2

Drawing

When designing a new interactive application the place to begin is with the model. However, students of interactive software generally insist on getting something up on the screen as soon as possible. Therefore we will assume that our model has been designed and implemented and it is now time to create the view. An expanded interactive architecture is shown in figure 2.1. Only on the most primitive of devices will the view directly access the screen. Generally this access is provided by the windowing system. The windowing system handles both the drawing from the view and the input to the controller. The input issues as well as their relationship between the View and the Controller will be discussed in chapter 3.

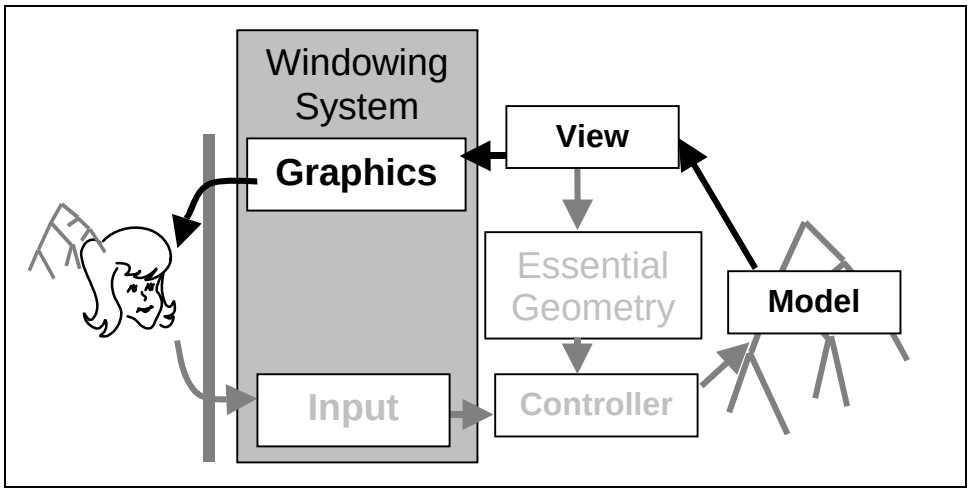

Figure 2.1 – Graphical presentation architecture

The windowing system provides a View with several services. The most obvious, is a list of overlapping rectangles called windows, like those in figure 2.2. This list is maintained in a back-to-front ordering so that the top-most windows always obscure any windows that they overlap. Of importance to the View implementation is that the windowing system presents a rectangular region on which the view can draw and insulates the View from all knowledge of the

existence of any other application or window. This greatly simplifies the View's drawing problem. In figure 2.2 the Calculator assumes that it has a full rectangle in which to draw and the windowing system ensures that the Windows Journal window in front is not damaged by the Calculator drawing its lower left corner.

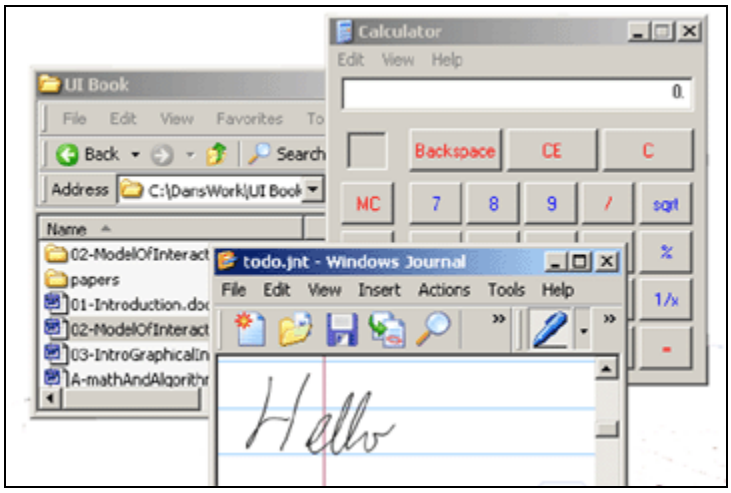

Figure 2.2 – Windowing system with overlap

Most windowing systems actually provide a tree of windows so that the interaction problem can be hierarchically decomposed into pieces. This simplifies both the visual layout and the software architecture. Figure 2.3 shows a simple paint application in a window. This window contains sub-windows for the menu, tools, colors and paint area. Each of these is further decomposed into other rectangular regions.

Each of these rectangular window regions has an associated widget that is responsible for all of the interaction within that rectangle. The widget is our basic unit of interaction. Part of the widget implements the View. One of the annoying characteristics of user interface software is that there is no uniformity of terms. In some systems widgets are called controls (Microsoft) and in others they are components (Java). In some cases the entire widget is called a view even though such views also include controller code. Widget is one of the earliest terms and the one we will use in this book. Whenever one is learning a new user interface system it is very important to find the names for common concepts. The important point here is that our view is provided with a rectangular drawing region that is independent of all other such regions.

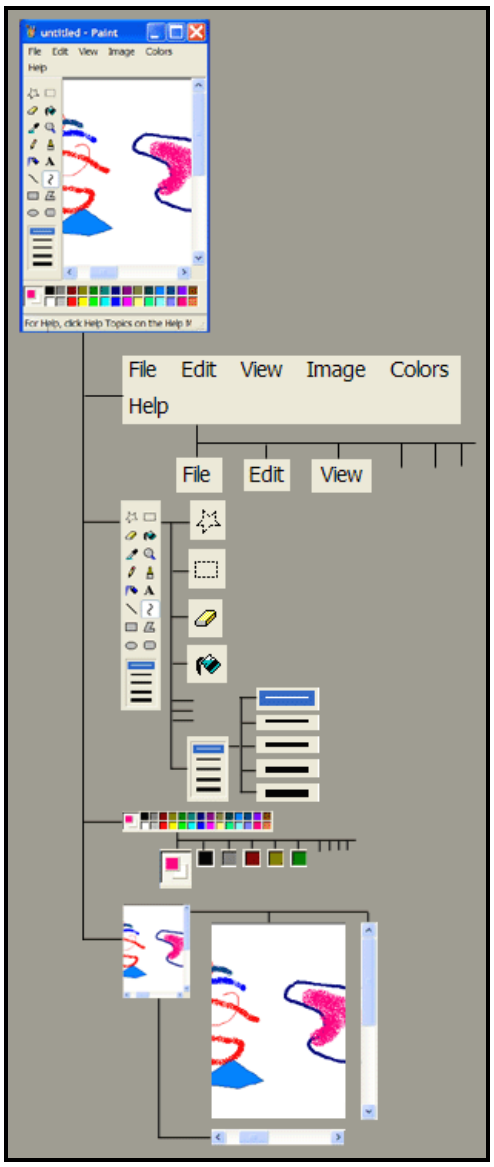

Figure 2.3 – Window tree

Redraw

Introduction of the overlapping window model for user interfaces created a problem like that shown in figure 2.4. On the left the address book window overlaps much of the pinball window. On the right the address book window has been moved and the outlined section of the pinball window has been revealed. The problem is that this outlined area must be redrawn and only the pinball application knows how to redraw it. However, nothing in the pinball application has caused this condition. Such redraw requests occur when window sizes are increased, overlapping windows are closed, or when a window is brought to the front and a variety of others. Redrawing the view also arises when the model changes.

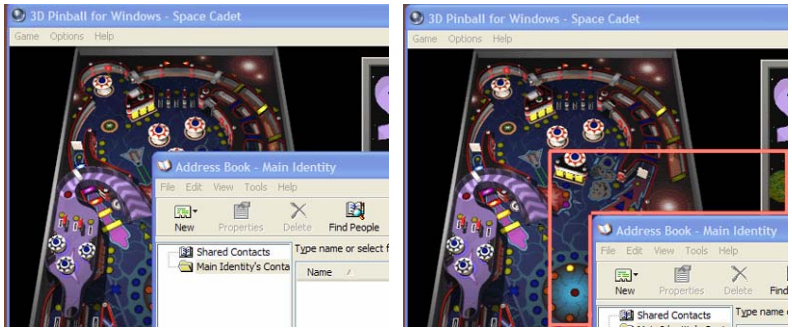

Figure 2.4 – Revealing hidden window space

Because there are so many reasons why a view must redraw itself a general architectural solution is required. In all modern graphical user interfaces each widget must implement a redraw(Graphics G) method. The windowing system is responsible for deciding when some region of a window needs to be redrawn and it will call the redraw method each time. The view then only draws when explicitly requested by the windowing system. No other part of the architecture should perform drawing. Every time redraw() is called it should completely draw the view from the current state of the model. The view has no idea which of the many possible reasons caused the redraw request. In a graphical application the view consist entirely of the redraw() method and any other methods that it may call. As with other user interface concepts, redraw goes by many names including paint, onPaint, update, draw or refresh. Every widget class has such a method.

