CS606: Computer Graphics / Term 2 (2020-21) / Programming Assignment 1

Shashank Reddy - IMT2018069
shashank.reddy@iiitb.org

International Institute of Information Technology, Bangalore. — January 30, 2021

Problem Statement

To generate an interactive 2D planar with 2D translation, rotation, scaling/zooming.
This program has 3 control modes namely mode 0, mode 1 and mode 2. The specifications for each mode are-

1. **Mode 0:**
   In this mode we can create certain fixed primitives, namely a Square, a Rectangle and a Circle. To do this, first we have to select the shape. For this, we can press "r" to select a rectangle, "c" to select a circle and "s" to select a sqaure respectively.
   We can then click anywhere on the canvas to draw the primitives.

   Mode: 1
   In this mode, we can translate and scale selected primitives.
   We can select a primtive by clicking on it.
   To scale the primitives about their centroid, we press the "+" key to enlarge and "-" key to minimize the primitive.
   To translate the objects, we can press the "UpArrow", "LeftArrow", "DownArrow", "RightArrow" to translate the primitives in the +y, -x, -y, +x direction(s) respectively.
   In this mode, we can also delete objects from the scene. To do this, we first select the object by clicking on it and then pressing "x".

   Mode: 2
   In this mode, the program automatically generates a tight rectangular bounding box for all the primitves in their object space and calculates the centroid of this bounding rectangle. We can then rotate all the primtives within this bounding rectangle about it's centroid by pressing the "RightArrow" and the "LeftArrow" key(s) to rotate them in the clockwise, anticlockwise direction(s) respectively.
   In this mode we can also exit the program and take a screenshot of the canvas by pressing the "d" key. This screenshot gets saved on your PC.

1 Problem Solution

We can discuss the problem solution by first dividiing the entire problem into smaller sub-problems that are easier to solve and later combine these solutions.

1.1 Getting the co-ordinates of a mouse click on the canvas

To solve this, we will use the mouseToClipCoord() function in renderer.js that converts the co-ordinates of the mouse click to co-ordinates in our clip space.

renderer.js

```
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]
 }
```

Once we have the co-ordinates of the mouse click in the clip space we can then pass these coordinates to various functions to construct the primitives (in Mode 0) or to select the primitives (Mode 1).

1.2 Rendering multiple objects at once

We store all generated objects in a list called sceneObjects[]. After each event (key press / mouse click), we clear the canvas and render all these primitives again.

1.3 Selecting primitives by a mouse click

For this, we calculate the Euclidean Distance between the clip coordinates of the mouse click and the centroid of the primitives. We then select the primitive whose Euclidean distance is the least. To show that a primitive has been selected, we color it black.

index.js

```
let min_distance = 1000.0;

marker = sceneObjects.length;

for(let i = 0; i < sceneObjects.length; ++i){
    let centroid = sceneObjects[i].getCentroid();

    let l2_norm = L2_Norm(clipCoordinates,centroid);

    if(min_distance > l2_norm){
        min_distance = l2_norm;
        marker = i;
    }
}

for(let i = 0; i < sceneObjects.length; ++i){
    sceneObjects[i].color_normal();
}
if (marker!=sceneObjects.length){
    sceneObjects[marker].color_black();
}
}
```

"Marker" is a pointer to the selected primitive.

1.4 Deleting a primitive from the scene

For this we simply remove the primitive from the list of primitives in the scene. (sceneObjects[]).

1.5 Translating a primitive

For this, we use transformation matrices. To generate these matrices, I used the gl matrix library.
I implemented the translation in such a way that it stores the previous translation vector (initially set to [0,0,0]). So every time we have to translate the primitive, I first get the previous translation vector and then update it by some translation factor in the required direction. w.l.o.g, we will look at the code for translation of the primitive in +x direction.

index.js

```
let translateDist = 0.1;

let prev = sceneObjects[marker].transform.getTranslate();
prev[0] += translateDist;
sceneObjects[marker].transform.setTranslate(prev);
sceneObjects[marker].transform.updateMVPMatrix();

let prevCentre = sceneObjects[marker].getCentroid();
prevCentre[0] += translateDist;
sceneObjects[marker].setCentroid(prevCentre);
```

