BBM414 Introduction to Computer Graphics Lab. Practical # 4- Model Drawing and Viewing

Barış Yıldız
2210356107
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
baris.byildiz@gmail.com

Overview

In this experiment, model drawing and perspective viewing in WebGL2 is going to be explored. The experiment consists of two parts.

In the first part, a source code that implements a camera viewing a cube in a perspective manner using sliders is given. The task is to make the camera view a pyramid instead of a cube, via mouse movement instead of sliders.

In the second part, the task is to create a scene consisting of three monkey heads, all performing different animations (1). Additionally, the scene is viewed from a perspective camera that can be moved by the player via pointer lock and mouse controls.

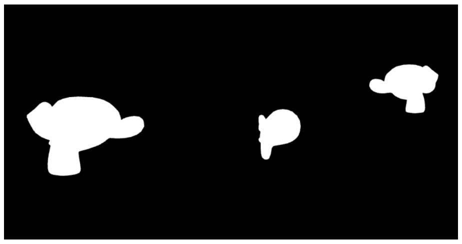

Figure 1: Monkey Heads (Note that lighting is not implemented.)

1 Part 1 - Perspective Pyramid

1.1 Assembling a Pyramid

The source code creates a cube object using vertices defined in a *vertices* array to form triangles that make up the cube using the *quad()* function. In the task, a *triangle()* method that assembles a triangle object is implemented and used in order to create the faces of the pyramid. The number of vertices to be drawn is halved. An additional vertex is added and the index of the magenta color in the *vertexColors* array is changed. The *triangle()* method is given below.

```javascript
function triangle(a, b, c, color) {
    pointsArray.push(vertices[a]);
    colorsArray.push(vertexColors[color]);
    pointsArray.push(vertices[b]);
    colorsArray.push(vertexColors[color]);
    pointsArray.push(vertices[c]);
    colorsArray.push(vertexColors[color]);
}
```

1.2 Mouse Controls

The mouse controls are implemented using the pointer lock API, mousemove events and the already implemented *phi* and *theta* variables. When the user presses the 'p' key, pointerlock is enabled. After this event, the camera sees the pyramid in different angles because of the change in *theta* and *phi* variables tied to the change in X and Y coordinates of the mouse respectively. Below are the added **event listener** functions.

```javascript
document.addEventListener("keyup", async (e) => {
    {
        if (e.key.toLowerCase() === "p") {
            if (!isPointerLocked) await canvas.
                requestPointerLock();
        }
    }
});

document.addEventListener("pointerlockchange", async(e) => {
    isPointerLocked = !isPointerLocked;
});

document.addEventListener("mousemove", (e) => {
    if (!isPointerLocked) return;
    if (e.movementX > 0) {
        theta += angleChange;
    } else if (e.movementX < 0) {
        theta -= angleChange;
    }

    if (e.movementY > 0) {
        phi += angleChange;
    } else if (e.movementY < 0) {
        phi -= angleChange;
    }
});
```

2 Part 2 - Scene with Three Monkey Heads

The task consists of three main steps, reading the .obj file that represents the monkey head, animating them and creating a mobile perspective camera.

2.1 Reading the .obj File

There are three types of "data lines" in the monkey head .obj file given.

- **Vertex Position Lines** that start with "v" and represent vertex positions of the object.
- **Normal Lines** that start with "vn" and represent the normal vectors of the object.

- **Face Lines** that start with "f" and represent triangles in the faces of the object.

In the experiment, **Vertex Position Lines** are read and pushed into an array. The space separated floats in these lines are the x, y and z coordinates of the positions of a vertex used by the object.

After the reading of position lines, **Face Lines** are read from the file. These lines consist of three "components" separated by spaces in the format **"<vertexIndex>//<textureIndex>//<normalIndex>"**. The <vertexIndex> part corresponds to the indexes of entries in **Vertex Position Lines**, in the order they appear on the file. Basically, the triangle vertex represented by this component has the position of the read position data line in the index <vertexIndex>. The idea is the same for the <textureIndex> and <normalIndex> parts. The object had no textures, so there were no texture coordinate data in the file and the <textureIndex> fields were empty in **Face Line** entries. Since lighting was not implemented, the <normalIndex> fields were not used, but were still read from the file for completeness.

2.2 Creating the Monkey Head

The *GameObject* class implemented in the experiment abstracts the process of reading the .obj file of an object, drawing it on the screen, binding shaders and setting certain properties such as position and rotation. Instanciating the *GameObject* class three times, three diferrent monkey head objects are spawned.

To draw the object on the screen, each position in each component of **Face Lines** are appended to a *triangleData* array and this array is used as the data for the vertex buffer of the object.

To apply rotation and translation to the object, the *u_translation* and *u_rotationAngle* uniforms of the shader attached to the object are accessed. For a proper rotation, the center of the object is also calculated.

Finally, for the object to appear in a perspective manner, the view and projection matrices of the camera are also linked to *u_viewMatrix* and *u_projectionMatrix* uniforms.

```
//Vertex Shader

#version 300 es
precision mediump float;
in vec4 a_position;
in vec3 a_normal;

uniform float u_rotationAngle;
uniform vec3 u_translation;
uniform vec3 u_center;
uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;

out vec4 v_color;

void main() {
    float cos_angle = cos(u_rotationAngle);
    float sin_angle = sin(u_rotationAngle);

    mat4 inverseTranslationMatrix = mat4(
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        -u_center.x, -u_center.y, 0.0, 1.0
    );

    mat4 translationMatrix = mat4(
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        u_center.x + u_translation.x, u_center.y +
            u_translation.y, u_center.z + u_translation.z, 1.0
    );

    mat4 rotationMatrix = mat4(cos_angle, 0.0, -sin_angle, 0.0,
        0.0, 1.0, 0.0, 0.0,
        sin_angle, 0.0, cos_angle, 0.0,
        0.0, 0.0, 0.0, 1.0
    );


    gl_Position = u_projectionMatrix * u_viewMatrix
        * translationMatrix * rotationMatrix *
            inverseTranslationMatrix * a_position;
    }

//Fragment Shader

#version 300 es
precision mediump float;
out vec4 color;
void main() {
    color = vec4(1.0, 1.0, 1.0, 1.0);
}
```

In the above shader attached to monkey head objects, a **model-view-projection** matrix is applied to the object that moves the object in the correct position on the screen.

2.3 Animations of the Monkey Heads

For all animations of the objects, linear interpolation (lerp) methods are utilized. The translating objects will go between two different points, and the rotation object will be rotated by an angle, both based on the time elapsed. Additionally, the monkey heads were positioned so that they are apart from each other, using the *setPosition()* method of the *GameObject* class.

For the leftmost monkey head, the z position (animation on z axis) is given by:

center = midpoint of the interpolation line

radius = (center + radius) and (center - radius) are the extreme points of the interpolation.

start = center - radius

end = center + radius

t = current application time

d = duration of the interpolation

if ((t mod 2) <= d):

f(t) = (t * (end - start) + d * start) / d

else:

f(t) = ((start-end) * (t-d) + end*d) / d

The z position of the object is given by the function of f(t).

For the rightmost object, the y position is given by the same f(t) function. For the middle object, the rotation is set to the time elapsed variable (t) using the *setRotation()* method of the *GameObject* class. Since the t variable keeps on increasing, the object keeps on rotating.

2.4 Perspective Camera

The perspective camera was implemented using several methods from the book. The perspective matrix of the camera was obtained using the *perspective(fovy, aspect, near, far)* function. The parameters *fovy, near and far* are custom values and *aspect* is the aspect of the canvas (width / height).

The view matrix of the camera was obtained using the *lookAt(eye, at, up)* function. The parameters *eye, at* are updated based on input wihle *up* is the global up vector (0.0, 1.0, 0.0) and is static.

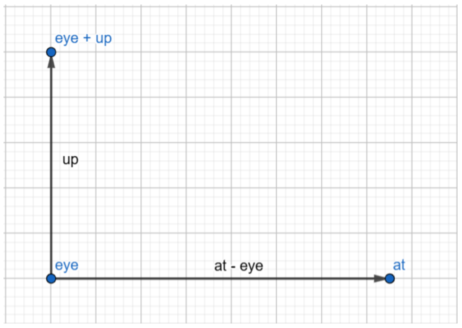

Figure 2: Visualization of the lookat() Parameters

As can be seen above, the *eye* parameter is the position of the camera, the *up* parameter is a vector always pointing upwards and the *at* parameter is a point in the direction of the camera. The subtraction *at - eye* gives the direction of the camera.

The camera is controlled by mouse inputs. When the left, middle or right mouse button is clicked, the application captures it and starts listening for mouse movement with pointer lock enabled.

- If the captured input is left mouse button click, the camera rotates around its axis. With horizontal mouse movement, the camera rotates horizontally and with vertical mouse movement, the camera rotate vertically. The vertical rotation is limited between 60 and -60 degrees from the zero point.
- If the captured input is middle mouse button click, the camera moves in the *at - eye* direction.
- If the captured input is right mouse button click, the camera is translated around the scene. The translation is relative to the camera itself, in other words it is horizontally or vertically with respect to the screen.

The camera rotation is controlled by two variables: *cameraHorizontalRotation* and *cameraVerticalRotation*. *camerHorizontalRotation* angle rotates the **at** point around the **up** vector while the *cameraVerticalRotation* angle adjusts the y coordinate of **at**.

The camera translation is controlled by an offset vector *cameraOffset*. This vector is added directly to the **eye** and **at** points after the orientation of **at** is completed.

References

[1] `https://webglfundamentals.org/webgl/lessons/webgl-load-obj.html`

[2] `https://interactivecomputergraphics.com/8E/Code%20update/Common`

[3] `https://www.youtube.com/watch?v=zUpJ2vx3wes&t=1028s`