Parametric Curves and Surfaces

In this course we will need a set of techniques to represent curves and surfaces in 2-d and 3-d. Some reasons for this include

- Describing curves in space that objects move along in an animation.
- Describing shapes in 2-d as a collection of curves.
- Describing surfaces in 3-d as a way to construct solid shapes: we break the shapes down into patches and describe each patch as a 2-d surface embedded in a three dimensional space.

The gold standard for constructing curves and surfaces is the parametric representation. For example, in two dimensions the parametric representation for a curve can be written

$$\tilde{p}(t) = \left(p_1(t)\ ,\ p_2(t)\right)$$

Likewise, in three dimensions a surface patch can be expressed as

$$\tilde{p}(s,t) = \left(p_1(s,t)\ ,\ p_2(s,t)\ ,\ p_3(s,t)\right)$$

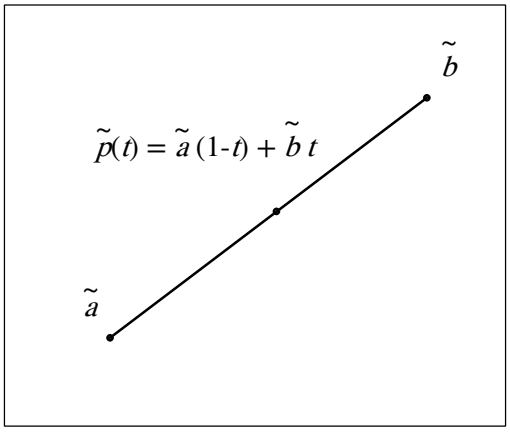
The challenge in each case is constructing the appropriate coordinate functions that will result in the curve or surface having a desired shape.

Bezier Curves

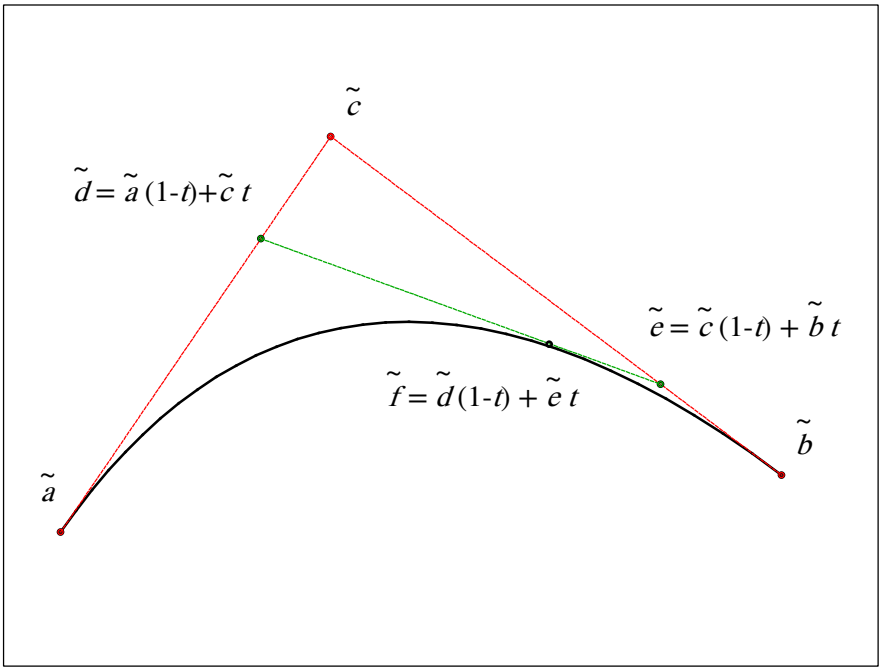
An alternative to constructing an explicit parametric representation is to use a constructive technique.

One such constructive technique is due to Bezier, who developed a method for constructing curves based on iterated linear interpolation.

