Drawing

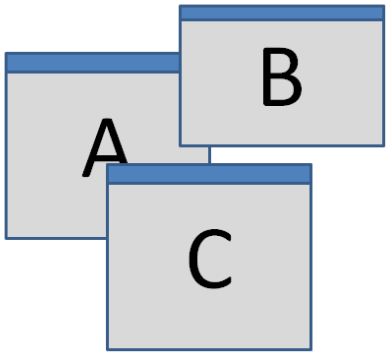
Obviously, to create a graphical user interface we must be able to draw on the screen. On modern computing devices such drawing is managed by a *windowing system*. Though there are a variety of these, they all share some common characteristics. Windowing systems need to be able to get any application to draw itself in the appropriate window. There are two basic approaches which are damage/redraw and presentation trees. We will look at both of these. Lastly we will review the geometry and styling of the basic shapes used for drawing in most system. We will save the drawing of text for the next chapter. Text has a variety of special needs.

Windowing System

On modern computing devices drawing involves the use of a *windowing system*. The windowing system on desktop and laptop devices behaves somewhat differently than and tablets and smartphones. In all of these cases there are a variety of similarities. Windowing systems are frequently integrated with the operating system as in Windows, Android, and Apple devices. Operating systems manage memory and processor time so that all processes get their needs met. Similarly a windowing system manages screen space so that all processes have their own drawing capability.

In figure 3-1 there are three applications A, B and C that each have their own windows. However, there is not enough room on the screen for all three applications to get what they want. The user has arranged that each of their windows overlaps. The advantage of using a windowing system is that each application thinks that they have their own rectangle of space in which they can draw. Application A knows nothing about applications B and C and the fact that they are occupying part of A's drawing space. Application A conducts its business as if it were the only application on the screen. This greatly simplifies the

implementation of graphical applications. If every application had to worry about every other application, the complexities would be overpowering.



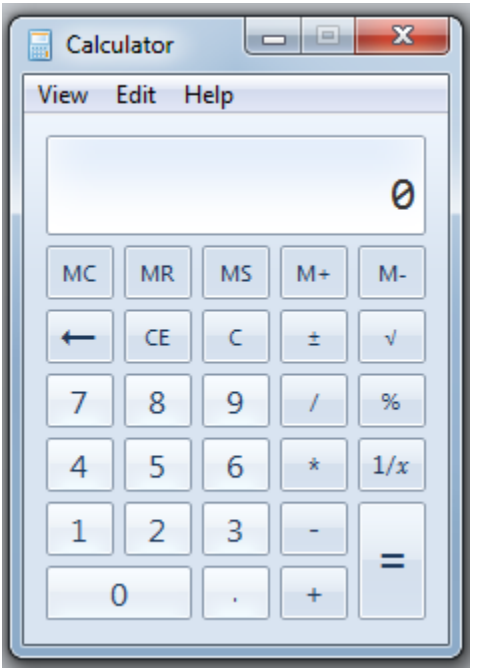
3-1 Overlapping windows managed by a windowing system

Clipping

An important capability of a windowing system is *clipping*. This is where any drawing that is done outside of the region belonging to that window is thrown away or clipped. In figure 3-1 the letter A has been clipped so that it does not overlap the window for application C. Most clipping is rectangular because most windows are rectangular. However, in the case of application A the clipping region is quite complex because the sections for B and C must be cut out of A's rectangular shape. In general, we do not concern ourselves with clipping other than to know it exists. We just assume that we have all the space we need to draw and let the windowing system take care of the rest.

In the case of most tablets and smartphones the top level windows occupy the entire screen space. In these cases the other windows are not drawn at all. We will see later how the damage/redraw mechanism deals with this case efficiently. Some tablets provide the ability to open multiple windows, but they do not overlap. This non-overlapping property means that clipping is always a rectangle which simplifies with windowing system's problems.

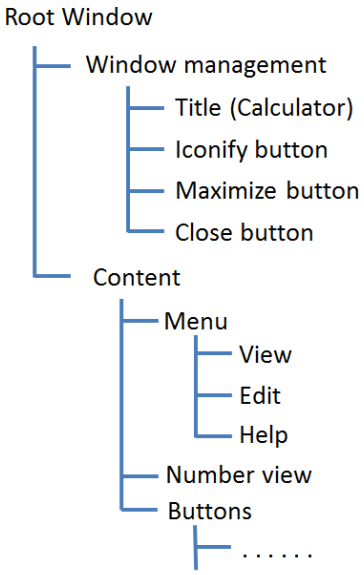
In some windowing systems, such as Microsoft Windows, the windows are not exactly rectangular. A careful look at the calculator in figure 3-2 shows that the window has rounded corners. In other systems the windows can be arbitrarily shaped. This can be handled in two ways. The first is that the window is actually rectangular, but is transparent in some places (the round corners). In other situations a more expensive clip to a shape region is used. Either will work fine. For our discussion we will treat windows as rectangles. The special cases are rare and differ only in their clipping algorithm.



3-2 – Windows with rounded corners

Window trees

Modern windowing systems actually support a tree of windows nested inside of each other. Figure 3-3 shows the tree that describes the calculator in figure 3-2.



3-3 – Window tree for calculator

Each of these smaller windows behaves independently to perform its own function. Each has its own rectangular region that it manages. Sometimes the inner windows do not work as hard as the root level windows in figure 3-1. There are frequently lighter weight objects that perform all of their own work, but are assumed to be well-behaved. The clipping effort is eliminated. This might be true for buttons and the other kinds of objects that we see in figure 3-2.

Units of interaction (Widgets)

Each of the windows that we see is a semi-independent unit of interaction. These have various names in various systems. In Java they are called components. In Windows and C# they are called controls. In Android they are called Views (just to complicate other uses of the word). In X-windows they were called widgets, which is the term we will use throughout the book. When learning any new system, find out the name they are using. It will make everything else more clear. A widget is a single unit of interaction that can draw itself and respond to interactive inputs and communicate with other parts of the application.

Drawable objects

The windowing system needs to be able to tell any part of any application to draw itself at any time. If the window for application B in figure 3-1 is moved, then application A will need to redraw the part of itself that was exposed by moving window. The problem is that the windowing system does not have any idea what application A is or its purpose.

If there is an interface declaration for Drawable as shown below then the windowing system can be given a pointer to a Drawable object and it will know how to get that object drawn.

```
public interface Drawable
{
   public void paint(Graphics g);
}
```

In essence this says that anything that is drawable will have a paint() method that receives a Graphics object as its parameter. There are various programming language mechanisms for how this happens but all of them essentially associate the address of the widget's paint method with the window. The windowing system then knows to call that method and the right thing will happen. The paint() method can be called whenever needed.

In many systems, the paint() method is where the View of Model-View-Controller is implemented. The View object might have a pointer to its model and implement Drawable. When its paint() method is called it retrieves information from the model and draws by making appropriate calls on the Graphics object.

Graphics object

The value lies in the Graphics object. Sometimes this is called a Canvas (HTML5, some Python libraries). The graphics object defines the interface between our widgets, the windowing system and the underlying display hardware. The Graphics object has methods for drawing all of the various shapes. In most systems there are many implementations of the Graphics object and our Drawable objects do

not care which it is. One implementation might have special code that takes advantage of features on a particular graphics card. Another implementation might package the calls and ship them over the network to be drawn on another display. Another implementation might convert the method calls to PostScript and send them to a printer. To our widgets it does not matter. We just make calls on the Graphics object to draw what we need.

Besides drawing, the Graphics object also contains the clipping region. At the top level, the windowing system creates a Graphics object for drawing on the screen and gives it the limits of the window as its clipping region. The windowing system will then remove any overlapping regions from the clipping region and call paint() with the new Graphics object. This ensures that the application gets redrawing and clipped appropriately. In many systems one can reduce the clipping region, but not expand it. For example in the calculator, when the Number View is to be drawn, the clip region is intersected with the rectangle for the Number View. If there is anything left then the Number View's paint() method is called with a Graphics object that has the smaller clip region suitable for Number View.

