BBM414 Computer Graphics Lab.
Programming Assignment 3 # - 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 scaling, translation and rotation using shader language. We will get GUI inputs and mouse clicks and movements from user and make an animation using the defined functions with respect to the given inputs and mouse cursor position.

1 Part 1

With the help of the square shaped buttons given to us in the first part of the assignment, we increased and decreased the rotation speed, changed its color to create different gradients and toggled the rotation direction.

```
in vec4 vPosition;
in vec4 vColor;
uniform float theta;

out vec4 fColor;

void main() {
    float s = sin(theta);
    float c = cos(theta);

    gl_Position.x = -s * vPosition.y + c * vPosition.x;
    gl_Position.y = s * vPosition.x + c * vPosition.y;
    gl_Position.z = 0.0;
    gl_Position.w = 1.0;

    fColor = vColor;
}
```

Vertex shaders are used to control the transformation of vertices. Here we use it to control the rotations of the vertices. While doing this, we get help from the following formula:

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

```
function toggle() {
```

```
    if (direction === 1) {
        direction = -1;
    } else {
        direction = 1;
    }
}

function adjustSpeed(adding) {
    speed = Math.max(0.02, speed + adding);
}

function change_color() {
    colors = [
        vec4(Math.random(), Math.random(), Math.random(), 1.0),
        vec4(Math.random(), Math.random(), Math.random(), 1.0),
        vec4(Math.random(), Math.random(), Math.random(), 1.0),
        vec4(Math.random(), Math.random(), Math.random(), 1.0)
    ];

    var colorBufferId = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBufferId);
    gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW);

    var vColor = gl.getAttribLocation(gl.getParameter(gl.CURRENT_PROGRAM),
        "vColor");
    gl.vertexAttribPointer(vColor, 4, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(vColor);
}
```

```
var colorBufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBufferId);
gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW);

var vColor = gl.getAttribLocation(program, "vColor");
gl.vertexAttribPointer(vColor, 4, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vColor);
```

We provide the implementations requested from us in the assignment through the above functions. Thanks to the Math.max function, we do not fall below the minimum rotation speed. With the change color function, the color gradient defined randomly with the buffers above is redefined and we get a new gradient image.

2 Part 2

2.1 Step 1 - Shaders

First of all, the shaders I have prepared look like the ones below. I wanted to put it at the top since I will be referring to it here frequently in the subsections where I explain the buttons. Fragment shader is not much different from what we use in other projects, but vertex shader is not like that.

The raw coordinates of each point are sent to this shader as an attribute called a_position. Transformations such as scaling (u_scale), rotation (u_rotation) and moving (u_offset) are applied to these coordinates. Shader performs these operations respectively: First, the shape is enlarged or reduced by multiplying the a_position value with the scale (u_scale). Then, all points of the shape are rotated by a certain angle using the rotation matrix. Finally, the move value (u_offset) is added to all coordinates, thus shifting the shape to the desired position. As a result of these operations, the shader prints the final coordinates to the gl_Position variable and WebGL draws this information on the screen.

```
const vertexShaderSource = `
```

```
attribute vec2 a_position;
uniform vec2 u_offset;
uniform float u_rotation;
uniform float u_scale;
void main() {
    // Rotation matrix
    float cosTheta = cos(u_rotation);
    float sinTheta = sin(u_rotation);
    mat2 rotationMatrix = mat2(
        cosTheta, -sinTheta,
        sinTheta, cosTheta
    );

    // Scale and rotate
    vec2 scaledPosition = a_position * u_scale;
    vec2 rotatedPosition = rotationMatrix * scaledPosition;

    // Apply translation
    gl_Position = vec4(rotatedPosition + u_offset, 0.0, 1.0);
}‘;

const fragmentShaderSource = ‘
precision mediump float;
uniform vec4 u_color;
void main() {
    gl_FragColor = u_color;
}‘;
```

2.2 Step 2 - Draw Button

It allows the user to activate or deactivate the drawing mode. In case the drawing mode is active (drawModeActive=true), the user can draw by holding down the mouse. In this case, the points that the user clicks and drags are added to the canvas and saved in the points array. These points are sent to the vertex shader with the a_position attribute in WebGL and the shape is made to appear on the screen. When drawing is stopped (drawModeActive=false), no new points are added and the canvas remains fixed. During the drawing process on the shader side, each new point is scaled, rotated and shifted under the influence of uniform variables such as u_scale, u_rotation and u_offset.

```
function addPoint(e) {
    const rect = canvas.getBoundingClientRect();
    const x = ((e.clientX - rect.left) / canvas.width) * 2 - 1;
    const y = -(((e.clientY - rect.top) / canvas.height) * 2 - 1);

    points.push(x, y);

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

    redrawScene();
}

document.getElementById("drawButton").addEventListener("click", () => {
    drawModeActive = !drawModeActive;
    if (drawModeActive) {
        document.getElementById("drawButton").textContent = "Stop Drawing";
    } else {
        document.getElementById("drawButton").textContent = "Start Drawing";
        isDrawing = false;
    }
});

canvas.addEventListener("mousedown", (e) => {
    if (!drawModeActive) return;
```

```
    isDrawing = true;
    addPoint(e);
});

canvas.addEventListener("mouseup", () => isDrawing = false);
canvas.addEventListener("mousemove", (e) => {
    if (!isDrawing || !drawModeActive) return;
    addPoint(e);
});
```