A line drawing application demonstrates redraw. The model is an array of lines. For each line we can access the end points, color and thickness. The view class must have a pointer to the model so that it can obtain needed information.

Figure 2.5 shows the model and view classes for our example. Because there is a widget associated with every window, the windowing system can get the view redrawn whenever necessary and the view will always draw the current state of the model.

```
class DrawModel
{
      int getNumberOfLines() { . . . }
      Point getEndPoint1(int lineIndex) { . . . }
      Point getEndPoint2(int lineIndex) { . . . }
      . . . other model methods . . .
}

class DrawView extends Widget
{
      DrawModel myModel;
      public void redraw(Graphics G)
      {
            for (int i=0;i<myModel.getNumberOfLines(); i++)
            {
                  G.drawLine(myModel.getEndPoint1(i),
                        myModel.getEndPoint2(i));
            }
      }
}
```

Figure 2.5 – Redraw method

Graphics Object

When redraw is called it is passed a Graphics object. This is sometimes called a device interface, a device context, a canvas or a sheet. They are all the same thing, which is a rectangular area on which the redraw method can paint the data that it needs. The Graphics object provides an abstract interface to the underlying drawing facility. This insulates applications from knowledge of other windows, screen controller hardware or graphics accelerator cards. The Graphics object defines a set of methods that views can use to draw. The underlying windowing system will take care of all of the interface issues.

The Graphics object architecture provides great software flexibility. The same redraw code can draw to a window on the screen, into an image to be saved, to a special file format from which the drawing can be recreated, to a printer or over the network to another screen. The redraw() code does not care; it just uses the interface provided. The Graphics object also provides the clipping of drawings to prevent damaging overlapping windows.

Due to the history of drawing hardware based on television screen controllers, the coordinate systems for Graphics objects always have the origin in the upper left hand corner with the positive X axis going to the right and the positive Y axis pointing down. The Graphics object always has a *clipping region*. This is the region of the coordinate system for which drawing will actually occur. In figure 2.4 the clipping region is the outlined shape in the right hand image. By controlling the clipping region the windowing system keeps all views "playing nice" in their own space.

```
public void redraw(Graphics G)
{
    Region clip=G.getClipRegion();
    for (int I=0;I<myModel.getNumberOfLines(); I++)
    {
        Point E1=myModel.getEndPoint1(i);
        Point E2=myModel.getEndPoint2(i);
        if (! clip.intersects(new Rectangle(E1,E2) )
        {
            G.drawLine(E1,E2);
        }
    }
}
```

Figure 2.6 – Clipping region drawing optimization

The presence of the clipping region allows us to optimize our redraw code somewhat. Figure 2.6 shows an alternate implementation of the redraw method that avoids drawing lines that will be completely discarded due to the clipping region. In this implementation if the bounding rectangle of the end points does not intersect the clipping region the line will not be drawn. In this example the optimization is not much use because with current graphics accelerator cards drawing the line is almost as cheap as building the rectangle and computing the intersection. However, in some applications with very complex models the cost of drawing may well be worth such optimizations. This is particularly true when large sections of the drawing can be ignored following a cheap intersection check. The recommended practice, however, is to implement redraw as a simple complete drawing of the model and then introduce clip region optimization if drawing speed becomes an actual issue. In very many cases the full draw operation happens faster than the user can see, making optimization a waste of effort and a source of bugs in the code.

Light and Color

Humans perceive images by sensing light and color. Understanding this tells us how to generate images that appear acceptable to the human eye. The human

retina contains rods and cones. There are about 120 million rods in the eye and they are broadly sensitive to all colors with their best sensitivity being in the blue wavelengths. Because rods are broadly sensitive to most colors, they do not distinguish between colors. Rods provide us with our night vision and our peripheral vision. The cones are sensitive to narrower wavelength bands of light and there are only 6-7 million of them. There are three types of cones, red (64%), green (32%) and blue (2%). The blue cones are more sensitive than the others and their signal is amplified so that all three are sensed in roughly equal proportion. However, because there are very few blue cones, the precision with which we see in blue is much lower[1].

Each type of cone senses a range of light wavelengths around their strongest value. This allows us to see a variety of colors other than red, green and blue. For example the wavelength of red light is about 650 nanometers. The wavelength of green is about 510 nanometers. Both of these are sensed by their respective types of cones. Yellow light has a wavelength of about 570 nanometers or close to half way between red and green. We do not have cones that sense yellow directly. However, because yellow is close to red, the red cones will fire and because yellow is close to green the green cones will fire. We sense yellow as a roughly equal firing of red and green.

