BBM414 Computer Graphics Lab.
Programming Assignment 2 # - 2024

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

Overview

The purpose of this assignment is to get familiar with simple translation and rotation using shader language. We will get keyboard keys and mouse movements from user and make an animation using the defined functions with respect to the given keys and mouse curser position.

1 Part 1

In this part we worked on provided 2D Sierpinski gasket. Our purpose was altering the color of the triangles from a single red color to an RGB color scheme.

```
<script id="vertex-shader" type="x-shader/x-vertex">#version 300 es
  in vec4 a_position;
  in vec4 a_color;
  out vec4 color;
  void main() {
       gl_Position = a_position;
       color = a_color;
}
</script>
```

The vertex shader in this setup is designed to manage the position and color of each vertex in 2D space. It takes in two main inputs: a-position, an attribute variable representing each vertex's position as a vec4 (where typically only the x and y coordinates are used, with z set to zero and w to one for 2D rendering), and a-color, another attribute variable representing the color of each vertex in vec4 format (RGBA, where each component ranges from 0.0 to 1.0 for red, green, blue, and alpha channels). The shader outputs color, a varying variable that is passed to the fragment shader and contains the vertex color. This color is then interpolated across the triangle surface during rasterization, creating a gradient effect between vertices.

The shader's techniques include utilizing attribute variables to receive vertex data from the buffer and using varying variables to pass interpolated data to the fragment shader. The position of each vertex is directly assigned to gl-Position, the built-in variable representing the vertex's final location in clip space, thus determining its on-screen position. Additionally, the a-color attribute is passed to the color variable, which the fragment shader will use to set each pixel's color within the triangle. Through this approach, the vertex shader defines each vertex's position and color, laying the groundwork for rendering shapes with gradient effects across surfaces.

```
<script id="fragment-shader" type="x-shader/x-fragment">#version 300 es
precision mediump float;
in vec4 color;
out vec4 outColor;

void main() {
    outColor = color;
}

</script>
```

The fragment shader in this setup is responsible for determining the final color of each pixel within a triangle. It receives color, a varying variable from the vertex shader that represents the interpolated color of each pixel. This interpolation is automatically handled by WebGL, which blends the vertex colors across the triangle's surface, resulting in smooth color transitions. The shader outputs outColor, the final color for each pixel (or fragment) that is written to the framebuffer, making it visible on the canvas.

The shader employs a few key techniques, including color interpolation and precision specification. The line precision mediump float; sets a medium precision for floating-point calculations in the fragment shader, offering a balance between performance and accuracy. The shader's main task is straightforward: it assigns the interpolated color received from the vertex shader to outColor, the output variable, defining the color of each pixel. This simplicity allows the shader to efficiently render a gradient effect across each triangle based on vertex colors.

```
function divideTriangle( a, b, c, count )
{

    // check for end of recursion

    if ( count === 0 ) {
        var color = all_colors[count % 3]
        triangle( a, b, c , color);
    }
    else {

        //bisect the sides

        var ab = mix( a, b, 0.5 );
        var ac = mix( a, c, 0.5 );
        var bc = mix( b, c, 0.5 );


        --count;

        // three new triangles

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

The divideTriangle function is designed to recursively subdivide an initial large triangle into smaller triangles, generating the fractal pattern known as the Sierpinski Gasket. It begins with three vec2 vertices, a, b, and c, that define the corners of the triangle. These vertices initially form a large equilateral triangle, but as the function recurses, they represent progressively smaller triangles. The function also takes an integer count, which specifies the number of times the triangle should be subdivided, determining the depth of recursion and thus the detail level of the fractal pattern.

The function operates by modifying two global arrays, points and colors, to store the vertices and colors of each triangle generated through the subdivision process. Through recursive subdivision, the function divides each triangle into smaller triangles until it reaches the smallest subdivision level, indicated by count reaching zero. To achieve this, it first calculates midpoints on each edge of the triangle using the mix function, which linearly interpolates between two vertices. This midpoint calculation forms new points that act as vertices for the smaller triangles. If count is zero, the function calls triangle(a, b, c, color), a helper function that appends the vertices and colors for the smallest triangles to points and colors arrays, finalizing their position in the fractal.

In the recursive case, if count is greater than zero, the function calculates the midpoints between vertices, producing three new points: ab, ac, and bc. It then decreases the recursion depth by one and makes three recursive calls, each with one of the smaller triangles formed by the vertices a, ab, ac, c, ac, bc, and b, bc, ab. Through this method, divideTriangle builds the Sierpinski Gasket, where the triangles reduce in size with each recursion level. Colors alternate through the RGB palette across the fractal pattern, creating visually distinct layers in the final rendering.

2 Part 2

2.1 Step 1 - Switching Default Position

The reset functionality triggered by pressing the "R" key allows users to restore the shape to its default state, resetting both its rotation angle and color settings. This feature is activated by a JavaScript keydown event listener that detects the "R" key press (event.key === "r" || event.key === "R"), ensuring responsiveness to both lowercase and uppercase "R". When triggered, the function resets the shape's rotation and color, providing a quick way to return it to its original appearance on the canvas.

The reset function has two primary outputs: resetting the rotation angle and restoring the color. First, the rotation angle is reset by setting rotationAngle to zero, which halts any active rotation by also setting rotationSpeed to zero. This aligns the shape to its default orientation, stopping further rotation until the user manually reactivates it.

For color resetting, the function addresses dynamic color changes by setting colorChangeMode to false, which stops any ongoing color transitions. Additionally, the time variable, used for continuous color modulation, is reset to zero, ensuring that if color transitions are later reactivated, they begin from the starting point rather than continuing from a previous state. The shape's color is then set to a predefined default, typically a static blue, by setting both staticColor and dynamicColor to this color, ensuring a uniform appearance.

In summary, when the "R" key is pressed, the event listener initiates a reset function that halts rotation by zeroing rotationAngle and rotationSpeed, stops dynamic color transitions by setting colorChangeMode to false, resets time to zero, and applies a default static color across the shape. This functionality provides a simple way to revert the shape to a known default state, creating a reliable baseline for subsequent interactions.

2.2 Step 2 - Rotation

The rotation functionality enables the shape to spin smoothly around its central axis on the canvas, with the rotation speed and direction controlled by the horizontal position of the cursor. The functionality is activated by pressing the "M" key, which toggles the spinning mode, allowing users to start or stop the rotation as desired. Additionally, the "R" key can be pressed at any point to reset the rotation, bringing the shape back to its initial orientation.

When the spinning mode is active, continuous rotation occurs around the shape's center point. The rotation speed and direction are directly influenced by the mouse's horizontal position relative to the canvas center. The further the cursor is from the center, the faster the shape spins, creating a responsive experience that enhances user interactivity.

To enable this, an event handling setup uses both keydown and mousemove listeners. Here's the key event handling code:

```
let spinningMode = false;
let rotationSpeed = 0;
const canvasCenterX = canvas.width / 2 + canvas.getBoundingClientRect().left;

document.addEventListener("keydown", (event) => {
    if (event.key === "m" || event.key === "M") {
        spinningMode = !spinningMode;
        if (!spinningMode) {
            rotationSpeed = 0;
        }
    }
});

document.addEventListener("mousemove", (event) => {
    if (spinningMode) {
        const distanceFromCenterX = event.clientX - canvasCenterX;
        const sensitivity = 0.0001;
        rotationSpeed = distanceFromCenterX * sensitivity;
    }
});
```

To apply this rotation, the vertex shader receives a rotation matrix calculated based on the current rotation angle. This rotation matrix is calculated in the calculateRotationMatrix function, which generates a 4x4 matrix based on the angle in radians. This matrix is then passed as a uniform variable, u-rotationMatrix, to the vertex shader, rotating each vertex around the shape's center point:

```
let rotationAngle = 0;

function calculateRotationMatrix(angle) {
    const cosA = Math.cos(angle);
    const sinA = Math.sin(angle);
    return new Float32Array([
        cosA, -sinA, 0, 0,
        sinA, cosA, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
    ]);
}
```

In each frame of the render loop, if spinningMode is true, rotationAngle is updated by adding rotationSpeed, allowing the shape to rotate smoothly. The calculateRotationMatrix function recalculates the matrix based on the updated angle, and the matrix is passed to the vertex shader as a uniform:

```
function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);

    if (spinningMode) {
        rotationAngle += rotationSpeed;
    }

    const rotationMatrix = calculateRotationMatrix(rotationAngle);

    const uRotationMatrix = gl.getUniformLocation(program, "u_rotationMatrix");
    gl.useProgram(program);
    gl.uniformMatrix4fv(uRotationMatrix, false, rotationMatrix);

    drawShape(gl, program, rotationMatrix);

    requestAnimationFrame(render);
}
```

The rotation of the shape around its center is achieved by leveraging a vertex shader in WebGL. The vertex shader receives a rotation matrix (u-rotationMatrix) as a uniform variable, which applies a transformation to each vertex of the shape. Rotation matrix is calculated in the main JavaScript code based on the rotationAngle, which changes continuously as the shape rotates. The matrix itself is generated in the calculateRotationMatrix function and updates on each frame in the render loop, allowing rotation around the shape's center point. The vertex shader multiplies each vertex position by the rotation matrix, effectively rotating the entire shape about the origin. Since WebGL's coordinate system is centered at the origin (0, 0), this method naturally rotates the shape around its central point. By handling rotation in the vertex shader, we ensure a continuous rotation effect that updates without problem in response to user interactions.

