Graphics and Game Technology

Assignment 5

Vertex and fragment shaders

1 Introduction

Modern Graphics Processing Units (GPU's) are programmable multiprocessors that can run "shaders" to compute not only light shading, but all sorts of effects in computer graphics. In this assignment we will use shaders with various techniques to create the scene shown in Figure 1.

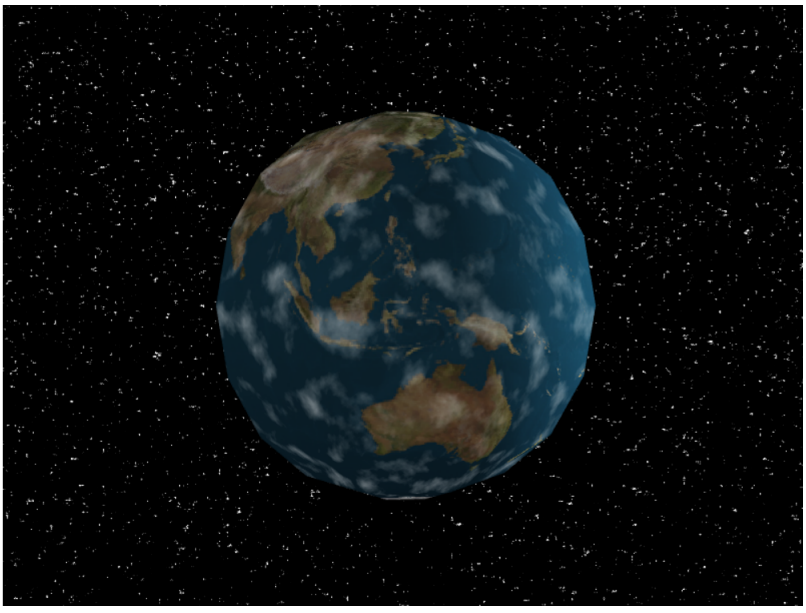

Figure 1: The final result of this assignment.

This assignment makes use of JavaScript and WebGL. WebGL is a subset of OpenGL (specifically "OpenGL ES") which is very easy to prototype with. We will start with the basic framework and a main file. The framework encapsulates most of the WebGL calls you need in an easy to use interface. At some point

you *will* need more information on the specifics of WebGL and the OpenGL Shading Language (GLSL). Use the following resources for that:

- WebGL API: `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`,
- Documentation for WebGL/OpenGL and GLSL functions: `http://docs.gl/`.

1.1 Getting started

Unpack the framework. In a terminal, go into the framework directory and run `./server.sh`. This will start a local web server [1]. You can then view the result of the framework by pointing a browser to `http://localhost:8080` where you should see a white rotating triangle (see Figure 2). Also open your browser's developer tools (F12 on most browsers). Errors with the program and shaders will be shown in the console.

For this assignment you will only need to modify `main.js` and the files in the `shaders/` folder.

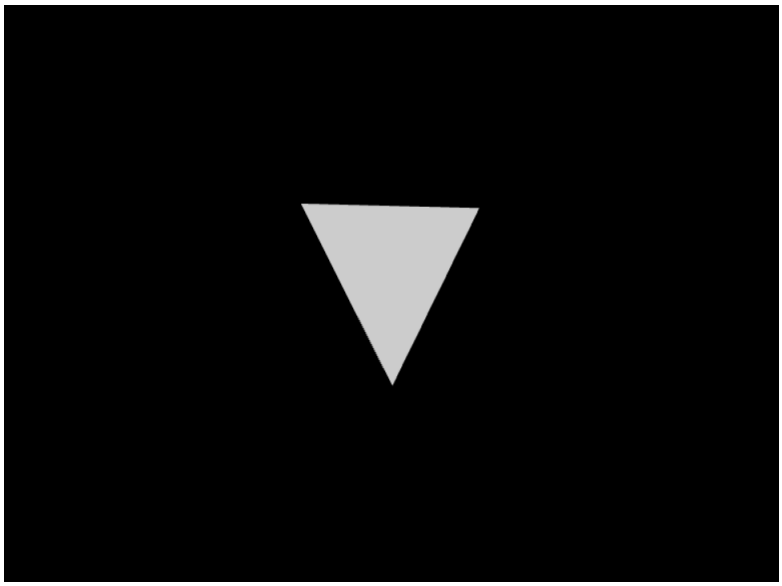

Figure 2: The framework produces a rotating triangle.

1.2 Overview of WebGL

Shaders can do more than what their name suggests. They are C-like programs that transform buffer data to pixel data in a programmable way that runs on

[1] You need to have Python version 3.x installed on your system to run this server.

GPUs. Buffers are arrays of data that are uploaded to the GPU. The framework defines a `Buffer` class as a wrapper to the WebGL interface. The only type of buffers we will use are those with `float` data.

1.2.1 Shader programs

There are always two source files per shader program: the vertex shader and the fragment shader. These source files are written in GLSL, a C-like programming language. The vertex shader takes the buffer data passed from WebGL as input. The input data is passed as *attribute* data, as defined by the attribute setup calls (described later). The data is then converted to screen coordinates using matrices and other calculations made in the vertex shader program. The fragment shader takes the vertex coordinates produced by the vertex shader and rasterizes the pixels between them. The fragment shader program then calculates the pixel colours on the screen, mostly by interpolation.

Shader programs run on the GPU. To pass data from your CPU code into the shader program and from the vertex to the fragment shader, the following qualifiers are used in the GLSL shader code:

attribute : this specifies the input for the vertex shader; this is where the data will reside that comes from the buffers created in the main CPU program.

uniform : this specifies data that is only changed between render calls, such as matrix data to specify a projection or a transformation; the data is updated from the CPU program with `program.setUniform*Data()`

varying : this specifies the output of the vertex shader and input to the fragment shader; you need matching definitions to communicate data from the vertex to the fragment shader. Note that the fragment shader linearly interpolates pixel values between the three calculated points of a triangle.

Both shader programs must first be compiled. This is done in `main.js` using the `Program` class which takes the paths to the source code of the vertex and the fragment shader and returns a `program` object that we later use to specify the input data:

```
program = await new Program().init('shaders/sphere.vs.glsl',
                                   'shaders/sphere.fs.glsl');
```

The shader programs are immediately checked. Any errors will be shown in the console of the browser.

1.2.2 Specifying input data: attributes

As mentioned earlier, attributes are the inputs to our vertex shader. To tell WebGL how to get to that data, we must call

```
program.attribute(bufferObj, attributeName, size, stride, offset);
```

where:

- `bufferObj` specifies the buffer that contains the input data; this is just a linear array with (float) values;
- `attributeName` specifies the name of the attribute used in the vertex shader;
- `size` specifies the number of elements for the attribute; for example 3 for a three dimensional vector;
- `stride` is the total number of elements per data point; for example 6 if we have a three dimensional position vector and a three dimensional normal vector per data point.
- `offset` is the starting offset of the attribute in the buffer data.

Take a look at `main.js` and see how this works for the triangle: the array contains 24 float values for three vertices, where each vertex is specified by eight values: three values for position, three for a normal vector and two for texture coordinates. Before you continue, make sure you understand the code that tells the program how these different attributes can be found in the array.

1.2.3 State binding

You may create as many shader programs and buffers as you like. However; at any point in time, only one set of shader programs can be "active". A similar restriction holds for buffers which need to be "bound" in order to be used by a shader program. Think of WebGL as a state machine that switches between a large collection of shader programs and buffers while a program is running. Modifying the state is done with different API calls, like `program.use()` to set the current shader and `buffer.bind()` to enable the use of a buffer. Both methods must be called before the objects may be used by other methods.

2 Creating a planet

Your first challenge for this assignment is to modify the program so that it renders a sphere instead of a triangle. In `main.js`, create a function `createSphere` that produces a buffer that contains the floating point values that represent the triangular patches that together make up a sphere. See Figure 3 for inspiration and section 2.5.8 ("3D Parametric Surfaces") of the FCG book on a method to do this. Center the sphere around (0.0, 0.0, 0.0) and give it a radius of 2.0. The normal vector of each point must point outwards and perpendicular to the sphere's surface so that the lighting calculations we add in the next section can be correctly applied. The coordinates for the texture (often refered to as u and v) must range from 0.0 to 1.0 for the horizontal and vertical axis, so we can later

apply a texture to the sphere. See also the section on "Spherical Coordinates" in section 11.2.1 of the FCG book.

It may be useful to create a helper function that creates the array data for a point in space (including position, normal and texture vectors) and a function that combines four of these points into a plane. Split each plane into two triangles to obtain a buffer that contains only triangles.

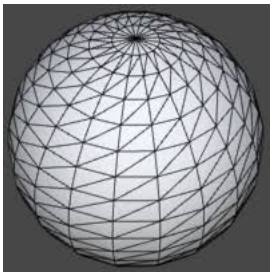

Figure 3: A sphere made out of triangles.

3 Gouraud and Phong shading

Now it's time to implement lighting. We are first going to implement it in the vertex shader.

Modify the vertex shader in `shaders/sphere.vs.glsl` to first implement Gouraud shading. Use the calculated position and normal vector that are already given in the framework's vertex shader. See the lecture slides for the calculation of the lighting using the ambient, diffuse and specular parameters. The result should look as shown in Figure 4. Notice the banding around the sides of the triangle patches that make up the sphere. When calculating the lighting in the vertex shader, we are only calculating the light properties at the vertices that make up the sphere. The fragment shader linearly interpolates the values between these points.

To improve this, move the light calculation from the vertex to the fragment shader in `shaders/sphere.fs.glsl`, thereby effectively creating Phong shading. To do this, use `varying` declarations to pass the position and normal data from the vertex to the fragment shader and have it calculate the lighting per pixel using the interpolated values of these.

4 Texturing

Next we will add a texture of the Earth to the sphere. First, use the `Texture` class of the framework to load the image:

```
// in callbacks.setup
sphereTexture = await new Texture().init('images/earthmap1k.jpg');
```

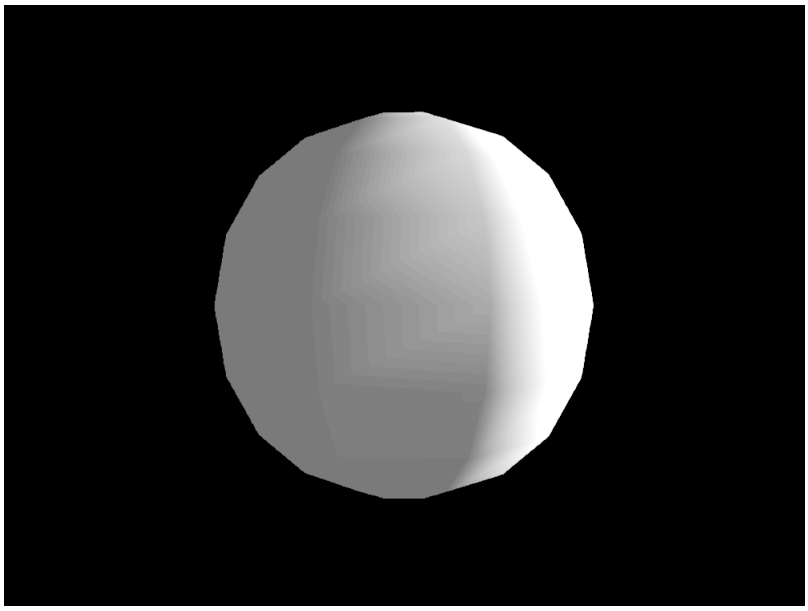

Figure 4: A sphere with lighting calculated in the vertex shader (Gouraud shading). Notice the banding where the triangles connect (slightly exaggerated in this image for illustration).

To use this image as a texture, we have to use a "sampler". First, bind the texture as a `uniform`:

```
program.setUniformSamplerData("uSampler", sphereTexture);
```

Then, in the fragment shader declare the `uniform`:

```
// Declare the sampler uniform as a global:
uniform sampler2D uSampler;
```

To use it, we use the `texture2D` function of GLSL:

```
// In main():
vec4 color = texture2D(uSampler, vTexture);
```

The first argument is the `uniform` we just declared, the second a `vec2` whose value we use as a position in the image. The coordinates go from 0.0 to 1.0, left to right and top to bottom.