1.6 Scaling a Primitive

For this as well I used translation matrices from the gl matrix library.
Just like in the case of translation, I implemented the scaling in such a way that it stores the vector by which the primitive was previously scaled by. After we get this vector, since we are scaling uniformly in the x and y direction, I add a scaling factor to the previous scaling vector in the x and y directions and use this for scaling.
Since, scaling is done about the origin, I first store the coordinates of it's centroid and then create a new primitive at the origin and scale it by the new scaling vector. I then translate this newly created primitive such that it's centroid is aligned with the centoid of the previous primtive that had to be scaled. I then replace the old primitive with the newly created primitive in the list of existing scene objects.

index.js

```
let scalingFactor = 0.1;

let name = sceneObjects[marker].getName();

let prevScale = sceneObjects[marker].transform.getScale();
let prevCentre = sceneObjects[marker].getCentroid();

var obj;

switch(name){
    case "circle":
        obj = new Circle(gl,[0,0]);
        break;
    case "square":
        obj = new Square(gl,[0,0]);
        break;
    case "rectangle":
        obj = new Rectangle(gl,[0,0]);
        break;
}

obj.color_black();

prevScale[0] += scalingFactor;
prevScale[1] += scalingFactor;
obj.transform.setScale(prevScale);
obj.transform.setTranslate([prevCentre[0],prevCentre[1],0]);
obj.transform.updateMVPMatrix();
obj.setCentroid(prevCentre);
sceneObjects[marker] = obj;
```

1.7 Regenerating the scene

Since, the rotations we apply in mode 2 are temporary, I wanted to recreate the scene in mode 2 with a new list of primitives so that the rotation operations on them do not affect my original set of scene objects. To do this, I made a custom deep copy function that returns a deepcopy of sceneObjects[]. In this function, I just create a new primitive with it's centroid at the centroid of the primitive we're trying to copy. Now, we set the dimensions of the primitive based on how the object we're trying to copy has been scaled previously. The calculations for this can be seen in the code.

```
function deepCopy(sceneObjects){
    var sceneObjectsdeepCopy = [];
    for(let i = 0; i < sceneObjects.length; ++i){
        let name = sceneObjects[i].getName();
        var new_obj;
        switch(name){
            case "circle":
                new_obj = new Circle(gl,sceneObjects[i].getCentroid());
                new_obj.setRadius(new_obj.getRadius()*sceneObjects[i].transform.getScale()[0]);
                break;

            case "square":
                new_obj = new Square(gl,sceneObjects[i].getCentroid());
                new_obj.setLength(new_obj.getLength()*sceneObjects[i].transform.getScale()[0]);
                break;

            case "rectangle":
                new_obj = new Rectangle(gl,sceneObjects[i].getCentroid());
                new_obj.setLB(new_obj.getDimensions()[0]*sceneObjects[i].transform.getScale()[0],new_obj.getDimensions()[1]*sceneObjects[i].transform.getScale()[0])
                break;
        }
        sceneObjectsdeepCopy.push(new_obj);
    }
    return sceneObjectsdeepCopy;

}
```

Figure 1: deepCopy Function

1.8 Calculating the centroid of the bounding rectangle

The calculations for this are quite straightforward. I assume that initially the bounding rectangle has sides parallel to the x and y axis. Now, to ensure that all the primitives are within the bounding rectangle, I calculate the rightmost point in all the primitives (greatest x value) , this can be done by looking at the rightmost point in each primitive (calculation in the code) and checking if the x coordinate of this point is greater than what you have already encountered, if yes then update this value, we can then set the right side of our bounding rectangle as x = the x coordinate of this point. w.l.o.g we can do this in all 4 directions and get the rightmost, leftmost, topmost, bottom-most points in all the primitives and then generate our bounding rectangle. Once we have this, it is trivial to find the centroid of this rectangle.