```
const vsSource = `
    attribute vec4 a_position;
    uniform mat4 u_rotationMatrix;

    void main() {
        gl_Position = u_rotationMatrix * a_position;
    }
`;
```

2.3 Step 3 - Color Changing

```
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));
}
```

The continuous color-changing function creates an effect that allows the shape to cycle through colors over time. By pressing the "C" key, which switches the colorChangeMode variable between true and false. When colorChangeMode is enabled, the colors cycle one by one; when disabled, the shape supposed to freeze at its current color. The problem is I couldn't figure how do it out, in my app handle, bar and fabric (whole shape) changes color.

The input driving this color change is a time-based calculation. Each frame in the render loop increments a time variable, which is used to transfer the color values through trigonometric functions, creating oscillations in the red, green, and blue channels.

To handle user input, an event listener for the "C" key toggles the color-changing mode:

```
let colorChangeMode = false;
document.addEventListener("keydown", (event) => {
    if (event.key === "c" || event.key === "C") {
        colorChangeMode = !colorChangeMode;
    }
});
```

Each frame, the time variable increments by a small constant (e.g., 0.04) if colorChangeMode is true. The red, green, and blue channels are calculated using Math.sin functions with different phase shifts to make color change:

```
let time = 0;
let dynamicColor = [0.0, 0.0, 1.0];
```

```
function updateColor() {
    time += 0.1; // Increment time to drive color change
    dynamicColor = [
        Math.abs(Math.sin(time * 10)),
        Math.abs(Math.sin((time + Math.PI / 3) * 10)),
        Math.abs(Math.sin((time + 2 * Math.PI / 3) * 10))
    ];
}
```

In the render loop, if colorChangeMode is enabled, updateColor is called each frame to recalculate dynamicColor based on the latest time value. The updated color is then passed as a uniform to the fragment shader, which applies the color to the entire shape:

```
function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);

    if (colorChangeMode) {
        updateColor();
    }

    const uColorLocation = gl.getUniformLocation(program, "u_color");
    gl.useProgram(program);
    gl.uniform4f(uColorLocation, dynamicColor[0], dynamicColor[1], dynamicColor[2],
         1.0);

    drawShape(gl, program);

    requestAnimationFrame(render);
}
```

In the fragment shader, the u-color uniform variable is used to apply the color to each pixel of the shape. The color is set via uniform4f, which assigns the current value of dynamicColor to the shader, producing the desired effect (which is in my opinion one of the most important part we implemented with vertex shader):

```
const fsSource = ‘
    precision mediump float;
    uniform vec4 u_color;
    uniform bool u_dynamicColorMode;
    uniform vec3 u_dynamicColor;