If you implemented this correctly, the sphere should now be textured with an image of the Earth.

5 Procedural textures in the shader

For our next challenge, we are going to use the shader to procedurally generate the effect of clouds moving over the surface of the Earth. Our clouds would not look very realistic if we created them from predefined triangle meshes or images, so we'll generate them randomly. For that, we need a way to get random values in the shader. Unfortunately, there are no built-in ways to generate random values, so we have to use something else. Read through chapter 10 and 11 of the book of shaders `https://thebookofshaders.com/10/` / `https://thebookofshaders.com/11/` to get an idea how random values in the shaders would work. Start by using the noise function described in chapter "2D":

```
// 2D Random
float random (in vec2 st) {
  return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) * 43758.5453123);
}

// 2D Noise based on Morgan McGuire @morgan3d
// https://www.shadertoy.com/view/4dS3Wd
float noise (in vec2 st) {
  vec2 i = floor(st);
  vec2 f = fract(st);

  // Four corners in 2D of a tile
  float a = random(i);
  float b = random(i + vec2(1.0, 0.0));
  float c = random(i + vec2(0.0, 1.0));
  float d = random(i + vec2(1.0, 1.0));

  // Smooth Interpolation
  vec2 u = smoothstep(0.,1.,f);

  // Mix 4 corner percentages
  return mix(a, b, u.x) +
  (c - a)* u.y * (1.0 - u.x) +
  (d - b) * u.x * u.y;
}
```

The `random` function accepts a `vec2` as a seed and uses random vectors and a combination of the `dot`, `sin` and `fract` functions to create a pseudorandom value. It is best to not change the constant values. The `noise` function uses the `random` function to create four corners with random values and interpolates between them with `smoothstep`.

Once you have a noise function, you can use the following function to add multiple layers of noise on top of each other with different scaling levels:

```
float noise_with_octaves(vec2 pos) {
```

```
  float value = 0.0;
  float c = 0.5;
  vec2 shift = vec2(100.0);

  for (int i = 0; i < 6; ++i) {
    value += c * noise(pos);
    pos = pos * 2.0 + shift;
    c *= 0.5;
  }
  return value;
}
```

When you use the u and v texture coordinates calculated for the sphere, be sure to scale the vertical axis by 0.5 to avoid a squashed noise output, as the horizontal axis is twice the length of the vertical axis for the sphere.

This is just one way to use a random number generator to generate cloud-like structures. There are many others. For example; this noise output may look better using gradient noise (also known as "Perlin" noise). Gradient noise uses gradient vectors instead of float values to interpolate between the corners. Feel free to use any other noise function, as long as you make your sources clear.

6 Adding stars to the background

For our final challenge we are going to add stars to the background of our planet. Create a new `Program` with a pair of shader programs called `starsProgram`. Create a new buffer, `starsBuffer`, that contains the coordinates of two triangles that together form a plane going from -1.0 to 1.0 horizontally and vertically. To place this plane just behind the sphere, taking up the whole viewport, use the following vertex shader code:

```
attribute vec2 aPosition; // binds to starsBuffer

varying vec2 vPos;

void main(void) {
  gl_Position = vec4(aPosition.x, aPosition.y, 0.9999, 1.0);
  vPos = aPosition.xy;
}
```

Note that this code also passes the coordinates of the plane to the fragment shader in `vPos`.

In the fragment shader, create a random number generator (see above) to generate white colours in random fragments. To make the stars appear at the same location each frame, you should use a constant seed value, e.g. `vPos`. Make the stars animate from left to right to create the illusion that the planet is slowly revolving (hint: pass the frame counter in `main.js` as a uniform to the fragment shader).

7 Grading

The grading for this assignment is as follows:

- **5 points** for a correct implementation of an algorithm that generates a sphere.
- **4 points** for correctly implementing Phong shading.
- **3 points** for correctly implementing texturing.
- **5 points** for correctly implementing procedurally generated clouds.
- **3 points** for correctly implementing the animated stars in the background.
- Again: points may be deducted if your code is not clearly formatted or commented!