BBM414 Introduction to Computer Graphics Lab. Practical # 2- Shape Drawing and Basic Shading

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

Overview

In this experiment, the concept of drawing shapes and using shaders to assign colors is being explored. The experiment consists of two parts.

In the first part, a source code that produces a gasket of red color is given. The task was to edit the code such that the gasket is multi-colored. More specifically, every triangle that make up the gasket should be of color red, green or blue based on some pattern.

In the second part, the task is to redraw the umbrella from Experiment 1 (1) and let the user adjust its *state* via keyboard keys. More specifically, the umbrella will have 3 different states: *reset* state, *move* state and *color* state. The user will be able to switch between these states using keyboard keys. The key 'r' switches to the *reset* state where the umbrella is stationary. The key 'm' switches to the *move* state where the umbrella is rotated depending on the position change of the mouse cursor. Finally, the key 'c' switches to the *color* state in which the umbrella will rotate just like in the *move* state, but now it's fabric will switch to a random color every frame. In the other two states, the color of the fabric is yellow.

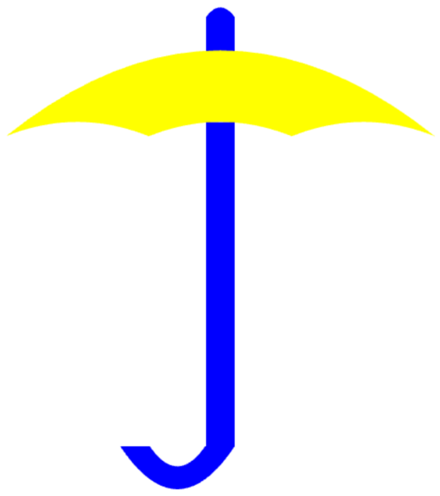
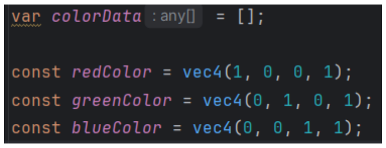


Figure 1: The umbrella to be redrawn

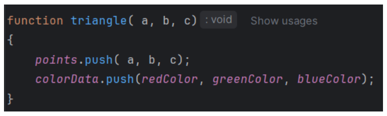
1 Part 1 - Coloring the Gasket

The approach was to first declare a *colorData* array and declare the Vector4 representations of colors red, green and blue (2a). After that, the colors in this order were added to the *colorData* array each time three triangle vertices are added to the main vertex buffer (2). In the end, a newly created vertex buffer that is based on *colorData* is created and is connected with the *a_Color* input attribute of the vertex shader which is also newly added.



```
var colorData :any[]  = [];

const redColor = vec4(1, 0, 0, 1);
const greenColor = vec4(0, 1, 0, 1);
const blueColor = vec4(0, 0, 1, 1);
```

(a) Newly added members.



```
function triangle( a, b, c) :void   Show usages
{
    points.push( a, b, c);
    colorData.push(redColor, greenColor, blueColor);
}
```

(b) Changed the triangle() function to include a new line of code.

Figure 2: Changed parts of code.

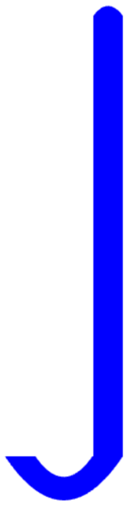
The below code is the changed shader code. In vertex shader, input attribute *a_Color* is added in order for the gasket to take in multiple different colors at once. This variable is then output to the fragment shader for the colorization of the gasket.

```
//Vertex Shader
#version 300 es
in vec4 vPosition;
in vec4 a_Color;

out vec4 v_Color;
void
main()
{
    gl_Position = vPosition;
    v_Color = a_Color;
}
//Fragment Shader
#version 300 es
precision mediump float;
out vec4 outColor;
in vec4 v_Color;

void
main()
{
    outColor = v_Color;
}
```

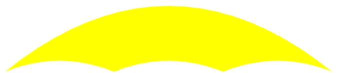
2 Part 2 - Implementing an Umbrella with Different States

2.1 Redrawing the Umbrella using Ear Clipping

In this experiment, a polygon triangulation algorithm known as ear clipping is used to draw the two different parts of the umbrella (3). First, the handle; then the fabric will be drawn for the correct blending to take place.



(a) The handle of the umbrella.



(b) The fabric of the umbrella.

Figure 3: Two parts of the umbrella.

The ear clipping algorithm used in the implementation takes in the vertices of the polygon (as Vector2 objects) the be drawn as input and outputs vertices that when formed into consecutive triangles, creates the polygon.

It is important to note that the precondition for the algorithm expects input vertices to be given in clockwise direction. In other words, the vertex input array can be partitioned into two arrays so that for one array, the x coordinates of the inputs are non-decreasing and for the other array, they are non-increasing. For example, a correct input order for the below polygon (4) is {2, 3, 4, 5, 1}.

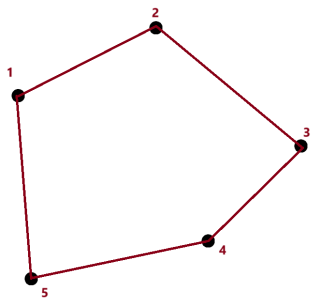


Figure 4: The polygon to be drawn

The algorithm, at each step, iteratively searches for three consecutive vertices that are vertices of a triangle which lies fully inside the polygon the be drawn. Such triangles are called *ears*. If an ear is correctly spotted, that ear gets cut off and another recursive search begins on the polygon represented by the remaining vertices. If at any step, no ear is found, the given polygon cannot be triangulated and the algorithm returns no vertices. For example, the below ear {2, 3, 4} is spotted (5), gets cut off, and a recursive search on the remaining polygon {2, 4, 5, 1} begins.

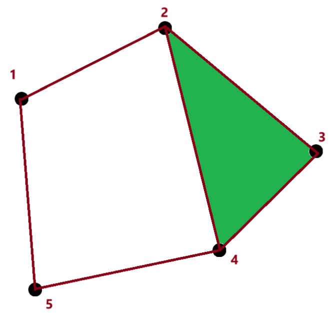
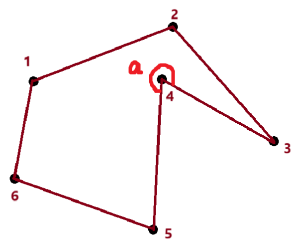


Figure 5: An ear of the polygon

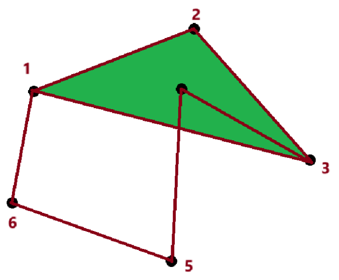
For three vertices to represent an ear, two conditions need to be satisfied:

1. The angle (inside the polygon) formed by the three vertices need to be less than 180 degrees (6a) .

2. No other vertex of the polygon is positioned inside the created triangle (6b) .



(a) The angle is bigger than 180 degrees.



(b) Triangle is outside the polygon.

Figure 6: Examples of invalid ear vertices.

If given polygon vertices represent a valid polygon, the algorithm will correctly find ears until there are three vertices left. Three vertices represent the base case, so they will be directly added and the triangulation of the polygon finishes.

In the implementation, the vertices that represent the handle and fabric of the umbrella are given as Bezier Curve points which are implemented in Experiment 1, in a clockwise fashion.

2.2 Adding States

The implementation keeps track of the current state using the *mode* global variable. There are also *angle* and *rotationChange* global variables that keep track of the rotation information of the umbrella. Variable *angle* is the rotation angle of the umbrella around the z-axis while *rotationChange* is the change applied to *angle* per frame.

Since the user controls the state of the umbrella via keystrokes, an event listener of type *keyup* is assigned to the document (7a). The listener function will take the key provided by the user, and set the global variable *mode* to the key if it is ’r’, ’m’ or ’c’.

The implementation also includes the method *render()* which is called every frame. The method is responsible for redrawing the umbrella, changing the rotation of it via *angle* variable, and changing the color of the umbrella depending on the current mode, depicted by the global variable *mode*. The *angle* variable is applied the change ’*rotationChange* * *deltaTime*’ at each call of the function. The multiplication with the *deltaTime* ensures smooth and equivalent animation across screens with different frame rates.

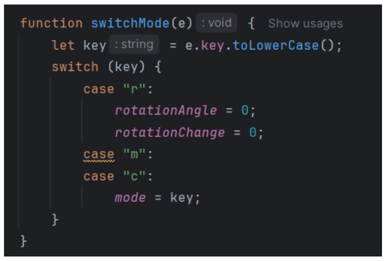
2.2.1 Reset Mode (R)

The global variable *mode* is equal to ’r’ in this mode. Since the umbrella should be stationary in this mode, *angle* and *rotationAngle* is set to zero. This is done in the *keyup* listener function. Also, the render function paints the umbrella fabric to yellow in this mode.

2.2.2 Move Mode (M)

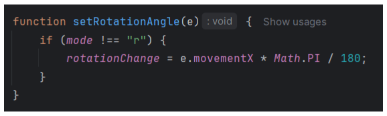
The global variable *mode* is equal to ’m’ in this mode. To implement this mode, an event listener of type *mousemove* ,which records the difference in mouse cursor position from the latest frame to the current one, is required (7b). The function of this listener , if the mode is not ’r’, will set the *rotationChange* variable which corresponds to change in the rotation angle (which corresponds to the *angle* variable) of the umbrella around the z-axis. The rotation is handled in the vertex shader (shown below) via the uniform *u_angle* which is set to *angle* in the *render()* function and the universal rotation matrix about the z-axis (named *rotationMatrix* in the shader).

```
//Vertex Shader
#version 300 es
precision mediump float;
in vec4 a_position;

uniform vec4 u_color;
out vec4 v_color;

uniform float u_angle;

void main() {

    float cos_angle = cos(u_angle);
    float sin_angle = sin(u_angle);

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

    gl_Position = rotationMatrix * a_position;
    v_color = u_color;
}
```



```
function switchMode(e) : void  {  Show usages
    let key : string  = e.key.toLowerCase();
    switch (key) {
        case "r":
            rotationAngle = 0;
            rotationChange = 0;
        case "m":
        case "c":
            mode = key;
    }
}
```

(a) Listener for keyup.



```
function setRotationAngle(e) : void  {  Show usages
    if (mode !== "r") {
        rotationChange = e.movementX * Math.PI / 180;
    }
}
```

(b) Listener for mousemove.

Figure 7: Listener Functions

2.2.3 Color Mode (C)

The global variable *mode* is ’c’ in this mode. Everything mode ’m’ allows is also allowed in color mode, so the setting of the *rotationChange* in the *mousemove* listener is only not performed when the mode is ’r’. When the *render()* function redraws the umbrella, it determines the color to be a randomly chosen one if the global variable *mode* is equal to ’c’.

References

Your references here.

[1] `https://www.youtube.com/watch?v=QAdfkylpYwc`

[2] `https://developer.mozilla.org/en-US/docs/Web/API/Element/mousemove_event`

[3] `https://developer.mozilla.org/en-US/docs/Web/API/Element/keyup_event`