SOMATOPIA

Contents

Introduction

This is an introduction to Somatopia, which is being developed in partnership with Cariad Interactive, Raspberry Pi, and Cardiff Metropolitan University, together with special schools and arts organisations.

The aim of Somatopia is to use the RPi to enable young people to create interactions that respond to sound and movement.

Quick Start

Launching Somatopia from the Raspberry Pi desktop.

1. Double click on the Somatopia folder
2. Double click on the Somatopia icon
3. Select "Open in Terminal". Lines of code will appear, then the Somatopia interface will launch with a choice of applications, Sound Wheel, Flow, Call and Response, Space. You will also see an Options button on the top left.
4. Go ahead and select an application, then return to the Somatopia interface, press the 'S' key.
5. Options will enable you to take your own pictures and to choose video on or off. If you are using Somatopia to take a picture you will need to quit out of Somatopia to add your image (explained below).
6. To quit, press ESC.

Making Simple Changes To Sound Wheel

You may be interested in modifying some aspects of the Somatopia app, which we encourage you to try! With the downloaded version from the Internet you won't be able to change any of the code, but you will be able to make some easy customisations by adding the names and portraits in the 'SoundWheel' interactions.

To do this, open the 'data' folder inside the Somatopia folder. Inside there will be several files but the ones we care about are Users.json and the Portraits folder. To change names and add portraits you need to make small edits to the text in the Users.json file. JSON is a way of structuring files so a computer can easily read them. In this file we already have a list of users with names, colours and shapes. You can modify the name, colour and shape associated with any user by simply changing the name with a text editor.

Open the Users.json file by double-clicking on it and try changing the first name to your name by replacing the word '`Placeholder1`' with your name (in quotation marks). The next time you open the application you'll see your name come up as the first member in sound wheel!

To modify colours and/or shapes you can do the same thing as modifying the names. Simply replace the colour and/or shape by selecting alternatives from the following list – do make sure that the spelling is exactly the same as written here:

Colours: orange, red, yellow, light blue, green, dark blue, blue, purple, white, grey, pink.

Shapes: circle, cross, heart, hexagon, square, triangle, asterix.

You can also add images, so that it appears within the shape. To do this you need to add your own .jpg file to the 'Portraits' folder. When you edit the text in the Users.json file it must be exactly the same as the .jpg name.

For example if I'd like to add Alex.jpg, I will change the JSON file so it looks like this:

```
{'Users':[
{ 'name': 'Alex', 'color': 'red', 'shape': 'square' } ]}
```

You can copy and paste this line of code as many times as you need to for all the names / colours / shapes you wish to include, remember to add a comma at the end of each line, but not the final line. For example...

```
{"Users":[
{ "name": "James", "color": "red", "shape": "square" },
{ "name": "Joel", "color": "pink", "shape": "heart"},
{ "name": "Wendy", "color": "green", "shape": "triangle"}
]}
```

Making sure that there is a corresponding image in the portraits folder, for example: Alex.jpg. When you run this in Somatopia, the .jpg image of Alex will appear on the screen inside the shape.

The simplest way to put a .jpg in the portraits folder is to use your Pi Camera as it is already set up for Somatopia. We will explain how to do this, however, if you would prefer to source your images from an external file, i.e USB or from the Internet, check the Raspberry Pi guides (link). Once your file is on the Pi and saved as a .jpg, you can add it to Somatopia by following the same steps.

Taking A Picture With Your Camera

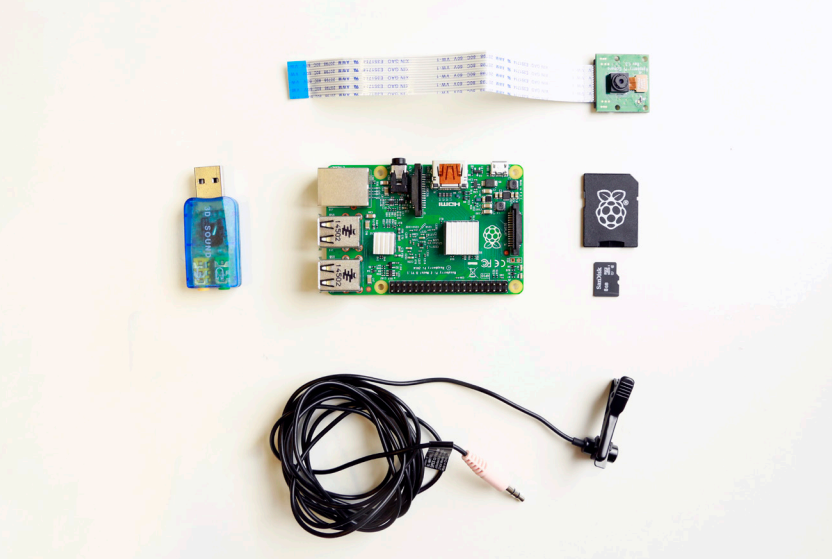
You can also take a picture within the Soamtopia app itself! Go to the options page and you'll see a live feed from the Pi cam in the bottom left corner along with some controls and black and white "background image". To take a photo press the check mark next to the words entitled "Save Your Portrait!" to take a snap-shot of the image you see in the bottom left. this will automatically save an image to the portraits folder entitled "`image[Date].jpg`" where the `[Date]` is the date and time that you took the image. To use it simply go into the data/portraits folder and rename this image to the name of the person you'd like to associate it with. In the example above our user is named Alex so we'd want to name his/her image "`Alex.jpg`".

Once you've renamed the file open up the app again and head over to sound wheel. When you reach that name their image should pop right up inside of their favorite shape!

Worksheet One
Setting Up Your Pi

Build:

We'll start by putting together the Raspberry Pi! There are several ingredients you'll need to bake your Pi to run with Somatopia and openFrameworks.



You will need the following items:

1 × RPi 2 Model B
1 × WiFi dongle
1 × Mouse
1 × Keyboard
1 × PiCam
1 × Microphone
1 × USB sound card
1 × Display with HDMI port
1 × HDMI cable
1 × microSD card Class 10 at least 8 GB
1 × microUSB power supply

Once you have those to hand you can begin putting your Pi together!

Step 1 – 'NOOBS' Software:

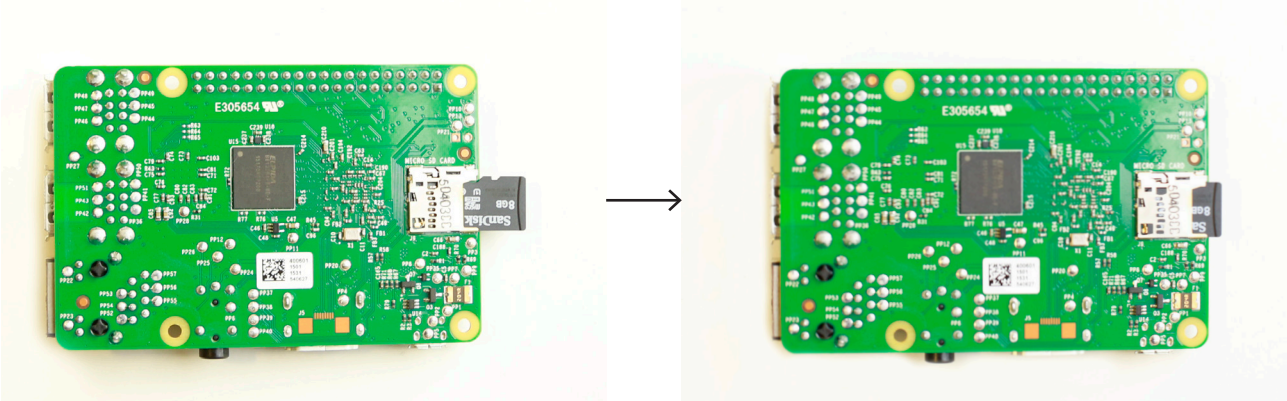
Your Pi may have been shipped with a 'NOOBS' card, which will have the RPi logo on it. If you have this you can jump to 'Step 2 – Setting Up The Hardware.' If not you will need to download the code that runs the Raspberry Pi onto your blank SD card. This software is called 'NOOBS' and you can download it from this link:

https://www.Raspberrypi.org/downloads/

When you have downloaded 'NOOBS' onto your computer, you can follow the instructions here to install it onto your SD card:

http://eLinux.org/RPi_Easy_SD_Card_Setup

Once you've installed it you can plug the SD card into the Pi.



Step 2 – Setting Up The Hardware:

Next we'll need to attach the camera. The PiCam connects to the Pi on the top of the device as illustrated in the images below. Open the clip by placing a finger on each side length-ways and gently pulling upwards. It should pop open but stay attached to the Pi. Insert the strip connected to the PiCam with the blue strip facing towards to the USB ports on the Pi. Ensure that the blue strip is flat and close the clip by gently pushing it back into place and waiting for it to click.

Once the PiCam is in place the rest is quite simple. Attach the microphone to the port on the USB sound card, this can then be inserted into one of the four USB ports, along with the other USB devices: mouse, keyboard, WiFi dongle. Attach the HDMI cable to the HDMI port, which sits next to the camera port.

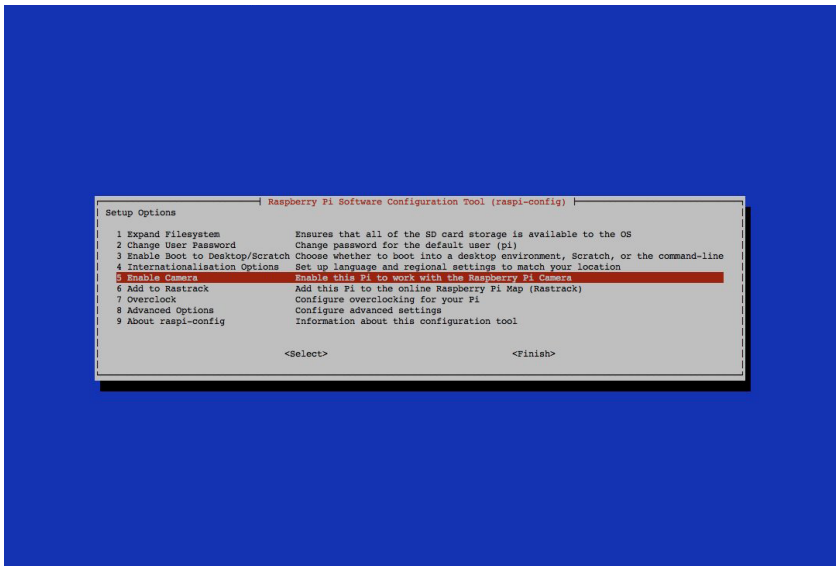
Finally, attach the power cable to the wall and plug it in to the micro-USB power port next to the HDMI port. Once the power is connected you should see a solid red light and a flickering green light near the SD card slot, which indicate your Pi is on! You should also see some scrolling text on your monitor with the RPi logo on the top left corner.

Operate:

Step 3 – Operating The Pi:

Your Raspberry Pi logo will appear on the top left corner of your monitor; the text scrolling up the screen is normal and will happen every time you boot up the Pi.

If this is the first time using your Raspberry Pi you'll reach a screen called '`raspi-config`', which looks like this:



This is an application that comes standard with your installation of NOOBS; it allows you to change the settings on the Raspberry Pi. The row highlighted in red is the currently selected option and you can navigate through the options using the arrow keys to move the selection and the Return/Enter key to interact with the selection.

To start using Somatopia and openFrameworks you will need to set several settings in this screen which will be described below, however if you want to just start using your Pi, go ahead and navigate to <finish> and press Enter to quit the '`raspi-config`' application. To do this use the left and right arrow keys to navigate to the bottom of the list of settings and press Enter/Return when you've selected finish. You can return to this screen any time from the Terminal by typing '`sudo raspi-config`' and pressing Enter/Return.

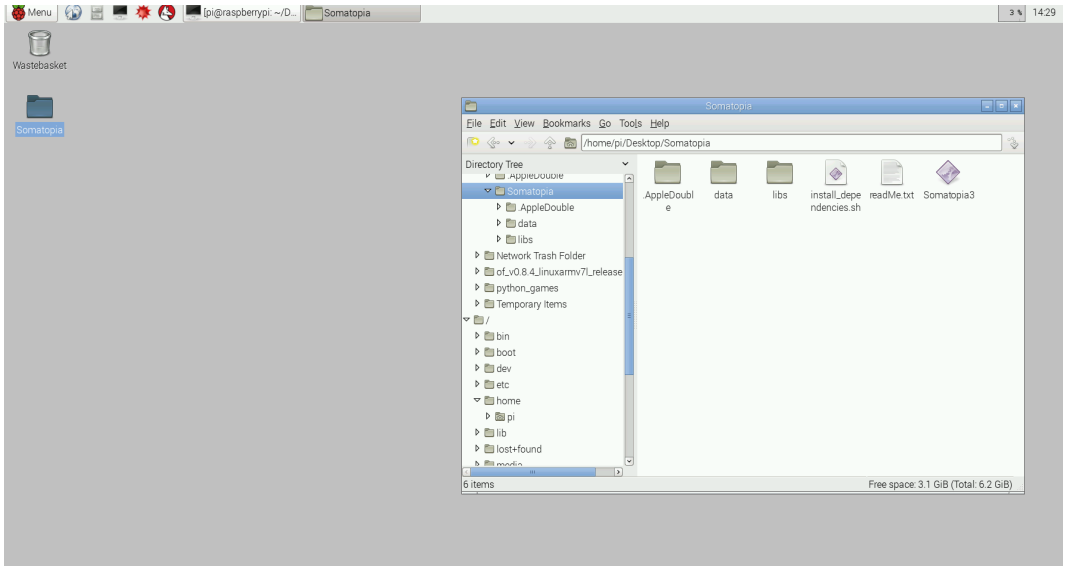
You should now see some green text with a blue dollar sign next to it and a blinking cursor at the bottom of your screen.

This is the Terminal and will allow you to control the computer using text commands. If you type a command and press Enter the computer will try to execute the command you input.

Try it now by typing '`hostname -I`' This is a command that prints out the IP address of your RPi on the current network. There is no network connection currently so you should just see a blank line and then another blinking cursor next to a blue dollar sign waiting for your next command. There are many commands you can send your computer via the Terminal, you can even use your computer with all its functionality from the Terminal alone!

We'll use some commands later but for now we'll go to a more familiar screen by typing '`startx`' and pressing Enter to bring up the desktop.

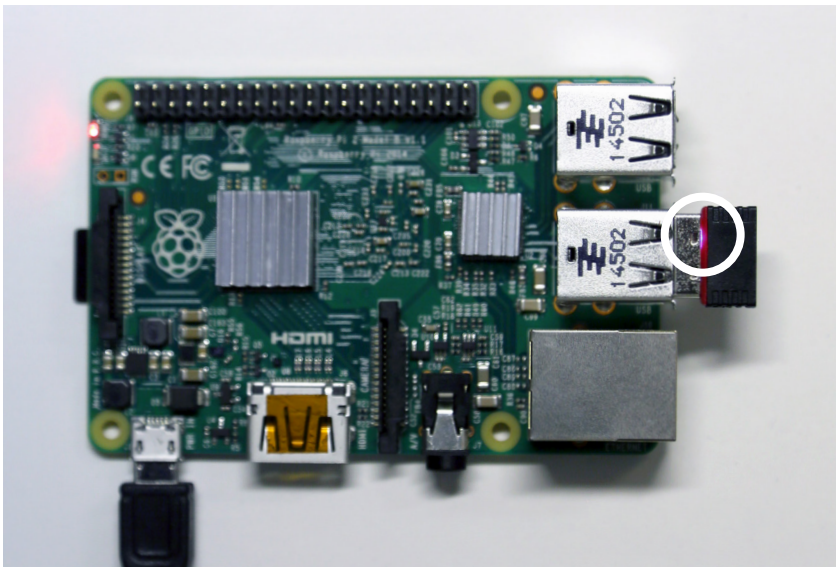
Now you have a mouse cursor and can see a few buttons. This is called a 'GUI' or 'Graphic User Interface' and all modern computers have one. You are now looking at your desktop.



Step 4 – Connecting To The Internet:

The next thing you need to do is connect to the Internet. Go to the top left corner and click on the menu button, next navigate to preferences. Select WiFi Configuration from the dropdown menu. First ensure your WiFi dongle is securely plugged in to a USB port and select Scan in the bottom right corner of the window. Press Scan again and this will display a list of networks that your RPi can join. Double click the network you would like to join and enter any information required, such as your password and press 'Add' at the bottom of the window. It may

take a few seconds to connect. Depending on the model, your WiFi dongle will usually show some kind of indication that it is on. Mine shows a blue light, as shown below:



You can check your Internet connection by opening the web browser. Click on the cursor and globe at the top left of the screen and try typing in any website. I use www.google.com to test my connection. If it loads you're connected!

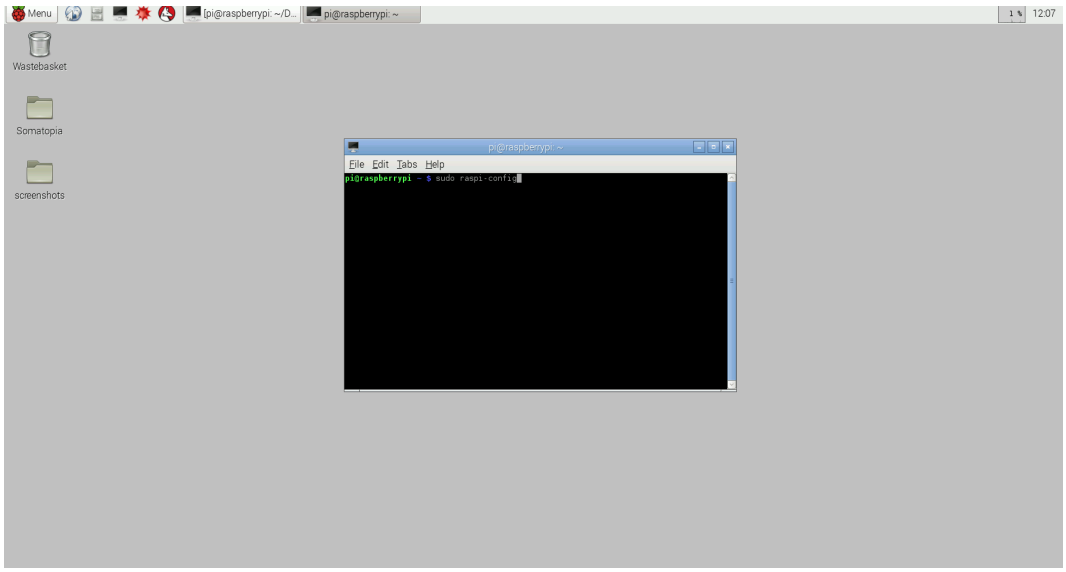
Step 5 – Installing Somatopia:

Now that we are connected to the Internet we can go ahead and download the software we'll need to get started with openFrameworks and Somatopia. First however, we'll need to change some settings so that the computer is ready to run these applications. Here we'll be returning to the Terminal from before and re-opening '`raspi-config`'.

In the top left corner there are several icons, one of which looks like a black computer screen. Click that icon to open a windowed version of the Terminal. Look familiar? That's because it's the same! The program you've just opened is a way of accessing the Terminal within the graphic user interface, which is very useful!

You can type directly into the Terminal here. Start by typing '`hostname -I`' and pressing Enter. The last time we did this we printed a blank line, now you should see an IP address if you're connected to the Internet. Great! You've just executed your first successful Terminal prompt!

Now we'd like to re-open the 'raspi-config' app and change some settings. Type 'sudo raspi-config' into the Terminal and [wait to] press Enter/Return. When we press Enter/Return this command will re-open the config menu for the RPi within the Terminal window. As a quick aside, let's discuss this command you just entered. There were two parts to it 'sudo' and 'raspi-config'. The second part 'raspi-config' is the command you need to open this menu; however, typing it alone would produce an error (you may try it if you are curious). The 'sudo' part is required for certain commands that the computer deems more important than others. Typing 'sudo' allows you to edit files and open applications that most users would not have permission to edit. Using 'sudo' allows you access to lots of things on your computer you normally wouldn't have access to. However, with great power comes great responsibility, so if you find yourself using the 'sudo' command make sure you understand what you're doing! For openFrameworks and Somatopia you will rarely need to use it so don't worry!

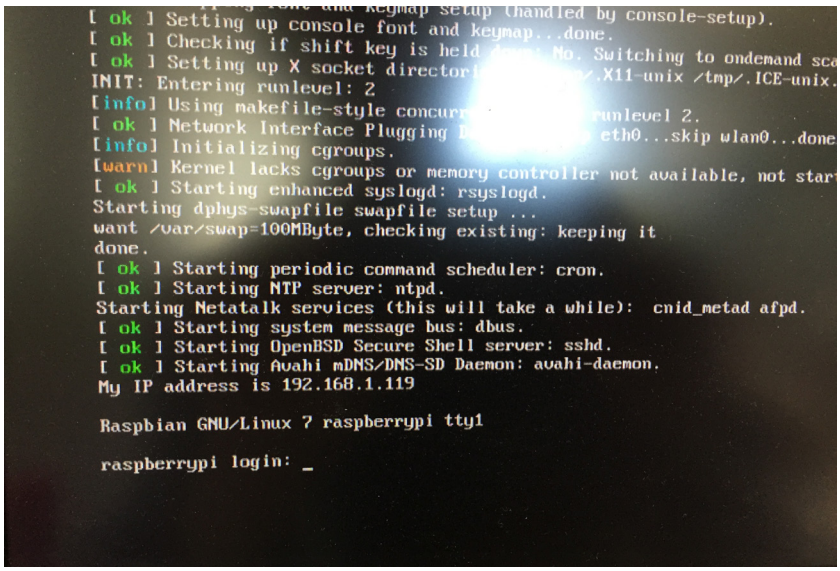


Now you can go ahead and enter the command 'sudo raspi-config' and press Enter. The first thing you'll need to do is Expand the Filesystem. Select Expand the Filesystem (Option 1) and press Enter. If you are using 'NOOBS' you may already have the file system expanded and you will be told this. You may also change the user password. The default password is 'raspberry', however you can change it to whatever you. Option 5 is entitled Enable Camera, which we will need to do to run Somatopia. Go ahead and press Enter when Enable Camera is selected - use the left and right arrows to select. When you press Enter you'll be brought back to the 'raspi-config' page. The last thing we'll need to do is go to Option 8 'Advanced

Options'. Enter the Advanced Options menu and select A3 Memory Split. Your default value here is usually 128 or 64 but we want it to be 256 (double 128). Change the value using the keyboard and double-check that it is correct then press ok.

Ok we're all done! Go ahead and select <Finish> and you'll be asked if you'd like to reboot. Reboot in order for the changes you made to take effect.

Now we're cooking with gas! Once your RPi boots back up you'll have the same code you saw before but we won't be brought back to the rasp-config window. Instead you'll be asked to log in.



The default username is '`pi`' and the default password is '`raspberry`' (unless you changed it in the last step). Enter the username and press Enter then enter the password and press Enter (the password will not show up when you enter it, it will look as if you are typing nothing but it's just a security measure so people can't read your password over your shoulder! You are actually typing). Once you're logged in you'll again be presented with the Terminal. Go ahead and enter the command '`startx`' again to open the GUI. You can set the GUI to open automatically every time you turn on your Pi by following an online guide, for now, logging in and executing startx is the procedure you'll follow every time you want to use your Pi.

From here we are ready to begin working with openFrameworks and Somatopia!

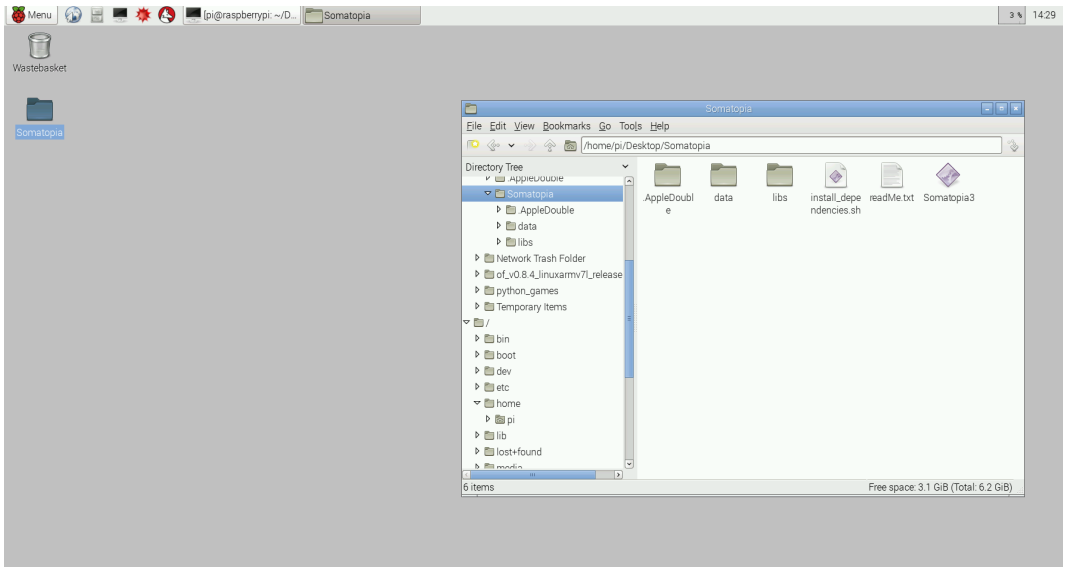
Worksheet Two
Playing With Somatopia

Play:

If you just want to start playing with Somatopia you can download the package from our website. Ensure you are connected to the Internet by opening up another Terminal window and executing the command '`hostname -I`' if you see an IP address pop up you're good to go!

If not go back to the WiFi configuration window and make sure you have everything set up correctly before proceeding. Once you download the folder place it on the Desktop from the downloads folder. In order to run the application you'll need to install some files that allow it to run called 'dependencies'. Lucky for you we've made a program known as a 'script' that will do this for you!

As a quick aside for those interested, 'scripts' are programs that execute commands in the Terminal automatically, so if you wanted to check what your IP address was and start the GUI you could write a 'script' that executed both of those commands so next time you want to do that, you can simply execute the script instead of executing the two separate commands.

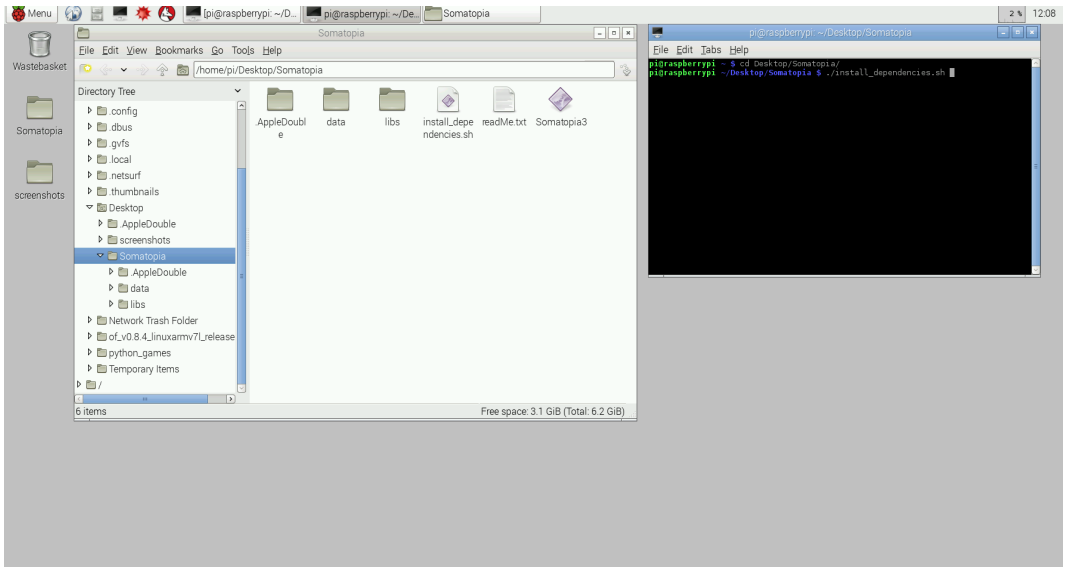


First of all let's have a look at what we're actually doing, on the top left of the screen there is an icon next to the Internet icon, which looks like a filing cabinet. If you open it up you'll open a file explorer window much like the one you have on a Mac or a PC. At the left of the window you'll see a list of folders and on the top you'll see a folder icon with the words /home/Pi written next to them in a clickable box.

Navigate to where you saved the downloaded Somatopia file and open it. There will be 5 objects inside, two folders: one script called '`install_dependencies.sh`' and a file called '`readMe.txt`' as well as an executable file entitled '`Somatopia`'.

We want to run the '`install_dependencies.sh`' script but we'll need special permission to do so, so we'll need to use the '`sudo`' command again in the Terminal. Use the Terminal window you already have open (or open a new one) and execute the following command '`cd`'; '`cd`' is a command we will use a little bit in this setup guide but you'll use it a lot if you want to do more programming, '`cd`' stands for 'change directory' and is used to navigate your computers file system.

