```
static inline mat4 ortho(float left,
                         float right,
                         float bottom,
                         float top,
                         float near,
                         float far) { ... }
```

Interpolation, Lines, Curves, and Splines

Interpolation is a term used to describe the process of finding values that lie between a set of known points. Consider the equation of the line passing through points A and B:

$$P = A + t\vec{D}$$

where P is any point on the line and $\vec{D}$ is the vector from A to B:

$$\vec{D} = (B - A)$$

We can therefore write this equation as

$$P = A + t\,(B - A) \quad \text{or}$$
$$P = (1 - t)\,A + tB$$

It is easy to see that when t is 0, P is equal to A; and when t is 1, P is equal to $A + B - A$, which is simply B. Such a line is shown in Figure 4.13.

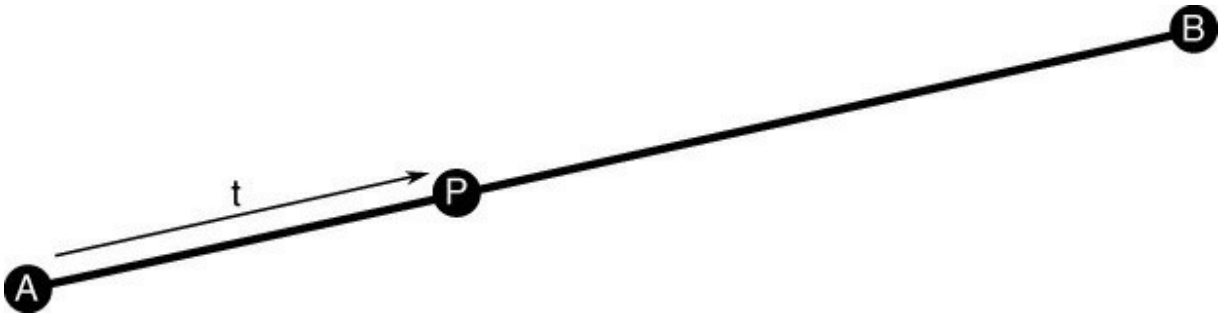

Figure 4.13: Finding a point on a line

If t lies between 0.0 and 1.0, then P will end up somewhere between A and B. Values of t outside this range will push P off the ends of the line. You should be able to see that by smoothly varying t, we can move point P from A to B and back. This is known as *linear interpolation*. The values of A and B (and therefore P) can have any number of dimensions. For example, they could be scalar values; two-dimensional values such as points on a graph; three-dimensional values such as coordinates in 3D space, colors, and so on; or even higher-dimension quantities such as matrices, arrays, or even whole images. In many cases, linear interpolation doesn't make much sense (for example, linearly interpolating between two matrices generally doesn't produce a meaningful result), but angles, positions, and other coordinates can normally be interpolated safely.

Linear interpolation is such a common operation in graphics that GLSL includes a built-in function specifically for this purpose, `mix`:

Click here to view code image

```
vec4 mix(vec4 A, vec4 B, float t);
```

The `mix` function comes in several versions taking different dimensionalities of vectors or scalars as the *A* and *B* inputs and taking scalars or matching vectors for *t*.

Curves

If moving everything along a straight line between two points is all we wanted to do, then this would be enough. However, in the real world, objects move in smooth curves and accelerate and decelerate smoothly. A curve can be represented by three or more *control points*. For most curves, there are more than three control points, two of which form the endpoints; the others define the shape of the curve. Consider the simple curve shown in Figure 4.14.

