BBM414 Computer Graphics Lab.
Assignment 2

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

Overview

This project explores WebGL2 techniques for rendering and animation. Part one focuses on generating a multicolored Sierpinski Gasket by recursively subdividing triangles and applying vertex colors. Part two builds on an umbrella shape, adding interactive animations: keyboard inputs switch modes, and mouse movement controls rotation and color changes, showcasing basic transformations and shader use for dynamic effects. Together, these parts highlight essential WebGL2 skills in rendering and interactivity.

1 Part 1 - Sierpinski Gasket Code Analysis

The following section analyzes the WebGL2 code for rendering a colorful Sierpinski Gasket [1]. This fractal pattern is created by recursively subdividing a triangle and assigning colors to each vertex.

1.1 Setup and Initialization

First, the code initializes the WebGL2 context on an HTML canvas and sets up the variables needed for rendering, including arrays for storing vertices and colors.

```
var canvas;
var gl;

var points = [];
var colors = [];
var NumTimesToSubdivide = 5;
```

The `NumTimesToSubdivide` variable sets the recursion depth for subdividing the triangle, determining the level of detail in the Sierpinski Gasket.

1.2 Defining Vertices and Colors

The main triangle's vertices and colors are defined. Each corner is assigned a unique color (red, green, blue), which will blend across the subdivided triangles.

```
var vertices = [
    vec2(-1, -1),
    vec2(0, 1),
    vec2(1, -1)
];
```

```
var vertexColors = [
    vec4(1.0, 0.0, 0.0, 1.0),
    vec4(0.0, 1.0, 0.0, 1.0),
    vec4(0.0, 0.0, 1.0, 1.0)
];
```

The `vertices` array represents the main triangle's corners, and `vertexColors` assigns a color to each vertex.

1.3 Recursive Subdivision of Triangles

The `divideTriangle` function recursively subdivides the triangle into smaller triangles by finding midpoints. The recursion ends when `count` reaches zero.

```
function divideTriangle(a, b, c, count, vertexColors) {
    if (count === 0) {
        triangle(a, b, c, vertexColors[0], vertexColors[1],
            vertexColors[2]);
    } else {
        var ab = mix(a, b, 0.5);
        var ac = mix(a, c, 0.5);
        var bc = mix(b, c, 0.5);

        --count;
        divideTriangle(a, ab, ac, count, vertexColors);
        divideTriangle(c, ac, bc, count, vertexColors);
        divideTriangle(b, bc, ab, count, vertexColors);
    }
}
```

This function finds midpoints between vertices and calls itself with smaller triangles, resulting in the Sierpinski pattern.

1.4 Storing and Rendering Triangles

The completed triangles are stored in arrays and rendered using WebGL. The `triangle` function pushes vertices and colors to the `points` and `colors` arrays.

```
function triangle(a, b, c, colorA, colorB, colorC) {
    points.push(a, b, c);
    colors.push(colorA, colorB, colorC);
}
```

The `render` function clears the canvas and draws the triangles using `gl.drawArrays`.

```
function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, points.length);
}
```

By repeatedly drawing triangles and applying vertex colors, the code renders a multicolored Sierpinski Gasket.

—

This section provides an overview of the Sierpinski Gasket code, explaining how recursion, buffer management, and shader attributes are combined to generate a dynamic fractal pattern.

1.5 Shader Programs

The rendering process in WebGL2 requires a vertex shader and a fragment shader. These shaders work together to process vertex data and assign colors to the rendered triangles.

1.5.1 Vertex Shader

The vertex shader processes each vertex, determining its position on the screen and passing color data to the fragment shader.

```
#version 300 es
in vec4 vPosition;
in vec4 vColor;
out vec4 vColorOut;

void main()
{
    gl_Position = vPosition;
    vColorOut = vColor;
}
```

In this shader: - `#version 300 es` specifies the shader version, making it compatible with WebGL2. - `in vec4 vPosition` is an input variable that receives the vertex position data from the WebGL program. - `in vec4 vColor` receives color data for each vertex. - `out vec4 vColorOut` passes the color data to the fragment shader. - The `main` function assigns `vPosition` to `gl_Position` to determine the vertex's position, and forwards `vColor` to `vColorOut`.