1.9 Rotating all the primitives about the centroid

For each rotation, I rotate each primitive by some rotation angle about the centroid of the bounding box. Just like translation and scaling, I store the angle by which the primitive had been rotated before and then increase it by some rotation factor.
Since rotation occurs about the centre, I tranform the axis such that the centroid is at the origin. To do this I create a new primtive with it's centroid at the coordinates (scene.centroid.x-primtive.centroid.x,scene.centroid.y-primtive.centroid.y). I also store the dimensions of the old primitive and use this to scale my primitive accordingly.
Then, we rotate the new primitive based on the new derived rotation angle and translate it with the translation vector $(oldprimitive.centroid.x, oldprimitive.centroid.y, 0)$.
We then replace the old primitive with the newly generated one.

2 QAs

1. **How did you program separate transformation matrices for all the object instances (primitives), and the scene?**
   I used the GL Matrix library to generate the transformation matrices, namely the rotate, scale, translate and the identity functions and also vec3 and mat4. On the shader side I used uniforms to store matrices cause they do not change from one shader invocation to another. Each object, has it's own transformation matrix which is passed to the GPU. Some more details of the use of the matrices are found in sections 1.5, 1.6 and 1.9.

```
getCentroid(){
    for(let i = 0; i < this.sceneObjects.length; ++i){
        let name = this.sceneObjects[i].getName();
        let centroid = this.sceneObjects[i].getCentroid();
        //console.log(centroid);
        switch(name){
            case "circle":
                let radius = this.sceneObjects[i].getRadius();
                this.maxX = Math.max(this.maxX,centroid[0]+radius);
                this.minX = Math.min(this.minX,centroid[0]-radius);
                this.maxY = Math.max(this.maxY,centroid[1]+radius);
                this.minY = Math.min(this.minY, centroid[1]-radius);
                break;

            case "square":
                let l = this.sceneObjects[i].getLength() / 2;
                this.maxX = Math.max(this.maxX,centroid[0]+l);
                this.minX = Math.min(this.minX,centroid[0]-l);
                this.maxY = Math.max(this.maxY,centroid[1]+l);
                this.minY = Math.min(this.minY, centroid[1]-l);
                break;

            case "rectangle":
                let len = this.sceneObjects[i].getDimensions()[0] / 2;
                let b = this.sceneObjects[i].getDimensions()[1] / 2;
                this.maxX = Math.max(this.maxX,centroid[0]+len);
                this.minX = Math.min(this.minX,centroid[0]-len);
                this.maxY = Math.max(this.maxY,centroid[1]+b);
                this.minY = Math.min(this.minY, centroid[1]-b);
                break;
        }
    }
    if(this.sceneObjects.length>0) this.centroid = [(this.maxX+this.minX)/2,(this.maxY+this.minY)/2];
    return this.centroid;
}
```

Figure 2: Centroid Calculation

2. **What API is critical in the implementation of "picking" using mouse button click?**
   An eventListener() for a click on the canvas is key in implementing the "picking" by using the mouse button click. Some other important elements are the mouseToClipCoord() [Section 1.1] and the Euclidean Distance [Section 1.3] which helps us to accurately pick primitives on the canvas.

3. **What would be a good alternative to minimize the number of key click events used in this application?**
   We can reduce the number of key presses by assigning a unique key for each mode, thus when we have to switch from Mode 0 to Mode 2 we can do it in one key press instead of 2. We can also add some kind of interactive GUI to reduce the number of key-click events.

4. **Why is the use of centroid important?**
   The use of the centroid is very important as scaling and rotation are done about a point in case of scaling and about an axis and a point in the case of rotation. This point about which the transformations occur remains the same [No change in coordinates]. Thus the centroid acts as a good choice as our primitves posses a certain amount of symmetry about the centroid. The origin may be another choice, however, in most cases it does not satisfy our requirements.

3 Conclusion

This assignment gave us a taste of computer graphics and taught us many things such as-

1. Introducition to JavaScript.
2. WebGL programming.
3. Creation and rendering of simple 2D geometric shapes as primitives.
4. Transformation of 2D objects and scene – instance-wise implementation.
5. Key(board) events for changing control modes, and transformation implementation.
6. Mouse events for picking points and objects.

I would like to thank Prof Jaya and Prof Srikanth for coming up with such an interesting assignment and the TAs for their tutorials and quick responses to doubts we had when attempting this assignment.