If you want to open a folder (known as a directory to programmers) all you need to do is type '`cd nameOfDirectory`' and you'll be there within the Terminal. At the top of the file explorer window that you opened you'll see a string of text describing where in the filesystem you are. It's like an address, it tells you what folder you are in and which folder that folder is in etc. until it gets to the largest folder home, mine says '/home/pi/Desktop/Somatopia' because my Somatopia folder is on the desktop.



We want to execute the '`install_dependencies.sh`' script in Terminal but we need to tell the Terminal we here to look for it. To do that we'll use the '`cd`' command: type '`cd`' and then a space into the Terminal window then copy and paste the string of character in the top of the file explorer into the Terminal. Once that's done press enter and the file path you entered should appear in blue to the right of the blue dollar sign in

the Terminal. This means you are in the directory inside the Terminal and the computer knows where to look for the script titled `install_dependencies.sh`.

Now to run the script you'll need to type '`sudo ./install_dependencies.sh`' which will run the script and install all of the stuff you'll need to run Somatopia. The Terminal will stop twice and ask you to confirm that you want to install certain things. Just type '`Y`' for yes and press enter each time it asks you and you'll know it's done when the blue dollar sign shows up again.

Now you're ready to play with Somatopia! Simply clicking on the icon entitled 'Somatopia' will launch the application.

Have fun!

Making Simple Changes To Sound Wheel

You may be interested in modifying some aspects of the Somatopia app, which we encourage you to try! With the downloaded version from the Internet you won't be able to change any of the code, but you will be able to make some easy customisations by adding the names and portraits in the 'SoundWheel' interactions.

To do this, open the '`data`' folder inside the Somatopia folder. Inside there will be several files but the ones we care about are Users.json and the Portraits folder. To change names and add portraits you need to make small edits to the text in the Users.json file. JSON is a way of structuring files so a computer can easily read them. In this file we already have a list of users with names, colours and shapes. You can modify the name, colour and shape associated with any user by simply changing the name with a text editor. Open the Users.json file by double-clicking on it and try changing the first name to your name by replacing the word '`Placeholder1`' with your name (in quotation marks). The next time you open the application you'll see your name come up as the first member in sound wheel!

To modify colours and / or shapes you can do the same thing as modifying the names. Simply replace the colour and / or shape by selecting alternatives from the following list – do make sure that the spelling is exactly the same as written here:

Colours: orange, red, yellow, light blue, green, dark blue, blue, purple, white, grey, pink.

Shapes: circle, cross, heart, hexagon, square, triangle, asterix.

You can also add images, so that it appears within the shape. To do this you need to add your own .jpg file to the 'Portraits' folder. When you edit the text in the Users.json file it must be exactly the same as the .jpg name.

For example if I'd like to add Alex.jpg, I will change the JSON file so it looks like this:

```
{'Users':[
{ 'name': 'Alex', 'color': 'red', 'shape': 'square' }
]}
```

Making sure that there is a corresponding image in the portraits folder, for example: Alex.jpg. When you run this in Somatopia, the .jpg image of Alex will appear on the screen inside the shape.

The simplest way to put a .jpg in the portraits folder is to use your Pi Camera as it is already set up for Somatopia. We will explain how to do this, however, if you would prefer to source your images from an external file, i.e USB or from the Internet, check the Raspberry Pi guides (link). Once your file is on the Pi and saved as a .jpg, you can add it to Somatopia by following the same steps.

Taking A Picture With Your Camera

You can also take a picture within the Somatopia app itself! Go to the options page and you'll see a live feed from the Pi cam in the bottom left corner along with some controls and black and white "background image". To take a photo press the check mark next to the words entitled "Save Your Portrait!" to take a snap-shot of the image you see in the bottom left. this will automatically save an image to the portraits folder entitled "image[Date].jpg" where the [Date] is the date and time that you took the image. To use it simply go into the data/portraits folder and rename this image to the name of the person you'd like to associate it with. In the example above our user is named Alex so we'd want to name his/her image "Alex.jpg".

Once you'd renamed the file open up the app again and head over to sound wheel. When you reach that name their image should pop right up inside of their favorite shape!

Worksheet Three
How It's Made, Introduction To openFrameworks

Now that you've had a play around with Somatopia, the more technically minded people may be thinking '*How did they make this?*'. Well you're in luck, that's exactly what we're going to talk about next!

We build Somatopia using openFrameworks, a wonderful framework for writing code that lets you do all sorts of great stuff really simply. You can check out the website at:

http://openFrameworks.cc.

In this worksheet we'll be downloading openFrameworks for our Raspberry Pi and building our first few apps!

To download the newest version of openFrameworks for your RPi you'll want to follow the guide they provide here:

http://forum.openFrameworks.cc/t/Raspberry-pi-2-setup-guide/18690

You can skip all the way to Step 6 because we've already done a lot of the setup. This guide assumes you have a Terminal (command line) open and you can simply copy-paste the commands directly into the Terminal. It glosses over a few finer points of the commands but just trust it for now and we'll clear things up in a bit. Step 10 may look a little scary but just copy and paste the command into the Terminal and execute it, we'll talk about it later.

Once you've reached the end of the setup guide you should have some awesome 3D shapes rotating around on the screen. Isn't openFrameworks awesome! Hell yeah!

Now that you've got openFrameworks setup we'll look at what we did. Steps 6, 7 and 8 of the openFrameworks guide just involved downloading and unpacking a files that openFrameworks includes. '`curl`' is a download command and '`tar`' is an unpack command (just like unzipping a compressed file). In Step 9 you installed the same dependencies that we installed before to run the Somatopia app. If there were any changes to those dependencies then they are all now updated, otherwise this step just checked to make sure there were no updates. In Step 10 we had to make sure our system was set up to work on the RPi 2 instead of the RPi 1 which openFrameworks was originally built to work on. We don't want to have to do this same command EVERY time we open a new Terminal window, that would be tedious so instead we'll add it to our .profile file which is executed every time we open a new Terminal. This makes the process automatic. Files with a . in front

of them are 'hidden' from the user which means that in order to see them you need to right click on the file explorer and select 'Show Hidden'. You can edit the .profile file by going to /home/pi/.profile and opening it up in a text editor or you can use the Terminal executing the following commands:

```
cd
nano .profile
```

'nano' is a command that open a text editor inside the Terminal, your Terminal will now be replaced by a text editor that you can navigate with the arrow keys (but not by clicking).

Scroll down to the bottom of the file the arrow keys and paste the line:

```
export MAKEFLAGS=-j4 PLATFORM_VARIANT=RPi2
```

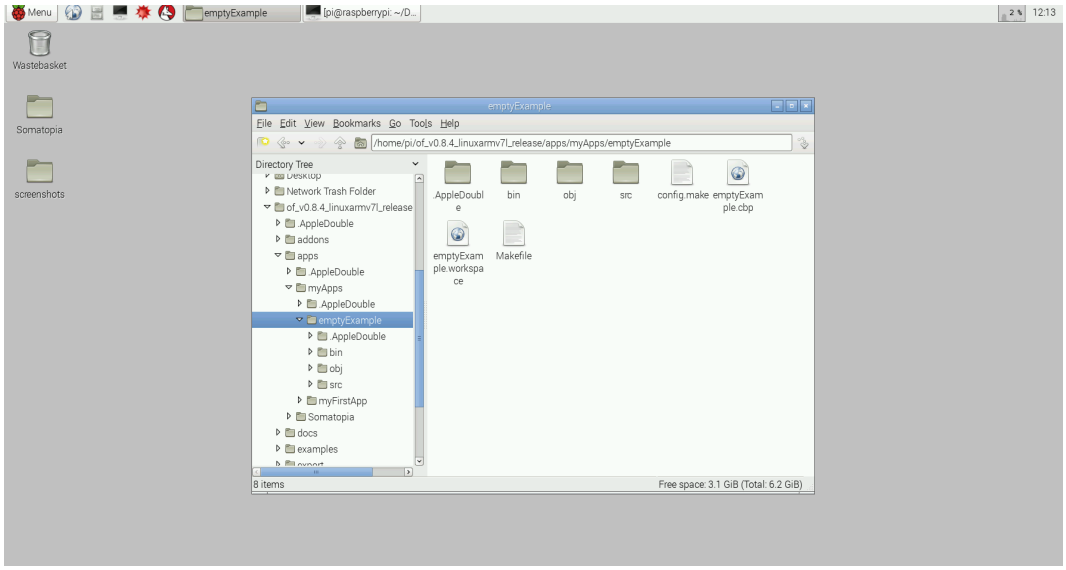
at the bottom now your system will be ready to go every time you open a new Terminal window.

Step 11 asks you to compile an example, if the word 'compile' means nothing to you then read on! Compiling is basically translating code you write into code the computer can read. As you probably have heard computers can only read 1's and 0's but programmers write in English! Cody English, but English all the same. We use letters and words and all sorts of symbols that make it easier for us to work on code. If coders wrote in 1's and 0's all the time we'd never get anywhere so we invent languages and translate the 1's and 0's into words and symbols that we can read.

These languages are called 'abstractions' and while they are great for us computers still live in 1 and 0 land so before we run any code on a computer we need to translate it back into 1's and 0's otherwise known as 'machine code'. That translating into 1's and 0's we call 'compiling'.

With that in mind, Step 11 just takes one of the example projects you downloaded with openFrameworks and translates it into machine code so the machine can run it. The other important thing which will use again that is introduced in Step 10 is the command 'make' this command might as well say 'compile' and you use it to compile your code. Step 12 makes your program run using the command 'make run' which starts up your program!

Ok now that we've clarified a little bit about compiling and we've got openFrameworks up and running we can go ahead and take a look at openFrameworks as a whole. Open up the openFrameworks folder and have a look at the file structure. There are a bunch of folders but the one we care about for now is apps. Apps stands for applications and this is where we will keep all our work. Opening up the apps folder you'll find another folder entitled myApps and inside that you'll see a file called emptyExample. Go ahead and open that folder.



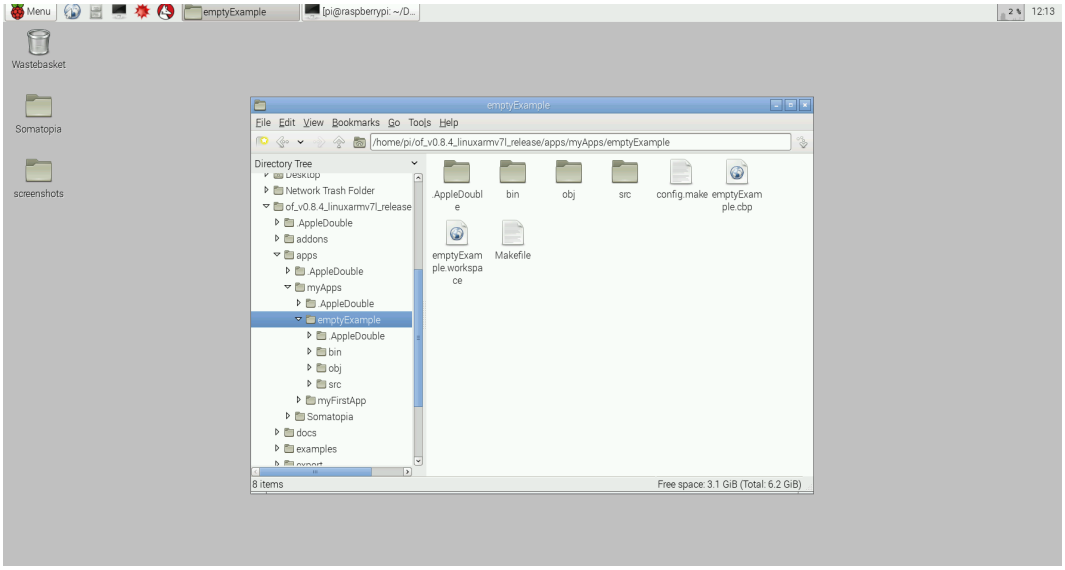
Inside this folder there is a Makefile a folder entitled 'src' and a file called config.make. Remember when we used the make command before to compile an example project? The Makefile in this folder is the file that tells the compiler exactly what needs to be done (makes sense when you think about, a Makefile controls the make command!). When you type the command '`make`' into the Terminal and execute it the compiler looks in your current directory in the Terminal to find a Makefile. If it finds one it will try to compile the project according to the rules in the makefile and if it doesn't it will yell at you! For this reason if we want to compile and run this empty example we need to navigate via the Terminal to this folder. Once again we can copy the file path at the top of the file explorer and type '`cd`' (don't forget the space) into the Terminal and paste the file path after it. Then execute the command to navigate to that file. Now let's take the time to learn a new Terminal command '`ls`'. Try it now typing '`ls`' into the Terminal and pressing enter. You should see a list of all the files and folders in the current directory which include the Makefile, the src folder and a few other files. That's what the '`ls`' command does! It lists all the files and folders in the current directory. If this is the case then

we're in the correct folder we can now type '`make`' to compile this empty example! Once it's done compiling and you've got the command line back you can type '`make run`' to run the example.

If you see a grey screen, don't worry you're doing it right! The empty Example is the most basic openFrameworks app possible and it does nothing, now we can add some of our code to the empty example and get it doing some fun stuff! You can close the empty example app by pressing escape.

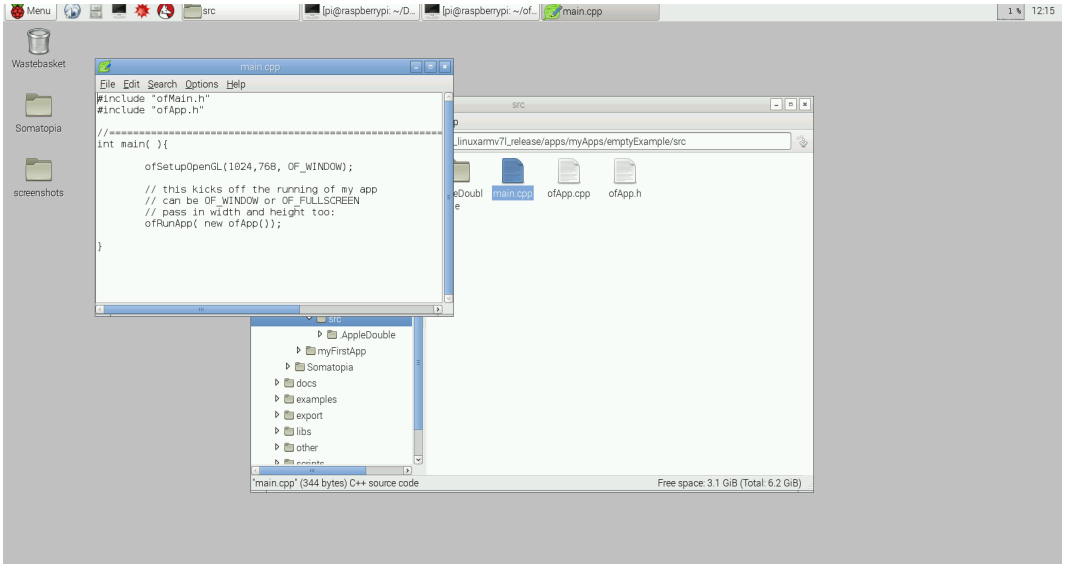
We want to keep this empty example folder unchanged so that we can copy it and use it as a starting point for all our projects, so we'll go back to the file explorer and copy paste the folder. It will ask you to add a new name and you can name it whatever you like, I named mine 'myFirstApp'. In the Terminal let's navigate to our new folder by typing '`cd ..`' here the .. tells '`cd`' to move back one folder. Then type '`cd myFirstApp`' (or whatever yours is called) to enter the folder. Just to test again execute the command '`make`' then execute '`make run`' once it's finished and you should have another grey window! Great, now exit the app by pressing escape and let's have some fun!

Open the src folder. This folder is the foler in which we are going to be making all of our changes. It's worth taking a moment now to talk about what language we'll be writting our code in. You've probably heard of 'programming languages' and there are many out there. Some of the most famous include Java, Javascript, Python, HTML, CSS, C, and C++. All programming languages are a little different and have different strengths and weaknesses. openFrameworks is written specifically in C++ so that is what we will be working with. C is one of the most basic programming languages out there and C++ is the upgraded version of C with a few extra bits. For the purposes of this tutorial, we'll only be using features of C so when you tell people 'I'm an epically good C++ programmer using openFrameworks' you'll be correct but basically you will have only used the C part of C++. This is a great place to start!



Now if we look into the folder entitled src we see three files one entitled main.cpp, ofApp.cpp and ofApp.h. Let's quickly describe what each of these files does. Firstly let's describe the extensions .cpp means it's a C++ file and .h means it's a header file. Header files are like tables of contents for .cpp files, they include a bit of information about all the things that .cpp file has in it and what it can do. .cpp files hold all of the actual computations and operations that make the program run! We use .h files so if you want to open a new project that someone else has worked on you can look at their .h files for a general overview of what they did without having to read through all the nitty-gritty technical stuff.

ofApp refers to openFrameworksApp, so the ofApp.cpp file contains all the computations that our openFrameworks app will do and the ofApp.h file is a table a content of all the functionality of openFrameworksApp will have. The main.cpp file is the most important file in the whole project which we'll explain in a minute. This is the file that allows us to run the application in C++. Go ahead and open up the main.cpp file. Here there are 5 lines of code that start our project running.



Computers are very complicated machines but they think very simply. The computer goes down this code one line at a time and executes whatever the line tells it to do. For that reason we'll look through the lines of this file one at a time to discuss some of the basic concepts of C++ programming then we'll jump into making our own changes.

The first two lines are called includes. Includes allow us to connect multiple files to each other. If we didn't have #include lines then we'd need to write all our code into one long file, instead we can write stuff into separate files which we #include in other files to keep our work organized.

The first line includes the file ofMain.h, this is the line that makes your project an openFrameworks project rather than just a normal C++ project, it gives you all the functionality that openFrameworks offers like nice drawing, text rendering, image rendering, sound processing etc. If you didn't have this line and you wanted to say, draw a circle, you would need to write all the code to do that yourself! What openFrameworks gives us is a bunch of helpful tools that allow us to do these tricky things really simply because someone in the openFrameworks community has already written that tricky code. The next line is `#include 'ofApp.h'`, this line simply includes all the code that we are about to write. We are going to modify the contents of ofApp.cpp which is included in ofApp.h (you'll see the `#include 'ofApp.h'` line is the first line inside the ofApp.cpp file when we look at that file later). So to quickly recap, the first two things we've done in our project are include all the powerful tools openFrameworks offers to our project and connect our app to our project.

The next line looks like this:

```
//=====================================================
```

Now you may be thinking 'what the heck does this mean?! Does this long string of = signs mean anything to the computer?!'. The answer is, perhaps surprisingly, no! This line is called a comment and is never ever read by the computer, it only exists to help humans reading the code understand it better! Any line of code with a double / in front of it i.e. '`// Hello`' this is a comment and will never be read by the computer. There are a few more lines like this in this file. Computers read one line at a time and the system is set up such that if it ever encounters a double / then it will just skip over everything written on the rest of that line. So often if you have some complicated line of code you'll write comments in plain English around it to clarify what exactly is going on. In this case, the comment is there just for visual clarity, like drawing a big line between two parts of your notes in a school-notebook.

