CS 411 – Computer Graphics

Assignment 2 – 2D Modelling and Viewing

Mayank Bansal – mbansal5@hawk.iit.edu

October 5, 2017

1. Problem Description

In this project, we were to write a WebGL program that satisfied the needs below:

a) Generate a random path made of connected line segments and move a 2D object about its center on the path. When reaching a boundary point the window should bounce the project.
b) Rotate the object about its center while it is moving.
c) Add buttons to increase/decrease speed, zoom, stop animation.
d) Add button which toggles mode in which past positions of the object are drawn as line segments
e) Add a button which toggles a mode that makes the object orient itself in the direction of motion by calculating the rotation matrix given the movement vector.

Support code, and a skeleton program was given to help start with the program.

2. Method and Problems Incurred

2.1 Moving the Triangle

To initially start moving the object, I simply altered ***mvMatrix*** to **translate** by ***curPosX*** and ***curPosY*** so that the object would move.

```
mvMatrix.translate(curPosX, curPosY, 0);
```

2.2 Getting the Board Scale Right

The board dimensions weren't giving an output, so I changed the **boardW** and **boardH** to 2.0.

```
var boardW = 2.0;          // board width
var boardH = 2.0;          // board height
```

2.3 Rotating the Triangle at Constant Angular Velocity

To get the object rotating at a constant angular velocity, I applied a **rotate t**ransformation on ***mvMatrix*** by ***curRotAngle***.

```
mvMatrix.rotate(curRotAngle, 0, 0, 1);
```

2.4 Render Mode Toggle

Next, I had to get the past segments to draw. The support code had already given code to draw the past path of the moving object, ***drawScene()*** function was not drawing the triangle if the render mode was toggled. To get that working, I created 2 different buffer objects for storing the line vertices and for storing the triangle:

```
// CREATE LINE & VERTEX BUFFER
vertexBuffer = gl.createBuffer();
lineBuffer = gl.createBuffer();
```

After that, I set the attribute pointer:

```
var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(a_Position);
```

In the ***drawScene()*** function, I bind two buffer objects for each of the objects we wanted to draw, i.e., the triangle and past points.

```
// BIND lineBuffer
gl.bindBuffer(gl.ARRAY_BUFFER, lineBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(past), gl.DYNAMIC_DRAW);

// BIND vertexBuffer
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```

To draw the lines, I would check if the ***renderMod***e was true, and then bind the buffer, apply transformations and draw the appropriate component.

```
if (renderMode > 0) {
    // BIND lineBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, lineBuffer);
    var len = past.length / 2;
    gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

    // DRAW LINES
    gl.uniformMatrix4fv(u_ModelMatrix, false, mvMatrix.elements);
    gl.uniform4f(u_FragColor, 1, 1, 0, 1); // set color
    gl.drawArrays(gl.LINE_STRIP, 0, len); // draw line
}
```

Similary, after drawing the lines, we would bind the line buffer and draw the triangle and its center of mass.

```
// BIND vertexBuffer
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

gl.uniformMatrix4fv(u_ModelMatrix, false, mvMatrix.elements);

// DRAW TRIANGLE
gl.uniform4f(u_FragColor, 1, 0, 0, 1); // set color
gl.drawArrays(gl.TRIANGLES, 0, 3); // draw triangle

// DRAW CENTER OF MASS
gl.uniform4f(u_FragColor, 1, 1, 1, 1); // set color
gl.drawArrays(gl.POINTS, 3, 1); // draw point
```

I found that the lines were being drawn on top of the triangle, so I decided to draw the lines first and then bind the triangle buffer to draw the triangle on top of the lines.

2.5 Moving Object in Direction of Motion

To move the object to that its always pointing in the direction of motion, we calculate the direction vector by the previous and past points of the current line, normalize the vector, and then calculate a perpendicular vector to it. After that, we set the rotation matrix and apply it to our ***mvMatrix***. Initially, the object started rotating based on the calculations but was always travelling 90 degrees perpendicular to the desired direction, so I rotated the whole object by -90 degrees after applying all the transformations.

```
if (rotateMode > 0) {

    ////////////////////////////////////////
    /// DEFINE ROTATION MATRIX FOR DIRECTION
    ////////////////////////////////////////

    var xDiff = past[past.length - 2] - past[past.length - 4];
    var yDiff = past[past.length - 1] - past[past.length - 3];

    var xDiffNorm = xDiff / (Math.sqrt(Math.pow(xDiff, 2) + Math.pow(yDiff, 2)));
    var yDiffNorm = yDiff / (Math.sqrt(Math.pow(xDiff, 2) + Math.pow(yDiff, 2)));

    var V = new Vector4([xDiffNorm, yDiffNorm, 0]);
    var V3DH = threeDto3DH(V);
    var M = new Matrix4();
    M.setRotate(90, 0, 0, 1);
    V3DH = M.multiplyVector4(V3DH);

    var xFormMatrix = new Float32Array([
        V.elements[0], V.elements[1], 0, 0,
        V3DH.elements[0], V3DH.elements[1], 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
    ]);

    M = new Matrix4();
    M.elements = xFormMatrix;
```

```
    mvMatrix.multiply(M);
    mvMatrix.rotate(-90, 0, 0, 1);
}
```

2.6 Scaling and Changing Speed of Animation

Changing the speed of animation was already given in the template and worked just as needed so there were no changes there.

To scale, initially I tried changing the viewport size, and setting a scale variable, but the viewport wouldn't scale correctly for scale < 1.0, so I simply added the following line to the ***drawScene()*** function:

```
mvMatrix.scale(scale, scale, 1);
```

To change the clipping box, I divided the boardW / boardH by the scale so that the object would move around accordingly:

```
if (curPosX < -(boardW / scale) / 2.0) { // left intersection
    curPosX = -(boardW / scale) / 2.0;
    dX *= -1;
}

if (curPosX > (boardW / scale) / 2.0) { // right intersection
    curPosX = (boardW / scale) / 2.0;
    dX *= -1;
}

if (curPosY < -(boardH / scale) / 2.0) { // bottom intersection
    curPosY = -(boardH / scale) / 2.0;
    dY *= -1;
}

if (curPosY > (boardH / scale) / 2.0) { // top intersection
    curPosY = (boardH / scale) / 2.0;
    dY *= -1;
}
```

The ***zoomIn()*** and ***zoomOut()*** function just changes the scale:

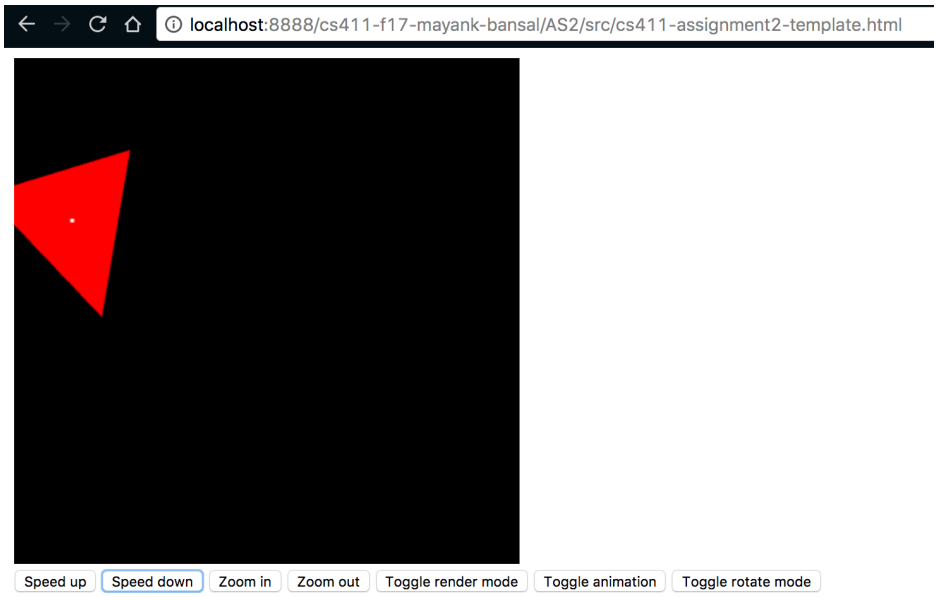
```
function zoomIn() {
    scale *= 1.25;
    console.log("Scale: " + scale);
}