2.3 Step 3 - Color

It allows the user to select both line and fill color. When the user selects a color, this color is transferred to the currentColor variable in RGBA format. This color is then passed to the u_color uniform variable in the fragment shader. This will ensure that the entire shape is painted with the new color. Both the line (gl.LINE_STRIP) and fill (gl.TRIANGLE_FAN) are redrawn using this new color. Thus, the user can change both the current line color and fill color at any time.

```
document.getElementById("colorPicker").addEventListener("input", (e) => {
    const hexColor = e.target.value;
    currentColor = [
        parseInt(hexColor.slice(1, 3), 16) / 255,
        parseInt(hexColor.slice(3, 5), 16) / 255,
        parseInt(hexColor.slice(5, 7), 16) / 255,
        1.0,
    ];
    redrawScene();
});
```

2.4 Step 4 - Move Buttons

Motion buttons (moveUpButton, moveDownButton, moveLeftButton, moveRightButton) are used to shift the shape in a certain direction. When these buttons are clicked, the x or y values in the offset array are increased or decreased. For example, the y value is increased for an upward movement, while it is decreased for a downward movement. These offset values are passed to the vertex shader through the u_offset uniform variable and all points of the shape are updated with this offset. On the shader side, each dot position is calculated by rotatedPosition + u_offset, so the entire shape shifts in the specified direction on the screen.

```
document.getElementById("moveUpButton").addEventListener("click", () => {
    offset[1] += 0.1;
    redrawScene();
});

document.getElementById("moveDownButton").addEventListener("click", () => {
    offset[1] -= 0.1;
    redrawScene();
});

document.getElementById("moveLeftButton").addEventListener("click", () => {
    offset[0] -= 0.1;
    redrawScene();
});

document.getElementById("moveRightButton").addEventListener("click", () => {
    offset[0] += 0.1;
    redrawScene();
});
```

2.5 Step 5 - Rotation Buttons

Rotate buttons (rotateClockwiseButton, rotateCounterClockwiseButton) are used to rotate the shape clockwise or counterclockwise. During the click process, the rotationAngle variable is incremented or decremented and this value is passed to the vertex shader as the u_rotation uniform variable. Shader creates a rotation matrix using this angle value and the rotated position is calculated by multiplying each point by this matrix. This process allows the entire shape to be rotated around a certain angle.

```
document.getElementById("rotateClockwiseButton").addEventListener("click", () => {
    rotationAngle -= Math.PI / 18; // 10 derece
    redrawScene();
});

document.getElementById("rotateCounterClockwiseButton").addEventListener("click",
     () => {
    rotationAngle += Math.PI / 18; // 10 derece
    redrawScene();
});
```

2.6 Step 6 - Scaling

The scaling slider (scaleSlider) is used to make the shape larger or smaller. When the user moves the slider, the selected value is assigned to the scale variable and this value is passed to the vertex shader as the u_scale uniform variable. Shader multiplies each point position by a_position * u_scale, which calculates the scaled position of the shape. As the user increases the scale value, the shape becomes larger, and as the user decreases it, it becomes smaller.

```
document.getElementById("scaleSlider").addEventListener("input", (e) => {
    scale = parseFloat(e.target.value);
    redrawScene();
});
```

2.7 Step 7 - Fill Button

Fill button fills the drawn shape with the current color. When the user clicks on this button, the isFilled flag is first set to true and the filling operation is performed using the gl.TRIANGLE_FAN mode. This mode takes the first point in the points array as the center and connects the other points to create a solid polygon. The fragment shader applies the fill color using the u_color uniform variable. The fill operation works in harmony with scaling, rotating, and shifting operations on the entire shape. I chose to use fan triangulation instead of the triangulation algorithms used in previous assignments.

```
    document.getElementById("fillButton").addEventListener("click", () => {
    if (points.length < 3) {
        alert("You need at least 3 points to fill the shape!");
        return;
    }
    isFilled = true;
    redrawScene();
});
```

```
    if (isFilled) {
        gl.uniform4f(colorUniformLocation, ...currentColor);
        gl.uniform2f(offsetUniformLocation, ...offset);
        gl.uniform1f(rotationUniformLocation, rotationAngle);
        gl.uniform1f(scaleUniformLocation, scale);
        gl.drawArrays(gl.TRIANGLE_FAN, 0, points.length / 2);
    }
```

2.8 Step 8 - Clear

It sets all the variables we use for the main operations equal to zero, then clears the canvas. In this way, we can start the new shape we will draw on a clean canvas with all values reset. If the reset process had not been performed, we would not be able to use the features of the buttons as we wish in the new shape, since the previous shape's information such as rotation, angle, whether it is full or not would be transferred.

```
document.getElementById("clearButton").addEventListener("click", () => {
    points = [];
    isFilled = false;
    offset = [0.0, 0.0];
    rotationAngle = 0.0;
    scale = 1.0;

    gl.clear(gl.COLOR_BUFFER_BIT);
});
```