In a simple linear interpolation we construct a straight line between two end points $\tilde{p}$ and $\tilde{q}$:



In an iterated interpolation we introduce one or more additional intermediate points (commonly referred to as control points) and do iterated interpolation using all of the points. The simplest example involves a single additional control point, $\tilde{c}$:



To start the construction, we first do a pair of linear interpolations from $\tilde{a}$ to $\tilde{c}$ and from $\tilde{c}$ to $\tilde{b}$ using the same factor t in both interpolations. Then, we do a third interpolation between the two interpolated points to construct a final point, $\tilde{f}$. As we vary the interpolation parameter t from 0 to 1, the final interpolated point will trace out a smooth curve from $\tilde{a}$ to $\tilde{b}$.

This construction results in a parameterized curve that takes the form

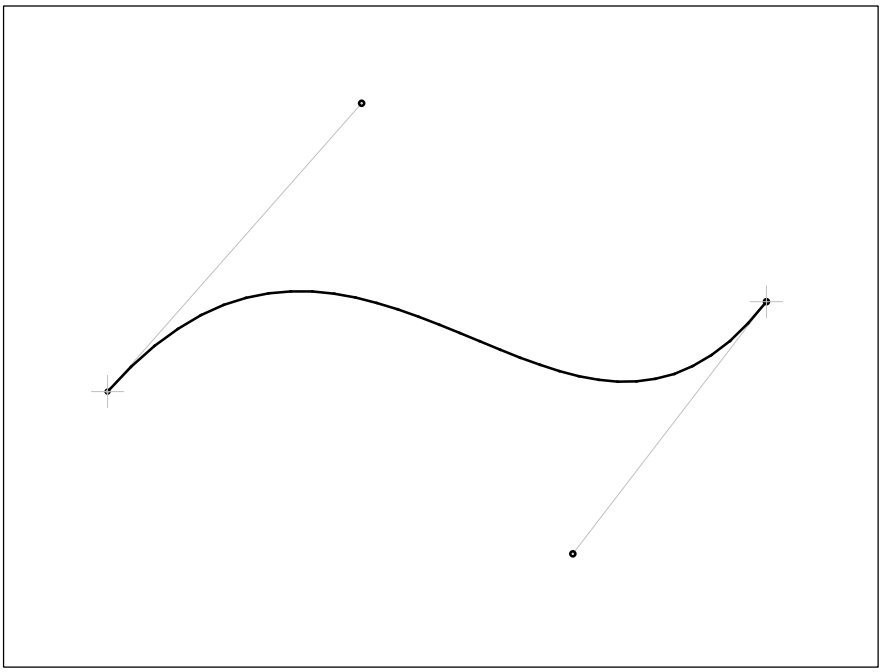
$$\tilde{f}(t) = \left(f_1(t)\ ,\ f_2(t)\right)$$

where $f_1(t)$ and $f_2(t)$ end up being second degree polynomials whose coefficients depend on the coordinates of the initial three points $\tilde{a}$, $\tilde{b}$ and $\tilde{c}$.

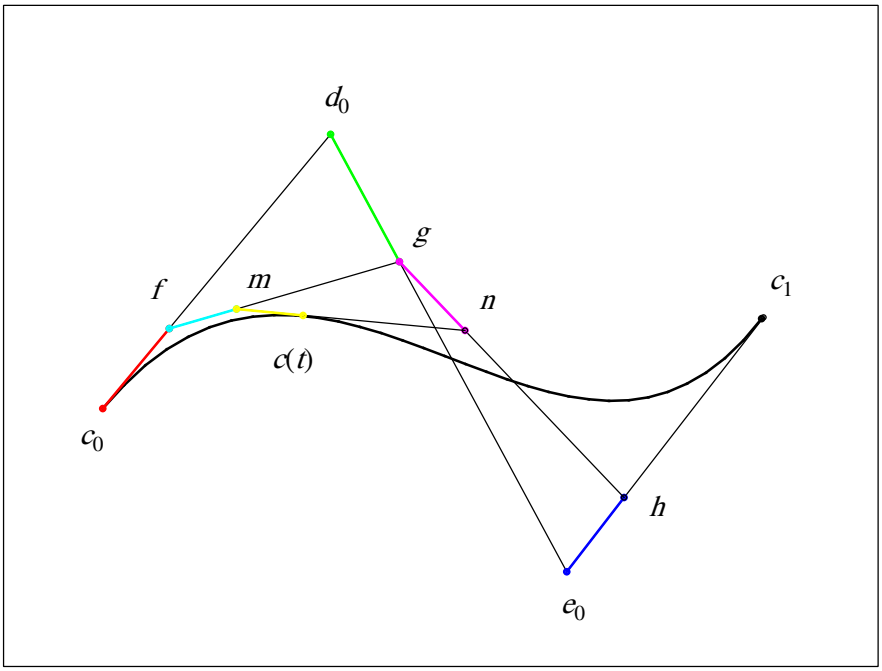
This construction is straightforward and very easy to implement in software, but suffers from a couple of serious problems.

- Although it is clear that moving the control point $\tilde{c}$ around changes the shape of the curve, the relationship between $\tilde{c}$ and the curve shape is not completely intuitive.
- The range of shapes that can be constructed in this way is somewhat limited. This is due to the fact that we are restricting ourselves to using a pair of quadratic polynomials in the parametric representation. Using higher degree polynomials would allow us to approximate a wider range of shapes.

The solution to both of these problems is to use more control points and more rounds of interpolation. The most commonly used version is the cubic Bezier curve, which uses two control points and several more rounds of interpolation.



Here is a diagram that illustrates the details of the iterated interpolation.



$$f = (1 - t)\, c_0 + t\, d_0$$

$$g = (1 - t)\, d_0 + t\, e_0$$

$$h = (1 - t)\, e_0 + t\, c_1$$

$$m = (1 - t)\, f + t\, g$$

$$n = (1 - t)\, g + t\, h$$

$$c(t) = (1 - t)\, m + t\, n$$

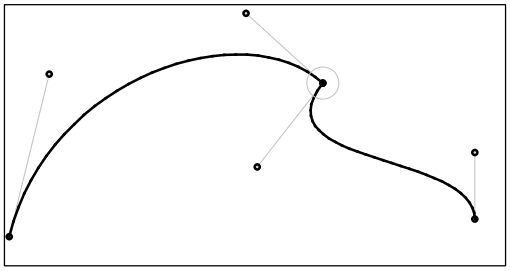
Back-substituting gives

$$c(t) = c_0\,(1 - t)^3 + 3\, d_0\, t\,(1 - t)^2 + 3\, e_0\, t^2\,(1 - t) + c_1\, t^3$$

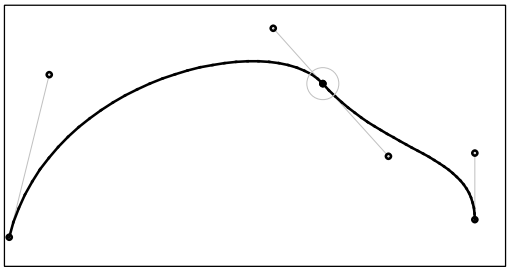
Catmull-Rom Splines

One common technique for constructing more complex curves is to break the curve into segments and then use a Bezier curve to construct each of the segments. One thing we usually try to do when applying this approach is to make sure that the segments join smoothly.

In the first example below I have constructed a pair of Bezier curves that share a common end point. The curves do not join smoothly at the common point.



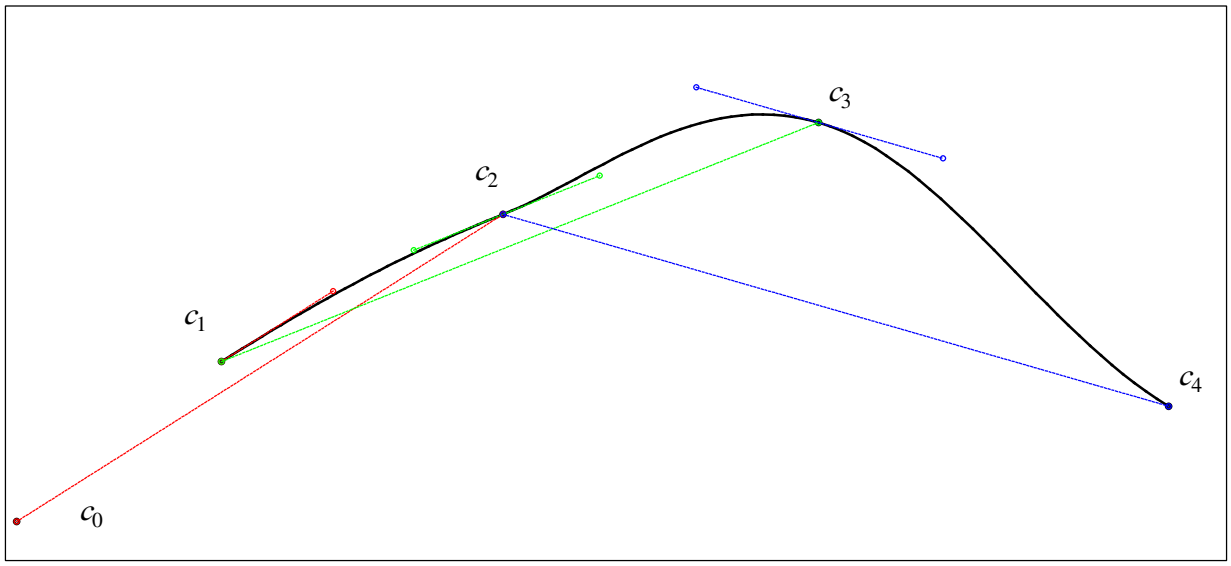
To get the curves to join more smoothly, it is helpful to constrain the two control points on either side of the common point to be colinear.



The Catmull-Rom algorithm couples this idea with a recipe to determine a slope for the colinear control points.

Here are the details.

- We seek to construct a curve that passes through a set of points c_1 to c_n.
- We do this by constructing Bezier curves that join each pair of points in the list.
- Where the Bezier curves join up we place the Bezier control points on either side of the join on a line whose slope is equal to the slope of the line connecting the points on either side of the join.



Here are the mathematical details, showing how to construct the slope of the colinear segment and how to actually compute the locations of the control points to achieve the desired effect.

$$c'(i) = \frac{c_{i+1} - c_{i-1}}{2}$$

$$d_i = \frac{1}{6}\left(c_{i+1} - c_{i-1}\right) + c_i$$

$$e_i = -\frac{1}{6}\left(c_{i+2} - c_i\right) + c_{i+1}$$

Multiple Dimensions

A Bezier curve is a space curve that is constructed by interpolating a set of four control points

$$c(s) = \sum_{n=0}^{3} B_n(s)\,\mathbf{p}_n$$

using the Bernstein polynomials

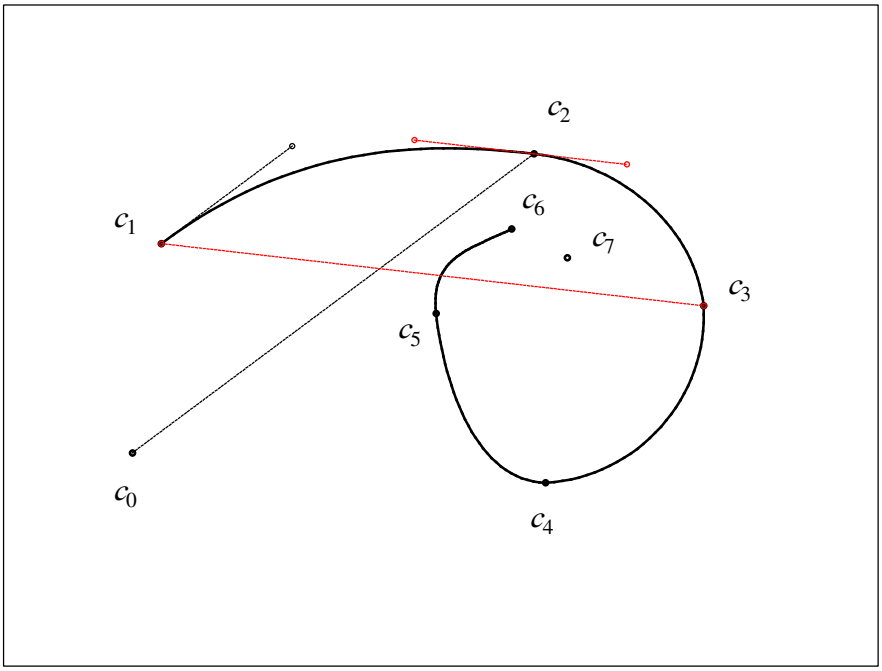
$$B_0(s) = \left(1 - s\right)^3$$

$$B_1(s) = s\left(1 - s\right)^2$$

$$B_2(s) = s^2\left(1 - s\right)$$

$$B_3(s) = s^3$$

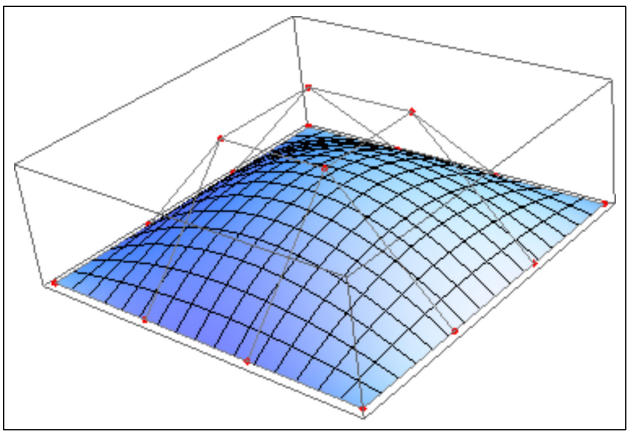
One advantage of this formulation is that it works equally well whether the control points $\mathbf{p}_n$ are points in two dimensions or three dimensions.



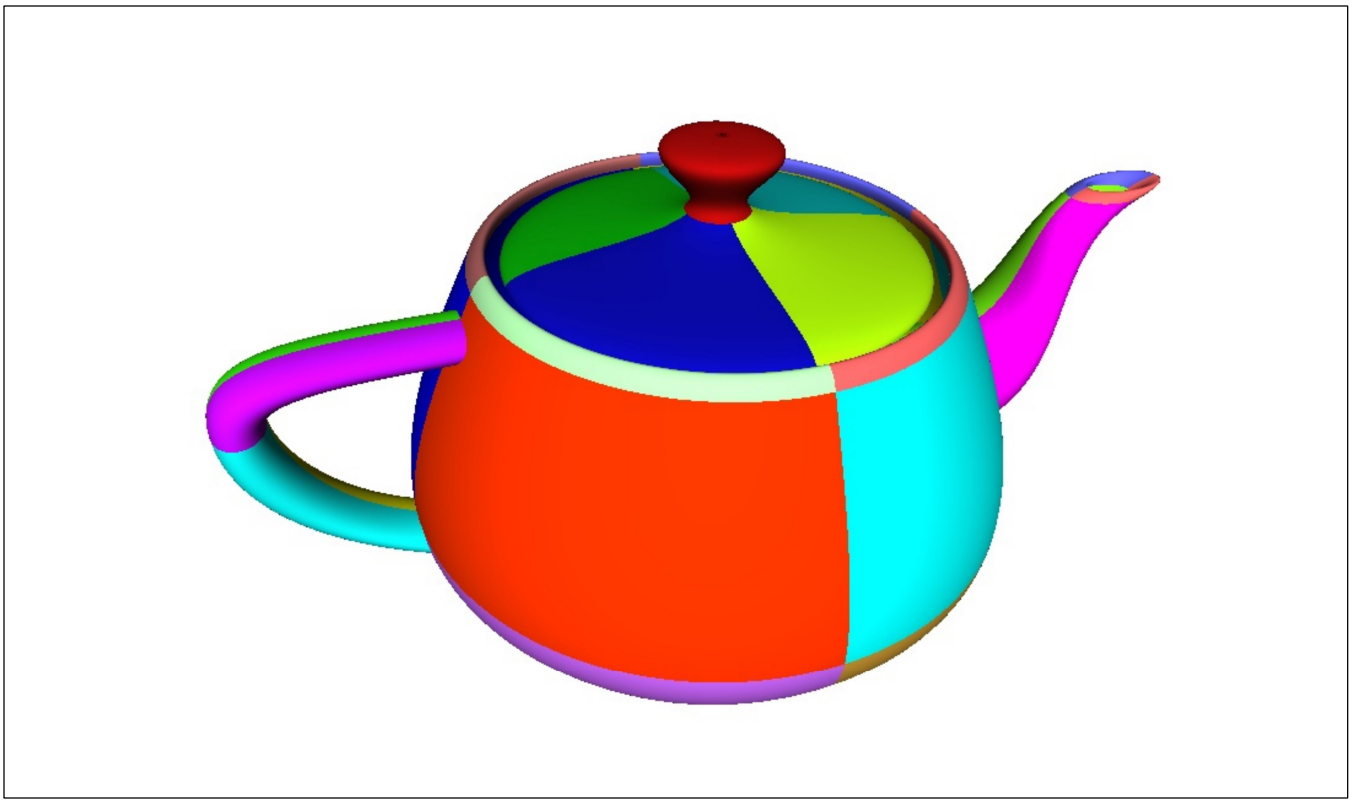
This idea also has a generalization to surfaces. A Bezier *patch* is a surface that is constructed by interpolating a set of 16 control points $\mathbf{p}_{n,m}$ arranged in a grid using two sets of Bernstein polynomials instead of just one:

$$c(s,t) = \sum_{n=0}^{3} \sum_{m=0}^{3} B_n(s)\, B_m(t)\, \mathbf{p}_{n,m}$$

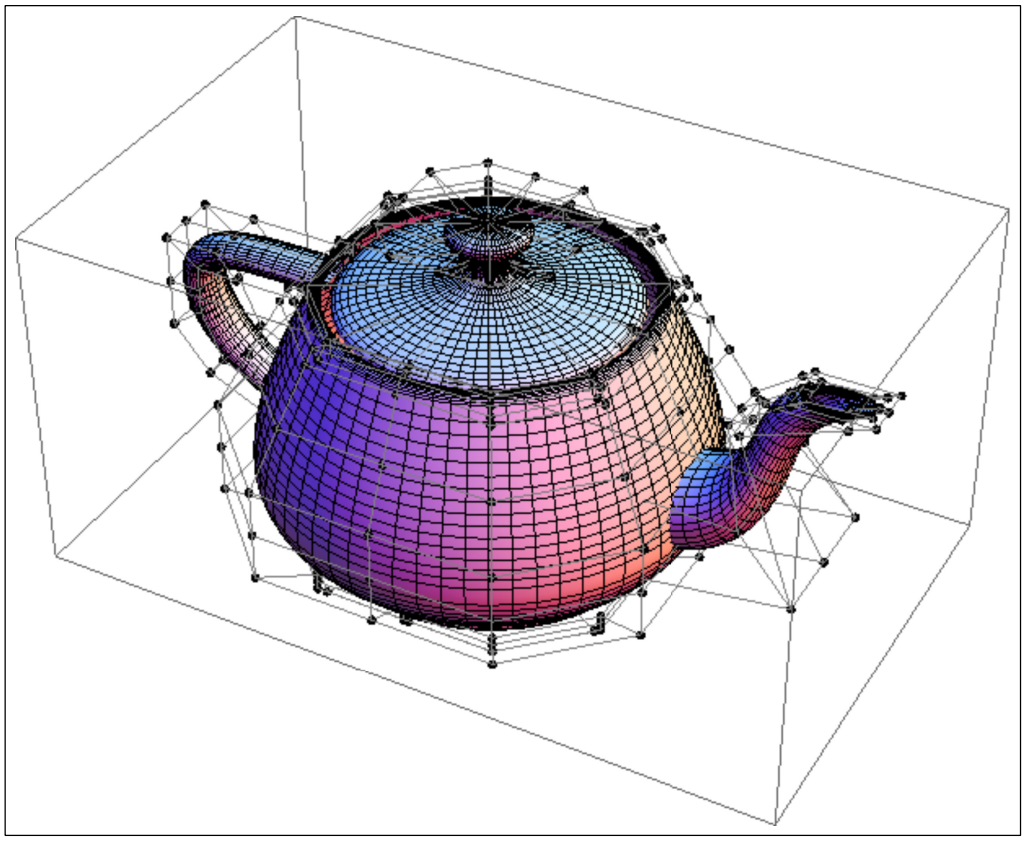
Here is a Bezier patch and its associated grid of 16 control points.



