically collects addons from GitHub by looking for repos that contain the term "ofx" in the title. Right now there are more than 1,500 addons.

2.0.7 Do it with others (DIWO)

<<<<<<< HEAD The driving philosophy behind openFrameworks is “do it with others” (DIWO). We love do it yourself (DIY) culture, which has been heavily promoted and facilitated by the rise of tutorial websites like Instructables or Make. But we’re also excited about the idea of “making socially” (“with others”). We practice DIWO through workshops, developer conferences, hackathons/labs, knitting circles and meetups in person, and online in the form of mailing lists, forum posts, and so on. We even have a gang sign. Because if you have a gang, you have to have a gang sign. The most important thing we want to stress is that you are not alone, that there’s a great group of people out there learning, teaching, hacking, making and exploring the creative side of code.

2.1 OF structure

The most important thing to understand about OF is that it has been designed to be a self contained structure. You download OF from the website and that version of OF can go anywhere on your hard drive. You shouldn’t mix different versions of OF and while older projects might work in newer versions of OF, that’s not always a guarantee, especially if there’s been a major release.

Because OF can go anywhere on your hard drive, all the internal links are relative. A project file, for example, looks to `../../../libs` rather then a fixed path like `C:Documents and Settings\OF` (on windows) or `/Users/name/Desktop/OF` (on linux / osx). This means that you have to be extremely careful about the depth that a project is away from the root of the OF folder. This is one of the most common mistakes beginners make, they have a project that they either move too shallow or too deeply, or they find and use other people’s code but don’t put it in the right spot. I simply can’t stress this point enough: project files have relative paths. It’s sweet, because it means you can share projects easily (it doesn’t have a fixed path with your name on it, for example) and you can move the whole OF folder around, but it still trips many beginners up.

[NOTE: I think below here belongs in Roy’s chapter.... this chapter is more historical and conceptual, these are more practical]

2.2 project generator

OF now ships with a simple project generator which is really useful for creating new projects. One of the larger challenges has always been making a new project and this tool takes a template and modifies it, chaning the name to a new name that you choose

and even allowing you to add addons, additional libraries that come with OF or that you can download. It allows you to pick where you want the project to go, and while we've structured all the examples to be a certain distance away from the root, you can change the height using this tool. It's designed to make it easy to start sketching in code without worrying too much about making a new project. In the past we've always recommend that you copy an old project and rename it, but this is a more civilized approach to making projects.

2.3 .h and .cpp

In OF (which is C++) you'll see .cpp and .h files (they are sometimes labeled as cxx or hpp files, respectively). The h files are the header files and cpp files are the implementation files. Header files are definitions – they show what's going to be in the code and implementation files are the actual steps. One anology is like a book, the header file is like the the table of contents of the book and describes the layout of the book and the implementation files are like the chapters, where the book is written. Another analogy is a recipe, where you have the list of ingedients (header files) and the actual steps (implementation files). It's useful to know about this split as a lot of modern languages don't have this split.

MORE

2.4 setup/update/draw

OF runs in a kind of game loop model - where we get you a context to draw into and try to run as fast as we can and repetedly draw. There are 3 main functions that you'll see a majority of the code in (as well as the event functions like mousePressed, keyPressed, etc)

- setup()
- update()
- draw()

Setup gets called once, at the first moment we have a window context and it's a good place for initializing variables and loading in files. Update and draw get called repeatedly. Update is for doing non visual changes, such as altering variables or performing analysis, draw is where we do any drawing. The order they get called in is:

setup->update->draw->update->draw->.....

Folks coming from processing, where there is just setup() and draw() often times wonder why we have two functions that repeat instead of one. There's a couple of reasons:

- Drawing in opengl is asynchronous, meaing after you fire off a bunch of commands to draw, they can be running in the background and return control back to your app. If you seperate out your drawing code from your non drawing code there's a potential your code will be faster.
- it's useful for debugging. If you want to know why your code is running slow, now you can comment out the drawing and see if the visual representation that's slow or the updating.

2.5 preprocessor/compiler/linker

When you write code, your end goal is a compiled application - an .exe or .app that you can click on an run. The job of the compiler is to make that executable for you, to turn text into compiled binary files. It's a 3 step process, and it's useful to know what's happening, especially since you can have errors at different steps along the way. Most IDEs output out a very length file of the compiling, and this can be really useful if you are posting to the forums, for example. Once you understand the process of how projects come to be, it can be easier to isolate errors. Nothing is as frustrating or daunting as looking at 500+ errors in a project when you go to compile, but when you notice that there's a missing include, it's clear why and usually one thing will fix many of the problems.

2.5.1 preprocess

The first step is that a preprocessor modifies the text files themselves. When you see the # symbol, that's a preprocessor operation. The most common preprocessor statement you'll see is:

```
#include "xxxxx.h"
```

which actually means, take the content of this file and put it right here. **[NOTE: more on "" vs <>]** You'll also see things like:

```
#define PI 3.1428
```

This means, when you see the word PI in the code, change it to this variable. This isn't a variable, this is literally modifying text.

Another common preprocessor step is asking a question. you can say things like:

```
#ifdef windows
    #include "windows.h"
#else
    #include "nonWindows.h"
#endif
```

As you can imagine this is increadibly useful for cross platform compilcation. If you want to see preprocessor craziness, look at ofConstants.h.

One common error you'll have in the preprocess phase is a file not found error, if you include a file like

```
#include "opencv.h"
```

and it can't find the file, you will get an error at the preprocessing stage. The way to fix this is to add header search paths, basically the places (folders) the IDE goes to look for a file. This is a common error when using a new library and one of the things the project generator is deisgned to help with when adding an addon.

[more on ofMain.h]

2.5.2 compile

Once the text has been modified, the job of the compiler is simply to take .cpp files and turn them into object code. It's taking the text and turning it into machine language instructions (also referred to as assembly). It doesn't touch the h files at all, it only thinks about .cpp files. In the previous phase the .cpp file has all the h files it uses added to it recursively.

This recursive h inclusion is one reason while you will see include guards on the top of h files. They will either look like:

```
#ifndef SOMEWORD
#define SOMEWORD
...
#endif
```

or the more modern

```
#pragma once
```

This is because if a file is included twice into a .cpp file the compiler could be confused. If it's sees the same definition twice, like:

```
float position;
float position;
```

it will not know which one is which. The include guard prevents the file from being included twice.

there are plenty of errors that can happen at compile time – using a variable that you haven't defined for example. The compiler will stop when it hits an error and the IDEs are designed to make it easy for you to see where the errors are and fix them.

The compilers job in life is to take the .cpp files and turn them into .o files. These are individual object files that it will compbine in the next phase, linking.

2.5.3 link

Finally, after we have a bunch of object files, our job is to link them into one thing – in our case an application (alternatively, compilers can compile code into a library, for example). This is what the linker does. As you can imagine, there are jumps from one thing to another. For example, in ofApp you could call a graphics call from ofGraphics:

```
void ofApp::draw(){
    ofCircle(100,100,20);
}
```

This code is calling a function in another object. The linker figures out the links from object to object (in this case between ofApp.o and ofGraphics.o) and links them together into one file.

In addition to header search paths, there are also setting in the IDE for dealing with linker paths and libraries to link against. A common error you might see is a link error, where the code in your project compiles fine, but it's having trouble linking because some object is missing. For example, if you forget to include a .cpp file from the source code, the other code will comiple fine, but when the linker goes to make that jump, it can't find where to jump to. Linker errors are described as "undefined reference" errors and occur at the end of the compile process.

======= The driving philosophy behind openFrameworks is "do it with others" (DIWO). We love do it yourself (DIY) culture, which has been heavily promoted and facilitated by the rise of tutorial website like Instructables or Make. But we're also excited about the idea of "making socially" ("with others"). We practice DIWO through workshops, developer conferences, hackathons/labs, knitting circles and meetups in person, and online in the form of mailing lists, forum posts, and so on. We even have a gang sign. Because if you have a gang, you have to have a gang sign. The most important thing we want to stress is that you are not alone, that there's a great group of people out there learning, teaching, hacking, making and exploring the creative side of code. >>>>>>> upstream/master

3 C++ Language Basics

by Josh Nimoy[1]

> The magician of the future will use mathematical formulas.
>
> **–Aleister Crowley, 1911**

3.1 Look Alive!

This chapter introduces you to writing small computer programs using the C++ language. Although I assume very little about your previous knowledge, the literacy you gain from this chapter will directly influence your comprehension in subsequent chapters of the book, as most other topics stand on the shoulders of this one. Furthermore, the lessons herein are cumulative, meaning you can't skip one of the topics or you will get lost. If you get stuck on one of the concepts, please seek help in understanding specifically the part that did not make sense before moving on to the next topic. Following the lessons with this sort of rigor will insure that you will get the most out of OpenFrameworks, but also computers in general.

3.2 Iteration

I did most of my drawing and painting in the mid-nineties, a high school AP art student sporting a long black ponytail of hair shaved with a step, round eyeglasses, and never an article of clothing without spill, fling, smattering, or splotch of Liquitex Basics acrylic paint. Bored out of my mind in economics class, playing with my TI-82 graphing calculator, I discovered something that flipped a light bulb on in my heart. Unlike smaller calculators around my house growing up, the TI-82 had a thick instruction manual. Amidst sections in this manual about trig functions and other dry out-of-reach science, something caught my thirsty, young eye: a sexy black-on-white pyramid with smaller upside-down pyramids infinitely nested inside, shown in Figure 1.

This fractal, the famous Sierpinski triangle[2], accompanied about twenty-five computer instructions making up the full SIERPINS program. I looked closer at the code, seeing

[1] http://jtnimoy.net

[2] https://www.wolframalpha.com/input/?i=sierpinski+triangle

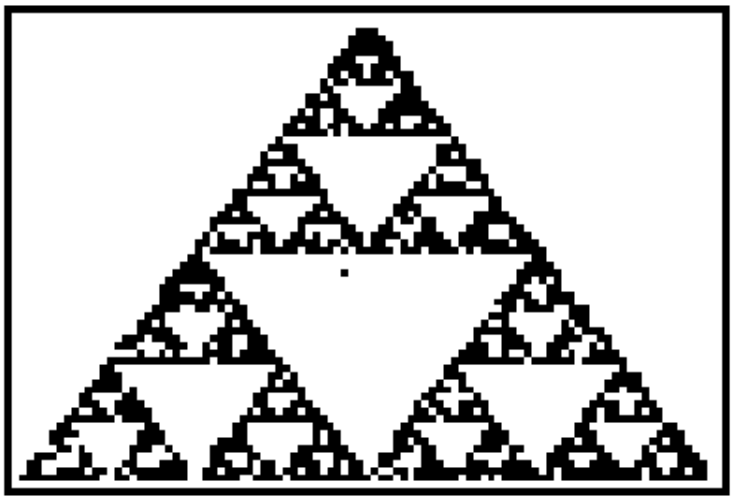


Figure 3.1: Figure 1: TI-82 rendering of the Sierpinski triangle, Courtesy of Texas Instruments

a few numeric operations – nothing too advanced, and most of it was commanding words, like “do this”, or “if something then do another thing”. I was able to key in the code from the book into the graphing calculator and run the program. At first, just a blank LCD panel. Slowly a few random pixels switched to black here and there, not really showing any pattern. After a few more seconds, the scene filled in and I could already see faint outlines of triangles. After a good long time, my calculator finally matched the picture in the book. My mind was officially blown. Certain things did not make sense. What sort of miracle of nature caused such a complex form to come from so little instruction? The screen had over six thousand pixels in it, so why is it that a mere twenty-five instructions was all it took to create this amazing, organism-like artwork? Whose artwork was it? Might I derive a new work from it? Rarely had I ever seen such a magical reward coming from so little work. I had found my new basics. I felt the need to understand the program because (I decided) it was important. I went back into the code and changed some of the numbers, then ran the program again. The screen went blank, then drew a different picture, only this time, skewed to the left, falling out of the viewport. Feeling more courageous, I attempted to change one of the English instructions, and the machine showed an error, failing to run.

The cycle illustrated in Figure 2 is an infinitely repeating loop that I have had a great pleasure of executing for a couple decades and I still love what I do. Each new cycle never fails to surprise me. As I pursue what it means to create a program, and what it means to create software art, the process of iteratively evolving a list of computer instructions always presents as much logical challenge as it does artistic reward. Very few of those challenges have been impossible to solve, especially with other people available to collaborate and assist, or by splitting my puzzle into smaller puzzles. If

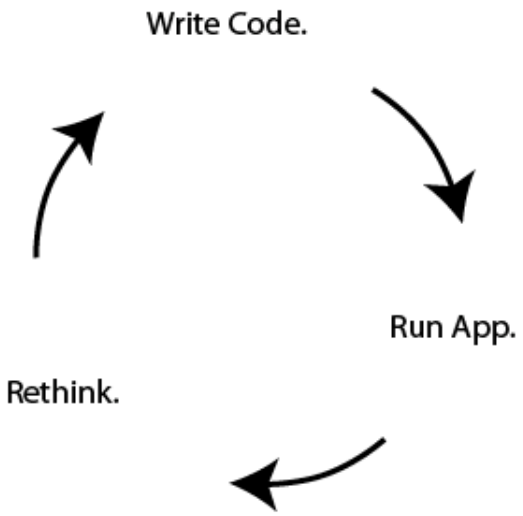


Figure 3.2: Figure 2: The human loop of a programmer.

you have already written code in another environment like Processing, Javascript, or even HTML with CSS, then this first important lesson might seem too obvious.

For those just now familiarizing themselves with what it means to write small programs, it is important to understand the iterative nature of the code writing process. The anecdote in Figure 3 shows what this process is *not*. Rarely would you ever enter some code into the editor just once, and expect to hit compile and see your finished outcome. It is natural, and commonly accepted for programs to start small, have plenty of mistakes (bugs), and evolve slowly toward a goal of desired outcome or behavior. In fact it is so commonplace that to make the former assumption is a downright programmer's mistake. Even in older days when programs were hand-written on paper, the author still needed to eyeball the code obsessively in order to work out the mistakes; therefore the process was iterative. In learning the C++ language, I will provide tiny code examples that you will be compiling on your machine. The abnormal part is typing the code from the book into the editor, and (provided your fingers do not slip) the program magically runs. I am deliberately removing the troubleshooting experience in order to isolate the subject matter of the C++ language itself. Later on, we will tackle the commonplace task of *debugging* (fixing errors) as a topic all its own.

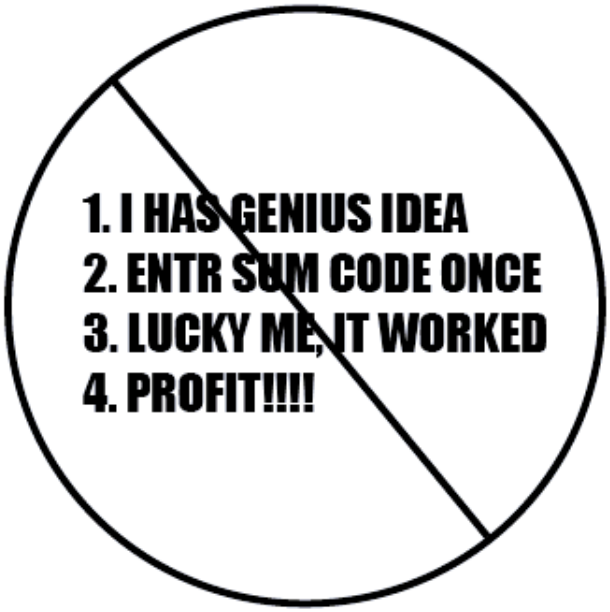


Figure 3.3: Figure 3: Don't get the wrong idea.

3.3 Compiling My First App

Let us start by making the smallest, most immediate C++ program possible, then use the convenient environment to test small snippets of C++ code throughout this chapter. In order to do that, we must have a *compiler*, which is a program that translates some code into an actual runnable app, sometimes referred to as the executable file. C++ compilers are mostly free of charge to download, and in a lot of cases, open source. The apps we generate will not automatically show up in places like Apple's App store, Google Play, Steam, Ubuntu Apps Directory, or Pi Store. Instead, they are your personal, private program files and you will be responsible for manually sharing them later on. In the following chapter *OF Setup and Project Structure*, the compiler will sit on your local computer, able to run offline. For now, we will be impatient and compile some casual C++ on the web using a convenient tool by Sphere Research Labs. Please open your web browser and go to ideone[3] (http://ideone.com).

You will notice right away that there is an editor already containing some code, but it may be set to another language. Let's switch the language to C++11 if it is not already in that mode. Down at the bottom left of the editor, press the button just to the left of "stdin", as shown in Figure 4. The label for this button could be any number of things.

A menu drops down with a list of programming languages. Please choose C++11, shown

[3] http://ideone.com

Figure 3.4: Figure 4

in Figure 5.

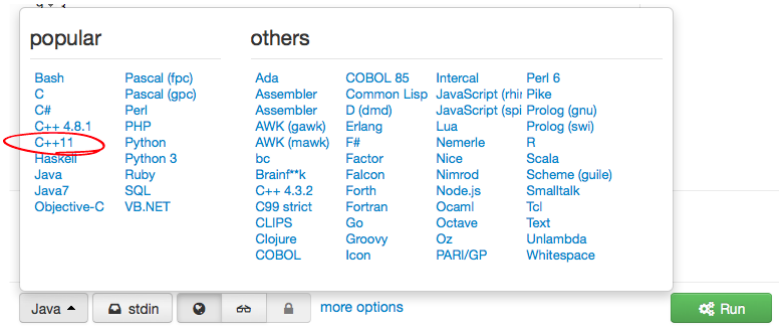
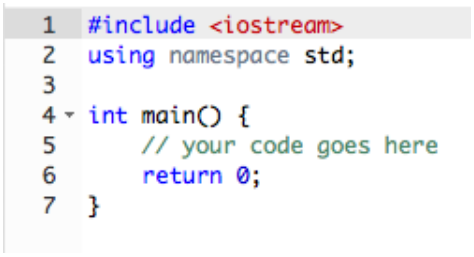


Figure 3.5: Figure 5

Notice that the code in the editor changed, and looks something like figure 6.



```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      // your code goes here
6      return 0;
7  }
```

Figure 3.6: Figure 6

This is just an empty code template that does nothing, and creates no errors. The numbers in the left hand gutter indicate the line number of the code. Press the green button labeled *Run*. You will see a copy of the code, "Success" in the comments, and the section labeled *stdin* (standard input) will be empty. *stdout* (standard output) will also be empty.

3.3.1 Interlude on Typography

Most fonts on the web are variable width, meaning the letters are different widths; the eye finds that comfortable to read. Fonts can also be fixed-width, meaning all the letters (even the W and the lowercase i) are the same width. Although this may look funny and quant like a typewriter, it serves an important purpose. A fixed width font makes a block of text into a kind of game board, like chess squares or graphing paper. Computer programming code is generally presented in fixed-width typesetting, because it is a form of ascii-art. The indentation, white space characters, and repetitive patterns are all important to preserve and easily eyeball for comparison. Every coder I know except artist Jeremy Rotsztain uses some manner of monospaced font for their code. Some typeface suggestions are Courier, Andale Mono, Monaco, Profont, Monofur, Proggy, Droid Sans Mono, Deja Vu Sans Mono, Consolas, and Inconsolata. From now on, you will see the font style switch to `this inline style` . . .

```
and this style encased in a block . . .
```

. . . and that just means you are looking at some code.

3.3.2 Comments

Now please press *Edit* (Figure 7) at the top left of the code editor.

Figure 3.7: Figure 7

You will see a slightly different editing configuration but the same template code will still be editable at the top. We will now edit the code. Find line 5, where it says:

```
// your code goes here .
```

A line beginning with a double forward slash is called a comment. You may type anything you need to in order to annotate your code in a way you understand. Sometimes

a it's useful to "comment out code" by placing two forward-slashes before it, because that deactivates the C++ code without deleting it. Comments in C++ can also take up multiple lines, or insert like a tag. The syntax for beginning and ending comment-mode is different. Everything between the `/* and the */` becomes a comment:

```
/*
this is a multi-line comment.
still in comment mode.
*/
```

Please delete the code on line 5 and replace it with the following statement:

```
cout << "Hello World" << endl;
```

This line of code tells the computer to say "Hello World" into an implied text-space known as *standard output* (aka. *stdout*). When writing a program, it is safe to expect *stdout* to exist. The program will be able to "print" text into it. Other times, it's just a window pane in your coding tool, only used to troubleshoot.

You may put almost anything between those quotes. The quoted phrase is a called a *string* of text. More specifically, it is a *c-string literal*. We will cover more on strings later in this chapter. In the code, the chunk `cout <<` part means "send the following stuff to stdout in a formatted way." The last chunk `<< endl` means "add a carriage return (end-of-line) character to the end of the hello world message." Finally, at the very end of this line of code, you see a semicolon (;).

In C++, semicolons are like a full stop or period at the end of the sentence. We must type a semicolon after each statement, and usually this is at the end of the line of code. If you forget to type that semicolon, the compile fails. Semicolons are useful because they allow multiple statements to share one line, or a single statement to occupy several lines, freeing the programmer to be flexible and expressive with one's whitespace. By adding a semicolon you ensure that the compiler does not get confused: you help it out and show it where the statement ends. When first learning C or C++, forgetting the semicolon can be an extremely common mistake, and yet it is necessary for the code to compile. Please take extra care in making sure your code statements end in semi-colons.

While you typed, perhaps you noticed the text became multi-colored all by itself. This convenient feature is called *syntax-coloring* and can subconsciously enhance one's ability to read the code, troubleshoot malformed syntax, and assist in searching. Each tool will have its own syntax coloring system so if you wish to change the colors, please expect that it's not the same thing as a word processor, whose coloring is something you add to the document yourself. A code editor will not let me assign the font "TRON.TTF" with a glowing aqua color to *just* `endl` (which means end-of-line). Instead, I can choose a special style for a whole category of syntax, and see all parts of my code styled that way as long as it's that type of code. In this case, both `cout` and `endl` are

considered keywords and so the tool colors them black. If these things show up as different colors elsewhere, please trust that it's the same code as before, since different code editors provide different syntax coloring. The entire code should now look like this:

```
#include <iostream.h>
using namespace std;

int main(){
    cout << "Hello World" << endl;
    return 0;
}
```

Now press the green *ideone it!* button at the bottom right corner and watch the output console, which is the bottom half of the code editor, just above that green button. You will see orange status messages saying things like "Waiting for compilation," "Compilation," and "Running". Shortly after, the program will execute in the cloud and the standard output should show up on that web page. You should see the new message in Figure 8.

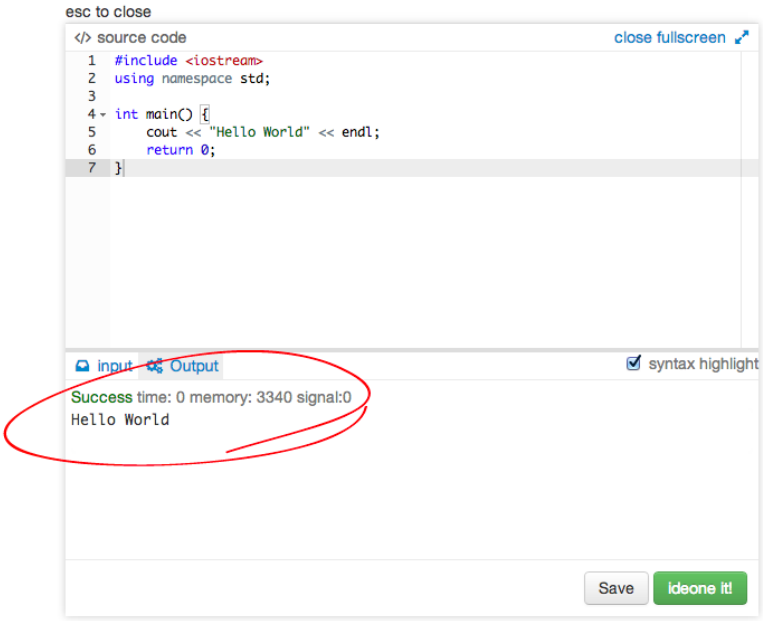


Figure 3.8: Figure 8

You made it this far. Now give yourself a pat on the back. You just wrote your first line of C++ code; you analyzed it, compiled it, ran it, and saw the output.

3.4 Beyond Hello World

Now that we've gotten our feet wet, let's go back and analyze the other parts of the code. The first line is an include statement:

```
#include <iostream>
```

Similar to *import* in Java and CSS, `#include` is like telling the compiler to cut and paste some other useful code from a file called *iostream.h* at that position in the file, so you can depend on its code in your new code. In this case, iostream.h *provides* `cout` and `endl` as tools I can use in my code, just by typing their names. In C++, a filename ending in **.h** is called a header file, and it contains code you would include in an actual C++ implementation file, whose filename would end in **.cpp**. There are many standard headers built into C++ that provide various basic services – in fact too many to mention here. If that wasn't enough, it's also commonplace to add an external library to your project, including its headers. You may also define your own header files as part of the code you write, but the syntax is slightly different:

```
#include "MyCustomInclude.h"
```

In OpenFrameworks, double quotes are used to include header files that are not part of the system installation.

3.4.1 What's with the # ?

It's a whole story, but worth understanding conceptually. The include statement is not really C++ code (notice the absence of semi-colon). It is part of a completely separate compiler pass called *preprocessor*. It happens before your actual programmatic instructions are dealt with. They are like instructions for the code compiler, as opposed to instructions for the computer to run after the compile. Using a pound/hash symbol before these *preprocessor directives*, one can clearly spot them in the file, and for good reason too. They should be seen as a different language, mixed in with the real C++ code. There aren't many C++ preprocessor directives — they are mostly concerned with herding other code. Here are some you might see.

`#define #elif #else #endif #error #if #ifdef #include #line #pragma #undef`

Let's do an experiment. In the code editor, please comment out the include directive on line 1, then run the code. To comment out the line of code, insert two adjacent forward-slashes at the beginning of the line.

```
//#include <iostream>
```

The syntax coloring will change to all green, meaning it's now just a comment. Run the code by pressing the big green button at the bottom right, and you'll see something new in the output pane.

```
prog.cpp: In function ‘int main()’:
prog.cpp:5:2: error: ‘’cout was not declared in this scope
  cout << "Hello World" << endl;
  ^
prog.cpp:5:27: error: ‘’endl was not declared in this scope
  cout << "Hello World" << endl;
                           ^
```

The compiler found an error and did not run the program. Instead, in attempt to help you fix it, the compiler is showing you where it got confused in attempt to help you fix it. The first part, *prog.cpp*: tells you the file that contains the error. In this case, ideone.com saved your code into that default file name. Next, it says `In function ‘int main()’`: file showing you the specific section of the code that contains the error, in this case, between the {curly brace} of a function called *main*. (We will talk about functions and curly braces later). On the next line, we see `prog.cpp:5:2:`. The 5 is how many lines from the top of the file, and 2 is how many characters rightward from the beginning of the line. Next, we see `error: ‘’cout was not declared in this scope`. That is a message describing what it believes it wrong in the code. In this case, it's fairly correct. iostream.h is gone, and therefore no `cout` is provided to us, and so when we try to send "Hello World", the compile fails. On the next couple of lines, you see the line of code containing the fallacious `cout`, plus an extra little up-caret character on the line beneath it, and that is supposed to be an arrow pointing at a character in the code. In this case, the arrow should be sitting beneath the 'c' in `cout`. The system is showing you visually which token is at fault. A second error is shown, and this time, the compiler complains that there is no endl. Of course, we know that in order to fix the error, we need to include `<iostream.h>` so let us do that now. Please un-comment line 1 and re-run the code.

```
#include <iostream>
```

When using OpenFrameworks, you have choice of tools and platforms. Each one shows you an error in a different way. Sometimes the editor will open up and highlight the code for you, placing an error talk bubble for more info. Other times, the editor will show nothing, but the compile output will show a raw error formatted similarly to the one above. While sometimes useful that we receive several errors from a compile, it can save a lot of grief if you focus on understanding and fixing the very first error that got reported. After fixing the top error, it is likely that all subsequent errors will elegantly disappear, having all been covered by your first fix. By commenting out that single line of code at the top, we caused two errors.

3.4.2 Namespaces at First Glance

Moving on to line 2, we see:

```
using namespace std;
```

Let's say you join a social website and it asks you to choose a username. My name is Joshua Nimoy — username might be JNIMOY. I submit the page and it returns an error, telling me that username is already taken, and I have to choose another, since my father, Joseph Nimoy, registered before I did and he's got JNIMOY. And so I must use my middle initial T, and create a unique username, JTNIMOY. I just created and resolved a *namespace conflict*. A namespace is a group of unique names — none are identical. It's possible to have identical names, as long as they are a part of two separate namespaces. Namespaces help programmers avoid stepping on each other's toes by overwriting one another's symbols or hogging the good names. Namespaces also provide a neat and tidy organization system to help us find what we're looking for. In OpenFrameworks, everything starts with `of` . . . like `ofSetBackground` and `ofGraphics`. This is one technique to do namespace separation because it's less likely that any other names created by other programmers would begin with `of`. The same technique is used by OpenGL. Every name in the OpenGL API (Application Programming Interface) begins with `gl` like `glBlendFunc` and `glPopMatrix`. In C++ however, it is not necessary to have a strictly disciplined character prefix for your names, as the language provides its own namespacing syntax. In line 2, `using namespace std;` is telling the compiler that this .cpp file is going to use all the names in the `std` namespace. Spoiler-alert! those two names are `cout` and `endl`. Let us now do an experiment and comment out line 2, then run the code. What sort of error do you think the compiler will return?

```
/* using namespace std; */
```

It's a very similar error as before, where it cannot find `cout` or `endl`, but this time, there are suggested alternatives added to the message list.

```
prog.cpp:5:2: note: suggested alternative:
In file included from prog.cpp:1:0:
/usr/include/c++/4.8/iostream:61:18: note: ‘  std::’cout
   extern ostream cout;  /// Linked to standard output
                  ^
```

The compiler says, "Hey, I searched for `cout` and I did find it in one of the namespaces included in the file. Here it is. `std::cout`" and in this case, the compiler is correct. It wants us to be *more explicit* with the way we type `cout`, so we express its namespace `std` (standard) on the left side, connected by a double colon (::). it's sort of like calling myself `Nimoy::Joshua`. Continuing our experiment, edit line 5 so that `cout` and `endl` have explicit namespaces added.

```
std::cout << "Hello␣World" << std::endl;
```

When you run the code, you will see it compiles just fine, and succeeds in printing "Hello World". Even the line that says `using namespace std;` is still commented out. Now

imagine you are writing a program to randomly generate lyrics of a song. Obviously, you would be using `cout` quite a bit. Having to type `std::` before all your `cout`s would get really tedious, and one of the reasons a programming language adds these features is to reduce typing. So although line 2 `using namespace std;` was not necessary, having it in place (along with other `using namespace` statements) can keep one's C++ code easy to type and read, through implied context.

Say I'm at a Scrabble party in Manhattan, and I am the only Josh. People can just call me Josh when it's my turn to play. However, if Josh Noble joins us after dinner however, it gets a bit confusing and we start to call the Joshes by first and last name for clarity. In C++, the same is also true. I would be `Nimoy::Josh` and he would be `Noble::Josh`. It's alright to have two different `cout` names, one from the `std`namespace, and another from the `improved` namespace, as long as both are expressed with explicit namespaces; `std::cout` and `improved::cout`. In fact, the compiler will complain if you don't.

You will see more double-colon syntax (::) when I introduce classes.

3.5 Functions

Moving on, let us take a look at line 4:

```
int main() {
```

This is the first piece of code that has a beginning and an end, such that it "wraps around" another piece of code. But more importantly, a function *represents* the statements enclosed within it. The closing end of this *function* is the closing curly brace on line 7:

```
}
```

In C++, we enclose groups of code statements inside functions, and each function can be seen as a little program inside the greater program, as in the simplified diagram in figure 9.

Each of these functions has a name by which we can call it. To call a function is to execute the code statements contained inside that function. The basic convenience in doing this is less typing, and we will talk about the other advantages later. Like a board game, a program has a starting position. More precisely, the program has an *entrypoint* expected by the compiler to be there. That entrypoint is a function called *main*. The code you write inside the *main* function is the first code that executes in your program, and therefore it is responsible for calling any other functions in your program. Who calls your *main* function? The operating system does! Let's break down the syntax of the main function in this demo. Again, for all you Processing coders, this is old news.

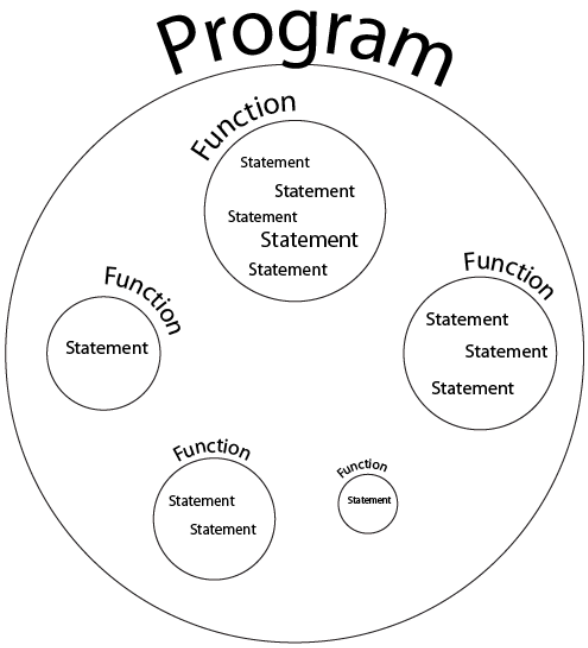


Figure 3.9: Figure 9: Many Functions

When defining a function, the first token is the advertised return type. Functions can optionally return a value, like an answer to a question, a solution to a problem, the result of a task, or the product of a process. In this case, *main* promises to return an `int`, or *integer* type, which is a whole number with no fraction or decimal component. Next token is the name of our function. The system expects the word "main" in all lower-case, but you will later define your own functions and we will get into naming. Next is an opening and closing parenthesis. Yes, it seems kind of strange to have it there, since there is nothing inside it. Later, we will see what goes in there — but never leave out the pair of parentheses with functions because in a certain way, that is the major hint to the human that it's a function. In fact, from now on, when I refer to a function by name, I'll suffix it with a ( ), for example `main()`.

Next, we see an opening curly bracket. Sometimes this opening curly bracket is on the same line as the preceding closing parenthesis, and other times, you will see it on its own new line. It depends on the personal style of the coder, project, or group — and both are fine. For a complete reference on different indent styles, see the the Wikipedia article on Indent Style (http://en.wikipedia.org/wiki/Indent_style).

In between this opening curly bracket and the closing one, we place our code statements that actually tell the computer to go do something. In this example, I only have one statement, and that is the required `return`. If you leave this out for a function whose return type is `int`, then the compiler will complain that you broke your promise

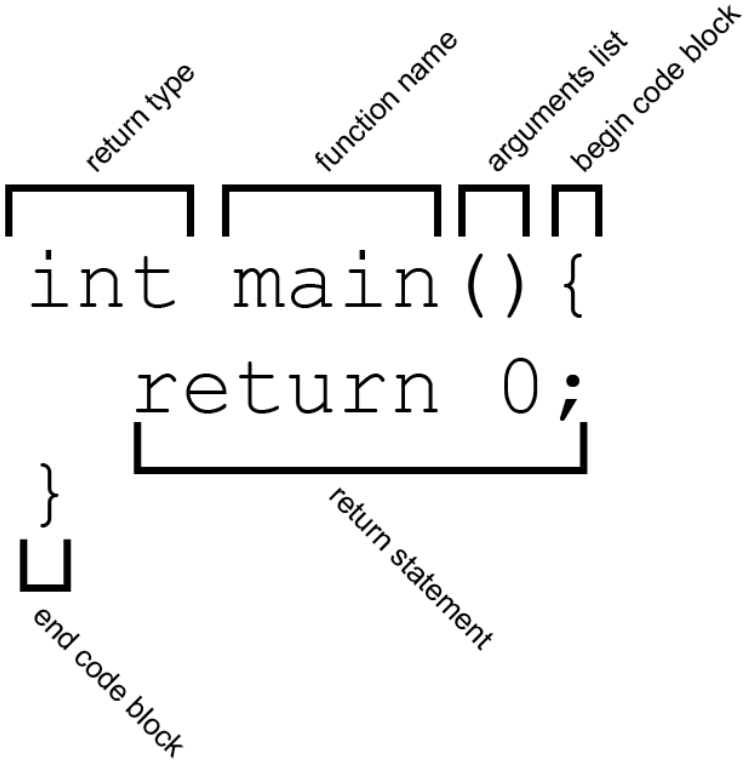


Figure 3.10: Figure 10: The Function

to return an int. In this case, the operating system interprets a 0 as "nothing went wrong". Just for fun, see what happens when you change the 0 to a 1, and run the code.

3.6 Custom Functions

We will now define our own function and make use of it as a word template. Type the sample code into your editor and run it.

```
#include <iostream>
using namespace std;

void greet(string person){
    cout << "Hi there " << person << "." << endl;
}

int main() {
    greet("moon");
    greet("red balloon");
    greet("comb");
    greet("brush");
    greet("bowl full of mush");
    return 0;
}
```

The output shows a familiar bedtime story.

```
Hi there moon.
Hi there red balloon.
Hi there comb.
Hi there brush.
Hi there bowl full of mush.
```

In this new code, notice the second function `greet()` which looks the same but different from `main()`. It has the same curly brackets to hold the code block, but the return type is different. It has the same pair of parentheses, but this time there is something inside. And what about that required return statement? The *void* keyword is used in place of a return type when a function does not return anything. So, since `greet()` has a *void* return type, the compiler will not complain should you leave out the `return`. In the parentheses, you see `string person`. This is a parameter, an input-value for the function to use. In this case, it's a bit like find-and-replace. Down in `main()`, you see I call `greet()` five times, and each time, I put a different string in quotes between the parentheses. Those are *arguments*.

As an aside, to help in the technicality of discerning between when to call them *arguments* and when to call them *parameters*, see this code example:

```
void myFunction(int parameter1, int parameter2){
    //todo: code
}

int main(){
    int argument1 = 4;
    int argument2 = 5;
    myFunction(argument1,argument2);
    return 0;
}
```

Getting back to the previous example, those five lines of code are all ***function calls***. They are telling `greet()` to execute, and passing it the one string argument so it can do its job. That one string argument is made available to `greet()`'s inner code via the argument called `person`. To see the order of how things happen, take a look at Figure 11.

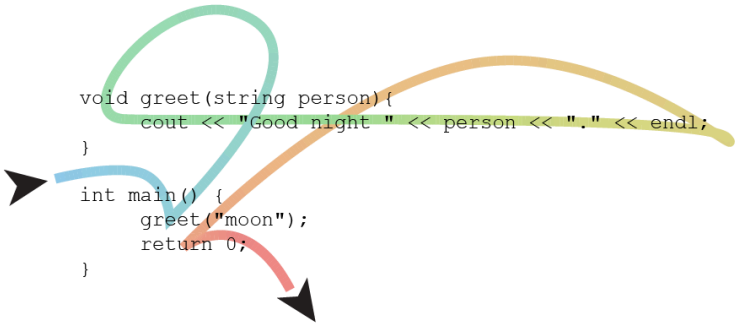


Figure 3.11: Figure 11. Function Call Flow

The colorful line in Figure 11 is the path drawn by an imaginary playback head that steps over the code as it executes. We start at the blue part and go in through the main entrypoint, then encounter `greet()`, which is where a *jump* happens. As the line turns green, it escapes out of `main()` temporarily so it can go follow along `greet()` for a while. About where the line turns yellow, you see it finished executing the containing code inside `greet()` and does a second jump (the return) this time going back to the previous saved place, where it continues to the next statement. The most obvious advantage we can see in this example is the reduction of complexity from that long `cout` statement to a simple call to `greet()`. If we must call `greet()` five times, having the routine *encapsulated* into a function gives it convenience power. Let's say you wanted to change the greeting from "Good night" to "Show's over". Rather than updating all the lines of code you cut-and-pasted, you could just edit the one function, and all the uses of the function would change their behavior along with it, in a synchronized way.

Furthermore, code can grow to be pretty complex. It helps to break it down into small routines, and use those routines as your own custom building blocks when thinking about how to build the greater software. By using functions, you are liberated from the need to meticulously represent every detail of your system; therefore a function is one kind of *abstraction* just like abstraction in art. This sort of abstraction is called *encapsulation of complexity* because it's like taking the big complex thing and putting it inside a nice little capsule, making that big complex thing seem smaller and simpler. It's a very powerful idea — not just in code.

3.7 Encapsulation of Complexity

Imagine actor Laurence Fishburne wearing tinted Pince-nez glasses, offering you two options that are pretty complicated to explain. On the one hand, he is willing to help you escape from the evil Matrix so that you may fulfill your destiny as the hacker hero, but it involves living life on life's terms and that is potentially painful but whatever. The story must go on and btw, there is a pretty girl. On the other hand, he is also willing to let you forget this all happened, and mysteriously plant you back in your tiny apartment where you can go on living a lie, none the wiser. These two options are explained in the movie *The Matrix* and then the main character is offered the choice in the form of colored pills, as a way to simplify an otherwise wordy film scenario. The two complex choices are encapsulated into a simple analogy that is much easier for movie audiences to swallow. See Figure 12.

Rather than repeating back the entire complicated situation, Neo (the main character) needed only to swallow one of the pills. Even if it were real medicine, the idea of encapsulating complexity still applies. Most of us do not have the expertise to practice medicine in the most effective way, and so we trust physicians and pharmacologists to create just the right blend of just the right herbs and chemicals. When you swallow a pill, it is like calling that function because you have the advantage of not needing to understand the depths of the pill. You simply trust that the pill will cause an outcome. The same is true with code. Most of the time, a function was written by someone else, and if that person is a good developer, you are free to remain blissfully ignorant of their function's inner workings as long as you grasp how to properly call their function. In this way, you are the *higher-level* coder, meaning that you simply call the function but you did not write it. Someone who creates a project in OpenFrameworks is sitting on the shoulders of the OpenFrameworks layer. OpenFrameworks sits on the shoulders of the OpenGL Utility Toolkit, which sits on OpenGL itself, and so on. In other words, an OpenFrameworks project is a *higher-level* application of C++, a language with a reputation for *lower-level* programming. As illustrated in Figure 13, I sometimes run into a problem when I tell people I wrote an interactive piece in C++.

There are a few advantages to using C++ over the other options (mostly scripting) for your new media project. The discussion can get quite religious (read: heated) among

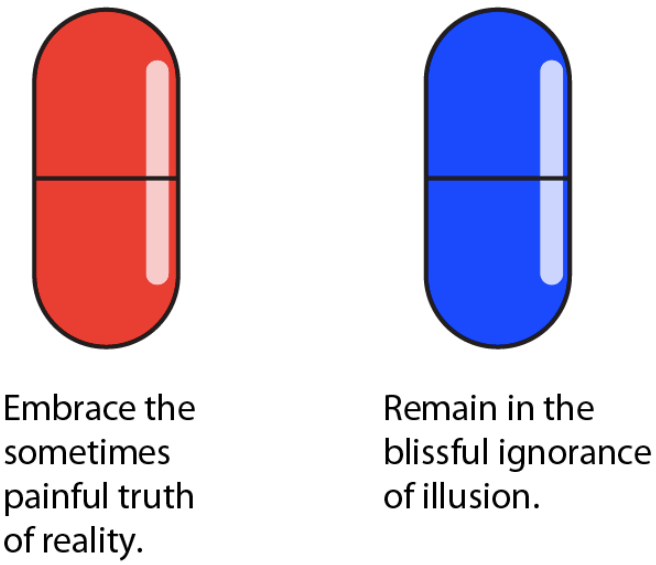


Figure 3.12: Figure 12. Red Pill and Blue Pill from The Matrix

those who know the details. If you seek to learn C++, then usually it is because you seek faster runtime performance, because C++ has more libraries that you can snap in to your project, or because your mentor is working in that language. An OF project is considered higher-level because it is working with a greater encapsulation of complexity, and that is something to be proud of.

3.8 Variables (part 1)

> A “thing” is a “think”, a unit of thought
>
> – Alan Watts

Please enter the following program into ideone and run it.

```
#include <iostream>
using namespace std;

int main(){
    cout << "My␣friend␣is␣" << 42 << "␣years␣old." << endl;
    cout << "The␣answer␣to␣the␣life␣the␣universe␣and␣everything␣is␣"
        << 42 << "." << endl;
    cout << "That␣number␣plus␣1␣is␣" << (42+1) << "." << endl;
    return 0;
```

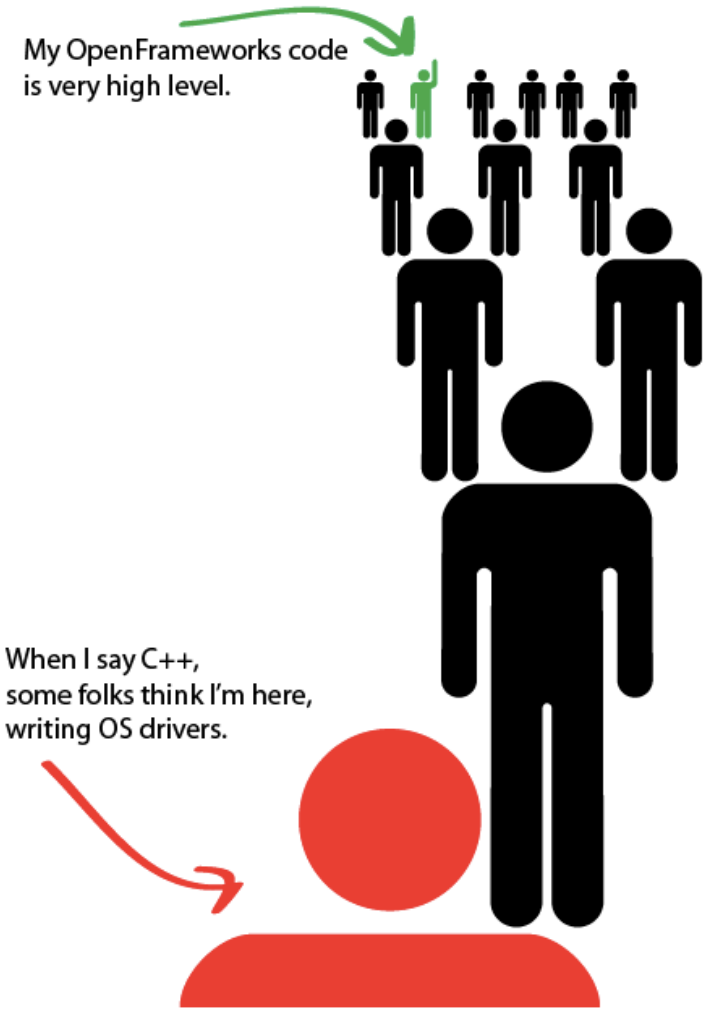


Figure 3.13: Figure 13. Standing on Shoulders of Giants

```
}
```

The output looks like this:

```
My friend is 42 years old.
The answer to the life the universe and everything is 42.
That number plus 1 is 43.
```

We understand from a previous lesson that stuff you put between the << operators will get formatted into the `cout` object, and magically end up in the output console. Notice in the last line, I put a bit of light arithmetic (42+1) between parentheses, and it evaluated to 43. That is called an *expression*, in the mathematics sense. These three lines of code all say something about the number 42, and so they all contain a *literal* integer. A literal value is the contents typed directly into the code; some would say "hard wired" because the value is fixed once it is compiled in with the rest.

If I want to change that number, I can do what I know from word processing, and "find-and-replace" the 42 to a new value. Now what if I had 100,000 particles in a 3d world. Some have 42s that need changing, but other 42s that should not be changed? Things can get both heavy and complex when you write code. The most obvious application of *variables* is that they are a very powerful find-and-replace mechanism, but you'll see that variables are useful for more than that. So let's declare an integer at the top of the code and use it in place of the literal 42s.

```
#include <iostream>
using namespace std;

int main(){

    int answer = 42;

    cout << "My␣friend␣is␣" << answer << "␣years␣old." << endl;
    cout << "The␣answer␣to␣the␣life␣the␣universe␣and␣everything␣is␣"
        << answer << "." << endl;
    cout << "That␣number␣plus␣1␣is␣" << (answer+1) << "." << endl;
    return 0;
}
```

Now that I am using the variable `answer`, I only need to change that one number in my code, and it will show up in all three sentences as 42. That can be more elegant than find-and-replace. Figure 18 shows the syntax explanation for declaring and initializing a variable on the same line.

It is also possible to declare a variable and initialize it on two separate lines. That would look like:

```
int answer;
answer = 42;
```

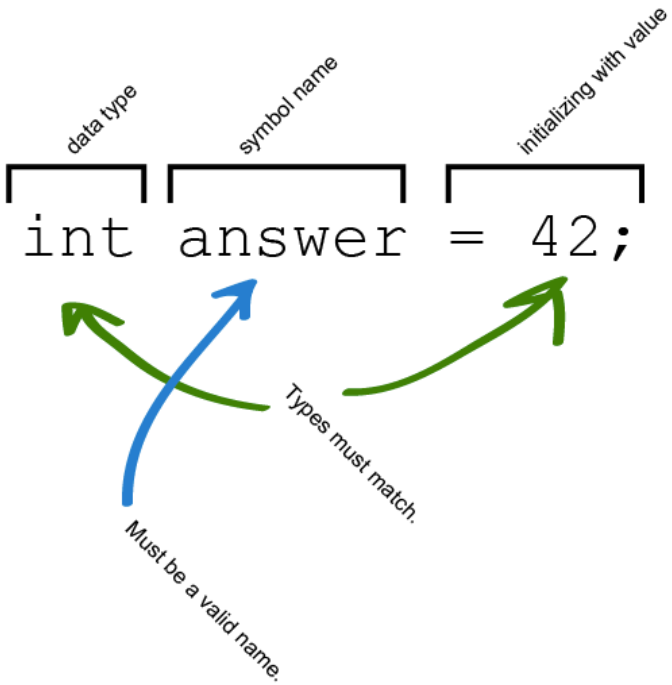


Figure 3.14: Figure 18. Variable declaration and initialization

In this case, there is a moment after you declare that variable when its answer may be unpredictable and glitchy because in C (unlike Java), fresh variables are not set to zero for free — you need to do it. If you don't, the variable can come up with unpredictable values — computer memory-garbage from the past. So, unless you intend to make glitch art, please always initialize your variable to some number upon declaring it, even if that number is zero.

3.8.1 Naming your variable

Notice the arrow below saying "must be a valid name". We invent new names to give our namespaces, functions, variables, and other constructs we define in code (classes, structs, enums, and other things I haven't taught you). The rules for defining a new identifier in code are strict in a similar way that choosing a password on a website might be.

- Your identifier must contain only letters, numbers, and underscores.
- it cannot begin with a number, but it can certainly begin with an underscore.
- The name cannot be the same as one of the language keywords (for example, the word `void`)

The following identifiers are okay.

```
a
A
counter1_
x_axis
perlin_noise_frequency_
         // a single underscore is fine___
       // several underscores are fine
```

Notice lowercase a is a different identifier than uppercase A. Identifiers in C++ are case-sensitive. The following identifiers are not okay.

```
1infiniteloop          // should not start with a number
transient-mark-mode    // dashes should be underscores
@jtnimoy               // should not contain an @
the locH of sprite 1   // should not contain spaces
void                   // should not be a reserved word
int                    // should not be a reserved word
```

naming your variable `void_int`, although confusing, would not cause any compiler errors because the underscore joins the two keywords into a new identifier. Occasionally, you will find yourself running into `unqualified id` errors. Here is a list of C++ reserved keywords to avoid when naming variables. C++ needs them so that it can provide a complete programming language.

```
alignas alignof and and_eq asm auto bitand bitor bool break case
    catch
char char16_t char32_t class compl const constexpr const_cast
    continue
decltype default delete do double dynamic_cast else enum explicit
export extern false final float for friend goto if inline int long
mutable namespace new noexcept not not_eq nullptr operator or or_eq
override private protected public register reinterpret_cast return
short signed sizeof static static_assert static_cast struct switch
template this thread_local throw true try typedef typeid typename
union unsigned using virtual void volatile wchar_t while xor xor_eq
```

3.8.2 Naming conventions

> Differences of habit and language are nothing at all if our aims are identical and our hearts are open.
>
> **–Albus Dumbledore**

Identifiers (variables included) are written with different styles to indicate their various properties, such as type of construct (variable, function, or class?), data type (integer or string?), scope (global or local?), level of privacy, etc. You may see some identifiers capitalized at the beginning and using `CamelCase`, while others remain all `lower_case_using_underscores_to_separate_the_words`. Global constants are found to be named with `ALL_CAPS_AND_UNDERSCORES`. Another way of doing lower-case naming is to start with a lowercase `letterThenCamelCaseFromThere`. You may also see a hybrid, like `ClassName__functionName__variable_name`. These different styles can indicate different categories of identifiers.

More obsessively, programmers may sometimes use what is affectionately nicknamed *Hungarian Notation*, adding character badges to an identifier to say things about it but also reduce the legibility, for example `dwLightYears` and `szLastName`. Naming conventions are not set in stone, and certainly not enforced by the compiler. Collaborators generally need to agree on these subtle naming conventions so that they don't confuse one another, and it takes discipline on everyone's part to remain consistent with whatever convention was decided. The subject of naming convention in code is still a comically heated debate among developers, just like deciding which line to put the curly brace, and whether to use tabs to indent. Like a lot of things in programming, someone will always tell you you're doing it wrong. That doesn't necessarily mean you are doing it wrong.

3.8.3 Variables change

We call them variables because their values *vary* during runtime. They are most useful as a bucket where we put something (let's say water) for safe keeping. As that usually goes, we end up going back to the bucket and using some of the water, or mixing a chemical into the water, or topping up the bucket with more water, etc. A variable is like an empty bucket where you can put your stuff. Figure 19 shows a bucket from the game *Minecraft*.

Figure 3.15: Figure 19. Bucket, courtesy of Mojang AB

If a computer program is like a little brain, then a variable is like a basic unit of remembrance. Jotting down a small note in my sketchbook is like storing a value into a variable for later use. Let's see an example of a variable changing its value.

```
#include <iostream>
using namespace std;

int main(){
    int counter = 0;
    cout << counter;
    counter = 1;
    cout << counter;
    counter = 2;
    cout << counter;
    counter = 3;
    cout << counter;
    counter = 4;
    cout << counter;
    counter = 5;
    cout << counter;
    return 0;
}
```

The output should be `012345`. Notice the use of the equal sign. It is different than what we are accustomed to from arithmetic. In the traditional context, a single equal sign means the expressions on both sides would evaluate to the same value. In C, that

is actually a double equal (==) and we will talk about it later. A single equal sign means "Solve the expression on the right side and store the answer into the variable named on the left side". It takes some getting used to if you haven't programmed before. If I were a beginning coder (as my inner child is perpetually), I would perhaps enjoy some alternative syntax to command the computer to store a value into a variable. Something along the lines of: `3 => counter` as found in the language *ChucK* by Princeton sound lab, or perhaps something a bit more visual, as my repurposing of the Minecraft crafting table in figure 20.

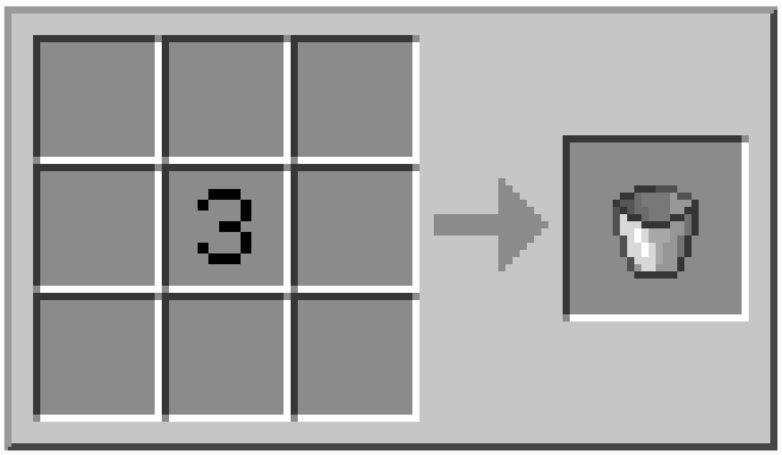


Figure 3.16: Figure 20. Minecraft crafting table repurposed for variable assignment

The usefulness of having the variable name on the left side rather than the right becomes apparent in practice since the expressions get get quite lengthy! Beginning a line with `varname =` ends up being easier for the eyeball to scan because it's guaranteed to be two symbols long before starting in on whatever madness you plan on typing after the equals sign.

Analyzing the previous code example, we see the number increments by 1 each time before it is output. I am repeatedly storing literal integers into the variable. Since a programming language knows basic arithmetic, let us now try the following modification:

```
#include <iostream>
using namespace std;

int main(){
    int counter = 0;
    cout << counter;
    counter = counter + 1;
    cout << counter;
    counter = counter + 1;
    cout << counter;
    counter = counter + 1;
```

```
    cout << counter;
    counter = counter + 1;
    cout << counter;
    counter = counter + 1;
    cout << counter;
    return 0;
}
```

The output should still be `012345`. By saying `counter = counter + 1`, I am incrementing `counter` by 1. More specifically, I am using `counter` in the right-hand "addition" expression, and the result of that (one moment later) gets stored into `counter`. This seems a bit funny because it talks about `counter` during two different times. It reminds me of the movie series, *Back to the Future*, in which Marty McFly runs into past and future versions of himself. See Figure 21.

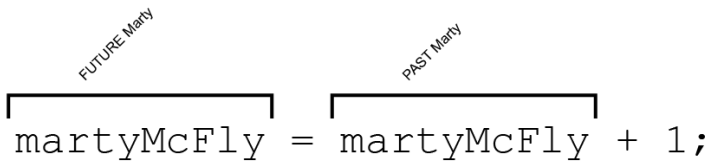


Figure 3.17: Figure 21. The future Marty uses the past Marty

Great Scott, that could make someone dizzy! But after doing it a few times, you'll see it doesn't get much more complicated than what you see there. This is a highly *practical* use of science fiction, and you probably aren't attempting to challenge the fabric of spacetime (unless you are Kyle McDonald, or maybe a Haskell coder). The point here is to modify the contents of computer memory, so we have `counter` from one instruction ago, in the same way that there might already be water in our bucket when we go to add water to it. Figure 22 shows `bucket = bucket + water`.

Incrementing by one, or adding some value to a variable is in fact so commonplace in all programming that there is even syntactic sugar for it. *Syntactic Sugar* is a redundant grammar added to a programming language for reasons of convenience. It helps reduce typing, can increase comprehension or expressiveness, and (like sugar) makes the programmer happier. The following statements all add 1 to `counter`.

```
counter = counter + 1; // original form
counter += 1;          // "increment self by" useful because it's
   less typing.
counter++;             // "add 1 to self" useful because you don't
   need to type a 1.
++counter;             // same as above, but with a subtle
   difference.
```

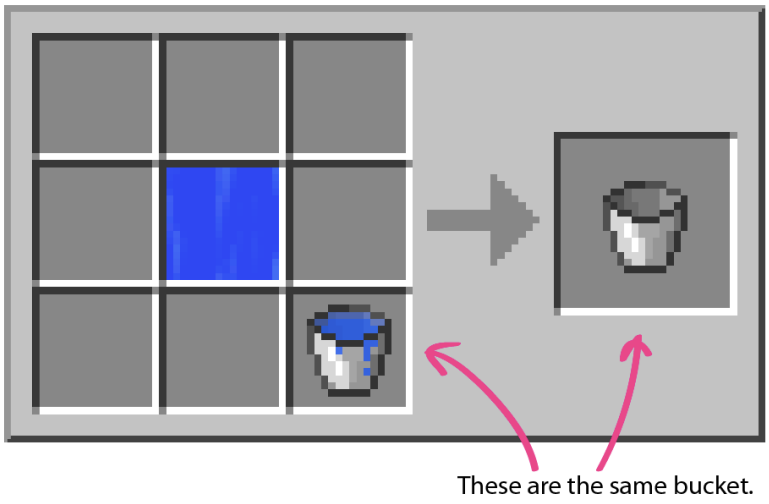


Figure 3.18: Figure 22. bucket = bucket + water

Let's test this in the program.

```
#include <iostream>
using namespace std;

int main(){
    int counter = 0;
    cout << counter;
    counter++;
    cout << counter;
    counter++;
    cout << counter;
    counter++;
    cout << counter;
    counter++;
    cout << counter;
    counter++;
    cout << counter;
    return 0;
}
```

Yes, it's a lot less typing, and there are many ways to make it more concise. Here is one way.

```
#include <iostream>
using namespace std;

int main(){
    int counter = 0;
    cout << counter++;
```

```
    cout << counter++;
    cout << counter++;
    cout << counter++;
    cout << counter++;
    cout << counter++;
    return 0;
}
```

The answer is still `012345`. The postfix incrementing operator will increment the variable even while it sits inside an expression. Now let's try the prefix version.

```
#include <iostream>
using namespace std;

int main(){
    int counter = 0;
    cout << ++counter;
    cout << ++counter;
    cout << ++counter;
    cout << ++counter;
    cout << ++counter;
    cout << ++counter;
    return 0;
}
```

If you got the answer `123456`, that is no mistake! The prefix incrementing operator is different from its postfix sister in this very way. With `counter` initialized as 0, `++counter` would evaluate to 1, while `counter++` would still evaluate to 0 (but an incremented version of `counter` would be left over for later use). The output for the following example is `1112`.

```
#include <iostream>
using namespace std;

int main(){
    int counter = 0;
    cout << ++counter; // 1: increments before evaluating
    cout << counter;   // 1: has NOT changed.
    cout << counter++; // 1: increments after evaluating
    cout << counter;   // 2: evidence of change.
    return 0;
}
```

For arithmetic completeness, I should mention that the subtractive *decrementing* operator (counter–) also exists. Also, as you might have guessed by now, if one can say `counter + 1`, then a C compiler would also recognize the other classic arithmetic like `counter - 3` (subtraction), `counter * 2` (asterisk is multiplication), `counter / 2` (division), and overriding the order of operations by using parentheses, such as

`(counter + 1)/ 2` evaluating to a different result than `counter + 1 / 2`. Putting a negative sign before a variable will also do the right thing and negate it, as if it were being subtracted from zero. C extends this basic palette of maths operators with boolean logic and bitwise manipulation; I will introduce them in Variables part 2.

There are a few more essentials to learn about variables, but we're going to take what we've learned so far and run with it in the name of fun. In the meantime, give yourself another pat on the back for making it this far! You learned what variables are, and how to perform basic arithmetic on them. You also learned what the ++ operator does when placed before and after a variable name.

The C++ language gets its name from being the C language plus one.

3.9 Conclusion

Congratulations on getting through the first few pages of this introduction to C++. With these basic concepts, you should be able to explore plenty far on your own, but I will admit that it is not enough to prepare you to comprehend the code examples in the rest of ofBook. Because of limited paper resources, what you've seen here is a "teaser" chapter for a necessarily lengthier introduction to the C++ language. That version of this chapter got so big that it is now its own book — available unabridged on the web, and possibly in the future as its own companion book alongside ofBook. Teaching the C++ language to non-programmers is indeed a large subject all itself, which could not be effectively condensed to 35 pages, let alone the 100+ page book it grew to be. If you're serious about getting into OpenFrameworks, I highly recommend you stop and read the unabridged version of this chapter before continuing in ofBook, so that you may understand what you are reading. You will find those materials at https://github.com/openframeworks/ofBook/tree/master/02_cplusplus_basics/unabridged.md

3.10 PS.

Stopping the chapter here is by no means intended to separate what is important to learn about C++ from what is not important. We have simply run out of paper. In lieu of how important the rest of this intro to C++ is, and based on ofZach's teaching experience, here is more of what you'll find in the unabridged version:

- Variables exist for different periods of time - some for a long time, and some for a small blip in your program's lifecycle. This subject of *scope* is covered in the unabridged version of this book, entitled *Variables (part 2)*.
- Variables have a *data type*. For example, one holds a number while another holds some text. More about that in *Fundamental Types*.

- It's important to reiterate that unlike Processing, variables do not necessarily start with a zero value. You must initialize them with your desired value, and otherwise there's no telling what will be waiting there for you. You'll find additional discussion of this phenomenon in the introduction to arrays.

  The best way to predict your future is to create it.

  –Abraham Lincoln

4 OF structure

by Roy Macdonald[1]

Let's get into openFrameworks (I'll refer to it as OF from now on). The philosophy chapter talks about OF in very abstract and conceptual manner, which is really useful for understanding the design of the OF environment. Now, let's take a look at what the OF download looks like.

I have found that it is very useful to explain OF by making analogies to cooking. Coding and cooking have a lot of things in common, and most people are familiar with the act of cooking. In this chapter, I'll be drawing connections between processes and terminology in cooking and openFrameworks.

4.1 First things first

You need to download the OF version and the IDE (Integrated Development Environment) that suits your platform. The IDE is a piece of software that will let you write, compile, run and debug the code you write. It is "integrated" because it uses other pieces of software to do each of the mentioned tasks. You can run your code without using the IDE, but the IDE will make your programming life much easier.

Go to www.openframeworks.cc/downloads and download the version that you need. By the side of each available version you will find a link to the where to download the needed IDE and how to install it.

4.2 Welcome to your new kitchen

4.2.1 IDE:

As said before, the *Integrated Development Environment*, IDE, is the application you will be using to build your openFrameworks projects. It will let you write code, compile (bake it), test it and debug it (find out what is giving you problems, if there is any, and fix it). There are several different IDEs, at least one for each platform you might be utilizing.

[1] http://github.com/roymacdonald/

The IDE is your new kitchen, where you'll find all the tools to cook incredible stuff. Yet there are a lot of different kitchen brands, just like IDEs. All do the same but things might be laid out and named in a slightly different way. If you know how to use one, you can use any other. For clarification, I will go through each IDE and show where you will find the most used commands, controls and settings. Only read for the IDE you are going to use.

All IDEs have a similar interface:

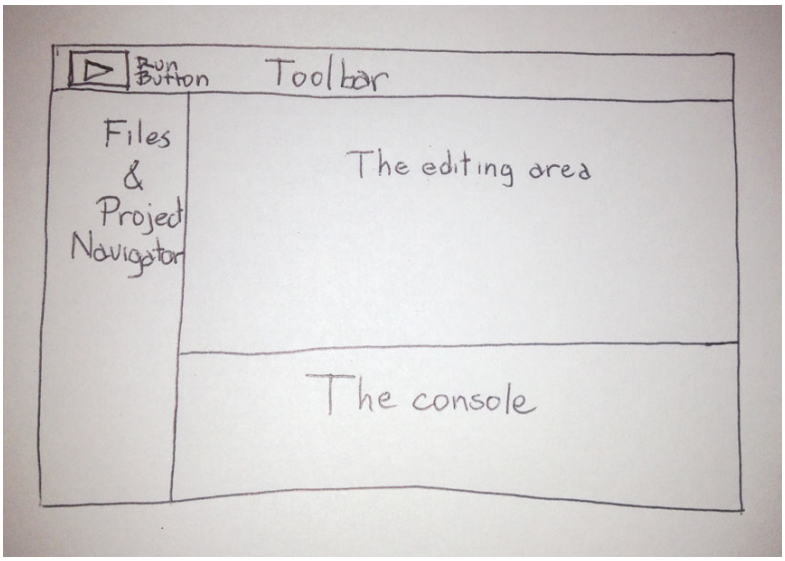


Figure 4.1: Abstract IDE interface

- Toolbar and Run Button: In the tool bar you'll find several useful buttons, such as open, save, save all, et cetera. The "run" button is very important. Usually it is labeled with a triangle pointing to the right, like the "play" button. When you press it, it will compile your code and if it went with no problem it will automatically run your code. Hence this is a frequently used button.
- File selector and project navigator area: Here you will see your project and the files associated with it. Usually it is displayed like a hierarchically ordered list of files. Here you'll find all the OF library files, as well as the files that are particular to your project.
- Editing area: When you open a file in the project navigation area, usually by double clicking it, it should open in the editing area. This looks just like any regular text-editing software, and behaves quite much the same.
- Console: This is where your app, when running, outputs messages. These messages are really useful for debugging, You can print text messages to the console using the `cout` comand or `ofLog(...)` function.

4.2.1.1 Apple Xcode

Xcode is Apple's IDE. Used both for iOS apps and desktop apps. Even though there are other IDEs for MacOSX, Xcode is a a pretty mature one with lots of nice and useful features, especially for dealing with iOS apps.

Use the latest version of Xcode and read the setup guide.

4.2.1.2 Microsoft Visual Studio 2012 Express

This is Microsoft's IDE, it is aimed for Windows development. It's a commercial product, but there's a free version you can download called "Express".

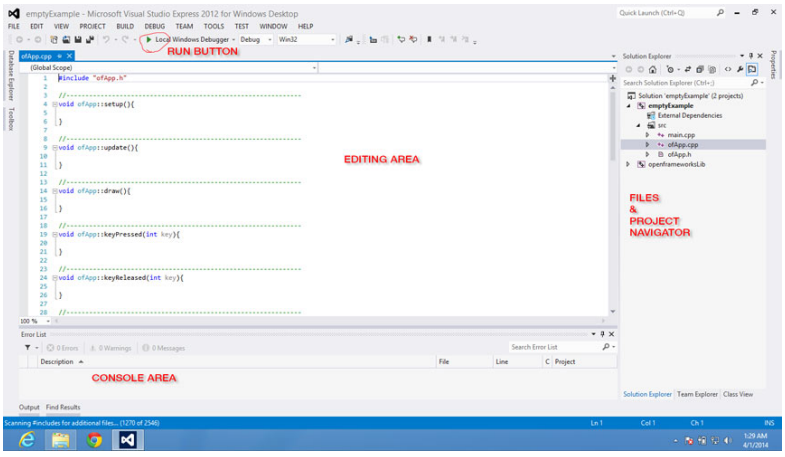


Figure 4.2: VS screenshot

4.2.1.3 Code::Blocks

Code::Blocks is a free IDE. It runs on several platforms, but OF supports it for windows and linux. It is quite "light" in terms of downloading and we use it in workshops over VS, which can be a bit intimidating for beginners. For windows, follow the setup instructions (including step "e") carefully. For linux, there are scripts that help install dependencies and the codeblocks IDE.

4.3 Running examples

Find the OF version that you downloaded and decompress it. From now on we will refer to this folder as the OF root folder. You can place the OF root folder anywhere

you like. One thing to stress is that OF is designed to be self contained – everything you need will stay in one folder and this folder can even be moved around on your drive if need be. If you download another version of openframeworks, it should stay in it's own folder and don't try to merge them.

Open it. Inside of it you will find several folders which we will describe below in more detail. For now, navigate to the examples folder and let's try to compile examples/-graphics/graphicsExample. If you are on OSX, click on the graphicsExample.xcodeproj. If you are using visual studio, choose the "sln" file. On code::blocks, choose the "workspace" file.

As a quick side note, about workspace files, the reason we ask you to open those rather then the project file is that they contain a sub-project to build the OF library also. If you have any doubts, please read the readme for your given platform.

Now your IDE should open and load this example. It should look like the IDE screen-shots above. Locate the "Run" button or menu option and click on it. The example should compile (which might take a while, since the first time you compile you are comiling the OF library also). You'll see alot of files being compiled the first time – don't worry, this will just happen once, when the OF library needs to be rebuilt. Feel free to get a cup of coffee or stretch. Long compile times are great moments to take a screen break.

If everything went well, a new window will pop up and display the example you just compiled. If this happened, congrats! You just have installed and compiled openFrame-works successfully and you are ready to go on. If this didn't happen, the first rule is, don't panic! check the notes below for each IDE and be sure to read the release notes that come with OF.

- Xcode: make sure that the popdown menu just at the right of the run button has selected the item with the name of your example and not the one named "openFrameworks." There might be more than one item with the name of the example you are trying to run. Select anyone as long as it is not the one named "openFrameworks". This popdown menu selects the target you want to compile. If "openFrameworks" is selected you will just compile the openFrameworks core and not the example code. When you select the other items xcode will compile both the OF core and the code for your example and when done it will run the example.
- VS: make sure you've opened the .sln file. Visual studio express doesn't have a triangle button by default (I think it looks like a gear for debugging). Locate the run without debugging option[2], which you can add to the menu bar if you want to customize the IDE.
- CB: make sure you've opened the .workspace file. If you are opening up other projects, be sure to close the current workspace first, as CB doesn't handle

[2] http://social.msdn.microsoft.com/Forums/vstudio/en-US/7b2182f9-0e46-4e6f-a8db-3ab5af39f14b/start-without-debugging-option-missing-from-debug-menu?forum=vsdebug

multiple projects open very well.

- With all IDEs, the play button will compile and run the project, but sometime you might need to hit the button twice if the window doesn't launch.

If the graphics example works, spend some time going through the other OF examples and running them. Usually it's good practice to close the current project / workspace completely before you open another one. Once the OF library is compiled It should be fun!

If you have trouble, please keep track of what errors you have, what platform you are on, and start with using the OF forum[3]. There's years of experience there, and really helpful folks who can help answer questions. First, try searching for a specific error and if you don't see it, post a question in the forum. When you are beginning it can be quite frustrating, but the community is very good at helping each other out.

Once done continue reading.

4.4 OF folder structure

Inside the OF root folder you will find several other folders, at least, the following:

4.4.0.4 Addons

The "addons" folder will contain the included "core" addons. Addons are extra pieces of code that extend OF's functionalities, allowing you to do almost anything with OF. Addons are usually written by third parties that have shared these. The "core" addons, the ones already included in your OF download, are addons that are used so frequently that it has been decided to include them as part of the official OF download. These are coded and maintained by OF's core developers.
Check the examples/addons folder in your OF root folder where you will find at least one example about how to use each of these addons. You can also go to ofxAddons[4] where you'll find a huge collection of additional addons from the community.

4.4.0.5 Apps

This is the folder where you put your project files as you make new projects. Your current OF download contains the folder named "myApps" inside of "apps", and this is

[3] http://forum.openframeworks.cc/
[4] http://ofxaddons.com/

where the project generator will default to when you make a new project. One important thing to note is that the project is positioned relatively to the libs folder, ie, if you were to look inside the project file you'd see relative folder paths, ie `../../../libs`. This image is showing how ../../../libs might work visually:

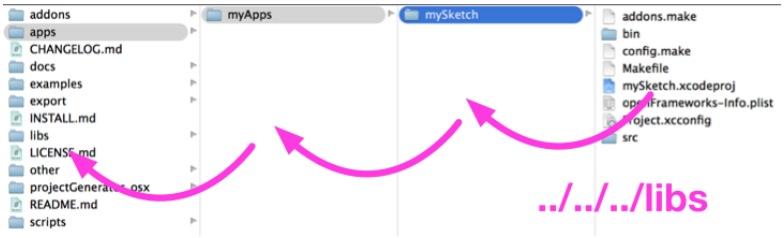


Figure 4.3: app position to root

A key thing to note is that your project files always have to live at this heigh away from the root. If you alter their heigh, they won't find the other pieces they need to compile. This is a very common mistake for beginners, especially as you start to find example projects and bring them in OF to test, etc. Please make sure you understand that projects are always setup relative to the root level. This is what makes the whole OF folder be able to be anywhere on your hard drive – it's all a self contained package. This is probably the #1 issue beginners have, so it's worth emphasizing.

4.4.0.6 Examples

This is a folder with examples, sorted out by topic. There are a big bunch of examples that cover almost all of OF's aspects. Each example is made with the idea of keeping it simple and focused to the particular aspect it tries to address, thus making it easily understandable and a good starting point when you want to do something similar in your project.

4.4.0.7 libs

These are the libs that openframeworks uses to compile your project. They include things like freeType for typography support, freeImage for image loading, glfw for windowing, etc. Also the code for openframeworks is in libs, as well as the project files that help compile OF. If you need to look at the source code, here's where to look.

4.4.0.8 other

Here you'll find an Arduino sketch for using with the serial example located at examples/communication/. This is handy to check that your serial communication with

Arduino is set up correctly and working.

4.4.0.9 projectGenerator

OF now ships with a simple project generator which is really useful for making new projects. One of the larger challenges has always been making a new project and this tool takes a template (located in the scripts folder) and modifies it, chaning the name to a new name that you choose and even allowing you to add addons. It allows you to place the project anywhere you want, and while we've structured all the examples to be a certain distance away from the root, you can set the position using this tool (ie, if you put the project deeper, it will change the `../../../libs` to `../../../../libs`, for examples). It's designed to make it easy / trivial to start sketching in code, without worrying too much about making a new project. In the past we've always recommend that you copy an old project and rename it, but this is a more civilized approach to making projects. Check the readme file where the usage of this app is described.

4.4.1 The OF Pantry:

Your default new kitchen will only have tools for coding, but the OF kitchen comes with a super nice pantry, filled up with really nice, cool and useful stuff.

Imagine that you want to cook something but your kitchen has no pantry or if it has it is completely empty. In such conditions, cooking anything would be quite difficult, as you'll have to go out and buy the things you need and you probably won't find everything in one outing. This is not a nice scenario, especially if you want to get creative and make awesome things.

So, what happens when you have your pantry filled with OF's components? You will be able to cook whatever you want because some really good ingredients are already there. Additionally, there are some really nice tools in there. This will let you complete recipes in a short amount of time, leaving you more time to get creative and try out new and more delicious recipes.

4.4.1.1 What is inside the OF pantry

Here you will find a lot of different things, from ingredients to tools, all ordered according to use. This is a breakdown of how openframeworks code is organized (as well as the examples) and should give you a sense of what OF contains:

- **3D**
  - Tools for drawing basic 3D polygonal objects, such as spheres, cubes, pyramids, etc.

  - `ofCamera`, `ofEasyCam` 3d cameras for navigating and viewing your 3D scene, either interactively or not.
  - `ofNode` a 3D point in space, which is the base type for any 3d object, allowing it to be moved, rotated, scaled, nested and drawn.
  - `ofMesh` A primitive for batching points in 3D space that allows you to draw them in several different ways such as points, lines, lines strips, triangles, triangles strips, and attach textures (images) to these. All of this is done very efficiently using your computer's GPU.
  - functions to help load and save 3D objects.

- **app**
  - Tools for setting and getting properties of your app such as window size, position, different drawing modes, framerate, et cetera.
  - different windowing systems, such as `ofAppNoWindow` which sets up openframeworks in a windowless context for example.

- **communication**
  - `ofSerial` which provides simple serial port communication
  - `ofArduino` which allows openframeworks to communicate via Firmata

- **events**
  - this is code for the OF event manager, allowing you to tap into app events if you need or even creating your own events.

- **gl**
  - OpenGL is the library for using the computer's GPU, this folder contains gl specific functionality such as VBOs (Vertex Buffer Object), FBOs (Frame Buffer Object), Renderers, Lights, Materials, Shaders, Textures, and several other GL utilities.
  - OF implements different rendering pipelines, Fixed and Programable rendering pipelines as well as OpenGL ES (used on less powerful devices such as smartphones and the Raspberry Pi) – most of this code is found in the gl folder.

- **graphics**
  - There are a lot of capabilities here, such as loading and saving images of almost any kind, implementing several different methods for drawing in 2D, and exhibiting colors and styles. Most of the drawing tools rely on OpenGL so these are usually very fast. Graphics also allows you to render as PDF, and it features typography with several kinds of rendering options and utilities.
  - There are useful objects like ofImage, a class for loading, saving and drawing images
  - `ofTrueTypeFont` is a library for loading and drawing true type fonts

- **math**
  - in ofMath you'll find things like vectors (ie `ofVec2f`, `ofVec3f`) , matrices (ie `ofMatrix3x3`, `ofMatrix4x4`), qaternions and some really useful math help functions like `ofRandom` and `ofNoise`.
- **sound**
  - openframeworks has both low level sound, `ofSoundStream`, for direct access to the sound card, as well as higher level code `osSoundPlayer` for playing samples and sound effects.
- **base types**
  - A lot of different base types used extensively within OF. For folks that want to understand the archtecture of OF, this is a useful place where you'll find base types for common elements.
- **utils**
  - Utilities for file input and output, logging, threading, system dialogs (open, save, alert), URL file loader, reading and saving XML files (super useful for storing and reading your app's settings)
  - `ofDirectory` which can help iterate through a directory
- **video**
  - Video Grabber and player, with behind-the-scenes implementations for all the supported platforms.
  - `ofVideoGrabber` helps with grabbing from a webcam or attached camera
  - `ofVideoPlayer` helps with playing video files

4.4.1.2 Addons

As mentioned before, addons extend OF core functionalities, and in each OF distribution there are several included addons, usually referred to as "core addons":

- **ofx3DModelLoader** Used for loading 3D models into your OF project. It only works with .3ds files.
- **ofxAssimpModelLoader** Also loads 3D models into your OF project, but it is done using the assimp[5] library, which supports a wide variety of 3D file formats, even animated 3D objects.
- **ofxGui** This is the default GUI (Graphical User Interface) for OF. It lets you add sliders and buttons so you can easily modify parameters while your project is running. It relies heavilly on ofParameters and ofParameterGroup. It allows you to save and load the values for the parameters that you've adjusted.

[5] http://assimp.sourceforge.net/

- **ofxKinect** Recently added as a core addon. As you probably infer, it's for using a Microsoft XBox Kinect 3D sensor with your OF project. This addon relies on libfreenect[6], so you can only access the depth and rgb images that the kinect reads and adjust some of its parameters, like tilt and light. It includes some handy functions that allow you to convert Kinect's data between several different kinds. Please note that ofxKinect doesn't perform skeleton tracking. For such thing you need to use ofxOpenNI.
- **ofxNetwork** Lets you deal with network protocols such as UDP and TCP. You can use it to communicate with other computers over the network. Check out network chapter for more information.
- **ofxOpenCv** This is OF's implementation of the best and most used Computer Vision code library, openCV. Computer Vision is a complete world by itself, and being able to use openCV right out-of-the-box is a super important and useful OF feature.
- **ofxOsc** OSC (Open Sound Control) implementation for OF. OSC easily comunicates with other devices or applications within the same network. OSC is used to send messages and parameters from one app to another one. Several chapters in this book discuss OSC.
- **ofxSvg** Loads and displays SGV files. These are vector graphics files, usually exported from vector drawing programs such as Adobe Illustrator.
- **ofxThreadedImageLoader** Loads images on a different thread, so your main thread (the one that draws to your screen) doesn't get stuck while loading images. Really useful when loading online images.
- **ofxVectorGraphics** Used to write out EPS vector graphics files. It the same drawing syntax as OF's regular drawing syntax, so it is really easy to use. Check chapter **[add correct chapter numbre]** for more info about OF's drawing capabilities.
- **ofxXmlSettings** This is OF's simple XML implementation used mostly for loading and saving settings.

That's what's in the pantry. what do you want to cook?

[6] http://openkinect.org/wiki/Main_Page

5 Graphics

By: Michael Hadley[1] with generous editor support from Abraham Avnisan[2], Brannon Dorsey[3] and Christopher Baker[4].

This chapter builds off of the *C++ Basics* and *Setup and Project Structure* chapters, so if you aren't familiar with basic C++ and creating openFrameworks projects, check out those chapters first.

In sections 1 and 2, we will create "paintbrushes" where the mouse is our brush and our code defines how our brush makes marks on the screen. In section 3, we will explore something called "coordinate system transformations" to create hypnotizing, spiraling rectangles. Source code for the projects is linked at the end of each section. If you feel lost at any point, don't hesitate to look at the completed code! You can check out the whole collection of code here[5] - both for standard development structure (Xcode, Code::Blocks, etc.) and for ofSketch.

If you are following along using ofSketch, great! There are a couple things to note. Coding in ofSketch is a bit different than coding in other Xcode, Code::Blocks, etc. 1) There will be a few points where variables are added to something called a header file (`.h`). When you see those instructions, that means you should put your variables above your `setup()` function in ofSketch. 2) You'll want to use `ofSetWindowShape(int width, int height)` in `setup()` to control the size of your application. 3) Some of the applications you write will save images to your computer. You can find them in your ofSketch folder, by looking for `ofSketch/data/Projects/YOUR_PROJECT_NAME/bin/data/`.

5.1 Brushes with Basic Shapes

To create brushes, we need to define some basic building blocks of graphics. We can classify the 2D graphics functions into two categories: basic shapes and freeform shapes. Basic shapes are rectangles, circles, triangles and straight lines. Freeform shapes are polygons and paths. In this section, we will focus on the basic shapes.

[1] http://www.mikewesthad.com/
[2] http://abrahamavnisan.com/
[3] http://brannondorsey.com/
[4] http://christopherbaker.net/
[5] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code

5.1.1 Basic Shapes

Before drawing any shape, we need to know how to specify locations on screen. Computer graphics use the Cartesian coordinate system[6]. Remember figure 1 (left) from math class? A pair of values `(x, y)` told us how far away we were from `(0, 0)`, the origin. Computer graphics are based on this same system, but with two twists. First, `(0, 0)` is the upper leftmost pixel of the screen. Second, the y axis is flipped such that the positive y direction is located below the origin figure 1 (center).

If we apply this to the top left of my screen figure 1 (right), which happens to be my browser. We can see the pixels and identify their locations in our new coordinate system. The top left pixel is `(0, 0)`. The top left pixel of the blue calender icon (with the white "19") is `(58, 5)`.

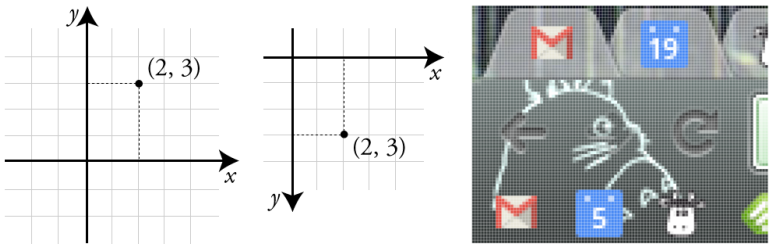


Figure 5.1: Figure 1: 2D screen coordinates

Now that we can talk about locations, let's jump into code. Create an openFrameworks project and call it "BasicShapes" (or something more imaginative). Open the source file, `ofApp.cpp`, and navigate to the `draw()` function. Add the following:

```
ofBackground(0);  // Clear the screen with a black color
ofSetColor(255);  // Set the drawing color to white

// Draw some shapes
ofRect(50, 50, 100, 100); // Top left corner at (50, 50), 100 wide x
    100 high
ofCircle(250, 100, 50); // Centered at (250, 100), radius of 50
ofEllipse(400, 100, 80, 100); // Centered at (400 100), 80 wide x
    100 high
ofTriangle(500, 150, 550, 50, 600, 150); // Three corners: (500,
    150), (550, 50), (600, 150)
ofLine(700, 50, 700, 150); // Line from (700, 50) to (700, 150)
```

When we run the code, we see white shapes on a black background. Success! Each time our `draw()` function executes, three things happen. First, we clear the screen by drawing a solid black background using `ofBackground(...)`[7]. The `0` represents a grayscale

[6] http://en.wikipedia.org/wiki/Cartesian_coordinate_system
[7] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofBackground

color where `0` is completely black and `255` is completely white. Second, we specify what color should be used for drawing with `ofSetColor(...)`[8]. We can think of this code as telling openFrameworks to pull out a specific colored marker. When we draw, we will draw in that color until we specify that we want another color. Third, we draw our basic shapes with `ofRect(...)`, `ofCircle(...)`, `ofEllipse(...)`, `ofTriangle(...)` and `ofLine(...)`. Check out the comments in the example to better understand how we are using the drawing functions. The functions can be used in other ways as well, so check out the openFrameworks documentation if you are curious.

`ofFill()`[9] and `ofNoFill()`[10] toggle between drawing filled shapes and drawing outlines. The colored marker analogy doesn't fit, but the concept still applies. `ofFill()` tells openFrameworks to draw filled shapes until told otherwise. `ofNoFill()` does the same but with outlines. So we can draw two rows of shapes on our screen (figure 2) - one filled and one outlines - if we modify our `draw()` function to look like:

```
ofFill(); // If we omit this and leave ofNoFill(), all the shapes
    will be outlines!
// Draw some shapes (code omitted)

ofNoFill(); // If we omit this and leave ofFill(), all the shapes
    will be filled!
// Draw some shapes (code omitted)
```

The circle and ellipse are looking a bit jagged, so we can fix that with `ofSetCircleResolution(...)`[11]. Circles and ellipses are drawn by connecting a series of points with straight lines. If we take a close look at the circle in figure 2, and we'll be able to identify the 20 tiny straight lines. That's the default resolution. Try putting `ofSetCircleResolution(50)` in the `setup()` function.

The individual lines that make up our outlines can be jagged too. We can fix that with a smoothing technique called anti-aliasing[12]. We probably don't need to worry about this since anti-aliasing will be turned on by default in recent versions of openFrameworks. If it isn't, just add `ofEnableAntiAliasing()`[13] to `setup()`. (For future reference, you can turn it off to save computing power: `ofDisableAntiAliasing()`[14].)

[Source code for this section[15]]

[8] http://openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofSetColor
[9] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofFill
[10] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofFill
[11] http://openframeworks.cc/documentation/gl/ofGLProgrammableRenderer.html#!show_setCircleResolution
[12] http://en.wikipedia.org/wiki/Spatial_anti-aliasing
[13] http://openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofEnableAntiAliasing
[14] http://openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofDisableAntiAliasing
[15] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/1_i_Basic_Shapes

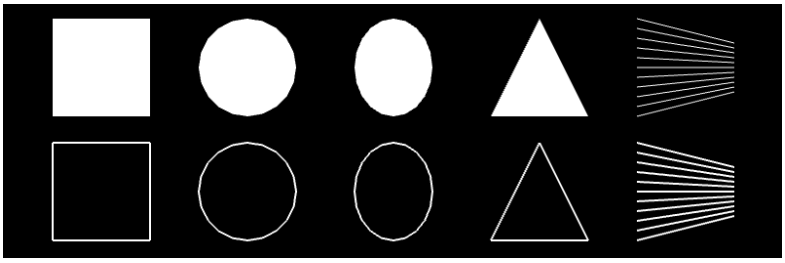


Figure 5.2: Figure 2: Basic shapes with and without a fill

[ofSketch file for this section[16]]

Extensions

1. We can change the thickness of lines using `ofSetLineWidth(...)`[17]. The default thickness is 1. We use this function like `ofFill()` and `ofSetColor(...)` in that it changes the thickness of the "marker" we use to draw lines. Note: the range of widths supported by this feature is dependent on your graphics card, so if it's not working, it might not be your fault!
2. Draw some rounded rectangles using `ofRoundedRect(...)`[18].
3. Explore the world of curved lines with `ofCurve(...)`[19] and `ofBezier(...)`[20]. You can control the resolution using `ofSetCurveResolution(...)`[21].

5.1.2 Brushes from Basic Shapes

We survived the boring bits, but why draw one rectangle, when we can draw a million (figure 3)? That is essentially what we will be doing in this section. We will build brushes that drop a burst of many small shapes whenever we press the left mouse button. To make things more exciting, we will mix in some randomness. Start a new openFrameworks project, called "ShapeBrush."

5.1.2.1 Single Rectangle Brush: Using the Mouse

We are going to lay down the foundation for our brushes by making a simple one that draws a single rectangle when we hold down the mouse. To get started, we are going to need to know 1) the mouse location and 2) if the left mouse button is pressed.

[16] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_i_Basic_Shapes.sketch
[17] http://openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofSetLineWidth
[18] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofRectRounded
[19] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofCurve
[20] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofBezier
[21] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofSetCurveResolution

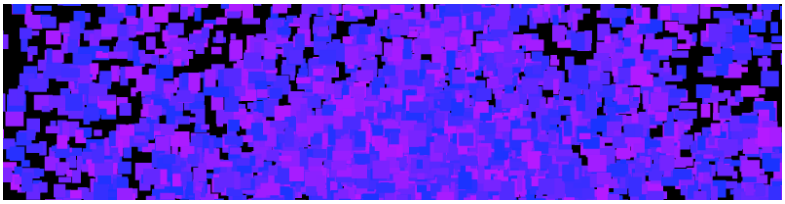


Figure 5.3: Figure 3: Okay, not actually a million rectangles

For 1), we can use two openFrameworks functions that return `int` variables: `ofGetMouseX()`[22] and `ofGetMouseY()`[23]. We will use them in `draw()`.

For 2), we can find out whether the left mouse button is pressed using `ofGetMousePressed(...)`[24]. The function asks us to pass in an `int` that represents which mouse button is we want to know about. openFrameworks provides some "public constants" for use here: `OF_MOUSE_BUTTON_LEFT`, `OF_MOUSE_BUTTON_MIDDLE` and `OF_MOUSE_BUTTON_RIGHT`. These public constants are just `int` variables that cannot be changed and can be accessed anywhere you have included openFrameworks. So `ofGetMousePressed(OF_MOUSE_BUTTON_LEFT)` will return `true` if the left button is pressed and will return `false` otherwise.

Let's add some graphics. Hop over to the `draw()` function where we can bring these new functions together:

```
if (ofGetMousePressed(OF_MOUSE_BUTTON_LEFT)) {  // If the left mouse
    button is pressed...
    ofSetColor(255);
    ofSetRectMode(OF_RECTMODE_CENTER);
    ofRect(ofGetMouseX(), ofGetMouseY(), 50, 50);  // Draw a 50 x 50
        rect centered over the mouse
}
```

`ofSetRectMode(...)`[25] allows us to control how the `(x, y)` we pass into `ofRect(...)` are used to draw. By default, they are interpreted as the upper left corner (`OF_RECTMODE_CORNER`). For our purposes, we want them to be the center (`OF_RECTMODE_CENTER`), so our rectangle is centered over the mouse.

Compile and run. A white rectangle is drawn at the mouse position when we press the left mouse button...but it disappears immediately. By default, the screen is cleared with every `draw()` call. We can change that with `ofSetBackgroundAuto(...)`[26]. Passing in a value of `false` turns off the automatic background clearing. Add the following lines into `setup()`:

[22] http://www.openframeworks.cc/documentation/events/ofEvents.html#show_ofGetMouseX
[23] http://www.openframeworks.cc/documentation/events/ofEvents.html#show_ofGetMouseY
[24] http://www.openframeworks.cc/documentation/events/ofEvents.html#show_ofGetMousePressedd
[25] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofSetRectMode
[26] http://openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofSetBackgroundAuto

```
ofSetBackgroundAuto(false);

// We still want to draw on a black background, so we need to draw
// the background before we do anything with the brush
ofBackground(0);
```

First brush, done! We are going to make this a bit more interesting by adding 1) randomness and 2) repetition.

Randomness can make our code dark, mysterious and unpredictable. Meet `ofRandom(...)`[27]. It can be used in two different ways: by passing in two values `ofRandom(float min, float max)` or by passing in a single value `ofRandom(float max)` where the min is assumed to be `0`. The function returns a random value between the min and max. We can inject some randomness into our rectangle color (figure 4) by using:

```
float randomColor = ofRandom(50, 255);
ofSetColor(randomColor);  // Exclude dark grayscale values (0 - 50)
    that won't show on black background
```

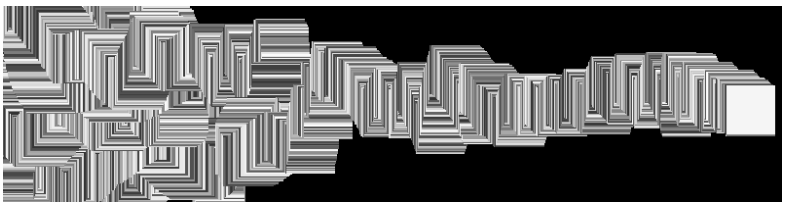


Figure 5.4: Figure 4: Drawing a rectangle snake

To finish off this single rectangle brush, let's add the ability to erase by pressing the right mouse button by adding this to our `draw()` function:

```
if (ofGetMousePressed(OF_MOUSE_BUTTON_RIGHT)) {  // If the right
    mouse button is pressed...
     ofBackground(0);  // Draw a black background over the screen
}
```

Source code for this section[28]]

[ofSketch file for this section[29]]

[27] http://openframeworks.cc/documentation/math/ofMath.html#!show_ofRandom
[28] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/1_ii_a_Single_Rectangle_Brush
[29] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_ii_a_Single_Rectangle_Brush.sketch

5.1.2.2 Bursting Rectangle Brush: Creating Randomized Bursts

We now have the basics in place for a brush, but instead of drawing a single rectangle in `draw()`, let's draw a burst of randomized rectangles. We are going use a `for` loop to create multiple rectangles whose parameters are randomly chosen. What can we randomize? Grayscale color, width and height are easy candidates. We can also use a small positive or negative value to offset each rectangle from mouse position. Modify `draw()` to look like this:

```
if (ofGetMousePressed(OF_MOUSE_BUTTON_LEFT)) {  // If the left mouse
    button is pressed...
     ofSetRectMode(OF_RECTMODE_CENTER);
     int numRects = 10;
     for (int r=0; r<numRects; r++) {
         ofSetColor(ofRandom(50, 255));
         float width = ofRandom(5, 20);
         float height = ofRandom(5, 20);
         float xOffset = ofRandom(-40, 40);
         float yOffset = ofRandom(-40, 40);
         ofRect(ofGetMouseX()+xOffset, ofGetMouseY()+yOffset, width,
            height);
     }
}
```

But! Add one more thing, inside of `setup()`, before hitting run: `ofSetFrameRate(60)`. The frame rate is the speed limit of our program, frames per second (fps). `update()` and `draw()` will not run more than `60` times per second. (ofSketch users - we'll talk about `update()` later.) Note: this is a speed *limit*, not a speed *minimum* - our code can run slower than `60` fps. We set the frame rate in order to control how many rectangles will be drawn. If `10` rectangles are drawn with the mouse pressed and we know `draw()` won't be called more than `60` times per second, then we will generate a max of `600` rectangles per second.

Compile, run. We get a box-shaped spread of random rectangles (figure 5, left). Why didn't we get a circular spread (figure 5, right)? Since `xOffset` and `yOffset` could be any value between `-40` and `40`, think about what happens when `xOffset` and `yOffset` take on their most extreme values, i.e. (xOffset, yOffset) values of (-40, -40), (40, -40), (40, 40) and (-40, 40).

If we want a random point within a circle, it helps to think in terms of angles. Imagine we are at the center of a circle. If we rotate a random amount (the *polar angle*) and then move a random distance (the *polar radius*), we end up in a random location within the circle (assuming we don't walk so far that we cross the boundary of our circle). We just defined a point by a polar angle and a polar radius instead of using `(x, y)`. We have just thought about space in terms of polar coordinates[30], instead of Cartesian

[30] http://en.wikipedia.org/wiki/Polar_coordinate_system

coordinates.

Back to the code. When we figure out our offsets, we want to pick a random direction (polar angle) and random distance (polar distance) which we can then convert to Cartesian coordinates (see code) to use as `xOffset` and `yOffset`. Our loop inside of `draw()` will look like this:

```
for (int r=0; r<numRects; r++) {
    ofSetColor(ofRandom(50, 255));
    float width = ofRandom(5, 20);
    float height = ofRandom(5, 20);
    float distance = ofRandom(35);

    // Formula for converting from polar to Cartesian coordinates:
    //  x = cos(polar angle) * (polar distance)
    //  y = sin(polar angle) * (polar distance)

    // We need our angle to be in radians if we want to use sin() or
       cos()
    // so we can make use of an openFrameworks function to convert
       from degrees
    // to radians
    float angle = ofRandom(ofDegToRad(360.0));

    float xOffset = cos(angle) * distance;
    float yOffset = sin(angle) * distance;
    ofRect(ofGetMouseX()+xOffset, ofGetMouseY()+yOffset, width,
       height);
}
```

Figure 5.5: Figure 5: Cartesian brush spread versus polar brush spread

[Source code for this section[31]]

[ofSketch file for this section[32]]

[31] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/1_ii_b_Bursting_Rect_Brush

[32] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_ii_a_Single_Rectangle_Brush.sketch

5.1.2.3 Glowing Circle Brush: Using Transparency and Color

Unlike what we did with the rectangle brush, we are going to layer colorful, transparent circles on top of each to create a glowing haze. We will draw a giant transparent circle, then draw a slightly smaller transparent circle on top of it, then repeat, repeat, repeat. We can add transparency to `ofSetColor(...)` with a second parameter, the alpha channel (e.g.`ofSetColor(255, 50)`), with a value from `0` (completely transparent) to `255` (completely opaque).

Before we use alpha, we need to enable something called "alpha blending." Using transparency costs computing power, so `ofEnableAlphaBlending()`[33] and `ofDisableAlphaBlending()`[34] allow us to turn on and off this blending at our discretion. We need it, so enable it in `setup()`.

Comment out the rectangle brush code inside the `if` statement that checks if the left mouse button is pressed. Now we can start working on our circle brush. We will use the `angle`, `distance`, `xOffset` and `yOffset` code like before. Our `for` loop will start with a large radius and step its value to `0`. Add the following:

```
int maxRadius = 100;  // Increase for a wider brush
int radiusStepSize = 5;  // Decrease for more circles (i.e. a more
    opaque brush)
int alpha = 3;  // Increase for a more opaque brush
int maxOffsetDistance = 100;  // Increase for a larger spread of
    circles
for (int radius=maxRadius; radius>0; radius-=radiusStepSize) {
    float angle = ofRandom(ofDegToRad(360.0));
    float distance = ofRandom(maxOffsetDistance);
    float xOffset = cos(angle) * distance;
    float yOffset = sin(angle) * distance;
    ofSetColor(255, alpha);
    ofCircle(ofGetMouseX()+xOffset, ofGetMouseY()+yOffset, radius);
}
```

We end up with something like figure 6, a glowing light except without color. Tired of living in moody shades of gray? `ofSetColor(...)` can make use of the Red Blue Green (RGB) color model[35] in addition to the grayscale color model. We specify the amount (`0` to `255`) of red, blue and green light respectively, e.g. `ofSetColor(255, 0, 0)` for opaque red. We can also add alpha, e.g. `ofSetColor(0, 0, 255, 10)` for transparent blue. Go ahead and modify the `ofSetColor(...)` in our circle brush to use a nice orange: `ofSetColor(255, 103, 0, alpha)`.

[33] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofEnableAlphaBlending
[34] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofDisableAlphaBlending
[35] http://en.wikipedia.org/wiki/RGB_color_model

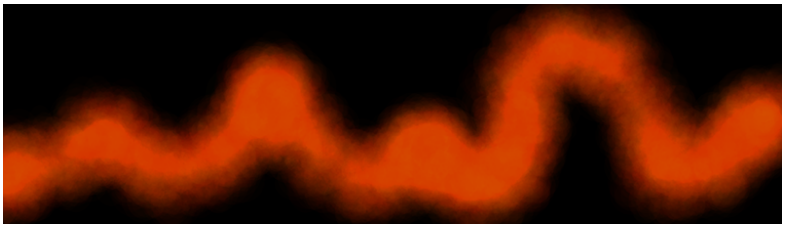


Figure 5.6: Figure 6: Results of using the circle glow brush

There's another way we can use `ofSetColor(...)`. Meet `ofColor`[36], a handy class for handling colors which allows for fancy color math (among other things). Here are some examples of defining and modifying colors:

```
ofColor myOrange(255, 132, 0); // Defining an opaque orange color -
    specified using RGB
ofColor myBlue(0, 0, 255, 50); // Defining a transparent blue color
    - specified using RGBA

// We can access the red, green, blue and alpha channels like this:
ofColor myGreen(0, 0, 255, 255);
cout << "Red channel:" << myGreen.r << endl;
cout << "Green channel:" << myGreen.g << endl;
cout << "Blue channel:" << myGreen.b << endl;
cout << "Alpha channel:" << myGreen.a << endl;

// We can also set the red, green, blue and alpha channels like this:
ofColor myYellow;
myYellow.r = 255;
myYellow.b = 0;
myYellow.g = 255;
myYellow.a = 255;

// We can also make use of some predefined colors provided by
    openFrameworks:
ofColor myAqua = ofColor::aqua;
ofColor myPurple = ofColor::plum;
// Full list of colors available at:
    http://www.openframeworks.cc/documentation/types/ofColor.html
```

If we wanted to make our brush fierier, we would draw using random colors that are in-between orange and red. `ofColor` gives us in-betweenness using something called "linear interpolation[37]" with a function called `getLerped(...)`[38]. `getLerped(...)` is a class method of `ofColor`, which means that if we have an `ofColor` variable, we

[36] http://openframeworks.cc/documentation/types/ofColor.html
[37] http://en.wikipedia.org/wiki/Linear_interpolation
[38] http://www.openframeworks.cc/documentation/types/ofColor.html#show_getLerped

can interpolate like this: `myFirstColor.getLerped(mySecondColor, 0.3)`. (For an explanation of classes and methods, see the *OOPS!* chapter.) We pass in two arguments, an `ofColor` and a `float` value between `0.0` and `1.0`. The function returns a new `ofColor` that is between the two specified colors, and the `float` determines how close the new color is to our original color (here, `myFirstColor`). We can use this in `draw()` like this:

```
ofColor myOrange(255, 132, 0, alpha);
ofColor myRed(255, 6, 0, alpha);
ofColor inBetween = myOrange.getLerped(myRed, ofRandom(1.0));
ofSetColor(inBetween);
```

[Source code for this section[39]]

[ofSketch file for this section[40]]

5.1.2.4 Star Line Brush: Working with a Linear Map

What about lines? We are going to create a brush that draws lines that radiate out from the mouse to create something similar to an asterisk or a twinkling star (figure 7). Comment out the circle brush and add:

```
int numLines = 30;
int minRadius = 25;
int maxRadius = 125;
for (int i=0; i<numLines; i++) {
    float angle = ofRandom(ofDegToRad(360.0));
    float distance = ofRandom(minRadius, maxRadius);
    float xOffset = cos(angle) * distance;
    float yOffset = sin(angle) * distance;
    float alpha = ofMap(distance, minRadius, maxRadius, 50, 0);  //
       Make shorter lines more opaque
    ofSetColor(255, alpha);
    ofLine(ofGetMouseX(), ofGetMouseY(), ofGetMouseX()+xOffset,
       ofGetMouseY()+yOffset);
}
```

What have we done with the alpha? We used `ofMap(...)`[41] to do a linear interpolation, similar to `getLerped(...)`. `ofMap(...)` transforms one range of values into a different range of values - like taking the "loudness" of a sound recorded on a microphone and using it to determine the color of a shape drawn on the screen. To get a "twinkle"

[39] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/1_ii_c_Glowing_Circle_Brush

[40] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_ii_c_Glowing_Circle_Brush.sketch

[41] http://www.openframeworks.cc/documentation/math/ofMath.html#show_ofMap

effect, we want our shortest lines to be the most opaque and our longer lines to be the most transparent. `ofMap(...)` takes a value from one range and maps it into another range like this: `ofMap(value, inputMin, inputMax, outputMin, outputMax)`. We tell it that distance is a `value` in-between `minRadius` and `maxRadius` and that we want it mapped so that a distance value of 125 (`maxRadius`) returns an alpha value of 50 and a distance value of 25 (`minRadius`) returns an alpha value of 0.

We can also vary the line width using: `ofSetLineWidth(ofRandom(1.0, 5.0))`, but remember that if we change the line width in this brush, we will need go back and set our line width back to `1.0` in our other brushes.

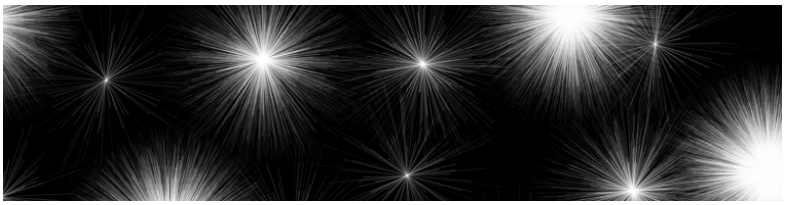


Figure 5.7: Figure 7: Results from using the line brush

[Source code for this section[42]]

[ofSketch file for this section[43]]

5.1.2.5 Fleeing Triangle Brush: Vectors and Rotations

Time for the last brush in section 1: the triangle. We'll draw a bunch of triangles that are directed outward from the mouse position (figure 8, left). `ofTriangle(...)` requires us to specify the three corners of the triangle, which means that we will need to calculate the rotation of the corners to make the triangle point away from the mouse. A new class will make that math easier: `ofVec2f`[44].

We've been defining points by keeping two separate variables: x and y. `ofVec2f` is a 2D vector, and for our purposes, we can just think of it as a point in 2D space. `ofVec2f` allows us to hold both x and y in a single variable (and perform handy math operations):

```
ofVec2f mousePos(ofGetMouseX(), ofGetMouseY());  // Defining a new
    ofVec2f

// Access the x and y coordinates like this:
cout << "Mouse␣X:␣" << mousePos.x << endl;
```

[42] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/1_ii_d_Star_Line_Brush

[43] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_ii_d_Star_Line_Brush.sketch

[44] http://openframeworks.cc/documentation/math/ofVec2f.html

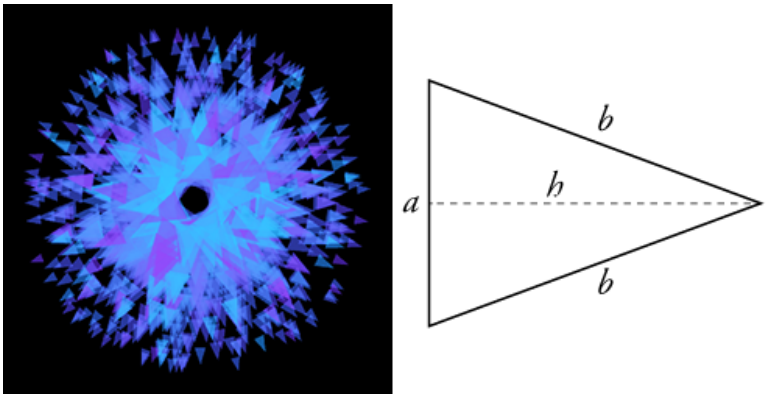


Figure 5.8: Figure 8: Isosceles triangles in a brush, left, and isolated in a diagram, right

```
cout << "Mouse␣Y:␣" << mousePos.y << endl;

// Or we can modify the coordinates like this:
float xOffset = 10.0;
float yOffset = 30.0;
mousePos.x += xOffset;
mousePos.y += yOffset;

// But we can do what we just did above by adding or subtracting two
   vectors directly
ofVec2f offset(10.0, 30.0);
mousePos += offset;
```

Let's start using it to build the triangle brush. The first step is to draw a triangle (figure 8, right) at the mouse cursor. It will become important later, but we are going to draw our triangle starting from the mouse cursor and pointing to the right. Comment out the line brush, and add:

```
ofVec2f mousePos(ofGetMouseX(), ofGetMouseY());

// Define a triangle at the origin (0,0) that points to the right
ofVec2f p1(0, 25.0);
ofVec2f p2(100, 0);
ofVec2f p3(0, -25.0);

// Shift the triangle to the mouse position
p1 += mousePos;
p2 += mousePos;
p3 += mousePos;

ofSetColor(255, 50);
ofTriangle(p1, p2, p3);
```

Run it and see what happens. We can add rotation with the `ofVec2f` class method `rotate(...)`[45] like this: `myPoint.rotate(45.0)` where `myPoint` is rotated around the origin, `(0, 0)`, by `45.0` degrees. Back to our code, add this right before shifting the triangle to the mouse position:

```
// Rotate the triangle points around the origin
float rotation = ofRandom(360); // The rotate function uses degrees!
p1.rotate(rotation);
p2.rotate(rotation);
p3.rotate(rotation);
```

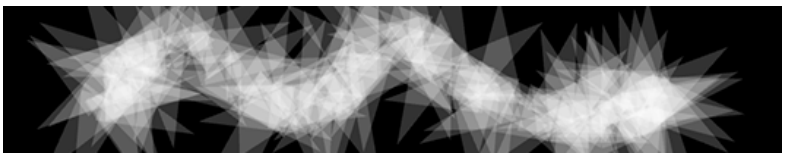


Figure 5.9: Figure 9: Results from using the rotating triangle brush

Our brush looks something like figure 8 (left). If we were to move that rotation code to *after* we shifted the triangle position, the code wouldn't work very nicely because `rotate(...)` assumes we want to rotate our point around the origin. (Check out the documentation for an alternate way to use `rotate(...)` that rotates around an arbitrary point.) Last step, let's integrate our prior approach of drawing multiple shapes that are offset from the mouse:

```
ofVec2f mousePos(ofGetMouseX(), ofGetMouseY());

int numTriangles = 10;
int minOffset = 5;
int maxOffset = 70;
int alpha = 150;
for (int t=0; t<numTriangles; ++t) {
    float offsetDistance = ofRandom(minOffset, maxOffset);

    // Define a triangle at the origin (0,0) that points to the
       right (code omitted)
    // The triangle size is a bit smaller than the last brush - see
       the source code

    // Rotate the triangle, then shift it to the mouse position
       (code omitted)

    ofVec2f triangleOffset(offsetDistance, 0.0);
    triangleOffset.rotate(rotation);
```

[45] http://www.openframeworks.cc/documentation/math/ofVec2f.html#show_rotate

```
    p1 += mousePos + triangleOffset;
    p2 += mousePos + triangleOffset;
    p3 += mousePos + triangleOffset;

    ofSetColor(255, alpha);
    ofTriangle(p1, p2, p3);
}
```

We are now using `ofVec2f` for our offset. We started with a vector that points rightward, the same direction our triangle starts out pointing. When we apply the rotation to them both, they stay in sync (i.e. both pointing away from the mouse). We can push them out of sync with: `triangleOffset.rotate(rotation+90)`, and we get a swirling blob of triangles. After that, we can add some color using `ofRandom(...)` and `getLerped(...)` again (figure 9) or play with fill and line width.

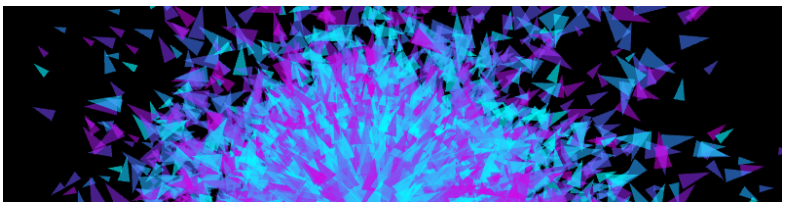


Figure 5.10: Figure 10: Results from using the final triangle brush

[Source code for this section[46]]

[ofSketch file for this section[47]]

Extensions

1. Define some public variables to control brush parameters like `transparency`, `brushWidth`, `offsetDistance`, `numberOfShapes`, etc.
2. Use the `keyPressed(int key)`[48] function (in `.cpp`) to control those parameters at run time (e.g. increasing/decreasing `brushWidth` with the `+` and `-` keys). If you are using ofSketch, see the next section for how to use that function.
3. Track the mouse position and use the distance it moves between frames to control those parameters (e.g. fast moving mouse draws a thicker brush).

[46] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/1_ii_e_Triangle_Brush
[47] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_ii_e_Triangle_Brush.sketch
[48] http://www.openframeworks.cc/documentation/application/ofBaseApp.html#!show_keyPressed

5.1.2.6 Raster Graphics: Taking a Snapshot

Before we move on, let's save a snapshot of our canvas. We'll want to use the `keyPressed(int key)`[49] function. This function is built into your application by default. Any time a key is pressed, the code you put into this function is called. The `key` variable is an integer that represents the key that was pressed.

If you are using project generator, you'll find `keyPressed(...)` in your `.cpp` file. If you are using ofSketch, you might not see the function, but it is easy to add. See the ofSketch file[50] for the last section.

In the `keyPressed(...)` function, add the following:

```
if (key == 's') {
    // It's strange that we can compare the int key to a character
       like `s`, right?  Well, the super short
    // explanation is that characters are represented by numbers in
       programming.  `s` and 115 are the same
    // thing.  If you want to know more, look check out the wiki for
       ASCII.
    glReadBuffer(GL_FRONT);  // HACK: only needed on windows, when
       using ofSetAutoBackground(false)
    ofSaveScreen("savedScreenshot_"+ofGetTimestampString()+".png");
}
```

`ofSaveScreen(...)`[51] grabs the current screen and saves it to a file inside of our project's `./bin/data/` folder with a filename we specify. The timestamp is used to create a unique filename, allowing us to save multiple screenshots without worrying about them overriding each other. So press the `s` key and check out your screenshot!

5.2 Brushes from Freeform Shapes

In the last section, we drew directly onto the screen. We were storing graphics (brush strokes) as pixels, and therefore working with raster graphics[52]. For this reason, it is hard to isolate, move or erase a single brush stroke. It also means we can't re-render our graphics at a different resolution. In contrast, vector graphics[53] store graphics as a list of geometric objects instead of pixel values. Those objects can be modified (erased, moved, rescaled, etc.) after we "place" them on our screen.

[49] http://www.openframeworks.cc/documentation/application/ofBaseApp.html#!show_keyPressed
[50] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/1_ii_e_Triangle_Brush.sketch
[51] http://www.openframeworks.cc/documentation/utils/ofUtils.html#show_ofSaveScreen
[52] http://en.wikipedia.org/wiki/Raster_graphics
[53] http://en.wikipedia.org/wiki/Vector_graphics

In this section, we are going to make a kind of vector graphics by using custom ("freeform") shapes in openFrameworks. We will use structures (`ofPolyline` and `vector<ofPolyline>`) that allow us to store and draw the path that the mouse takes on the screen. Then we will play with those paths to create brushes that do more than just trace out the cursor's movement.

5.2.1 Basic Polylines

Create a new project called "Polylines," and say hello to `ofPolyline`[54]. `ofPolyline` is a data structure that allows us to store a series of sequential points and then connect them to draw a line. Let's dive into some code. In your header file, define three `ofPolylines`:

```
ofPolyline straightSegmentPolyline;
ofPolyline curvedSegmentPolyline;
ofPolyline closedShapePolyline;
```

We can fill those `ofPolylines` with points in `setup()`:

```
straightSegmentPolyline.addVertex(100, 100);  // Add a new point:
    (100, 100)
straightSegmentPolyline.addVertex(150, 150);  // Add a new point:
    (150, 150)
straightSegmentPolyline.addVertex(200, 100);  // etc...
straightSegmentPolyline.addVertex(250, 150);
straightSegmentPolyline.addVertex(300, 100);

curvedSegmentPolyline.curveTo(350, 100);  // These curves are
    Catmull-Rom splines
curvedSegmentPolyline.curveTo(350, 100);  // Necessary Duplicate for
    Control Point
curvedSegmentPolyline.curveTo(400, 150);
curvedSegmentPolyline.curveTo(450, 100);
curvedSegmentPolyline.curveTo(500, 150);
curvedSegmentPolyline.curveTo(550, 100);
curvedSegmentPolyline.curveTo(550, 100);  // Necessary Duplicate for
    Control Point

closedShapePolyline.addVertex(600, 125);
closedShapePolyline.addVertex(700, 100);
closedShapePolyline.addVertex(800, 125);
closedShapePolyline.addVertex(700, 150);
closedShapePolyline.close();  // Connect first and last vertices
```

We can now draw our polylines in the `draw()` function:

[54] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html

```
ofBackground(0);
ofSetLineWidth(2.0);  // Line widths apply to polylines
ofSetColor(255,100,0);
straightSegmentPolyline.draw();  // This is how we draw polylines
curvedSegmentPolyline.draw();  // Nice and easy, right?
closedShapePolyline.draw();
```

We created three different types of polylines (figure 11). `straightSegmentPolyline` is composed of a series points connected with straight lines. `curvedSegmentPolyline` uses the same points but connects them with curved lines. The curves that are created are Catmull–Rom splines[55], which use four points to define a curve: two define the start and end, while two control points determine the curvature. These control points are the reason why we need to add the first and last vertex twice. Lastly, `closedShapePolyline` uses straight line segments that are closed, connecting the first and last vertices.

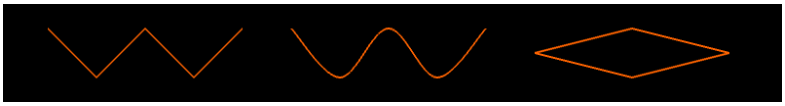


Figure 5.11: Figure 11: Examples of polylines - straight, curved and closed straight

The advantage of drawing in this way (versus raster graphics) is that the polylines are modifiable. We could easily move, add, delete, scale our vertices on the the fly.

[Source code for this section[56]]

[ofSketch file for this section[57]]

Extensions

1. Check out `arc(...)`[58], `arcNegative(...)`[59] and `bezierTo(...)`[60] for other ways to draw shapes with `ofPolyline`.

[55] http://en.wikipedia.org/wiki/Centripetal_Catmull%E2%80%93Rom_spline
[56] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/2_i_Basic_Polylines
[57] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/2_i_Basic_Polylines.sketch
[58] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_arc
[59] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_arcNegative
[60] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_bezierTo

5.2.2 Building a Brush from Polylines

5.2.2.1 Polyline Pen: Tracking the Mouse

Let's use polylines to draw brush strokes. Create a new project, "PolylineBrush." When the left mouse button is held down, we will create an `ofPolyline` and continually extend it to the current mouse position. We will use a `bool` to tell us if the left mouse button is being held down. If it is being held down, we'll add the mouse position to the polyline, but instead of adding *every* mouse position, we'll add the mouse positions where the mouse has moved a distance away from the last point in our polyline.

Let's move on to the code. Create four variables in the header:

```
ofPolyline currentPolyline;
bool leftMouseButtonPressed;
ofVec2f lastPoint;
float minDistance;
```

Initialize `minDistance` and `currentedAddingPoints` in `setup()`:

```
minDistance = 10;
currentlyAddingPoints = false;
```

Now we are going to take advantage of two new functions - `mousePressed(int x, int y, int button)`[61] and `mouseReleased(int x, int y, int button)`[62]. These are functions that are built into your application by default. They are event handling functions, so whenever a mouse button is pressed, whatever code you put into `mousePressed(...)` is called. It's important to note that `mousePressed(...)` is only called when the mouse button is initially pressed. No matter how long we hold the mouse button down, the function is still only called once. The same goes for `mouseReleased(...)`.

The functions have a few variables `x`, `y` and `button` that allow you to know a bit more about the particular mouse event that just occurred. `x` and `y` are the screen coordinates of the cursor, and `button` is an `int` that represents the particular button on the mouse that was pressed/released. Remember the public constants like `OF_MOUSE_BUTTON_LEFT` and `OF_MOUSE_BUTTON_RIGHT`? To figure out what `button` is, we'll compare it against those constants.

Let's turn back to the code. If you are using project generator, you'll find these mouse functions in your `.cpp` file. If you are using ofSketch, you might not see these functions, but they are easy to add. See the ofSketch file[63] for this section. Inside of `mousePressed(...)`, we want to start the polyline:

[61] http://www.openframeworks.cc/documentation/application/ofBaseApp.html#!show_mousePressed
[62] http://www.openframeworks.cc/documentation/application/ofBaseApp.html#show_mouseReleased
[63] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/2_ii_a_Polyline_Pen.sketch

```
if (button == OF_MOUSE_BUTTON_LEFT) {
    leftMouseButtonPressed = true;
    currentPolyline.curveTo(x, y);  // Remember that x and y are the
       location of the mouse
    currentPolyline.curveTo(x, y);  // Necessary duplicate for first
       control point
    lastPoint.set(x, y);  // Set the x and y of a ofVec2f in a
       single line
}
```

Inside of `mouseReleased(...)`, we want to end the polyline:

```
if (button == OF_MOUSE_BUTTON_LEFT) {
    leftMouseButtonPressed = false;
    currentPolyline.curveTo(x, y); // Necessary duplicate for last
       control point
    currentPolyline.clear();  // Erase the vertices, allows us to
       start a new brush stroke
}
```

Now let's move over to the `update()` function. For ofSketch users, this another default function that you might not see in your sketch. It is a function that is called once per frame, and it is intended for doing non-drawing tasks. It's easy to add - see the ofSketch file[64] for this section.

Let's add points to our polyline in `update()`:

```
if (leftMouseButtonPressed) {
    ofVec2f mousePos(ofGetMouseX(), ofGetMouseY());
    if (lastPoint.distance(mousePos) >= minDistance) {
        // a.distance(b) calculates the Euclidean distance between
           point a and b.  It's
        // the straight line distance between the points.
        currentPolyline.curveTo(mousePos);  // Here we are using an
           ofVec2f with curveTo(...)
        lastPoint = mousePos;
    }
}
```

Note that this only adds points when the mouse has moved a certain threshold amount (`minDistance`) away from the last point we added to the polyline. This uses the `distance(...)`[65] method of `ofVec2f`.

All that is left is to add code to draw the polyline in `draw()`, and we've got a basic curved polyline drawing program. But we don't have the ability to save multiple polylines, so

[64] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/2_ii_a_Polyline_Pen.sketch

[65] http://openframeworks.cc/documentation/math/ofVec2f.html#show_distance

we have something similar to an Etch A Sketch. We can only draw a single, continuous line. In order to be able to draw multiple lines that don't have to be connected to each other, we will turn to something called a `vector`. This isn't the same kind of vector that we talked about earlier in the context of `of2Vecf`. If you haven't seen vectors before, check out the stl::vector basics tutorial[66] on the site.

Define `vector <ofPolyline> polylines` in the header. We will use it to save our polyline brush strokes. When we finish a stroke, we want to add the polyline to our vector. So in the if statement inside of `mouseReleased(...)`, before `currentPolyline.clear()`, add `polylines.push_back(currentPolyline)`. Then we can draw the polylines like this:

```
ofBackground(0);
ofSetColor(255);  // White color for saved polylines
for (int i=0; i<polylines.size(); i++) {
    ofPolyline polyline = polylines[i];
    polyline.draw();
}
ofSetColor(255,100,0);  // Orange color for active polyline
currentPolyline.draw();
```

And we have a simple pen-like brush that tracks the mouse, and we can draw a dopey smiley face (figure 12).

Figure 5.12: Figure 12: Drawing a smilie with the polyline brush

[Source code for this section[67]]

[ofSketch file for this section[68]]

Extensions

1. Add color!
2. Explore `ofBeginSaveScreenAsPDF(...)`[69] and `ofEndSaveScreenAsPDF(...)`[70]

[66] http://openframeworks.cc/tutorials/c++%20concepts/001_stl_vectors_basic.html
[67] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/2_ii_a_Polyline_Pen
[68] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/2_ii_a_Polyline_Pen.sketch
[69] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofBeginSaveScreenAsPDF
[70] http://openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofEndSaveScreenAsPDF

to save your work into a vector file format.
3. Try using the `keyPressed(...)` function in your source file to add an undo feature that deletes the most recent brush stroke.
4. Try restructuring the code to allow for a redo feature as well.

5.2.2.2 Polyline Brushes: Points, Normals and Tangents

Since we have the basic drawing in place, now we play with how we are rendering our polylines. We will draw points, normals and tangents. We'll talk about what normals and tangents in a little bit. First, let's draw points (circles) at the vertices in our polylines. Inside the `for` loop in `draw()` (after `polyline.draw()`), add this:

```
vector<ofVec3f> vertices = polyline.getVertices();
for (int vertexIndex=0; vertexIndex<vertices.size(); ++vertexIndex) {
    ofVec3f vertex = vertices[vertexIndex];  // ofVec3f is like
       ofVec2f, but with a third dimension, z
    ofCircle(vertex, 5);
}
```

`getVertices()`[71] returns a `vector` of `ofVec3f` objects that represent the vertices of our polyline. This is basically what an `ofPolyline` is - an ordered set of `ofVec3f` objects (with some extra math). We can loop through the indices of the vector to pull out the individual vertex locations, and use them to draw circles.

What happens when we run it? Our white lines look thicker. That's because our polyline is jam-packed with vertices! Every time we call the `curveTo(...)` method, we create 20 extra vertices (by default). These help make a smooth-looking curve. We can adjust how many vertices are added with an optional parameter, `curveResolution`, in `curveTo(...)`. We don't need that many vertices, but instead of lowering the `curveResolution`, we can make use of `simplify(...)`[72].

`simplify(...)` is a method that will remove "duplicate" points from our polyline. We pass a single argument into it: `tolerance`, a value between 0.0 and 1.0. The `tolerance` describes how dis-similar points must be in order to be considered 'unique' enough to not be deleted. The higher the `tolerance`, the more points will be removed. So right before we save our polyline by putting it into our `polylines` vector, we can simplify it. Inside of the if statement within `mouseReleased(...)` (before `polylines.push_back(currentPolyline)`), add: `currentPolyline.simplify(0.75)`. Now we should see something like figure 13 (left).

We can also sample points along the polyline using `getPointAtPercent(...)`[73], which takes a `float` between `0.0` and `1.0` and returns a `ofVec3f`. Inside the `draw()` function, comment out the code that draws a circle at each vertex. Below that, add:

[71] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_getVertices
[72] http://openframeworks.cc/documentation/graphics/ofPolyline.html#show_simplify
[73] http://openframeworks.cc/documentation/graphics/ofPolyline.html#show_getPointAtPercent

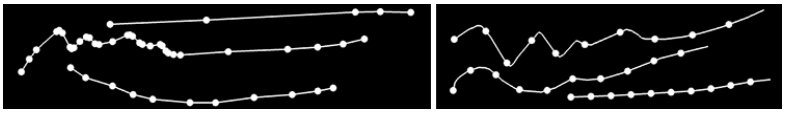


Figure 5.13: Figure 13: Drawing circles at the vertices of a polyline, without and with resampling points evenly

```
    for (int p=0; p<100; p+=10) {
        ofVec3f point = polyline.getPointAtPercent(p/100.0);  //
            Returns a point at a percentage along the polyline
        ofCircle(point, 5);
    }
```

Now we have evenly spaced points (figure 13, right). Let's try creating a brush stroke where the thickness of the line changes. To do this we need to use a normal vector[74]. Figure 14 shows normals drawn over some polylines - they points in the opposite (perpendicular) direction to the polyline. Imagine drawing a normal at every point along a polyline, like figure 15. That is one way to add "thickness" to our brush. We can comment out our circle drawing code in `draw()`, and add these lines of code instead:

```
    vector<ofVec3f> vertices = polyline.getVertices();
    float normalLength = 50;
    for (int vertexIndex=0; vertexIndex<vertices.size();
        ++vertexIndex) {
        ofVec3f vertex = vertices[vertexIndex];  // Get the vertex
        ofVec3f normal = polyline.getNormalAtIndex(vertexIndex) *
            normalLength;  // Scale the normal
        ofLine(vertex-normal/2, vertex+normal/2);  // Center the
            scaled normal around the vertex
    }
```

We are getting all of the vertices in our `ofPolyline`. But here, we are also using `getNormalAtIndex`[75] which takes an index and returns an `ofVec3f` that represents the normal vector for the vertex at that index. We take that normal, scale it and then display it centered around the vertex. So, we have something like figure 14 (left), but we can also sample normals, using the function `getNormalAtIndexInterpolated(...)`[76]. So let's comment out the code we just wrote, and try sampling our normals evenly along the polyline:

```
float numPoints = polyline.size();
float normalLength = 50;
```

[74] http://en.wikipedia.org/w/index.php?title=Normal_vector
[75] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_getNormalAtIndex
[76] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_getNormalAtIndexInterpolated

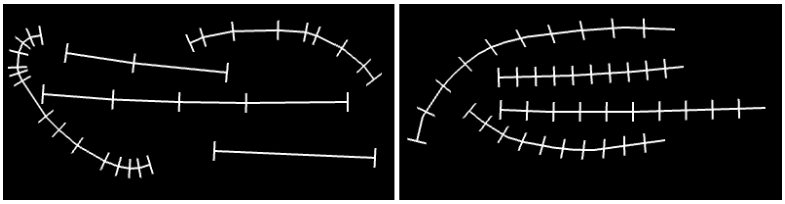


Figure 5.14: Figure 14: Drawing normals at the vertices of a polyline, without and with resampling points evenly

```
for (int p=0; p<100; p+=10) {
    ofVec3f point = polyline.getPointAtPercent(p/100.0);
    float floatIndex = p/100.0 * (numPoints-1);
    ofVec3f normal =
       polyline.getNormalAtIndexInterpolated(floatIndex) *
       normalLength;
    ofLine(point-normal/2, point+normal/2);
}
```

We can get an evenly spaced point by using percents again, but `getNormalAtIndexInterpolated(...` is asking for an index. Specifically, it is asking for a `floatIndex` which means that we can pass in 1.5 and the polyline will return a normal that lives halfway between the point at index 1 and halfway between the point at index 2. So we need to convert our percent, `p/100.0`, to a `floatIndex`. All we need to do is to multiply the percent by the last index in our polyline (which we can get from subtracting one from the `size()`[77] which tells us how many vertices are in our polyline), resulting in figure 14 (right).

Now we can pump up the number of normals in our drawing. Let's change our loop increment from `p+=10` to `p+=1`, change our loop condition from `p<100` to `p<500` and change our `p/100.0` lines of code to `p/500.0`. We might also want to use a transparent white for drawing these normals, so let's add `ofSetColor(255,100)` right before our loop. We will end up being able to draw ribbon lines, like figure 15.

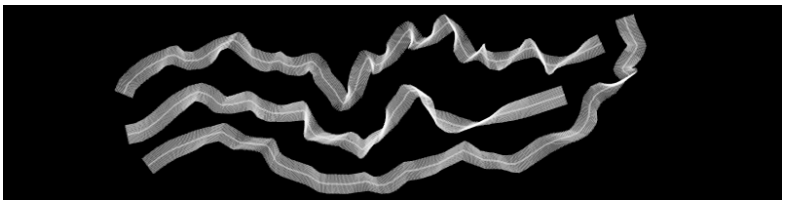


Figure 5.15: Figure 15: Drawing many many normals to fill out the polyline

We've just added some thickness to our polylines. Now let's have a quick aside about tangents, the "opposite" of normals. These wonderful things are perpendicular to the

[77] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_size

normals that we just drew. So if we drew tangents along a perfectly straight line we wouldn't really see anything. The fun part comes when we draw tangents on a curved line, so let's see what that looks like. Same drill as before. Comment out the last code and add in the following:

```
vector<ofVec3f> vertices = polyline.getVertices();
float tangentLength = 80;
for (int vertexIndex=0; vertexIndex<vertices.size(); ++vertexIndex) {
    ofVec3f vertex = vertices[vertexIndex];
    ofVec3f tangent = polyline.getTangentAtIndex(vertexIndex) *
       tangentLength;
    ofLine(vertex-tangent/2, vertex+tangent/2);
}
```

This should look very familiar except for `getTangentAtIndex(...)`[78] which is the equivalent of `getNormalAtIndex(...)` but for tangents. Not much happens for straight and slightly curved lines, however, sharply curved lines reveal the tangents figure 16 (left).

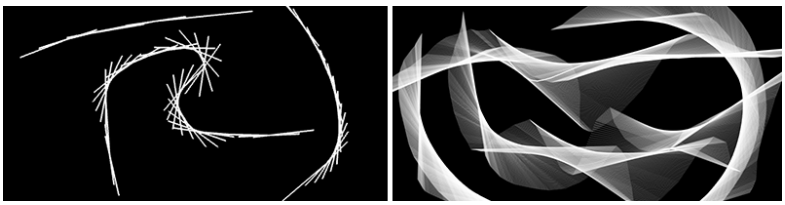


Figure 5.16: Figure 16: Drawing tangents at vertices of polylines

I'm sure you can guess what's next... drawing a whole bunch of tangents at evenly spaced locations (figure 16, right)! It's more fun that it sounds. `getTangentAtIndexInterpolated(...)`[79] works like `getNormalAtIndexInterpolated(...)`. Same drill, comment out the last code, and add the following:

```
ofSetColor(255, 50);
float numPoints = polyline.size();
float tangentLength = 300;
for (int p=0; p<500; p+=1) {
    ofVec3f point = polyline.getPointAtPercent(p/500.0);
    float floatIndex = p/500.0 * (numPoints-1);
    ofVec3f tangent =
       polyline.getTangentAtIndexInterpolated(floatIndex) *
       tangentLength;
    ofLine(point-tangent/2, point+tangent/2);
}
```

[78] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_getTangentAtIndex
[79] http://www.openframeworks.cc/documentation/graphics/ofPolyline.html#show_getTangentAtIndexInterpolated

[Source code for this section[80]]

[ofSketch file for this section[81]]

Extensions

1. Try drawing shapes other than `ofLine(...)` and `ofCircle(...)` along your polylines. You could use your brush code from section 1.
2. The density of tangents or normals drawn is dependent on the length of the brush stroke. Try making it independent (hint: you may need to adjust your loop and use `getPerimeter()` to calculate the length).
3. Check out how to draw polygons using `ofPath` and try drawing a brush stroke that is a giant, closed shape.

5.2.2.3 Vector Graphics: Taking a Snapshot (Part 2)

Remember how we saved our drawings that we made with the basic shape brushes by doing a screen capture? Well, we can also save drawings as a PDF. A PDF stores the graphics as a series of geometric objects rather than as a series of pixel values. So, if we render out our polylines as a PDF, we can open it in a vector graphics editor (like Inkscape[82] or Adobe Illustrator) and modify our polylines in all sorts of ways. For example, see figure 17 where I colored and blurred the polylines to create a glowing effect.

Once we have a PDF, we could also use it to blow up our polyines to create a massive, high resolution print.

To do any of this, we need to use `ofBeginSaveScreenAsPDF(...)`[83] and `ofEndSaveScreenAsPDF()`[84]. When we call `ofBeginSaveScreenAsPDF(...)`, any subsequent drawing commands will output to a PDF *instead of* being drawn to the screen. `ofBeginSaveScreenAsPDF(...)` takes one required argument, a `string` that contains the desired filename for the PDF. (The PDF will be saved into `./bin/data/` unless you specify an alternate path). When we call `ofEndSaveScreenAsPDF()`, the PDF is saved and drawing commands begin outputting back to the screen.

Let's use the polyline brush code from the last section to save a PDF. The way we saved a screenshot previously was to put `ofSaveScreen()` inside of `keyPressed(...)`. We can't do that here because `ofBeginSaveScreenAsPDF(...)`

[80] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/2_ii_b_Polyline_Brushes
[81] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/2_ii_b_Polyline_Brushes.sketch
[82] http://www.inkscape.org/en/
[83] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#!show_ofBeginSaveScreenAsPDF
[84] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofEndSaveScreenAsPDF

and `ofEndSaveScreenAsPDF()` need to be before and after (respectively) the drawing code. So we'll make use of a `bool` variable. Add `bool isSavingPDF` to the header (.h) file, and then modify your source code (.cpp) to look like this:

```
void ofApp::setup(){
    // Setup code omitted for clarity...

    isSavingPDF = false;
}

void ofApp::draw(){
    // If isSavingPDF is true (i.e. the s key has been pressed), then
    // anything in between ofBeginSaveScreenAsPDF(...) and
       ofEndSaveScreenAsPDF()
    // is saved to the file.
    if (isSavingPDF) {
        ofBeginSaveScreenAsPDF("savedScreenshot_
           "+ofGetTimestampString()+".pdf");
    }

    // Drawing code omitted for clarity...

    // Finish saving the PDF and reset the isSavingPDF flag to false
    // Ending the PDF tells openFrameworks to resume drawing to the
       screen.
    if (isSavingPDF) {
        ofEndSaveScreenAsPDF();
        isSavingPDF = false;
    }
}

void ofApp::keyPressed(int key){
    if (key == 's') {
        // isSavingPDF is a flag that lets us know whether or not
           save a PDF
        isSavingPDF = true;
    }
}
```

[Source code for this section[85]]

[ofSketch file for this section[86]]

[85] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/2_ii_c_Save_Vector_Graphics

[86] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/2_ii_c_Save_Vector_Graphics.sketch

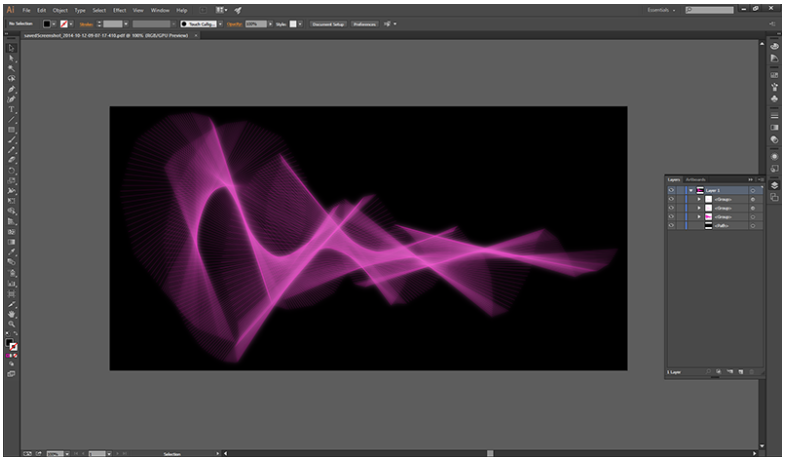


Figure 5.17: Figure 17: Editing a saved PDF from openFrameworks in Illustrator

5.3 Moving The World

We've been making brushes for a long time, so let's move onto something different: moving the world. By the world, I really just mean the coordinate system (though it sounds more exciting the other way).

Whenever we call a drawing function, like `ofRect(...)` for example, we pass in an `x` and `y` location at which we want our shape to be drawn. We know (0,0) to be the upper left pixel of our window, that the positive x direction is rightward across our window and that positive y direction is downward along our window (recall figure 1). We are about to violate this established knowledge.

Imagine that we have a piece of graphing paper in front of us. How would we draw a black rectangle at (5, 10) that is 5 units wide and 2 units high? We would probably grab a black pen, move our hands to (5, 10) on our graphing paper, and start filling in boxes? Pretty normal, but we could have also have kept our pen hand stationary, moved our paper 5 units left and 10 units down and then started filling in boxes. Seems odd, right? This is actually a powerful concept. With openFrameworks, we can move our coordinate system like this using `ofTranslate(...)`, but we can *also* rotate and scale with `ofRotate(...)` and `ofScale(...)`. We will start with translating to cover our screen with stick figures, and then we will rotate and scale to create spiraling rectangles.

5.3.1 Translating: Stick Family

`ofTranslate`[87] first. `ofTranslate(...)` takes an x, a y and an optional z parameter, and then shifts the coordinate system by those specified values. Why do this? Create a new project and add this to our `draw()` function of our source file (.cpp):

```
// Draw the stick figure family
ofCircle(30, 30, 30);
ofRect(5, 70, 50, 100);
ofCircle(95, 30, 30);
ofRect(70, 70, 50, 100);
ofCircle(45, 90, 15);
ofRect(30, 110, 30, 60);
ofCircle(80, 90, 15);
ofRect(65, 110, 30, 60);
```

Draw a white background and color the shapes, and we end up with something like figure 18 (left).

Figure 5.18: Figure 18: Arranging a little stick figure family

What if, after figuring out where to put our shapes, we needed to draw them at a different spot on the screen, or to draw a row of copies? We *could* change all the positions manually, or we could use `ofTranslate(...)` to move our coordinate system and leave the positions alone:

```
// Loop and draw a row
for (int cols=0; cols<10; cols++) {

    // Draw the stick figure family (code omitted)

    ofTranslate(150, 0);
}
```

So our original shapes are wrapped it in a loop with `ofTranslate(150, 0)`, which shifts our coordinate system to the left 150 pixels each time it executes. And we'll end up with figure 18 (second from left). Or something close to that, I randomized the colors in the figure - every family is different, right?

If we wanted to create a grid of families, we will run into problems. After the first row of families, our coordinate system will have been moved quite far to the left. If we

[87] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofTranslate

move our coordinate system up in order to start drawing our second row, we will end up drawing off the screen. It would look like figure 18 (third from left).

So what we need is to reset the coordinate system using `ofPushMatrix()`[88] and `ofPopMatrix()`[89]. `ofPushMatrix()` saves the current coordinate system and `ofPopMatrix()` returns us to the last saved coordinate system. These functions have the word matrix in them because openFrameworks stores all of our combined rotations, translations and scalings in a single matrix. So we can use these new functions like this:

```
    for (int rows=0; rows<10; rows++) {
        ofPushMatrix(); // Save the coordinate system before we
            shift it horizontally

            // It is often helpful to indent any code in-between
                push and pop matrix for readability

            // Loop and draw a row (code omitted)

        ofPopMatrix(); // Return to the coordinate system before we
            shifted it horizontally
        ofTranslate(0, 200);
    }
```

And we should end up with a grid. See figure 18, right. (I used `ofScale` to jam many in one image.) Or if you hate grids, we can make a mess of a crowd using random rotations and translations, figure 19.

Figure 5.19: Figure 19: A crowd

[Source code for this section[90]]

[ofSketch file for this section[91]]

[88] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofPushMatrix
[89] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofPopMatrix
[90] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/3_i_Translating_Stick_Family
[91] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/3_i_Translating_Stick_Family.sketch

5.3.2 Rotating and Scaling: Spiraling Rectangles

Onto `ofScale(...)` and `ofRotate(...)`! Let's create a new project where rotating and scaling rectangles to get something like figure 20.

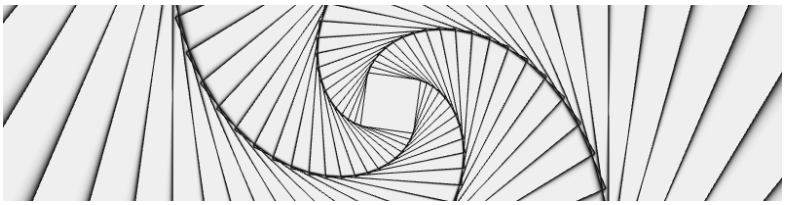


Figure 5.20: Figure 20: Drawing a series of spiraling rectangles

Before knowing about `ofRotate(...)`, we couldn't have drawn a rotated rectangle with `ofRect(...)`. `ofRotate(...)`[92] takes an angle (in degrees) and rotates our coordinate system around the current origin. Let's attempt a rotated rectangle:

```
ofBackground(255);
ofPushMatrix();
    // Original rectangle in blue
    ofSetColor(0, 0, 255);
    ofRect(500, 200, 200, 200);

    // Rotated rectangle in red
    ofRotate(45);
    ofSetColor(255, 0, 0);
    ofRect(500, 200, 200, 200);
ofPopMatrix();
```

Hmm, not quite right (figure 21, left). `ofRotate(...)` rotates around the current origin, the top left corner of the screen. To rotate in place, we need `ofTranslate(...)` to move the origin to our rectangle *before* we rotate. Add `ofTranslate(500, 200)` before rotating (figure 21, second from left). Now we are rotating around the upper left corner of the rectangle. The easiest way to rotate the rectangle around its center is to use `ofSetRectMode(OF_RECTMODE_CENTER)` draw the center at (500, 200). Do that, and we finally get figure 21, third from left.

Push, translate, rotate, pop - no problem. Only thing left is `ofScale(...)`[93]. It takes two arguments: the desired scaling in x and y directions (and an optional z scaling). Applying scaling to our rectangles:

```
ofSetRectMode(OF_RECTMODE_CENTER);
ofBackground(255);
```

[92] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofRotate
[93] http://www.openframeworks.cc/documentation/graphics/ofGraphics.html#show_ofScale

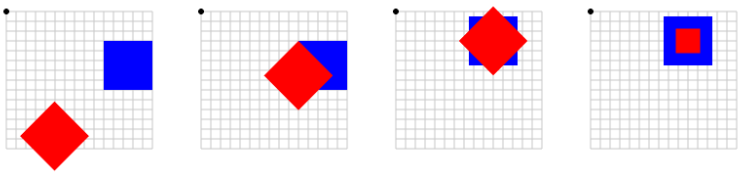


Figure 5.21: Figure 21: Steps along the way to rotating and scaling a rectangle in place

```
ofPushMatrix();
    // Original rectangle in blue
    ofSetColor(0, 0, 255);
    ofRect(500, 200, 200, 200);

    // Scaled down rectangle in red
    ofTranslate(500, 200);
    ofScale(0.5, 0.5);  // We are only working in x and y, so let's
        leave the z scale at its default (1.0)
    ofSetColor(255, 0, 0);
    ofRect(0, 0, 200, 200);
ofPopMatrix();
```

We'll run into the same issues that we ran into with rotation and centering. The solution is the same - translating before scaling and using `OF_RECTMODE_CENTER`. Example scaling shown in figure 21 (right).

Now we can make trippy rectangles. Start a new project. The idea is really simple, we are going to draw a rectangle at the center of the screen, scale, rotate, draw a rectangle, repeat and repeat. Add the following to our `draw()` function:

```
ofBackground(255);

ofSetRectMode(OF_RECTMODE_CENTER);
ofSetColor(0);
ofNoFill();
ofPushMatrix();
    ofTranslate(ofGetWidth()/2, ofGetHeight()/2);  // Translate to
        the center of the screen
    for (int i=0; i<100; i++) {
        ofScale(1.1, 1.1);
        ofRotate(5);
        ofRect(0, 0, 50, 50);
    }
ofPopMatrix();
```

That's it (figure 20). We can play with the scaling, rotation, size of the rectangle, etc. Three lines of code will add some life to our rectangles and cause them to coil and uncoil over time. Put these in the place of `ofRotate(5)`:

```
// Noise is a topic that deserves a section in a book unto itself
// Check out Section 1.6 of "The Nature of Code" for a good
   explanation
// http://natureofcode.com/book/introduction/
float time = ofGetElapsedTimef();
float timeScale = 0.5;
float noise = ofSignedNoise(time * timeScale) * 20.0;
ofRotate(noise);
```

Next, we can create a visual smear ("trail effect") as it rotates if we will turn off the background automatic clearing and partially erase the screen before drawing again. To do this add a few things to `setup()`:

```
ofSetBackgroundAuto(false);
ofEnableAlphaBlending(); // Remember if we are using transparency,
   we need to let openFrameworks know
ofBackground(255);
```

Delete `ofBackground(255)` from our `draw()` function. Then, add this to the beginning of our `draw()` function:

```
float clearAlpha = 100;
ofSetColor(255, clearAlpha);
ofSetRectMode(OF_RECTMODE_CORNER);
ofFill();
ofRect(0, 0, ofGetWidth(), ofGetHeight());  // ofBackground doesn't
   work with alpha, so draw a transparent rect
```

Pretty hypnotizing? If we turn up the `clearAlpha`, we will turn down the smear. If we turn down the `clearAlpha`, we will turn up the smear.

Now we've got two parameters that drastically change the visual experience of our spirals, specifically: `timeScale` of noise and `clearAlpha` of the trail effect. Instead of manually tweaking their values in the code, we can use the mouse position to independently control the values during run time. Horizontal position can adjust the `clearAlpha` while vertical position can adjust the `timeScale`. This type of exploration of parameter settings is super important (especially when making generative graphics), and using the mouse is handy if we've got one or two parameters to explore.

`mouseMoved(int x, int y )`[94] runs anytime the mouse moves (in our app). We can use it to change our parameters, but we need them to be global first. Delete the code that defines `timeScale` and `clearAlpha` locally in `draw()` and add them to the header. Initialize the values in `setup()` to `100` and `0.5` respectively. Then add these to `mouseMoved(...)`:

[94] http://openframeworks.cc/documentation/application/ofBaseApp.html#!show_mouseMoved

```
clearAlpha = ofMap(x, 0, ofGetWidth(), 0, 255);  // clearAlpha goes
    from 0 to 255 as the mouse moves from left to right
timeScale = ofMap(y, 0, ofGetHeight(), 0, 1);  // timeScale goes
    from 0 to 1 as the mouse moves from top to bottom
```

One last extension. We can slowly flip the background and rectangle colors, by adding this to the top of `draw()`:

```
ofColor darkColor(0,0,0,255);  // Opaque black
ofColor lightColor(255,255,255,255);  // Opaque white
float time = ofGetElapsedTimef();  // Time in seconds
float percent = ofMap(cos(time/2.0), -1, 1, 0, 1);  // Create a
    value that oscillates between 0 to 1
ofColor bgColor = darkColor;  // Color for the transparent rectangle
    we use to clear the screen
bgColor.lerp(lightColor, percent);  // This modifies our color "in
    place", check out the documentation page
bgColor.a = clearAlpha;  // Our initial colors were opaque, but our
    rectangle needs to be transparent
ofColor fgColor = lightColor;  // Color for the rectangle outlines
fgColor.lerp(darkColor, percent);  // Modifies color in place
```

Now use `bgColor` for the transparent rectangle we draw on the screen and `fgColor` for the rectangle outlines to get figure 22.

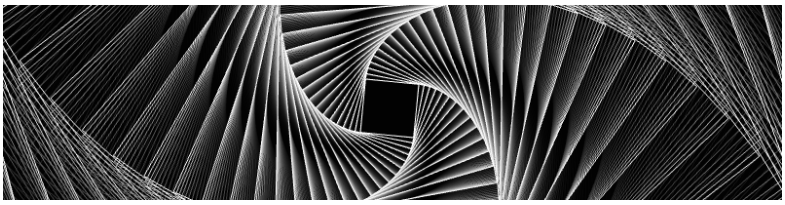


Figure 5.22: Figure 22: A single frame from animated spiraling rectangles where the contrast reverses over time

[Source code for this section[95]]

[ofSketch file for this section[96]]

Extensions

1. Pass in a third parameter, `z`, into `ofTranslate(...)` and `ofScale(...)` or rotate around the x and y axes with of `ofRotate(...)`.

[95] https://github.com/openframeworks/ofBook/tree/master/chapters/intro_to_graphics/code/3_ii_Rotating_and_Scaling

[96] https://github.com/openframeworks/ofBook/blob/master/chapters/intro_to_graphics/code/3_ii_Rotating_and_Scaling.sketch

2. Capture animated works using an addon called ofxVideoRecorder[97]. If you are using Windows, like me, that won't work for you, so try screen capture software (like fraps) or saving out a series of images using `ofSaveScreen(...)` and using them to create a GIF or movie with your preferred tools (photoshop, ffmpeg etc.)

5.4 Next Steps

Congratulations on surviving the chapter :). You covered a lot of ground and (hopefully) made some fun things along the way - which you should share on the forums[98]!

If you are looking to learn more about graphics in openFrameworks, definitely continue on to *Advanced Graphics* chapter to dive into more advanced graphical features. You can also check out these three tutorials: Basics of Generating Meshes from an Image[99], for a gentle introduction to meshes; Basics of OpenGL[100], for a comprehensive look at graphics that helps explain what is happening under the hood of openFrameworks; and Introducing Shaders[101], for a great way to start programming with your graphical processing unit (GPU).

[97] https://github.com/timscaffidi/ofxVideoRecorder
[98] http://forum.openframeworks.cc/
[99] http://openframeworks.cc/tutorials/graphics/generativemesh.html
[100] http://openframeworks.cc/tutorials/graphics/opengl.html
[101] http://openframeworks.cc/tutorials/graphics/shaders.html

6 Ooops! = Object Oriented Programming + Classes

6.1 Overview

This tutorial is an quick and practical introduction to Object Oriented Programming in openFrameworks and a how-to guide to build and use your own classes. By the end of this chapter you should understand how to create your own objects and have a lot of balls bouncing on your screen!

6.2 What is OOP

Object Oriented Programming is a programming paradigm based on the use of objects and their interactions. Some terms and definitions used within OOP are listed below:

-A Class defines the characteristics of a thing - the object - and its behaviors; it defines not only its properties and attributes but also what it can do.

-An Object is an instance of a class.

-The Methods are the objects abilities and how we can call them.

A recurring analogy is to see a Class as a the cookie cutter and the cookies as the Objects.

Note: please see chapter (Josh Nimoy's) for amore detailed explanation of Objected Oriented languages.

6.3 How to build your own Classes (simple Class)

Classes and objects are similar to the concepts of movie clips and instances in Flash and are also a fundamental part of Java programming. Because like coding, cooking is fun and we tend to experiment in the kitchen let's continue with the classic metaphor of a cookie cutter as a class and cookies as the objects. Every class has two files: a header file, also known as a Declarations file with the termination '.h' and an implementation file, terminating in '.cpp'. A very easy way of knowing what these two files do is to think

of the header file (.h) as a recipe, a list of the main ingredients of your cookie. The implementation file (.cpp) is what we're going to do with them, how you mix and work them to be the perfect cookie! So let's see how it works:

first of all let's create the two class files: If you're using XCODE as your IDE (it stands for: Integrated Development Environment), select the src folder and left Click (or CTRL + click), on the pop menu select 'New File' and you'll be taken to a new window menu, choose the appropriate platform you're developping for (OSX or iOS) and select C++ class and finally choose a name (we used 'ofBall'). You'll automatically see the two files in your 'src' folder: ofBall.h and ofBall.cpp . Now let's edit your class header (.h) file. Feel free to delete all its contents and let's start from scratch: Declare a class in the header file (.h). In this case, the file name should be ofBall.h. Folllow the code below and type into your own ofBall.h file, please note the comments I've included to guide you along.

```
#ifndef _OF_BALL // if this class hasn't been defined, the program
    can define it
#define _OF_BALL // by using this if statement you prevent the class
    to be called more than once which would confuse the compiler
#include "ofMain.h" // we need to include this to have a reference
    to the OpenFrameworks framework
class ofBall {

    public: // place public functions or variables declarations here

    // methods, equivalent to specific functions of your class
        objects
    void update();  // update method, used to refresh your objects
        properties
    void draw();    // draw method, this where you'll do the
        object's drawing

    // variables
    float x;      // position
    float y;
    float speedY; // speed and direction
    float speedX;
    int dim;      // size
    ofColor color; // color using ofColor type

    ofBall(); // constructor - used to initialize an object, if no
        properties are passed the program sets them to the default
        value
    private: // place private functions or variables declarations
        here
}; // don't forget the semicolon!!
#endif
```

We have declared the Ball class header file (the list of ingredients) and now lets get to the cooking part **[KL: I wouldn't use an arrow symbol within this text.] [RX: KL: do you mean excluding the point exmples to keep it simple?]** to see what these ingredients can do! Please notice the '#include' tag, this is a way to tell the compiler which file to include for each implementation file. When the program is compiled these '#include' tags will be replaced by the original file they're referring to. The 'if statement' (#ifndef) is a way to prevent the repetition of header files which could easily occur, by using this expression it helps the compiler to only include the file once and avoid repetition. Don't worry about this now, we'll talk about it later on!

Here's how you can write the class *.cpp file, the implementation file:

[KL: did the previous chapter talk about how to create a new class in oF? If so, no worries, but if it didn't, it's a good idea to address how to do that, and that the example below is ofBall.cpp. - see below] [JTN: yes it did but only in the unabridged version - see below] [RX: added xcode instructions below, anyone up to help with other IDEs?]

```
#include "ofBall.h"

ofBall::ofBall(){
    x = ofRandom(0, ofGetWidth());      // give some random
       positioning
    y = ofRandom(0, ofGetHeight());

    speedX = ofRandom(-1, 1);           // and random speed and
       direction
    speedY = ofRandom(-1, 1);

    dim = 20;

    color.set(ofRandom(255),ofRandom(255),ofRandom(255)); // one way
       of defining digital color is by adddressing its 3 components
       individually (Red, Green, Blue) in a value from 0-255, in
       this example we're setting each to a random value
}


void ofBall::update(){
    if(x < 0 ){
        x = 0;
        speedX *= -1;
    } else if(x > ofGetWidth()){
        x = ofGetWidth();
        speedX *= -1;
    }

    if(y < 0 ){
        y = 0;
```

```
        speedY *= -1;
    } else if(y > ofGetHeight()){
        y = ofGetHeight();
        speedY *= -1;
    }

    x+=speedX;
    y+=speedY;
}

void ofBall::draw(){
    ofSetColor(color);
    ofCircle(x, y, dim);
}
```

Now, this is such a simple program that we could have wirtten it inside our OfApp(.h and .cpp) files and it wouldn;t be senseless to do if we didn;t want to re-use this code. In there lyes one of the advantages of Object Oriented Programming: re-use: Imagine we want to create thousands of these balls and how easily the ocde could get messy and extended, by creating our own class we can later re-create as many objects as need from it and just call the appropriate methods when needed keeping our code clean and eficient. In a more pragamatic example think of creating a class for each of your UI elements (button, slider, etc) and hopw easy it would be to them deploy them in your program but also to include and re-use them in future programs.

[KL: Explain exactly why we are creating this class outside of ofApp. This and the explanation below seem kind of rushed and OOP can benefit by using some real life analogies to demonstrate class relationships.]

6.4 make an Object from your Class

Now that we've created a class let's make the real object! In your testApp.h (header file) we'll have to declare a new object and get some free memory for it. But first we need to include (or give the instructions to do so) your ofBall class in our program. To do this we need to write:

```
#include "ofBall.h"
```

on the top of your testApp.h file. Then we can finally declare an instance of the class i our progam:

```
ofBall myBall;
```

Now let's get that ball bouncing on screen! Go to your project testApp.cpp (implementation) file. Now that we've created the object, we just need to draw it and update its values by calling its methods. In the update() function, add:

```
myBall.update();  // calling the object's update method
```

and in the draw() function lets add:

```
myBall.draw();  // call the draw method to draw the object
```

Compile and run! By now you must be seeing a bouncing ball on the screen! Great!

6.5 make objects from your Class

By now, you're probably asking yourself why you went to so much trouble to create a bouncing ball. You could have done this (and probably have) without using classes. In fact one of the advantages of using classes is to be able to create multiple individual objects with the same characteristics. So, let's do that now! Go back to your ofApp.h file and create a couple of new objects:

```
ofBall myBall1;
ofBall myBall2;
ofBall myBall3;
```

In the implementation file (ofApp.cpp), call the corresponding methods for each of the objects

in the ofApp's update() function:

```
myBall1.update();
myBall2.update();
myBall3.update();
```

and also in the draw() function:

```
myBall1.draw();
myBall2.draw();
myBall3.draw();
```

6.6 make more Objects from your Class

We've just created 3 objects but what if we wanted to created 10, 100 or maybe 1000's of them?! Hardcoding one by one would be a painful and long process that can be easily solved by automating the object creation and function calls. Just by using a couple for loops we'll make this process simpler and cleaner. Instead of declaring a list of objects one by one we'll create an array of objects of type 'ofBall'. We'll also

introduce another new element: a constant. Constants are defined after the #includes as #define CONSTANT_NAME value. This is a way of defining a constant value that won't ever change in the program:

[KL: The pseudo code-like explanation above is an effective approach. This is a good method to use before writing out the ofBall class above, too. Also, I've been taking out words like "just" before steps and simplifying verb tenses for clarity. I'd keep that in mind as you continue writing this chapter. The more concise, the better.] [KL: Restate which file this is happening in.] in the testApp class header file, where you define the balls objects also define the constant that we'll use for the number of objects:

```
#define NBALLS 5
```

we'll now use the CONSTANT value to define the size of our array of objects:

```
ofBall myBall[NBALLS];
```

back to our implementation file we'll just need to create an array of objects and call their methods through 'for' loops. in the update() function:

```
for(int i=0; i<NBALLS; i++){
    myBall[i].update();
}
```

in the draw() function:

```
for(int i=0; i<NBALLS; i++){
    myBall[i].draw();
}
```

6.7 make even more Objects from your Class: properties and constructors

As we've seen, each of the objects has a set of properties defined by its variables (position, speed, direction, and dimension). Another advantage of object oriented programming is that the objects created can have different values for each of its properties. For us to have better control of each object, we can have a constructor that defines these characteristics and lets us access them. In the ofBall definitions file (*.h) we can change the constructor to include some of the object's properties (let's say position and dimension):

```
ofBall(float x, float y, int dim);
```

Since we've changed the constructor, we'll need to update the ofBall implementation (*.cpp) file to reflect these.

```
ofBall::ofBall(float _x, float _y, int _dim){
    x = _x;
    y = _y;
    dim = _dim;

    speedX = ofRandom(-1, 1);
    speedY = ofRandom(-1, 1);
}
```

Your ofBall.cpp file should look like this by now:

```
#include "ofBall.h"

ofBall::ofBall(float _x, float _y, int _dim){
        x = _x;
        y = _y;
        dim = _dim;

        speedX = ofRandom(-1, 1);
        speedY = ofRandom(-1, 1);

        color.set(ofRandom(255), ofRandom(255), ofRandom(255));

}


void ofBall::update(){

    if(x < 0 ){
        x = 0;
        speedX *= -1;
    } else if(x > ofGetWidth()){
        x = ofGetWidth();
    speedX *= -1;
    }

    if(y < 0 ){
        y = 0;
        speedY *= -1;
    } else if(y > ofGetHeight()){
        y = ofGetHeight();
        speedY *= -1;
    }

    x+=speedX;
    y+=speedY;

}
```

```
void ofBall::draw(){
    ofSetColor(color);
    ofCircle(x, y, dim);
}
```

By implementing these changes we'll also need to create space in memory for these objects. We'll do this by creating a pointer (a reference in memory) for each object. Back to the ofApp.h (definitions) file we'll declare a new object like this:

```
ofBall *myBall;
```

The star(*) means it will be created in a reserved part of memory just for it, we'll dynamically allocate this instance of the ofBall class.

[KL: specify why we'd make it into a pointer vs not a pointer]

Now in the TestApp.cpp file we will need to create the object in the setup and we'll call the object's methods on the draw() and update() functions in a different way than before. Instead of using the (.) dot syntax like we have been doing so far, from now on we'll use the (->) arrow syntax. Also, we'll also be creating a new instance way more explicitily. So, in setup()

```
            // x-position,        y-position,           size
myBall = new ofBall(ofRandom(300,400), ofRandom(200,300),
   ofRandom(10,40));
```

As you see it is now possible to directly control the objects properties on its creation. and now we'll just need to update and draw it.

```
myBall->update();


myBall->draw();
```

[KL: We've changed myBall.update() to myBall->update(). That's kind of a big deal and warrants explanation concerning pointers.]

[JTN: no harm in explaining it twice, but i introduced it at the end of my unabridged chapter https://github.com/openframeworks/ofBook/blob/master/02_cplusplus_basics/unabridg]

6.8 make even more Objects from your Class

In this part of our OOPs! tutorial **[KL: I simplified two statements into one for concision.]** we'll demonstrate an automation process to create objects from our previously built class. We'll create more **[KL: "We'll create" works better than "we'll be creating."**

This is an example of verb tenses I've been changing. It's shorter and clearer.] objects by using arrays like we did in part 2.1 but this time we'll have to do some minor changes:

```
ofBall** myBall;   // an array of pointers of type ofBall
int nBalls;        //variable for the number of balls
```

When creating an array of objects, instead of creating one pointer, we'll create an array of pointers. That's why we have two 'stars' and not one in the declarations(*.h) file. We have created a pointer to an array of pointers. Let's see how we'll create and call these objects in the implementation* (.cpp) file:

```
nBalls = 5; // the number of ball objects we want to create

myBall = new ofBall*[nBalls]; // an array of pointers for the objects

for (int i = 0; i < nBalls; i++){
    float x = 20+(100*i); // using the value of the counter
       variable(i) to differentiate them
    float y = 20+(100*i);
    int dim = 10+(i*10);

    myBall[i] = new ofBall(x,y,dim);  // create each object from the
       array
}
```

similarly when we want to draw and update the objects we've created we'll need 'for' loops to run through the array.

```
for (int i = 0; i < nBalls; i++){
    myBall[i]->update();
}



for (int i = 0; i < nBalls; i++){
    myBall[i]->draw();
}
```

[KL: Great tutorial so far. The organization works well. I'm eager to see the rest. The main thing so far would be focusing on concision in your writing.]

6.9 Make and delete as you wish - using vectors

In this part we'll look into more dynamic ways of creating and destroying objects from our class. Vectors are special arrays that don;t need a pre-fixed number of elements,

that's their magic: vectors are elastic! note: You'll be hearing baout two different types of vectors throughout this book. Please don't confuse stl::vectors (the elastic arrays type we're talking about) with the math vectors (forces).

Back to our beloved testApp.h file, let's define a vector of ofBall objects by typing:

```
vector <ofBall*> myBall;
```

In this expression we're creating a type (vector) of type (ofBall pointers) and naming it myBall. Now, let's head to our (.cpp) and start cooking! Ignore the setup, update and draw methods for now, let's jump to

```
void testApp::mouseDragged(int x, int y, int button){

}
```

In this method we're listening to the dragging activity of your mouse or trackpad and we'll use this simplicity to create interaction! So let's just create some code to create ofBalls and add them to our porgram when we drag the mouse.

```
void testApp::mouseDragged(int x, int y, int button){
    ofBall *tempBall;
        tempBall = new ofBall(x,y, ofRandom(10,40));
        myBall.push_back(tempBall);
}
```

A few new things in our code, first we declare a temporary object pointer, we them create it and assign 'x' and 'y' mouse Coordinates to it's contructor variables. We later use this temporary object as a shortcut to add ofBall objects to our vector. Back to our Update and Draw methids we can add the needed 'for loops' to iterate over the objects in the vector to update and draw them liek we would do with arrays. This time though we didn't declare a variable that stores the maximum number of objects but instead we call a method that vectors have that allwos us to know their size. See code below for Update:

```
for (int i = 0 ; i<myBall.size(); i++) {
        myBall[i]->update();
    }
```

and for Draw:

```
for (int i = 0 ; i<myBall.size(); i++) {
    myBall[i]->draw();
}
```

Now let's also implement a way to delete them before we have way too many ofBalls: On the testApp::MousePressed Call we will loop though our vector and check the distance between the coordinates of the mouse with the ofBall position, if this distance

is smaller than the ofBall dimension then, we know that we're clicking inside it, we can delete it. Because we're using the vecotr.erase method we need to use an iterator (myBall.begin()), a shortcut that references to the first element of the vector as a starting point to access the vector element we really want to erase ('i').

```
for (int i =0; i < myBall.size(); i++) {
    float distance = ofDist(x,y, myBall[i]->x, myBall[i]->y); // a
       method OF give us to check the distance between two
       coordinates

    if (distance < myBall[i]->dim) {
        myBall.erase(myBall.begin()+i); // we need to use an
           iterator/ reference to the vector position we want to
           delete
    }
}
```

To learn more about stl::vector check xxx chapter or this online shory tutorial : http://www.openframeworks.cc/tutorials/c++%20concepts/001_stl_vectors_basic.html

6.10 Quick intro to polymorphism (inheritance)

You're now discovering the power of OOP, making a class and creating as many objects from that in an instant, adding and deleting by your application needs. Now, for a second let's go back to our cooking metaphor (yummi!) and imagine that your cookies, even sharing the same cookie cutter and dough using some different sprinkles on each won't hurt and add some desired variation to our cookie jar selection! This is also the power of OOP and inheritance: by allowing to use a base class and add some specific behaviours overwrite some of the behaviours of a class, creating a subset of instances / objects with slightly different behaviors. The great thing about this is it's repurposability, we're using the 'mother' class as a starting point, using all its capabilities but we overwrite one of its methods to give it more flexibility. Going back to our oFBall class intial version (step 1) we'll build some 'daughter' classes based on its main characteristics (motion behaviors and shape) but we'll distinct each inherited subClass by using a different color on its drawing method. Your ofBall header file should look like this:

```
#ifndef _OF_BALL // if this class hasn't been defined, the program
    can define it
#define _OF_BALL // by using this if statement you prevent the class
    to be called more than once which would         confuse the
    compiler
#include "ofMain.h"
```

```
class ofBall {

public: // place public functions or variables declarations here

void update();
void draw();

// variables
float x;
float y;
float speedY;
float speedX;
int dim;

ofColor color;

ofBall();

private:

};
#endif
```

And let's make some slight changes on the implementation file: lets; chage the min and maximum values of the random size to larger values and set the position to the center of the screen.Make it look like this:

```
#include "ofBall.h"

ofBall::ofBall(){
    x = ofGetWidth()*.5;
    y = ofGetHeight()*.5;
    dim = ofRandom(200,250);

    speedX = ofRandom(-1, 1);
    speedY = ofRandom(-1, 1);

    color.set(ofRandom(255), ofRandom(255), ofRandom(255));
}
```

We can leave the update() and draw() functions as they were. Now,let's start making 'daughter' versions of this 'mother' class. Create a new Class set of files and name them 'ofBallBlue'. Feel free to copy the code below and it's '.h' shoudl look like this:

```
#pragma once                // another and more modern way to
   prevent the compiler from including this file more than once

#include "ofMain.h"
```

```
#include "ofBall.h"            // we need to include the 'mother'
    class, the compiler will include the mother/base class so we have
    access to all the methods inherited

class ofBallBlue : public ofBall {     // we set the class to
    inherit from 'ofBall'

public:

    void draw();              // this is the only methid we actually
        want to be different from the 'mother class'

};
```

On the '.cpp' file we'll need to them specify what we want the new 'draw()' method to do uniquely.

```
#include "ofBallBlue.h"


void ofBallBlue::draw(){
    ofSetColor(ofColor::blue);    // this is a shortcut for full
        blue color ;)
    ofCircle(x, y, dim);
}
```

Now, on your own, create two new classes: ofBallRed and ofBallGreen based on ofBall class like ofBlue is. Back to your testApp.h, include the newly made classes and create one instance of each and in your testApp.cpp file initialize them and call their update() and draw() methods. A quick trick! right before you call the draw method, make this call:

```
ofEnableBlendMode(OF_BLENDMODE_ADD);
```

This will make your application drawing methods have an Additive Blending Mode. For more on this check Chapter??.

Hope you enjoyed this short tutorial! have fun!

7 Animation

by Zach Lieberman[1]

with edits from Kayla Lewis

7.1 Background

The word animation is a medieval term stemming from the Latin animare, which means 'instill with life'. In modern terms, it's used to describe the process of creating movement from still, sequential images. Early creators of animation used spinning discs (phenakistoscopes) and cylinders (zoetropes) with successive frames to create the illusion of a smooth movement from persistence of vision. In modern times, we're quite used to other techniques such as flip books and cinematic techniques like stop motion. Increasingly, artists have been using computational techniques to create animation – using code to "bring life" to objects on the screen over successive frames. This chapter is going to look at these techniques and specifically try to address a central question: how can we create compelling, organic, and even absurd movement through code?

As a side note, I studied fine arts, painting and printmaking, and it was accidental that I started using computers. The moment that I saw how you could write code to move something across the screen, even as simple as silly rectangle, I was hooked. I began during the first dot-com era working with flash / actionscript and lingo / director and have never looked back.

This chapter will first explain some basic principles that are useful to understanding animation in OF, then attempt to show a few entrypoints to interesting approaches.

7.2 Animation in OF / useful concepts:

7.2.1 Draw cycle

The first point to make about animation is that it's based on successive still frames. In openFrameworks we have a certain loop cycle that's based roughly on game programming paradigms. It goes like this:

[1] http://thesystemis.com

- setup()
- update()
- draw()
- update()
- draw()
- ....

Setup gets called once, right at the start of an OF apps lifecycle and update / draw get called repeatedly. Sometimes people ask why two functions get called repeatedly, especially if they are used to Processing, which has only a setup and a draw command. There are a few reasons. The first is that drawing in openGL is asynchronous, meaning there's a chance, when you send drawing code to the computer, that it can return execution back to your program so that it can perform other operations while it draws. The second is that it's generally very practical to have your drawing code separated from your non-drawing code. If you need to quickly debug something–say, for example, your code is running slow–you can comment out the draw function and just leave the update running. It's separating out the update of the world from the presentation and it can often help clean up and organize your code. Think about it like a stop frame animator working with an overhead camera that might reposition objects while the camera is not taking a picture then snap a photograph the moment things are ready. In the update function you would be moving things around and in the draw function you draw things exactly as they are at that moment.

7.2.2 Variables

The second point to make about animation is that it requires variables. A variable is a placeholder for a value, which means that you can put the value in and you can also get the value out. Variables are essential for animation since they "hold" value from frame to frame–e.g., if you put a value in to a variable in the setup function or update function, you can also get it out from memory in the draw function. Take this example:

[note: simple animation example here]

7.2.3 Frame rate

The third point to make about OF and animation is frame rate. We animate in openframeworks using successive frames. Frame rate refers to how quickly frames get drawn. In OF there are several important functions to know about.

- `ofGetFrameRate()` returns the current frame rate (in frames per second). Set it 0 to run as fast as possible

- `ofSetFrameRate( float targetFrameRate )` sets the maximum frame rate. If the software is animating faster than this, it will slow it down. Think of it like a speed limit. It doesn't make you go faster, but it prevents you from going too fast.

In addition, openGL works with an output display and will attempt to synchronize with the refresh rate of the monitor – sometimes called vertical-sync or vertical blanking. If you don't synchronize with the refresh rate, you can get something called frame tearing, where the non-synchronization can mean frames get drawn before and after a change, leading to horizontal lines of discontinuity, called screen tearing[2]

[note: frame rip graphic here]

We have a function in OF for controlling this. Some graphics card drivers (see for example Nvidia's PC drivers) have settings that override application settings, so please be sure to check your driver options.

- `ofSetVerticalSync (bool bUseSync)` set this true if you want to synchronized vertically, false if you want to draw as fast as possible.

By default, OF enables vertical sync and sets a frame rate of 60FPS. You can adjust the VSYCN and frame rate settings if you want to animate faster, but please note that by default OF wants to run as fast as possible. It's not uncommon if you are drawing a simple scene to see frame rates of 800 FPS if you don't have VSYNC enabled (and the frame rate cap set really high or disabled).

Another imporant point which is a bit hard to cover deeply in this chapter is frame rate independence. If you animate using a simple model – say for example, you create a variable called xPos, increase it by a certain amount every frame and draw it.

```
void testApp::setup(){
    xPos = 100;
}

void testApp::update(){
    xPos += 0.5;
}

void testApp::draw(){
    ofRect(xPos, 100, 10, 10);
}
```

this kind of animation works fine, but it assumes that your frame rate is constant. If you app runs faster, say by jumping from 30fps to 60fps, the object will appear to go twice as fast, since there will be 2x the number of update and draw functions called per second. Typically more complex animation will be written to take this into account, either by using functions like time (explained below) or mixing the frame rate or elapsed time into your update. For example, a solution might be something like:

[2] http://en.wikipedia.org/wiki/Screen_tearing

```
void testApp::update(){
    xPos += 0.5 * (30.0 / ofGetFrameRate());
}
```

if ofGetFrameRate() returns 30, we multiply 0.5 by 1, if ofGetFrameRate() returns 60, we multiply it by 1/2, so although we are animating twice as fast, we take half sized steps, therefore effectively moving at the same speed regardless of framerate. Framerate indepence is fairly important to think about once you get the hang of things, since as observers of animation, we really do feel objects speeding up or slowwing down even slightly, but in this chapter I will skip it for the sake of simplicity in the code.

7.2.4 Time functions

Finally, there are a few other functions that are useful for animation timing:

- `ofGetElapsedTimef()` returns the elapsed time in floating point numbers, starting from 0 when the app starts.
- `ofGetElapsedTimeMillis()` similarly returns the elapsed time starting from 0 in milliseconds.
- `ofGetFrameNum()` returns the number of frames the software has drawn. If you wanted, for example, to do something every other frame you could use the mod operator, e.g., `if (ofGetFrameNum()% 2 == 0)`.

7.2.5 Objects

In these examples, I'll be using objects pretty heavily. It's helpful to feel comfortable with OOP to understand the code. One object that is used heavily is `ofPoint`, which contains an x,y and z variable. In the past this was called "ofVec3f" (vector of three floating point numbers), but we just use the more convenient ofPoint. In some animation code, you'll see vectors used, and you should know that ofPoint is essentially a vector.

You will also see objects that have basic functionality and internal variables. I will typically have a setup, update and draw inside them. A lot of times, these objects are either made because they are useful recipies to have many things on the screen or they help by putting all the variables and logic of movement in one place. I like to have as little code as possible at the testApp / ofApp level. If you are familiar with actionscript / flash, this would be similar to having as a little as possible in your main timeline.

7.3 linear movement

7.3.1 getting from point a to point b

One of the most important things to think about when it comes to animation is answering the simple question:

how do you get from point A to point B?

In this chapter we will look at animating movement (changing position over time) but we could very well be animating any other numeric property, such as color, the width or height of a drawn shape, radius of a circle, etc.

The first and probably most important lesson of animation is that we **love** numbers between 0 and 1.

[note: love picture here]

The thing about numbers between 0 and 1 is that they are super easy to use in interesting ways. We typically refer to these kinds of numbers as percent, and you'll see me use the shorthand `pct` in the code–this is a floating point number between 0 and 1. If we wanted to get from point A to point B, we could use this number to figure out how much of one point and how much of another point to use. The formula is this:

```
((1-pct) * A) + (pct * B)
```

To add some detail if we are 0 pct of the way from A to B, we calculate

```
((1-0) * A) + (0 * B)
```

which simplifies to `(1*A + 0*B)` or A. If we are 25 percent of the way, it looks like

```
((1-0.75) * A) + (0.25 * B)
```

which is 75% of A + 25% of B. Essentially by taking a mix, you get from one to the other. The first example shows how this is done.

[note: linear example code here]

As a side note, the function `ofMap`, which maps between an input range, uses pct internally. It takes a value, converts it into a percentage based on the input range, and then uses that pct to find the point between the output range. **[note: see omer's chapter]**

7.3.2 Curves

One of the interesting properties of numbers between 0 and 1 is that they can be easily adjusted / curved.

The easiest way to see this is by raising the number to a power. A power, as you might remember from math class, is multiplying a number by itself, e.g., 2^3 = `2*2*2 = 8`. Numbers between 0 and 1 have some interesting properties. If you raise 0 to any power it equals 0 (`0x0x0x0 = 0`). The same thing is true for 1 (`1*1*1*1 = 1`), but if you raise a number between 0 and 1 to a power, it changes. For example, 0.5 to the 2nd power = 0.25.

Let's look at a plot of pct raised to the second power:

[note: plot grpahic here]

[note: better explanation of how to read the chart] Think about the x value of the plot as the input and y value as the output. If put in 0, we get out a y value of 0, if we put in 0.1, we get out a y value of 0.01, all the way to putting in a value of 1 and getting out a value of 1.

As side note, it's important to note that things in the world often don't move linearly. They don't take "even" steps. Roll a ball on the floor, it slows down. It's accelerating in a negative direction. Sometimes things speed up, like a baseball bat going from resting to swinging. Curving pct leads to interesting behavior. The objects still take the same amount of time to get there, but they do it in more lifelike, non-linear ways.

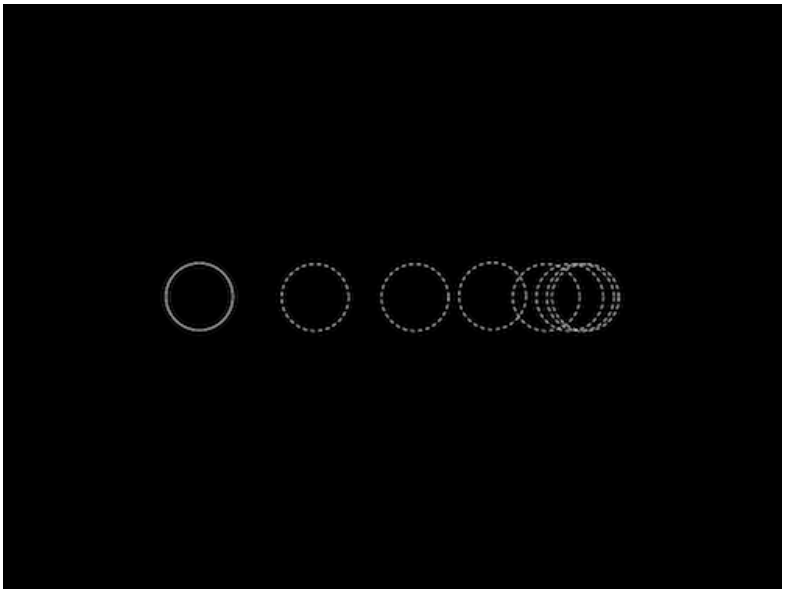


Figure 7.1: nonlinear

If you raise the incoming number between 0 and 1 to a larger power it looks more extreme. Interestingly, if you raise this value between 0 and 1 to a fractional (rational) power (i.e., a power that's less than 1 and greater than 0), it curves in the other direction.

The second example shows an animation that uses pct again to get from A to B, but in this case, pct is raised to a power:

In the 4th example (**4_rectangleInterpolatePowfMultiple**), you can see a variety of these rectangles, all moving with different shaping functions. They take the same amount of time to get from A to B, but do it in very different ways. I usually ask my students to guess which one is moving linearly – see if you can figure it out without looking at the code:

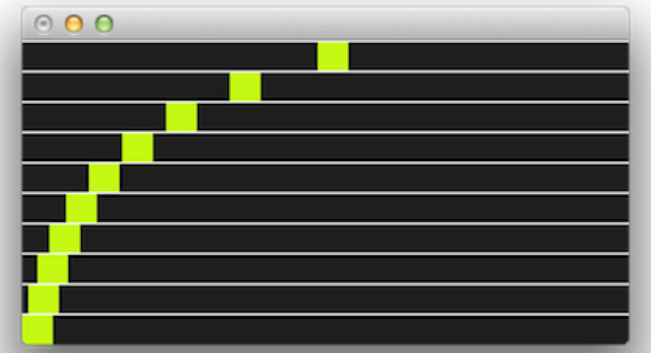


Figure 7.2: xeno diagram

http://en.wikipedia.org/wiki/12_basic_principles_of_animation#Slow_in_and_slow_out

[note: can we get rights for a screenshot of masahiko sato curves DVD ?]

Raising percent to a power is one of a whole host of functions that are called "shaping functions" or "easing equations." Robert Penner wrote about and derived many of these functions so they are also commonly reffered to as "Penner Easing Equations." Easings.net[3] is a good resource, as well there are several openFrameworks addons for easing.

- http://sol.gfxile.net/interpolation/#c1
- http://easings.net/

7.3.3 Zeno

A small twist on the linear interpolation is a technique that I call "Zeno" based on Zeno the greek philosopher's *dichotomy paradox*:

> Imagine there is a runner running a race and the runner runs 1/2 of the distance in a certain amount of time, and then they run 1/2 of the remaining

[3] http://easings.net/

distance in the same amount of time, and run 1/2 of the remaining the distance the distance, etc. Do they finish the race? There is always some portion of the distance remaining left to run one half of. The idea is that you can always keep splitting the distance.

If we take the linear interpolation code but always alter our own position instead (e.g., take 50% of our current position + 50% of our target position), we can animate our way from one value to another. I usually explain the algorithm in class by asking someone to do this:

1. Start at one position in your room.
2. Pick a point to move to.
3. Calculate the distance between your current position and that point.
4. Move 50% closer.
5. Go to (3).

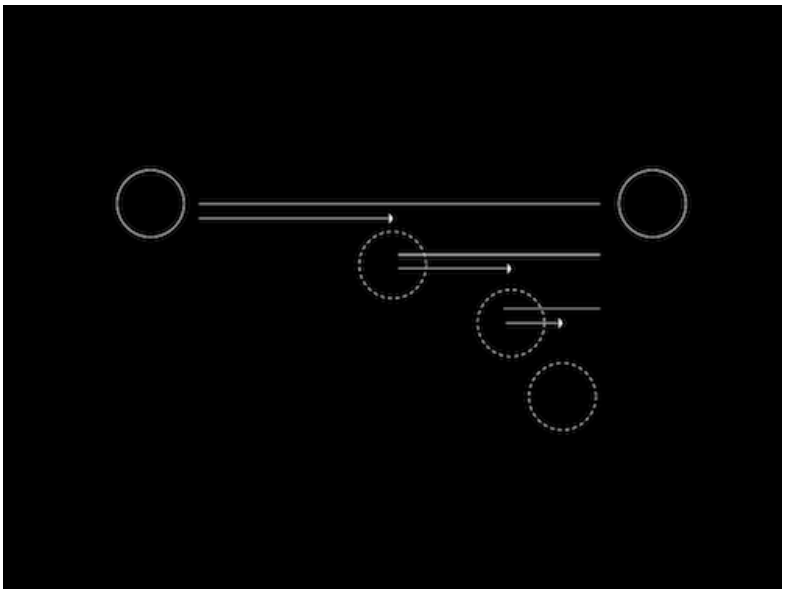


Figure 7.3: xeno diagram

In code, that's basically the same as saying

```
currentValue = currentValue + ( targetValue - currentValue ) * 0.5.
```

In this case `targetValue - currentValue` is the distance. You could also change the size of the step you make every time, for example, taking steps of 10% instead of 50%:

```
currentValue = currentValue + ( targetValue - currentValue ) * 0.1.
```

If you expand the expression, you can write the same thing this way:

```
currentValue = currentValue * 0.9 + targetValue * 0.1.
```

This is a form of smoothing: you take some percentage of your current value and another percentage of the target and add them together. Those percentages have to add up to 100%, so if you take 95% of the current position, you need to take 5% of the target (e.g., currentValue * 0.95 + target * 0.05).

In Zeno's paradox, you never actually get to the target, since there's always some remaining distance to go. On the computer, since we are dealing with pixel positions on the screen and floating point numbers at a specific range, the object appears to stop.

In the 5th example **(5_rectangleXeno)**, we add a function to the rectangle that uses xeno to catch up to a point:

```
void rectangle::xenoToPoint(float catchX, float catchY){
    pos.x = catchUpSpeed * catchX + (1-catchUpSpeed) * pos.x;
    pos.y = catchUpSpeed * catchY + (1-catchUpSpeed) * pos.y;
}
```

Here, we have a value, `catchUpSpeed`, that represents how fast we catch up to the object we are trying to get to. It's set to 0.01 (1%) in this example code, which means take 99% of my own postion, 1% of the target position and move to their sum. If you alter this number you'll see the rectangle catch up to the mouse faster or slower. 0.001 means it will run 10 times slower, 0.1 means ten times faster.

This technique is very useful if you are working with noisy data – a sensor for example. You can create a variable that catched up to it using xeno and smoothes out the result. I use this quite often when I'm working with hardware sensors / physical computing, or when I have noisy data. The nice thing is that the catch up speed becomes a knob that you can adjust between more real-time (and more noisey data) and less real-time (and more smooth) data. Having that kind of control comes in handy!

7.4 Function based movement

In this section of the book we'll look at a few examples that show function based movement, which means using a function that takes some input and returns an output that we'll use for animation. For input, we'll be passing in counters, elapsed time, position, and the output we'll use to control position.

7.4.1 Sine and Cosine

Another interesting and simple system to experiment with motion in openframeworks is using sin and cos.

Sin and cos (sine and cosine) are trigonometric functions, which means they are based on angles. They are the x and y position of a point moving in a constant rate around a circle. The circle is a unit circle with a radius of 1, which means the diameter is `2*r*PI` or `2*PI`. In OF you'll see this constant as `TWO_PI`, which is 6.28318...

As a side note, sometimes it can be confusing that some functions in OF take degress where others take radians. Sin and cos are part of the math library, so they take radians, whereas most openGL rotation takes degrees. We have some helper constants such as `DEG_TO_RAD` *and* `RAD_TO_DEG`*, which can help you convert one to the other.*

Here's a simple drawing that helps explain sin and cos.

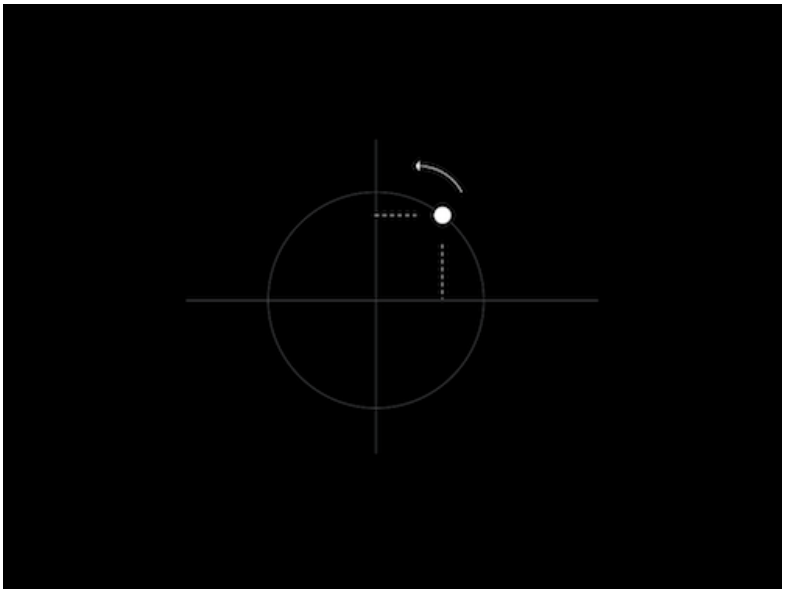


Figure 7.4: sin

All you have to is imagine a unit circle, which has a radius of 1 and a center position of 0,0. Now, imagine a point moving counter clockwise around that point as a constant speed. If you look at the height of that point, it goes from 0 at the far right (3 o'clock position), up to 1 at the top (12 o'clock), back at 0 at the left (9 o'clock) and down to -1 at the bottom (6 o'clock). So it's a smooth, curving line that moves between -1 and 1. That's it. Sin is the height of this dot and cos is the horizontal position of this dot. At the far right, where the height of this dot is 0, the horizontal position is 1. When sin is 1, cos is 0, etc. They are in sync, but shifted.

7.4.1.1 Simple examples

It's pretty easy to use sin to animate the position of an object.

Here, we'll take the sin of the elapsed time `sin(ofGetElpasedTimef())`. This returns a number between negative one and one. It does this every 6.28 seconds. We can use of map to map this to a new range. For example

```
void ofApp::draw(){
    float xPos = ofMap(sin(ofGetElpasedTimef()), -1, 1, 0,
       ofGetWidth());
    ofRect(xPos, ofGetHeight/2, 10,10);
}
```

This draws a rectangle which move sinusoidally across the screen, back and forth every 6.28 seconds.

You can do simple things with offseting the phase (how shifted over the sin wave is). In example 7 **(7_sinExample_phase)**, we calculate the sin of time twice, but the second time, we add PI: `ofGetElapsedTimef()+ PI`. This means to the two values will be offset from each other by 180 degrees on the circle (imaginging our dot, when one is far right, the other will be far left. When one is up, the other is down). Here we set the background color and the color of a rectangle using these offset values. It's useful if you start playing with sin and cos to start to manipulate phase.

```
//--------------------------------------------------------------
void testApp::draw(){


    float sinOfTime                = sin( ofGetElapsedTimef() );
    float sinOfTimeMapped        = ofMap( sinOfTime, -1, 1, 0, 255);


    ofBackground(sinOfTimeMapped, sinOfTimeMapped, sinOfTimeMapped);


    float sinOfTime2            = sin( ofGetElapsedTimef() + PI);
    float sinOfTimeMapped2        = ofMap( sinOfTime2, -1, 1, 0,
       255);

    ofSetColor(sinOfTimeMapped2, sinOfTimeMapped2, sinOfTimeMapped2);
    ofRect(100,100,ofGetWidth()-200, ofGetHeight()-200);

}
```

As a nerdy detail, floating point numbers are not linearly precise, e.g., there's a different number of floating point numbers between 0.0 and 1.0 than 100.0 and 101.0. You actually lose precision the larger a floating point number gets, so taking sin of elapsed time can start looking crunch after some time. For long running installations I will sometimes write code that looks like `sin((ofGetElapsedTimeMillis()% 6283)/ 6283.0)` *or something similar, to account for this. Even though ofGetElapsedTimef()*

gets larger over time, it's a worse and worse input to sin() as it grows. ofGetElapsedTimeMillis() doesn't suffer from this problem since it's an integer number and the number of integers between 0 and 10 is the same as between 1000 and 1010.

7.4.1.2 Circular movement

Since sin and cos are derived from the circle, if we want to move things in a circular way, we can figure this out via sin and cos. We have four variables we need to know:

- the origin of the circle (xOrig, yOrig)
- the radius of the circle (radius)
- the angle around the circle (angle)

The formula is fairly simple:

```
xPos = xOrig + radius * cos(angle);
yPos = yOrig + radius * sin(angle);
```

This allows us to create something moving in a circular way. In the circle example, I will animate using this approach.

```
float xorig = 500;
float yorig = 300;
float angle = ofGetElapsedTimef()*3.5;
float x = xorig + radius * cos(angle);
float y = yorig + radius * sin(angle);
```

Note: In OF, the top left corner is 0,0 (y axis is increasing as you go down) so you'll notive that the point travels clockwise instead of counter-clockwise. If this bugs you (since above, I asked you imagine it moving counter clockwise) you can modify this line `float y = yorig + radius * sin(angle)` *to* `float y = yorig + radius * -sin(angle)` *and see the circle go in the counter clockwise direction.*

For these examples, I start to add a "trail" to the object by using the ofPolyline object. I keep adding points, and once I have a certain number I delete the oldest one. This helps us better see the motion of the object.

If we increase the radius, for example by doing:

```
void testApp::update(){
    radius = radius + 0.1;
}
```

we get spirals.

7.4.1.3 Lisajous figures

Finally, if we alter the angles we pass in to x and y for this formula in different rates, we can get interesting figures, called "Lissajous" figures, named after the French mathematician, Jules Antoine Lissajous. These formulas look cool. Often times I joke with my students in algo class about how this is really a course to make cool screen savers.

7.4.2 Noise

Noise is similar sin/cos in that it's a function that takes some input and produces output, which we can then use for movement. In the case of sin/cos you are passing in an angle and getting a results back that goes back and forth between -1 and 1. In openframeworks we wrap code that uses simplex noise[4], which is comparable to Perlin noise and we have a function `ofNoise()` that takes an input and produces an output. Both algorithms (Perlin, Simplex) provide a psueduo random noise pattern – they are quite useful for animation, because they are continuous functions, unlike something like ofRandom, which just returns random values.

When I say continuous function, what I mean is if you pass in smaller changes as input, you get smaller output and if you pass in the same value you get the same result. For example, `sin(1.7)` always returns the same value, and `ofNoise(1.7)` also always returns the same result. Likewise if you call `sin(1.7)` and `sin(1.75)` you get results that are continuous (meaning, you can call `sin(1.71)sin(1.72)... sin(1.74)` to get intermediate results).

You can do the same thing with ofNoise – here, I write a for loop to draw noise as a line. `ofNoise` takes an input, here i/10 and produces an output which is between 0 and 1. `ofSignedNoise` is similar but it produces an output between -1 and 1.

Figure 7.5: noise line

```
ofBackground(0,0,0);
ofSetColor(255);

ofNoFill();
ofBeginShape();
for (int i = 0; i < 500; i++){
```

[4] http://en.wikipedia.org/wiki/Simplex_noise

```
    float x = i;
    float noise = ofNoise(i/10.0);
    float y = ofMap(noise, 0,1, 0, 100);
    ofVertex(x,y);
}
ofEndShape();
```

If you alter the i/10.0, you can adjust the scale of the noise, either zooming in (ie, i/100.0), so you see more details, or zooming out (ie, i/5.0) so you see more variation.

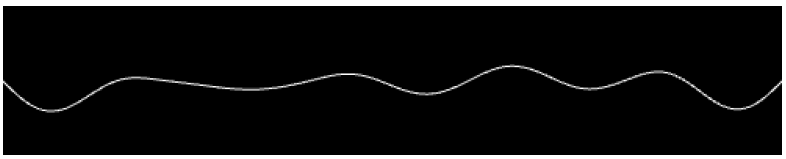


Figure 7.6: noise with i dividied by 100

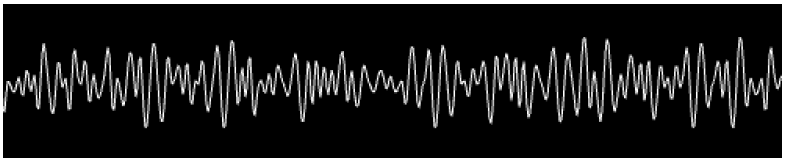


Figure 7.7: noise with i dividied by 5

We can use noise to animate, for example, here, we move an object on screen using noise:

```
float x = ofMap( ofNoise( ofGetElapsedTimef()), 0, 1, 0,
   ofGetWidth());
ofCircle(x,200,30);
```

If we move y via noise, we can take a noise input value somewhere "away" from the x value, ie further down the curved line:

```
float x = ofMap( ofNoise( ofGetElapsedTimef()), 0, 1, 0,
   ofGetWidth());
float y = ofMap( ofNoise( 1000.0+ ofGetElapsedTimef()), 0, 1, 0,
   ofGetHeight());
ofCircle(x,y,30);
```

Alternatively, ofNoise takes multiple demensions. Here's a quick sketch moving something in a path via ofNoise using the 2d dimensions

The code for this example (note the 2 inputs into ofNoise, this is a 2-dimensional noise call. it allows us to use the same value for time, but get different results):

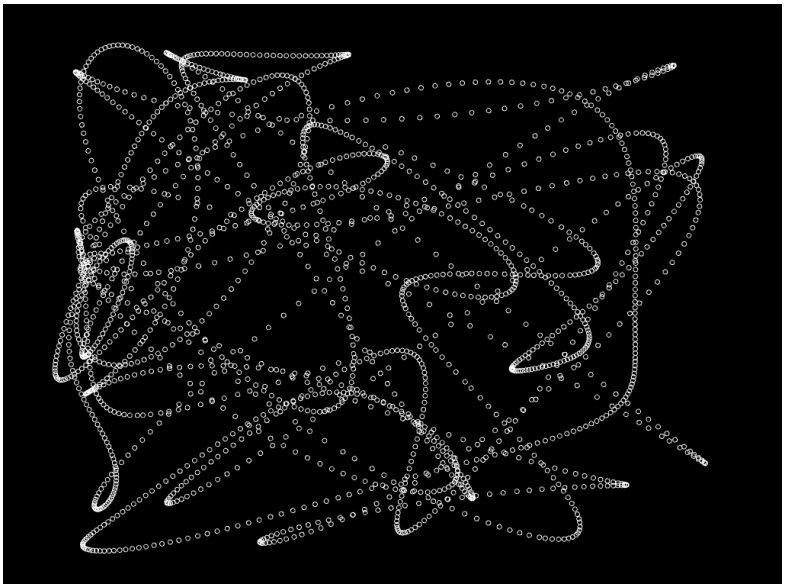


Figure 7.8: noise via 2d

```
//--------------------------------------------------------------
void ofApp::setup(){
    ofBackground(0);
    ofSetBackgroundAuto(false);
}

//--------------------------------------------------------------
void ofApp::update(){
}

//--------------------------------------------------------------
void ofApp::draw(){

    float x = ofMap( ofNoise( ofGetElapsedTimef()/2.0, -1000), 0, 1,
       0, ofGetWidth());
    float y = ofMap( ofNoise( ofGetElapsedTimef()/2.0, 1000), 0, 1,
       0, ofGetHeight());
    ofNoFill();
    ofCircle(x,y,3);

}
```

There's a ton more we can do with noise, we'll leave it for now but encourage you to look at the noise examples that come with openframeworks, which show how noise can be use to create lifelike movement. Also, we encourage readers to investigate the

work of Ken Perlin[5], author of the simplex noise algorithm – he's got great examples of how you can use noise in creative playful ways.

7.5 Simulation

If you have a photograph of an object at one point in time, you know its position. If you have a photograph of an object at another point in time and the camera hasn't changed, you can measure its velocity, i.e., its change in distance over time. If you have a photograph at three points in time, you can measure its acceleration, i.e., how much the speed changing over time.

The individual measurements compared together tell us something about movement. Now, we're going to go in the opposite direction. Think about how we can use measurements like speed and acceleration to control position.

If you know how fast an object is traveling, you can determine how far it's traveled in a certain amount of time. For example, if you are driving at 50 miles per hour (roughly 80km / hour), how far have you traveled in one hour? That's easy. You've traveled 50 miles. How far have you traveled in two or three hours? There is a simple equation to calculate this distance:

```
position = position + (velocity * elapsed time)
```

e.g.:

```
position = position + 50 * 1;   // for one hour away
```

or

```
position = position + 50 * 2;   // for two hours driving
```

The key expression–position = position + velocity–in shorthand would be `p=p+v`.

Note, the elapsed time part is important, but when we animate we'll be doing p=p+v quite regularly and you may see us drop this to simplify things (assume every frame has an elapsed time of one). This isn't entirely accurate but it keeps things simple. See the previous section on frame rate (and frame rate independence) for more details

In addition, if you are traveling at 50 miles per hour (apologies to everyone who thinks in km!) and you accelerate by 5 miles per hour, how fast are you driving in 1 hr? The answer is 55 mph. In 2 hrs, you'd be traveling 60 mph. In these examples, you are doing the following:

```
velocity = velocity + acceleration
```

[5] http://mrl.nyu.edu/~perlin/

In shorthand, we'll use `v=v+a`. So we have two important equations for showing movement based on speed:

```
p = p + v;   // position = position + velocity
v = v + a;   // velocity = velocity + acceleration
```

The amazing thing is that we've just described a system that can use acceleration to control position. Why is this useful? It's useful–if you remember from physics class–because Newton had very simple laws of motion, the second of which says

```
Force = Mass x Acceleration
```

In shorthand, `F = M x A`. This means force and acceleration are linearly related. If we assume that an object has a mass of one, then force equals acceleration. This means we can use force to control velocity and velocity to control position.

The cool, amazing, beautiful thing is that are plenty of forces we can apply to an object, such as spring forces, repulsion forces, alignment forces, etc.

I have several particle examples that use this approach, and while I won't go deeply into them, I'll try to explain some interesting ideas you might find

7.5.1 particle class

The particle class in all of the examples is designed to be pretty straight forward. Let's take a look at the H file:

```
class particle{

    public:

        ofPoint pos;
        ofPoint vel;
        ofPoint frc;
        float damping;

        particle();
        void setInitialCondition(float px, float py, float vx, float
           vy);

        void resetForce();
        void addForce(float x, float y);
        void addDampingForce();

        void update();
        void draw();
};
```

For variables, it has ofPoint objects for position, velocity and force (abbreviated as pos, vel and frc). It also has a variable for damping, which represents how much this object slows down over time. A damping of 0 would mean not slowing down at all, and as damping gets higher, it's like adding more friction - imagine rolling a ball on ice, conrete or sand, it would slow down at different rates.

In terms of functions, it has a contructor which sets some internal variables like damping and a setInitialCondition() that allows you to set the position and velocity of the particle. Think about this as setting up its initial state, and from here you let the particle play out. The next three functions are about forces (we'll see more) – the first one, `resetForce()`, clears all the internal force variable frc. Forces are not cummualtive across frames, so at the start of every frame we clear it. `addForce()` adds a force in a given direction, useful for constant forces, like gravity. `addDampingForce()` adds a force opposite velocity (damping is a force felt opposite the direction of travel). Finally, update takes forces and adds it to velocity, and takes velocity and adds it to position. Draw just draws a dot where position is.

The particle class is really simple, and throughout these examples, we add complexity to it. In general though formula you will see in all the examples is:

```
for (int i = 0; i < particles.size(); i++){
    particles[i].resetForce();
}

// <------ magic happens here --------->

for (int i = 0; i < particles.size(); i++){
    particles[i].update();
}
```

where the magic is happening between the reset force and update. Although these examples increase in complexity, they do so simply by adding new functions to the particle class, and adding more things between reset and update.

[note: add screenshot of simple particle examples]

7.5.2 simple forces, repulsion and attraction

In the next few examples, I added a few functions to the particle object:

```
void addRepulsionForce( float px, float py, float radius, float
   strength);
void addAttractionForce( float px, float py, float radius, float
   strength);
void addClockwiseForce( float px, float py, float radius, float
   strength);
```

```
void addCounterClockwiseForce( float px, float py, float radius,
    float strength);
```

They essentially adds forces the move towards or away from a point that you pass in, or in the case of clockwise forces, around a point.

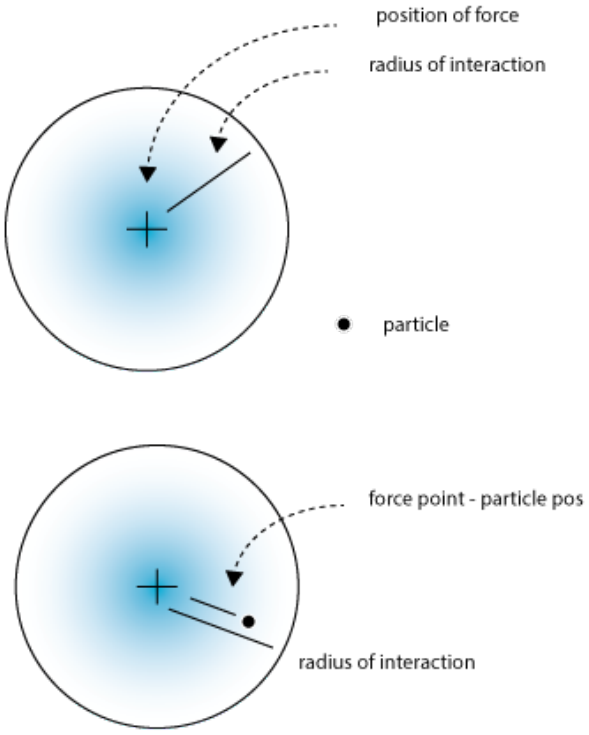


Figure 7.9: sin

The calculation of these forces is fairly straight forward - first, we figure out how far away from a point is from the center of the force. If it's outside of the radius of interaction, we disregard it. If it's inside, we figure out its percentage , ie, the distance between the force and the particle devided by the radius of interaction. This gives us a number that's close to 1 when we towards the far edge of the circle and 0 as we get towards the center. If we invert this, by taking 1 - percent, we get a number that's small on the outside, and larger as we get closer to the center.

This is useful because often times forces are proportional to disctance. For example, a magnetic force will have a radius at which it works, and the closer you get to the magnet the stronger the force.

Here's a quick look at one of the functions for adding force:

```
void particle::addAttractionForce( float px, float py, float radius,
    float strength){

    ofVec2f posOfForce;
    posOfForce.set(px, py);
    ofVec2f diff = pos - posOfForce;

    if (diff.length() < radius){
        float pct = 1 - (diff.length() / radius);
        diff.normalize();
        frc.x -= diff.x * pct * strength;
        frc.y -= diff.y * pct * strength;
    }
}
```

`diff` is a line between the particle and the position of the force. If the length of diff is less then the radius, we calculate the pct as a number that goes between 0 and 1 (0 on the outside of the radius of interaction, 1 as we get to the center of the force). We take the line `diff` and normalize it to get a "directional" vector, its magnitude (distance) is one, but the angle is still there. We then multiply that by pct * strength to get a line that tells us how to move. This gets added to our force.

You'll notice that all the code is relatively similar, but with different additions to force. For example, repulsion is just the opposite of attraction:

```
        frc.x += diff.x * pct * strength;
        frc.y += diff.y * pct * strength;
```

We just move in the oppisite direction. For the clockwise and counter clockwise forces we add the perpindicular of the diff line. The perpindicular of a 2d vector is just simply switching x and y and making one of them negative.

[more: show example]

7.5.3 particle particle interaciton

Now that we have particles interacting with forces, the next step is to give them more understanding of each other. For example, if you have a broad attraciton force, they will all converge on the same point without any respect for their neighbors. The trick is to add a function that allows the particle to feel a force based on their neighbor.

We've added new functions to the particle object (looking in the h file):

```
void addRepulsionForce(particle &p, float radius, float scale);
void addAttractionForce(particle &p, float radius, float scale);
```

This looks really similar to the code before, except here we pass in a particle instead of an x and y position. You'll notice that we pass by reference (using &) as opposed to passing by copy. This is because internally we'll alter both the particle who has this function called as well as particle p – ie, if you calculate A vs B, you don't need to calculate B vs A.

```
void particle::addRepulsionForce(particle &p, float radius, float
    scale){

    // ----------- (1) make a vector of where this particle p is:
    ofVec2f posOfForce;
    posOfForce.set(p.pos.x,p.pos.y);

    // ----------- (2) calculate the difference & length

    ofVec2f diff    = pos - posOfForce;
    float length    = diff.length();

    // ----------- (3) check close enough

    bool bAmCloseEnough = true;
    if (radius > 0){
        if (length > radius){
            bAmCloseEnough = false;
        }
    }

    // ----------- (4) if so, update force

    if (bAmCloseEnough == true){
        float pct = 1 - (length / radius);  // stronger on the inside
        diff.normalize();
        frc.x = frc.x + diff.x * scale * pct;
        frc.y = frc.y + diff.y * scale * pct;
        p.frc.x = p.frc.x - diff.x * scale * pct;
        p.frc.y = p.frc.y - diff.y * scale * pct;
    }
}
```

The code should look very similar to before, except with these added lines:

```
frc.x = frc.x + diff.x * scale * pct;
frc.y = frc.y + diff.y * scale * pct;
p.frc.x = p.frc.x - diff.x * scale * pct;
p.frc.y = p.frc.y - diff.y * scale * pct;
```

This is modifying both the particle you are calling this on and the particle that is passed in.

This means we can cut down on the number of particle particle interactions we need to calculate:

```
for (int i = 0; i < particles.size(); i++){
    for (int j = 0; j < i; j++){
        particles[i].addRepulsionForce(particles[j], 10, 0.4);
    }
}
```

you'll notice that this 2d for loop, the inner loop counts up to the outer loop, so when i is 0, we don't even do the inner loop. When i is 1, we compare it to 0 (1 vs 0). When i is 2, we compare it to 0 and 1 (2 vs 0, 2 vs 1). This way we never compare a particle with itself, as that would make no sense (although we might know some poeple in our lives that have a strong self attraction or repulsion).

[note: maybe a diagram to clarify]

One thing to note is that even through we've cut down the number of calculations, it's still quite a lot! This is a problem that doen't scale linearly. In computer science, you talk about a problem using "O" notation, ie big O notation. This is a bite more like Oˆ2 / 2 – the complexity is like 1/2 of a square. If you have 100 particles, you are doing almost 5000 calculations (100 * 100 / 2). If you have 1000 particles, it's almost half a million. Needless to say, lots of particles can get slow...

We don't have time to get into it in this chapter, but there's different approaches to avoiding that many calculations. Many of them have to deal with spatial hashing, ways of quickly identifying which particles are far away enough to not even consider (thus avoiding a distance calculation).

7.5.4 local interactions lead to global behavior

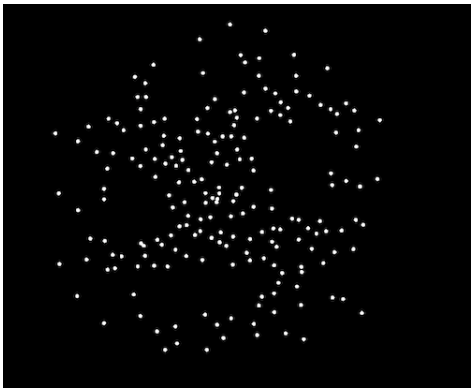


Figure 7.10: sin

7.6 where to go further

7.6.1 physics and animation libraries

8 Information Visualization Chapter

by Tega Brain[1]

This chapter gives a brief overview of working with data in OpenFrameworks and introduces some basic information visualisation techniques. It describes steps in the data visualisation process, common file formats and useful functions for converting data. It is structured using two specific examples; a time based plot and a map with geolocated data.

8.1 Intro

8.1.1 What is data? What is information?

Computation has driven a huge increase in our capacity to collect, sort and store data and yet our ability to understand it remains limited by our sensory and cognitive capacities. A visual process that is used across multiple fields, data visualisation is a way of interpreting and presenting data and can potentially reveal trends and patterns that might otherwise remain invisible.

Data are symbols or numerical interpretations that represent the properties of objects and environments (Ackoff, 1989). Information is produced from analysing the context and descriptive qualities of data, it relates to why the data was collected. Although these terms are often used interchangeably, in the field of information science data is generally thought of as a raw material from which information is produced through analytical processes.

8.1.2 Steps of visualising data

Ben Fry is a data artist and the author of Visualizing Data (2008), a well-known text outlining data visualisation approaches for the Processing programming environment. In this excellent reference text Fry describes seven stages for visualising data and these provide a useful structure for approaching data-driven projects. These steps are:

*Acquire: Obtain the data. Data is commonly made available in files downloadable from online sources such as weather services, newspapers, census records and from

[1] http://www.tegabrain.com/

social media platforms. However there are also times when you may need to compile and format data from hardware or sensors. Real-time data is often available via an Application Programming Interfaces (API), which is an interface or a set of rules that define the process of how other applications can communicate with it. Every API is designed differently and therefore can be communicated with in different ways. This chapter presents two examples of visualising a static dataset imported into OF from external files, and an example that reads data from the New York Times API.

*Parse: Provide some structure for the data's meaning, and order it into categories. Once you have obtained your data, before you load it into OpenFrameworks it is important to parse the data. Parsing means checking the file's format. Is the dataset tagged correctly? Check that each line of your data is broken up consistently across columns. This can be done in a number of ways such as, printing your file out in the terminal or opening your file in a text editor or spreadsheet program and checking for inconsistencies or gaps.

*Filter: Remove all but the data of interest. Your dataset is likely to contain extra information not relevant to your visualisation. For example in the tab separated (.tsv) file shown in figure 1, the file has columns like station ID and latitude and longitude that are not relevant to the first visualisation example. As the data is from only one location, location information can be removed so that they do not interfere with the your visualisation process.

*Mine: Apply methods from statistics or data mining to discern patterns in your data and place the data in mathematical context. As Fry (2008) outlines, the mining stage of visualising data involves applying statistical methods and math to your dataset to analyse patterns and trends within it. This might be as simple as identifying the minimum and maximum values so that you know the range of variation in its values. Depending on your data, you may wish to calculate an average or a median value.

*Represent: Choose a basic visual model, such as a bar graph, list, or tree.

*Refine: Improve the basic representation to make it clearer and more visually engaging.

*Interact: Add methods for manipulating the data or controlling what features are visible.

8.2 Working with data files in OpenFrameworks

8.2.1 Common data file structures: tsv, csv, xml, json

Data is available and stored in specific file types that have particular structures and syntax. The following file types are some of the most common forms of structuring data.

CSV: Comma separated values (csv) files are files where entries in the lines of the file are separated by commas. These can be directly imported into OF by using the ofxCsv add-on. This add-on allows for the reading and writing of CSV file. TSV: Tab separated value files are text files where entries in the lines of the file are separated by tabs. These can be directly imported into OF. *XML: XML files are written in EXtensible Markup Language. XML files are composed of tags that define a data hierarchy for the values within them. A tag has a name, attributes and values within it. If tags are nested the enclosing tags are called parent tags and the nested tags are the children. The tags next to one another are siblings.

```
<parentTagName>
    <childtagName
       attributeName="attributeValue">TagValue</childtagName>
    <siblingTag />
</parentName>
```

Reading an XML file in OF requires the use of an OF addon called ofXmlSettings.

*JSON: JSON stands for 'javascript object notation'. This is a human readable file that is built on two structures, a collection of name/value pairs which can be realised in OF as a struct and an ordered list of values, realised as a vector. Json files also are parsed using an OF addon called ofxJSON, see example 2.XX for how to implement this.

8.2.2 Example - Visualising Time Series Plot

Step 1 Acquire: This section works through an example of a data visualisation of US population data downloaded from the United States Census service here: http://www.nber.org/data/census-decennial-population.html

Step 2 Parse and Filter: Open this file in a spreadsheet program and inspect its contents. You will see that there is an population data for all the regions of the USA from 1900-1990. This example visualises the total population data and data from New York, Louisiana and Alabama so we must construct the data file with only the data from those particular states. You will want to copy and past the selected data into a new spreadsheet so that you are working with a file structure that looks like Figure 1. If you are working in Excel to parse the data, this program has a useful way of transposing the table. Copy a row from the original spreadsheet, and then paste it into your new file by selecting the "Paste Special" option in the Edit menu and selecting "Transpose" before hitting ok.

IMAGE HERE.

Check the data for any gaps or strange characters.

Ensure that you do not have any extra labelling or text at the top and bottom of the data columns. Your final file should only have a row of data labels in the first row.

Step 3 Mine: Check each variable for minimum and maximum values so that you know the approximate range of variation. Check for any strange outlying values.

Save your file as a tsv file.

Loading Your file into an OF Project Firstly generate a new project remembering to include the add-ons if your data is in csv or json format. Then save the parsed population data file to the 'bin' folder of your OF project.

Once your project file is set up, we will now work through writing the code.

Organising your data. Firstly we will need several structures to keep our data organised and allow easy access to it.

*Vectors? Explain here – or has this happened elsewhere?

Vectors are an important data structure for storing lists of data in OF. Here we define a vector of structs, where each struct is a list of variables, and each list of variables holds the values from each line of our data file.

Structs ? Structs are useful way for declaring lists of related variables with one name and stored in one block of memory. In this case, each list contains data points from each column of our data file. Each variable of a struct can then be accessed by a single pointer. The following defines a struct called popData that contains five variables. Like all declarations, structs are declared in the h file of your program. This struct is then wrapped in a vector called dataPoints.

IN THE MAIN.CPP FILE: We will add fAppGlutWindow.h to the main.cpp file. This makes the default window manager based on glut. This class provides all the functionality to create a window, change/query it's size and position. Our file will look like this:

```
#include "ofMain.h"
#include "testApp.h"
#include "ofAppGlutWindow.h"


int main( ){

    ofAppGlutWindow window;
    ofSetupOpenGL(&window, 1024,768, OF_WINDOW);          //
       <-------- setup the GL context

    // this kicks off the running of my app
    // can be OF_WINDOW or OF_FULLSCREEN
    // pass in width and height too:
    ofRunApp( new testApp());

}
```

IN THE H FILE: We define a struct called timeData which will hold the values from each line of our file.

```
typedef struct {

    int year;
    float ny;
    float lou;
    float ala;

} timeData;
```

We will then declare a vector that contains a list of structs, one for each line of our data file.

```
class testApp : public ofBaseApp{
    public:
vector < popData > dataPoints;
};
```

Explain typedef ???

We will also need to declare some variables to contain some minimum and maximum values from our dataset.

```
int minYear;
    int maxYear;
    float maxValue;

    ofRectangle dimensions;
```

In summary, our H FILE will look like this:

```
#pragma once

#include "ofMain.h"


typedef struct {

    int year;
    float ny;
    float lou;
    float ala;

} popData;


class testApp : public ofBaseApp{
```

```
public:
    void setup();
    void update();
    void draw();

    void keyPressed  (int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y );
    void mouseDragged(int x, int y, int button);
    void mousePressed(int x, int y, int button);
    void mouseReleased(int x, int y, int button);
    void windowResized(int w, int h);
    void dragEvent(ofDragInfo dragInfo);
    void gotMessage(ofMessage msg);


    vector < popData > dataPoints;

    int minYear;
    int maxYear;
    float maxValue;

    ofRectangle dimensions;


};
```

IN THE TESTAPP FILE:

We will now load the data file into OF using the ofBuffer class.

8.2.2.1 ofBuffer Class

ofBuffer will read the data into a buffer which is temporary storage for it as we write code to restructure and process it. ofBuffer is what is known as a convenience class, and provides easy methods for reading from and writing to files. A convenience class simply means that this is a class that doesn't do anything by itself but wraps or allows access to the functionality of a group of other classes.

8.2.2.2 Buffer Functions

ofBufferFromFile(); is a function that allows you to load your data file.

```
ofBuffer file = ofBufferFromFile“(population.tsv");
cout␣<<␣file.getText();
```

This loads the population.tsv file into a variable called 'file'. Then we have printed out the contents of the variable 'file' using getText() which allows us to check the file has loaded correctly.

getFirstLine(); Returns all text up to the first new line which ends at the first carriage return.

```
string nameLine = file.getFirstLine();
```

We have used getFirstLine(); to copy the first line of the file containing the labels into a string called 'nameLine'.

getNextLine(); Returns the next row of the data file as marked by \n or \r (new line characters). isLastLine(); Returns the last line of the file.

These functions can be combined to run through each line of data in the buffer and . We can nest this function in a conditional function that checks we are not at the last line of our file, here the while() loop is used. When the last line of the file is reached, our program will leave the buffer and this loop.

```
    while (!file.isLastLine()){
            string line = file.getNextLine();
            vector < string > split = ofSplitString(line, "\t");
            popData data;
            data.year = ofToInt(split[0]);
            data.ny = ofToFloat(split[1]);
            data.lou = ofToFloat(split[2]);
            data.ala = ofToFloat(split[3]);
            dataPoints.push_back(data);
        }
```

This block of code has arranged our data into a vector called dataPoints which contains a struct.

Using this structure, any data points can now be accessed by referring to the vector, the variable name and the index number.

For example dataPoints[0].pop; returns the first entry of the pop list.

Now we have loaded the data from our file into a structure that allows us to access it and manipulate it easily from the rest of our program. Putting this all together in the testApp::setup() is shown below. The last five lines of this code sets up the dimensions and colour of the graph.

```
void testApp::setup(){


    ofBuffer file = ofBufferFromFile("population1.tsv");
    cout << file.getText();
```

```
    // grab the first line, which is just names.
    string nameLine = file.getFirstLine();

    while (!file.isLastLine()){
        string line = file.getNextLine();
        vector < string > split = ofSplitString(line, "\t");
        popData data;
        data.year = ofToInt(split[0]);
        data.ny = ofToFloat(split[1]);
        data.lou = ofToFloat(split[2]);
        data.ala = ofToFloat(split[3]);
        dataPoints.push_back(data);
    }

// let's round up to the next "10" on the max value
    maxValue = ceil(maxValue / 10) * 10;


    // let's find the min and max years, and the max value for the
       data.
    // years are easy, we know it's the first and last year of the
       array.

    minYear = dataPoints[0].year;
    maxYear = dataPoints[dataPoints.size()-1].year;

    // search linealy through the data to find the max value;

    maxValue = 0;
    for (int i = 0; i < dataPoints.size(); i++){
        if (dataPoints[i].ny > maxValue){
            maxValue = dataPoints[i].ny;
        }
        if (dataPoints[i].lou > maxValue){
            maxValue = dataPoints[i].lou;
        }
        if (dataPoints[i].ala > maxValue){
            maxValue = dataPoints[i].ala;
        }
    }

    // let's round up to the next "10" on the max value
    maxValue = ceil(maxValue / 10) * 10;


    dimensions.x = 150;
    dimensions.y = 150;
    dimensions.width = 700;
    dimensions.height = 400;
```

```
    ofBackground(180,180,180);

}
```

Step 4 Represent.

The draw() part of the code incorporates a for loop maps the full range of values across the first variable pop to the dimensions of the graph. This is a way to make our graph responsive. If we were to now load in different data, it would be remapped according to how many data points it contains.

```
    void testApp::draw(){

    ofSetColor(255,255,255);
    ofRect(dimensions.x, dimensions.y, dimensions.width,
       dimensions.height);

    ofSetColor(90,90,90);
    for (int i = 0; i < dataPoints.size(); i++){

        float x = dimensions.x + ofMap( dataPoints[i].year, minYear,
           maxYear, 0,dimensions.width);
        float y = dimensions.y + ofMap( dataPoints[i].pop, 0,
           maxValue, dimensions.height, 0);

        ofCircle(x,y, 2);
    }

}
```

Now you have a very basic working graph, the next steps are to add labels and interactivity.

Step 6 Refine. We need to declare a font in testApp.h file:

```
    ofTrueTypeFont font;
    ofTrueTypeFont labelFont;
```

and then in the testApp.cpp file in setup we load the font:

```
    font.loadFont("bfont.ttf", 20);
    labelFont.loadFont("bFont.ttf", 10);
```

We add x axis labels using the following block of code:

```
    for (int i = 0; i < dataPoints.size(); i++){
```

```
        if (dataPoints[i].year % 10 == 0){
            float x = dimensions.x + ofMap( dataPoints[i].year,
               minYear, maxYear, 0,dimensions.width);
            float y = dimensions.y + dimensions.height;
            ofSetColor(90,90,90);
            labelFont.drawString(ofToString( dataPoints[i].year), x,
               y + 20);
            ofSetColor(220,220,220);
            ofLine(x, y, x,  dimensions.y);
        }
    }
```

So what does this mean? First we are iterating though each line of the dataPoints vector using a for loop. For each value of i, we access each line of the vector. As we do this we check:

```
if (dataPoints[i].year % 10 == 0){
```

This if statement asks if the year is divided by 10, is there a remainder? In other words, it is a conditional that will only be true for every 10 year interval (no remainder). Within this conditional, we are then calculating an x value that is mapped from the start of our graph area to the end (0 to dimensions.width). This evenly spaces our x values. The y value is calculating the top of the graph (dimensions.y + dimensions.height).

We then print the labels to the screen with:

```
labelFont.drawString(ofToString( dataPoints[i].year), x, y + 20);
```

So this selects the font (labelFont.) writes to the screen (drawString) converts the data to a string variable type (ofToString) and then takes the year from each line of the vector (dataPoints[i].year) and positions it at the x and (y+20) calculated.

Lastly we draw some grid lines at each x value from the top of the graph (y) to the bottom of the graph (dimensions.height).

We add y axis labels using this code:

```
for (int i = 0; i <= (int)maxValue; i++){
        if (i % 1000000 == 0){
            float x = dimensions.x;
            float y = dimensions.y + ofMap(i, 0, maxValue,
               dimensions.height, 0);

            ofSetColor(90,90,90);
            labelFont.drawString(ofToString( i ), x - 30, y + 5);
            ofLine(x,y, x-5,y);
        }
    }
```

Here we have a for loop generating values for i that range from 0 to maxValue. Similarly, the if statement will only generate ticks and labels for every 1000000 of these values. What interval is chosen for this statement will depend on the range of the data. As we are dealing with population data, we will choose to have a label at intervals of 1 million. This time y values are mapped from 0 to the height of the graph so they are spread evenly. And then text and a line is drawn using the same functions as on the x axis.

Step 6 Interact.

Finally we can add interactivity by creating clickable tabs that will switch between the different datasets in our file. This section turns our code into a state machine, where we define a variable called 'which' which is toggled between the values 0,1 and 2. The value of 'which', dictates what dataset will be displayed.

In testApp.h, declare which:

```
int which;
```

Then insert this block of code to the mousePressed part of the openFrameworks template. These conditionals define regions around the title labels and turns them into buttons. When the button is clicked, 'which' changes state to the value shown.

```
void testApp::mousePressed(int x, int y, int button){


ofRectangle rect = font.getStringBoundingBox("NewYork,
   dimensions.x, dimensions.y-15);
    if (rect.inside(ofPoint(x,y))){
        which = 0;
    }

    rect = font.getStringBoundingBox("Louisiana, dimensions.x + 80,
       dimensions.y-15);
    if (rect.inside(ofPoint(x,y))){
        which = 1;
    }

    rect = font.getStringBoundingBox("Alabama, dimensions.x + 160,
       dimensions.y-15);
    if (rect.inside(ofPoint(x,y))){
        which = 2;
    }
```

Finally we must return to void testApp::draw() and make some changes. In the for loop where we draw the data points we change from this:

```
    for (int i = 0; i < dataPoints.size(); i++){
```

```
        float x = dimensions.x + ofMap( dataPoints[i].year, minYear,
            maxYear, 0,dimensions.width);
        float y = dimensions.y + ofMap( dataPoints[i].ny, 0,
            maxValue, dimensions.height, 0);

        ofCircle(x,y, 2);
    }
```

to this:

```
    for (int i = 0; i < dataPoints.size(); i++){

        float value;
        if (which == 0) value = dataPoints[i].ny;
        if (which == 1) value = dataPoints[i].lou;
        if (which == 2) value = dataPoints[i].ala;

        float x = dimensions.x + ofMap( dataPoints[i].year, minYear,
            maxYear, 0,dimensions.width);
        float y = dimensions.y + ofMap( value, 0, maxValue,
            dimensions.height, 0);


        ofCircle(x,y, 2);
    }
```

We have created a new float 'value' to hold each data point. Depending on the value of 'which', value is assigned data from one of the three data sets in the tsp file.

Finally the last step here is to draw the titles to the screen which is done by adding the last block of code underneath the for loop we just changed.

```
    if (which == 0) ofSetColor(180,90,90);
        else ofSetColor(90,90,90);
        font.drawString("Milk", dimensions.x, dimensions.y-15);

        if (which == 1) ofSetColor(180,90,90);
        else ofSetColor(90,90,90);
        font.drawString("Tea", dimensions.x + 80, dimensions.y-15);

        if (which == 2) ofSetColor(180,90,90);
        else ofSetColor(90,90,90);
        font.drawString("Coffee", dimensions.x + 160,
            dimensions.y-15);
```

8.3 More Useful functions for working with data

8.3.1 Conversion functions (ofSplitString, ofToString, ofToInt)

Conversion functions enable the manipulation of each line of data in the buffer. They allow each line to be split and for parts of it to be placed into string or integer variables.

```
ofSplitString(line, "\"t );
```

This function splits a string at a specified character. It has two arguments, the first is the name of the string to be split and the second is the character at which it is to be split. (indicates split at a tab)

```
string ofToString(integer, value);
```

ofToString object takes a number and turns it into a string representation of that number. The first argument is the integer to be transformed and the second indicates to how many decimals places you want to use. If you do not specify the second value, then the default value used is 7.

```
ofToInt(const string &intString);
ofToInt(string);
```

Similar to the previous, this object converts string or a string representation into an actual integer variable.

```
ofToFloat(const string &intString);
ofToFloat(string);
```

This object converts another variable type into an a float variable.

8.4 Working with APIs

8.4.1 What are APIs?

An API is an Application Programming Interface. This means it is a software to software interface allowing one piece of software to interact automatically with another online software. It takes the form of a set of instructions for how a program can be designed to interface with the online service. Every API is different with some being very well documented while others are not.

You can write programs in OpenFrameworks so that data an be pulled from an API automatically and used in your sketch. For example if we want to be able to pull data from the New York Times API we would inspect the instructions for doing so here: http://developer.nytimes.com/docs.

8.5 Further resources

JSON validation tools like: http://jsonlint.com/

8.6 References

Fry, B. (2008). *Visualizing Data,* O'Reilly Media. Ackoff, R. L. (1989). *From Data to Wisdom. Journal of Applied Systems Analysis,* 16, 3–9.

9 Experimental Game Development in openFrameworks

by Phoenix Perry[1] and Jane Friedhoff[2]

Game developers are, in greater and greater numbers, turning to openFrameworks' creative coding toolkit to develop their games. Unlike platforms like Unity, GameMaker, and Construct2, oF was not specifically developed for game makers. However, oF's ability to port to mobile, manipulate video, utilize camera input, support generative graphics, and hook in with devices like Arduino and Kinect (among other features) makes it a very attractive option for developers who want to be able to rapidly produce compelling, unique games.

9.0.1 Popular games in open frameworks

In this chapter, we'll learn about game development in openFrameworks. We'll cover what goes into making a game, as well as how to code a simple space shooter. Finally, we'll put an experimental oF twist on our game by implementing OSC functionality, which will allow you to alter the difficulty of the game live—while a player is playing it.

Ready? Let's go!

9.1 How do game developers actually make games?

There are as many ways to make games as there are game developers. However, many developers follow an iterative process: that is, adding a single component, testing it, adding an additional component, testing it again, and so on. Regardless of the platform, this method allows game developers to quickly figure out what parts of the initial idea are worth keeping and rapidly test additions they think might be interesting–without having to risk wasting time on building out a complete game that, in retrospect, isn't compelling.

This iterative process can be done digitally or physically. Paper prototyping is the process of testing mechanics and interactions with paper models and analogs. Although

[1] http://www.phoenixperry.com
[2] janefriedhoff.com

Figure 9.1: Spell Tower by Zach Gage

Figure 9.2: Particle Mace by Andy Wallace

these paper prototypes don't necessarily look like the final game, they can be mocked up quickly and thrown away cheaply, allowing developers to experiment with core mechanics more rapidly than they could with code. For example, a puzzle game's board and pieces can likely be mocked up with paper and dice more quickly than it can be implemented in even a basic mobile app. When a developer makes a digital prototype, or one made with code, they will similarly typically start by refining game mechanics, keeping assets rough until they get closer to the end. Finally, developers enter the long process of tuning their game, tweaking various parameters about the game until it feels just right.

We're going to use openFrameworks to play with the final step of this process. In the game we're making, we're not going to settle on one set of parameters that stay static from game to game. We're going to use openFrameworks' OSC library to allow us to communicate wirelessly from another device (e.g. a smartphone or table) so we can tune those parameters live, giving our players experiences tailored just for them.

9.2 So what is OSC, anyway?

OSC, or Open Sound Control, came about as an advancement to MIDI, so let's talk about MIDI first. MIDI is a data protocol that sends and receives information between devices, typically electronic musical instruments. MIDI is what allowed things like keyboards and drum machines to fire in sync. If you've heard pop music, you've heard MIDI in action.

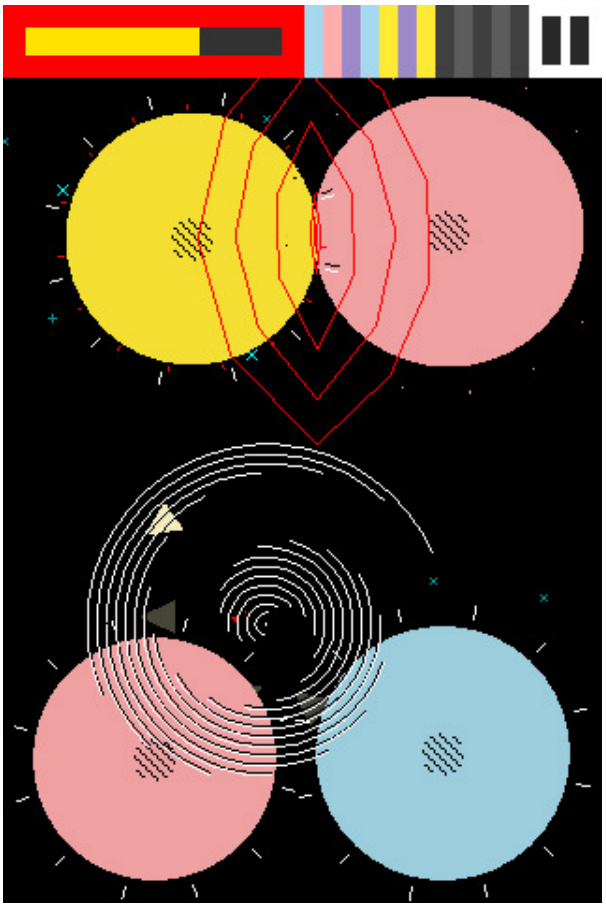


Figure 9.3: Eliss by Steph Thirion

Figure 9.4: Scream Em Up by Jane Friedhoff

MIDI has data channels, on which you can send or receive single messages, or events. Programmers could associate these MIDI events with actions that their electronic instruments could take. For example, you could set up your keyboard to send data on channel 1, and receive data on MIDI channel 2. More specifically, you could program a specific key (say, the 'a' key) to send out a MIDI event on channel 1. If your drum machine is set up to receive on channel 1, it will receive that message and perform the appropriate action (e.g. playing). A pretty cool system, but one that was limited by its pre-defined and discrete message types.

As time advanced, so did computers and the speed of data transfers, leading us to OSC. OSC was designed to allow for more expressive performance data, with different, flexible kinds of messages sent over networks. OSC is a thin layer on top of the UDP protocol, and allows users to send information over networks just by specifying the network address and the incoming and outgoing ports. (UDP is used frequently in games, and it is possible to use both of these protocols at the same time in the same code base with no issues.)

OSC messages consist of the following:

- An address pattern. This is a hierarchical name space, and looks a bit like a Unix filesystem or URL (e.g. `/Address1`). These patterns can effectively be anything you want (e.g. `/EnemySpeed`)–think of them as names for what you send.
- A Type tag string. This simply represents the kind of data being sent (e.g. `int`, `string`).
- Arguments. The actual value that is being transmitted (e.g. `6`, `"Hello world"`, etc.).

There are plenty of inexpensive apps for smartphones and tablets that provide customizable GUIs (complete with buttons, sliders, etc.) for sending different kinds of MIDI messages. Download one (we like TouchOSC) so we have something to send our messages with. With this in mind, let's start making our game!

9.3 Our basic game–& making it not-so-basic

OpenFrameworks handles OSC as an included addon, so our first step will be to run the project generator and create a project with the OSC addon. (If you haven't had a chance to read about addons, now would be a good time to jump over to [here] and do just that!) Launch the project generator, then, in the main menu, click the word "Addons." A popup will appear. Select ofxOsc and then click back. Now, next to the word Addons, you should see ofxOsc. Press "generate". When it completes the project creation process, close the generator and open up the project in either Visual Studio or Xcode. The project will be set up in your myApps folder. Open it now.

Here's what our game will have:

- A player, who has an on-screen position, a movement speed, and an image to represent it
- Some enemies, who have an on-screen position, a movement speed (with the horizontal value based on a sine wave), an image to represent them, and an interval to keep track of when they can shoot next
- A level controller, which has an interval to keep track of when an enemy should be spawned next
- Bullets (for the player and the enemies), which have an on-screen position, images to represent them, a way to keep track of where they come from (player or enemy), and a speed
- Bonus lives, which have an on-screen position, an image to represent them, and a speed

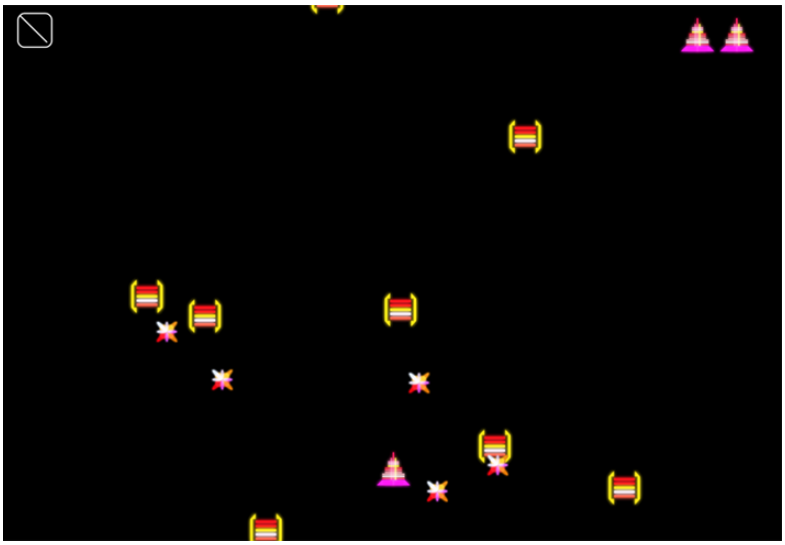


Figure 9.5: Space Game in action!

With all that written out, let's use OSC to affect the following:

- The horizontal movement of our enemies–whether they move in a more exaggerated sin wave, or whether they move in more of a straight line
- The frequency with which our enemies shoot
- The frequency with which our level controller spawn enemies
- Whether a life bonus is on screen or not

These three parameters will allow the developer to, second-by-second, tailor the difficulty of the game to the individual playing it.

Let's start with our testApp. There are a few things we definitely know we'll want classes for, so make corresponding .h and .cpp files for Player, Bullet, Life, Enemy, and LevelCon-

troller. Remember to `#include "ofMain.h"` in each of those classes, and to include the .h file of each of those classes in `testApp.h`.

9.3.1 Gamestates

First let's create the basic structure of our game. Games typically have at least three parts: a start screen, the game itself, and an end screen. We need to keep track of which section of the game we're in, which we'll do using a variable called a game state. In this example, our game state variable is a string, and the three parts of our game are `"start"`, `"game"`, and `"end"`. Let's add a score and a player at this point as well.

```
string game_state;
int score;
Player player_1;
```

We'll then divide up `testApp`'s `update()` and `draw()` loops between those game states:

```
//--------------------------------------------------------------
void testApp::update(){
   if (game_state == "start") {

   } else if (game_state == "game") {
   } else if (game_state == "end") {

   }
}
//--------------------------------------------------------------
void testApp::draw(){
   if (game_state == "start") {
   } else if (game_state == "game") {
   } else if (game_state == "end") {

   }
}
```

Let's set the initial value of `game_state` to `"start"` right when the app begins.

```
//--------------------------------------------------------------
void testApp::setup(){
   game_state = "start";
  score = 0;
}
```

Finally, let's make sure that we can move forward from the start screen. In this example, when the player is on the start screen and releases the space key, they'll be taken to the game.

```
//--------------------------------------------------------------
void testApp::keyReleased(int key){

   if (game_state == "start") {
       game_state = "game";
   } else if (game_state == "game") {
      // blank for now
   }
}
```

9.3.2 Player movement

Great! Let's move onto our player. Our player's class looks like this:

```
class Player {
public:
   ofPoint pos;
   float width, height, speed;
   int lives;

   bool is_left_pressed, is_right_pressed, is_down_pressed, is_up_
      pressed;

   void setup(ofImage * _img);
   void update();
   void draw();
   void shoot();

   void calculate_movement();

   bool check_can_shoot();

   ofImage * img;

};
```

Taking this one step at a time:

- Our player's position will be stored in an `ofPoint` called `pos`. ofPoints are handy datatypes that contain `x` and `y` values, letting us access our player's position through `pos.x` and `pos.y`.
- Our player will have `width`, `height`, and `speed` variables (which we'll use for collision detection and movement, respectively).
- Our player will have an integer number of lives (since it wouldn't make any sense for them to have 4.33333333333 lives).

- Our player will keep track of what movement keys are currently pressed in separate booleans.
- Our player will have `setup`, `update`, `draw`, `shoot`, and `calculate_movement` methods.
- Finally, our player will have a pointer to the image we're using for the player.

You may be wondering why we're using all these booleans–why not just check and see which keys are pressed?

The problem is that, in openFrameworks, `keyPressed()` does not return all the keys currently being pressed–just the last key that was pressed. That means that if the player presses up and left (intending to move diagonally), openFrameworks will only report one of the keys being pressed. You can try printing out the result of keyPressed to see this in action. What we'll do to avoid this is instead base the player's movement on the booleans we wrote earlier. If the player presses a certain key, that boolean will be true; if they release that key, that boolean will be false. That way, if the player presses up and left, we'll report up and left as being true until those keys are released.

Here's what our new `keyPressed()` and `keyReleased()` functions look like:

```
//--------------------------------------------------------------
void testApp::keyPressed(int key){
   if (game_state == "game") {
       if (key == OF_KEY_LEFT) {
           player_1.is_left_pressed = true;
       }

       if (key == OF_KEY_RIGHT) {
           player_1.is_right_pressed = true;
       }

       if (key == OF_KEY_UP) {
           player_1.is_up_pressed = true;
       }

       if (key == OF_KEY_DOWN) {
           player_1.is_down_pressed = true;
       }
   }

}
//--------------------------------------------------------------
void testApp::keyReleased(int key){
   if (game_state == "start") {
       game_state = "game";
   } else if (game_state == "game") {
       if (key == OF_KEY_LEFT) {
           player_1.is_left_pressed = false;
```

```
        }

        if (key == OF_KEY_RIGHT) {
            player_1.is_right_pressed = false;
        }

        if (key == OF_KEY_UP) {
            player_1.is_up_pressed = false;
        }

        if (key == OF_KEY_DOWN) {
            player_1.is_down_pressed = false;
        }
    }
}
```

Add `ofImage player_image` to `testApp.h`, then load the player's image and instantiate the player in `testApp`'s `setup()`:

```
void testApp::setup(){
    game_state = "start";
    player_image.loadImage("player.png");

    player_1.setup(&player_image);
}
```

Finally, update and draw your player in the appropriate part of `testApp::update()` and `testApp::draw()`:

```
//--------------------------------------------------------------
void testApp::update(){
    if (game_state == "start") {

    } else if (game_state == "game") {
        player_1.update();
    }
}

//--------------------------------------------------------------
void testApp::draw(){
    if (game_state == "start") {

    } else if (game_state == "game") {
        player_1.draw();
    } else if (game_state == "end") {

    }
}
```

You should have a player who moves around on-screen. Sweet!? ###Player bullets

Let's make our bullets next. In order to have a variable number of bullets on screen at a time, we need to add a `vector<Bullet> bullets;` to testApp.h. Let's also create a `void update_bullets();` function, which will update our vector of bullets (and, shortly, trigger the check for bullet collisions). We also want our player and enemy bullets to look different, so we'll add `ofImage enemy_bullet_image;` and `ofImage player_bullet_image;` to our `testApp.h` file.

Our bullet class will look a lot like the player class, having a position, speed, width, pointer to an image, and various functions. The big difference is that the bullets will keep track of who they came from (since that will affect who they can hurt and which direction they move).

```
class Bullet {
public:
   ofPoint pos;
   float speed;
   float width;
   bool from_player;

   void setup(bool f_p, ofPoint p, float s, ofImage * bullet_image);
   void update();
   void draw();

   ofImage * img;
};
```

Our `Bullet.cpp` will look like this:

```
void Bullet::setup(bool f_p, ofPoint p, float s, ofImage * bullet_
    image) {
   from_player = f_p;
   pos = p;
   speed = s + 3;
   img = bullet_image;
   width = img->width;

}
void Bullet::update() {
   if (from_player) {
       pos.y -= speed;
   } else {
       pos.y += speed;
   }
}
void Bullet::draw() {
   img->draw(pos.x - width/2, pos.y - width/2);
```

```
}
```

Again, this is much like the code for the player. The two differences are:

- We keep track of where the bullet comes from, and alter the code based on that variable (meaning we can keep all the bullets in the same vector)
- When instantiating a bullet, we check to see the position of the shooter, as well as the shooter's current speed (so it will always move faster than the thing that shot it)

Now that our bullet class is implemented, we can go back to `testApp::setup()` and add in `enemy_bullet_image.loadImage("enemy_bullet.png");` and `player_bullet_image.loadImage("player_bullet.png");` right underneath where we loaded in our `player_image`. For now, our `update_bullets()` function will call the `update()` function in each bullet, and will also get rid of bullets that have flown offscreen in either direction.

```
//--------------------------------------------------------------
void testApp::update_bullets() {
   for (int i = 0; i < bullets.size(); i++) {
       bullets[i].update();
       if (bullets[i].pos.y - bullets[i].width/2 < 0 ||
           bullets[i].pos.y + bullets[i].width/2 > ofGetHeight()) {
            bullets.erase(bullets.begin()+i);
       }
   }
   // 'well call a collision check function here shortly
}
```

Our `testApp::update()` and `testApp::draw()` will now look like this:

```
//--------------------------------------------------------------
void testApp::update(){
   if (game_state == "start") {

   } else if (game_state == "game") {
       player_1.update();
       update_bullets();
   }
}
//--------------------------------------------------------------
void testApp::draw(){
   if (game_state == "start") {

   } else if (game_state == "game") {
       ofBackground(0,0,0);
       player_1.draw();
```

```
        for (int i = 0; i < bullets.size(); i++) {
            bullets[i].draw();
        }
    } else if (game_state == "end") {

    }
}
```

Finally, let's add an if-statement to our `keyPressed()` so that when we press the space-bar during the game, we spawn a player bullet:

```
//--------------------------------------------------------------
void testApp::keyPressed(int key){
    if (game_state == "game") {
        if (key == OF_KEY_LEFT) {
            player_1.is_left_pressed = true;
        }

        if (key == OF_KEY_RIGHT) {
            player_1.is_right_pressed = true;
        }

        if (key == OF_KEY_UP) {
            player_1.is_up_pressed = true;
        }

        if (key == OF_KEY_DOWN) {
            player_1.is_down_pressed = true;
        }

        if (key == ' ') {
            Bullet b;
            b.setup(true, player_1.pos, player_1.speed, &player_
                bullet_image);
            bullets.push_back(b);
        }
    }
}
```

Remember, the first parameter in the bullet's setup is whether it comes from the player (which, in this case, is always true). Run your app and fly around shooting for a bit to see how it feels.

9.3.3 Adding adversaries

Let's move on to our enemy. This process should be familiar by now. Add an `ofImage enemy_image;` and a `vector<Enemy> enemies;` to `testApp.h`. Addition-

ally, add `float max_enemy_amplitude;` and `float max_enemy_shoot_interval;` to `testApp.h`–these are two of the enemy parameters we'll affect with OSC. Your enemy class will look like this:

```
class Enemy {
public:
   ofPoint pos;
   float speed;
   float amplitude;
   float width;

   float start_shoot;
   float shoot_interval;

   void setup(float max_enemy_amplitude, float max_enemy_shoot_
      interval, ofImage * enemy_image);
   void update();
   void draw();
   bool time_to_shoot();

   ofImage * img;
};
```

Our enemy's horizontal movement will be shaped by the values fed to a sine wave (which we'll see in a moment). We'll keep track of our amplitude variable (so different enemies can have different amplitudes). We'll also want to keep track of whether enough time has passed for this enemy to shoot again, necessitating the start_shoot and shoot_interval variables. Both of these variables will actually be set in our setup() function. Finally, we'll have a boolean function that will tell us whether the enemy can shoot this frame or not. Our enemy class will look like this:

```
void Enemy::setup(float max_enemy_amplitude, float max_enemy_shoot_
   interval, ofImage * enemy_image) {
   pos.x = ofRandom(ofGetWidth());
   pos.y = 0;
   img = enemy_image;
   width = img->width;
   speed = ofRandom(2, 7);
   amplitude = ofRandom(max_enemy_amplitude);
   shoot_interval = ofRandom(0.5, max_enemy_shoot_interval);
   start_shoot = ofGetElapsedTimef();
}
void Enemy::update() {
   pos.y += speed;
   pos.x += amplitude * sin(ofGetElapsedTimef());
}
void Enemy::draw() {
   img->draw(pos.x - width/2, pos.y - width/2);
}
```

```
bool Enemy::time_to_shoot() {
    if (ofGetElapsedTimef() - start_shoot > shoot_interval) {
        start_shoot = ofGetElapsedTimef();
        return true;
    }
    return false;
}
```

In update, we're using the current elapsed time in frames to give us a constantly increasing number to feed to the sine function, which in turn returns a value between -1 and 1. We multiply it by the amplitude of the wave, making this curve more or less exaggerated.

In `time_to_shoot()`, we check to see whether the difference between the current time and the time this enemy last shot is greater than the enemy's shooting interval. If it is, we set `start_shoot` to the current time, and return true. If not, we return false. Let's integrate our enemies into the rest of our `testApp.cpp`:

```
//--------------------------------------------------------------
void testApp::setup(){
    game_state = "start";

    max_enemy_amplitude = 3.0;
    max_enemy_shoot_interval = 1.5;

    enemy_image.loadImage("enemy0.png");
    player_image.loadImage("player.png");
    enemy_bullet_image.loadImage("enemy_bullet.png");
    player_bullet_image.loadImage("player_bullet.png");

    player_1.setup(&player_image);
}
//--------------------------------------------------------------
void testApp::update(){
    if (game_state == "start") {

    } else if (game_state == "game") {
        player_1.update();
        update_bullets();

        for (int i = 0; i < enemies.size(); i++) {
            enemies[i].update();
            if (enemies[i].time_to_shoot()) {
                Bullet b;
                b.setup(false, enemies[i].pos, enemies[i].speed,
                    &enemy_bullet_image);
                bullets.push_back(b);
            }
        }
```

```
    } else if (game_state =="draw") {
    }
}
//--------------------------------------------------------------
void testApp::draw(){
    if (game_state == "start") {
    } else if (game_state == "game") {
        ofBackground(0,0,0);
        player_1.draw();
        for (int i = 0; i < enemies.size(); i++) {
            enemies[i].draw();
        }
        for (int i = 0; i < bullets.size(); i++) {
            bullets[i].draw();
        }
    } else if (game_state == "end") {
    }
}
```

9.3.4 Collisions

Let's implement our bullet collision checks. Add a void `check_bullet_collisions();` to your `testApp.h`, then write the following function:

```
//--------------------------------------------------------------
void testApp::check_bullet_collisions() {
    for (int i = 0; i < bullets.size(); i++) {
        if (bullets[i].from_player) {
            for (int e = enemies.size()-1; e >= 0; e--) {
                if (ofDist(bullets[i].pos.x, bullets[i].pos.y,
                    enemies[e].pos.x, enemies[e].pos.y) <
                    (enemies[e].width + bullets[i].width)/2) {
                    enemies.erase(enemies.begin()+e);
                    bullets.erase(bullets.begin()+i);
                    score+=10;
                }
            }
        } else {
            if (ofDist(bullets[i].pos.x, bullets[i].pos.y, player_
                1.pos.x, player_1.pos.y) < (bullets[i].width+player_
                1.width)/2) {
                bullets.erase(bullets.begin()+i);
                player_1.lives--;

                if (player_1.lives <= 0) {
                    game_state = "end";
                }
```

```
            }
        }
    }
}
```

This code is a bit nested, but actually pretty simple. First, it goes through each bullet in the vector and checks to see whether it's from the player. If it's from the player, it starts a for-loop for all the enemies, so we can compare the player bullet position against all the enemy positions. We use `ofDist()` to see whether the distance between a given bullet and a given enemy is less than the sum of their radii–if it is, they're overlapping. If a bullet is not from the player, the function does a distance calculation against the player, to see whether a given enemy bullet and the player are close enough to count it as a hit. If there is a hit, we subtract a player's life and erase that bullet. If the player has less than or equal to 0 lives, we change the game state to the end.

Don't forget to call `check_bullet_collisions()` as part of `update_bullets()`:

```
//--------------------------------------------------------------
void testApp::update_bullets() {
    for (int i = 0; i < bullets.size(); i++) {
        bullets[i].update();
        if (bullets[i].pos.y - bullets[i].width/2 < 0 ||
            bullets[i].pos.y + bullets[i].width/2 > ofGetHeight()) {
             bullets.erase(bullets.begin()+i);
        }
    }
    check_bullet_collisions();
}
```

9.3.5 Our game's brain

Great! Except... we don't have any enemies yet! Definitely an oversight. This is where our level controller comes in. Add `LevelController level_controller;` to your `testApp.h`. Our level controller class is super-simple:

```
class LevelController {
public:
    float start_time;
    float interval_time;

    void setup(float e);
    bool should_spawn();
};
```

As you might guess, all it'll really do is keep track of whether it's time to spawn another enemy yet.

Inside our `LevelController.cpp`:

```
void LevelController::setup(float s) {
   start_time = s;
   interval_time = 500;
}
bool LevelController::should_spawn() {
   if (ofGetElapsedTimeMillis() - start_time > interval_time) {
       start_time = ofGetElapsedTimeMillis();
       return true;
   }
   return false;
}
```

When we set up our level controller, we'll give it a starting time. It'll use this time as a baseline for the first enemy spawn. The should_spawn code should look familiar from the enemy bullet section.

We'll wait to set up our level controller until the game actually starts–namely, when the game state changes from `"start"` to `"game"`.

```
void testApp::keyReleased(int key){
   if (game_state == "start") {
       game_state = "game";
       level_controller.setup(ofGetElapsedTimeMillis());
   }

  ...
}
```

Next we'll integrate it into our `testApp::update()`:

```
//--------------------------------------------------------------
void testApp::update(){
   if (game_state == "start") {

   } else if (game_state == "game") {
       player_1.update();
       update_bullets();

       for (int i = 0; i < enemies.size(); i++) {
           enemies[i].update();
           if (enemies[i].time_to_shoot()) {
               Bullet b;
               b.setup(false, enemies[i].pos, enemies[i].speed,
                   &enemy_bullet_image);
               bullets.push_back(b);
           }
       }
```

```
        if (level_controller.should_spawn() == true) {
            Enemy e;
            e.setup(max_enemy_amplitude, max_enemy_shoot_interval,
                &enemy_image);
            enemies.push_back(e);
        }
    }
}
```

Awesome! We're close to done!

9.3.6 Bonus lives

Before we finish, let's add in our last OSC feature: the ability to throw in bonus lives on the fly. Add `vector<Life> bonuses;` and `ofImage life_image;` to your `testApp.h`. To keep our code modular, let's also add `void update_bonuses();` in the same place. Don't forget to `life_image.loadImage("life_image.png");` in `testApp::setup()`.

`Life.h` should look like this:

```
class Life {
public:
    ofPoint pos;
    float speed;
    float width;

    ofImage * img;

    void setup(ofImage * _img);
    void update();
    void draw();
};
```

And it'll function like this–a lot like the bullet:

```
void Life::setup(ofImage * _img) {
    img = _img;
    width = img->width;
    speed = 5;
    pos.x = ofRandom(ofGetWidth());
    pos.y = -img->width/2;
}
void Life::update() {
    pos.y += speed;
}
void Life::draw() {
    img->draw(pos.x - img->width/2, pos.y - img->width/2);
```

```
}
```

Our `update_bonuses()` function works a lot like the bullet collision function:

```
//--------------------------------------------------------------
void testApp::update_bonuses() {
   for (int i = bonuses.size()-1; i > 0; i--) {
       bonuses[i].update();
       if (ofDist(player_1.pos.x, player_1.pos.y, bonuses[i].pos.x,
          bonuses[i].pos.y) < (player_1.width + bonuses[i].width)/2)
          {
           player_1.lives++;
           bonuses.erase(bonuses.begin() + i);
       }

       if (bonuses[i].pos.y + bonuses[i].width/2 > ofGetHeight()) {
           bonuses.erase(bonuses.begin() + i);
       }
   }
}
```

All that's left for our lives functionality is to alter `testApp::update()` and `testApp::draw()`.

```
//--------------------------------------------------------------
void testApp::update(){
   if (game_state == "start") {

   } else if (game_state == "game") {
       player_1.update();
       update_bullets();
       update_bonuses();

       for (int i = 0; i < enemies.size(); i++) {
           enemies[i].update();
           if (enemies[i].time_to_shoot()) {
               Bullet b;
               b.setup(false, enemies[i].pos, enemies[i].speed,
                  &enemy_bullet_image);
               bullets.push_back(b);
           }
       }

       if (level_controller.should_spawn() == true) {
           Enemy e;
           e.setup(max_enemy_amplitude, max_enemy_shoot_interval,
              &enemy_image);
           enemies.push_back(e);
       }
```

```
    }
}
//--------------------------------------------------------------
void testApp::draw(){
    if (game_state == "start") {
        start_screen.draw(0,0);
    } else if (game_state == "game") {
        ofBackground(0,0,0);
        player_1.draw();
        draw_lives();

        for (int i = 0; i < enemies.size(); i++) {
            enemies[i].draw();
        }

        for (int i = 0; i < bullets.size(); i++) {
            bullets[i].draw();
        }

        for (int i = 0; i < bonuses.size(); i++) {
            bonuses[i].draw();
        }
    } else if (game_state == "end") {

    }
}
```

9.3.7 Let's get visual

Finally, we've been a bit stingy with visual feedback, so let's add in a start screen, a score, a visual representation of the lives left, and an end screen. Add `ofImage start_screen;`, `ofImage end_screen;`, `void draw_lives();`, and `void draw_score();` to `testApp.h`.

Change `testApp::setup()` to load in those assets:

```
//--------------------------------------------------------------
void testApp::setup(){
    ...
    player_1.setup(&player_image);
    start_screen.loadImage("start_screen.png");
    end_screen.loadImage("end_screen.png");
    score_font.loadFont("Gota_Light.otf", 48);
}
```

Draw them in the appropriate game states using `start_screen.draw(0, 0);` and `end_screen.draw(0, 0);`.

Add in the last two functions:

```
//--------------------------------------------------------------
void testApp::draw_lives() {
   for (int i = 0; i < player_1.lives; i++) {
       player_image.draw(ofGetWidth() - (i * player_image.width) -
          100, 30);
   }

}
//--------------------------------------------------------------
void testApp::draw_score() {
   if (game_state == "game") {
       score_font.drawString(ofToString(score), 30, 72);
   } else if (game_state == "end") {
       float w = score_font.stringWidth(ofToString(score));
       score_font.drawString(ofToString(score), ofGetWidth()/2 -
          w/2, ofGetHeight()/2 + 100);
   }
}
```

By using `stringWidth()`, we can calculate the width of a string and shift the text over–handy for centering it.

All that's left after that is to call `draw_score();` and `draw_lives();` during the `testApp::draw()`'s game state, and to call `draw_score();` during the end state.

Congrats–you made a game!

9.3.8 Linking oF and OSC

Now let's add in the OSC functionality. We are going to set our application up to receive messages from our iPad and then make changes in real-time while our game is running to test some possible player scenarios. As mentioned before, this can trump going into your application and making manual changes because you skip the need to recompile your game and playtest live. In fact, you can use Touch OSC to even open up new ways to interact with your players.

Touch OSC is used to switch game levels on the fly and to run challenges.

To accomplish this we are going to create a new class that will contain our OSC functionality. Create a .cpp and .h file for this class now and name it LiveTesting. Open `LiveTesting.h` And let's add the line to import the OSC at the top of your file after your preprocessor directives and also a line for using iostream for testing purposes. As we add the code we will explain in inline in the code comments.

Add the following:

Figure 9.6: Nightgame developer interface by Phoenix Perry

```
#include <iostream>
#include "ofxOsc.h"
```

Next let's set up all of our variables we are going to use to receive OSC data and map it to in game values. Add the following code into your class.

```
class LiveTesting
{
public:
    LiveTesting();
    //a default c++ constructor
    void setup();   //for setup
    void update(); //for updating

    ofxOscSender sender;
    //you can set up a sender!
    //We are going to use this network connection to give us
    //some visual feedback of our current game values.

    ofxOscReceiver receiver;
    //this is the magic! This is the port on which your game gets
       incoming data.

    ofxOscMessage m;
    //this is the osc message your application gets from your device.


    //these are the values we will be tweaking during testing
    float max_enemy_amplitude;
    int interval_time;
    float max_enemy_shoot_interval;
    bool triggerBonus;

};
```

Now let's jump over to the `LiveTesting.cpp` file. In this file we are going to set up our network address and the ports we are sending and receiving data on as the first order of business. However, to go any further we are going to need to do some housekeeping and install additional software. For OSC to work it will need a local wifi network to send the messages across. Note this tactic may not work for a network outside of your own because often a sysadmin will stop this kind of traffic from being transmitted on a very public network. We suggest brining an Airport Express or similar with you so you can quickly and wirelessly establish a local network for playtesting.

For the purpose of this chapter and to allow us to create an experience that will work on both Android and iOS we are going to use a piece of software called TouchOSC from this URL: http://hexler.net/software/touchosc

The desktop editor software is free however the matching software for your device will be $4.99. Get both now. As a mater of principle, we endorse building your own tools and you could easily build a second oF project to be your OSC sender and receiver on your mobile device. With that said, nothing beats TouchOSC for speed, ease of use and complete, platform independent flexibility. If you are someone who often moves between an iOS and Android device on both Windows and Mac, this tool will become indispensible to you. As a games designer it can open up possibilities like changing levels on the fly, updating game variables, adjusting for player feedback and adding new features into and taking them out of your game as it's running. We highly endorse using it and support the continued advancement of the tool. You can also use it with music production tools like Ableton Live and it comes with great presets for things like DJing and mixing music live. Go to the app store of your device and purchase the mobile version now if you would like to continue down this route.

After we get all of the tools downloaded and installed. Let's start setting everything up. You are going to need two bits of information. You are going to need to know the IP address of your computer and the ip address of your laptop. If you are on a mac, just open up your System Preferences. Go to the Network setting and click on your wifi connection in the left sidebar. On the right side it will display your IP address. You can also get this setting by opening up Terminal and entering in the command ifconfig. Terminal will list of every network that's a possible connection for your machine from the past, even if it's not currently active. For example, if you have ever connected your phone, it will be in the list with some flag and listed as inactive. Look for the connection that's currently active. It will look something like this:

```
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu
    1500
    ether 60:33:4b:12:e5:3b
    inet6 fe80::6233:4bff:fe12:e53b%en1 prefixlen 64 scopeid 0x5
    inet 192.168.0.5 netmask 0xffffff00 broadcast 192.168.0.255
    media: autoselect
    status: active
```

The inet address is your current IP. On windows, open the `charms` bar. In search type `cmd` and open the command prompt. Type in `ipconfig`. This information is much clearer than the data dump from terminal. The connected listed as your Wireless LAM adapter Wi-Fi will list your current IPV4 address. This is your IP address. Finally, obtain your mobile device's IP address as well from your device settings. Make a note of your IPAddress for the next section.

At this point, go ahead and launch TouchOSC on your device and the Touch OSC desktop editor on your computer. If you are on Windows, you will need to make sure you have java installed first. Once the software is open, click the open icon in the top tool bar. In the file containing the code for this chapter you will see a file called `ofBook.touchosc`.

We are going to make this interface now and deploy it to our phone. We will make this interface to control these parameters in our game:

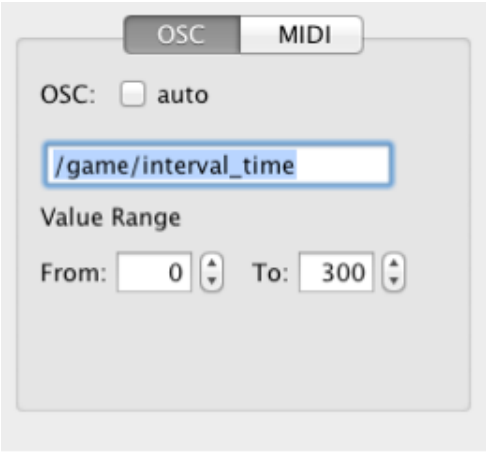
```
//these are the values we will be tweaking during testing
float max_enemy_amplitude;
int interval_time;
float max_enemy_shoot_interval;
bool triggerBonus;
```

To build the app, let's start by adding our first knob. Right click in the black empty space to the right. Choose to make a `rotaryH`. Next make two `labelH` objects. The first one will be the name of our knob. The second one will be for displaying the value of the current variable in our game. Place one label above the knob and one below. It should look like the below image:

Now look to the left side of the app. At this point, it's time to set all of the values this knob will be sending and what the labels will display. Let's start with `label1`. We

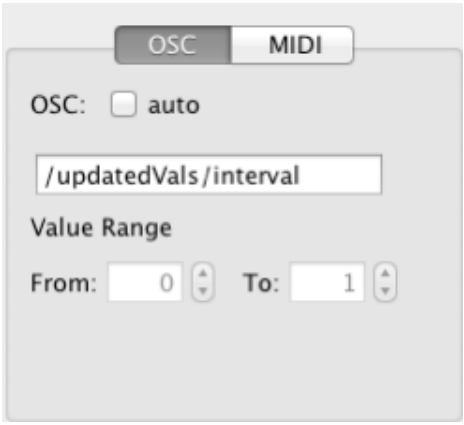
will name our knob on screen to make things easier to read. The first value in our game we want to control, level controller interval time, should be what this label reads onscreen. Changing the name field in the app interface will do little. However, note under the name field you can change the color of the label. For this example, use yellow. Next, jump down to the next to last field on screen called `Text`. You will want to set this to level controller interval time.

Moving on, select the knob. This one will require more set up because it will actually be sending values to our game. Color it yellow first. In the dark grey OSC box set all of the values we need to receive for the game. If auto is checked, uncheck it. Now customize the text in that box to `/game/interval_time`. In the `From` fields set the parameters to a range of values to try out in the game during a playtest. We will use from 0 to 300. These two elements, the tag and the parameters will get packed up into a message and sent over our OSC network to our game when values change.



The last thing to set up will be the bottom label to display what our interval variable is currently set to in our running game. Select it. We will change the settings and the address tag to reflect that it is not game data being sent to our game but rather data

being sent out of our game. Select the label on screen to pull up the parameters for it on the right. In the darkened OSC box change the parameters to those below:



This is the pattern we are going to use for all of our knobs and labels. Essentially, the pattern is

- Create 3 interface elements for each parameter
  - A label for the name of the parameter you will be controlling
  - An interface element like a knob to change it with
  - An output label to display the current in game variable setting

Do this now for the other two knobs. The settings are below for each one.

Label / Knob Set 2

- Label H
  - Color: orange
  - Text: Max Enemy Shoot Interval
- Rotary H
  - Color: orange
  - OSC: /game/max_enemy_shoot_interval
  - Value Range: From: 0 To: 1
- Label H
  - Color: Orange
  - OSC: /updatedVals/max_enemy_shoot_interval

Label / Knob Set 3

- Label H
  - Color: Green

  - Text: max enemy amplitude

- Rotary H
  - Color: Green
  - OSC: /game/max_enemy_amplitude
  - Value Range: From: 0 To: 1

- Label H
  - Color: Green
  - OSC: /updatedVals/max_enemy_amplitude

Set 4

We are going to add one more but this one will be a Push Button verses a RotaryH. Right click to create it just like the knob. Make that now and 2 labels. Here are the settings:

- Label H
  - Color: Pink
  - Text: Trigger Bonus

- Push Button
  - Color: Pink
  - OSC: /game/triggerBonus
  - From: 0 To: 1

- Label H
  - Color: Pink
  - OSC: /updatedVals/triggerBouns

Save your file to your hard drive desktop and name it `PlaytestInterface`. You are done building your interface for play testing. Now let's deploy it. On your mobile device, launch Touch OSC. It will launch and open a settings screen.

This is when we need the network address of your computer we retrieved earlier. Under Connections touch OSC: and set it to the IPAddress of your computer to link the two. This should be something like `192.165.0.3`

The ports should also get set. Tap each one an set them up to these values:

```
Port (outgoing) 8001
Port (incoming) 8000
```

Next tap on TouchOSC in the upper left corner of the app to go back to the settings.

Now click on `Layout`. Then tap `Add`. It will start to search for your computer. Switch back over to your computer now and into the TouchOSC Editor. Press the green `Sync` arrow.

Switch back to your device. You should see your computer listed under FOUND HOSTS. Select it. It will pop back to the settings screen. Scroll down and find PlaytestInterface in the list of interfaces. Select it and it will take you back to the main menu. Press Done in the upper left corner and your interface will now launch. If you want to get back to the settings screen at any point the white dot in the upper right hand corner will return the user interface to that screen.

Finally, TouchOSC is set up. Let's link it to our game and run our very first playtest. Go back to the programming IDE. Open up `LiveTesting.cpp`. In our default constructor, we will now set up our game to send and receive values over the network. To do this we will need to know which Ip address and port on our device we will send to as well as set up a port on our local computer's network to receive incoming data. Your computer will have only one IP address but it can send and receive data on thousands of ports. While we aren't going too deep into ports there, you can think of the IP address like a boat pier. Lots of boats can be docked at a single pier. This is no different. Your ports are your docks and your IP address is your pier. You can think of the data like the people departing and arriving. You'll need a separate port for each activity in this scenario. If a port isn't used by your operating system, you can send and receive data there. We are going to use `8000` and `8001`. The final thing to establish is the Address Pattern. It will look like a file path and it will allow us to specify the address pattern match our messages to their right values. Add this code:

```
#include "LiveTesting.h"

LiveTesting::LiveTesting(){

    sender.setup("192.168.0.11", 8000);
    //this is the ip address of your ipad/android and the port it
       should be
    //set to receive on

    receiver.setup(8001);
    /*this is the port you're game will receive data on.
    For us this is the important one! Set your mobile device to send
       on this port.*/

    m.setAddress("/game");
```

```
    /*This is OSC's URL like naming convention. You can use a root
       url address like
     structure and then everything under that address will be
        accessible by that message.

     It's very similar to a folder path on your hard drive. You can
        think of the
     game folder as your root directory and all the bits that are
     /game/someOtherName are inside of it.*/
}
```

In the above code we simply set up our network address, incoming and out going ports and created a default address pattern. From here we should be good to go to set up the messages we'd like to send and receive in our code.

Let's move on to the next major function we want to write. We need to run an update function in this class to update every frame so we can make sure that if we move a slider on our ipad that change becomes reflected within the game. Also, we might want to send that value back out once we receive it so we can get some visual feedback on our tablet to let us know what our current settings are.

Each time we make a change on our device, it will send over the updates to our code via Touch OSC. We want to make sure we get all of the incoming messages that are being sent so we will create a simple while loop. We will loop through the whole list of messages that came into our game that frame and match it to the corresponding variable in our game via if statements.

```
  while (receiver.hasWaitingMessages()) {
        //get the next message
        ofxOscMessage m;
        receiver.getNextMessage(&m);
```

Every incoming message will come with its own unique address tag and new arguments. You can get access to a message's address via the getAddress function. For example,`if(m.getAddress()== "/game/max_enemy_amplitude")`, will test to see if the message address is /game/max_enemy_amplitude. If it is, set the variable equal to that value in your game's codebase and they are linked together. Every swipe of the knob will translate to direct changes in your game. We do this for every single value we want to set.

```
    if(m.getAddress() == "/game/max_enemy_amplitude")
        {
            max_enemy_amplitude = m.getArgAsFloat(0);

            //these values send back to OSC to display the
            //current settings for visual feedback
            sendBack.addFloatArg(max_enemy_amplitude);
            sendBack.setAddress("/updatedVals/max_enemy_amplitude");
```

```
            sender.sendMessage(sendBack);

            cout << max_enemy_amplitude << endl;
        }
```

At the same time, we are also going to send those exact same values back out to our device so we can see the numbers that the settings in our game are currently at. This is handy for two reasons. One, you get visual feedback of the current variables values' on your device. Two, if you happen to land on settings that feel right in your game, you can take a screen cap on your device. After stopping the game, go back and change the variables to match in your code and the next time you run your program, it will start with those parameters.

To pack up all of the values in our current running game and send them back to the device every frame we will create a variable of type `ofxOscMessage` called `sendBack`. When we have a string match for the address in the `ofxOscMessage m`, we just copy the arguments over to `sendBack` via the right function (in this case usually `addFloatArg`) and set the address pattern using the `setAddress` function. Finally, we use the built in `sendMessage` function to send the message out over OSC.

Here's the complete code to add to your LiveTesting.cpp file

```
void LiveTesting::update()
{
    //our simple while loop to make sure we get all of our messages
    while (receiver.hasWaitingMessages()) {

        //get the message, which will hold all of our arguments
           inside of it.
        //It's a collection of data!

        ofxOscMessage m;
        //pass a reference to that message to the receiver
        //we set up above using the getNextMessage function in the
           OSC add on.

        receiver.getNextMessage(&m);

        //this will be the message we send back from our game
        //to our device letting it know what value we received
        //from it and displaying that back to us so we know what our
        //current game setting are at

        ofxOscMessage sendBack;

        //remember or address tags are unique.
        //we set up the /game tag as our root address and each /
           denotes a sub tag
```

```
        //if theses strings are a match, we know the message that
            came in is our
        //amplitude

        if(m.getAddress() == "/game/max_enemy_amplitude")
        {

            //this is critical.
            //Each type must match if you want to be able to run
                your code.
            //We know the first argument in our array of messages
            //will be a float if the above if statement evaluates to
                true

            max_enemy_amplitude = m.getArgAsFloat(0);

            //now we are going to pack up a collection of data to
                send back to
            //our device. sendBack is also a collection of data we
            //add arguments to.
            //Add the value we set our amplitude to the message and
                move on.

            sendBack.addFloatArg(max_enemy_amplitude);
            sendBack.setAddress("/updatedVals/max_enemy_amplitude");
            sender.sendMessage(sendBack);

            cout << max_enemy_amplitude << endl;
        }

        else if (m.getAddress() == "/game/interval_time")
        {
            //this is exactly the same as above.
            //We just simply are testing to see if the address
            //tag is this value and if so doing the exact
            //process of setting our ingame value to match the value
                of the
            //incoming argument and sending back our interval_time
                to our device.

            interval_time = m.getArgAsInt32(0);

            //send visual feedback
            sendBack.addIntArg(interval_time);
            sendBack.setAddress("/updatedVals/interval");
            sender.sendMessage(sendBack);
        }
        else if (m.getAddress() == "/game/max_enemy_shoot_interval")
        {
```

```
            //again the same process of testing the address tag
            max_enemy_shoot_interval = m.getArgAsFloat(0);

            //send visual feedback
            sendBack.addFloatArg(max_enemy_shoot_interval);
            sendBack.setAddress("/updatedVals/max_enemy_shoot_
                interval");
            sender.sendMessage(sendBack);

        }
        else if (m.getAddress() == "/game/triggerBonus")
        {
            //and finally we rap it up this is last test.
            triggerBonus = m.getArgAsInt32(0);
            cout << triggerBonus << endl;
            //send visual feedback
            sendBack.addIntArg(triggerBonus);
            sendBack.setAddress("/updatedVals/triggerBouns");
            sender.sendMessage(sendBack);


        }
    }
```

You have reached the end of the tutorial. Now do a real testing session. Run the game and have a friend play it while you change the knobs. Once you have settings you like, quit the game and add those values into the code to make them permanent updates.

For a bonus challenge, find a few settings you like, and create a difficulty ramp for game using those values of time.

9.3.9 Resouces

We've reached the end of the chapter but not the end of the journey. A few great resources for independent games scene are listed here:
Come Out And Play[3]
Kill Screen[4]
Indiecade[5]
Babycastles[6]
Polygon[7]

[3] http://www.comeoutandplay.org/
[4] http://killscreendaily.com/
[5] http://www.indiecade.com
[6] http://www.babycastles.com
[7] http://www.polygon.com/

IDGA[8]
Game Dev Net[9]
Game Dev Stack Exchange[10]
Gamasutra[11]
Digra[12]
Different Games[13]

9.3.10 About us

This chapter was written by two members of the Code Liberation Foundation[14], Phoenix Perry[15] and Jane Friedhoff[16]. This organization teaches women to program games for free. Featuring game art by Loren Bednar. We build community, create a safe spaces for women who want to learn to program in a non-male dominated setting and generally rock.

[8] http://www.igda.org/
[9] http://www.gamedev.net/page/index.html
[10] http://gamedev.stackexchange.com/
[11] http://gamasutra.com/
[12] http://www.digra.org/
[13] http://www.differentgames.org/
[14] http://www.codeliberation.org
[15] http://www.phoenixperry.com
[16] janefriedhoff.com

10 Image Processing and Computer Vision

By Golan Levin[1]

Edited by Brannon Dorsey[2]

10.1 Preliminaries to Image Processing

10.1.1 Digital image acquisition and data structures

This chapter introduces techniques for manipulating (and extracting certain kinds of information from) *raster images*. Such images are sometimes also known as *bitmap images* or *pixmap images*, though we'll just use the generic term **image** to refer to any array (or *buffer*) of numbers that represenRat the color values of a rectangular grid of *pixels* ("picture elements"). In openFrameworks, such buffers come in a variety of flavors, and are used within (and managed by) a wide variety of convenient container objects, as we shall see.

10.1.1.1 Loading and Displaying an Image

Image processing begins with, well, *an image*. Happily, loading and displaying an image is very straightforward in OF. Let's start with this tiny, low-resolution (12x16 pixel) grayscale portrait of Abraham Lincoln:

Figure 10.1: Small Lincoln image

Below is a simple application for loading and displaying an image, very similar to the *imageLoaderExample* in the OF examples collection. The header file for our program, *ofApp.h*, declares an instance of an `ofImage` object, *myImage*:

```
// Example 1: Load and display an image.
// This is ofApp.h
```

[1] http://www.flong.com/

[2] http://brannondorsey.com

```
#pragma once
#include "ofMain.h"

class ofApp : public ofBaseApp{
    public:
        void setup();
        void draw();
        ofImage myImage;
};
```

Below is our complete *ofApp.cpp* file. The Lincoln image is *loaded* from our hard drive (once) in the `setup()` function; then we *display* it (many times per second) in our `draw()` function. As you can see from the filepath provided to the `loadImage()` function, the program assumes that the image *lincoln.png* can be found in a directory called "data" alongside your executable:

```
// Example 1: Load and display an image.
// This is ofApp.cpp

#include "ofApp.h"

void ofApp::setup(){
    myImage.loadImage("lincoln.png");
    myImage.setImageType(OF_IMAGE_GRAYSCALE);
}

void ofApp::draw(){
    ofBackground(255);
    ofSetColor(255);

    int imgW = myImage.width;
    int imgH = myImage.height;
    myImage.draw(10, 10, imgW * 10, imgH * 10);
}
```

Compiling and running the above program displays the following canvas, in which this (very tiny!) image is scaled up by a factor of 10, and rendered so that its upper left corner is positioned at pixel location (10,10). The positioning and scaling of the image are performed by the `myImage.draw()` command. Note that the image appears "blurry" because, by default, openFrameworks uses linear interpolation[3] when displaying upscaled images.

If you're new to working with images in OF, it's worth pointing out that you should try to avoid loading images in the `draw()` or `update()` functions, if possible. Why? Well, reading data from disk is one of the slowest things you can ask a computer to do. In many circumstances, you can simply load all the images you'll need just once, when

[3] http://en.wikipedia.org/wiki/Linear_interpolation

Figure 10.2: Pixel data diagram

your program is first initialized, in `setup()`. By contrast, if you're repeatedly loading an image in your `draw()` loop — the same image, again and again, sixty times per second — you're hurting the performance of your app, and potentially even risking damage to your hard disk.

10.1.1.2 Where (Else) Images Come From

In openFrameworks, raster images can come from a wide variety of sources, including (but not limited to):

- an image file (stored in a commonly-used format like .JPEG, .PNG, .TIFF, or .GIF), loaded and decompressed from your hard drive into an `ofImage`;
- a real-time image stream from a webcam or other video camera (using an `ofVideoGrabber`);
- a sequence of frames loaded from a digital video file (using an `ofVideoPlayer`);
- a buffer of pixels grabbed from whatever you've already displayed on your screen, captured with `ofImage::grabScreen()`;
- a synthetic computer graphic rendering, perhaps obtained from an `ofFBO` or stored in an `ofPixels` or `ofTexture` object;
- a real-time video from a more specialized variety of camera, such as a 1394b Firewire camera (via `ofxLibdc`), a networked Ethernet camera (via `ofxIpCamera`), a Canon DSLR (using `ofxCanonEOS`), or with the help of a variety of other community-contributed addons like `ofxQTKitVideoGrabber`, `ofxRPiCameraVideoGrabber`, etc.;
- perhaps more exotically, a *depth image*, in which pixel values represent *distances* instead of colors. Depth images can be captured from real-world scenes with special cameras (such as a Microsoft Kinect via the `ofxKinect` addon), or extracted from synthetic CGI scenes using (for example) `ofFBO::getDepthTexture()`.


An example of a depth image (left) and a corresponding RGB color image (right), captured simultaneously with a Microsoft Kinect. In the depth image, the brightness of a pixel represents its proximity to the camera.

Incidentally, OF makes it easy to **load images directly from the Internet**, by using a URL as the filename argument, as in `myImage.loadImage("http://en.wikipedia.org/wiki/File:Exa`
Keep in mind that doing this will load the remotely-stored image *synchronously*, meaning your program will "block" (or freeze) while it waits for all of the data to download from the web. For an improved user experience, you can also load Internet images *asynchronously* (in a background thread), using the response provided by `ofLoadURLAsync()`; a sample implementation of this can be found in the openFrameworks *imageLoaderWebExample* graphics example. Now that you can load images stored on the Internet, you can fetch images *computationally* using fun APIs (like those of Temboo[4], Instagram[5] or Flickr[6]), or from dynamic online sources such as live traffic cameras.

10.1.1.3 Acquiring and Displaying a Webcam Image

The procedure for **acquiring a video stream** from a live webcam or digital movie file is no more difficult than loading an `ofImage`. The main conceptual difference is that the image data contained within an `ofVideoGrabber` or `ofVideoPlayer` happens to be continually refreshed, usually about 30 times per second (or at the framerate of the footage).

The following program (which you can find elaborated in the OF *videoGrabberExample*) shows the basic procedure. In this example below, for some added fun, we also retrieve the buffer of data that contains the `ofVideoGrabber`'s pixels, then "invert" this data (to produce a "photographic negative") and display it with an `ofTexture`.

The header file for our app declares an `ofVideoGrabber`, which we will use to acquire video data from our computer's default webcam. We also declare a buffer of unsigned chars to store the inverted video frame, and the `ofTexture` which we'll use to display it:

[4] https://temboo.com/library/
[5] http://instagram.com/developer/
[6] https://www.flickr.com/services/api/

```
// Example 2. An application to capture, display,
// and invert live video from a webcam.
// This is ofApp.h

#pragma once
#include "ofMain.h"

class ofApp : public ofBaseApp{
    public:

        void setup();
        void update();
        void draw();

        ofVideoGrabber      myVideoGrabber;
        ofTexture           myTexture;

        unsigned char*      invertedVideoData;
        int                 camWidth;
        int                 camHeight;
};
```

Does the `unsigned char*` declaration look unfamiliar? It's important to recognize and understand, because this is a nearly universal way of storing and exchanging image data. The `unsigned` keyword means that the values which describe the colors in our image are exclusively positive numbers. The `char` means that each color component of each pixel is stored in a single 8-bit number—a byte, with values ranging from 0 to 255—which for many years was also the data type in which *char*acters were stored. And the `*` means that the data named by this variable is not just a single unsigned char, but rather, an *array* of unsigned chars (or more accurately, a *pointer* to a buffer of unsigned chars). For more information about such datatypes, see Chapter 9, *Memory in C++*.

Below is the complete code of our webcam-grabbing .cpp file. As you might expect, the `ofVideoGrabber` object provides many more options and settings, not shown here. These allow you to do things like listing and selecting from available camera devices; setting your capture dimensions and framerate; and (depending on your hardware and drivers) adjusting parameters like camera exposure and contrast.

Note that the example segregates our heavy computation into `update()`, and rendering our graphics into `draw()`. This is a recommended pattern for structuring your code.

```
// Example 2. An application to capture, invert,
// and display live video from a webcam.
// This is ofApp.cpp

#include "ofApp.h"
```

```
void ofApp::setup(){

    // Set capture dimensions of 320x240, a common video size.
    camWidth  = 320;
    camHeight = 240;

    // Open an ofVideoGrabber for the default camera
    myVideoGrabber.initGrabber (camWidth,camHeight);

    // Create resources to store and display another copy of the data
    invertedVideoData = new unsigned char [camWidth*camHeight*3];
    myTexture.allocate (camWidth,camHeight, GL_RGB);
}

void ofApp::update(){

    // Ask the grabber to refresh its data.
    myVideoGrabber.update();

    // If the grabber indeed has fresh data,
    if (myVideoGrabber.isFrameNew()){

        // Obtain a pointer to the grabber's image data.
        unsigned char* pixelData = myVideoGrabber.getPixels();

        // For every byte of the RGB image data,
        int nTotalBytes = camWidth*camHeight*3;
        for (int i=0; i<nTotalBytes; i++){

            // pixelData[i] is the i'th byte of the image;
            // subtract it from 255, to make a "photo negative"
            invertedVideoData[i] = 255 - pixelData[i];
        }

        // Now stash the inverted data in an ofTexture
        myTexture.loadData (invertedVideoData, camWidth,camHeight,
           GL_RGB);
    }
}

void ofApp::draw(){
    ofBackground(100,100,100);
    ofSetColor(255,255,255);

    // Draw the grabber, and next to it, the "negative" ofTexture.
    myVideoGrabber.draw(10,10);
    myTexture.draw(340, 10);
}
```

This application continually displays the live camera feed, and also presents a live, "filtered" (photo negative) version. Here's the result, using my laptop's webcam:

Figure 10.3: Video grabber screenshot

Acquiring frames from a Quicktime movie or other digital video file stored on disk is an almost identical procedure. See the OF *videoPlayerExample* implementation or `ofVideoGrabber` documentation[7] for details.

A common pattern among developers of interactive computer vision systems is to enable easy switching between a pre-stored "sample" video of your scene, and video from a live camera grabber. That way, you can test and refine your processing algorithms in the comfort of your hotel room, and then switch to "real" camera input when you're back at the installation site. A hacky if effective example of this pattern can be found in the openFrameworks *opencvExample*, in the addons example directory, where the switch is built using a `#define` preprocessor directive[8]:

```
    //...
    #ifdef _USE_LIVE_VIDEO
        myVideoGrabber.initGrabber(320,240);
    #else
        myVideoPlayer.loadMovie("pedestrians.mov");
        myVideoPlayer.play();
    #endif
    //...
```

Uncommenting the `//#define _USE_LIVE_VIDEO` line in the .h file of the *opencvExample* forces the compiler to attempt to use a webcam instead of the pre-stored sample video.

[7] http://openframeworks.cc/documentation/video/ofVideoGrabber.html
[8] http://www.cplusplus.com/doc/tutorial/preprocessor/

10.1.1.4 Pixels in Memory

To begin our study of image processing and computer vision, we'll need to do more than just load and display images; we'll need to *access, manipulate and analyze the numeric data represented by their pixels*. It's therefore worth reviewing how pixels are stored in computer memory. Below is a simple illustration of the grayscale image buffer which stores our image of Abraham Lincoln. Each pixel's brightness is represented by a single 8-bit number, whose range is from 0 (black) to 255 (white):

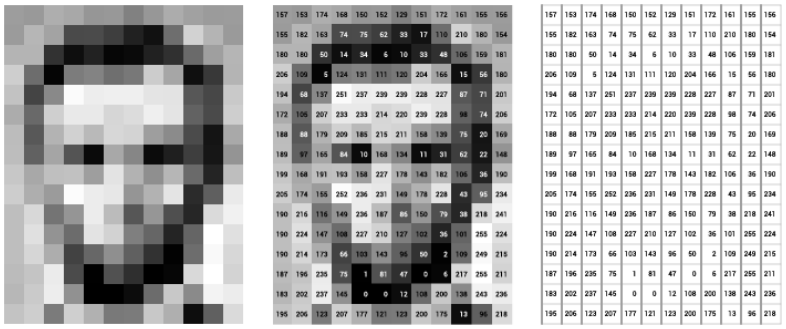


Figure 10.4: Pixel data diagram

In point of fact, pixel values are almost universally stored, at the hardware level, in a *one-dimensional array*. For example, the data from the image above is stored in a manner similar to this long list of unsigned chars:

```
{157, 153, 174, 168, 150, 152, 129, 151, 172, 161, 155, 156,
 155, 182, 163,  74,  75,  62,  33,  17, 110, 210, 180, 154,
 180, 180,  50,  14,  34,   6,  10,  33,  48, 106, 159, 181,
 206, 109,   5, 124, 131, 111, 120, 204, 166,  15,  56, 180,
 194,  68, 137, 251, 237, 239, 239, 228, 227,  87,  71, 201,
 172, 105, 207, 233, 233, 214, 220, 239, 228,  98,  74, 206,
 188,  88, 179, 209, 185, 215, 211, 158, 139,  75,  20, 169,
 189,  97, 165,  84,  10, 168, 134,  11,  31,  62,  22, 148,
 199, 168, 191, 193, 158, 227, 178, 143, 182, 106,  36, 190,
 205, 174, 155, 252, 236, 231, 149, 178, 228,  43,  95, 234,
 190, 216, 116, 149, 236, 187,  86, 150,  79,  38, 218, 241,
 190, 224, 147, 108, 227, 210, 127, 102,  36, 101, 255, 224,
 190, 214, 173,  66, 103, 143,  96,  50,   2, 109, 249, 215,
 187, 196, 235,  75,   1,  81,  47,   0,   6, 217, 255, 211,
 183, 202, 237, 145,   0,   0,  12, 108, 200, 138, 243, 236,
 195, 206, 123, 207, 177, 121, 123, 200, 175,  13,  96, 218};
```

This way of storing image data may run counter to your expectations, since the data certainly *appears* to be two-dimensional when it is displayed. Yet, this is the case, since computer memory consists simply of an ever-increasing linear list of address spaces.

Note how this data includes no details about the image's width and height. Should this list of values be interpreted as a grayscale image which is 12 pixels wide and 16 pixels tall, or 8x24, or 3x64? Could it be interpreted as a color image? Such 'meta-data' is specified elsewhere — generally in a container object like an `ofImage`.

10.1.1.5 Grayscale Pixels and Array Indices

It's important to understand how pixel data is stored in computer memory. Each pixel has an *address*, indicated by a number (whose counting begins with zero):

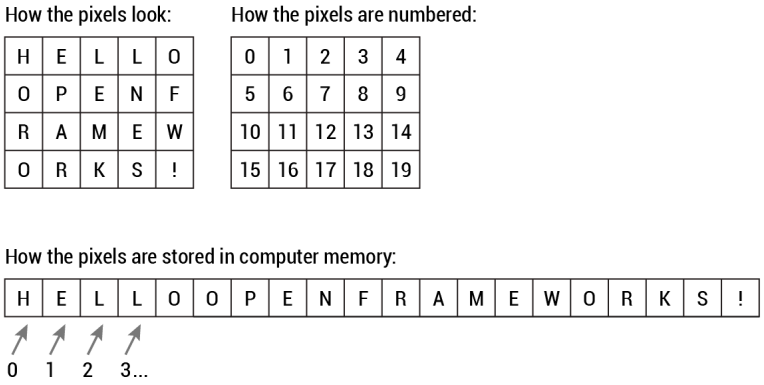


Figure 10.5: Based on Shiffman's image in the Processing tutorial

Observe how the (one-dimensional) list of values have been distributed to successive (two-dimensional) pixel locations in the image — wrapping over the right edge just like English text.

It frequently happens that you'll need to determine the array-index of a given pixel *(x,y)* in an image that is stored in an `unsigned char*` buffer. This little task comes up often enough that it's worth committing the following pattern to memory:

```
// Given:
// unsigned char *buffer, an array storing a one-channel image
// int x, the horizontal coordinate (column) of your query pixel
// int y, the vertical coordinate (row) of your query pixel
// int imgW, the width of your image

int arrayIndex = y*imgW + x;

// Now you can GET values at location (x,y), e.g.:
unsigned char pixelValueAtXY = buffer[arrayIndex];

// And you can also SET values at that location, e.g.:
buffer[arrayIndex] = pixelValueAtXY;
```

Reciprocally, you can also fetch the x and y locations of a pixel corresponding to a given array index:

```
// Given:
// A one-channel (e.g. grayscale) image
// int arrayIndex, an index in that image's array of pixels
// int imgW, the width of the image

int y = arrayIndex / imgW; // NOTE, this is integer division!
int x = arrayIndex % imgW;
```

Most of the time, you'll be working with image data that is stored in a higher-level container object, such as an `ofImage`. There are *two* ways to get the values of pixel data stored in such a container. In one method, we can ask the image for its array of unsigned char pixel data, using `.getPixels()`, and then fetch the value we want from this array. Many image containers, such as `ofVideoGrabber`, also support a `.getPixels()` function.

```
int arrayIndex = y*imgW + x;
unsigned char* myImagePixelBuffer = myImage.getPixels();
unsigned char pixelValueAtXY = myImagePixelBuffer[arrayIndex];
```

The second method is a high-level convenience operator that returns the color stored at a pixel location:

```
ofColor colorAtXY = myImage.getColor(x, y);
float brightnessOfColorAtXY = colorAtXY.getBrightness();
```

10.1.1.6 Finding the Brightest Pixel in an Image

Using what we know now, we can write a simple computer-vision program that locates the brightest pixel in an image. This elementary concept was used to great artistic effect by the artist collective, Graffiti Research Lab (GRL), in the openFrameworks application they built for their 2007 project *L.A.S.E.R Tag*[9]. The concept of *L.A.S.E.R Tag* was to allow people to draw projected graffiti on a large building facade, using a laser pointer. The bright spot from the laser pointer was tracked by code similar to that shown below, and used as the basis for creating projected graphics.

The .h file for our app loads an ofImage (`laserTagImage`) of someone pointing a laser at the building. (In the real application, a live camera was used.)

```
// Example 3. Finding the Brightest Pixel in an Image
// This is ofApp.h
```

[9] http://www.graffitiresearchlab.com/blog/projects/laser-tag/

Figure 10.6: Laser Tag by GRL

```
#pragma once
#include "ofMain.h"

class ofApp : public ofBaseApp{
    public:
        void setup();
        void draw();
        ofImage laserTagImage;
};
```

Here's the .cpp file:

```
// Example 3. Finding the Brightest Pixel in an Image
// This is ofApp.cpp

#include "ofApp.h"

//--------------------
void ofApp::setup(){
    laserTagImage.loadImage("images/laser_tag.jpg");
}

//--------------------
void ofApp::draw(){
    ofBackground(255);
```

```
    int w = laserTagImage.getWidth();
    int h = laserTagImage.getHeight();

    float maxBrightness = 0;  // these are used in the search
    int   maxBrightnessX = 0; // for the brightest location
    int   maxBrightnessY = 0;

    // Search through every pixel. If it is brighter than any
    // we've seen before, store its brightness and coordinates.
    for (int y=0; y<h; y++) {
        for(int x=0; x<w; x++) {
            ofColor colorAtXY = laserTagImage.getColor(x, y);
            float brightnessOfColorAtXY = colorAtXY.getBrightness();
            if (brightnessOfColorAtXY > maxBrightness){
                maxBrightness = brightnessOfColorAtXY;
                maxBrightnessX = x;
                maxBrightnessY = y;
            }
        }
    }

    // Draw the image.
    ofSetColor (255);
    laserTagImage.draw(0,0);

    // Draw a circle at the brightest location.
    ofNoFill();
    ofEllipse (maxBrightnessX, maxBrightnessY, 40,40);
}
```

Our application locates the bright spot of the laser (which, luckily for us, is the brightest part of the scene) and draws a circle around it. Of course, now that we know where the brightest (or darkest) spot is, we can can develop many interesting applications, such as sun trackers, turtle trackers...

Being able to locate the brightest pixel in an image has other uses, too. For example, in a depth image (such as produced by a Kinect sensor), the brightest pixel corresponds to the *foremost point*—or the nearest object to the camera. This can be extremely useful if you're making an interactive installation that tracks a user's hand.

The brightest pixel in a depth image corresponds to the nearest object to the camera.

Unsurprisingly, tracking *more than one* bright point requires more sophisticated forms of processing. If you're able to design and control the tracking environment, one simple yet effective way to track up to three objects is to search for the reddest, greenest and bluest pixels in the scene. Zachary Lieberman used a technique similar to this in his *IQ Font*[10] collaboration with typographers Pierre & Damien et al., in which letterforms

[10] https://vimeo.com/5233789

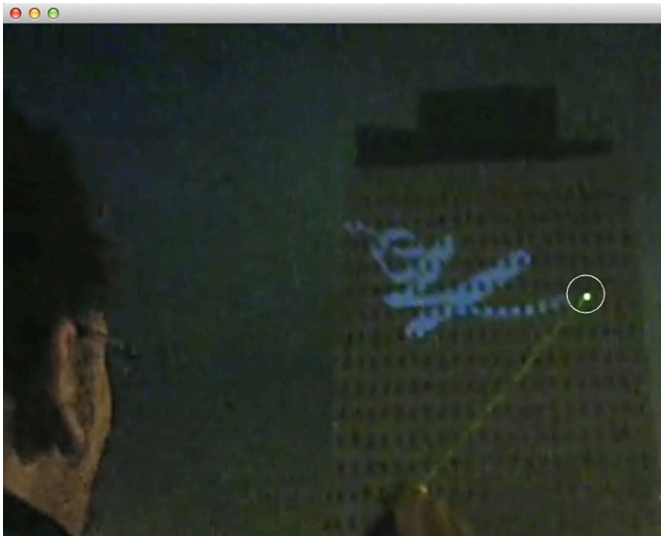


Figure 10.7: Laser Tag by GRL

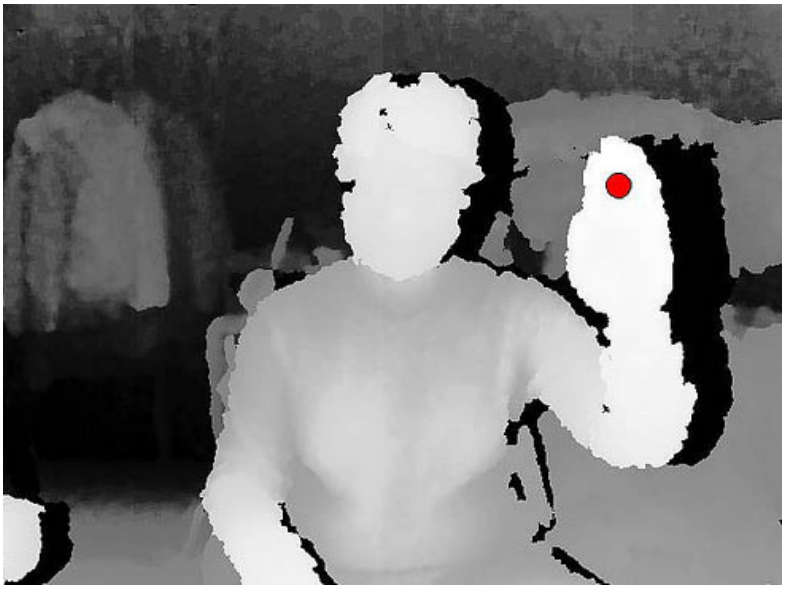


Figure 10.8: Not mine

were created by tracking the movements of a specially-marked sports car.

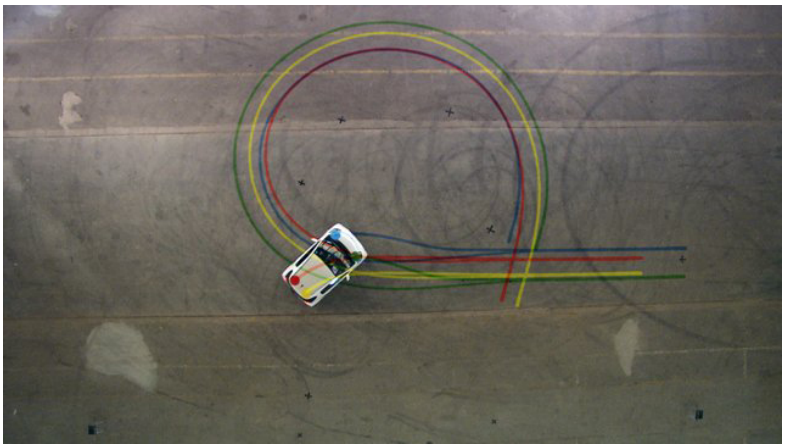


Figure 10.9: Not mine

10.1.1.7 Three-Channel (RGB) Images.

Our Lincoln portrait image shows an 8-bit, 1-channel image. Each pixel uses a single round number (technically, an unsigned char) to represent a single luminance value. But other data types and formats are possible.

For example, it is common for color images to be represented by 8-bit, *3-channel* images. In this case, each pixel brings together 3 bytes' worth of information: one byte each for red, green and blue intensities. In computer memory, it is common for these values to be interleaved R-G-B. As you can see, color images necessarily contain three times as much data.

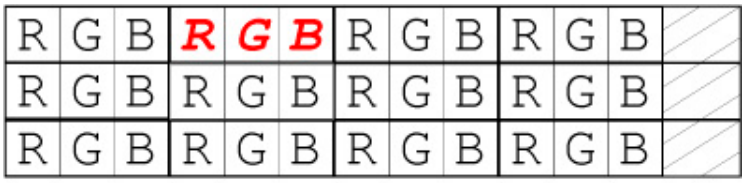


Figure 10.10: Not mine

Take a very close look at your LCD screen, and you'll see how this way of storing the data is directly motivated by the layout of your display's phosphors:

Because the color data are interleaved, accessing pixel values in buffers containing RGB data is slightly more complex. Here's how you can retrieve the values representing the individual red, green and blue components of pixel at a given (x,y) location:

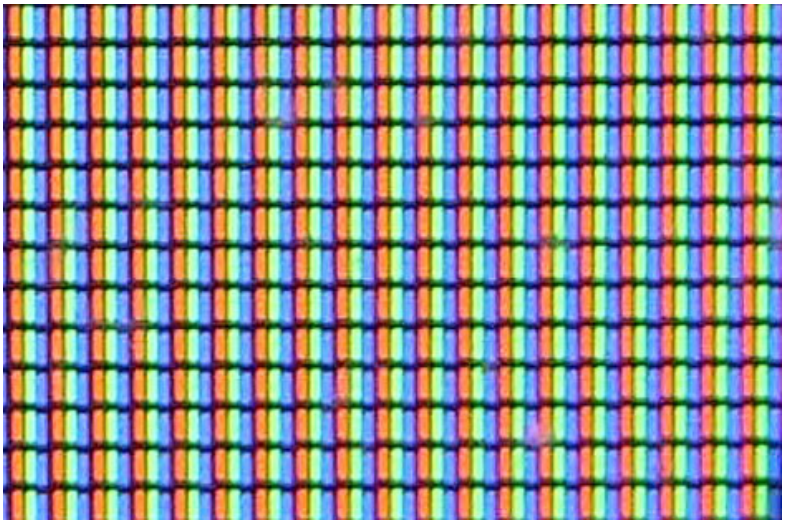


Figure 10.11: Not mine

```
// Given:
// unsigned char *buffer, an array storing an RGB image
// (assuming interleaved data in RGB order!)
// int x, the horizontal coordinate (column) of your query pixel
// int y, the vertical coordinate (row) of your query pixel
// int imgWidth, the width of the image

int rArrayIndex = (y*imgWidth*3) + (x*3);
int gArrayIndex = (y*imgWidth*3) + (x*3) + 1;
int bArrayIndex = (y*imgWidth*3) + (x*3) + 2;

// Now you can get and set values at location (x,y), e.g.:
unsigned char redValueAtXY   = buffer[rArrayIndex];
unsigned char greenValueAtXY = buffer[gArrayIndex];
unsigned char blueValueAtXY  = buffer[bArrayIndex];
```

10.1.1.8 Other Kinds of Image Formats and Containers

8-bit 1-channel and 8-bit 3-channel images are the most common image formats you'll find. In the wide world of image processing algorithms, however, you'll eventually encounter an exotic variety of other types of images, including: - 8-bit *palettized* images, in which each pixel stores an index into an array of (up to) 256 possible colors; - 16-bit (unsigned short) images, in which each channel uses *two* bytes to store each of the color values of each pixel, with a number that ranges from 0-65535; - 32-bit (float) images, in which each color channel's data is represented by floating point numbers.

For a practical example, consider Microsoft's popular Kinect sensor, which produces a depth image whose values range from 0 to 1090. Clearly, that's wider than the range of 8-bit data (from 0 to 255) one might typically encounter; in fact, it's approximately 11 bits of resolution. To accommodate this, the `ofxKinect` addon employs a 16-bit image to store this information without losing precision. Likewise, the precision of 32-bit floats is almost mandatory for computing high-quality video composites.

You'll also find: - 2-channel images (commonly used for luminance plus transparency); - 3-channel images (generally for RGB data, but occasionally used to store images in other color spaces, such as HSB or YUV); - 4-channel images (commonly for RGBA images, but occasionally for CMYK); - *Bayer images*, in which the RGB color channels are not interleaved R-G-B-R-G-B-R-G-B... but instead appear in a unique checkerboard pattern.

It gets even more exotic. "Hyperspectral" imagery from the Landsat 8 satellite[11], for example, has 11 channels, including bands for ultraviolet, near infrared, and thermal (deep) infrared!

In openFrameworks, images can be stored in a variety of different *container classes*, which allow their data to be used (captured, displayed, manipulated, and stored) in different ways and contexts. Some of the more common containers you may encounter are:

- **unsigned char*** An array of unsigned chars, this is the raw format used for storing buffers of pixel data. It's not very smart, but it's often useful for exchanging data with different libraries. Many image processing textbooks will assume your data is stored this way.
- **ofPixels** This is a container for pixel data which lives inside each ofImage, as well as other classes like ofVideoGrabber. It's a wrapper around a buffer that includes additional information like width and height.
- **ofImage** The `ofImage` is the most common object for loading, saving and displaying static images in openFrameworks. Loading a file into the ofImage allocates an (internal) ofPixels object to store the image data. ofImage objects are not merely containers, but also support methods for displaying their pixel data.
- **ofxCvImage** This is a container for image data used by the ofxOpenCV addon for openFrameworks, which supports certain functionality from the popular OpenCV library for filtering, thresholding, and other image manipulations.
- **cv::Mat** This is the data structure used by OpenCV to store image information. It's not used in openFrameworks, but if you work a lot with OpenCV, you'll often find yourself placing and extracting data from this format.

To the greatest extent possible, the designers of openFrameworks (and OF addons for image processing, like ofxOpenCV and ofxCv) have provided simple operators to help make it easy to exchange data between these containers.

[11] https://www.mapbox.com/blog/putting-landsat-8-bands-to-work/

10.1.1.9 RGB, grayscale, and other color space conversions

Many computer vision algorithms (though not all!) are commonly performed on grayscale or monochome images. Converting color images to grayscale can significantly improve the speed of many image processing routines, because it reduces both the number of calculations as well as the amount of memory required to process the data. Except where stated otherwise, *all of the examples in this chapter assume that you're working with monochrome images*. Here's some simple code to convert a color image (e.g. captured from a webcam) into a grayscale version:

[Code to convert RGB to grayscale using openFrameworks]

[Code to convert RGB to grayscale using ofxCV]

[Code to convert RGB to grayscale using OpenCV]

Of course, there are times when

10.1.2 Image arithmetic: mathematical operations on images

A core part of the workflow of computer vision is *image arithmetic*. These are the basic mathematical operations we all know – addition, subtraction, multiplication, and division – but interpreted in the image domain. Here are two very simple examples:

[Code to add a constant value to an image]

[Code to multiply an image by a constant]

TIP: Watch out for blowing out the range of your data types.

