CS606 Computer Graphics
Spring 2020
Agam Kashyap
IMT2018004

Introduction to OpenCV

Problem Statement

The given task is to create an interactive 2D plane wherein the user can add 2D shapes by clicking on the plane, can move the objects using arrow keys and scale the same objects using +/- keys. The final task is to consider the objects drawn until now as one singular entity and enable user to rotate them around its center using arrow keys.

Approach

Mode 0: Inserting Figures

There are two methods that I used to draw the shapes before deciding on the final method. The first method involved creating constructors utilizing the Mouse Click Coordinates. The coordinates of the vertices were defined with respect to the mouse click coordinates. This method was perfect until I reached **scaling**.

To explain the second method, I will also answer the following question that was asked.
Question 4: Why is the use of centroid important?
First let us look at it from the perspective of drawing a figure and apply transformation to it. We use the one centroid to draw a complete shape, computing all of its vertices using that centroid. So when we have to apply any transformation to the figure, we just have to do one computation for the centroid, the rest of the vertices will then be calculated from the new centroid.
Now, this use case specifically helps in the method 1 that I mentioned. The issue that it brings forth is, since the transformation matrices are applied based on the origin, the scaling also happens with respect to the origin and this leads to the object appearing to move while it is getting scaled. to solve this issue, the other approach that can be taken is to draw the object always on origin by default and then translate the object to the given mouse click value, which becomes its centroid. So when the transformation matrix is called, the object gets scaled or rotated at the origin itself and then translates to the mouse click.

The issue that basing the object construction based on mouse click, is that the scaling happens with respect origin. So, if my object is drawn with the mouse click coordinates, it's going to look like its moving from its position while scaling.

To solve this issue and to further make the whole process simpler, I utilised the method of drawing the object first at the origin and then translating it to the mouse click coordinates.

The shape objects that are being drawn are getting added to an array called *Figures* which I then iterate over to draw them all.

Mode 1: Translation and Scaling

I used the matrix transformations here for performing the operations of translations and scaling. I will answer this question along with explaining the approach.

Question 1: How did you program separate transformation matrices for all the object instances (primitives), and the scene?
I created a separate class for handling the transformations, that is, multiplication of matrices. For each of Shape objects, I defined a *translation* variable that would be modified whenever a user would press the arrow keys. Similarly I defined their specific *rotation* and *scale* variables, and a *Transform* object. So, when I am drawing these objects, the *Transform* object of each Figure is getting called, so correspondingly their transformation matrices are getting created and multiplied to the objects original vertex matrix.

Question 2: What API is critical in the implementation of "picking" using mouse button click?
In my program, the method of "picking" an object is intuitive. Say you created a square which overlaps partially an already existing circle. When you click on part of the circle that is visible, the circle will be selected and when you click on the square the square will be, even if it is over the part that hides the circle. To achieve this I used simple geometry, wherein, when the user is in mode 1 and a click is detected, I iterate through all the figures in the array *Figures* and call their is_inside() function which return if the mouse click happened inside those objects and their **order**. **Order** is defined as the ranking in which the elements were drawn on the canvas or in the other words their indices in the *Figures* array. Now, we have all the shapes over whom the mouse click happened and their order. The one with the highest order is the one which was added the latest and is the one which will on top of all the other shapes. Hence I return this object as the selected figure and change its colour to gray.

Mode 2: Scene Rotation

This task involved several trials. I approached this problem by splitting it into different parts. First I decided to find the bounding box. For doing that, I retrieved the minimum and maximum X,Y coordinates i.e. bottom left corner and top right corner, which for the circle are the vertices present at 0, 90, 180, 270 degrees with respect to positive X axis. Amongst all these values, I select the minimum for X and Y and similarly the maximum. This will give me the perfectly fitting bounding box for all the figures. Now, I get the center of this box which is around which we have to rotate all the figures.

Now for the second task, I tried moving the figures from this bounding box center

to the origin because the flow that is required to achieve the task of scene rotation is:
Location: Origin
Action: Scale $\longrightarrow$ Translate to figure's original position
Location: Figure's original position
Action: Translate by negative of bounding box center's coordinates
Location: Scenery shifted with the bounding box center as origin
Action: Apply Rotation $\longrightarrow$ Translate by positive bounding box's center coordinates

So in the exact same flow, I approached the transformations, and got it working. Since we also had to reset the positions of the objects as they were before the mode 2 transformations, all I had to do was set the *rotationAngle* variable of each figure back to 0 and reset the rotation angle and axis in its transform object as well.

In my project you can see the Mode value and the Shape value in the top left corner, for ease of using the program. The code structure is modular with the different files handling the subsequent tasks plus some more functionalities:

1. **Renderer.js** : The first class object that is created, which handles creating the canvas and setting it up
2. **Shader.js** : The shader class object, which for the purpose of this assignment, is the same for all types of figures.
3. **vertex.js** : Contains the GLSL code for the vertex shader program
4. **fragment.js** : Contains the GLSL code for the fragment shader, wherein I have declared color as a uniform type variable, which lets me use the same fragment shader for all figures.
5. **transform.js** : Contains transform class with transformation functionalities
6. **Rectangle.js** : Rectangle class being used for Square and Rectangle
7. **Circle.js** : circle class being used for circle
8. **index.js** : The main file combining all the different classes to produce the program
9. **index.html globalStyle.css** : HTML and CSS code for webpage

Question 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.

One obvious solution to this is to provide an interface with buttons to the users wherein:
For Mode change we will have a button
In mode 0, user could select one of three buttons for each figure instead of using keyboard
Similarly in mode 1, the user could be provided with a sliding bar for translations

and scaling instead of using arrow keys and +/-. But from the perspective of the user, using arrow keys and +/- keys makes our program more Intuitive
And for the same reasons, even though we could provide two buttons for rotation in mode 2, I believe using the arrow keys is a better choice.

But one thing that could be done in mode 1 is using the left mouse button to hold and drag an object around the screen. So, for implementing it, we could set a timeout on mousedown, and if the time crosses it, we'll start to translate the object to where we move the mouse cursor to before releasing the mouse.

Similarly in mode 2, we could use the dragging function for performing our rotation of scene. The implementation could get a bit complicated, but the basic approach would be to register the mouse coordinates where it was held down and as the mouse moves, use the *new mouse coordinates*, *old mouse coordinates* and the *bounding box center coordinates* to calculate the angular distance moved, and send that value to the transformation matrix.

Conclusion

In the end I was able to complete the required tasks. Through this project I got a clear understanding of the matrix multiplication concepts underlying the transformation, and the fact that the matrices in WebGL store elements column wise, since I wrote the multiply functions on my own. Further on I understood the stacking process applied in the transformations, because of which the action that we want the figure to do first shall be written last. I was also able to get an understanding of the programmability offered by the Shaders.