INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY, BANGALORE

Assignment 1

Sushranth P Hebbar

January 30, 2021

1 PROBLEM STATEMENT

1.1 2D interactive planar rendering with 2D translation, rotation, scaling/zooming.

Objectives:- WebGL programming, Creation and rendering of simple 2D geometric shapes as primitives, Transformation of 2D objects and scene – instance-wise implementation, Key(board) events for changing control modes, and transformation implementation Mouse events for picking points and objects.

2 APPROACH

2.1 Introduction

Each primitive has it's own class. The attributes of this class are the initial coordinates of the centroid, color, unit(which is used to determine the initial spawn size of the primitive), buffer and the vertices of the triangle it is composed of. Additionally, it contains an instance of the Transform class. This contains the translation vector, scaling vector, rotation angle, axis and the transformation matrix. The renderer class deals with initialization and resizing of the canvas. The vertex and fragment code are available in vertex.js and fragment.js. Lastly, the index.js file acts as the glue for running the previous classes and files.

2.2 Drawing Mode

When the state of the application is in this mode, an instance of a primitive is created based on the value of the key press. The coordinates of the mouse click are obtained and then converted to clip space. It's then passed as arguments to create a primitive object which is then added to a global list of primitives. This list it then iterated and each primitive is rendered on the screen.

2.3 Instance Transformation Mode

The euclidean distance is computed between the mouse coordinates of the clicked point and the centroid of each primitive. Then the primitive with the minimum euclidean distance is picked. As mentioned in the introduction section, each primitive is composed of an instance

```
let mouse_coord = renderer.mouseToClipCoord(event.clientX,event.clientY);
if(control_key==0)
{
    if(shapeKey=="KeyS")
    {
        const square = new Square(gl,mouse_coord[0],mouse_coord[1], vec3.fromValues(1.0, 0.0, 1.0));
        primitives.push(square);
    }
    else if(shapeKey=="KeyR")
    {
        //console.log("HERE in rectangle");
        const rectangle = new Rectangle(gl,mouse_coord[0],mouse_coord[1], vec3.fromValues(1.0, 0.0, 0.0));
        primitives.push(rectangle);
    }
    else if(shapeKey=="KeyC")
    {
        const circle = new Circle(gl, mouse_coord[0], mouse_coord[1], vec3.fromValues(0.0, 0.0, 1.0));
        primitives.push(circle);
    }
}
```

Figure 2.1: Primitive Instance.

of the Transform class. When translation or scaling is applied the components of these vectors are appropriately incremented/decremented. As displayed in figure 3.1, the primitive is first translated from it's spawn point to the origin of the clip space. Then the scaling transformation is applied. This way, the primitive is scaled while preserving the centroid. Then it is translated back to it's original spawn point. Lastly, the translation vector is added to the centroid after which it is rendered. For more details, check out the updateMVPMatrix function which can be found in transform.js.

2.4 Scene Transformation Mode

In this mode, the centroid of the tightly fitting bounding box is computed by finding the minimum and maximum of the x and y coordinates of the endpoints of each primitive. Then the next step is to find the midpoint which is nothing but the centroid. For each primitive, the current centroid is the sum of the spawned centroid and the translation vector. The endpoints of the primitive are then computed by multiplying the size of the primitive(primitive.unit as shown in figure 2.2) with the scaling vector. For more details check out Circle, Rectangle and Square classes. Finally, the xmax,xmin... variables are updated and used to compute the centroid of the bounding box.
The updateAroundPoint function manages the transformation matrix for rotation about the centroid of the bounding box for each primitive as displayed in figure 3.1. First the spawn point is translated to the origin. Then it is scaled. Now consider the difference vector between the current position of the centroid of the primitive and the centroid of the bounding box. Add this difference vector to the centroid at the origin. Now this position of the centroid of the primitive about the origin is equivalent to the position of the current centroid of the primitive about the centroid of the bounding box. Now apply the rotation transformation and shift this about the centroid of the bounding box.

3 Questions

3.1 How did your program separate transformation matrices for all the object instances(primitives), and the scene?

Their exists a class called Transform which contains the transformation matrix as an attribute. Their are three primitive classes - Square, Rectangle and Circle. Each of these classes is composed of an instance of the Transform class as an attribute. In the below image, the updateMVP-Matrix function deals with translation and scaling as required by the instance transformation mode. The updateAroundPoint function deals with rotation of all the primitives around the centroid of the bounded rectangle. For each primitive, based on the current mode value, the

```
var xmin=1e9, xmax=-1e9, ymin=1e9, ymax=-1e9;
primitives.forEach((primitive,index) => {

    var x = primitive.transform.translate[0]+primitive.centerX;
    var y = primitive.transform.translate[1]+primitive.centerY;
    var width, height;
    width = ((primitive.unit)*primitive.transform.scale[0]);
    height = width;
    if(primitive instanceof Rectangle)
    {
        //console.log("Rectangle");
        height += width;
    }
    xmax = Math.max(xmax, x+width);
    xmin = Math.min(xmin, x-width);
    ymax = Math.max(ymax, y+height);
    ymin = Math.min(ymax, y-height);
});
var rect_x = (xmax+xmin)/2.0;
var rect_y = (ymax+ymin)/2.0;
```

Figure 2.2: Centroid Calculation.

appropriate function is invoked and this sets the values of the transformation matrix. Having two different functions is convenient when transitioning from scene transformation to drawing mode as invoking the updateMVPMatrix function restores the transformation done in the instance transformation mode.

```
updateMVPMatrix()
{
    //console.log((-1)*this.origin);
    mat4.identity(this.modelTransformMatrix);
    mat4.translate(this.modelTransformMatrix, this.modelTransformMatrix, this.translate);
    //mat4.rotate(this.modelTransformMatrix, this.modelTransformMatrix, this.rotationAngle, this.rotationAxis);
    mat4.translate(this.modelTransformMatrix, this.modelTransformMatrix, this.invOrigin);
    mat4.scale(this.modelTransformMatrix, this.modelTransformMatrix, this.scale);
    mat4.translate(this.modelTransformMatrix, this.modelTransformMatrix, this.origin);
    //console.log(this.translate);
}

updateAroundPoint(xPoint, yPoint, xdPoint, ydPoint)
{
    var diff = vec3.fromValues(xdPoint,ydPoint,0.0);
    //console.log(diff);
    var centroid = vec3.fromValues(xPoint, yPoint, 0.0);
    mat4.identity(this.modelTransformMatrix);
    mat4.translate(this.modelTransformMatrix, this.modelTransformMatrix, centroid);
    mat4.rotate(this.modelTransformMatrix, this.modelTransformMatrix, this.rotationAngle, this.rotationAxis);
    mat4.translate(this.modelTransformMatrix, this.modelTransformMatrix, diff);
    mat4.scale(this.modelTransformMatrix, this.modelTransformMatrix, this.scale);
    mat4.translate(this.modelTransformMatrix, this.modelTransformMatrix, this.origin);
}
```

Figure 3.1: Transformation Matrix.

3.2 What API is critical in the implementation of "picking" using mouse button click?

The mouse click is detected using javascripts event listener. But these coordinates have to be converted to clip space. In figure 3.2, the coordinates are first normalized using the canvas width and height. Then they are multiplied by two and then subtracted by one. Now the range of the mouse coordinates is from -1 to 1. Finally, the y coordinate is reversed.
As given in figure 3.3, iterate over all the primitives and find the euclidean distance between the centroid of each primitive and the coordinates of the mouse click after it has been converted to clip space coordinates. Pick the primitive with the minimum euclidean distance.

3.3 What would be a good alternative to minimize the number of key click events used in this application? Your solution should include how the mode-value changes are incorporated.

Buttons can be an alternative to key clicks. There can be separate buttons for each mode. For example, if you click on the instance transformation button, it further displays +/- buttons for

```
mouseToClipCoord(mouseX,mouseY) {

    // convert the position from pixels to 0.0 to 1.0
    mouseX = mouseX / this.canvas.width;
    mouseY = mouseY / this.canvas.height;

    // convert from 0->1 to 0->2
    mouseX = mouseX * 2;
    mouseY = mouseY * 2;

    // convert from 0->1 to 0->2
    mouseX = mouseX - 1;
    mouseY = mouseY - 1;

    // flip the axis
    mouseY = -mouseY; // Coordinates in clip space

    return [mouseX, mouseY]
}
```

Figure 3.2: Converting to clip space.

```
else if(control_key==1)
{
    var distanceToCentroid = 1e9;
    primitives.forEach((primitive,index) =>
    {
        var x = primitive.transform.translate[0]+primitive.centerX;
        var y = primitive.transform.translate[1]+primitive.centerY;
        var x1 = mouse_coord[0];
        var y1 = mouse_coord[1];

        var temp = ((x-x1)*(x-x1) + (y-y1)*(y-y1));
        if(temp<distanceToCentroid)
        {
            distanceToCentroid = temp;
            selectedPrimitive = index;
        }
    });
    //console.log(selectedPrimitive);
    //var primitive = primitives[selectedPrimitive];
    //console.log(primitive.centerX);
    //console.log(primitive.centerY);
```

Figure 3.3: Picking primitive.

translation and scaling. Sliders are another option for +/- buttons. Similarly for scene transformation, their can be buttons/ sliders for the rotation. A next button can be used to change the current mode to the next one. Minimal changes would have to be applied for the code. Instead of detecting key presses, button events will have to be detected.

3.4 Why is the use of centroid important? (Hint: Consider it with respect to rotation and scaling.)

Centroids are useful when the primitives are symmetric. It's easier to compute the endpoints of the primitive, then divide the primitive into triangles and render. In other words, it is sufficient to know the centroid of a primitive when performing transformations. This is beneficial for transformations like scaling and rotation which by default occur around the origin. If you have to perform these transformations around any other point, then additional sequence of vector transformations have to be applied to get the desired result. Instead of applying these on every single point in the primitive, it is sufficient to apply on the centroid. Once the final position of the centroid is computed, the rest of the points can be rendered around the centroid.

4 REFERENCES

https://webglfundamentals.org/
https://tex.stackexchange.com/questions/107832/how-to-create-internet-link-in-pdf