    void main() {
        vec4 color = u_dynamicColorMode ? vec4(u_dynamicColor, 1.0) : u_color;
        gl_FragColor = color;
    }
‘;
```

And there are some screenshot from outcome:

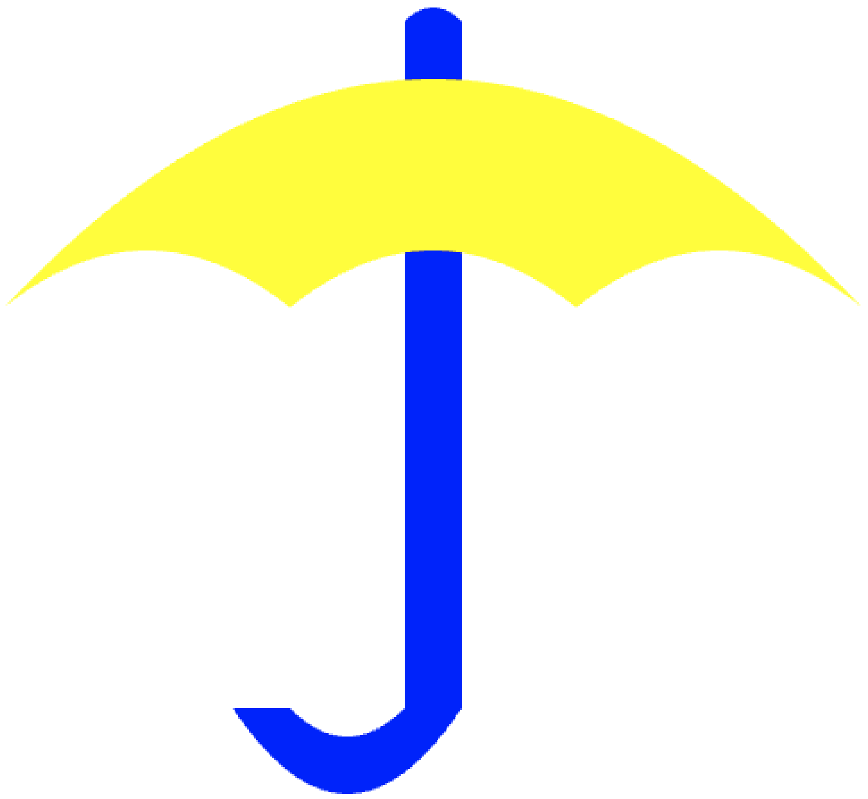

Figure 1: Default

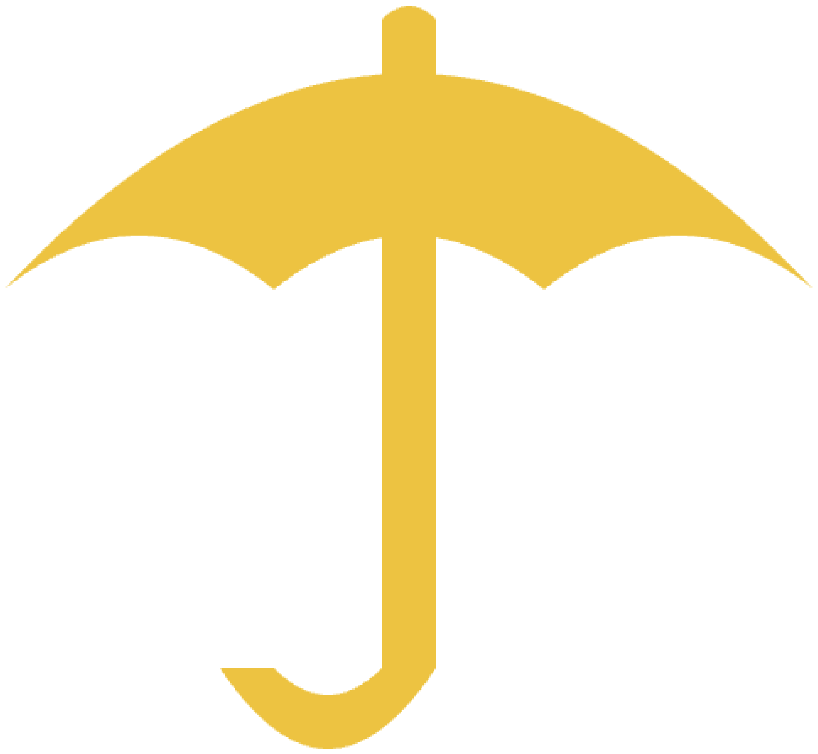

Figure 2: Color Change

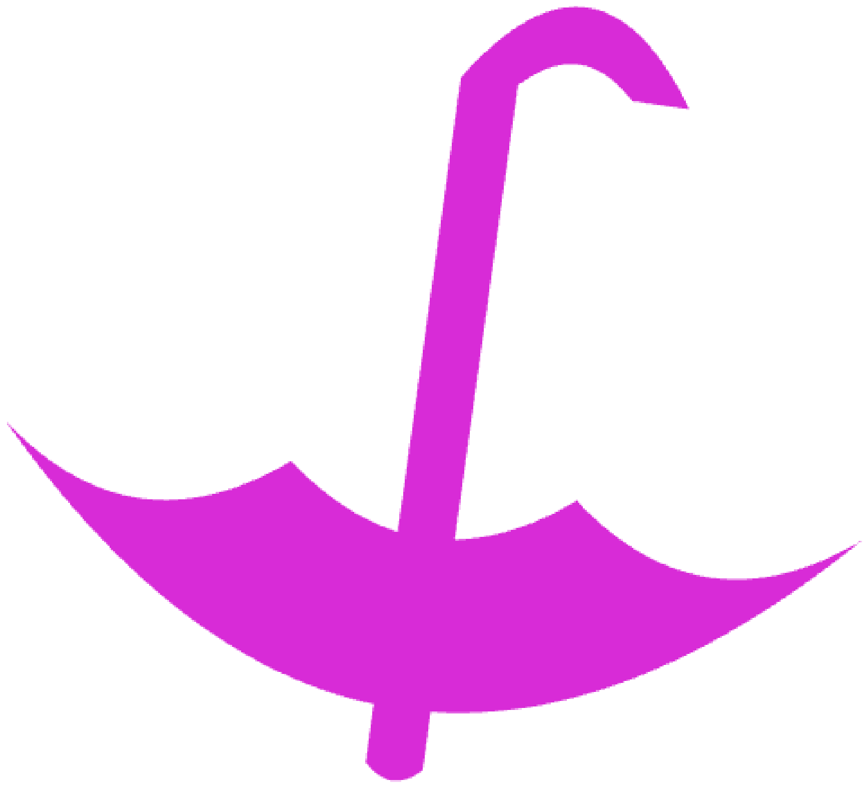

Figure 3: Rotation with Color Change