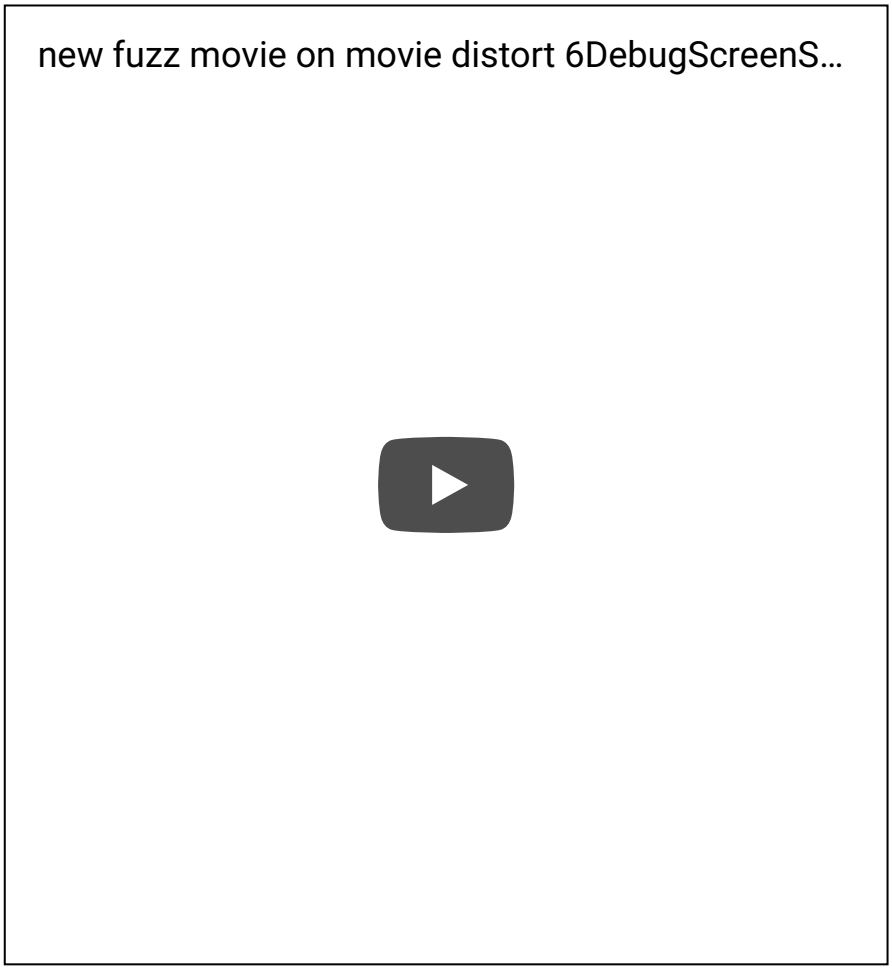
openFrameworks tutorial, part II: Texture Mapping

In part II of this tutorial we will explore mapping a regular 2D image to a vertex mesh, transforming it into a 3D object. This tutorial is more advanced than part I and assumes you already have OF (openframeworks) installed on your machine with an IDE and are familiar with with basic object oriented programming concepts. The result of this tutorial can be seen in this video:



And the source code can be downloaded here.

Fun part

That's the great thing about part II's, you skip right to the fun part! This where we get to play with code and learn cool tricks.

Instead of using a still image, this time we are going to use a video file. The process is very similar to using a regular ofImage, it just require a few more steps. First declare an ofVideoPlayer object in your header file.

```
ofVideoPlayer video; //Prerecorded video
```

Then load the video into the object using loadMovie(). Remember the video must reside in your bin/data folder. call the play() method to get things going.

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

video.loadMovie( "polar bear.mov" ); //Load the
video.play(); //Start the video
}
```

Call update() in update() and draw() in draw(). Simple isn't it?

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

//update video
```

```
video.update();
}

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

//Set background to black and rendering color t
ofBackground(0);
ofSetHexColor(0xffffff);

video.draw(0, 0);
}
```

The video is now drawn to the screen and loops automatically. Go make yourself a pina colada, sit back and enjoy the movie 😉

Now we're going to create and FBO (Frame Buffer Object) that we will use to create a fade-out effect. Look at ofFbo in the OF online reference for a description of this wondrous little object. While you're there, have a look at the class methods in the left-hand column to see what it can do. The OF reference is a great place to see what classes / methods are available in OF, although unfortunately there is not nearly as much descriptive information as Processing's reference. Hopefully some noble, philanthropic contributor will help fill in some of the gaps in the future (sacrificing her weekends for the greater good of the OF community). Welcome to the world of Open Source!

Ok, back to work, let's setup the FBO. First declare it in the header file, together with two private variables: videoWidth and videoHeight.

```
ofVideoPlayer video;
ofFbo fbo;

private:
```

```
int videoWidth = 720;
int videoHeight = 480;
```

Then allocate memory for it in setup() and clear the junk that resides in it (if you're into glitch art, then don't clear it). begin() and end() do what you think they do: everything that happens in between affects the internal state of the FBO.

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

video.loadMovie( "polar bear.mov" ); //Load the
video.play(); //Start the video

//setup fbo
fbo.allocate(videoWidth, videoHeight);
// clear fbo
fbo.begin();
ofClear(255,255,255, 0);
fbo.end();
}
```

Notice that we set the alpha component to 0. This is because we are going to draw our video into it with a low alpha without clearing the background, resulting in a fade-out effect. in setup() we call the method isFrameNew() which checks if the video is offering us a new frame. This avoids discrepancies between the video's framerate and the program's framerate (which may be much higher, or indeed much lower). This avoids unnecessary computation and increases performance.

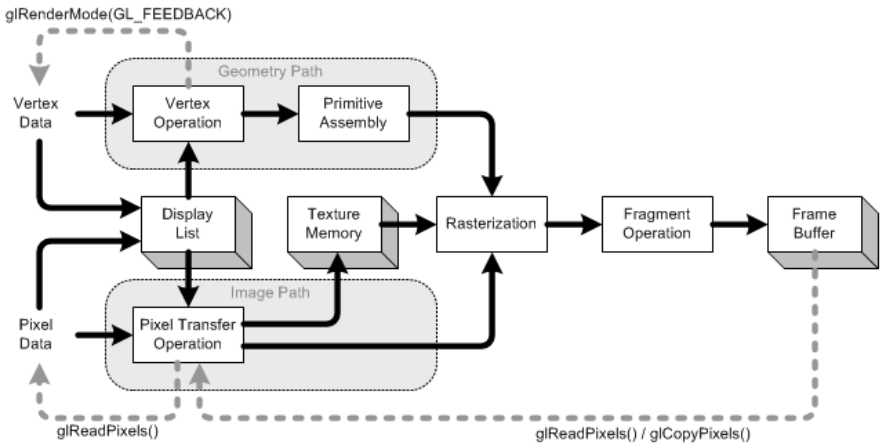
```
if (video.isFrameNew()) {
// draw video into fbo
fbo.begin();

int alpha = 20; // amount of smoothing
ofEnableAlphaBlending();
ofSetColor(255, 255, 255, alpha);

// draw image into fbo
video.draw(0, 0);
ofDisableAlphaBlending();
fbo.end();
}
```

As you may have noticed, the video is drawn into the FBO, which will then itself be drawn in draw(). You know what to do: replace video.draw() with fbo.draw(0, 0) and run the program to see the result.

Now we will create a vertex mesh that we will use to map our video onto. Vertices are points in 3 dimensional space that can be used to create 3D objects that are processed on the graphics card. This is called the rendering pipeline and involves several stages of operations including primitive assembly, where the vertices get connected into triangles, and rasterization, where 3D scenes get converted into pixel data. This is a good illustration of the process:



The descriptions of each stage of the process are worth reading, even if not all of it makes complete sense. This process is handled by OpenGl, one of the core OF libraries that allows us to draw graphics to the screen (kind of important!). OF provides a thin wrapper for OpenGl, meaning it simplifies the process while not completely obscuring it. And it is worth noting that it is possible to make native OpenGl function calls directly from you OF program (which is also true of all the other core libraries).

Have a look at the reference for ofImage; you will see that this object packages a pixel array and a texture object, and that these can be converted either way using the appropriate methods. As explained in the descriptions of the stages of the rendering pipeline, textures can be loaded into texture memory and applied onto geometric objects, such as 3D meshes (made from an array of vertices). That is exactly what we are going to do. We are also going to map the z-position of the vertex array to the brightness of the video pixels, creating a distortion in the bright regions of the video.

Ok, theory aside, here are the necessary steps to achieve this. First create an ofMesh object called mesh and add the local variables W, H and meshSize.

```
ofVideoPlayer video; //Prerecorded video
ofFbo fbo;
ofMesh mesh;

private:

int videoWidth = 720;
int videoHeight = 480;
int W = 100; //Grid size
int H = 100;
int meshSize = 6;
```

Next, set up the vertices and their indices in setup() using our local variables. This part of the code is largely copied from the section called *Using ofMesh* in the excellent book mastering-openframeworks-creative-coding-demystified. There you will find a much better explanation of this code than I could offer, and in any case it is mostly outside the scope of this tutorial. Buy the book, you won't regret it! Basically We are mapping vertices to texture coordinates – notice how mesh.addVertex() and mesh.addTexCoord() are called one after the other. We are only using 10 000 vertices (100 x 100) but we have 345 600 pixels (720 x 480), so we cannot map every pixel in the video. Instead we map every 34th or 35th pixel (or 'texture coordinate') and the graphics card will interpolate between these values to give us a smooth image.

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

video.loadMovie( "polar bear.mov" ); //Load the
video.play(); //Start the video

//setup fbo
fbo.allocate(videoWidth, videoHeight);
// clear fbo
fbo.begin();
ofClear(255,255,255, 0);
fbo.end();

//Set up vertices
for (int y=0; y<H; y++) {
for (int x=0; x<W; x++) {
mesh.addVertex(ofPoint((x - W/2) * meshSize, (
mesh.addTexCoord(ofPoint(x * (videoWidth / W),
mesh.addColor(ofColor(255, 255, 255));
}
}
```

```
//Set up triangles' indices
for (int y=0; y<H-1; y++) {
for (int x=0; x<W-1; x++) {
int i1 = x + W * y;
int i2 = x+1 + W * y;
int i3 = x + W * (y+1);
int i4 = x+1 + W * (y+1);
mesh.addTriangle( i1, i2, i3 );
mesh.addTriangle( i2, i4, i3 );
}
}
}
```

Draw the mesh's wireframe in draw() to see what it looks like. Because we are now using the OpenGl coordinate system, it is necessary to frist shift the 0,0 coordinates from the top-left corner into the centre of the screen using push/popmartix() and translate().

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

//Set background to black and rendering color t
ofBackground(0);
ofSetHexColor(0xffffff);


fbo.draw(0, 0);

ofPushMatrix(); //Store the coordinate system

//Move the coordinate center to screen's center
ofTranslate( ofGetWidth()/2, ofGetHeight()/2, (
mesh.drawWireframe();
ofPopMatrix(); //Restore the coordinate system
}
```

Now see what happens when you apply noise to the mesh. Add this code to update().

