BBM414 Computer Graphics Lab.
Assignment 1

Uygar ERSOY
2220356114
Department of Computer Engineering
Hacettepe University
Ankara, Turkey

Overview

This assignment demonstrates the use of WebGL2 to create a canvas and draw an umbrella with a white background. The umbrella handle and fabric are constructed with different colors, where the border points are generated using the Bézier curve algorithm and the interior is triangulated using polygon triangulation (ear clipping) algorithm.

1 Part 1 - Code Explanation

In this section, the key components of the code used to draw the umbrella is described.

1.1 Triangle Area Calculation and Point Inclusion

The `areaOfATriangle` function calculates the area of a triangle given three points (`point1`, `point2`, `point3`) in a 2D space. These points are expected to have `x` and `y` coordinates, and the function employs a mathematical formula derived from the determinant of a matrix that represents the triangle. This formula ensures that the area is positive regardless of the order in which the points are provided. The absolute value (`Math.abs()`) is used because the determinant can be negative depending on the orientation (clockwise or counterclockwise) of the points, but area must always be a positive value. [1]

```
function areaOfATriangle(point1, point2, point3) {
    return Math.abs((point1.x * (point2.y - point3.y) +
                     point2.x * (point3.y - point1.y ) +
                     point3.x * (point1.y - point2.y)) /2.0);
}
```

The next function, `isPointInTriangle`, uses the area calculation to check if a given point lies inside a triangle formed by three other points. It compares the area of the triangle with the sum of areas formed by the point and the triangle's vertices. Three different triangles are formed with the given parameter, point, with other parameters, namely, v1, v2, v3. If this point lies inside of the triangle formed by v1-v2-v3, then the sum of the area of those three triangles will be equal to the area of the v1-v2-v3 triangle. With the help of `areaOfATriangle` function, `isPointInTriangle` function checks this condition.

```
function isPointInTriangle(point, v1, v2, v3) {
    const v1v2v3 = areaOfATriangle(v1, v2, v3);
    const pointV1V2 = areaOfATriangle(point, v1, v2);
    const pointV1V3 = areaOfATriangle(point, v1, v3);
    const pointV2V3 = areaOfATriangle(point, v2, v3);

    return (pointV1V2 + pointV1V3 + pointV2V3) === v1v2v3;
}
```

1.2 Ear Clipping Triangulation

The `earClippingTriangulation` function implements the ear clipping algorithm to decompose a polygon into triangles. It iterates over the vertices, identifies ears, and ensures no other points are inside these ears. The `isCounterClockwise` function ensures proper vertex ordering.

The `earClippingTriangulation` function is designed to divide a polygon into triangles using the ear clipping algorithm. It starts by converting the input vertices into pairs of (x, y) coordinates, then iteratively looks for "ears"—triangles formed by three consecutive vertices that do not contain any other points from the polygon inside them. To find these ears, the function checks the orientation of each triplet of vertices to ensure they form a valid triangle using a counterclockwise test. It then verifies that no other polygon vertices fall inside the triangle using the isPointInTriangle function. When a valid ear is found, its vertices are stored in a result array, and the ear's vertex is removed from the polygon. The process repeats until the polygon is reduced to just three vertices, which form the final triangle. This algorithm ensures that the entire polygon is decomposed into a set of non-overlapping triangles suitable for rendering. [2, 3]

```
function earClippingTriangulation(vertices) {
    let pairedVertices = pairVertices(vertices);
    const triangles = [];
    while (pairedVertices.length > 3) {
        for (let i = 0; i < pairedVertices.length; i++) {
            const previous = pairedVertices[(i - 1 + pairedVertices.
                length) % pairedVertices.length];
            const previousIndex = (i - 1 + pairedVertices.length) %
                pairedVertices.length;
            const current = pairedVertices[i];
            const next = pairedVertices[(i + 1) % pairedVertices.
                length];
            const nextIndex = (i + 1) % pairedVertices.length;

            if (isCounterClockwise(previous, current, next)) {
                let flag = true;
                for (let j = 0; j < pairedVertices.length; j++) {
                    if (j !== i && j !== previousIndex && j !==
                        nextIndex) {
                        if (isPointInTriangle(pairedVertices[j],
                            previous, current, next)) {
                            flag = false;
                            break;
                        }
                    }
                }
                if (flag) {
                    triangles.push(previous.x, previous.y);
                    triangles.push(current.x, current.y);
                    triangles.push(next.x, next.y);
                    pairedVertices.splice(i, 1);
                    break;
                }
            }
        }
    }
    triangles.push(pairedVertices[0].x, pairedVertices[0].y);
    triangles.push(pairedVertices[1].x, pairedVertices[1].y);
    triangles.push(pairedVertices[2].x, pairedVertices[2].y);
    return triangles;
}
```

