Technical University of Denmark

02561 Computer Graphics

Extending Drawing Program with Quadratic Bezier Curves & Shader-Computed Circles

Author:
Esben Damkjær Sørensen (s233474)

Portfolio link:
https://cse-cg-worksheets.pages.dev/

December 17th, 2023

Table of Contents

1. Introduction

This report presents a solution to extend part 4 of worksheet 2 to incorporate quadratic Bezier curves into the drawing tool. Additionally, it suggests a different way of rendering circles than originally developed in the worksheet.

The original drawing tool developed as part of the worksheet utilized WebGL to interact with the underlying graphics hardware of the computer and render this onto an HTML canvas. Thus, this project similarly uses WebGL.

The original implementation only performed a single draw call after a user operation. In the worksheet implementation, the user can draw three types of figures; that is, points, triangles, and circles. The circles are n-sided regular polygons, made up of a sufficient number, *n*, of triangles to appear circular. Points consist of two right triangles to form a small square.

The user can draw these, by choosing a drawing mode. E.g., when in the circle drawing mode, and the user places a point of one color, and a second point of another color, the program will create a circle, with its center in the first point, and the periphery through the second point. The color of circle is a linearly interpolated gradient from the center to the periphery of the two points' colors.

In figure 1 is shown an example drawing made with this drawing tool, however besides the circle, it doesn't provide any way to draw custom curves, which this report seeks to provide a solution for. The figure also clearly demonstrates how the circle is made up of an n-gon, of which the sides are easily countable, which this report will look into eliminating.

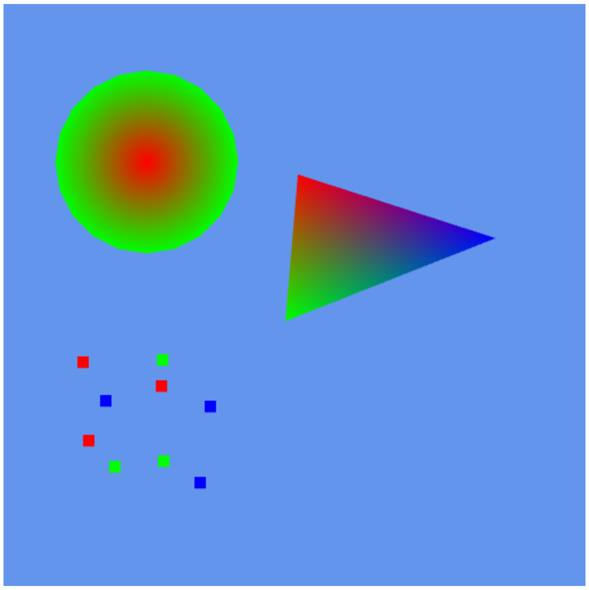

figure 1: Example drawing from the original drawing tool

2. Approach

As this report discusses a solution to extend the drawing program developed in part 4 of worksheet 2, the solution will similarly use standard web technologies to enable execution of the program directly in the browser.

In short, HTML is used to describe the structure of the interface, that the user will interact with to choose what tools to use on the canvas.

Whereas HTML describes the structure of a webpage, JavaScript is a Turing complete programming language, that can be used to implement the functional behavior of the webpage.

As mentioned in the introduction, the original drawing program utilized the WebGL API to interact with the GPU. It does so by providing an API, that wraps the lower-level graphics API, OpenGL, and as such the APIs are very similar. This API enables the developer to write GPU accelerated programs called shaders, which can be used to render an image to a HTML canvas.

To achieve the functionality to draw quadratic Bezier curves in the drawing program, this report proposes a solution, that uses a HUD (Head-Up Display). That is a transparent overlaying canvas on top of the WebGL canvas, that shapes can be drawn onto, which will appear on top of the below canvas. Rendering quadratic curves with a shader is not the most trivial task, and the Canvas API provides a 2D context, which exposes methods to draw various figures including quadratic Bezier curves onto the canvas by only specifying the start, control, and end points of the curve.

To achieve a better approximation of the circle, the new drawing program will try to render the circle from the circle equation seen in equation 1 using a fragment shader.

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

equation 1: circle equation

3. Implementation

To implement the HUD, another canvas was introduced in the HTML-file of the project, as showcased in listing 1. Both canvases were given the absolute positioning property, such that they both are positioned relative to their common parents. By not specifying any more than the z-index, the canvases are positioned on top of each other, ordered by the z-index.

```
<div id="canvasStack" style="width: 512px; height: 512px">
    <canvas id="glCanvas" width="512" height="512"
            style="position: absolute; z-index: 0;"
    ></canvas>
    <canvas id="hudCanvas" width="512" height="512"
            style="position: absolute; z-index: 1"
    >
    </canvas>
</div>
```

listing 1: Stacked canvases

Inside the JavaScript file, is the code that initializes the WebGL context, compiles the shaders etc. Additionally, it also handles the logic that is executed upon events firing such as click events.

3.1. Bezier curves

Inside the event listener, that listens for mouse clicks on the canvas, is an if statement, that handles the logic for each drawing mode. listing 2 shows how the quadratic Bezier is drawn using the Canvas API's 2D context by initializing a path with a given start point. The quadratic curve is then specified with the control and end point.

```
} else if (modeSelect.value === "bezier") {
    if (shapeBuilderPoints.length >= 3) {
        let deleteCount = shapeBuilderPoints.length * pointVertices.length;
        let start = vertices.length - deleteCount;
        vertices.splice(start);
        colors.splice(start);


        let startPoint = toHudCoordinates(shapeBuilderPoints[0], hud);
        let controlPoint = toHudCoordinates(shapeBuilderPoints[1], hud);
        let endPoint = toHudCoordinates(shapeBuilderPoints[2], hud);

        hudCtx.beginPath();
        hudCtx.moveTo(startPoint[0], startPoint[1]);

        hudCtx.quadraticCurveTo(
            controlPoint[0],
            controlPoint[1],
            endPoint[0],
            endPoint[1]
        );

        hudCtx.strokeStyle = rgbToHex(shapeBuilderColors[0]);
        hudCtx.stroke();

        shapeBuilderPoints = [];
        shapeBuilderColors = [];
    }
}
```

listing 2: Drawing mode – Bezier curves

The variable *shapeBuilderPoints* is a list, that keeps track of mouse click points since last shape; thus, the curve is only drawn once the user has clicked three times on the canvas.

However, the points stored in the *shapeBuilderPoints* are specified according to the OpenGL coordinate system, but the 2D context uses screen pixel coordinates. That's why the auxiliary function *toHudCoordinates* is called on the points.

3.2. Shader-computed circles

Just like the Bezier curve, the shader-computed circles were added as another case to the drawing mode if-statement as seen in listing 3. Just like the original circle implementation, this drawing mode assumes the first point is the center of the circle, and the second lies on the periphery of the circle.

The radius is calculated from these two points, and a circle object is pushes to a list of all the circles. When the circle is rendered, it is done using the draw call in triangles mode. For that reason, another list called *circleBounds*, contains all the vertices, that make up the triangle-pairs, that make up the bounding box for every circle. These triangles simply form a square, that has a *2r* side-length.

```
} else if (modeSelect.value === "shadercircle") {
    if (shapeBuilderPoints.length >= 2) {
        let deleteCount = shapeBuilderPoints.length * pointVertices.length;
        let start = vertices.length - deleteCount;
        vertices.splice(start);
        colors.splice(start);

        let p1 = shapeBuilderPoints[0];
        let p2 = shapeBuilderPoints[1];

        let radius = Math.abs(
            length(
                subtract(p1, p2)
            )
        );

        circles.push(
            {
                center: p1,
                radius: radius,
                innerColor: shapeBuilderColors[0],
                outerColor: shapeBuilderColors[1],
            }
        );

        circleBounds.push(
            ...
            translate(
                squareVertices.map(vert => scale(radius, vert)),
        p1
      )
    );

    shapeBuilderPoints = [];
    shapeBuilderColors = [];
  }
}
```

listing 3: Drawing mode - Shader-computed circle

To achieve a higher separation of concerns and better readability, the shaders have been moved into separate files. There are four shaders, two vertex shaders and two fragment shaders. One of the shader pairs is responsible for rendering the triangles of the original implementation. The other shader pair named circle.[frag/vert], is responsible for rendering the circles in the new implementation. As seen in listing 4, the shader has four inputs in the form of uniforms. To render a circle, it uses the circle equation to determine if a fragment is inside or outside the circle. In case it is outside the circle, it discords that fragment. If it is inside the circle, it colors the fragment a mix of the inner and outer colors based on how far the fragment is from the center of the circle.

```
 #version 300 es
precision mediump float;

uniform vec2 center;
uniform float radius;

uniform vec3 innerColor;
uniform vec3 outerColor;

in vec2 fragCoord;
out vec4 fragColor;

void main() {
  float a = (fragCoord.x - center.x) * (fragCoord.x - center.x);
  float b = (fragCoord.y - center.y) * (fragCoord.y - center.y);
  float c = radius * radius;

  float res = a + b - c;

  if (res > 0.0) {
    discard;
  } else {
    vec3 color = mix(innerColor, outerColor, sqrt(a + b) / radius);

    fragColor = vec4(color, 1.0);
  }

}
```

listing 4: Fragment shader for circle rendering

Since uniforms are static across a draw call, all circles must be drawn by separate draw calls. So, in addition to the original implementations single draw-call, there's not a loop that iterates over all circles, and draws them to the screen.

4. Results

The following two sections showcase the results after using newly implemented tools.

4.1. Quadratic Bezier Curves

figure 2 shows a number of quadratic curves drawn with the new HUD tool in various colors specified by the user, as well as a few other figures drawn with the program.

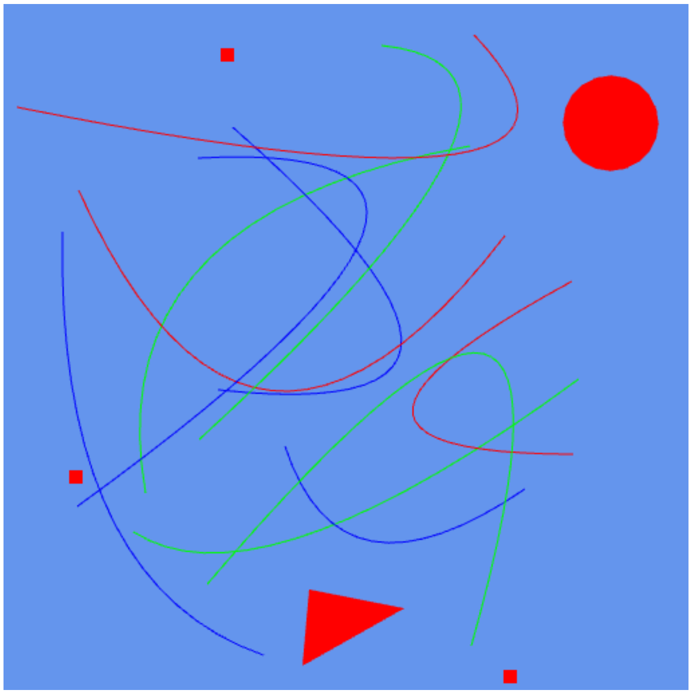

figure 2: Screenshots of the result after drawing a few quadratic Bezier curves

4.2. Shader-computed circles

In figure 3 is an attempt to draw two similar circles, one with the new circle rendering implementation on the left-hand side, and another with the old implementation on the right-hand side, which uses a 10-sided regular polygon.

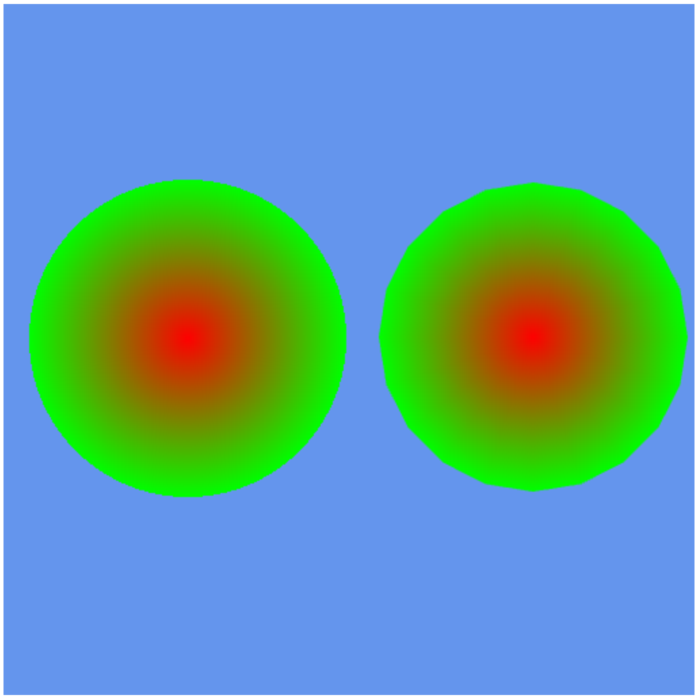

figure 3: Screenshot of the shader-computed circles (lhs) next to the original implementation of circles (rhs).

5. Discussion

One obvious downside of using the head-up display, is that everything drawn on the head-up display, will always appear on top of the canvas below, and this cannot be changed. For that reason, the quadratic Bezier curves will always appear on top of everything else regardless of the drawing order. This is not ideal for a general-purpose drawing program. This could be fixed by either drawing all shapes using the canvas 2D context or writing a custom implementation of the quadratic Bezier curves, that uses the WebGL context.

In general, drawing order wasn't considered in this project, and thus the new implementation of the circle rendering, will also cause all the circles to appear on top of every other shape drawn with the program besides the quadratic curves. This could however be fixed fairly simple, by specifying the z-coordinate of the vertices drawn by the program. By keeping track of the drawing order, the z-coordinate could be set accordingly, but mapped to the interval [-1, 1], as this is within the bounds of the NDC cube, which is rendered to the screen.

In figure 3, it can be seen, that the new implementation of the circle is a closer approximation of an actual circle, only limited by the resolution of the screen. If you look closely, you can see the pixels clearly, as there's no antialiasing on the circle. If anti-aliasing was applied, the circle would appear even smoother to the viewer.

Without having performed any benchmarks, one can imagine, that the new implementation of the circle rendering is less efficient. This is especially due to the square root operation, which is performed for every single fragment inside a circle. However, it is no match for my laptop, in the simple use case of this drawing program.

It is possible to achieve a similar result with the old circle implementation, by simple increasing the number of sides in the regular polygon. However, in case a zoom feature is added, this count would have to be increased, as the user zooms in, or the linear sides will become noticeable again.

6. Conclusion

This project successfully utilized the canvas API's 2D context to implement a Bezier curves head-up display. However, with the downside, they will always appear on top of everything else, which is not ideal for a drawing program.

Additionally, using the new shader-computed circles proved a better approximation of an actual circle for the given zoom level and circle size, when using 10-sided regular polygons. The edge of the circle appears a little coarse due to, and future improvement could be to add anti-aliasing to smoothen out the edge.

The deployed version of the full portfolio can be found at:
https://cse-cg-worksheets.pages.dev/