UNIVERSITA' DEGLI STUDI DI ROMA
"LA SAPIENZA"

Interactive Graphics: Homework #1 Report

Student: Giulia CASSARA'

Matricola: 1856973

1

> Add the viewer position (your choice), a projection (your choice) and compute the ModelView and Projection matrices in the Javascript application. The viewer position and viewing volume should be controllable with buttons, sliders or menus.

My application takes from the sliders of the html page the values of Radius, Theta, and Phi angles - although they have a default value-. Then, those values are passed to the javascript applications, they are used to compute the ModelView matrix as follows:

```
var render = function() {
  gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    eye = vec3(radiusCamera*Math.sin(thetaCamera)*Math.cos(phiCamera),
          radiusCamera*Math.sin(thetaCamera)*Math.sin(phiCamera),
          radiusCamera*Math.cos(thetaCamera));

    modelView = lookAt(eye, at, up);
  ....
}
```

Listing 1: ModelView

Where at and up are declared as follows: at = vec3(0.0, 0.0, 0.0), up = vec3(0.0, 1.1, 0.0);
For the projection matrix I simply take the values from the sliders of the parameters fovy, aspect, near, far and put them on the perspective function.

2

> Include a scaling (uniform, all parameters have the same value) and a translation Matrix and control them with sliders.

Scaling is nothing but increasing or decreasing the size of an object. I used a 4×4 diagonal matrix having the scaling factors of x,y,z coordinates in the diagonal positions (as it is the last diagonal position 1).
By default the diagonal matrix is identity matrix but with the sliders from the html page we can regulate the scale value and adjust the scaling of the cube.

In the vertex shader program, we multiply the scale matrix with the transformation matrix to compute then the gl_Position which holds the final position of the vertices.

```
var scaleMat = [
    scaleFactor, 0.0, 0.0, 0.0,
    0.0, scaleFactor, 0.0, 0.0,
    0.0, 0.0, scaleFactor, 0.0,
    0.0, 0.0, 0.0, 1.0
    ];

    gl.uniformMatrix4fv(gl.getUniformLocation(program, "scaleMat"),
       false, scaleMat);
}
```

Listing 2: Scale Matrix

3

Define an orthographic projection with the planes near and far controlled by sliders.

The way we compute the orthogonal projection is similar to the perspective projection but we use the ortho function with values of near and far regulated by the user in the html page.

```
ProjectionMat = perspective(fovy, aspect, near, far);
gl.uniformMatrix4fv(gl.getUniformLocation(program, "ProjectionMat"),
    false, flatten(ProjectionMat));
```

Listing 3: Orthogonal Projection Matrix

4

Split the window vertically into two parts. One shows the ortographic projection defined above, the second uses a perspective projection. The slider for near and far should work for both projections. Points 5 to 7 use the splitted window with two different projections

We splitted the screen into two parts. The left side cube has the perspective projection, the right side cube has the orthogonal projection. To do the splitting we used the following methods:
The WebGLRenderingContext.scissor() method of the WebGL API sets a scissor box, which limits the drawing to a specified rectangle. When you first create a WebGL context, the size of the viewport will match the size of the canvas. However, if you resize the canvas, you will need to tell the WebGL context a new viewport setting. In this situation, you can use gl.viewport. Finally, The WebGLRenderingContext.drawArrays() method of the WebGL API renders primitives from array data.

```
    ProjectionMat = perspective(fovy, aspect, near, far);
    gl.uniformMatrix4fv(gl.getUniformLocation(program, "ProjectionMat"),
        false, flatten(ProjectionMat));

    gl.scissor(0, height/2, width/2, height/2);
    gl.viewport(0, height/2, width/2, height/2);
    gl.drawArrays(gl.TRIANGLES, 0, numVertices);

    ProjectionMat = ortho(-1.0, 1.0, -1.0, 1.0, near, far);
    gl.uniformMatrix4fv(gl.getUniformLocation(program, "ProjectionMat"),
        false, flatten(ProjectionMat));

    gl.scissor(width/2, height/2, width/2, height/2);
    gl.viewport(width/2, height/2, width/2, height/2);
    gl.drawArrays(gl.TRIANGLES, 0, numVertices);
}
```

Listing 4: Splitting screen

5

Introduce a light source, replace the colors by the properties of the material (your choice) and assign to each vertex a normal.

Implement both the Gouraud and the Phong shading models, with a button switching between them.

In order to implement the Gouraud and Phong shading models we have to make some computation on vertex shader and fragment shader. In the html page we use a switch botton to choose between the two types of shading models. To do that, we use a flag that tells us if we are computing the Gouraud or the Phong shading models. For the vertex shader we have the following computations:

```
void main() {

          TransfMat = transMat * scaleMat;
          gl_Position = ProjectionMat * modelView * TransfMat *
             vPosition;

          vec3 pos = -(modelView * TransfMat * vPosition).xyz;
          vec3 light = lightPositionVec.xyz;

          vec3 L = normalize( light - pos );

          vec3 E = normalize( -pos );
          vec3 H = normalize( L + E );

          vec3 N = normalize((modelView * TransfMat * vColor).xyz);

          if (!shadingType) {        // Gouraud

            vec4 ambient = v_ambientProductVec;
            float Kd = max(dot(L, N), 0.0);

            vec4 diffuse = Kd * v_diffuseProductVec;
            float Ks = pow(max(dot(N, H), 0.0), v_materialShininessVal);

            vec4 specular = Ks * v_specularProductVec;
            if (dot(L, N) < 0.0) specular = vec4(0.0, 0.0, 0.0, 1.0);

            fColor = ambient + diffuse + specular;
            fColor.a = 1.0;

          } else {
            fL = L;
            fN = N;
            fE = E;
          }

          fTexCoord = vTexCoord;
      }
```

```

}
```

Listing 5: Vertex shader

For the fragment shader:

```
void main() {
        vec4 fColorFinal;

        // Step 6
          if (!shadingType) {
            fColorFinal = fColor;
          } else {
            vec4 fColorF;
            vec3 H = normalize(fL + fE);
            vec4 ambient = f_ambientProductVec;
            float Kd = max(dot(fL, fN), 0.0);
            vec4 diffuse = Kd * f_diffuseProductVec;
            float Ks = pow(max(dot(fN, H), 0.0), f_materialShininessVal)
                ;
            vec4 specular = Ks * f_specularProductVec;
            if (dot(fL, fN) < 0.0) specular = vec4(0.0, 0.0, 0.0, 1.0);
            fColorF = ambient + diffuse + specular;
            fColorF.a = 1.0;
            fColorFinal = fColorF;
          }

          gl_FragColor = fColorFinal * texture2D( texture, fTexCoord );
      }
}
```

Listing 6: Fragment shader

6

Add a procedural texture (your choice) on each face, with the pixel color a combination of the color computed using the lighting model and the texture

We have chosen a chess texture which is defined in the javascript applications. The pixel color is a combination of the pearl material and the texture itself. It is clear that this combination is done in this line of code in the fragment shader.

```
void main() {
                    gl_FragColor = fColorFinal * texture2D( texture,
                        fTexCoord );
}
```

Listing 7: Texture

7 Conclusions

The homework works successful. But there are some limitations and bugs. One of the bugs is that both cubes disappears completely with certain values of ”far” parameter without appearing anymore, the page must be refreshed. Another limitation is the shading, which seems to be too flat.