BBM414 Computer Graphics Lab.
Practical # - Practical Info

Ahmet Uman
2210356129
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b2210356129@cs.hacettepe.edu.tr

Overview

The purpose of this assignment is to draw a simple umbrella shape. We aim to draw the curved sides of the umbrella using a bezier curve and fill the shape we have using triangulaziton methods. With this we will get familiar with WebGL2 canvas and learn simple drawing

1 Part 1 - Drawing Bezier Curves

The WebGL program we use for the course can only draw straight lines and triangles due to its structure. For more complex shapes, different algorithms are used. To draw curved objects, we used Bezier Curve, which is frequently used because it is efficient, easy to use and implement, and modular.

The quadratic Bezier curve is defined by three control points: P_0, P_1, and P_2. The parametric equation for a quadratic Bezier curve is:

$$B(t) = (1-t)^2 \cdot P_0 + 2(1-t)t \cdot P_1 + t^2 \cdot P_2, \quad \text{where } t \in [0,1]$$

- **At $t = 0$:**
$$B(0) = (1-0)^2 P_0 + 2(1-0)(0)P_1 + 0^2 P_2 = P_0$$
The curve starts at P_0.
- **At $t = 1$:**
$$B(1) = (1-1)^2 P_0 + 2(1-1)(1)P_1 + 1^2 P_2 = P_2$$
The curve ends at P_2.
- **Influence of P_1:** For $0 < t < 1$, the point P_1 influences the curvature of the Bézier curve, pulling it toward P_1 and determining its shape.

In my JavaScript function, I calculate the x and y coordinates separately using the quadratic Bezier formula:

```
function calculateQuadraticBezierPoints(p0, p1, p2, numPoints) {
    let points = [];
    for (let i = 0; i <= numPoints; i++) {
        let t = i / numPoints;
        let x = (1 - t) * (1 - t) * p0[0] + 2 * (1 - t) * t * p1[0] + t * t * p2[0];
        let y = (1 - t) * (1 - t) * p0[1] + 2 * (1 - t) * t * p1[1] + t * t * p2[1];
        points.push([x, y]);
```

```
    }
    return points;
}
```

$$x(t) = (1-t)^2 P_{0x} + 2(1-t)tP_{1x} + t^2 P_{2x}$$
$$y(t) = (1-t)^2 P_{0y} + 2(1-t)tP_{1y} + t^2 P_{2y}$$

Where:

- P_{0x}, P_{0y} are the x and y coordinates of the starting point P_0.
- P_{1x}, P_{1y} are the coordinates of the control point P_1.
- P_{2x}, P_{2y} are the coordinates of the ending point P_2.
- t is the parameter ranging from 0 to 1.

2 Part 2 - Fill Curvy Shapes Using Triangulaziton Methods

As I mentioned in the first part, WebGL can only draw the triangle shape. We will use triangles to fill in the curvy shapes we obtained with the Bezier curve.

First of all, we should keep in mind that the Bezier curves we created are not perfect curves, and that dozens of points (100 in my code) come together in the direction of the function to create a curve-like appearance. When we think of it, they are like a polygon consisting of 100 sides. Triangularization methods are used to divide polygons into triangle sets. In this regard, if we divide the curvy-like shapes we have into triangles and the insides of these triangles are colored, we can fill the inside of it with color.

Ear Clipping Method

2.1 Winding Order

```
function signedArea(polygon) {
    let area = 0;
    for (let i = 0; i < polygon.length; i++) {
        const [x1, y1] = polygon[i];
        const [x2, y2] = polygon[(i + 1) % polygon.length];
        area += (x1 * y2 - x2 * y1);
    }
    return area / 2;
}
```

This function calculates the signed area of a polygon using the Shoelace (or Gauss's area) formula. If the area value is positive, it means the winding order of the vertices is counter-clockwise; if the area is negative, the vertices are in clockwise order.

2.2 Ensure Winding Order

```
function ensureWindingOrder(polygon) {
    if (signedArea(polygon) < 0) {
        polygon.reverse();
    }
    return polygon;
}
```

In the previous section, we determined if a vertex is in clockwise or counterclockwise order. Using this function, if a vertex is in clockwise order, we will reverse the vertex order. The purpose is making

sure polygon vertices are in counter-clockwise order for consistency in calculations like determining convexity.

2.3 Calculate Internal Angle

```
function getInternalAngle(vector1, vector2) {
    const dotProduct = vector1[0] * vector2[0] + vector1[1] * vector2[1];
    const magnitude1 = Math.sqrt(vector1[0] ** 2 + vector1[1] ** 2);
    const magnitude2 = Math.sqrt(vector2[0] ** 2 + vector2[1] ** 2);
    return Math.acos(dotProduct / (magnitude1 * magnitude2));
}
```

This function calculates the internal angle between two vectors at a vertex using the dot product. We will use this function during the triangulation process.

The angle θ between two vectors $\vec{v_1}$ and $\vec{v_2}$ is calculated using the dot product formula:

$$\cos\theta = \frac{\vec{v_1} \cdot \vec{v_2}}{\|\vec{v_1}\|\|\vec{v_2}\|}$$

2.4 Check a Point Inside Triangle

```
function isPointInsideTriangle(triangle, point) {
    const [a, b, c] = triangle;
    const area = 0.5 * (-b[1] * c[0] + a[1] * (-b[0] + c[0]) + a[0] * (b[1] - c[1])
        + b[0] * c[1]);
    const s = 1 / (2 * area) * (a[1] * c[0] - a[0] * c[1] + (c[1] - a[1]) *
        point[0] + (a[0] - c[0]) * point[1]);
    const t = 1 / (2 * area) * (a[0] * b[1] - a[1] * b[0] + (a[1] - b[1]) *
        point[0] + (b[0] - a[0]) * point[1]);
    const u = 1 - s - t;
    return s > 0 && t > 0 && u > 0;
}
```

This function determine whether a given point lies inside a specified triangle. We will use this function in triangulate function.

Let the triangle be defined by its three vertices: points A, B, and C, with coordinates $A(x_a, y_a)$, $B(x_b, y_b)$, and $C(x_c, y_c)$. Let $P(x_p, y_p)$ be the point we want to test for inclusion within the triangle.

We calculate the barycentric coordinates s, t, and u of point P with respect to triangle $\triangle ABC$. The point P lies inside the triangle if and only if all the barycentric coordinates are positive and their sum is 1.

First, compute twice the area $2A$ of triangle $\triangle ABC$:

$$2A = x_a(y_b - y_c) + x_b(y_c - y_a) + x_c(y_a - y_b)$$

Compute the barycentric coordinates s, t, and u:

$$\begin{aligned} s &= \frac{1}{2A}\left[(y_b - y_c)(x_p - x_c) + (x_c - x_b)(y_p - y_c)\right] \\ t &= \frac{1}{2A}\left[(y_c - y_a)(x_p - x_c) + (x_a - x_c)(y_p - y_c)\right] \\ u &= 1 - s - t \end{aligned}$$

The point P lies inside triangle $\triangle ABC$ if:

$$s > 0, \quad t > 0, \quad u > 0$$

2.5 Triangulate

```
function triangulate(polygon) {
    const triangles = [];
    let vertices = ensureWindingOrder([...polygon]);

    while (vertices.length > 3) {
        let earFound = false;
        for (let i = 0; i < vertices.length; i++) {
            const prevIndex = (i - 1 + vertices.length) % vertices.length;
            const nextIndex = (i + 1) % vertices.length;

            const prevVertex = vertices[prevIndex];
            const vertex = vertices[i];
            const nextVertex = vertices[nextIndex];

            const vector1 = [vertex[0] - prevVertex[0], vertex[1] - prevVertex[1]];
            const vector2 = [nextVertex[0] - vertex[0], nextVertex[1] - vertex[1]];
            const angle = getInternalAngle(vector1, vector2);

            if (angle >= Math.PI) continue;

            const triangle = [prevVertex, vertex, nextVertex];
            if (!isAnyPointInside(triangle, vertices)) {
                triangles.push(triangle);
                vertices.splice(i, 1);
                earFound = true;
                break;
            }
        }

        if (!earFound) break;
    }

    if (vertices.length === 3) {
        triangles.push([vertices[0], vertices[1], vertices[2]]);
    }

    return triangles;
}
```

This function triangulates a polygon into a set of triangles using the ear clipping method.

The algorithm starts by ensuring the polygon has counter-clockwise winding and makes a copy of the vertices.

A while loop continues until the polygon is reduced to a triangle (3 vertices). It calculates previous, current, and next vertex indices using modulo arithmetic. Vectors are computed from the current vertex to the previous and next vertices. The internal angle at the current vertex is calculated using `getInternalAngle`. If the angle is less than π (convex vertex), it proceeds to check for ears.

A triangle is formed with the previous, current, and next vertices. `isAnyPointInside` is used to ensure no other polygon vertices lie inside the triangle. If a valid ear is found, it is added to the list of triangles, the current vertex is removed, and the loop restarts. If no ear is found in a complete iteration, the loop breaks to prevent infinite looping.

When only three vertices remain, they form the last triangle, which is added to the list.

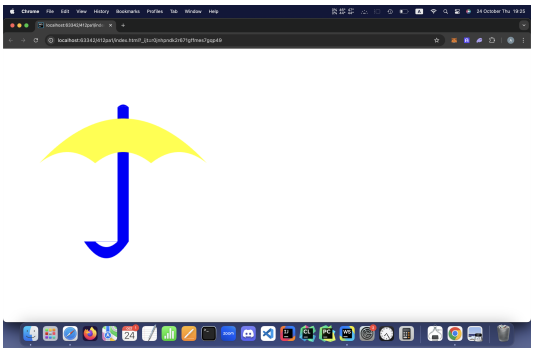

Figure 1: Screenshot of my project result from a web browser

References

[1] https://javascript.info/bezier-curve

[2] https://webglfundamentals.org/webgl/lessons/webgl-3d-geometry-lathe.html

[3] https://en.wikipedia.org/wiki/Barycentric_coordinate_system

[4] https://jtsorlinis.github.io/rendering-tutorial/

[5] https://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/

[6] https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf