Interactive Graphics Project Documentation

George Paul Pislariu 1647679

1 Project layout

The HTML page is structured in the following way: there are three sliders groups each one varying the main parameters of the represented objects.

1.1 Viewing position

This group of sliders allows us to control the camera position:

- Theta: with range between -90 and +90 allows to control the camera's theta angle of both projections;
- Phi: with range between -90 and +90 allows to control the camera's phi angle of both projections;
- Radius: with range between 1 and 2 allows to control the camera's radius (distance from the origin) of the perspective projection.

1.2 Projections

This group of sliders allows us to control the main projections parameters.

- Perpective Fovy: with range between 10 and 150 allows to control the fovy angle (angle between the top and bottom planes of the clipping volume);
- Perspective Aspect: with range between 0.5 and 2 allows to control the aspect ratio (width divided by height) of the perspective projection;
- Far Plane: with range between 1 and 10 allows to control the far plane distance from the camera for both objects;
- Near Plane: with range between 0.5 and 3 allows to control the near plane distance from the camera for both objects;
- Orthogonal Width: with range between 0 and 20 allows to controls the width in the orthogonal projection viewing volume;
- Orthogonal Height: with range between 0 and 20 allows to controls the height in the orthogonal projection viewing volume;

1.3 Translation and Scaling

This group of sliders allows us to scale and translate on the three axes both objects.

- Scale: with range between 0.1 and 1.5 allows to scale both objects;
- Translate x: with range between -2 and +2 allows to translate both objects along the x axis;
- Translate y: with range between -2 and +2 allows to translate both objects along the y axis;
- Translate z: with range between -2 and +2 allows to translate both objects along the z axis;

2 Implementation details

2.1 Canvas

In order to achieve the point 4 of this homework, the canvas was split in two sections. We can have more than one WebGL context on a page, but each context must manage it's own resources. We cannot create buffers or textures on one context and then use them on the other: we would need to load all your resources twice. This approach can be very intensive for memory. A better solution, that we implement in this case, is to use gl.viewport to render to half of it for each view of the scene. With this solution the advantages are clear: we can define buffers and textures just one time and use it for both objects at same time. In order to render at the same time both objects with a different projection each, it has been defined the following function:

renderScene (drawX, drawY, drawWidth, drawHeight, pMatrix, mvMatrix)

This function has 4 parameters in input that define the viewport dimensions and position and two matrices, projection and modelView. The function's task is to define the viewport and render the scene, considering input's parameters. In our case it has been invoked twice: the first one defines the left area of canvas when draws the cube with an orthogonal view, the second one defines the right area of canvas when draws the cube with a perspective view.

renderScene(0, 0, canvas.width/2, canvas.height, pMatrix1, mvMatrix)
renderScene(canvas.width/2, 0, canvas.width/2, canvas.height, pMatrix2, mvMatrix)

2.2 Light

When we look at a point on an object, the color that we see is determined by multiple interactions among light sources and reflective surfaces. For every light source, we must specify its color and either its location or its direction. We can specify the position of the light using a vec4 type defined in MV.js:

```
var lightPosition = vec4(2.0, 2.0, 0.0, 0.0);
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0);
var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0);
var lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);
```

Material properties should match up directly with the supported light sources and with the chosen reflection model:

```
var materialAmbient = vec4(1.0, 0.0, 1.0, 1.0);
var materialDiffuse = vec4(1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4(1.0, 0.8, 0.0, 1.0);
var materialShininess = 100.0;
```

Because light from multiple sources is additive, we can repeat the calculation for each source and add up the individual contributions. We have three choices as to where we do the calculation: in the application, in the vertex shader, or in the fragment shader. Although the basic model can be the same for each, there will be major differences both in efficiency and appearance depending on where the calculation is done.

2.3 Shading models

In this project two shading models have been implemented: Gouraud and Phong shading. Gouraud shading is a per-vertex color computation. What this means is that the vertex shader must determine a color for each vertex and pass the color as an out variable to the fragment shader. Since this color is passed to the fragment shader as an in varying variable, it is interpolated across the fragments thus giving the smooth shading.

In contrast, Phong shading is a per-fragment color computation. The vertex shader provides the normal and position data as out variables to the fragment shader. The fragment shader then interpolates these variables and computes the color.

In other words in Gouraud Shading, the color for the fragment is computed in the Vertex Shader. Whereas, in Phong Shading, the color for the fragment is computed in the Fragment Shader.

In this application we can switch between those shading models using a button. Depending on the button's value the computation of the shading model takes place in the fragment shader or in the vertex shader.

2.4 Procedural texture

The main advantage of the procedural texture mapping requires much less memory as compared to image based texture mapping. There is no image to download or store in RAM or in the GPU's memory. Otherwise the disadvantage is the texture maps are calculated at rendering time for each individual fragment. If the calculations are complex, rendering speeds become slower.

In order to apply a procedural texture at our objects first the uniforms sampler2D were defined in the fragment shader. It has been used the texture coordinates passed from the vertex shader and has been called the function texture2D to look up a color from that texture.

```
uniform sampler2D Tex0;
uniform sampler2D Tex1;
gl_FragColor = fColor*(texture2D(Tex0, fTexCoord)*texture2D(Tex1, fTexCoord));
```

In the JavaScript file it has been defined two functions "image1" and "image2" in order to create a checkerboard pattern using floats and then to convert floats to ubytes for texture. Then the texture coordinates has been set:

```
var tBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, tBuffer);
gl.bufferData( gl.ARRAY_BUFFER, flatten(texCoordsArray), gl.STATIC_DRAW
);

var vTexCoord = gl.getAttribLocation( program, "vTexCoord");
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vTexCoord);
```

At the end the function "configureTexture(image)" invokes the gl.createTexture() method of the WebGL API that creates and initializes a WebGLTexture object.