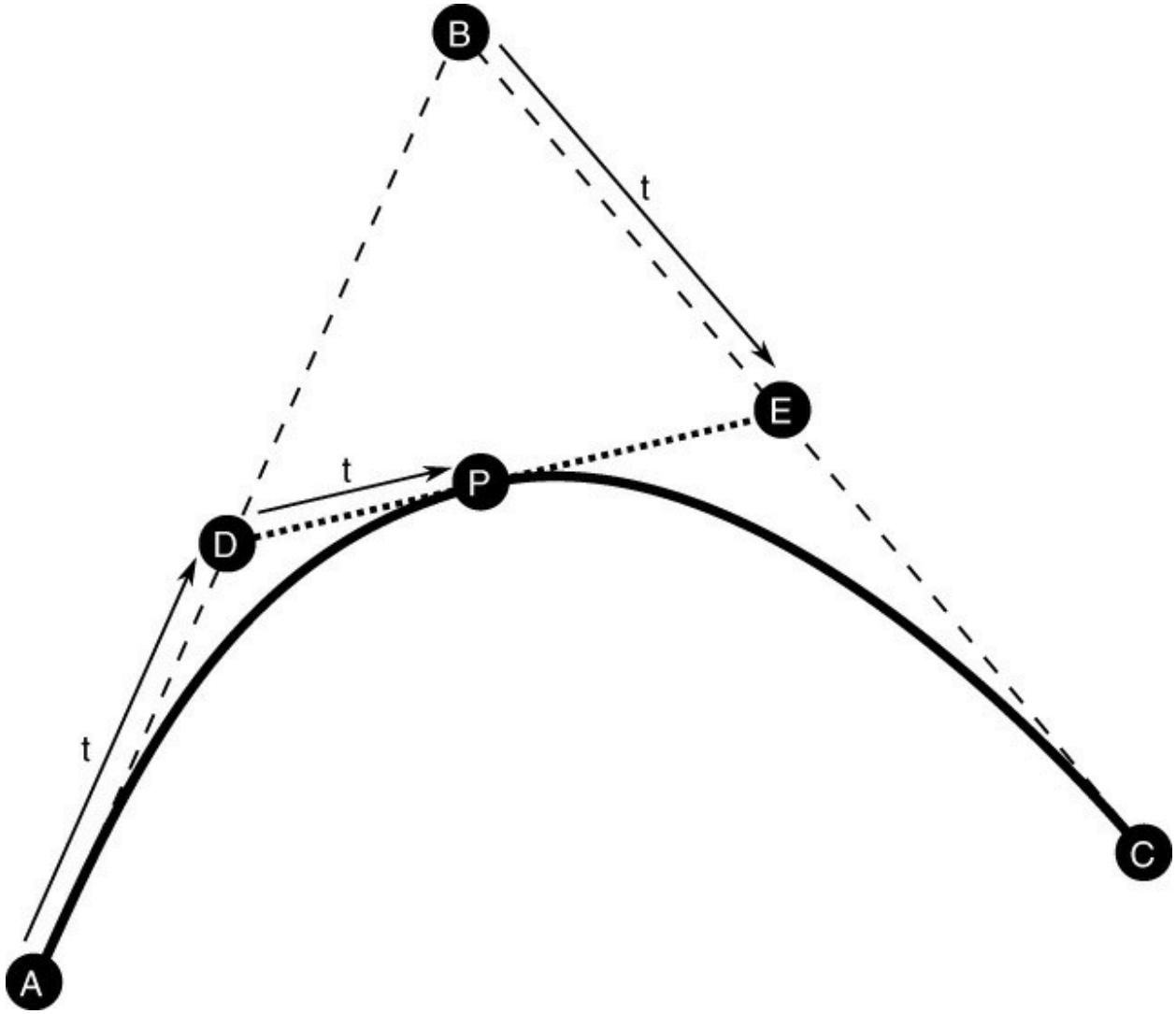

Figure 4.14: A simple Bézier curve

The curve shown in Figure 4.14 has three control points, *A*, *B*, and *C*. *A* and *C* are the

endpoints of the curve and B defines the shape of the curve. If we join points A and B with one line and points B and C together with another line, then we can interpolate along the two lines using a simple linear interpolation to find a new pair of points, D and E. Now, given these two points, we can join them with yet another line and interpolate along it to find a new point, P. As we vary our interpolation parameter, t, point P will move in a smooth curved path from A to D. Expressed mathematically, this is

$$D = A + t(B - A)$$
$$E = B + t(C - B)$$
$$P = D + t(E - D)$$

Substituting for D and E and doing a little crunching, we come up with the following:

$$P = A + t(B - A) + t((B + (t(C - B))) - (A + t(B - A))))$$
$$P = A + t(B - A) + tB + t^2(C - B) - tA - t^2(B - A)$$
$$P = A + t(B - A + B - A) + t^2(C - B - B + A)$$
$$P = A + 2t(B - A) + t^2(C - 2B + A)$$

You should recognize this as a *quadratic* equation in t. The curve that it describes is known as a *quadratic Bézier curve*. We can actually implement this very easily in GLSL using the `mix` function, as all we're doing is linearly interpolating (mixing) the results of two previous interpolations.

Click here to view code image

```
vec4 quadratic_bezier(vec4 A, vec4 B, vec4 C, float t)
{
    vec4 D = mix(A, B, t);       // D = A + t(B - A)
    vec4 E = mix(B, C, t);       // E = B + t(C - B)

    vec4 P = mix(D, E, t);       // P = D + t(E - D)

    return P;
}
```

By adding a fourth control point as shown in Figure 4.15, we can increase the order by 1 and produce a *cubic* Bézier curve.

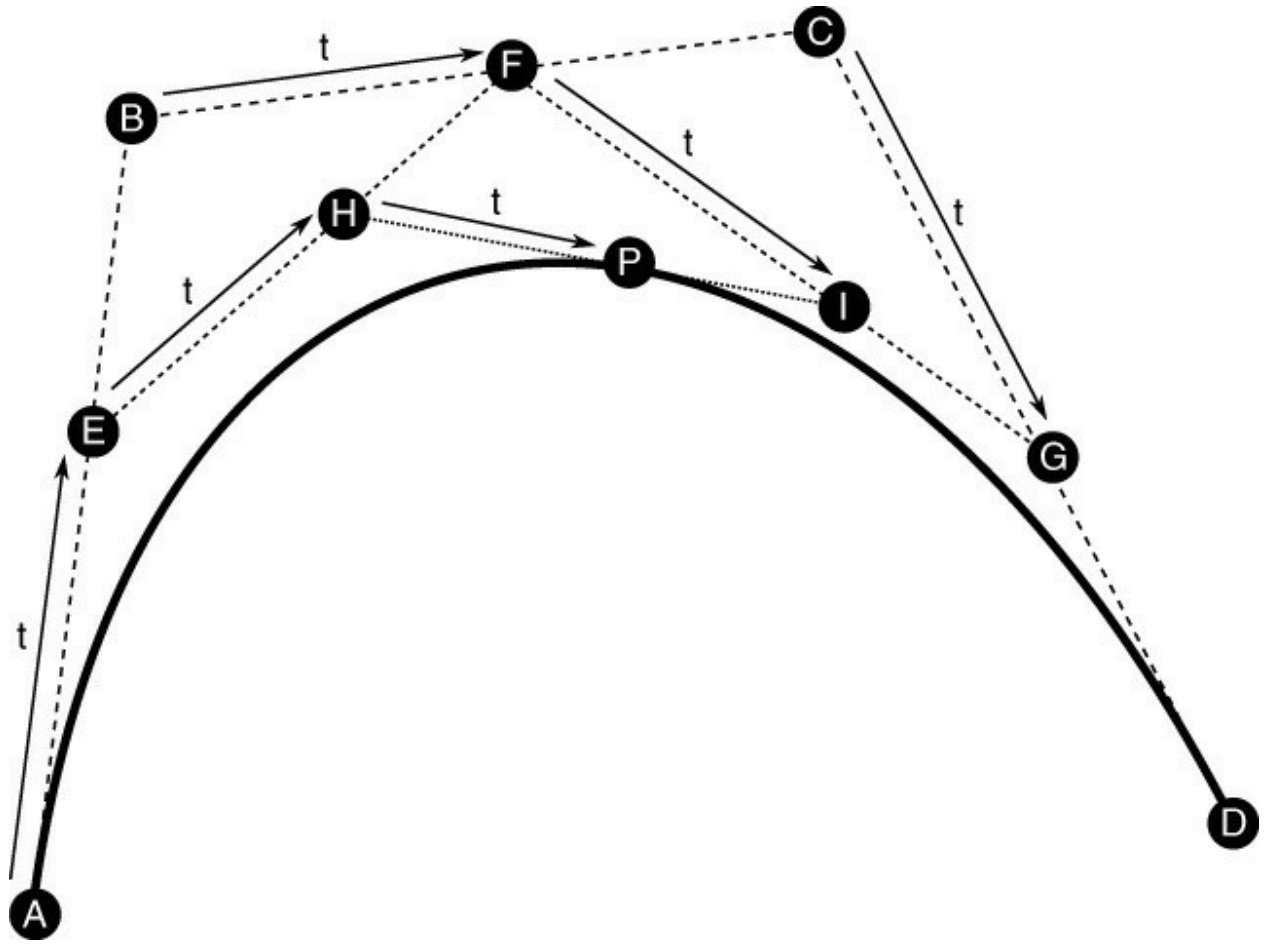

Figure 4.15: A cubic Bézier curve

We now have four control points, *A*, *B*, *C*, and *D*. The process for constructing the curve is similar to that for the quadratic Bézier curve. We form a first line from *A* to *B*, a second line from *B* to *C*, and a third line from *C* to *D*. Interpolating along each of the three lines gives rise to three new points, *E*, *F*, and *G*. Using these three points, we form two more lines, one from *E* to *F* and another from *F* to *G*, interpolating along which gives rise to points *H* and *I*, between which we can interpolate to find our final point, *P*. Therefore, we have

$$
\begin{aligned}
E &= A + t(B - A) \\
F &= B + t(C - B) \\
G &= C + t(D - C) \\
H &= E + t(F - E) \\
I &= F + t(G - F) \\
P &= H + t(I - H)
\end{aligned}
$$

If you think these equations look familiar, you're right: Our points *E*, *F*, and *G* form a quadratic Bézier curve that we use to interpolate to our final point *P*. If we were to substitute the equations for *E*, *F*, and *G* into the equations for *H* and *I*, then substitute

those into the equation for *P*, and crunch through the expansions, we would be left with a cubic equation with terms in t^3—hence the name *cubic Bézier curve*. Again, we can implement this simply and efficiently in terms of linear interpolations in GLSL using the `mix` function:

Click here to view code image

```
vec4 cubic_bezier(vec4 A, vec4 B, vec4 C, vec4 D, float t)
{
    vec4 E = mix(A, B, t);       // E = A + t(B - A)
    vec4 F = mix(B, C, t);       // F = B + t(C - B)
    vec4 G = mix(C, D, t);       // G = C + t(D - C)

    vec4 H = mix(E, F, t);       // H = E + t(F - E)
    vec4 I = mix(F, G, t);       // I = F + t(G - F)

    vec4 P = mix(H, I, t);       // P = H + t(I - H)

    return P;
}
```

Just as the structure of the equations for a cubic Bézier curve "includes" the equations for a quadratic curve, so, too, does the code to implement them. In fact, we can layer these curves on top of each other, using the code for one to build the next.

Click here to view code image

```
vec4 cubic_bezier(vec4 A, vec4 B, vec4 C, vec4 D, float t)
{
    vec4 E = mix(A, B, t);       // E = A + t(B - A)
    vec4 F = mix(B, C, t);       // F = B + t(C - B)
    vec4 G = mix(C, D, t);       // G = C + t(D - C)

    return quadratic_bezier(E, F, G, t);
}
```

Now that we see this pattern, we can take it further and produce even higher-order curves. For example, a *quintic* Bézier curve (one with five control points) can be implemented as

Click here to view code image

```
vec4 quintic_bezier(vec4 A, vec4 B, vec4 C, vec4 D, vec4 E, float t)
{
    vec4 F = mix(A, B, t);       // F = A + t(B - A)
    vec4 G = mix(B, C, t);       // G = B + t(C - B)
    vec4 H = mix(C, D, t);       // H = C + t(D - C)
    vec4 I = mix(D, E, t);       // I = D + t(E - D)

    return cubic_bezier(F, G, H, I, t);
}
```

This layering could theoretically be applied over and over for any number of control points. However, in practice, curves with more than four control points are not commonly used. Rather, we use *splines*.

Splines

A spline is effectively a long curve made up of several smaller curves (such as Béziers) that locally define their shape. At least the control points representing the ends of the curves are shared between segments,[4] and often one or more of the interior control points are either shared or linked in some way between adjacent segments. Any number of curves can be joined together in this way, allowing arbitrarily long paths to be formed. Take a look at the curve shown in Figure 4.16.

4. This is what sticks the curves together to form a spline. These control points are known as *welds* and the control points in between are referred to as *knots*.

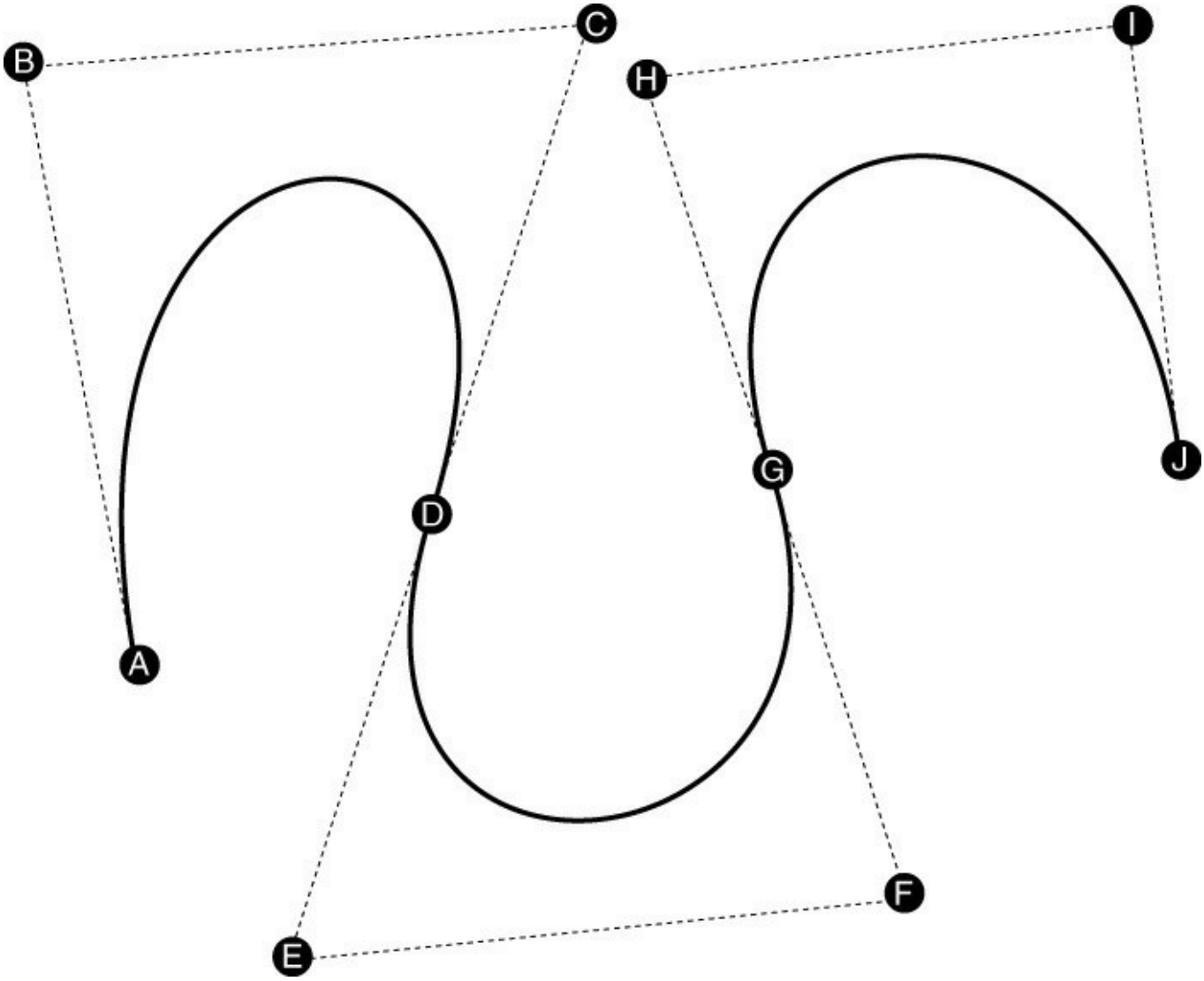

Figure 4.16: A cubic Bézier spline

In Figure 4.16, the curve is defined by ten control points, *A* through *J*, which form three cubic Bézier curves. The first is defined by *A*, *B*, *C*, and *D*, the second shares *D* and further uses *E*, *F*, and *G*, and the third shares *G* and adds *H*, *I*, and *J*. This type of spline is known as a *cubic Bézier spline* because it is constructed from a sequence of cubic Bézier curves. This is also known as a *cubic B-spline*—a term that may be familiar to anyone who has read much about graphics in the past.

To interpolate point *P* along the spline, we simply divide it into three regions, allowing *t* to range from 0.0 to 3.0. Between 0.0 and 1.0, we interpolate along the first curve, moving from *A* to *D*. Between 1.0 and 2.0, we interpolate along the second curve, moving from *D* to *G*. When *t* is between 2.0 and 3.0, we interpolate along the final curve between *G* and *J*. Thus, the integer part of *t* determines the curve segment along which we are interpolating and the fractional part of *t* is used to interpolate along that segment. Of course, we can scale *t* as we wish. For example, if we take a value between 0.0 and 1.0 and multiply it by the number of segments in the curve, we can continue to use our original range of values for *t* regardless of the number of control points in a curve.

The following code will interpolate a vector along a cubic Bézier spline with ten control points (and thus three segments):

Click here to view code image

```
vec4 cubic_bspline_10(vec4 CP[10], float t)
{
    float f = t * 3.0;
    int i = int(floor(f));
    float s = fract(t);

    if (t <= 0.0)
        return CP[0];

    if (t >= 1.0)
        return CP[9];

    vec4 A = CP[i * 3];
    vec4 B = CP[i * 3 + 1];
    vec4 C = CP[i * 3 + 2];
    vec4 D = CP[i * 3 + 3];

    return cubic_bezier(A, B, C, D, s);
}
```

If we use a spline to determine the position or orientation of an object, we will find that we must be very careful about our choice of control point locations to keep motion smooth and fluid. The rate of change in the value of our interpolated point *P* (i.e., its velocity) is the differential of the equation of the curve with respect to *t*. If this function

is discontinuous, then P will suddenly change direction and our objects will appear to jump around. Furthermore, the rate of change of P 's velocity (its acceleration) is the second-order derivative of the spline equation with respect to t. If the acceleration is not smooth, then P will appear to suddenly speed up or slow down.

A function that has a continuous first derivative is known as C^1 continuous; similarly, a curve that has a continuous second derivative is known as C^2 continuous. Bézier curve segments are both C^1 and C^2 continuous, but to ensure that we maintain continuity over the welds of a spline, we need to ensure that each segment starts off where the previous ended in terms of position, direction of movement, and rate of change. A rate of travel in a particular direction is simply a velocity. Thus, rather than assigning arbitrary control points to our spline, we can assign a velocity at each weld. If the same velocity of the curve at each weld is used in the computation of the curve segments on either side of that weld, then we will have a spline function that is both C^1 and C^2 continuous.

This should make sense if you take another look at Figure 4.16—there are no kinks and the curve is nice and smooth through the welds (points D and G). Now look at the control points on either side of the welds. For example, take points C and E, which surround D. C and E form a straight line and D lies right in the middle of it. In fact, we can call the line segment from D to E the velocity at D, or $\vec{V}_D$. Given the position of point D (the weld) and the velocity of the curve $\vec{V}_D$ at D, then C and E can be calculated as

$$C = D - \vec{V}_D$$

$$E = D + \vec{V}_D$$

Likewise, if $\vec{V}_A$ represents the velocity at A, B can be calculated as

$$B = A + \vec{V}_A$$

Thus, you should be able to see that given the positions and velocities at the welds of a cubic B-spline, we can dispense with all of the other control points and compute them on the fly as we evaluate each of the control points. A cubic B-spline represented this way (as a set of weld positions and velocities) is known as a *cubic Hermite spline*, or sometimes simply a cspline. The cspline is an extremely useful tool for producing smooth and natural animations.

Summary

In this chapter, you learned some mathematical concepts crucial to using OpenGL for the creation of 3D scenes. Even if you can't juggle matrices in your head, you now know what matrices are and how they are used to perform the various transformations. You also learned how to construct and manipulate the matrices that represent the viewer and viewport properties. You should now understand how to place your objects in the scene

and determine how they are viewed on screen. This chapter also introduced the powerful concept of a frame of reference, and you saw how easy it is to manipulate frames and convert them into transformations.

Finally, we introduced the use of the `vmath` library that accompanies this book. This library is written entirely in portable C++ and provides a handy toolkit of miscellaneous math and helper routines that can be used along with OpenGL.

Surprisingly, we did not cover a single new OpenGL function call in this entire chapter. Yes, this was the math chapter, and you might not have even noticed if you think math is just about formulas and calculations. Vectors and matrices, and the application thereof, are absolutely crucial to being able to use OpenGL to render 3D objects and worlds. However, it's important to note that OpenGL doesn't impose any particular math convention upon you and does not itself provide any math functionality. If you use a different 3D math library, or even roll your own, you will still find yourself following the patterns laid out in this chapter for manipulating your geometry and 3D worlds. Now, go ahead and start making some!