Interactive Graphics Homework 1

A.Y. 2019/2020

Mauro Ficorella

Introduction

This relation describes the choices made for developing the Interactive Graphics' homework1; this project is written using HTML, JavaScript and WebGL.

I've to implement an irregular geometry of 20 to 30 (maximum) vertices. This geometry was implemented adding several features to it, as described below in the solution.

Solution

Aesthetic aspect:

Regarding the aesthetic aspect of the html, I have used a particular style for sliders and buttons, and these styles are implemented in the html file near to each slider/button definition; I have also used some scripts to write next to each slider the precise value of that slider in every moment it is used (this also for a practical reason, so in this way I can see in every moment the exact value of a lot of parameter in my geometry).

I've also chose a light variant of brown for the vertices' color, and "pearl" for the material, in order to obtain a clearly visible surface, also with the texture, directional light and spotlight enabled at the same time.

Requirement 1:

In order to make an irregular geometry made of 20 to 30 vertices, I chose a figure representing an "M", as the initial of my name.

This geometry is made of 24 (visible) vertices.

I chose normal of 3 coordinates and texture of 2 coordinates.

To compute normals I needed to subtract vertices of two sides (as two vectors) of each geometry's face and then I needed to perform cross product between these two vectors, as it follows in the following formula:

```
var t1 = subtract(vertices[b], vertices[a]);
var t2 = subtract(vertices[c], vertices[b]);
var normal = cross(t1, t2);
normal = vec3(normal);
```

Finally I pushed, for each vertex, the normal to the array normalsArray.

Regarding texture coordinates, I used the following method for choosing them: since my geometry is made up of squares, rectangles and parallelograms, I have calculated texture coordinates in four different ways, one for squares, one for rectangles and two for parallelograms (depending on the parallelogram's inclination in my geometry). After this, I have made four different functions, square rectangle parallelogram1 and parallelogram2, in which I added (via pushing them) vertices, vertices' colors, normals' and texture's coordinates in the relative arrays.

Requirement 2:

In order to calculate model view and projection matrices, I needed to position the viewer, in this case the camera; for the camera position I've used model view matrix.

These two matrices are defined in the vertex shader and can be found in the homework1.html file with these usages:

```
uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
gl_Position = uProjectionMatrix * uModelViewMatrix * aPosition;
```

Model view matrix is calculated in the homework1.js file in the render() function, using the function lookAt(eye,at,up) stored in MVnew.js, like this:

```
modelViewMatrix = lookAt(eye, at, up);
```

This lookAt function is aimed to move all objects in front of the camera.

Also the projection matrix is calculated in homework1.js in the render() function, using the function perspective(fovy,aspect,near,far) stored in MVnew.js, like this:

```
projectionMatrix = perspective(fovy, aspect, near, far);
```

Regarding viewing position and viewing volume, I made them controllable through some sliders, computed in html file. I also set those parameters in the proper way in the js file, so that the object is clearly visible at the starting of the program.

I've also implemented the rotation of the geometry respect to theta and phi angles: another reason for this choice was that with rotation, directional light could have been seen way better.

Requirement 3:

First of all, regarding lights, I've not implemented specular component neither in directional light nor in spotlight, because in the cartoon shading it wasn't present specular component, so it would have been useless.

I've implemented a spotlight in a finite position and a directional light.

Regarding the spotlight I also used attenuation and exponent. The exponent is used to set how concentrated the spotlight is (via "spotFactor" in the code), as explained in the following formula:

```
float cos = max(dot(normalize(spotD), normalize(-spotL)),0.0);
float spotFactor = pow(cos,spotExponent);
```

The attenuation (attenuation with constant parameter set to 2.0, linear parameter set to 1.0 and quadratic parameter set to 0.5) is used in order to obtain an attenuation of the spotlight due to the spotlight's distance from the geometry, as it follows in the following formulas:

```
pLightDirection = vec3(spotLightPosition.xyz - aPosition.xyz);
float dist = length(pLightDirection);
float constantAtt = 2.0;
float linearAtt = 1.0;
float quadraticAtt = 0.5;
float spotAttenuation = 1.0/(constantAtt + linearAtt*dist + quadraticAtt*pow(dist,2.0));
```

After these computations, either spotFactor (that contains exponent) and spotAttenuation are multiplied in the ambient and diffuse components of the spotlight in the fragment shader.

I've also made one button and seven sliders to better control the spotlight, in particular:

- Spotlight X position to move spotlight's position along the x-axis;
- Spotlight Y position to move spotlight's position along the y-axis;
- Spotlight Z position to move spotlight's position along the z-axis; with this slider it's possible to see how attenuation varies when the spotlight moves away along the z-axis positive direction;
- Spotlight X direction to control spotlight's direction along the x-axis;
- Spotlight Y direction to control spotlight's direction along the y-axis;
- Spotlight Cutoff to increase or decrease spotlight's cutoff;
- Spotlight Exponent to increase or decrease spotlight's exponent (spotlight concentration);

For both directional light and spotlight I've used those parameters, located in the js file:

```
var dirLightPosition = vec4(1.0, 1.0, 1.0, 0.0 );
var dirLightAmbient = vec4(0.2, 0.2, 0.2, 1.0 );
var dirLightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var spotLightPosition = vec4( 0.0, 0.0, 0.0, 1.0);
var spotLightAmbient = vec4( 0.2, 0.2, 0.2, 1.0 );
var spotLightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var spotLightDirection = vec4( 0.0, 0.0, -1.0, 1.0 );
var lCutOff = 2.5 * Math.PI/180.0;
var spotExponent = 80.0;
```

Note that, in order to implement directional light, I used 0.0 value as the last element of the vector dirLightPosition. For the same reason, in the last element of the vector spotLightPosition, there is a 1.0, meaning that, in this case, spotlight is in a finite position.

Indeed, I've implemented a button that turns on/off the spotlight, and in the code I've implemented this using a boolean value "sFlag" in both html and js file (initialized to false in the js file, so when the program starts up the spotlight is off) to control properly this button's behaviour.

Requirement 4:

For the material I've chose "pearl" (for the reasons explained in the aesthetic section at the beginning of this document) material, with parameters set in the proper way in the js file, as it follows:

```
var materialAmbient = vec4(0.25, 0.20725, 0.20725, 0.922);
var materialDiffuse = vec4(1.0, 0.829, 0.829, 0.922);
var materialShininess = 11.264;
```

In order to obtain reflected colors, I have to multiply light/material properties as it follows in the js:

```
var dirAmbientProduct = mult(dirLightAmbient, materialAmbient);
var dirDiffuseProduct = mult(dirLightDiffuse, materialDiffuse);
var spotAmbientProduct = mult(spotLightAmbient, materialAmbient);
var spotDiffuseProduct = mult(spotLightDiffuse, materialDiffuse);
```

Requirement 5:

Concerning the shading model, I've implemented, in the per-fragment way, in the fragment shader, the cartoon shading using formulas written in the pdf that I've received together with the homework assignment.

First of all, I calculated in the vertex shader normal at each vertex, direction to light and to vertex; then I sent normal and direction to the light to the fragment shader. The rest of the computation takes place in the fragment shader.

I've declared, in the fragment shader, the following variables, whose value is computed in the js file:

```
uniform vec4 spotAmbientProduct, spotDiffuseProduct;
uniform vec4 dirAmbientProduct, dirDiffuseProduct;
```

Then, regarding cartoon shading, I've declared those variables :

```
float directionalConstant = max( dot(dirL, N), 0.0 );
float spotlightConstant = max( dot(spotL, N), 0.0 );
vec4 AG = vec4(0.0, 0.0, 0.0, 1.0);
```

I've used two constants instead of one because I've two light sources, so using only one would have produced an incorrect illumination result. For global ambient light I've used a vector made of zeros since I already have two light sources, so another light would have been unnecessary.

I've repeated this reasoning also for computing Ci and Cs values provided in the general formula: since I have two light sources, I've made directionalCi, directionalCs, spotlightCi and spotlightCs. These parameters are computed in the following way (with dirAmbient=AL*AM, dirDiffuse=DL*DM, computed using directional light values, and spotAmbient=AL*AM, spotDiffuse=DL*DM, computed using spotlight values, as specified in html and js files):

```
vec4 directionalCi = (AG * AM) + dirAmbient + dirDiffuse;
vec4 directionalCs = (AG * AM) + dirAmbient;
vec4 spotlightCi = (AG * AM) + spotAmbient + spotDiffuse;
vec4 spotlightCs = (AG * AM) + spotAmbient;
```

After those computation, I used those two "if" functions and did the other computations to calculate spotlight and directional light in terms of cartoon shading:

```
if (directionalConstant >= 0.5){
    cartoonDirectional = directionalCi;
}
else{
    cartoonDirectional = directionalCs;
}
if (spotlightConstant >= 0.5){
    cartoonSpotlight = spotlightCi;
}
else{
    cartoonSpotlight = spotlightCs;
}
```

Requirement 6:

After I've wrote texture coordinates in the js file, as described at the beginning of this document, I've pushed those values to the texture coordinates array; using configureTexture() function, I bound together all these texture coordinates.

I've computed pixel color as a combination of the color computed using the lighting model and the texture, via the following formula in the html file, located in the fragment shader:

```
fColor = (cartoonDirectional + cartoonSpotlight + vColor)*texture(uTextureMap, vTexCoord);
```

Specifically, I've multiplied texture function with the sum between vertices color and the lights (calculated via cartoon Shading in the fragment shader) so that pixel color would have been a combination of colors computed using both lighting model and texture.

I've also implemented two button that activate/deactivate the texture from my geometry, using a boolean value "tFlag" in both html and js file (initialized to true in the js file, so when the program starts up the texture is enabled).

Advantages and disadvantages of my solution:

In my project I have implemented perspective projection, in which objects far from the viewer are projected smaller than objects with the same size positioned closer to the viewer; this projection looks very realistic; regarding angles, they aren't kept in plans that aren't parallel to the projection plan.

My solution has the main advantage of being a lot flexible; it allows to choose between different positions of the viewer, to activate/deactivate texture and spotlight, and also allows to read in real time every single value that is set through sliders.

Features and results:

In this project, when rotating the figure through sliders, light changes according to directional light; when enabling spotlight, it correctly shows and it increments/decrements intensity of the amount of light shown according to the distance between it and the geometry; there is also the possibility to modify the cutoff size so spotlight looks bigger/smaller in real time, and the same possibility is available also for the exponent in the spotlight formulas used in the code.

Indeed there is also the possibility to activate/deactivate the texture, implemented in the proper way on each face of the geometry.