openFrameworks tutorial, part I: Wave Distortion Effect

This tutorial is intended as a primer in openFrameworks, or 'OF'. It is intended for people who are new to OF but who already have some basic knowledge of object-orientated programming. It will be ideal for people transitioning from Processing. If you already know how to setup an openFrameworks project, feel free to skip ahead to the section called Fun part. In part one of this tutorial we will be building an image processing tool that deforms a given image into a wavy pattern. You can see the result in this video:

You can download the source code here.

Introduction to OF

openFrameworks as described by its creators:

"openFrameworks is an open source C++ toolkit designed to assist the creative process by providing a simple and intuitive framework for experimentation."

OF is inspired by the Processing project and can be considered as Processing's 'big brother', giving you more power and flexibility in your code-based creative projects. It doesn't come with an IDE, so you will have to use Xcode, Eclipse, Code::Blocks or Visual Studio (depending on your Operating system). I will be using Xcode 5 in this series of tutorials, together with OF version 8.0. You will need Xcode and OF for this tutorial, which can be. You will find an 'Xcode setup guide' in the downloads section of OF, which will help you get started.

You can get more information about OF on their website. In the Tutorial section there is some helpful information for beginners, including one page for those transitioning from Processing to OF:

http://openframeworks.cc/tutorials/introduction/000_introduction.html
http://openframeworks.cc/tutorials/introduction/001_chapter1.html
http://openframeworks.cc/tutorials/first%20steps/002_openFrameworks_for_processing_users.html

Structure of an OF program

Assuming you have OF correctly setup on your machine with Xcode installed, you are now ready to get your hands dirty building an OF program.

Create a new OF/Xcode project using the project generator app (found in main-OF-folder/projectGenerator/projectGenerator). Name your project 'waveDistortion' and select ofxGui in the addons section (since we will be using this library later on).

Tip: It is much easier to add addons using the project generator (when you create a new project), then to add them manually later, although there is now a plugin you can download that facilitates this task. If you need to add addons later in a project, create a new project (using the project generator) and copy/paste your code from your current project.

Unless you selected a different path in the project generator, your OF/Xcode project will now appear in main-OF-folder/apps/myApps/your-new-project-folder/. You will see that there is an Xcode project file packages together with various other files and two folders: the src folder is where your code lives; the bin/data folder is where you should place images, videos and sounds that you want to want to use.

When you open the Xcode project file you will see the files structure in the left hand column, which includes all of the files in your OF/Xcode project folder and all of the linked libraries (both system libraries and OF libraries) that allow your program to function and do and all sort of magical things. Navigate to the files in the src folder like so:

navigate to src files

Click on each of these files to view their contents. You will see that a template has been created for you: a main function (the starting point of every c++ program) and and a class called testApp with method declarations. main() sets up the windowing system (allowing you to draw things to the screen), and launches your program. testApp is equivalent to the blank window you get in the Processing IDE when you open up a new project (this is also a class, only it is hidden from you). Like all classes in OF projects, testApp is composed of a header file (with a .h extension) and a c++ file (with a .cpp extension). The header file is where the class, together with its methods and instance variables, is declared; the c++ file is where the methods are defined. There are a few good reasons for this separation of code, which we might refer to as 'interface : implementation', but this is outside the scope of this tutorial. Two methods that will be familiar to Processing coders are setup() and draw(). There is a third one called update(). setup() gets called once at the beginning of the program; update() and draw() get called repeatedly in an infinite loop until the program terminates. As the names suggest, update() is where variables should be updated, and draw() should be reserved for drawing objets to the screen. Try running the program by clicking on the play button in the top-left hand corner; if you see an empty grey window then you are good to go. This window is actually your OF application: it can be launched directly from the bin folder and, like all applications, it must be quit using cmd+Q (on a Mac).

Fun part

Ok, so now that you have a basic understanding of the structure of an OF program, we can actually start coding, yay!

In the testApp.h file, declare two ofImage objects after the method declarations.

```
#pragma once

#include "ofMain.h"
```

```
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);


ofImage image1; //Original image
ofImage image2; //Modified image
};
```

In the testApp.cpp file, load the image file into the ofImage object in setup().

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

//Load image
image1.loadImage( "Sylvie_Pic.jpg" );
}
```

Draw the image to the screen call the draw method of the ofImage object starting in the top-left hand corner (with the width and height of the original image by default). OfSetColor() should always be called before drawing images because it affects the color of all subsequent drawing operations, so to avoid accidentally modifying the color of object down the line it is best to call this function every time you draw an image. ofBackgroundGradient (unsurprisingly) sets the background color of the drawing context as a gradient.

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

//set background color
ofBackgroundGradient(ofColor::white, ofColor::

// draw image
ofSetColor( 255, 255, 255);
image1.draw(0, 0);
}
```

Awesome, you're done! Go and celebrate…

…Ok, the thrill has worn off and you don't feel like you've mastered OF quite yet. Let's make our image move. Create a function that processes the original image on a pixel by pixel basis and returns a new image. First declare it in the testApp.h file (I called it wave) and give it an ofImage object as an argument.

```
ofImage wave(ofImage sourceImg);
```