One of the more common techniques used to make sophisticated shapes on computer graphics is to glue together a collection of Bezier patches to make a complex shape. Here is the classic teapot example. The teapot is composed of 32 separate Bezier patches:



Here is the teapot surrounded by its control grid.



Constructing polygons and computing normals

Now that we have a set of mathematical techniques that we can use to describe surfaces in three dimensions we will want to render those surfaces using OpenGL. One method for doing this is to use the mathematical representation of the surface to construct vertices, polygons, and normals.

Constructing the vertices is easy: all we have to do is to sample the surface equation at a grid of points

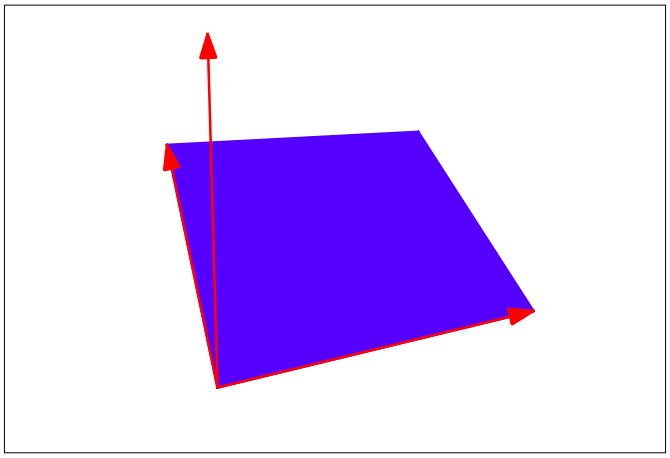
$$(s_i, t_j)$$

in parameter space and then use those to construct vertices:

$$v_{i,j} = \begin{bmatrix} x(s_i, t_j) \\ y(s_i, t_j) \\ z(s_i, t_j) \end{bmatrix}$$

From the vertices we can then go about systematically constructing polygons.

To compute normals we can use one of two techniques. One technique is to construct vectors running from each vertex of the polygon to its two immediate neighbors and then compute the cross product of those vectors:



An alternative technique is *Newell's method*, which computes a normal for the entire polygon by adding contributions from each of the N vertices:

$$n_x = \sum_{i=0}^{N-1} \left(y_i - y_{next(i)}\right)\left(z_i + z_{next(i)}\right)$$

$$n_y = \sum_{i=0}^{N-1} \left(z_i - z_{next(i)}\right)\left(x_i + x_{next(i)}\right)$$

$$n_z = \sum_{i=0}^{N-1} \left(x_i - x_{next(i)}\right)\left(y_i + y_{next(i)}\right)$$

An alternative to constructing the vertices ourselves is to use tessellation shaders to construct our Bezier patches.

We will see this in chapter 12.