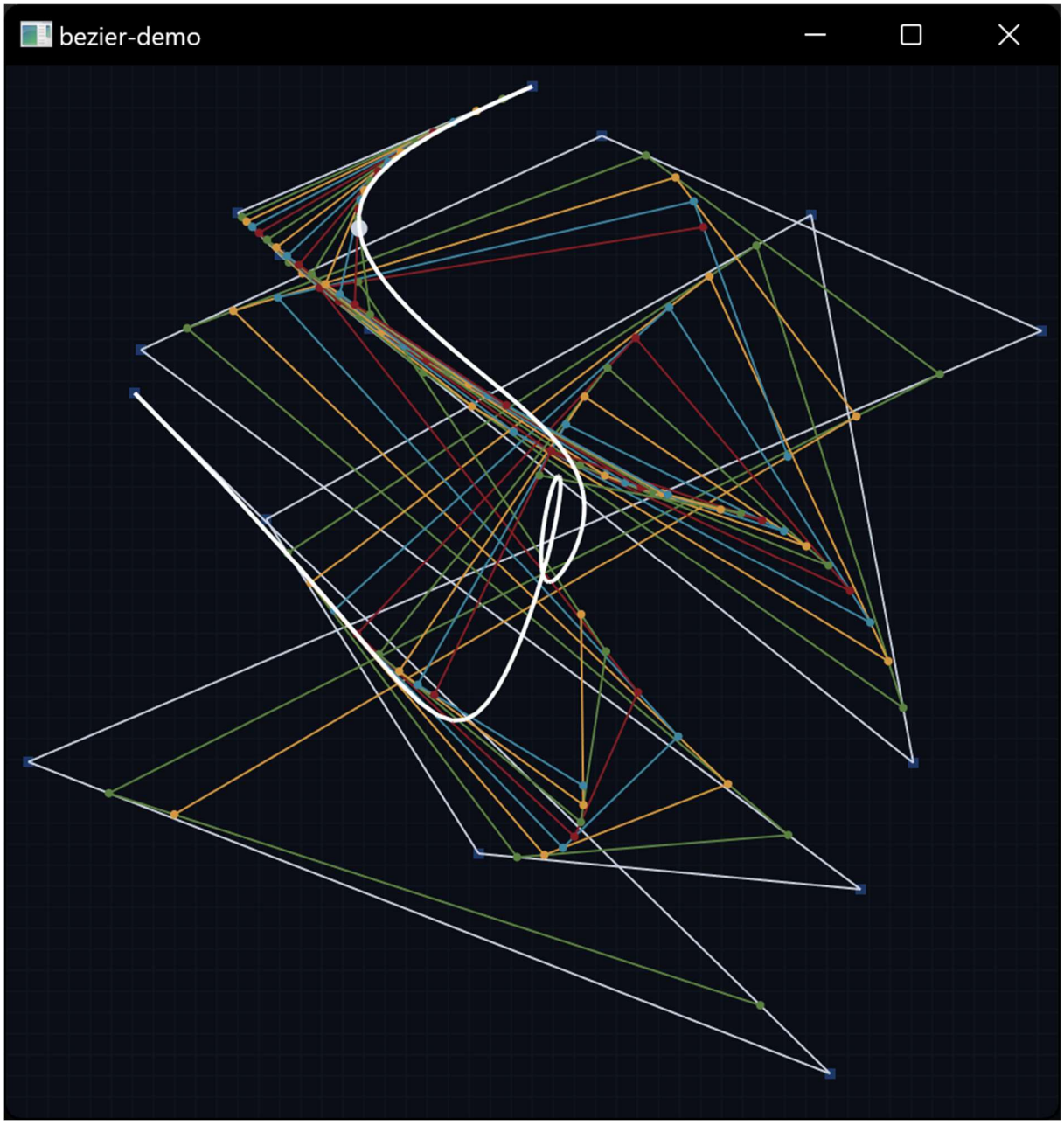
Brodric Foster
April 30th, 2024
Professor Reza Entezari

Graphics Assignment #4 – Curve Design



This project was completed with Rust, using Piston for the graphics. At the heart of the program is the BezierCurve struct, containing only a vector of control points, represented as pairs of floating point numbers. The two approaches, the blending/matrix form and de Casteljau's, are captured in two methods.

Brodric Foster
April 30th, 2024
Professor Reza Entezari

```
    // Return a point on the curve at time t
    // Blending function format.
    // https://en.wikipedia.org/wiki/B%C3%A9zier_curve#Explicit_definition
    pub fn point(&self, t: f64) -> [f64; 2] {
        let n = self.control_points.len() - 1;
        let mut point = [0.0, 0.0];
        for (i, &control_p) in self.control_points.iter().enumerate() {
            // B(t) = sum from i=0 to n of { (n choose i) * (1-t)^(n-i) * t^i * P_i }
            let b = binomial(n, i) as f64
                  * (1.0 - t).powf((n - i) as f64)
                  * t.powf(i as f64);
            point[0] += b * control_p[0];
            point[1] += b * control_p[1];
        }
        point
    }
```

By having an explicit formula for each point at every time value, we're able to get whatever precision is desired. The formula itself is also only computed once, dependent only on the curve's number of control points.

```
pub fn de_casteljaus(&self, t: f64, i: usize) -> Vec<[f64; 2]> {
    // returns the points used in the de Casteljau algorithm at time t and subdiv. i.
    // i = 0 returns the control points.
    // i = 1 returns interpolation between the control points, etc.\
    // i = n-1 returns the final point on the curve.
    // base case: i = 0
    if i == 0 {
        return self.control_points.clone();
    }
   // recursive case: i > 0
   let mut points = vec![];
   let previous = self.de_casteljaus(t, i - 1);
   for segment in previous.windows(2) {
            let p1 = segment[0];
            let p2 = segment[1];
        points.push([(1.0 - t) * p1[0] + t * p2[0], (1.0 - t) * p1[1] + t * p2[1]]);
    }
    points
}
```

Brodric Foster
April 30th, 2024
Professor Reza Entezari

This method, by contrast, is really pretty to look at. It involves only linear interpolations, which is nice for performance, but it doesn't scale well with large orders. Additionally, we must calculate every single midpoint in order to get the final point that's actually on the curve.

Rendering is done utilizing these functions given by BezierCurve. The curve itself is rendered by sampling the point() function at various t-values, then drawing line segments between them.

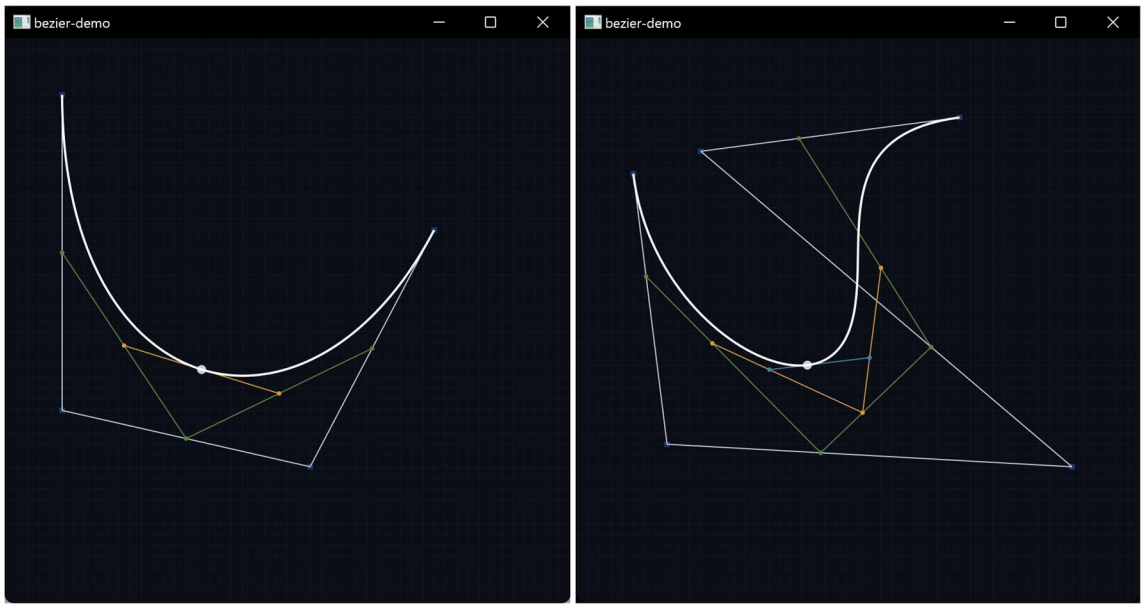
```rust
// Draws the curve itself. Uses the blending function form.
fn render_curve(&self, curve: &BezierCurve, args: &RenderArgs, gl: &mut GlGraphics) {
    use graphics::*;
    const LINE_WIDTH: f64 = 1.0;
    let line_color = [1.0, 1.0, 1.0, 1.0];

    // generate a list of points on the curve
    const NUM_POINTS: usize = 100;
    let mut points = [[0.0; 2]; NUM_POINTS];
    for i in 0..NUM_POINTS {
        let t = i as f64 / (NUM_POINTS - 1) as f64;
        points[i] = curve.point(t);
    }

    // draw the curve
    for segment in points.windows(2) {
        let p1 = segment[0];
        let p2 = segment[1];
        gl.draw(args.viewport(), |c, gl| {
            line(
                line_color,
                LINE_WIDTH,
                [p1[0], p1[1], p2[0], p2[1]],
                c.transform,
                gl,
            );
        });
    }
}
```

Another way of generating the curve involves the de Casteljau's method. In order to visualize it, we only render for a single t-value. However, we recursively display the midpoints and connecting lines of each level of subdivision.

Brodric Foster
April 30th, 2024
Professor Reza Entezari

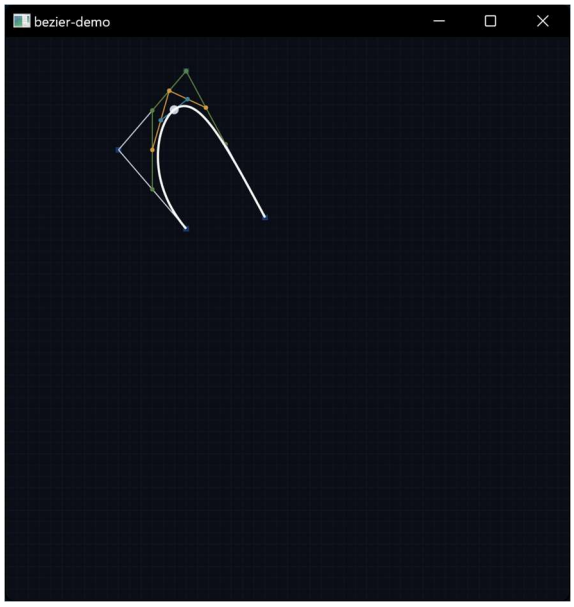
```rust
// This draws everything but the actual curve, including the visualization of the de
Casteljau's algorithm.
fn render_cage(&self, curve: &BezierCurve, args: &RenderArgs, gl: &mut GlGraphics) {
    for subdivision in 0..curve.control_points.len() {
        let points = curve.de_casteljaus(self.time, subdivision);
        let color = self.get_color(subdivision, curve.control_points.len() - 1);
        // special case for control points, draw boxes instead of circles.
        if subdivision == 0 {
            self.draw_control_points(&points, args, gl);
            self.draw_lines(&points, color, args, gl);
            continue;
        }
        // set color, thickness, based on subdivision level
        self.draw_lines(&points, color, args, gl);
        self.draw_points(&points, color, args, gl);
    }
}
```

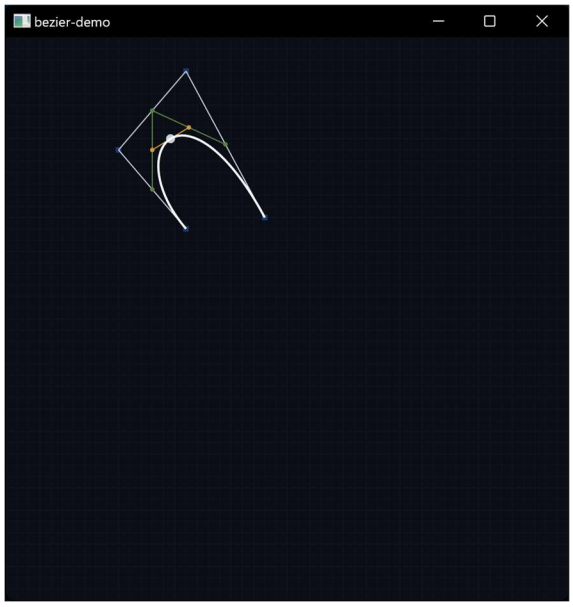
We can view the fruits of our labor below:



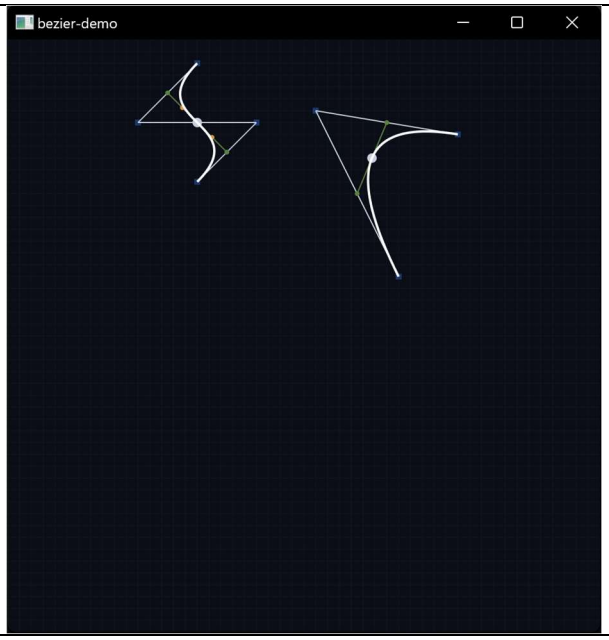
The method I've implemented allows for Bezier curves of any degree. Additionally, repeating consecutive control points results in a lower order curve, but this is not equal to the lower order curve that shares the same control points.

Brodric Foster
April 30th, 2024
Professor Reza Entezari

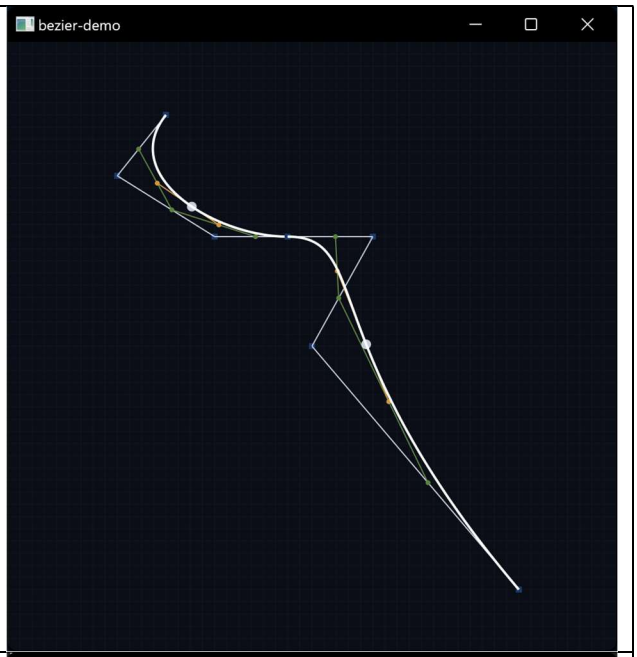




Multiple splines can be added, as well, allowing us to view different types of continuity.

Not continuous	

Brodric Foster
April 30th, 2024
Professor Reza Entezari

G1 Continuous	
C1 continuous