Technical University of Denmark

02561

Computer Graphics

Drawing program with Rational Quadratic Bezier Curves

Authors:

Jeff Gyldenbrand

Student Nr.:

s202790

December 19, 2021

Contents

1 Introduction

The purpose of this project is to further develop upon my implementation of the 2D drawing program from worksheet 2[3]. My original drawing program is implemented in javascript / HTML and with the WebGL Shader. WebGL[2] is an API for rendering interactive 2D and 3D graphics to most browsers. The original implementation has the features:

- **3 drawing modes:**
  - Points: drawing single points
  - Triangles: place three points to create a triangle
  - Circles: place a center point and another point to define a circle
- **Coloring:**
  - Drawing: choosing between a set of colors before drawing points, triangles or circles. The last two modes can blend multiple colors
  - Canvas: choose between a set of colors for the background
- **Clear canvas:**
  - This resets the canvas to the selected canvas color

Placing points is done by clicking left mouse-button. The further implementation to the drawing program is an implementation of rational quadratic bezier curves. When this mode is chosen, the first point is the starting point, the next is a control point, and the last the end point. This gives a nice curvature between the points. Left image in figure 1.1 is the original drawing program. The right image is the final drawing program.

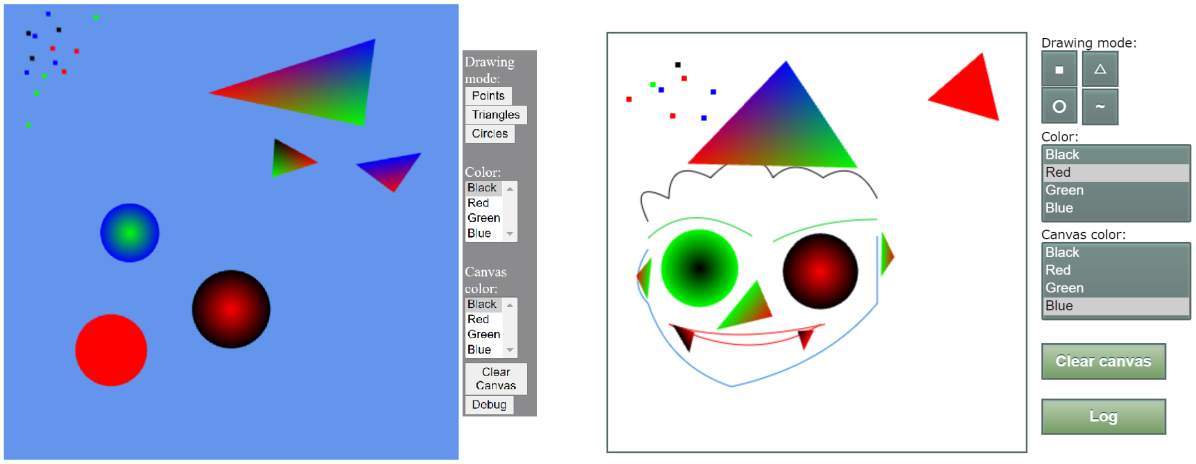

Figure 1.1: Left: original drawing program. Right: final program with bezier curves

This report will focus on the methods and math behind creating a drawing program as just described.

2 Methods

In order to draw interactive graphics on a HTML canvas to run directly in a browser, several methods needs to be implemented. In this section I will provide an abstracted overview of the different methods implemented. In depth explanations of these methods will be elaborated in section[3]

HTML / JS The HTML (HyperText Markup Language) describes a structure of the webpage semantically and can embed text, images, sounds, videos and other objects to be rendered to the browser. HTML works well with the javascript scripting language. The HTML-page[1] (project.html) for this project defines a table that contains two canvases. One for the main drawing program (webgl) and one for a head-up-display (hud). The main program, webgl, is responsible for all the original functionality from the W2 exercise, as mentioned earlier. In order to add the functionality for bezier curves I decided to implement this in the HUD-canvas. The smart thing is that the two canvases can be drawn on top of each other, such that the webgl-canvas can be responsible for 3D-graphics and the HUD-canvas can be responsible for 2D. In my case, the latter, is just responsible for drawing the bezier curves. Both implementations resides in the same javascript-file[2], but the logic is handled separately for the two canvases.

The HTML-page also contains all interactive elements such as, buttons, for choosing drawing modes and clearing canvas, and selectors for choosing colors. These all have an ID-tag that is referenced by the javascript-file. Some utility library files also resides in the HTML file in order to make use of the webgl functionalities.

To style up the design of the webpage I've created some CSS-files[3] (Cascading Style Sheets) which defines coloring, fonts, font-size, positition of elements, etc. Furthermore I've used the Bootstrap[1][4] framework to present the whole webpage (all exercises and the project) in a nice way. Files for this framework is downloaded such that the project webpage can be viewed offline.

Lastly, are the vertex and fragment shader directly defined in the HTML file.

SHADERS Shaders are programs that runs on the graphical processor unit (GPU). The GPU is a specialized processor that is designed to accelerate graphics rendering. GPU's are excellent at vector and matrix manipulations,

[1]Project HTML-page location: 'graphics\$js\project\project.html'$
[2]Project JavaScript-file location: 'graphics\$js\project\hud.js'$
[3]CSS location: 'graphics\$css\'$
[4]External CSS location: 'graphics\$assets\extern_css\'$

which is a great part of webgl programming, thus vertex- and fragment shaders is a natural choice for creating such a drawing program to render in a browser. The vertex shader is responsible for manipulating the attributes of the vertices. These can be operations such as transforming the vertex positions, generating texture coordinates and lighting the vertex. The result of these is a set of pixels, called a fragment, which the fragment shader can work on. The fragment shader knows the location of a fragment and can thus handle depths, interpolation of colors etc, between fragments.

WEBGL As mentioned, this project uses the WebGL library which compiles the shader instructions to GPU code. The WebGL is defined in the javascript file and the methods for this library will be elaborated in the next section.

3 Implementation

3.0.1 HTML

The vertex- and fragment shader is implemented in the HTML-file. It is possible to define this elsewhere, for instance directly in the javascript file. However then it needs to be treated as a string, thus making it a bit tedious to edit. The shader recieves data from WebGL in the javascript file through the attribute variables *vPosition*, which is the position of the vertices and *vColor*, which is the color of the vertices. Through the varying-variable, the color vertices are shared to the fragment shader.

```
<script id="vertex-shader" type="x-shader/x-vertex">
  attribute vec2 vPosition;
  attribute vec3 vColor;
  varying vec4 color;
  void main(){
    gl_PointSize = 5.0;
    gl_Position = vec4(vPosition, 0.0, 1.0);
    color = vec4(vColor, 1.0); }
</script><script id="fragment-shader" type="x-shader/x-fragment">
  precision mediump float;
  varying vec4 color;
  void main(){
    gl_FragColor = color;
  }
 ...
```

The two canvases, *webgl* and *hud* is defined within a table along with a menu for interactive functionalities such as buttons and selectors. In order to give the impression of a single drawing program, the two canvases are placed on top of each other. This is done by setting the position of the canvases to an absolute position,

which means the position is fixed. Then we can provide an z-index to determine the order of the canvases. I have chosen to render the webgl canvas on top of the hud. This is possible not a good way of doing it, my thoughts why will be explained in section[6]. The body of the HTML-document uses a onload-function that calls the main-function of the javascript file.

```
...
<body onload="main()">
  <table>
    <tbody>
      <tr>
        <td>
          <canvas id="webgl" width="400" height="400"
          style="position: absolute; z-index: 1;
          border:2px solid #566963;">canvas error</canvas>

          <canvas id="hud" width="400" height="400"
          style="position: absolute; z-index: 0">canvas error</canvas>
  ...
```

3.0.2 JavaScript

The main-function is the first function to be executed. The first thing it does is to call *getDocumentElements()* (see A.0.1, line 121 and 68), which is responsible for making a connection and adding an event listener to the elements of the HTML-page, such as buttons, selectors.

To be able to differentiate between the two canvases we define a variable, *gl*, that get the rendering context for the WebGL canvas, and a variable, *ctx*, that gets the rendering context for the *hud*-canvas.

```
...
  gl = getWebGLContext(canvas);
  var ctx = hud.getContext('2d');
...
```

After initialising the shaders and setting up buffers for vertices and colors (see A.0.1, line 142-164), an event listener for mouse-input is setup. Everytime a mouse-button is clicked, it is checked that the click is happening within the canvas frame, if so, we check which drawing mode is currently active, and call the responsible function of that mode:

```
...
    // Switcing between drawing modes:
    switch (mode) {
    case 1:
        placeSinglePoints(cBuffer, vColor, vBuffer, vPosition);
        break;
    case 2:
        placeTriangle(cBuffer, vColor, vBuffer, vPosition);
```

```
9           break;
10      case 3:
11          placeCircle(cBuffer, vColor, vBuffer, vPosition);
12          break;
13      case 4:
14          placeCurve(cBuffer, vColor, vBuffer, vPosition, ctx);
15          break;
16  ...
```

The script maintains a points-, triangle- and circle-array, defined globally. These contains the index the respective positions that the render-function will draw ultimo.

placeSinglePoints-function simply adds single points to its array. *placeTriangle*-function adds to the point-array on the first and second point, with the third click of the mouse, it removes the two points and adds the index position to the triangle array. **placeCircle**-function. The same logic as in the triangle function. However, here we only place one initial point, the second click with the mouse replaces the point and add the index to the circle array. The variables *first*, *second*, *third*, is responsible for keeping track of when you are on first, second or third mouse click.

placeCurve-function is for drawing the bezier curves. I mentioned earlier that the bezier curve is rendered on the HUD-canvas, which is true. However, before rendering the curve, the function places a single point for the starting of the curve and a single point for the control point of the curve. This is done in the webgl-canvas for visual purposes, such that the user can see where he clicks. When the third mouse-click is placed, these points are removed again, and the *placeBezierCurve(ctx, startPoint, controlPoint, endPoint)*-function is called with the coordinates of the three chosen points.

placeBezierCurve(ctx, startPoint, controlPoint, endPoint-function is responsible for drawing the curve to the HUD-canvas. The only problem is that the points giving as arguments to the function is in a coordinate system ranging from -1,1 top left, to 1,-1 bottom right, and needs to be converted to the canvas height and width (400px, 400px). This is illustrated in picture[1.1]

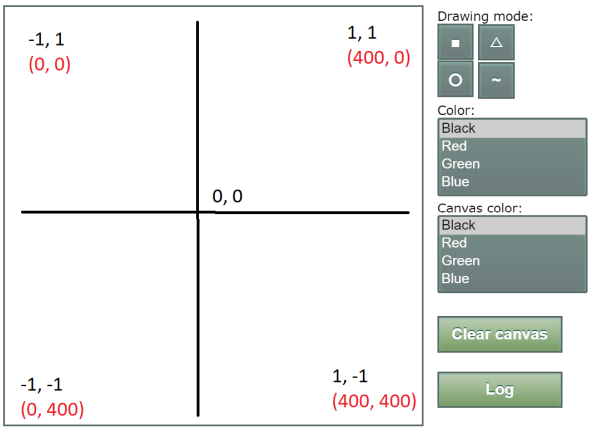

Figure 3.1: Red: Canvas 400x400 px. Black: Space of mouse-clicks

To address this we simply convert the coordinates to match the canvas size:

```
...
  // convert startpoint
  var startPointX = startPoint[0] * 200 + 200;
  var startPointY = -(startPoint[1] * 200 - 200);

  // convert controlpoint
  var controlPointX = controlPoint[0] * 200 + 200;
  var controlPointY = -(controlPoint[1] * 200 - 200);

  // convert endpoint
  var endPointX = endPoint[0] * 200 + 200;
  var endPointY = -(endPoint[1] * 200 - 200);
...
```

and pass along the new coordinates to the context, which provides the function to create quadratic curves:

```
...
  ctx.beginPath();
  ctx.moveTo(startPointX, startPointY);
  ctx.quadraticCurveTo(controlPointX, controlPointY, endPointX,
    endPointY);
  ctx.strokeStyle = color;
  ctx.stroke();
```

4 Result

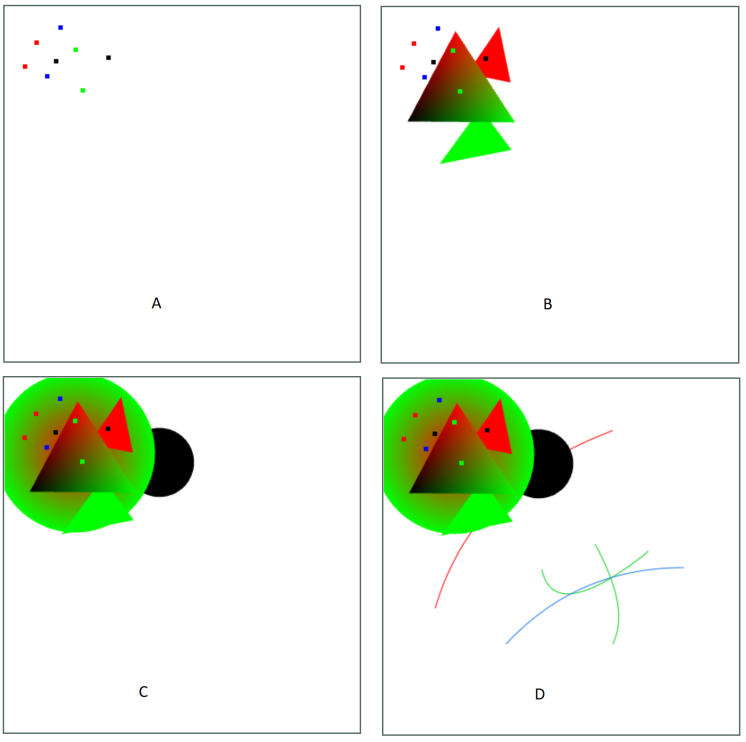

Figure 4.1: Test of the different drawing modes

Late I realised some minor issues with my drawing program. As observed in picture[4.1] the program creates single points, triangles, single colored and interpolated, circles, single colored and interpolated and lastly curves. However, as seen in (A) the single points are added first. In (B) the triangles, and these are rendered behind the dots. This is the case for all modes. All new addition to the canvas is drawn behind the old ones. This can easily be fixed by fixing the rendering order. There is however another issue. The bezier curves are rendered on the HUD-canvas, and still rendered behind all other objects. This can't be fixed be changing the rendering order of the webgl-canvas alone. The problem is that the webgl-canvas is placed on top of the hud-canvas, and thus will always be behind anything on the webgl-canvas. My ideas of fixing this is explained in section[6]

5 Discussion

All in all a drawing program using WebGL for rendering to web-browsers is implemented and working. The program can create different colored canvases, place single dots in different colors. Place triangles, colored and interpolated. Place circles, colored

and interpolated. And lastly place bezier curves by placing a starting, control and endpoint.

As mentioned, I lately realised a few issues. One that is easy to fix, the ordering of rendering in the webgl-canvas. For the issue on curves always being rendered behind the webgl-canvas, my idea is to reverse the z-index of the two canvases; webgl and hud. Then listen for mouse clicks on the hud-canvas and pass these events to the respective functions of the webgl-canvas. Then curves will always be drawn on top.

For possible future work, I could extend on the function for drawing bezier curves, to instead of taking only tree coordinates, making the function take an arbitrary amount of control points. Also, generally, add an option for choosing the thickness of dots and the curves.

6 Link to online project

The project is attached as a ZIP-file to DTU learn, where you can open index.html at the root folder. This will open the webpage that presents all of the worksheets and the project.

You will also be able to find the project at my student page at DTU:

`https://www.student.dtu.dk/~s202790/graphics/`

A Appendix

A.0.1 Code

Project.html

```
<!DOCTYPE html>

<script id="vertex-shader" type="x-shader/x-vertex">
  attribute vec2 vPosition;
  attribute vec3 vColor;
  varying vec4 color;

  void main(){
    gl_PointSize = 5.0;
    gl_Position = vec4(vPosition, 0.0, 1.0);
    color = vec4(vColor, 1.0);
  }
</script>
<script id="fragment-shader" type="x-shader/x-fragment">
  precision mediump float;
  varying vec4 color;

  void main(){
    gl_FragColor = color;
```

```
  }
</script>


<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>HUD</title>
     <link rel="stylesheet" href="../../css/style.css">
  </head>

  <body onload="main()">

  <table>
    <tbody>
      <tr>
        <td>
          <canvas id="webgl" width="400" height="400"
          style="position: absolute; z-index: 1; border:2px solid
    #566963;">
          canvas error</canvas>
          <canvas id="hud" width="400" height="400"
          style="position: absolute; z-index: 0">canvas error</canvas>
        </td>
        <td>
          <table style="position: absolute; left: 425px; z-index: 2">
            <tbody>
              <tr>
                <td class="menu-project">
                  Drawing mode:
                    <br>
                  <!--  <button id="addPoints">Points</button> -->
                    <input class="myButton" type="submit"
                    value="   " id="addPoints">
                    <input class="myButton" type="submit"
                    value="   " id="addTriangles"><br>
                    <input class="myButton" type="submit"
                    value="   " id="addCircles">
                    <input class="myButton" type="submit"
                    value="~" id="addCurves">
                </td>
              </tr>
              <tr>
                <td class="menu-project">
                Color:
                  <br>
                  <select id="colorMenu" size="4" class="select-style">
                  <option value="0" selected>Black</option>
                  <option value="1">Red</option>
                  <option value="2">Green</option>
                  <option value="3">Blue</option>
                  </select>
```

```
                </td>
              </tr>
              <tr>
                <td class="menu-project">
                  Canvas color:
                    <br>
                  <select id="clearMenu" size="4" class="select-style">
                    <option value="0" selected>Black</option>
                    <option value="1">Red</option>
                    <option value="2">Green</option>
                    <option value="3">Blue</option>
                  </select>
                </td>
              </tr>
              <tr>
                <td>
                  <br>
                  <input class="myButton-clear-canvas" type="submit"
                  value="Clear canvas" id="clearButton">
                  <br><br>
                  <input class="myButton-clear-canvas" type="submit"
                  value="Log" id="logButton">
                </td>
              </tr>
            </tbody>
          </table>
        </td>
      </tr>
    </tbody>
  </table>

    <script src="lib/webgl-utils.js"></script>
    <script src="lib/webgl-debug.js"></script>
    <script src="lib/cuon-utils.js"></script>
    <script src="lib/cuon-matrix.js"></script>
    <script type="text/javascript" src="../../common/webgl-utils.js">
    </script>
    <script type="text/javascript" src="../../common/MV.js">
    </script>
    <script type="text/javascript" src="../../common/initShaders.js"
    ></script>
    <script src="hud.js"></script>

  </body>
</html>
```

HUD.js

```
var gl;
var vertices = [];
var points = [];

var canvas;
```

```
var hud;
var clearMenu;
var clearButton;
var addPoints;
var addTriangles;
var addCircles;
var debugButton;

var mode = 1;

//var mode = true;
var first = true;
var second = false;
var third = false;

var max_triangles = 100000;
var max_verts = 3 * max_triangles;
var index = 0;

var t1 = [];
var t2 = [];
var t3 = [];
var t4 = [];
var t = [];

var points = [];
var triangles = [];
var circles = [];
var colors = [];

var startPoint = [];
var controlPoint = [];
var endPoint = [];

var baseColors = [
      vec3(0.0, 0.0, 0.0),  // black
      vec3(1.0, 0.0, 0.0),  // red
      vec3(0.0, 1.0, 0.0),  // green
      vec3(0.0, 0.0, 1.0),  // blue
    ];

function colorConverter(colorCode){

  switch (colorCode) {
  case 0:
      return '#000000';
      break;
  case 1:
      return '#FF0000';
      break;
  case 2:
      return '#0CCF20';
```

```
58          break;
59      case 3:
60          return '#1D7CE6';
61          break;
62      }
63
64  }
65
66
67
68  function getDocumentElements(){
69    // Retrieve <canvas> element
70    canvas = document.getElementById('webgl');
71    hud = document.getElementById('hud');
72
73    // get HTML elements
74    clearMenu = document.getElementById("clearMenu");
75    clearButton = document.getElementById("clearButton");
76    addPoints = document.getElementById("addPoints");
77    addTriangles = document.getElementById("addTriangles");
78    addCircles = document.getElementById("addCircles");
79
80    // for printing debuggin-information in browser
81    debugButton = document.getElementById("logButton");
82
83    // triangle-button clicked
84    addTriangles.addEventListener("click", function(event){
85      console.log("Triangle button clicked");
86      mode = 2;
87      //mode = false;
88    });
89
90    // points-button clicked
91    addPoints.addEventListener("click", function(event){
92      console.log("Points button clicked");
93      mode = 1;
94    });
95
96    // points-button clicked
97    addCircles.addEventListener("click", function(event){
98      console.log("Circles button clicked");
99      mode = 3;
100   });
101
102   // points-button clicked
103   addCurves.addEventListener("click", function(event){
104     console.log("Curves button clicked");
105     mode = 4;
106   });
107
108   // debug-button clicked
109   debugButton.addEventListener("click", function(event){
```

```
110     console.log("Points: [" + points + "] Size: " + points.length +
111     "\nTriangles: [" + triangles + "] Size: " + triangles.length +
112     "\nCircles: [" + circles + "] Size: " + circles.length);
113   });
114
115
116 }
117
118 function main() {
119
120   // Get HTML document elements and setup event listeners
121   getDocumentElements();
122
123   if (!canvas || !hud) {
124     console.log('Failed to get HTML elements');
125     return false;
126   }
127
128   // Get the rendering context for WebGL
129   gl = getWebGLContext(canvas);
130
131   // Get the rendering context for 2DCG
132   var ctx = hud.getContext('2d');
133
134   if (!gl || !ctx) {
135     console.log('Failed to get rendering context');
136     return;
137   }
138
139   gl.viewport(0,0, canvas.width, canvas.height);
140   gl.enable(gl.DEPTH_TEST);
141
142   program = initShaders(gl, "vertex-shader", "fragment-shader");
143   gl.useProgram(program);
144
145   // color buffer setup
146   var cBuffer = gl.createBuffer();
147   gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
148   gl.bufferData( gl.ARRAY_BUFFER, sizeof['vec3']*
149     max_verts, gl.STATIC_DRAW );
150
151   // vertex color setup
152   var vColor = gl.getAttribLocation( program, "vColor" );
153   gl.vertexAttribPointer( vColor, 3, gl.FLOAT, false, 0, 0 );
154   gl.enableVertexAttribArray( vColor );
155
156   // vertex buffer setup
157   var vBuffer = gl.createBuffer();
158   gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
159   gl.bufferData( gl.ARRAY_BUFFER, max_verts, gl.STATIC_DRAW );
160
161   // vertex position setup
```

```
162  var vPosition = gl.getAttribLocation( program, "vPosition" );
163  gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
164  gl.enableVertexAttribArray( vPosition );
165
166    // clear the canvas
167    clearButton.addEventListener("click", function(event) {
168      var bgcolor = baseColors[clearMenu.selectedIndex];
169      gl.clearColor(bgcolor[0], bgcolor[1], bgcolor[2],
170                    bgcolor[3],bgcolor[4],bgcolor[5]);
171
172      // reset everything
173      first = true;
174      second = false;
175      third = false;
176
177      mode = 1;
178      index = 0;
179
180      t1 = [];
181      t2 = [];
182      t3 = [];
183      t4 = [];
184      t = [];
185
186      points = [];
187      triangles = [];
188      circles = [];
189      colors = [];
190      render();
191    });
192
193    // get mouseclick and draw points/triangles/circles
194    canvas.addEventListener("click", function (ev) {
195    // set boundaries
196    var bbox = ev.target.getBoundingClientRect();
197    mousepos = vec2(2*(ev.clientX - bbox.left)/canvas.width - 1, 2*
198    (canvas.height - ev.clientY + bbox.top - 1)/canvas.height - 1);
199
200    // set boundaries
201    var bbox = ev.target.getBoundingClientRect();
202    mousepos = vec2(2*(ev.clientX - bbox.left)/canvas.width - 1, 2*
203    (canvas.height - ev.clientY + bbox.top - 1)/canvas.height - 1);
204
205    // Switcing between drawing modes:
206    switch (mode) {
207    case 1:
208        placeSinglePoints(cBuffer, vColor, vBuffer, vPosition);
209        break;
210    case 2:
211        placeTriangle(cBuffer, vColor, vBuffer, vPosition);
212        break;
213    case 3:
```

```
214             placeCircle(cBuffer, vColor, vBuffer, vPosition);
215             break;
216         case 4:
217             placeCurve(cBuffer, vColor, vBuffer, vPosition, ctx);
218             break;
219         }
220         });
221         render();
222 }
223
224
225 function placeSinglePoints(cBuffer, vColor, vBuffer, vPosition){
226   t = vec3(baseColors[colorMenu.selectedIndex]);
227   gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
228   gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
229     index, flatten(t));
230   gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
231
232   points.push(index);
233   t1 = mousepos;
234   gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
235     index, flatten(t1));
236   index++;
237 }
238
239 function placeTriangle(cBuffer, vColor, vBuffer, vPosition){
240
241   // Placing first point
242   if(first){
243
244     t = vec3(baseColors[colorMenu.selectedIndex]);
245     gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
246     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
247       index, flatten(t));
248     gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
249
250
251     points.push(index);
252     t1 = vec2(mousepos);
253     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
254       index, flatten(t1));
255     index++;
256
257     first = false;
258     second = true;
259
260     // Placing second point
261   } else if (second){
262
263     colors.push(index);
264     t = vec3(baseColors[colorMenu.selectedIndex]);
265     gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
```

```
266     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
267       index, flatten(t));
268     gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
269
270     points.push(index);
271     t2 = vec2(mousepos);
272     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
273       index, flatten(t2));
274     index++;
275
276     second = false;
277     third = true;
278
279     // Placing third point
280   } else {
281     t = vec3(baseColors[colorMenu.selectedIndex]);
282     gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
283     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
284       index, flatten(t));
285
286     gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
287
288     points.pop();
289     triangles.push(points.pop());
290     t3 = vec2(mousepos);
291
292     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
293       index, flatten(t3));
294     index++;
295
296     first = true;
297     third = false;
298   }
299 }
300
301 function placeCircle(cBuffer, vColor, vBuffer, vPosition){
302
303         // Placing first point
304       if(first){
305
306         // colors
307         t = vec3(baseColors[colorMenu.selectedIndex]);
308         gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
309         gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
310           index, flatten(t));
311         gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
312
313
314         // push first point
315         points.push(index);
316         t1 = vec2(mousepos);
317         gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
```

```
318             index, flatten(t1));
319           index++;
320
321           first = false;
322           second = true;
323
324           // Placing second point
325         } else {
326
327           // colors
328           t = vec3(baseColors[colorMenu.selectedIndex]);
329           gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
330           gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
331             index, flatten(t));
332
333           // vertex
334           gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
335           circles.push(points.pop());
336           t2 = vec2(mousepos);
337
338           // calculate radius from point1 to point2
339           var r = Math.sqrt(Math.pow((t2[0]-t1[0]),2) +
340                   Math.pow((t2[1]-t1[1]),2));
341
342           // make circle
343           for (i = 0; i <= 200; i++){
344
345             t = vec3(baseColors[colorMenu.selectedIndex]);
346             gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
347             gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
348               index, flatten(t));
349
350             gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
351             t2 = vec2(t1[0] + r*Math.cos(i*2*Math.PI/200),
352                       t1[1] + r*Math.sin(i*2*Math.PI/200));
353             gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
354               index, flatten(t2));
355             index++;
356           }
357           second = false;
358           first = true;
359         }
360 }
361
362 // place points to calculate the bezier curve
363 function placeCurve(cBuffer, vColor, vBuffer, vPosition, ctx){
364
365   // Placing first point
366   if(first){
367
368     t = vec3(baseColors[colorMenu.selectedIndex]);
369     gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
```

```
370     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
371       index, flatten(t));
372     gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
373
374     // add single start point (for visual experience)
375     points.push(index);
376     t1 = vec2(mousepos);
377     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
378       index, flatten(t1));
379     index++;
380
381     startPoint.push(mousepos[0].toPrecision(1));
382     startPoint.push(mousepos[1].toPrecision(1));
383
384     first = false;
385     second = true;
386
387   // Placing second point
388   } else if (second){
389
390     colors.push(index);
391     t = vec3(baseColors[colorMenu.selectedIndex]);
392     gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
393     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec3'] *
394       index, flatten(t));
395     gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
396
397     // add single control point (for visual experience)
398     points.push(index);
399     t2 = vec2(mousepos);
400     gl.bufferSubData(gl.ARRAY_BUFFER, sizeof['vec2'] *
401       index, flatten(t2));
402     index++;
403
404     controlPoint.push(mousepos[0].toPrecision(1));
405     controlPoint.push(mousepos[1].toPrecision(1));
406
407     second = false;
408     third = true;
409
410   // Placing third point
411   } else {
412
413     // remove start and control point
414     points.pop();
415     points.pop();
416
417     endPoint.push(mousepos[0].toPrecision(1));
418     endPoint.push(mousepos[1].toPrecision(1));
419
420     placeBezierCurve(ctx, startPoint, controlPoint, endPoint);
421
```

```
422       first = true;
423       third = false;
424
425       startPoint = [];
426       controlPoint = [];
427       endPoint = [];
428     }
429 }
430
431 // draw the actual curve to the canvas
432 function placeBezierCurve(ctx, startPoint, controlPoint, endPoint) {
433   //  ctx.clearRect(0, 0, 400, 400);
434
435   // convert startpoint
436   var startPointX = startPoint[0] * 200 + 200;
437   var startPointY = -(startPoint[1] * 200 - 200);
438
439   // convert controlpoint
440   var controlPointX = controlPoint[0] * 200 + 200;
441   var controlPointY = -(controlPoint[1] * 200 - 200);
442
443   // convert endpoint
444   var endPointX = endPoint[0] * 200 + 200;
445   var endPointY = -(endPoint[1] * 200 - 200);
446
447   var color = colorConverter(colorMenu.selectedIndex);
448
449   ctx.beginPath();
450   ctx.moveTo(startPointX, startPointY);
451   ctx.quadraticCurveTo(controlPointX, controlPointY, endPointX,
      endPointY);
452   ctx.strokeStyle = color;
453   ctx.stroke();
454 }
455
456 function render() {
457   gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
458
459   // iterate thru all indexes of points-array and draw each point
460   for(var i = 0; i < points.length; i++){
461     gl.drawArrays(gl.POINTS, points[i], 1);
462   }
463
464   // iterate thru all indexes of triangle-array and draw each triangle
465   for (var i = 0; i < triangles.length; i++){
466     gl.drawArrays(gl.TRIANGLE_FAN, triangles[i], 3);
467   }
468
469   // iterate thru all indexes of triangle-array and draw each triangle
470   for (var i = 0; i < circles.length; i++){
471     gl.drawArrays(gl.TRIANGLE_FAN, circles[i], 202);
472   }
```

```
473   window.requestAnimFrame(render);
474 }
```

References

[1] Bootstrap. “Bootstrap”. In: (). URL: https://getbootstrap.com/.
[2] Kronos Group. “WebGL”. In: (). URL: https://www.khronos.org/webgl/.
[3] Jeff Gyldenbrand. “Worksheet 2. Part 4”. In: (). URL: https://www.student.dtu.dk/~s202790/graphics/js/worksheet2/part4/part4.html.