12

2D Geometry

Up to this point we have treated the view and the controller separately by speaking of "essential geometry" that links the two. The controller calls the view's essential geometry method(s) to determine what object the controller should manipulate and how. For our scroll-bar example in chapter 11, the essential geometry consisted of checking the mouse against 5 rectangles. In many applications, the essential geometry is not so simple. In this chapter we will work through the necessary mathematics for dealing with geometric questions in most two-dimensional graphics interfaces. In chapter 13 we will address geometric transformations, which are a powerful model for a variety of interactive manipulations. In chapter 14 we will look at interactive techniques based on these geometric concepts.

To understand the geometry discussions in this book a basic knowledge of linear algebra is required. The basic concepts are presented in this chapter. More detailed discussions are found in the appendix (A1.1 and A1.2). The appendix also contains code for all of the operations. Following the matrix algebra we then look at the kinds of geometric problems that arise in interactive programs. All of the geometry problems in this chapter are solutions of either implicit or parametric equations. The differences between these two will be discussed. Lastly we will work through lines, circles, ellipses, rectangles, cubic curves and polygons deriving solutions for the geometric problems of interaction.

Basic matrix algebra

Before addressing the geometry issues we must review some basic matrix algebra. Those already familiar with linear algebra can skip this section. We will only be using real matrices through most of this book and will confine the linear algebra discussion to those cases. Our most rudimentary structure is the vector, which is a one-dimensional array of real numbers. Vectors are written as column matrices or just as a letter as in figure 12.1. For 2D problems vectors will have two or three elements as needed. On occasion we need a row vector, which is represented as the transpose of the column vector ($\mathbf{A}^T$).

$$\mathbf{A} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \qquad \mathbf{A}^T = \begin{bmatrix} u & v & w \end{bmatrix}$$

Figure 12.1 – Vectors and their transpose

A vector is a degenerate case of a matrix, which is a two dimensional array of real numbers. Many matrices are square, but they need not be. A matrix also has a transpose as shown in figure 12.2. The transpose is constructed simply by exchanging the rows and columns for each item in the matrix. We reference any element of a matrix M as $M_{r,c}$. For example, $M_{1,2}$ from figure 12.2 would be q. Note that the matrix indices start at 1 rather than the 0 used in most programming languages.

$$M = \begin{bmatrix} p & q \\ r & s \\ t & u \end{bmatrix} \qquad M^T = \begin{bmatrix} p & r & t \\ q & s & u \end{bmatrix}$$

Figure 12.2 – Matrices and their transpose

We will frequently construct matrices from row or column vectors. This will be true when we build the geometry matrix for various spline forms and when we construct representations for simultaneous equations. Figure 12.3 shows matrices constructed from two vectors

$$A = \begin{bmatrix} c \\ d \\ e \end{bmatrix} \qquad B = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \qquad S = \begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} c & f \\ d & g \\ e & h \end{bmatrix}$$

$$T = \begin{bmatrix} A^T \\ B^T \end{bmatrix} = \begin{bmatrix} c & d & e \\ f & g & h \end{bmatrix} = S^T$$

Figure 12.3 – Matrices from vectors

If we have an n by m matrix S and an m by p matrix T then we can compute the matrix product of S and T to produce an n by p matrix result. The definition of matrix product is:

$$S \bullet T = P \text{ such that } P_{r,c} = \sum_{i=1}^{m} S_{r,i} \bullet T_{i,c}$$

$$S \bullet T = \begin{bmatrix} c & f \\ d & g \\ e & h \end{bmatrix} \bullet \begin{bmatrix} c & d & e \\ f & g & h \end{bmatrix} = \begin{bmatrix} (c^2 + f^2) & (cd + fg) & (ce + fh) \\ (dc + gf) & (d^2 + g^2) & (de + gh) \\ (ec + hf) & (ed + hg) & (e^2 + h^2) \end{bmatrix}$$

Figure 12.4 – Matrix multiply

Figure 12.4 shows the most common matrix multiply for 2D geometry. The more general matrix multiplication algorithm is found in appendix A1.2b. There is a special case of matrix multiply called the dot-product that is the product of two vectors of equal dimension. The result is a scalar real number.

$$A \bullet B = A^T \bullet B = \begin{bmatrix} c & d & e \end{bmatrix} \bullet \begin{bmatrix} f \\ g \\ h \end{bmatrix} = cf + dg + eh$$

Figure 12.15 – Vector dot product

There is a square (same number of rows and columns) identity matrix I that is all zeros except for ones down the diagonal. The identity matrix is defined as $I \bullet A = A \bullet I = A$ for any matrix A. Matrix multiplication is associative but not commutative. That is

$$A \bullet (B \bullet C) = (A \bullet B) \bullet C$$

however

$$A \bullet B \neq B \bullet A$$

Figure 12.16 – Associative but not commutative

For many square matrices M there exists an inverse matrix M^{-1}, such that $M \bullet M^{-1} = I$ and $M^{-1} \bullet M = I$. There is not always an inverse for a square matrix, but where one exists we can use it to solve many problems. The algorithm for computing the inverse of a matrix is given in appendix A1.2e.

The above set of matrix algebra operations is sufficient for most of the geometry to be discussed in the next three chapters. A good grounding in linear algebra beyond what is given here can be very helpful in many applications.

Geometric problems for user interface work

For a given geometric shape there are seven basic problems that may arise in user interfaces. They are:

- Scan conversion
- Constructing a shape from a set of points
- Distance between a shape and a point
- Nearest point on a shape
- Intersection of two shapes
- bounding box
- Inside or outside of a shape

Scan conversion is an algorithm for deciding which pixels on a screen or in an image correspond to some geometrically described shape. We will assume that our underlying Graphics object has code to solve this problem and we will not further discuss this here. The algorithms can be found in most good graphics texts[1].

Our next interactive task is to construct or change a shape by manipulating points. Virtually all 2D interaction consists of manipulating points with a mouse or pen and then inferring the change to some shape from the new point set. The simplest case is manipulating a line by its two end points.

Computing the distance between a shape and a point is critical for selecting geometric objects on the screen. Users do not want to exactly hit a line with the mouse, they want to hit close to it and have the computer figure it out. Computing the distance between a shape and a point is critical to determining when a mouse position is close enough for selection.

In figure 12.17 a user is trying to draw a line that ends exactly on a circle. The mouse position, however, is not exactly on the circle, it is near the circle. Again the user does not want to hit the circle exactly because that takes time and distracts from the task. They want the computer to “snap” the mouse point to the nearest point on the circle. For such snapping operations we need to compute the nearest point on some shape from some mouse or pen point. Frequently we will compute the distance to a shape by computing the nearest point and then computing the distance between the two points.

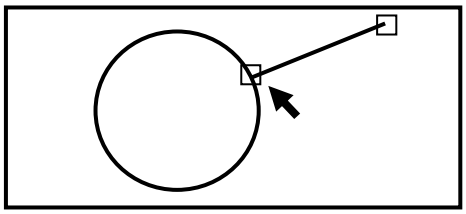


Figure 12.17 – Snapping to nearest point on a shape

In some situations we need to intersect two shapes. Shape intersection allows us to construct a point that meets some criteria rather than forcing the user to interactively approximate such a point.

Selecting shapes can be computationally expensive. This is particularly true when there are thousands of shapes on the screen and we must compare them all to the mouse position. It is sometimes faster to compute a bounding box around the shape and test the bounding box against the mouse point. If the mouse point is outside of the bounding box then we need not spend any more time on that shape. Checking a bounding box against a point involves only 4 compares, which is very efficient. What we need then is an efficient mechanism for computing the bounding box.

When we have closed shapes such as in figure 12.18 we need to determine whether a mouse point is inside or outside of a shape. This is our basic selection mechanism for closed shapes.

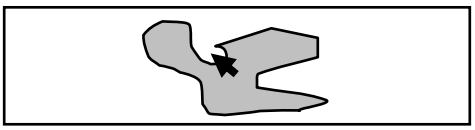


Figure 12.18 – Inside or outside of a closed shape

Forms of equations and their solutions

To represent geometric shapes we will use two forms of equation: implicit and parametric. The implicit form in 2D is an equation of the form $f(x,y)=0$. An example would be the equation of a 2D line which is $ax+by+c=0$. Implicit equations have the property of dividing the space into two half-spaces $f(x,y)>0$ and $f(x,y)<0$. For several shapes this is useful for separating inside and outside. For some shapes $|f(x,y)|$ is distance between $[x\ y]$ and the shape. Given the

implicit equations of two shapes we have two equations and two unknowns, which can be solved algebraically to produce the intersection point.

We also represent geometric shapes with parametric equations. In parametric equations there is an additional parameter *t* that can be any real number. For a variety of finite shapes we constrain *t* to the range 0.0 to 1.0. In 2D there are two equations *g(t)=x* and *h(t)=y*. Using these two equations we can pick any real value of *t* and get a point *[x y]*. Parametric equations can represent shapes for which the implicit form is awkward and they also are the basis for many blending functions.

Parametric equations also have a general form for computing the nearest point on a shape to some point *[x y]*. Given two parametric equations and some parameter value *t* the distance is $dist(x,y,t) = \sqrt{(g(t)-x)^2 + (h(t)-y)^2}$. Using calculus we can replace *g(t)* and *h(t)* with the parametric equations for our shape, take the first derivative with respect to *t*, set it equal to zero and solve. We can simplify matters by throwing away the square root since minimizing the square will produce the same answer.

For some geometric forms the calculus approach is problematic because we end up trying to solve polynomial equations of degree 4 or higher. In this case we remember that in a user interface any answer within a pixel or two is good enough because the user cannot do any better on their own. There is a simple recursive algorithm that will produce an answer that is close enough for interactive work. Assuming that our parameter *t* lies in some range (usually 0.0 to 1.0) we can slice that range into small parts and find the nearest point. We can then take the values of *t* on either side of that nearest point and create a new much smaller range and try again recursively. We stop when the endpoints of the range are less than one pixel apart. This algorithm is shown in figure 12.19 and works for virtually any parametric shape. This algorithm is guaranteed to converge. However, there is a possibility that it may converge to a local rather than true minimum. However, chopping the shape into 10 regions will prevent such problems for all but the most bizarre shapes.

```
public Point nearestPoint(Point p, float lowerT, float upperT)
{
	int N=10; // any integer in this range is fine
	float inc = (upperT-lowerT)/N;
	Point lowP = computePoint(lowerT);
	Point hiP=computePoint(upperT);
	if ( dist(lowP, hiP)<1.0 ) return lowP; // close enough for pixel resolution

	float nearT = lowerT;
	Point nearP=lowP;
	float nearD = dist(nearP,p);

	for (float t=lowerT+inc; t<=upperT; t=t+inc)
	{
		Point tp=computePoint(t);
		if (dist(tp,p)<nearD)
		{
			nearD=dist(tp,p);
			nearT=tp
			nearP=tp;
		}
	}
	float newLow=nearT-inc;
	if (newLow<lowerT) newLow=lowerT;
	float newHi=nearT+inc;
	if (newHi>upperT) newHi=upperT;
	return nearestPoint(p,newLow, newHi);
}
private float dist(Point a, Point b)
{	// returns the square of the distance which is acceptable for minimization of distance
	float dx=a.x-b.x;
	float dy=a.y-b.y;
	return dx*dx+dy*dy;
}
private Point computePoint(float t)
{
	Point rslt;
	rslt.x=g(t);	// X component of the parametric equation
	rslt.y=h(t);	// Y component of the parametric equation
	return rslt;
}
```

Figure 12.19 – Nearest point on a parametric shape

Geometric shapes

Using our matrix tools, as well as implicit and parametric forms we can now describe a set of useful geometric shapes and address each of the interactive tasks for each shape. The shapes we will consider are:

- Points and vectors
- Lines
- Circles and ellipses
- Cubic curves
- Rectangles
- Polygons
- Curvilinear shapes

Points and vectors

In some sense points are trivial in that they have an X and a Y in 2-space. However, to simplify their use with matrix algebra we will frequently represent points in *homogeneous coordinates* where each point is a triple *[x y 1]*. The constant 1 will prove useful in many equations and in the geometric transformations of chapter 13. It is also sometimes helpful to represent vectors as well as points. A vector has a direction and a magnitude and is frequently represented by two components *[dx dy]*. A vector has no position. For example Northwest at 50 kph has a direction (Northwest) and a magnitude (50 kph) and is the same no matter where in the city you are driving. It can be represented as $[-50\sqrt{2}, 50\sqrt{2}]$. We will frequently represent vectors in homogeneous coordinates as *[x y 0]*. The zero multiplier will eliminate position information (irrelevant for vectors) in many of our equations. A vector is frequently constructed by the difference between two points. Suppose we have two points *A=[p q 1]* and *B=[r s 1]* , their difference *A-B* would be the vector *V=[p-r q-s 0]*.

Lines

Lines are the next simplest geometric shape. A line is constructed from two endpoints as shown in figure 12.20, which also shows the parametric representation for a line. The bounding box for a line is simply the minimum and maximum X and Y components of the end points.

Figure 12.21 shows the vector algebra equivalent of the parametric representation. By subtracting point *A* from point *B* we get a vector along the line that has the same length as the distance between *A* and *B*. The parameter *t* adjusts how much of that vector is added to point *A*. If *t* is 0.0 then the result is simply

point *A*. If *t* is 1.0 then the resulting point is point *B*. Various values of *t* between 0.0 and 1.0 will produce all of the points on the line that lie between point *A* and point *B*. Other values of *t* will produce other points on the infinite line. It is a nice property of the parametric form that bounding *t* directly defines a finite line segment.

