Computer Graphics
Hand-in-4

xFrednet

October 16, 2020

1 Hand-in 4

1.1 (20%) Shaders (2015 No. 4)

On the following two pages there are shaders for use in an OpenGL pipeline, a vertex shader and a fragment shader. You can add both uniform and varying variables to these shaders above their main() function definition and you can add any number of variables and code inside the main functions.

The following types can be used:

`float`	a floating point number
`vec4`	a 4 coordinate vector, used for positions, directions and colors
`mat4`	a 4x4 matrix

The following GLSL functions can be used:

`float`	`dot(vec4 v1, vec4 v2):`	Returns the dot product of two vectors
`vec4`	`normalize (vec4 v):`	Returns the normalized vector
`float`	`length(vec4 v):`	Returns the length of the vector
`vec4`	`cross(vec4 v1, vec4 v2):`	Returns the cross product of two vectors
`float`	`max(float a, float b):`	Returns the higher value
`float`	`min(float a, float b):`	Returns the lower value
`float`	`pow(float num, float exp):`	Returns num to the power of exp

`+, -, *, /` are done component-wise on vec4 but as matrix multiplications when one or both sides are mat4.

Variables are already defined for the attributes `a_position` and `a_normal` for each vertex that will be sent through the pipeline.

The current shaders both have the same initial definitions at this point. Strike out those that are unnecessary in each shader and add whatever definitions you need for each of the following problems.

1.1.1 a) (5%) Add variables and calculations needed to transform the position attribute (vertices) to global coordinates, eye coordinates and clip coordinates and set the value for the built-in output variable gl_Position.

1.1.2 b) (10%) Add variables and calculations needed for the shaders to do per-fragment lighting. Include ambient, diffuse and specular lighting for a single light source. Set the final color value to the built-in output variable gl_FragColor.

1.1.3 c) (5%) Fog is a color that is mixed with the final fragment color based on distance from the camera. Anything closer than u_startFog is 100% the fragment color, anything further away than u_endFog is 100% the fog color. Anything in between is a weighted average of the two based on the proportion of the distance in the startFog to endFog range. Add variables and calculations to the shaders to implement this functionality.

Vertex shader

```
    // xxx attribute vec3 a_position;
    // xxx attribute vec3 a_normal;

+   in vec3 in_position;
+   in vec3 in_normal;

+   out vec4 surface_normal; // Also called n
+   out vec4 to_light;   // Also called S
+   out vec4 to_camera;  // Also called v

    uniform vec4 u_lightPosition;
+   uniform vec4 u_camera_position;
    // xxx uniform vec4 u_lightColor;

    // xxx uniform vec4 u_materialAmbient;
    // xxx uniform vec4 u_materialDiffuse;
    // xxx uniform vec4 u_materialSpecular;
    // xxx uniform float u_materialShininess;

+   uniform mat4 u_transformation_matrix;
+   uniform mat4 u_view_matrix;
+   uniform mat4 u_projection_matrix;

    main() {
+       vec4 global_coordinates = u_transformation_matrix * vec4(in_position, 1.0);
+       vec4 eye_coordinates = u_view_matrix * eye_coordinates;
+       vec4 clip_coordinates = u_projection_matrix * eye_coordinates;

+       to_light = u_lightPosition - global_coordinates;
+       to_camera = u_camera_position - global_coordinates;

+       surface_normal = normalize(u_transformation_matrix * vec4(in_normal, 0.0));
+       gl_Position = clip_coordinates;
    }
```

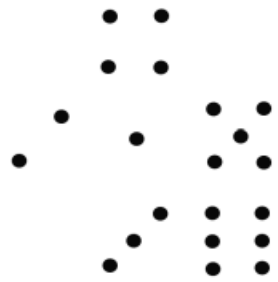
Fragment shader

```
    // xxx attribute vec3 a_position;
    // xxx attribute vec3 a_normal;

+   in vec4 surface_normal;
+   in vec4 to_light;
+   in vec4 to_camera;

    // xxx uniform vec4 u_lightPosition;
    uniform vec4 u_lightColor;

    uniform vec4 u_materialAmbient;
    uniform vec4 u_materialDiffuse;
    uniform vec4 u_materialSpecular;
    uniform float u_materialShininess;

+   uniform vec4 u_startFog;
+   uniform vec4 u_endFog;
+   uniform vec4 u_fog_color;

    main() {
+       // ##############
+       // Lighting
+       // ##############
+       // diffuse lighting
+       float lambert = max(dot(surface_normal * normalize(to_light)), 0.0);
        vec4 diffuse = u_materialDiffuse * lambert;
+
+       // specular lighting
+       vec4 h = to_light + to_camera;
+       float phong = max(dot(surface_normal * normalize(h)), 0.0);
+       vec4 specular = u_materialSpecular * pow(phong, u_materialShininess);

+       // ambient
+       vec4 ambient = u_materialAmbient * u_lightColor;
+       vec4 fragment_color = diffuse + specular + ambient;

+       // #############
+       // Fog
+       // #############
+       float camera_distance = length(to_camera);
+       float fog_diff = u_endFog - u_startFog;
+       float in_fog_distance = camera_distance - u_startFog;
+       float fog_value = max(min(in_fog_distance / fog_diff, 0.0), 1.0);
+
+       // #############
+       // Output
+       // #############
+       //              <--         Light              -->  <--       Fog       -->
+       gl_FragColor = fragment_color * (1.0 - fog_value) + u_fog_color * fog_value;
+       // Looking back on this let's me see some optimizations but I'll keep it like this :)
    }
```

This doesn't look that good with latex tbh... but I'm to stubborn to change it at this point. Stubbornness is not always a good trait.

1.2 Graphics programming (10%) (2013 No. 7)

The class D6 (previous/opposite page) contains data to draw a 3D cube. Imagine that the texture sent into it's constructor is this image:



1.2.1 a) (5%) Add to the class D6 the code needed to use this texture and to map a different numbered face to each of the cube's sides. You can add code between any lines in the code on the opposite page.

Hint: You need to set up the UV coordinates (between 0 and 1) for all the cube's vertices.

1.2.2 b) (5%) A separate dice rolling engine returns a list of the structure RolledD6:

```
public class coords {
    public float x;
    public float y;
    public float z;
}
public class RolledD6 {
    public float size;
    public coords position;
    public coords rotationAxis;
    public float rotationAngle;
}
```

Finish code that will display each of the rolled dice of the size indicated, at the correct position and rotated by the rotation angle around the vector called rotation axis.

```
void drawDiceSet {
    List<RolledD6> diceList = DiceRollingEngine.getDice();
    foreach(RolledDice d in diceList) {
+       glTranslate(d.position.x, d.position.y, d.position.z);
+       glRotate(d.rotationAngle, d.rotationAxis.x, d.rotationAxis.y, d.rotationAxis.z);
+       glScale(d.size, d.size, d.size);
    }
}
```

```
public class D6 {
    FloatBuffer vertexBuffer;
    FloatBuffer texCoordBuffer;
    Texture texture;

    public D6(Texture tex)
    {
        vertexBuffer = BufferUtils.newFloatBuffer(72);
        vertexBuffer.put(new float[] {
            -0.5f, -0.5f, -0.5f, -0.5f,  0.5f, -0.5f, // back  | 6
             0.5f, -0.5f, -0.5f,  0.5f,  0.5f, -0.5f, //
             0.5f, -0.5f, -0.5f,  0.5f,  0.5f, -0.5f, // left  | 2
             0.5f, -0.5f,  0.5f,  0.5f,  0.5f,  0.5f, //
             0.5f, -0.5f,  0.5f,  0.5f,  0.5f,  0.5f, // front | 1
            -0.5f, -0.5f,  0.5f, -0.5f,  0.5f,  0.5f, //
            -0.5f, -0.5f,  0.5f, -0.5f,  0.5f,  0.5f, // right | 5
            -0.5f, -0.5f, -0.5f, -0.5f,  0.5f, -0.5f, //
            -0.5f,  0.5f, -0.5f, -0.5f,  0.5f,  0.5f, // up    | 4
             0.5f,  0.5f, -0.5f,  0.5f,  0.5f,  0.5f, //
            -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,  0.5f, // down  | 3
             0.5f, -0.5f, -0.5f,  0.5f, -0.5f,  0.5f  //
        });
        vertexBuffer.rewind();

        texCoordBuffer = BufferUtils.newFloatBuffer(48);
        texCoordBuffer.put(new float[] {
+           0.66f, 0.66f,    1.00f, 0.66f, // back  | 6
+           0.66f, 1.00f,    1.00f, 1.00f, //
+           0.00f, 0.33f,    0.33f, 0.33f, // left  | 2
+           0.00f, 0.66f,    0.33f, 0.66f, //
+           0.33f, 0.33f,    0.66f, 0.33f, // front | 1
+           0.33f, 0.66f,    0.66f, 0.66f, //
+           0.66f, 0.33f,    1.00f, 0.33f, // right | 5
+           0.66f, 0.66f,    1.00f, 0.66f, //
+           0.33f, 0.00f,    0.66f, 0.00f, // up    | 4
+           0.33f, 0.33f,    0.66f, 0.33f, //
+           0.33f, 0.66f,    0.66f, 0.66f, // down  | 3
+           0.33f, 1.00f,    0.66f, 0.66f //
        });
        texCoordBuffer.rewind();
+       this.texture = tex;
    }
    public void draw()
    {
        Gdx.gl11.glVertexPointer(3, GL11.GL_FLOAT, 0, vertexBuffer);
+       Gdx.gl11.glTexCoordPointer(2, GL11.GL_FLOAT, 0, texCoordBuffer);
+       Gdx.gl11.glActiveTexture(GL11.GL_TEXTURE0);
+       Gdx.gl11.glBindTexture(GL11.GL_TEXTURE_2D, texture);

        Gdx.gl11.glNormal3f(0.0f, 0.0f, -1.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 0, 4);
        Gdx.gl11.glNormal3f(1.0f, 0.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 4, 4);
        Gdx.gl11.glNormal3f(0.0f, 0.0f, 1.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 8, 4);
        Gdx.gl11.glNormal3f(-1.0f, 0.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 12, 4);
        Gdx.gl11.glNormal3f(0.0f, 1.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 16, 4);
        Gdx.gl11.glNormal3f(0.0f, -1.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 20, 4);

+       // I don't unbind the things here because there isn't much space in the
+       // exam and it'll just be overwritten by the next render :)
    }
}
```