1.5.2 Fragment Shader

The fragment shader determines the final color of each fragment (pixel) in the triangle.

```
#version 300 es
precision mediump float;
in vec4 vColorOut;
out vec4 outColor;

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

In this shader: - `precision mediump float;` sets the precision for floating-point calculations. - `in vec4 vColorOut` receives color data from the vertex shader. - `out vec4 outColor` is the output color of each fragment. - The `main` function assigns `vColorOut` to `outColor`, ensuring that each fragment's color matches the interpolated vertex colors, creating a smooth color transition across the triangles.

These shaders work together to create the Sierpinski Gasket by processing each vertex's position and color and passing the interpolated colors to produce a gradient effect across the fractal pattern.

2 Part 2 - Umbrella Rendering and Interaction Analysis

This section describes the code that renders an umbrella shape in WebGL2 and provides user-controlled rotation and color-changing functionality. The code initializes the canvas, configures the shaders and buffers, and sets up user interactions via keyboard and mouse.

2.1 Canvas and WebGL2 Context Initialization

The `main` function begins by obtaining a WebGL2 context for rendering and clearing the canvas with a white background color.

```
const canvas = document.getElementById("glCanvas");
const gl = canvas.getContext("webgl2");

if (!gl) {
    alert("Webgl2 is not supported!");
```

```
    return;
}

gl.clearColor(1.0, 1.0, 1.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

2.2 Shader Program Initialization

Two shader programs are initialized: `program1` for the umbrella's body and handle, and `program2` for the fabric part, allowing it to change color independently.

```
const program1 = initProgram(gl, vsSource, fsSourceRed);
const program2 = initProgram(gl, vsSource, fsSourceYellow);
```

2.3 Vertices and Curves for the Umbrella Shape

Various segments of the umbrella are represented as vertex arrays. The umbrella's components include the rectangle for the body, the top curve, and additional curves for the fabric and handle. Each component is formed using the `generateBezierPoints` function for smoother curved shapes.

```
const rectangleVertices = new Float32Array([
    0.02, 0.6,
    -0.02, 0.6,
    -0.02, -0.3,
    0.02, -0.3
]);

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

Each component is subsequently passed through an ear-clipping triangulation algorithm to convert it into triangles suitable for rendering.

2.4 User Interaction for Rotation and Color Change

Keyboard and mouse events control the umbrella's rotation and color change: - Pressing the `r` key resets rotation and color change. - Pressing the `m` key enables rotation based on horizontal mouse movement. - Pressing the `c` key enables both rotation and random color changes for the fabric.

```
window.addEventListener("keydown", (event) => {
   if (event.key === "r") {
       rotationEnabled = false;
       rotationAngle = 0;
       lastMouseX = null;
       colorFlag = false;
   } else if (event.key === "m") {
       rotationEnabled = true;
   } else if (event.key === "c") {
       colorFlag = true;
       rotationEnabled = true;
   }
});
```

2.5 Mouse Event Listener for Rotation

The rotation of the umbrella is controlled by the user's horizontal mouse movement. The `mousemove` event listener tracks the mouse's position and adjusts the rotation angle of the umbrella accordingly.

The mouse's position relative to the canvas is calculated by subtracting the canvas's position from the mouse's `clientX` coordinate using `getBoundingClientRect()` [2]. The difference (`deltaX`) between the current and previous mouse positions is computed, which determines the horizontal movement. This difference is then scaled by a factor of `0.01` to control the rotation speed. If the mouse moves to the right (`deltaX > 0`), the rotation angle increases, and if the mouse moves to the left (`deltaX < 0`), the angle decreases. The `rotationAngle` is updated accordingly, resulting in a smooth real-time rotation of the umbrella based on horizontal mouse movement.

```
canvas.addEventListener("mousemove", (event) => {
    if (rotationEnabled) {
        const canvasRectangle = canvas.getBoundingClientRect();
        const mouseX = event.clientX - canvasRectangle.left;

        if (lastMouseX !== null) {
            const deltaX = - mouseX + lastMouseX;
            const speed = Math.abs(deltaX) * 0.01;
            rotationAngle += deltaX > 0 ? speed : -speed;
        }
        lastMouseX = mouseX;
    }
});
```

2.6 Rendering the Umbrella

The `draw` function renders the umbrella by iterating through each part in the `umbrellaParts` array. For each part, it calculates the rotation matrix based on `rotationAngle` using `sin` and `cos`, and applies the rotation around the center point defined by `centerPoint`. The translation is handled by updating the `uTranslation` uniform, ensuring parts rotate around this center.

For the fabric part, if the `colorFlag` is set, a random color is generated and passed to the fragment shader using the `uColor` uniform. Otherwise, the fabric retains its default color. Each part is drawn individually with `gl.drawArrays` after binding its corresponding buffer and setting the necessary attributes [3].

The `requestAnimationFrame` [4] function ensures continuous updates to the canvas, enabling smooth animations as the umbrella responds to user inputs like mouse movement for rotation and key presses for color changes.

```
umbrellaParts.forEach(part => {
    gl.useProgram(part.program);
    const cos = Math.cos(rotationAngle);
    const sin = Math.sin(rotationAngle);

    const rotationLoc = gl.getUniformLocation(part.program, "uRotation
        ");
    const translationLoc = gl.getUniformLocation(part.program, "
        uTranslation");

    if (part.program === program2) {
        const fabricColorLoc = gl.getUniformLocation(part.program, "
            uColor");
        gl.uniform4fv(fabricColorLoc, fabricColor);
    }

    gl.uniform2f(rotationLoc, sin, cos);
    gl.uniform2f(translationLoc, centerPoint.x, centerPoint.y);

    const positionLoc = gl.getAttribLocation(part.program, "aPosition
        ");
    gl.enableVertexAttribArray(positionLoc);
    gl.bindBuffer(gl.ARRAY_BUFFER, part.buffer);
    gl.vertexAttribPointer(positionLoc, 2, gl.FLOAT, false, 0, 0);
```

```
    gl.drawArrays(gl.TRIANGLES, 0, part.count);
});
```

2.7 Shaders

The shaders used in this project consist of a vertex shader and fragment shaders for different colors.

2.7.1 Vertex Shader

The vertex shader is responsible for transforming the vertex positions using a rotation and translation matrix. It takes the input vertex position (`aPosition`), applies the rotation using the `uRotation` uniform (representing the rotation matrix components), and adds the translation vector `uTranslation` to the result. The transformed position is then assigned to `gl_Position`.

```
const vsSource = `#version 300 es
    in vec2 aPosition;
    uniform vec2 uRotation;
    uniform vec2 uTranslation;
    void main() {
        float x = aPosition.x * uRotation.y - aPosition.y * uRotation.
            x;
        float y = aPosition.x * uRotation.x + aPosition.y * uRotation.
            y;
        vec2 position = vec2(x,y) + uTranslation;
        gl_Position = vec4(position, 0.0, 1.0);
    }
`;
```

2.7.2 Fragment Shaders

The fragment shaders determine the color output for the pixels.

The `fsSourceYellow` shader sets the color of the fabric to a uniform color (`uColor`) passed from the JavaScript code. It is used to color the umbrella fabric with a random or predefined color.

```
const fsSourceYellow = `#version 300 es
    precision highp float;
    uniform vec4 uColor;
    out vec4 fragColor;
    void main() {
        fragColor = uColor;
    }
`;
```

The `fsSourceRed` shader sets a fixed red color for other parts of the umbrella. The color is hardcoded in the fragment shader, rendering a specific shade of red.

```
const fsSourceRed = `#version 300 es
    precision highp float;
    out vec4 fragColor;
    void main() {
        fragColor = vec4(169.0/255.0, 4.0/255.0, 50.0/255.0, 1.0);
    }
`;
```

3 Conclusion

This project demonstrates the use of WebGL2 for rendering an interactive umbrella model with dynamic rotation and color changes. Through the implementation of vertex and fragment shaders, the umbrella responds to user inputs, including mouse movements for rotation and keyboard events

for color modification. The project highlights the capabilities of WebGL2 in creating real-time, interactive graphics with user-driven transformations.

References

1. https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle
2. https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect
3. https://webgl2fundamentals.org/
4. https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame