BBM 414 - Assignment3 WebGL2 Transformations and Basic GUI

Muzaffer Berke Savas - b2220356044

28/11/2024

Overview

This project uses WebGL2 to create an interactive application that allows users to draw, transform, and fill 2D shapes. The application supports transformations such as scaling, rotation, and translation. Additionally, it provides functionality to fill polygons using the ear-clipping triangulation algorithm.

Shaders

Vertex Shader

The vertex shader is responsible for transforming vertex positions. A transformation matrix, `uTransform`, is applied to the vertices.

Listing 1: Vertex Shader Code

```
#version 300 es
in vec2 aPosition;
uniform mat3 uTransform;
void main() {
    vec3 transformedPosition = uTransform * vec3(aPosition, 1.0);
    gl_Position = vec4(transformedPosition.xy, 0.0, 1.0);
}
```

Fragment Shader

The fragment shader assigns colors to pixels on the screen. A uniform variable, `uColor`, is used to set the color of the drawn shapes.

Listing 2: Fragment Shader Code

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

Event Listeners for Interactivity

The following event listeners are implemented to provide interactivity to the WebGL program:

- **Fill Button:**
  - Adds an event listener for the `fillButton` to trigger filling of the drawn shape using the ear clipping triangulation method.
  - Ensures the vertices are processed in clockwise or counter-clockwise order.
- **Rotation Buttons:**
  - `rotateClockwiseButton` and `rotateCounterclockwiseButton`:
    - Adjust the rotation of the shape.
    - Re-renders the scene and updates filled triangles if needed.
- **Translation Buttons:**
  - `upButton`, `downButton`, `leftButton`, and `rightButton`:
    - Modify the translation vector to move the shape in the respective direction.
    - Re-renders the scene and updates filled triangles if needed.
- **Draw Button:**
  - Toggles the drawing mode, allowing users to add vertices by clicking on the canvas.
  - Adjusts the cursor style to indicate drawing mode.
- **Canvas Click Event:**
  - Captures the click coordinates, normalizes them, and adds the points to the vertex buffer.
  - Updates the scene to reflect the newly added vertices.
- **Scale Slider:**
  - Adjusts the scaling factor based on user input.
  - Dynamically updates the display value and slider style.
  - Re-renders the scene and updates filled triangles if necessary.
- **Color Picker:**
  - Changes the shape's color based on the selected color.
  - Re-renders the scene and updates filled triangles with the new color.
- **Clear Button:**
  - Clears all vertices and resets transformations.

  - Resets scaling, translation, and rotation to default values.
  - Updates the slider and re-renders the scene.

- **Document Ready Event:**
  - Ensures that the initial state of the scale slider and its value display is set up when the document is loaded.

Matrix Transformations

The transformation matrix is a 3×3 matrix used for 2D affine transformations. It combines scaling, rotation, and translation into a single operation:

$$\mathbf{T} = \begin{bmatrix} s_x \cos(\theta) & -s_y \sin(\theta) & t_x \\ s_x \sin(\theta) & s_y \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where:

- s_x, s_y: Scaling factors.
- θ: Rotation angle.
- t_x, t_y: Translation offsets.

Drawing and Triangulation

Drawing Shapes

Shapes are drawn by capturing mouse clicks on the canvas. Each click adds a vertex to the polygon. The vertices are stored in a buffer, which is then passed to the GPU for rendering.

Listing 3: Mouse Click Event for Drawing Shapes

```
canvas.addEventListener('click', (event) => {
    const rect = canvas.getBoundingClientRect();
    const x = (event.clientX - rect.left) / canvas.width * 2 - 1;
    const y = (event.clientY - rect.top) / canvas.height * -2 + 1;
    vertices.push(x, y);
    draw();
});
```

Filling Shapes with Ear-Clipping

The ear-clipping algorithm identifies and removes *ears* (triangles within the polygon) iteratively until only one triangle remains. The following function identifies an ear and clips it:

Listing 4: Ear-Clipping Algorithm

```
function clipEar(vertices) {
    const indices = [];
    while (vertices.length > 6) { // At least 3 vertices (6 coordinates
       )
        for (let i = 0; i < vertices.length / 2; i++) {
            if (isEar(vertices, i)) {
                indices.push(i, (i + 1) % vertices.length, (i + 2) %
                   vertices.length);
                vertices.splice((i + 1) * 2, 2);
                break;
            }
        }
    }
    indices.push(0, 1, 2); // Last triangle
    return indices;
}
```

References

1. Angel, Edward, and Dave Shreiner. *Interactive Computer Graphics: A Top-Down Approach with WebGL.* Pearson, 2016.
2. WebGL2 Documentation. Available at: https://www.khronos.org/webgl/
3. Moeller, Tomas, and Eric Haines. *Real-Time Rendering.* A K Peters/CRC Press, 2018.
4. Triangulation Algorithms: https://en.wikipedia.org/wiki/Polygon$_t$*riangulation*