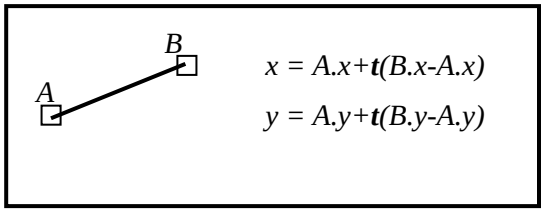


Figure 12.20 – Parametric equation of a line

$$\boxed{L = A + t(B - A)}$$

Figure 12.21 – Vector equation for a parametric line

We can solve for the nearest point *L* to some mouse point *M* as shown in figure 12.22. We formulate the square of the distance between *L* and *M*, take the first derivative with respect to *t*, set the derivative equal to zero, solve for *t* and substitute *t* back into the original equation of the line to produce a function for *L*. The equation in figure 12.22 is cheaper than the iterative solution in figure 12.19

$$dist^2(L,M) = |L-M|^2 = (L.x - M.x)^2 + (L.y - M.y)^2$$

$$dist^2(L,M) = (A.x + t(B.x - A.x) - M.x)^2 + (A.y + t(B.y - A.y) - M.y)^2$$

$$\frac{ddist^2}{dt} = 2(A.x + t(B.x - A.x) - M.x)(B.x - A.x) + 2(A.y + t(B.y - A.y) - M.y)(B.y - A.y)$$

$$0 = t\left[(B.x - A.x)^2 + (B.y - A.y)^2\right] + (A.x - M.x)(B.x - A.x) + (A.y - M.y)(B.y - A.y)$$

$$t = \frac{-(A.x - M.x)(B.x - A.x) - (A.y - M.y)(B.y - A.y)}{\left[(B.x - A.x)^2 + (B.y - A.y)^2\right]}$$

$$L = A + t(B - A) = A + \left[\frac{-(A.x - M.x)(B.x - A.x) - (A.y - M.y)(B.y - A.y)}{\left[(B.x - A.x)^2 + (B.y - A.y)^2\right]}\right](B - A)$$

Figure 12.22 – Nearest point *L*, on a parametric line, to point *M*

Knowing the nearest point *L* to the point *M* we can compute the distance of *M* to the line. This is generally not used, however, because the implicit solution is much cheaper.

Given parametric equations for two lines we can solve for their intersection simply by setting them equal to each other and solving for their two parameters as shown in figure 12.23. Note that the vector equation of a line is actually two equations (one for X and one for Y) so there are two equations and two unknowns (*s* and *t*) with a simple algebraic solution.

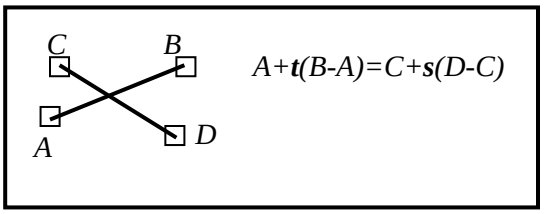


Figure 12.23 – Intersecting two parametric lines

Implicit form of a line

The implicit equation for a line is any point *[x y]* such that *ax+by+c=0* for three real coefficients *a, b* and *c*. This equation can also be represented in vector form using homogeneous coordinates $[a \quad b \quad c] \bullet [x \quad y \quad 1] = 0$. The equation of a line in 2D is a special case of the hyperplane equation in *N*-space. The hyperplane equation has many uses in dividing spaces. A most important property of the hyperplane equation of a 2D line is that the vector *[a b]* is a normal vector to the line or perpendicular to the line. We can normalize the equation of the line so that the normal vector is a unit vector (length 1). The normalized formulation is shown in figure 12.24.

$$\frac{a}{\sqrt{a^2+b^2}}x + \frac{b}{\sqrt{a^2+b^2}}y + \frac{c}{\sqrt{a^2+b^2}} = 0$$

Figure 12.24 – Normalized implicit equation of a line

Note that there are an infinite number of coefficients *a, b* and *c* that can represent the same line. Multiplying a set of coefficients by any non-zero real number will produce another set of coefficients that describes the same line. This is a problem when we try to construct an implicit equation for a line given two points because we end up with two equations (one for each point) and three unknowns (*a, b* and *c*).

To construct the implicit equation we first take advantage of the fact that *[a b]* is known to be a vector perpendicular to the line. We also know that (*B*-*A*) is a vector along the line. What we need is a vector perpendicular to (*B*-*A*). For any vector *[u v]* there is a perpendicular vector *[-v u]*. Using this we can construct the implicit equation for a line containing two points *A* and *B* as shown in figure 12.25.

$$\begin{aligned}
&a = B.y - A.y \\
&b = -(B.x - A.x) = A.x - B.x \\
&ax + by + c = (B.y - A.y)x + (A.x - B.x)y + c = 0 \\
&(B.y - A.y)x + (A.x - B.x)y = -c \\
&(B.y - A.y)A.x + (A.x - B.x)A.y = -c \\
&\\
&(B.y - A.y)x + (A.x - B.x)y - (B.y - A.y)A.x - (A.x - B.x)A.y = 0
\end{aligned}$$

Figure 12.25 – Implicit equation of a line

If we have a normalized equation of a line as shown in figure 12.24 then the normal vector is of length 1. One of the properties of a normalized implicit

equation is that is really a function for the distance from the line. Figure 12.26 shows a function for the distance from a mouse point *M* to a line.

$$dist(M,a,b,c) = \frac{a}{\sqrt{a^2+b^2}} M.x + \frac{b}{\sqrt{a^2+b^2}} M.y + \frac{c}{\sqrt{a^2+b^2}}$$

Figure 12.26 – Distance from *M* to the line *a,b,c*

We can compute a point *L* on the line that is nearest to mouse point *M* using similar techniques. We have two unknowns (*L.x* and *L.y*) and two equations (the equation of the line and the first derivative of the distance). However, there is an advantage to the parametric solution to the nearest point problem. Lines are infinite. The implicit equation will give us a nearest point on the line but does not directly tell us if the point in on the line segment. The parametric solution in figure 12.22 gives us a value for *t* that can be used to determine if the nearest point is actually on the line segment that we are interested in. Therefore we use the parametric rather than implicit form for nearest point problems.

Circles/ellipses

We discuss circles and ellipses together because a circle is just a special case of an ellipse. In this section we will only discuss ellipses whose axes are aligned with the X and Y axes. This is the most common interactive form for an ellipse. Rotated ellipses that are not axis aligned are dealt with more easily using the scale and rotation transformations described in chapter 13.

We can best describe a circle either as a center point with a radius or as a center point and some point on the diameter of the circle as shown in figure 12.27. The center/radius form is more comfortable mathematically, but the two point form is easier to manipulate interactively.

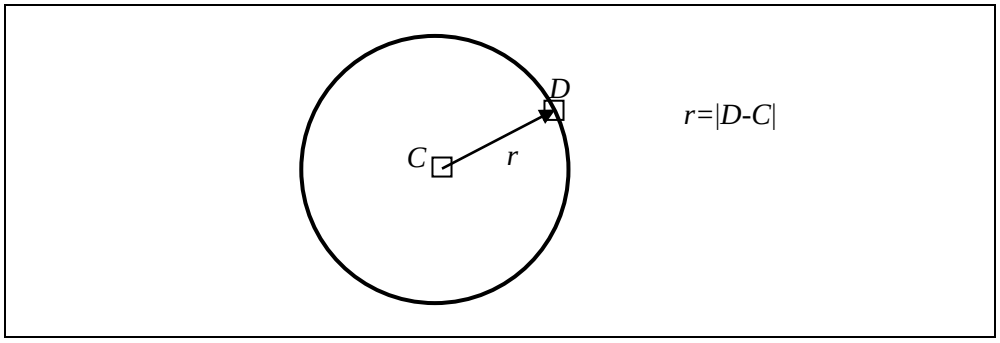


Figure 12.27 – Circle constructed from two points

The implicit equation of a circle is constructed by asserting that the distance between any point *[x y]* and the center *C* must be equal to the radius of the circle. The implicit equation is shown in figure 12.28. Figure 12.18 also shows the squared formulation that asserts the same thing, but removes the redundant square root. The final formulation divides the equation by r^2 to provide an alternative formulation that we will use later with ellipses.

$$\sqrt{(x - C.x)^2 + (y - C.y)^2} - r = 0$$

$$(x - C.x)^2 + (y - C.y)^2 - r^2 = 0$$

$$\left(\frac{x - C.x}{r}\right)^2 + \left(\frac{y - C.y}{r}\right)^2 - 1 = 0$$

Figure 12.28 – Implicit equation of a circle

Distance from a mouse point *M* to a circle with an implicit equation can be handled in the same way we handled lines. The first formulation of the implicit equation is actually a distance from the shape formulation as shown in figure 12.29.

$$dist(M, C, r) = \sqrt{(M.x - C.x)^2 + (M.y - C.y)^2} - r$$

Figure 12.29 – Distance from a point *M* to a circle *C,r*

The nearest point on a circle is only slightly more complicated. As shown in figure 12.30, the nearest point *N* is along the vector *M-C* but is only the distance *r* away from the center. To find *N* we compute *M-C*, normalize it to a unit vector, multiply it by *r* to make it the right length and then add it to *C* to find the point *N*.

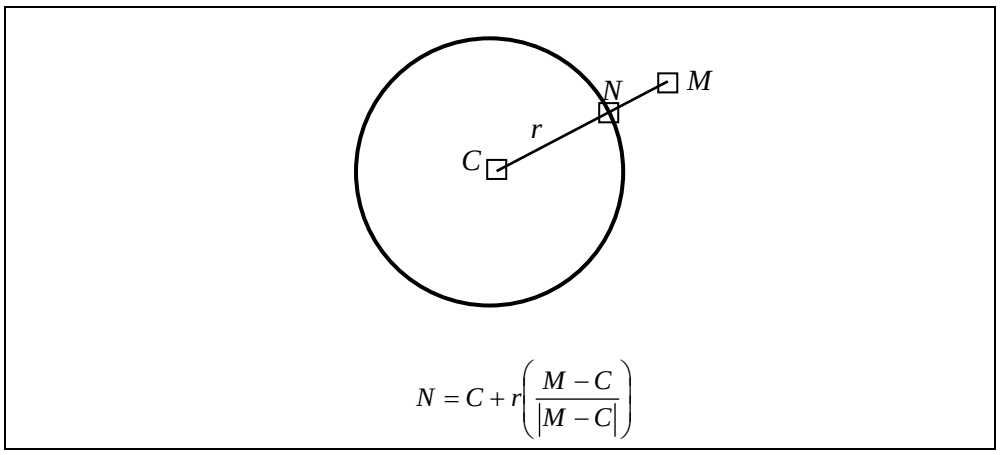


Figure 12.30 – Nearest point *N* to *M* on circle *C,r*

The circle is our first shape that has an inside and an outside. The implicit form of the circle gives us the inside condition shown in figure 12.31. If the distance to the diameter is less than zero then the point is inside of the circle.

$$\left(\frac{M.x - C.x}{r}\right)^2 + \left(\frac{M.y - C.y}{r}\right)^2 - 1 \le 0$$

Figure 12.31 – Inside test for a point *M* and a circle *C,r*

Parametric form

The parametric form of a circle is based on trigonometry. Our parameter *t* is the angle (in radians) around the circle. Using this we can describe the circle by the equations shown in figure 12.32.

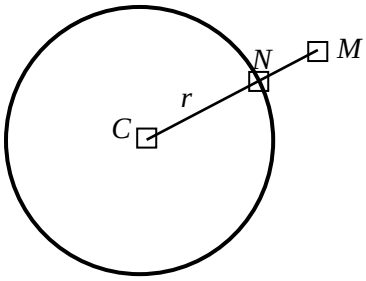
$$\begin{aligned} x &= r \cdot \cos(2\pi t) + C.x \\ y &= r \cdot \sin(2\pi t) + C.y \end{aligned}$$

Figure 12.32 – Parametric form of a circle

The parameter *t* is multiplied by 2π so that as *t* ranges from 0.0 to 1.0, the angle will range from 0.0 to 2π radians. This preserves our convention of parameters that range from 0.0 to 1.0.

Finding the nearest point on the circle is best solved by the technique in figure 12.30, which uses neither the implicit nor the parametric form. However, it is frequently useful to determine the parameter *t* at the nearest point *N*. This can

be used for interactively defining arc segments. The tangent of the angle is easily computed using the *x* and *y* components of *M*-*C*. However, if *M.x*-*C.x* is zero, the tangent is infinite. Fortunately most trigonometry libraries provide an *arctan2* function that will compute the arctangent directly without the division. This rather directly gives us a value for *t* as shown in figure 12.33.



$$\tan(2\pi t) = \frac{M.y - C.y}{M.x - C.x}$$

$$2\pi t = \tan^{-1}\left(\frac{M.y - C.y}{M.x - C.x}\right)$$

$$2\pi t = \arctan 2\left(M.y - C.y, M.x - C.x\right)$$

$$t = \frac{\arctan 2\left(M.y - C.y, M.x - C.x\right)}{2\pi}$$

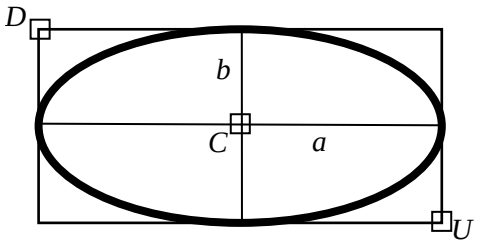
Figure 12.33 – Nearest point

The parametric form of a circle is quite convenient for intersections with lines or other circles simply by equating the *x* and *y* equations to produce two equations and the two unknown parameters for the two shapes. The parametric form is not useful for determining inside/outside.

Ellipses

As mentioned earlier, a circle is a special case of an ellipse. In most interactive graphics we use ellipses that are aligned with the X and Y axes. Normally such ellipses are interactively constructed in terms of a bounding rectangle for the ellipse as shown in figure 12.34. The user has specified point *D* (where the mouse went down) and point *U* (where the mouse went up). From these we compute the center *C* and the two radii *a* and *b*. The bounding box for an ellipse is computed by addition/subtraction or the radii from the center. In

many cases it is the bounding box that is retained as the representation of the ellipse.



$$C = \frac{D+U}{2} \qquad a = \frac{U.x - D.x}{2} \qquad b = \frac{U.y - D.y}{2}$$

Figure 12.34 – Coefficients of an ellipse *C,a,b* from two points *U,D*

In this formulation an ellipse is the same as a circle except that the radius in *x* is different from the radius in *y*. By taking the implicit equation of a circle found in figure 12.28 we can derive an implicit equation for an ellipse shown in figure 12.35. Using a similar technique we convert the parametric equation for a circle from figure 12.32 into the parametric equations for an ellipse shown in figure 12.36.

$$\left(\frac{x - C.x}{a}\right)^2 + \left(\frac{y - C.y}{b}\right)^2 - 1 = 0$$

Figure 12.35 – Implicit equation for an axis aligned ellipse *C,a,b*

$$\begin{aligned} x &= a \cdot \cos(2\pi t) + C.x \\ y &= b \cdot \sin(2\pi t) + C.y \end{aligned}$$

Figure 12.36 – Parametric equations for an axis aligned ellipse *C,a,b*

The simple angle tricks for computing the nearest point and distance to the shape that we used for circles will not work on ellipses. The best solution is to use the general numeric technique for parametric shapes shown in figure 12.19. This will produce the parameter *t* of the nearest point from which the nearest point and the closest distance are easily computed. The implicit form of an ellipse does easily produce an inside test as shown in figure 12.37.

$$\left(\frac{x-C.x}{a}\right)^2+\left(\frac{y-C.y}{b}\right)^2-1\le 0$$

Figure 12.37 – Inside test for an axis-aligned ellipse *C,a,b*

Arcs

It is common to desire an arc which is only a portion of a circle or ellipse. Arcs are easily represented using parametric equations and placing bounds on the parameter *t*. When considering an arc, however, there is the problem of the zero angle which is the same as the angle 2π. Thus the point for parameter 0.0 is the same as the point for parameter 1.0. This is a problem when we desire an arc that extends through the zero angle as shown in figure 12.38. The points *S* and *E* might define two possible arcs *U* and *V*. This is easily handled by recognizing that *sin* and *cos* are periodic and we can use parameter values larger than 1.0. We can define *V* using the bounds for *t* of 0.125 to 0.875. We can cross the zero angle for arc *U* with the bounds 0.857 to 1.125.

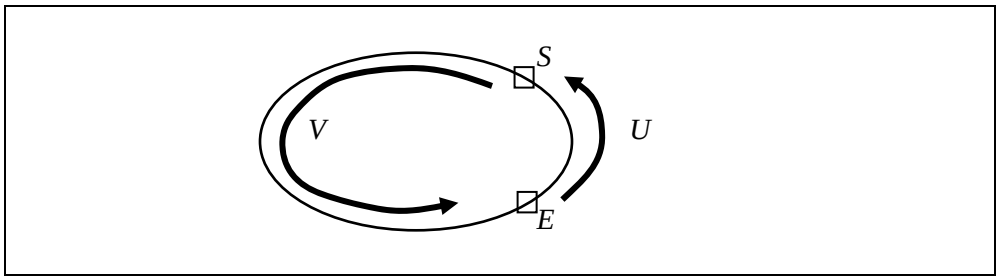


Figure 12.38 – Parameterization of arcs

Rectangles

Rectangles are one of the simplest closed shapes. They are represented by *minx, miny, maxx* and *maxy*. Finding the distance to an edge is simply a matter of comparing the corresponding coordinate of the mouse point to the edges and taking the smallest distance. Finding the nearest point is equally trivial. Rectangles are generally constructed aligned with the axes as with ellipses and they are created interactively in the same manner as an ellipse, as shown in figure 12.34. The inside test checks the point for larger than both mins and less than both maxes.

Curves

The simple shapes of lines, circles, ellipses and rectangles are generally not interesting enough for many needs. For creating arbitrary curved shapes such as those found in figure 12.39 we use curves defined by cubic polynomials that are connected together piecewise to form a particular shape.

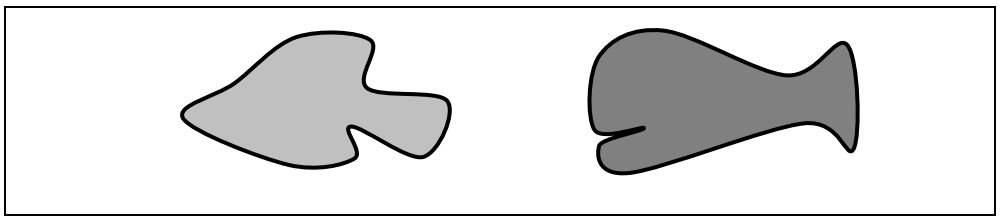


Figure 12.39 – Curved shapes

A quadratic polynomial (degree 2) can only have one bend. A cubic polynomial (decree 3) can only have up to two bends. However, the shapes in figure 12.39 have many bends. To get more bends in our shape we could use higher-degree polynomials. However, higher degree polynomials are very difficult to manage and specify interactively. Instead we use many cubic polynomials and stick them together end-to-end to form a more complex shape. The shapes in figure 12.39 consist of 10-20 cubic curves stitched together.

Quadratic polynomials have the very nice property that many geometric problems can be solved using the quadratic equation. Cubic polynomials do not have this property. However, because we are going to construct shapes by piecing together many curves we want a representation that will easily blend between pieces. Figure 12.40 shows two thick curves that we want to blend together smoothly with the thin curve. There is no way to blend these two curves with only one bend (quadratic), however, we can blend any two curves with two bends (cubic) as shown. The cubic curve is the lowest degree polynomial that will smoothly blend two arbitrary curves. That is why the cubic polynomial is the most common representation for curved shapes.

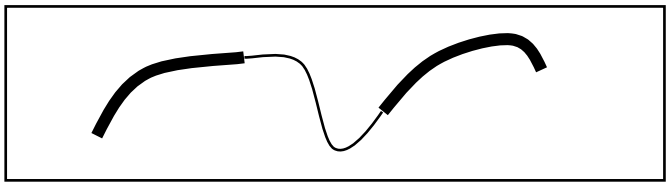


Figure 12.40 – Blending curves with cubic polynomials

Although there are implicit representations for cubic curves, they are not as easy to work with interactively; therefore we will only concern ourselves with parametric representations. The parametric representation of a cubic curve is shown in figure 12.41 along with its matrix representation.

$$x = at^3 + bt^2 + ct + d$$
$$y = et^3 + ft^2 + gt + e$$

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix} \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$C \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 12.41 – Parametric equations for cubic curves

The coefficients *a* through *h* are formed into the *coefficient matrix C*. The coefficient matrix forms a compact representation for the cubic curve. As with other shapes, the cubic curve is actually infinite in length. We will only use the part of the curve where the value of *t* lies between 0.0 and 1.0. Though the coefficient matrix *C* is a nice mathematical representation, it is an extremely poor interactive representation of a curve. No human can reliably specify a curve by manipulating the coefficients. We will discuss three forms of *spline* that accept a set of 4 control points and produce a coefficient matrix for a cubic curve. Each of these forms has different handles for how users control the shape of a curve.

As a mathematical representation the cubic polynomial is quite convenient. To find the nearest point on a curve we use the iterative solution shown in figure 12.19. Because there are up to two inflection points in a cubic curve there are local minima to the nearest point problem. That is why the algorithm in figure 12.19 sets the number of slices to higher than two. Using 10 or more initial slices of the parameter range ensures that the true minimum will be found in all but the most violently distorted curves, which very rarely occur in practice. With a nearest point solution we also can get the distance to the shape. Intersections are also handled using the standard parametric form. However, because cubic

polynomials are involved the solution is generally solved numerically rather than algebraically.

To compute the bounding box of a curve we must compute the maxima and minima in both *X* and *Y*. The first derivative with respect to the parameter *t* of a cubic polynomial is always a quadratic polynomial that can be solved using the quadratic equation to give zero, one or two values of *t* for each of the equations for *X* and *Y*. Any solutions for *t* that are outside the range 0.0 to 1.0 are discarded. The maxima and minima values of *t* from the *X* equation are added to the *X* coordinates of the curve endpoints to give 2 to 4 values. We take the largest and smallest of these as the right and left edge of our bounding box. We do the same in *Y* for the top and bottom edges. This gives us the smallest bounding box on the curve. For some spline formulations there are cheaper ways to compute a bounding box, but it is not always the tightest bounding box.

Continuity

Because curved shapes are created by piecing together a series of cubic polynomial curves, ensuring continuity at the joints is very important. In working with cubic curves we consider C(0), C(1) and C(2) continuity. C(0) continuity between curves *S* and *T* holds if the ending point (parameter $s = 1.0$) of *S* and the starting point (parameter $t = 0.0$) of *T* are the same point. C(0) means that the zeroeth derivatives are the same. This gives a curve with no gap at the joint, but there may be a sharp corner. C(1) continuity holds when the first derivatives of the two curves are the same at the shared point. C(1) gives a curve that is "smooth" across the joint with no sharp corner. In most interactive situations C(1) continuity is sufficient. However, in some design situations C(2) continuity is required. C(2) continuity holds if the second derivatives are the same at the joint point. We also assume that C(2) continuity implies C(1) and C(1) implies C(0). There are no formulations of cubic curves that can ensure C(3) or higher, therefore we do not consider them.

Spline formulations

As mentioned earlier the coefficient matrix has good mathematical properties but has a terrible user interface. What we want is a set of control points that can be manipulated interactively by the user and then a mechanism for converting those control points into a coefficient matrix. We can accomplish this by decomposing the coefficient matrix *C* into two matrices, the *geometry matrix G* and the *spline matrix S*, as shown in figure 12.42. As shown, the geometry matrix is constructed from four points *P0* through *P3*. These are our control points. The

spline matrix *S* is a constant matrix for each form of spline. Each form of spline uses a different arrangement of the four control points to accomplish different purposes. There are spline formulations that do not use points, but we will not consider them because they do not map as directly to a user interface. To create a cubic curve, we populate the geometry matrix with the positions of four control points and then multiply by the appropriate spline matrix. This will produce the 8 coefficients in the coefficient matrix and we have a parametric representation of our cubic curve. The interesting work is in selecting the correct spline formulation for a given problem.

$$G \bullet S = C$$

$$G \bullet S \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} P0 & P1 & P2 & P3 \end{bmatrix} \bullet S \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 12.42 – Decomposition of the coefficient matrix of a cubic curve

Bezier

The Bezier spline geometry consists of two end points (*P0,P3)* and two additional interior control points (*P1,P2*), as shown in figure 12.43. The interior control points are defined such that the vector *P1-P0* is tangent to the curve at *P0* and the vector *P3-P2* is tangent to the curve at *P3*.

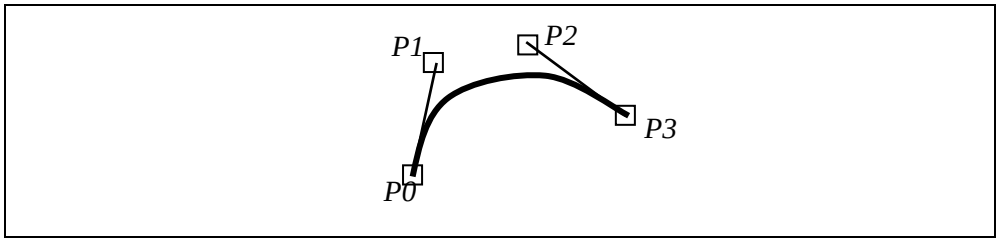


Figure 12.43 – Bezier control points

The Bezier curve has the most flexible interactive controls of the three forms we will discuss. The endpoints and their tangents are easily manipulated by moving the four control points. Increasing and decreasing the distance between an interior control point and its corresponding end point also controls the shape

of the curve. Figure 12.44 shows *P2* at various positions along the same line. This changes the shape of the curve by increasing or decreasing the "influence" of the tangent vector at *P3*.

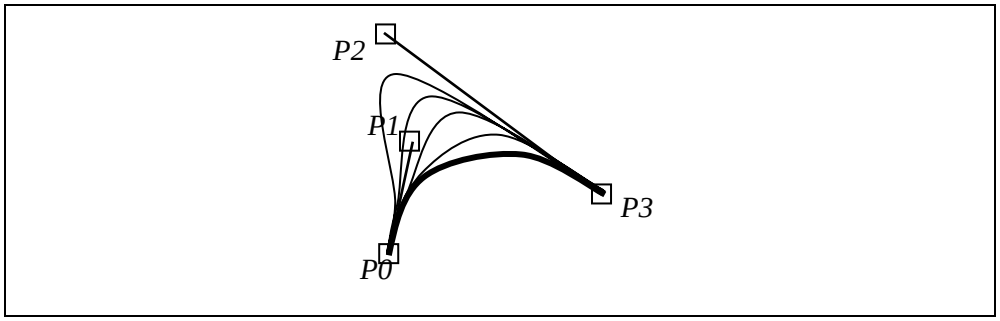


Figure 12.44 – Extending Bezier control points

Two Bezier curves *A* and *B* will have C(0) continuity if *A3* and *B0* are the same point. C(1) continuity is also easy to achieve by C(0) continuity and by *A2, A3, B0* and *B1* all being collinear, as shown in figure 12.45. Many user interfaces will maintain this relationship automatically. When a user moves *A2* the end points *A3* and *B0* stay in position, but *B1* is automatically repositioned to maintain colinearity and to maintain the distance between *B0* and *B1*. It is only possible for two Bezier curves to have C(2) continuity in very special cases.

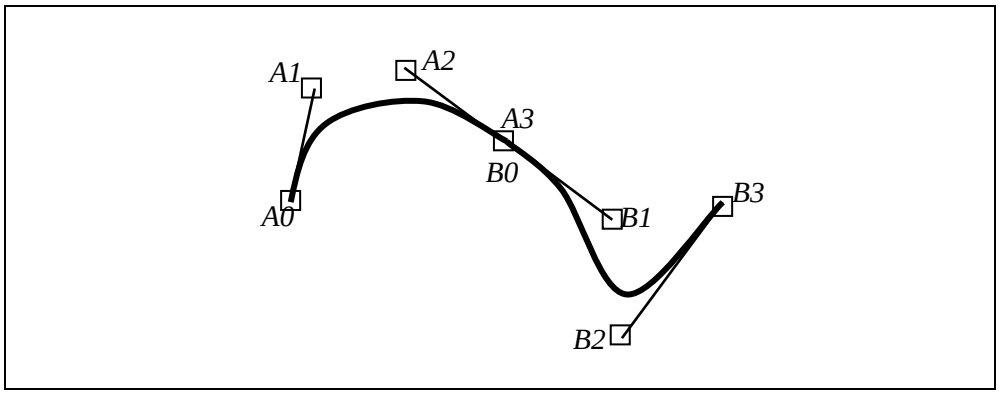


Figure 12.45 – C(0) and C(1) continuity in Bezier curves

Bezier curves also have the convex-hull property, which is that the *convex hull* of the control points is guaranteed to contain all of the points on the curve segment from 0.0 to 1.0. The convex hull of a set of points is the smallest polygon that contains all of the points in the set. We are not particularly

interested in computing the convex-hull, but by taking the maximums and minimums in X and Y of the four control points, we get a bounding box that contains the convex hull of the polygon. By the convex hull property we also have a cheaply computed bounding box for the curve. Because of their flexibility many graphics packages use Bezier control points rather than the coefficient matrix as their representation for drawing cubic curves. The Bezier representation also has a very efficient algorithm for scan-converting cubic curves that avoids repeated evaluation of the cubic polynomial and thus increases drawing speed. The constant spline matrix **Bz** is shown in figure 12.46.

$$\begin{bmatrix} P0 & P1 & P2 & P3 \end{bmatrix} \bullet Bz \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} P0.x & P1.x & P2.x & P3.x \\ P0.y & P1.y & P2.y & P3.y \end{bmatrix} \bullet \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 12.46 – The Bezier spline matrix

B-Spline

Our second spline organizes its points in a very different way. The control points are arranged in sequences with the control points of individual curves sharing control points with their neighbors. Curve segments drawn from the sequence of control points automatically have C(0) through C(2) continuity. The B-spline is very useful in this regard because with a minimum of user effort high continuity of the curve is achieved. Figure 12.47 shows such a curve composed of 4 curve segments and 6 control points. Note, however, that with B-splines the curves do not necessarily pass through the control points.

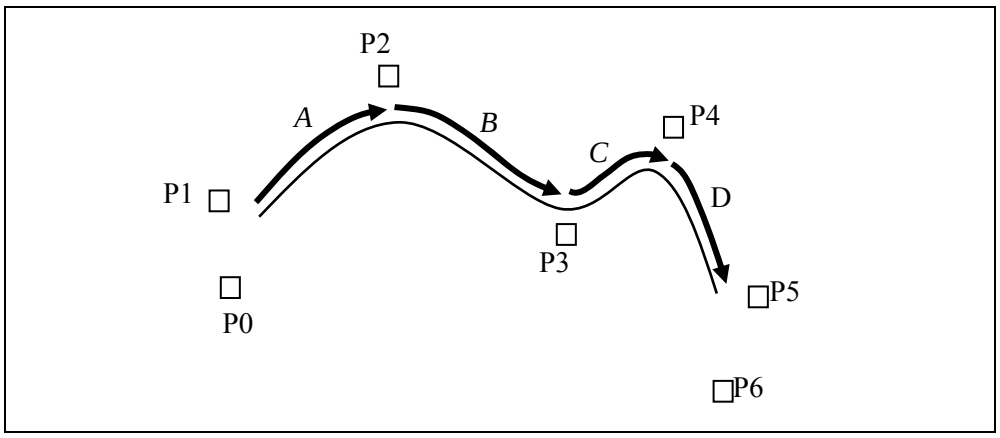


Figure 12.47 – Control points for four B-spline curves

Given a sequence of control points like that shown in figure 12.47, the geometry for a particular curve is defined relative to its start point at *Pi*. For example, curve *B* is defined relative to *P2* (*i*=2). Note also that there are two control points (*P0* and *P6*) that extend beyond the ends of the curve. These control the direction at the ends of the curve. The reason that the continuity comes automatically is because adjacent curves share control points. In figure 12.47 the points for *A* are [*P0 P1 P2 P3*] and the points for *B* are [*P1 P2 P3 P4*]. These two adjacent curves have three control points in common. The full geometry matrix for a B-spline curve along with the B-spline matrix **Bs** is shown in figure 12.48.

$$\begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix} \bullet Bs \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix} \bullet \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 12.48 – Matrix for B-spline curve

As with the Bezier spline, the B-spline has the convex hull property. We thus can compute a bounding box for a curve segment directly from the control points. It is important to note that this bounding box is not a tight bound. The points of the curve are inside of the box, but the box is generally larger than the curve. All of the other interactive geometry problems are solved by computing the coefficient matrix from the geometry matrix and then solving directly in the parametric form of the curve.

The B-spline uses a sequence of points to produce a sequence of curves that are automatically C(2) continuous. The sequence of points from which overlapping sets of control points are drawn is interactively more efficient than the Bezier because there are fewer controls for the user to manipulate. However, the B-spline does not pass through the control points. This makes the B-spline harder to control for a particular task. It is more difficult to get the curve to go exactly where you want it to go.

Catmull-Rom

The Catmull-Rom spline sets up its geometry matrix in a way similar to the B-spline in that it uses overlapping sets of control points from a sequence of points. The Catmull-Rom spline does pass through every control point. This makes it an excellent choice for interactive curves. The Catmull-Rom automatically C(0) and C(1) continuous. It does not have the convex-hull property. To find a bounding box of a Catmull-Rom curve segment we must convert it to one of the other representations. Figure 12.49 shows the matrix **Cr** for the Catmull-Rom spline.

$$\begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix} \bullet Cr \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix} \bullet \frac{1}{2} \begin{bmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \bullet \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 12.49 – Spline matrix for Catmull-Rom spline

Conversion between spline forms

All three of the curve forms that we have discussed are simply different controls for the same formulation of cubic curves. They are different ways of converting control points into polynomial coefficients. In fact, any curve represented by a coefficient matrix can be converted into any of the three geometry matrices. Because all of the forms represent the same polynomials we can convert the geometry matrix of one form of spline into the geometry matrix of any other. Figure 12.50 shows how to convert a Catmull-Rom G_{cr} into a Bezier G_{bz}. This, for example, would allow us to compute a bounding box for a Catmull-Rom spline or draw a Catmull-Rom spline using a graphics package that only supports Bezier curves.

$$G_{cr} \bullet Cr = C$$
$$G_{bz} \bullet Bz = C$$
$$G_{cr} \bullet Cr = G_{bz} \bullet Bz$$
$$G_{cr} \bullet Cr \bullet Bz^{-1} = G_{bz} \bullet Bz \bullet Bz^{-1}$$
$$G_{cr} \bullet Cr \bullet Bz^{-1} = G_{bz}$$

Figure 12.50 – Converting Catmull-Rom to Bezier

Because both splines compute the same coefficient matrix, we set them equal to each other. We then multiply both sides by the inverse of the Bezier matrix *Bz* and we get a function that will compute Bezier geometry from Catmull-Rom geometry. A similar technique will convert any curve into any other.

It is very important to note that the continuity properties are not changed by these conversions. Converting Catmull-Rom to B-spline will not introduce C(2) continuity. The curve itself does not change. Given two adjoining Catmull-Rom splines *A* and *B* we can convert them into two B-splines *A'* and *B'*. However, the splines *A'* and *B'* do not share any control points therefore there are no additional continuity guarantees. Constructing two curves using overlapping control points is not the same as constructing them from other representations.

Polygons

One of the most important closed shapes is the polygon. Polygons are represented by a sequence of vertex points and it is assumed that adjacent vertex points are connected by a straight line. Nearest point and distance to the polygon border problems are solved using each of the line segments and taking the smallest distance. The bounding box is defined by the maximums and minimums of the vertex points in X and Y. The most challenging problem is to determine

whether a point is inside or outside of the polygon. The approach, as shown in figure 12.51 is to intersect a horizontal line that passes through the point *M* with all of the edges of the polygon. This produces a set of intersection points. We then count the number of intersection points whose X coordinate is less than *M.x*. If the number of points to the left (less than) is odd then the point is inside the polygon. If it is even then *M* is outside. The idea is that if you start from outside of the polygon then each time you cross a boundary you transition from outside to inside, inside to outside and so on.

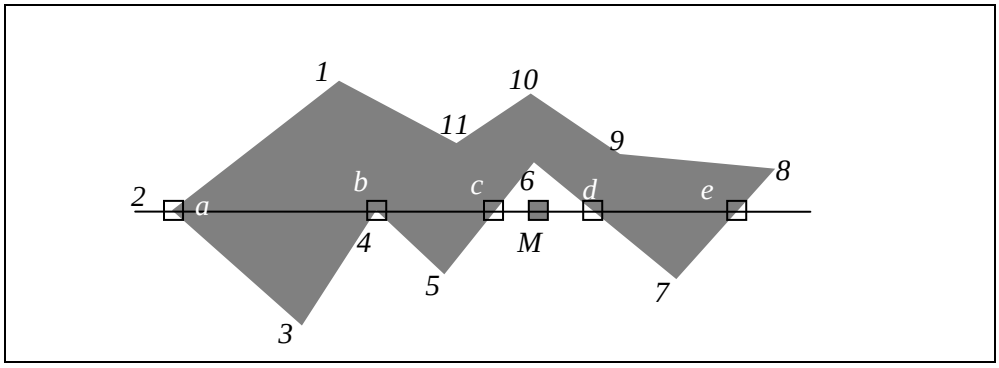


Figure 12.51 – Inside/Outside test on a polygon

Figure 12.51 shows an odd number of intersection points (*a,b* and *c*) to the left of *M*. An odd number would indicate that *M* should be inside of the polygon when obviously it is outside. The problem is that points *a* and *b* actually represent two intersection points each because they are each on two line segments (1-2,2-3 and 3-4,4-5). However, they each represent a different case. In the case of point *a,* the horizontal line entered the shape and thus had only one inside/outside transition. At point *b* the horizontal line transitions outside and then right back inside and thus should be treated as two transitions.

Our problem occurs when the horizontal line passes through one of the vertices of the polygon. These vertices fall into two cases. In the first case the sign of the change in Y is the same for both line segments on the vertex. For example the sign of change in Y for 1-2 is negative and for 2-3 it is also negative. Therefore this vertex should only be counted once. The same is true for vertex 9 where the change in Y for 8-9 is positive and for 9-10 is positive. The other case is where the change in Y reverses at the vertex. At vertex 4 we have a positive change in Y for 3-4 and a negative change for 4-5. In this case the vertex should be counted twice. For vertex 7 the change in Y reverses from negative (6-7) to

positive (7-8). If our horizontal line had passed through vertex 7 then we would count it twice.

With careful checking of the vertices, the inside/outside test is an easy algorithm for determining whether a polygon has been selected. It is almost always more efficient to first check to see if point *M* is inside of the bounding box of the polygon. For most polygons on the screen the point *M* is outside and the bounding box check can avoid the more costly analysis of the intersection points.

Curvilinear enclosed shapes

Polygons are somewhat restrictive in the kinds of shapes that they can represent. For most graphics packages the most general shape is the curvilinear shape. Essentially it is like a polygon except that the edges need not be only lines, they can be any cubic polynomial. Note that lines are degenerate cubic polynomials with zero coefficients for t^3 and t^2. Curvilinear shapes form the basis for most type-face systems. Looking closely at the letter "T" in figure 12.52 we see 12 straight lines filled in by 4 curves. The letter "S" is mostly curves with only 6 straight lines. The letter "a" has a hole in the middle of the shape and the letter "i" is composed of two separate shapes.

Figure 12.52 – Curvilinear shapes for Times-Roman letters

We treat curvilinear shapes in much the same way that we did polygons. For nearest point we address each edge (curved or linear) separately and take the closest. Selection, however, can be a problem because a given cubic curve can have up to two inflection points that may or may not create local minima/maxima in Y. Figure 12.53 shows the issues that we must address in adapting the inside/outside test to shapes with cubic curved edges.

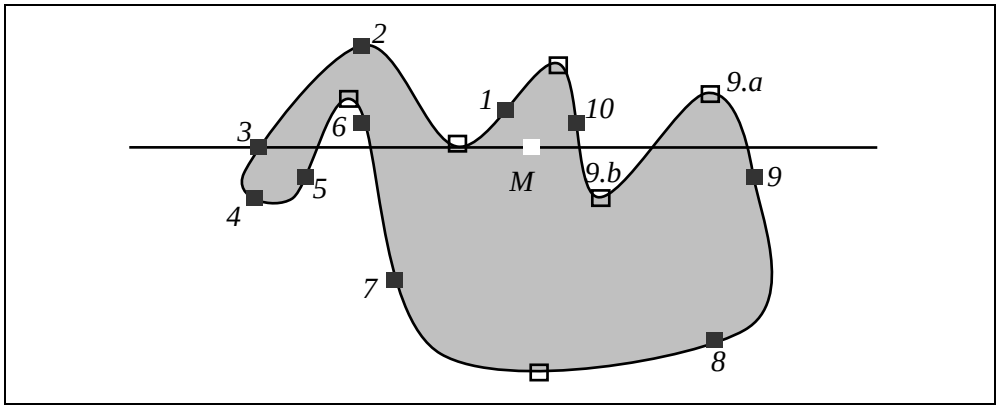


Figure 12.53 – Inside/Outside on curved shapes

The same problems occur in curved shapes as in polygons when the horizontal test line goes through the endpoints of two adjoining curves (vertex 3). We can solve those problems using the same techniques as we used with polygons. The cubic curves offer two additional problems not found in polygons. The first is that a cubic curve may intersect a line up to 3 times (curve 9-10). The second is that local minima and maxima may intersect the line (curve 1-2).

Our first step is to break each curved edge at its minima and maxima. Finding the minima and maxima of the curve relative to Y is easy. We take the first derivative of Y component of the curve equation, set it equal to zero and apply the quadratic equation. This will produce up to 2 values of the parameter *t* where maxima or minima are located. We discard any solutions that do not lie between 0.0 and 1.0 because they are outside of our curve segment. These maxima and minima are shown by the hollow squares in figure 12.53. We now have subsegments of curves that are monotonic in Y. Being monotonic in Y there is at most one intersection with the line in each subsegment. Consider for example the subsegements (9-9.a, 9.a-9.b, and 9.b-10). A subsegment of a curve is defined by a range of *t* that is smaller than 0.0 to 1.0.

For each curve or subsegment of a curve we can now test to see if the endpoints lie on opposite sides of our test line. If both endpoints of a curve segment lie on the same side of our line, there can be no intersection of that curve segment with the line. We can do this test because we have placed segment endpoints at all the maxima and minima so there are no intermediate intersection. For segments that do straddle the test line we can iteratively solve for the value of *t* whose Y component is equal to *M.y*. This algorithm is similar to that shown

in figure 12.19. If the line passes through a maxima or minima we always count it twice as we did with similar cases for polygons. We now can count the intersection points to the left of $M.x$ to determine odd/even for inside/outside.

Summary

We now have a set of primitive shapes that we can use to draw a variety of model presentations. We also have a set of mathematical tools for manipulating and selecting those shapes.

Exercises

1. If most of our shapes are not rectangular, why is computing the bounding box important? Based on this answer why is a loose bounding box usually just as good?
2. Describe how to use a parametric representation of a shape to produce answers that are good enough for interactive use.
3. What is the difference between a point and a geometric vector?
4. How does normalizing the implicit equation of a line help us to find the distance between some point and that line?
5. Given the implicit equation of a line, how do we find a vector perpendicular to that line?
6. Write an interactive program that will place a set of lines, ellipses and cubic curves on the screen. Use the geometry in this chapter to report when a mouse click is within three pixels of one of the shapes.
7. Why do we use arctan2 in computing angles rather than arctangent?
8. Why are cubic polynomials used rather than the simpler quadratic polynomials when creating curves?
9. Why is C(1) continuity easier to achieve for Catmull-Rom and B-spline than it is for Bezier curves?
10. Why do we care about the convex hull property when interactively working with curves?
11. When performing the inside/outside test on a shape with curved edges why do we need to solve for the minimum and maximum points.

---

[1] Foley, J.S., van Dam, A., Feiner, S. K., and Hughes, J. F. *Computer Graphics: Principles and Practice*, Addison-Wesley, (1995).