# Chapter 1

Line Rendering

Drawing lines on a computer is an easy problem to solve. However, drawing lines well is significantly more difficult. OpenGL comes with native support for rendering lines, as well as methods to alter the width and color of the drawn line. First, this is not enough control over the line rendering that we will need for this project, since the lines drawn with this method are just simple, solid lines. We would like to have a wide variety of line styles for the user to draw with, for example: lines with feathered edges, variable width, and end caps. The common approach for making production quality lines is to transform the lines into triangles. Using triangles gives a much higher level of control over how the line is drawn because we are not bound to the original definition of the line.
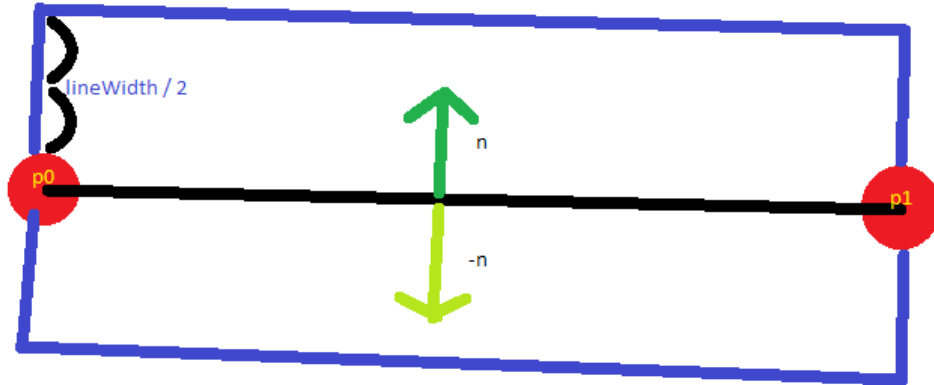
**Figure 1.1:** Diagram for creating the line geometry

1.1 Creating Geometry From the Line Definition

Starting the line is straight forward. Assuming we are given a set of points and a line width, we can start drawing the line by creating a line segment between a set of two points. We then extrude the segment along the positive and negative normal by half of the line width. However, a problem arises when multiple of these segments are formed along the line points. We begin to get gaps along the drawn line. To solve this, we need to create a join between each of the line segments. There are three main join types commonly used to solve this problem: miter, bevel, and round joins.

The round join is formed by rounding out the gap such that a smooth curve is created. It is calculated by forming a circle whose center is the line
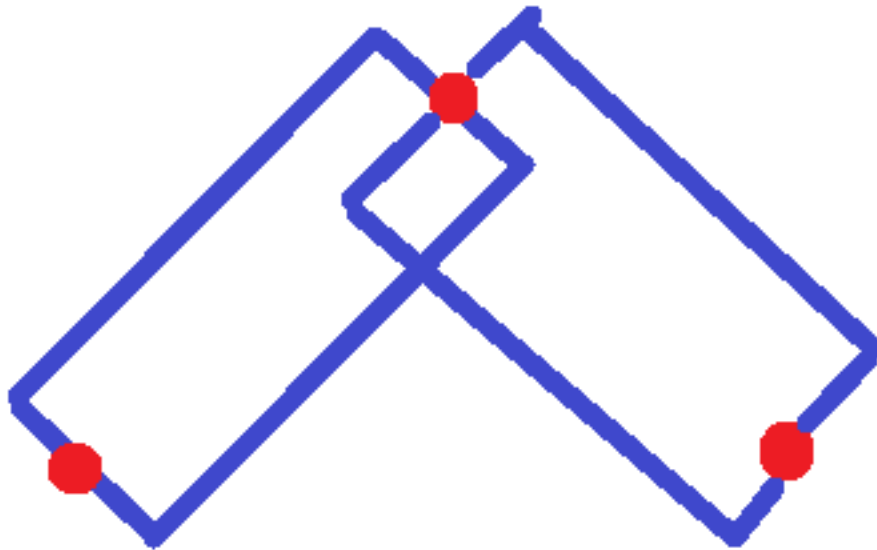
Figure 1.2: The gap that forms when connecting segments
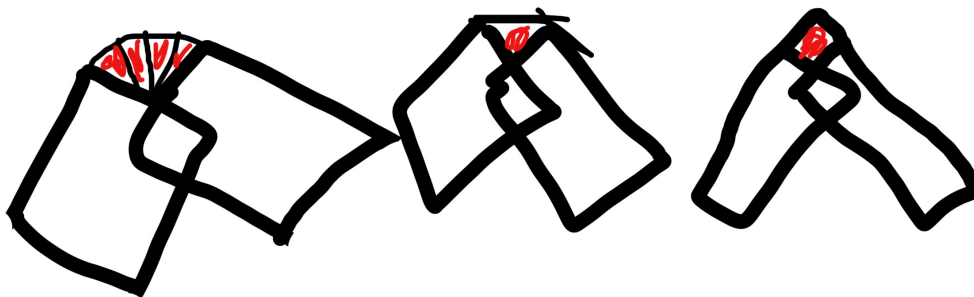
**Figure 1.3:** Types of joins from left to right: round, bevel, miter

Figure 1.4: An example of a miter join artifact from a sharp curve

point at the gap, with a radius of half of the total line width. From there, the part of the circle that fills in the gap is polygonalized. The bevel join is formed by simply filling in the triangular shaped hole with a polygon. The polygon's third edge is the line segment formed from $p_1 a$ and $p_1 b$ (note change names when adding better diagram with labels).

**TODO**: Add derivation and diagrams for miter join.

Miter joins have the particular issue that artifacts can arise from exceptionally sharp points in the line segments. This should be avoided with splines since the spline should subdivide in these cases.
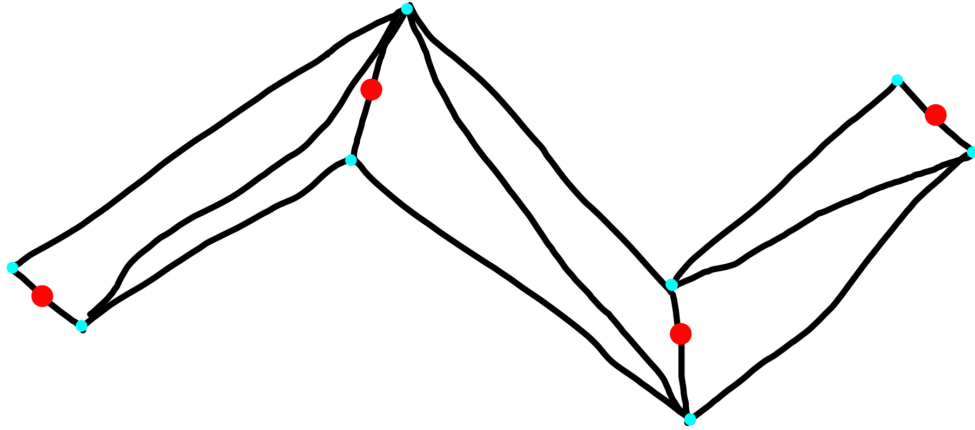
Figure 1.5: Polygonization of a line with miter joins using only two vertices per control point

1.2   Implementation

Actually creating this geometry can get very expensive, especially as more and more lines are drawn. To make this as cheap as possible we take advantage of the properties of both miter joins and our spline storage. The miter join has the advantage of being able to be added by simply translating and fusing vertices in the rectangular geometry. This allows for the least complexity in forming line geometry by simply creating two vertices for each defined line point, which are used to define a triangle strip. Then, using a vertex shader, the vertex positions are transformed appropriately to form the line polygon.
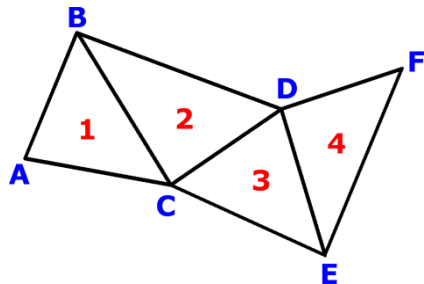
Figure 1.6: A standard triangle strip. Note how the definition matches our polygonalization

The system can now display polygonalized, three dimensional lines and does so efficiently. However, there are two problems with this approach. First, lines only exist in the plane they are drawn in. This means if the camera were to rotate such that it is perpendicular to the plane the lines were drawn on, the lines would disappear from our rendering. This happens because we currently expand the lines in world space, based on the plane the line was drawn on, instead of based on the current position of the camera. Second, this approach assumes we expand our lines on a plane, limiting the sketching surfaces available. Expanding the lines if the sketch was done on a sphere becomes a significantly harder problem. how is line width defined on curved geometry? How do we create them without creating too many polygons and running out of memory? Luckily, we can avoid both of these problems by instead doing a screen space projection of the points that form our lines, and then expand them in the "two-dimensional" screen space. This means that no matter how the line is drawn in the 3-D environment, their definition doesn't change.

TODO: Describe algorithm for projection and add code here

1.3 Various Pen Types