When assume that these operations are performed *pixelwise* – meaning, for every pixel in an image. When

In the examples presented here, for the sake of simplicity, we'll assume that the images upon which we'll perform these operations are all the same size – for example, 640x480 pixels, a typical capture size for many webcams. We'll also assume that these images are monochrome or grayscale.

- adding two images together
- subtracting two images
- multiplying an image by a constant
- mentioning ROI
- Example: creating an average of several images (e.g. Jason Salavon)
- Example: creating a running average
- Example: creating a circular alpha-mask from a computed Blinn spot

10.1.3 Filtering and Noise Removal Convolution Filtering

- Blurring an image
- Edge detection
- Median filtering
- Advanced sidebar: dealing with boundary conditions

========== 3. Scenario I. Basic Blobs (e.g. Manual Input Sessions)

3.1. The Why - Some examples of projects that use blob-tracking - and some scenarios that call for it.

10.1.4 3.2. Detecting and Locating Presence and Motion

10.1.4.1 3.2.1. Detecting presence with Background subtraction

sfdflkj #### 3.2.2. Detecting motion with frame-differencing sfdflkj #### 3.2.3. Binarization, blob detection and contour extraction sfdflkj - Area thresholds for contour extraction (min plausible area, max plausible area, as % of capture size) - Finding negative vs. positive contours

10.1.5 3.3. Image Processing Refinements

10.1.5.1 3.3.1. Using a running average of background

10.1.5.2 3.3.2. Erosion, dilation, median to remove noise after binarization

10.1.5.3 3.3.3. Combining presence and motion in a weighted average

10.1.5.4 3.3.4. Compensating for perspectival distortion and lens distortion

10.1.6 3.4. Thresholding Refinements

- Some techniques for automatic threshold detection
- Dynamic thresholding (per-pixel thresholding)

3.5. The Vector space: Extracting information from Blob Contours - Area, Perimeter, Centroids, Bounding box - Calculcating blob orientation (central axis) - Locating corners in contours, estimating local curvature - 1D Filtering of contours to eliminate noise, i.e local averaging. - Other shape metrics; shape recognition

3.6. Using Kinect depth images - Finding the “fore-point” (foremost point) - Background subtraction with depth images - Hole-filling in depth images - Computing normals from depth gradients

3.7. Suggestions for further experimentation: - Tracking multiple blobs with ofxCv.tracker - Box2D polygons using OpenCV contours, e.g. https://vimeo.com/9951522

Automatic thresholding is

========== 4. Scenario II. Face Tracking.

4.1. Overview Some examples of projects that use face-tracking - *Face Substitution*[12] by Kyle McDonald & Arturo Castro (2011). The classic - *Google Faces*[13] by Onformative (2012). This project, which identifies face-like features in Google Earth satellite imagery, explores what Greg Borenstein has called *machine pareidolia* – the possibility that computer algorithms can “hallucinate” faces in everyday images.

10.1.7 A basic face detector.

In this section we’ll which implements face detection using the classic “Viola-Jones” face detector that comes with OpenCV. - Face detection with classic OpenCV viola-Jones detector - How it works, and considerations when using it. - cvDazzle;

How does the Viola-Jones face-tracker work?

The cvDazzle[14] project by Adam Harvey

Ada writes: “OpenCV is one of the most widely used face detectors. This algorithm performs best for frontal face imagery and excels at computational speed. It’s ideal for real-time face detection and is used widely in mobile phone apps, web apps, robotics, and for scientific research.

OpenCV is based on the the Viola-Jones algorithm. This video shows the process used by the Viola Jones algorithm, a cascading set of features that scans across an image at increasing sizes. By understanding how the algorithm detects a face, the process of designing an “anti-face” becomes more intuitive."

10.1.7.1 SIDEBAR

> *Orientation-dependence in the OpenCV face detector: Bug or Feature?* - Kyle & Aram (“How to Avoid Facial Recognition”

4.3. Advanced face analysis with the Saraghi FaceTracker

[12] https://vimeo.com/29348533
[13] http://www.onformative.com/lab/googlefaces/
[14] http://cvdazzle.com/

10.1.8 4.4. Suggestions for Further Experimentation

Now that you can locate faces in images and video, consider using the following exercises as starting-points for further exploration:

- Make a face-controlled puppet
- Mine an image database for faces
- Make a kinetic sculpture that points toward a visitor's face.

10.1.9 Suggestions for Further Experimentation

I sometimes assign my students the project of copying a well-known work of interactive new-media art. Reimplementing projects such as the ones below can be highly instructive, and test the limits of your attention to detail. As Gerald King writes[15], such copying "provides insights which cannot be learned from any other source." *I recommend you build...*

10.1.9.1 A Slit-Scanner.

Slit-scanning — a type of "time-space imaging" — has been a common trope in interactive video art for more than twenty years. Interactive slit-scanners have been developed by some of the most revered pioneers of new media art (Toshio Iwai, Paul de Marinis, Steina Vasulka) as well as by literally dozens[16] of other highly regarded practitioners. The premise remains an open-ended format for seemingly limitless experimentation, whose possibilities have yet to be exhausted. It is also a good exercise in managing image data, particularly in extracting and copying pixel ROIs. In digital slit-scanning, thin slices are extracted from a sequence of video frames, and concatenated into a new image. The result is an image which succinctly reveals the history of movements in a video or camera stream.

10.1.9.2 Text Rain by Camille Utterback and Romy Achituv (1999).

Text Rain[17] is a now-classic work of interactive art in which virtual letters appear to "fall" on the visitor's "silhouette". Utterback writes: "In the Text Rain installation, participants stand or move in front of a large projection screen. On the screen they see a mirrored video projection of themselves in black and white, combined with a color animation of falling letters. Like rain or snow, the letters appears to land on participants' heads and arms. The letters respond to the participants' motions and can be caught, lifted,

[15] http://www.geraldking.com/Copying.htm
[16] http://www.flong.com/texts/lists/slit_scan/
[17] http://camilleutterback.com/projects/text-rain/

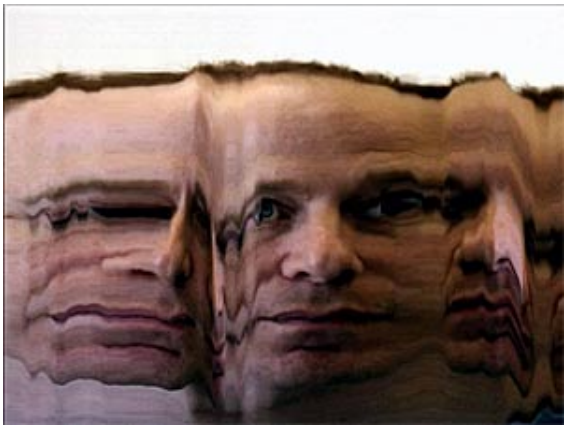


Figure 10.12: Daniel Rozin, Time Scan Mirror (2004)

and then let fall again. The falling text will 'land' on anything darker than a certain threshold, and 'fall' whenever that obstacle is removed."

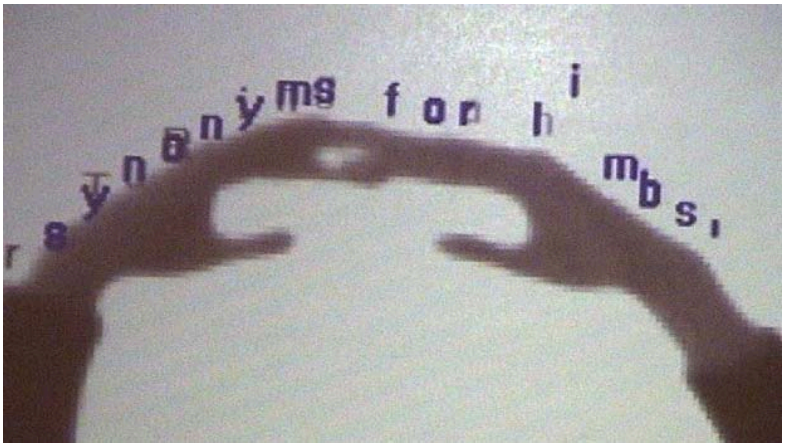


Figure 10.13: Camille Utterback and Romy Achituv, Text Rain (1999)

==========

A Computer-Vision lexicon, and where to find out more information

Computer vision is a huge field and we can't possibly cover all useful examples here. Sometimes people lack the terminology to know what to google for.

– Camera calibration. – Homography transforms and re-projection.

11 hardware

by Caitlin Morris[1] and Pierre Proske[2]

11.1 introduction

This chapter will give you an introduction to working with openFrameworks outside of your computer screen and into the physical world. Why exactly would you want to do this? Well, given that we are physical creatures ourselves, having software control, sense and actuate real-world things can be pretty exciting and create truly visceral experiences. Screen based work can be captivating, but physical installations have the potential to deliver greater impact due to their more tangible nature.

There are a number of ways of taking your openFrameworks app out of the frame of your own personal computer and getting it to interact with the outside world. Largely this involves some kind of communication from openFrameworks to whatever hardware you've decided to hook up to. The different types of computer based communications (or protocols) vary, but the most common is what's known as 'serial' communication, so called because each bit of data sent is transferred one after the other (as opposed to multiple bits being sent in parallel).

The first hardware that we'll look at interfacing with is the excellent **Arduino** prototyping platform. Arduino is, in its own words, an *"open-source electronics prototyping platform based on flexible, easy-to-use hardware and software... intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments.*" It's easy to see why there's a lot of overlap between communities of people interested in using openFrameworks and Arduino! With Arduino, it's quick to get your openFrameworks app hooked up to sensors (like light sensors, proximity sensors, or other methods of environmental input), actuators (like lights, motors, and other outputs), and real-time interaction. You're free to move out of the realm of pixels and into the physical world.

This chapter assumes that you have the Arduino IDE installed, in addition to the environment that you normally use for working with openFrameworks. If not, you can download it from the Arduino website (arduino.cc) or the Arduino github (github.com/arduino).

[1] http://www.caitlinmorris.net/
[2] http://www.digitalstar.net/

Additionally, following along with the examples in this chapter requires you to have a few pieces of basic hardware. You'll need an Arduino (any generation; an Uno, Leonardo, or Duemilanove will look the most like the graphics in this chapter but any USB-connected Arduino will work just fine) and the appropriate USB cable to connect to your computer.

[** Callout - The Leonardo differs from earlier boards in that it has built-in USB communication, eliminating the need for a secondary processor. This allows the Leonardo to appear to a connected computer as a mouse and keyboard, in addition to a virtual serial port. This makes setting up interactive applications a very simple procedure - in your openFrameworks app all you need to do is check for a key press! **]

11.2 getting started with serial communication

SERIAL: ONE AFTER THE OTHER

Serial, in the most basic language sense, refers to things that come one after another; it's a term often used to describe magazines, crimes, and television programs. That meaning also applies when talking about serial data: "serial communication" means that all information between two entities is being sent one piece at a time, following in a single stream. One piece of data, or one bit, is just a binary piece of information: you're either sending a 0 or a 1. Using the terminology of digital electronics, these are frequently referred to as "high" and "low"; 0 is low (like turning a light off) and 1 is high (flipping it back on). 8 bits (for example the stream 01000001, which represents the letter A) are sometimes packaged together to create a single byte.

Serial communication is actually a very broad topic and there are many serial protocols, including audio-visual protocols such as DMX (based on RS-485) and MIDI (serial at 31,250 bits per second) which we'll briefly cover in this chapter. The most common serial protocol is called RS-232 and computers used to be equipped with RS-232 serial ports (remember them?) but today they are rarely present, which is why serial communications involving a computer will typically require an RS-232 to USB adaptor (found on-line or at your local electronics store).

Figure 11.1: RS-232 to USB adaptor

However, if you're connecting to an Arduino, it already appears to the computer as a virtual serial port and you just need a regular USB cable (the exact type is dependent on which model Arduino you have). The Arduino also has a built-in library which handles reading and writing to the serial port that appears on your computer. Additionally, the Arduino has bi-directional RS-232 serial ports which can be used to connect to other external serial devices. In short - the Arduino is well equipped for serial communications and does most of the hard work for you!

note: expand on Serial library

The speed at which data is transmitted between the Arduino and your software is measured in bits per second, or bps, a fairly self-explanatory unit of measurement. The rate of bits per second is commonly referred to as the baud rate, and will vary based on your application. For example, the standard baud rate of 9600bps will transfer data more slowly than a rate of 115200, but the faster baud rate may have more issues with byte scrambling.

```
-- editor joshuajnoble I think adding some explanation of what rs232
   is (a picture of an oscilloscope would be good) the flow of using:

enumerateDevices()
setup()
available()
close()

So you can find all serial devices, open the device, check if it has
   data, close the port and release the file handle.

Might be nice to have the Arduino serial example mirror the DMX
   example, like:


here's some Arduino code to kick this off

int redPin   = 9;   // Red LED
int greenPin = 10;  // Green LED
int bluePin  = 11;  // Blue LED

int color[4];
long int inByte;
int wait = 10; //10ms

void setup()
{
  pinMode(redPin,   OUTPUT);   // sets the pins as output
  pinMode(greenPin, OUTPUT);
  pinMode(bluePin,  OUTPUT);

  Serial.begin(9600);
```

```
}

void outputColour(int red, int green, int blue) {
  analogWrite(redPin, red);
  analogWrite(bluePin, blue);
  analogWrite(greenPin, green);
}

void setColor() {
  int i = 0;

  //wait and be patient
  while (i < 4)
  {
    if (Serial.available() > 0) {
        color[i] = Serial.read();
        i++;
    }
  }
}

// Main program
void loop()
{
  if (Serial.available() > 0) {
    // get incoming byte:
    inByte = Serial.read();

     if (inByte == 'C') {
      getColour();
      analogWrite(redPin, color[1]);
      analogWrite(bluePin, color[2]);
      analogWrite(greenPin, color[3]);
    }
  }
  delay(wait);
}

-- end editor
```

11.3 digital and analog communication

USING SERIAL MONITOR WITH ARDUINO

The Arduino IDE has a built-in Serial monitor, which enables you to “tune in” to the data coming across a serial port at a specified baud rate. You can find the Serial Monitor

either under Tools - Serial Monitor in the Arduino menu bar, or in the "magnifying glass" icon at the top of the IDE.

In the Arduino sketch, set up Serial communication and print a basic "Hello world!" phrase to the Serial monitor in the setup() function:

```
Serial.begin(9600);
Serial.println("Hello world!");
```

Open the Serial monitor and make sure that your monitor is set to the same baud rate as your sketch (the variable that you set in Serial.begin()). Note that the TX/RX lights on your Arduino flash once on setup - you can press the reset button on your Arduino (on most devices) to run setup again to see this more clearly. The Arduino itself is sending the printed text over Serial and the Serial Monitor is picking up that text and printing it for us to read.

Of course, you can also use the Serial monitor to reflect more interesting data coming from devices that are connected to your Arduino. You can print both digital and analog input data to Serial.

Digital inputs like pushbuttons, or a switch that only has open and closed positions, will send a binary value as their status. You can walk through an example of connecting a pushbutton to your Arduino in the excellent Arduino tutorial found here:

http://arduino.cc/en/tutorial/button

To print the value of the pushbutton to Serial, store the returned value from digitalRead in a variable and print that variable to Serial, just as done previously with the "hello world!" example. Here, though, print in the loop() function rather than setup() so that the value updates continuously rather than only once on reset.

```
buttonState = digitalRead(buttonPin);
Serial.println( buttonState );
```

If you're feeling limited by the binary nature of the pushbutton, you can also read from an analog input, using a component like a potentiometer, photoresistor, or any of the wide variety of sensors that provide non-binary input.

Analog inputs can only be connected on pins A0-A5 on a standard Arduino. These analog-specific pins have an ADC (analog-digital converter) chip which enables them to convert the amount of voltage returning from the circuit into a digital-readable number between 0 and 1023.

There are a number of examples and tutorials for working with analog input; a basic one using a potentiometer can be found here:

http://arduino.cc/en/Tutorial/AnalogInput

Printing the incoming variables to the Serial monitor is the same as with a digital input, except that you'll be using the Arduino function for analogRead() rather than digitalRead():

```
sensorValue = analogRead(sensorPin);
Serial.println( sensorValue );
```

11.4 using serial for communication between arduino and openframeworks

In the same way that Arduino uses Serial communication for communication between hardware and the Serial monitor, it can also use Serial communication to communicate between the Arduino board and any other running application, including openFrameworks. This can be done quite simply using the ofSerial class, native to openFrameworks. This class sets up a Serial listener at a specified baud rate and Serial port, giving it access to the same streaming data as the Serial library in the native Arduino IDE.

There's a good, heavily commented demonstration of this in the communications folder of examples that comes bundled with openFrameworks. The basic components needed to get this working are a Serial object, a toggle for knowing whether a message has been sent or not, and an array for storing the data that we receive.

USING FIRMATA AS A SERIAL PROTOCOL

Though it's possible to navigate all serial communication manually, you'll reach the limitations of what you're able to do fairly quickly - as soon as you start wanting to address different devices or have multiple inputs, you'll fall into a spiral of packet management and be much more prone to getting corrupt packets or inaccurate and scrambled data. Rather than deal with this manually, it's much simpler to use Firmata, an open source protocol for managing multiple Serial streams.

OFARDUINO

ofArduino, the built-in Arduino communication class for openFrameworks, is based on Firmata protocol. Using ofArduino assumes that a default Firmata sketch is loaded onto the Arduino. This sketch, found in the Arduino examples folder, enables all of the pins of the Arduino for communication through both analog and digital communication, as well as more specific addressing of pins for servomotor control.

EXAMPLE: Work through the same LED blink sketch as done previously with only Arduino, but with OF.

The basic flow of what we're going to do looks like this: (graphic missing)

- Make an ofArduino object
- Connect to the Arduino object at the correct port and baud rate
- Set up an event listener to determine whether we're successfully connected to the Arduino

- Set up a pin to communicate with, and specify whether that communication is analog or digital
- Poll for data from the serial port
- Send HIGH and LOW (or analog value) arguments to that pin

Make an ofArduino object

The first step is to add an ofArduino object into the header file of your project (usually, testApp.h). I'll call this myArduino.

```
void setup();
void update();
void draw();

ofArduino myArduino;
```

Now we've extended the capabilities of the native openFrameworks ofArduino class into our sketch, and we can work with the object myArduino.

Connect to the Arduino object at the correct port and baud rate

In the setup() of testApp.cpp, use the ofArduino `connect()` function to set up a connection at the appropriate port and baud rate. `connect()` takes two parameters: the first is a String of the serial port name, which should match the serial port name you connected to in the Arduino application; the second is the baud rate. Firmata uses a standard baud rate of 57600 bps.

```
ard.connect("/dev/tty.usbserial-a700fiyD", 57600);
```

Set up an event listener to determine whether we've successfully connected to the Arduino

If you're working only within the Arduino IDE, it's easy to have functions (like setting up the pin modes) called only once at the start of the program – you can just call those functions from within `setup()` with the confidence that they'll always be run once when the device initializes. When you're communicating with other software like openFrameworks, however, it's important to have a checking system to ensure that any setup functions only occur after a connection has been established. openFrameworks uses the ofEventUtils class to make this easier, relying on the default `ofAddListener()` and `ofRemoveListener()` functions to check for the connection event.

Within the openFrameworks app, we'll want to create an Arduino-specific `setup()` function, which is only called once as a result of the serial connection being established. We'll declare this function first in testApp.h:

```
void setupArduino(const int & version);
```

... and call it from testApp.cpp:

```
void testApp::setupArduino(const int & version) {
    // Arduino setup tasks will go here
}
```

The argument that's being passed to the function, `const int & version`, is a default return from the listener we're about to set up, which always responds to a connection event by sending back an argument with the connected firmware version. That can stay as it is.

In the `setup()` of testApp.cpp, create a listener using `ofAddListener()`. `ofAddListener()` is a function of ofEventUtils, which takes the arguments (event object, callback object, callback function). When the event object happens (in this case, when the ofArduino EInitialized event is triggered), ofAddListener tells the callback object (here, a pointer to the testApp itself, referred to as "this") to perform the setupArduino function that we created in the last step.

```
ofAddListener(myArduino.EInitialized, this, &testApp.setupArduino);
```

When the EInitialized event is triggered (when the connection to the Arduino is complete, and the Arduino responds by sending back information about its firmware version), the listener sends us to the callback function, which in this case is `setupArduino()`.

Within `setupArduino()`, we can remove the listener, because we know a connection has been established. `ofRemoveListener()` takes the same arguments as its counterpart.

```
ofRemoveListener(myArduino.EInitialized, this,
    &testApp.setupArduino);
```

Set up a pin to communicate with, and specify whether that communication is analog or digital

Now it's time to treat our Arduino setup function just like we would within a standard Arduino app, and set up our pins and pin modes. Here, I'm going to set up my Arduino Pin 13 as a digital output, in preparation for making a basic LED blink.

```
myArduino.sendDigitalPinMode(13, ARD_OUTPUT);
```

The other options for pin setup follow in line with standard Arduino pin settings:

```
sendDigitalPinMode(PIN_NUMBER, ARD_INPUT) // digital input
sendAnalogPinMode(PIN_NUMBER, ARD_OUTPUT) // analog output
sendAnalogPinMode(PIN_NUMBER, ARD_INPUT) // analog input
```

Poll for data from the serial port

In order to continuously update with new information on the serial port, it's important to periodically call the ofArduino `update()` function. This can be done in its own Arduino-specific function, or can be called directly from `testApp::update()`:

```
myArduino.update();
```

That's it! Now you're ready to start sending digital signals to pin 13 on your Arduino.

There are any number of triggers that you can use to control this signalling: you could set up a timer, integrate it into a game event, use a camera input... the possibilities are endless! Here, I'm going to trigger my Pin 13 LED to turn on and off based on the up and down arrow keys.

Because I'm controlling activity with keyboard keys, I'm going to use the `void testApp::keyPressed (int key)` function, but you could also place your triggers within `draw()` or another function depending on your desired effect.

```
void testApp::keyPressed  (int key){
    switch (key) {
        case OF_KEY_UP:
                ard.sendDigital(13, ARD_HIGH);  // turn LED on
                break;
            case OF_KEY_DOWN:
                ard.sendDigital(13, ARD_LOW); // turn LED off
                break;
            default:
                break;
        }
    }
```

When all the parts are together, run the app and toggle your UP and DOWN arrow keys to turn the on-board LED on your Arduino on and off! You can also put in a 3mm or 5mm LED on pin 13 to make the effect more obvious. Remember that pin 13 is the only Arduino pin with a built-in resistor, so if you want to add LEDs or other components on other pins, you'll need to build a full circuit with resistors to avoid burning out your parts.

11.5 Lights On - controlling hardware via DMX

DMX (which stands for Digital Multiplex), also known as DMX512 (512 being the number of channels each output can accommodate), is a protocol for controlling lighting and stage equipment. It's been around since the 80's, and is sometimes referred to as the MIDI of the lighting world as it achieves a fairly similar outcome - the sequencing and controlling of hardware through the use of a computer. DMX can be used to control anything from strobes to RGB par-can lights to LED fixtures. It's even possible to drive

LED strips by Pulse Width Modulation if you have the right hardware. The advantage of sending DMX through a custom openFrameworks app is that you can then integrate it via all the other goodness OF has to offer, including custom GUI's, custom sequencing algorithms, camera tracking - you name it.

Overview of the DMX protocol

In order to send DMX first of all you need a DMX to USB control interface. This is a special box that you'll need to purchase in order to enable your computer to send DMX data via a USB port. These interfaces can be easily purchased on-line in case you can't track one down locally. You'll also need some DMX cables to connect between the interface and the DMX device you want to control. Microphone cables with XLR connectors can be used to send DMX data, although the official standard for DMX is a 5-pin cable, unlike the 3-pins that XLR has to offer. There does exist adaptors between 5 and 3-pin connectors in case you need to mix and match them. In any case, hook up your hardware via cables to your DMX to USB device, install your drivers if required (Operating System dependent) and you are ready to send. As previously mentioned, each DMX output from your controller can send up to 512 channels of DMX data. In DMX terminology, each group of 512 channels is known as a "Universe". Multiple DMX Universes are often used in complex setups requiring lots of channels. Typically you won't need more than a single universe as a single coloured light will only use up around 3 channels (one each for red, green and blue).

DMX data format

A DMX packet, in other words the data sent to the hardware each frame, consists of 512 channels, with an 8-bit value sent per channels (i.e. 0-255). One of the idiosyncracies of DMX is that the channel numbering starts at 1, channel 0 being a start code and not a data channel. This means that when setting up an array to hold your per-frame DMX data, you'll need to make it a size of 513 bytes. In openFrameworks we almost always represent a byte as an unsigned char, though you can also represent this with other types.

```
//setup the data structure
unsigned char dmxData[513];

// zero the first value
dmxData[0] = 0;

// channels are valid from here on up
dmxData[1] = 126;
```

A number of OF addons have sprung up around DMX, a quick search of ofxAddons.com will reveal the most up to date. Typically these addons will have set up all the necessary data structures, including the one above, so you won't need to worry about anything other than sending the right data to the right channels. The hardest part will probably be installing the drivers for your controller!

Figure 11.2: DMX Lighting

Structure of an OF DMX application

No matter which code or which addon you use, the way in which you'll send DMX data will be very similar to the following pseudo-code (replace the comments with the relevant code):

```
void ofApp::setup() {
    //connect to your DMX controller
}

void ofApp::update() {

    //assign the relevant values to your DMX data structure

    //update the DMX controller with the new data
}
```

The only concern then becomes what colour you'll be setting your lights and how you'd like to dim them.

Using a colour picker to set up your lights

> editor – see above, I think could be cool to mirror this with the ofSerial example

TODO

> editor joshuajnoble I really feel like we should have rpi in its own chapter. It's so tricky to get setup. I do think that talking about something like wiringPi in the context of hardware is a really good idea though, for sure.

11.6 Raspberry Pi - getting your OF app into small spaces

The Raspberry Pi is a popular small sized computer (also known as a single board computer) running on hardware not entirely dissimilar to that which powers today's smartphones. The processor at least, is part of the same ARM family of chips. Originally the Raspberry Pi (abbreviated as RPi) was originally developed as an educational platform to be able to teach the basics of computing hardware in a simple and affordable package. The Raspberry Pi is part of a much larger ecosystem of ARM devices, and the Model B Pi, the most popular version available shortly after launch, is technically classified as an ARM6 device. OpenFrameworks currently supports ARM6 and ARM7 devices, of which the latter are typically more recent and faster hardware designs. While there are plenty of small form-factor alternatives to the Pi, it's a good choice as a computing platform due to the community that's formed around it and the various hardware and software extensions that have been developed for it. The Raspberry Pi is also completely open source, including the source code for the Broadcom graphics stack that

it contains, which is quite unusual in the hardware world. The advantages of this are again that it enables enthusiasts and professionals from within the RPi community to extend this device to its fullest potential. Having a platform that is well tested and can be used in many different applications is also of benefit, particularly for installations that need to run for extensive periods of time. However, as with any technology, there are advantages and there are caveats, which we'll cover here, along with some practical scenarios which might be useful to anyone interested in taking this mini-computer into the wilds.

Setting up a Raspberry Pi

The Raspberry Pi is typically loaded with a version of the Linux operating system that has been tailored for its particular hardware requirements. The most common is the Raspbian distribution (recommended by the Raspberry Pi Foundation, the organisation producing the device) however whichever distribution you choose you'll end up having to engage with the intricacies of Linux at some point, and becoming familiar with using the command-line will be necessary. However fear not, you won't need to dig too deep and most likely you'll just be following well documented guides that others have set up - one of the aforementioned benefits of using this gadget. Having approximately the surface area of an Arduino Mega (although having a slightly higher profile), while far slower than a desktop computer, the RPi is a relatively powerful device with out-of-the-box features (USB 1.1 hub, full networking stack, HDMI and audio out etc.) than you won't see on an Arduino yet similarly priced. More importantly, within the context of this chapter, the RPi is both small and has a serial port, which lends itself to well to hardware applications where often both of those are prerequisites. There's not much point in describing the full setup process of a Raspberry Pi as it's better described elsewhere. The following links will help you get started:

Raspberry Pi quick start[3]

OpenFrameworks Raspberry Pi setup[4]

Raspberry Pi and Serial Ports

There are multiple ways to set up a serial connection on the RPi. The first method requires a USB to serial cable and involves a virtual serial communication port. If you plug in your USB to serial cable, it should automatically create a software serial port . The naming and location of virtual serial ports on the Raspberry Pi is similar to what you'd find on any stardard Linux operating system. It's also not unlike the setup found on OSX, which has a unix-like foundation. If you're coming from Windows however, it's a little different to the system of COM ports (COM1, COM2 etc) used there.

Assuming you've successfully installed your RPi, if you've booted into the graphical environment, then open up a terminal window (double click the LXTerminal icon on

[3] http://www.raspberrypi.org/help/quick-start-guide/
[4] http://www.openframeworks.cc/setup/RPI

the desktop if you're using Raspbian). Otherwise you'll already be on the command-line. Either way, type the following command:

```
ls -la /dev/ttyUSB*
```

This should list all of the serial devices on your system, starting from ttyUSB0 and counting upwards. If you see such a device, your USB serial cable has been installed correctly. Plugging in an Arduino to a Raspberry Pi will also create such a serial device, so if you're heading down that path this is where you should look to confirm it is connected. The next thing you should check before moving on however is check what permissions the serial port has. The Linux file system, much like OSX under the hood, uses the concept of an "owner" of a file and what "group" it belongs to. If you are a member of a file group, you'll be able to access that file even if you are not its owner. File permissions are there for security reasons, which is a good thing, but sometimes they'll trip you up. The above command should return with something like this:

```
crw-rw---- 1 root dialout 4, 64 Apr  6 23:03 /dev/ttyUSB0
```

In order to be able to read and write to this device normally, you'll need to be a member of group it belongs to, in this case, the "dialout" group.

To check which groups you belong to, type in the following command:

```
groups
```

This will return something like:

```
adm dialout cdrom sudo dip video plugdev lpadmin sambashare
```

The only thing of note here is to make sure that the group assigned to the serial port (in this case "dialout") is in your groups list. Typically the default "pi" user wil be a member of this group. If it isn't, just run the following command to add it, appending your user login name at the end:

```
usermod -a -G dialout yourusername
```

Now you should be able to connect to other serial devices using your cable.

Raspberry Pi and GPIO

The RPi has a hardware serial port as well, literally two of the GPIO (General Purpose Input Output) pins on the board are dedicated to hardware based serial transmit and receive. They are listed as pins 8 and 10 or GPIO14 and GPIO15 in the documentation. Don't get confused by the two different names, one is relative to all the pins (8 and 10) and the other is relative to the usable programmable pins (GPIO14 and GPIO15). You might be inclined to use these if you've already used up the Pi's USB ports, or you might want to use them to interface directly with electronics. A word of caution however to

anyone connecting any voltages directly to the GPIO pins - there's no protection here against wrong or incorrect voltages. It's nothing to be alarmed about, you're just going to be interfacing directly with the board and the advantages this opens up also exposes you to potentially damaging the board if you do something wrong. Always check your connections and voltages before hooking things up, its a good habit to get into and will avoid frying things. Although truth be told, the only real way to learn is to make mistakes, and there's nothing that'll imprint your memory more effectively than the smell of burning silicon, but I digress...

Another thing to keep in mind when using the hardware serial port is that all of the GPIO pins operate at around 3 volts. This means that a logic "high" will be approximately 3 volts, which differs from the more typical 5 volts used by most (but not all) Arduino devices. There are quite a few sensors and other electronics that operate at 3 volts, and a quick web search will point you in the right direction. This lower than normal voltage is quite useful in low power electronic setups, where the slightly lower voltage will save you a bit of energy, useful when running off batteries. However again be careful when hooking up 3 volt based sensors directly to the Pi as it may be safer to include a bit of protection between the electronics and the Pi to avoid damaging the board.

In order to compensate for the difference in voltage between a regular 5V Arduino and the Pi, you'll need a logic level converter between the two. A quick web search should turn up a converter that'll suit your needs, all you need to then do is hook the two devices up to the converter, with the Pi connecting to the 3V side of the converter and the Arduino hooked up to the 5V side. Remember to always make sure that the serial transmit (or TX for short) from one board is connected to the serial receive (or RX for short) on the other. You'll need power and ground supplied from both sides too. Once this is done, you're ready to start communicating between the two. All the openFrameworks procedures of communicating between an Arduino and OF, detailed above, now apply. As a fail-safe test of whether the serial port is set up correctly, you can always do a test on the Pi side by using the venerable minicom command-line application. It's a text only interface to the serial port, useful for quick low-level serial debugging. It might feel a little archaic initially, but it does its job and it's very lightweight.

To install minicom, just open a terminal window and type the following:

```
sudo apt-get install minicom
```

You can then send characters to the Arduino via Minicom, and view them using the serial monitor on the Arduino. Make sure both devices have the same baud rate. Launch minicom using the following command (if operating at 9600 baud):

```
minicom -b 9600 -o -D /dev/ttyAMA0
```

where the number 9600 represents the baud rate and /dev/ttyAMA0 is the address of our GPIO serial port. What you type into the minicom terminal screen should appear on the Arduino serial monitor and vice versa.

Case Study: Raspberry PI as a master DMX controller

One of the nifty uses of a Raspberry Pi is to use it as a master DMX controller. It takes up little space and sending serial data doesn't require much processing power so it's a perfect candidate. Add to that the ability to control your OF app running on the PI remotely via OSC, for example using an Android or IOS tablet, and you have a fully fledged custom DMX set up which would otherwise require expensive commercial software to replicate.

Headless Apps

Given that the Raspberry Pi can be viewed as a Swiss-army knife in the installation world and has less processing power than a desktop computer, you may often find yourself using it for a specific task and that task may not require any graphical output (like churning out DMX commands for example). In this case, you can disable the graphical capabilities of openFrameworks in order to streamline your application for the task at hand. Using an application in this fashion is know as running the application "headless". In order to build a headless application (which works for all target platforms, not just the RPi), all you'll need to do is open up your project's main.cpp and change it to the following:

```
//ofSetupOpenGL(1024,768, OF_WINDOW);            // <--------
   comment out the setup of the GL context

// this kicks off the running of my headless app:
ofRunApp( new ofAppNoWindow());
```

Voila! Your application will now run without opening a graphical window. This can be particularly useful when launch it from a script or the command-line, as well see shortly.

Running your app on start-up

So, you've created an OF app that does some amazing stuff, and chances are you've turned to this little over-achieving box because of it's size and now you want to incorporate it into an installation. Ideally, technical installations, whether interactive or not, should be super easy to set up and turn on, in order to place little demands on gallery or event staff that might be minding the piece, and to reduce the chances of something going wrong. Ideally, you'll turn the power of the Pi on, and once it has booted up your app will run. Additionally, if your app doesn't make use of any graphics or visual output, you might want to run it in what's called "headless" mode, where the graphical capabilities of OF are turned off. This will save power and reduce processor demand on the Pi.

Running an application on start up can be done in many different ways on a Linux based system such as the Pi. However here are a couple of methods that you could use depending on your needs. Firstly, there's a file called "rc.local" in amongst the Pi's system files where you can list applications that'll be run on start up. In order to make

use of this you'll need to have the Pi booting into the command line. If the Pi is running the GUI, this won't work. To configure your Pi to start up in command-line mode, run the So to make use of this method, open up a terminal and do the following:

```
sudo nano /etc/rc.local
```

This will open up this file using the command-line text editor "nano" and because it is a system file you'll need to use the "sudo" (short for Super User Do ....i.e. do something as if you were the super user, a user with permission to modify system files) command to give you access to an otherwise protected file. Once you've opened this file, after the initial comments (lines beginning with '#') add the following lines (replacing "myProject" with the name of your app):

```
# Auto run our application
/home/pi/openFrameworks/apps/myApps/myProject/bin/myProject &
```

Add these lines and then hit CTRL-X to save (hit 'Y to confirm that you want to overwrite the current file). The ampersand indicates thatThen if all has gone well, next time you reboot your application will launch. You won't have any window decorations (such as a border and buttons for minimise and close), but if you run the application full-screen that won't matter anyway, and you have the advantage of not running a GUI environment which eats up resources on the otherwise busy Pi.

The other way to run applications is to use the "cron" system. Cron, if enabled (which it normally will be by default on a Pi), is a daemon (or persistent application that runs in the background) is a piece of software whose only purpose is to carry out actions at particular times of the day. Essentially it's a scheduler, much like a calendar reminder system except instead of reminders you'll be scheduling various tasks. All you need to do then will be to tell Cron to run your app whenever the system boots up. All cron actions are stored in special files which exist on the Pi in the /var/spool/cron/contab/ directory. You aren't allowed to edit these files directly, you need to run a special application called *crontab* which you can then use to create and modify those files.

In the case of starting our app on boot, all we need to do then is type the following in a terminal (replace 'pi' with your username if not logged in with the default account):

```
sudo crontab -e -u pi
```

Then a file will open up and you can edit it in the same way as you would using nano (if fact *crontab* is making use of *nano* in this process). Just add the following line to the file:

```
@reboot  /home/pi/openFrameworks/apps/myApps/myExampleApp &
```

Substitute the right app and and the right folder names depending on where your OF app is located. You'll need to get the full path right, so make sure it's correct before

hitting CTRL-X to save the file. Once saved, again all you'll need to do is reboot to witness your app being kicked off on start-up. Presto!

In either case above, if you wanted to make sure that your app restarted upon crashing, you could wrap the application in a shell script running a while loop, such as this:

```
#!/bin/bash
cd /home/pi/openFrameworks/app/myApps/myExampleApp/bin
while true; do
    ./myExampleApp
done
```

All you would need to do would be to copy the above code into a file and save it as something like "myApp.sh". Then make it executable by changing the file's permissions:

```
chmod a+x myApp.sh
```

To test the shell script, just try running it in a terminal window by typing "./myApp.sh" in the same directory as the script. If it launches successfully, you can then replace any direct references to the app in the above auto-start examples with this script. Keep in mind, the only way to fully kill the app will be to kill the process running this shell script with a command such as

```
kill -9 1234
```

where you need to replace '1234' with the process id (PID) of your script. The PID can be found by typing

```
ps -A
```

in a terminal. This will list all running processes on the system. Anyhow, that's enough system administration for the moment, time to get creative instead.

12 Sound

by Adam Carlucci[1]

This chapter will demonstrate how to use the sound features that you'll find in openFrameworks, as well as some techniques you can use to generate and process sound.

Here's a quick overview of the classes you can use to work with sound in openFrameworks:

`ofSoundPlayer` provides simple access to sound files, allowing you to easily load and play sounds, add sound effects to an app and extract some data about the file's sound as it's playing.

`ofSoundStream` gives you access to the computer's sound hardware, allowing you to generate your own sound as well as react to sound coming into your computer from something like a microphone or line-in jack.

As of this writing, these classes are slated to be introduced in the next minor OF version (0.9.0):

`ofSoundBuffer` is used to store a sequence of `float` values, and perform audio-related things on said values (like resampling)

`ofSoundFile` allows you to extract uncompressed ofSoundBuffers from files.

`ofSoundObject` is an interface for chaining bits of sound code together, similar to how a guitarist might use guitar pedals. This is mostly relevant for addon authors or people looking to share their audio processing code.

12.1 Getting Started With Sound Files

Playing a sound file is only a couple lines of code in openFrameworks. Just point an `ofSoundPlayer` at a file stored in your app's data folder and tell it to play.

```
class ofApp : public ofBaseApp {
  ...
  ofSoundPlayer soundPlayer;
};
```

[1] http://adamcarlucci.com

```
void ofApp::setup() {
  soundPlayer.loadSound("song.mp3");
  soundPlayer.play();
}
```

This is fine for adding some background music or ambiance to your app, but ofSoundPlayer comes with a few extra features that are particularly handy for handling sound effects.

"Multiplay" allows you to have a file playing several times simultaneously. This is great for any sound effect which might end up getting triggered rapidly, so you don't get stuck with an unnatural cutoff as the player's playhead abruptly jumps back to the beginning of the file. Multiplay isn't on by default. Use `soundPlayer.setMultiPlay(true)` to enable it. Then you can get natural sound effect behaviour with dead-simple trigger logic like this:

```
if ( thingHappened )
  soundPlayer.play();
}
```

Another feature built-in to ofSoundPlayer is speed control. If you set the speed faster than normal, the sound's pitch will rise accordingly, and vice-versa (just like a vinyl record). Playback speed is defined relative to "1", so "0.5" is half speed and "2" is double speed.

Speed control and multiplay are made for each other. Making use of both simultaneously can really extend the life of a single sound effect file. Every time you change a sound player's playback speed with multiplay enabled, previously triggered sound effects continue on unaffected. So, by extending the above trigger logic to something like...

```
if( thingHappened ) {
  soundPlayer.setSpeed(ofRandom(0.8, 1.2));
  soundPlayer.play();
}
```

...you'll introduce a bit of unique character to each instance of the sound.

One other big feature of ofSoundPlayer is easy spectrum access. On the desktop platforms, you can make use of ofSoundGetSpectrum() to get the *frequency domain* representation of the sound coming from all of the currently active ofSoundPlayers in your app. An explanation of the frequency domain is coming a little later in this chapter, but running the openFrameworks *soundPlayerFFTExample* will give you the gist.

Ultimately, ofSoundPlayer is a tradeoff between ease-of-use and control. You get access to multiplay and pitch-shifted playback but you don't get extremely precise control or access to the individual samples in the sound file. For this level of control, ofSoundStream is the tool for the job.

12.2 Getting Started With the Sound Stream

ofSoundStream is the gateway to the audio hardware on your computer, such as the microphone and the speakers. If you want to have your app react to live audio input or generate sound on the fly, this is the section for you!

You may never have to use the ofSoundStream directly, but it's the object that manages the resources needed to trigger `audioOut()` and `audioIn()` on your app. These two functions are optional members of your ofApp, like `keyPressed()`, `windowResized()` and `mouseMoved()`. They will start being called once you implement them and initiate the sound stream. Here's the basic structure for a sound-producing openFrameworks app:

```
class ofApp : public ofBaseApp {
  ...
  void audioOut( float * output, int bufferSize, int nChannels );
  double phase;
}

void ofApp::setup() {
  phase = 0;
  ofSoundStreamSetup(2, 0); // 2 output channels (stereo), 0 input
      channels
}

void ofApp::audioOut( float * output, int bufferSize, int nChannels
    ) {
  for(int i = 0; i < bufferSize * nChannels; i+=2) {
    float sample = sin(phase); // generating a sine wave sample
    output[i] = sample; // writing to the left channel
    output[i+1] = sample; // writing to the right channel
    phase += 0.05;
  }
}
```

When producing or receiving audio, the format is floating point numbers between -1 and 1 (the reason for this is coming a little later in this chapter). The sound will arrive in your app in the form of *buffers*. Buffers are just arrays, but the term "buffer" implies that each time you get a new one, it represents the chunk of time after the previous buffer. The reason openFrameworks asks you for buffers (instead of individual samples) is due to the overhead involved in shuttling data from your program to the audio hardware, and is a little outside the scope of this book.

The buffer size is adjustable, but it's usually a good idea to leave it at the default. The default isn't any number in particular, but will usually be whatever the hardware on your computer prefers. In practice, this is probably about 512 samples per buffer (256 and 1024 are other common buffer sizes).

Sound buffers in openFrameworks are *interleaved* meaning that the samples for each channel are right next to each other, like:

```
[Left] [Right] [Left] [Right] ...
```

This means you access individual sound channels in much the same way as accessing different colours in an ofPixels object (i.e. `buffer[i]` for the left channel, `buffer[i + 1]` for the right channel). The total size of the buffer you get in `audioIn()` / `audioOut()` can be calculated with `bufferSize * nChannels`.

An important caveat to keep in mind when dealing with ofSoundStream is that audio callbacks like `audioIn()` and `audioOut()` will be called on a seperate *thread* from the standard `setup()`, `update()`, `draw()` functions. This means that if you'd like to share any data between (for example) `update()` and `audioOut()`, you need to make use of an `ofMutex` to keep both threads from getting in each others' way. You can see this in action a little later in this chapter, or check out the threads chapter for a more in-depth explanation.

12.3 Why -1 to 1?

In order to understand *why* openFrameworks chooses to represent sound as a continuous stream of `float` values ranging from -1 to 1, it'll be helpful to know how sound is created on a physical level.

[a minimal picture showing the mechanics of a speaker, reference: http://wiki.backyardbrains.com/im]

At the most basic level, a speaker consists of a cone and an electromagnet. The electromagnet pushes and pulls the cone to create vibrations in air pressure. These vibrations make their way to your ears, where they are interpreted as sound. When the electromagnet is off, the cone is simply "at rest", neither pulled in or pushed out.

[footnote] A basic microphone works much the same way: allowing air pressure to vibrate an object held in place by a magnet, thereby creating an electrical signal.

From the perspective of an openFrameworks app, it's not important what the sound hardware's specific voltages are. All that really matters is that the speaker cone is being driven between its "fully pushed out" and "fully pulled in" positions, which are represented as 1 and -1. This is similar to the notion of "1" as a representation of 100% as described in the animation chapter, though sound introduces the concept of -100%.

[footnote] Some other systems use an integer-based representation, moving between something like -65535 and +65535 with 0 still being the representation of "at rest". The Web Audio API provides an unsigned 8-bit representation, which ranges between 0 and 255 with 127 being "at rest".

A major way that sound differs from visual content is that there isn't really a "static" representation of sound. For example, if you were dealing with an OpenGL texture which represents 0 as "black" and 1 as "white", you could fill the texture with all 0s or all 1s and end up with a static image of "black" or "white" respectively. This is not the case with sound. If you were to create a sound buffer of all 0s, all 1s, all -1s, or any single number, they would all sound like exactly the same thing: nothing at all.

[footnote] Technically, you'd probably hear a pop right at the beginning as the speaker moves from the "at rest" position to whatever number your buffer is full of, but the remainder of your sound buffer would just be silence.

This is because what you actually hear is the *changes* in values over time. Any individual sample in a buffer doesn't really have a sound on its own. What you hear is the *difference* between the sample and the one before it. For instance, a sound's "loudness" isn't necessarily related to how "big" the individual numbers in a buffer are. A sine wave which oscillates between 0.9 and 1.0 is going to be much much quieter than one that oscillates between -0.5 and 0.5.

12.4 Time Domain vs Frequency Domain

When representing sound as a continuous stream of values between -1 and 1, you're working with sound in what's known as the "Time Domain". This means that each value you're dealing with is referring to a specific moment in time. There is another way of representing sound which can be very helpful when you're using sound to drive some other aspect of your app. That representation is known as the "Frequency Domain".

[image of a waveform vs an FFT bar graph, reference http://upload.wikimedia.org/wikipedia/commons/8/8c/]

In the frequency domain, you'll be able to see how much of your input signal lies in various frequencies, split into separate "bins" (see above image).

You can transform a signal from the time domain to the frequency domain by a ubiquitous algorithm called the Fast Fourier Transform. You can get an openFrameworks-ready implementation of the FFT (along with examples!) in either the ofxFFT or ofxFft addons (by Lukasz Karluk and Kyle McDonald respectively).

In an FFT sample, bins in the higher indexes will represent higher pitched frequencies (i.e. treble) and the lower ones will represent bassy frequencies. Exactly *which* frequency is represented by each bin depends on the number of time-domain samples that went into the transform. You can calculate this as follows:

```
frequency = (binIndex * sampleRate) / totalSampleCount
```

So, if you were to run an FFT on a buffer of 1024 time domain samples at 44100Hz, bin 3 would represent 129.2Hz (`(3 * 44100)/ 1024 ≈129.2`). This calculation demonstrates a property of the FFT that is very useful to keep in mind: the more time domain samples you use to calculate the FFT, the better frequency resolution you'll get (as in, each subsequent FFT bin will represent frequencies that are closer together). The tradeoff for increasing the frequency resolution this way is that you'll start losing track of time, since your FFT will be representing a bigger portion of the signal.

Note: A raw FFT sample will typically represent its output as Complex numbers[2], though this probably isn't what you're after if you're attempting to do something like audio visualization. A more intuitive representation is the *magnitude* of each complex number, which is calculated as:

```
magnitude = sqrt( pow(complex.real, 2) + pow(complex.imaginary, 2) )
```

If you're working with an FFT implementation that gives you a simple array of float values, it's most likely already done this calculation for you.

You can also transform a signal from the frequency domain *back* to the time domain, using an Inverse Fast Fourier Transform (aka IFFT). This is less common, but there is an entire genre of audio synthesis called Additive Synthesis which is built around this principle (generating values in the frequency domain then running an IFFT on them to create synthesized sound).

The frequency domain is useful for many things, but one of the most straightforward is isolating particular elements of a sound by frequency range, such as instruments in a song. Another common use is analyzing the character or timbre of a sound, in order to drive complex audio-reactive visuals.

The math behind the Fourier transform is a bit tricky, but it is fairly straightforward once you get the concept. I felt that this explanation of the Fourier Transform[3] does a great job of demonstrating the underlying math, along with some interactive visual examples.

12.5 Reacting to Live Audio

12.5.1 RMS

One of the simplest ways to add audio-reactivity to your app is to calculate the RMS of incoming buffers of audio data. RMS stands for "root mean square" and is a pretty straightforward calculation that serves as a good approximation of "loudness" (much better than something like averaging the buffer or picking the maximum value). The

[2] http://en.wikipedia.org/wiki/Complex_number
[3] http://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/

"square" step of the algorithim will ensure that the output will always be a positive value. This means you can ignore the fact that the original audio may have had "negative" samples (since they'd sound just as loud as their positive equivalent, anyway). You can see RMS being calculated in the *audioInputExample*.

```
// modified from audioInputExample
float rms = 0.0;
int numCounted = 0;

for (int i = 0; i < bufferSize; i++){
    float leftSample = input[i * 2] * 0.5;
    float rightSample = input[i * 2 + 1] * 0.5;

    rms += leftSample * leftSample;
    rms += rightSample * rightSample;
    numCounted += 2;
}

rms /= (float)numCounted;
rms = sqrt(rms);
// rms is now calculated
```

12.5.2 Onset Detection (aka Beat Detection)

Onset detection algorithms attempt to locate moments in an audio stream where an *onset* occurs, which is usually something like an instrument playing a note or the impulse of a drum hit. There are many onset detection algorithms available at various levels of complexity and accuracy, some fine-tuned for speech as opposed to music, some working in the frequency domain instead of the time domain, some made for offline processing as opposed to realtime, etc.

A simple realtime onset detection algorithm can be built on top of the RMS calculation above.

```
class class ofApp : public ofBaseApp {
    ...
    float threshold;
    float minimumThreshold;
    float decayRate;
}

void ofApp::setup() {
    ...
    decayRate = 0.05;
    minimumThreshold = 0.1;
    threshold = minimumThreshold;
}
```

```
void ofApp::audioIn(float * input, int bufferSize, int nChannels) {
    ...
    threshold = ofLerp(threshold, minimumThreshold, decayRate);

    if(rms > threshold) {
        // onset detected!
        threshold = rms;
    }
}
```

This will probably work fine on an isolated drum track, sparse music or for something like detecting whether or not someone's speaking into a microphone. However, in practice you'll likely find that this won't really cut it for reliable audio visualization or more intricate audio work.

You could of course grab an external onset detection algorithm (there's quite a few addons available for it), but if you'd like to experiment, try incorporating the FFT into your algorithm. For instance, try swapping the RMS for the average amplitude of a range of FFT bins.

12.5.3 FFT

Running an FFT on your input audio will give you back a buffer of values representing the input's frequency content. A straight up FFT *won't* tell you which notes are present in a piece of music, but you will be able to use the data to take the input's sonic "texture" into account. For instance, the FFT data will let you know how much "bass" / "mid" / "treble" there is in the input at a pretty fine granularity (a typical FFT used for realtime audio-reactive work will give you something like 512 to 4096 individual frequency bins to play with).

When using the FFT to analyze music, you should keep in mind that the FFT's bins increment on a *linear* scale, whereas humans interpret frequency on a *logarithmic* scale. So, if you were to use an FFT to split a musical signal into 512 bins, the lowest bins (bin 0 through bin 40 or so) will probably contain the bulk of the data, and the remaining bins will mostly just be high frequency content. If you were to isolate the sound on a bin-to-bin basis, you'd be able to easily tell the difference between the sound of bins 3 and 4, but bins 500 and 501 would probably sound exactly the same. Unless you had robot ears.

[footnote] There's another transform called the *Constant Q Transform* (aka CQT) that is similar in concept to the FFT, but spaces its bins out logarithmically which is much more intuitive when dealing with music. As of this writing I'm not aware of any openFrameworks-ready addons for the CQT, but it's worth keeping in mind if you feel like pursuing other audio visualization options beyond the FFT.

12.6 Synthesizing Audio

This section will walk you through the creation of a basic musical synthesizer. A full blown instrument is outside the scope of this book, but here you'll be introduced to the basic building blocks of synthesized sound.

A simple synthesizer can be implemented as a *waveform* modulated by an *envelope*, forming a single *oscillator*. A typical "real" synthesizer will have several oscillators and will also introduce *filters*, but many synthesizers at their root are variations on the theme of a waveform + envelope combo.

12.6.1 Waveforms

Your synthesizer's waveform will define the oscillator's "timbre". The closer the waveform is to a sine wave, the more "pure" the resulting tone will be. A waveform can be made of just about anything, and many genres of synthesis[4] revolve around techniques for generating and manipulating waveforms.

A common technique for implementing a waveform is to create a *Lookup Table* containing the full waveform at a certain resolution. A *phase* index is used to scan through the table, and the speed that the phase index is incremented determines the pitch of the oscillator.

Here's a starting point for a synthesizer app that we'll keep expanding upon during this section. It demonstrates the lookup table technique for storing a waveform, and also visualizes the waveform and resulting audio output. You can use the mouse to change the resolution of the lookup table as well as the rendered frequency.

```
class ofApp : public ofBaseApp {
public:
    void setup();
    void update();
    void draw();

    void updateWaveform(int waveformResolution);
    void audioOut(float * output, int bufferSize, int nChannels);

    std::vector<float> waveform; // this is the lookup table
    double phase;
    float frequency;

    ofMutex waveformMutex;
    ofPolyline waveLine;
    ofPolyline outLine;
};
```

[4] http://en.wikipedia.org/wiki/Category:Sound_synthesis_types

```
void ofApp::setup() {
    phase = 0;
    updateWaveform(32);
    ofSoundStreamSetup(1, 0); // mono output
}

void ofApp::update() {
    ofScopedLock waveformLock(waveformMutex);
    updateWaveform(ofMap(ofGetMouseX(), 0, ofGetWidth(), 3, 64,
       true));
    frequency = ofMap(ofGetMouseY(), 0, ofGetHeight(), 60, 700,
       true);
}

void ofApp::draw() {
    ofBackground(ofColor::black);
    ofSetLineWidth(5);
    ofSetColor(ofColor::lightGreen);
    outLine.draw();
    ofSetColor(ofColor::cyan);
    waveLine.draw();
}

void ofApp::updateWaveform(int waveformResolution) {
    waveform.resize(waveformResolution);
    waveLine.clear();

    // "waveformStep" maps a full oscillation of sin() to the size
    // of the waveform lookup table
    float waveformStep = (M_PI * 2.) / (float) waveform.size();

    for(int i = 0; i < waveform.size(); i++) {
        waveform[i] = sin(i * waveformStep);

        waveLine.addVertex(ofMap(i, 0, waveform.size() - 1, 0,
           ofGetWidth()),
                           ofMap(waveform[i], -1, 1, 0,
                              ofGetHeight()));
    }
}

void ofApp::audioOut(float * output, int bufferSize, int nChannels) {
    ofScopedLock waveformLock(waveformMutex);

    float sampleRate = 44100;
    float phaseStep = frequency / sampleRate;

    outLine.clear();
```

```
    for(int i = 0; i < bufferSize * nChannels; i += nChannels) {
        phase += phaseStep;
        int waveformIndex = (int)(phase * waveform.size()) %
            waveform.size();
        output[i] = waveform[waveformIndex];

        outLine.addVertex(ofMap(i, 0, bufferSize - 1, 0,
            ofGetWidth()),
                          ofMap(output[i], -1, 1, 0, ofGetHeight()));
    }
}
```

Once you've got this running, try experimenting with different ways of filling up the waveform table (the line with `sin(...)` in it inside `updateWaveform(...)`). For instance, a fun one is to replace that line with:

```
waveform[i] = ofSignedNoise(i * waveformStep, ofGetElapsedTimef());
```

This will get you a waveform that naturally evolves over time. Be careful to keep your waveform samples in the range -1 to 1, though, lest you explode your speakers and / or brain.

12.6.2 Envelopes

We've got a drone generator happening now, but adding some volume modulation into the mix will really bring the sound to life. This will let the waveform be played like an instrument, or otherwise let it sound like it's a living being that reacts to events.

We can create a simple (but effective) envelope with `ofLerp(...)` by adding the following to our app:

```
class ofApp : public ofBaseApp {
    ...
    float volume;
};

void ofApp::setup() {
    ...
    volume = 0;
}

void ofApp::update() {
    ...
    if(ofGetKeyPressed()) {
        volume = ofLerp(volume, 1, 0.8); // jump quickly to 1
    } else {
```

```
        volume = ofLerp(volume, 0, 0.1); // fade slowly to 0
    }
}

void ofApp::audioOut(float * output, int bufferSize, int nChannels) {
    ...
    output[i] = waveform[waveformIndex] * volume;
    ...
}
```

Now, whenever you press a key the oscillator will spring to life, fading out gradually after the key is released.

The standard way of controlling an envelope is with a relatively simple state machine called an ADSR, for "Attack, Decay, Sustain, Release".

- **Attack** is how fast the volume reaches its peak after a note is triggered
- **Decay** is how long it takes the volume to fall from the peak
- **Sustain** is the resting volume of the envelope, which stays constant until the note is released
- **Release** is how long it takes the volume to drop back to 0 after the note is released

A full ADSR implementation is left as an exercise for the reader, though this example from earlevel.com[5] is a nice reference.

12.6.3 Frequency Control

You can probably tell where we're going, here. Now that the app is responding to key presses, we can use those key presses to determine the oscillator's frequency. We'll introduce a bit more `ofLerp(...)` here too to get a nice *legato* effect.

```
class ofApp : public ofBaseApp {
    ...
    void keyPressed(int key);
    float frequencyTarget;
}

void ofApp::setup() {
    ...
    frequency = 0;
    frequencyTarget = frequency;
}

void ofApp::update() {
    ...
```

[5] http://www.earlevel.com/main/2013/06/03/envelope-generators-adsr-code/

```
    // replace the "frequency = " line from earlier with this
    frequency = ofLerp(frequency, frequencyTarget, 0.4);
}

void ofApp::keyPressed(int key) {
    if(key == 'z') {
        frequencyTarget = 261.63; // C
    } else if(key == 'x') {
        frequencyTarget = 293.67; // D
    } else if(key == 'c') {
        frequencyTarget = 329.63; // E
    } else if(key == 'v') {
        frequencyTarget = 349.23; // F
    } else if(key == 'b') {
        frequencyTarget = 392.00; // G
    } else if(key == 'n') {
        frequencyTarget = 440.00; // A
    } else if(key == 'm') {
        frequencyTarget = 493.88; // B
    }
}
```

Now we've got a basic, useable instrument!

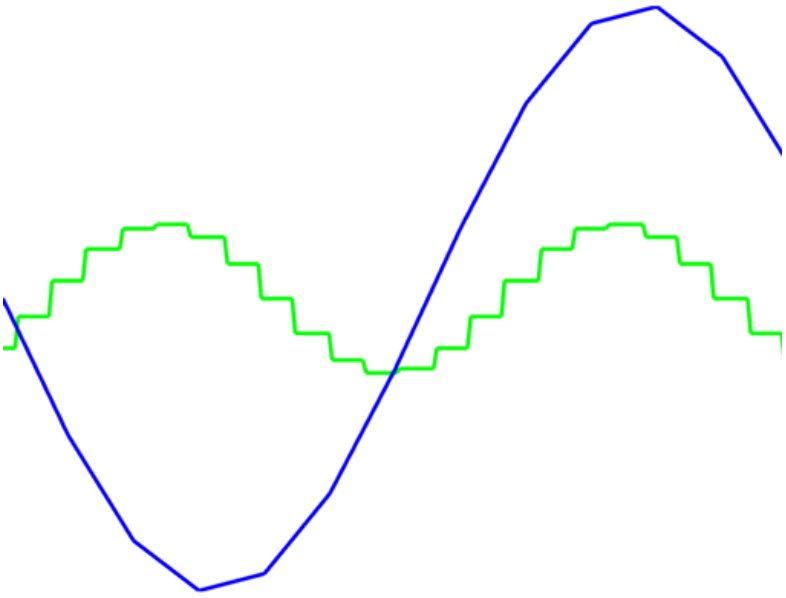


Figure 12.1: Synthesis

A few things to try, if you'd like to explore further:

- Instead of using `keyPressed(...)` to determine the oscillator's frequency, use

ofxMidi to respond to external MIDI messages. If you want to get fancy, try implementing pitch bend or use MIDI CC messages to control the frequency lerp rate.

- Try filling the waveform table with data from an image, or from a live camera (`ofMap(...)` will be handy to keep your data in the -1 to 1 range)
- Implement a *polyphonic* synthesizer. This is one which uses multiple oscillators to let you play more than one note at a time.
- Keep several copies of the `phase` index, and use `ofSignedNoise(...)` to slightly modify the frequency they represent. Add each of the waveforms together in `output`, but average the result by the number of phases you're tracking.

For example:

```
void ofApp::audioOut(float * output, int bufferSize, int nChannels) {
    ofScopedLock waveformLock(waveformMutex);

    float sampleRate = 44100;
    float t = ofGetElapsedTimef();
    float detune = 5;

    for(int phaseIndex = 0; phaseIndex < phases.size();
       phaseIndex++) {
        float phaseFreq = frequency + ofSignedNoise(phaseIndex, t) *
           detune;
        float phaseStep = phaseFreq / sampleRate;

        for(int i = 0; i < bufferSize * nChannels; i += nChannels) {
            phases[phaseIndex] += phaseStep;
            int waveformIndex = (int)(phases[phaseIndex] *
               waveform.size()) % waveform.size();
            output[i] += waveform[waveformIndex] * volume;
        }
    }

    outLine.clear();
    for(int i = 0; i < bufferSize * nChannels; i+= nChannels) {
        output[i] /= phases.size();
        outLine.addVertex(ofMap(i, 0, bufferSize - 1, 0,
           ofGetWidth()),
                          ofMap(output[i], -1, 1, 0, ofGetHeight()));
    }
}
```

12.7 Audio Gotchas

12.7.1 "Popping"

When starting or ending playback of synthesized audio, you should try to quickly fade in / out the buffer, instead of starting or stopping abruptly. If you start playing back a buffer that begins like `[1.0, 0.9, 0.8...]`, the first thing the speaker will do is jump from the "at rest" position of 0 immediately to 1.0. This is a *huge* jump, and will probably result in a "pop" that's quite a bit louder than you were expecting (based on your computer's current volume).

Usually, fading in / out over the course of about 30ms is enough to eliminate these sorts of pops.

If you're getting pops in the middle of your playback, you can diagnose it by trying to find reasons why the sound might be very briefly cutting out (i.e. jumping to 0, resulting in a pop if the waveform was previously at a non-zero value).

12.7.2 "Clipping" / Distortion

If your samples begin to exceed the range of -1 to 1, you'll likely start to hear what's known as "clipping", which generally sounds like a grating, unpleasant distortion. Some audio hardware will handle this gracefully by allowing you a bit of leeway outside of the -1 to 1 range, but others will "clip" your buffers.

[clipped waveform image, reference http://www.st-andrews.ac.uk/~www_pa/Scots_Guide/audio/clipping/fig]

Assuming this isn't your intent, you can generally blame clipping on a misbehaving addition or subtraction in your code. A multiplication of any two numbers between -1 and 1 will always result in another number between -1 and 1.

If you *want* distortion, it's much more common to use a waveshaping algorithm[6] instead of trying to find a way to make clipping sound good.

12.7.3 Latency

No matter what, sound you produce in your app will arrive at the speakers sometime after the event that triggered the sound. The total time of this round trip, from the event to your app to the speakers is referred to as *latency*.

In practice, this usually isn't a big deal unless you're working on something like a musical instrument with very tight reaction time requirements (a drum instrument, for

[6] http://music.columbia.edu/cmc/musicandcomputers/chapter4/04_06.php

instance). If you're finding that your app's sound isn't responsive enough, you can try lowering the buffer size of your ofSoundStream. Be careful, though! The default buffer size is typically the default because it's determined to be the best tradeoff between latency and reliability. If you use a smaller buffer size, you might experience "popping" (as explained above) if your app can't keep up with the extra-strict audio deadlines.

13 Network

by Arturo Castro[1]

corrections by Brannon Dorsey

13.1 TCP vs UDP

TCP and UDP are 2 of the most used protocols to comunicate through a network. Indeed TCP is so common that the suite of protocols on which internet is based is usually called TCP/IP. The network protocols are classified in layers by something called the OSI model[2] TCP and UDP belong to layer 4, the transport layer, and are usually the most used in OF along with protocols from layer 7, the application protocol, like HTTP, FTP or OSC that actually work on top of the other layers, for example HTTP and FTP on top of TCP and OSC usually on top of UDP.

13.1.1 TCP

Transmission Control Protocol, is without doubt the most used network protocol on the Internet, it is a protocol based on a connection, stream based and resistant to errors, package reordering and package lose. Let's see what all that means.

To understand all that we might need to know a bit about how a TCP/IP network works. First of all we need to know that when we send something it's usually divided in packages, each segment of the network might support a different package size so it might subdivide our packages into smaller packages. A package is just a segment of the information we are trying to send plus some headers depending on the protocol we are using. This division in packages is used among other things so it's easier to recover from errors. For example if we are sending a file and we sent it in one go, if some parts of it get corrupted we'll need to resend the full file again. Dividing it in packages and adding some headers to them allows us to detect errors per package so we only need to resend the corrupted packages instead of the whole thing.

When we send a package from one computer to another, even with a connection based protocol like TCP, there's no way of knowing in advance which path it's going to arrive

[1] http://arturocastro.net
[2] http://en.wikipedia.org/wiki/OSI_model

through. On the Internet there's several paths to arrive from one point to another and packages go through whatever path is more optimal at the moment they are sent. But that path might no longer be the ideal path some milliseconds later, so the next package could go through a different route and even arrive before than packages that were sent before.

Another problem is that packages might get corrupted on their way to the destination computer, for example, because of electrical noise in some of the cables.

With all that let's say we send packages ABCD in that order, it might be that at the other end we get something like GCB: package A got corrupted and turned into G, packages B and C arrived ok but in the wrong order, and package D was totally lost.

TCP is able to solve all of those problems. when TCP sends packages it numbers them so that they can be correctly ordered when the other computer recieves them. It also adds something called a CRC to each package that allows the other computer to know if that package is corrupt.

When the destination receives a package and that package is correct, it sends a confirmation, also called an ACK. If after some time the sender hasn't received that confirmation, it sends the package again. Whic solves the problem of corrupted and lost packages.

This ACKs also allows to regulate the speed with which packages are sent so if the clienthas less bandwith than the server, the server can slow down sending packages till it arrives to the speed at which the client can receive them

As we see, using a TCP connection ensures that everything we send is received correctly on the other side.

So why not just always use TCP? Well TCP has some limitations, for example TCP is connection oriented, that means that in order to communicate with another machine we need to open a connection explicitly to that machine and that machine only.

TCP is also stream oriented. That means that we cannot send individual messages and expect them to arrive in one piece, they will arrive eventually but not exactly as we sent them. For example if we send something like:

```
    "Hello world!! this is an openFrameworks network message"
```

On the other side, the application using TCP, we may receive it like:

```
    "Hello w"
    "orld!"
    "!this is a"
    "n openFr"
    "ameworks ne"
    "twork mess"
    "age"
```

We can't even be sure of which size those packages are going to have. There's tricks to send full messages, like adding a delimiter to them, for example openFrameworks when doing:

```
tcpClient.send("Hello␣world!!␣this␣is␣an␣openFrameworks␣network␣
    message");
```

Internally is sending:

```
    "Hello world!! this is an openFrameworks network message[/TCP]\0"
```

The last `\0` is actually added for compatibility reasons with old versions of flash! The `[\TCP]` allows the other side to read the stream until it receives the full message. So when you use:

```
string message = tcpServer.receive();
```

Internally ofxTCPClient/Server will read the stream, keep the partial messages in memory, and wait until it gets the delimiter before returning a full message. This is done transparently, but if you are interfacing with some other protocol coming from a non OF application you might need to do this your self by using `sendRaw()` `sendRawMsg()` `receiveRaw()` and `receiveRawMsg()` which don't send or expect a terminator.

If the protocol you are working with uses TCP and a delimiter, chances are you might be able to use this same trick by using:

```
tcpClient.setMessageDelimiter(myDelimiter);
```

Something important that you might know already: in order to connect applications through a TCP/IP transport network protocol, you usually need, an IP address and a port, the IP address is specific to each machine, the port to each application. So with an IP/port pair we can define an application running in a specific address along all Internet, almost. For example an application that is running a web server usually runs on port 80, if the machine is being executed in has the IP 15.6.8.2, 15.6.8.2:80 defines that web server among any other application running in any other machine in all the Internet. So if we want to connect two machines, usually all we need to know is the IP and port of the application running in the server and use it in the client to connect ot it.

There's an exception though. In most internal networks, like your home network for example, there's a router that connects the machines in that network to the Internet. These routers usually do something called NAT: Network Address Translation. NAT was invented because the IPv4 protocol has a limited number of IP addresses. Internally, the network uses a reserved range of addresses: 192.168.x.x/24 or 10.x.x.x/32, which are addresses that won't be found directly on the internet. When we try to connect to an external address it acts as a kind of proxy between your computer and the server that we want to connect. The router has it's own external address, and when it receives a

response it translates the address and port in which it has received it to an internal one, the one from our computer and sends the packages back to us.

While this is really practical, it means that if we have 2 computers behind NAT routers, it's imposible to open a connection between them (in principle) . There's ways to configure a router to send any package sent to a specific port to the same internal address. There's also libraries like ofxNice[3] that allow you to do NAT transversal, but that will only work using UDP.

13.1.2 UDP

UDP or **User Datagram Protocol**, is a non-connection datagram oriented, non error resistant protocol. It is more or less the total opposite to TCP. We don't need to establish a connection, instead we just send messages to a specific address and port. As long as there's a process listening in that machine and that port it will receive the message.

Datagram oriented means that whatever we send, that fits in the package size supported by the network, by all the subnetworks in the path from one computer to another, will arrive in one piece on the other side. In openFrameworks, if we do:

```
string message = "Hello␣world!!␣this␣is␣an␣openFrameworks␣network␣
    message";
udpManager.SendAll(message, message.size());
```

The other side will receive that message in one piece. That is, if it receives it at all.

As we've said before, UDP is not resistant to errors so the package may not arrive. It may get corrupted and there's no way to recover it. If we send several packages they may arrive in a different order then they were sent. UDP also doesn't adjust for bandwidth, so if one side is sending faster than what the other side can receive, or even if it fills the local buffers while sending, some packages will get lost. What's worse, we won't receive any advice that they got lost, nor in the sender nor in the receiver.

UDP might seem not useful but it has some advantages. Sometimes, we don't mind some packages being lost. For example, if we are sending a package every few frames with the state of our application we don't mind if sometimes we don't receive one of them. It's also really hard to loose a package on a local network, but if you need total reliability don't trust UDP, just use TCP.

Some of the advantages of UDP come from the fact that is connectionless. That means, among other things, that we can broadcast messages to all of the computers in the local network using a broadcast address. To calculate the broadcast address we need to know the IP address of the machine from where we are sending and the subnetwork mask by doing a bit xor on them you get the broadcast address. For example, if the IP address of our machine is 192.168.0.3, and our network mask is 255.255.255.0, the

[3] https://github.com/arturoc/ofxNice

broadcast address will be 192.168.0.255. We can also use multicast if we are working across networks although that's more difficult to setup. We can reuse ports, so we can have more than one process in the same machine using the same port, or use the same port to send and receive...

UDP, as we mentioned before, allows us to do NAT transversal using some clever tricks although is not something that can be done with raw UDP and requires a third party library.

A case where UDP might be preferable over TCP is in very time critical applications, you might have heard that UDP is faster than TCP. That is not exactly true, at least is not that fast to make much difference. The real difference is that when using TCP, if a package gets corrupted or lost the next messages won't get delivered to the application until the lost one is resent so that might introduce a slight delay. In most applications that delay is not noticeable but in some very time critical applications we might prefer to loose some packages than having to wait for them to be resnet. We are usually talking of milliseconds here so as we've said it is usually not a problem.

Another possibility is implementing part of the error recovery in TCP while using UDP, for example we might not mind loosing some packages or getting some of them corrupt but we care about the order in which they arrive, in those cases we can implement package reordering in UDP simply by using a sequence number in each messsage and reorder the packages in the destination by having a buffer so we can wait a little before actually processing a message to see if any other message thath might arrive later needs to be processed before.

In general use TCP if you need your messages to arrive no matter what, when loosing even one package might be critical and UDP if you need some of the most advanced uses like broadcasting, multicasting, NAT transversal, or when waiting for lost packages to be resent, even a few milliseconds, might make be critical for the application while loosing some packages is ok.

13.2 OSC

OSC is an application level protocol. It is of a higher level than UDP or TCP, and it's main characteristic is that it allows to send types like int, float, string... without worrying about the underlying architecture of the sender and receiving machine. It's usually based on UDP so it has some of the same problems.

It's usage in openFrameworks is really simple, so just check the examples in the examples/addons folder to know how it works.

Another advantage of using OSC is that there's lots of commercial and open source projects that support OSC. Using OSC you might be able to easily control some other

software or receive results from it, for example you can have a Pure Data patch to generate audio and control it's parameters from openFrameworks by sending OSC messages to it.

As well as OSC there's other application level protocols for specific applications and it's usually easier to use those than trying to use transport protocols like TCP or UDP. For example streaming video is a really complex problem to solve but there's protocols like RTP that already solve or at least mitigate all the complications that it involves so using a library that implements RTP will be easier than trying to send video directly using TCP or UDP.

14 Advanced graphics

by Arturo Castro[1]

corrections by Brannon Dorsey

14.1 2D, immediate mode vs ofPolyline/ofPath

Traditionally, in frameworks like openFrameworks or processing, the way of drawing things has been something like:

```
void ofApp::draw(){
    ofFill();
    ofSetColor(255,0,0);
    ofBeginShape();
    ofVertex(20,20);
    ofVertex(40,20);
    ofVertex(40,40);
    ofVertex(20,40);
    ofEndShape(true);
}
```

`openFrameworks version`

Which will draw a red square of side 20 at 20,20. For simple primitives like a rectangle we can use `ofRect()`, but if we want to draw more complex shapes the above method is common. This kind of syntax comes from the openGL equivalent:

```
void ofApp::draw(){
    glColor4f(1.0,0.0,0.0,1.0);
    glBegin(GL_TRIANGLE_FAN);
    glVertex(20,20);
    glVertex(40,20);
    glVertex(40,40);
    glVertex(20,40);
    glEnd();
}
```

[1] http://arturocastro.net

`GL version

However, that method is deprecated since OpenGL 3. The openFrameworks version actually does something else. This is because while drawing a rectangle like that works in triangle fan mode, if we try to draw something more complex (mostly any concave shape), it won't work. Because OpenGL only knows how to draw triangles, drawing a concave shape needs one more step called tessellation. The tessellation process involves converting a shape into several triangles before sending to the graphics card.

As we've said the GL syntax is now actually deprecated in openFrameworks if you are using since OpenGL 3, through the programmable renderer, or if you are using openGL ES (Android, iPhone, or ARM Linux in openFrameworks).

The openFrameworks version continues to work but it's ineffective depending on what we are doing. Internally, the openFrameworks version is tessellating the shape, then storing all the triangles in an ofMesh, and then drawing that ofMesh. If you are using openGL 3+ instead of an ofMesh that will be drawn through a VBO using an ofVboMesh, since that's the only possible way of drawing in newer openGL.

Tessellation is kind of slow, but also depending on the number of vertices our shape has it doesn't make much sense to send them to the graphics card every frame. The paradigm that newer versions of openGL use is something like this: create the shape once, upload it to the graphics card, and then draw it every frame without having to reupload again, this is usually don through some kind of buffer in the graphics card, usually vbo's.

In openFrameworks, the ofPolyline and ofPath classes do this in 2D and ofVboMesh for 3D.

14.1.1 ofPolyline

ofPolyline, allows us to represent the contour of a shape. The equivalent to the previous example using a polyline would be something like:

```
//ofApp.h

ofPolyline polyline;

//ofApp.cpp

void ofApp::setup(){
    polyline.lineTo(20,20);
    polyline.lineTo(40,20);
    polyline.lineTo(40,40);
    polyline.lineTo(20,40);
    polyline.close();
}
```

```
void ofApp::draw(){
    polyline.draw();
}
```

Now, instead of calculating the vertices every frame, we are creating them once in setup and drawing them every frame.

However, an ofPolyline still sends its vertices to the graphics card. ofPolyline is really a class meant to be used to do operations over polylines, like simplifications, smoothing... Also ofPolyline can only draw outlines, not filled shapes. ofPath is the recommended way of drawing shapes.

14.1.2 ofPath

ofPath is a complex class internally, it would be the equivalent of an ofImage for 2D geometry. The same way that an ofImage holds a copy of the image in RAM as an ofPixels and a copy in the GPU as an ofTexture, an ofPath contains several representations of the geometry. It's use is simple, and pretty similar to ofPolyline. It follows the same paradigm of creating the shape once and drawing it multiple times:

```
//ofApp.h

ofPath path;

//ofApp.cpp

void ofApp::setup(){
    path.moveTo(20,20);
    path.lineTo(40,20);
    path.lineTo(40,40);
    path.lineTo(20,40);
    path.close();
}

void ofApp::draw(){
    path.draw();
}
```

Unlike ofPolyline, ofPath draws filled shapes by default. As you may have noticed by now, ofFill/NoFill doesn't effect ofPolyline or ofPath. That's because they also follow the more modern openGL paradigm where most global state values are deprecated. For example if you use openGL 3+, `glColor4f` or `glLineWidth` don't exist anymore. Instead, you can set the color per vertex on every shape or use a shader to specify a color or line thickness. We'll see this when tallking about ofMesh.

ofPath allows us to specify if we want to draw it with outline, fill, color, and width per path, so those properties are local to each path instead of specifying them globally:

```
//ofApp.h

ofPath path;

//ofApp.cpp

void ofApp::setup(){
    path.moveTo(20,20);
    path.lineTo(40,20);
    path.lineTo(40,40);
    path.lineTo(20,40);
    path.close();
    path.setStrokeColor(ofColor::blue);
    path.setFillColor(ofColor::red);
    path.setFilled(true);
    path.setStrokeWidth(2);
}

void ofApp::draw(){
    path.draw();
}
```

This avoids several problems, for example when using global colors, a function needs to store the current color, draw something, and then restore the previous color. Storing the color of a shape in the same object makes it easier to draw several things without having to care about the global color or keeping the global state as it was.

Globals in programming are usually a bad idea, and the way openGL has worked untill now was heavily based on globals. Associating every attribute of a shape to the object that represents it solves several problems and is a more object oriented way of doing things.

ofPath packages even more interesting stuff. For example, when we draw a path the first time, ofPath internally calculates it's tessellation and stores it in an ofVboMesh, keeping its vertices in the GPU. If the vertices haven't changed when we draw an ofPath the next time, the vertices don't need to be uploaded again to the graphics card. This makes things really fast.

You can actually access that tessellation using:

```
//ofApp.h

ofPath path;
ofVboMesh tessellation;

//ofApp.cpp
```

```
void ofApp::setup(){
    path.moveTo(20,20);
    path.lineTo(40,20);
    path.lineTo(40,40);
    path.lineTo(20,40);
    path.close();
    path.setStrokeColor(ofColor::blue);
    path.setFillColor(ofColor::red);
    path.setFilled(true);
    path.setStrokeWidth(2);
    tessellation = path.getTessellation();
}

void ofApp::draw(){
    tessellation.drawWireframe();
}
```

The tessellation only represents the fill of our shape. If the path has no fill, it'll return an empty mesh.

We can also access the outlines of an ofPath as a vector of ofPolylines using `path.getOutline()`

> Advanced note: ofPath works with similar API to other formats and libraries for 2D drawing, like SVG, cairo, or nvidia's path rendering openGL extension. That makes it easier to use it not only to draw to the screen using openGL but also to other formats like vectorial formats like PDF or SVG through the cairo renderer in openFrameworks. The use of the cairo renderer is outside of the scope of this chapter, but the important thing to know is that ofPath stores primitives in their original format as ofPath::Commands. Those commands are things like lineTo, bezierTo... and usually end up decomposed in polylines, and later on tesselated if we want to draw them as filled shapes. That's mainly because openGL doesn't know how to decompose things like a bezier into line segments or tessellate a shape but other formats like an SVG or PDF do. When rendering through the cairo renderer ofPath won't decompose or tessellate shapes. Instead it will just send the original primitives to the renderer which will later scale well no matter how big or small we want to show them. Usually we don't need to be aware of this, since openFrameworks will know internally which kind of representation of ofPath it's better to use. If you are working with openGL only there's a flag that can be activated `path.setMode(ofPath::POLYLINES)` which will make that path override the creation of primitives and work directly with ofPolylines which can be slightly faster in certain cases, mostly if you are creating a really high number of paths and modifying them frequently.

14.2 3D

The same way that we have objects to create 2D shapes and draw them later, there's similar classes to work with 3D like ofMesh, ofVboMesh, and of3dPrimitive. Before looking at them, let's see a another topic related to how things are done in the newest versions of openGL and learn the openFrameworks equivalent.

14.2.1 Transformation matrices

If you've worked with processing, openFrameworks or similar frameworks you are probably used to position things in the screen doing something like:

```
ofTranslate(20,20);
ofRotate(45);
ofRect(20,20,20,20);
```

This draws a square rotated 45 degrees around it's top-left corner. Usually you would enclose that between `ofPush`/`PopMatrix` so later drawings won't be affected by the transformations that we've just applied.

This comes from the openGL equivalent:

```
glTranslatef(20,20);
glRotatef(45);
ofRect(20,20,20,20);
```

This is also deprecated since openGL 3. What!? "I can't do use translate/rotate/scale anymore?", you might ask. Well, in openFrameworks you can still use the equivalent `ofTranslate`/`Rotate`/`Scale` if you want, but that has a number of problems and that's why they've been deprecated. Let's see why:

We've seen how, when drawing things in most modern versions of openGL, the paradigm is to create the shape once and then draw it several times with transformations. Each call to ofTranslate, Rotate, Scale, or the gl equivalents for that matter, are doing a multiplication of 4x4 matrices. This is not really that slow, unless you are doing it tons of times. But we can avoid it somehow. Instead of doing all of the multiplications of the matrices every frame, we can use an ofMatrix4x4 for each shape we use, do all of that shape's transformations once (or every time the shape moves), and apply them later when we want to draw that frame:

```
//ofApp.h
ofPath path
ofMatrix4x4 m;

//ofApp.cpp
void ofApp::setup(){
```

```
    path.moveTo(20,20);
    path.lineTo(40,20);
    path.lineTo(40,40);
    path.lineTo(20,40);
    path.close();
    m.rotate(45);
    m.translate(20,20);
}

void ofApp::draw(){
    ofMultMatrix(m);
    path.draw();
}
```

Now we are avoiding 1 matrix multiplications every frame. That's not much really, and probably for something like this is just easier to keep using ofTranslate and ofRotate, but you get the idea. If we have hundreds of transformations, storing them only when they change makes things faster.

Also if we encapsulate each geometry with it's transformation by having objects that contain an ofPath and an ofMatrix4x4 we'll avoid confusing global states. Each shape sets it's transformations before drawing.

In openFrameworks, the classes that apply transformations still return the matrix to it's original state so things will work as before.

If you want to know more about how transformation matrices work you should check out the chapter on mathematics. The purpose of this chapter is not so much to show how they work, but rather the newest paradigms in the latest versions of openGL.

In openFrameworks, there's a utility class called ofNode, that allows you to apply complex transformations like set an object to look to another, set a hierarchy of nodes... When working with 3D it's useful to keep an ofNode along with every mesh that represents it's transformations, so when you draw each mesh, instead of using ofTranslate, rotate, scale you can just apply the transformation of it's node using `node.transformGL()`. This will multiply the current matrix by the one in the node. When you are done you can use `node.restoreTransformGL()`to go back to the previous state.

The most important idea of this section is that when working with complex transformations, instead of using ofTranslate/Rotate/Scale, it is usually easier to implement an ofNode associated to each mesh or shape that you draw. This is also much easier for readability. For meshes, there's a new class in openFrameworks since 0.8.0 called of3dPrimitive, that internally has an ofVboMesh and an ofNode so you can use this pattern in an easy way.

14.2.2 ofCamera

When using openGL, we always have a perspective matrix that affects how 3D objects are projected into the 2d surface, that is the screen, to give appearance of 3D. There's several ways to setup that matrix, but usually we need to know the FOV. The FOV, or field of view, is the angle that the virtual camera, that we are looking through, can see. We also need the near and far clip planes. These define the distance at which things begin and end to be drawn. Finally, we need the width and height of the viewport. All of those parameters define a frustrum, a 6 sides polyhedra that defines the bounding box of things that will appear in the screen as well as how they'll be projected from 3D into 2D.

We also have a second matrix, called the model view, which defines the location of the virtual camera through which we look at the scene. The view matrix is actually the inverse of the matrix that defines the position of the camera, so when we alter it we actually transform the position, rotation and scale of things being drawn. It's this matrix that gets modified by default when we use ofTranslate, ofRotate and ofScale. Again there's more information about this in the maths chapter.

By default, openFrameworks sets a projection matrix with a FOV of 60, width and height of the screen, and clip planes automatically calculated from the other parameters. It then calculates a model view that "moves the virtual camera" back from 0,0 to a position where the top left of the screen matches with 0,0 and the bottom right with width,height.

In openGL however, by default, those matrices are set to the identity matrix which makes the center of the screen (0,0), the top left corner (1,-1) and the bottom right corner (-1,1). You might have noticed that, in openGL, the y coordinate grows upward while in openFrameworks, it grows downwards. This is to avoid confusion and make it easier to work with images or mouse coordinates in openFrameworks, as they also grow downwards. Other libraries we use also follow that convention. Since 0.8.0 you can change that by calling: `ofSetOrientation(OF_ORIENTATION_DEFAULT,false)` being false a false vertical flip so y will grow upwards.

So most of the time, especially when working with 2D, these perspective settings are enough. We can draw things and the coordinates will match nicely with the size of the screen in pixels. When working with 3D though, we might need something more complex, like moving the camera along a scene or changing the field of view... That's what ofCamera allows.

ofCamera is actually an ofNode, so you can do with it anything that you might do with an ofNode. For instance, you can set it to look to another object, or you could add it in a hierarchy of nodes so that when its parent moves it moves relatively to its parent position... the ofNode in the end defines where the camera is and where it's looking at, which in turn is the inverse of the view matrix that will get uploaded to openGL.

On top of that, ofCamera allows you to set a perspective matrix. That's the matrix that

defines how things will be projected to the 2D screen. To use it, usually we set it up like so:

```
//ofApp.h
ofCamera camera;

// ofApp.cpp

void ofApp::setup(){
    camera.setFov(60);  // this will actually do nothing since 60 is
        the default
}

void ofApp::draw(){
    camera.begin();
    // draw something
    camera.end();
}
```

When using an ofCamera, 0,0 will be at the center of the screen, and y will grow upwards. With the default settings, the top,left of the screen will be (-w/2,h/2) and the bottom,right (w/2,-h/2).

As we see in the example to draw things as if they were looked at from the camera we call `camera.begin()` draw them and then call `camera.end()` to stop using that camera and go back to the perspective that openFrameworks sets by default, or whatever we had setup before.

While our application runs, we can tweak the camera parameters in update to move it, make it look at some object, rotate it, or even change the fov which will look like changing the "zoom" of a real camera.

14.2.3 ofMesh

In openFrameworks, the ofMesh class allows us to represent a 3D model. Internally, it's just a bunch of vectors. Each vector represents one mesh attribute. Those attributes are: vertices, colors, texture coordinates and normals. Each mesh should usually have the same number of each of those attributes unless it's not using one of them in which case it'll be empty.

For example to define a mesh that draws a red square we can do:

```
// ofApp.h

ofMesh mesh;

// ofApp.cpp
```

```
void ofApp::setup(){
    mesh.addVertex(ofVec3f(20,20));
    mesh.addColor(ofColor::red);
    mesh.addVertex(ofVec3f(40,20));
    mesh.addColor(ofColor::red);
    mesh.addVertex(ofVec3f(40,40));
    mesh.addColor(ofColor::red);
    mesh.addVertex(ofVec3f(20,40));
    mesh.addColor(ofColor::red);
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
}

void ofApp::draw(){
    mesh.draw();
}
```

or

```
// ofApp.h

ofMesh mesh;

// ofApp.cpp

void ofApp::setup(){
    mesh.addVertex(ofVec3f(20,20));
    mesh.addVertex(ofVec3f(40,20));
    mesh.addVertex(ofVec3f(40,40));
    mesh.addVertex(ofVec3f(20,40));
    mesh.addColor(ofColor::red);
    mesh.addColor(ofColor::red);
    mesh.addColor(ofColor::red);
    mesh.addColor(ofColor::red);
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
}

void ofApp::draw(){
    mesh.draw();
}
```

Remember that the mesh is just several vectors, one per attribute of the vertices so every color we add is applied to the vertex in the same position, that way we can do things like define gradients:

```
// ofApp.h

ofMesh mesh;
```

```
// ofApp.cpp

void ofApp::setup(){
    mesh.addVertex(ofVec3f(20,20));
    mesh.addColor(ofColor::red);
    mesh.addVertex(ofVec3f(40,20));
    mesh.addColor(ofColor::red);
    mesh.addVertex(ofVec3f(40,40));
    mesh.addColor(ofColor::blue);
    mesh.addVertex(ofVec3f(20,40));
    mesh.addColor(ofColor::blue);
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
}

void ofApp::draw(){
    mesh.draw();
}
```

Same goes for texture coordinates and normals. Each of them applies, again, to the vertex in the same position:

```
// ofApp.h

ofMesh mesh;
ofImage img;

// ofApp.cpp

void ofApp::setup(){
    mesh.addVertex(ofVec3f(20,20));
    mesh.addTexCoord(ofVec2f(0,0));
    mesh.addVertex(ofVec3f(40,20));
    mesh.addTexCoord(ofVec2f(20,0));
    mesh.addVertex(ofVec3f(40,40));
    mesh.addTexCoord(ofVec2f(20,20));
    mesh.addVertex(ofVec3f(20,40));
    mesh.addTexCoord(ofVec2f(0,20));
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
    img.loadImage("some20x20img.png");
}

void ofApp::draw(){
    img.bind();
    mesh.draw();
    img.unbind();
}
```

When we add texture coordinates, we probably want to use a texture while drawing that mesh, to use a texture we use `bind()` on an ofImage or ofTexture and call `unbind()`

when we are done using it. We can even draw several meshes that use the same texture by calling bind/unbind once and drawing all of them in between and it's actually recomended since changing the openGL state, the binded texture in this case, it's relatively slow.

We could even combine color and texture tinting the texture with the color we apply to each vertex.

There's more information about how ofMesh works in this tutorial[2].

14.2.4 ofVboMesh

ofVboMesh is a simple class that encapsulates a vbo and inherits from ofMesh. That means that we can use it exactly the same as an ofMesh, that it is actually an ofMesh, but when it's drawn, instead of uploading all the vertices to the graphics card every time call draw on it, it uploads them once when we draw for the first time and only uploads them again if they change. Usually when working with openGL it is advised to use ofVboMesh instead of ofMesh.

There's a case where using an ofVboMesh might be slower, and that's if we want to draw an ofVboMesh, modify it's vertices and then draw it again in the same frame. The problem here is that openGL doesn't really draw things as soon as we tell it to draw. Instead, it stores all the drawing commands and then draws all of them at once and in parallel with the execution of our program. When we try to draw a vbo, modify it's contents and then draw it again in the same frame, openGL would need to really draw the vbo at that exact moment, which means drawing everything else up to that point. That would slow things down a lot. If you need to do something like this, make a copy of the vbo and modify the copy instead of the original. In general don't draw, modify and redraw a vbo in the same frame:

```
// ofApp.h

ofVboMesh mesh;

// ofApp.cpp

void ofApp::setup(){
    mesh.addVertex(ofVec3f(20,20));
    mesh.addVertex(ofVec3f(40,20));
    mesh.addVertex(ofVec3f(40,40));
    mesh.addVertex(ofVec3f(20,40));
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
}

void ofApp::draw(){
```

[2] http://openframeworks.cc/tutorials/graphics/opengl.html

```
    mesh.draw();
    mesh.getVertices()[1].x+=0.1;
    mesh.draw(); // slow!!
}
```

instead do:

```
// ofApp.h

ofVboMesh mesh;
ofVboMesh mesh2;

// ofApp.cpp

void ofApp::setup(){
    mesh.addVertex(ofVec3f(20,20));
    mesh.addVertex(ofVec3f(40,20));
    mesh.addVertex(ofVec3f(40,40));
    mesh.addVertex(ofVec3f(20,40));
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
    mesh2 = mesh;
}

void ofApp::update(){
    mesh.getVertices()[1].x+=0.1;
    mesh2.getVertices()[1].x=mesh.getVertices()[1].x + 0.1;
}

void ofApp::draw(){
    mesh.draw();
    mesh2.draw(); // fast!!
}
```

14.2.5 of3dPrimitive

As we've mentioned before, of3dPrimitive is a helper class that encapsulates an ofVboMesh and inherits from ofNode. You can call any method you would call on an ofNode, because of how inheritance works, it is actually an ofNode so we can change it's position, rotate it, make it look to some other node, add it to a node hierarchy... And when you call `draw` on it, it'll draw the mesh it contains applying the transformation defined by it's node.

There's several predefined 3D primitives, like `ofPlanePrimitive`, `ofSpherePrimitive`, `ofIcoSpherePrimitive` or `ofCylinderPrimitive` which know about the particulars of the geometry of the mesh they contain. This makes it easy to apply textures to it or change the resolution of the mesh...

Or you can create your own using `of3dPrimitive` directly:

```
// ofApp.h

of3dPrimitive primitive;

// ofApp.cpp

void ofApp::setup(){
    primitive.getMesh().addVertex(ofVec3f(20,20));
    primitive.getMesh().addVertex(ofVec3f(40,20));
    primitive.getMesh().addVertex(ofVec3f(40,40));
    primitive.getMesh().addVertex(ofVec3f(20,40));
    primitive.getMesh().setMode(OF_PRIMITIVE_TRIANGLE_FAN);
}

void ofApp::update(){
    primitive.move(ofVec3f(10,0,0));
}

void ofApp::draw(){
    primitive.draw();
}
```

Note: While the example above aims to show how to use of3dPrimitive to create custom geometries while being simple enough to fit in this context, usually is not a good idea to use of3dPrimitive for simple primitives like the one above. Calculating the transformations of an ofNode is kind of expensive in terms of CPU usage. For primitives with lots of vertices it's the way to go, but for something like the previous example it is usually just faster to recalculate all the points in their new position using an ofVboMesh

15 That Math Chapter: From 1D to 4D

by Omer Shapira[1]

NOTE: This chapter is formatted with MD and LaTeX. Github won't render it properly. Try stackedit.io[2] **instead**

15.1 How Artists Approach Math

Math is a curious thing in arts. Many artists reference it directly as inspiration for their work, from Leonardo Da Vinci's *Vitruvian Man*, through Escher's different views of fields of numbers, and many other highlighted, topical representations in art. It is otherwise known as a tool for bringing order into most arts: musicians religiously follow Chromatic Circles (which are just cyclic groups of order 12, $\mathbb{Z}/12\mathbb{Z}$), Architects create rhythms in harmonic series, $\frac{1}{2^n}$ or $\frac{1}{3^n}$, and product designers train their loved ones to wake them up in the middle of the night and ask them questions about The Golden Ratio, $\frac{1+\sqrt{5}}{2}$ (Seriously guys, stop it). But just as it is important for artists to appreciate the order that Mathematics can bring, it is significantly more important to observe the chaos Mathematics contains.

Randomness, events in large scales and unpredictabilty were Mathematical concepts that were inaccessible for rapid exploration until very recently. The concept of *simulation* – letting a thing happen, bound to some conditions, on a massive scale – is something that computers enabled humans to explore. When Benoît Mandelbrot worked for IBM, his attempt at printing the density map of a self-repeating sequence of complex numbers – something that would have taken forever for a human hand and head – resulted in a scientific measurement being reappropriated for its aesthetic value, in what we now call Fractal Art[3]. The same conseqeunce, that people could create a drawing faster than they could think, enabled an entire family of simulation arts, like the tree-like structures generated from L-Systems[4], concieved by Aristid Lindenmayer, and many, many flavors of computer-generated biomimetic artifacts. It's acceptable to call these things *Art*, because the thought that mathematicians had done this deliberately in their work would simply confuse the audience.

[1] http://omershapira.com
[2] http://stackedit.io
[3] https://en.wikipedia.org/wiki/Fractal_art
[4] http://en.wikipedia.org/wiki/L-system

But just by picking up this book, any reader already knows better than to create this distinction. Math is everywhere in Art, just like Art is everywhere in Math. When using a brush or pen or chisel, we're taking advantage of the hard work that nature is doing, calculating physics, rendering things perfectly for us, all in real time. In the computer world, none of that is true. Things like L-Systems had to be created for us to use, because our hands can't reach into the computer. If you're doing any bit of digital art, the math is happening somewhere, whether you control it or not. This chapter will try to expose basic inner-workings of the black arts of graphics programming. I wish I could make this chapter 5 times as long and include the really neat stuff, but that would make this book too heavy to put in a rucksack.

15.2 About this Chapter

A Math chapter for a book about graphics will always miss out on many ideas. In fact, there are entire books covering "math for graphics", mostly consisting of references to other books, focusing on a specific topic (like Linear Algebra, Multivariable Calculus, Differential Geometry, and many other words mysteriously connected to other words). This chapter must therefore be very concise about ideas. All topics here are explained in a friendly way, but please - never fear googling a thing for which you need better examples.

This chapter will be divided into *'numbers of D's'* : we'll start from one dimension, and slowly explore the possibilities enabled by the amount of dimensions we're operating in. We'll explore concepts of *scale* and *change* and learn how much can be described just with these two words. Depending on how you choose to read it, this chapter contains hundreds years of Mathematical research, or in other words, several classes of college math, so it's worth bookmarking.

Note: When bringing math to innocent readers, most programming books will try to explain the idea, not necessarily the exact implementation. This book is no different. This chapter contains detailed breakdowns of concepts, but if you want to find out what's going on under the hood, there's no alternative to reading the source code - in fact, since all the math here is only a few lines long - it's actually *encouraged* to have a look at the source.

15.3 One Dimension: Using Change

Let's start our journey by looking at the number line. It's a stretch of numbers going to infinity in both the positive and negative direction. Suppose we were ants or microbes, so that we could stand on exactly one value here, and travel to any other value by walking in that direction. That's pretty much the definition of a *dimension*. It's an

infinite collection of values that are all accessible, and any value of it can be described with one number. As you're about to see, these properties are going to enable quite a lot of options.

[GRAPHICS: The Number Line. Reference added]

15.3.1 Interpolation

15.3.1.1 Linear Interpolation: The `ofLerp`

```
float ofLerp(float start, float stop, float amt)
```

As Randall Munroe (the author of xkcd) once put it, if you see a number larger than 7 in your page, you're not doing real math. To prove ourselves worthy, this part will only involve numbers between 0 and 1.

Those of you that have already done a little time-based or space-based work have probably noticed that you can often describe elements of your work as sitting on a line between two known points. A frame on a timeline is at a known location between 00:00:00 and the runtime of the film, a scrollbar is pointing to a known location between the beginning and the end of a page. That's exactly what `lerp` does.

With the `lerp` function, you can take any two quantities, in our case `start` and `stop`, and find any point between them, using amounts (`amt`) between 0 and 1. To be verbose:

$$\text{lerp}\,(a, b, t) = t \cdot b + (1 - t) \cdot a$$

15.3.1.1.1 Note: What does *linear* really mean? Engineers, Programmers and English Speakers like to think of *linear* as *anything you can put on a line*. Mathematicians, having to deal with all the conceptual mess the former group of people creates, define it *anything you can put on a line* ***that begins at (0,0)***. There's good reasoning behind that, which we will see in the discussion about Linear Algebra. In the meantime, think of it this way:

> A *Linear Transform* takes any line that has a value 0 at the point 0 and returns a line with the same property, $f(x) = ax$. If it returns a line value different from 0 at $x = 0$, $f(x) = ax + b$, it's an *Affine Transform* instead.

At this point you probably know, but it's worth repeating: Those two transformations may either change lines into lines, or in some degenerate cases, lines to points. For example, $f(x) = x^2$ is totally not linear.

15.3.1.1.2 Exercise: Save NASA's Mars Lander In 1999, an masterpiece of engineering was making its final approach to Mars. All instruments were showing that the approach distance matched the speed, and that it's just about to get there and do some science. But instead, it did something rather rude: it crashed into the red planet. An investigation made later by NASA revealed that while designing the lander, one team worked with their test equipment set to *centimetres*, while the other had theirs set to *inches*. **By the way, this is all true.**

Help the NASA teams work together: write a function that converts centimetres to inches. For reference, $1_{\text{in}} = 2.54_{\text{cm}}$. Test your result against three different real-world values. Tip: Its much easier to start solving with pen and paper than it is with a keyboard.

Think:

1. Why can we use `lerp` outside the range of 0 and 1?
2. What would it take to write a function that converts inches into centimetres?

15.3.1.2 Affine Mapping: The `ofMap`

```
float ofMap(float value, float inputMin, float inputMax, float
    outputMin, float outputMax, bool clamp = false)
```

In the last discussion, we saw how by using `lerp`, any value between two points can be *linearly* addressed as a value between 0 and 1. That's very convenient, and therefore the reason we build most of our arbitrary numerical ranges (`ofFloatColor`, for example) in the domain of 0 and 1.

However, when dealing with real world problems, programmers run into domains of values that they wish to map to other ranges of values, neither of which are confined to 0 and 1. For example, someone trying to convert the temperature in Celsius to Fahrenheit won't be able to use a `lerp` by itself - the domain we care about isn't between 0 and 1. Surely, the way of doing that must involve a `lerp`, but it needs a little help.

If we want to use the `lerp` function, we're aiming to get it to the range between 0 and 1. We can do that by knocking `inputMin` off the input `value` so that it starts at 0, then dividing by the size of the domain:

$$x = \frac{\text{value} - \text{inputMin}}{\text{inputMax} - \text{inputMin}}$$

Now that we've tamed the input domain to be between 0 and 1, we do the exact opposite to the output: `ofMap(value, inputMin, inputMax, outputMin, outputMax)` $= \frac{\text{value}-\text{inputMin}}{\text{inputMax}-\text{inputMin}} \cdot (\text{outputMax} - \text{outputMin}) + \text{outputMin}$

Here's an example. Let's say we're given a dataset in Farenheit. Farenheit sets 0 to be the freezing point of brine and 100 to be the body temperature of a slightly ill British human (duh?). In order to do *anything* with that, we first need to convert that to Celsius, which at least uses *The Same Damn Substance™* for 0 and 100: Water. Now, we happen to know that water freezes at 32_{f} and boils at 212_{f}, so we have the same exact objective range, now it's time to map. We'll use an array for this:

```
vector<float> farenheitValues;
// we'll skip over the code that fills them up, as somebody else has
   done it
vector<float> celsiusValues; //Sets up an empty C++ vector, which is
   a dynamic array
for (int i = 0 ; i < farenheitValues.size() ; ++i){
    celsiusValues.pushBack(ofMap(32, 212, 0, 100,
       farenheitValues[i]));
}
```

Watch what we did here. We took the *domain* of 32 to 212 and converted it to a *range* of 0 to 100. There are two things to note about that:

- In Mathematics, we often use the words *domain* and *range* as origin and target. Using those terms allows us to introduce another concept we care about: *Separation of Concern*. If we know that every input a function takes is guaranteed to be in a certain *domain*, we can engineer it so it guarantees an output in a certain *range*, and make sure it doesn't fail. In fact, this is the mathematical definition of a function:

  A Function is a Mathematical object that maps *every* value of a certain domain to a *single* value of a certain range.

- We defined the range of 32 to 212 as two points we know on a line. The actual range of temperatures is -459.67 (the absolute zero, in Farenheits) to somewhere very, very large (known as the planck temperature) - it's not very conventient to calculate that. So instead of choosing the whoe range, we mapped a known area of it to a known area of it in the range. We are allowed to use an `ofMap()` for that, because the scale is linear. Some scales are not linear; For example, the decibel (dB), commonly used in sound measurement, is logarithmic, so converting between a range of 0_{dB} - 6_{dB} to 6_{dB}-15_{dB} would not convey any meaning.

15.3.1.3 Range Utilities

15.3.1.3.1 Clamping You'll notice that the previous explanation is missing the `clamp` parameter. This may not matter to us if we're using the `ofMap` function in the range that we defined, but suppose we select a `value` smaller than `inputMin`: would it be ok if the result was also smaller than `outputMin`? If our program is telling an elevator

which floor to go to, that might be a problem. That's why we add `true` to the tail of this function whenever we need to be careful.

Just in case, oF offers another specific clamp function:

```
float ofClamp(float value, float min, float max)
```

15.3.1.3.2 Range Checking Two important functions we unjustly left out of this chapter:

```
bool ofInRange(float t, float min, float max);
```

Tells you whether a number `t` is between `min` and `max`.

```
float ofSign(float n);
```

Returns the sign of a number, as `-1.0` or `1.0`. Simple, eh?

15.3.2 Beyond Linear: Changing Change

[mh: I recognize that you are trying to be general here by talking about change, but at least throwing the word motion around as a type of change would give readers something upon which to anchor the concept.]

So far we've discussed change that is bound to a line. But in Real Life™ there's more than just straight lines: For one, we can't even describe periodic events with straight lines If we need to describe the vibration of a guitar string [footnote: this example sounds kinda old. Of course I meant "the wobble of a dubstep instrument"] or the changing speed of a biliiard ball after impact, we're going to need to use higher orders of change.

In this discussion, we're about to see how we can describe higher orders of complexity, via a cunning use of `lerp`s. You will see that some types of change can be reproduced this way (like that billiard ball) - while other types of motion, like harmonic motion, will need a separate mechanism. Keep in mind that some of the code here is conceptual, not necessarily efficient.

15.3.2.1 Quadratic and Cubic Change Rates

Consider this function:

```
float quadratic (float t){
    float a1 = ofLerp(t, 5, 8);
    float a2 = ofLerp(t, 2, 9);
```

```
    float b = ofLerp(t, a1, a2);
    return b;
    }
```

This function used a defined range and a parameter to create `a1`, then used another defined range with the *same* parameter to create `a2`. Their result looks surprising:

[GRAPHIC: QuadraticSpline.pdf to be processed]

We've done something remarkable here. We used the way one parameter changes on two fixed lines to control a third, totally mobile line, and draw one point on it at each point in time between 0 and 1. In Mathspeak, it looks like this:

$$\begin{aligned}
&\text{lerp}\left(t, \text{lerp}\left(t, 5, 8\right), \text{lerp}\left(t, 2, 9\right)\right) = \\
&\text{lerp}\left(t, 8 \cdot t + 5 \cdot (1-t), 9 \cdot t + 2 \cdot (1-t)\right) \\
&\quad = (9 \cdot t + 2 \cdot (1-t)) \cdot t + (8 \cdot t + 5 \cdot (1-t)) \cdot (1-t) \\
&\quad = \left(9t^2 + 2t - 2t^2\right) + (8t + 5 - 5t) - \left(8t^2 + 5t - 5t^2\right) \\
&\quad = 4t^2 + 5
\end{aligned}$$

Something interesting happened here. Without noticing, we introduced a second order of complexity, a *quadratic* one. Seriously, give it a second look, draw the entire process on paper. It's remarkable.

The same manipulation can be applied for a third order:

```
float cubic (float t){
    float a1 = ofLerp(t, 5, 8);
    float a2 = ofLerp(t, 2, 9);
    float a3 = ofLerp(t, 3, -11);
    float a4 = ofLerp(t, -2, 4);
    float b1 = ofLerp(t, a1, a2);
    float b2 = ofLerp(t, a3, a4);
    float c = ofLerp(t, b1, b2);
    return c;
    }
```

We'll skip the entire solution, and just reveal that the result will appear in the form of

$$at^3 + bt^2 + ct + d$$

See the pattern here? The highest exponent is the number of successive `ofLerp`s we applied, i.e. the number of times we nested the `lerp` function in itself.

15.3.2.1.1 ...And So On The general notion in Uni level Calculus is that *you can do anything if you have enough of something*. So fittingly, there's a curious little idea

in Mathematics which allows us, with enough of these nested control points, to approximate any curve segment we can imagine. In the original formulation of that idea (called a *Taylor Series*), we only reach a good approximation if the amount of degrees (successive `lerp`s we applied) is close to infinity.

In Computer Graphics, as you're about to see - 3 is close enough.

15.3.3 Splines

What we've done in the previous chapter is really quite remarkable. We have built a rig of points, on top of which we built a rig for controlling these points in pairs, and we continued to do so until we ended up with one parameter, t, to control them all, subsequently controlling the process of drawing. In the domain we defined all of that to happen, we can clearly make this a physical metaphor: for example, a bunch of articulating sliderules connected to eachother at a certain point. However, for reasons you're about to see, Mathematicians will often shy away from the description of polynomials as a physical metaphor.

The reason is what happens to polynomials soon after they step away from their engineered control points. Outside the range of control, every polynomial will eventually go to infinity - which is a broad term, but for us designers it means that slightly off it's range, we'll need a lot more paper, or computer screen real estate, or yarn, or cockroaches (true story[5]) in order to draw it.

[GRAPHICS: PolynomialToInfinity.pdf, to be processed]

So instead of using polynomials the way they are, some mathematicians thought of a clever thing to do: use only the good range (one that's between the control points), wait for the polynomial to do something we don't like (like turn from positive to negative), **then mix it with another polynomial**. That actually works pretty well:

[GRAPHICS: Spline.ai - Please talk to me before processing this]

In the illustration, we've taken a few parts of the same cubic (3rd degree) polynomial, moved it around and scaled it to taste, and added all of them together at each point (let's call it 'mixing').

The resulting curve is seamless and easy to deal with. It also carries some sweet properties: using it, one can use the absolute minimum of direction changes to draw any cubic polynomial between any two points. **[mh: maybe add another sentence here to unpack this]** In other words, *it's smooth*.

These properties make this way of creating curves pretty popular in computer graphics, and you may find its variants under different names, like *Beziér Curves* or Spline Curves. The code for implementing this is a little long and tedious in C++, so this chapter won't go into it - but in case you were wondering, it's just the same code for making

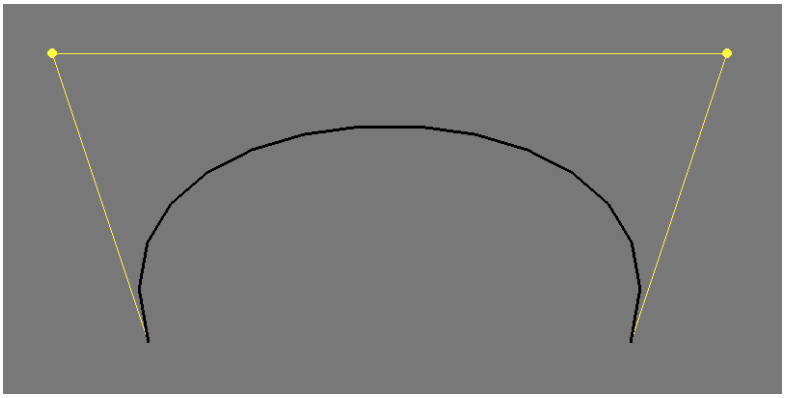
[5] http://www.andrewcerrito.com/itpblog/itp-winter-show-nyc-food-crawl/

polynomials we discussed above, only with a lot of `if` statements to check if `t` is in the correct range.

Using the curve functions in openFrameworks is pretty straightforward: All you have to do is start from a point, and then add a destination, along with the control points **[mh: worth defining a control point somewhere in here]** to reach it:

```
//The beginning point
line.addVertex(ofPoint(200, 400));
//A sequence of two control points and a destination:
//control 1's x and y, control 2's x and y, and the destination
line.bezierTo(100, 100, 800, 100, 700, 400);
```

This generates this image:



This is just one example of use though. All of the different combinations are documented extensively in the *Advanced Graphics* chapter.

15.3.4 Tweening

So far we've learned how to use a bunch of control points to create an interesting curve in space. We've used our parameter t to simulate the time it takes to draw each point on the curve. We also implicitly assumed that time runs only in the period that we let it run in, in the case of our examples, between 0 and 1. But we never really tested what it looks like to run all of that in real time.

[GRAPHICS: Smoothstep.ai to be processed] The smoothstep function, $f(x) = 3x^2 - 2x^3$, used to create a smooth transition between 0 and 1

Apparently, it's not that different from running it through space. By looking at the x dimension of this graph as time and the y dimension of this graph as a value for our creation, we can see some patterns of animation forming. The trick is simple: think of all of the values that need to change in your animation, and control them using functions.

Tweening is not yet a standard part of the openFrameworks library. In the meantime, some nice utility functions for tweening are available in the ofxTween library.

15.3.4.1 Other Types of Change

This chapter provides examples of rates of change achievable by using linear change in succession. But there's so much more to rates of change that can't fit here - for example, trigonometric functions (sines and cosines) can only be approximated using polynomials, but in order to actually describe them, we need to get into a discussion about *frequency*. The topic of *change* is so broad that there's not a single branch of mathematics that deals with it – it encompases most of those I can think of. In the Sound chapter, you'll learn a litte more about dealing with frequency. Until then, you've already acquired a toolset that allows you to do a lot.

15.4 More Dimensions: Some Linear Algebra

Until now, we explored several ideas on how to change what's going on the number line. That's cool, but we want to know how to do graphics, and graphics has more than one dimension. Our ancient Mathematician ancestors (Just kidding, most important Mathematicians die before 30. Not kidding) also faced this problem when trying to address the space of shapes and structures, and invented some complex machinery to do so. The fancy name for this machinery is *Linear Algebra*, which is exactly what it sounds like: using algebraic operations (add and multiply, mostly), in order to control many lines.

In this part you're going to learn many concepts in how to store and manipulate multidimensional information. You'll later be able to use that information to control realtime 3d graphics using OpenGL, and impress the opposite (or same) sex with your mastery of geometry.

15.4.1 The Vector

You may have heard of vectors before when discussing directions or position, and after understanding that they can represent both, may have gotten a little confused. Here's the truth about Vectors™:

> A vector is just an array that stores multiple pieces of the same type of information.

Seriously, that's all it is. Quit hiding.

This simplicity is also their great power. Just like the number 5 can be used to describe five Kilometres, the result of subtracting 12 and 7, or the number of cookies in a jar - the same works with vectors.

It's up to the user of that mathematical object to choose what it is used as. The vector

$$v = \begin{bmatrix} 5 \\ -3 \\ 1 \end{bmatrix}$$

can represent a point in space, a direction of a moving object, a force applied to your game character, or just three numbers. And just like with numbers, algebraic operations such as addition and multiplication may be applied to vectors.

Oh, but there's a catch. You see, everyone was taught what $a + b$ means. In order to go on with vectors, we need to define that.

15.4.1.1 Vector Algebra

Generally speaking, when dealing with Algebra of numerical structures that aren't numbers, we need to pay close attention to the *type* of things we're cooking together. In the case of vectors, we'll make a distinction between *per-component* and *per-vector* operations.

15.4.1.1.1 Scalar Multiplication The product between a vector and a scalar is defined as:

$$a \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az \end{bmatrix}$$

That falls into the category of *per-vector* operations, because the entire vector undergoes the same operation. Note that this operation is just a scaling.

```
ofVec3f a(1,2,3);
cout << ofToString( a * 2 ) << endl;
//prints (2,4,6)
```

15.4.1.1.2 Vector Addition Adding vectors is pretty straightforward: it's a *per-component* operation:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{bmatrix}$$

```
ofVec3f a(10,20,30);
ofVec3f b(4,5,6);
cout << ofToString( a + b ) << endl;
//prints (14,25,36)
```

15.4.1.1.2.1 Example: `ofVec2f` as position Vector addition serves many simple roles. In this example, we're trying to track our friend Lars as he makes his way home from a pub. Trouble is, Lars is a little drunk. He knows he lives south of the pub, so he ventures south; But since he can't walk straight, he might end up somewhere else.

```
/* in testApp.h: */
ofVec2f larsPosition;
void larsStep(ofVec2f direction);

/* in testApp.cpp: */
void testApp::setup(){
    larsPosition = ofVec2f( ofGetWidth() / 2., ofGetHeight() / 3. );
}

void testApp::update(){
    if (larsPosition.y < ofGetHeight * 2. / 3.){
        //As Lars attempts to take one step south,
        //He also deviates a little to the right,
        //or a little to the left.
        ofVec2f nextStep(ofRandom(-1.,1.),1.);
        larsStep(nextStep);
    }

void ofApp::larsStep(ofVec2f direction){
    position += direction;
}

void testApp::draw(){
    //Draw Lars any way you want. No one's judging you.
}
```

15.4.1.1.3 Note: C++ Operator Overloading [mh: this proke the flor a bit for me, so I'd recommend pushing it later]

Just like we had to define the meaning of a product of a scalar quantity and a vector, programming languages - working with abstract representations of mathematical objects, also need to have definitions of such an operation built in. C++ takes special care of these cases, using a feature called *Operator Overloading*: defining the `*` operation to accept a scalar quantity and a vector as left-hand side and right-hand side arguments:

```
ofVec3f operator*( float f, const ofVec3f& vec ) {
    return ofVec3f( f*vec.x, f*vec.y, f*vec.z );
}
```

The same is defined, for example, between two instances of `ofVec3f`:

```
ofVec3f ofVec3f::operator+( const ofVec3f& pnt ) const {
    return ofVec3f( x+pnt.x, y+pnt.y, z+pnt.z );
}
```

naturally representing the idea of vector addition.

The basic arithmetic operations, `+`, `-`, `*`, `/`,`+=`, `-=`, `*=`, `/=`, exist for both combinations of `ofVec2f`, `ofVec3f` and `ofVec4f`s and between any vector object and a scalar quantity. Whenever an operation is postfixed with `=`, it modifies the left-hand side with the operation, *and only then*. The operations `+`,`-`,`*`,`/` will *always* return a copy.

Some excellent examples of operator overloading done right exist in the source files for the `ofVec` types. It's encouraged to check them out.

Warning: Overloading operators will make you go blind. Programmers use operators without checking what they do, so bugs resulting from bad overloads take a long time to catch. If the expression `a + b` returns a reference instead of a copy, a `null` instead of a value, or modifies one of the input values – someone will use it one day, and that someone will cry for many days. Unless the operator can do one arithmetic thing and that alone, do not overload it with a different meaning. openFrameworks may or may not have a feature that tweets for you whenever you've written a silly operator overload. No one knows[6].

15.4.1.1.4 Distance Between Points

```
float ofVec3f::distance( const ofVec3f& pnt) const
float ofVec3f::squareDistance( const ofVec3f& pnt ) const
float ofVec3f::length() const
float ofDist(float x1, float y1, float x2, float y2);
float ofDistSquared(float x1, float y1, float x2, float y2);
```

Let's start by a definition. You may remember the *Pythagorean Theorem*, stating what the length of a line between point a and b is:

$$\text{Distance}\left(\begin{bmatrix} x_a \\ y_a \end{bmatrix}, \begin{bmatrix} x_b \\ y_b \end{bmatrix}\right) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

Here's the good news: It's the exact same definition in three dimensions! just add the

[6] https://code.google.com/p/keytweeter/

z term.

$$\text{Distance}\left(\begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix}, \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix}\right) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

Vector Length, then, can be naturally defined as the distance between the vector and the point $(0, 0, 0)$:

$$\text{Length}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \sqrt{x^2 + y^2 + z^2}$$

And that's exactly what using `.length()` as a property of any `ofVec` will give you.

15.4.1.1.5 Vector Products: There's More Than One So you're multiplying two numbers. Simple, right? Five million and seven times three equals something you know. Even if you need a calculator for the result, you still know *it's a number* that's not the case with vectors. If we just want to resize vectors (the way we do with numbers), we multiply a vector by a scalar and it grows. But what does it mean, geometrically, to multiply by a vector?

If we were to follow the *per-component* convention that we created, we would get an operation like this:

```
cout << ofToString(ofVec3f(1,2,3) * ofVec3f(1,2,3)) << endl;
//prints (1,4,9)
```

It's also known as the *Hadamard product*. It's intuitive, but not particularly useful. One case it is useful for is if we want to scale something individually in every dimension.

In the next section we describe something more helpful.

15.4.1.1.6 The Dot Product

```
float ofVec3f::dot( const ofVec3f& vec )
```

The dot product of two vectors has a definition that's not too clear at first. On one hand, the operation can be defined as

$$v_a \bullet v_b = x_a \cdot x_b + y_a \cdot y_b + z_a \cdot z_b$$

which is really easy to implement (in fact, graphics cards have special circuitry for doing just that!). On the other hand, it can also bet defined as

$$v_a \bullet v_b = \|v_a\| \cdot \|v_b\| \cdot \cos\theta$$

where θ is the angle between the two vectors. Soon you'll see that this is a rather lucky coincidence. In the meantime, here's how you *shoud* remember dot products:

A dot product of a and b reflects how one vector projects in the other vector's direction.

Hold it. That's not the end of the story. As you can see, the $\|v_a\| \cdot \|v_b\|$ part of $\|v_a\| \cdot \|v_b\| \cdot \cos\theta$ should tell you that both vectors' lengths have equal parts in determining the final size of the thing, but in most practical cases, you'll be using dot products to determine either vector length or angles between vectors.

That's why dot products are such an amazing coincidence: If you know the lengths of v_a and v_b, you're given $\cos\theta$ for free. If you know the plane on which v_a and v_b lie, one vector and the angle to the other, you get the other one for cheap, and so on. In typical use, if we were to take two vectors that each have length 1 (*normalized* vectors, in Mathspeak), the dot product ab would basically a cosine of the angle between them. That relationship, described by $\cos\theta$, is easy to think of as a projection: Imagine shining a light from the top of one axis, and observing the shadow on another axis. How long it is, and which direction it's going, is exactly consistent with the dot product (in fact, most lighting models use dot products for just about everything).

15.4.1.1.7 Example: Finding out if a point is above or below a plane This is a problem we'll often run into in 3D graphics: given a point p and a plane, we need to figure out which side of the plane the point is on. This may be really useful if you're trying to decide what *not* to render in a scene, in order to save processing time.

There are many equivalent ways to describe a plane. The most elegant one in this case is by using the plane's normal (the direction perpendicular to both axis of the plane) which is a vector we'll mark as n, and the distance from the origin, d (note that because the plane can pass below the origin, this distance can be negative). This is a valid definition: a plane can be defined as all of the points in the world that form a perpendicular vector to the normal.

Now the math:

If the point p is on the plane. We know that every line on the plane is perpendicular to (has a 90-degree angle with) the normal. Specifically, every line connecting some point on a plane to the point where we put our normal (which is the same on the plane) so if we extract the direction vector from the line and call it v, we can say:

$$nv = \|n\| \ \|v\| \ \cos 90^\circ = \|n\| \ \|v\| \, 0 = 0$$

.

If the point p isn't on a plane. In that case we know that it definitely doesn't have a 90-degree angle with the plane's normal n, therefore the dot product $n \bullet v$ won't be zero. So all we need to know is: does it project on the normal's positive direction, or it's negative direction? In order to do that, we first find a point on the plane. That's easy, we defined our plane such that we can follow the normal n from the origin $(0, 0, 0)$ for a length d and we'll get there.

Therefore the point $q = dn$ is most definitely on the plane.

Now, let's make up a vector from the point p to that point: $v = q - p$. This equation holds because when subtracting two points, we get the difference between them, hence the direction from p to q. Now we calculate:

$$vn = v_x n_x + v_y n_y + v_z n_z$$

- *If the dot product is positive*, the normal and the line to a point on the plane are both pointing in the same direction, that means that the point p is *below* the plane.
- *If the dot product is negative*, the line from the point to the plane has to go back, therefore the point is above the plane.

Here's the code:

```
//we define a margin for numerical error
float const EPSILON = 0.00001;

// a function that returns -1 if a point is below a plane,
// 1 if it's above a plane and 0 if it's on a plane
int whichSide(ofVec3f planeNormal, float planeDistance, ofVec3f
    testPoint){
    ofVec3f directionToPlane = planeDistance * planeNormal -
        testPoint;
    float dot = directionToPlane.dot(planeNormal);
    if (abs(dot) < EPSILON){ //Check if the dot product is very near
        zero
        return 0;
    } else { // else return the opposite of its sign!
    return (dot < 0 ? 1 : -1);
    }
}
```

Note that in the code we had to take care of numerical inaccuracy of the computer, and give it a margin for deciding that a vector is sometimes perpendicular to that normal. An alternative would be to ignore those cases, and chose that anything with `dot > 0` is below the plane. I like it better this way.

As you can see from the example, the dot product is magical: just from knowing the coordinates, we get an insight about the angle between vectors for free.

15.4.2 The Matrix™

In the computer world, a program needs the two things to function: Algorithms and Data Structures (it also needs I/O, but we're talking about computation, not engineering). In the 3D Maths world it's exactly the same: we call our data structures 'vectors' and our algorithms are operations.

At the core of the heavy machinery built to control 3d space, a matrix is just a data structure, like a vector. However, the 'algorithms' applied to this data structure (operations, in Mathland) make it an extremely powerful one. All of the *affine* operations we care about in 3D can be described in the form of a matrix: translation, rotation, scaling, inversion, squeezing, shearing, projection and more and more. Here's a simple way to remember this:

> A Matrix is a mathematical object that stores a geometric transformation of points.

Notation Convention: When dealing with matrices, most authors usually mark vectors with lowercase letters and matrices with uppercase letters.

15.4.2.1 Matrix Multiplication as a dot product

The easiest way to look at a matrix is to look at it as a bunch of vectors. Depending on what we care about, we can either look at the columns or rows as vectors.

//TODO: Draw 2x2 example

15.4.2.1.1 Identity Let's start from the simplest case. Just like with numbers, it is a very important property of any algebraic structure to have a *neutral* member for each operation. For example, in Numberland, multiplication of any x by 1 returns x, same goes for addition to 0. In Matrixland, that identity element is a matrix with 1s along the diagonal zeroes elsewhere. For example, the identity matrix for 3 dimensions is:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

So for any matrix M,

$$MI = IM = M$$

.

Note: You may have noticed that we're very careful right now with the order of multiplications, like when we took extra care to describe that $MI = IM$. There's a good reason for that, and you'll discover it in a few pages.

15.4.2.1.2 Scale You might remember that when scaling a vector (i.e point in space and/or velocity and/or force and/or brightness value for a colour, etc), we may choose to scale it uniformly by scalar multiplication:

```
ofVec3f v(1, 2, 3);
cout << ofToSting(v * 5) << endl; //prints (5, 10, 15)
```

or, because of a weird language design choice, most graphics applications will allow you to scale non-uniformly on a per-component basis:

```
ofVec3f v1(1, 2, 3);
ofVec3f v2(10, 100, 1000);
cout << ofToString(v1 * v2) << endl //prints (10, 200, 3000)
```

To put an end to this insanity, scaling in matrix multiplication is well-defined [footnote: "well-defined" is not just a compliment, it's a Mathematical term that essentially means that if you follow the rules, the computation will never *crash*, so to speak – it will never produce results outside the form you defined. For example, dividing by zero is undefined, therefore *division is not well defined on a domain of real numbers that contains zero*. Somebody closed that loop and defined division by zero to be exactly ∞, so for any range that contains infinity, *division is well defined*. You'll find the same issue in programming languages: some operations produce incompatible results. Many criticize C++ for allowing that to happen too often, but other languages (such as Haskell, Erlang and OCaml) are more strict about only using well-defined operations, so the compiler is able to catch your errors, not the end user.] in openFrameworks (also in math!). It goes like this: The matrix S that scales $(x, y, z)^T$ to $(ax, by, cz)^T$ is:

$$S \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax \\ by \\ cz \end{bmatrix}$$

There's logic behind this. Recall that a vector multiplied by a matrix, $M \cdot v$ is just a collection of dot products:

$$M \cdot v = \begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix} \cdot v = \begin{bmatrix} M_1 \cdot v \\ M_2 \cdot v \\ M_3 \cdot v \end{bmatrix}$$

So, in order to get a multiplication through that only affects x, we tune the vector (upper row of the matrix) M_1 to be zero anywhere but the interface with x:

$$M_1 = (a, 0, 0)$$

so the entire calculation would be:

$$M \cdot v = \begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix} \cdot v = \begin{bmatrix} a & 0 & 0 \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix} \cdot v = \begin{bmatrix} a \cdot v_x + 0 \cdot v_y + 0 \cdot v_z \\ M_{2,1} \cdot v_x + M_{2,2} \cdot v_y + M_{2,3} \cdot v_z \\ M_{3,1} \cdot v_x + M_{3,2} \cdot v_y + M_{3,3} \cdot v_z \end{bmatrix}$$

Making the x component of the resulting vector be $a \cdot v_x$, just as promised.

Scalar multiplication of any matrix M becomes really easy, then: it's essentially right multiplication by a diagonal matrix full of a's:

$$a \cdot M = a \cdot I \cdot M$$

15.4.2.1.3 Skew matrices [mh: this could use an image]

Odd operations like skewing are where things need to get a little less intuitive and more logical. When we think of *skewing* we normally imagine adding to a certain dimension, suppose x, a proportion of a quantity from another dimension, let's say y. Suppose that proportion is some $0 < a \leq 1$, as if to say, 'I want to nudge it a little, the more it leaves the ground'. The matrix for doing that in 2 dimensions would look like this:

$$S = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix}$$

See what we did there? We made the resulting x value depend on the input x value (being multiplied by 1), but also slightly depend on the input y, exactly how slightly being determined by the magnitude of a:

$$S \cdot v = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \cdot x + a \cdot y \\ 0 \cdot x + 1 \cdot y \end{bmatrix} = \begin{bmatrix} x + ay \\ y \end{bmatrix}$$

Pretty neat, right? Try to remember this trick, as we're going to use it quite a lot when we move stuff around in the fourth dimension. I'm not even joking.

15.4.2.1.4 Rotation matrices We now see that any operation in Matrixland can really be expressed in a collection of vectors. We also know that dot products of vectors express *the angle between two vectors times their magnitude*. A slightly surprising fact is that those two properties are enough to describe any rotation.

In order to comprehend this last statement, let's first explain how rotating one vector works. Let's take the vector

$$v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

I picked a vector that coincides entirely with the x axis. Now, suppose that we would like to rotate the vector by an angle θ, starting from the origin. Using our knowledge about the unit circle[7], we can describe the rotated vector as

$$v_\theta = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

[7] http://en.wikipedia.org/wiki/Unit_circle

Now we found a target for the x axis to go to. Using the same rotation, let's try to find where the old y axis (the vector $u_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$), we only need to know the angle between them. Luckily, we all know that it's 90 degrees, or in radians: $\frac{\pi}{2}$. The new home will then have to be at angle $\theta + \frac{\pi}{2}$ from the x axis (angle 0):

$$u_\theta = \begin{bmatrix} \cos\left(\theta + \frac{\pi}{2}\right) \\ \sin\left(\theta + \frac{\pi}{2}\right) \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$$

That last equality is due to trigonometric equalities.

15.4.2.1.4.1 2D Rotation Matrices We now have all of the information we need to build a matrix that moves the vectors $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$ to $\left\{ \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}, \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \right\}$:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Now, hold on. Check out what we did here: we placed the targets for the source vectors as *columns* in the matrix, and then we took the resulting *rows* of the matrix to do the rotation. Why did we do that?

Recall that a matrix is just a stack of dot products. How did we construct these dot products? We just aligned all of the entries that should be affecting the *resulting* entry in one row of the matrix. That means that when considering the resulting y entry, our vectors defined *the mixture of y components* from the target vectors that we would like to see in the resulting operation. This makes sense: Think of the vectors that compose the matrix as a new coordinate system, and what we're calculating is how the 'natural' coordinate system is projected onto them.

15.4.2.1.5 3D Rotation Matrices In *Flatland*, Edwin A. Abbott describes a land in which two-dimensional beings live obliviously and happily, until one of them encounters a three-dimensional thing. His mind is boggled – he quickly understands the implications; When he tries to explain it to the other squares living in his world, they are apalled and he is cast away as a heretic.

Rotation in three dimensions is more complex to understand than rotations in two dimensions. There are much more cases, and in order to understand nontrivial rotations, one has to actually look at things in four dimensions. This seems to anger many people who don't normally think in 4 dimensions. After all we've been through, please stick around for this one too.

15.4.2.1.5.1 Euler Angles The easiest way to think about rotations in 3D is to just think about them as a series of 2D-rotations. Let's called that *nesting a dimension*.

The trick for rotating about one axis in 3D-land works the exact same way it does in 2d land: In order to rotate around one axis, all we need to do is to use a 2d rotation matrix (think about it: a rotation about one axis doesn't depend on the others just yet), and add a neutral dimension to it. Here's what rotation matrices in 3d look like when we use one axis of rotation each time:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And this is indeed a useful way for rotating about one axis. Leonhard Euler, a famous Mathematician working on these types of rotations, noted early on that while good for rotating about one axis, this method (later named after him) was not trivial to state for multiaxial rotations. To understand that, it's easiest to grab a Rubik's cube and twist it about its x dimension, and then about it's y dimension. Now make a note of where the unique tiles have moved, revert the changes, and try it again with first y and then x. Your results will be different!

Noting that, we know that when rotating things in more than 2 dimensions, we need to know not only the angles of the rotations, but the order in which to apply them. The rotation matrices $R_{zx}(\theta) = R_z(\theta) R_x(\theta)$ and $R_{xz}(\theta) = R_x(\theta) R_z(\theta)$ are not the same. The reason for the difference will be explained in the next section.

15.4.2.1.5.2 Other Methods of Rotation: Axis-Angles and Quaternions We can only end this one-page section with a defeating note: rotations in 3D are a big subject. Even though one matrix can only mean one thing, there are multiple ways of getting to it. Euler Angles demonstrated above are one common and easy-to-comprehend way; A slightly more general way is given by defining an arbitrary axis and rotating around it, called *Axis-Angle Rotation*.

//TODO: Draw difference between angle-axis and normal-axis euler rotations

Constructing the matrix for that kind of rotation is slightly hairy, which is why programmers often prefer not to use matrices for describing those rotations, but more compact Algebraic objects called *Quaternions*. Those exist in openFrameworks under

`ofQuaternion`, and can mostly be used without actually investigating the underlying math.

As far as usage goes, it's important to note that Quaternions are sometimes more efficient (and actually easier to think with) than Matrices, but their Mathematical underpinnings are far beyond the scope of a basic math chapter.

15.4.2.2 Matrix Algebra

This chapter introduced a different kind of math from what you were used to. But while introducing *a new thing to do things with* we opened up a lot of unexplored dangers. Notice that we always multiplied vectors by matrices in a certain order: It's always the vector *after* the matrix, the vector is always transposed, and any new operation applied to an existing situation always happens with a matrix to the left of our result. There's a reason for all of that: *Commutativity*.

15.4.2.2.1 Commmumamitativiwha? In high school Algebra, we used to think that $a \cdot b = b \cdot a$. No reason not to think that: The amount of uranium rods that you have times the amount of specially trained monkeys that I have equals the same amount of casualties, no matter the order of multiplication. That's because quantities are *commutative*, the order in which they apply operations to each other doesn't matter.

But, in matrixland we're not talking about things we counted - instead, we're talking about operations, and here's the deal:

> Operations (like Rotation, Translation and Scaling) are generally not commutative.

There's a difference between scaling a square by x and then rotating it by 90 degrees and doing it the other way around:

//TODO: Draw this

What's more, doing it the other way around is not always defined. Matrices and vectors with unequal sizes have very special conditions in which they could be multiplied. We're not dealing with them now, so I'll let you read about it in Wikipedia, but it's important to know that whenever using matrices for manipulating a space, order of operands is really important.

15.4.2.2.2 What else is weird? Nothing. We can still multiply the matrices and vectors in any order that we want to:

$$M_1 M_2 v = (M_1 M_2)\, v = M_1\, (M_2 v)$$

as long as we don't change the order in which they appear. That property is called *Associativity*, and it's one of the defining properties of algebraic structures that describe geometric operations, structures which mathematicians call *Groups*. *Commutativity* is an optional property for groups, it just happens to be a given when dealing with operations between numbers, which is why you've never been told that you need it. There's a lesson here: simulations take a lot of properties for granted. It's sometimes good to ask why.

Now grab a pack of ice, place it on your head for 15 minutes and go on reading the next part.

15.4.3 "The Full Stack"

Now that we know how to construct its major components, let's have a look at all the math that constructs graphics in openFrameworks before sending it to the screen. For that, we'll have to – once again – increase our number of D's.

15.4.3.1 Translation matrices

If you recall the comment in the beginning of this chapter, mathematicians are very careful when calling things linear. In 2D, a linear operation can basically do 2 things: Rotation and Scaling (including negative scaling - "mirroring"). The reason for this is that these are all operations that can be done in n dimensions to any n-dimensional shape (replace n with 3 for our example).

If the entire shape lifts itself magically and moves away from the origin - it can't be done with a matrix, therefore it's not linear. This presents a problem to people who want to use matrices as an algebraic system for controlling 3d: in real life we need to move some stuff around.

15.4.3.1.1 Homogenous coordinates: Hacking 3d in 4d This problem has caused hundreds of years of agony to the openFrameworks community, until in 1827 a hacker called Möbius pushed an update to the ofMäth SVN repo: use the matrix in 4 dimensions to control a 3 dimensional shape. Here's the shtick: a 3d operation can be described as a 4d operation which doesn't do anything to the 4th dimension. Written as a matrix, we can describe it like this:

$$A_{4\times 4} = \left[\begin{array}{ccc|c} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Now we already know that a 1D Skew can move all lines in that axis in a certain direction without affecting the other dimensions, and that a 2D skew will do that for all things on a certain plane, so it's easy to imagine that a 3D skew will do that to 3D spaces embedded in a space with more dimension. Möbius figured that feature is useful, and he proposed on the bianual openFrämeworks meeting in Tübingen that all operations will be conducted in 4D space, and then projected back into 3D space, like this:

$$T = \left[\begin{array}{ccc|c} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

The 3D Transform vector t is placed in the 4th dimension, with it's 4th entry as 1 (because 1 is neutral to multiplication). The bottom row that is added has zeroes in the x, y, z entries, in order to avoid interfering with other operations. Check out what happens when a vector is multiplied by this matrix:

$$T \cdot v = \left[\begin{array}{ccc|c} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{c} x \\ y \\ z \\ \hline 1 \end{array}\right] = \left[\begin{array}{c} x + t_x \\ y + t_y \\ z + t_z \\ \hline 1 \end{array}\right]$$

Now all we need to do is discard the 4th dimension to get our translated point. Quite cool, innit?

Notice that because we placed a 1 at the w (4th) dimension, all of the multiplication operations went through that component transparently. This trick became the standard of most computer geometry systems. Möbius actually has a lot more going in this theory: if we change that 1 into something else, we're able to simulate a projection into a camera pinhole. This chapter won't cover that fascinating part of math, but when you get to building cameras in OpenGL, keep this amazing hack in mind.

15.4.3.2 SRT (Scale-Rotate-Translate) operations

Now we've defined the operations we like the most to describe (sort-of) real world objects moved around in space. Let's spend a few paragraphs talking about how to combine all of the operations together.

If you recall, geometric operations are *non-commutative*, which means that if we defined them in a specific order, there's no guarantee that changing the order will provide us with similar results. ~~That means that when building a graphics system we need to exercise systematic vigilance when executing human stuff like "Turn that spindle around so I may see its refractions of the sun" without accidentally turning the sun around its' axis, incinerating the good people of Uranus.~~ **[mh: this felt distracting]**

The way we execute that vigilance is by a predefined order for handling objects. If you grab a pen and paper it won't take too long to figure that order out: 1. Modify the scale (if need be). 2. Modify the orientation (if need be). 3. Modify the position (if need be). 4. Rejoice.

Any other order will cause weird effects, like things growing and spinning off their axes (anchor point / pivot, if animation is your jam). This may seem like common sense, but Ken Perlin notes that it was only the late 80s when that system became a standard for 3d.

This set of operations is called *SRT*, or *Scale-Rotate-Translate*, because it can be described by the following sequence of matrices:

$$TRSv$$

Where the first operation occuring is the one most adjacent to the vector v.

If you recall, we can multiply all of these matrices to get one matrix representing all the entire operation:

$$M = TRS$$

We call this matrix M, because it places objects we give it in their place in the *Model*. Whenever you call `ofTranslate()`, `ofRotate()`, `ofScale()` (or equivalent) on an object, that operation is applied to the **currently active *Model* matrix**. Whenever you execute `ofPushMatrix()`, a copy of that matrix is saved in *the matrix stack*, so that you can go back to it when necessary. And when necessary, you will then use `ofPopMatrix()`, which will cause the current matrix M to be deleted and replace it with a matrix from the top of the matrix stack. That is the entire mystery about matrices. That's it.

15.4.3.3 Using Matrices and Quaternions in openFrameworks

While this chapter was supposed to show the underlying representation of grpahics operations, it did intentionally avoid showing matrix examples in code. Now that you know how matrices look on the inside, it'll be a lot easier for you to figure out how to debug your 3d code, but most of the time using matrices in raw form won't be necessary.

While you could construct a matrix via `ofMatrix4x4`, using:

```
ofMatrix4x4( const ofQuaternion& quat ) {
    makeRotationMatrix(quat);
}

ofMatrix4x4(    float a00, float a01, float a02, float a03,
             float a10, float a11, float a12, float a13,
             float a20, float a21, float a22, float a23,
             float a30, float a31, float a32, float a33);
```

You'll mostly find that what matters to you is the Algebra of the Operation, not the actual numbers, so you'll be much better off using these:

```
void ofMatrix4x4::makeScaleMatrix( const ofVec3f& );
void ofMatrix4x4::makeScaleMatrix( float, float, float );

void ofMatrix4x4::makeTranslationMatrix( const ofVec3f& );
void ofMatrix4x4::makeTranslationMatrix( float, float, float );

void ofMatrix4x4::makeRotationMatrix( const ofVec3f& from, const
    ofVec3f& to );
void ofMatrix4x4::makeRotationMatrix( float angle, const ofVec3f&
    axis );
void ofMatrix4x4::makeRotationMatrix( float angle, float x, float y,
    float z );
void ofMatrix4x4::makeRotationMatrix( const ofQuaternion& );
void ofMatrix4x4::makeRotationMatrix( float angle1, const ofVec3f&
    axis1,
                  float angle2, const ofVec3f& axis2,
                  float angle3, const ofVec3f& axis3);
```

All these things do is form Operations you can later multiply your `ofVec4f` objects with.

Here's the same example for Quaternions, using the `ofQuaternion` class:

```
/* Axis-Angle Rotations*/
void ofQuaternion::makeRotate(float angle, float x, float y, float
    z);
void ofQuaternion::makeRotate(float angle, const ofVec3f& vec);
void ofQuaternion::makeRotate(float angle1, const ofVec3f& axis1,
    float angle2, const ofVec3f& axis2, float angle3, const ofVec3f&
    axis3);

/* From-To Rotations */
void ofQuaternion::makeRotate(const ofVec3f& vec1, const ofVec3f&
    vec2);
```

Just like with Matrices, any of these objects create a *rotation operation* that can later be applied to a vector:

```
ofVec3f myUnrotatedVector(1,0,0);
ofQuaternion quat;
quat.makeRotate(ofVec3f(1,0,0), ofVec3f(0,1,0));
ofVec3f myRotatedVector = quat * myUnrotatedVector;
cout << ofToString(myRotatedVector) << endl;
//prints out (0,1,0)
```

15.4.3.3.0.1 Ok, Now What? This chapter is just the tip of the iceberg in what math can do for graphics.

In the 'Advanced Graphics' chapter you'll learn about two similar matrices: * The *View* matrix tramsforms the result of the *Model* matrix to simulate where our camera is supposed to be at. * The *Projection* matrix applies the optical properties of the camera we defined and turns the result of the *View* matrix from a 3D space to a 2D image. The Projection matrix is built slightly different than the *Model-View* matrix, but if you've made it this far, you won't have trouble reading about it in a special Graphics topic.

15.4.3.3.0.2 Also, Thanks Learning Math is hard. Teaching Math is therefore excruciating: having so many ideas you want to put in someone else's head, and the slow and sticky nature of it all. I'd like to thank Prof. Bo'az Klartag and Prof. Ken Perlin and for giving me ideas on how to teach mathematics intuitively.

16 Memory in C++

by Arturo Castro[1]

Correctly using memory is one of the trickiest parts of working with c++. The main difference with other languages like Java, Python and in general any languages that are "garbage collected" is that in c++ we can explicitly reserve and free memory while in those an element called garbage collector does the work for us.

There's also an important difference, in c++ we have two different memory areas, the heap and the stack, if you are used to work with processing, Java or Python among others you'll be used to only have heap memory.

We'll see later what the main differences are, but first let's see what's memory and what happens when we create variables in our program.

16.1 Computer memory and variables

It's helpful to understand at least at a high level how computer memory works.

A computer has different types of memory, in this section we are going to be talking about RAM (Random Access Memory) memory. The kind of memory where the computer stores the instructions for the programs that are executing at every moment and the data those programs are using.

Your computer probably has something like 4Gb of RAM, in c++ we can access most of that memory, and to access it what we do is create variables. Memory is divided in bytes, which are the minimum memory size that we can usually use in a c++ application. Each data type like char, int, float... has a different size all measured in bytes. Those sizes can be different for different platforms but the most usual is something like:

- char: 1 byte
- short: 2 bytes
- int: 4 bytes
- float 4 bytes
- double 8 bytes

[1] http://arturocastro.net

Other types might have variable sizes depending on their contents like for example an array or a string.

For example when we create a variable like:

```
int i;
```

what we are doing is reserving 4 bytes of those 4Gb to store an int, it doesn't really matter if we are storing ints or other types of data, the sizes will be different for and int a char, a float or a string but the type of memory is always the same.

Internally the computer doens't really now about that memory area as `i` but as a memory address. A memory address is just a number that points to a specific byte in the 4Gb of memory.

When we create a variable like `int i` we are telling our program to reserve 4 bytes of memory, associate the address of the first byte of those 4 to the variable name `i` and restrict the type of data that we are going to store in those 4 bytes to only ints.

Usually memory addresses are represented in hexadecimal[2]. In c++ you can get the memory address of a variable by using the `&` operator, like:

```
cout << &i << endl;
```

The output of that cout is the memory address of the first byte of the variable `i` we just created.

Later on, when we asign a value to that variable, what it's happening is that we are storing that value in the memory area that we've just reserved by declaring the variable, so when we do:

```
i = 0;
```

Our memory will look like:

The order in which the bytes that form the int are layed out in the memory depends on the architecture of our computer, you'll prpbably seen little endian and big endian[3] mentioned sometime. Those terms refer to how the bytes of a data type are ordered in memory, if the most significative bytes come first or last. Most of the time we don't really need to know about this order but most modern computer architectures use little endian.

If you've used c++ for a while you've probably had crashes in your programs because of bad memory accesses. Usually the message you'll see is something like `segmentation fault...`. What does that mean?

When you create variables in a program, even in c++, you can't really access all the memory in the computer, for security reasons. Imagine you had your bank account

[2] http://en.wikipedia.org/wiki/Hexadecimal
[3] http://en.wikipedia.org/wiki/Endianness

opened in your browser, if any program could access all the memory in the computer a malign application could just access the memory of the browser and get that information or even modify it. To avoid it the operating system asigns chuncks of memory to every program. When your application starts it's asigned a `segment` of memory, later on as you create variables if there's enough memory in that `segment` your variables will be created there. When there's not more memory available in that segment the operating system asigns the application a new one and the application starts using that. If you try to access a memory address that doesn't belong to a segment asigned to your application, the operating system just kills the application to avoid possible security risks.

How does that happen usually? Well most of the time you just don't try to access memory addresses by their number, so how's it possible that sometimes you try to access a variable and you get a segmentation fault. Most of the time this happens because you try to access a varible that doesn't exist anymore, usually because you stored a pointer to a memory area and then free or move that memory somewhere else. We'll talk in more detail about this later

16.2 Stack variables, variables in functions vs variables in objects

As we said at the beginning of the chapter there's two types of memory in c++ the stack and the heap. Let's talk first about the stack since that's the easiest type of memory to use and what you'll use more frequently in openFrameworks.

The stack is the type of memory that you use when creating variables inside a function or in the .h of your class as long as you don't use pointers and the keyword new.

It's called stack because it's organized like a stack[4]. When in our application we call a function, there's an area in memory asigned to that function **call**. That specific function **call**, and only it, during the time it lasts can create variables in that area.

Those variables stop existing when the function call ends. So for example you can't do:

```
void ofApp::setup(){
    int a = 0;
}

void ofApp::update(){
    a = 5; // error a doesn't exist outside setup
}
```

[4] http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29

Also since we are talking about function calls you can't store a value in an stack variable and expect it to be there when the function is called again.

In general we can say that variables exist in the block they've been defined, a block in c++ is defined by the {} inside which a variable was defined, so for example, doing this won't be valid either:

```
for (int i=0;i<10;i++){
    int a = 5;
}
cout << a << endl; // error a doesn't exist outside the {} of the for
```

We can even do this:

```
void ofApp::setup(){
   {
        int a = 0;
        // do something with a
   }

   {
        int a = 0;
        // do something with a
   }
}
```

which is not very common but is used sometimes to define the life of a variable inside a function, mostly when that variable is an object that holds resources and we want to only hold them for a specific duration.

The life of a variable is called `scope`.

Apart from creating variables inside functions we can also create variables in the class declaration in our .h like:

```
class Ball{
public:
    void setup();

    float pos_x;
}
```

These kind of variables are called `instance variables` because every instance or object of our class will get a copy of it. The way they behave is more or less the same as the stack variables created in functions. They exist for the duration of the {} in which they were defined, in this case the {} of the object to which they belong.

These variables can be accessed from anywhere in the class so when we need to have data that is accessible from any function in an object of this class we create it like this.

The memory in the stack is limited, the exact size, depends on the architecture of our computer, the operating system and even the compiler we are using. In some systems it can even be changed during the runtime of the application, but most of the time we won't reach that limit.

Even if we create all of our variables in the stack, usually the objects that consume most memory like for example a vector or in openFrameworks something like ofPixels, will create most of the memory they use internally in the heap which is only limited by the amount of memory available in the computer so we don't need to worry about it.

We'll see in the next sections how the heap works and what are the advantages of using the heap or the stack.

16.3 Pointers and references

Before talking about heap memory let's see how pointers and references work in c++, what's their syntax and what's really happening with memory when we create a pointer or a reference.

As we've seen before we can get the address of a variable by doing:

```
cout << &i << endl;
```

And that will give us the memory address of the first byte used by that variable no matter it's type. When we store that memory address in another variable that's what we call in c++ a pointer. The syntax is:

```
int i = 0;
int * p = &i;
```

And what we get in memory is something like:

A pointer usually occupies 4/8 bytes (depending if we are on a 32 or 64bits application), we are representing it as 1 byte only to make things easier to understand, but as you can see it's just another variable, that instead of containing a value contains a memory address that points to a value. That's why it's called pointer.

A pointer can point to heap or stack memory.

Now, let's explain something that it's really important to take into account when programming in c++. As we've seen till now, when we declare a variable like:

```
int i;
```

We get a memory layout like:

As we see there's no value in that memory area yet. In other languages like processing doing something like:

```
int i;
println(i)
```

is illegal, the compiler will tell us that we are trying to use a variable that is not initialized. In c++ though, that's perfectly legal but the contents of that variable are undefined. Most of the times we'll get 0 because the operating system will clear the memory before assigning it to our program, again, for security reasons. But if we are resuing memory that we had already assigned, then, that memory area will contain anything, and the results of our program will be undefined.

If for example we have a variable that defines the position of something we are going to draw, failing to initialize it will lead to that object being drawn anywhere.

Most objects have default constructors that will initialize their value to, for example 0, so in the case of objects it's usually not necessary to give them a value.

What happens when we use an uninitialized pointer? Well, since a pointer contains a memory address, if the value we get in that memory area points to an address that doesn't belong to our program and we try to retrieve or modify the value stored in that address the OS will kill our application with a segmentation fault signal.

Back to pointers, we've seen that, we can create a pointer like:

```
int i = 5;
int * p = &i;
```

now, if we try to use the pointer directly like;

```
cout << p <<< endl;
```

what we'll get is a memory address not the value 5. So how do we access the value pointed by a pointer, well we can use the opposite operator to &: as & gives us the address of a variable, * gives us the value pointed by a memory address, so we can do:

```
cout << *p << endl;
```

and we'll get the value 5 printed now. We can also do:

```
int j = *p;
cout << j << endl;
```

and again will get the value 5 since we made a copy of the value pointed by p in j.

The &operator is called the *reference operator* since it gives us a reference to a variable, it's memory address. The * operator is it's opposite, the *dereference operator* and it gives us the value pointed by a pointer, it dereferences a reference, a memory address, so we can access it's value instead of the address.

Till now, we've work with primitive values, ints really, but the behaviour will be the same for any other primitive value, like float, short, char, unsigned int... In c++ in fact, the behaviour is also the same for objects.

If you are used to Java, for example you've probably noticed that while in Java and C++ this:

```
int a = 5;
int b = a;
a = 7;
cout << "a: " << a << " b: " << b << endl;
```

will behave the same (of course changing cout for the equivalent in java). That is: a will end up being 7 and b will be 5. When we use objects the behaviour in c++ is different to that of Java. For example, let's say we have a class Ball:

```
class Ball{
public:
    void setup();
    //...

    ofVec2f pos;
}
```

or the similar class in processing;

```
class Ball{
    void setup();

    PVector pos;
}
```

if in c++ you do:

```
Ball b1;
b1.pos.set(20,20);
Ball b2;
b2 = b1;
b2.pos.set(30,30);
```

b1 pos will end up being 20,20 and b2 30,30 while if you do the equivalent in java both b1 and b2 will have position 30,30:

```
Ball b1 = new Ball();
b1.pos.set(20,20);
Ball b2;
b2 = b1;
b2.pos.set(30,30);
```

Notice how in the case of Java we have made new for the first ball but not for the second, that's because in Java everything that is an object is a pointer in the heap so when we do `b2 = b1`we are actually turning b2 into a reference to b1, and when we later change b2, we are also changing b1.

In c++, instead when we do `b2 = b1` we are actually copying the values of the variables of b1 into b2 so we still have 2 different variables instead of a reference. When we modify b2, b1 stays the same.

In both languages the = means copy the value of the right side into the variable on the left side of the =. The difference is that in Java an object is really a pointer to an object the contents of `b1` or `b2` are not the object itself but it's memory address, while in c++ b1 actually contains the object itself.

This is more or less what memory would look like in Java and C++:

As you can see in c++ objects in memory are just all their member variables one after another. When we make an object variable equal to another, by default, c++ copies all the object to the left side of the equal operator.

Now what would happen if we have a class like:

```
class Particle{
public:
    void setup();
    //...

    ofVec2f pos;
    ParticleSystem * parent;
}
```

And we do:

```
Particle p1;
Particle p2;
ParticleSystem ps;

p1.pos.set(20,20);
p1.parent = &ps;
p2 = p1;
```

Well as before c++ will copy the contents of p1 on p2, the contents of p1 are an ofVec2f which consits of 2 floats x and y and then a pointer to a ParticleSystem, and that'w what gets copied, the ParticleSystem itself won't get copied only the pointer to it, so p2 will end up having a copy of the position of p2 and a pointer to the same ParticleSystem but we'll have only 1 particle system.

The fact that things are copied by default and that objects can be stored in the stack as oposed to being always a pointer has certain adavantages. For example, in c++ a

vector or an array of particles like the ones we've used in the last example will look like:

```
vector<Particle> particles;
```

in memory all the particles will be contiguous, among other things, that makes accessing them faster than if we had pointers to a different location in memory. It also makes it easier to translate c++ vectors to openGL memory structures but that's the topic for another chapter.

Among other things we need to be aware of the fact that c++ copies things by default, when passing objects to functions as parameters. For example this:

```
void moveParticle(Particle p){
    p.x += 10;
    p.y += 10;
}

...

Particle p1;
moveParticle(p1);
```

Is perfectly valid code, but won't have any effect since the function will receive a copy of the particle and modify that copy instead of the original.

We can do this:

```
Particle moveParticle(Particle p){
    p.x += 10;
    p.y += 10;
    return p;
}
...

Particle p1;
p1 = moveParticle(p1);
```

So we pass a copy of our particle to the function which modifies it's values and returns a modified copy which we then copy into p1 again. See how many times we've mentiponed copy in the previous sentence? The compiler will optimize some of those out and for small objects it's perfectly ok to do that but imagine we had something like this:

```
vector<Particle> moveParticles(vector<Particle> ps){
    for(int i=0;i<ps.size();i++){
        ps[i].x += 10;
        ps[i].y += 10;
    }
```

```
    return ps;
}
...

vector<Particle> ps;
...
ps = moveParticles(ps);
```

If we have 1 million particles that will be awfully slow, memory is really slow compared to the cpu, so anything that involves copying memory or allocating new memory should be usually avoided. So what can we do to avoid all that copies?

Well we could use pointers right?

```
void moveParticle(Particle * p){
    p->x += 10;
    p->y += 10;
}
...

Particle p1;
moveParticle(&p1);
```

Now, here's something new, notice how to refer to the variables of a pointer to an object instead of using the dot, we use the -> operator, everytime we want to access a variable in a pointer to an object instead of having to dereference it like:

```
(*p).x +=10
```

we can use the ->

```
p->x += 10
```

So that solves our problem, using a pointer instead of passing a copy of the object, we are passing a reference to it, it's memory address, so the function will actually modify the original.

The main problem with this is that the syntax is kind of weird, imagine how would look like if we passed a pointer for the second example, the one with the vector:

```
vector<Particle> moveParticles(vector<Particle> ps){
    for(int i=0;i<ps.size();i++){
        ps[i].pos.x += 10;
        ps[i].pos.y += 10;
    }
    return ps;
}
...
```

```
vector<Particle> ps;
...
ps = moveParticles(ps);
```

Now, the function will look like:

```
void moveParticles(vector<Particle> * ps){
```

the problem is that now we can't use the [] operator to access the elements in the vector cause ps is not a vector anymore but a pointer to a vector. What's more this

```
ps[i].x += 10;
```

would actually compile but would mostly sure give as a memory access error, a segmentation fault. ps is now a pointer and when using pointers the `[]' behaves like if we had an array of vectors!

We'll explain this in more depth in the section about memory structures, but let's see how to pass a reference that doens't have pointer syntax. In c++ is called a reference and it looks like:

```
void moveParticles(vector<Particle> & ps){
    for(int i=0;i<ps.size();i++){
        ps[i].pos.x += 10;
        ps[i].pos.y += 10;
    }
}

vector<Particle> ps;
...
moveParticles(ps);
```

Now we are passing a reference to the original object but instead of having to use pointer syntax we can still use it as if it was a normal object.

> Advanced note: Some times we want to use references to avoid copies but still be sure that the function we pass our object to, won't modify it's contents, in that case it's recomendable to use const like:
>
> ```
> ofVec2f averagePosition(const vector<Particle> & ps){
>     ofVec2f average;
>     for(int i=0;i<ps.size();i++){
>         average += ps[1].pos;
>     }
>     return average/float(ps.size());
> }
> vector<Particle> ps;
> ...
> ofVec2f averagePos = averagePosition(ps);
> ```

const only makes it imposible to modify the variable, even if it's a reference, and tells anyone using that function that they can pass their data into it and it won't be changed, also anyone modifying that function knows that in the future it should stay the same and the input, the particle system shouldn't be modified.

Outside of parameters, references have a couple of special characteristics.

First we can't modify the content of a reference once it's created, for example we can do:

```
ofVec2f & pos = p.pos;
pos.x = 5;
```

but trying to change the reference itself like in:

```
ofVec2f & pos = p.pos;
pos.x = 5;
pos = p2.pos  // error, a reference can only be asigned on it's
   declaration
```

Also you can return a reference but depending on what that reference it's pointing to it can be a bad idea:

```
ofVec2f & averagePosition(const vector<Particle> & ps){
    ofVec2f average;
    for(int i=0;i<ps.size();i++){
        average += ps[1].pos;
    }
    average/=float(ps.size());
    return average;
}
```

Will actually compile but will probably result in a segmentation fault at some point or even just work but we'll get weird values when calling this funciton. The problem is that we are creating the variable `average` in the stack so when the function returns it'll be *deleted* from memory, the reference we return will be pointing to a memory area that is not reserved anymore for average and as soon as it gets overwritten we'll get invalid values or a pointer to a memory area that doesn't belong to our program anymore.

This is one of the most annoying problems in c++ it's called dangling pointers or in this case references and it's caused when we have a pointer or a reference that points to a memory area that is later freed somehow.

More modern langauges solve this with diferent strategies, for example Java won't let this happen since objects are only deleted once the last reference to them goes out of scope, it uses something called a garbage collector that from time to time goes through

the memory looking for objects which have no more references pointing to them, and deletes them. This solves the problem but makes it hard to know when objects are going to get really deleted. c++ in it's latest versions, and more modern languages try to solve this using new kinds of pointers that define ownership of the object, we'll talk about it in the latest section of this chpater, smart pointers.

16.4 Variables in the heap

Now that we now the syntax and semantics of pointers lets see how to use the heap. The heap is an area of memory common to all of our application, any function can create variables in this space and share it with others, to use it we need a new keyword `new`:

```
Particle * p1 = new Particle;
```

If you know processing or Java that looks a little bit like it, right? indeed this is exactly the same as a Java object: when we use `new` we are creating that variable in the heap instead of the stack. `new` returns a pointer to a memory address in the heap and in c++ we explictly need to specify that the variable `p1` where we are going to store it, is a pointer by using the `*` in the declaration.

To access the variables or functions of a pointer to an object, as we've seen before, we use the `->` operator so we would do:

```
Particle * p1 = new Particle;
p1->pos.set(20,20);
```

or:

```
Particle * p1 = new Particle;
p1->setup();
```

A pointer as any variable can be declared without initializing it yet:

```
Particle * p1;
p1->setup() // this will compile but fail when executing the
    application
```

We can imagine the heap as some kind of global memory as opposed to the stack being local memory. In the stack only the block that owned it could access it while things created in the heap outlive the scope in which they were created and any function can access them as long as they have a reference (a pointer) to them. For example:

```
Particle * createParticle(){
    return new Particle;
```

```
}

void modifyParticle(Particle * p){
    p->x += 10;
}

...

Particle * p = createParticle();
modifyParticle(p);
```

`createParticle` is creating a new `Particle` in the heap, so even when createParticle finishes that `Particle` still exists. We can use it outside the function, pass a reference to it to other functions...

So how can we say that we don't want to use that variable anymore? we use the keyword `delete`:

```
Particle * p1 = new Particle;
p1->setup();
...
delete p1;
```

This is important when using the heap, if we fail to do this we'll get with what is called a memory leak, memory that is not referenced by anyone but continues to leave in the heap, making our application use more and more memory over time till it fills all the available memory in our computer:

```
void ofApp::draw(){
    Particle * p1 = new Particle;
    p1->setup();
    p1->draw();
}
```

every time we call `draw`, it'll create a new particle, once each `draw` call finishes we loose the reference *p1 to it but the memory we allocated using new is not freed when the function call ends so our program will slowly use more and more memory, you can check it using the system monitor.

As we've mentioned before the stack memory is limited so sometimes we need to use the heap, trying to create 1 million particles in the stack will probably cause a stack overflow. In general, though, most of the time in c++ we don't need to use the heap, at least not directly, classes like vector, ofPixels and other memory structures allow us to use heap memory but still have stack semantics, for example this:

```
void ofApp::draw(){
    vector<Particle> particles;
    for(int i=0; i<100; i++){
```

```
        particles.push_back(Particle())
        particles.back().setup();
        particles.back().draw();
    }
}
```

is actually using heap memory since the vector is internally using that, but the vector destructor will take care of freeing that memory for us as soon as the particles variable goes out of scope, when the current call to draw finishes.

16.5 Memory structures, arrays and vectors

Arrays are the most simple way in c++ to create collections of objects, as any other type in c++ they can also be created in the stack or in the heap. Arrays in the stack have a limitation though, they need to have a predifined size that needs to be specified in it's declaration and can't change afterwards:

```
int arr[10];
```

the same as with any other type, the previous declaration already reserves memory for 10 ints, we don't need to use new, and that memory will be uninitialized. To access them, as you might now from previous chapters you can just do:

```
int arr[10];
arr[0] = 5;
int a = arr[0]
```

if we try to do:

```
int arr[10];
int a = arr[5];
```

the value of a will be undefined since the memory in the array is not initialized to any value when it's created. Also if we try to do:

```
int arr[10];
int a = arr[25];
```

most probably our application will crash if the memory address at arr + 25 is outside the memory that the operating system has asigned to our application.

We've just sayd arr + 25? what does that mean? As we've seen before a variable is just some place in memmory, we can get it's memory address which is the first byte that is asigned to that variable in memory. With arrays is pretty much the same, for example

since we know that an int occupies 4 bytes in memory, an array of 10 ints will occupy 40 bytes and those bytes are contiguous:

Remember that memory addresses are expressed as hexadecimal so 40 == 0x0028. Now to take the address of an array, as with other variable we might want to use the `&` operator and indeed we can do it like:

```
int arr[0];
int * a = &arr[0];
```

That gives us the address of the first element of the array which is indeed that of the array, but with arrays, the same variable is actually a pointer itself:

```
int arr[10];
int * a = &arr[0];
cout << "a: " << a << " arr: " << arr << endl;
```

will print the same value for both a and arr. So an array is just a pointer to a memory address with the only difference that, that memory address is the beginning of reserved memory enough to allocate, in our case, 10 ints. All those ints will be one after another, so when we do `arr[5]` we are just accessing the value that is in the memory address of our array + the size of 5 ints. If our array started in `0x0010`, and ints ocupy `4 bytes`, arr[5] would be `10 + 4 * 5 = 30` which in hexadecimal is `0x001E`. We can actually do this in our code:

```
int arr[10]
arr[5] = 20;
cout << "&arr[5]: " << &arr[5] << "arr+5: " << arr+5 << endl
cout << "arr[5]: " << arr[5] << "*(arr+5): " << *(arr+5) << endl
```

now, that's really weird and most of the time you won't use it, it's called pointer arithmetic. The first cout will print the address in memory of the int at position 5 in the array in the first case using the `&` operator to get the address of `arr[5]` and in the second directly by adding 5 to the first address of `arr` doing `arr+5`. In the second cout we print the value at that memory location, using `arr[5]` or dereferencing the address `arr+5` using the `*` operator.

Note that when we add `5` to the adress of the array it's not bytes we are adding but the size in bytes of the type it contains, in this case `+5` actually means `+20` bytes, you can check it by doing:

```
int arr[10]
arr[5] = 7;
cout << "arr: " << arr << "arr+5: " << arr+5 << endl
```

and substracting the hexadecimal values in a calculator. If you try to substract them in your program like:

```
int arr[10]
arr[5] = 20;
cout << "arr: " << arr << "arr+5: " << arr+5 << endl
cout << "(arr+5) - arr: " << (arr+5) - arr << endl
```

You will end up with 5 again because as we've said pointer arithmetic works with the type size not bytes.

The syntax of pointer arithmetics is kind of complicated, and the idea of this part wasn't really to show pointer arithmetics itself but how arrays are just a bunch of values one after another in memory, so don't worry if you haven't understood fully the syntax, is probably something you won't need to use. It is also important to remember that an array variable acts as a pointer so when we refer to it without using the [] operator we end up with a memory address not with the values it contains.

The arrays we've created till now are created in the stack so be careful when using big arrays like this cause it might be problematic.

Arrays in the heap are created like anything else in the heap, by using new:

```
int arr[] = new int[10];
```

or

```
int * arr = new int[10]
```

As you can see this confirms what we've said before, an array variable is just a pointer, when we call new int[10] it allocates memory to store 10 integers and returns the memory address of the first byte of the first integer in the array, we can keep it in a pointer like in the second example or using int arr[] which declares an array of unkown size.

The same as other variables created in the heap we'll need to delete this manually so when we are done with it we need to use delete to deallocate that memory, in the case of arrays in the heap the syntax is slightly special:

```
int arr[] = new int[10];
...
delete[] arr;
```

if you fail to use the [] when deleting it, it'll only deallocate the first value and you'll end up with a memory leak.

There's also some problems with the syntax of arrays, for example this:

```
int arr[10]
int arrB[10];
arrB = arr;
```

will fail to compile. And this:

```
int arr[] = new int[10];
int arrB[] = new int[10];
arrB = arr;
```

will actually compile but as with other variables we are not copying the values that arr points to into arrB but instead the memory address. In this case will end up with 2 pointers pointing to the same memory location, the one that we created when creating arr and loose the memory that we allocated when initializing arrB. Again we have a memory leak, the memory allocated when doing `int arrB[] = new int[10];` is not referenced by any variable anymore so we can't delete it anymore. To copy arrays there's some **c** (not c++) functions liek memcpy but their syntax is kind of complex, that's why when working with c++ is recomended to use vectors.

C++ vectors are very similar to arrays, indeed their layout in memory is the same as an array, they contain a bunch of values contiguous in memory and always allocated in the heap. The main difference is that we get a nicer syntax and *stack semantics*. To allocate a vector to contain 10 ints we can do:

```
vector<int> vec(10);
```

We can even give an initial value to those 10 ints in the initialization like:

```
vector<int> vec(10,0);
```

And for example copying a vector into another, works as expected:

```
vector<int> vec(10,0);
vector<int> vecB = vec;
```

Will create a copy of the contents of vec in vecB. Also even if the memory that the vector uses is in the heap, when a vector goes out of scope, when the block in which it was declared ends, the vector is destroyed cause the vector itself is created in the stack, so going out of scope, triggers it's destructor that takes care of deleting the memory that it has created in the heap:

```
void ofApp::update(){
    vector<int> vec(10);
    ...
}
```

That makes vectors easier to use than arrays since we don't need to care about deleting them, end up with dangling pointers, memory leaks... and their syntax is easier.

Vectors have some more features and using them properly might be tricky mostly if we want to optimize for performance or use them in multithreaded applications, but that's not the scope of this chapter, you can find some tutorials about vectors, this is

an introductory one in the openFrameworks site: vectors basics[5] and this one explains more advanced concepts std::vector[6]

16.6 Other memory structures, lists and maps

Having objects in memory one after another is most of the time what we want, the access is really fast no matter if we want to access sequentially to each of them or randomly to anyone, since a vector is just an array internally, accesing let's say position 20 in it, just means that internally it just needs to get the memory address of the first position and add 20 to it. In soime cases though, vectors are not the most optimal memory structure. For example, if we want to frequnetly add or remove elements in the middle of the vector, and you imagine the vector as a memory strip, that means that we need to move the rest of the vector till the end one position to the right and then insert the new element in the free location. In memory there's no such thing as move, moving contiguous memory means copying it and as we've said before, copying memory is a relatively slow operation.

Sometimes, if there's not enough memory to move/copy the elements, one position to the right, the vector will need to copy the whole thing to a new memory location. If we are working with thousands of elements and doing this very frequently, like for example every frame, this can really slow things down a lot.

To solve that, there's other memory structures like for example lists. In a list, memory, is not contiguous but instead each element has a pointer to the next and previous element so inserting one element just means changing those pointers to point to the newly added element. In a list we never need to move elements around but it's main disadvantage is that not being the elements contiguous in memory it's access can be slightly slower than a vector, also that we can't use it in certain cases like for example to upload data to the graphics card which always wants contiguos memory.

Another problem of lists is that trying to access an element in the middle of the list (what is called random access) is slow since we always have to go through all the list till we arrive to the desired element. Lists are used then, when we seldom need to access randomly to a position of it and we need to add or remove elements in the middle frequently. For the specifics of the syntax of a list you can check the c++ documentation on lists[7]

There's several memory structures in the c++ standard library or other c++ libraries, apart from vectors and lists we are only going to see briefly maps.

Sometimes, we don't want to access things by their position or have an ordered list of elements but instead have something like an index or dictionary of elements that we

[5] http://openframeworks.cc/tutorials/c++%20concepts/001_stl_vectors_basic.html
[6] http://arturocastro.net/blog/2011/10/28/stl::vector/
[7] http://www.cplusplus.com/reference/list/list/

can access by some key, that's what a map is. In a map we can store pairs of (key,value) and look for a value by it's key. For example let's say we have a collection of objects which have a name, if that name is unique for all the objects, we can store them in a map to be able to look for them by their name:

```
map<string, NamedObject> objectsMap;

NamedObject o1;
o1.name = "object1";
objectsMap[o1.name] = o1;
```

Later on we can look for that object using it's name like:

```
objectsMap["object1"].doSomething();
```

Be careful though, if the object didn't exist before, using the `[]` operator will create a new one. If you are not sure, it's usually wise to try to find it first and only use it if it exists:

```
if(objectsMap.find("object1")!=objectsMap.end()){
    objectsMap["object1"].doSomething();
}
```

You can find the complete reference on maps in the c++ documentation for maps[8]

16.7 smart pointers

As we've said before, traditional c pointers also called now *raw pointers* are sometimes problematic, the most frequent problems are dangling pointers: pointers that probably were once vañlid but now point to an invalid memory location, trying to dereference a NULL pointer, posible memory leaks if we fail to deallocate memory before loosing the reference to that memory address...

Smart pointers try to solve that by adding what we've been calling stack semantics to memory allocation, the correct term for this is RAII: Resource Acquisition Is Initialization[9] And means that the creation of an object in the stack, allocates the resources that it'll use later. When it's destructor is called because the variable goes out of scope, the destructor of the object is triggered which takes care of deallocating all the used resources. There's some more implications to RAII but for this chapter this is what matters to us more.

Smart pointers use this technique to avoid all the problems that we've seen in raw pointers. They do this by also defining better who is the owner of some allocated

[8] http://www.cplusplus.com/reference/map/map/
[9] http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

memory or object. Till now we've seen how things allocated in the stack belong to the function or block that creates them we can return a copy of them (or in c++11 or later, move them) out of a function as a return value but their ownership is always clear.

With heap memory though, ownership becomes way more fuzzy, someone might create a variable in the heap like:

```
int * createFive(){
    int * a = new int;
    *a = 5;
    return a;
}
```

Now, when someone calls that function, who is the owner of the `new int`? Things can get even more complicated, what if we pass a pointer to that memory to another function or even an object?

```
// ofApp.h
int * a;
SomeObject object;

// ofApp.cpp
void ofApp::setup(){
    a = createFive();
    object.setNumber(a);
}
```

who is now the owner of that memory? ofApp? object? The ownership defines among other things who is responsible for deleting that memory when it's not used anymore, now both ofApp and object have a reference to it, if ofApp deletes it before object is done with it, object might try to access it and crash the application, or the other way around. In this case it seems logical that ofApp takes care of deleting it since it knows about both object and the pointer to int a, but what if we change the example to :

```
// ofApp.h
SomeObject object;

// ofApp.cpp
void ofApp::setup(){
    int * a = createFive();
    object.setNumber(a);
}
```

or even:

```
// ofApp.h
SomeObject object;

// ofApp.cpp
```

```
void ofApp::setup(){
    object.setNumber(createFive());
}
```

now ofApp doesn't know anymore about the allocated memory but both cases are possible so we actually need to know details of the implementation of object to know if we need to keep a reference of that variable to destroy it later or not. That, among other things breaks encapsulation, that you might now from chapter 1. We shouldn't need to know how object works internally to be able to use it. This makes the logic of our code really complicated and error prone.

Smart pointers solve this by clearly defining who owns and object and by automatically deleting the allocated memory when the owner is destroyed. Sometimes, we need to share an object among several owners. For that cases we have a special type of smart pointers called shared pointers that defined a shader ownership and free the allocated memory only when all the owners cease to use the variable.

We are only going to see this briefly, there's lots of examples in the web about how to use smart pointers and reference to their syntax, the most important is to understand how they work by defining the ownership clearly compared to raw pointers and the problems they solve.

16.7.1 unique_ptr

A unique_ptr, as it's name suggests, is a pointer that defines a unique ownership for an object, we can move it around and the object or function that has it at some point is the owner of it, no more than one reference at the same time is valid and when it goes out of scope it automatically deletes any memory that we might have allocated.

To allocate memory using a unique_ptr we do:

```
void ofApp::setup(){
    unique_ptr<int> a(new int);
    *a = 5;
}
```

As you can see, once it's created it's syntax is the same as a raw pointer, we can use the * operator to dereference it and access or modiify it's value, if we are working with objects like:

```
void ofApp::setup(){
    unique_ptr<Particle> p(new Particle);
    p->pos.set(20,20);
}
```

We can also use the -> to access it's member variables and functions.

When the function goes out of scope, being unique_ptr an object, it's destructor will get called, which internally will call `delete` on the allocated memory so we don't need to call delete on unique_ptr at all.

Now let's say we want to move a unique_ptr into a vector:

```
void ofApp::setup(){
    unique_ptr<int> a(new int);
    *a = 5;

    vector<unique_ptr<int> > v;
    v.push_back(a);  // error
}
```

That will generate a long error, depending on the compiler, really hard to understand. What's going on, is that `a` is still owned by ofApp::setup so we can't put it in the vector, what we can do is move it into the vector by explicitly saying that we want to move the ownership of that unique_ptr into the vector:

```
void ofApp::setup(){
    unique_ptr<int> a(new int);
    *a = 5;

    vector<unique_ptr<int> > v;
    v.push_back(move(a));
}
```

There's a problem that unique_ptr doesn't solve, we can still do:

```
void ofApp::setup(){
    unique_ptr<int> a(new int);
    *a = 5;

    vector<unique_ptr<int> > v;
    v.push_back(move(a));

    cout << *a << endl;
}
```

The compiler won't fail there but if we try to execute the application it'll crash since `a` is not owned by ofApp::setup anymore, having to explicitly use `move` tries to solve that problem by making the syntax clearer. After using move, we can't use that variable anymore except through the vector. More modern langauages like Rust[10] completely solve this by making the compiler detect this kind of uses of moved variables and producing a compiler error. This will probably be solved at some point in c++ but by now you need to be careful to not use a moved variable.

[10] http://www.rust-lang.org/

16.7.2 shared_ptr

As we've seen before, sometimes having unique ownership is not enough, sometimes we need to share an object among several owners, in c++11 or later, this is solved through `shared_ptr`. The usage is pretty similar to `unique_ptr`, we create it like:

```
void ofApp::setup(){
    shared_ptr<int> a(new int);
    *a = 5;

    vector<shared_ptr<int> > v;
    v.push_back(a);
}
```

The difference is that now, both the vector and ofApp::setup, have a reference to that object, and doing:

```
void ofApp::setup(){
    shared_ptr<int> a(new int);
    *a = 5;

    vector<shared_ptr<int> > v;
    v.push_back(a);

    cout << *a << endl;
}
```

Is perfectly ok. The way a shared_ptr works is by keeping a count of how many references there are to it, whenever we make a copy of it, it increases that counter by one, whenever a reference is destroyed it decreases that reference by one. When the reference cound arrives to 0 it frees the allocated memory. That reference counting is done atomically, which means that we can share a shared_ptr across threads without having problems with the count. That doesn't mean that we can access the contained data safely in a multithreaded application, just that the reference count won't get wrong if we pass a shared_ptr accross different threads.

17 Threads

by Arturo Castro[1]

corrections by Brannon Dorsey

17.1 What's a thread and when to use it

Sometimes in an application we need to execute tasks that will take a while to finish. The perfect example is reading something from disk. In the computer the CPU is way faster than accessing the memory which is way faster than accessing the hard disk. So accessing, for example, an image from the HD can take a while compared to the normal flow of the application.

In openFrameworks, and in general, usually when working with openGL, our application will run in an infinite loop calling update/draw every cycle of the loop. If we have vertical sync enabled, and our screens works at 60Hz, each of those cycles will last around 16ms (1s/(60frames/s))*1000(ms/s). Loading an image from disk can take way more than those 16ms, so if we try to load an image from our update method, for example, we'll notice a pause in our animation.

To solve this we usually use threads. Threads are a way of executing certain tasks inside an application outside of the main flow. That way we can run more than one task at once so things that are slow don't stop the main flow of the application. We can also use threads to accelerate tasks by dividing them in several smaller tasks and running each of those at the same time. You can think of a thread as a subprogram inside your program.

Every application has at least 1 thread. In openFrameworks, that thread is where the setup/update/draw loop happens. We'll call this the main (or openGL) thread. But we can create more threads and each of them will run separately from the others.

So if we want to load an image in the middle of our application, instead of loading our image in update, we can create a thread that loads the image for us. The problem with this is that once we create a thread, the main thread doesn't know when it has finished, so we need to be able to communicate the results from our auxiliary thread to the main one. There's also problems that might arrise from different threads accessing

[1] http://arturocastro.net

the same areas in memory. We'll need some mechanisms to synchronize the access to shared memory between 2 or more threads.

First let's see how to create a thread in openFrameworks.

17.2 ofThread

Every application has at least one thread, the main thread (also called the GL thread), when it's using openGL.

But as we've said we can create auxiliary threads to do certain tasks that would take too long to run in the main thread. In openFrameworks we can do that using the ofThread class. ofThread is not meant to be used directly, instead we inherit from it and mplement a `threadedFunction` which will later get called from the auxiliary thread once we start it:

```
class ImageLoader: public ofThread{
    void setup(string imagePath){
        this->imagePath = imagePath;
    }

    void threadedFunction(){
        ofLoadImage(path,image);
    }

    ofPixels image;
    string path;
}

//ofApp.h

ImageLoader imgLoader;


// ofApp.cpp
void ofApp::keyPressed(int key){
    imgLoader.setup("someimage.png");
    imgLoader.startThread();
}
```

When we call `startThread()`, `ofThread` starts a new thread and returns immediately, that thread will call our `threadedFunction` and will finish when the function ends.

This way the loading of the image happens simultaneously to our update/draw loop and our application doesn't stop while waiting till the image is loaded from disk.

Now, how do we know when our image is loaded? The thread will run separately from the main thread of our application:

As we see in the image the duration of loading of the image and thus the duration of the call to threadedFunction is not automatically known to the main thread. Since all our thread does is load the image, we can check if the thread has finished running which will tell us that the image has loaded. For that ofThread has a method: `isThreadRunning()`:

```
class ImageLoader: public ofThread{
    void setup(string imagePath){
        this->imagePath = imagePath;
    }

    void threadedFunction(){
        ofLoadImage(path,image);
    }

    ofPixels image;
    string path;
}

//ofApp.h
bool loading;
ImageLoader imgLoader;
ofImage img;

// ofApp.cpp
void ofApp::setup(){
    loading = false;
}

void ofApp::update(){
    if(loading==true && !imgLoader.isThreadRunning()){
        img.getPixelsRef() = imgLoader.img;
        img.update();
        loading = false;
    }
}

void ofApp::draw(){
    img.draw(0,0);
}

void ofApp::keyPressed(int key){
    if(!loading){
        imgLoader.setup("someimage.png");
        loading = true;
        imgLoader.startThread();
    }
}
```

Now as you can see we can only load a new image when the first one has finished loading. What if we want to load more than one? A possible solution would be to start a new thread and ask it if it's been loaded already:

```
class ImageLoader: public ofThread{
    ImageLoader(){
        loading = false;
    }

    void load(string imagePath){
        this->imagePath = imagePath;
        loading = true;
        startThread();
    }

    void threadedFunction(){
        ofLoadImage(path,image);
        loaded = true;
    }

    ofPixels image;
    string path;
    bool loading;
    bool loaded;
}

//ofApp.h
vector<unique_ptr<ImageLoader>> imgLoaders;
vector<ofImage> imgs;

// ofApp.cpp
void ofApp::setup(){
    loading = false;
}

void ofApp::update(){
    for(int i=0;i<imgLoaders.size();i++){
        if(imgLoaders[i].loaded){
            if(imgs.size()<=i) imgs.resize(i+1);

            imgs[i].getPixelsRef() = imgLoaders[i].img;
            imgs[i].update();
            imgLoaders[i].loaded = false;
        }
    }
}

void ofApp::draw(){
    for(int i=0;i<imgLoaders.size();i++){
```

```
        imgs[i].draw(x,y);
    }
}

void ofApp::keyPressed(int key){
    imgLoaders.push_back(move(unique_ptr<ImageLoader>(new
        ImageLoader)));
    imgLoaders.back().load("someimage.png");
}
```

Another possibility would be to use 1 thread only. To do that a possible solution would be to use a queue in our loading thread whenever we want to load a new image. To do this we insert it's path in the queue and when the threadedFunction finishes loading one image it checks the queue. If there's a new image it loads it and it is removed from the queue.

The problem with this is that we will be trying to access the queue from 2 different threads, and as we've mentioned in the memory chapter, when we add or remove elements to a memory structure there's the possibility that the memory will be moved somewhere else. If that happens while one thread is trying to access it we can easily end up with a dangling pointer that will cause the application to crash. Imagine the next sequence of instruction calls from the 2 different threads:

```
    loader thread: finished loading an image
    loader thread: pos = get memory address of next element to load
    main thread:   add new element in the queue
    main thread:   queue moves in memory to an area with enough
        space to allocate it
    loader thread: try to read element in pos  <- crash pos is no
        longer a valid memory address
```

At this point we might be accessing a memory address that doesn't contain a string anymore, or even trying to access a memory address that is outside of the memory assigned to our application. In this case the OS will kill it sending a segmentation fault signal as we've seen in the memory chapter.

The reason this happens is that since thread 1 and 2 run simultaneously we don't know in which order their instructions area going to get executed. We need a way to ensure that thread 1 cannot access the queue while thread 2 is modifying it and viceversa. For that we'll use some kind of lock: In C++ usually a mutex, in openFrameworks an ofMutex.

But before seeing mutexes, let's see briefly some particulars of using thread while using openGL.

17.3 Threads and openGL

You might have noticed in the previous examples:

```
class ImageLoader: public ofThread{
    ImageLoader(){
        loaded = false;
    }
    void setup(string imagePath){
        this->imagePath = imagePath;
    }

    void threadedFunction(){
        ofLoadImage(path,image);
        loaded = true;
    }

    ofPixels image;
    string path;
    bool loaded;
}

//ofApp.h
ImageLoader imgLoader;
ofImage img;

// ofApp.cpp
void ofApp::setup(){
    loading = false;
}

void ofApp::update(){
    if(imgLoader.loaded){
        img.getPixelsRef() = imgLoader.img;
        img.update();
        imgLoader.loaded = false;
    }
}

void ofApp::draw(){
    img.draw(0,0);
}

void ofApp::keyPressed(int key){
    if(!loading){
        imgLoader.setup("someimage.png");
        imgLoader.startThread();
    }
}
```

Instead of using an ofImage to load images, we are using an ofPixels and then in the main thread we use an ofImage to put the contents of ofPixels into it. This is done because openGL, in principle, can only work with 1 thread. That's why we call our main thread the GL thread.

As we mentioned in the advanced graphics chapter and other parts of this book, openGL works asynchronously in some kind of client/server model. Our application is the client sending data and drawing instructions to the openGL server which will send them to the graphics card in it's own times.

Because of that, openGL knows how to work with one thread, the main thread from which the openGL context was created. But if we try to do openGL calls from a different thread we will most surely crash the application, or at least not get the desired results.

When we call `img.loadImage(path)` on an ofImage, it'll actually do some openGL calls, mainly create a texture and upload to it the contents of the image. If we did that from a thread that isn't the GL thread, our application will probably crash or just don't load the texture properly.

There's a way to tell ofImage, and most other objects that contain pixels and textures in openFrameworks, to not use those textures and instead work only with pixels. That way we could use an ofImage to load the images to pixels and later in the main thread activate the textures to be able to draw the images:

```
class ImageLoader: public ofThread{
    ImageLoader(){
        loaded = false;
    }
    void setup(string imagePath){
        image.setUseTexture(false);
        this->imagePath = imagePath;
    }

    void threadedFunction(){
        image.loadImage(path);
        loaded = true;
    }

    ofImage image;
    string path;
    bool loaded;
}

//ofApp.h
ImageLoader imgLoader;

// ofApp.cpp
```

```
void ofApp::setup(){
    loading = false;
}

void ofApp::update(){
    if(imgLoader.loaded){
        imgLoader.image.setUseTexture(true);
        imgLoader.image.update();
        imgLoader.loaded = false;
    }
}

void ofApp::draw(){
    imageLoader.image.draw(0,0);
}

void ofApp::keyPressed(int key){
    if(!loading){
        imgLoader.setup("someimage.png");
        imgLoader.startThread();
    }
}
```

There are ways to use openGL from different threads, for example creating a shared context to upload textures in a different thread or using PBO's to map a memory area and later upload to that memory area from a different thread but that's out of the scope of this chapter. In general remember that accessing openGL outside of the GL thread is not safe. In openFrameworks you should only do operations that involve openGL calls from the main thread, that is, from the calls that happen in the setup/update/draw loop, the key and mouse events, and the related ofEvents. If you start a thread and call a function or notify an ofEvent from it, that call will also happen in the auxiliary thread, so be careful to not do any GL calls from there.

A very specific case is sound, sound APIs in openFrameworks, in particular ofSoundStream, create their own threads since sound's timing needs to be super precise. So when working with ofSoundStream be careful not to use any openGL calls and in general apply the same logic as if you where inside the threadedFunction of an ofThread. We'll see more about this in the next sections.

17.4 ofMutex

Before we started the openGL and threads section we were talking about how accessing the same memory area from 2 different threads can cause problems. This mostly occurs if we write from one of the threads causing the data structure to move in memory or make a location invalid.

To avoid that we need something that allows to access that data to only one thread simultaneously. For that we use something called mutex. When one thread want's to access the shared data, it locks the mutex and when a mutex is locked any other thread trying to lock it will get blocked there until the mutex is unlocked again. You can think of this as some kind of token that each thread needs to have to be able to access the shared memory.

Imagine you are with a group of people building a tower of cards, if more than one at the same time tries to put cards on it it's very possible that it'll collapse so to avoid that, anyone who wants to put a card on the tower, needs to have a small stone, that stone gives them permission to add cards to the tower and there's only one, so if someone wants to add cards they need to get the stone but if someone else has the stone then they have to wait till the stone is freed. If more than one wants to add cards and the stone is not free they queue, the first one in the queue gets the stone when it's finally freed.

A mutex is something like that, to *get the stone* you call lock on the mutex, once you are done, you call unlock. If some other thread calls lock while another thread is holding it, they are put in to a queue, the first thread that called lock will get the mutex when it's finally unlocked:

```
thread 1: lock mutex
thread 1: pos = access memory to get position to write
thread 2: lock mutex <- now thread 2 will stop it's execution
   till thread 1 unlocks it so better be quick
thread 1: write to pos
thread 1: unlock mutex
thread 2: read memory
thread 2: unlock mutex
```

When we lock a mutex from one thread and another thread tries to lock it, that stops it's execution. For this reason we should try to do only fast operations while we have the mutex locked in order to not lock the execution of the main thread for too long.

In openFrameworks, the ofMutex class allows us to do this kind of locking. The syntax for the previous sequence would be something like:

```
thread 1: mutex.lock();
thread 1: vec.push_back(something);
thread 2: mutex.lock(); // now thread 2 will stop it's execution
   until thread 1 unlocks it so better be quick
thread 1: // end of push_back()
thread 1: mutex.unlock();
thread 2: somevariable = vec[i];
thread 2: mutex.unlock();
```

We just need to call `lock()` and `unlock()` on our ofMutex from the different threads, from `threadedFunction` and from the update/draw loop when we want to access a

piece of shared memory. ofThread actually contains an ofMutex that can be locked using lock()/unlock(), we can use it like:

```
class NumberGenerator{
public:
    void threadedFunction(){
        while (isThreadRunning()){
            lock();
            numbers.push_back(ofRandom(0,1000));
            unlock();
            ofSleepMillis(1000);
        }
    }

    vector<int> numbers;
}

// ofApp.h

NumberGenerator numberGenerator;

// ofApp.cpp

void ofApp::setup(){
    numberGenerator.startThread();
}

void ofApp::update(){
    numberGenerator.lock();
    while(!numberGenerator.empty()){
        cout << numberGenerator.front() << endl;
        numberGenerator.pop_front();
    }
    numberGenerator.unlock();
}
```

As we've said before, when we lock a mutex we stop other threads from accessing it. It is important that we try to keep the lock time as small as possible or else we'll end up stopping the main thread anyway making the use of threads pointless.

17.5 External threads and double buffering

Sometimes we don't have a thread that we've created ourselves, but instead we are using a library that creates it's own thread and calls our application on a callback. Let's see an example with an imaginary video library that calls some function whenever

there's a new frame from the camera, that kind of function is called a callback because some library *calls us back* when something happens, the key and mouse events functions in OF are examples of callbacks.

```
class VideoRenderer{
public:
    void setup(){
        pixels.allocate(640,480,3);
        texture.allocate(640,480,GL_RGB);
        videoLibrary::setCallback(this, &VideoRenderer::frameCB);
        videoLibrary::startCapture(640,480,"RGB");
    }

    void update(){
        if(newFrame){
            texture.loadData(pixels);
            newFrame = false;
        }
    }

    void draw(float x, float y){
        texture.draw(x,y);
    }

    void frameCB(unsigned char * frame, int w, int h){
        pixels.setFromPixels(frame,w,h,3);
        newFrame = true;
    }

    ofPixels pixels;
    bool newFrame;
    ofTexture texture;
}
```

Here, even if we don't use a mutex, our application won't crash. That is because the memory in pixels is preallocated in setup and it's size never changes. For this reason the memory won't move from it's original location. The problem is that both the update and frame_cb functions might be running at the same time so we will probably end up seeing tearing[2]. Tearing is the same kind of effect we can see when we draw to the screen without activating the vertical sync.

To avoid tearing we might want to use a mutex:

```
class VideoRenderer{
public:
    void setup(){
        pixels.allocate(640,480,3);
```

[2] http://en.wikipedia.org/wiki/Screen_tearing

```
        texture.allocate(640,480,GL_RGB);
        videoLibrary::setCallback(this, &VideoRenderer::frameCB);
        videoLibrary::startCapture(640,480,"RGB");
    }

    void update(){
        mutex.lock();
        if(newFrame){
            texture.loadData(pixels);
            newFrame = false;
        }
        mutex.unlock();
    }

    void draw(float x, float y){
        texture.draw(x,y);
    }

    void frameCB(unsigned char * frame, int w, int h){
        mutex.lock();
        pixels.setFromPixels(frame,w,h,3);
        newFrame = true;
        mutex.unlock();
    }

    ofPixels pixels;
    bool newFrame;
    ofTexture texture;
    ofMutex mutex;
}
```

That will solve the tearing, but we are stopping the main thread while the `frameCB` is updating the pixels and stopping the camera thread while the main one is uploading the texture. For small images this is usually ok, but for bigger images we could loose some frames. A possible solution is to use a technique called double or even triple buffering:

```
class VideoRenderer{
public:
    void setup(){
        pixelsBack.allocate(640,480,3);
        pixelsFront.allocate(640,480,3);
        texture.allocate(640,480,GL_RGB);
        videoLibrary::setCallback(this, &VideoRenderer::frameCB);
        videoLibrary::startCapture(640,480,"RGB");
    }

    void update(){
        bool wasNewFrame = false;
```

```
        mutex.lock();
        if(newFrame){
            swap(pixelsFront,pixelsBack);
            newFrame = false;
            wasNewFrame = true;
        }
        mutex.unlock();

        if(wasNewFrame) texture.loadData(pixelsFront);
    }

    void draw(float x, float y){
        texture.draw(x,y);
    }

    void frameCB(unsigned char * frame, int w, int h){
        pixelsBack.setFromPixels(frame,w,h,3);
        mutex.lock();
        newFrame = true;
        mutex.unlock();
    }

    ofPixels pixelsFront, pixelsBack;
    bool newFrame;
    ofTexture texture;
    ofMutex mutex;
}
```

With this we are locking the mutex for a very short time in the frame callback to set `newFrame = true` in the main thread. We do this to check if there's a new frame and then to swap the front and back buffers. `swap` is a c++ standard library function that swaps 2 memory areas so if we swap 2 ints `a` and `b`, `a` will end up having the value of `b` and viceversa, usually this happens by copying the variables but `swap` is overridden for ofPixels and swaps the internal pointers to memory inside `frontPixels` and `backPixels` to point to one another. After calling `swap`, `frontPixels` will be pointing to what `backPixels` was pointing to before, and viceversa. This operation only involves copying the values of a couple of memory addresses plus the size and number of channels. For this reason it's way faster than copying the whole image or uploading to a texture.

Triple buffering is a similar technique that involves using 3 buffers instead of 2 and is useful in some cases. We won't see it in this chapter.

17.6 ofScopedLock

Sometimes we need to lock a function until it returns, or lock for the duration of a full block. That is exactly what a scoped lock does. If you've read the memory chapter you probably remember about what we called initially, *stack semantics*, or RAII Resource Adquisition Is Initialization[3]. A scoped lock makes use of that technique to lock a mutex for the whole duration of the block, even any copy that might happen in the same `return` call if there's one.

For example, the previous example could be turned into:

```
class VideoReader{
public:
    void setup(){
        pixelsBack.allocate(640,480,3);
        pixelsFront.allocate(640,480,3);
        texture.allocate(640,480,GL_RGB);
        videoLibrary::setCallback(&frame_cb);
        videoLibrary::startCapture(640,480,"RGB");
    }

    void update(){
        bool wasNewFrame = false;

        {
        ofScopedLock lock(mutex);
            if(newFrame){
                swap(fontPixels,backPixels);
                newFrame = false;
                wasNewFrame = true;
            }
        }

        if(wasNewFrame) texture.loadData(pixels);
    }

    void draw(float x, float y){
        texture.draw(x,y);
    }

    static void frame_cb(unsigned char * frame, int w, int h){
        pixelsBack.setFromPixels(frame,w,h,3);
        ofScopedLock lock(mutex);
        newFrame = true;
    }

    ofPixels pixels;
```

[3] http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

```
    bool newFrame;
    ofTexture texture;
    ofMutex mutex;
}
```

A ScopedLock is a good way of avoiding problems because we forgot to unlock a mutex and allows us to use the {} to define the duration of the lock which is more natural to C++.

There's one particular case when the only way to properly lock is by using a scoped lock. That's when we want to return a value and keep the function locked until after the value was returned. In that case we can't use a normal lock:

```
ofPixels accessSomeSharedData(){
    ofScopedLock lock(mutex);
    return modifiedPixels(pixels);
}
```

We could make a copy internally and return that later, but with this pattern we avoid a copy and the syntax is shorter.

17.7 Poco::Condition

A condition, in threads terminology, is an object that allows to synchronize 2 threads. The pattern is something like this: one thread waits for something to happen before starting it's processing. When it finishes, instead of finishing the thread, it locks in the condition and waits till there's new data to process. An example of this could be the image loader class we were working with earlier. Instead of starting one thread for every image, we might have a queue of images to load. The main thread adds image paths to that queue and the auxiliary thread loads the images from that queue until it is empty. The auxiliary thread then locks on a condition until there's more images to load.

Such an example would be too long to write in this section, but if you are interested in how something like that might work, take a look at ofxThreadedImageLoaded (which does just that).

Instead let's see a simple example. Imagine a class where we can push urls to pings addresses in a different thread. Something like:

```
class ThreadedHTTPPing: public ofThread{
public:
    void pingServer(string url){
        mutex.lock();
        queueUrls.push(url);
```

```
        mutex.unlock();
    }

    void threadedFunction(){
        while(isThreadRunning()){
            mutex.lock();
            string url;
            if(queueUrls.empty()){
                url = queueUrls.front();
                queueUrls.pop();
            }
            mutex.unlock();
            if(url != ""){
                ofHttpUrlLoad(url);
            }
        }
    }

private:
    queue<string> queueUrls;
}
```

The problem with that example is that the auxiliary thread keeps running as fast as possible in a loop, consuming a whole CPU core from our computer which is not a very good idea.

A typical solution to this problem is to sleep for a while at the end of each cycle like:

```
class ThreadedHTTPPing: public ofThread{
public:
    void pingServer(string url){
        mutex.lock();
        queueUrls.push(url);
        mutex.unlock();
    }

    void threadedFunction(){
        while(isThreadRunning()){
            mutex.lock();
            string url;
            if(queueUrls.empty()){
                url = queueUrls.front();
                queueUrls.pop();
            }
            mutex.unlock();
            if(url != ""){
                ofHttpUrlLoad(url);
            }
            ofSleepMillis(100);
```

```
        }
    }

private:
    queue<string> queueUrls;
}
```

That alleviates the problem slightly but not completely. The thread won't consume as much CPU now, but it sleeps for an unnecesarily while when there's still urls to load. It also continues to run in the background even when there's no more urls to ping. Specially in small devices powered by batteries, like a phone, this pattern would drain the battery in a few hours.

The best solution to this problem is to use a condition:

```
class ThreadedHTTPPing: public ofThread{
    void pingServer(string url){
        mutex.lock();
        queueUrls.push(url);
        condition.signal();
        mutex.unlock();
    }

    void threadedFunction(){
        while(isThreadRunning()){
            mutex.lock();
            if (queue.empty()){
                condition.wait(mutex);
            }
            string url = queueUrls.front();
            queueUrls.pop();
            mutex.unlock();

            ofHttpUrlLoad(url);
        }
    }

private:
    Poco::Condition condition;
    queue<string> queueUrls;
}
```

Before we call `condition.wait(mutex)` the mutex needs to be locked, then the condition unlocks the mutex and blocks the execution of that thread until `condition.signal()` is called. When the condition awakens the thread because it's been signaled, it locks the mutex again and continues the execution. We can read the queue without problem because we know that the other thread won't be able to access it. We copy the next url to ping and unlock the mutex to keep the lock time to

a minimum. Then outside the lock we ping the server and start the process again.

Whenever the queue gets emptied the condition will block the execution of the thread to avoid it from running in the background.

17.8 Conclusion

As we've seen threads are a powerfull tool to allow for several tasks to happen simultaneously in the same application. They are also hard to use, the main problem is usually accessing shared resouces, usually shared memory. We've only seen one specific case, how to use threads to do background tasks that will pause the execution of the main task, there's other cases where we can parallelize 1 task by dividing it in small subtasks like for example doing some image operation by dividing the image in for subregions and assigning a thread to each. For those cases there's special libraries that make the syntax easier, OpenCv for example can do some operations using more than one core through TBB[4] and there's libraries like the same TBB or OpenMP[5] that allow to specify that a loop should be divided and run simultaneol¡usly in more than one core

[4] https://www.threadingbuildingblocks.org/
[5] http://openmp.org/wp/

18 ofxiOS

by Lukasz Karluk[1]

18.1 OpenFrameworks on iOS devices.

18.2 Intro

The beauty behind OpenFrameworks is its cross-platform nature and the ability to run the same code on your desktop and mobile devices while achieving the same results.

Support for the iPhone in OpenFrameworks started when the very early iPhones (iPhone 2 or 3 - fact check!) were being released. Some clever people from the OF community realised that the iPhone supported OpenGL which also happened to be what OF was using for rendering graphics, and even more importantly, that C++ code could be mixed with Obj-C code. With these two very important pieces of the puzzle in place it was a matter of tweaking the core OF code to begin it's support for iOS devices. It was then when ofxiPhone was born.

Since then Apple have released a number of other devices like the iPad and so the ofxiPhone title became less accurate and was eventually changed to ofxiOS, OpenFrameworks support for all iOS devices.

18.3 Intro to Objective-C

We've briefly mentioned Objective-C or Obj-C for short and some people may know or may not know what it is. Obj-C is the main programming language used by Apple for OSX and iOS systems. Obj-C is also a superset of the C programming language which makes it possible to compile C and C++ code with an Obj-C compiler, which means its possible to mix OF C++ code with native Obj-C code.

Some people right about now may be letting off long desperate sighs, thinking - "I'm just starting to get my head around C++ and now I have to learn Obj-C... what tha?!" Lucky for you OpenFrameworks has done all the hard work of keeping most Obj-C

[1] http://www.julapy.com/

Figure 18.1: Figure 1: OF on iPhone.

code hidden so you don't have to worry about it. By using the OpenFrameworks API, you can access most iOS device functionality like the gyro, accelerometer or camera without having to type any Obj-C code what so ever. BUT, the time will come when you're comfortable with the iOS environment and you want to access something really specific, so specific that OF hasn't even considered wrapping for you. In this case you'll need to put your Obj-C hat on and get those hands dirty.

Obj-C syntax can look a little daunting when you first look at it. It certainly scared the hell out of me the first time I saw it. And you may notice its unusually very long which is due to it's very explicit nature, meaning that all function names are very descriptive of the functionality they perform and all those words add up. You may think that it would take longer to work with an explicit language but its actually the opposite, because the functions are easier to find and in XCode you can usually start typing the function you are looking for and XCode will give you a list of suggestions and complete the function name for you.

18.3.1 Obj-C Class structure

Like in C++, in Obj-C your code is broken down into two files, the header file and the implementation file. The header file still has the same .h extension but a implementation file has a .m extension. Here is a very basic example of how these two files look like.

I created a class called MyClass which extends a UIView class.

In the header (.h) file below we need to define our class interface and we do this between the `@interface` and `@end` tags. The class interface defines instance variables and public methods, but for now we're going to leave it empty.

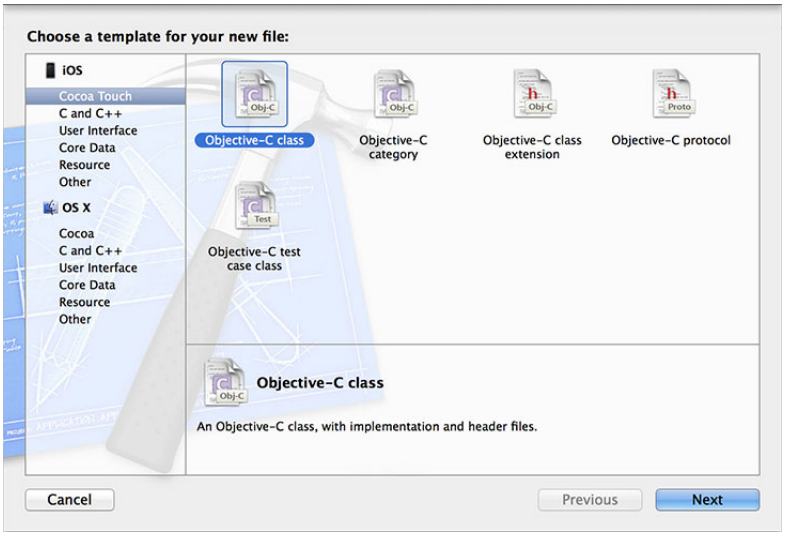
```
@interface MyClass : NSObject {
    //
}

@end
```

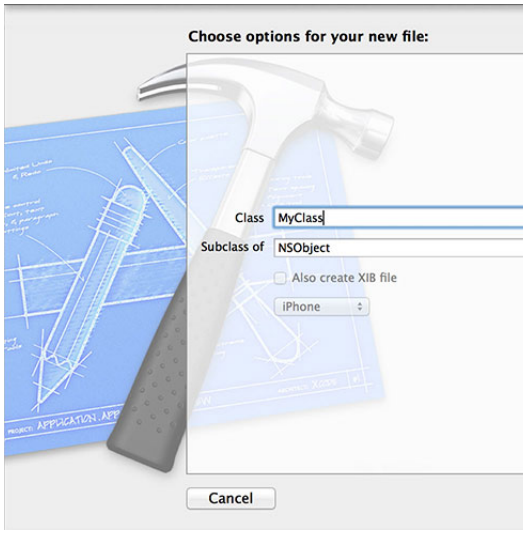
The implementation (.m) file is where you add your actual code for the methods defined in the header file. The first thing you need to do is import your `MyClass.h` header file which defines the structure and then write your implementation between the `@implementation` and `@end` tags.

```
#import "MyClass.h"

@implementation MyClass

@end
```

18.3.2 Make new Obj-C Class in XCode

Nice thing about XCode is that it makes programming easier by creating the basic structure of a class when you first create it, so you don't have to type out the above code structure everytime. By going to File menu, selecting New and then File... a dialogue will appear showing all the files that XCode can create for you. Select `Objective-C class` and another dialogue will appear where you can name your new Class and specify which Subclass it will extend. MyClass will then be automatically generated, ready for you to enter your code into.





18.3.3 Variables and Methods

So now that we have the bare bones of our class, lets add some methods and variables to it so it actually does something. Lets start simple and say that MyClass contains two string variables, one for my first name and one for my last name. We will also want to create some methods for setting and retrieving these variables from the class.

MyClass header (.h) file now looks like this,

```
@interface MyClass : NSObject {
    NSString * firstName;
    NSString * lastName;
}

- (void)setFirstName:(NSString *)nameStr;
- (void)setLastName:(NSString *)nameStr;

- (NSString *)getFirstName;
- (NSString *)getLastName;
```

```
@end
```

Inside MyClass interface, you can now see two `NSString` variables being defined, `firstName` and `lastName`. `NSString` is the Obj-C equivalent of `string` in C++. The `*` means that `firstName` and `lastName` are pointers to `NSString` objects. In Obj-C, all instance variables are private by default, which means you can not access them directly from outside of the class object, so you need to create accessor methods to set and get these values.

Lets look at how methods are structured in Obj-C and take `setFirstName` as an example,

```
- (void)setFirstName:(NSString *)nameStr;
```

The very first thing that comes before every Obj-C method is a dash `-` or a plus `+`. A dash means that the method is a instance method and a plus means that the method is a class method. Next in line we have the return type of the method. In this instance method we are setting the first name and not returning any value, so therefor the return type is `void`. Next is the name of the method `setFirstName` and then separated by a colon `:` are the variables that we pass into the method. In this example we are passing in a `nameStr` variable and we have to specify the type of that variable which is `NSString`.

Now that we have defined our variables and methods in the class interface, lets look at the implementation file where our functional code will live.

```
#import "MyClass.h"

@implementation MyClass

- (void)setFirstName:(NSString *)nameStr {
    [firstName autorelease];
    firstName = [nameStr retain];
}

- (void)setLastName:(NSString *)nameStr {
    [lastName autorelease];
    lastName = [nameStr retain];
}

- (NSString *)getFirstName {
    return firstName;
}

- (NSString *)getLastName {
    return lastName;
}
```

```
@end
```

In terms of structure, the methods look almost exactly the same as the in the class interface, only now each method has curly braces on the end { and } which symbolise the begining and end of the method code. The two getter methods (`getFirstName` and `getLastName`) are pretty straight forward and simply return a pointer to a `NSString` variable. The setter methods (`setFirstName` and `setLastName`) contain code which is more specific to Obj-C and here is where we first touch upon the topic of memory managemen in Obj-C.

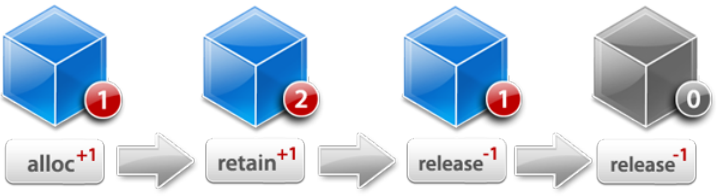
18.3.4 Memory Management

Lets look at what is going on inside the `setFirstName` method.

```
[firstName autorelease];
firstName = [nameStr retain];
```

All that the above is doing is assigning a new string value to `firstName` but it's also making sure the previous value is released before a new one is retained to prevent memory leaks. Calling `autorelease` on a object is telling the object to `release` at some stage in the not too distant future when it is no longer being used, usually at the end of the method when it is no longer needed. We then need to `retain` the new string which you can think of as binding it to the `NSString * firstName` pointer reference. Retaining and releasing objects is at the core of the Obj-C memory management system and is know as reference counting.

The basic theory behind reference counting is that when ever an object is retained, the reference count goes up by +1 and everytime it is released, the reference count is goes down by -1. When the reference count is back down to zero, the object is released from memory.



NEED TO RECREATE THIS DIAGRAM.

There are a couple way of creating an Obj-C object and we'll use the NSString class to demonstrate. Below is a code sample of how a `NSString` object is created using the `alloc` method. Calling `alloc` on a `NSString` class returns a new `NSString` object. A very important thing to note here is that when an object is created using `alloc`, it's reference count is at +1. So behind the scenes, Obj-C has created a new string object and has already called `retain` on the object for us. The final line in the code example is initialising the string object with some text which says `"I'm a string"`.

```
firstName = [NSString alloc];
[firstName initWithString:@"I'm a string"];
```

Another way of creating a string is using `NSString` class methods shown in the code sample below. When a object is created using class method, it is created in a `autorelease` state which means it's reference count is at +1 but because it has been marked as `autorelease`, it will be released from memory soon after if not retained. This is why we need to call `retain` on the new string object, so that we can hold onto its reference and use it somewhere else in our code.

```
firstName = [NSString stringWithString:@"I'm a string"];
[firstName retain];
```

The general rule when it comes to Obj-C memory management is if you create an object using the `alloc` method or call `retain` on a object, you have taken responsibility for that object and sometime in the the future you will have to `release` it.

18.3.5 Ins and Outs

With everything that was just discussed, lets take another look at MyClass which will now include the `init` and `dealloc` methods, the entry and exit points of all Obj-C objects.

```
#import "MyClass.h"

@implementation MyClass

- (id)init {
    self = [super init];
    if(self != nil) {
        firstName = [[NSString alloc] initWithString:@"Lukasz"];
        lastName = [[NSString alloc] initWithString:@"Karluk"];
    }
    return self;
}

- (void)dealloc {
    [firstName release];
    [lastName release];
    [super dealloc];
}

- (void)setFirstName:(NSString *)nameStr {
    [firstName autorelease];
    firstName = [nameStr retain];
}
```

```
- (void)setLastName:(NSString *)nameStr {
    [lastName autorelease];
    lastName = [nameStr retain];
}

- (NSString *)getFirstName {
    return firstName;
}

- (NSString *)getLastName {
    return lastName;
}

@end
```

Both the `init` and `dealloc` methods are already defined in every Obj-C object so we are then extending these methods and overriding their behaviour. The `init` method is the first method to be called on every Obj-C object which makes it the ideal place to intialise your variables. The `init` method always returns a reference of itself with type `id`, which in C++ is equivalent to returning `void *`. Because we are extending the `init` method, we need to make sure we call it's super method first, otherwise the object will not initialise correctly. We then make sure that `[super init]` is called successfully without any issues before initialising variables `firstName` and `lastName`. The last thing an `init` method needs to do is return a reference to itself.

The `dealloc` method is called when an object is about to be released from memory, which makes it the perfect place to release any other memory the object is holding onto. `firstName` and `lastName` objects are released and the last order of business is calling the `[super dealloc]` method, before the object is completely removed from memory.

18.3.6 Properties

Nice thing about Obj-C is that it really makes programming a lot faster by providing syntax shortcuts where possible. In `MyClass` we had to create getter and setter methods for passing in the `firstName` and `lastName` into the object. In Obj-C there is actually a much faster way of declaring getters and setters with the use of properties. Properties are a syntax feature that allow to automatically declare getter and setter accessors. Here is how the `@property` syntax looks like in the header file,

```
@interface MyClass : NSObject {
    NSString * firstName;
    NSString * lastName;
}
```

```
@property (retain) NSString * firstName;
@property (retain) NSString * lastName;

@end
```

You can see that we've ditched the old getter and setter methods and have now replaced it with the `@property` syntax. After the `@property` tag we can also declare some extra setter attributes, where we have `retain` in brackets. This means that every time we use the `firstName` property to set a new value, it will automatically `retain` the new `NSString` which is super handy and means we're writing less code to get the same result.

Next lets jump into the implementation file,

```
#import "MyClass.h"

@implementation MyClass

@synthesize firstName;
@synthesize lastName;

- (id)init {
    self = [super init];
    if(self != nil) {
        self.firstName = [[[NSString alloc]
            initWithString:@"Lukasz"] autorelease];
        self.lastName = [[[NSString alloc] initWithString:@"Karluk"]
            autorelease];
    }
    return self;
}

- (void)dealloc {
    self.firstName = nil;
    self.lastName = nil;
    [super dealloc];
}

@end
```

The first thing we need to do inside the implementation file is `@synthesize` the `@property` that we declared in the header file. `@synthesize` tells the Obj-C compiler to generate the getter and setters methods defined through the `@property` directive in the header.

Now we can access the `firstName` and `lastName` string objects via the getter and setter methods created by the `@property` directive. These methods are generated internally and we don't actually see them but everytime we write `self.firstName` or

`self.lastName`, we are accessing the getter and setter methods.

In the `init` method, one thing that has changed is that an `autorelease` method is being called on the `NSString` object as soon as it is created. This might initially look incorrect as it appears that we are creating an object, retaining it and then releasing it which will bring the reference count back down to zero and means the object will be released from memory. But we have to realise that we are using the setter method created by the `@property` directive which automatically retains the object. This means the final reference count will be +1.

In the `dealloc` method you will notice that `firstName` and `lastName` are not actually being released but are set to `nil` via the `@property` setter. Behind the scenes, when the setter receives a `nil` value it first checks if the object is valid and if so it automatically calls `release` on the object and invalidates the object by setting it to nil. If we were to write out this logic it would look like this,

```
if(firstName != nil) {
    [firstName release];
    firstName = nil;
}
```

Properties definitely take a little while to get used to but when mastered are very powerful tool to faster and flexible coding.

18.3.7 Delegates

18.3.8 Automatic Reference Counting (ARC)

All this talk of memory management can get pretty heavy, so you'll be happy to know that Obj-C have made programming easier using Automatic Reference Countng (ARC). ARC does all the memory management for you so you no longer have to worry about retaining and releasing objects, its all done by the compiler. ARC works by looking at your code at compile time and making sure that each object is retained for as long as it needs to be but also that its released as soon as it no longer used.

By default ARC is turned off inside ofxiOS XCode projects, but can be easily turned on in the project's Build Settings. Worth noting is that even though ARC can be turned on, ofxiOS source files are still compiled with non-ARC. ARC is only applied to Obj-C files in the main XCode project.

When ARC is turned on, it is possible to specify which Obj-C class should use ARC and which should use regular memory management. You can disable ARC for a specific class using the -fno-objc-arc compiler flag for that class.

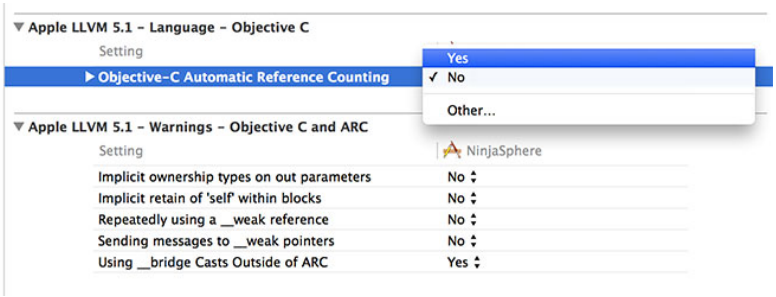


Figure 18.2: Figure 1: OF on iPhone.

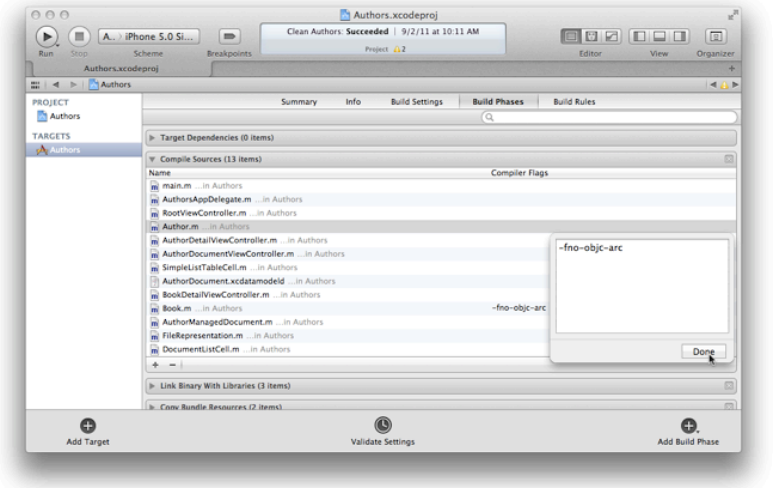


Figure 18.3: Figure 1: OF on iPhone.

18.3.9 Mixing Obj-C and C++ (Objective-C++)

Since both the Obj-C and C++ languages are a subset of the C language, it is possible to mix the two together. First of all before you start typing C++ code into your Obj-C classes or vise versa you need to rename your implementation file extension from (`*.m`) => (`*.mm`). This lets the XCode compiler know that the file is a combination of Obj-C and C++.

`ofApp.mm` by default is already to setup this way so you can start using Obj-C code inside your app. One example of this might be that you would like to use UIKit to add some kind of user interface over the top of your ofApp. There are many very useful possibilities of mixing Obj-C with C++ and we'll go into more detail later in this chapter.

18.3.10 TODO

- what does objective-C look like?
- differences between C++ and Obj-C (string, arrays)
- brief overview of Obj-C memory management compared to C++ (retain/release and ARC)
- How C++ and Obj-C can be mixed together. Mention .mm files.

good reference => http://cocoadevcentral.com/d/learn_objectivec/

18.4 Under the Hood

UIKit is the backbone of all iOS apps. It is a collection of classes or framework that provide a standardised structure for creating and running applications. UIKit provides the skeleton structure into which you can insert your custom application code and makes it easy to receive system events like device orientation changes or memory warning as two examples amongst many.

UIKit organises its classes using the MVC (model-view-controller) design pattern. When you get into iOS programming you will see the MVC patterns everywhere, especially when working with UIViewControllers. MVC breaks up code into one of the three categories and makes the code more extensible and reusable.

When you run a iOS app, it always begins with the UIApplication class which listens for system events and passes them into the app code for further handling. The first class you can start writing your own code into is the Application Delegate. The App Delegate is responsible for creating and managing the UIWindow as well as the root UIViewController, two very important objects in the iOS app structure. UIWindow's job is to coordinate and display content on the screen. As for a UIViewController, you can think of it as a single app screen and using the this analogy the root UIViewController

can be thought of as the home screen for the app. The root UIViewController is the bottom most view controller on top of which you can stack other view controllers, aptly named the view controller stack. When stacking UIViewController objects on top of one another you get the beginnings of an app. You now have a few screens with different UIView objects that you can navigate between.

(explain UIViews)

So that's the super compressed summary which is only skiming the surface of iOS development. It can be a very steap learning curve and this is where ofxiOS comes to the rescue as an elevator that takes you straight to the top of that curve. ofxiOS allows you to make native iOS apps using OpenFrameworks without knowing anything about UIKit or Obj-C at all.

Inside ofxiOS are classes which extend UIKit and take care of creating the iOS application structure. The three core classes are ofxiOSAppDelegate which extends the application delegate, ofxiOSViewController which extends a UIViewController and is the root view controller for an OF application, and ofxiOSEAGLView which extends a UIView and is the view to which OpenGL content is drawn to.

ofxiOSAppDelegate is mainly in responsible for listening to and handling global events like orientation changes, memory warnings, and events for when the application is exited or moves to the background state. ofxiOSAppDelegate alerts the ofApp that that these events have happened and it is then up to the programmer to handle these events as they chose inside the ofApp.

ofxiOSViewController is the OF UIViewController and like the name suggest, its main responsibility is to create and control the OF UIView. It also takes care of orientation changes and can rotate an OF application to match the orientation changes on the device.

ofxiOSEAGLView is the OF UIView which displays all OF rendered content. ofxiOSEAGLView is respinsible for creating a ESRenderer which encapsulates low-level OpenGL setup and makes it possible to render OpenGL graphics into a UIView. ofxiOSEAGLView also listens out for touch events which it passes into the ofApp to be handled by the programmer.

TODO // need to make a diagram to visualise all this.

18.4.1 ofxiOSApp

When you open up a empty ofxiOS project you will immediatly notice some differences in the ofApp header file. You will see that ofApp extends ofxiOSApp instead of ofBaseApp as you would see when running a desktop app. This is because iOS apps and desktop apps are slightly different, desktop apps receive mouse and keyboard events and iOS apps receive touch events as well a orientation events and memory warnings.

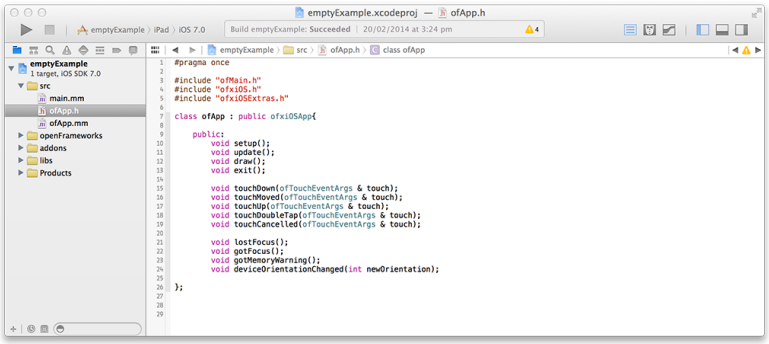


Figure 18.4: Figure 1: OF on iPhone.

To handle these new events, new methods had to be defined inside ofxiOSApp which the ofApp inherits from.

```
void touchDown(ofTouchEventArgs & touch);
void touchMoved(ofTouchEventArgs & touch);
void touchUp(ofTouchEventArgs & touch);
void touchDoubleTap(ofTouchEventArgs & touch);
void touchCancelled(ofTouchEventArgs & touch);
```

Touch events are passed into the ofApp through these methods. Each method receives a ofTouchEventArgs object which contains all the information about the touch event, such as the touch ID and the x and y position of the touch on the screen.

```
void lostFocus();
void gotFocus();
```

Focus events are passed into the ofApp when the application goes from active to inactive state and vise versa. `gotFocus()` method tells the ofApp that the application has becomes active, which happens when the application first launches. `lostFocus()` method tells the ofApp that the application has become inactive which happens when a phone call or SMS interrupts the app or when the user exits the app.

```
void gotMemoryWarning();
```

Each iOS application is allocated a limited amount of memory for it to run. When an application exceeds the allocated amount, the operating systems lets the application know by giving it a memory warning. Memory warnings are passed into the ofApp via the `gotMemoryWarning()` method at which point the application needs to free up some memory otherwise the operating system can terminate the application.

```
void deviceOrientationChanged(int newOrientation);
```

iOS dispatches orientation events when ever the device orientation changes. Orientation events are passed into the ofApp through `deviceOrientationChanged()` method. It is then up to the user to handle these orientation changes as they see fit. iosOrientationExample inside examples/ios/ folder demonstrates how the orientation events can be used.

18.4.2 OpenGL ES and iOS

- Intro > lead to types
- iOS Support for ES 1.1
- iOS Support for ES 2.0
- iOS Support for ES 2.0 > Lead to Hardware
- Apple Devices Hardware Limitations (A7 VS PowerVR)
  - PowerVR
  - A7 > Lead to Device Specific Limitions
- Device Specific Texture Limitations
  - Ref Apple Docs https://developer.apple.com/library/ios/documentation/DeviceInformation/Refer
- Conclusion > Lead to Shaders and Crossovers.

- Shaders using ES2. Crossover between web ES2 shaders and iOS ES2 shaders.
- https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences

18.5 OF & UIKit

- Adding UIViews to an OF app, above and below the OF glView.
- openFrameworks as part of a larger app, several openFrameworks apps in one iOS app
- addons for ofxiOS
- dispatching on main queue from OF to UIKIT using blocks.

http://www.creativeapplications.net/iphone/integrating-native-uikit-to-your-existing-openframeworks-ios-project/

18.6 Media Playback and Capture

A large chunk of ofxiOS support is media playback and capture. ofxiOS has good support for video playback, sound playback, camera capature and sound input. All Obj-C code that makes these features possible is wrapped and abstracted so a regular OF user can continue using the OF API the same way across all supported platforms.

For example, lets say you have a very simple ofApp that plays a video. To achieve this you would use the `ofVideoPlayer` class, create a object instance of the class, call the `loadMovie()` method to load the video file and then call the `play()` method to begin playback of the video. Now to do this across desktop OF apps or iOS OF apps, the code is exactly the same. This is because we are using the `ofVideoPlayer` API which is common across all supported OF platforms. Although the thing to know here is that even though the code works the same way across the different platforms, the actual code used to play a video on OSX and iOS (for example) is very different.

18.6.1 ofxiOSVideoPlayer

`ofxiOSVideoPlayer` is the video player class used for video playback inside ofxiOS. When you're using `ofVideoPlayer` inside a iOS OF project, you are actually using `ofxiOSVideoPlayer`. OF automatically selects the correct video player class to use depending on the platform you are using.

A iOS example that demonstrates the use of the video player on iOS can be found in the folder, examples/ios/moviePlayerExample

When looking inside the ofApp header file you will notice that we are using the `ofxiOSVideoPlayer` class instead the generic `ofVideoPlayer` class. You can use both but it's probably better to use the `ofxiOSVideoPlayer` class instead. The reason being is that `ofxiOSVideoPlayer` has a few extra methods which are specific to iOS, which you may or may not want to use, but it's always good to have that option.

Let go through some of the basic functionality.

To load and play a video it's exactly the same as using the `ofVideoPlayer`.

```
void ofApp::setup() {
    video.loadMovie("hands.m4v");
    video.play();
}
```

On every single frame we need to update the video player.

```
void ofApp::update(){
    video.update();
}
```

Figure 18.5: Figure 1: OF on iPhone.

And to draw the video to screen, we need to first get a reference to the video texture and call draw on the texture object.

```
void ofApp::draw(){
    video.getTexture()->draw(0, 0);
}
```

Now for those extra iOS specific methods.

If you poke around inside `ofxiOSVideoPlayer` you will see a method called `getAVFoundationVideoPlayer()` which is responsible for returning a reference of the underlaying `AVFoundationVideoPlayer`. `AVFoundationVideoPlayer` is the Obj-C implementation for the iOS video player and is the class that sits below `ofxiOSVideoPlayer` and pretty much does all the work. Now, some reasons you may want have accesss to the `AVFoundationVideoPlayer` is that you want to work directly with the Obj-C code to get the most out of iOS video player features or you want to display the video inside a UIView instead of rendering it to OpenGL.

Here we are getting a pointer reference to the `AVFoundationVideoPlayer` which also happens to extends a `UIView`. This means we can add the video player to a UIView hiarchy and display the video natively.

```
AVFoundationVideoPlayer * avVideoPlayer;
avVideoPlayer = (AVFoundationVideoPlayer
    *)video.getAVFoundationVideoPlayer();
[avVideoPlayer setVideoPosition:CGPointMake(0, 240)];
[ofxiOSGetGLParentView() insertSubview:avVideoPlayer.playerView
    belowSubview:controls.view];
```

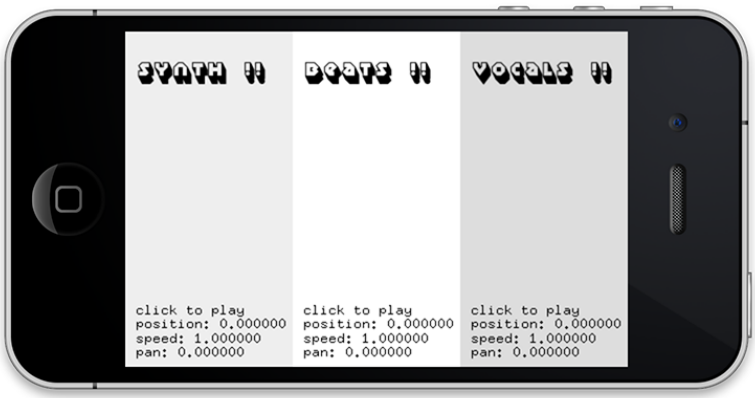
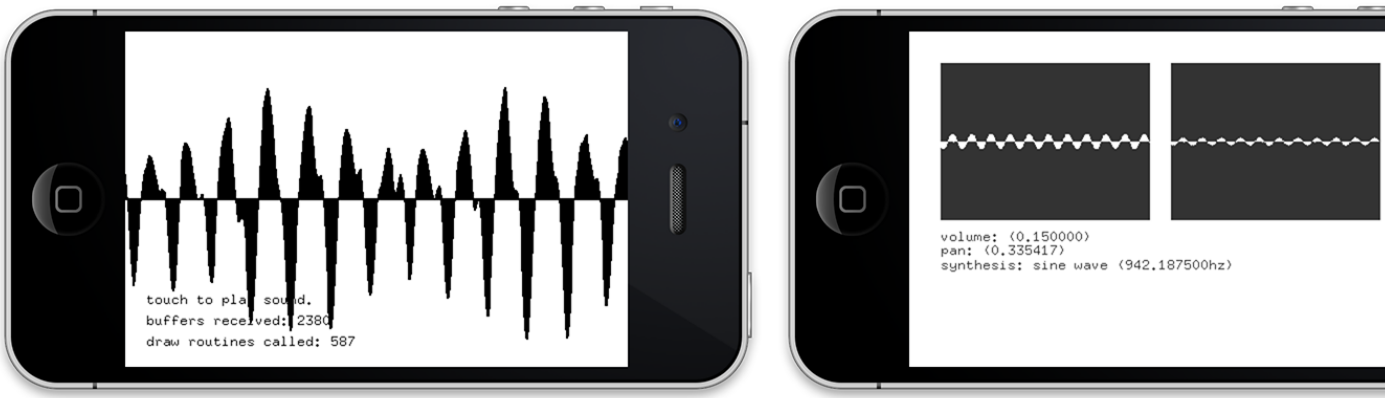


Figure 18.6: Figure 1: OF on iPhone.

18.6.2 ofxiOSVideoGrabber

18.6.3 ofxiOSSoundPlayer and ofxOpenALSoundPlayer

18.6.4 ofxiOSSoundStream



18.7 Life Hacks

- ofxiOS utils, ofxiOSExtras, ofxiOSImagePicker, ofxiOSMapKit etc.

18.8 App Store

- App distribution, preparing your OF app for the app store.

- examples of OF iOS apps already in the app store.

18.9 Case Studies

https://itunes.apple.com/au/app/john-lennon-the-bermuda-tapes/id731652276?mt=8

https://itunes.apple.com/au/app/sadly-by-your-side/id687252928?mt=8

https://itunes.apple.com/us/app/swipin-safari/id635434195?mt=8

https://itunes.apple.com/au/app/tunetrace/id638180873?mt=8

https://itunes.apple.com/au/app/hana/id556557031?mt=8

https://itunes.apple.com/gb/app/starry-night-interactive-animation/id511943282

https://itunes.apple.com/au/app/snake-the-planet!/id528414021?mt=8

https://itunes.apple.com/au/app/horizons/id391748891?mt=8

https://itunes.apple.com/us/app/spelltower/id476500832?mt=8 # C++ 11

the following needs readability improvements (flow of text)... i'll take another pass at it soon

18.10 Blah blah

C++ is a pretty old language, it's been around since XXX, and perhaps because of that (but certainly for many other reasons), it is often seen as archaic, obtuse, or perhaps just plain rubbish by today's standards. Contrary to that, many people believe that it is still offers the best balance of performance and clarity on the coding high street, and (in part thanks to the success of Unix and its mates) has an incredibly strong ecosystem of 3rd party libraries, device support and general acceptance, even up to the point where current shader languages and CUDA use C++ as their language of choice.

Some more modern languages (such as JavaScript and C#) make programs which run in a very different way to C/C++. They have a 'virtual machine', which is a very different type of computer than the one which crunches electronic signals on a circuit board. The virtual machine receives and processes program instructions like a real machine, but allows for all sorts of ideas which don't directly translate to silicon electronics, such as dynamic typing and reflection. The virtual machine abstracts the constraints of the processor away from the thinking of the programmer.

C/C++ does not have a virtual machine, which (for the time being) often gives it a performance edge over these newer languages. It is quite strict in that ultimately the C code itself (somewhere down the chain) translates 1:1 with processor instructions, the

design of the language is inflexible in this way, but clinging to this is the achievement of C++, that code can be both understood naturally by a human, and clearly translate to machine code.

In this chapter we'll look at some of the new patterns in the C++ language introduced in C++11, which retain this promise whilst offering new ways of writing code.

18.11 auto

Perhaps the most used, and simplest new pattern in C++11 is `auto`. You'll love it. And probably won't remember life without it after a day or 2. Consider the following...

```
ofRectangle myRectangle = ofGetCurrentViewport();
ofVec2f rectangleCenter = myRectangle.getCenter();
float rectangleCenterX = rectangleCenter.x;
```

In this code block, we are declaring 3 variables: * `myRectangle` * `rectangleCenter` * `rectangleCenterX`

On each line of code we are:

1. Getting a variable on the right hand side. which is of a certain type (`ofRectangle`, `ofVec2f`, `float` respectively)
2. Declaring a new variable which is explicitly typed to match the value on the right
3. Assigning the value to the variable

What we may notice, is that the type of data on the right and left side of the = is the same. Since C++ is strictly typed (e.g. a function which returns a `float` will always return a `float` no matter what), it is impossible for the value on the right hand side to ever be anything different. The compiler **knows** what type of value the right hand will give, e.g. it knows that on line 1 that on the right hand side of the = is an `ofRectangle`. So perhaps if we were to write something like:

```
auto myRectangle = ofGetCurrentViewport();
auto rectangleCenter = myRectangle.getCenter();
auto rectangleCenterX = rectangleCenter.x;
```

Then the compiler can do some of the coding for us. In fact, thanks to `auto`, we can do this now. This code block compiles to exactly the same result as the first code block did. The compiler notices what's on the right hand side and substitutes in the correct type wherever it sees `auto`.

18.11.0.1 How this helps

Well obviously auto's going to save you keystrokes. Imagine the following:

```
vector<shared_ptr<ofThread> > myVectorOfThreads;
```

```
// the old way
vector<shared_ptr<ofThread>>::iterator firstThreadIterator =
    this->myVectorOfThreads.begin();

// the new way
auto firstThreadIterator = this->myVectorOfThreads.begin();
```

Now this makes the code more readable (by decent humans), but also you could take advantage of `auto` in other ways, for example you could make changes things in the h file, and the cpp file code would automatically correct itself. For example in the h you might change the `vector` to a `list`, or change `shared_ptr<ofThread>` to `ofThread *`. These changes would perpetuate automatically to wherever an `auto` is listening out in the code. Nifty huh?

18.11.1 Watch out for this

18.11.1.1 `auto` is not a new type

Note that the following doesn't work:

```
auto myRectangle = ofGetCurrentViewport();
myRectangle = "look␣mum!␣i'm␣a␣string!!"; // compiler error!
```

Remember that `auto` isn't a type itself, it's not a magic container that can take any kind of thing (such as `var` in C# or another dynamic typed language), it is simply a keyword which gets substituted at compile time, you can imagine that the compiler just writes in for you whatever makes sense on that line where it is. In this case, the first line makes sure that `myRectangle` is an `ofRectangle`, and you can't assign a `string` to an `ofRectangle`.

18.11.1.2 You can't use `auto` in function arguments

Since the `auto` must be implicitly defined by the line of code where it is used, and that this decision is made at compile time, it can not be a function argument, let's see what that means..

Imagine that the following was valid: (**NOTE** : it isn't!)

```
float multiplyBy2(auto number) {
    return number * 2;
}
```

```
int firstNumber = 1;
float secondNumber = 2.0f;

cout << multiplyBy2(firstNumber) << endl;
cout << multiplyBy2(secondNumber) << endl;
```

Now if this code were valid, then the first time the function is called, the `auto` would mean `int`, and the second time it would mean `float`. Therefore saying that the text `auto` is simply substituted with an explicit type where it is written doesn't make sense. So basically **you can't use `auto` in function arguments** (you might want to look into `template` instead, which would automatically generate 2 pieces of code for the 2 different types).

18.11.1.3 You can't use `auto` as a function return type

I'm not sure why, you just can't. It kinda makes sense that you should be able to, but you just can't, move along :).

18.11.2 `const` and references

Let's do a `const auto`:

```
ofRectangle rectangle = ofGetCurrentViewport();
//is the same as
auto rectangle = ofGetCurrentViewport();

//and

const ofRectangle rectangle = ofGetCurrentViewport();
//is the same as
const auto rectangle = ofGetCurrentViewport();
```

Next look at `auto &` for when we want reference types.

```
float x = rectangle.x;
//is the same as
auto x = rectangle.x;

//and

float & x = rectangle.x;
//is the same as
auto & x = rectangle.x;
```

18.11.3 Summary

- Save keystrokes by using `auto` in variable declarations
- `auto` takes the type of the right hand side of the = assignment, and replaces the text `auto` with that type at compile time.
- `auto` is not a magic container which can carry any type of data, it simply gets replaced at compile time by whatever is implied in the code
- You can't use `auto` in function arguments, return types and a few other places.
- Use `const auto` for a const type, and `auto &` for a reference type

18.12 for (thing : things)

Consider the following common pattern:

```
vector<ofPixels> mySelfies;
/*
take some sexy snaps
*/

//oh dear, my photos are all in portrait
for(int i=0; i<mySelfies.size(); i++) {
    //rotate them from landscape to portrait
    mySelfies[i].rotate90(1);
}

vector<ofxSnapChat::Friend> myFriends = snapChatClient.getFriends();

//now let's send them to all my friends
for(int i=0; i<myFriends.size(); i++) {
    if (myFriends[i].isHot()) {
        for(int i=0; i<mySelfies.size(); i++) {
            myFriends[i].sendImage(mySelfies[i]);
        }
    }
}
```

Well we're forever doing things like `for(int i=0; i<mySelfies.size(); i++){`, so let's see if we can find a neater way of doing this with `for thing in things` which in C++11 we can write as `for (thing : things)`...

```
vector<ofPixels> mySelfies;
/*
take some sexy snaps
*/
```

```
//oh dear, my photos are all in portrait
for(auto & mySelfie : mySelfies) {
    //rotate them from landscape to portrait
    mySelfie.rotate90(1); // (1)
}

auto myFriends = snapChatClient.getFriends();

//now let's send them to all my friends
for(auto & myFriend : myFriend) {
    if (myFriend.isHot()) {
        for(auto & mySelfie : mySelfies) {
            myFriend.sendImage(mySelfie); // (2)
        }
    }
}
```

Notice that `for(thing : things)` gels so well with `auto`. Also notice that I'm using `auto &` since:

- At `(1)` I want to be able to change the contents of the vector, so I need a reference to the vector item rather than a copy of it.
- At `(2)`, it makes more sense to use a reference rather than a copy because it's computationally cheaper, and means I don't have to allocate new instances of `ofxSnapChat::Friend` (which I presume is quite a complex object, since it can do things like send images over the internet, and understand societal dispositions of what it means to be attractive).

18.12.1 Summary

- Use `for(auto thing : vectorOfThings)`
- This works with all the STL containers (`vector`, `map`, `list`, `deque`, `set`, etc) and in some more purpose built containers (e.g. `ofxGrayCode::DataSet`)
- Often you'll want to do `for(auto & thing : vectorOfThings)` to use a reference rather than a copy

18.13 override

`override` saves you time not by reducing the amount of typing you do, but by reducing the amount of head-scratching you might do when dealing with `virtual` functions. Imagine the following:

```
class BuildingProjectionMapper {
```

```
public:
//...
    virtual void mapTheGreekColumns();
//...
};

class AutoBuildingProjectionMapper : public BuildingProjectionMapper
    {
public:
    void mapTheGreekColums(); // woops, I spelt column incorrectly
};
```

Now if I implement `AutoBuildingProjectionMapper::mapTheGreekColums`, it may never get called, and I may be wondering why my function calls are all being handled by the base class. The problem is that the compiler never told me that the function that I was trying to override didn't exist. Here comes `override` to the rescue.

```
class AutoBuildingProjectionMapper : public BuildingProjectionMapper
    {
public:
    void mapTheGreekColums() override;
};
```

This tells the compiler that I'm intending to override a `virtual` function. In this case, the compiler will tell me that no `virtual` function called `mapTheGreekColums` exists, and that therefore my `override` is faulty. So following the compiler's complaint I can go in and fix the spelling mistake. Then I can get on with making my Projection Mappening on the town library facade.

18.13.1 Summary

- Use the keyword `override` at the end of function definitions in a derived class' h file when you are intending to override a `virtual` function in the base `class`
- The compiler will warn you if your `override` is invalid, which might just save you a lot of time hunting for errors

18.14 Lambda functions

18.14.1 Worker threads

18.14.2 Callbacks

18.14.3 Summary

19 Case Study : Line Segments Space

by Mimi Son and Elliot Woods (Kimchi and Chips)[1]

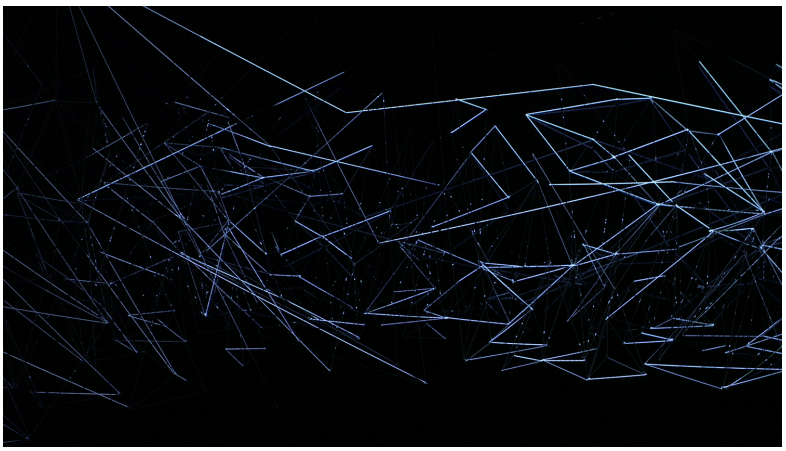


Figure 19.1: Line Segments Space

19.1 Foreward

Line Segments Space is an artwork created by studio Kimchi and Chips[2] (Mimi Son, Elliot Woods), and is the third installation within a series of works titled 'Digital Emulsion', preceded by Lit Tree (2011) and Assembly (2012). The pricipal technician for this project is me, Elliot, the voice of this chapter.

This chapter will discuss some of the technical details of the project to varying levels of detail. Before continuing, I hope that you will first take a little time to view the work in a non-technical context and watch the video on our website at http://kimchiandchips.com/#LSS.

The work contains a number of technical solutions, the principal one being an implementation of the Digital Emulsion technique. Others include a custom CAD application, generative 'brushes' for volumetric content, optical layout of the room and equipment,

[1] http://www.kimchiandchips.com/
[2] http://kimchiandchips.com

sound control and spatialisation, and calibration by a team of client computers. This chapter touches on a few of these challenges.

==perhaps this should be a list of what is actually in the chapter== **[BD: The chapter is short enough that I don't think that this is needed]**

19.2 Artist statement

An architectural web of threads spans a gallery space. It hangs abstract and undefined, a set of thin positive elements segmenting the dark negative space between. Dynamic imaginary forms are articulated into physical volume by the material of this thread, and the semi-material of the light. The visual gravity of the filaments occupying the space between.

A 2D canvas is reduced from a surface piece into a line segment, but then constructed into another dimension, a volume. Light creates contrast and order on the lines to articulate digital matter. Digital forms inhabit the interconnected boundaries of space, moulding visual mass,

The artists reference Picasso's light painting, and Reticuláreas of Gego who's work offers a contemplation of the material and immaterial, time and space, origin and encounter and art and technology.

Kimchi and Chips create technology which paints into different dimensions, bringing new canvases and expanding the possibilities for artists to articulate form. These technologies become a corpus of code, offered without restriction on the internet. Their code is adopted by other artists and corporations, spreading values and ideas implicit with the artists' work into shared cultural idea space. *Line Segments Space* lives both as a dynamic gallery object, and as an encapsulation of the techniques as new computer code and tools on the internet.

[BD: This artist statement may be better at the top. It feels a bit weird to go from talking about the work in the forward, to a brief technical overview, and then an artist statement.]

19.3 Digital Emulsion

Digital Emulsion (or Re-projection Scanning) is a technique which combines 3D scanning with projection mapping in order to create new canvases for visual expression. *Line Segments Space* employs Digital Emulsion to accurately aim light from projectors onto individual threads, whilst also determining the 3D geometry of the web.

The technique combines the use of a video projector and an imaging camera (e.g. DSLR or machine vision camera) to augment a physical object. The steps for this are generally:

1. Calibrate the camera and projector (e.g. using OpenCV)
2. Perform a Structured Light scan of a scene (e.g. using ofxGraycode[3])
3. Triangulate the 3D location of every projector pixel in the scene (e.g. using ofxTriangulate[4])
4. Render a graphical response to the scene using the triangulation data
5. Project this response back onto the scene using the structured light data to perform a pixel-precise mapping between the projector and the scene.

19.3.1 Structured Light

Structured Light refers to a set of techniques which couple projectors with sensors to take visual and spatial readings of the physical world.

A very simple structured light technique is to project a thin white line onto a scene and to take a photo of it. Within the photo, we can see that the line kinks and bends within the cameras image as it passes over 3D features. Using some trigonometry we could perhaps calculate something about the 3D shape of the object based on the displacement of this line.

(insert photo of line projected onto object e.g. http://www.david-3d.com/gfx/slides/4.jpg)
[BD: This photo illustrates Structered Light well! I would use it.]

If we took many images (e.g. a video) whilst moving the line across the whole scene, then we could recover a lot of 3D information about the scene, and make a mesh (e.g. ofMesh).

Generally for Digital Emulsion, we use a structured light technique called Graycode Structured Light. If you're interested in learning more, I suggest checking out either ofxGraycode[5] or David laser scanner[6] (a free to download standalone scanning app which employs structured light).

The specific advantage of using Graycode (rather than 3-phase) structured light for Digital Emulsion projects, is that it gives you accurate information of the location of the **projector's pixels** rather than of the **camera's pixels**. Folowing the Graycode scan, we can now consider that our projector's pixels are sensing the scene but are still also controllable as visible pixels, that they in fact sensor-pixels, also known as 'sexels'.

[3] https://github.com/elliotwoods/ofxGraycode
[4] https://github.com/elliotwoods/ofxTriangulate
[5] http://github.com/elliotwoods/ofxGraycode
[6] http://www.david-3d.com/

19.4 Technical solution

19.4.1 Constraints

The first presentation of *Line Segments Space* was at Seoul Art Space Gumcheon between September and October 2013, the exhibition had the following constraints:

- 1 week installation time
- 4 week run time
- Temporary room built by gallery
- Limited production budget from gallery for material and equipment costs, other costs covered by artists

19.4.2 System overview

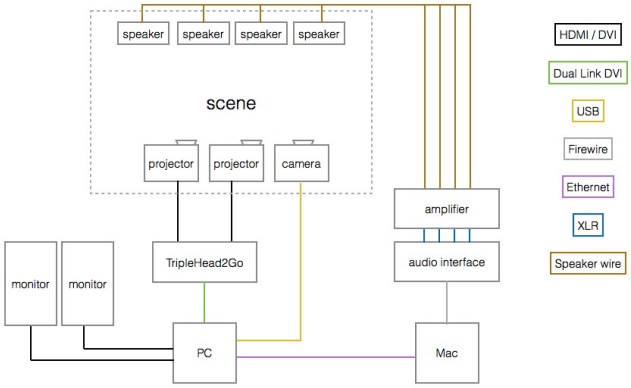


Figure 19.2: System Diagram

==please redraw==

19.4.2.1 Software frameworks

Generally I split processes into 2 categories:

[BD: Here, the difference between online + offline tasks is slightly muddled. Perhaps it is because of the associations with those words. Online + offline is slightly confusing. Perhaps outlining your definition for them here in more depth would be helpful.]

Online : * The task is performed with the installation hardware * Other people are likely to be involved in the process * It's best if software edits can be made quickly and freely * Edits are made whilst continuously observing the output (like a pilot manouvering a plane)

An example of an online process might be the final runtime of the installation.

Offline : * The task can be performed away from the installation hardware. * Some offline tasks may require intense computing time * Often these tasks require more concentration * Edits are made and the results are viewed asynchronously (like a chef tasting the soup)

An example of an offline process would be processing the scan data.

My personal preference is often to use openFrameworks for developing offline tasks, and to use another toolkit called VVVV[7] for developing online processes.

19.4.2.2 Hardware

Component	Reasoning
PC, Windows	PC's are selected for flexible graphics options and for VVVV compatability
GeForce GTX 680	Moderately strong, so good at geavy shader pipelines such as used in this projectIt has
TripleHead2Go	Keeping all output in a single context (i.e, 1 'Display' in Windows) reduces rendering ov
2 portrait monitors	Extra screenspace makes working environments more productive, and a significant por
Mac Mini, OSX	The second computer is for sound design and uses an audio interface which requires

19.5 Design time applications

During the early development stages of the project, we create some applications which are not intended to feed directly into the final work, but exist to facilitate the sketching process of developing the concept and design of the work. These help us to identify possible unexpected directions we may go in, to understand how much effort may be required to realise the work both physically and technically, and to understand the material requirements (e.g. how much rope do we need to buy).

Some examples of these 'design time applications' are: * A simple Digital Emulsion

[7] http://vvvv.org/

scanning app which worked with 2 projectors, a DSLR and After Effects. This was used to develop the tone and manner of the artwork by enabling semi-functional prototypes to be built in the studio. * Several prototypes for calibrating the camera and projectors * A bespoke CAD app for designing the physical web of strings

19.5.1 addLinesToRoom

Let's talk about the CAD app a little more by discussing some of its features and how they are implemented. The full source of this app is up at ==URL for source== and you can download an OSX version at ==URL for build==. This app was written on a long flight, then tweaked as and when it was used to add further features.

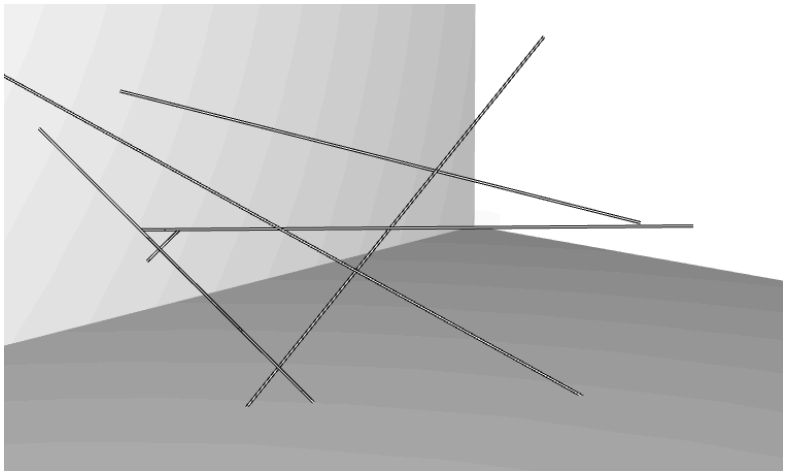
19.5.1.1 Laying down lines

ofxGrabCam

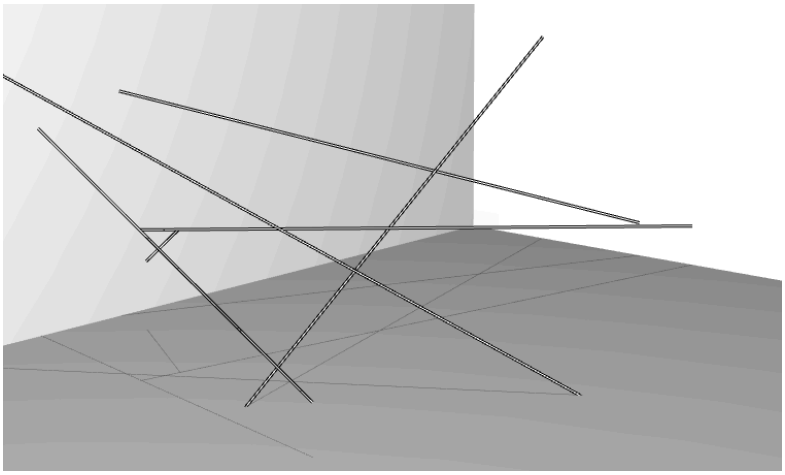
19.5.1.2 Shadows

Editing a 3D scene through a computer monitor is often confusing, especially when we're editing thin lines. We can't naturally see the depth in the scene without constantly moving the camera. Ideally we could see the scene from 2 views simultanaously, enabling us to judge depth.

One simple way of seeing the scene from 2 'views' is to draw shadows into the scene, enabling us to judge depth in the scene much more easily.



Without shadows


With shadows

==suggest arranging these 2 images side by side==

There are a number of standard ways to render shadows in computer graphics, but I chose a super-naive method due to the very simple nature of the scene. Essentially every line is drawn twice, once as a normal 3D line, then again but with the y value clamped to the floor of the room, and the colour set to a dark grey colour.

```
//---------
void Thread::draw(float edgeThickness, ofColor center, ofColor
    border) const {
    const ofVec3f start = this->s;
    const ofVec3f end = this->s + this->t;

    ofPushStyle();

    ofSetLineWidth(edgeThickness);
    ofSetColor(border);
    ofLine(start, end);

    ofSetLineWidth(1.0f);
    ofSetColor(center.r, center.g, center.b);
    ofLine(start, end);

    ofPopStyle();
}

//---------
void Thread::drawShadow(float floorHeight) const {
    ofVec3f start = this->s;
    ofVec3f end = this->t + start;

    //clamp the y value to the floor y value, so that the line
        sticks to the floor
    start.y = floorHeight;
```

```
    end.y = floorHeight;

    ofPushStyle();

    ofSetColor(20, 20, 20, 100);
    ofLine(start, end);
    ofPopStyle();
}
```

19.5.1.3 Shift to zoom

Often it's necessary in an application to perform an action more accurately than can be easily done with the normal mouse/trackpad and screen. In these scenarios, I generally add a "hold [SHIFT] to zoom" mode, which performs an appropriate action to assist the task at hand.

In this case, the [SHIFT] key makes the line wider, and simultaneously the renderer presents a zoomed view in the corner of the screen.

19.5.1.4 Layers feature

Rebuilding gui objects

19.5.1.5 Final notes

The drawing tool tries to mirror the actual physical workflow

Unexpected outcomes (things we had been imagining differently from each other. some layouts turned out to be nearly impossible to make

Apps you write should make you happy whenever you look at it, so make them a touch pretty and use a little subtle colour.

20 Case Study: Choreographies for Humans and Stars

Permanent interactive outdoor installation developed by Daily tous les jours[1] for Montreal's planetarium (2014).

Chapter by Eva Schindling[2] (with help from Pierre Thirion[3])

Figure 20.1: "Choreographies for Humans and Stars" in action at the Montreal's Rio Tinto Alcan Planetarium (January 2014)

20.1 Project Overview

Choreographies for Humans and Stars[4] is a permanent interactive outdoor installation hosted at Montreal's Rio Tinto Alcan Planetarium. The interactive projection on the building's facade invites passers-by to a series of collective dance performances inspired by the different mechanics of planets and stars.

Seven stones anchored into the ground delimit the dance area in front of the projection. A series of instructions on the screen guide participants through a set of choreographies that combine dance and astronomy. The participants use their bodies to understand celestial dynamics like eclipses, forces of attraction and combustion. A

[1] http://dailytouslesjours.com/
[2] http://evsc.net
[3] http://www.21h42.fr
[4] http://www.dailytouslesjours.com/project/choregraphies-pour-des-humains-et-des-etoiles/

camera system tracks the movements across the dance stage and controls the images and animations on the projection. The original image material has been produced through workshops with local kids.

Figure 20.2: The interactive projection invites passers-by to a series of collective dance performances

This chapter documents the various stages of *Choreographies for Humans and Stars* in chronological order. It talks about project logistics, prototyping processes and technological choices, before diving into some OF implementation details towards the end.

20.1.1 Call, Competition and Commission

The project started out as an official call by the Public Art Bureau of the City of Montreal[5], which is in charge of commissioning and maintaning permanent artworks around the city. For the opening of the new planetarium they wanted to commission Montreal's very first interactive and permanent artwork.

The official brief asked for an interactive digital installation utilizing the building facade for projection and simultaneously offering an intervention on the plaza in front of the venue's entrance. The artist needed to ensure that the work lasts a minimum of 3 years in the public space, operating year-round. Sound was excluded, and the work should produce no light pollution. The budget for realizing the project was set at $262.000 CAD (before taxes).

The selection process took ~9 months and included three phases:

1. Request for qualifications (RFQ): jury select 6 artists based on portfolio

[5] http://ville.montreal.qc.ca/artpublic

2. Request for proposals (RFP): jury select 3 finalists based on preliminary artistic concept
3. Final concept: jury selects winner based on complete project proposal (photo montage, video simulation, detailed budget, technical details, production calendar, supplier and collaborator list)

After passing all phases we were officially commissioned by the Public Art Bureau in June 2012.

20.1.2 Timeline

From first brainstorms to final hand-over the mammoth project took an impressive 28 months to complete. That's 10 months longer than the official brief planned for. When you work with that many players (the city, the planetarium, collaborators..) your first and second attempt at a project timeline is bound to fail. A lot of the delay was due to elongated contract negotations with the city, as neither we nor they had negotiated the maintenance of permanent digital artworks before (not your typical bronze statue).

Our more pragmatic goal was to get it all done by November 2013, with the main intention of avoiding all the snow and coldness that comes along with Montreal's winter season. Naturally we slipped right past that goal, and had our big opening amidst lots of snow mid January, with temperatures ranging between -15 to -25.

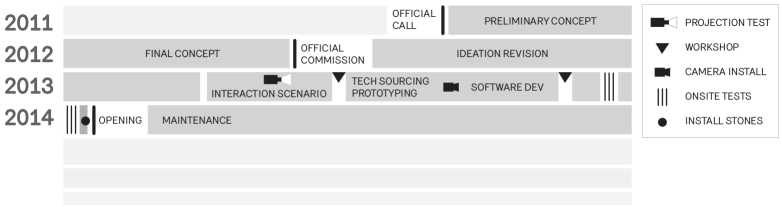


Figure 20.3: The project timeline spanning impressive 28 months

20.1.3 Everyone involved

Credit lists quickly grow long. The internal DTLJ team includes Mouna and Melissa being the main artists with the vision, Eva handling software, Pierre being heavily involved with visual identity and video production, Michael handling the LEDs in the outdoor furniture. The list of external collaborators include a producer (Nicolas), a choreographer (Dana), a technical director (Frederick), a software engineer (Emmanuel), a film animator (Patrick), an industrial design studio (Dikini[6]), a graphic designer (Studio Atelier), a concrete workshop (M3Beton[7]), engineers, a camera man, ...

[6] http://www.studiodikini.com/
[7] http://m3beton.ca/

20.2 Ideation and Prototyping

Choreographies for Humans and Stars is inspired by space as the great unknown. We felt that our role could be to bring a physical experience that would help the planetarium visitors to not only understand but also feel what space is about. One of our early inspirations was the opening scene of the Béla Tarr's movie "*Werckmeister Harmonies*", where a party ends in dancing a solar system waltz with the Earth and the Moon turning around an eclipsing Sun.

Very early in the process we started collaborating with a choreographer and together we explored how participants could use their bodies in ways that mimic celestial dynamics. In a choreography-driven narrative each scene would represent a recognizable space phenomena, instigating a journey from spinning like revolving planets, to lining up to cause an eclipse.

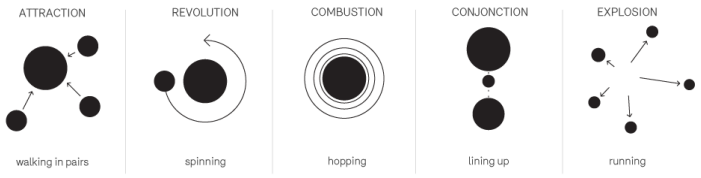


Figure 20.4: The celestial mechanics and their corresponding body movements

The individual choreographies would be communicated with text instructions and should allow participants to dance within a group, yet maintain enough freedom to indulge in personal interpretations.

20.2.1 Challenges in the Interaction design

Once the overall structure and narrative had been decided on we moved into the iterative process of prototyping, testing and finetuning the interaction design. We went from paper drawings to full scale prototypes and had several test sessions with users of all ages.

One of the main challenges of the project was to find a balance between providing interesting reactive visuals while also giving people the freedom to perform choreographies without having their eyes constantly stuck on the screen. Being unable to use sound, all interaction feedback needed to be visual. But in order to encourage true freedom of movement, the colorful images of explosions and shooting stars needed to be tuned down in their *reactiveness* to provide more of a backdrop instead of the main attraction of the piece.

Similar challenging was the task to communicate the instructions to the participants. While some actions could be phrased as one-worders - *"FREEZE!"* - others were more elaborate and cryptic - *"Walk with someone, keep the same distance between you (No hands!)"*. Creating the piece for a bilingual audience also highlighted the possible interpretive differences between English and French instructions. Having test sessions with uninitiated users was important to adjust the exact wording of instructions, and also to refine the timing of the individual scenes. (The factor that eventually ended up influencing the timing the most, was January's outside temperature).

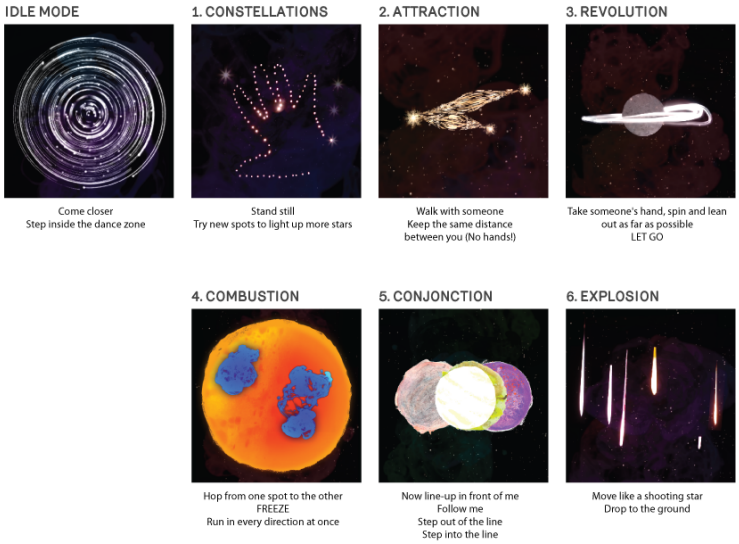


Figure 20.5: The seven scenes representing celestial movements and the instructions animating participants to perform

20.2.2 Outlining the dance zone

The projection being intangible, it was important to give the project a physical presence on the ground. Some sort of tangible intervention that would serve as interface for people to interact, and further represent the project during daylight hours.

At the beginning we imagined a series of stones and platforms arranged to form celestial pathways and encouraged hopping from stone to stone. Yet this would have introduced too many physical obstacles (tripping over, slipping) and severely limited the free movements and interactions in the space. Over the course of prototyping the importance of the physical presence shifted from being an interface to providing a delimiting perimeter around the active dance zone. After going through many design proposals (a stage, a ring, a ballet pole!) we landed on 7 inch-high concrete stones positioned in a circular formation. A single white LED on each stone enhanced their

presence. Installing the underground cabling for those 7 LEDs proved a big challenge and required the $10k rental of a ground-unfreezing device.

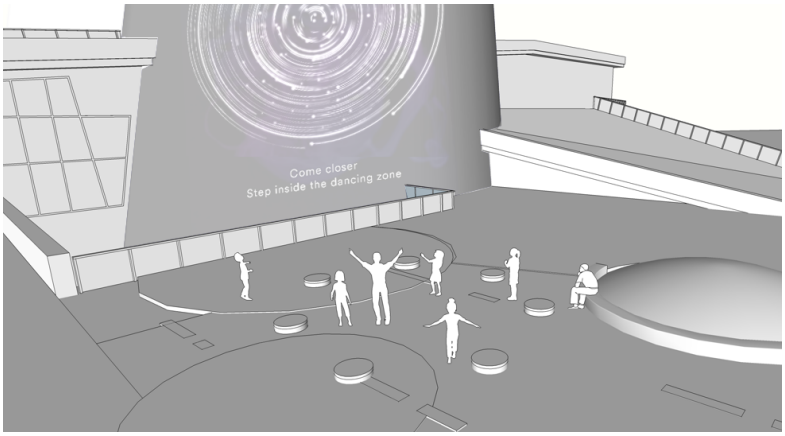


Figure 20.6: The circular dance zone in front of the projection, outlined by 7 concrete stones

20.2.3 Producing video content

Since the project would be permanently installed, we involved locals in the making of these images, aiming to create a sense of ownership within the community that will be living next to the project for many years to come. The image and video content was created using simple analog animation techniques in a series of workshops[8] with local kids aged 7 to 12.

20.3 Finding the Technical Solutions

The technical challenges of creating a public artwork definitely multiply when the project is meant to be permanent. Suddenly you can't just hide cables under cable-trays or choose the perfect location for your camera. Your limits are set by year-round weather conditions, architectural building codes and the intended path of the snow-plow.

20.3.0.1 Put the Projector with the animals

The projection surface on the planetarium is 20 meter high and covered with shiny tiles. The shininess of these tiles was first worrisome, but an early projection test

[8] http://www.dailytouslesjours.com/to-community-and-beyond

settled any doubts of projection quality. To cover a large area of the building surface, we required a high-lumen projector (Barco HDX-W20[9]) which naturally ate almost half the budget. To save further costs - and for weather-protection and easy-access reasons - the projector was placed in the neighbouring building: the Montreal biodome. We had to build a wooden platform for the projector and cut a glass replacement window out of the slightly tinted facade of the biodome. A simple $15 heater now ensures that that window is kept clear of ice and condensation. Additionally we negotiated with the landscape architects the trimming of any trees that were likely to grow into our projection cone over the next 3 years.

Working inside the biodome turned out to be quite entertaining: my access route to our equipment led directly by the penguin compound, and work sessions were accompanied by a constant backdrop of bird chirping.

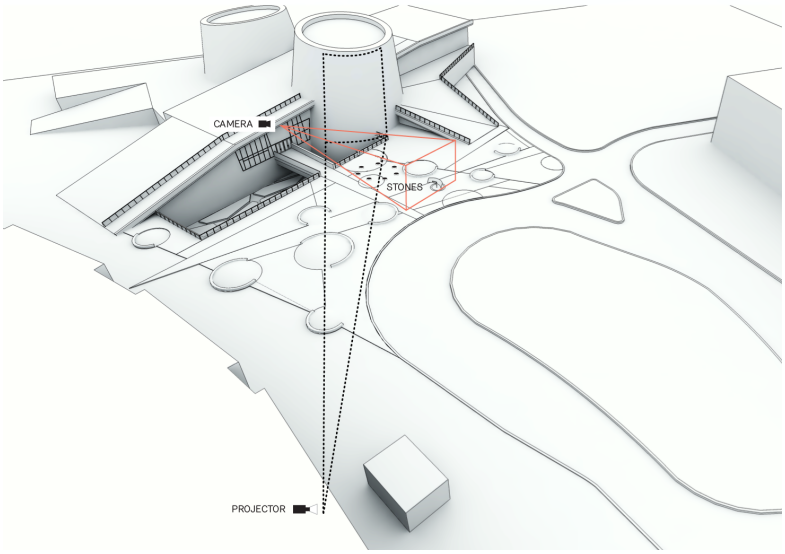


Figure 20.7: The camera is mounted at the planetarium, while the projector is installed in the neighboring biodome

20.3.0.2 Camera style and placement

In an ideal camera tracking scenario you have a controlled indoor environment that provides a clean top-down camera view and lets you create the lighting and background design of your choice. Our site at the planetarium is outdoors and therefore subject to all possible weather conditions. The foreground-background contrast can invert based

[9] http://www.barco.com/en/Products-Solutions/Projectors/Large-venue-projectors/20000-lumens-WUXGA-3-chip-DLP-projector-with-light-on-demand-option.aspx

on if snow covers the dark pavement or not. The general lighting conditions are poor, but the area can temporarily get lit up by the headlights of passing cars.

When first brainstorming technical solutions Kinects were quickly excluded due to a distance of at least 20 meters between dance stage and any possible camera location. More viable options included thermal imaging cameras ($$ and low-res), laser range finders ($$ and limited to one dimension), stereoscopic 3d cameras (too dark environment, also too large distance), and cameras placed at 2 different angles to allow for dynamic mapping of two perspectives into one (double the fun/noise).

Finally we settled on going with one single camera (Basler Scout scA1600-28gm)[10], with a high sensitivity lens (Fuji HF12.5SA-1)[11] for low-light situations and a tracking solution that could convert the 2d information into 3 dimensions. Strict architectural codes prohibited us from placing the camera on top or along the surface of the planetarium. After long negotiations we were limited to placing the camera at a quite low angle slightly to the left of the projection site. Surveillance style, we packed the camera into an ugly weatherproof housing, together with a power supply, a heater and a fan system.

20.3.0.3 Network setup and negotiations

After calculating our camera's bandwidth requirements (resolution 800x600px * framerate 28fps * color depth 8bit * raw compression = 13 MB/sec) we discovered that the local network wouldn't allow us to send the camera data directly to the projector site. We had to place one computer (Intel Core i5 3570K 3.40G/6M/S1155 with 8GB ram, Ubuntu 12.04 LTS) in close proximity to the camera and another computer (Intel Core i7 3770K 3.40G/8M/S1155 with 16GB ram and an Asus GTX680 graphics card, Ubuntu 12.04 LTS) next to the projector. The two sites were only a 3-4 minute footwalk apart, but required keycards and the occasional security guard to open locked doors. In hindsight we would have preferred to stick to the original plan of installing our own fiber optics link to place all computer equipment in the same location.

The network being part of the city network, was heavily controlled, subject to 15min timeout internet access. A couple of request forms later we had a LAN connection between our two computers. VPN access for remote maintenance and remote updates took about 2-3 months, and we are still in negotiation to get SSH access. (Cities protect their networks).

20.3.1 Choice of tracking software

For the tracking software we found a collaborator in Emmanuel Durand, part of the research lab at Society for Arts and Technologies[12]. Emmanuel had developed blob-

[10] http://www.baslerweb.com/products/scout.html?model=130
[11] https://www.fujifilmusa.com/products/optical_devices/machine-vision/2-3-5/hf125sa-1/
[12] http://www.sat.qc.ca/

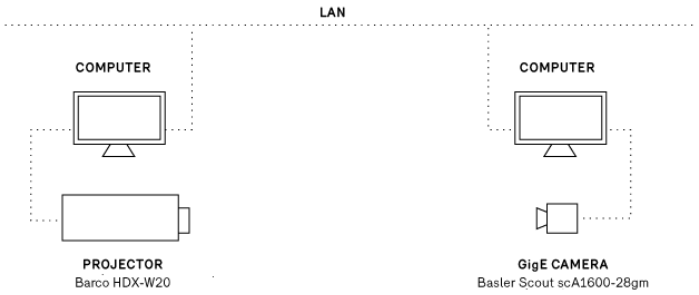


Figure 20.8: The quite simple technical system diagram, the only obstacle was the city-run LAN network

server[13] - a opencv based software to implement various realtime detection algorithms - and was looking for first test projects. For the project he further developed and adapted blobserver to our needs. Blobserver is designed to take in multiple camera or video sources, detect entities and then report its findings via OSC. Live configuration of blobserver can be done through OSC as well, which allows for easy integration with any OSC-friendly software.

20.3.1.1 Method of Tracking

To track participants in our dance zone we used blobserver's HOG detector (histogram of oriented gradients[14]) which learns from a database of human shapes to detect human outlines. The HOG detection is processing optimized by limiting its areas of interest to areas with recent movements, detected by background subtraction.

First tracking tests were done based on publicly available databases of human shapes [link?], but to get better results we created our own image database. We trained the system on images taken with our camera on site, providing the specific camera angle and the same specific background. Future project updates will include further training of the tracking model by including images showing people and environment in their summer-attire.

20.3.1.2 Tracking challenges

The tracking algorithm performs great when dealing with ideal-case scenarios: maximum 3 people, moving at a distance from each other, their silhouettes standing out with high contrast from the background. But naturally, in a public setting without any

[13] https://github.com/paperManu/blobserver
[14] http://en.wikipedia.org/wiki/Histogram_of_oriented_gradients

Figure 20.9: Tracking of participants worked extremely well in the high-contrast winter landscape

sort of supervision you can't control how the audience uses your artwork. As soon as too many people enter the dance zone at once, the system can get confused by overlapping shapes. Similarly, small children, or clothes in the same brightness as the background can cause the detection algorithm to sporadically fail. Configuring the system to forgive those mistakes and let it pretend it still detects those participants, also widens the door for unintentional noise (fake positives).

That balance between detection precision and detection forgiveness and the fact that we deal with a fuzzy system in an uncontrolled environment, took a while to settle in (Side note: this is our first camera tracking project). During early on-site tests we'd settle on tracking settings that performed well, only to come back the next day to a different weather scenario and discover that the same settings didn't apply anymore.

After learning this the hard way, we understood to lower our expectations and also figured out ways to make use of the limitations of the system. Certain instructions ask participants to change into untypical shapes (*"Hop from one spot to the other"*, *"Drop to the ground"*). Instead of training the system on the shapes of people hopping or lying on the ground, we'd use the fact that the detection system lost them, in order to trigger visual reactions.

Similarly we'd detect people spinning - "*Take someone's hand, spin and lean out as far as possible*" - by counting how many times the system loses and re-detects their shapes. As we'd only look for one specific choreography in each scene, we were free to interpret the incoming tracking results as needed.

20.3.2 Choice of visualization software

This project provided the perfect excuse to jump back into openFrameworks. Previous company projects relying on computation mostly lived in the realm of music (max/MSP) or the web (python, node.js). On the rare occasion that visuals were involved, a too short timeline asked for a quick solution (processing). And in general we fall victim to the mistake of over-polluting the never ending prototype and simply turning it into production software.

Figure 20.10: Correct execution of “Drop to the ground” and spinning instructions - in both cases we use the fact that the system loses detection to trigger visual reactions

Choreographies for Humans and Stars with its demands of high-res video animations provided a decent time frame allowing for proper project development and gave me all the right reasons to return to openFrameworks. Helpful at this stage was that most prototyping of content and interaction happened with non-interactive videos produced with video editing software. Any truly interactive prototype had to wait for all the pieces (camera placement on site, trained tracking software) to come together.

OpenFrameworks was chosen as the programming environments because of C++’s fast processing speed, it’s addons, the open-source nature of many similar projects dealing with video and animation content, and mostly its avid and rarely not-helpful community forum. A main reason was also openFrameworks cross-platform ability, as i am personally on a Windows 7 laptop, while the office is reigned over by Macs, and the decision had been made to give it a try with Linux computers for the installation. So being able to jump between the different operating systems while still developing the same software was naturally a plus.

20.3.3 Additional software used

- processing[15] ... for creating OSC communication dummies
- switcher[16] ... to stream video files to shared memory
- libshmdata[17] ... to share video via shared memory
- Photoshop, Final Cut Pro ... to produce and edit image / video content

[15] http://processing.org/
[16] https://code.sat.qc.ca/redmine/projects/switcher
[17] https://github.com/sat-metalab/libshmdata

20.4 Developing the Visualization Software

20.4.1 Development setup

The openFrameworks linux install is build for codeblocks, yet as i have come to like the code editor Sublime Text[18] for its lightweightness and simplicity, i chose to program in Sublime and then compile (`$ make`) and run the program (`$ ./bin/appName`) from the terminal (or terminator[19]). On my win7 laptop i code with Sublime, but compile and run the software from within Codeblocks. Besides its purpose of providing a history of the code, i use github mainly to push code between development and production computers. Alongside Sublime and a bunch of terminal windows, my typical programming setup includes a browser with tabs open on the openFrameworks forum, the openFrameworks documentation page, and github (to search specific function uses).

20.4.2 Quick summary of what the app does

The application navigates the projection through a sequence of 6 scenes that have different visuals and choreography instructions. When participants follow the instructions (and hop, or line-up, or run around ..) the application receives their position data, analyses it for each scene's interaction requirements, and then controls video elements on the projection. In some scenes the participant's location is directly mapped to a video element location, in other scenes participant movements simply cause videos to appear/disappear on screen.

The addons used for the application:

- **ofEvents** ... for controlling the animation
- **ofxOsc** ... for communication between computers
- **ofxOpenCv** ... only for running a perspective transformation
- **ofxGui** ... to build a GUI

20.4.3 Sequential structure

The transition from one scene and segment to the next is either time-dependent (elapsed time comparison with `ofGetUnixTime()`) or based on the participants successful execution of instructions (did they all freeze?). Yet even if no interaction goal is achieved, a set maximum timer will still cause the transition to the next scene.

While the system usually goes through all scenes sequentially, the scene requiring at least 2 participants will be skipped if not enough people are detected in the dance

[18] http://www.sublimetext.com/
[19] http://gnometerminator.blogspot.ca/p/introduction.html

zone. Additionally the system will fall into an idle mode, if no participants have been detected during the last 30 seconds. After the idle mode it restarts the sequence with scene 1.

20.4.4 Incoming tracking data

The tracking software *blobserver* on the camera computer acts as OSC server and is configured to send tracking data to the IP address of the projection computer. Similarly the openFrameworks app registers itself as OSC client on the OSC server and is able to tune the tracking parameters according to specific scene requirements.

To be able to test and simulate the two-way OSC communication i created several processing dummies, which turned out to be very useful for occasions without the full technical setup. (1: dummy to print out received message. 2: dummy to send tracking parameters. 3: dummy to simulate incoming tracking data).

20.4.4.1 Dealing with split message blocks and framerate differences

The OSC messages sent by the tracking software take this format:

```
/blobserver/startFrame      389 1
/blobserver/hog             128 140 287 0.4 1.2 77 4 1
/blobserver/hog             135 418 103 2.2 2.8 20 0 0
/blobserver/hog             136 356 72 0.3 0.2 18 0 0
/blobserver/endFrame
```

Each message with the address line `/blobserver/hog` signifies the tracking data for one recognized shape, and communicates blob id, position, velocity, age, etc. Bounded by the `/blobserver/startFrame` and `/blobserver/endFrame` messages, an arbitrary amount of tracking messages (= current number of *people* recognized) can be received at any time. The frequency of those message blocks depends on the framerate of *blobserver*.

As it can't be guaranteed that *blobserver* and openFrameworks always run on the same framerate, it could happen that multiple tracking updates arrive before the openFrameworks baseApp calls `update()` again. It was therefore necessary to *store away* new incoming data and only trigger the actual processing of that data after all incoming messages have been parsed.

Similarly it could happen that half of the tracking messages are received before, and the other half after the baseApp's `update()` loop. To avoid this splitting of data to cause glitches (system thinks a specific blob-id disappeared, while it just hasn't been updated yet), it was necessary to hold off all processing of data, before at least one `/blobserver/endFrame` has been received during each openFrameworks frame.

20.4.4.2 Storing and updating tracking data

The received tracking data is stored in a map of `Blob` objects `std::map<int,Blob> blobs`. Maps give all the flexibility of vectors (loop, iterator, etc.) but also allow for easy access of entities via their id.

If new tracking data arrives, the system first checks if the blob-id already exists in the map or if it needs to be created. Then it updates the instance with the new data.

```
while (receiver.hasWaitingMessages()) {

    // ...
    if(m.getAddress() == "/blobserver/hog") {

        // parse incoming message arguments
        int blobid = m.getArgAsInt32(0);
        int posx = m.getArgAsInt32(1);
        int posy = m.getArgAsInt32(2);

        // first look if object with that ID already exists
        std::map<int,Blob>::iterator iter = blobs.find(blobid);
        if( iter == blobs.end() ) {
            // didn't exist yet, therefore we create it
            blobs[blobid].id = blobid;
            //....
            ofAddListener( blobs[blobid].onLost, this,
                &planeApp::blobOnLost );
        }

        // now update the blob (old or new)
        Blob* b = &blobs.find(blobid)->second;
        b->setRawPosition(posx, posy);
        b->setVelocity(velx, vely);
        b->age = age;
        //....

    }

}
```

After the new tracking information has been filed away into the blobs map, the blobs map is cleaned of all non-updated members.

20.4.4.3 Perspective transformation

The blob location data received from the tracking software is based on the angled view of the slightly off-center mounted camera. To be able to better tell the participants'

position within the dance stage and their distance to each other, it was necessary to map that skewed 3d location data into a cleaner top-down perspective. Additional rotation of the now 2dimensional data enabled it to easily tell if participants aligned themselves along the axis facing the projection. The skewing and rotating of the data is achieved via `cv::perspectiveTransform`.

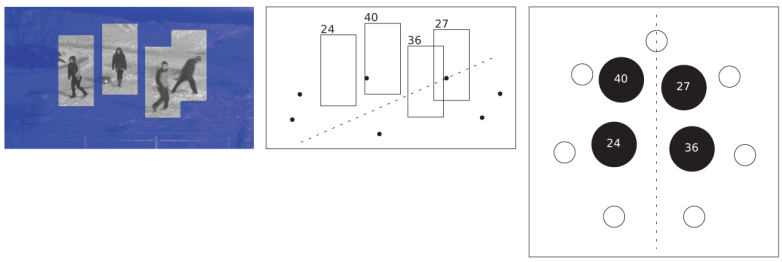


Figure 20.11: Transforming the perspective from the off-center camera to a correctly aligned top-down view

20.4.5 Implementing video content

All our visual raw material exists in the form of videos (and some images). Besides written instructions that are drawn, everything on the projection is direct video display without any effects.

20.4.5.1 The quest for the right codec

When we first received the computer hardware i did a series of performance tests with video files of different codecs to determine how we would prepare our video content. The mistake i made was that i primarily focused on video playback. And once the setup would play multiple video instances at HD resolution, 30fps, in every codec (H.264, photoJPEG, quicktimePNG, animation, raw) while still providing a framerate of 60FPS, i was convinced the top-notch CPU and GPU would be able to handle it all.

What i didn't consider was the load on the processor that comes from loading, decoding, starting, releasing and deleting of multiple video objects within short time spans. This insight finally dawned close to the project's opening date, when more and more photoJPEG and quicktimePNG (alpha) video files were added to the project and the framerate suddenly started to dwindle. Luckily for us, that drop in performance was not clearly visible to the *unknowing* eye, as the project's overall handmade look'n'feel (due to the material's origin from analog stop-motion techniques) allowed for more tolerance regarding the framerate.

Another round of video performance tests (post opening) led to the conclusion that we'd re-encode all videos with the animation codec at the lowest framerate the video

content allowed (5, 10, 15fps). Even though the video files were larger in size, the less strong compression factor minimized the processor's decoding time.

Still, we encountered a few platform and player-dependent idiosyncrasies. an unsolved mystery is still why gstreamer doesn't like certain custom resolutions and displays a green line underneath those video (our fix: find a new resolution).

20.4.5.2 Dynamic video elements

While the background videos appear on schedule and are preloaded by the system, most foreground videos appear dynamically based on participant actions. To be able to handle them easily all dynamic video elements are stored in a vector of shared pointers `std::vector< ofPtr<mediaElement> > fgMedia`. The whole vector of pointers can then be updated and drawn, no matter how few or many videos of shootings stars or planets are currently being displayed. By using `ofPtr` one doesn't need to worry about properly releasing dynamically allocated memory.

Example: Everytime a user stands still long enough during scene 1, a video element displaying a blinking star gets added to the vector:

```
// add new STAR video
fgMedia.push_back(ofPtr<mediaElement>( new
    videoElement("video/stars/STAR_01.mov")));

// set position of video, and press play
( *fgMedia[fgMedia.size()-1] ).setPosition( blobMapToScreen(
    blobs[blobID].position ) );
( *fgMedia[fgMedia.size()-1] ).playVideo();

// link blob to video, to be able to control it later
blobs[blobID].mediaLink = fgMedia[fgMedia.size()-1];
```

20.4.5.3 Preloading versus dynamic loading

In general all video sources that are used in a controlled way (as in: used only as one instance) are preloaded at startup of the software. For video sources that are called up dynamically in possibly multiple instances at once, a combination of two approaches were used:

1. Load the video content when needed … freezes whole app momentarily while loading file, unless the video loading is executed within a thread
2. Preload a large enough vector of multiple instances of the video, then cycle through them with a pointer … allows for faster access/display, yet slows down the application if used for too big or too many video files

20.4.6 Event-driven animation

The control of video elements by the blobs (detected participants) is implemented with `ofEvent()` calls. Events are an good way of giving objects a way of controlling elements in the baseApp without having to query each objects possible states from the baseApp's update() loop, or without having to give objects a pointer to the whole baseApp.

Blob objects have multiple events they can trigger actions in the baseApp:

- **onEnterStage** ... enter dance zone, used to make videos appear
- **onLeaveStage** ... leave dance zone, used to make videos disappear
- **updatePosition** ... called on every frame, to update position-mapped video elements
- **onLost** ... not detected anymore (but still alive!), used for Hop! Run! Drop! video triggers
- **onFreeze** ... stopped moving, used to make videos appear or force transitions (all freeze!)
- **unFreeze** ... started moving again, used to make videos disappear
- **overFreeze** ... hasn't moved for x seconds, used to make constellations appear
- **onSteady** ... at same distance to neighbor for x seconds, used to create star bridges between neighbors
- **onBreakSteady** ... broke steady distance with neighbor, let star bridge disappear
- **prepareToDie** ... make sure to disconnect all connected videos
- ...

`ofEvent` instances are defined in the blob object header:

```
class Blob {

    ofEvent<int> onLost;
    ofEvent<int> onFreeze;
    ofEvent<int> unFreeze;

}
```

When a new blob object is created in the baseApp, `ofAddListener()` connects the object's events to functions within the baseApp.

```
// create new blob
blobs[blobid].id = blobid;

ofAddListener( blobs[blobid].onLost, this, &planeApp::blobOnLost );
ofAddListener( blobs[blobid].onFreeze, this, &planeApp::blobOnFreeze
    );
ofAddListener( blobs[blobid].unFreeze, this, &planeApp::blobUnFreeze
    );
```

```
// ...
```

When a blob then updates and analyses its position, velocity etc. it can trigger those events with `ofNotifyEvent()`.

```
void Blob::evalVelocity(float freezeMaxVel, float freezeMinTime) {

    if ( this->vel < freezeMaxVel ) {
        if ( !frozen ) {
            frozen = true;
            ofNotifyEvent(onFreeze,this->id,this);
        }
    } else {
        if ( frozen ) {
            frozen = false;
            ofNotifyEvent(unFreeze,this->id,this);
        }
    }

}
```

The `ofNotifyEvent()` call then triggers the connected function in the baseApp:

```
void planeApp::blobOnFreeze(int & blobID) {

    if (scene==STARS && blobs[blobID].onStage) {
        // add new STAR video
        fgMedia.push_back(ofPtr<mediaElement>( new
           videoElement("video/stars/STAR_01.mov")));
        // ...
    }

}
```

20.4.7 Debug screen and finetuning interaction

For testing and tuning purposes the application is run on 2 screens: the projection and the debug screen (by using `window.setMultiDisplayFullscreen(true)`). The debug screen shows a visualization of the tracking data, the states of the blobs relevant for the current scene, two GUI panels for finetuning parameters, and a smaller preview of the projection view.

When doing live testruns on site it is important to have all interaction parameters easily accessible via a GUI. Thresholds like: what velocity defines “standing still”, or how exact does the alignment need to be to activate the eclipse - are easier to tune if comparison runs don’t need to get interrupted by software compilation time.

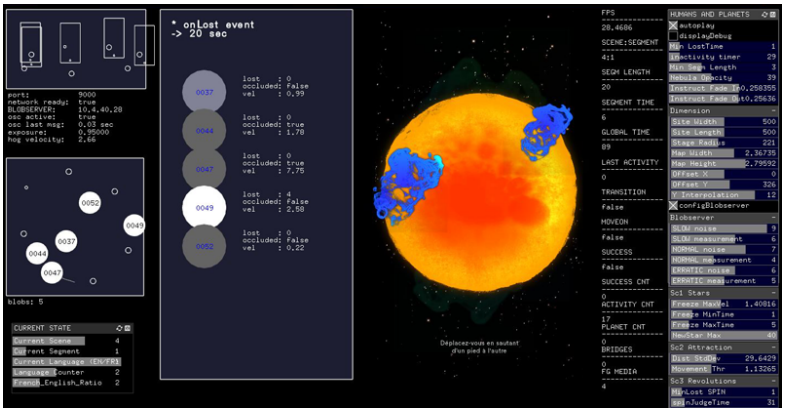


Figure 20.12: The debugscreen showing (from the top left): the abstracted camera view, the transformed top-down view, tracking analysis, the output visuals, system states and tunable parameters

20.5 Fail-safes and dirty fixes

The nights before the opening were naturally used for heavy test sessions that led to some restructuring, some video updates and lots of parameters finetunings. Late night coding over remote desktop while dressing up every 5 minutes to run outside into the cold to test the changes with camera vision and projection - not the best of scenarios. Combined with the natural occurrence of bugs, suddenly the software didn't seem as stable and fast as just a few days ago. A mysterious segmentation fault error kept appearing (not often enough to be easily traceable), but other pressing issues didn't allow for proper investigations into the error's roots.

The opening day had me glued next to the projection computer, ready to hit a button or change a parameter, in case of unexpected crashes or lagging framerates. The next day - before the project went live on its daily schedule of 5pm to midnight - turned into an exercise of setting priorities. Instead of going into debug mode and investigating errors, the main goal was to to keep the app going seamlessly during showtime.

20.5.1 First: Keep your App alive

The one good thing about segmentation faults is that they kill your application very fast. The software crashes and in the next frame you are left with your desktop background (which should be black). The perfect fail-safe solution for this is something like daemontools[20] (linux), which is a background process that monitors your application and restarts it within a second in case it crashes. After setting up supervision with

[20] http://cr.yp.to/daemontools.html

daemontools, the application could crash, but all people would see is a few seconds of black (depending on how long the preloading of video files takes on startup).

20.5.2 Second: Framerate cheats

The second concern was the prevent or hide the drops in framerate that would be caused by too many dynamic videos being active at the same time.

- Erratic framerate variations can be hidden by updating animations with a FPS-dependent value `ofGetLastFrameTime()`
- If the application's memory usage grows over time it probably has hidden memory leaks. To counter a steady drop in framerate it's not a bad idea to regularly terminate the app voluntarily. We found a natural and seamless opportunity to restart our application at the end of each 6-scene sequence.
- By putting realistic limits on the number of your processed objects (blobs, video elements), you can avoid major framerate drops when mysterious glitches suddenly report the simultaneous detection of 100 blobs.

20.5.3 Always: Investigate

In order to understand and fix what's going wrong in your application, it's necessary to find ways of monitoring what's going on under the hood.

- I use `ofLogToFile()` to make each run save out its stack trace to a timestamp-named log file. Being able to go back and search for similar interaction sequences in history, allows me to compare if a certain hack solved a problem.
- When implementing memory-related changes (threaded objects, preloading of multiple videos, etc.) it is good to execute extreme use cases while having an eye on the application's CPU and RAM stats (`$ top`). This allows for a systematic comparison and early spotting of potential bottlenecks.

20.5.4 Finally: Optimize

Once an installation has been open to the public for a few days, it hopefully has revealed all its weak points. After the stress of finding quick fixes to hide those weaknesses - and a few days of healthy mental detachment from the project - it was time to tackle and optimize the software with a clear mind.

- By replacing all video files that can easily also be drawn with opengl, your application will perform better.
- What to implement with threads?

20.6 Results and Reception

At the point of writing this chapter *Choreographies for Humans and Stars* has been on show for 2-3 winter months. The projection on the shiny building looks very stunning and the snow landscape gives the site a beautiful lunar feeling. The project seems successful in addressing different kinds of audiences and manages to engage kids, teens and parents alike.

While for most people the choreography instructions seem easy to follow, we've also observed several people mistaking the 7 concrete stones as interactive trigger elements. The confusion might be due to the delicate LEDs in the stones looking like sensors, or the descriptive text relief sculpted on top of the stones.

Figure 20.13: Participants mistaking the stones as sensor objects

The reception from the audience is very good even though the usage count is not very high yet. Besides at events like Montreal's Nuit Blanche - luring with fires and hot beverages - the winter season with its low temperatures currently prohibits people from hanging out and letting the site become a destination in itself.

For objective analysis of usage and interaction behaviours we are gathering data with a simple logging system. It will be interesting to observe the usage over time and analyse the difference between seasons and succeeding years. Besides looking at number, we are also curious to see how people (the planetarium staff, visitors and passersby) will live with the piece over time.

Figure 20.14: Explosions on the sun are triggered by jumping

Figure 20.15: The final instruction of the cycle asks participants to drop to the ground, exhale and look at the sky

21 Case Study: Anthropocene, an interactive film installation for Greenpeace as part of their field at Glastonbury 2013

21.1 Project Overview

Anthropocene

Adjective

Relating to or denoting the current geological age, viewed as the period during which human activity has been been the dominant influence on climate and the environment.

To see the finished project as part of a wider video all about the Greenpeace Field at Glastonbury 2013, please see the YouTube link below:

http://youtu.be/LwokmKqT0og?t=1m12s

Or an exceprt from inside the dome here:

https://vimeo.com/97113402

All source code can be found here:

https://github.com/HellicarAndLewis/Anthropocene

21.2 The Project

21.2.1 Initial Brief from Client

On 9th April 2013 we were approached by Paul Earnshaw of Greenpeace about an installation as part of Greenpeace's Field at Glastonbury 2013, a large music festival in the South West of England. Another interaction design studio had previously been in place to create a five day experience due to go live for the public duration of the festival on the 25th of June, but a scheduling conflict had emerged that had meant that they had to reluctantly withdraw.

Paul already had a budget and a unique space picked out for the installation, a large geodesic dome:

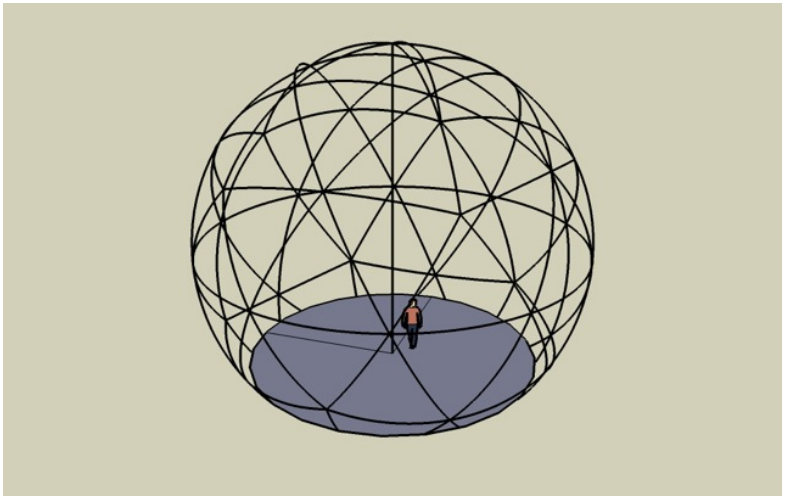


Figure 21.1: External Shell Structure for Installation from Client

21.2.2 Our response

We initially sought out a projector hire firm, who responded with a quote and a plan for a projection setup that met our requirements for maximum visual impact on a budget:

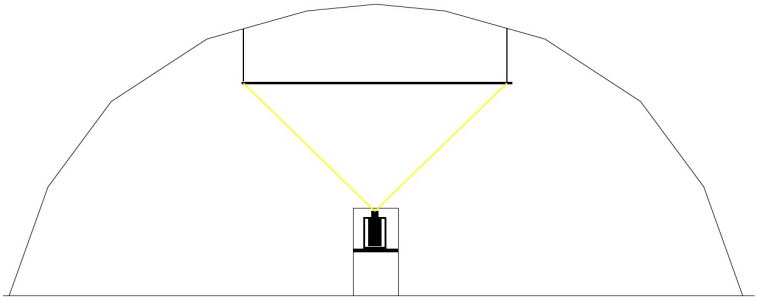


Figure 21.2: Initial Projection Plan from Projector Firm

After some studio thinking, by 16th April we responded with the following document:

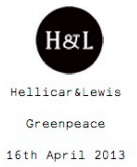


Figure 21.3: First Page of Presentation

We would like this installation to be a relaxing and immersive space. An oasis where the viewer can relax on bean bags looking up at a screen.

We will use a mix of existing Greenpeace footage and a generative soundscape to build a beautiful abstraction of the Arctic.

We would like to project onto the ceiling of the space, using either a rectangular, square or circular projection surface. We will experiment with different projection >shapes and see what fits best aesthetically as well as meeting the budget.

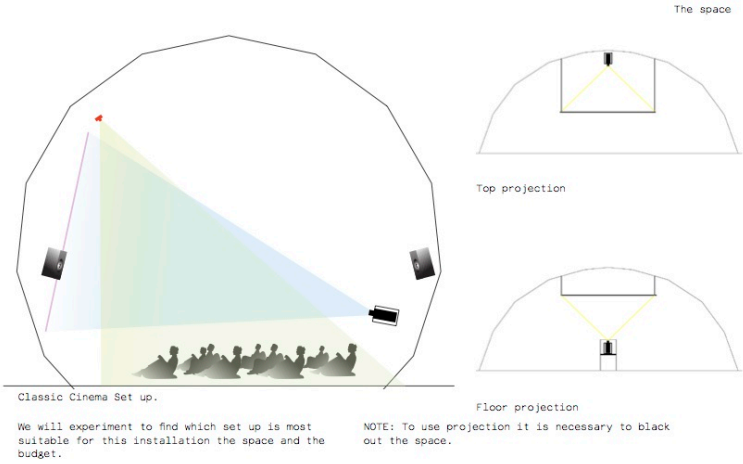
We would like to explore the following ideas within the imagery, sound and feeling of the space.

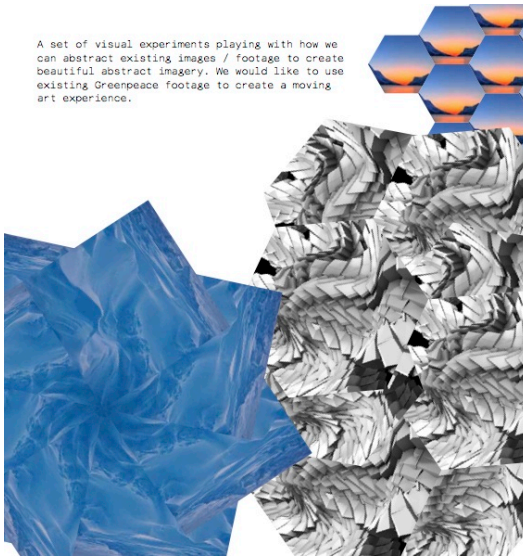
1: The space as a timepiece - trying to have a cycle of sunset, night and dawn - each lasting around five minutes and having a single interaction between the floor and >ceiling that is explored graphically and interactively.

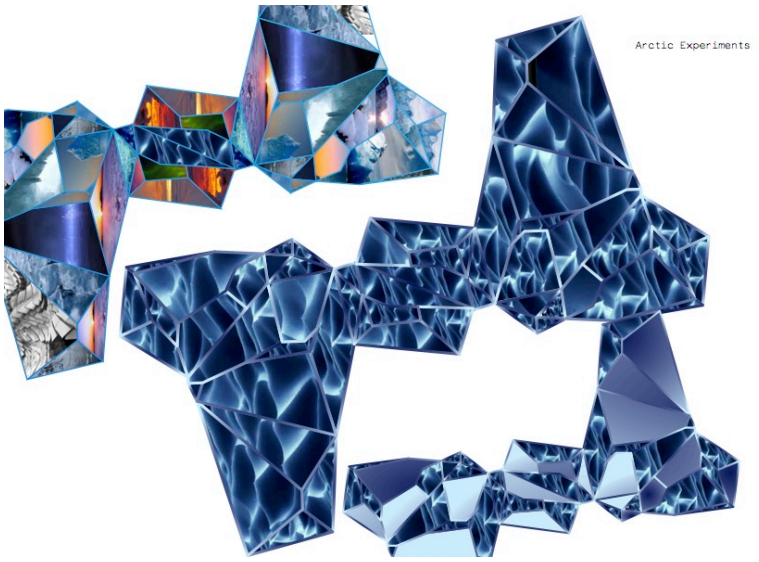
2: Kaleidoscopes, shattering or delaying or time stretching footage. Breaking it up into blocks of time. Arranging in grids, or having different delays in different parts. >The possibility of peoples movement being mirrored into the video playback in interesting ways, playing with time.

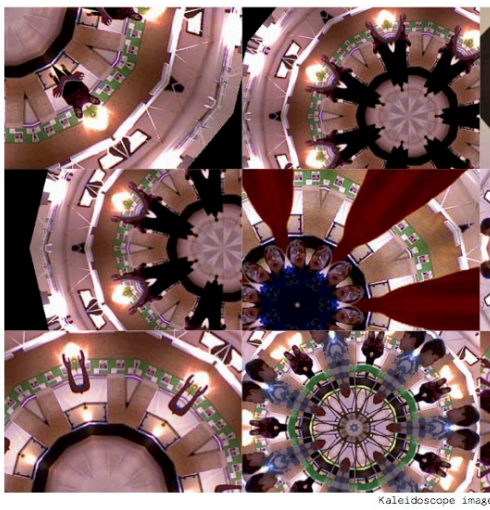
3: Making an oasis away from the rest of the festival that would last around 15 minutes, but raise some points about how the cycle of seasons of the Arctic are being >affected.

4: Generative audio - a four channel speaker system that adds depth and texture the visuals.









On April 30th, we recieved an email from Paul:

"..we would love you to implement your proposal in our main feature of the dome at Glastonbury Festival this year..."

We had the project! Now it was time to get real about the budget, and see if we could get some interesting musical collaborators...

During May, we concentrated on discovering what film footage was available, and finalising the production design and kit list. My business partner Pete Hellicar spent many hours working on the edit of the film, aiming to complete it while other negotiations continued.

21.2.3 Audio negotiations

On May 10th, Pete reached out to our friends at Warp Records to see if any of their artists would be interested in donating their music to the project, and by the nick of project time we had permission from several artists to use their sounds.

21.2.4 Supplier change, Final Budget Negotiations and Interaction Plan

By the end of May, we had changed hardware suppliers to ones already working with Greenpeace on their field, and had found replacement kit for our production. After experimenting with a circular projection screen, we'd arrived at a traditional projector set-up within the dome - a large rectangular projection screen about halfway up the dome wall with seating arranged in front of it. We'd also reached a final budget, and I was ready to start coding.

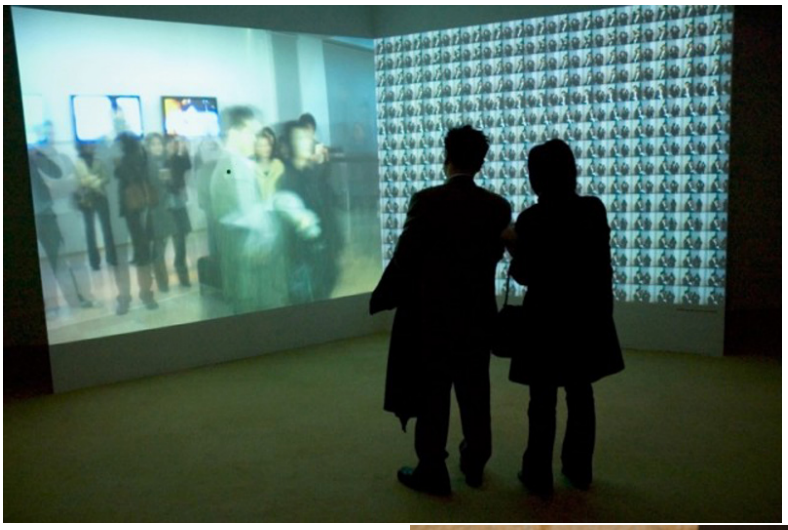
Pete and I had arrived at a final interactive concept after some discussions with Paul, who stated that a "show time" of about 15 minutes was desirable - enough time to get a detailed message across, but not so long as to bore a casual visitor. Pete took his film edit to 15 minutes and had it approved by the Greenpeace team for pacing and content. We decided to use the Microsoft Kinect to allow the openFrameworks application to distort or effect the film footage in real time - based on viewers movements in front of the projection screen. To make the dome a bit more comfortable Paul arranged the donation of several jumbo size bean bags - meaning that visitors could lie comfortably and wave their hands in the air to interact with the film - we angled the Kinect to hopefully pick up unintended user interaction first, surprising users and gently guiding them to stand in front of the bean bags and use their whole bodies to interact. We knew we had to strike a balance between a pre-scripted show and a completely spontaneous one - so we decided on developing several visual looks which could be placed onto a time line for easy repetition, editing and playback. The aim was to get to a system with the reliability of a static linear film and the responsivity of a live "VJ" system - albeit one that used the viewers silhouette rather than pre-rendered matts to affect the edited Greenpeace film.

At the beginning of June 2014 we received the following image from Paul:

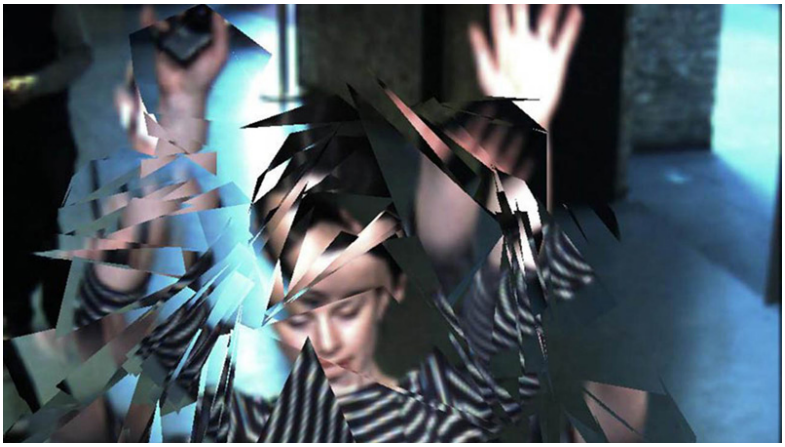
The site awaited us.

Figure 21.4: Site Visit by Client

21.2.5 Interactive Background to Delay Maps, and the possibility of generating a Delay Map from the Kinect Depth Image



We are the time. We are the famous. Created at Fabrica.



Hereafter by United Visual Artists. Feedback by Hellicar&Lewis.

Pete started the project by doing some sketches in Photoshop of how two dimensional angular shapes could “shatter” video footage - I realised that we could make similar effects in real time by using a delay map. This brought me back to previous projects around varying video delay across a whole image - starting with “We are the time. We are the famous”[1] at Fabrica[2], continuing with Hereafter[3] at UnitedVisualArtists[4] and finally Feedback[5] at Hellicar&Lewis[6]. Many people have been interested in this area for some time, Golan Levin[7] has compiled a list of related works[8].

A delay map is simply grey-scale image that is used in combination with a digital video file to decide how much the video file should be delayed on a per-pixel basis. In this projects case a white pixel in a certain position in the delay map meant that there would be zero delay on the corresponding pixel of the video file currently being played back. Conversely, a black pixel in the delay map image would mean the maximum frame delay on the corresponding pixel of the video file. I.e. a completely white delay map image would combine with a video file to play back with zero delay, whilst a black image would give a uniform maximum delay - a linear horizontal grey-scale gradient would give a gradated delay from 0 on the white side to maximum on the black side - with all divisions smoothly displayed in between.

