BBM414 Introduction to Computer Graphics Lab.
Practical # 1- Drawing An Umbrella

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

Overview

In this assignment, we were tasked to draw the image of the umbrella provided below (1) using WebGL. The umbrella consists of a handle (in blue color) and fabric (in yellow color).

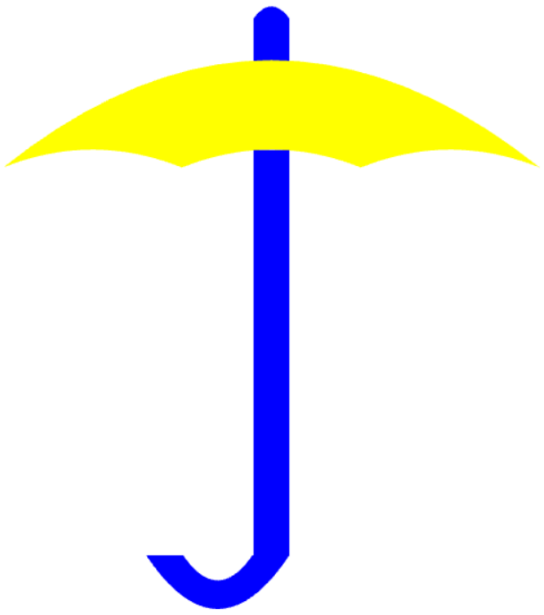

Figure 1: The umbrella to be drawn

1 Part 1 - Implementation Details

In the implementation, the image was broken down to two parts: the handle and the fabric (2). It can be observed that if we first draw the handle, then draw the fabric, the handle part behind the fabric will not be seen by the camera.

(a) The handle of the umbrella.

(b) The fabric of the umbrella.

Figure 2: Two parts of the umbrella.

This is utilized in the implementation, where first all the vertices needed to produce the handle are gathered in the data array, then the vertices needed to produce the fabric are appended. The vertices include 2 floats corresponding to the position and 4 floats corresponding to the color. The color components of the handle correspond to the blue color, and in the fabric they correspond to yellow. After the data is gathered, the vertex and fragment shaders are compiled into a WebGL program.

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

in vec4 a_position;
in vec4 a_color;
out vec4 v_color;

void main() {
    gl_Position = a_position;
    v_color = a_color;
}
```

The vertex shader is designed to take two position floats and assign them to *a_position*, and to take four color floats and assign them to *a_color*. In the main function, the vertex positions are generated and *a_color* is output to the fragment shader.

```
//Fragment Shader
#version 300 es
precision mediump float;

in vec4 v_color;
out vec4 color;

void main() {
    color = v_color;
}
```

The fragment shader is designed only to take the color attribute from the vertex shader and output it to the fragment processor.

After the shader compilation is done, a vertex buffer object (VBO) is created using the data array that holds all vertices. The configurations needed to pass the correct data to the correct shader attributes are done with the WebGL method *gl.vertexAttribPointer*. Finally, the whole data array in the VBO is drawn using a single draw call. The following two parts explain how the vertex data is generated for the handle and fabric parts.

2 Part 2 - Drawing the Handle

2.1 Intuition

The handle of the umbrella is further broken down to 3 parts as shown below : a semicircle-like shape at the top, a quad in the middle and semi-ring-like shape at the bottom (3).

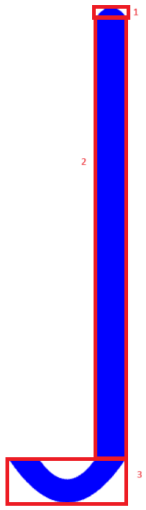

Figure 3: Parts of the handle

To draw the quad in the middle, the implementation needs the center point, the width and the height of it. Using these, the 4 corner points of the quad are generated and 2 triangles that make up the quad are drawn using them.

To draw a quad with the center (x,y), of width w and height h, the points

$$(x - w/2, y + h/2)$$
$$(x - w/2, y - h/2)$$
$$(x + w/2, y + h/2)$$

are connected to form the first triangle. Likewise, the points

$$(x + w/2, y + h/2)$$
$$(x + w/2, y - h/2)$$
$$(x - w/2, y - h/2)$$

are connected to form the second triangle. These two triangles make up the quad.

To draw the semicircle-like shape at the top, the implementation utilizes Bézier Curves with 3 two-dimensional control points P_0 P_1 and P_2.

$$L_0(t) = (1 - t)P_0 + tP_1$$
$$L_1(t) = (1 - t)P_1 + tP_2$$
$$Q_0(t) = (1 - t)L_0(t) + tL_1(t)$$
$$0 <= t <= 1$$

Here, $L_0(t)$ is the linear interpolation (lerp) function between P_0 and P_1. The function outputs a 2D point in the line segment P_0P_1 according to the t value. Likewise, $L_1(t)$ is the lerp function between P_1 and P_2. Since $L_0(t)$ and $L_1(t)$ are also 2D points, $Q_0(t)$ is defined to be the lerp function between them. Moreover, the graph of $Q_0(t)$ over the interval $[0, 1]$ gives the Bézier Curve with the control points P_0 P_1 and P_2 (4).

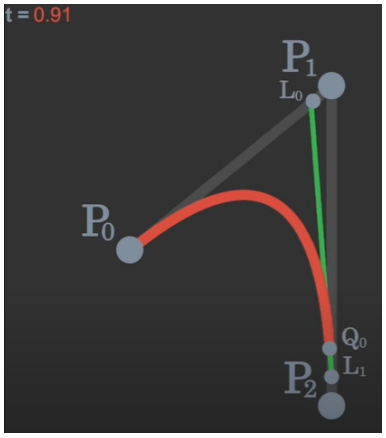

Figure 4: Bézier Curve

In order to draw a semicircle-like shape, the area between the curve and the line segment P_0P_2 needs to be filled in. For this, the implementation uses an algorithm to create a point C that divides P_0P_2 into two equal parts, and connects that point and each of the two sequential points in the curve into triangles. (5)

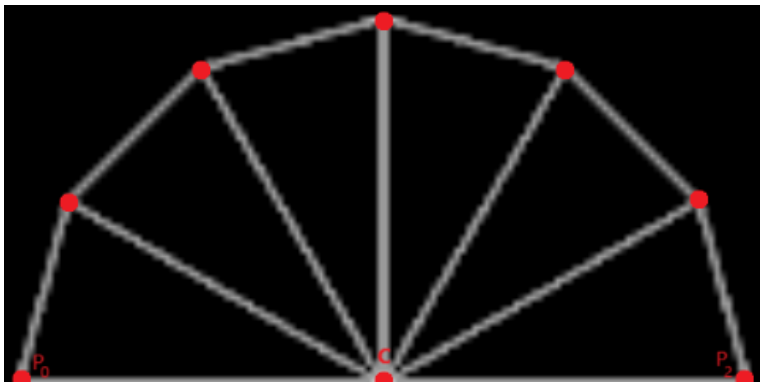

Figure 5: Triangulation Example

The red point marked as C is iteratively combined with all the other red marked points (points in the curve) to form triangles that fill the area. The more the triangles are, the smoother and more realistic the shape gets. Since generating every point in the curve is impossible, the best is to generate points as close each other as possible.

Lastly, the ring-like shape at the bottom needs to be drawn. This shape is equivalent to two semicircle-like shapes with the same center (the point C mentioned above), but one of them being smaller and having the same color as the background. The implementation first draws the big one, of blue color; then the smaller one of the background color. The smaller area subtracts a portion of the bigger area, creating a ring-like shape.

2.2 Implementation

The function below (6) adds the vertices needed for a quad with position, width and height attributes. The position is actually the center of the quad. This function is used to draw the quad in the middle of the handle.

```
function addQuadPosition(pos, width, height, colorID) {
    var x = pos.x;
    var y = pos.y;

    bufferData.push(x - width / 2, y + height / 2);
    addColor(colorID);

    bufferData.push(x + width / 2, y + height / 2);
    addColor(colorID);

    bufferData.push(x - width / 2, y - height / 2);
    addColor(colorID);

    bufferData.push(x + width / 2, y + height / 2);
    addColor(colorID);

    bufferData.push(x + width / 2, y - height / 2);
    addColor(colorID);

    bufferData.push(x - width / 2, y - height / 2);
    addColor(colorID);
}
```

Figure 6: Function to Add Quads

The function below (7) adds the vertices needed for a semicircle-like shape with position, diameter and height attributes. From these attributes, the control points P_0, P_1 and P_2 are calculated, the Bézier Curve between them is drawn and filled in. (8)

```
function addBezierCurveData(pos, diameter, height, colorID) {
    var x = pos.x;
    var y = pos.y;

    // linear interpolation lambda function.
    let lerp = (t, p0, p1) => new vec2(p0.x + t * (p1.x - p0.x), p0.y + t * (p1.y - p0.y));

