Mesh Morphing

An interactive 3D scene with a morphing sphere.

Introduction

I made this sketch to create some visuals to use in a live performance. This sketch is audio reactive and has some parameters which are accessible from a graphic user interface and, in a future release from a MIDI controller. I decided to use a configuration file to set some aspects of my sketch that could vary from scenario to scenario, so for example at this time I could change the dimensions of the window, set it fullscreen, get the application to run in eco mode and so on. No one likes laggy visuals, so i decided to write my own shaders to achieve greater performance and scalability.

Description

This sketch involves a 3D Scene with a spherical shape as protagonist. The shape appearance is constantly changing due to a gradual displacement of its vertexes. This is done by plugging every vertex of the shape into a *perlin noise* function which will use the vertex coordinates to output a number between 0.0 and 1.0. These values will be used to push all the vertexes from their original position by some amount, therefore morphing the sphere into a less perfect, less boring shape.

This algorithm could be implemented in 2 ways:

- Using *the processing way* through the PShape class methods
- Modifying the rendering behavior through a **Vertex Shader**

In this case i preferred to go for the second approach. That's because a custom shader allow the program to perform all the vertex displacement calculations on the GPU, resulting in much higher performance. In addition to that I have found no way to access the vertexes of a PShape created like this `new PShape(SPHERE, x)`, so i would be stuck with having to create all my 3D shapes manually which could be time expensive and could result in less scalable code.

Explaining openGL is well beyond the scope of this paper, so i'll leave some useful links for whomever would like to look into the topic with more depth. To give a glimpse of what a vertex shaders does we could say that they are small programs that have to put together vertexes to construct some geometries. These geometries will then be the source of other calculations that will eventually result in a picture of the scene created.

```
attribute vec4 position;

uniform mat4 transformMatrix;
uniform float u_noise_amount;
uniform float u_time;

varying vec4 fragPosition;

float noise(vec3 st) {...}

void main() {
 float displacement = u_noise_amount * (noise(position.xyz + u_time) - 0.5);

 gl_Position = transformMatrix * position + vec4(displacement);

 fragPosition = normalize(position + displacement);
}
```

Anyhow, here above is a simplified version of my vertex shader code; The language is called **GLSL**, which stands for OpenGL Shading Language. When you create a PShape and you place it on the canvas through `shape()`, this shader will be applied to all vertexes of the shape. So Processing will send the coordinates of a specific vertex through the `position` attribute. At the same time it will send some other parameters that are the same throughout all vertexes: these variables are referred to as uniforms and in this case are `transformMatrix`, `u_noise_amount` and `u_time`. *transformMatrix* is passed automatically by processing , on the contrary *u_noise_amount* and *u_time* are variables that i manually send from my *Mesh* class through the PShape's set method.

The main function is where calculations are done, and their result is stored in the `gl_Position` variable. Here I defined a noise function and fed it with the vertex position coordinates. The result of the noise is then multiplied by the uniform *u_noise_amount*, summed to the original position and stored in *gl_Position*.

You should have noticed that i also set a variable `fragPosition` which is defined as a varying vector. Well this is because i decided to use the morphing effect to be also applied to the shape's texture, so by defining that variable as a varying type i exposed it to the **fragment shader**, which is another GLSL file that cooperates with the vertex shader to render color.

```
varying vec4 fragPosition;

uniform float u_time;
uniform float u_frag_noise_amount;
uniform float u_frag_noise_density;

float noise(vec3 st) {...}

void main() {
  float noise_density = 0.1 + 10. * u_frag_noise_density;
  float noise_amount = 1.0 - 0.8 * u_frag_noise_amount;
  float noise = noise(fragPosition.xyz * noise_freq + u_time * 0.01);

  vec4 colorA = vec4(0.0, 0.0, 1.0, 1.);
  vec4 colorB = vec4(1.0) - colorA;
  vec4 color  = mix(colorA, colorB, smoothstep(0.0, noise_amount, noise));

  gl_FragColor = vec4(color.rgb, 1.0);
}
```

This one above is the code for the fragment shader i mentioned earlier. You should notice that i defined a variable called `fragPosition` which has the same name of the one in the vertex shader and it's also of varying type. These are the required conditions to share values between fragment and vertex shaders. I also included some uniforms to be able to tweak the shader from the outside in, as i did with the vertex shader. So in the main function of a fragment shader a color should be calculated and stored in the `gl_FragColor`. In this case that's done by plugging the fragPosition coordinates into, again, a noise function, then its result is stored and used to mix two arbitrary colors, defined as `colorA` and `colorB`. To mix the colors i used the glsl *mix* function which performs linear interpolation of color values. The linearity of the mix function is distorted through another glsl function called *smoothstep* which implements Hermite polynomial interpolation, this one will use the u_noise_amount uniform to define the steepness of its slope.

Conclusions

I think that the structure of this sketch is strong, so there is plenty of space for new features in the future.
Here are some of them:

- Color slider to change the colors of the textures
- Midi support with custom mapping through the settings.json file
- Video Export and Screenshot support
- Drop controlP5 deprecated methods
- Fix issue #1

Requirements

- Processing 3.5.x
- Processing Sound
- PeasyCam
- ControlP5

Installation

Open a terminal and cd into the path you want this sketch to be located.

```
cd your-path-of-choice
git clone https://github.com/lorenzorivosecchi/mesh_morphing.git
```

Make sure you have installed all the libraries listed above, if you don't just add them using the Contribution Manager.

Usage

Launch sketch by double clicking any `.pde` file in the directory.
Use the graphic user interface to change some parameters of the scene.
Check out the on screen console for tips and shortcuts

Reference

- Shaders - Andrea Colubri
- The Book of Shaders
- Shaderific
- controlP5

Developed by Lorenzo Rivosecchi in June 2019 https://github.com/lorenzorivosecchi/mesh_morphing