All implementations of a Graphics object have methods to draw lines, polygons, ellipses, curves. Most have the capability to draw more complex shapes. When learning a new system it is important to find whatever it is that they call the Graphics object and then review the methods provided to see what kind of drawing support is available.

Simple Styling

There are some styling issues that must be addressed at this point. We will cover styling in more detail later. When one draws a line, for example, there is generally a drawLine() method of some form that takes the coordinates of the two end points. This information alone is not sufficient. Lines that are infinitely thin (as the mathematics would indicate) are easy to draw but very hard to see. Lines have other properties or attributes such as the thickness, the color, any dot/dash pattern to the line and the shape of the ends (round, square, flush etc.).

If all of this information was included as parameters to the drawLine() method, the use of that method would be very tedius.

There are two techniques used for providing such styling information. The first is the current settings method. In this approach the Graphics object has methods to set color, line thickness, etc. When drawLine() is called the current settings of those styling values are used. This is the approach that Java and many other systems use. A second approach is the use of a Pen object. The Pen object contains all of the attributes necessary to draw a line. Then when drawLine() is called there is a fifth pen parameter that contains all the other information. This is a powerful technique because it allows us to define several styles of line drawing for our application and then use a particular one for a given line. This is the technique that C# and .Net applications use. The two approaches have equal power. It is important to check to see which approach is being used on a system you are trying to learn and then adapt appropriately.

Pens or other attributes are not confined to the drawLine() method. Anything that has a border, such as a rectangle, circle, polygon or other shape will use these same attributes when drawing. There are also attributes for the fill color, fill image or pattern, gradient fills and a variety of other controls. These also can be treated using the current settings approach (as in Java) or can be collected together into a Brush, Paint, or Fill object, depending on the system's naming. A Brush object contains all of the filling attributes and can be passed to any drawing method that fills in an area.

With filled shapes, such as circle or rectangle we frequently want a filled shape with a border. Many systems provide two methods for each shape such as drawCircle() which draws the circle as a stroke or fillCircle() that fills the circle shape. The get a filled circle with a border it is usual to first call fillCircle() and then drawCircle() with the same geometry.

The styling of how shapes are drawn is generally more complex that what is defined in the Graphic object. The drawing methods provide the raw material for our presentations. For an application to look good it must present a consistent style and for a good design process we need to be able to change that style across the whole application. For this we will need a more comprehensive styling mechanism that will be discussed in chapter xxxx.

Drawing representations

Before we move forward into the details of actual drawing we need to look at how drawings are represented. There are three basic forms: *pixels*, *strokes* and *regions*. Each of these has strengths and weaknesses and we will move back and forth among them in defining the presentations that we wish to draw.

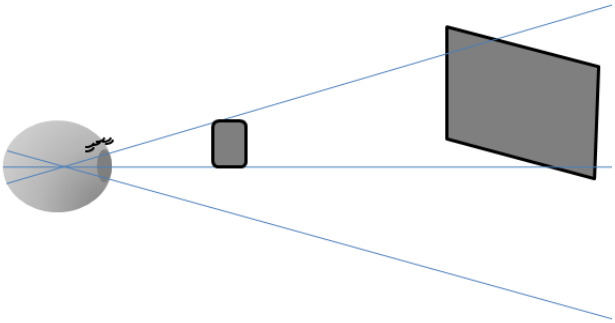
Pixels

The pixel representation is the most universal. Anything that human beings can see can be represented as pixels. The pixel representation is simply a two dimensional array of pixels where each pixel contains a color. There are some representations where each pixel contains only a gray value (varying from white to black) but they are increasingly rare. A gray-scale pixel image takes 1/3 the amount of space as a color image, but space no longer matters much and color pixels can easily represent gray if that is what is desired.

There are two key properties to a pixel representation: the resolution and the color depth. The resolution of a pixel representation is simply the number of pixels available. The more pixels we have the finer the details we can represent and the more space required. We frequently talk of pixel resolution in dpi (dots per inch). This a useful measure for displays and printers but does not actually tell us how well people can see a given pixel resolution.