Then define it in the testApp.cpp file. It can be placed anywhere – I put it between draw() and keyPressed().

```
//--------------------------------------------------
ofImage testApp::wave(ofImage sourceImg){

//create a clone of the source image (cloning a
ofImage newImg;
newImg.clone(sourceImg);

// returns how much time has elapsed since the
float time = ofGetElapsedTimef();

// iterate through every pixel in the image1 ar
for (int y=0; y<image1.height; y++) {
for (int x=0; x<image1.width; x++) {

// calculate wave distortion
float noise = ofNoise(time + y * 0.001) * noise
float noiseAmp = noise * amplitude; // scale th
float waveform = sin((y * wavelength) + time *
float wave = (waveform + distortion) * noiseAmp

//xWave represents the distorted pixel locatior
int xWave = x + wave;

//Set newImg pixel color (at location x,y) equa
ofColor color = sourceImg.getColor( xWave, y );
newImg.setColor( x, y, color );
}
}

newImg.update();
return newImg;
}
```

Clearly there are a few things going on here, but don't worry, we will walk through the code one step at a time. First however, to see the animation we need to add some local variables – you should put these in the testApp.h file under the keyword 'private'. (Public variables are available to other classes, whereas private variable are only available within the class).

```
ofImage image1; //Original image
ofImage image2; //Modified image

ofImage wave(ofImage sourceImg);

private:

float wavelength = 0.05;
float amplitude = 40.0;
float noiseScale = 8.0;
float distortion = 25;
float ySpeed = 3.0;
```

Call the function in update(), assigning the returned value (an ofImage object) to image2.

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


// image2 is now the result of the processed in
image2 = fuzz(image1);
}
```

Then replace image1 with image2 in draw().

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

```
//set background color
ofBackgroundGradient(ofColor::white, ofColor::g

// draw image
ofSetColor( 255, 255, 255);
image2.draw(0, 0);
}
```

Now if you run the program you should see the result. Not bad eh!

So what's going on in the wave function? First we clone the source image (that we pass in as an argument), assigning the value to a local ofImage object called newImg, the function's return value. We store the current 'time' of the program using ofGetElapsedTimef() (similar to millis() in Processing) in a local variable called time, which we later use to 'drive' the sin wave forward. We use a nested for loop to iterate through every pixel in the image1 pixel array. We use the distortion value (wave + x) and use this to get the colour of a pixel further along in the array, which we assign to newImg's x,y, location. Because newImg is a clone of sourceImg, it has the same dimensions as sourceImg, i.e. the pixel array is the same length, meaning we will not go 'out of bounds' and get the dreaded 'null pointer exception'. NewImg therefore represents the distorted version of image1, which we assign to image2 in update().

You may be wondering about the distortion code:

```
// calculate wave distortion
float noise = ofNoise(time + y * 0.001) * noise
float noiseAmp = noise * amplitude; // scale th
float waveform = sin((y * wavelength) + time *
float wave = (waveform + distortion) * noiseAmp
```

This is where the magic happens, but I'm not going to go into it because if you don't understand it then you need to learn and experiment with trigonometry functions such as sin() and the noise function ofNoise(). Hopefully the comments help you make sense of it. I encourage you to play around with it as it is quite sensitive to change (small changes will generally have a large impact).

The final step will be to add gui elements so that we can have control parameters and change the wave distortion values during runtime. This is really easy with the ofxGui library. (I am assuming you added this library when you created the project; if not, refer to the Structure of an OF program for guidance). First we have to link the ofxGui library by using the preprocessor directive #include at the top of the testApp.h file.

```
#include "ofMain.h"
#include "ofxGui.h"
```

Remove the private variables that we created earlier and replace them with these ofxGui slider objects at the bottom of the testApp.h file

```
ofxPanel gui;
ofxFloatSlider wavelength, amplitude, noiseScal
```

Then initialize them in setup().

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

```
//setup UI elements
gui.setup();

//setup each gui element: args = name, default,
gui.add(wavelength.setup( "wavelength", 0.125,
gui.add(amplitude.setup( "amplitude", 0, 0, 50
gui.add(noiseScale.setup( "noiseScale", 0, 0, 1
gui.add(distortion.setup( "distortion", 0, 0, 5
gui.add(ySpeed.setup( "ySpeed", 0, 0, 10 ));

//Load image
image1.loadImage( "Sylvie_Pic.jpg" );




}
```

These gui objects will now control the values of our wave distortion variables. The last step is to draw the ofxPanel object called gui that will contain the gui sliders.

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

//set background color
ofBackgroundGradient(ofColor::white, ofColor::g

// draw image
ofSetColor( 255, 255, 255);
image2.draw(0, 0);

gui.draw();
}
```

You can move this gui panel around and play with the slider values to see variations of this wave distortion effect.

Congratulations, you are now an OF hero and have graduated to level II: openFrameworks tutorial, part II: Vertex Mapping.