BBM414 Computer Graphics Lab. Assignment 3

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

Overview

This project involves developing an interactive graphics application using WebGL and shaders. In Part 1, users can manipulate a square with buttons to change its direction, speed, and color. Part 2 expands on this by adding a GUI for drawing shapes with mouse clicks, applying transformations (translation, scaling, rotation), and dynamically changing colors. The application emphasizes user interaction and animation through a responsive interface.

1 Part 1 - Transformations on a Square

The following section analyzes the WebGL2 code for rendering and transforming a square. The program includes interactive functionality to manipulate the square's rotation, speed, and color through user inputs.

1.1 Setup and Initialization

The code begins by initializing the WebGL2 context on an HTML canvas and defining variables required for rendering and transformation. Key variables include `theta` for rotation angle and `addValue` for controlling the rotation speed and direction.

```
var canvas;
var gl;

var theta = 0.0;
var addValue = 0.1;
var thetaLoc;
```

Additionally, an array of random RGBA colors is generated for the square's vertices. Each vertex is assigned a unique color to create a gradient effect.

```
var 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)
];
```

1.2 Vertices and Buffers

The square is defined using four vertices arranged in a TRIANGLE_STRIP format. These vertices, along with the color data, are uploaded to the GPU using buffer objects.

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

1.3 Shader Integration

The program uses shaders to handle vertex positions and colors. The vertex shader receives the vertex positions and color attributes, interpolating colors across the square.

```
var program = initShaders(gl, "vertex-shader", "fragment-shader");
gl.useProgram(program);

var vPosition = gl.getAttribLocation(program, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);

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

1.4 Shader Program Analysis

The WebGL2 program uses two shaders: a vertex shader and a fragment shader. These shaders work together to process vertex data and determine the final color of each fragment on the rendered square.

1.4.1 Vertex Shader

The vertex shader is responsible for transforming the positions of the square's vertices and passing color data to the fragment shader. The shader receives the vertex position (`vPosition`) and vertex color (`vColor`) as inputs, and the rotation angle (`theta`) is provided as a uniform variable.

```
#version 300 es
in vec4 vPosition;
uniform float theta;
in vec4 vColor;
out vec4 outColor;

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;
    outColor = vColor;
}
```

Key Functionality:

- **Rotation Transformation:** The vertex position is rotated using the rotation matrix. The sine (`s`) and cosine (`c`) of the angle `theta` are computed, and the new position is calculated as:

$$\text{gl\_Position.x} = -s \cdot vPosition.y + c \cdot vPosition.x$$
$$\text{gl\_Position.y} = s \cdot vPosition.x + c \cdot vPosition.y$$

This performs a 2D rotation around the origin.

- **Output Position:** The rotated position is assigned to `gl_Position`, with `z` and `w` components set to 0.0 and 1.0 respectively for proper rendering in normalized device coordinates.
- **Color Passing:** The input color (`vColor`) is passed to the fragment shader through the `outColor` variable.

1.4.2 Fragment Shader

The fragment shader determines the color of each fragment (pixel) based on the interpolated color values from the vertex shader.

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

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

Key Functionality:

- **Precision Declaration:** The `mediump` precision is used for floating-point calculations, balancing performance and accuracy.
- **Color Output:** The interpolated color (`outColor`) is assigned to the output variable `outputColor`, which is used to display the final color of the fragment.

Overall Workflow: The vertex shader rotates the square's vertices and interpolates the vertex colors, while the fragment shader ensures smooth color blending across the surface of the square. Together, these shaders enable the real-time transformations and dynamic visual effects seen in the application.

1.5 Interactive Controls

The program includes buttons to manipulate the square's behavior:

- **Toggle Rotation:** Flips the rotation direction by negating `addValue`.

  ```
  function toggle() {
      addValue *= -1;
  }
  ```

- **Speed Up:** Increases `addValue`, accelerating the rotation.

  ```
  function speedUp() {
      addValue += (addValue > 0 ? 0.01 : -0.01);
  }
  ```

- **Slow Down:** Decreases `addValue` to a minimum threshold, slowing the rotation.

  ```
  function slowDown() {
      if (addValue > 0.02) {
          addValue -= 0.01;
      } else if (addValue < -0.02) {
          addValue += 0.01;
      } else {
          addValue = (addValue > 0 ? 0.02 : -0.02);
      }
  }
  ```

- **Change Color:** Generates new random colors for each vertex and updates the GPU buffer.