Take for example a modern display screen that is about 24” across and with about 2,000 pixels horizontally. That give us 83 dpi for a desktop screen. A smartphone screen is about 4.5 inches in length with that

same 2,000 pixels across for a resolution of 444 dpi. Most laser printers have 300-600 dpi resolution. For interaction, however, the surface resolution is not as important as the angular resolution.



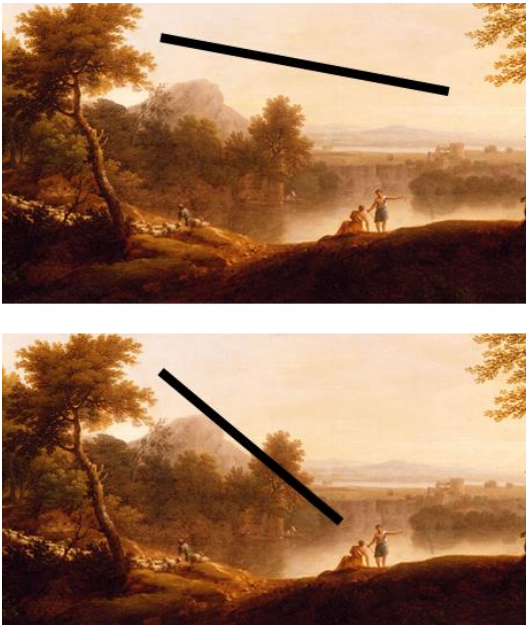
3-4 – Angular resolution of the eye

Figure 3-4 shows how the eye sees the smart phone and the screen. To understand how well the eye can see, we need to divide the dpi measure by the viewing distance. Desktop display screens are viewed from about 25” away. Actual distance varies widely. Smartphones are viewed from 8-12” away. Once we divide by distance we see that the smartphone appears about twice as sharp to the user as the display screen even though they both have the same number of pixels. A similar analysis of projected screens or televisions across the room shows that fewer pixels are needed in many such situations because of the distance.

The color depth or *dynamic range* of an image is controlled by two things: the sensing capability of the retina and the opening of the eye’s iris. The sensors in the retina are capable of resolving about 64 levels of intensity (6 bits). However, the iris can open or close letting in more or less light. However, for interactive work we do not want the iris opening and closing. We want it at a steady setting so as to reduce fatigue. Therefore 64 levels for each of red, green, blue. We actually use 256 levels because that fits in a byte and simplifies computing.

Strokes

Pixels can represent anything that the eye can see, but they are not very flexible to manipulate interactively. In figure 3-5 we see a line on the top that we want to change to the line on the bottom.



3-5 – Moving a line

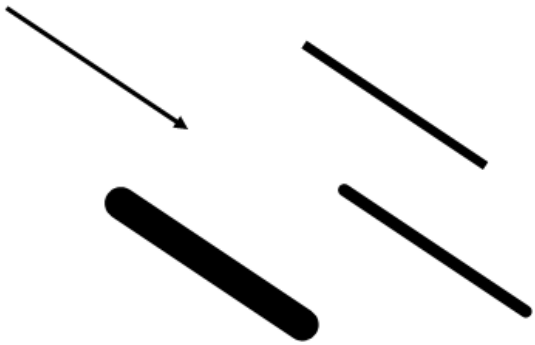
If we attempt to move this line in a pixel representation, it is extremely difficult. First we need to identify all of the pixels that belong to the line and transform those pixels to the new position. In addition, we must recover or replace the background pixels that formerly were covered by the line. This is all possible but computationally and interactively painful.

An alternative representation is to store the image separately in pixel format and represent the line by its two end-points. We interactively move the line by changing the values for one of its end-points. This also matches one of our most common interaction techniques, which is to move a control point from one position to another. Representing a line by its end-points uses a stroke representation. Rectangles, arcs, circles,

ellipses and curves all have geometric stroke representations which are easy to manipulate. Once we have changed the representation, we redraw everything again (image then line) and the result is a new picture that was interactively easy to change. Note that we had to redraw everything after the change. For actual drawing on a screen, printer or projector we eventually must convert stroke representations back to pixel representations for display. The stroke to pixel conversion happens when we make calls to methods in the Graphics object.

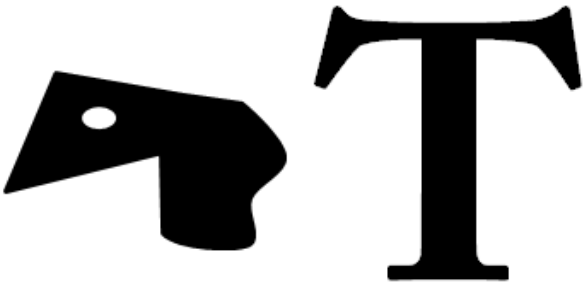
Regions

Strokes also have their limitations. Mathematically a line is infinitely thin. We do not actually want an infinitely thin line. We want lines of many thicknesses. Once we start drawing lines in many thicknesses we must think about what the shape of the endpoints of the line should be. Figure 3-6 shows several possible line shapes including one with an arrow head.



3-6 – Line shapes

For each of the lines in figure 3-6 the geometry is specified by two end points, but the shapes differ. We need a representation that captures this difference without resorting to pixels that are hard to manipulate. There are also other shapes that must be captured such as in figure 3-7.



3-7 – Region shapes

All of these shapes can be represented by describing the border of the shape and then filling it. There are a variety of such border representations but the most common are quadratic or cubic curves. By fully describing the border, the round ends of the fat line in figure 3-6 and the little fillets on the shape of the T can all be correctly represented. Regions (using a border description) also take less space and less internet resource than pixel representations. Regions are not quite as easy to manipulate as strokes, but are easier than pixels. Virtually all laser printers use region representations.

In most cases, stroke representations are converted to region representations so as to get the shape right. Region representations are then converted to pixels for printing or display. All of this happens inside of the Graphics object. For purposes of interaction, we do not really care how the graphics experts perform the necessary conversions.

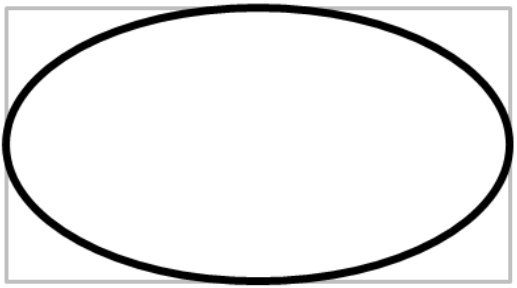
Simple shapes

Almost all graphics systems are capable of drawing lines, rectangles, ellipses, arcs, polylines, polygons and curves. Line geometry consists of two endpoints.

Rectangles are always represented aligned with the X and Y axes. The common rectangle representations are top, left, bottom, right or top,

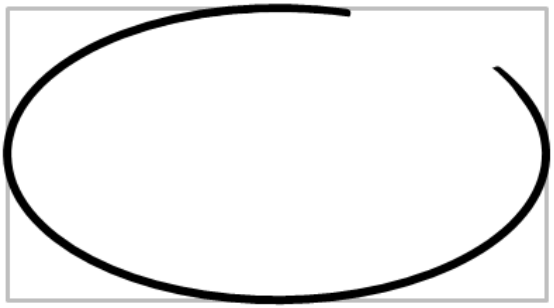
left, width and height. There are other combinations, but they are less commonly used.

Ellipses are also drawn relative to the X and Y axes. The primary reason for this is that it is the most common case and it has the simplest interaction. The ellipse, as shown in figure 3-8 is specified by its bounding rectangle. Circles are a special case of an ellipse where the bounding rectangle is a square. Some systems provide a circle method, but most just provide the more general ellipse.



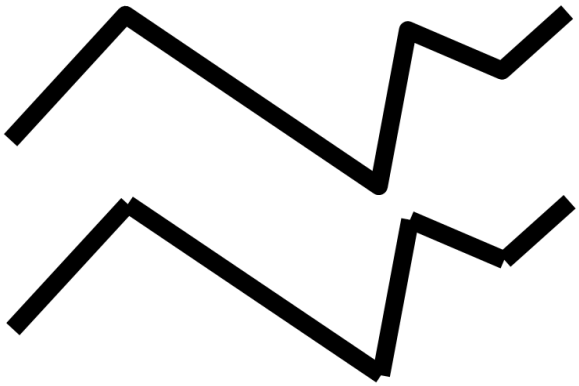
3-8 – Ellipse defined by its rectangle

A portion of an ellipse can be drawn to create an arc. Arcs are specified starting with the definition of the ellipse and then specifying a starting angle and an ending angle. Angles usually start from the right edge and go counter clockwise. The most common case is to use radians. As shown in 3-9 the ending angle may continue on and wrap around.



3-9 Elliptical arcs

A polyline is just a sequence of line segments. However, the lines are grouped together because it takes fewer points (two adjacent segments share the same end point.) Polylines are also grouped together because it gives the drawing system the ability to cleanly join the line segments into a single shape. Figure 3-10 shows the differences between using line segments and a polyline.



3-10 – Line segment and polyline differences

Polygons are simply closed versions of a polyline and are represented by a sequence of points.

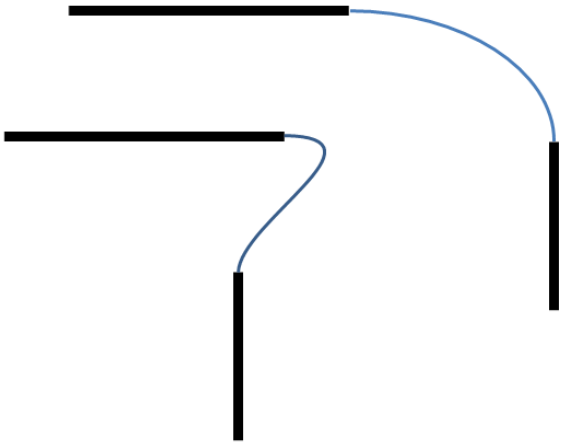
Curves

Creating arbitrary curved shapes like the ones in figure 3-7 requires a little more work. Mathematically there are no equations that capture all of the shapes that we might want to create. Instead we define shapes by piecing together simpler curves to create shapes with as much complexity as we want.

When we piece curves together we want the pieces to match well. For this we talk about G(0) and G(1) continuity. G(0) continuity means that the zeroth derivatives are the same at the point where two curves connect. G(1) continuity means that the first derivatives of the two curves are proportional to each other at the point where they join. G(0) continuity simply means that the two curves touch exactly. G(1) means

that the joint is smooth because the tangent vectors of both curves at the point are parallel to each other.

In drawing systems we will always find cubic curves and frequently find quadratic curves. Quadratic curves are attractive because most geometric problems can be solved using the quadratic equation. That is very attractive. Cubic (degree 3) equations do not have such a simple solution. Quadratic curves do not have sufficient power to blend any curve together. Consider the lines in figure 3-11.

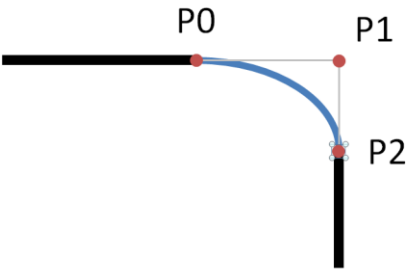


3-11 – Smoothly blending lines

In the upper right case, the two straight lines are smoothly blended by a quadratic curve. In the lower left case, there is no quadratic curve that can smoothly connect these lines. A cubic curve and smoothly blend any two lines in any position. That gives us a very powerful tool that is a little more computationally expensive. The lower left case in 3-11 could be solved using two quadratic curves pieced together. All of the curves needed for the region outline of the T in figure 3-7 could be quadratic curves. Another reason for using cubic curves is that they can more accurately approximate circles and arcs than quadratic curves. This will be useful in some of our later geometry problems.

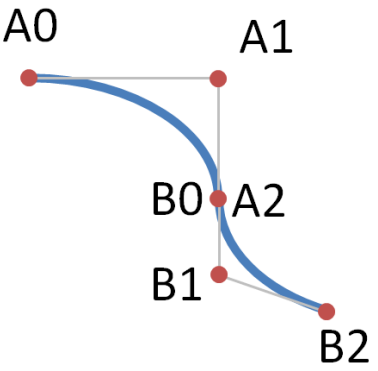
Bezier curves

The most common curve found in a drawing package is the Bezier curve. It comes in a quadratic and a cubic form. The quadratic curve has three control points P0, P1 and P2. The endpoints of the curve are at P0 and P2. The point P1 is consider the “bulge”point. Figure 3-12 shows how a quadratic Bezier curve is constructed to join two lines.



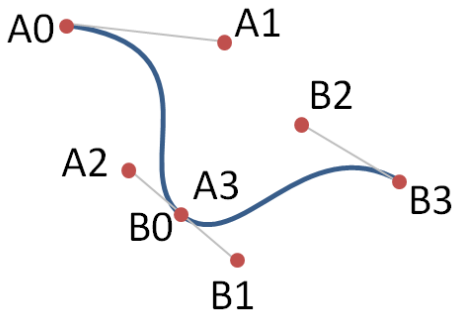
3-12 – Quadratic Bezier blending two lines

P0 is the starting point for the curve and P2 is the ending point. To get G(0) continuity we just make these points the same as the end points of the two lines being blended. The line P0-P1 is tangent to the curve at point P0 and the line P2-P1 is tangent to the curve at point P2. It is point P1 that controls the smooth G(1) continuity. If P1 is on both lines (at the intersection) then the tangents will be smooth at both P0 and P2. Figure 3-13 shows how we piece together two quadratic Bezier curves A and B.



To achieve G(0) continuity points A2 and B0 are the same. Curve B starts at the same point as curve A ends. To achieve G(1) continuity (make it smooth) points A1, A2, B0 and B1 must all be collinear.

A cubic Bezier curve is constructed similarly except that it has 4 control points rather than three. As with the quadratic curve the first and last points define the end points of the curve. With the cubic curve each end point has its own control point for the tangent rather than sharing a single point. Figure 3-14 shows how cubic curves A and B can be pieced together smoothly. The tangent of the curve at A0 is defined by the line A0-A1, but the tangent at A3 is defined by a different line A3-A2. To get G(1) continuity points A2, A3, B0, and B1 must be collinear.



3-14 – Blending two cubic Bezier curves.

Region shapes

With these two forms of curves we can build arbitrary shapes with the following 4 methods.

- moveTo(x,y)
- lineTo(x,y)
- quadraticTo(cx, cy, x, y)
- cubicTo(c1x, c1y, c2x, c2y, x,y)

The moveTo() method is usually for defining the starting point of a shape. It just moves the current point to be (x,y). The lineTo() methods uses the ending point of whatever came before and draws a line to the point (x,y). This pattern of using the last point of the previous primitive as the starting point for the next primitive is common. The quadraticTo() and cubicTo() methods draw quadratic and cubic curves. To draw the curve combination in figure 3-13 we would use the following code.

```
Graphics g;
g.moveTo(A0x,A0y);
g.quadraticTo(A1x,A1y,A2x,A2y);
g.quadraticTo(B1x,B1y,B2x,B2y);
```

By using the last point of the previous primitive no points every get repeated. Similarly the cubic curves in 3-14 would be drawn as

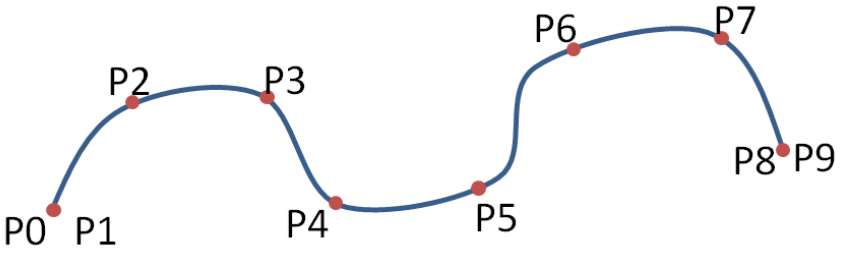
```
Graphics g;
g.moveTo(A0x,A0y);
g.cubicTo(A1x,A1y,A2x,A2y,A3x,A3y);
g.cubicTo(B1x,B1y,B2x,B2y,B3x,B3y);
```

If the very last point is the same as the starting point, then a closed shape is created that can be filled.

Notice that the left-hand shape in figure 3-7 has a hole in it. This is created by drawing the outside shape with lines and curves and then using another moveTo() to start the second inside shape. The filling rules for shapes will create the hole.

Catmull-Rom curves

There is another form of cubic curve that is interactively useful. The Bezier curve has two additional control points for each curve. Interactively this is a little awkward. It gives us lots of control over the curve but requires many more points. Suppose we want a curve where we only indicate points that are actually on the curve. For this we use the Catmull-Rom curve that has exactly the same mathematics as all other cubic curves but only uses points actually on the curve as its control points. Figure 3-15 shows how such curves are constructed.



3-15 – Catmull-Rom curves

As with the Bezier we build up a complex curve by piecing together several simple cubic curves. In the case in figure 3-15 the control points for each curve are:

```
P0, P1, P2, P3
P1, P2, P3, P4
P2, P3, P4, P5
P3, P4, P5, P6
P4, P5, P6, P7
P5, P6, P7, P8
P6, P7, P8, P9
```

Each curve still has four control points but they borrow two of their points from their neighbors. For example the curve between P4 and P5 has the control pointe P3 (from the previous curve), P4, P5 and P6 (from the next curve). The problem is at the ends where there is no previous curve or next curve. In those cases we just use the end point twice. P0 and P1 are the same point. If we want to create a closed shape out of control points as in figure 3-16 we simply wrap around.

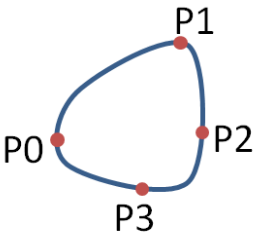


Figure 3-16 – Closed shapes with Catmull-Rom curves

In this shape there are 4 curves that have the following control points.

```
P3, P0, P1, P2
P0, P1, P2, P3
P1, P2, P3, P0
P2, P3, P0, P1
```

One of the nice features of the way Catmull-Rom curves borrow control points from their neighbors is that we get G(0) and G(1) continuity automatically. This also simplifies our interaction with the shape.

Summary

Drawing is generally done through a windowing system. It is the windowing system's responsibility to manage limited screen space among many processes and interactive units. A key mechanism that the windowing system uses is clipping, which forbids a piece of an application from drawing outside the bounds it has been given.

Generally windows are arranged in trees. Each widget is its own window and is part of a parent window. This decomposition is generally based on rectangles and allows each interactive unit to operate as if it is its own pixel space.

Windowing systems need to get windows redrawn on demand based on system needs. The is handled by implementing a paint() method on every component/widget. The paint() method receives a Graphics object that forms the interface between the interactive software and the underlying technology.

Drawings are represented as pixels, strokes or regions. Pixels can represent anything and are the ultimate representation for screens and printers. Pixels, however, do not interact very well. Stroke-based representations represent the geometry of the shape and then redraw the shape as needed to convert to pixels. Strokes are easy to interact with by just changing their geometry points. Strokes do not accurately represent their own shapes. Regions representations use lines and curve to define the border of a shape, which is then filled. Strokes are

converted to regions and regions to pixels as part of the drawing process.

Our simple shapes are lines, rectangles, ellipses, arcs, and curves. Lines and curves can be assembled into the borders for filled shapes.