The `isCounterClockwise` function checks if three vertices v0, v1, and v2 form a counterclockwise orientation by calculating the signed area of the triangle they form. This is based on the cross product of the vectors from v0 to v1 and from v0 to v2. A positive result means the points are counterclockwise, ensuring correct vertex ordering for triangulation. [4]

```
function isCounterClockwise (v0, v1, v2) {
    return (v1.x - v0.x) * (v2.y - v0.y) -
           (v1.y - v0.y) * (v2.x - v0.x) > 0;
}
```

1.3 Bézier Curve Generation

The function that generates the curved segments of the umbrella uses a quadratic Bézier curve, which is defined by three points: a start point, a control point, and an end point. The curve is calculated by interpolating between these points based on a parameter t, which ranges from 0 to 1. As t increases, the formula blends the points to create a smooth curve. The function computes multiple points along this curve by iterating over values of t, resulting in a set of vertices that approximate the curved parts of the umbrella. [5]

```
function generateBezierPoints(startPoint, controlPoint, endPoint) {
    const points = [];
    for (let i = 0; i <= 100; i++) {
        const t = i / 100;
        const x = Math.pow(1 - t, 2) * startPoint.x +
                  2 * (1 - t) * t * controlPoint.x +
                  Math.pow(t, 2) * endPoint.x;
        const y = Math.pow(1 - t, 2) * startPoint.y +
                  2 * (1 - t) * t * controlPoint.y +
                  Math.pow(t, 2) * endPoint.y;
        points.push(x, y);
    }
    return points;
}
```

1.4 Combining Curves for the Umbrella

The `createSmallSetOfCurves` function creates multiple Bézier curves to form the umbrella's bottom fabric part. It computes the start, control, and end points for each segment and combines them into a smooth border. Returned smallerCurvesPoint array contains the vertices of three continuous curved segment of the fabric part of the umbrella. Given values are given arbitrarily by the programmer to match the given shape.

```
function createSmallSetOfCurves() {
    const smallerCurvesPoints = [];

    for (let i = 0; i < 3; i++) {
        const segmentStartX = -0.7 + (1.4) * (i / 3);
        const segmentEndX = -0.7 + (1.4) * ((i + 1) / 3);
        const segmentStartPoint = { x: segmentStartX, y: 0.25 };
        const segmentControlPoint = { x: (segmentStartX + segmentEndX)
             / 2.0, y: 0.35 };
        const segmentEndPoint = { x: segmentEndX, y: 0.25 };

        const segmentBezierPoints = generateBezierPoints(
           segmentStartPoint, segmentControlPoint, segmentEndPoint);
        segmentBezierPoints.splice(0, 2);
        smallerCurvesPoints.push(...segmentBezierPoints);
    }
    return smallerCurvesPoints;
}
```

1.5 Example: Applying Bézier Curve and Ear Clipping for Drawing

The following example illustrates how the previous functions are utilized to generate and render a curved segment of the umbrella using WebGL. In this case, the `generateBezierPoints` function computes a series of points along a quadratic Bézier curve, and the `earClippingTriangulation` function is applied to triangulate these points, enabling them to be rendered as triangles in WebGL.

```
const topCurveVertices = generateBezierPoints(
    { x: 0.02, y: 0.6 },
    { x: 0.0, y: 0.65 },
    { x: -0.02, y: 0.6 }
);

const topCurveVerticesTriangles = earClippingTriangulation(
    topCurveVertices);

const topCurveBuffer = initBuffer(gl, topCurveVerticesTriangles);

gl.bindBuffer(gl.ARRAY_BUFFER, topCurveBuffer);
gl.vertexAttribPointer(positionAttributeLocation, 2, gl.FLOAT, false,
    0, 0);
gl.drawArrays(gl.TRIANGLES, 0, topCurveVerticesTriangles.length / 2);
```

In this example, the curve is defined by three key points: the start, control, and end points. The `generateBezierPoints` function generates 100 points along the curve, and these are passed into `earClippingTriangulation` to decompose the curve into triangles. The resulting triangulated vertices are stored in a buffer and rendered using WebGL's `drawArrays` method, which draws the umbrella's top curved segment.

Conclusion

The combination of Bézier curves for the umbrella's border and ear clipping triangulation for filling the interior allows for a dynamic and smooth rendering of the umbrella shape using WebGL2. This approach highlights the power of geometric algorithms in computer graphics.

References

[1] https://www.cuemath.com/geometry/area-of-triangle-in-coordinate-geometry/

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

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

[4] https://en.wikipedia.org/wiki/Curve_orientation

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