BBM414 Introduction to Computer Graphics Lab. Practical # 3- Transformations and Basic GUI

Barış Yıldız
2210356107
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
baris.byildiz@gmail.com

Overview

In this experiment, transformations (translations, rotations and scaling) with matrices and basic GUI with HTML elements are implemented. The experiment consists of two parts.

In the first part, a source code that implements a white square rotating in the counterclockwise direction at a high speed is provided. The task is to implement four buttons the form a GUI: a **Toggle** button that reverses the rotation direction, a **Speed Up** button that increases the rotation speed, a **Slow Down** button that decreases the rotation speed and a **Color** button that gives all four corner vertices of the square different colors.

In the second part, the task is to implement a *basic drawing web application*. The application consists of a drawing canvas and a control panel. The control panel consists of a color picker, several buttons and a slider.

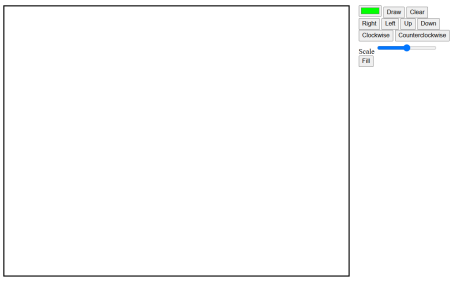

Figure 1: Drawing Canvas (left side) and Control Panel (right side)

- **The color picker** sets the color of the drawn shape.
- The **Draw** button sets the mode to "Draw". In this mode, the user can click on the drawing canvas to insert a vertex. This inserted vertex will be joined to the vertex added before it with a line.
- The **Clear** button will clear the drawing canvas and reset any other settings such as applied translations, scales and rotations.
- The **Right**, **Left**, **Up** and **Down** buttons will translate (displace) the shape in the corresponding directions.
- The **Clockwise** and **Counterclockwise** buttons will rotate the shape in the corresponding directions.

- The **Scale** slider will scale the shape. This will turn back to the default setting upon pressing the "Clear" button.
- The **Fill** button will set the mode to "Fill". In this mode, the shape is filled in with the selected color. This mode also doesn't allow the user to change back to the "Draw" mode because the shape is already drawn.

1 Part 1 - Rotating Square

Firstly, the buttons are added as HTML elements and they are assigned onclick functions from Javascript.

- The onclick of **Toggle** button is a function that multiplies the value of a variable called *thetaChange* by -1. This essentially inverts the direction of rotation change, making the square rotate in the opposite direction.
- The onclick of **Speed Up** button is a function that adds a predefined factor (0.01 in this case) to the absolute value of *thetaChange*. In other words, the value of *thetaChange* will be decreased by the factor if it is negative, increased otherwise.
- The onclick of **Slow Down** button is a function that has the inverse logic of the onclick of **Speed Up**. As a difference, the *thetaChange* value is clamped at the end so that it is not 0 or not of the opposite sign.
- The onclick of **Color** button is a function that assigns four Vector4 objects that represent colors of the four vertices, in a *colors* array. The assigned Vector4 objects have random float values between 0 and 1 as their r, g and b (x, y, z) components.

Secondly, in the render function, the given *theta* variable is increased by the newly added *thetaChange* variable. This will make the buttons have an effect on the speed of the square's rotation.

Lastly, in order to change the color of the square, the newly added *colors* Vector4 array is made a WebGL buffer. The vertex shader is provided a *vColor* input attribute that is then passed to the fragment shader and output from it as the color of the square. At the end of this process, the color buffer is linked to the *vColor* attribute of the vertex shader via *gl.vertexAttribPointer()*.

2 Part 2 - Web Drawing Application

The application consists of two parts: **a drawing canvas** and a **control panel**. These two parts are contained in a div element with the id *container*. This div has the **display** style property set to *flex* which allows the two parts to be placed horizontally. Below subsections explain the implementations of the two parts.

2.1 The Drawing Canvas

The drawing canvas on the left is a div element with the id *drawDiv* that contains a single canvas element inside with the id *glCanvas*. This canvas is used as the WebGL canvas of the application. The canvas also has the **cursor** style property as *crosshair* for a more precise cursor. The **border-style** style property of the *drawDiv* element is *solid* which separates the canvas from the background.

2.2 The Control Panel

The control panel is a div element with the id *buttonDiv* which has the **padding-left** style property set to *20px* to apply a left padding to the items for them to be separated from the drawing canvas. The div element contains five other div elements which form the five different rows of control panel elements.

- The first of these div elements contain **the color picker**, the **Draw** button and the **Clear** button.
  - The color picker is an HTML input element of type color. The **onchange** attribute of this element is the function *setColor(value)* which takes the selected color in hexadecimal format, converts it to decimal format and assigns the red, green, blue components to the *color* variable in the program. This variable determines the color of the filled shape or the drawn lines.
  - The Draw button has the **onclick** function *setMode(modeToSet)* which sets the *mode* variable in the program. This variable represents the mode of the application, being **MODE.DRAW**, **MODE.CLEAR** or **MODE.FILL**. The button calls the method as setMode(MODE.DRAW). Similarly, the Clear button's **onclick** function is setMode(MODE.CLEAR).
- The second of these div elements contains buttons that displace the shape in a direction. For example, the **Up** button moves the shape in the +y direction and the **Right** button translates in the +x direction. These buttons use the function *displaceObject(displacement)* as onclick functions. This function does *positionChange* += *displacement*, *positionChange* being a variable of type Vector2 in the program. This variable is the value to translate the shape by in the translation/displacement matrix. The parameter *displacement* can have the any of the values **DISPLACEMENT.LEFT**, **DISPLACEMENT.RIGHT**, **DISPLACEMENT.DOWN** or **DISPLACEMENT.UP**.
- The third of the div elements contains buttons that rotate the shape in a direction. The buttons use the function *setAngleChange(direction)* which increments or decrements the *angleChange* variable in the program depending on *direction*. The variable is used as the angle to rotate by in the rotation matrix to rotate the object.
- The fourth and fifth div elements contain the Scale slider and the Fill button respectively. The scale slider uses the function *setScale(value)* as its **onchange** callback. The function assigns *value* to the x and y attributes of the *scale* Vector2 variable in the program. This variable is used in the scale matrix that scales the shape. The Fill button simply sets the *mode* variable to **MODE.FILL** by having setMode(MODE.FILL) as its **onclick** callback.

2.3 Transformations and Shader Code

The transformations are done in the vertex shader of the program while the fragment shader controls the color change. The below matrices are used and implemented in the shader:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2: Displacement Matrix

The displacement matrix is both used to translate the shape to a direction, and to translate the object to the origin point and back for a proper rotation to take place (a rotation around the object's center instead of the origin).

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3: Rotation Matrix

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4: Scaling Matrix

```
//Vertex Shader
precision mediump float;
in vec4 a_position;

uniform float u_angle;
uniform vec2 u_dv;
uniform vec2 u_scale;
uniform vec2 u_center;
uniform vec4 u_color;

out vec4 v_color;

void main() {
    float cos_angle = cos(u_angle);
    float sin_angle = sin(u_angle);

    mat4 inverseTranslationMatrix = mat4(
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        -u_center.x, -u_center.y, 0.0, 1.0
    );

    mat4 translationMatrix = mat4(
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        u_center.x + u_dv.x, u_center.y + u_dv.y, 0.0, 1.0
    );

    mat4 scaleMatrix = mat4(
        u_scale.x, 0.0, 0.0, 0.0,
        0.0, u_scale.y, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    );

    mat4 rotationMatrix = mat4(
        cos_angle, sin_angle, 0.0, 0.0,
        -sin_angle, cos_angle, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    );

    gl_Position = translationMatrix * rotationMatrix
    * scaleMatrix * inverseTranslationMatrix * a_position;
    v_color = u_color;
}

//Fragment Shader
precision mediump float;
in vec4 v_color;
out vec4 color;
void main() {
    color = v_color;
}
```

The value *u_center* is the center point of the drawn shape calculated from the minimum and maximum extreme values of the shape's x and y values. The value *u_dv* corresponds to the value of the *positionChange* variable in the program. Likewise, the value *u_scale* is *scale* in the program, *u_angle* is *angleChange* and *u_color* is *color*.

$$u\_center = Vector2((x\_min + x\_max)/2, (y\_min + y\_max)/2)$$

The gl_Position of the object is first multiplied by *inverseTranslationMatrix*, which translates the object to the origin. Secondly, the object is applied scaling, rotation, and translation using corresponding matrices and uniform values *u_scale*, *u_dv* and *u_angle*. Lastly, the object's center is translated back by *u_center*.

2.4 Adding Vertices and Filling In The Shape

The function *addVertex()* is added to the drawing canvas as a **mousedown** event callback function. This function is responsible for creating the vertices of the shape to be drawn at the mouse click position. The X and Y position of the mouse click is transformed to the canvas coordinates using the following formula:

$$CanvasX = (mouseX - canvasWidth/2)/(canvasWidth/2)$$
$$CanvasY = (mouseY - canvasHeight/2)/(canvasHeight/2)$$

The polygon triangulation algorithm accepts only vertices given in clockwise direction, yet in this drawing app it can be given in counterclockwise direction. In order to also accept these type of shapes, the triangulation algorithm is tried on the reversed list of vertices if it fails on the normal list.

2.5 Render Function

In the *render()* function, the added vertices are drawn either as lines, as triangles or are erased depending on the mode.

- If the *mode* variable is **MODE.DRAW**, the drawing mode is *gl.LINE_STRIP*.
- If it is **MODE.FILL**, the drawing mode is *gl._TRIANGLES*, and the data to be drawn is triangles created from given vertices. If the given vertices are unable to be converted to triangles (invalid shape for the triangulation algorithm), *mode* is switched back to **MODE.DRAW**. Else, it is converted from lines to a filled shape.
- If it is **MODE.CLEAR**, the vertex data is erased and all applied settings are reset (position change, angle change, scale).

References

[1] *BBM412 Slides, Specifically the slide regarding Transformations*

[2] `https://developer.mozilla.org/en-US/docs/Web/API/Element/mousedown_event`

[3] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input`