BBM414 Computer Graphics Lab. Programming Assignment #1 - WebGL2 Canvas and Simple Drawing

Muzaffer Berke Savaş
2220356044
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b2220356044@cs.hacettepe.edu.tr

Overview

The aim of this assignment is to draw an umbrella shape using Bézier curves formulas and the polygon triangulation algorithm in WEBGL2.

1 Bézier Curves and Polygon Triangulation Algorithm Implementations

The quadratic Bézier curve is defined by three control points P_0, P_1, and P_2. The following code calculates the points on the curve using the quadratic Bézier formula:

```
function bezier_quadratic(p0,p1,p2) {
    const points =[];
    for(let t = 0; t <= 1 ; t += 0.001){
        const x = (((1 - t) ** 2) * p0[0]) + (2 * (1-t) * t * p1[0]) + ((t ** 2) * p2[0]);
        const y = (((1 - t) ** 2) * p0[1]) + (2 * (1-t) * t * p1[1]) + ((t ** 2) * p2[1]);
        points.push(x,y);
    }
    return points;
}
```

Parameters: The function takes three control points $p0$, $p1$, and $p2$, each represented as an array with two elements: $[x, y]$.

Loop: A loop iterates over the parameter t from 0 to 1 in steps of 0.001. For each value of t, the function calculates the x and y coordinates of the point on the curve using the quadratic Bézier equation:

$$B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$$

The linear Bézier curve is defined by two control points P_0 and P_1, and the following code calculates the points along this line:

```
function bezier_linear(p0,p1) {
    const points =[];
    for(let t = 0; t <= 1 ; t += 0.001){
        const x = p0[0] + (t * (p1[0] - p0[0]));
        const y = p0[1] + (t * (p1[1] - p0[1]));
        points.push(x,y);
    }
    return points;
}
```

The following code implements a basic polygon triangulation algorithm by generating the indices that form the triangles from a set of points:

```
function triangulation(points) {
    const indices = [];
    for (let i = 2; i < points.length / 2; i++) {
        indices.push(0, i - 1 , i);
    }
    return indices;
}
```

Loop: The `for` loop iterates over the points starting from the third point (index 2). The loop condition is `i < points.length / 2`, where the length is halved because each point has two coordinates (x and y).

Triangle Formation: In each iteration, the function adds three indices to the `indices` array: the index 0 (representing the first point), `i - 1`, and `i`. This forms a triangle by connecting the current point `i` and the previous point `i-1` to the first point.

2 Bézier Curves and Polygon Triangulation Usages

This code block sample defines and renders the top part of a handle using a quadratic Bézier curve in WebGL. The curve is created by the `bezier_quadratic` function with three 2D control points, and the resulting points are stored in `bezierPoints_0`. The `triangulation` function is applied to generate index data (`triangulationIndexes_0`) for triangle-based rendering. Two buffers are initialized: one for the Bézier curve points (`Float32Array`) and one for the triangulation indices (`Uint16Array`). Finally, the `draw` function renders the shape using these buffers with a black color [0.0, 0.0, 0.0, 1.0].

```
const bezierPoints_0 = bezier_quadratic( p0: [-0.03,0.9], p1: [0.0,1.0], p2: [0.03,0.9]);
const triangulationIndexes_0 = triangulation(bezierPoints_0);                    // Top of the handle using bezier_quadratic curve
initBuffer(gl,bezierPoints_0,Float32Array);
initBuffer(gl,triangulationIndexes_0,Uint16Array);
draw(gl,positionAttributeLocation,colorUniformLocation,triangulationIndexes_0.length, color: [0.0, 0.0, 0.0, 1.0]);
```

Figure 1: Bézier Quadratic Curve Sample

This code defines and renders a rectangular shape using Bézier curves and triangulation in WebGL. Four Bézier lines are created using the `bezier_linear` function, each defined by two 2D points, representing the top, bottom, left, and right edges of the rectangle. These lines are combined into the `bezierLines` array. The `triangulation` function is then applied to these lines to generate index data for triangle-based rendering. Two buffers are initialized: one for the Bézier line coordinates (`Float32Array`) and one for the triangulation indices (`Uint16Array`). Finally, the `draw` function renders the rectangle with a black color [0.0, 0.0, 0.0, 1.0].

```
const bezierLine_1 :[]  = bezier_linear( p0: [-0.03, 0.9],  p1: [0.03, 0.9]);
const bezierLine_2 :[]  = bezier_linear( p0: [-0.03, -0.7],  p1: [0.03, -0.7]);
const bezierLine_3 :[]  = bezier_linear( p0: [-0.03, 0.9],  p1: [-0.03, -0.7]);
const bezierLine_4 :[]  = bezier_linear( p0: [0.03, 0.9],  p1: [0.03, -0.7]);          // Handle rectangle part using bezier_linear function
const bezierLines :any[]  = [];
bezierLines.push(...bezierLine_1);
bezierLines.push(...bezierLine_2);
bezierLines.push(...bezierLine_3);
bezierLines.push(...bezierLine_4);
const triangulationIndexes :[]  = triangulation(bezierLines);
initBuffer(gl, bezierLines, Float32Array);
initBuffer(gl, triangulationIndexes, Uint16Array);
draw(gl, positionAttributeLocation, colorUniformLocation, triangulationIndexes.length,  color: [0.0, 0.0, 0.0, 1.0]);
```

Figure 2: Bézier Linear Curve Sample

3 Initializing the Buffer, Draw Function and Final Shape

This `initBuffer` function initializes a WebGL buffer based on the specified `arrayType`. It first creates a new buffer using `gl.createBuffer`. If the `arrayType` is `Float32Array`, the function binds the buffer to `gl.ARRAY_BUFFER` and stores the given `positions` in it as a float array using `gl.bufferData` for static drawing. If the `arrayType` is `Uint16Array`, it binds the buffer to `gl.ELEMENT_ARRAY_BUFFER`, computes the triangulation of the `positions`, and stores the indices in the buffer as a `Uint16Array` for static drawing. The function returns the initialized buffer in both cases.

```
function initBuffer(gl, positions, arrayType) :{..}|undefined {                    // Initialize the buffer with respect to arrayType

  const buffer :AudioBuffer|WebGLBuffer  = gl.createBuffer();


  if (arrayType === Float32Array){

    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

    return {
      position : buffer
    };
  } else if (arrayType === Uint16Array){

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffer);

    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(triangulation(positions)), gl.STATIC_DRAW);

    return {
      position: buffer
    };
  }
}
```

Figure 3: InitBuffer Function

This function, `draw`, is responsible for rendering shapes in WebGL. It first sets up the vertex attribute pointer using `gl.vertexAttribPointer`, which defines how vertex data is pulled from the buffer, specifying 2 components per vertex (for 2D coordinates) and using floating-point data. The vertex attribute array is then enabled with `gl.enableVertexAttribArray`. The function also sets the shape's color by passing the `color` array to `gl.uniform4fv`, which applies a uniform color to the shape. Finally, it renders the shape as triangles using `gl.drawElements`, with indices of type `UNSIGNED_SHORT` and the specified `length` for the number of elements.

```
function draw(gl, positionAttributeLocation, colorUniformLocation, length, color) :void { Show usages

  gl.vertexAttribPointer(positionAttributeLocation, size: 2, gl.FLOAT, normalized: false, stride: 0, offset: 0);
  gl.enableVertexAttribArray(positionAttributeLocation);
  gl.uniform4fv(colorUniformLocation, color);
  gl.drawElements(gl.TRIANGLES, length, gl.UNSIGNED_SHORT, offset: 0);

}
```

Figure 4: Draw Function

Final shape of Umbrella looks like :

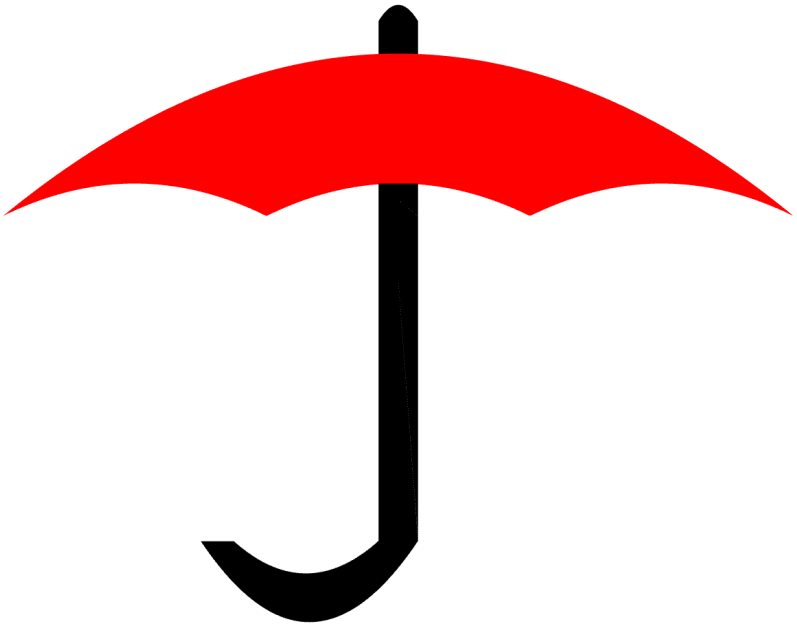

Figure 5: Final Shape of Umbrella

References

[1] `https://en.wikipedia.org/wiki/BÃľzier_curve`

[2] `https://en.wikipedia.org/wiki/Polygon_triangulation`

[3] BBM412 Computer Graphics ~Programming with WEBGL PDF