```
//Change vertices
for (int y=0; y<H; y++) {
for (int x=0; x<W; x++) {

//Vertex index
int i = x + W * y;
ofPoint p = mesh.getVertex( i );

//Change z-coordinate of vertex
p.z = ofNoise(x * 0.05, y * 0.05, ofGetElapsedT
mesh.setVertex( i, p );

//Change color of vertex
mesh.setColor(i , ofColor(255, 255, 255));
}
}
```

We get the current vertex from the vertex array using the standard expression i = x + width * y (because all arrays are actually one-dimensional in memory), and map its z position to perlin noise. Run the program to see the result. Pretty cool, eh.

Now, we are going to connect the video with the mesh. We are going to use the same nested for loop we just created for our vertex array to iterate through the FBO pixel array and map the brightness values to the z position of the vertex (instead of perlin noise). Create two objects in the header file.

```
ofImage image;
ofPixels fboPixels;
```

Use these objects to convert an FBO to ofPixel (that we can use to access pixel information), and then again to ofImage (that we will use later on in draw() to bind the video to the mesh.

```
//convert fbo to ofImage format
fbo.readToPixels(fboPixels);
image.setFromPixels(fboPixels);
```

Create a new index variable for our new pixel array object (using the same expression i = x + width * y) and scale in proportion to the size of our video. Multiply by 4 because the FBO contains four components: Red, Green, Blue and Alpha. Then extract the brightness value and map to the z position of the corresponding vertex.

```
//Change vertices
for (int y=0; y<H; y++) {
for (int x=0; x<W; x++) {

//Vertex index
int i = x + W * y;
ofPoint p = mesh.getVertex( i );

float scaleX = videoWidth / W;
float scaleY = videoHeight / H;

// get brightness
int index = ((x * scaleX) + videoWidth * (y * s
```

```
int brightness = fboPixels[index] / 4; // 4 is

//Change z-coordinate of vertex
p.z = brightness; // ofNoise(x * 0.05, y * 0.05
mesh.setVertex( i, p );

//Change color of vertex
mesh.setColor(i , ofColor(255, 255, 255));
}
}
```

Run the program and see how the mesh has taken on the form of the video. Don't get mesmerized, you're not finished yet! The last important thing to do is to bind the video to the mesh. Do this in draw() and comment out the FBO and Wireframe draw() methods.

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

//Set background to black and rendering color t
ofBackground(0);
ofSetHexColor(0xffffff);

// fbo.draw(0, 0);

ofPushMatrix(); //Store the coordinate system

//Move the coordinate center to screen's center
ofTranslate( ofGetWidth()/2, ofGetHeight()/2, 0

//Draw mesh
image.bind();
mesh.draw();
image.unbind();
// mesh.drawWireframe();
```

```
ofPopMatrix(); //Restore the coordinate system
}
```

There you have it, your video is not mapped to your mesh. Awesome. However, to make it 100x more awesome, add controls to transform the perspective and view your 3D video from different angles. Add these local variables to your header file.

```
float tiltCurrent = 0;
float tiltTarget = 0;
float turnCurrent = 1;
float turnTarget = 1;
```

Add these lines to draw(), inside the push and pop matrix calls after translate().

```
tiltCurrent = ofLerp(tiltCurrent, tiltTarget, (
turnCurrent = ofLerp(turnCurrent, turnTarget, (
ofRotateX(tiltCurrent);
ofRotateZ(turnCurrent);
```

And add these lines to your keyPressed() stub. If you're wondering what ofLerp() does, it interpolates between two values (in this case tiltCurrent and tiltTarget) at a certain speed. This gives us a smooth motion when changing the viewing angle with the up, down, left and right keys.

```
//
```

```
//--------------------------------------------
void testApp::keyPressed(int key){

if (key == OF_KEY_DOWN){
tiltTarget -= 5;
} else if (key == OF_KEY_UP){
tiltTarget += 5;
} else if (key == OF_KEY_LEFT){
turnTarget -= 5;
} else if (key == OF_KEY_RIGHT){
turnTarget += 5;
}



}
```

Now you are an OF ninja! What I suggest it to add GUI controls, so that you can play with alpha in the FBO, change the scale of brightness and and change the meshSize variable during runtime. You might also do any of 1000 other things that you can think of, such as experimenting with different input sources or combining input sources or, or, or…