  ```
  function 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)
      ];

      gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
      gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.
          STATIC_DRAW);
  }
  ```

1.6 Animation Loop

The animation is driven by a `requestAnimationFrame` loop that clears the canvas, updates the rotation angle, and redraws the square on each frame.

```
function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    theta += addValue;
    gl.uniform1f(thetaLoc, theta);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
    requestAnimationFrame(render);
}
```

This loop ensures smooth and continuous rotation, with user interactions dynamically affecting the animation in real time.

2 Part 2 - Interactive Drawing Application

The following section analyzes the implementation of an interactive drawing application using WebGL2. This program allows users to draw polygons, manipulate their transformations, and interact with various GUI elements.

2.1 Initialization and Setup

The initialization step involves setting up global variables, event listeners, and default parameters required for the application's functionality.

```
let currentColor = "#000000";
let scale = 1;
let closed = false;
let fill = false;
let translation = { x: 0, y: 0 };
let theta = 0;
let hasFilled = false;
let canTransform = false;
const canvas = document.getElementById("glCanvas");
const gl = canvas.getContext("webgl2");
```

Purpose: These global variables define the initial state and configuration of the drawing application.
Details:

- `currentColor`: Stores the currently selected color for rendering the polygons.

- `scale`: Represents the scaling factor for transformations.
- `closed` and `fill`: Boolean flags to indicate whether the polygon should be closed and/or filled.
- `translation` and `theta`: Specify translation (`x`, `y`) and rotation (`theta`) transformations.
- `hasFilled` and `canTransform`: Flags indicating whether the polygon has been filled and whether transformations are allowed.
- `canvas` and `gl`: References to the HTML canvas element and the WebGL2 rendering context.

Dynamic Initialization on Document Load The application also sets up key event listeners during the `DOMContentLoaded` event to prepare for user interactions.

```
document.addEventListener("DOMContentLoaded", () => {
    const colorPicker = document.getElementById("colorPicker");
    colorPicker.addEventListener("input", () => {
        currentColor = colorPicker.value;
    });

    let vertices = [];
    let isDrawing = false;
    let filledTriangles = [];
```

Purpose: Initializes dynamic user-driven elements like the color picker and vertex storage.
Details:

- `colorPicker`: Binds an input listener to the color picker to dynamically update the selected color (`currentColor`).
- `vertices`: An array for storing user-defined vertices during drawing.
- `isDrawing`: A flag to indicate whether the user is actively drawing a polygon.
- `filledTriangles`: Stores triangulated vertex data when the polygon is filled.

Conclusion: This setup phase ensures that the application initializes with default settings and prepares essential elements for user interaction, enabling a seamless drawing experience.

2.2 The `drawPolygon` Function

The `drawPolygon` function is the core rendering mechanism in the application. It handles the rendering of polygons, applying transformations, and supporting both closed polygons and filled polygons through triangulation. Below is the explanation of its components.

2.2.1 Canvas Preparation and Early Exit Conditions

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

if (vertices.length < 2) {
    return;
}
```

Purpose: Prepares the canvas for rendering and ensures sufficient vertices are available for drawing.
Details: The canvas is cleared with the default color to prevent overlapping drawings. If fewer than two vertices exist, the function exits early, as a polygon cannot be formed.

2.2.2 Handling Closed Polygons

```
if (closed && vertices.length > 2 && isDrawing) {
    vertices.push(vertices[0]);
    isDrawing = false;
}
```

Purpose: Closes the polygon by connecting the last vertex to the first when required.
Details: When a polygon is marked as `closed`, (happens when user stops drawing by clicking any of the transformation buttons), the first vertex is appended to the vertex list. This ensures the polygon forms a loop and it becomes a closed polygon.

2.2.3 Vertex Data Preparation

```
const vertexArray = [];
for (let i = 0; i < vertices.length; i++) {
    vertexArray.push(vertices[i].x, vertices[i].y);
}
```

Purpose: Converts the array of vertex objects into a flat array format suitable for WebGL.
Details: Each vertex's `x` and `y` coordinates are appended sequentially to a flat array. This format is required for WebGL's buffer operations.

2.2.4 Shader Program Initialization and Buffer Binding

```
const program = initProgram(gl, vsSource, fsSource);
gl.useProgram(program);

const buffer = initBuffer(gl, vertexArray);
const positionLocation = gl.getAttribLocation(program, "aPosition");
gl.enableVertexAttribArray(positionLocation);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);
```

Purpose: Prepares the GPU for rendering by setting up shaders and uploading vertex data.
Details:

- `initProgram`: Initializes and links the vertex and fragment shaders into a WebGL program.
- `initBuffer`: Creates and uploads the vertex array into a GPU buffer.
- `getAttribLocation`: Retrieves the location of the `aPosition` attribute in the vertex shader.
- `enableVertexAttribArray` and `vertexAttribPointer`: Enable and bind the vertex buffer data to the `aPosition` attribute, ensuring it is accessible in the vertex shader.

2.2.5 Applying Transformations

```
const translationLocation = gl.getUniformLocation(program, "
   uTranslation");
const thetaLocation = gl.getUniformLocation(program, "theta");
const scaleLocation = gl.getUniformLocation(program, "uScale");

gl.uniform2f(translationLocation, translation.x, translation.y);
gl.uniform1f(thetaLocation, theta);
gl.uniform1f(scaleLocation, scale);
```

Purpose: Applies user-defined transformations such as translation, rotation, and scaling.
Details:

- `getUniformLocation`: Retrieves the locations of the transformation-related uniforms (`uTranslation`, `theta`, `uScale`).
- `uniform2f` and `uniform1f`: Set the values for translation (`x, y`), rotation (`theta`), and scaling (`scale`) for use in the vertex shader.

2.2.6 Setting Polygon Color

```
const colorLocation = gl.getUniformLocation(program, "uColor");
const color = hexToRgb(currentColor);
gl.uniform4fv(colorLocation, color);
```

Purpose: Sets the color for the polygon rendering.
Details:

- `hexToRgb`: Converts the selected hex color code to an RGBA format.
- `uniform4fv`: Passes the RGBA color to the fragment shader's `uColor` uniform.

2.2.7 Rendering Polygons

- **For Filled Polygons:**

  ```
  if (filledTriangles.length > 0) {
      const filledBuffer = initBuffer(gl, filledTriangles);
      gl.bindBuffer(gl.ARRAY_BUFFER, filledBuffer);
      gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT,
          false, 0, 0);
      gl.drawArrays(gl.TRIANGLES, 0, filledTriangles.length /
           2);
  }
  ```

  Details: Filled polygons are rendered using triangulated vertex data stored in `filledTriangles`. These triangles are uploaded to a buffer and rendered as `gl.TRIANGLES`.

- **For Line Segments:**

  ```
  for (let i = 0; i < vertices.length - 1; i++) {
      gl.drawArrays(gl.LINES, i, 2);
  }
  ```

  Details: Open polygons are rendered by iterating through consecutive vertex pairs and drawing lines (`gl.LINES`) between them.

Conclusion: The `drawPolygon` function integrates user input, transformations, and WebGL rendering capabilities to dynamically render and transform polygons in real-time.

2.3 Color Conversion: `hexToRgb` Function

The `hexToRgb` function converts a hexadecimal color value into its equivalent RGBA format, suitable for use in WebGL shaders.

```
function hexToRgb(hex) {
    return [
        parseInt(hex.slice(1, 3), 16) / 255,
        parseInt(hex.slice(3, 5), 16) / 255,
        parseInt(hex.slice(5, 7), 16) / 255,
        1.0
    ];
}
```

Purpose WebGL shaders require colors in normalized RGBA format, where each component is a floating-point value between 0.0 and 1.0. This function provides a way to convert user-selected hexadecimal colors (common in HTML color pickers) into this normalized format.

Explanation

- The input `hex` is a string representing a hexadecimal color code (e.g., `"#FF5733"`).

- The `slice` method extracts the red (`hex.slice(1, 3)`), green (`hex.slice(3, 5)`), and blue (`hex.slice(5, 7)`) components from the hexadecimal string.
- `parseInt(..., 16)` converts each extracted substring into an integer based on base-16 (hexadecimal) representation.
- Each integer is divided by 255 to normalize the value into the range $[0.0, 1.0]$.
- The returned array includes the normalized red, green, and blue values, along with a fixed alpha value of `1.0` to ensure full opacity.

Use Case This function is invoked whenever a user selects a new color via the color picker. The resulting RGBA values are passed to the WebGL shaders, ensuring that the selected color is correctly rendered on the canvas.

Conclusion By handling the conversion internally, the `hexToRgb` function simplifies color management and ensures compatibility with WebGL's rendering pipeline.

2.4 Event Listeners for User Interaction

The application provides interactive controls to draw, manipulate, and transform shapes on the canvas. These functionalities are implemented through event listeners that respond to user actions such as mouse clicks, button presses, and input changes.

Drawing Controls

- `Draw Button`: Initializes the drawing process by resetting the vertices array, enabling transformations, and clearing the canvas. The cursor changes to a crosshair when it enters the canvas element to indicate drawing mode.

```
document.getElementById("Draw").addEventListener("click",
    () => {
    if (vertices.length > 0) { return; }
    canTransform = true;
    isDrawing = true;
    canvas.style.cursor = "crosshair";
    vertices = [];
    gl.clear(gl.COLOR_BUFFER_BIT);
});
```

- `Canvas Click`: Adds a new vertex at the clicked position, normalized to WebGL's coordinate system. Vertices are stored in an array and used to render the polygon dynamically.

```
canvas.addEventListener("click", (event) => {
    if (!isDrawing) { return; }
    const rect = canvas.getBoundingClientRect();
    const x = (event.clientX - rect.left) / canvas.width *
        2 - 1;
    const y = (rect.bottom - event.clientY) / canvas.height
         * 2 - 1;
    vertices.push({x, y});
    drawPolygon(gl, vertices, closed);
});
```

- `Clear Button`: Resets all variables, clears the canvas, and exits drawing mode.

Transformation Controls The transformation buttons allow the user to manipulate the drawn polygon. These controls include translation, rotation, scaling, and filling.

- `Translation`: Buttons for moving the shape (`Up`, `Down`, `Left`, `Right`) adjust the `translation` values and redraw the polygon.

```
document.getElementById("Right").addEventListener("click",
    function() {
     if (!canTransform) { return; }
     closed = true;
     translation.x += 0.1;
     drawPolygon(gl, vertices);
});
```

- `Rotation`: Buttons for clockwise and counterclockwise rotation update the `theta` value in radians and redraw the polygon.
- `Scaling`: A slider input adjusts the `scale` factor, allowing dynamic resizing of the shape.
- `Filling`: The `Fill` button applies the ear clipping algorithm to triangulate the polygon and fills it with the current color. Filling is only enabled for closed polygons with three or more vertices.

Summary These event listeners provide an intuitive interface for users to draw and interact with shapes on the canvas. Each listener modifies the state variables or directly invokes the `drawPolygon` function to update the rendering in real-time.

3 Additional Notes

In the program, `canTranform` variable is used to prevent beforehand modifications to the shape before it is drawn. After each transformation button is clicked, `closed` variable is set to true and with this change, first vertex point is pushed to the `vertexArray`. By doing this, a line between the last point and the first point is drawn and polygon became closed. When fill button is clicked, `filledTriangles` array is populated using `earClippingTriangulation` function to be drawn later. By setting `hasFilled` to true, any side effect by clicking fill button again is prevented.

4 Summary

In summary, user can start drawing by clicking `draw` button. When user clicks to any transformation button, program will close the polygon by the methods mentioned above. After that, user could apply any transformation to that polygon by clicking its respective button. When s/he selects a new color, last selected color will be applied to the whole polygon. Also, ear clipping method generally expects vertices to be in the counterclockwise rotation. If they are not, it causes problems. To prevent this, a checking process is added to `helper.js` file that checks the condition and if they are not in counterclockwise direction, it reverses the given vertices array.

For program to work the way it was intented, user should follow the rules. Sometimes, when the given shape is a little bit complex, ear clipping algorithm behaves strangely that causes infinite loops. When fill is clicked and given shape is not filled, my suggestion is closing the page and trying with another shape.

5 Conclusion

This project demonstrates the capabilities of WebGL2 through two parts: foundational transformations and an interactive drawing application.

In **Part 1**, we explored fundamental WebGL2 concepts, implementing transformations like rotation, scaling, and translation using shaders. This provided a clear understanding of the WebGL2 rendering pipeline.

In **Part 2**, these concepts were applied to create a polygon drawing tool, enabling user-defined transformations, color selection, and dynamic filling using ear clipping triangulation. This showcased the integration of WebGL2 with JavaScript for interactive applications.

Together, these parts highlight WebGL2's flexibility, from basic rendering to interactive graphics. Future enhancements could include texture mapping, animations, and 3D rendering, expanding its practical applications.

References

1. `https://github.com/esangel/WebGL`
2. `https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect`
3. `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener`