CS 411 – Computer Graphics

Assignment 3 – Curve and Surface Interpolation

Mayank Bansal – mbansal5@hawk.iit.edu

October 22, 2017

1. Problem Description

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

a) Let the user click the canvas at arbitrary locations. Store and display the clicked locations as points.
b) Upon receiving 4 points generate a Cardinal spline between the middle 2.
c) Thereafter, upon receiving any additional points continue to add additional curve segments while guaranteeing C1 continuity between the added curve segments. Each added point should result in an added curve segment.
d) Add two buttons for increasing/decreasing the tension parameter, and update the curve upon changing this parameter.
e) Add the ability to move an object along the spline path you generated:
    i. Add a button to toggle animating an object moving along the spline curve path.
    ii. Add two buttons for increasing/decreasing the speed of the object.

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

2. Method and Problems Incurred

2.1 Adding & Displaying User Points

The points are added using the ***click()*** function as shown below:

```
function click(ev, gl, canvas, a_Position) {
    // get display coordinates
    var x = ev.clientX; // x coordinate of a mouse pointer
    var y = ev.clientY; // y coordinate of a mouse pointer

    // convert viewport to world coordinates (viewport-to-window transformation)
    var rect = ev.target.getBoundingClientRect();
    x = ((x - rect.left) - canvas.width / 2) / (canvas.width / 2) * boardW / 2;
    y = (canvas.height / 2 - (y - rect.top)) / (canvas.height / 2) * boardH / 2;

    // store the point
    console.log('Storing point (%f,%f)', x, y);
```

```
    ctrlPts.push(x);
    ctrlPts.push(y);

    // process new points
    processPoints();
}
```

This function stores the points into the ***ctrlPts*** array. After that the points are processed to create the interpolation.

2.2 Generating Cardinal Spline Between Middle 2 Points

To first generate the Cardinal Spline, we must generate the Mc Matrix which will help us calculate the interpolated points. In the ***main()*** function, we update the Mc Matrix as follows:

```
// update Mc Matrix
updateMc();
console.log("Mc Matrix: ", Mc);
```

The ***updateMc()*** function calculates the Mc Matrix depending on the tension that is set at that point of time using the Cardinal Spline matrix as follows:

```
// update Mc Matrix function
function updateMc() {
    Mc = [
        -tension, 2 - tension, tension - 2, tension,
        2 * tension, tension - 3, 3 - 2 * tension, -tension,
        -tension, 0, tension, 0,
        0, 1, 0, 0
    ];
}
```

After the Mc Matrix is set, we can then process the points through the following function:

```
function processPoints() {

    intrPts = [];
    console.log("Redrawing...");

    // generate interpolation points
    var nCtrlPts = ctrlPts.length / 2;
    if (nCtrlPts > 3) {
        for (var i = 0; i < nCtrlPts - 3; i++) {
            // there are 4 points - interpolate between pk and pk+1
            console.log('adding interpolation points');
            var pk_n1 = {x: ctrlPts[0 + i * 2], y: ctrlPts[1 + i * 2]}; // p_{k-1}
            var pk_p0 = {x: ctrlPts[2 + i * 2], y: ctrlPts[3 + i * 2]}; // p_{k+0}
            var pk_p1 = {x: ctrlPts[4 + i * 2], y: ctrlPts[5 + i * 2]}; // p_{k+1}
            var pk_p2 = {x: ctrlPts[6 + i * 2], y: ctrlPts[7 + i * 2]}; // p_{k+2}
            interpolate(pk_n1, pk_p0, pk_p1, pk_p2);
```

```
        }
    }

    // indicate plot update
    refreshFlag = 1;
}
```

Here, if the number of points is greater than 3, then we run a for loop through all consecutive set of 4 points each. For example, if we have 7 points drawn at the time, then we interpolate each of these sets of points in this order:

p0, p1, p2, p3 → p1, p2, p3, p4 → p2, p3, p4, p5 → p3, p4, p5, p6

After getting the 4 set of points to interpolate, we call the interpolate function as defined below:

```
function interpolate(pk_n1, pk_p0, pk_p1, pk_p2) {

    var xCoords = [pk_n1.x, pk_p0.x, pk_p1.x, pk_p2.x];
    var yCoords = [pk_n1.y, pk_p0.y, pk_p1.y, pk_p2.y];

    var xCoordsMc = McMultiplyPoints(xCoords);
    var yCoordsMc = McMultiplyPoints(yCoords);

    for (var u = 0; u <= 1; u += uStep) {

        var Ux = [Math.pow(u, 3), u * u, u, 1];
        var Uy = [Math.pow(u, 3), u * u, u, 1];

        var xNewCoords = findPoint(xCoordsMc, Ux);
        var yNewCoords = findPoint(yCoordsMc, Uy);

        intrPts.push(xNewCoords);
        intrPts.push(yNewCoords);
    }
}
```

We first separate all the ***xCoords***, and ***yCoords*** from the given points. We then multiply those vectors with the Mc matrix, resulting in 4x1 matrices ***xCoordsMc*** and ***yCoordsMc***. We then multiply the ***u*** coefficients 1x4 vector with the ***xCoordsMc*** and ***yCoordsMc***, resulting in a 1x1 matrix giving us the ***x, y*** values for that value of ***u***.

The function ***findPoint()*** and ***McMultiplyPoints()*** simply multiply 1x4, 4x1 and 4x4 x 4x1 matrices respectively and are defined below as:

```
function McMultiplyPoints(v) {
    return [
        Mc[0] * v[0] + Mc[1] * v[1] + Mc[2] * v[2] + Mc[3] * v[3],
```

```
        Mc[4] * v[0] + Mc[5] * v[1] + Mc[6] * v[2] + Mc[7] * v[3],
        Mc[8] * v[0] + Mc[9] * v[1] + Mc[10] * v[2] + Mc[11] * v[3],
        Mc[12] * v[0] + Mc[13] * v[1] + Mc[14] * v[2] + Mc[15] * v[3]
    ];
}

function findPoint(v1, v2) {
    return v1[0] * v2[0] + v1[1] * v2[1] + v1[2] * v2[2] + v1[3] * v2[3];
}
```

Initially, I was having issues with generating the curve after the 4th point was drawn, and instead the curves were drawn only after 5 points were drawn, and it was because I had made a mistake with the interpolation for loop indices.

2.3 Adding Continuous Curve Segments

Since we used the last 3 drawn points for every 5th point, the for loop in the previous section, interpolates the points generating a continuous curve while guaranteeing C1 continuity between the added curve segments.

2.4 Increasing/Decreasing Tension

The increase and decrease tension function are defined below as:

```
function tensionUp() {
    tension += 0.1;
    console.log('tension = %f', tension);
    updateMc();
    processPoints();
}

function tensionDown() {
    tension -= 0.1;
    console.log('tension = %f', tension);
    updateMc();
    processPoints();
}
```

These function increment or decrement the tension variable, and the call the ***updateMc()*** function to update the values in the matrix based on the new tension set. After updating the Mc Matrix, we call ***processPoints()*** to then redraw the curves according to the new tension.

Initially, when I wrote the functions, the tension was increasing and decreasing, and the curves were getting updated but the previous curves with the previous tensions remained. For that, I

simply reset the interpolated points for every redraw in the ***processPoints()*** function as below:

```
intrPts = [];
console.log("Redrawing...");
```

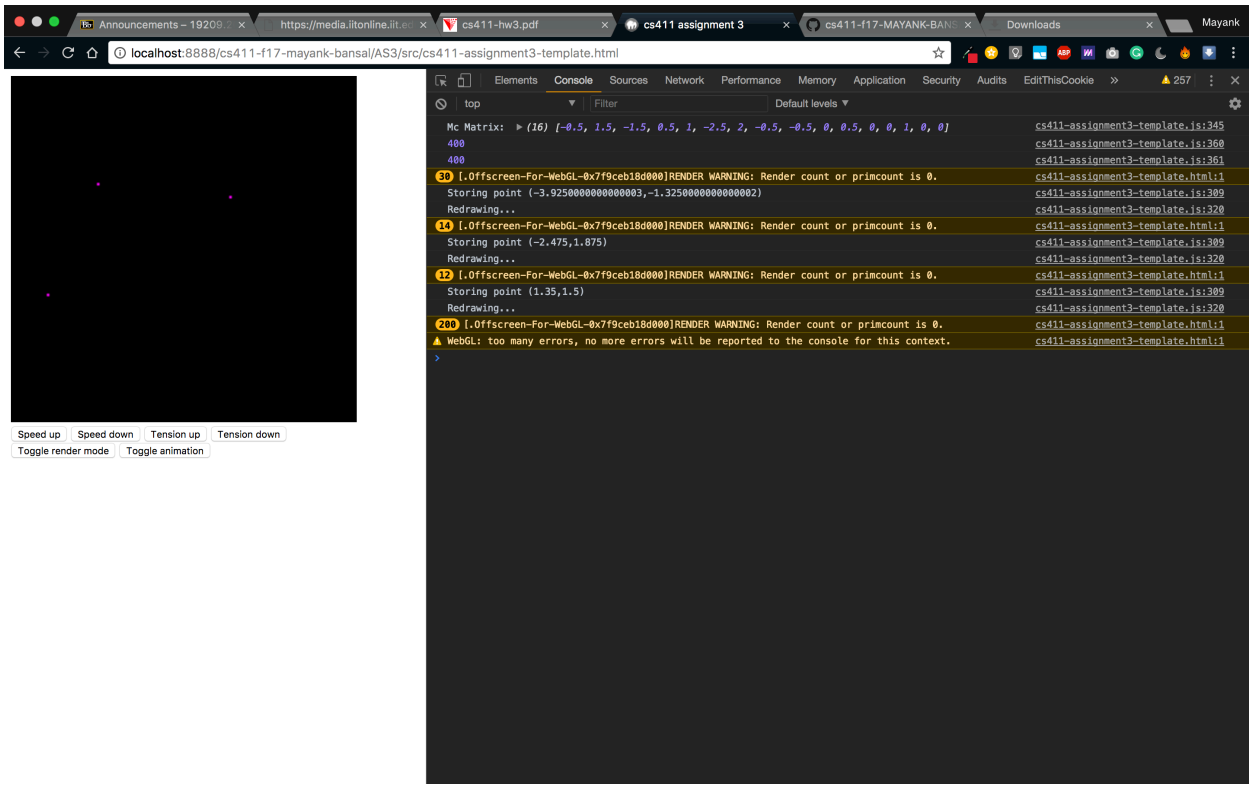
2.5 Moving Object on Spine

Once the curve is generated, the triangle object is drawn using the vertex shader program just as in Assignment 2. The object bounces off the last drawn point and retraces the path in the opposite direction.

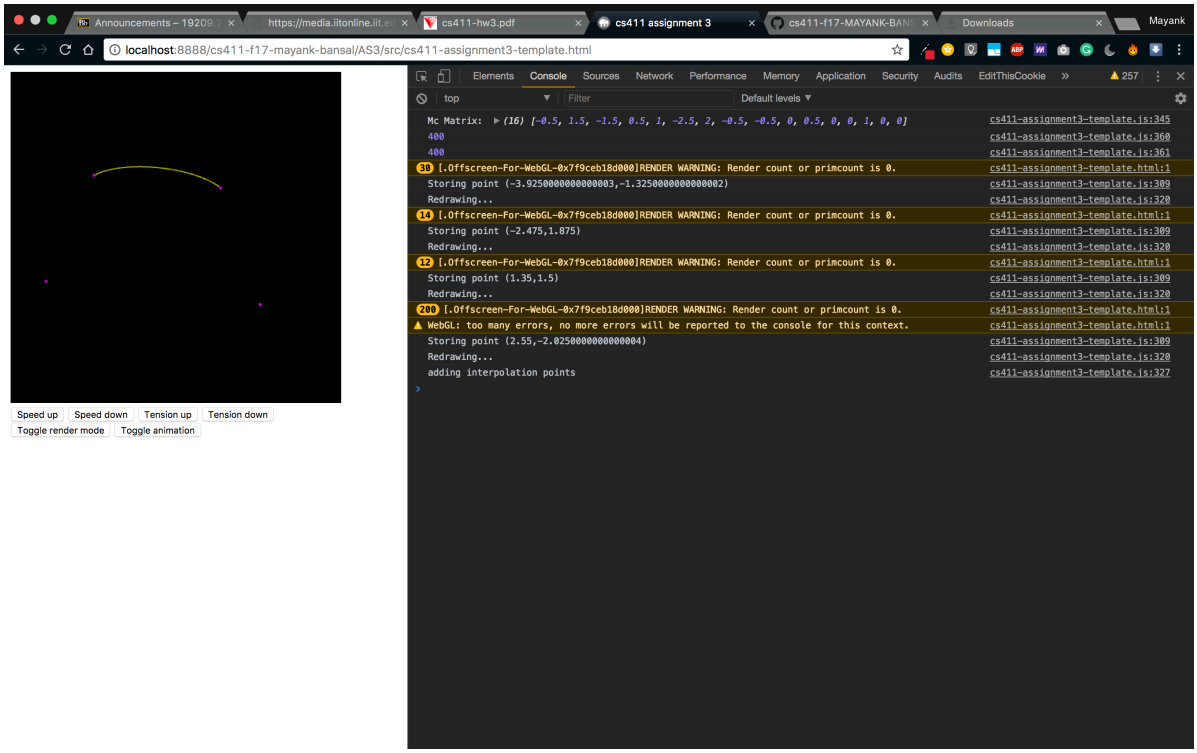
3. Results and Evaluation

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

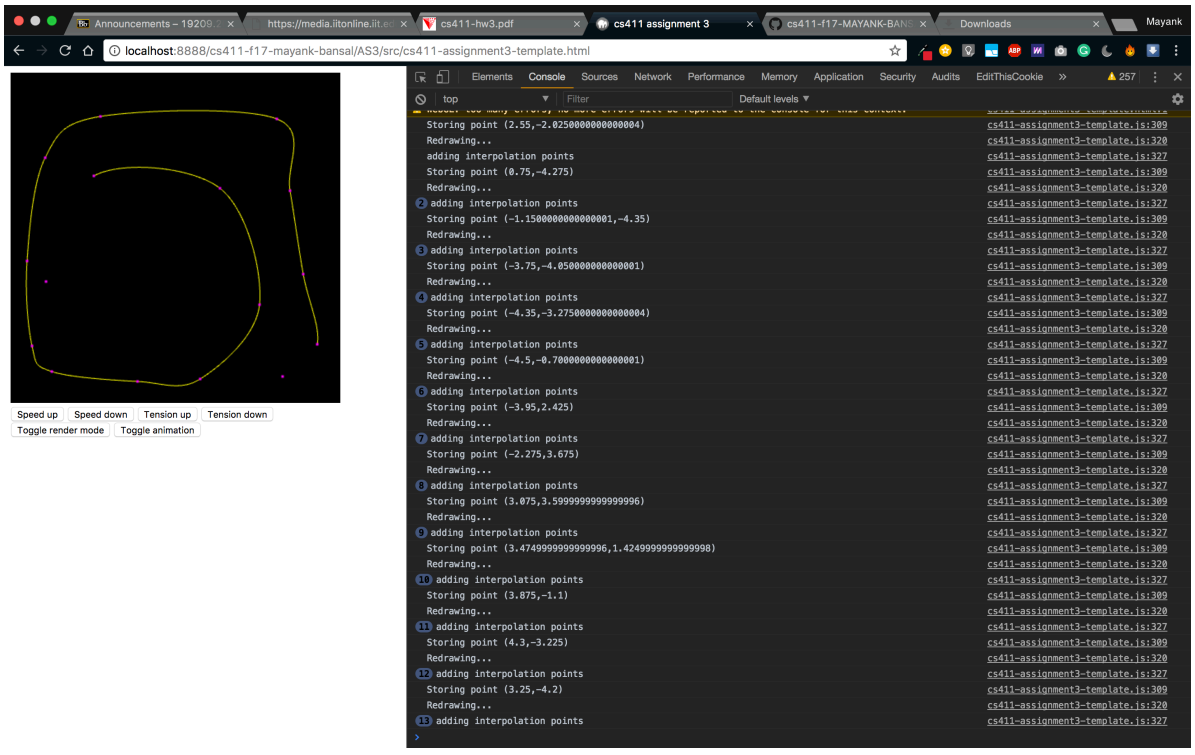
3.1 Adding & Displaying User Points



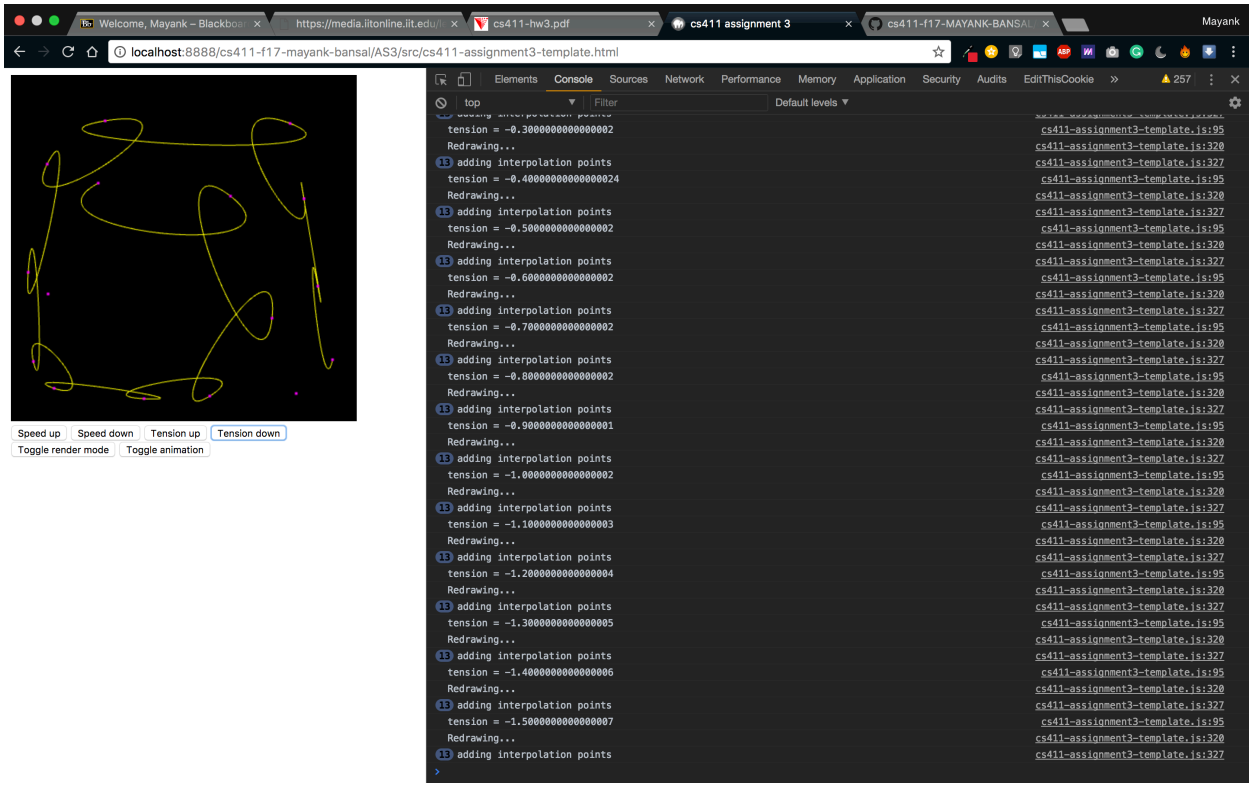
3.2 Generating Cardinal Spline Between Middle 2 Points



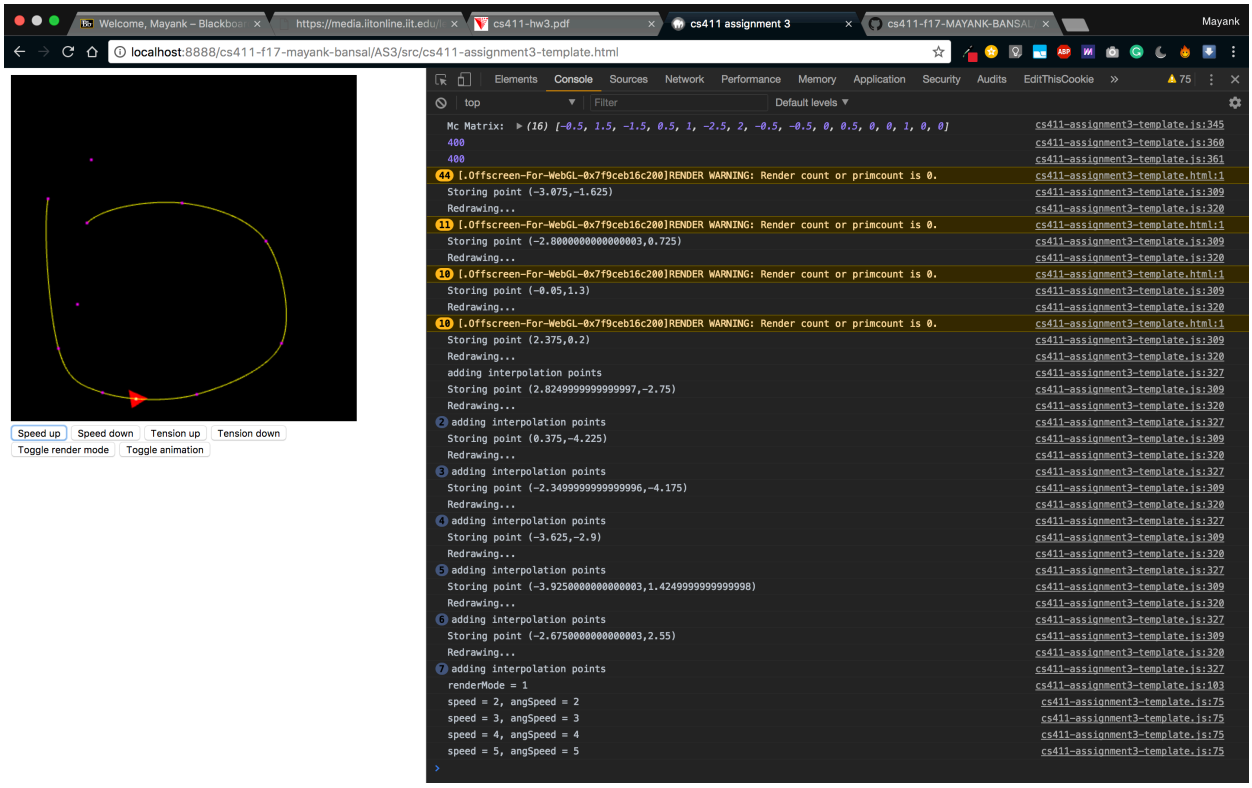
3.3 Adding Continuous Curve Segments

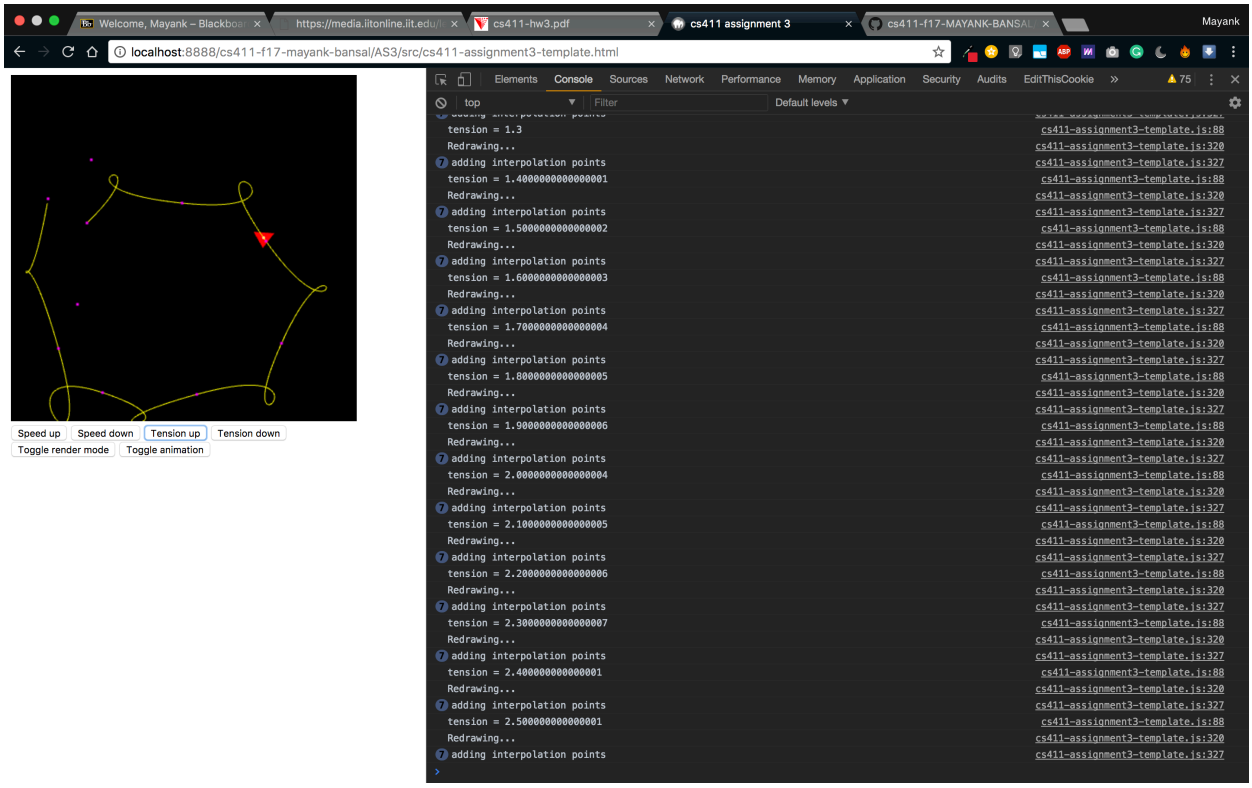


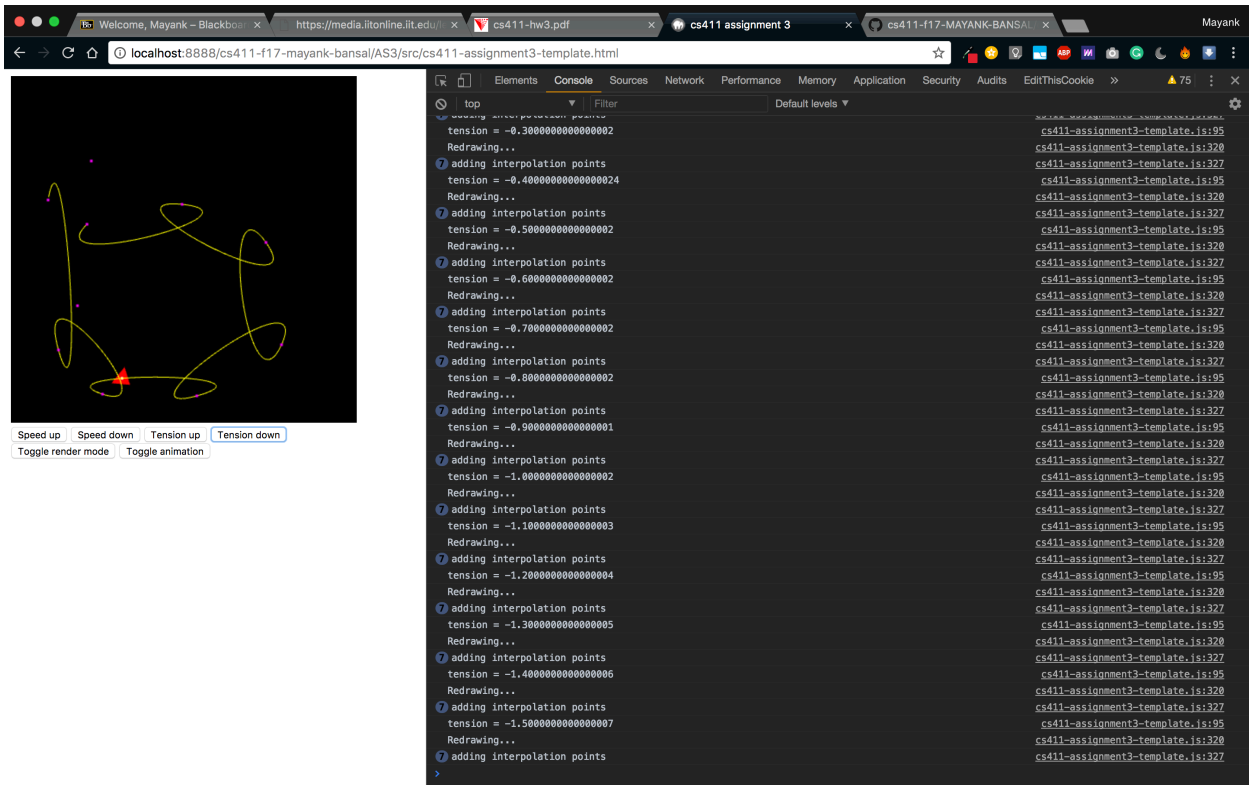
3.4 Increasing/Decreasing Tension



3.5 Moving Object on Spine







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-assignment3-problems.html

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

cs411-assignment3-template.html

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