Delay maps are a great way of allowing an art director to quickly “paint” several grey-scale images in Photoshop or some similar image editing program and see the effects of that map on any video file - abstracting away the technical details of the underlying video delay code. This approach of using imagery to control underlying code is a particularly effective technique - making new tools for Art Directors to interface with

[1] http://www.benettongroup.com/40years-press/fabrica_yeux_ouverts.html
[2] http://fabrica.it/
[3] http://uva.co.uk/work/hereafter
[4] http://uva.co.uk/
[5] http://www.hellicarandlewis.com/the-roundhouse/
[6] http://hellicarandlewis.com
[7] http://www.flong.com/
[8] http://www.flong.com/texts/lists/slit_scan/

code using visual techniques rather than syntax and text heavy traditional software engineering techniques.

The breakthrough after this initial thinking was to try to think of what other grey-scale maps I had seen - the live depth image of the Kinect! This would allow peoples 3D silhouettes to create per pixel delay maps that would change in real-time as they moved in front of the 3D sensors of the Microsoft device. The addition of James Georges ofxSlitScan[9] made swapping in and out static grey scale maps very simple, all I had to do was combine the depth map with his existing code on a frame by frame basis.

21.2.6 Actual Timeline

Here are the folder names of all the folders in my GreenpeaceArcticGlastonbury2013 folder.

- 2013_04_11_PlansAndContentFromGreenpeace
- 2013_04_16_ProjectorQuotes
- 2013_04_30_PeteQuoteAndIdeas
- 2013_05_08_GlastoOverviewPlan
- 2013_05_14_PetePlanAndTechList
- 2013_05_20_GuestList
- 2013_05_28_CrewDetailsFromPete
- 2013_05_29_addons
- 2013_05_29_addonsAfterPragmatism
- 2013_05_29_ofxGUIFromDevelopGitHubBranch
- 2013_05_31_AddMaps
- 2013_06_02_BaficInvoice
- 2013_06_03_PeteEffectsFromSomantics
- 2013_06_04_HomeHigherResForPete
- 2013_06_06_CallToActionScript
- 2013_06_12_CrewForFieldReadup
- 2013_06_12_Font
- 2013_06_12_GreenpeaceLogos
- 2013_06_12_MoreCrewBriefing
- 2013_06_13_HuntResult
- 2013_06_13_MoreDurationBits
- 2013_06_13_obviousJimAudioReactiveRing
- 2013_06_16_ofxTimelineVideo
- 2013_06_19_Singleton
- 2013_06_19_VoiceOverOutro

[9] https://github.com/obviousjim/ofxSlitScan

- 2013_06_20_CateringMenu
- 2013_06_20_NewAddonsToTry
- 2013_06_24_CodeForArtFromJeffTimesten
- 2013_06_24_DeadFlock
- 2013_06_24_newFilmAndAudio
- 2013_06_24_ofxAddonsOFXContourUtil
- 2013_07_31_Final50Invoice
- 2013_08_18_ThankYouFromGreenpeace

21.3 Development

21.3.1 Development Hardware and Software setup

MacBook Pro * 15-inch, Mid 2009 * Processor: 3.06 GHz Intel Core 2 Duo * Memory: 4 GB 1067 MHz DDR3 * Graphics: NVIDIA GeForce 9600M GT 512 MB

- XCode for Development
- Chrome for Web Browsing
- Sublime Text for logging

21.3.2 Explanation and Discussion of Development in Detail

21.3.2.1 ofxKinect, as a possible input to ofxSlitScan

One of the benefits of using a platform like openFrameworks is that when people do release extras or “addons” they inevitably interface with the core - interesting results can be found by thinking about how addons can interface with each other using the core as a bridge.

In ofxKinect and ofxSlitScan’s case, both addons used the same type of data:

```
unsigned char* getDepthPixels();       ///< grayscale values //from
    ofxKinect.h
```

and

```
void setDelayMap(unsigned char* map, ofImageType type); //from
    ofxSlitScan.h
```

So connecting them was simple:

```
slitScan.setDelayMap(depthPixels); //from testApp::update() in
    testApp.cpp
```

This kind of separation demonstrates encapsulation or the information hiding qualities of software - the utility of not having to know the specifics of the implementation of the functionality described, merely the inputs required and outputs produced.

http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming) http://en.wikipedia.org/wiki/In

21.3.2.2 ofxSlitScan, using PNG's and moving to generating realtime delay maps, making a Aurora

Starting is often the hardest thing to do with programming. To combat this, I try to do the stupidest, most obvious thing first and then build from there. In this project, I started by prototyping various looks using static PNGs - feeding new data into the examples provided with ofxSlitScan. The provided an easy sketching ability - combined with a paint program to quickly produce many input variations.

The next place to experiment was making the input real-time and interactive - using the blobs from a sliced section of the live Kinect depth image from ofxKinect. Drawing these simple blobs as an image allowed them to be inputted into ofxSlitscan on a frame by frame basis - producing a time warping effect over the playback of the film that Pete Hellicar edited for the project. As so often happens, when the input to the interaction becomes real-time it was far more engaging, which is exactly what we wanted users to do - see SLITSCANKINECTDEPTHGREY mode below for more details on the precise implementation, and in the other cases that follow.

What else could be done with the depth information applied to the delay map of the slit scan? Experiments with effecting the blobs outline yielded the SPIKYBLOBSLITSCAN mode. Using the input from the Kinect as an input to a paint simulator was something that I had worked on with Marek Bereza in the Somantics project - it made sense to try it as an input to a slitscan, as can be seen in the PAINT mode. This Paint mode made something that very much resembled the appearance of a human aurora when mixed with the beautiful Borealis footage that Pete Hellicar had sourced with the help of Greenpeace. SPARKLE mode was another example of a successful port from Somantics to Anthropocene.

Another good strategy for finding new interesting things is to feed the output of a system back into its input - this is demonstrated well by the visual feedback effects produced by using video frames as the delay maps back into their own history - implemented in SELFSLITSCAN mode.

21.3.2.3 ofxBox2d, making ice, previous projects with Todd Vanderlin

I had previously worked with Todd Vanderlin on the Feedback project, where we had experimented with using Box2D (via Todd's ofxBox2D) as a way of "shattering" live video. Feedback used a screen orientated in portrait mode that encouraged the repeating of

familiar existing behaviour - moving the experience from a tech demo to a playful joyous one. Having earlier experimented with ice like static PNG's I knew that using real-time triangles from ofxBox2D would work well - this time I had the advantage via the Kinect of a slice of 3D space as input, something that Todd had to work much harder to simulate using only 2D live camera input in Feedback. This aspect of constantly improving novel hardware inputs means that previous work can often be revisited and explored.

21.3.2.4 ofxTimeline, understanding how cuing works

To combine the film and the various real-time effects, it was essential to develop a cuing system to allow different effects to combine with different scenes in a reliably repeatable way. I began by experimenting with Duration, but after emailing the author of the addon (see development notes above), it become apparent that ofxTimeline would be a much better fit for the project - a subset of Durations code base.

After dealing with Quicktime performance issues (see below), the main challenge was cuing the effects. The structure of how ofxTimeline passes messages meant that the signal to switch scenes would only be sent when the play-head passed over the cue - clicking to a point after a cue meant that the signal to switch scenes would not be despatched. Deadlines of other functionality meant that this couldn't be fixed in time for show time - meaning that show operators would have to be careful when shuffling playback during the show proper.

21.3.2.5 ofxGui, running the Latest branch from Github, multiple input methods and GUI addons

I knew that I wanted to augment ofxTimelines interface with controls for the setup of the Kinect and other custom requirements for the project. Watching the GitHub development branch revealed the release of an official core GUI addon - something I wanted to experiment with, which meant that I had to switch from an official static release of OF to the live development branch via Github. The project ended up with multiple interfaces - two graphical ones (ofxTimeline and ofxKinect control mainly) and a keyboard based one (consisting mainly of single boolean switches together with playback and editing shortcuts). With further development, a unified GUI would be desirable, but development pressures meant it wasn't a priority.

21.3.2.6 ofxOpticalFlowFarneback, making a polar bear

During development and testing, I realised a furry look could serve well for making people feel like they were polar bears. I had seen "spikey" outline looks before - all

achieved by drawing normals along the circumference of a blob. I'd also experimented with optical flow in previous projects and started thinking about how the two could be combined - I looked for optical flow addons on ofxaddons.com[10] and discovered a flurry of recent activity since I'd last checked. Development tends to flow like this - periods of fallow followed by simultaneous parallel development from several quarters.

- ofxCvOpticalFlowLK by James George[11]
- ofxOpticalFlowFarneback by Tim Scaffidi[12]
- ofxOpticalFlowLK by Lukasz Karluk[13]

Tim Scaffidi's version immediately stood out to Pete, so I developed two simple colourings for Aurora and Polar Bear modes, merely tweaking Tim's excellent demo code.

21.3.3 XML Issues around the Naming of Scenes

Mid development, I found that saving the XML wasn't functioning as expected - it turned out to be the fault of non alpha numeric characters in scene names. I learnt the hard way that it's always good to avoid punctuation and spaces altogether and use CamelCase[14].

21.3.4 Video Performance, using the HighPerformanceExample

Right from the beginning of the project, it was obvious that video decoding would be significant portion of processing time per frame. Others in the openFrameworks community had been investigating performance in recent years, with James George contributing an OSX only High Performance video example[15]. This used native Quicktime playback features, enabling far higher performance on compatible hardware. While this undoubted enabled the film playback to function smoothly, it did make the code less platform independent - one of the inevitable compromises that happens during development.

21.3.5 Counting the items in an Enum

I knew that I would have to switch between different visual looks as the film was played back by the program. C++ provides the ENUM keyword to allow the coder to define

[10] http://ofxaddons.com
[11] https://github.com/Flightphase/ofxCvOpticalFlowLK
[12] https://github.com/timscaffidi/ofxOpticalFlowFarneback
[13] https://github.com/julapy/ofxOpticalFlowLK
[14] http://en.wikipedia.org/wiki/CamelCase
[15] https://github.com/openframeworks/openFrameworks/commit/4e02db8d82c520bef6c09d58b37076a84fe37571

a data set of named elements, but I needed a way to count the number of modes programmatically. Stack Overflow[16] provided the solution.

```
enum GreenpeaceModes {BLANK, GUI, VIDEO, VIDEOCIRCLES,
    KINECTPOINTCLOUD, SLITSCANBASIC, SLITSCANKINECTDEPTHGREY,
    SPARKLE, VERTICALMIRROR, HORIZONTALMIRROR, KALEIDOSCOPE,
    COLOURFUR, DEPTH, SHATTER, SELFSLITSCAN, SPIKYBLOBSLITSCAN,
    MIRRORKALEIDOSCOPE, PARTICLES, WHITEFUR, PAINT, GreenpeaceModes_
    MAX = PAINT}; //best to use ALL CAPS for enumerated types and
    constants so you can tell them from ClassNames and variableNames.
    Use camelCase for variableNames -
    http://en.wikipedia.org/wiki/CamelCase

/*
    http://stackoverflow.com/questions/2102582/how-can-i-count-the-items-in-an-enum
 For C++, there are various type-safe enum techniques available, and
    some of those (such as the proposed-but-never-submitted
    Boost.Enum) include support for getting the size of a enum.

 The simplest approach, which works in C as well as C++, is to adopt
    a convention of declaring a ...MAX value for each of your enum
    types:

 enum Folders { FA, FB, FC, Folders_MAX = FC };
 ContainerClass *m_containers[Folders_MAX + 1];
 ....
 m_containers[FA] = ...; // etc.
 Edit: Regarding { FA, FB, FC, Folders_MAX = FC} versus {FA, FB, FC,
    Folders_MAX]: I prefer setting the ...MAX value to the last
    legal value of the enum for a few reasons:

 The constant's name is technically more accurate (since Folders_MAX
    gives the maximum possible enum value).
 Personally, I feel like Folders_MAX = FC stands out from other
    entries out a bit more (making it a bit harder to accidentally
    add enum values without updating the max value, a problem Martin
    York referenced).
 GCC includes helpful warnings like "enumeration value not included
    in switch" for code such as the following. Letting Folders_MAX
    == FC + 1 breaks those warnings, since you end up with a bunch
    of ...MAX enumeration values that should never be included in
    switch.
 switch (folder)
 {
 case FA: ...;
 case FB: ...;
 // Oops, forgot FC!
 }
```

[16] http://stackoverflow.com/questions/2102582/how-can-i-count-the-items-in-an-enum

```
*/
```

I used the Stack Overflow tip in the `void testApp::keyPressed (int key)` method.

```
case 'a': //used to be key left, but it interferes with ofxtimeline
{
    currentMode = (GreenpeaceModes)((int)currentMode - 1);
    if(currentMode < 0){
        currentMode = GreenpeaceModes_MAX;//see .h file for
            stackoverflow justification
    }
    break;
}
case 's': //used to be key right, but it interferes with ofxtimeline
{
    currentMode = (GreenpeaceModes)((int)currentMode + 1);
    if(currentMode > GreenpeaceModes_MAX){
        currentMode = (GreenpeaceModes)0;//see .h file for
            stackoverflow justification
    }
}
```

While I could have gone down the polymorphic[17] custom class route, I felt that the ENUM apporach provided good performance (through compiler optimisation of common C++ coding paradigms), speed of development (lower file overhead) and clarity of code.

21.3.6 Sequencing

Kieran and Pete completed the main sequencing on-site.

21.4 Show time

21.5 Post Event

The biggest PR boost to the project while it was live was a review[18] from Timeout:

"A highlight of the Greenpeace field was undoubtedly the Arctic Dome, voted by Time Out as the second best non-musical thing to do at the Festival and previewed by NME. It offered people the opportunity to disappear through a crack in the ice and take a magical 15-minute trip to the North Pole, where ice towered and the Northern Lights danced."

[17] http://en.wikipedia.org/wiki/Polymorphism_(computer_science)
[18] http://www.glastonburyfestivals.co.uk/news/greenpeace-at-glastonbury-2013

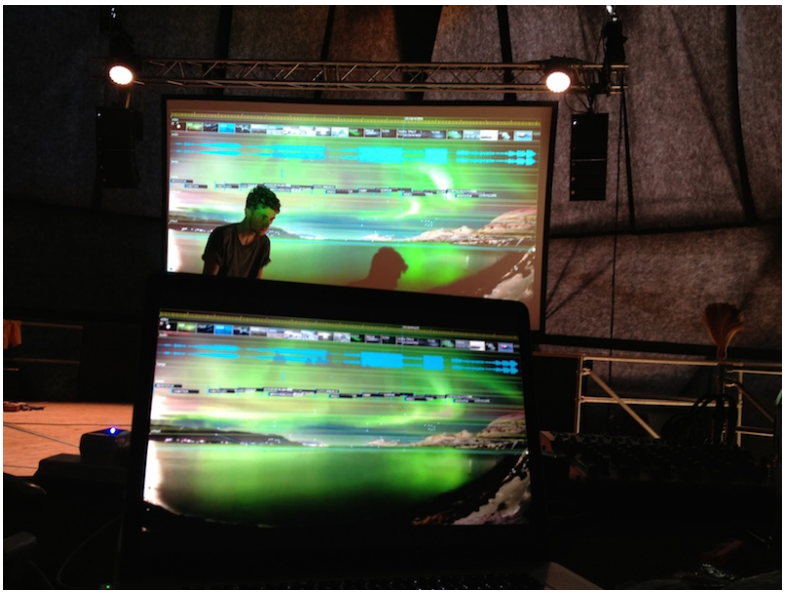


Figure 21.5: Kieran in front of the projection screen, final sequencing

Figure 21.6: The Installation in Action, 27th June 2013

21.5.1 Testimony from Show Operators

Kieran and Bafic were the people who ran the show for the general public, below is their testimony, with Kieran starting:

Did you have a routine before show time?

Before the first show of the day we'd double check the connection between the laptop and the Kinect and test with the skeleton tracking that everything was working correctly. Before show time we'd dim the lights, make sure the sound was turned on, switch to the point cloud setting so people could see themselves as they walked in and then we'd turn the beanbags upright so as to 'set the scene'. Then, as people started to walk in we'd raise the lights as though they were walking on stage. And then before we pressed play we'd dim the lights to black.

Any problems during shows? Crashes? Funny stories?

A couple of times the connection between the Kinect and the laptop failed due to the cable being under tension so we just had to let the show run to the end before we could fix it. The main problem we had was the projector overheating and displaying a lamp warning which involved having to find the technician to sort it. At one point the projector overheated so badly that we had to leave it switched off for 40 minutes before we could run the show again.

Off the top of my head I can't think of anything I'd like to change about it, the GUI had quite a steep learning curve so it took a while to remember all the keys to press to hide each part of the interface but once we'd mastered that everything was fine. I guess the only thing that would be good but most likely ultimately un-achieveable would be full automation in the sense that the station wouldn't have to be manned at all times.

Following is Bafic's post show report:

Did you have a routine before show time?

Before every show we would go through different ways to layout the bean bags. It's started off as just a small idea but as we kept on doing it we noticed that it would affect how people acted with the film. Some were semi circles some were bean bags set up in rows like cinema seats sometimes we pushed all bean bags to the back and told people they had to stand up and use their full body to interact with the film.

When seated in rows people mostly used their arms (a few people were moving the legs in air sitting down) but never was it a full body movement until we moved bean bags to the back . Some excited people would stand up and run to in front of the Kinect and interact with it that way, after they had finished they would sit down and someone else would follow due to the sheer curiosity of seeing what the previous person had done. It was interesting because everyone was so curious as to what would happen. I was sitting their amazed because their were a few loops/back and forths happening.

1. You had the back and forth between the one person who would stand up interact with the Kinect and then that would show up on the projection.
2. They would sit down and the next back and forth would be the next person to stand up start off with maybe replicating the previous persons techniques and movement AND Then coming up with the own ideas and movement.
3. then their was us who was watching and getting excited and seeing what they were doing and changing effects depending on what the user was doing and what we felt like could be interesting then obviously what we put on screen would effect how the person would dance/move/use their body. The whole thing was like a 3x over Möbius strip of events and occurrences that kept affecting the previous element and also the next element at the same time!

Any problems during shows? Crashes? Funny stories?

I can't think of any crashes or problems that happened. Their was a time when someone came in with a puppet on a long stick and they waved it at the Kinect and that would egg on the rest of the audience because this funny puppet would appear on screen. The whole experience was really amazing and interesting.

21.5.2 Open Source discussions with Client

Greenpeace were happy for us to Open Source, as we do with all our projects. Greenpeace does not have a GitHub of it's own, but we were able to suggest that that should be part of their future strategy. The problem was the film that formed the backdrop for the interaction - while musicians were happy to license music for a live only experience, getting those rights in perpetuity has been challenging. Negotiations continue.

21.5.3 Re-running remotely in Australia and New Zealand

The project has been re-exhibited twice in the Southern Hemisphere - in Australia and New Zealand. Getting the code up and running wasn't a problem - but training someone to use the two layers of mouse GUI and on layer of Keyboard GUI was a challenge, especially over a painfully slow Skype connection.

21.5.4 Future development

Paul Valery said 'Poems are never finished - just abandoned'. This is sadly true for all artistic endeavours. Below are three areas for future development.

21.5.4.1 Social interaction

The Hello Wall[19] and Hello Cube[20] projects showed how making feedback loops between users and installations via social networks is not only fun, but helps spread awareness of the installation beyond the physical bounds of the project. Imagine allowing users to post comments to the projection as it happening via Twitter and receiving bespoke screen grabs showing evidence of their interaction in return - or even choosing which of the interactive effects is active at a certain time. The meta data of these interactions could be used to come up with the most enaging timeline, or to deliver messages to users in the days, weeks and months following the installation - particularly useful for an organisation such as Greenpeace that relies on public support to lobby Governments and Corporations.

21.5.4.2 Broadcast

Pete and I discussed how we could transform the installation into one that broadcast itself to a wider audience when we were in the planning stage. Unfortunately, securing a reliable Internet connection at the Glastonbury Music festival proved impossible. Post and Previous Hellicar&Lewis projects for Nike[21] and Coca-Cola[22] show how broadcasting an installation with the addition of social network interaction can dramatically increase engagement. We hope to be able to make such a socially activated broadcast interaction with Greenpeace in the near future - imagine several locations around the world witnessing the same film simultaneously with body movement from each location feeding back into the others - live video portals of depth maps crossing continents and time zones to produce a truly global event.

21.5.4.3 Raspberry Pi

With the advent of a Raspberry Pi[23] port of openFrameworks, a port of the project to the platform would allow for the deployment of the project to events that have even smaller budgets than this iteration. This would also entail a port of the Kinect code to 2D computer vision, but I'm confident this would be a spur for other interactions and visual effects.

[19] http://www.hellicarandlewis.com/the-hello-wall/
[20] http://www.hellicarandlewis.com/tate-modern/
[21] http://www.hellicarandlewis.com/nikefeeltv/
[22] http://www.hellicarandlewis.com/coke/
[23] http://www.openframeworks.cc/setup/raspberrypi/

21.5.5 Conclusion

All in all, for a low budget project, using openFrameworks was the differentiator that enabled me to collaborate with the rest of the team at Hellicar&Lewis to make the installation come to life. The key factors were being able to draw upon so many external addons, previous projects and the community as a whole.

21.6 Team and Credits

- Pete Hellicar and Joel Gethin Lewis
- Commissioned by Paul Earnshaw of Greenpeace
- Produced by Sarah Toplis
- Assisted by Bafic[24] and Kieran Startup[25]

Project uses addons and other code Contributions from:

- Marek Bereza aka Mazbox[26] as part of Cariad Interactive
- ofxKinect[27] by Theo Watson[28]
- ofxSlitScan[29] by James George[30]
- ofxBox2d[31] by Todd Vanderlin[32]
- ofxTimeline[33] by James George[34]
- ofxOpticalFlowFarneback[35] by Tim Scaffidi[36]

Thanks to: * All our families and friends. * The Greenpeace Family * Microsoft for being Open * Theo Watson[37] * The entire openFrameworks community * Marshmallow Laser Feast[38] * Tim Scaffidi[39] * James George[40] * YCAM InterLab[41]

[24] http://www.bafic.co.uk/
[25] http://www.kieranstartup.co.uk/
[26] http://www.mazbox.com/
[27] https://github.com/ofTheo/ofxKinect
[28] http://www.theowatson.com/
[29] https://github.com/obviousjim/ofxSlitScan
[30] http://jamesgeorge.org/
[31] https://github.com/vanderlin/ofxBox2d
[32] http://vanderlin.cc/
[33] https://github.com/YCAMInterlab/ofxTimeline
[34] http://jamesgeorge.org/
[35] https://github.com/timscaffidi/ofxOpticalFlowFarneback
[36] http://timothyscaffidi.com/
[37] http://www.theowatson.com/
[38] http://marshmallowlaserfeast.com/
[39] http://timothyscaffidi.com/
[40] http://jamesgeorge.org/
[41] http://interlab.ycam.jp/en

21.7 Hardware selection

- 1 x 3D Camera - Microsoft XBox360 Kinect
- 1 x Playback and Interaction Computer - MacBook Pro Retina
- 1 x 10K projector
- 1 x Projection Screen
- Sound - 4 x D&B T-10 Top + Amp 2 x Subs

21.8 Appendix 1: Code structure, main loop

The structure of setup(), update() and draw() methods is common to openFrameworks code - with the addition of two large switch statements for switching between modes at runtime.

```
//--------------------------------------------------------------
void testApp::update() {
    //kinect
    kinect.update();
    // there is a new frame and we are connected
    if(kinect.isFrameNew()) {
        // load grayscale depth image from the kinect source
        depthPreCrop.setFromPixels(kinect.getDepthPixels(),
           kinect.width, kinect.height);

        if(mirror){
            depthPreCrop.mirror(false, true);
        }

        maskGrayImage();
        depthPreCrop.flagImageChanged();

        // save original depth, and do some preprocessing
        depthOrig = depthPreCrop; //copy cropped image into orig
        depthProcessed = depthOrig; //copy orig into processed
        colorImageRGB = kinect.getPixels(); //getting colour pixels

        if(invert) depthProcessed.invert();
        if(mirror) {
            colorImageRGB.mirror(false, true);
            //greyIRSingleChannel.mirror(false, true);
        }

        depthOrig.flagImageChanged();
        depthProcessed.flagImageChanged();
        colorImageRGB.flagImageChanged();
```

```
        if(preBlur) cvSmooth(depthProcessed.getCvImage(),
            depthProcessed.getCvImage(), CV_BLUR , preBlur*2+1);
        if(topThreshold) cvThreshold(depthProcessed.getCvImage(),
            depthProcessed.getCvImage(), topThreshold * 255, 255, CV_
            THRESH_TRUNC);
        if(bottomThreshold) cvThreshold(depthProcessed.getCvImage(),
            depthProcessed.getCvImage(), bottomThreshold * 255, 255,
            CV_THRESH_TOZERO);
        if(dilateBeforeErode) {
            if(dilateAmount) cvDilate(depthProcessed.getCvImage(),
                depthProcessed.getCvImage(), 0, dilateAmount);
            if(erodeAmount) cvErode(depthProcessed.getCvImage(),
                depthProcessed.getCvImage(), 0, erodeAmount);
        } else {
            if(erodeAmount) cvErode(depthProcessed.getCvImage(),
                depthProcessed.getCvImage(), 0, erodeAmount);
            if(dilateAmount) cvDilate(depthProcessed.getCvImage(),
                depthProcessed.getCvImage(), 0, dilateAmount);
        }
        depthProcessed.flagImageChanged();

        // find contours
        depthContours.findContours(depthProcessed,
                                   minBlobSize * minBlobSize *
                                       depthProcessed.getWidth() *
                                       depthProcessed.getHeight(),
                                   maxBlobSize * maxBlobSize *
                                       depthProcessed.getWidth() *
                                       depthProcessed.getHeight(),
                                   maxNumBlobs, findHoles,
                                       useApproximation);
        //now do the diff bits for the PAINT mode
        ofxCvGrayscaleImage thresholdedDepthImageForPaint;
        thresholdedDepthImageForPaint.setFromPixels(depthProcessed.getPixelsRef())
        thresholdedDepthImageForPaint.resize(paintCanvas.getWidth(),
            paintCanvas.getHeight());
        thresholdedDepthImageForPaint.flagImageChanged();
        // loop through pixels
        //  - add new colour pixels into canvas
        unsigned char *canvasPixels = paintCanvas.getPixels();
        unsigned char *diffPixels =
            thresholdedDepthImageForPaint.getPixels();

        int r = 255;

        for(int i = 0; i < paintCanvas.width*paintCanvas.height;
            i++) {
            if(diffPixels[i]) {
```

```
                //paint in the new colour if
                canvasPixels[i*3] = r;
                canvasPixels[i*3+1] = r;
                canvasPixels[i*3+2] = r;
            }else{
                int greyScale = (int)(canvasPixels[i*3]*0.9f);
                canvasPixels[i*3] = greyScale;
                canvasPixels[i*3+1] = greyScale;
                canvasPixels[i*3+2] = greyScale;
            }
        }
        paintCanvas.blur();
        paintCanvas.flagImageChanged();
        paintCanvasAsOfImage.setFromPixels(paintCanvas.getPixelsRef());
        paintCanvasAsOfImage.update();
        flowSolver.setPyramidScale(pyramidScale);
        flowSolver.setPyramidLevels(pyramidLevels);
        flowSolver.setWindowSize(windowSize);
        flowSolver.setExpansionArea(expansionAreaDoubleMe*2);
        flowSolver.setExpansionSigma(expansionSigma);
        flowSolver.setFlowFeedback(flowFeedback);
        flowSolver.setGaussianFiltering(gaussianFiltering);
        flowSolver.update(depthProcessed);
    }


    //Dirty filthy hack
    if(currentMode != SLITSCANBASIC){
        prevSlitScan = -1;
    }
    switch(currentMode){
```

see below for mode by mode update details

```
        default:
            break;
    }
}
```

```
void testApp::draw() {
    ofBackground(0, 0, 0);
    ofSetColor(255, 255, 255);

    switch (currentMode) {
```

see below for descriptions of various modes drawing

```
    }
```

```
    if( bShowNonTimelineGUI ){
        nonTimelineGUI.draw();
    }


    if( timeline.getIsShowing() ){
        ofSetColor(255, 255, 255);

        //timeline
        timeline.draw();

        string modeString;
        modeString = "Mode␣is␣";

        switch (currentMode) {
            case BLANK: //blank mode
                modeString += "BLANK";
                break;
```

edited for sanity.

```
        }

        ofSetColor(ofColor::red);
        ofDrawBitmapString(modeString,20,100);
    }
}
```

21.9 Appendix 2: Modes, with screen grabs and code explanation

21.9.0.1 BLANK

Blank mode simply displayed a blank screen. A useful default for measuring idle performance.

Mode update:

```
        case BLANK: //image drawing mode
            break;
```

21.9.0.2 GUI

GUI displayed several program variables and image previews of various stages of Kinect image and blob outline processing.

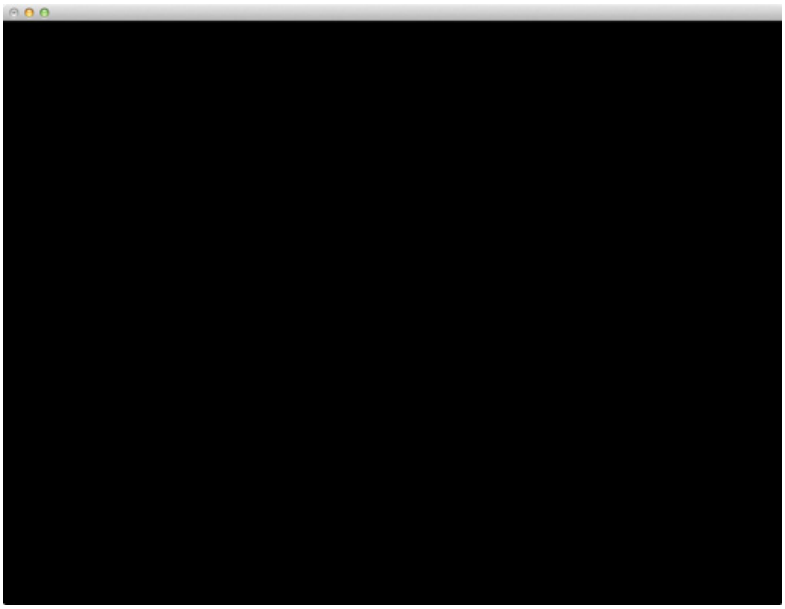


Figure 21.7: BLANK Mode

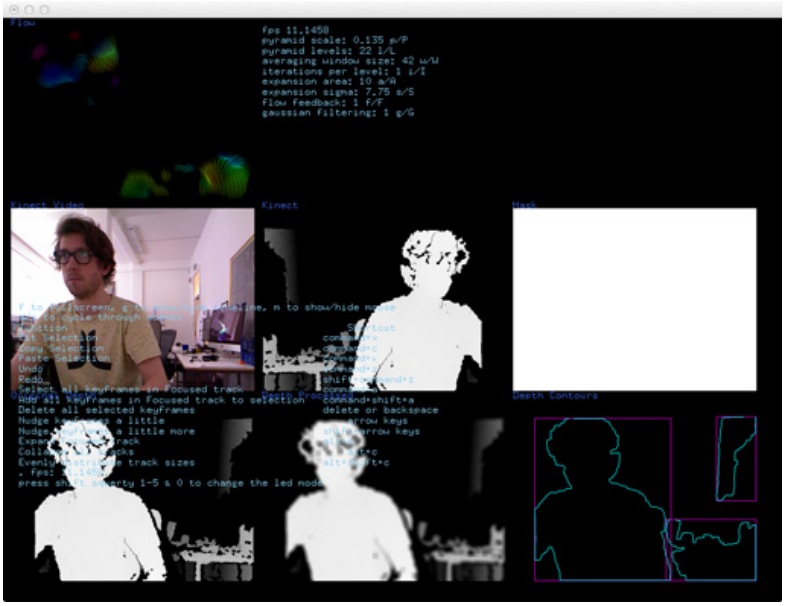


Figure 21.8: GUI Mode

Mode update:

```
        case GUI: //GUI MODE
            break;
```

Mode draw:

```
        case GUI: //image drawing mode
        {
            ofFill();
            ofSetColor(0);
            ofRect(0,0,ofGetWidth(),ofGetHeight()); //draw a black
               rectangle

            int imageOffSet = 10;
            int imageWidth = 320;
            int imageHeight = 240;
            int imageX = imageOffSet;

            // draw everything
            ofSetColor(ofColor::white);
            ofEnableAlphaBlending();
            flowSolver.drawColored(imageWidth, imageHeight, 10, 3);
            ofDisableAlphaBlending();
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Flow", imageX, imageOffSet);
            ofSetColor(ofColor::white);
            colorImageRGB.draw(imageX, imageHeight+imageOffSet,
               imageWidth, imageHeight);
            //greyIRSingleChannel.draw(imageX,
               imageHeight+imageOffSet, imageWidth, imageHeight);
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Kinect␣Video", imageX,
               imageHeight+imageOffSet);
            imageX += imageOffSet+imageWidth;
            ofSetColor(ofColor::white);
            kinect.drawDepth(imageX, imageHeight+imageOffSet,
               imageWidth, imageHeight);
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Kinect", imageX,
               imageHeight+imageOffSet);
            imageX += imageOffSet+imageWidth;
            ofSetColor(ofColor::white);
            maskImage.draw(imageX,imageHeight+imageOffSet,
               imageWidth, imageHeight);
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Mask", imageX,
               imageHeight+imageOffSet);
            imageX = imageOffSet;
            ofSetColor(ofColor::white);
```

```
            depthOrig.draw(imageX,imageHeight+imageOffSet+imageHeight+imageOffSet,
               imageWidth, imageHeight);
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Original␣Depth", imageX,
               imageHeight+imageOffSet+imageHeight+imageOffSet);
            imageX += imageOffSet+imageWidth;
            ofSetColor(ofColor::white);
            depthProcessed.draw(imageX,imageHeight+imageOffSet+imageHeight+imageOffSet,
               imageWidth, imageHeight);
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Depth␣Processed", imageX,
               imageHeight+imageOffSet+imageHeight+imageOffSet);
            imageX += imageOffSet+imageWidth;
            ofSetColor(ofColor::white);
            depthContours.draw(imageX,
               imageHeight+imageOffSet+imageHeight+imageOffSet,
               imageWidth, imageHeight);
            ofSetColor(ofColor::royalBlue);
            ofDrawBitmapString("Depth␣Contours", imageX,
               imageHeight+imageOffSet+imageHeight+imageOffSet);
            ofSetColor(ofColor::skyBlue);
            // draw instructions
            stringstream reportStream;
            reportStream
            << "f␣to␣fullscreen,␣g␣to␣show/hide␣timeline,␣m␣to␣
               show/hide␣mouse" << endl
            << "a/s␣to␣cycle␣through␣scenes" << endl
            << "Function␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               Shortcut" << endl
            << "Cut␣Selection␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               command+x" << endl
            << "Copy␣Selection␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               command+c" << endl
            << "Paste␣Selection␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               command+v" << endl
            << "Undo␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               command+z" << endl
            << "Redo␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               shift+command+z" << endl
            << "Select␣all␣keyframes␣in␣Focused␣track␣␣␣␣␣␣␣␣␣␣␣␣␣
               command+a" << endl
            << "Add␣all␣keyframes␣in␣Focused␣track␣to␣selection␣␣␣
               command+shift+a" << endl
            << "Delete␣all␣selected␣keyframes␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               delete␣or␣backspace" << endl
            << "Nudge␣keyframes␣a␣little␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               arrow␣keys" << endl
            << "Nudge␣keyframes␣a␣little␣more␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
               shift+arrow␣keys" << endl
```

```
            << "Expand␣Focused␣track␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
                alt+e" << endl
            << "Collapse␣all␣tracks␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
                alt+c" << endl
            << "Evenly␣distribute␣track␣sizes␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
                alt+shift+c" << endl
            << ",␣fps:␣" << ofGetFrameRate() << endl
            << "press␣shift␣squerty␣1-5␣&␣0␣to␣change␣the␣led␣mode"
                << endl;
            ofDrawBitmapString(reportStream.str(),20,ofGetHeight()/2.f);

            stringstream m;
            m << "fps␣" << ofGetFrameRate() << endl
            << "pyramid␣scale:␣" << flowSolver.getPyramidScale() <<
                "␣p/P" << endl
            << "pyramid␣levels:␣" << flowSolver.getPyramidLevels()
                << "␣l/L" << endl
            << "averaging␣window␣size:␣" <<
                flowSolver.getWindowSize() << "␣w/W" << endl
            << "iterations␣per␣level:␣" <<
                flowSolver.getIterationsPerLevel() << "␣i/I" << endl
            << "expansion␣area:␣" << flowSolver.getExpansionArea()
                << "␣a/A" << endl
            << "expansion␣sigma:␣" << flowSolver.getExpansionSigma()
                << "␣s/S" << endl
            << "flow␣feedback:␣" << flowSolver.getFlowFeedback() <<
                "␣f/F" << endl
            << "gaussian␣filtering:␣" <<
                flowSolver.getGaussianFiltering() << "␣g/G";

            ofDrawBitmapString(m.str(), 20+320, 20);
        }
            break;
```

21.9.0.3 VIDEO

Video mode displayed the current frame of the unprocessed video file.

Mode update:

```
        case VIDEO:
            break;
```

Mode draw:

```
        case VIDEO: //the film
            ofFill();
            ofSetColor(255);
```

Figure 21.9: VIDEO Mode

```
            timeline.getVideoPlayer("video")->draw(0, 0,
                ofGetWidth(),ofGetHeight());
            break;
```

21.9.0.4 VIDEOCIRCLES

VideoCircles was a direct cut and paste from the examples/video/osxHighPerformanceVideoPlayerExamp
This code was useful during initial development to discover the performance hit for individual pixel array access. A lot of my early development during projects is based around finding what the limits of various prospective coding functionality is - getting to a happy mix of performance and functionality.

Mode update:

```
        case VIDEOCIRCLES: //the film as circles
            break;
```

Mode draw:

```
        case VIDEOCIRCLES: //the film as circles
            {
                ofFill();
                ofSetColor(0);
                ofRect(0,0,ofGetWidth(),ofGetHeight()); //draw a
                    black rectangle
```

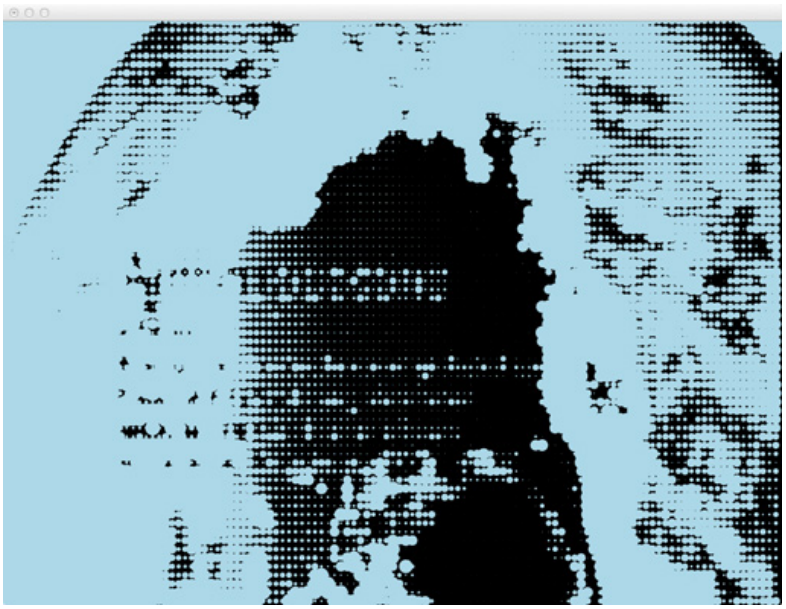


Figure 21.10: VIDEOCIRCLES Mode

```
if (timeline.getVideoPlayer("video")->isLoaded()) {
    unsigned char * pixels =
        timeline.getVideoPlayer("video")->getPixels();
    ofPixelsRef pixelsRef =
        timeline.getVideoPlayer("video")->getPixelsRef();

    // let's move through the "RGB(A)" char array
    // using the red pixel to control the size of a
        circle.
    //ofSetColor(timeline.getColor("colour"));
    ofSetColor(ofColor::lightBlue);

    float circleSpacing = 10.f;

    float widthRatio =
        ofGetWidth()/timeline.getVideoPlayer("video")->getWidth();
    float heightRatio =
        ofGetHeight()/timeline.getVideoPlayer("video")->getHeight(

    for(int i = 0; i <
        timeline.getVideoPlayer("video")->getWidth();
        i+= 8){
         for(int j = 0; j <
             timeline.getVideoPlayer("video")->getHeight();
             j+= 8){
              ofColor pixelColor =
```

```
                                timeline.getVideoPlayer("video")->getPixelsRef().getColo
                                j);
                            int b = pixelColor.b;
                            float val = 1 - ((float)b / 255.0f);
                                //more blue in the arctic!
                            ofCircle(i*widthRatio, j*heightRatio,
                                circleSpacing * val);
                        }
                    }
                }
            }
            break;
```

21.9.0.5 KINECTPOINTCLOUD

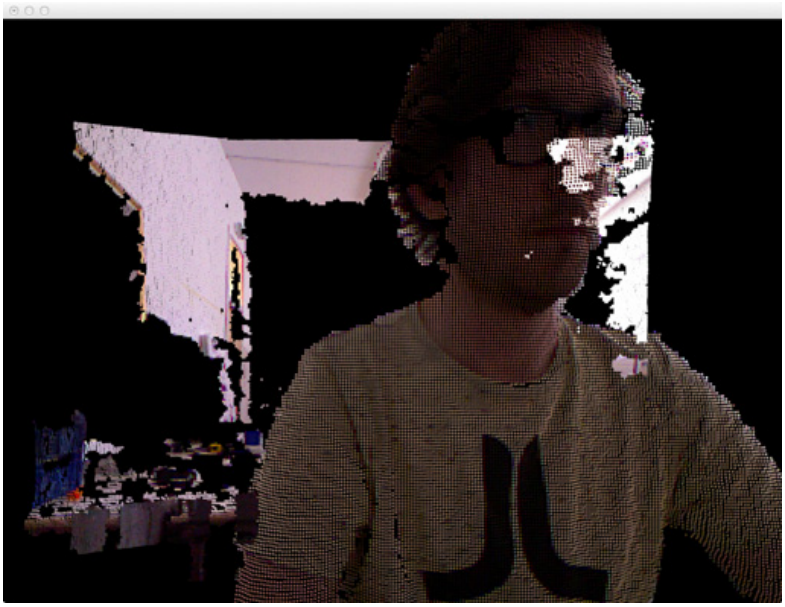


Figure 21.11: KINECTPOINTCLOUD Mode

Another cut and paste from addon example code, this time from the now core `ofxKinect`.

Mode update:

```
        case KINECTPOINTCLOUD: //draw the kinect camera depth cloud
            break;
```

Mode draw:

```
        case KINECTPOINTCLOUD: //draw the kinect camera depth cloud
            easyCam.begin();
            drawPointCloud();
            easyCam.end();
            break;
```

21.9.0.6 SLITSCANBASIC

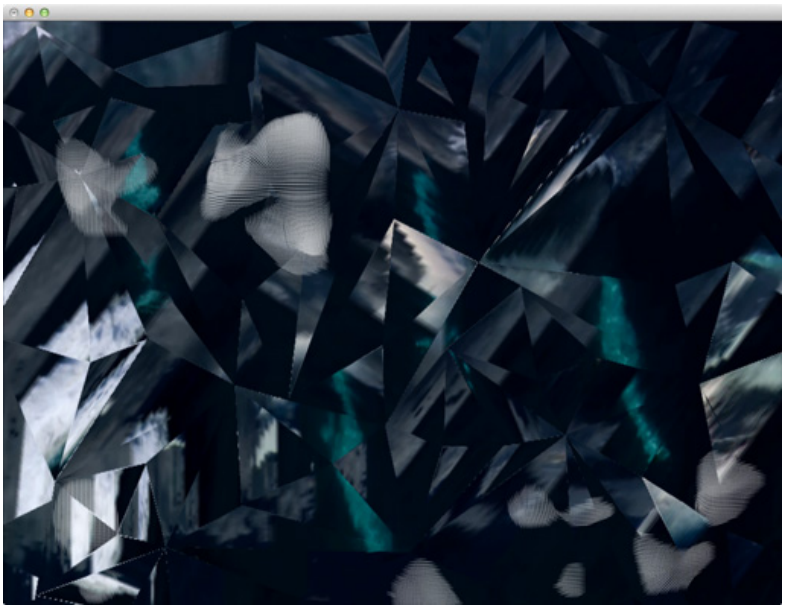


Figure 21.12: SLITSCANBASIC Mode

The most basic of the slitscan modes on this project - a direct port of example functionality in ofxSlitscan - but with the possibility of changing the slitscan PNG source file on the ofxTimeline GUI.

Mode update:

```
        case SLITSCANBASIC: //slit scan the movie on the grey from
            the kinect depth grey
        {
            //check slit scan...
            int theCurrentSlitScan = timeline.getValue("slitscan");
            if(prevSlitScan != theCurrentSlitScan){
                slitScanSliderSlid(); //only update when you have
                    to...
                prevSlitScan = theCurrentSlitScan;
            }
```

```
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef());
            }
        }
            break;
```

Mode draw:

```
        case SLITSCANBASIC: //slit scan the movie on depth png
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
               ofGetHeight());

            //white fur
            ofEnableAlphaBlending();
            flowSolver.drawGrey(ofGetWidth(),ofGetHeight(), 10, 3);
            ofDisableAlphaBlending();

            break;
```

21.9.0.7 SLITSCANKINECTDEPTHGREY

Figure 21.13: SLITSCANKINECTDEPTHGREY Mode

The most basic of novel slitscan modes developed for this project - feeding the Kinect depth image into ofxSlitscan on a per frame basis - once I realised this would still result in interactive frame rates I knew the project would succeed.

Mode update:

```
        case SLITSCANKINECTDEPTHGREY: //slit scan the movie on the
            grey from the kinect depth grey
        {
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                //kinect slitscan
                //depthPixels.setFromPixels(kinect.getDepthPixelsRef());
                depthPixels.setFromPixels(depthProcessed.getPixelsRef());
                depthPixels.resize(timeline.getVideoPlayer("video")->getWidth(),
                    timeline.getVideoPlayer("video")->getHeight());
                //
                    slitScanDepthGrey.setDelayMap(depthPixels);
                //
                    slitScanDepthGrey.addImage(timeline.getVideoPlayer("video")->ge
                slitScan.setDelayMap(depthPixels);
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef()
            }
        }
            break;
```

Mode draw:

```
        case SLITSCANKINECTDEPTHGREY: //slit scan the movie on the
            grey from the kinect depth grey
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
                ofGetHeight());
            //slitScanDepthGrey.getOutputImage().draw(0, 0,
                ofGetWidth(), ofGetHeight());
            break;
```

21.9.0.8 SPARKLE

An experiment with using previously developed Somantics functionality into ofxTimeline.

Mode update:

```
        case SPARKLE: //sparkles on the slitscan
        {
            //update the sparkles come what may...
            someSparkles.update(&depthContours);
            someSparkles.draw(ofColor::white);
            //someSparkles.draw(timeline.getColor("colour"));

            ofImage distortionMap;
            distortionMap.allocate(someSparkles.theFBO.getWidth(),
                someSparkles.theFBO.getHeight(), OF_IMAGE_COLOR);
```

Figure 21.14: SPARKLE Mode

```
            someSparkles.theFBO.readToPixels(distortionMap.getPixelsRef());

            distortionMap.resize(timeline.getVideoPlayer("video")->getWidth(),
                timeline.getVideoPlayer("video")->getHeight());
            slitScan.setDelayMap(distortionMap);

            if(timeline.getVideoPlayer("video")->isFrameNew()){
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef());
            }
        }
            break;
```

Mode draw:

```
        case SPARKLE:
            //do some sparkles - used the slit scan to hold it....
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
                ofGetHeight());
            //ofSetColor(255,255,255);
            //someSparkles.theFBO.draw(0, 0, ofGetWidth(),
                ofGetHeight());
            break;
```

21.9.0.9 VERTICALMIRROR

Figure 21.15: VERTICALMIRROR Mode

A vertical mirror on the video playback - again ported directly from Somantics.

Mode update:

```
        case VERTICALMIRROR: //vertical mirror
        {
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                verticalMirrorImage.setFromPixels(timeline.getVideoPlayer("video")
                   verticalMirrorImage.getWidth(),
                   verticalMirrorImage.getHeight());

                verticalMirrorImage.updateTexture();
            }
        }
            break;
```

Mode draw:

```
        case VERTICALMIRROR:
        {
            bool usingNormTexCoords =
               ofGetUsingNormalizedTexCoords();

            if(!usingNormTexCoords) {
                ofEnableNormalizedTexCoords();
```

```
            }

            verticalMirrorImage.getTextureReference().bind();

            ofMesh mesh;
            mesh.clear();
            mesh.addVertex(ofVec3f(0, 0));
            mesh.addVertex(ofVec3f(0, ofGetHeight()));
            mesh.addVertex(ofVec3f(ofGetWidth()/2, 0));
            mesh.addVertex(ofVec3f(ofGetWidth()/2, ofGetHeight()));
            mesh.addVertex(ofVec3f(ofGetWidth(), 0));
            mesh.addVertex(ofVec3f(ofGetWidth(), ofGetHeight()));


            mesh.addTexCoord(ofVec2f(0.25, 0.0));
            mesh.addTexCoord(ofVec2f(0.25, 1.0));
            mesh.addTexCoord(ofVec2f(0.75, 0.0));
            mesh.addTexCoord(ofVec2f(0.75, 1.0));
            mesh.addTexCoord(ofVec2f(0.25, 0.0));
            mesh.addTexCoord(ofVec2f(0.25, 1.0));

            mesh.setMode(OF_PRIMITIVE_TRIANGLE_STRIP);
            ofSetColor(ofColor::white);
            mesh.draw();

            verticalMirrorImage.getTextureReference().unbind();

            // pop normalized tex coords
            if(!usingNormTexCoords) {
                ofDisableNormalizedTexCoords();
            }

            //white fur
            ofEnableAlphaBlending();
            flowSolver.drawGrey(ofGetWidth(),ofGetHeight(), 10, 3);
            ofDisableAlphaBlending();
        }
            break;
```

21.9.0.10 HORIZONTALMIRROR

A horizontal mirror on the video playback - again ported directly from Somantics.

Mode update:

```
        case HORIZONTALMIRROR: //HORIZONTALMIRROR mirror
        {
            if(timeline.getVideoPlayer("video")->isFrameNew()){
```

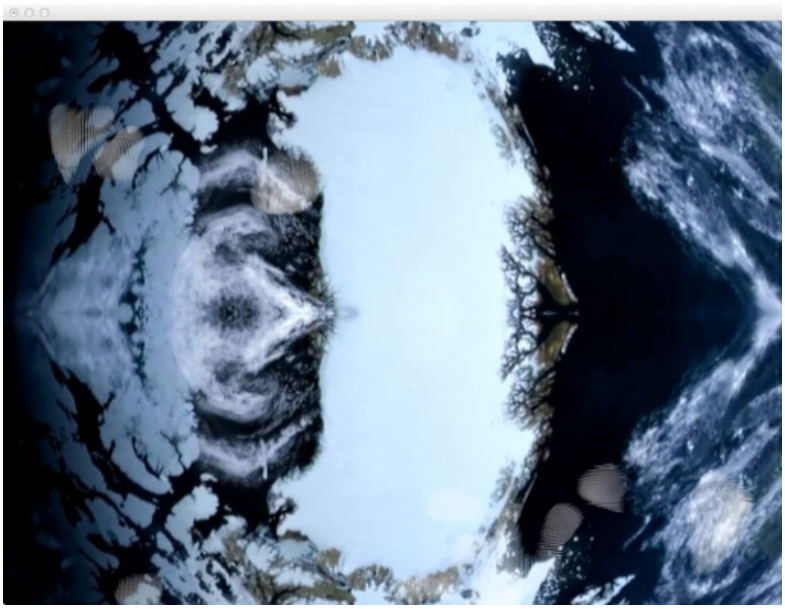


Figure 21.16: HORIZONTALMIRROR Mode

```
                horizontalMirrorImage.setFromPixels(timeline.getVideoPlayer("video
                    horizontalMirrorImage.getWidth(),
                    horizontalMirrorImage.getHeight());

                horizontalMirrorImage.updateTexture();
            }
        }
            break;
```

Mode draw:

```
        case HORIZONTALMIRROR:
        {
            bool usingNormTexCoords =
                ofGetUsingNormalizedTexCoords();

            if(!usingNormTexCoords) {
                ofEnableNormalizedTexCoords();
            }

            horizontalMirrorImage.getTextureReference().bind();

            ofMesh mesh;
            mesh.clear();
            mesh.addVertex(ofVec3f(ofGetWidth(), 0));
            mesh.addVertex(ofVec3f(0, 0));
```

```
            mesh.addVertex(ofVec3f(ofGetWidth(), ofGetHeight()/2));
            mesh.addVertex(ofVec3f(0, ofGetHeight()/2));
            mesh.addVertex(ofVec3f(ofGetWidth(), ofGetHeight()));
            mesh.addVertex(ofVec3f(0,ofGetHeight()));

            mesh.addTexCoord(ofVec2f(1.0, 0.25));
            mesh.addTexCoord(ofVec2f(0.0, 0.25));
            mesh.addTexCoord(ofVec2f(1.0, 0.75));
            mesh.addTexCoord(ofVec2f(0.0, 0.75));
            mesh.addTexCoord(ofVec2f(1.0, 0.25));
            mesh.addTexCoord(ofVec2f(0.0, 0.25));

            mesh.setMode(OF_PRIMITIVE_TRIANGLE_STRIP);
            ofSetColor(ofColor::white);
            mesh.draw();

            horizontalMirrorImage.getTextureReference().unbind();

            // pop normalized tex coords
            if(!usingNormTexCoords) {
                ofDisableNormalizedTexCoords();
            }

            //white fur
            ofEnableAlphaBlending();
            flowSolver.drawGrey(ofGetWidth(),ofGetHeight(), 10, 3);
            ofDisableAlphaBlending();
        }
            break;
```

21.9.0.11 KALEIDOSCOPE

A Kaleidoscope mirror on the video playback - again ported directly from Somantics, using Marek Bereza's[42] logic.

Mode update:

```
        case KALEIDOSCOPE: //kaleidsocope
        {
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                kaleidoscopeMirrorImage.setFromPixels(timeline.getVideoPlayer("video")->
                    kaleidoscopeMirrorImage.getWidth(),
                    kaleidoscopeMirrorImage.getHeight());

                kaleidoscopeMirrorImage.updateTexture();
            }
```

[42] http://mazbox.com/

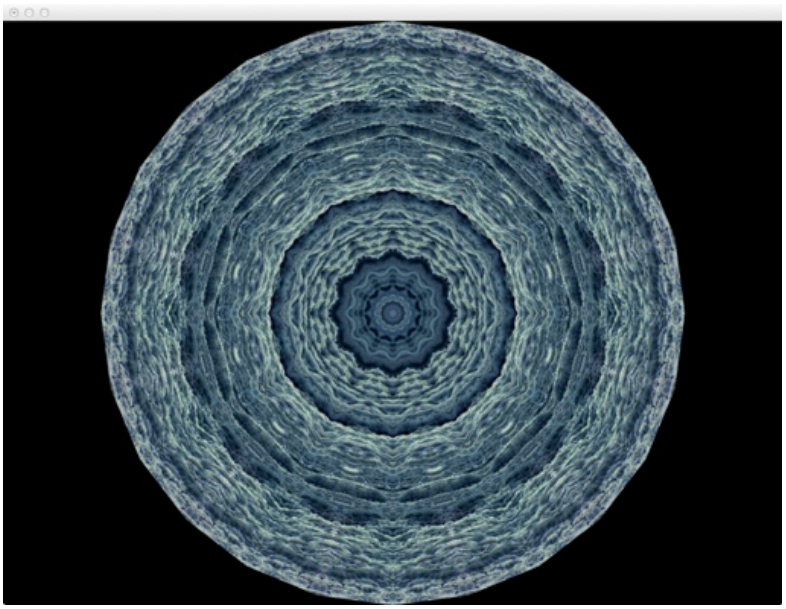


Figure 21.17: KALEIDOSCOPE Mode

```
        }
            break;
```

Mode draw:

```
        case KALEIDOSCOPE:
        {
            bool usingNormTexCoords =
               ofGetUsingNormalizedTexCoords();

            if(!usingNormTexCoords) {
                ofEnableNormalizedTexCoords();
            }

            kaleidoscopeMirrorImage.getTextureReference().bind();

            int star = ((int)timeline.getValue("star")*2);//8; //get
               star from the timeline gui, but multiply by 2 to get
               to always even
            float offset = timeline.getValue("offset");//0.5f; //
               get offset from the timeline gui
            float angle = 360.f/star; //8 sides to start



            ofMesh mesh;
```

```
ofVec3f vec(0,0,0);
mesh.addVertex(vec);
vec.x += ofGetHeight()/2;

for(int i = 0; i < star; i++) {
    mesh.addVertex(vec);
    vec.rotate(angle, ofVec3f(0,0,1));
}

// close the loop
mesh.addVertex(vec);



// now work out the texcoords
/*____________________

 |   \        /   |
 |    \      /    |
 |     \    /     |
 |      \  /      |
 |       \/       |
 +----------------+

 A v shape out of the centre of the camera texture
 */



float realOffset = 0.5;
// normalized distance from the centre (half the width
   of the above 'V')
float dist =
   ABS((float)kaleidoscopeMirrorImage.getHeight()*tan(ofDegToRad(angle)*0.5


// the realOffset is where the (normalized) middle of
   the 'V' is on the x-axis
realOffset = ofMap(offset, 0, 1, dist, 1-dist);


// this is the point at the bottom of the triangle - our
   centre for the triangle fan
mesh.addTexCoord(ofVec2f(realOffset, 1));


ofVec2f ta(realOffset-dist, 0);
ofVec2f tb(realOffset+dist, 0);
```

```
            for(int i = 0; i <= star; i++) {
                if(i%2==0) {
                    mesh.addTexCoord(ta);
                } else {
                    mesh.addTexCoord(tb);
                }
            }


            glPushMatrix();
            glTranslatef(ofGetWidth()/2, ofGetHeight()/2, 0);
            mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
            mesh.draw();
            glPopMatrix();

            kaleidoscopeMirrorImage.getTextureReference().unbind();

            // pop normalized tex coords
            if(!usingNormTexCoords) {
                ofDisableNormalizedTexCoords();
            }

            //white fur
            ofEnableAlphaBlending();
            flowSolver.drawGrey(ofGetWidth(),ofGetHeight(), 10, 3);
            ofDisableAlphaBlending();
        }
            break;
```

21.9.0.12 COLOURFUR

A direct port of Tim Scaffidi's ofxOpticalFlowFarneback[43] demo code.

Mode update:

```
        case COLOURFUR: //COLOURFUR
        {
        }
            break;
```

Mode draw:

```

        case COLOURFUR:
        {
            ofSetColor(ofColor::white);
```

[43] https://github.com/timscaffidi/ofxOpticalFlowFarneback

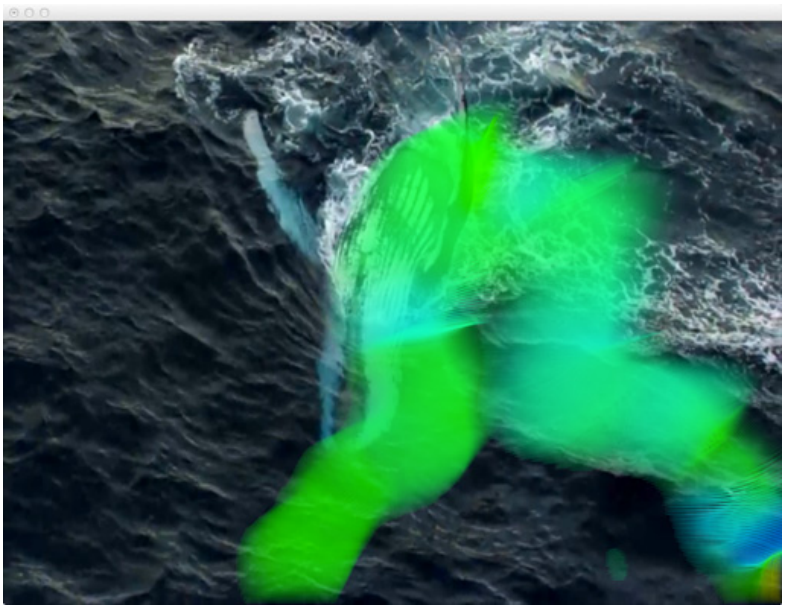


Figure 21.18: COLOURFUR Mode

```
            timeline.getVideoPlayer("video")->draw(0, 0,
                ofGetWidth(),ofGetHeight());
            ofEnableAlphaBlending();
            flowSolver.drawColored(ofGetWidth(),ofGetHeight(), 10,
                3);
            ofDisableAlphaBlending();
        }
            break;
```

21.9.0.13 DEPTH

A simple mode to display the depth image directly - useful for debugging when onsite.

Mode update:

```
        case DEPTH: //DEPTH
        {
        }
```

Mode draw:

```
        case DEPTH:
        {
            depthProcessed.draw(0,0,ofGetWidth(), ofGetHeight());
        }
```

Figure 21.19: DEPTH Mode

```
            break;
```

21.9.0.14 SHATTER

A direct port of Todd Vanderlin's[44] code that he wrote for the Feedback project, but using it as live delay map input to the Slitscan.

Mode update:

```
        case SHATTER:
        {
            //update the shatter
            theShatter.update(&depthContours);
            theShatter.draw(ofColor::white);

            ofImage distortionMap;
            distortionMap.allocate(theShatter.theFBO.getWidth(),
               theShatter.theFBO.getHeight(), OF_IMAGE_COLOR);

            theShatter.theFBO.readToPixels(distortionMap.getPixelsRef());

            distortionMap.resize(timeline.getVideoPlayer("video")->getWidth(),
               timeline.getVideoPlayer("video")->getHeight());
```

[44] http://vanderlin.cc/projects/feedback/

Figure 21.20: SHATTER Mode

```
            slitScan.setDelayMap(distortionMap);

            if(timeline.getVideoPlayer("video")->isFrameNew()){
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef());
            }
        }
            break;
```

Mode draw:

```
        case SHATTER:
        {
            //do some shattering - used the slit scan to hold it....
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
               ofGetHeight());
            //ofSetColor(255,255,255);
            //theShatter.theFBO.draw(0, 0, ofGetWidth(),
               ofGetHeight());
        }
            break;
```

Figure 21.21: SELFSLITSCAN Mode

21.9.0.15 SELFSLITSCAN

Feeding the greyscale image of the current film frame back into the SlitScan delay map made for some interesting feedback effects.

Mode update:

```
        case SELFSLITSCAN:
        {
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                //self slitscan
//                ofImage selfSlitScanDelayMap;
//
    selfSlitScanDelayMap.allocate(timeline.getVideoPlayer("video")->getWidth(),
    timeline.getVideoPlayer("video")->getHeight(), OF_IMAGE_COLOR);
//
    selfSlitScanDelayMap.setFromPixels(timeline.getVideoPlayer("video")->getPixelsR

                slitScan.setDelayMap(timeline.getVideoPlayer("video")->getPixelsRe
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef()
            }
        }
            break;
```

Mode draw:

```
        case SELFSLITSCAN:
```

```
        {
            //do some SELFSLITSCAN - used the slit scan to hold
                it....
            ofSetColor(255,255,255);
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
                ofGetHeight());
        }
            break;
```

21.9.0.16 SPIKYBLOBSLITSCAN

Figure 21.22: SPIKYBLOBSLITSCAN Mode

Feeding the Spiked blob outline back into the SlitScan delay map.

Mode update:

```
        case SPIKYBLOBSLITSCAN:
        {
            //SPIKYBLOBSLITSCAN
            //update the spikes come what may...
            theSpikey.update(&depthContours);
            theSpikey.draw(ofColor::white);

            ofImage distortionMap;
            distortionMap.allocate(theSpikey.theFBO.getWidth(),
                theSpikey.theFBO.getHeight(), OF_IMAGE_COLOR);
```

```
            theSpikey.theFBO.readToPixels(distortionMap.getPixelsRef());

            distortionMap.resize(timeline.getVideoPlayer("video")->getWidth(),
               timeline.getVideoPlayer("video")->getHeight());
            slitScan.setDelayMap(distortionMap);

            if(timeline.getVideoPlayer("video")->isFrameNew()){
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef()
            }
        }
```

Mode draw:

```
        case SPIKYBLOBSLITSCAN:
        {
            //do some SPIKYBLOBSLITSCAN - used the slit scan to hold
               it....
            ofSetColor(255,255,255);
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
               ofGetHeight());
            //theSpikey.theFBO.draw(0,0,ofGetWidth(), ofGetHeight());
        }
            break;
```

21.9.0.17 MIRRORKALEIDOSCOPE

Combining Mirror and Kaleidoscope modes.

Mode update:

```
        case MIRRORKALEIDOSCOPE: //MIRRORKALEIDOSCOPE mirror
        {
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                verticalMirrorImage.setFromPixels(timeline.getVideoPlayer("video")
                   verticalMirrorImage.getWidth(),
                   verticalMirrorImage.getHeight());

                verticalMirrorImage.updateTexture();

                kaleidoscopeMirrorImage.setFromPixels(timeline.getVideoPlayer("vid
                   kaleidoscopeMirrorImage.getWidth(),
                   kaleidoscopeMirrorImage.getHeight());

                kaleidoscopeMirrorImage.updateTexture();
            }
        }
            break;
```

Figure 21.23: MIRRORKALEIDOSCOPE Mode

Mode draw:

```
        case MIRRORKALEIDOSCOPE:
        {
            bool usingNormTexCoords =
               ofGetUsingNormalizedTexCoords();

            if(!usingNormTexCoords) {
                ofEnableNormalizedTexCoords();
            }

            verticalMirrorImage.getTextureReference().bind();

            ofMesh mirrorMesh;
            mirrorMesh.clear();
            mirrorMesh.addVertex(ofVec3f(0, 0));
            mirrorMesh.addVertex(ofVec3f(0, ofGetHeight()));
            mirrorMesh.addVertex(ofVec3f(ofGetWidth()/2, 0));
            mirrorMesh.addVertex(ofVec3f(ofGetWidth()/2,
               ofGetHeight()));
            mirrorMesh.addVertex(ofVec3f(ofGetWidth(), 0));
            mirrorMesh.addVertex(ofVec3f(ofGetWidth(),
               ofGetHeight()));


            mirrorMesh.addTexCoord(ofVec2f(0.25, 0.0));
            mirrorMesh.addTexCoord(ofVec2f(0.25, 1.0));
```

```
            mirrorMesh.addTexCoord(ofVec2f(0.75, 0.0));
            mirrorMesh.addTexCoord(ofVec2f(0.75, 1.0));
            mirrorMesh.addTexCoord(ofVec2f(0.25, 0.0));
            mirrorMesh.addTexCoord(ofVec2f(0.25, 1.0));

            mirrorMesh.setMode(OF_PRIMITIVE_TRIANGLE_STRIP);
            ofSetColor(ofColor::white);
            mirrorMesh.draw();

            verticalMirrorImage.getTextureReference().unbind();

            kaleidoscopeMirrorImage.getTextureReference().bind();

            int star = ((int)timeline.getValue("star")*2);//8; //get
                star from the timeline gui, but multiply by 2 to get
                to always even
            float offset = timeline.getValue("offset");//0.5f; //
                get offset from the timeline gui
            float angle = 360.f/star; //8 sides to start

            ofMesh mesh;

            ofVec3f vec(0,0,0);
            mesh.addVertex(vec);
            vec.x += ofGetHeight()/2;

            for(int i = 0; i < star; i++) {
                mesh.addVertex(vec);
                vec.rotate(angle, ofVec3f(0,0,1));
            }

            // close the loop
            mesh.addVertex(vec);

            // now work out the texcoords
            /*_________________

             |   \        /   |
             |    \      /    |
             |     \    /     |
             |      \  /      |
             |       \/       |
             +----------------+

             A v shape out of the centre of the camera texture
             */

            float realOffset = 0.5;
            // normalized distance from the centre (half the width
```

```
            of the above 'V')
        float dist =
            ABS((float)kaleidoscopeMirrorImage.getHeight()*tan(ofDegToRad(angle)*0.5
        // the realOffset is where the (normalized) middle of
            the 'V' is on the x-axis
        realOffset = ofMap(offset, 0, 1, dist, 1-dist);

        // this is the point at the bottom of the triangle - our
            centre for the triangle fan
        mesh.addTexCoord(ofVec2f(realOffset, 1));

        ofVec2f ta(realOffset-dist, 0);
        ofVec2f tb(realOffset+dist, 0);
        for(int i = 0; i <= star; i++) {
            if(i%2==0) {
                mesh.addTexCoord(ta);
            } else {
                mesh.addTexCoord(tb);
            }
        }

        glPushMatrix();
        glTranslatef(ofGetWidth()/2, ofGetHeight()/2, 0);
        mesh.setMode(OF_PRIMITIVE_TRIANGLE_FAN);
        mesh.draw();
        glPopMatrix();

        kaleidoscopeMirrorImage.getTextureReference().unbind();

        // pop normalized tex coords
        if(!usingNormTexCoords) {
            ofDisableNormalizedTexCoords();
        }

        //white fur
        ofEnableAlphaBlending();
        flowSolver.drawGrey(ofGetWidth(),ofGetHeight(), 10, 3);
        ofDisableAlphaBlending();
    }
        break;
```

21.9.0.18 PARTICLES

Using Somantics particle functionality as a SlitScan delay map.

Mode update:

Figure 21.24: PARTICLES Mode

```
        case PARTICLES:
        {
            //PARTICLES
            theParticles.update(&depthContours);
            theParticles.draw(ofColor::white);
            ofImage distortionMap;
            distortionMap.allocate(theParticles.theFBO.getWidth(),
               theParticles.theFBO.getHeight(), OF_IMAGE_COLOR);
            theParticles.theFBO.readToPixels(distortionMap.getPixelsRef());
            distortionMap.resize(timeline.getVideoPlayer("video")->getWidth(),
               timeline.getVideoPlayer("video")->getHeight());
            slitScan.setDelayMap(distortionMap);
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef()
            }
        }
            break;
```

Mode draw:

```
        case PARTICLES:
        {
            //do some PARTICLES - used the slit scan to hold it....
            ofSetColor(255,255,255);
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
               ofGetHeight());
```

```
            //theParticles.theFBO.draw(0,0,ofGetWidth(),
                ofGetHeight());
        }
            break;
```

21.9.0.19 WHITEFUR

Figure 21.25: WHITEFUR Mode

Turning the ofxOpticalFlowFarneback demo code, but making the graphical output monochrome.

Mode update:

```
        case WHITEFUR: //WHITEFUR, nowt
        {
        }
            break;
```

Mode draw:

```
        case WHITEFUR:
        {
            ofSetColor(ofColor::white);
            timeline.getVideoPlayer("video")->draw(0, 0,
                ofGetWidth(),ofGetHeight());
            ofEnableAlphaBlending();
```

```
            flowSolver.drawGrey(ofGetWidth(),ofGetHeight(), 10, 3);
            ofDisableAlphaBlending();
        }
            break;
```

21.9.0.20 PAINT

Figure 21.26: PAINT Mode

Porting the Paint mode from Somantics as a delay map.

Mode update:

```
        case PAINT: //body painting diff
        {
            slitScan.setDelayMap(paintCanvasAsOfImage);
            if(timeline.getVideoPlayer("video")->isFrameNew()){
                slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef()
            }
        }
            break;
```

Mode draw:

```
        case PAINT:
        {
            //do some paint - used the slit scan to hold it....
```

```
            slitScan.getOutputImage().draw(0, 0, ofGetWidth(),
                ofGetHeight());
        }
            break;
```

21.10 Appendix 3: Edited development notes

21.10.0.21 29th May 2013

oF/of_v0.7.4_osx_release/apps/ofxKinect-demos

Also downloaded ofxKinect. Get gui working first, with ofxKinect, then start on:

- https://github.com/toruurakawa/ofxFakeMotionBlur
- Don't use, use jamezilla https://github.com/kylemcdonald/ofxBlur
- https://github.com/jamezilla/ofxBlurShader
- https://github.com/kylemcdonald/ofxCameraFilter
- https://github.com/vanderlin/ofxBox2d
- https://github.com/NickHardeman/ofxBullet
- https://github.com/fishkingsin/ofxPBOVideoPlayer
- https://github.com/arturoc/ofxPlaymodes
- Don't use, in core now https://github.com/Flightphase/ofxQTKitVideoPlayer
- https://github.com/after12am/ofxTLGlitch
- https://github.com/bakercp/ofxVideoBuffer
- https://github.com/bakercp/ofxVideoUtils
- https://github.com/obviousjim/ofxSlitScan

Doing gui - having to make the projectGenerator to make the projects, generating the examples now.... Recopy over examples after! Did it, just copying in the empty example xcode project, all in here now:

oF/of_v0.7.4_osx_release/examples/gui

Email of notes on development:

On 29 May 2013, at 20:44, Joel Gethin Lewis wrote:

- All ofFloatColor or ofFloatImages
- HSB blob shifts as a mode - crazy colours, also try whole image on slow change using ofmath demos
- Blobs cracking off
- Just blackness on blob
- Slit scan obvs

- ofxbox2d? Kinect demos? Look at memos
- Look at ofxaddons for time ones
- Use ofGui official one
- Have different GUI panes per constructor for ofxScenes (make that)

Think in addon way - indeed that every scene might have addons inside it. That's the way the should be. Addons inside scenes. Scenes are subclasses of ofxScene. Draw it out. Start with slitscan as first scene. Just get that working then use that as basis for ofxScene. Pragmatic! Will need central image creator as input for each scene. Kinect in this case. Don't worry about that for now.

ofparameter is missing! Looking at old OF folder from other project: openFrameworks-develop/apps/devApps/projectGenerator, looking in there in the oF project to try to find what is going on...

openFrameworks-develop/libs/openFrameworks/types contains:

- ofBaseTypes.cpp
- ofBaseTypes.h
- ofColor.cpp
- ofColor.h
- ofParameter.cpp
- ofParameter.h
- ofParameterGroup.cpp
- ofParameterGroup.h
- ofPoint.cpp
- ofPoint.h
- ofRectangle.cpp
- ofRectangle.h
- ofTypes.h

looking for ofpanel

openFrameworks-develop/addons/ofxGui/src

is where it is...opening:

oF/of_v0.7.4_osx_release/examples/gui/guiExample

again, just trying to add it in, in the addon... Nooo thats bad.. should use the develop version... space is low...now working here:

oF/openFrameworks-develop/apps/devApps/projectGenerator

trying to build that and run it, had to select the root oF folder, it was defaulting to a weird one, so selected:

oF/openFrameworks-develop

seems to be working, leaving it for a bit...

error ofFile::copyFromTo source file/folder doesn't exist: oF/openFrameworks-develop/scripts/osx/template/emptyExample.xcodeproj/xcshareddata/WorkspaceSettings.xcsettings

is the error....it's correct:

oF/openFrameworks-develop/scripts/osx/template/emptyExample.xcodeproj/xcshareddata

doesn't have it

openFrameworks-develop/apps/devApps/projectGenerator/bin/data/xcode/template/emptyExample.xcode

copied that in, and another file inside

openFrameworks-develop/apps/devApps/projectGenerator/bin/data/xcode/template/emptyExample.xcode

xcschememanagement.plist

as well...so trying to generate again...seems to be working now.....won't paste in the log!
(-; trying this now...

oF/openFrameworks-develop/examples/gui/guiExample

nice!

oF/openFrameworks-develop/examples/gui/guiFromParametersExample

next - not that interesting...

oscParametersReceiver oscParametersSender

together... Neat demo! synchronised gui controls....both crash on exit

sender:

```
void testApp::setup(){
    parameters.setName("parameters");
    parameters.add(size.set("size",10,1,100));
    parameters.add(number.set("number",10,1,100));
    parameters.add(check.set("check",false));
    parameters.add(color.set("color",ofColor(127),ofColor(0,0),ofColor(255)));
    gui.setup(parameters);
    // by now needs to pass the gui parameter groups since the panel
       internally creates it's own group
    sync.setup((ofParameterGroup&)gui.getParameter(),6667,"localhost",6666);
    ofSetVerticalSync(true);
}

void testApp::update(){
    sync.update();
}
```

receiver:

```
void testApp::setup(){
    parameters.setName("parameters");
    parameters.add(size.set("size",10,0,100));
    parameters.add(number.set("number",10,0,100));
    parameters.add(check.set("check",false));
    parameters.add(color.set("color",ofColor(127),ofColor(0,0),ofColor(255)));
    gui.setup(parameters);
    // by now needs to pass the gui parameter groups since the panel
       internally creates it's own group
    sync.setup((ofParameterGroup&)gui.getParameter(),6666,"localhost",6667);
    ofSetVerticalSync(true);
}

void testApp::update(){
    sync.update();
}

void testApp::draw(){
    gui.draw();
    ofSetColor(color);
    for(int i=0;i<number;i++){
        ofCircle(ofGetWidth()*.5-size*((number-1)*0.5-i),
           ofGetHeight()*.5, size);
    }
}
```

subtle difference in port lines in sync setups...

oF/openFrameworks-develop/examples/gui/parameterEdgeCasesExample

doesn't work...

oF/openFrameworks-develop/examples/gui/parameterGroupExample

Is very intersting - two renderers running at once! Only thing missing is multiple parameters, and images being drawn? could always do that with bools, and the images being displayed on top, sliders and the like could work with that too... Moving big greenpeace video into:

oF/openFrameworks-develop/examples/video/osxHighPerformanceVideoPlayerExample/bin/data/m

to save space, rather than copying!

oF/openFrameworks-develop/examples/video/osxHighPerformanceVideoPlayerExample

Trying this now...builds with standard movie file in demo, fingers.mov. Now trying, Greenpeace.m4v - works great! audio back too...and pixel access! MOVED video file out of the folder for safety..

copied in this:

oF/openFrameworks-develop/apps/ofxKinect-demos

trying normal ofxKinect first...

oF/openFrameworks-develop/addons/ofxKinect oF/openFrameworks-develop/addons/ofxKinect/example

trying that... works fine, with motor and everything...so making a mega mix up of:

ofxKinect, ofxGUI and ofHighPerformanceVideo demo

oF/openFrameworks-develop/apps/HAndLGreenpeace/001fromofxKinectExampleAndofxGUI/bin/data/movie

copied that in, changed name to:

oF/openFrameworks-develop/apps/HAndLGreenpeace/001fromofxKinectExampleAndofxGUIAndHighPerform

oF/openFrameworks-develop/examples/video/osxHighPerformanceVideoPlayerExample
oF/openFrameworks-develop/examples/gui/guiExample

copying over gui data...that works with gui.. now lets try with high performance video...all works! nice debug screen! saved it out to making of....

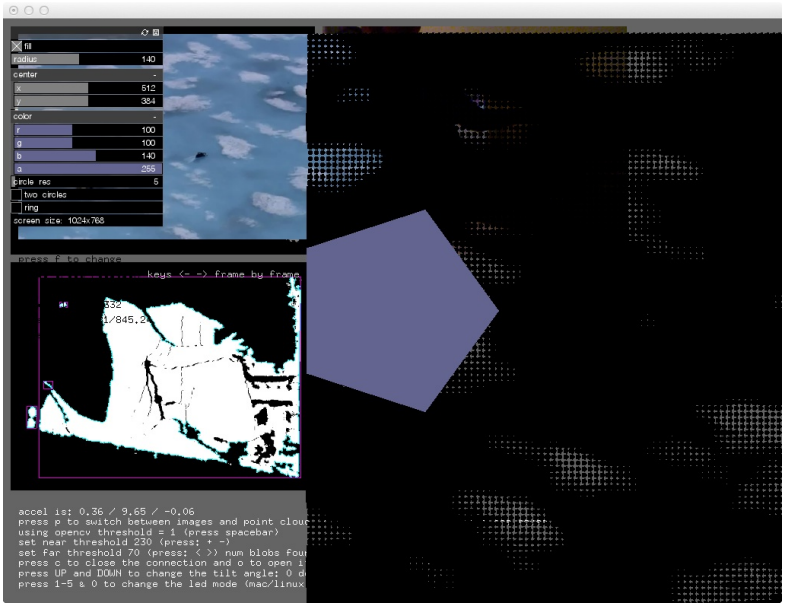


Figure 21.27: Kinect, GUI and High Performance Video Debug Screen

21.10.0.22 30th May 2013

Doing modes, tidying up gui, need to do more on gui tidy up and keys. Made:

oF/openFrameworks-develop/apps/HAndLGreenpeace/002FirstModesAndSlitScan

Builds (-; Pixel drawing is messed up, checking the original high perf demo. Recoded nicely with bits that made sense and were easier to understand...now slitsan! fixed a few gui bugs...

oF/openFrameworks-develop/addons/ofxSlitScan

Made that... Image is PNG RGB for slitscan delay map, so kinect depth map is good for that... right? (-; lucky birthday boy! works great...did grab..

Figure 21.28: Slit scan generated from Kinect Depth Map Slice

21.10.0.23 31st May 2013

Showed Pete, performance better on his laptop, no optimisation yet, tried out some maps with bafic...

21.10.0.24 6th June 2013

First lets do GUI that corresponds to big maps, so we can switch between them... Duplicating multicoloured magic into the folder, so got all that lovely code to work with.. After lunch lets look at mirror Somantics code... Too complicated for now, need some time to sit down and make it work... For now on the Virgin SA flight, lets try some of the addons!

ofxBlur By Kyle McDonald had to add accelerate, qtkit and corevideo frameworks to make it work..

ofxBlurShader This is a very lightly refactored version of Kyle McDonald's ofBlur example (https://github.com/kylemcdonald/SubdivisionOfRoam/tree/master/GaussianBlur). It hasbeen updated to OF 007. Didn't build!

ofxBox2d trying every example:

- oF/openFrameworks-develop/addons/ofxBox2d/ComplexPolgonExample - useful for making shatter like effects - how do I texture them?
- oF/openFrameworks-develop/addons/ofxBox2d/ContactListenerExample - useful for tiggering audio samples on ofxBox2D interactions
- oF/openFrameworks-develop/addons/ofxBox2d/CustomDataExample - useful for storing data withing objects, nothing particularly useful there for us at the moment..
- oF/openFrameworks-develop/addons/ofxBox2d/ForcesExample - bunch of forces on mouse action
- oF/openFrameworks-develop/addons/ofxBox2d/JointExample -long chain of pieces
- oF/openFrameworks-develop/addons/ofxBox2d/ofxBox2dExample - line that you can draw and lots of various shapes
- oF/openFrameworks-develop/addons/ofxBox2d/PolygonExample - more line drawing...
- oF/openFrameworks-develop/addons/ofxBox2d/SimpleExample - simple!

ofxBullet

- oF/openFrameworks-develop/addons/ofxBullet/SimpleExample - is simple, several different basic shapes...
- oF/openFrameworks-develop/addons/ofxBullet/CustomShapesExample - needs ofxAssimpMeshHelper - that is in the assimp model loader addon... cool - very fast simulation and * loading of custom shapes - perhaps pete could make custom 3D shapes?
- oF/openFrameworks-develop/addons/ofxBullet/EventsExample - smashing of objects into each other, mouse animation of objects within cube
- oF/openFrameworks-develop/addons/ofxBullet/JointsExample - has long chain of shapes, similar to ofxbox2d example...

ofxCameraFilter

- oF/openFrameworks-develop/addons/ofxCameraFilter/example-graphics - simple camera effects on some rotating shapes, aberation and the like...
- oF/openFrameworks-develop/addons/ofxCameraFilter/example-live - does the same but live, and with an interesting use of an ofMesh

ofxFakeMotionBlur

- oF/openFrameworks-develop/addons/ofxFakeMotionBlur/example - no work

ofxPBOVideoPlayer

- oF/openFrameworks-develop/addons/ofxPBOVideoPlayer/example - seems speedy

ofxPlaymodes

- oF/openFrameworks-develop/addons/ofxPlaymodes/example-pmAV - needs more addons, come back to this..

ofxTLGlitch

- trying oF/openFrameworks-develop/addons/ofxVideoBuffer/example-multi-tap - had to add empty example, couldn't get building....

DONE addon off....

21.10.0.25 12th June 2013

Greenpeace logos look nice as slit scans! saved all graphics and fonts into:

2013_06_12_Font 2013_06_12_GreenpeaceLogos

Pete gave me new audio and the film for working with duration

21.10.0.26 13th June 2013

Duration demo is up from James George too:

- Posted demo code from Duration.cc demo github.com/obviousjim/Dur... cc @JGL

got that, put it here:

2013_06_13_obviousJimAudioReactiveRing

and copied into:

OF/openFrameworks-develop/apps/jamesGeorgeDurationDemo/DurationAudioReactiveRing-master

made:

Duration_004_OSX Duration_004_OSX.zip durationData

too.. The readme sez:

```
Duration: Timeline for Creative Code Demonstration

Code used in the demo of Duration:
```

```
http://vimeo.com/59654979

Requires ofxRange and ofxDuration
https://github.com/YCAMInterlab/ofxDuration
https://github.com/Flightphase/ofxRange

Download Duration
http://www.duration.cc/ // https://github.com/YCAMInterlab/Duration

Supported by YCAM InterLab Guest Research Project 2012
```

Getting those.. put in here:

2013_06_13_MoreDurationBits

ofxRange-master.zip ofxDuration-master.zip

trying this first

OF/openFrameworks-develop/addons/ofxDuration/example-simpleReceiver

totally did it, totally worked - have to show Pete Hellicar it tomorrow, and disuss which controls he wants...made a new track:

Duration/durationData/FirstTry

audio all loaded in fine (-; need to test with film sync, see if that works OK.. try to set the movie time on each frame? will it fuck everything? Basically should make a new version of the app:

OF/openFrameworks-develop/apps/HAndLGreenpeace/003WithOFXDuration

Added:

GUI SimpleReceiverPort.txt

to data folder too...need to compare with: OF/openFrameworks-develop/addons/ofxDuration/example-simpleReceiver and duplicate the functionality - start with scene control and colour....

MORNING TIME

OF/openFrameworks-develop/addons/ofxDuration/example-simpleReceiver

opening that and taking the functionality over...

```
ofxDurationTrack sceneTrack = duration.getTrack("/scene");
string currentScene = sceneTrack.flag;

if(currentScene == "VIDEO"){
    currentMode = VIDEO;
}

if(currentScene == "SLITSCANBASIC"){
```

```
    currentMode = SLITSCANBASIC;
}
```

totally works!

21.10.0.27 16th June 2013

Lets try the video syncing over osc.. Didn't seem to work with:

```
float remoteTime = sceneTrack.lastUpdatedTime;
cout << "Remote time is:" << remoteTime << endl
float totalLengthOfVideo = greenpeaceVideo.getDuration();
float percentToSeekTo = remoteTime/totalLengthOfVideo;
greenpeaceVideo.setPosition(percentToSeekTo);
```

Hmmm. Sent this to james and got a response:

On 16 Jun 2013, at 19:00, James George wrote: yea it's impossible to call setPosition on a video every frame and have it playback smoothly. Quicktime needs to control its own time. Try this: play the video back normally in openframeworks and then update Duration every frame based on it's position:

```
https://github.com/YCAMInterlab/Duration#controlling-duration-through-osc

Specifically make sure Duration has its incoming OSC port set and
   from OF send it a /duration/seektosecond. Get the seconds from
   the video player.getPosition()*player.getDuration() then create
   an outgoing OSC message directed at Duration:

Seek    /duration/seektosecond  Second (Float)   Sets playhead
   position to the specified second
Sending the /seektosecond message will then trigger an update to
   come back from Duration to your app and update all the other
   params.
```

On Sun, Jun 16, 2013 at 1:49 PM, Joel Gethin Lewis wrote: Hey James, I've been trying to get a Duration app to be able to sync the video playback on an OF app - I used your example and have started trying to sync to the time from a track:

```
ofxDurationTrack sceneTrack = duration.getTrack("/scene");
string currentScene = sceneTrack.flag;
float remoteTime = sceneTrack.lastUpdatedTime;
cout << "Remote time is:" << remoteTime << endl;
float totalLengthOfVideo = greenpeaceVideo.getDuration();
cout << "totalLengthOfVideo time is:" << totalLengthOfVideo << endl;
float percentToSeekTo = remoteTime/totalLengthOfVideo;
cout << "percentToSeekTo time is:" << percentToSeekTo << endl;
greenpeaceVideo.setPosition(percentToSeekTo);
```

But it results in stuttering, playback - do you have any tips? How often are the control packets sent? Should I be getting the remote time in a better way? Cheers, Joel

looking at:

https://github.com/YCAMInterlab/Duration#controlling-duration-through-osc

/duration/seektosecond

Is what we want...so need to setup osc, trying to get that working with a simple sender, having problems gaining control from the OF app. Sent this:

On 16 Jun 2013, at 20:48, Joel Gethin Lewis wrote: Hey James, Thanks! It kind of works, but not really. I have my app jumping around it's video when I press t:

```
case 't':
{
   float newseekposition = (float)mouseX/(float)ofGetWidth();
   ofClamp(newseekposition, 0.f, 1.f); //safety
   greenpeaceVideo.setPosition(newseekposition);
   cout << "New seek position is: " << newseekposition << endl;
}
```

If the Duration app is set to paused, it updates fine, the playhead moving around when I press T in my app- but I don't get the messages back from Duration! If it isn't paused (the duration app), I get the messages, but I can't move the Duration playhead around with the above messages! Catch 22? What should I do? I want to get the messages back, have it be playing on both ends and be able to seek. At the moment, I can have seeking in my app and Duration, but without messages back. Or messages back, without seeking.

Sending the /seektosecond message will then trigger an update to come back from Duration to your app and update all the other params.

Doesn't seem to be happening? Two little Duration suggestions:

1) Shouldn't ofxDuration have a send to DurationApp method? That would be useful, no? Auto osc.
2) Can I mute the audio of the Duration app in its GUI?

Any thoughts gratefully recieved. Ideally, I'd like either side to be Master if it sends messages to the other. Make sense? My app the true master, but seeking to Quicktime if it gets an occasional timeline change from the Duration app - but how to do that only some of the time? Cheers, Joel

Made new osc send:

On 16 Jun 2013, at 20:51, Joel Gethin Lewis wrote: This is my send, in my update:

```
//update duration based on the position of the quicktime player
```

```
float videoTimeToSend =
    greenpeaceVideo.getPosition()*greenpeaceVideo.getDuration();
ofxOscMessage m;
m.setAddress("/duration/seektosecond");
m.addFloatArg(videoTimeToSend);
senderToDuration.sendMessage(m);
```

Got this reply back, and replied:

On 16 Jun 2013, at 21:07, Joel Gethin Lewis wrote: Hey James, I'll take a look. BUT! Looking at this video:

https://vimeo.com/59653952

It looks like I might be better off doing everything in a single OF app. What do you think? Do you think my massive video file (785,526,769 bytes (785.5 MB on disk)) will kill your thumbnail maker? Lets see...Cheers, Joel

On 16 Jun 2013, at 20:53, James George wrote:

Huh! Duration should definitely update when you move the playhead even if it's not playing... Definitely a bug. Must be a bug in the way seektosecond works. This may be a rabbit hole, but try downloading the source from the Duration website (its the entire OF bundle) and see if you can give it a look. it's probalby a simple change to make sure that handleOscOut() works even when it's not playing.

Watched that video above, did this:

jglmacbookprocore2:addons joel$ git clone https://github.com/YCAMInterlab/ofxTimeline.git

Trying:

OF/openFrameworks-develop/addons/ofxTimeline/example-videoRenderer

Worked...tried:

OF/openFrameworks-develop/apps/tryingBigVideoIntoOfxTimeline/example-allTracks

Totally worked! OK - so need to get audio file and video files separately... autosaves..... apple z for undo even works!

21.10.0.28 17th June 2013

On 17 Jun 2013, at 00:12, James George wrote: the video player posted on the oF list is really nice, but it doesn't support the getCurrentFrame() command which may cause some issues. give it a shot! On Sun, Jun 16, 2013 at 5:02 PM, Joel Gethin Lewis wrote: IT TOTALLY ROCKS! It totally works with the thumbnailer. Great work. I am going to code it up as a pure OF app, with maybe a little OSC Remote. Did you see the discussion about the new OSX high performance video player? How gnarly is the hookup to the ofVideoPlayer? I just glanced at the code and it didn't seem too bad.. AMAZING. Cheers,

Joel On Sun, 16 Jun 2013, at 21:26, James George wrote: that's me in the video btw ;) On Sun, Jun 16, 2013 at 4:26 PM, James George wrote: no it'll be fine, the thumbnails generator is really light, it pulls only as it needs. you can also disable it.

Need to decide on the different modes - and do a mirror mode and a proper sparkles mode with direction.. First off, make a new version, with everything in it and timeline and duration stripped out...RIGHT! made this:

OF/openFrameworks-develop/apps/HAndLGreenpeace/004BuiltInOfxTimeLine

All working nice for demo, need to re-add GUI elements for showing and hiding etc... did it on mouse hide... works great with the slitscan control on too...So next, it's really time to do effects...mirror first.. add an x variable for the point....

21.10.0.29 18th June 2013

doing sparkles first: copied over:

/Users/joel/Documents/Projects/HellicarAndLewis/greenpeaceArcticGlastonbury2013/OF/openFrameworks
develop/apps/HAndLGreenpeace/004BuiltInOfxTimeLine/bin/data/particles

- blob.png
- glitter.png
- sparkle.png
- star.png

as the images for the particles

https://github.com/HellicarAndLewis/MulticolouredMagic/blob/master/Somantics/src/somantics/Sparkles

Do this with the depth image as input to the blob tracker - or the IR image?

Get all three modes working first, then have a think about how to get them working as MODES - make a mode object? Look at somantics for how to have scenes. Maybe call it a mode? Construct with a pointer to the test app for easier data steal. Have a vector of things. Just making sparkles for now, made a sparkle cloud, duplicated the spartkcles logic from marekes sparkles from somantics - the one that spawns along the edges of the blobs... great way of doing it! Going to need an FBO to draw the Sparkles into, so looking at:

/Users/joel/Documents/Projects/HellicarAndLewis/greenpeaceArcticGlastonbury2013/OF/openFrameworks
develop/examples/gl/fboTrailsExample

Lets make it first, then do a InstallationMode object, based on what Sparkles actually needed. tightly coupling into testApp at the moment with a passed pointer, but whatever works for now...Compilation problems, forward declaration because of pointers to testApp...

http://stephanschulz.ca/downloads/singleton.zip

had a look, from :

http://forum.openframeworks.cc/index.php/topic,12466.msg54372.html#msg54372

the first one i found was about singletons which allows you to have global variables that can be accessed by all classes; i.e. all .cpp files. Bollocks to singletons... bad for test app, decoupled...trying to get the FBO to play nicely with the slitscan and the sparkles

FBO->SLITSCAN is working:

RGB fbo!

```
ofImage distortionMap;
distortionMap.allocate(someSparkles.theFBO.getWidth(),
    someSparkles.theFBO.getHeight(), OF_IMAGE_COLOR);
someSparkles.theFBO.readToPixels(distortionMap.getPixelsRef());
distortionMap.resize(timeline.getVideoPlayer("video")->getWidth(),
    timeline.getVideoPlayer("video")->getHeight());
slitScan.setDelayMap(distortionMap);
```

setup:

```
theFBO.allocate(aWidth, aHeight, GL_RGB);
```

draw:

```
theFBO.begin();
ofSetColor(ofColor::black);
ofRect(0,0,theFBO.getWidth(), theFBO.getHeight());
ofSetColor(ofColor::white);
float circleX = theFBO.getWidth()/2.f;
float circleY = theFBO.getHeight()/2.f;
float circleRadius = min(circleX, circleY);
ofCircle(circleX,circleY, circleRadius);
theFBO.end();
```

So the bug is currently with how the cloud of sparkles is being drawn - is the contour finder being read properly? I'm trying to draw at:581814,23197.4, at size:13.0935 is where things were trying to draw! co-ordinates must be in pixels inside the contour tracker! dumb.....sorted it with:

```
void Sparkles::update(ofxCvContourFinder* aContourFinder){
    float cloudWidth = theFBO.getWidth();
    float cloudHeight = theFBO.getHeight();

    float contourWidth = aContourFinder->getWidth();
    float contourHeight = aContourFinder->getHeight();

    float widthRatio = cloudWidth/contourWidth;
    float heightRatio = cloudHeight/contourHeight;
```

```
    // now just stick some particles on the contour and emit them
       randomly
    for(int i = 0; i < aContourFinder->nBlobs; i++) {
        int step = 10;//contourFinder.blobs[i].pts.size()/10;
        for(int j = 0; j < aContourFinder->blobs[i].pts.size();
           j+=step) {
            cloud.spawn(
                    (aContourFinder->blobs[i].pts[j].x)*widthRatio,
                    (aContourFinder->blobs[i].pts[j].y)*heightRatio,
                    ofRandom(-5, 5), ofRandom(-5, 5));
        }
    }
}
```

OK that works

SECOND:

On 17 Jun 2013, at 19:01, Joel Gethin Lewis wrote: This is the mirror: https://github.com/HellicarAndLewis/Multi
Just do a vertical scene and a horizontal scene for now - kaledscope later...DONE... just the vertical one for now...

THIRD:

Paint as a slitscan input. Lets do paint! it's fun... - it is fun! It looks nice....DONE Quick optimisation - why are both slitscans done separately? Changed it, all seems fine. BUG: when switching to slitscan basic from sparkles, you don't get any update of the slitscan image, but it works initially...dirty hack to make work - change prevslitscan to -1 if it's not slitscan basic mode...

OK next! sleep...

21.10.0.30 20th June 2013

FOURTH:

Use the blobs from the depth image - make a bunch of triangles in box2d as greyscale image that floats up and ADD that to slitscan...dropped in box2d, all working ok: OF/openFrameworks-develop/apps/HAndLGreenpeace/006AddingOFXBox2D bit slow, need to look at optimising...

21.10.0.31 21st June 2013

grabbing some addons:

- https://github.com/maxillacult/ofxPostGlitch

- https://github.com/outsidecontext/ofxPSLevels
- https://github.com/neilmendoza/ofxPostProcessing - http://www.neilmendoza.com/ofxpostpro
- https://github.com/julapy/ofxOpticalFlowLK
- https://github.com/timscaffidi/ofxOpticalFlowFarneback
- https://github.com/Flightphase/ofxCvOpticalFlowLK

doing a quick look at them before supper.... lazy! look at in the morning....the next evening! ok addons first....

doing this:

ofxCvOpticalFlowLK - no readme, no draw, moving on - could use the draw image into an FBO easily

ofxOpticalFlowFarneback - OF/openFrameworks-develop/apps/bunchOfAddonsTrying/ofxOpticalFlow looks beautiful! definitely develop this one post stripped down version.... easy conversion to greyscale for the coloured one

ofxOpticalFlowLK - OF/openFrameworks-develop/apps/bunchOfAddonsTrying/ofxOpticalFlowLK similar look to ofxOpticalFlowFarneback but not as pretty, use the other...

ofxPostGlitch - OF/openFrameworks-develop/apps/bunchOfAddonsTrying/ofxPostGlitch lots of fun effects and already in an FBO! just do these effects on either the live video OR the depth image, but put into the slitscan

ofxPostProcessing 3D demo, with

```
post.createPass<FxaaPass>()->setEnabled(false);
post.createPass<BloomPass>()->setEnabled(false);
post.createPass<DofPass>()->setEnabled(false);
post.createPass<KaleidoscopePass>()->setEnabled(false);
post.createPass<NoiseWarpPass>()->setEnabled(false);
post.createPass<PixelatePass>()->setEnabled(false);
post.createPass<EdgePass>()->setEnabled(false);
```

nice, but all in 3D - doing a quick hack to draw the video grabber in the scene. No, couldn't get it working, need to draw it to a texture and draw in space, no thank you...

ofxPSLevels

```
s += "\nbrightness␣(b/B)␣:␣" + ofToString(levels.brightness);
s += "\ncontrast␣(c/C)␣:␣" + ofToString(levels.contrast);
s += "\nsaturation␣(s/S)␣:␣" + ofToString(levels.saturation);
s += "\ngamma␣(g/G)␣:␣" + ofToString(levels.gamma);
```

nice to have this as a post effect for everything.

OK, lets get on with the other direction mirror AND the kaleidoscope..... should be relatively straight forward, just drop in for now. get rid of box2d?

taking out ofxbox2d - that;s better, but why is just video playback so slow? Having a look now... optimised the video only draw section...Still not fast, do it through the fucking still scan with the image that does nothing....taking out colour.....made a few more slitscans...

- ALLBLACK.png
- ALLWHITE.png
- NOHelvetica.png
- Rewind.png
- YesGillSans.png

left in bangs..turned on snapping...they look ok, can work on these...working on horizontal mirror, made notes, did it not quite right...

This is wrong:

```
case SLIGHTLY BUGGERED RERVERSED VERTICAL MIRROR:
{
    ofxCvColorImage mirrorImage;
    mirrorImage.allocate(timeline.getVideoPlayer("video")->getWidth(),
       timeline.getVideoPlayer("video")->getHeight());
    mirrorImage.setFromPixels(timeline.getVideoPlayer("video")->getPixels(),
       mirrorImage.getWidth(), mirrorImage.getHeight());
    mirrorImage.updateTexture();
    bool usingNormTexCoords = ofGetUsingNormalizedTexCoords();

    if(!usingNormTexCoords) {
        ofEnableNormalizedTexCoords();
    }

    mirrorImage.getTextureReference().bind();

    ofMesh mesh;
    mesh.clear();
    mesh.addVertex(ofVec3f(0, 0));
    mesh.addVertex(ofVec3f(0, ofGetHeight()));
    mesh.addVertex(ofVec3f(ofGetWidth()/2, 0));
    mesh.addVertex(ofVec3f(ofGetWidth()/2, ofGetHeight()));
    mesh.addVertex(ofVec3f(ofGetWidth(), 0));
    mesh.addVertex(ofVec3f(ofGetWidth(), ofGetHeight()));
    mesh.addTexCoord(ofVec2f(0, 0.25));
    mesh.addTexCoord(ofVec2f(0, 0.75));
    mesh.addTexCoord(ofVec2f(1.0, 0.25));
    mesh.addTexCoord(ofVec2f(1.0, 0.75));
    mesh.addTexCoord(ofVec2f(0, 0.25));
    mesh.addTexCoord(ofVec2f(0, 0.75));
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_STRIP);
    ofSetColor(ofColor::white);
    mesh.draw();
```

```
    mirrorImage.getTextureReference().unbind();

    // pop normalized tex coords
    if(!usingNormTexCoords) {
        ofDisableNormalizedTexCoords();
    }
    break;
}
```

This is right

```
case HORIZONTALMIRROR:
{
    ofxCvColorImage mirrorImage;

    mirrorImage.allocate(timeline.getVideoPlayer("video")->getWidth(),
       timeline.getVideoPlayer("video")->getHeight());
    mirrorImage.setFromPixels(timeline.getVideoPlayer("video")->getPixels(),
       mirrorImage.getWidth(), mirrorImage.getHeight());
    mirrorImage.updateTexture();

    bool usingNormTexCoords = ofGetUsingNormalizedTexCoords();
    if(!usingNormTexCoords) {
        ofEnableNormalizedTexCoords();
    }

    mirrorImage.getTextureReference().bind();

    ofMesh mesh;
    mesh.clear();
    mesh.addVertex(ofVec3f(ofGetWidth(), 0));
    mesh.addVertex(ofVec3f(0, 0));
    mesh.addVertex(ofVec3f(ofGetWidth(), ofGetHeight()/2));
    mesh.addVertex(ofVec3f(0, ofGetHeight()/2));
    mesh.addVertex(ofVec3f(ofGetWidth(), ofGetHeight()));
    mesh.addVertex(ofVec3f(0,ofGetHeight()));
    mesh.addTexCoord(ofVec2f(1.0, 0.25));
    mesh.addTexCoord(ofVec2f(0.0, 0.25));
    mesh.addTexCoord(ofVec2f(1.0, 0.75));
    mesh.addTexCoord(ofVec2f(0.0, 0.75));
    mesh.addTexCoord(ofVec2f(1.0, 0.25));
    mesh.addTexCoord(ofVec2f(0.0, 0.25));
    mesh.setMode(OF_PRIMITIVE_TRIANGLE_STRIP);
    ofSetColor(ofColor::white);
    mesh.draw();

    mirrorImage.getTextureReference().unbind();
```

```
    // pop normalized tex coords
    if(!usingNormTexCoords) {
        ofDisableNormalizedTexCoords();
    }
}
```

Getting a bit better...fixing the controls - some of the keys were clashing. Red lines on the screen indicate track in a and out below the main timeline.

```
Keys for Duration/ofxTimeline:

Note on OS X the COMMAND key is used, on Linux and Windows the CTRL
   key is used

Function    Shortcut
Cut Selection   command+x
Copy Selection  command+c
Paste Selection command+v
Undo    command+z
Redo    shift+command+z
Select all keyframes in Focused track   command+a
Add all keyframes in Focused track to selection command+shift+a
Delete all selected keyframes   delete or backspace
Nudge keyframes a little    arrow keys
Nudge keyframes a little more   shift+arrow keys
Expand Focused track    alt+e
Collapse all tracks alt+c
Evenly distribute track sizes   alt+shift+c
```

Sped things up by taking off vertical sync and smoothing too, 30fps. Did kaleidoscope, little bugs I think...turned the update into a proper switch statement, really improved performance! All good, enough for tonight....

21.10.0.32 23rd June 2013

OK first thing to do is to take over all the Kinect stuff from:

cariad/reactickles/oF/openFrameworks-develop/apps/zHarp/006withMemoLogic

So taking that over now....Making it all in:

OF/openFrameworks-develop/apps/HAndLGreenpeace/008NewKinectAndPsychBear

Taking over the code, adding the display to the blank scene.... Trying to get the saving working....got it working - it was the bad characters! : and &. That's working, now neatening up the gui screen, adding a blank screen and taking out pointless Kinect modes. OK thats nice, now lets get the psych fur working...All in and the gui in too!

OF/openFrameworks-develop/apps/HAndLGreenpeace/009ShatterExperiment

trying shatter...trying to make it work but there seems to be a conflict when I try to include box2d. Hmmm

all i had to do was change shatter.h to :

```
include "ofMain.h"
include "ofxOpenCv.h"
include "ofxBox2D.h"
```

from:

```
include "ofMain.h"
include "ofxBox2D.h"
include "ofxOpenCv.h"
```

Via OF Forum post: http://forum.openframeworks.cc/index.php?topic=7165.0 :

paulf london Posts: 22 Re: Weird codeblocks 007 build errors Reply #5 on: April 05, 2012, 02:12:39 PM in testApp.h having #include “ofxOpenCv.h” at the top of my include list solved the issue for me

Crazy... OK. got that working, but way too slow...

```
float timeSinceLastShatter = ofGetElapsedTimef() - timeOfLastShatter;

if(timeSinceLastShatter > 10.f){ //every 2 seconds make some more....
    float shatterWidth = theFBO.getWidth();
    float shatterHeight = theFBO.getHeight();
    float contourWidth = aContourFinder->getWidth();
    float contourHeight = aContourFinder->getHeight();
    float widthRatio = shatterWidth/contourWidth;
    float heightRatio = shatterHeight/contourHeight;

    // now just stick some particles on the contour and emit them
       randomly
    for(int i = 0; i < aContourFinder->nBlobs; i++) {
        int step = 20;

        shape.clear();

        for(int j = 0; j < aContourFinder->blobs[i].pts.size();
           j+=step) {
            shape.addVertex((aContourFinder->blobs[i].pts[j].x)*widthRatio,
                            (aContourFinder->blobs[i].pts[j].y)*heightRatio);
        }

        // This is the manual way to triangulate the shape
        // you can then add many little triangles
        // first simplify the shape
```

```
        shape.simplify();
        // save the outline of the shape
        ofPolyline outline = shape;
        // resample shape
        ofPolyline resampled = shape.getResampledBySpacing(256);
            //dude
        //ofPolyline resampled = shape.getResampledBySpacing(100);
        // triangulate the shape, return am array of triangles
        vector <TriangleShape> tris =
            triangulatePolygonWithOutline(resampled, outline);
        // add some random points inside
        addRandomPointsInside(shape, 255);

        // now loop through all the tri's and make a box2d triangle
        for (int i=0; i<tris.size(); i++) {
            ofxBox2dPolygon p;
            p.addTriangle(tris[i].a, tris[i].b, tris[i].c);
            p.setPhysics(1.0, 0.3, 0.3);
            p.setAsEdge(false);
            if(p.isGoodShape()) {
                p.create(box2d.getWorld());
                triangles.push_back(p);
            }
        }

        // done with shape clear it now
        shape.clear();
    }

    timeSinceLastShatter = ofGetElapsedTimef();
}
```

Let's just spray triangles out from the top of the blobs...like sparkles but with triangles....triangles lame, circles work! Had it running on pete's laptop all lovely...

21.10.0.33 24th June 2013

OK, things to try this morning before lunch:

DONE 1. feed in current frame as greyscale for the slitscan DONE 2. try a slitscan mode where I make a spikey slitscan mode - like in divide by zero, going to need OF/openFrameworks-develop/addons/ofxContourUtil from julapy make the triangles shaded? make an ofMesh of it? DONE 3. try the full video, or the mirror vertical/horizontal for the background of the kaleidescope DONE - flock it .4. try a flock attracted to blobs....

OF/openFrameworks-develop/apps/HAndLGreenpeace/010WithSpikyBlobsFlockAndSelfSlitScan

Made that. starting with 1. SELFSLITSCAN - super easy:

```
if(timeline.getVideoPlayer("video")->isFrameNew()){
    slitScan.setDelayMap(timeline.getVideoPlayer("video")->getPixelsRef());
    slitScan.addImage(timeline.getVideoPlayer("video")->getPixelsRef());
```

Next on to spikey mode! Was going to use:

ofxContourUtil-master

From julapy, but it's all in:

```
void ofPolyline::simplify(float tol){
```

In core, so lets have a go with that...also have:

```
ofPolyline ofPolyline::getSmoothed(int smoothingSize, float
   smoothingShape)
```

This is the logic from Divide by Zero:

```
// contour simplification/manipulation

int numberOfBlobs = videoContourFinder.blobs.size();

if(numberOfBlobs > 0){
        //if we have at least one blob
    curve.resize(numberOfBlobs);
    curveSmooth.resize(numberOfBlobs);
    curveSimplify.resize(numberOfBlobs);
    curveCvSimplify.resize(numberOfBlobs);
    float mx = gui.getValueF("AURA_SIMPLIFICATION");
    float scale1    = mx;
    float scale2    = mx * 140;
    float scale3    = mx * 0.1;
    bool noneSmooth = gui.getValueB("AURA_IS_SMOOTH");
    bool simplifyCV = gui.getValueB("AURA_IS_CV");
    float auraScale = gui.getValueF("AURA_SCALE");
    bool scaleFromStage = gui.getValueB("AURA_SCALE_FROM_STAGE");

    for(int i = 0; i< numberOfBlobs; i++){
        curve[i] = videoContourFinder.blobs[i];
        ofPoint centreOfStage = ofPoint(camWidth/2.f, camHeight);

        if(scaleFromStage){
            curve[i].scaleBlob(centreOfStage, auraScale); //scale
               from the base of stage
        }else {
            curve[i].scaleBlob(curve[i].centroid, auraScale); //else
               do it from the centroid
```

```
        }

        if(noneSmooth){ //smooth it
            cu.smooth( curve[i].pts, curveSmooth[i].pts, scale1 );
        }else{
                //do nothing.
        }

        if(simplifyCV){
                //cv simplify it
            simplifyDP_openCV( curve[i].pts, curveCvSimplify[i].pts,
               scale3 );
        }else{
                //just simplify it
            cu.simplify( curve[i].pts, curveSimplify[i].pts, scale2
               );
        }
    }
}
```

So lets have a look at the demo here:

/Users/joel/Documents/Projects/HellicarAndLewis/greenpeaceArcticGlastonbury2013/OF/openFrameworks develop/examples/graphics/polylineBlobsExample

Very useful demo....used all the code from demo and ofPolyline in general. Looks nice, ended up doing a simplify down to 10 points.... Let's move onto changing the background for the kaleidoscope...did it with the vertical mirror as the background, pretty.. Downloaded:

https://github.com/jefftimesten/CodeForArt https://github.com/jefftimesten/CodeForArt/tree/master/Chap physics/012-flock/src

Is what i used for Whiteheat anyhow... jefftimesten = jeff crouse. Lets use it again...flock it, too slow too lame no fun... lets do it with particles instead...Looks ok. Going to crack on with the movie....changed name to:

OF/openFrameworks-develop/apps/HAndLGreenpeace/010WithSpikyBlobsParticlesAndSelfSlitScan

Doing the audio now and new film in:

2013_06_24_newFilmAndAudio

Copied in and took over to Pete's computer so he could have a play....he is sequencing...

21.10.0.34 25th June 2013

Shower! was lovely. Long drop too.

TODO today: lets do white fur first...looks great...copying over to pete...

DONE 1) kaledscope is always, (2n)+1 : 3,5,7,9,11,13,15,17,19 19 as the limit... which is up to 9 DONE 2) add WHITE FUR to: VERTICALMIRROR, HORIZONTALMIRROR, KALEIDOSCOPE, MIRRORKALEIDOSCOPE, SLITSCANBASIC

copied in the new slitscans:

- 00_ALLBLACK.png
- 01_ALLWHITE.png
- 01_random_grid.png
- down_to_up.png
- left_to_right.png
- right_to_left.png
- soft_noise.png
- Triangle_001.png
- Triangle_002.png
- Triangle_003.png
- Triangle_004.png
- Triangle_005.png
- up_to_down.png

look amazing!

First, kaleidoscope - want always even! timeline.addCurves("star", ofRange(2, 12)); - so just double it... Second, adding white fur.....to VERTICALMIRROR, HORIZONTALMIRROR, KALEIDOSCOPE, MIRRORKALEIDOSCOPE, SLITSCANBASIC

21.10.0.35 26th June 2013

Just changed the fur to not have any alpha (white fur that is) also added non-ofxTimeline GUI to everything...

22 Version control with Git

by Christoph Buchner (bilderbuchi[1])

In this chapter, you will learn about version control and why you should use it. You will get a short introduction to Git, the version control system of choice for openFrameworks. The major concepts and keywords are explained, enabling you to easily dig deeper into the subject using available online resources. A number of tools for working with Git are presented. You will learn about Github, a web service for hosting Git repositories and one of the major platforms for "social coding". You will be shown how you can start hosting your own projects on Github and leverage its features. Finally, you will learn how you can build upon the things you just learned and where you can get help if you get stuck.

22.1 What is version control, and why should you use it?

How do you track the state of your projects over time? Have you ever made several copies of your project folder (or text document, Photoshop file,...)? Did you name them something like `Awesome_project_1`, `Awesome_project_may10`, `Awesome_project_final`, `Awesome_project_final2`, `Awesome_project_really_final`,...? Did you mail around zipped projects to collaborators, and had some trouble synchronizing the changes they (or you) made in the meantime? Have you run into problems when you had to go back to a previous version, change things, copy those changes to other version, or generally keep tabs on changes?

If you nodded at some of these questions, you've come to the right place - version control is there to help you! Version control[2] (also called revision control) is the management of changes to documents, computer programs, large web sites, and other collections of information. It is used to easily, efficiently and reproducibly track changes to all kinds of information (like a really advanced "undo" function). Specifically, in programming it is (primarily) used to track changes to your source code, but it's generally applicable to most kinds of files on your computer. Version control also enables programmers to effectively collaborate in teams, because it offers methods to distribute changes, merge different development versions together, resolve conflicts if two (or more) programmers modified the same file, sync state between computers, etc.

[1] https://github.com/bilderbuchi
[2] http://en.wikipedia.org/wiki/Revision_control

I hope you'll agree by now that it is a very good idea to use some manner of version control when developing your programs. In the next section, I'll talk a bit about the different choices you have when choosing a particular system.

22.2 Different version control systems

There is a number of version control systems out there, some of which you've maybe encountered already. They can be divided into two big camps: "centralized" and "distributed" systems.

In *centralized version control systems*[3], a central server orchestrates the various tasks the version control system performs, and programmers synchronize with this server. Common operations often need a network connection to the central server. Typically, programmers only have a part of the whole project locally available. Some popular centralized version control[4] systems are:

- Concurrent Versions System (CVS)[5], which has introduced the "branching" concept[6] into version control systems.
- Subversion (SVN)[7], a popular successor to CVS.

Distributed version control systems[8], on the other hand, take a peer-to-peer approach. There is no central server, and every (local) repository contains all files and history (thus acting as a backup, too!). Network access is only needed for syncing changes with other programmers. Distributed version control systems have recently gained much popularity. Some notable systems[9] are:

- Git[10] was initially designed and developed by Linus Torvalds for Linux kernel development. Its popularity has recently boomed.
- Bazaar[11] is a distributed version control system created by Canonical (the company behind Ubuntu). It is primarily used on Launchpad[12], a code hosting platform primarily used for developing projects around Ubuntu.
- Mercurial[13] was started around the same time as Git. It is quite similar to Git[14], especially to a newcomer.

[3] http://en.wikipedia.org/wiki/Revision_control
[4] http://en.wikipedia.org/wiki/List_of_revision_control_software
[5] http://savannah.nongnu.org/projects/cvs
[6] http://en.wikipedia.org/wiki/Concurrent_Versions_System#History_and_status
[7] http://subversion.apache.org/
[8] http://en.wikipedia.org/wiki/Distributed_revision_control
[9] http://en.wikipedia.org/wiki/List_of_revision_control_software
[10] http://en.wikipedia.org/wiki/Git_%28software%29
[11] http://bazaar.canonical.com/
[12] launchpad.net
[13] http://mercurial.selenic.com/
[14] http://stackoverflow.com/questions/35837/what-is-the-difference-between-mercurial-and-git

22.3 Introduction to Git

openFrameworks uses Git to version-control the code base, and relies on Github[15] as a hosting platform for the code and the issue tracker. In this section, I'll give you an introduction on how to use version control with your openFrameworks project, and introduce the relevant concepts and commands as they are encountered.

Note that this chapter is only an introduction, and as such only touches the surface of Git's capabilities, both in the presented commands, and their options. Much more detailed information, including in-depth tutorials and a command reference, can be found online. Some links are collected at the end of the chapter, and most commands presented have a link to their online reference.

In what follows, I'll explain the basic concepts of Git. After that, I'll show the typical operations involved in using Git with an openFrameworks project in a walk-through fashion. Then I will show you how to work with remote Git servers.

22.3.1 Basic concepts

When you put your project (which is contained in a directory on your disk) under version control, Git creates a **repository** in your project directory. This means that the contents of that folder are tracked with Git. Most of the files associated with Git are in the `.git` folder in your project root (the leading dot means this folder is by probably hidden from view in your file browser by default).

The basic element for tracking the history of the repository is the **commit**. This is basically a snapshot of the repository's state at the time of the commit, including a **commit message** and any parent commit(s). Think of it as a checkpoint for saving in a videogame. It has a unique identifier called the **hash** (or **SHA**). This is a checksum calculated from the commit's contents. It's impossible to change any part of the commit without the hash changing. Thus, a commit hash uniquely defines a commit and the whole history preceding it.

As your work proceeds, you will be adding commits, describing the things you change. These commits will form a chain of commits, making up the project history. A chain of such commits is called a **branch**, and the default branch is called `master`. Branches can also be created when you decide to diverge from a line of development, and try something different (for example a new feature, or a bug fix) while preserving the state of the project. This new chain of commits, which *branches off* at a certain commit of the original branch, now forms its own branch.

Branches can be **merged** into another branch. When this happens, Git analyzes the two different branches and merges their different histories/changes together.

This figure visualizes how this looks like:

[15] www.github.com

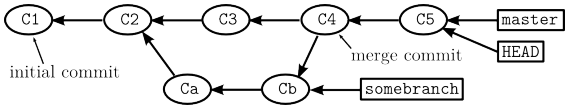


Figure 22.1: Simple Git branch diagram.

Finally, there are three different "areas" in Git, which you will encounter often when reading about Git:

The **repository** in the `.git` directory contains all the commits. The **HEAD** points to the current commit of the branch you are currently on. This represents the *latest committed state* of your repository. If you create a new commit, it will become this commit's parent (and HEAD will be moved to the new commit).

The **working directory** contains the files and folders under version control, the stuff you modify and work with when writing code for your project.

When you prepare a commit, you first have to **stage** any changes you want that commit to contain. This means that these changes will be put into the **index** (or **staging area**).

So, you modify your files in the working directory, you stage those modifications, putting them into the index, and then you **create a commit**, taking the files from the index and storing them in the repository. To get files *back* from the repository (i.e. restore the state as it was at some previous point), you **check out** files, putting them into the working directory. This is shown graphically in the following figure:

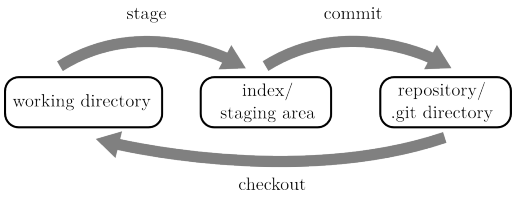


Figure 22.2: The three areas of Git.

Armed with these basic facts, we can dive right in, and start working on our first project!

22.3.2 Getting started: project setup

You can follow along with this section by entering the given commands (in the line starting with `$`) into your terminal. You can also use a Git program with a GUI if you want (some will be presented later in the chapter), but you will have to figure out which actions correspond to the respective terminal commands.

Much of what follows will be less tedious to achieve, and presented in a prettier way, if you're using a GUI to interact with Git. Nevertheless, I am presenting this intro with a terminal-based approach for several reasons:

- I think it's actually more instructive to follow some typed commands than pages after pages of (rapidly outdated) screenshots of a GUI app you probably don't even use (as there are quite a lot of them out there).
- Many GUI programs don't offer the full range of functions that Git provides, so you will probably have to drop down into a terminal sooner or later. At that point it's quite handy to know what your GUI does in the background.
- Most of the online documentation and advice on Git focus on the command-line interface.

First, we have to set up Git itself for our operation system. This mainly involves downloading and installing[16] and setting the username and email address. The instructions vary slightly depending on the operating system[17].

When we have successfully set up Git, we create a new, **empty project** with the OF project generator. We will end up with a project folder containing some C++ files and some IDE files depending on OS and chosen IDE (in my case: Linux and Code::Blocks). This will look similar to this:

```
$ tree -a --charset ascii
.
|-- addons.make
|-- bin
|   `-- data
|       `-- .gitkeep
|-- config.make
|-- demoProject.cbp
|-- demoProject.workspace
|-- Makefile
`-- src
    |-- main.cpp
    |-- ofApp.cpp
    `-- ofApp.h

3 directories, 9 files
```

Now, it's time to create our Git repository, using the `git init` command:

```
$ git init
Initialised empty Git repository in
   /home/bilderbuchi/demoProject/.git/
```

[16] http://git-scm.com/downloads
[17] https://help.github.com/articles/set-up-git#platform-all

22.3.2.1 .gitignore

One thing we should do right at the beginning is add a special Git file called `.gitignore`[18] to the root of our repository.

It's important that the Git repository contains all files necessary to successfully compile our program, but no unnecessary stuff. Generally, this means that files we edit by hand (e.g. source and header files, Readme files, images,...) should be included in the repository, but files which are *generated from* our code (e.g. compiled binaries, pdfs generated from some source file, video files or image sequences created with our program) should stay out. Also, user-specific files like IDE files describing the location of windows in our IDE, or backup copies of our files that the OS creates, don't really belong in the repository.

If we take care of this right at the beginning, we can easily make sure that only "proper" files end up in our repo. Git handles this file exclusion with the aforementioned `.gitignore` files (there can be several), which contains patterns describing which files are ignored by Git. Those ignored files will still exist in our working directory, that means we can still use them, but Git will not track them.

If, later down the line, we see files appearing in our list of changes which should not be there, or if we can't seem to add a file that belongs in the repository, we don't force Git to do what it doesn't want to, rather fine-tune the `.gitignore` pattern to match our expectations. Note that the `.gitignore` pattern does *not* affect files that have already been committed.

Because it can be daunting to come up with a generally useful `.gitignore` template, it's currently planned[19] that OF offers to add a pre-made `.gitignore` file when we create our project. This file will look similar to this (formatted into three columns for convenience):

```
$ pr -tW84 -s"|" -i"␣"1 -3 .gitignore
############################|                    |.externalToolBuilders
# ignore generated binaries|# XCode             |
# but not the data folder  |*.pbxuser           |##################
############################|*.perspective       |# operating system
                           |*.perspectivev3     |##################
/bin/*                     |*.mode1v3           |
!/bin/data/                |*.mode2v3           |# Linux
                           |# XCode 4           |*~
#########                  |xcuserdata          |# KDE
# general                  |*.xcworkspace       |.directory
#########                  |                    |.AppleDouble
                           |# Code::Blocks       |
[Bb]uild/                  |*.depend            |# OSX
[Oo]bj/                    |*.layout            |.DS_Store
```

[18] http://git-scm.com/docs/gitignore
[19] https://github.com/openframeworks/openFrameworks/issues/2791

```
*.o                          |                    |*.swp
[Dd]ebug*/                   |# Visual Studio     |*~.nib
[Rr]elease*/                 |*.sdf               |# Thumbnails
*.mode*                      |*.opensdf           |._*
*.app/                       |*.suo               |
*.pyc                        |*.pdb               |# Windows
.svn/                        |*.ilk               |# Image file caches
*.log                        |*.aps               |Thumbs.db
                             |ipch/               |# Folder config file
#########################    |                    |Desktop.ini
# IDE files which should     |# Eclipse           |
# be ignored                 |.metadata           |# Android
#########################    |local.properties    |.csettings
```

This might look like magic to you, but let us just continue for now, you can always look up more information on the `.gitignore` syntax later, for example here[20].

22.3.2.2 `git status`

A command which we will use very often is `git status`[21]. This command enables us to see the current state of a repository at a glance. It offers some flags to fine-tune its output (like most Git commands).

Alright, let's use `git status -u` to see what's going on in our repository. The `-u` flag makes sure we see *all* untracked files, even in subdirectories:

```
$ git status -u
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    .gitignore
#    Makefile
#    addons.make
#    bin/data/.gitkeep
#    config.make
#    demoProject.cbp
#    demoProject.workspace
#    src/main.cpp
#    src/ofApp.cpp
#    src/ofApp.h
```

[20] http://git-scm.com/docs/gitignore
[21] http://git-scm.com/docs/git-status

```
nothing added to commit but untracked files present (use "git␣add"
    to track)
```

The output tells us which branch we are currently on (`master`), that we haven't committed anything yet, and that there are a couple of untracked files (i.e. not yet known to Git) in the repository and, importantly, what we should do next. Using `git status -s` is an option to get more condensed output.

The list of files looks correct, so far so good! You might have noticed the `.gitkeep` file in `bin/data/`. Git only tracks files, not directories, which means that empty directories are not visible to Git. A common technique to work around this, if you want to have empty directories (e.g. for future output files) in your repository, is to place an empty file there, which makes sure that that directory can be added to Git. Naming that file `.gitkeep` is just a convention, and has no special meaning to Git.

If we compile the OF project now, some additional files will be created in the /bin folder. Because we added a `.gitignore` file in the previous step, these files will not be picked up by Git. We can check this by running `git status -u` again.

22.3.2.3 `git add`

The next step is to *stage* the untracked files using `git add`[22]. This will put those files into the *index*, as discussed previously.

We stage untracked files and modifications to files already in the repository with the command `git add <filespec>`, where `<filespec>` describes one or more files or directories, so could be for example `addons.make`, `src` or `*.cpp`.

We can also add *all* files and modifications in the repository with `git add .`, but as this is a catch-all filespec, we will have to check the output of `git status -u` first, to make sure no unwanted files are missed by the `.gitignore` pattern and would slip into the repository! Since we just made sure our git status looks alright, let's do it:

```
$ git add .
```

You will notice a change when we run `git status` next:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
```

[22] http://git-scm.com/docs/git-add

```
#    new file:   .gitignore
#    new file:   Makefile
#    new file:   addons.make
#    new file:   bin/data/.gitkeep
#    new file:   config.make
#    new file:   demoProject.cbp
#    new file:   demoProject.workspace
#    new file:   src/main.cpp
#    new file:   src/ofApp.cpp
#    new file:   src/ofApp.h
#
```

All those untracked files are now in the "Changes to be committed" section, and so will end up in the next commit we make (if we don't unstage them before that).

To unstage changes we have accidentally staged, we use `git rm --cached <file>` (for newly added files) or `git reset HEAD <file>` (for modified files). As usual, `git status` reminds us of these commands where appropriate.

22.3.2.4 `git commit`

Now that we've prepared the staging area/index for our first commit, we can go ahead and do it. To this end we will use `git commit`[23]. We can supply a required commit message at the same time by using the `-m` flag, otherwise Git will open an editor where we can enter a message (and then save and exit to proceed).

```
$ git commit -m "Add␣initial␣set␣of␣files."
[master (root-commit) 3ef08e9] Add initial set of files.
 10 files changed, 388 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Makefile
 create mode 100644 addons.make
 create mode 100644 bin/data/.gitkeep
 create mode 100644 config.make
 create mode 100644 demoProject.cbp
 create mode 100644 demoProject.workspace
 create mode 100644 src/main.cpp
 create mode 100644 src/ofApp.cpp
 create mode 100644 src/ofApp.h
```

The first line of the output shows us the branch we are on (`master`), and that this was our first commit, creating the root of our commit tree. Also, we see the hash (i.e. the unique ID) of the commit we just created (`3ef08e9`) and the commit message. The hash is given in a short form, as it's often sufficient to only supply the first seven or so characters of the hash to uniquely identify a commit (Git will complain if that's

[23] http://git-scm.com/docs/git-commit

not the case). The next line roughly describes the changes that were committed, how many files were changed and how many insertions and deletions were committed. The rest lists the files new to Git, the mysterious `mode 100644` is a unix-style description of the file permissions, `100644` is a regular, non-executable file (`100755` would be an executable file).

Now, let's check our status to see what's going on in the repository:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Hooray, that's the all-clear, all systems green message! It means that the working directory as it is right now is already committed to Git (with the exception of ignored files). It's often a good idea, whenever you start or stop working in a repository, to start from this state, as you will always be able to fall back to this point if things go wrong.

Now that we have made our initial commit, we can make our first customizations to the OF project. Onwards!

22.3.3 First edits

OK, we have a clean slate now, so let's start playing around with our OF project. A programming tutorial wouldn't be complete without saying hello to the world, so let's do that: Open `ofApp.cpp`, and in the implementation of `void ofApp::setup()`, add an appropriate message, e.g. `cout << "Hello world";`, and save the file.

We have just made a modification to a file that Git is tracking, so it should pick up on this, right? Let's check, using `git status` (you hopefully already guessed that part):

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
   directory)
#
#    modified:   src/ofApp.cpp
#
no changes added to commit (use "git␣add" and/or "git␣commit␣-a")
```

Alright, Git tells us it knows that we modified `ofApp.cpp`. Note that now the entry is in a section called "Changes not staged for commit", not "untracked files" as before. Again, `git status` offers instructions for what we could want to do next, very convenient.

22.3.3.1 `git diff`

Now, let's find out what exactly we changed in `ofApp.cpp`. For this, `git diff`[24] is used. It can be used to compare states between all kinds of areas (check out the examples section of the man page[25]), but in its simplest form, `git diff`, allows us to view the changes we made relative to the index (staging area for the next commit). In other words, the differences are what we could tell Git to further add to the index but we still haven't." (from the `man page`[26]). (Use the `--staged` option to see the diff of *already staged* changes.) Let's check it out:

```
$ git diff
diff --git a/src/ofApp.cpp b/src/ofApp.cpp
index b182cce..8018cf7 100644
--- a/src/ofApp.cpp
+++ b/src/ofApp.cpp
@@ -3,6 +3,7 @@
 //--------------------------------------------------------------
 void ofApp::setup(){

+    cout << "Hello world!";
 }

 //--------------------------------------------------------------
```

This output shows the difference between two files in the unified diff format[27] (`diff` is a popular Unix tool for comparing text files). It looks pretty confusing, but let's pick it apart a bit to highlight the most useful parts.

The first line denotes the two different files being compared, denoted as `a/...` and `b/...`. If we have not just renamed a file, those two will typically be identical.

The next couple of lines further define what exactly is being compared, but that's not interesting to us until we come to the line starting with `@@`. This defines a so-called "hunk", which means it tells us what part of the file is being shown next. In the *original* state (prefixed by `-`), this section starts at line 3, and goes on for 6 lines. In the new state (prefix `+`), the section starts at line 3, and goes on for 7 lines.

Next, we see the actual changes, with a couple of lines around it for context. Unchanged lines start with a space, added lines with a `+`, and removed lines with a `-`. A modified line will show up as a removed line, followed by an added line. We can see that one line containing a hello world message was added.

Now that we have made some changes, we can compile and run the program to confirm it works as expected, and we didn't make a mistake. Then, we can prepare a commit

[24] http://git-scm.com/docs/git-diff
[25] http://git-scm.com/docs/git-diff
[26] http://git-scm.com/docs/git-diff
[27] http://en.wikipedia.org/wiki/Diff#Unified_format

with the modification, as before:

```
$ git add src/ofApp.cpp

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   src/ofApp.cpp
#
```

Before we commit again, a couple of words regarding commits and commit messages:

First, a good mantra to remember is to **"commit early, commit often"**. This means you should create frequent small commits, whenever a somewhat *self-contained element of code* is finished (e.g. implementing a new function, fixing a small bug, refactoring something) as opposed to whole features (e.g. "Implement optical flow tracking for my installation.") or mixing different changes together ("Updated Readme, increased performance, removed unused Kinect interaction.") . The reasoning behind this is that it creates a fine-grained trail of changes, so when something breaks, it is easier to find out which (small) change caused the problem (e.g. with tools like `git bisect`[28]), and fix it.

Second, **write good commit messages**. This will make it easier for everybody else, and future you in a couple of months, to figure out what some commit does. A good, concise convention for how a good commit message looks can be found here[29]. In short, you should have a short (about 50 characters or less), capitalized, imperative summary (e.g. "Fix bug in the file saving function"). If this is not enough, follow this with a blank line(!) and a more detailed summary, wrapping lines to about 72 characters. *(If your terminal supports this, omitting the second quotation mark allows you to enter multiple lines. Otherwise, omit the* `-m` *flag to compose the commit message in the editor.)*

Now that that is out of the way, we can commit the change we just added, and check the status afterwards:

```
$ git commit -m "Add␣welcome␣message"
[master e84ba14] Add welcome message
 1 file changed, 1 insertion(+)

$ git status
# On branch master
nothing to commit, working directory clean
```

[28] http://git-scm.com/docs/git-bisect
[29] http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

22.3.4 Branches and merging

Branches and merging[30] are the bread and butter of Git, so you will be branching and merging a lot. Branching and merging often is a workflow encouraged by Git, as those are computationally cheap operations. We are getting into some slightly more-advanced stuff, so if you don't quite grasp it right away don't be worried.

For example, if we want to create some new feature, or fix a bug in our program, it is prudent to start this work on a branch separated from the main branch. This has several advantages:

- Our work on is contained in this branch.
- We can quickly and easily switch to another topic of work if needed.
- The main branch is unaffected by our work as long as it's not merged, so normal operations can continue in the meantime (e.g. when we create an experimental addition to an OF addon other people are using).

22.3.4.1 `git branch` and `git checkout`

To get a list of the branches in a repository, we use `git branch`[31] without arguments (`*` denotes the current branch). Since we only have one branch for now, this is not very exciting:

```
$ git branch
* master
```

To create a new branch, we use `git branch <branchname>`. To then check out that branch, to make it the current one, we use `git checkout <branchname>`[32]. There is a shorter way to achieve both operations in one, using `git checkout -b <branchname>`, so let's use that to create **and** check out a new branch called `celebration`:

```
$ git checkout -b celebration
Switched to a new branch 'celebration'

$ git branch
* celebration
  master
```

To celebrate, let's add a second message after the "Hello World", e.g. `cout << "Yeah, it works";`. Let's confirm that Git has picked this up, using `git diff` as before:

[30] http://git-scm.com/book/en/Git-Branching
[31] http://git-scm.com/docs/git-branch
[32] http://git-scm.com/docs/git-checkout

```
$ git diff
diff --git a/src/ofApp.cpp b/src/ofApp.cpp
index 8018cf7..59d84fa 100644
--- a/src/ofApp.cpp
+++ b/src/ofApp.cpp
@@ -4,6 +4,7 @@
 void ofApp::setup(){

     cout << "Hello world!";
+    cout << "\nYeah, it works!";
 }

 //--------------------------------------------------------------
```

If we run `git status` again, it will show that we are on the `celebration` branch now:

```
$ git status
# On branch celebration
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
   directory)
#
#    modified:   src/ofApp.cpp
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Because we have confirmed that the list of changes to be staged and committed is alright, we can take a little shortcut. Using the `-a` flag of `git commit`[33], we can tell Git to automatically stage modified or deleted files (new files are not affected) when committing, so we can skip the `git add` step:

```
$ git commit -am "Let us celebrate!"
[celebration bc636f4] Let us celebrate!
 1 file changed, 1 insertion(+)

$ git status
# On branch celebration
nothing to commit, working directory clean
```

22.3.4.2 Merging branches

Let us meanwhile add a feature on the `master` branch. First, we check out the `master` branch:

[33] http://git-scm.com/docs/git-commit

```
$ git checkout master
Switched to branch 'master'
```

Now, we can add some code to switch the background color if the space key is pressed. You should be able to recognize what I did by looking at the output of `git diff`. Try to approximately replicate these changes:

```
$ git diff
diff --git a/src/ofApp.cpp b/src/ofApp.cpp
index 8018cf7..f7e3dbc 100644
--- a/src/ofApp.cpp
+++ b/src/ofApp.cpp
@@ -18,12 +18,16 @@ void ofApp::draw(){

 //--------------------------------------------------------------
 void ofApp::keyPressed(int key){
-
+    if (key == ' ') {
+        ofBackground(255);
+    }
 }

 //--------------------------------------------------------------
 void ofApp::keyReleased(int key){
-
+    if (key == ' ') {
+        ofBackground(127);
+    }
 }

 //--------------------------------------------------------------
```

Next, we again commit quickly (as we already checked the modifications to be committed with `git diff`):

```
$ git commit -am "Add␣background␣switching"
[master 7cc405c] Add background switching
 1 file changed, 6 insertions(+), 2 deletions(-)
```

22.3.4.3 `git log`

To show commit logs, we can use the `git log`[34] command. In it's default form, `git log` shows a plain list of commits, printing their hashes, timestamp, author and commit message. Its output is heavily customizable, and one nice thing we can do is generate

[34] http://git-scm.com/docs/git-log

a primitive tree view with this incantation *(which you could save under an alias[35] to make it shorter, but this is out of scope for this tutorial)*:

```
$ git log --all --graph --decorate --oneline
* bc636f4 (celebration) Let us celebrate!
| * 7cc405c (HEAD, master) Add background switching
|/
* e84ba14 Add welcome message
* 3ef08e9 Add initial set of files.
```

We see the whole history of the repository here, showing the branch structure, including brief commit messages. We can also see the branch tips in parentheses, and also the current branch (where `HEAD` points to). We realize that the "Let us celebrate" commit is not yet included in `master`, so let's do that now!

22.3.4.4 `git merge`

We can now merge the `celebration` branch back into our `master` branch to make our celebratory message available there too. This happens with the `git merge`[36] command. We use `git merge <branchname>` to merge another branch into the current branch, like so:

```
$ git merge celebration
Auto-merging src/ofApp.cpp
Merge made by the 'recursive' strategy.
 src/ofApp.cpp | 1 +
 1 file changed, 1 insertion(+)
```

Git automatically figured out how to merge `ofApp.cpp` so that `master` now contains the modifications from both branches (go ahead and take a look at `ofApp.cpp` now).

The tree view now looks like this:

```
$ git log --all --graph --decorate --oneline
*   1c6d4aa (HEAD, master) Merge branch 'celebration'
|\
| * bc636f4 (celebration) Let us celebrate!
* | 7cc405c Add background switching
|/
* e84ba14 Add welcome message
* 3ef08e9 Add initial set of files.
```

We can see that the two branches have been merged together successfully, so `master` now contains all our modifications. Also note that the `celebration` branch is unaffected by the merge, it's still pointing to its original commit.

[35] http://stackoverflow.com/questions/2553786/how-do-i-alias-commands-in-git
[36] http://git-scm.com/docs/git-merge

Next, we shall find out what happens if merging does *not* go so smoothly.

22.3.4.5 `git reset`

First, purely for demonstration purposes, we use `git reset`[37] to undo the merge commit we just made. This can be a dangerous command, because we can erase commits with it, so we have to be careful. It's always useful to do a `git status` immediately before `git reset`, just to make sure the repository is in the state we think it is. `git reset --hard HEAD~<N>` sets the current branch back by `<N>` commits, discarding the rest of the commits in the process if they are not part of another branch. *They can still be recovered using* `git reflog`[38]*, but that's a bit too complicated to show here. Generally, it's hard to really lose things you have previously committed, so if you accidentally deleted some important history, don't despair immediately. :-)*

In contrast, the `--soft` flag just moves the `HEAD` pointer to another commit, but leaves our working directory and index unchanged. This can be useful e.g. for undoing commits[39].

Anyway, let's reset our `master` branch back one commit now:

```
$ git reset --hard HEAD~1
HEAD is now at 68d2674 Add background switching
```

You can consult the tree view again to see that the merge commit has disappeared, and `master` is back at "Add background switching". Now, let's try to make a merge fail.

22.3.4.6 Merge conflicts

Git is pretty smart when merging branches together, but sometimes (typically when the same line of code was edited differently in both branches) it does not know how to merge properly, which will result in a **merge conflict**. It's up to us to resolve a merge conflict manually.

Now, let's create a commit which will create a conflict. We just add a second output line after the "Hello world" statement. Since in the `celebration` branch, another statement was *also* added right after "Hello world", Git will not know how to correctly resolve this. Our `cout` statement looks like this:

```
$ git diff
diff --git a/src/ofApp.cpp b/src/ofApp.cpp
index f7e3dbc..6e232ba 100644
--- a/src/ofApp.cpp
```

[37] http://git-scm.com/docs/git-reset
[38] http://gitready.com/advanced/2009/01/17/restoring-lost-commits.html
[39] http://stackoverflow.com/a/927386/599884

```
+++ b/src/ofApp.cpp
@@ -4,6 +4,7 @@
 void ofApp::setup(){

     cout << "Hello world!";
+    cout << "\nThis is not going to end well!";
 }

 //--------------------------------------------------------------
```

Now we will create a commit (we are still on `master`):

```
$ git commit -am "Trigger a conflict"
[master 2608b52] Trigger a conflict
 1 file changed, 1 insertion(+)
```

When we attempt to merge `celebration` into `master`, bad things happen:

```
$ git merge celebration
Auto-merging src/ofApp.cpp
CONFLICT (content): Merge conflict in src/ofApp.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

When a conflict is detected by Git, it will stop the merging process and put "conflict markers"[40] into the conflicted files. Those markers look like this:

```
$ head -n 14 src/ofApp.cpp
#include "ofApp.h"

//--------------------------------------------------------------
void ofApp::setup(){

    cout << "Hello world!";
<<<<<<< HEAD
    cout << "\nThis is not going to end well!";
=======
    cout << "\nYeah, it works!";
>>>>>>> celebration
}

//--------------------------------------------------------------
```

The part between `<<<` and `===` shows the file as it is in `HEAD`, the current branch we want to merge *into*. The part between `===` and `>>>` shows the file as it is in the named branch, in our case `celebration`. What we have to do now next is resolve the conflict by implementing the conflicted section in a way which makes sense for our program,

[40] http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging#Basic-Merge-Conflicts

remove the conflict markers and save the file. For example, we can make the conflicted section look like this:

```
void ofApp::setup(){

    cout << "Conflict␣averted!";
    cout << "\nHello␣world!";
    cout << "\nYeah,␣it␣works!";

}
```

After doing this, Git still knows that there has been a conflict, and `git status` again tells us what to do next:

```
$ git status
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#    both modified:      src/ofApp.cpp
#
no changes added to commit (use "git␣add" and/or "git␣commit␣-a")
```

Obediently, we run `git add src/ofApp.cpp` to stage our conflict-free file and mark the conflict as resolved. Now, we can finish the merge. If we omit the `-m <message>` part, `git commit` will open an editor (which one depends on your setup) with a proposed commit message which mentions the files for which conflicts had occured. You can either try that way, or just create a self-made commit message directly, as usual:

```
$ git commit -m "Merge␣after␣resolving␣conflict"
[master 29d152e] Merge after resolving conflict
```

All that remains is to check if everything worked alright, and take a last look at our tree view:

```
$ git status
# On branch master
nothing to commit, working directory clean

$ git log --all --graph --decorate --oneline
*   29d152e (HEAD, master) Merge after resolving conflict
|\
| * d9be50b (celebration) Let us celebrate!
* | 2608b52 Trigger a conflict
* | e822ea4 Add background switching
|/
```

```
* 1964a43 Add welcome message
* f6caa7b Add initial set of files.
```

Congratulations, you have just resolved your first merge conflict! This concludes the walk-through portion of this chapter, I will continue with more high-level explanations of Git features.

22.3.4.7 `git tag`

During your work, you will encounter special commits, which mark significant stations in your project's history, for example released versions, or the state of your code when it was installed somewhere, or handed off to your customer. You can do this easily with Git tags[41]. Use `git tag somename` to put a tag on the current commit. This tag now permanently points to that commit, and you can (mostly) use it in Git commands just like commit hashes and branch names. For example, `git checkout v1.2` will check out the repository's state (if the tag exists) just like it was when you published version 1.2.

22.3.5 Remote repositories and Github

An important aspect of your work may involve collaboration with others. With Git, this typically involves one or more remote repositories (short **remotes**), to which you **push** your modifications, and from which you **fetch** the modifications of others.

One of several popular hosting platforms for Git repositories is Github[42]. Github offers repository hosting (public and private), project wiki pages, an issue tracker, social features, project web pages, etc. OpenFrameworks primarily uses Github to host its source code repositories[43] and the openFrameworks issue tracker[44].

Delving deeper into Github's features would lead too far here, so I'll just outline the typical operations you will deal with when interacting with Github repositories.

22.3.5.1 Setting up and remotes

To start a project on Github, you have several options:

[41] http://git-scm.com/docs/git-tag
[42] www.github.com
[43] https://github.com/openframeworks
[44] https://github.com/openframeworks/openFrameworks/issues

- If you want to have your own copy of the source code of a project that already lives online, you **fork**[45] that repository, including all history, ending up with a copy of it under your account.
- If you want to start a fresh project, you can create a new repository[46]. Github will display instructions for creating an empty local repository, or for connecting the new repository to an existing local one.

If there's already a Git repository online somewhere, you can also clone that repository[47] to get a copy of it on your local machine. This command is not limited to Github repositories, but can be used with all Git repositories, see the `git clone` docs[48] for what you can do with `git clone`.

The remote repositories are added as so-called `remotes` to your local repository's configuration (either automatically, or using `git remote add`). Think about it as a target identifier you supply to Git commands if you want to work with remote repositories. A remote is just an identifier that points to the Github (or other) URL where that repository lives.

It is customary that a "parent" repository (i.e. the repository under your Github account) is called `origin`, and a repository you forked from is called `upstream`. You can get the list of current remotes using `git remote`[49] (add `-v` to see more info).

22.3.5.2 Fetching and pulling

Now that you have a remote repository configured, you can interact with it via `git push` and `git fetch`. As the names imply, `git fetch`[50] fetches branches *from* a remote, so to get the latest version of the `master` branch of your Github repo, you'd do `git fetch origin master` (the syntax is `git fetch <remote> <branchName>`). If you wanted to obtain the newest modifications from your upstream remote, instead, you'd do `git fetch upstream master`. After this has finished, you'll have an additional branch called `origin/master` in your repository. You can check this with `git branch -a` - remote branches are listed with a `remotes/` prefix.

To integrate the newest changes of this remote branch into your local `master`, you first do `git checkout master`, to be sure you're on the correct branch. Then, you should make sure that the state of the repository is in order, using `git status`. Next, you merge the remote branch, just like any other branch, using `git merge origin/master`. If all went well, you now have all the latest changes integrated into

[45] https://help.github.com/articles/fork-a-repo
[46] https://help.github.com/articles/create-a-repo
[47] https://help.github.com/articles/fork-a-repo#step-2-clone-your-fork
[48] http://git-scm.com/docs/git-clone
[49] http://git-scm.com/docs/git-remote
[50] http://git-scm.com/docs/git-fetch

your `master` branch. If not, you'll probably have to fix some conflicts, as you already learned above.

A commonly used shortcut for the subsequent operations `git fetch` and `git merge` is `git pull`[51] - you can use that instead if you like. Personally, I tend to use `fetch` and `merge` separately, as it gives you a bit more control over what happens.

22.3.5.3 Pushing

When you have commits or branches you want to share with others, you will push them onto your remote repository (e.g. on Github) using `git push`[52], for example with `git push origin awesome-feature`. If the branch does not exist yet, it gets created in the remote repository, else it gets updated. Others can then fetch the new branch from your remote repository to integrate into their repositories. Note that Git tags are only pushed to a remote if you supply the `--tags` flag.

22.3.5.4 Pull requests

A central feature of the Gitub collaboration model are **pull requests**[53]. Pull requests (or "PRs" for short) are ways to get your personal changes integrated into a repository you forked (it's important that you forked the repository into your own account instead of getting a copy by other means).

Let's walk through this with an example: Say you found a bug in openFrameworks[54], and want to fix it. You have already forked openFrameworks to your account, created a local copy, and created a branch from `master`, called `fix-uglybug`, following our contribution guidelines[55] (ideally there's a bug report first where we discuss the proper way to fix the bug, but let's leave that part aside for now). To get the openFrameworks developers to review (and hopefully integrate) your bugfix, you push your branch to your remote (`git push origin fix-uglybug`), then switch to the freshly uploaded branch in the Github web interface, and click the green "compare, review, create a pull request" button. Following the instructions, you will be able to compare your branch to the target branch (typically openFrameworks' `master` branch), review the changes you made, type up a description of the pull request, and send it.

A pull request enables an easy review of your changes and offers a discussion platform where the changes can be discussed, updated, and finally merged with one click, incorporating your changes into the upstream repository.

[51] http://git-scm.com/docs/git-pull
[52] http://git-scm.com/docs/git-push
[53] https://help.github.com/articles/using-pull-requests
[54] https://github.com/openframeworks/openFrameworks
[55] https://github.com/openframeworks/openFrameworks/blob/master/CONTRIBUTING.md

22.4 Popular GUI clients

While working with the console commands offers the whole power of Git, it is sometimes more convenient to do at least part of the version control work in an application with a GUI. There are a couple of GUI applications available (depending on platform), and which one you use is often a matter of taste (and functionality of the individual programs), so I will just enumerate a couple of popular candidates. There's also a curated list of applications here[56], and a pretty exhaustive list here[57].

- git-gui[58] and gitk[59] are simple interfaces distributed with Git.
- Github offers clients for Windows[60] and Mac[61] which also offer integration with Github (not only Git).
- Gitg[62] is an open-source Git GUI for Linux.
- Tower[63] is a Git GUI for Mac.
- GitExtensions[64] is an open-source Git GUI for Windows.
- smartGit[65] runs on Windows, Mac, Linux.

You should try a few of the options and use what you like best. Personally, I use a combination of console commands and Gitg on Linux. I use the GUI mainly for branch navigation, selecting and staging modifications, and committing, and the command line interface for working with remotes and more complicated operations.

22.5 Conclusion

22.5.1 Tips & tricks

This section contains a loose collection of tips and tricks around avoiding common pitfalls and working with Git:

- Collaborating with users using different operating systems can be tricky because MacOS/Linux and Windows use different characters to indicate a new line in files (`\n` and `\r\n`, respectively). You can configure Git according to existing guidelines[66] to avoid most problems.

[56] http://git-scm.com/downloads/guis
[57] https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools#Graphical_Interfaces
[58] http://git-scm.com/docs/git-gui
[59] http://git-scm.com/docs/gitk
[60] http://windows.github.com/
[61] http://mac.github.com/
[62] https://wiki.gnome.org/Apps/Gitg
[63] http://www.git-tower.com/
[64] http://code.google.com/p/gitextensions/
[65] http://www.syntevo.com/smartgithg/
[66] https://help.github.com/articles/dealing-with-line-endings

- Some editors automatically remove trailing whitespace in files when saving. This can lead to commits containing unintentional modifications, which can make browsing a file's change history more confusing. Most relevant Git commands (e.g. `git diff`) accept the `-w` flag to ignore whitespace changes.
- When you realize that you want to add some more changes to your last commit, you can use the `--amend` flag when committing to add your staged changes to the last commit and adjust the commit message. This rewrites that commit, so only do that if you haven't pushed your commit yet!
- When you want to stage only part of the modifications in a file, you can use `git add -p <file>`. This switches to an interactive view where you can decide whether or not to add each change chunk.
- Git can be told to colorize the terminal output[67], which is pretty helpful.
- Use `git rm`[68] and `git mv`[69] when removing or moving files, respectively. If you don't, the index does not get properly updated. You can run `git add -u` to update the index manually.

22.5.2 Further reading

Now we are at the end of this quick introduction to Git, and while I have covered the most important things you need to know to get you up and running, I have only touched the surface of what Git can do.

Probably the most important thing left now is to point out where you can learn more about Git, and where you can turn to when things don't work out as expected:

- Learning resources:
  - GitRef[70] is an awesome short primer on Git fundamentals.
  - The Git home page[71] is probably the most unified but comprehensive online resource. Among others, it hosts:
  - The free ProGit book[72], readable online. Awesome to get in-depth information about all things Git.
  - The Git reference[73], which has the documentation about all Git commands, their options and usage.
  - Try Git[74] is an excellent interactive tutorial.
  - Github offers a Hello World[75] introduction to Git and Github.

[67] http://git-scm.com/book/en/Customizing-Git-Git-Configuration#Colors-in-Git
[68] http://git-scm.com/docs/git-rm
[69] http://git-scm.com/docs/git-mv
[70] http://gitref.org/
[71] {[}git-scm.com
[72] http://git-scm.com/book
[73] http://git-scm.com/docs
[74] http://try.github.io/levels/1/challenges/1
[75] https://guides.github.com/activities/hello-world/

  - There are some websites available which visualize/animate the workings of Git, see here[76], here[77] or here[78].
  - Gitignore patterns for a lot of different situations can be found e.g. here[79] or at gitignore.io[80].

- Get help:

  - Google[81] the errors you get!
  - Stack Overflow[82] is an awesome resources to find answers to problems you encounter (probably someone had the same problem before), and to ask questions yourself! There's even a separate tag[83] for Git.
  - If you're not successful with Stackoverflow, the openFrameworks forum has a separate category called "revision control"[84] for questions around this topic.

Finally, I hope that this chapter made you realize how useful it can be to integrate version control into your creative coding workflow, and that you will one day soon look fondly back on the days of zip files called `Awesome_project_really_final.zip`.

[76] http://www.wei-wang.com/ExplainGitWithD3/
[77] http://pcottle.github.io/learnGitBranching/
[78] http://ndpsoftware.com/git-cheatsheet.html
[79] http://github.com/github/gitignore
[80] www.gitignore.io
[81] https://www.google.com/
[82] http://stackoverflow.com/
[83] http://stackoverflow.com/questions/tagged/git
[84] http://forum.openframeworks.cc/category/revision-control

23 ofSketch

By Brannon Dorsey

Edited by Michael Hadley[1].

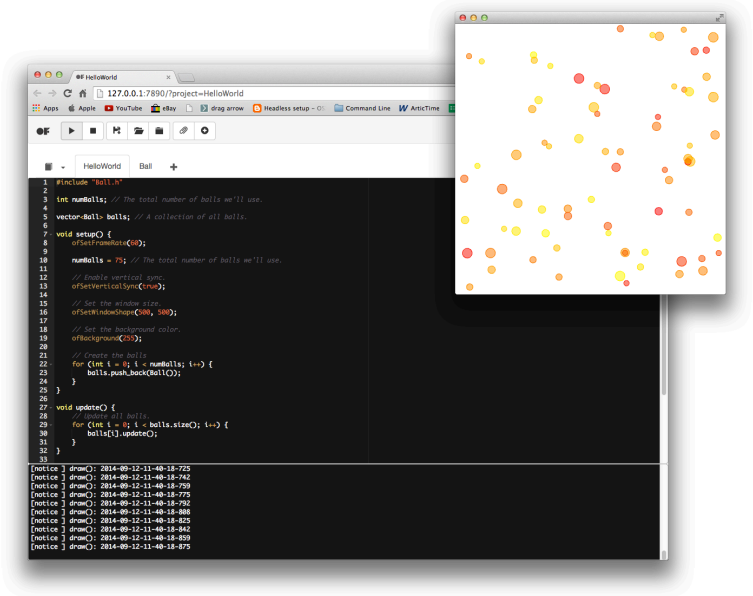


Figure 23.1: Hello World Example

23.1 What is ofSketch?

ofSketch[2] is a barebones development environment created specifically for building and running openFrameworks sketches. Its a minimal installation openFrameworks application that allows you to spend more time coding and less time with configuration.

One of the main goals in developing ofSketch is to decrease the barriers to entry for openFrameworks. For this reason, it should be noted that ofSketch is primarily geared towards beginners and new coders.

[1] http://www.mikewesthad.com/

[2] https://github.com/olab-io/ofSketch

23.1.1 What is ofSketch Good For?

- Teaching or learning openFrameworks
- Making the leap[3] from Processing[4] to openFrameworks
- Rapid code prototyping
- Creating code snippets, experiments, and examples
- Running openFrameworks on the Raspberry Pi[5]

23.1.2 What is ofSketch NOT Good For?

- Replacing a professional IDE like Xcode, Code::Blocks, Visual Studio, etc...
- Building sizable projects or applications

23.1.3 How does ofSketch work?

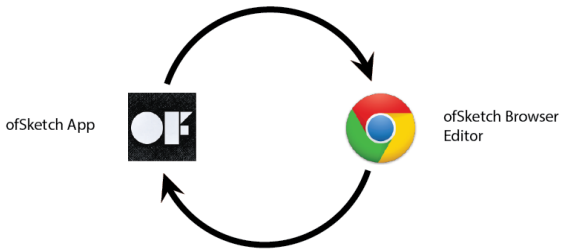


Figure 23.2: App to Browser Diagram

ofSketch works by internally communicating back and forth between the ofSketch application and the browser editor. The application, among other things, reads/writes files to your computer, and compiles and launches your projects. The browser editor acts as the graphical user interface (GUI), where you code, run, and manage your projects. When you interact with ofSketch you are doing so from the browser, however the distinction between the application and the browser should not be overlooked. For more info, check out the ARCHITECTURE.md[6] file on the ofSketch GitHub repository.

[3] http://openframeworks.cc/tutorials/first%20steps/002_openFrameworks_for_processing_users.html
[4] http://processing.org
[5] http://www.raspberrypi.org/
[6] https://github.com/olab-io/ofSketch/blob/master/ARCHITECTURE.md

23.2 Download

Visit the ofSketch releases page[7] to download the ofSketch app for your platform.

ofSketch comes packaged with the following files:

- CHANGELOG.md[8]
- CONTRIBUTING.md[9]
- data/
- LICENSE.md[10]
- ofSketch app
- README.md[11]

It is important for the ofSketch app and the "data" folder to stay in the same directory. I recommend that you leave the app in the uncompressed folder that you download.

Double-click ofSketch to launch the editor. If you are on a Mac, you may need to right click the app, and then press "Open".

That's it! Go code.

23.2.1 Getting Started With ofSketch

Code in ofSketch looks a bit different than what you may be used to with openFrameworks. If you are new to openFrameworks, great! We think that ofSketch code is easier to learn than the normal `.h` and `.cpp` openFrameworks code structure.

23.2.2 ofSketch Style Code

ofSketch uses "header style[12]" C++, where code implementation is written along with declarations inside of the header file, instead of the matching `.cpp` source file. If this doesn't make since to you, don't worry. Essentially, this allows us to write simple, easy-to-read code, that is great for beginners!

23.2.3 Project File

Every ofSketch project starts with an empty project file that looks like this:

[7] https://github.com/olab-io/ofSketch/releases
[8] https://github.com/olab-io/ofSketch/blob/master/CHANGELOG.md
[9] https://github.com/olab-io/ofSketch/blob/master/CONTRIBUTING.md
[10] https://github.com/olab-io/ofSketch/blob/master/LICENSE.md
[11] https://github.com/olab-io/ofSketch/blob/master/README.md
[12] http://hanxue-it.blogspot.com/2014/04/why-include-cc-implementation-code-in.html

```
void setup() {
        // put your setup code here, to run once:

}

void draw() {
        // put your main code here, to run once each frame:

}
```

If you are coming from Processing, this should be extremely familiar. To add global functions and variables, simply add them to this project file. You don't need to prefix any identifiers (variables, functions, etc...) with `ofApp`.

```
// global variables go up here
std::string text;

void setup() {
        // put your setup code here, to run once:
        text = "Hello World!";
        printHelloWorld(); // function call
}

void draw() {
        // put your main code here, to run once each frame:

}

// global functions go down here
void printHelloWorld() {
        cout << text << endl;
}
```

23.2.4 Classes

Using classes in ofSketch is easy! Press the "+" button in the tab bar in the ofSketch browser window to add a new class. When you do this a class template is automatically generated for you. Here is an example class template for a "Particle" class:

```
class Particle{

public:

    Particle(){

    }
```

```
};
```

This is essentially a regular `.h` file. The default constructor is explicitly defined in the generated template, but adding class members is easy. Just remember to both declare and implement all of the functions that you write in this file. Here is an example of a basic "Particle" class that could be used in a particle system.

```
class Particle{

public:

    ofVec2f location;
    ofVec2f velocity;
    ofVec2f acceleration;

    ofColor color;

    float maxSpeed;

    int radius;

    // default constructor
    Vehicle() {};
    // overloaded constructor
    Vehicle(float x, float y) {

        acceleration = ofVec2f(0,0);
        velocity = ofVec2f(0, -2);
        location = ofVec2f(x,y);
        color = ofColor(ofRandom(255), 0, 255);
        radius = 6.0;

        maxSpeed = 4;
    }

    void update() {

        velocity += acceleration;
        velocity.limit(maxSpeed);
        location += velocity;
        acceleration *= 0;
    }

    void draw() {

        ofFill();
        ofSetColor(color);
        ofCircle(location.x, location.y, radius);
    }
```

```
// etc...

};
```

23.2.5 Includes

Every ofSketch file includes "ofMain.h" by default. To include custom classes, simply put `#include "ClassName.h"` at the top of any file that needs to use that class. Below is an example of how to include the Particle class file above in the project file.

```
#include "Particle.h"

Particle p;

void setup() {
     // put your setup code here, to run once:
     // create a particle at the center
     p = Particle(ofGetWidth()/2, ofGetHeight()/2);
}

void update() {
    p.update();

}

void draw() {
        // put your main code here, to run once each frame:
    p.draw();
}
```

Here we include the "Particle.h" file, use it to instantiate a Particle object "p", and then place it in the middle of the screen.

Note that we also added an `update` function. As you may know by now, it is customary in openFrameworks to separate operations that update logic from operations that render graphics to the screen. This is for performance reasons, however, it is not necessary, and all of the code placed in `void update()` can instead live inside of `void draw()` if you prefer.

23.2.6 Examples

ofSketch comes packaged with a few basic examples that illustrate the ofSketch code style. Most of them are ported from the regular openFrameworks examples, but there

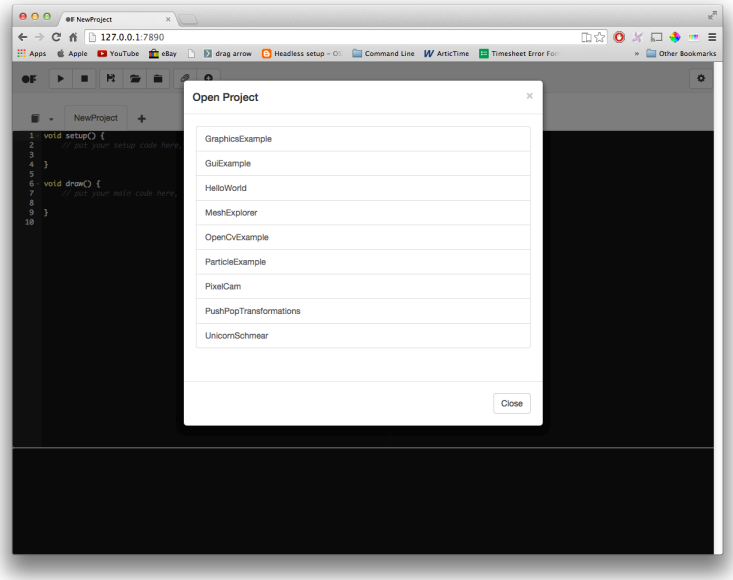


Figure 23.3: Open Project

are a few new ones too. Press the “Open Project” button inside ofSketch to open one of them.

While the code in this chapter highlights the difference between ofSketch code and a normal implementation of C++ code, reviewing the examples should give you a better idea of the general ofSketch style.

23.3 Sketch Format

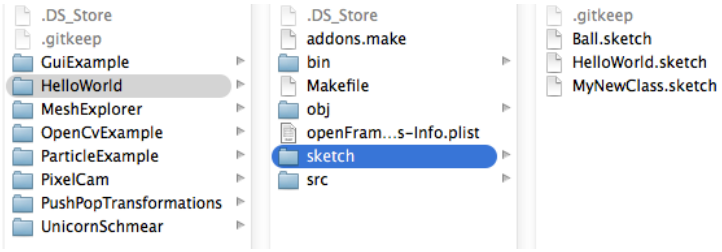


Figure 23.4: Sketch Folder Structure

If you take a look at an app in your Projects folder, you will see a “sketch” directory. This is where all of the ofSketch source files are saved, each with a “.sketch” file extension. When you use ofSketch, you are editing the files in this directory. Whenever you run

an app from inside ofSketch, all of the files in the “sketch” directory are processed to generate source files inside of the “src” directory.

Because of this workflow, it is important to edit ofSketch projects through **ofSketch only**. You could easily get yourself into trouble if you edited an ofSketch project with Xcode (modifying the files in the “src” directory) and then opened it in ofSketch again, pressed the play button, and then overwrote all of our changes.

Soon, ofSketch will include fancy project import and export feature which will allow you to import/export a project from/to a professional IDE target of your choice. Until then, it is best to just copy a project if you want to edit it with something other than ofSketch.

23.4 Remote Coding

One of the highlights of the ofSketch browser editor is the ability to edit code on a remote machine through a network connection. This is especially helpful when coding with the Raspberry Pi, or when tweaking live installation code. The figure below illustrates this in practice.

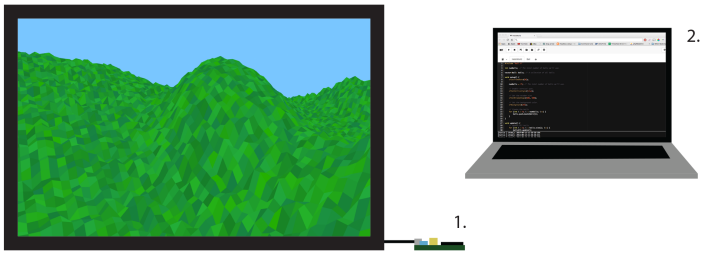


Figure 23.5: Remote Coding Diagram

1. Raspberry Pi running an ofSketch app that draws a landscape to a connected to a TV
2. Laptop with web browser pointed to RPi’s IP address and port 7890 (e.x. http://192.168.0.204:7890)

To code with ofSketch remotely, start the ofSketch application on the machine that you want to run the openFrameworks app. In order to connect to ofSketch from that machine, you need to know its unique IP address on the network. You can use an application like Bonjour Browser[13] on device 2 to discover device 1’s IP address.

[13] https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=bonjour+browser

Once you have the IP address of device 1, open a web browser on device 2 and visit `http://CPU_1_IP_ADDRESS:7890`.

You can now create, edit, and run projects using device 2.

23.5 Future

The ofSketch project is still in its infancy. In fact, at the time of writing this chapter, the most recent ofSketch release is v0.3.2. Undoubtedly the application will change as we approach a stable release (v1.0), but we are particularly excited about some of the things we have in store. Below is a list of planned features to watch out for over the next year.

23.5.1 App Export

App Export will allow users to export executables and resources that can be transferred to and run on other computers. The exported project will be downloaded as zipped bundle for easy transport.

23.5.2 Project File Export

Project File Export will use an integrated version of the openFrameworks Project Generator in the ofSketch app to export a project for Xcode, Code::Blocks, and Visual Studio 2012. These project files will be useful to allow new users to access more advanced editing features available in professional IDEs.

23.5.3 Custom .h & .cpp Files

To aid in the transition from ofSketch to more advanced IDEs, ofSketch users will be given the option to create and work with `.h` and `.cpp` files. Eventually, using this functionality will be as simple as including the appropriate extension when creating the file.

23.5.4 Clang Indexed Autocomplete

We intend to use some of these[14] Clang resources to index the openFrameworks Core and use it to better provide autocomplete and syntax highlighting tools in the editor. Ideally, this system would also auto-index user code.

[14] https://github.com/brannondorsey/ofSketch/wiki/Clang-Resources

24 Installation up 4evr - OSX

The original version of the article is here[1].

This article is intended as a starter guide to help keep a software based installation up for as long as possible in a public facing setting. This guide applies primarily to software running on Mac OSX 10.8+. The software is likely running as a fullscreen, single display app. Although I hope many different people find parts of the article useful in terms of prepping for the challenge of long term installs.

At work I recently had to set up a four installations of different configurations that would need to run all day, every day, 24 hours a day for a couple months with as few crashes or glitches as possible and without anyone going to check on them. This is something that a lot of media artists need to do all the time, and there are a bunch of different tricks and tips to keeping things up for an extended period, I figured I'd share my findings. There are alternate ways to do many of these tasks and this is only one road so please share some tips you've picked up out in the field down in the comments box below.

I had to do several searches in a couple different places to find all the information I needed to keep everything consistently up and bug free. Luckily most of the installations I was dealing with this time were fairly light in terms of resources and complications, but it's always best practices to have a safety net.

I usually run these off brand new, unboxed computers so this is sort of starting from scratch. Most of the things I point out are set to the opposite by default.

Tip: if you're doing multiple computers, do these prep steps on one of them and just boot the others into target disk mode and use something like Carbon Copy Cloner[2] to mirror the first one on the next so everything is as consistent as possible.

24.1 Step 1: Prep your software and the computer

When building your software or whatever it might be, always keep the long running installation in mind. Plan which things will need to be adjusted by whoever is watching over the installation from the beginning (or at least don't save it for the end). In my experience, keep it as simple as possible, so that it's easy for the caretaker to get in

[1] http://blairneal.com/blog/installation-up-4evr/
[2] http://www.bombich.com/

there to fix or adjust what they need without opening Xcode and compiling or even exiting out of your app. Time you spend now to make things simple will save you hours of remote debugging when something breaks.

You'll need to go through and turn off or disable several different automatic settings to keep things from popping up over top of your application. This can differ depending on whether you're running 10.6, 10.7, 10.8, 10.9 etc etc.

In System Preferences:

- **Desktop and Screensaver:** Disable your screensaver. Set it's time to "Never." I also suggest changing your desktop background to either black/a screenshot of your app/you client's logo - you can even set these to change automatically - remember - **it's not broken until someone notices** :)
- **Energy Saver:** Turn Display Sleep and Computer Sleep to Never. Enable "Start up automatically after power failure" and "Restart automatically if the computer freezes" (these are only available in 10.7 and later)
- **Users and Groups:** ->Login Options: Enable Automatic Login
- **Software update:** Disable automatic updates.
- **Notifications:** Disable any potential Notification Center alerts (banners or pop-ups) from specific apps
- **Sharing:** If you are running your main computer without a monitor or in an inaccessible area, don't forget to turn on File sharing and Screen sharing. This will allow you to access the computer and control it if you're on the same network (optional if you're concerned about security).
- **Network:** If you don't need remote access or don't need Internet access for the installation, it's not a bad idea to disable the Wifi so the "please select a wireless network" window doesn't pop up when you least expect it. You can also turn off the option to ask you to join a new network if the proper one isn't found.
- **Bluetooth** :If running without a mouse or keyboard plugged in, sometimes you can get the annoying "Bluetooth keyboard/mouse setup" pop up over your application. You can temporality disable these by going to the advanced settings within the Bluetooth Preferences. See below for it's location in 10.6.
- **Security:** I would make sure that "Disable Automatic Login" is unchecked so you don't hit any surprises on reboots. If you're really paranoid, you can even disable things like the IR remote receiver that still exists on some macs and definitely on Macbooks. This would keep pranksters with Apple TV remotes from "Front Rowing" your installation. To disable, go to Security->General->Advanced (in >10.8) and "Disable remote control IR receiver".

You can also disable the "This Application Unexpectedly Quit" and the subsequent bug report that comes with it by running this command in terminal OR renaming the Problem Reporter app:

```
sudo chmod 000 /System/Library/CoreServices/Problem\ Reporter.app
```

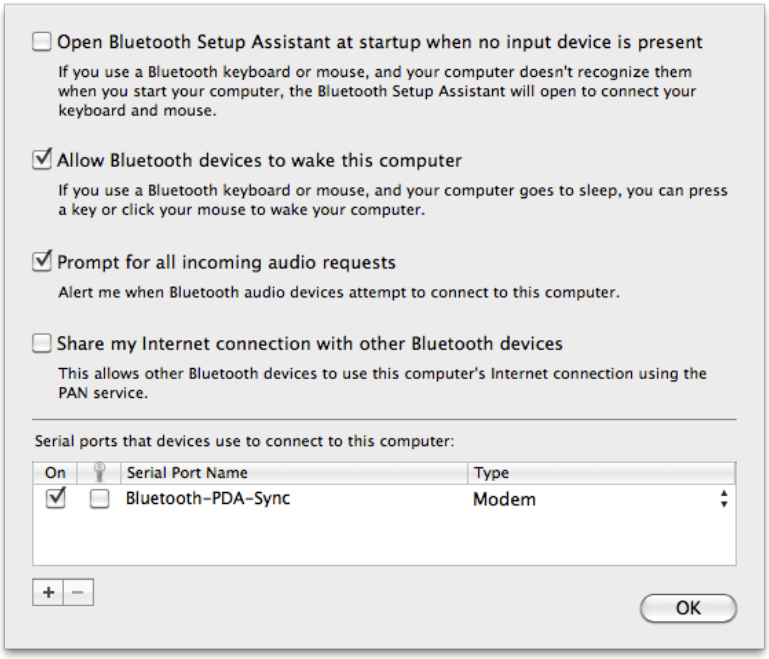


Figure 24.1: BluetoothSettings

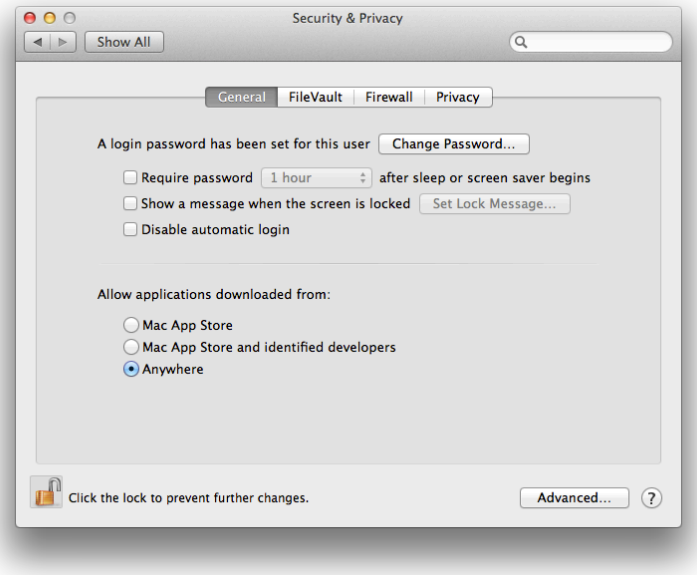


Figure 24.2: SecuritySettings

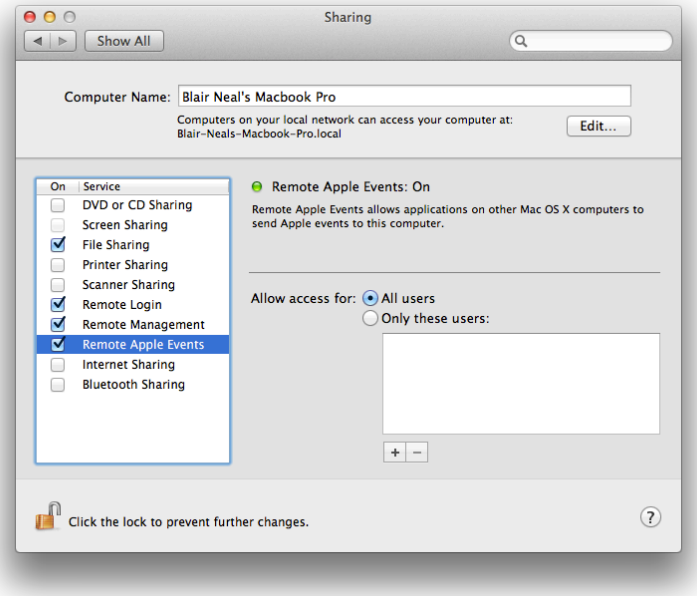


Figure 24.3: SharingSettings

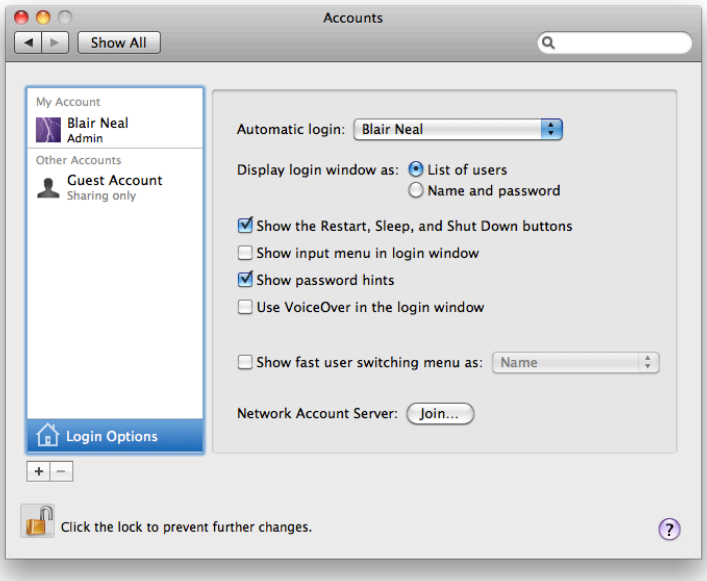


Figure 24.4: Login_items

Another useful tool for modifying certain OSX .plists for disable or enabling certain things is Tinkertool[3] You can use this to disable or enable certain things that System Preferences doesn't cover.

I would also look at this filepath and you can rename files in here to temporarily disable them on the computer you're using: /System/Library/CoreServices

You can rename "Notification Center" to "Notification Center_DEACTIVATE" or something (or you can move it) - and then you won't get any obnoxiously "helpful" Notification Center popups.

If necessary, You can also hide all of the desktop icons with this terminal command:

```
defaults write com.apple.finder CreateDesktop -bool false
```

24.2 Step 2: Boot into your software

Things get unplugged, power goes out, not everyone has budget or space for a battery backup etc etc. Above, I covered how to have everything reboot automatically after power failures or freezes, but you'll also need your app to be ready to go from boot and not leave the desktop open to prying eyes. There are many ways to have your application load automatically - the simplest is using OSX's built in tools: In the System Preferences "Accounts" panel, select "Login Items" and drag your application into there to have it open automatically on launch.

24.3 Step 3: Keep it up (champ!)

There are several ways to make sure your application goes up and stays up -

Launchd

Using Launch Daemons is an alternate way to get apps to load on boot and to continuously re-open them if they go down. Launchd plists are very useful alternatives to cron jobs and can be used to run things on a periodic basis or on calendar days. You could achieve similar results with a combination of automator and iCal, but it depends on what you're comfortable with.

Here is an Apple Doc[4] on using Launch Agents and Launch Daemons in various ways.

[3] http://www.bresink.com/osx/TinkerTool.html
[4] http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html

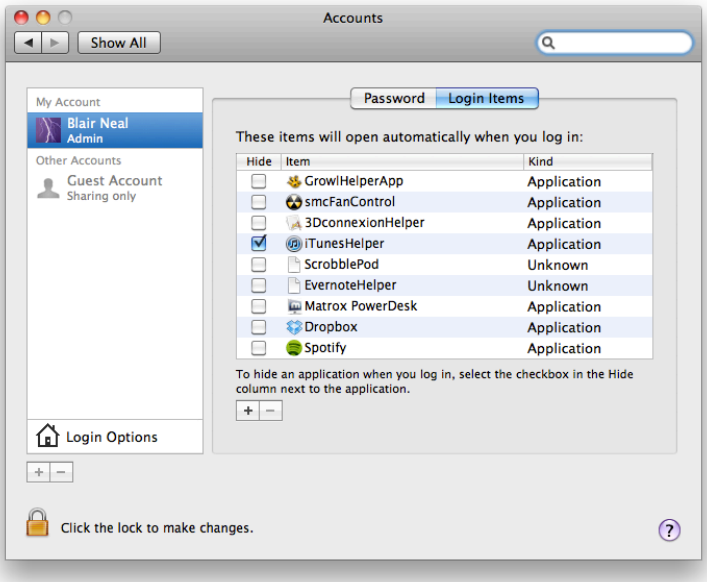


Figure 24.5: Login Items

The difference between a Launch Daemon and a Launch Agent[5] (Basically whether you need it to run when a user is logged in or not...for most simple options like launching a regular app, you'll just want a Launch Agent)

Also note (!) that you may need to point your launch daemon to a file within your .app package, not just the app itself - you have to point it to the file in the MacOS folder inside the .app package (right-click your app and select "Show package Contents") Otherwise you might be wondering why the launchdaemon isn't launching your app.

A launchd example from admsyn: https://gist.github.com/4140204

Of course you could make the launchd plist yourself for free from a template like above. You can read all about them with the command "man launchd.plist" typed into terminal to get an idea of what each toggle controls. One quick method to setting up Launchd is to use Lingon ($4.99 in the App Store) or Lingon X[6]

In Lingon, hit the + to create a new launchd plist. Just make it a standard launch agent. Now Set up your plist like so:

One additional/optional thing you can add to this is to put an additional key in the plist for a "Successful Exit". By adding this, your app won't re-open when it has detected that it closed normally (ie You just hit escape intentionally, it didn't crash). Can be useful if you're trying to check something and OS X won't stop re-opening the app on

[5] http://techjournal.318.com/general-technology/launchdaemons-vs-launchagents/
[6] http://www.peterborgapps.com/lingon/

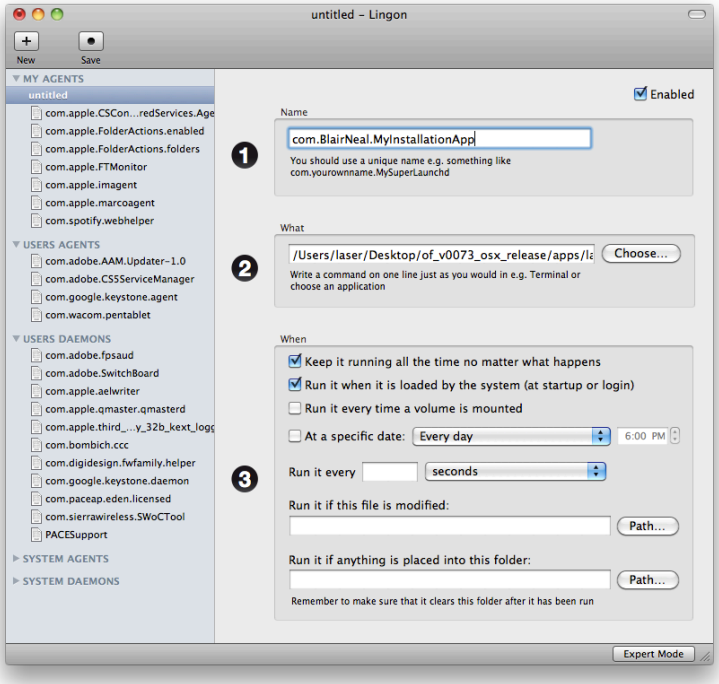


Figure 24.6: LingonSetup

you. To easily add this to the key, just hit “expert mode” on the bottom of the Lingon window after selecting your newly made script on the left. Then modify the relevant bits highlighted in the screenshot:

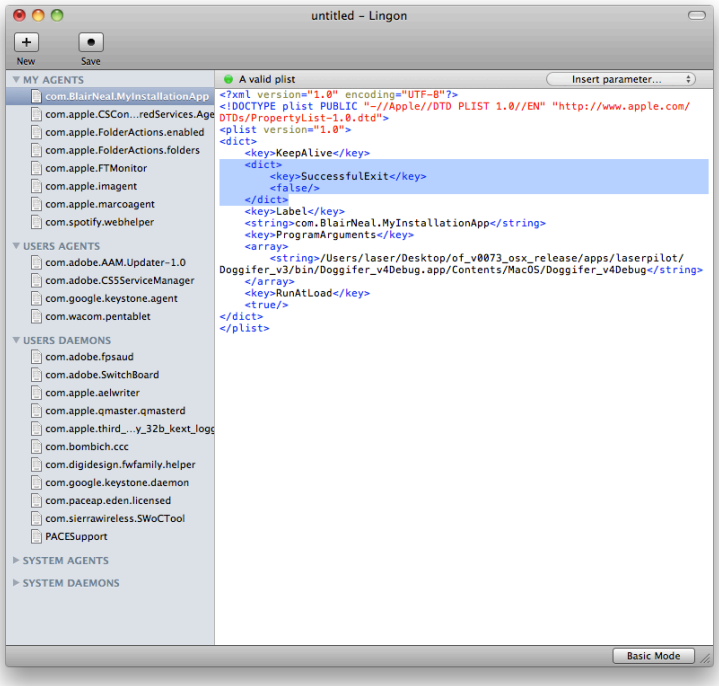


Figure 24.7: LingonplistSetup

Shell script+Cron Job method

(I got the following super helpful tip from Kyle McDonald[7])

This method is sort of deprecated in relation to the launchd method - you can run shell scripts with Lingon and launchd in the same manner as what we’ve got here. Shell scripting is your best friend. With the help of the script below and an application called CronniX (or use Lingon) , you will be able to use a cronjob to check the system’s list of currently running processes. If your app does not appear on the list, then the script will open it again, otherwise it won’t do anything. Either download the script or type the following into a text editor, replacing Twitter.app with your app’s name and filepath. Don’t forget the “.app” extension in the if statement!:

```
\#!/bin/sh if [ $(ps ax | grep -v grep | grep "Twitter.app" | wc -l)
    -eq 0 ] then
echo "Twitter␣not␣running.␣opening..."
open /Applications/Twitter.app else
```

[7] http://kylemcdonald.net/

```
echo "Twitter running" fi
```

Save that file as something like “KeepOpen.sh” and keep it next to your application or somewhere convenient.

After creating that file, you’ll need to make it executable. To do this, open the Terminal and in a new window type “chmod +x” and then enter the path to the shell script you just created (you can either drag the shell script into the terminal window or manually type it). It would look something like this:

```
Laser-MacBook-Pro:~ laser$ chmod +x /Users/laser/Desktop/KeepOpen.sh
```

After you have made it executable, you’re now ready to set it up as a cronjob. Tip: to test the script, you can change the extension at the end to KeepOpen.command as an alternative to opening it with Terminal, but the same thing gets done.

Cronjobs are just low level system tasks that are set to run on a timer. The syntax for cronjobs is outside of the scope of this walkthrough, but there are many sites available for that. Instead, the application CronniX can do a lot of the heavy lifting for you.

After downloading CronniX, open it up and create a new cronjob. In the window that opens, in the command window, point it to your KeepOpen.sh file and check all of the boxes in the simple tab for minute, hour, month, etc. This tells the job to run every minute, every hour, every day, every month. If you want it to run less frequently or at a different frequency, play around with the sliders.

Now just hit “New” and then make sure to hit “Save” to save it into the system’s crontab. Now if you just wait a minute then it should open your app every minute on the minute. Maybe save this one for the very end if you have more to do :)

This is a great tool if there is an unintended crash because the app will never be down longer than a minute.

Non-Cronjob - Shell Script Method

```
\#!/bin/bash

while true
do
#using open to get focus
echo "Trying to open empty example"
open -a emptyExample
sleep 10
done
```

Just type this into a plaintext document and save it as something like ”KeepMyAppAlive-Plz.command” and then use chmod as above to make the file executable and then drop this in your login items as above. This one will just continuously try and open your app

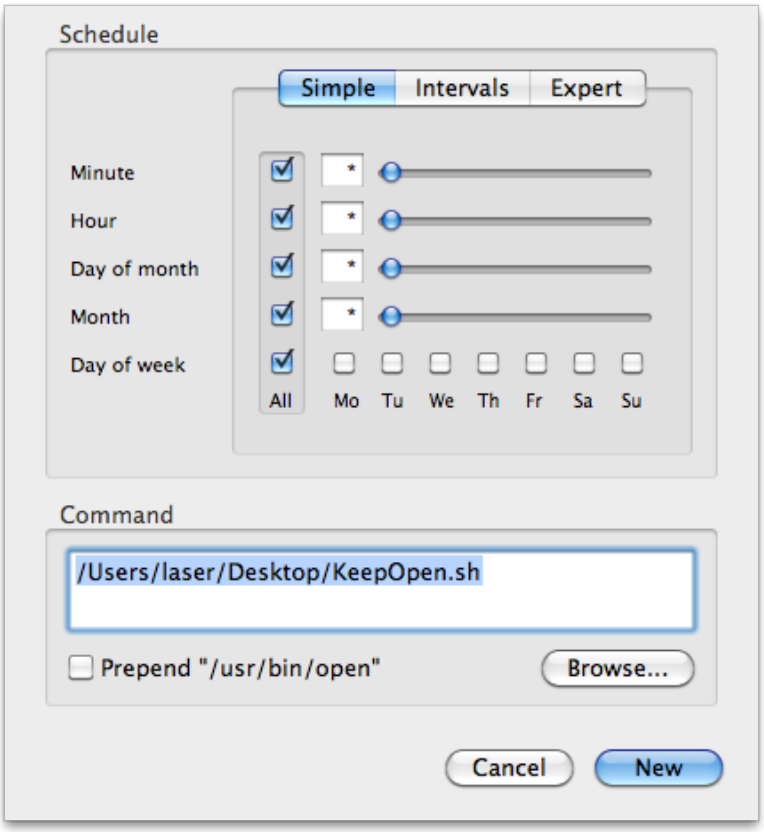


Figure 24.8: Cronnix_link

every 10ms, but if it is already open, the OS knows to not try opening it a second, third, fourth time.

Make sure to check the Console.app for any errors that may have come through when no one caught them, whenever you check the installation in person or remotely. This is not a fix-all for buggy programming, just a helper to keep things running smooth. The more things you can do to leave yourself notes about why the crash happened, the faster you can address the core issue.

Applescript is also a very solid choice for doing some more OS specific work in terms of having odd menus clicked or keypresses sent in some order.

24.4 Step 4: Reboot periodically

This one is a little more preventative, or maybe superstitious so hopefully someone can point out a concrete reason why this is a good idea. Depending on your app and the amount of stuff it reaches into, there could be some memory leaks or other OS bugs that you haven't accounted for. Rebooting every day or week is a good idea to keep everything tidy, system wise.

The simplest option by far would be to go to System Preferences->Energy Saver and then click "Schedule..." and enter in some values if you need to turn the computer off to rest for a longer period of time to save it some stress when it might not be used at night time or something. Heat can do funny things sometimes, so if you have a chance to get your computer to rest and the time to test it, definitely give this a shot...saves some energy too which is nice.

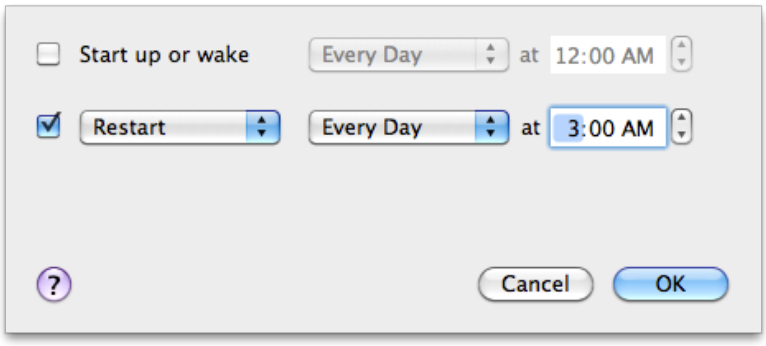


Figure 24.9: Auto-reboot

You could also set up another shell script with a crontab as above with CronniX that reboots the system with as often as you specify.

Another option (if you don't want to deal with the terminal and shell scripting) is to use iCal to call an Automator iCal event. This method is perhaps a little easier to schedule

and visualize when you will reboot. Within Automator, create a new file with the iCal event template to do something like this:

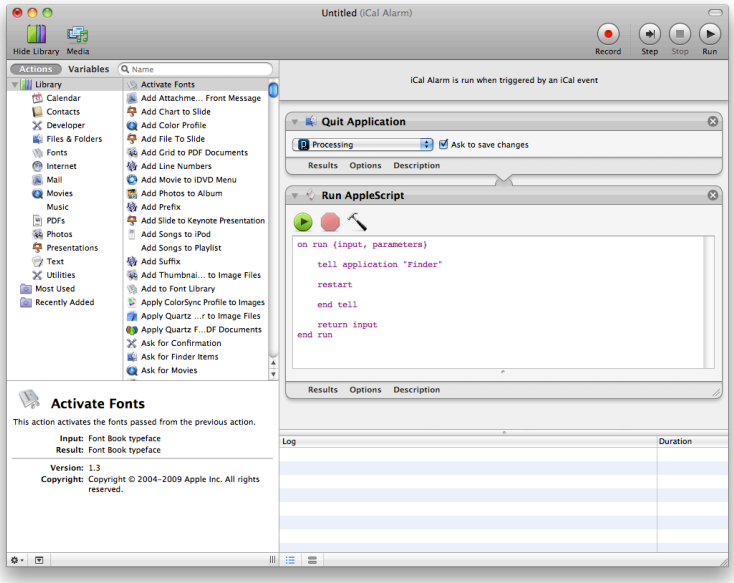


Figure 24.10: Automator Shell Script

Run it to see if it does what you expect and then save it out. When you save,it will open in iCal as an action that gets opened. Just set it to repeat as often as you'd like. You can also set things like having it email you when it reboots or runs the script.

If you'd like to just close your programs and re-open them and there is a background and foreground do something like this (pauses are so the quitting and re-opening stuff has time to actually execute):

24.5 Step 5: Check in on it from afar

There are a bunch of options here from various paid web services (like Logmein[8] or Teamviewer[9]), to VNC (many options for this: RealVNC[10] and Chicken of the VNC tend to come up a bunch) to SSHing[11]. The choice here depends on your comfort level and how much you need to do to perform maintenance from far away. Also - see below for tips on logging the status of your app as an alternative way

[8] http://www.logmein.com/
[9] http://teamviewer.com/
[10] http://realvnc.com/
[11] http://www.mactricksandtips.com/2009/06/ssh-into-your-mac.html

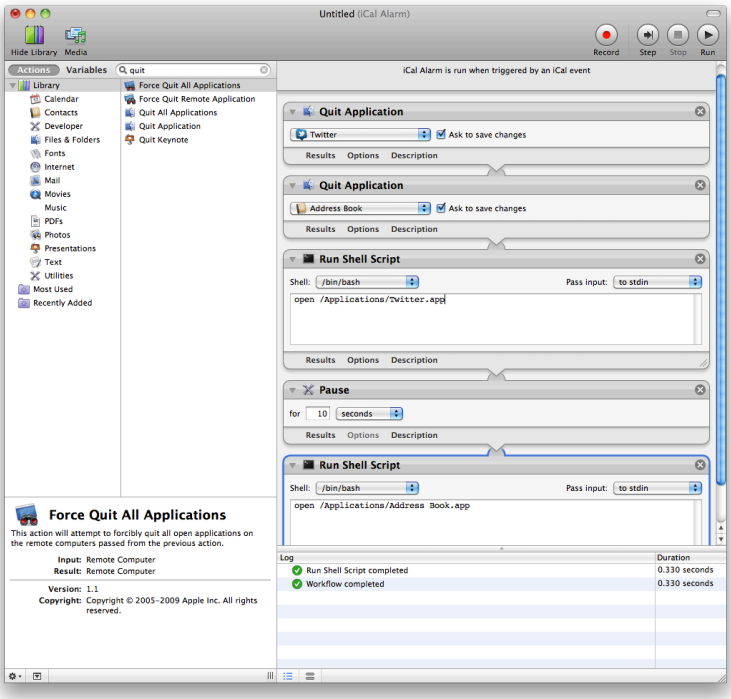


Figure 24.11: enter image description here

Leaving a Dropbox connected to the computer and your own is super useful for file swaps between the two computers. Although most remote screensharing services have file sharing built in, Dropbox is just a nice, fairly transparent option.

24.6 Step 6: Test, test, test.

You've already tested and perfected your software for the installation, so make sure to test all of the above methods and automatic scripts in as realistic manner as you can before you leave it alone for the first day at school.

You can't account for everything, so don't beat yourself up if something does eventually happen, but this list will hopefully alleviate a little bit of frustration. Good luck!

24.7 Additional Tips: Logging

If you have an installation that runs for weeks or months, you might want a way to keep tabs on it that doesn't involve remotely logging in and checking on it. A good thing to have would be to have something on the system that writes certain info to a text file (kept on a linked Dropbox), or better write that file to a web server that you can then check.

There are a couple things you can do depending on what you want to know about the state of your installation.

There is a terminal command you can use to get a list of all of the currently running processes on your computer:

```
ps aux (or ps ax)
```

(more info above ps commands here[12]) – Further more you can filter this list to only return applications you're interested in learning about:

```
ps aux | grep "TweetDeck"
```

This will return a line like this:

```
USER               PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED
        TIME COMMAND
laser            71564   0.4  1.7  4010724 140544   ??  S    Sun03PM
    14:23.76 /Applications/TweetDeck.app/Contents/MacOS/TweetDeck
    -psn_0_100544477
```

[12] https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/ps.1.html

```
laser          95882   0.0  0.0  2432768    600 s000  S+   12:11PM
   0:00.00 grep TweetDeck
```

Now you have the following useful info: CPU usage, Memory usage (as percentage of total memory), Status, Time Started, Time Up

All that is left is to write this output to a text file, which you can do with a line like this:

```
ps aux | grep 'TweetDeck' >>
   /Users/laser/Dropbox/InstallationLogs/BigImportantInstall/Number6ProcessLog.txt
```

This line basically says - tell me the open processes (px aux) - only give me the lines that have "Tweetdeck" in them (grep Tweetdeck) - and then append them to a text file located at this location (>> path_to_text_file)

Now we just need to make this an executable shell script and set it up as a launch daemon or cron job – see above at Step 3 to learn how to run the shell script at a regular interval using Lingon and launchd. If the app isn't running, it will only return the "grep YourAppName" process which is a good thing to log because if your app isn't open you won't know how long it's been out (nothing will be logged), but having the grep process logged will at least tell you it was checking for it. Grep will also more accurately tell you what time it checked – the other app will only give you a start time and up time.

Let's also take this one step further and say, hypothetically, that the Triplehead2Go display adapter you have is fairly wonky and you don't always get the displays or projectors to connect after reboot – or maybe a projector is shutting itself off and disrupting things. Well we can log the currently available resolutions too! Try entering the line below in your own terminal:

```
system_profiler SPDisplaysDataType
```

This will return a list of connected displays and some metadata about them including resolution and names.

Let's say you want to make sure you're running a resolution of 3840×720 at all times...or you want a log of resolution changes. You would do something like:

```
system_profiler SPDisplaysDataType | grep Resolution
```

This will return "Resolution: 3840×720" which you can combine with the above lines to write it all to a text file. So here would be your shell script file if you wanted to record the currently running processes and the current resolutions:

```
    \#!/bin/bash
ps aux | grep 'YourAppName' >>
   /Users/you/filepath/Install6ProcessLog.txt
system_profiler SPDisplaysDataType | grep Resolution >>
   /Users/you/Dropbox/Install6ProcessLog.txt
```

And now you're feeling excited, maybe you want to grab a fullscreen screenshot at a regular interval too, just to make sure there is no funkiness happening that you can't see...well you could add this line to the above as well:

```
screencapture ~/Desktop/$(date +%Y%m%d-%H%M%S).png
```

This will save a screenshot to the desktop (specify your own file path) with a formatted date and time. You may want to do this every hour instead of every 5 minutes since it's a big chunk of data and it may cause some issue with your screens. As usual – test before deploying!

Bonus points would be to create an auto-generated table and webpage that takes all of this info and puts it into a nice view that you can use to check all of your installations at a glance.

If the process logger isn't enough, we can use what we learned in that process to actually set up a system to email you if something is amiss so you don't have to manually check it. We can do this all with the command line and internal tools, it's just a more involved setup. This is going to be fairly general and will need some tuning in your specific case.

First you will need to configure postfix so you can easily send emails from the terminal – follow the instructions here as closely as possible: http://benjaminrojas.net/configuring-postfix-to-send-mail-from-mac-os-x-mountain-lion/

If you were using a gmail account you would do:

InstallationSupport@gmail.com

pass: yourpassword

The line in the passwd file mentioned in the article would be: smtp.gmail.com:587 installationSupport@gmail.com:yourpassword

Now send a test email to yourself by running: echo "Hello" | mail -s "test" "InstallationSupport@gmail.com"

Second step is to combine this new found ability to send emails from the Terminal with a process to check if your application is still running...something like the below would work with some tweaking for what you're looking to do:

```
\#!/bin/sh
if [ $(ps ax | grep -v grep | grep "YourApp.app" | wc -l) -eq 0 ] ;
   #Replace YourApp.app with your own app's name
then
        SUBJECT="Shit␣broke"
        EMAIL="InstallationSupport" #this is the receiver
     EMAILMESSAGE="This␣could␣be␣for␣adding␣an␣attachment/logfile"
     echo "The␣program␣isn't␣open␣-␣trying␣to␣re-open">$SUBJECT
     date | mail -s "$SUBJECT" "$EMAIL"  "$EMAILMESSAGE"
```

```
        echo "YourApp␣not␣running.␣Opening..."

    open /Applications/YourApp.app #reopen the app - set this to an
        exact filepath
else
    echo "YourApp␣is␣running"
fi
```

Now you just need to follow the instructions from Step 3 above to set this shell script up to run with launchd – you can check it every 5 minutes and have it email you if it crashed. You could also adapt the If statement to email you if the resolution isn't right or some other process condition.

24.8 Memory leak murderer

See this article[13] about combining the above process with something that kills and restarts an app if it crosses a memory usage threshold

Bonus – if using MadMapper – see this link[14] for an AppleScript that will open MadMapper and have it enter fullscreen – and enter "OK" on a pesky dialog box.

24.9 Alternate resources

This is an amazing addon for openFrameworks apps that keeps your application open even after a large range of failures: https://github.com/toolbits/ofxWatchdog

http://vormplus.be/blog/article/configuring-mac-os-x-for-interactive-installations

http://www.evsc.net/home/prep-windows-machine-for-fulltime-exhibition-setup

If you're looking for help with this task with Windows, check out this awesome script StayUp[15] from Stephen Schieberl. Also for Windows: http://www.softpedia.com/get/System/File-Management/Restart-on-Crash.shtml and this tool for scripting OS operations on windows http://www.nirsoft.net/utils/nircmd.html

Check out this great step by step from EVSC: http://www.evsc.net/home/prep-windows-machine-for-fulltime-exhibition-setup

[13] http://blairneal.com/blog/memory-leak-murderer/
[14] http://blairneal.com/blog/applescript-to-automatically-fullscreen-madmapper-for-installations/
[15] http://www.bantherewind.com/stayup

25 Installation up 4evr - Linux

by Arturo Castro[1]

In linux you can install a barebones desktop which makes things way easier to setup for installations, since among other things, you won't need to worry about deactivating annoying notifications. Also things like deactivating screen blanking can be done via scripts so everything can just be copy and pasted and it's easy to automate.

1. Install a linux distribution of your choice, for installations i usually use ubuntu since the drivers for the graphics cards come preinstalled. While installing, choose the option to login automatically to your user, that way the installation can be started later just by turning on the computer.
2. Update all the packages to the latest versions. In ubuntu you can use the Software Updater tool or via the command line do:

```
sudo apt-get update
sudo apt-get upgrade
```

3. Install the propietary drivers if you are using nvidia or ati cards. In latest ubuntu versions you can install it through the "Software & Updates" tool in the Additional Drivers tab
4. Ubuntu Unity, the default desktop is usually bloated with features that are not used in a computer running an installation, i've been recently using Openbox which also makes OpenGL slihtly faster since the desktop is not composited and even solves some problems with vsync:

```
sudo apt-get install openbox
```

5. You also probably want to install openssh server to be able to access the machine remotely:

```
sudo apt-get install openssh-server
```

6. Now download and install openFrameworks via the install_dependencies.sh script

[1] http://arturocastro.net

7. Logout the session and choose openbox instead of unity in the greeter screen
8. You'll get a grey screen with no decorations, bars... you can access a context menu pressing with the right button of the mouse anywhere in the desktop although i find it easier at this point to just log in through ssh from my laptop.
9. Once you've installed your application in the computer you probably want it to start when the machine boots. With Openbox you just need to create a script in ~/.config/openbox/autostart there add the path to the binary for your application:

```
~/openFrameworks/apps/myapps/myapp/bin/myapp &
```

```
don't forget the & at the end so the application is started in the
    background.
```

10. Disable the screen blank, in the same autostart file add this lines:

```
xset s off
xset -dpms
```

And that's it now the computer will start you app everytime it starts. Most PC's have a BIOS setting where you can choose to start it automatically after a power cut so if the installation is somewhere where they just cut the power at night it'll immediately start automatically in the morning when they turn it on again.

25.1 Some additional tricks:

- Linux can be installed in an SD card so you can save some money by buying a 16Gb SD Card instead of an HD, most SD Cards are also pretty fast so boot times will be really short. Just boot from a USB stick or CD and with the SD card in the reader, if there's no HD the ubuntu installer will just install in the sdcard. Installing to an SD card makes it also really easy to make copies of an installation. In ubuntu you can use the "Disks" tool to create and restore backups or from the command line with:

```
sudo dd bs=4M if=/dev/sdcardDevice of=sdcardimg.bin
```

```
To create the backup where sdcardDevice is the name of the device
    that you can find out by mounting the device and showing it's
    properties from nautilus. And then:
```

```
sudo dd bs=4M if=sdcardimg.bin of=/dev/sdcardDevice
```

```
To restore the backup. If you have 2 sdcard readers you can just
   copy from one to another using dd
```

- When accesing a machine via ssh you can start applications that have a graphical output if you set the display first:

```
export DISPLAY=:0
```

```
also you can tunnel the graphical output to your own computer if you
   add -X when starting the ssh session
```

- You can make an application restart in case it crashes really easy from the same autostart script:

```
~/openFrameworks/apps/myapps/myapp/bin/myapp.sh &
```

```
and now create a myapp.sh script with this contents:
```

```
cd ~/openFrameworks/apps/myapps/myapp/bin/
ret=1
while [ ret -neq 0 ]; do
    ./myapp
    ret=$?
done
```

```
that will restart your application in case it crashes and you can
   still quit it pressing escape (unless your app crashes when
   exiting)
```