I'm not sure if these are the additions that where expected for this task. However, I believe that this should render the dice correctly :)

2 TGRA Hand-in 3

2.1 Lighting calculations (10%) (2014 No. 6)

A single light is in the light model in an OpenGL program. It has the ambient values (0.0, 0.0, 0.0), diffuse values (0.5, 0.3, 0.7), specular values (0.3, 0.8, 0.7) and position (5.0, 8.0, -1.0). There is also a global ambient factor of (0.3, 0.2, 0.4) in the light model. A camera is positioned in (4.0, 6.0, 5.0) and looks towards P.

P has the color values: ambient (0.4, 0.2, 0.3), diffuse (0.4, 0.7, 0.2) and specular (0.6, 0.6, 0.6). It has a shininess value of 13. It has the position (4.0, 4.0, 3.0) and a normal (0.0, 1.0, 0.0).

2.1.1 What will be the blue color value for P on the screen ?

I'm going to calculate the RGB value for the given point. This is not required but probably a good practice :)

$$n = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$S = light_{pos} - P = \begin{pmatrix} 1 \\ 4 \\ -4 \end{pmatrix}$$

diffused lighting

$$lambert = max\left(\frac{n \circ S}{|n| * |S|}, 0\right) = max\left(\frac{4}{\sqrt{33}}, 0\right) \approx 0.6963$$

specular lighting

$$v = camera_{pos} - P = \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix}$$

$$h = v + S = \begin{pmatrix} 1 \\ 6 \\ -2 \end{pmatrix}$$

$$phong = max\left(\frac{n \circ h}{|n| * |h|}\right) = max\left(\frac{6}{\sqrt{41}}, 0\right) \approx 0.9370$$

$$f = 13(shininess)$$

light

$$I = (I_d * material_d * lambert) + \left(I_s * material_s * phong^f\right) + (I_a * material_a) + (I_{gaf} * material_a)$$

$$I \approx \left(\begin{pmatrix} 0.5 \\ 0.3 \\ 0.7 \end{pmatrix} * \begin{pmatrix} 0.4 \\ 0.7 \\ 0.2 \end{pmatrix} * 0.6963\right) + \left(\begin{pmatrix} 0.3 \\ 0.8 \\ 0.7 \end{pmatrix} * \begin{pmatrix} 0.6 \\ 0.6 \\ 0.6 \end{pmatrix} * 0.9370^{13}\right) + \left(\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} * \begin{pmatrix} 0.4 \\ 0.2 \\ 0.3 \end{pmatrix}\right) + \left(\begin{pmatrix} 0.3 \\ 0.2 \\ 0.4 \end{pmatrix} * \begin{pmatrix} 0.4 \\ 0.2 \\ 0.3 \end{pmatrix}\right)$$