function zoomOut() {
    scale /= 1.25;
    console.log("Scale: " + scale);
}
```

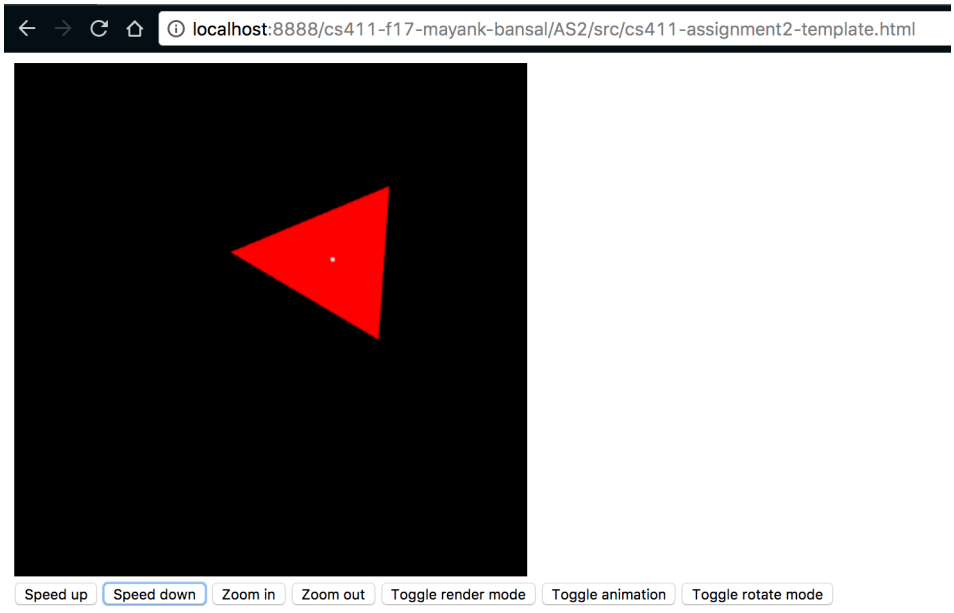
3. Results and Evaluation

Below there are the screenshots of different required outputs mentioned in the problem statement:

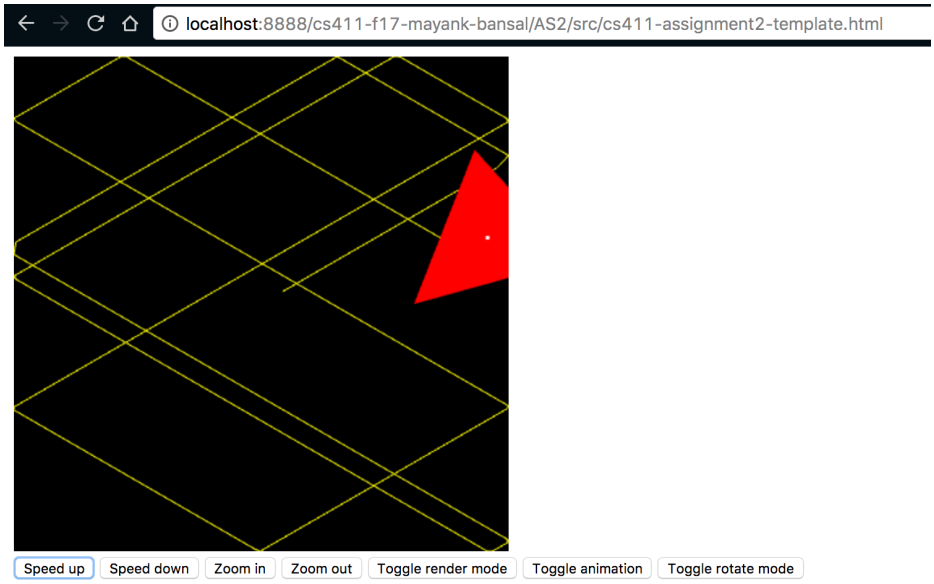
3.1 Moving Triangle



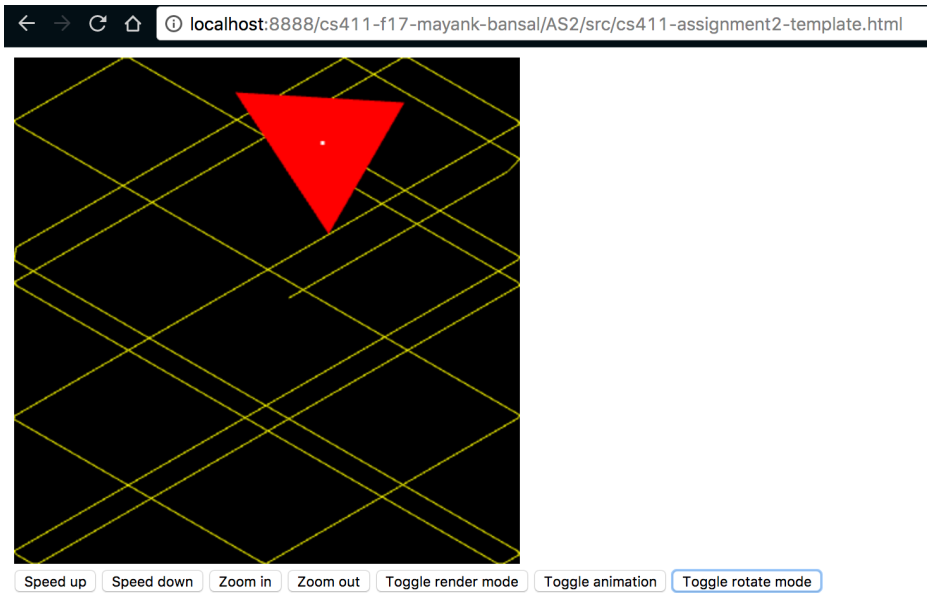
3.2 Constant Rotation



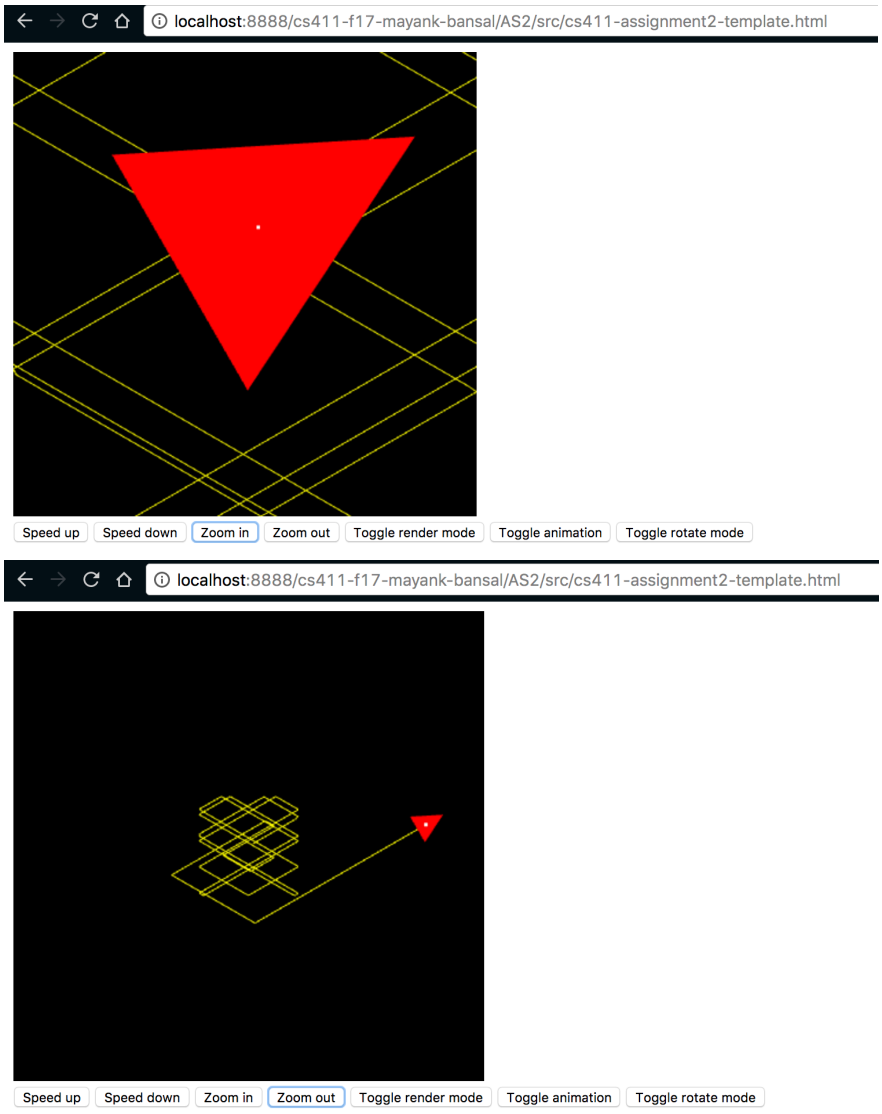
3.3 Path History



3.4 Rotation Mode Toggle



3.5 Zoom In / Zoom Out



4. Conclusion

The outputs shown in the previous section mimic the reference output given to us with no documented problems or missed cases.

5. Environment & Organization

This project was run and tested on ***Google Chrome Version 61.0.3163.100 (Official Build) (64-bit)***. Run out of a MAMP stack server. Simply pull the repository and run out of the SRC file.

The SRC folder contains 2 files:

cs411-assignment2-problems.html

Contains solutions to the first problem set, written in JavaScript. Output is located in the JavaScript console.

cs411-assignment2-template.html

Contains solutions to the programming WebGL problem set, written in JavaScript.