Interactive Graphics Homework 1

Michal Ostyk-Narbutt (1854051)
Prof. Marco Schaerf

May 6, 2020

SAPIENZA
UNIVERSITÀ DI ROMA

1 Introduction

This is a documentation report describing the techniques used in the First Homework for the Interactive Graphics course. Given a baseline file of a cube, the task was to modify it by

- expanding the number of vertices (20-30) with each having a normal and texture coordinates.
- adding a viewer position and a a perspective projection
- computing the ModelView and Projection matrices in the Javascript application
- adding two lights sources: Directional and a Spotlight
- assigning to the object a material with the relevant properties
- Implementing a per-fragment shading model
- adding a texture loaded from file, with the pixel color a combination of the color computed using the lighting model and the texture.

2 Documentation

2.1 Creating an irregular object

My idea was to re-create an octagon Hourglass as seen in Figure 1. In order to accomplish that task I decided to create out of a series of triangles (`tri function`) and quadrilaterals (`quad function`). However, unlike in Figure 1, mine would have a more visible inner part.

I modeled the vertices based 4 faces of different widths, with the top and bottom sharing one value for the width, and the inner "neck" of the hour composed of two octagons also sharing one value. To accomplish this task I decided to run the following python code which allowed me obtain various initial sizes. Moreover, the texture and normal coordinates were based on quadileteral and triangular properties respectively of each (`tri function`) and (`quad function`) functions. These were all modeled in the same clockwise manner in order for the normal matrix to work appropriately.

Figure 1: Octagon Hourglass [1]

```
def getshapevalues(height_value, vertices=8, width=1):
    coords = []
    for ind, value  in enumerate(range(vertices), start=1):
        x = cos(2*value*pi/vertices) * width
        y = sin(2*value*pi/vertices) * width
        coords.append([x, y])
       return coords
```

Listing 1: Getting values for the vertices

The code snippet in Listing 1 in Python is just a visual version of obtaining a polygon coordinates. These in reality would be checked for value and using strings formatted into JavaScript ready code. The result is show in Listing 2 below.

```
vec4(0.6, 0.0, -1.0, 1.0), // point 0
vec4(0.3*Math.sqrt(2), 0.3*Math.sqrt(2), -1.0, 1.0), // point 1
vec4(0.0, 0.6, -1.0, 1.0), // point 2
vec4(-0.3*Math.sqrt(2), 0.3*Math.sqrt(2), -1.0, 1.0), // point 3
vec4(-0.6, 0.0, -1.0, 1.0), // point 4
vec4(-0.3*Math.sqrt(2), -0.3*Math.sqrt(2), -1.0, 1.0), // point 5
vec4(0.0, -0.6, -1.0, 1.0), // point 6
vec4(0.3*Math.sqrt(2), -0.3*Math.sqrt(2), -1.0, 1.0), // point 7
vec4(0, 0, -1.0, 1.0), // point (centroid) 8
```

Listing 2: Bottom Octagon example vertex defintions

Hence, using this procedure, from the four octagon faces, and inner points of the bottom and top face, a total of 34 vertices (inner octagon centroids were not needed). The points were connected using.

2.2 ModelView and Projection matrices

Next, I added a viewer position, a projection and computed the ModelView and Projection matrices using a Javascript application. The ModelView was computed using the `lookAt` function which can concatenate with modeling transformations. Subsequently, sliders were added to the HTML file, which include the following transformations: radius, theta, phi, Near, Far, Scale, FOV (field of view), aspect. Additionaly for animation and asthetic reasons enabled buttons which through automatic increments of theta, and appropriate rotation matrices (`rz, ry, rx`) allow the rotation of the hourglass around the roll pitch yaw angles. Moreover, the shape can be translated along each of the X, Y, and Z world axes.

2.3 Lighting: Directional and spotlight and Material

The material and light are very much interconnected hence they will be described in one section together. Both directional light and spotlight were required to be implemented with necessary elements depicted in Listing 3. The directional light and spotlight were calculated appropriately with these parameters as passing from each `gl.uniform4fv` to the vertex shader. These were hence sent to the fragment shader via `N, L, L_Directional, E_Directional, L_SpotLight, E_SpotLight` parameters. There the material was blended with the shader model described in the next subsection, and multiplied by the spotlight outcome `spotfactor` which itself was based on the distance to the light source, and constant attenuation.

```
var lightPosition, lightAmbient, lightDiffuse; // main light
var lightPositionDirectional, lightAmbientDirectional, lightAmbientDirectional;
var SpotlightAmbient, SpotlightDiffuse, SpotlightDirection;
var materialAmbient, materialDiffuse; //material
var constantAttenuation; // a single non changing  attenuation
var spotLightAngle, spotLightCutOff; // spotlight parameters
```

Listing 3: Light and Material parameters

2.3.1 Calculating the spotlight

The spot direction is specified as a vector which simplifies calculations. The size of the cone is specified by a cutoff angle, and the light is only emitted from the light position in directions whose angle with the spot direction is less than the cutoff angle.

Here, I used the cosine of the cutoff angle instead of the angle itself, then as in Listing 4 compare the cutoff value (predefined) using the dot product (as in the the slides of the course `Angel_UNM_14_8.pdf`) of the `E_SpotLight` and -L (since +L is coming from the object itself that represents the cosine of the angle between the light ray and the spot direction. Then we compute the spotFactor which is multiplied by basic light color to give the effective light intensity of the spotlight at a point on a surface.

```
1  float spotCosine =  max(dot(E_SpotLight , -L), 0.0);
2  float spotFactor;
3  if (acos(spotCosine) < radians(spotLightAngle)){
4    spotFactor = pow(spotCosine, spotLightCutOff) * attenuation;
5  }else{ spotFactor = 0.0;}
```

Listing 4: Spotfactor assignment

2.4 Per-fragment shading model

The per-fragment shading model used is called the Cartoon Shade Algorithm [2] First, we had to calculate the illuminated diffuse color as in Equation 1, which is the vertex color. Next, the shadowed diffuse color in Equation 2. Finally once, we computed the value of the product of the unit vector from the light source to the vertex $\overrightarrow{L}$ and the unit vector normal to the surface at the vertex $\overrightarrow{n}$, we assign to the fragment the value based on the check in Equation 3. Note that the result of this product is the cosine of the angle between the two vectors. The resulting value would be either multiplied by the texture (described in the following section) or left as is a sum of C_i and C_s. Note that in the spotlight shading these would also be multiplied by the spot factor.

$$C_i = a_g \times a_m + a_l \times a_m + d_l \times d_m \tag{1}$$

$$C_s = a_g \times a_m + a_l \times a_m \tag{2}$$

$$\text{fragment assignment} = \begin{cases} C_i, & \text{if } \max(\overrightarrow{L} \cdot \overrightarrow{n}, 0) \geq 0. \\ C_s, & \text{if } \max(\overrightarrow{L} \cdot \overrightarrow{n}, 0) < 0. \end{cases} \tag{3}$$

2.5 Texture from file

The texture is simulated via a PNG loaded from file resource which depicts the the surface of sand beach. The coordinates were pushed from predefined coordinates in both the (`tri function`) and quadrilaterals (`quad function`) functions. Next, thanks to the `configureTexture` function there were mapped and sent to the fragment shader. The texture can be toggled on and off as requested. The mapping I used was `gl.NEAREST_MIPMAP_LINEAR` which means no filtering on the texture image but linear filtering between the two nearest MIP levels. This resulted in a sand like vertical texture.

References

[1] `http://evershinegift.com/Index.asp?Product420/742.html`

[2] Lake, Adam & Marshall, Carl & Harris, Mark & Blackstein, Marc. (2000). Stylized Rendering Techniques For Scalable Real-Time 3D Animation. Proceedings of the Symposium on Non-Photorealistic Animation and Rendering. 10.1145/340916.340918. `https://dl.acm.org/doi/pdf/10.1145/340916.340918`

[3] `https://www.khronos.org/registry/webgl/specs/latest/2.0`

[4] `https://www.w3schools.com/css`