    // p0, p1, p2 are three control points used to draw the semicircle-like object. c is the midpoint of p0 and p1.
    const p0 = new vec2(x - diameter/2, y - height);
    const p1 = new vec2(x + diameter/2, y - height);
    const c = new vec2((p0.x + p1.x) / 2, (p0.y + p0.y) / 2);
    const p2 = new vec2(c.x, y + height);

    // precision is the number of triangles to fill the semicircle. The more this value is, the smoother the curve gets
    const precision = 20;

    // q0 and q1 represent points in the bezier curve.
    var q0 = new vec2(p0.x, p0.y);
    for (var i = 1; i <= precision; i++) {
        var t = i/precision;
        var q1 = lerp(t, lerp(t, p0, p2), lerp(t, p2, p1));

        bufferData.push(q0.x, q0.y);
        addColor(colorID);

        bufferData.push(c.x, c.y);
        addColor(colorID);

        bufferData.push(q1.x, q1.y);
        addColor(colorID);

        q0 = q1;
    }
}
```

Figure 7: Function to Add a Semicircle-Like Shape

Inside the function, the control points and the center C is calculated. The local lambda function $lerp$ defines a lerp function between two points. The $precision$ local variable defines how many triangles to use to fill the area. There is a loop that is iterated $precision$ number of times, each time using $t = i/precision$ as the time variable to calculate the next point q_1 in the curve. q_0 holds the previous point in the curve. The points q_0, q_1, c are connected together to form a triangle inside the area.

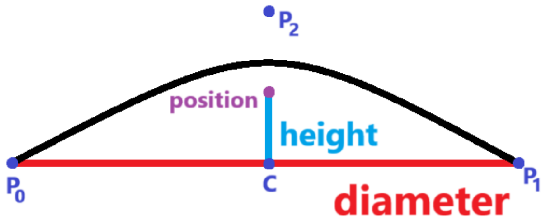

Figure 8: Semicircle-Like Shape Attributes and Control Points (NOTE: P_1 and P_2 swapped places, this is only a name change)

3 Part 3 - Drawing the Fabric

Like the handle, the fabric can also be broken down to smaller parts such as (9):

Figure 9: Parts of the fabric

The fabric contains a big semicircle-like shape of yellow color and 3 equally sized smaller ones of the same color as the background. The same procedure (7) to draw a semicircle-like shape can be used 4 times to completely draw the fabric.

However, when the fabric drawing procedure is finished, a problem occurs. A small part of the existing handle is deleted because the middle smaller semicircle-like shape gets drawn on top of a portion of the handle. This problem can be seen in the figure below (10):

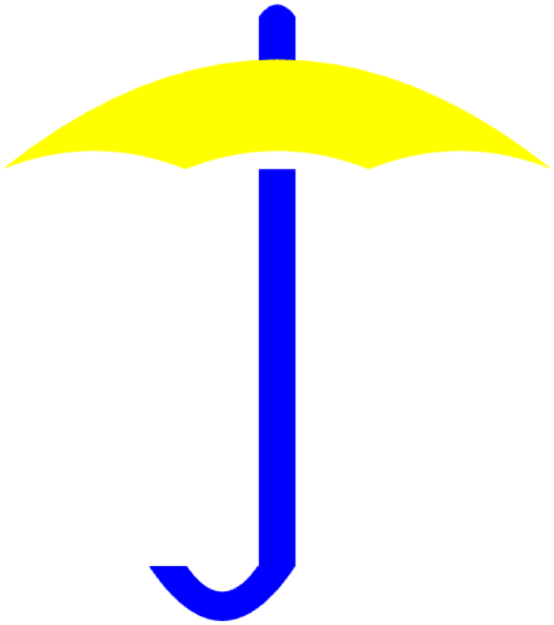

Figure 10: Error in Drawing

Redrawing that portion of the handle eliminates the problem.

References

Your references here.

[1] https://www.w3schools.com/js/js_objects.asp

[2] https://www.geeksforgeeks.org/lambda-expressions-in-javascript/

[3] https://stackoverflow.com/questions/44447847/enums-in-javascript-with-es6

[4] https://www.youtube.com/watch?v=pnYccz1Ha34

[5] https://stackoverflow.com/questions/75496846/circle-triangulation

[6] https://en.wikipedia.org/wiki/B%C3%A9zier_curve

[7] https://www.overleaf.com/learn/latex/Code_listing