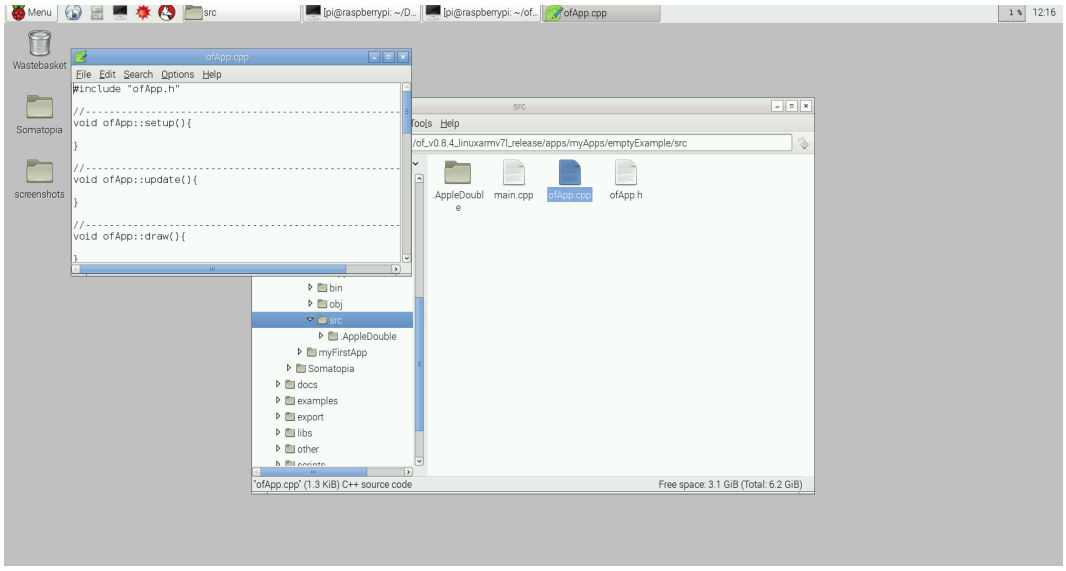
The next line is the following:

```
int main ( ) {
```

This is called a function declaration. Functions are very useful things that exist in computer programming which allow you to package up functionality into little pieces to be used over and over again. Say you wanted to draw ten circles on the screen, instead of writting all the code required to draw a circle ten times, you could write the code once as a function called '`drawCircle()`' then just call '`drawCircle()`' 10 times. This makes code much easier to read and more compact. Let's look through this line at the different pieces of the code: firstly there is the word '`int`'. You'll see this word a lot and it's a shortening of the 'integer'. Integers are whole numbers (0, 1, -3, 8 etc.) 0.5 is not an integer because it's not a whole number! We'll use this word a lot but here it is used to describe the 'return type' of the function '`main`'. The return type is what a function gives us back, i.e. if you want a function that adds two numbers together you'll want that function to give you back (i.e. return) another number which is the sum of the two you put in. In this case we want our main function to return an integer that describes whether or not it's run correctly. This is a standard part of all C++ programs, that they have a function called main that returns an int. The next word '`main`' is the name of the function. In general we can name functions anything we want, except in this specific case. Every C and C++ program ever written has a function called '`main`' that must return an '`int`', it's the function that runs

EVERYTHING. The next bits are parentheses which follow every function and denote it as a function. The final character is a curly brace '{'. After each function their are a pair of curly braces '{ }' where you put all the functionality of the functions (if you scroll down to the bottom of the file you'll see a closing curly brace. This is called 'scope' and it denotes what operations are in which functions, when you call a function your function does whatever you've written in between the curly braces.

The next few lines are comments and functions that are called inside of the main function. Basically they set up the windows that our app will live in and start our app running! For now let's pass over these functions. Close the main.cpp file and have a look at the ofApp.cpp file.



This is where all the magic happens! Let's pause here and talk a bit about what we'd like to do. The main strength of openFrameworks is that it allows you do draw things on the screen easily. There are LOTS of complex things that go into drawing to a computer screen, and openFrameworks does a lot of the hard stuff for you and let's you write really simple code that will get some amazing results. If we want to have some animations in openFrameworks we are going to need to do 3 things, we'll need to set up all the stuff we want to draw, we need to update what we're drawing between frames and we need to actually draw the frames. Lucky for us this is exactly the structure of our program that is presented here! Let's look through this file one line at a time just like we did the main.cpp file. The first line is a familiar one:

```
#include 'ofApp.h'
```

This means that at the beginning of this file we're adding a table of contents of all the operations this file can perform.

The next line:

```
//--------------------------------------------------------
```

is another comment which the computer does not read.

The next line is another function declaration, just like main!

```
void ofApp::setup(){
```

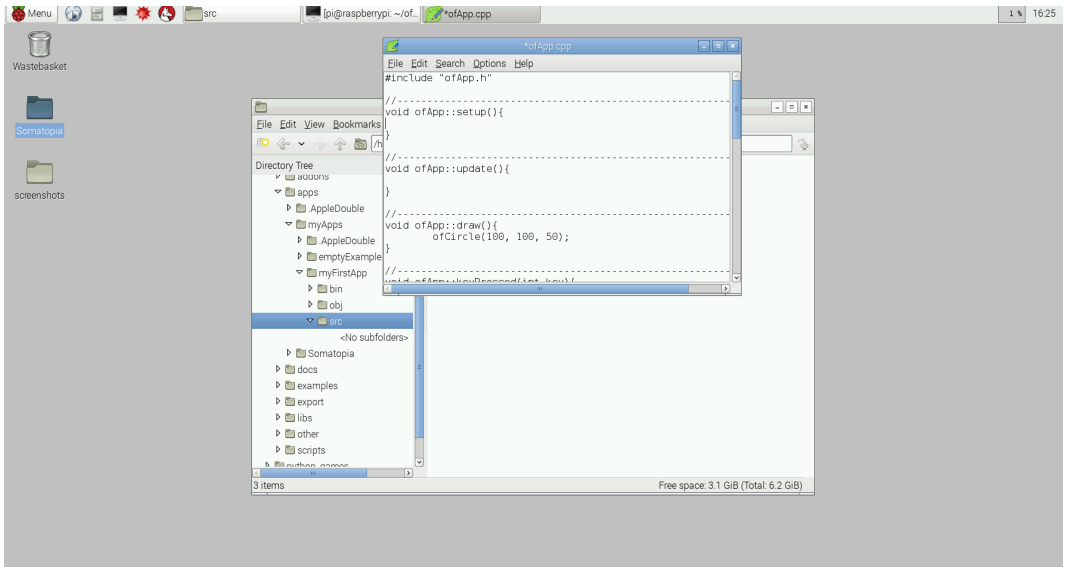
Here we're going to learn a new word '`void`' which is the '`return type`' of this function the same way '`int`' was the return type of our main function. Here void means nothing, i.e the function returns nothing. Easy! Can you guess the name of this function? The same way '`main`' was the name of the last function '`ofApp::setup`' is the name of this function, However we'll just call it '`setup`' Then we see the same pair of parentheses as before denoting is as a function, then the opening and closing curly braces. It is inside these curly braces that we'll write all the functionality that we want to have happen when we execute our setup function, but for now it's empty so if we execute '`setup`' nothing will happen.

Have a look at the next two functions in this file, they are names '`update`' and '`draw`'... how coincidental just the structure we wanted in the first place! This is how openFrameworks works, it has 3 primary functions, '`setup`', '`update`' and '`draw`' built in and when yo start your app running it runs them in this order: setup -> update -> draw -> update -> draw -> update -> draw -> update -> draw ... on forever! There are a bunch more functions below that allow you to do thinks when a key is pressed or when the mouse is moved etc. but for now we'll just use the first three functions.

So now that we understand the basics of openFrameworks and a little bit about programming and functions we're ready to build our very first app. The first thing we'll do with openFrameworks is something we've spoken a bit about before, and it's drawing a circle! As I mentioned before there are lots of complexities that come into play when you're drawing something on the screen, but openFrameworks gets rid of all of these and packages them up in nice, simple functions.

The function we're looking for is called: '`ofCircle`' this function, when called draws a circle at a given x and y position with a

given radius. Since this will be the first time we call a function let's walk through it slowly. We'll want to write our line of code in the 'draw' function because we'll want to 'draw' it every frame! So in the 'draw' function we'll need to write our line of code in between ther two curly braces on it's own line. First we write the name of our function 'ofCircle()' followed by the parantheses. Bewteen the parantheses we'll list the parameters that we want our circle to be drawn with separated by commas. To define a circle we need 3 numbers: an 'x' position, a 'y' position and a 'radius' (in that order). All our units are pixels and the origin of our coordinate system starts at the top left of the window with the positive x direction towards the right and the positive y direction towards the bottom of the window.



Let's start with a circle at the point (100, 100) so 100 pixels to the left, and 100 pixels down and let's say we want the radius of our circle to be 50 pixels. The radius of a circle is the length of a straight line from the center to the edge. To do this we will add the three parameter to our function call so the line will look like this: 'ofCircle(100, 100, 50)' The one last thing we're forgetting is a semi-colon at the end. At the end of each line of executing code we need to put a semi-colon to tell the computer that that line of code is over. So the final line will look like this:

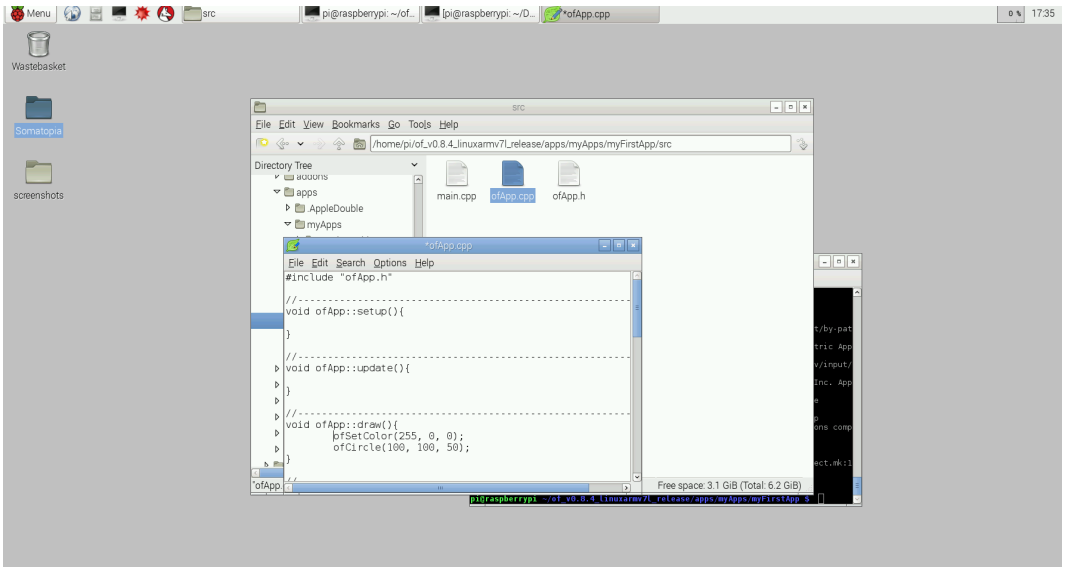
```
ofCircle(100, 100, 50);
```

Let's give that a try! Navigate to the folder entitled myFirstApp in the Terminal and execute the command 'make' to compile the new code! Then once it's finished compiling type 'make run' and you should see a white circle near the top left corner of your oF

window! So that was pretty easy, there are lots of things we can do with openFrameworks that will teach us a bit more about computer programming now that we have this circle. In the next few sections we'll make the circle move, bounce and change it's color whcih will introduce a few more concepts of programming!

Changing The Circle's Color:

A common concept in programming is that of a 'State Engine'. A state engine is a system wherein you can change certain settings which are saved for later use. One such setting we may want to change is the color we are drawing the circle in. We havn't told the app anything about what color we want to draw the circle with so it's defaulted to white but if we wanted to tell it what color to draw in all we'd need to do is set the drawing color using the function '`ofSetColor();`' This function takes 3 parameters which in this case are numbers between 0 and 255 representing how red, green and blue we want our color to be. A value of 0 means none of that color while a value of 255 means the maximum amount of that color. If we add the line: '`ofSetColor(255, 0, 0);`' Inside the draw function before we draw our circle we're saying 'let's set the color to 255 red, 0 green and 0 blue' or 'everything I draw from here on our should be 100% red 0% green and 0% blue until I change the color setting again'. Let's give it a try, add the line '`ofSetColor(255, 0, 0); just above the ofDrawCircle(100, 100, 50);`' line but still inside the '`draw()`' function. Compile the project again using the '`make`' command and then use make run to run it and tahdah! a red circle! Have a play around with some other colors by changing the numbers you give the ofSetColor function. You can also change the background color by using the function '`ofBackground()`' and passing it 3 numbers for red, green and blue aswell.



Making The Circle Move:

So we've had a lot of fun with our static circle but if we could get it to move around that would be even cooler. Let's say we want to move our ball downwards in a striaght line. This means that each frame we want to change the position of the ball by some amount in the positive y direction. All we can do is tell the ball where we want to draw it next so we'll need to save the position of the ball at any given moment using a 'variable' a variable is a letter or word one can put in place of another value that stores the value for later use. To do this we'll need to open up the ofApp.h file and add our new variable to the table of contents of our program. Scroll to the bottom of the page and before the line that reads '`};`' insert a line that reads '`int circleY;`'. This line is called a variable declaration and it tells the computer that we are going to be saving a value with integer type (a whole number) under the name '`circleY`'. Save that file and re-open the ofApp.cpp file. Inside the '`setup()`' function we'll add set the value of circleY to an initial value by adding the line: '`circleY = 100;`'. Here we're using the = sign in a way that is common in computing. '`=`' in C++ and C is the '`assignment operator`' which means that it assigns values to variables. Here is stores the value 100 which we want to be the initial value of our circles Y position inside the variable '`circleY`', now if we ever use the variable '`circleY`' in place of a number it will be just like using the value 100! That is in fact exactly what we will do, let's replace the second '`100`' in '`ofCircle`' with the variable name: '`circleY`'.

Let's compile and run the app again. There won't be a change because we've just replaced the value 100 with a variable with the same value! What we can do now is change the value of '`circleY`' and keep it saved each frame. In our update function we can write the following line: '`circleY = circleY + 1;`'. Now what we're doing is setting the value of circleY to be itself plus 1 pixel just before the draw the circle each time. Let's compile and run that and see what happens. Amazing! we've got a moving circle! If you let it run for a little while you may notice one issue and that's that the circle disappears off the bottom of the screen never to be seen again! This is because when it reaches the bottom we keep incrementing the variable making the y position of the circle larger and larger but the screen in only so large so we can't see it when the y position gets too large!

Making The Circle Come Back:

Let's make the circle loop around the window like it would if it were in Pac-Man so that is always stays on the screen. To do this we'll want to check when the position of the circle is equal to the height of the window and then set the value back to the top of the window. To do this we'll use what is known in programming as a 'conditional statement'. We want to check if a certain condition is met (ie if the circle is at the bottom of the screen) and execute a command only if that condition is met. To do this we'll use an '`if`' statement. Just after the line: '`circleY = circleY + 1;`' let's add the following bit of code:

```
if(circleY > ofGetHeight() + 50) {
circleY = -50;
}
```

This is a bit tricky to let's tackle it on piece at a time. The if statement here consists of the word if and then everything in the enclosing parentheses after it: '`if(circleY > ofGetHeight() + 50)`' this can be read almost like normal English: 'if the variable circleY has a value that is greater than the height of the screen plus 50 pixels...' The stuff inside the parentheses is a comparison between two numbers, the value contained in '`circleY`' and value returned by the function '`ofGetHeight()`' (a built in openFrameworks function that tells you the height of the screen) plus 50 more pixels to account for the radius of the circle. If it is true that the value contained in '`circleY`' is larger than the value returned by '`ofGetHeight() + 50`' then the computer executes the code between the curly braces, i.e. it sets circleY to by -50 pixels. Let's compile and run this again and you should see the smooth motion of the circle. Pretty cool huh!? Can you make the circle move faster or slower? how anout making it change it's color when it loops around? What about making it move in the x-direction aswell or allowing it to change directions when the mouse is pressed?

These are all things that you can do pretty simply with openFrameworks. check out the website openFrameworks.cc and click on 'documentation' to see a list of all the functions that openFrameworks offers.

Worksheet Four

Extending Somatopia: An Intro To Object Oriented Programming

Now that you've gotten an intro to openFrameworks we're going to dive a bit deeper into the code by adding our own interaction to the 5 existing ones in Somatopia! In this section we're going to get a bit more advanced and it will assume that you've played around with openFrameworks a bit, maybe made some of your own basic projects and now you're ready to jump in on something a little more advanced.

The main topics in this worksheet will be as follows:
1. Using Addons With openFrameworks
2. Using Object Oriented Programming With C++

These are all important topics for programming with openFrameworks and programming in general. Let's start by talking about addons.

Using Addons With openFrameworks:

One of the dirty secrets of programming is that programmers, in general, are very lazy people. We don't want to make extra work for ourselves when we don't have to, in fact that's sort of the whole reason computing got started in the first place! Lets get a computer to do all the tedious tasks like calculating where to draw the ball next so you don't have to!

Because of this programmers have built a myriad of tools to get around doing real work and one such tool in openFrameworks is called "addons". Addons are a way of including other peoples code in your own projects so you don't need to reinvent the wheel every time you go to do a project. Say you want to build a project which is a magic mirror in which people can look and become happy. We want to draw a live camera feed of people looking at our mirror and draw a smiley face over the faces of people who aren't smiling. Without addons this would be quite the challenge, you'd need to firstly figure out a way of looking at an image a determining which parts of the image are peoples faces (no mean feat) then you'd also need to develop an algorithm that allows the computer to determine whether or not those people are smiling. This could take a lot of work and you may find yourself thinking that you should perhaps go back to your simple bouncing ball project but the good news is some very clever people have already figured out how to do both of these things long before you ever sat down to code your project!

Not only have these people figured out all of these tricky algorithms already, but they've also made much of their code available through "addons" which are basically chunks of other

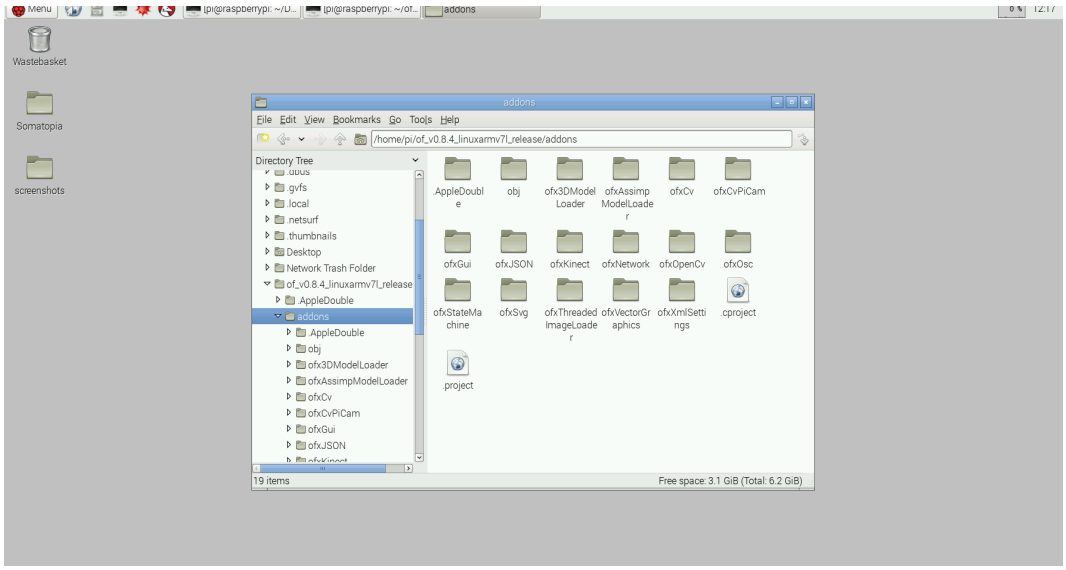
peoples code that you can use for your own projects because these clever and generous people have "open sourced" (i.e. released to the public for free) their work. With respect to the smiling project I described above there is a great addon called ofxSmile created by backercp which you can check out if interested but for now let's talk about the addons we've used to build Somatopia.

We use 5 addons for Somatopia which I'll list below with a brief description:

ofxJSON – This addon allows us to read files of the type .json which are text files that hold data. You may remember the Users.json file which we modified earlier to include more users in our Sound wheel interaction and now you know that this is the addon that allows us to read that file!

ofxOpenCv, ofxCv and ofxCvPiCam – This set of three addons allows us to use code in "openCV" libraries which stands for "Open Computer Vision". These are the addons which we use to do analysis of images which allows us to do camera interaction in the interactions Flow and Space. We'll talk about these a bit later.

ofxStateMachine – This addon allows us to structure our App in a such a way that we can have multiple interactions which we switch between. We've set up our app such that each interaction is a "State" and the state machine allows us to switch between these states in a clean way.



To include an addon in your own project is very straightforward on a raspberry pi. All you need to do is download the addon from the internet (many addons have their source code available on GitHub) and place it's folder in the "addons" folder.

Check out http://ofxaddons.com/ for a huge list of all the addons that openFrameworks has to offer (there are even more non-official ones floating around GitHub too but I think you'll find this is a good place to start)! Once it's in your "addons" folder you'll be able to add any addon within that folder to your project by adding the name of the folder at the bottom of the addons.make file within your project and following the instructions provided by the author of the addon. Sometimes these instructions are very straightforward and usually just involve adding a line to `#include addons .h` file at the top of your `ofApp.h` file and sometimes you need to do a few more things like download additional libraries etc. depending upon the complexity of the addon.

We won't need any additional addons to add our new interaction but it will be important to understand that they exist and how we're using them in our code. To add an interaction we'll need to add a new "State" to our state machine. Let's quickly have a look at our testApp.cpp file. We'll gloss over some of the details here because you're already a rocking openFrameworks coder but quickly we'll have a look at the structure of the code. All your openFrameworks apps have this file (maybe it's called ofApp.cpp) and they all have this setup function at the top. What we're doing in our setup function is setting up our state machine. You can think of each of our interactions as individual openFrameworks apps with their own setup, update and draw functions as well as other functions like mousePressed stateEnter and stateExit. We use our state machine addon to connect all of these apps together and swap between them easily. If we scroll down to line 100 we'll see when we add all of these states:

```
stateMachine.addState<SplashState>();
stateMachine.addState<OptionsState>();
stateMachine.addState<FlowState>();
stateMachine.addState<CRState>();
stateMachine.addState<SpaceState>();
stateMachine.addState<SoundWheelState>();
stateMachine.addState<MirrorState>();
```

Splash state is the main page with all the buttons on it, Options state is the option page found in the top left corner, Flow state is out flow interaction etc. To add our own state we'll need to

add it here. We've already made a state for you to add yourself called "Rhythm" which you'll find by looking in the src folder there is a `Rhythm.cpp` and a `Rhythm.h` file as well as a `Ball.cpp` and a `Ball.h` file. To add this state to our app we'll need to add a line here that looks like this:

```
stateMachine.addState<RhythmState>();
```

The next step we'll need to take is connecting the `RhythmState.h` file with a `#include` line in the `testApp.h` file. Now we knew this was coming and we've cheated a bit by adding it to the `.h` file already so you don't need to do it yourself.

Now if we compiled and ran our app our Rhythm state would be in the app but we'd have no way to access it! We'll need to add a button to access it in our Spash State. Let's open the `SplashState.h` file and you'll see where we initialize all of our "Statebuttons" which are the buttons on the splash page which allow us to jump to any interaction we want. Now, we're getting dangerously close here to discussing something called "Object Oriented Programming" and we're not quite ready to do that yet (but we will in a moment!) for now look at how we declared the intergers "`offSet`", "`buttonWidth`" and "`buttonHeight`".

This should be totally familiar to you, you declare 3 integers which we'll give values to later. In object oriented programming we can do the same kind of thing but with any object we create ourselves, not just integers, floating point values strings etc. but also stuff like "bouncing balls", "states" (like in our state machine) and even "StateButtons" as we have here. You'll see we've commented out one of the lines "`StateButton rhythm;`" which we'll want to comment back in now! Now have a look at our `splashState.cpp` file and you'll see some more commented out lines, line 19, 27, 49 - 51 and 88 - 91. These are the lines that control initializing, drawing and allowing us to interact with our new state button. Once these lines are all un-commented go ahead and compile the code and run it.

And voila! Try clicking on our new rhythm button and you'll see a red ball bouncing around the screen which should bounce off of objects in the frame! The ball will behave strangely right now because the background is not set properly but we can fix that. Leave the rhythm state by pressing the right mouse button and go to the options state. You'll see in the bottom right there is a background image in black and white. If you place the RPi and the camera in a place where they will not move and move out of a frame pressing the space bar will change this background image to the image on the left (the current camera

feed). Now if you re-enter the frame and return to the rhythm state you'll find the ball does indeed bounch off of you and the sides of the screen! Pretty cool huh?

Using Object Oriented Programming With C++

Now we've spoken a bit about state machines, added our new mode and introduced the idea of object oriented programming very briefly. Now we're going to dive into the interaction that we just added to talk a bit more about object oriented programming. Let me give a quick theoretical introduction to what object oriented programming is to a practical programmer such as myself.

Object oriented programming is more or less a way of organizing programs in a lucid and modular way. The idea is simple, let's start with the idea of an integer. An integer is a data type in C and C++ that refers to a piece of data stored on 4 bytes of data which can be interpreted as a number between −32767 and 32767. There are operators for integers, you can add them together, you can subtract them, you can assign them values you can even divide and multiply them. There are lots of data types native to C++ including ints, floats, doubles, longs, bools, strings, chars, the list goes on. While these basic data types are excellent for holding values like numbers or pieces of text quite often in computing we want to store more information than just a simple number.

We may, in a very basic example want to store the value of a 2 dimensional vector. A vector is defined as an object that has both direction and magnitude, it is essentially an arrow that points a certain distance in a certain direction. To define a vector in euclidean space all we need is two integers that describe where it is in the two dimensions, we call these the x value and the y value. You may remember from the previous worksheet that we drew a circle which moved around the screen and we defined it's position using two integers `circleX` and `circleY`. Those two number constitute the vector that starts at the top left hand corner of the screen and points to the position of the circle.

Now using a pair of integers is all well and good for one circle, but let's say we wanted to have 100 circles, then suddenly we'd need `circleX1`, `circleY1`, `circleX2`, `circleY2`, `circleX3`... and on and on and every time we wanted to draw a circle in it's correct position we'd have to pick the correct integers. Instead, we'd rather bundle that pair of of integers that define the circles position together into a new object called a vector.

That way all we need is position1, position2, position3... and we can easily keep track of our list. To do this we'll use what's called a class.

The class is a tool in C++ that let's you bundle together different data types and keep them together. Let's call our class "myVector" so classing it would look like this:

```
class myVector {
public:
int x;
int y;
};
```

That's it! It's pretty simpe the only thing to note is the line "public:", in C++ you need to tell a program whether it is allowed to directly access fields in your class. Fields are by default private which means no other classes can touch them but we don't really mind making these fields public so we will. Now we can declare a new vector just the same way we'd declare an integer:

```
myVector circlePosition;
```

To set the values of the x and y we use '.' notation which allows us to access the "fields" or data types within the classes, to set the circle position to (50, 100) we would do the following:

```
circlePosition.x = 50;
circlePosition.y = 100;
```

now instead of having two lists, one of all the x values and one of all the y values of say 100 circles we can just have a single list of all the position of the circles and access the x and y values using '.' notation.

This is a great start, but we may want to create even cooler classes. If we can put two integers together to create a vector, why not put a bunch of numbers together to make a dog? We can do this in just the same way, let's make a class called "dog" that has several fields, dogs have names, a number of legs, a maximum running speed, perhaps they have a level of hunger which indicates how much they want to eat. Maybe they even have a color and a sensitivity of smell, all of this is possible with a class:

```
class dog {
public:
```

```
int numLegs;
string name;
float hungerLevel;
float maxRunningSpeed;
double smellSensitivity;
ofColor color;
};
```

So now we can store not only simple objects like vectors but much more complex objects like dogs! Now what is we also want our dog to be able to do things? It's all well and good to have a dog that has a hunger level but if he can't eat how can he ever change his hunger level? For this we have things called methods. Methods are functions that can be called on a particular class. So if we wanted our dog to be able to eat we would modify our dog class by adding a method declaration inside it like so:

```
class dog {
public:
int numLegs;
string name;
float hungerLevel;
float maxRunningSpeed;
double smellSensitivity;
ofColor color;

void eat() {
hungerLevel++;
if(hungerLevel > 1.0) hungerLevel == 1.0;
}
};
```

Now our dog can eat! we can use the method the same way we accessed the fields so if we initialize a dog:

```
dog Sparky;
// initialize all the fields of sparky our dog
Sparky.eat();
```

This will call the functionality of our dog to eat!

Super cool right? This is object oriented programming at it's most simple. You can have a look in the source code for the the rhythm state we just added and see this kind of coding in action. In this case we define a new class called "`Ball`" which contains all the code we want our bouncing ball to execute it includes fields that hold it position and color and methods that allow it to bounce off the walls and off of people in the image.

Object Oriented Programming has a lot of further topics but it's really a great way to write programs and you'll find that it helps you organize your mind as well! It also means that you can pick up bits of old code you've written and reuse them just like addons (in fact most every addon is just someone elses class that you get to use!) even the entire app that we're building is itself an instance of an "`ofApp`" class! Isn't that something!

Thank You!

Somatopia has been funded through the Raspberry Pi Foundation.

We would also like to thank Ashgrove School, Cardiff and Trinity Fields School, Caerphilly for their participation.