$$I \approx \left(\begin{pmatrix} 0.20 \\ 0.21 \\ 0.14 \end{pmatrix} * 0.6963\right) + \left(\begin{pmatrix} 0.18 \\ 0.48 \\ 0.42 \end{pmatrix} * 0.9370^{13}\right) + \begin{pmatrix} 0.12 \\ 0.04 \\ 0.12 \end{pmatrix}$$

$$I \approx \begin{pmatrix} 0.3365 \\ 0.3922 \\ 0.3977 \end{pmatrix}$$

The blue value of the object in point P is $\approx$ 0.3977

2.2 (10%) Cohen-Sutherland Clipping (2015 No. 2)

A clipping window has the following geometry:

- Window(left, right, bottom, top) = (200, 600, 100, 400)

A line with the following end points is drawn in the world:

- P1: (240, 480)
- P2: (140, 300)

2.2.1 Show how the Cohen-Sutherland clipping algorithm will clip these lines and what their final endpoints, if any, are. Show the coordinate values of P1 and P2 after each pass of the algorithm.

	left	**center**	**right**
top	1001	1000	1010
center	0001	0000	0010
bottom	0101	0100	0110

Iteration 1

```
P1_code = 1000
P2_code = 0001

// => Clip P1 to the top

P1.x += (top - P1.y) * (P2_x - P1_x) / (P2_y - P1_y) // -44.4444
P1.y = top

// P1: (195.5, 400)
// P2: (140  , 300)
```

Iteration 2

```
P1_code = 0001
P2_code = 0001

// Both points are outside on the same side
// => The entire line gets clipped. The points from the last
//    Iteration still stand
//      * P1: (195.5, 400)
//      * P2: (140  , 300)
```

Okay, this is the end of this hand-in. I have to say that I've gotten used to writing LaTeX by now. It's still slower than writing by hand or in the case of No.2 just using a monospace environment. But this is a good exercise and I had some fun writing these 300 lines ^^

3 TGRA Handin 2

3.1 (10%) Transformations and matrices (2015 No. 1)

You have access to the same matrix and graphics classes as we built together this semester, that is a BoxGraphic class with a drawSolidCube that sends vertices into the pipeline for a cube of the size 1x1x1, centered in (0,0,0), and a ModelMatrix class with the basic transformation operations, matrix stack operations and shader manipulation operations. These classes have been created and initialized elsewhere. You don't need to remember the exact names of functions, just the idea behind each one you need to call, and what parameters they take.

What you want to draw is the following scene:

a) A box of size 2x2x2, centered in (9,5,-2) and under it a big flat floor which is 10x10 in the x-z plane but only 0.8 thick. Its corner should be in (0,0,0) and its top edge level with the base x-z plane.

3.1.1 (5%) Write the code that would draw the scene described. Imagine you're already inside the display function and lights, colors and cameras have already been set.

I assume that the BoxGraphic instances have been setup with the following scale and position class members: I ignore the rotation because it wasn't mentioned :)

$$Scale_{Box} = \begin{pmatrix} 2.0 \\ 2.0 \\ 2.0 \end{pmatrix} Position_{Box} = \begin{pmatrix} 9.0 \\ 5.0 \\ -2.0 \end{pmatrix}$$

$$Scale_{Plane} = \begin{pmatrix} 10.0 \\ 0.8 \\ 10.0 \end{pmatrix} Position_{Plane} = \begin{pmatrix} 5.0 \\ -0.4 \\ 5.0 \end{pmatrix}$$

I've created two solutions for the code. The first one is just the code I created without looking at my code or any material. The seconds solution is a revamped and detailed solution. I would like to get feedback for both if possible.

First solution:

```
vertex_count = 6 * 2 * 3
for e in entities:
    matrix = (ModelMatrix.identity()
        .translate(e.position)
        .scale(e.scale))
    gl.glLoadUniform(self.transformation_matrix_loc, matrix.c_ptr())
    e.drawSolidCube()
```

Second solution:

```
for e in entities:
    self.model_matrix.load_identity()
    self.model_matrix.translate(e.get_position())
    self.model_matrix.scale(e.get_scale())

    self.shader.set_model_matrix(self.model_matrix.matrix)
    e.drawSolidCube()

    pygame.display.flip()
```

3.1.2 b) (5%) Show the values that are in the shader's model matrix when the first box is drawn.

$$model\_matrix = \begin{pmatrix} 2 & 0 & 0 & 9 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Check with python for validation:

```
>>> import glm
>>> mat = glm.mat4()
>>> mat = glm.translate(mat, e.position)
>>> mat = glm.scale(mat, e.scale)
>>> print(mat)

[           2 |           0 |           0 |           0 ]
[           0 |           2 |           0 |           0 ]
[           0 |           0 |           2 |           0 ]
[           9 |           5 |          -2 |           1 ]
```

LaTeX is so much fun :)

3.2 5. Camera Transformations (30%) (2015, No. 5)

3.2.1 a) (15%) A camera is set up to be positioned in (0,8,4) looking at the point (0,3,-1). It has an up vector (0,0,-1). Find the point of origin and vectors for the camera's coordinate frame.

$$origin = \begin{pmatrix} 8 \\ 40 \end{pmatrix}$$

I'm not a 100% sure which vectors I should calculate for the coordinate frame but I expect that these are $\vec{v}, \vec{n}, \vec{u}$ for the view matrix. This would also kind of make sense. So here we go:

$$\vec{n} = normalize(\vec{origin} - \vec{looking\_at}) = normalize \begin{pmatrix} 0-0 \\ 8-3 \\ 4--1 \end{pmatrix} = normalize \begin{pmatrix} 0 \\ 5 \\ 5 \end{pmatrix} \approx \begin{pmatrix} 0 \\ 0.707 \\ 0.707 \end{pmatrix}$$

$$\vec{u} = normalize\,(\vec{up} \times \vec{n}) \approx normalize \begin{pmatrix} 0.707 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\vec{v} = normalize\,((\vec{n} \times \vec{u})) \approx \begin{pmatrix} 0 \\ 0.707 \\ -0.707 \end{pmatrix}$$

3.2.2 (5%) Set up the values in a matrix that represents this position and orientation of a camera. Which matrix in your shader should be set to these values?

From formula sheet: $\begin{pmatrix} u_x & u_y & u_z & -(eye \circ u) \\ v_x & v_y & v_z & -(eye \circ v) \\ n_x & n_y & n_z & -(eye \circ n) \\ 0 & 0 & 0 & 1 \end{pmatrix}$

$$\texttt{ViewMatrix} \approx \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.707 & -0.707 & -2.828 \\ 0 & 0.707 & 0.707 & -8.485 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The view matrix should be set to these values.

3.2.3 (10%) The camera should have a field of view of 75 grad, an aspect ratio of 16:9, a near plane at 3 and a far plane at 25. Find the exact values for a matrix that calculates this camera. Which matrix in your shader should be set to these values?

From formula sheet: $\begin{pmatrix} \frac{2N}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2*F*N}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$

N = 3

F = 25

fov = 75

aspect = $\frac{16}{9}$

top = $N * tan\left(\frac{fov}{2}\right) \approx 3 * 0.767 \approx 2.301$

bottom = $-top \approx -2.301$

right = $top * aspect \approx 4.091$

left $= -right \approx -4.091$

$$\texttt{ProjectionMatrix} \approx \begin{pmatrix} 0.733 & 0 & 0 & 0 \\ 0 & 1.304 & 0 & 0 \\ 0 & 0 & -1.273 & 6.812 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The projection matrix should be set to these values.

$d := distance$

$f := shininess$

$lambert = max\left(\frac{n \circ S}{|n| * |S|}, 0\right)$

$phong = max\left(\frac{n \circ h}{|n| * |h|}, 0\right)$

$attFactor = attenuation.x * d^2 + attenuation.y * d + attenuation.z$

$I = \sum \left(\left(I_d * \frac{material_d * lambert}{attFactor}\right) + \left(I_s * \frac{material_s * phong^f}{attFactor}\right)\right) + (I_{gaf} * material_a)$

$Reflection(\vec{a}) = \vec{a} - 2 * (\vec{a} \circ \vec{n}) * \vec{n}$