Bezier Curves Documentation

How to run?

- This program is very simple webGL program. Please go to "./Source" and open index.html. You should able to run it without installing any external libraries.

User Interface: 2 Buttons

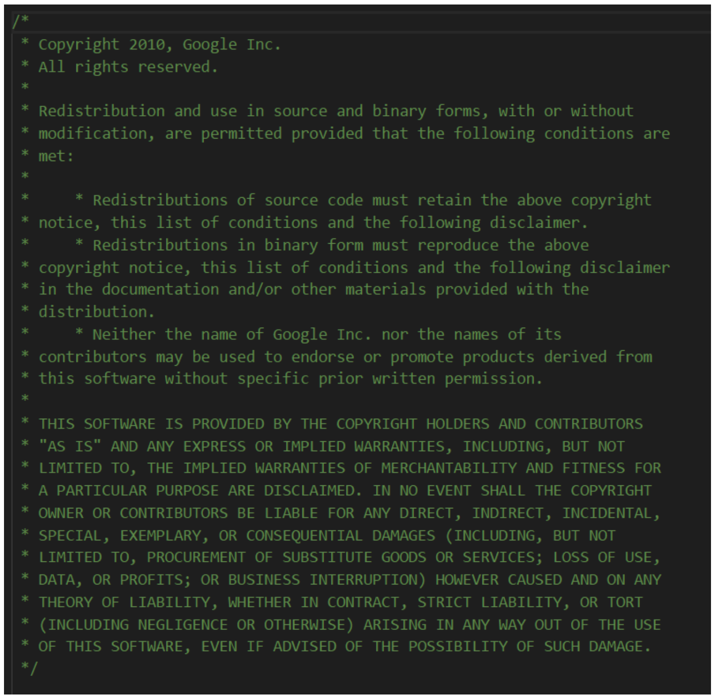
1. Bezier Curvature: this button will render a near-perfect curvature.
2. Wireframe: this button will render a wireframe which is drawn by connecting each control point.

Hardcoded Data

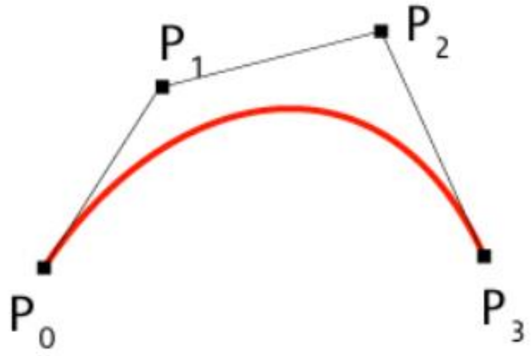
- The data are in "data.js" file inside "./Source" directory.
- Currently, I have hardcoded "teapot" and "teaspoon". You may uncomment either one to render whichever object you want.

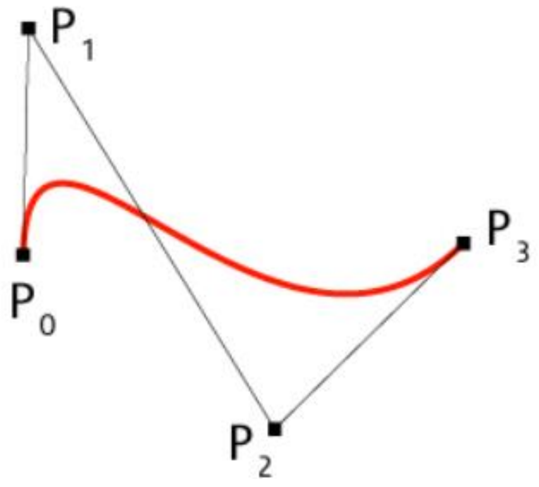
What I did

❖ As you observe, this program is very simple WebGL program. "./Helper" directory is an open source code that I have got from Google.



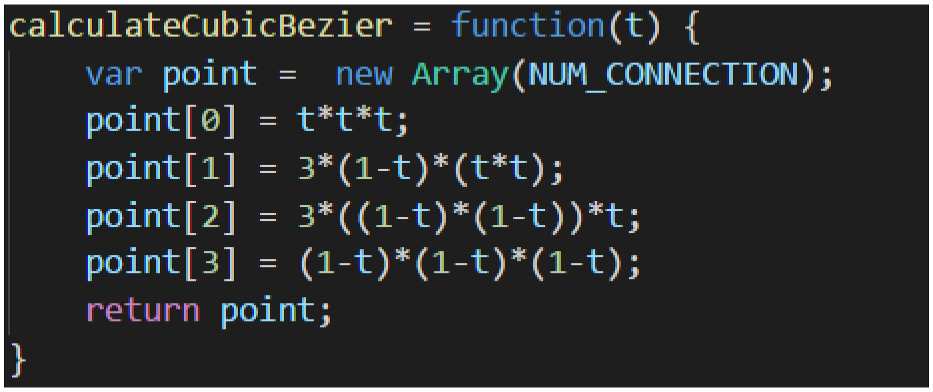
The main functionalities are in “bezier.js” file. What I have done is to use cubic Bezier formula to draw the curve. If you look at the global variables, you may see “NUM_CONNECTION = 4”. This means that I am using every 4 point to connect like the figure below.





If you look at “calculateCubicBezier” function inside “bezier.js” file, it’s pretty obvious what I am calculating.

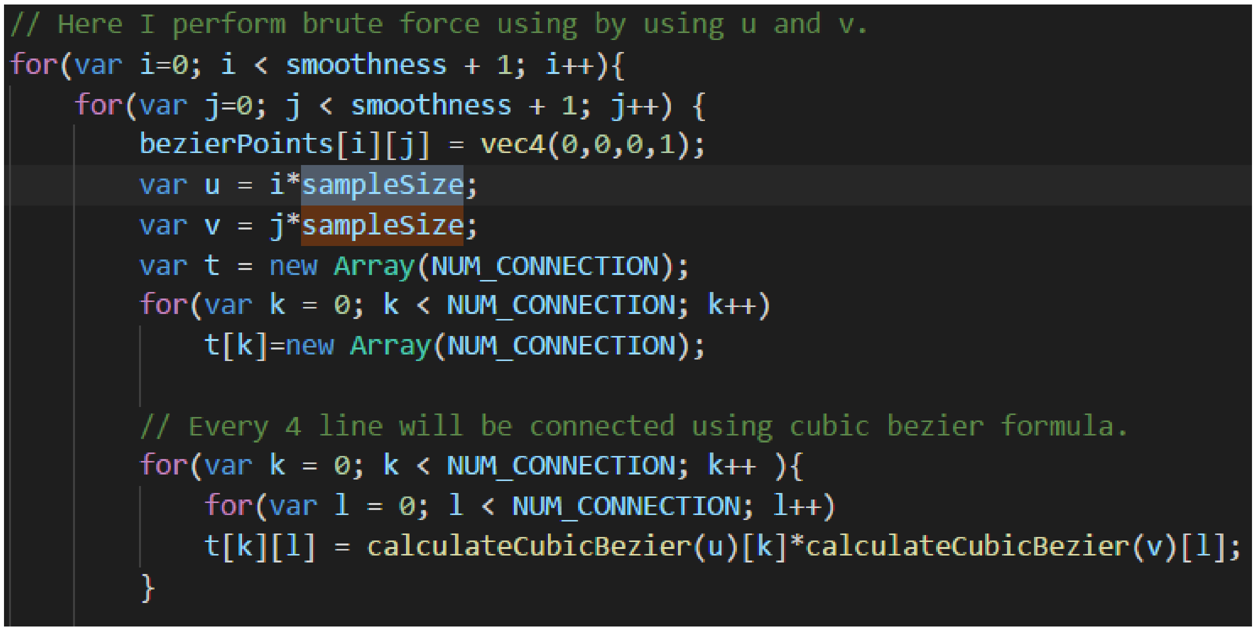
$$(1-t)^3\mathbf{P}_0 + 3t(1-t)^2\mathbf{P}_1 + 3t^2(1-t)\mathbf{P}_2 + t^3\mathbf{P}_3$$



```
calculateCubicBezier = function(t) {
    var point =  new Array(NUM_CONNECTION);
    point[0] = t*t*t;
    point[1] = 3*(1-t)*(t*t);
    point[2] = 3*((1-t)*(1-t))*t;
    point[3] = (1-t)*(1-t)*(1-t);
    return point;
}
```

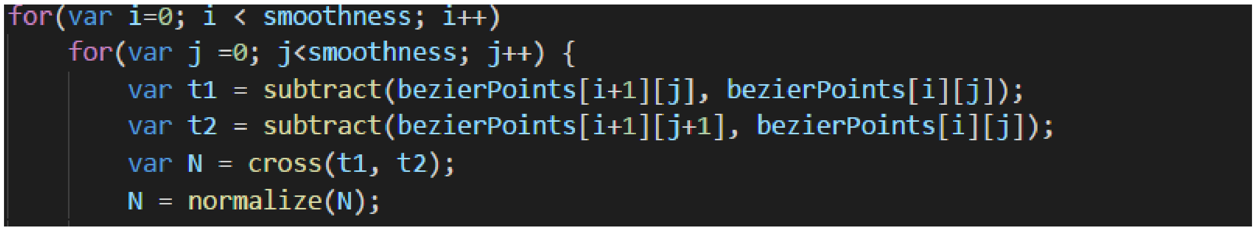
Now, using this function, I can find the point, but how did I actually draw the smooth curve using these points? If you look at the global variables section again, there exists “smoothness = 100”. This represents how smooth I want the curves to be. Using “smoothness = 100”, I calculate the sample size by “1.0 / smoothness”. So this means that I am calculating up to 0.001 floating point, which should be pretty accurate curve.

Using this smoothness and sample size, I have used a brute force method to draw the curvatures. I have decided to use the brute force method because it made more sense to me. I increment u and v gradually by sample size and find each point using Bezier formula, and connect each line to draw a curvature.



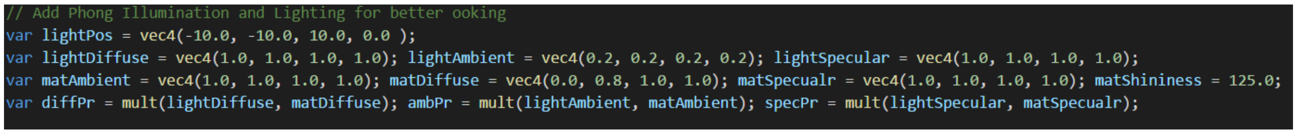
```
// Here I perform brute force using by using u and v.
for(var i=0; i < smoothness + 1; i++){
    for(var j=0; j < smoothness + 1; j++) {
        bezierPoints[i][j] = vec4(0,0,0,1);
        var u = i*sampleSize;
        var v = j*sampleSize;
        var t = new Array(NUM_CONNECTION);
        for(var k = 0; k < NUM_CONNECTION; k++)
            t[k]=new Array(NUM_CONNECTION);

        // Every 4 line will be connected using cubic bezier formula.
        for(var k = 0; k < NUM_CONNECTION; k++ ){
            for(var l = 0; l < NUM_CONNECTION; l++)
            t[k][l] = calculateCubicBezier(u)[k]*calculateCubicBezier(v)[l];
        }
```

Then, I have filled the patches using shaders. In order to fill the patches with colours, I had to calculate the normals. I calculate the normal for every point (the points that have been calculated by "calculateCubicBezier" function) in order to render more realistic object.



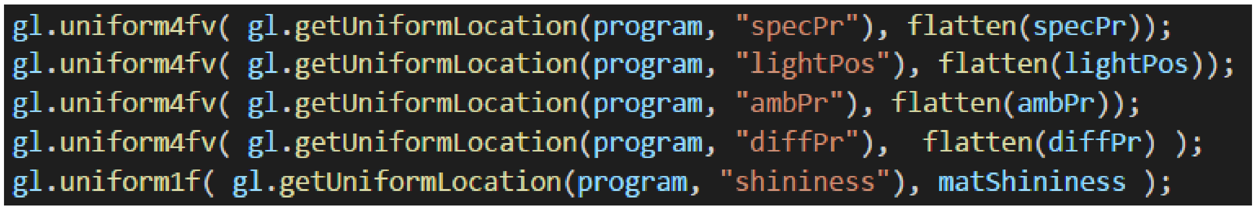
```
for(var i=0; i < smoothness; i++)
    for(var j =0; j<smoothness; j++) {
        var t1 = subtract(bezierPoints[i+1][j], bezierPoints[i][j]);
        var t2 = subtract(bezierPoints[i+1][j+1], bezierPoints[i][j]);
        var N = cross(t1, t2);
        N = normalize(N);
```

Then, I have used Phong Illumination using these declarations.



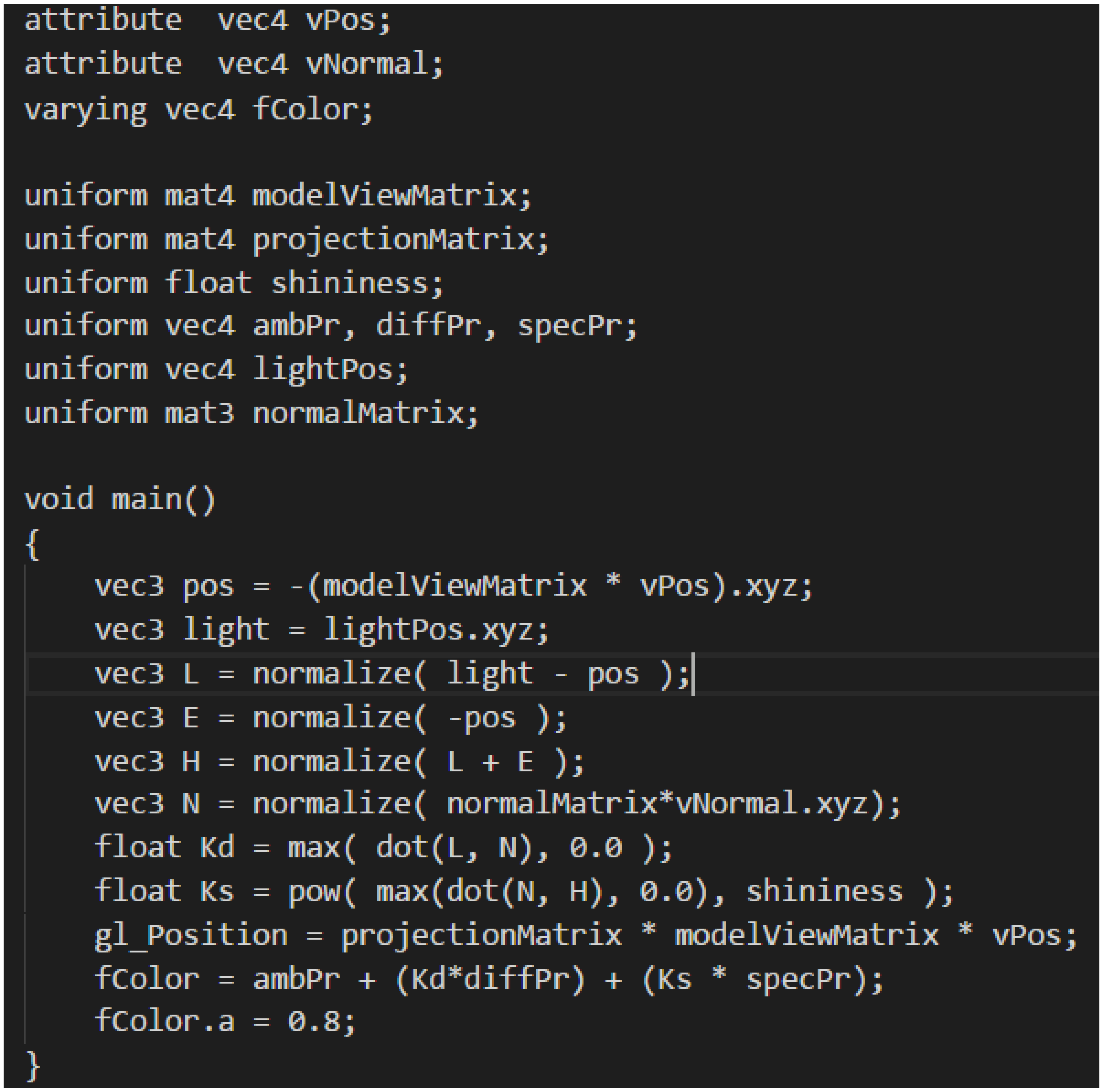
```
// Add Phong Illumination and Lighting for better ooking
var lightPos = vec4(-10.0, -10.0, 10.0, 0.0 );
var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0); lightAmbient = vec4(0.2, 0.2, 0.2, 0.2); lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);
var matAmbient = vec4(1.0, 1.0, 1.0, 1.0); matDiffuse = vec4(0.0, 0.8, 1.0, 1.0); matSpecualr = vec4(1.0, 1.0, 1.0, 1.0); matShininess = 125.0;
var diffPr = mult(lightDiffuse, matDiffuse); ambPr = mult(lightAmbient, matAmbient); specPr = mult(lightSpecular, matSpecualr);
```

These diffuseProduct, ambientProduct, specularProduct, shininess and etc. are then passed onto the vertex shader.



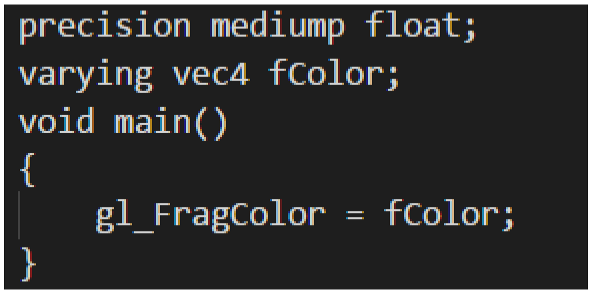
```
gl.uniform4fv( gl.getUniformLocation(program, "specPr"), flatten(specPr));
gl.uniform4fv( gl.getUniformLocation(program, "lightPos"), flatten(lightPos));
gl.uniform4fv( gl.getUniformLocation(program, "ambPr"), flatten(ambPr));
gl.uniform4fv( gl.getUniformLocation(program, "diffPr"),  flatten(diffPr) );
gl.uniform1f( gl.getUniformLocation(program, "shininess"), matShininess );
```

This vertex shader code is in “index.html” file. As you see, I am basically calculating initiating the scene with lights, 3d space and phong illumination.



```
attribute  vec4 vPos;
attribute  vec4 vNormal;
varying vec4 fColor;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform float shininess;
uniform vec4 ambPr, diffPr, specPr;
uniform vec4 lightPos;
uniform mat3 normalMatrix;

void main()
{
    vec3 pos = -(modelViewMatrix * vPos).xyz;
    vec3 light = lightPos.xyz;
    vec3 L = normalize( light - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );
    vec3 N = normalize( normalMatrix*vNormal.xyz);
    float Kd = max( dot(L, N), 0.0 );
    float Ks = pow( max(dot(N, H), 0.0), shininess );
    gl_Position = projectionMatrix * modelViewMatrix * vPos;
    fColor = ambPr + (Kd*diffPr) + (Ks * specPr);
    fColor.a = 0.8;
}
```

Then, of course there is fragment shader as well.



```
precision mediump float;
varying vec4 fColor;
void main()
{
    gl_FragColor = fColor;
}
```