Color blindness occurs when one of the types of cones is defective. Loosing the red cones for example does not mean that one cannot see red but rather that one cannot distinguish red from green or from blue. The red is being sensed by the rods and the other cones but not in a way that can distinguish the red.

Human vision also has a limited dynamic range. That is, the rods and cones can only distinguish different levels of color within a limited range of intensities. If the light is too bright then the sensors all fire at maximum value and we just see blinding whiteness. If the light is too dark then we just see blackness. The eye compensates for this to some extent by using its iris to control the amount of light that enters the eye. This increases the range of brightness that we can see, but for a given iris setting the dynamic range has only so much precision. To prevent eye fatigue we want a brightness range that does not force frequent iris adjustments. Most graphics screens only support such a range for both ease of use and cost containment.

RGB

All computer displays and all televisions use the RGB or red, green, blue color system for displaying light. Since the human eye can only distinguish red, green and blue we can deceive the retina into reporting any color that we want by

appropriately mixing red, green and blue light. The approach in all display systems is to generate spots of red, green and blue light in a region small enough so that they are not seen as separate spots. We call this red, green, blue triple a *pixel* or picture cell. We call the density of RGB pixels the *spatial resolution* of a display. It generally is expressed as dots per inch (DPI) or in dots per centimeter.

The dynamic range of the eye also has an impact on representing colors. The retina cannot distinguish more than 64 distinct levels of intensity. This is less for some types of cones. For a given iris diameter, only 6 bits are needed to represent all visible light intensities. Because the iris can rapidly adapt within small ranges and for simplicity in computing hardware we typically allocate 8 bits (256 levels), which is more than enough to represent all of the colors that can be seen. However, we must represent intensities for red, green and blue for a total of 3 bytes or 24 bits for each visible color.

There are three ways for expressing RGB colors. Display hardware always represents RGB colors as three (generally 8 bit) integers. Thus red, green and blue each range from 0 to 255. In some cases red, green and blue are represented as real numbers between 0.0 and 1.0. This representation is independent of the number of bits. This is particularly useful when some color representations cheat on the number of bits allocated for blue, since humans cannot distinguish levels of blue as easily as other colors. A third color representation uses percentages ranging from 0% to 100% for each primary color.

HSB

The RGB color system is not very convenient for humans to use in selecting desired colors. People generally do not think about how much red, green or blue they want. They thinking in terms of light colors, yellow colors, pastel colors etc. The HSB or hue, saturation, and brightness system provides a more human-friendly way to express colors. The HSB values are then converted to RGB to control computer displays. The hue corresponds to the primary wavelength or color of the light to be generated. It is expressed as degrees around a color wheel as shown in figure 2.7. Yellow, for example would have a hue of 60 or half way between red and green. Orange would be about 30 or half way between red and yellow. Hue is generally expressed as degrees, but sometimes as a percentage of the way around the color wheel and less frequently as a 0-255 number that is linearly mapped to the distance around the color wheel.

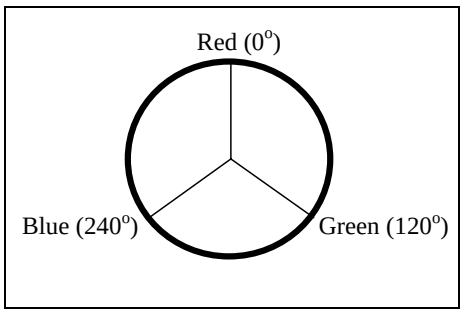

Figure 2.7 – Hue color wheel

Saturation is generally the most confusing to new users of the HSB system. Saturation is the amount of color being displayed as opposed to the amount of gray or white. For example the color red has very high saturation; there is no green or blue light involved. The color pink has the same hue as red, but it has lots of green and blue mixed in to give it a more whitish color. Red has high saturation while pink has low saturation. Pastel colors have low saturation. Grayscale images like those produced by ordinary laser printers have zero saturation. There is no color. Saturation is generally represented using 0-1.0, 0%-100% or as an integer 0-255.

Brightness is the amount of light. Grayscale images are composed exclusively of variations in brightness. Brown, for example, is a low brightness version of orange. Navy blue is a low brightness version of blue. As with saturation, brightness is generally represented using 0-1.0, 0%-100% or as an integer 0-255.

Using HSB we independently select the desired hue such as purple (hue=300 or half way between blue and red). We can control the brightness to get a bright purple or a deep dark purple. We can also control the saturation to get an intense, highly saturated royal purple or a pale pastel purple with low saturation.

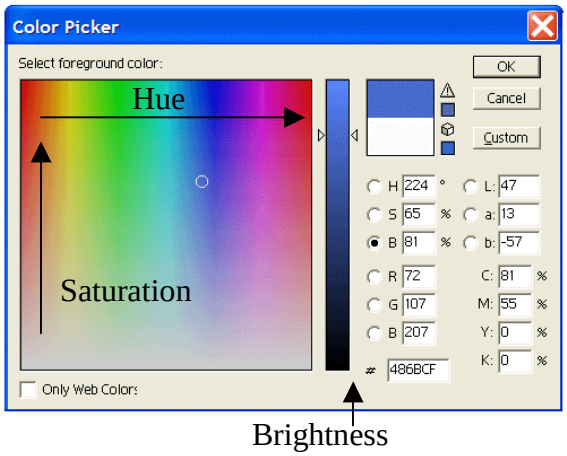

Figure 2.8 – Adobe Photoshop color picker

Conversion to RGB

Conversions from HSB to RGB are generally hard for people to do accurately. However, many systems do not support HSB directly and one must frequently specify RGB values. The best way to do this is to get close to the desired color and then adjust the RGB values to get the desired result; much like one adjusts the hot and cold water faucets to get to the right temperature and flow.

Assume that we want the color of a pumpkin. The hue of a pumpkin is orange (30). A pumpkin color has medium-high saturation (70%) and about medium brightness (50%). The brightness will vary a lot depending on where we look on the pumpkin. Shadows and shading are primarily controlled using brightness or the total amount of light. The hue of our pumpkin is closest to red so we will set red (our primary color) to be equal to the brightness (red=50%). Our saturation is 70% so we will set blue and green to 70% of the brightness, which would be 35% (R=50%, G=35%, B=35%). However, this gives us a red hue rather than an orange hue. To get to more orange (between red and green but mostly red) we raise green to partway between red and the saturation value (R=50%, G=40%, B=35%). If we wanted yellow we would make red and green equal (R=50%, G=50%, B=35%). In essence the approach is to make the dominant color (in this case red) equal to the brightness, make the least dominant color equal to the saturation (based on the brightness) and then adjust the intermediate color to get the right hue.

This is still difficult for most people to do accurately. Therefore all modern graphical tool kits provide a color picking widget. The widget for Adobe Photoshop is shown in figure 2.8. The easiest way to select colors is to spread them out on a surface and have the user select the desired color. Unfortunately display surfaces are 2D and the color space is 3D. In figure 2.8, the color choices are laid out with the X-axis being the hue and the Y-axis being saturation. The bar just to the right of the color space controls the brightness. The user can select a desired brightness on this bar and then all possible hues and saturations for that brightness. This particular color picker widget allows users to work in HSB or in RGB. Many only support RGB. However, this is less important because the user primarily is selecting the color that looks right, regardless of the RGB values.

The algorithms for converting from HSB to RGB and back again are found in appendix A4.

CMYK

The RGB system is based on producing colors by mixing light. However, when printing or painting we produce colors by mixing pigment. Pigments work by absorbing some wavelengths of light. Thus their color is determined not by the light that they generate but by the light that they take away. RGB are the additive primaries. When adding them together we get white (all colors). The subtractive primaries cyan, magenta, yellow (CMY) work by absorbing or taking away light. Cyan for example is white light with all of the red removed. Magenta removes all of the green and yellow removes all of the blue. If we mix them all together we get black because all light has been absorbed. However, most pigments are not exact in the colors that they absorb and frequently do not completely absorb the light that we want. When mixing most CMY pigments we get muddy dark gray, brown or green. To get really sharp images we need very distinct blacks. For that reason most printing systems introduce a special black pigment to mitigate the pigment mixing problems. Thus we have the CMYK or Cyan, Magenta, Yellow, blacK color system. This is the system used by most printers. One can see this color system in the lower right corner of figure 2.8.

Generally users do not work in CMYK. However, since much of what we do is printed we should be aware of this complementary color system. Most good visual designers are trained to some extent in the visual arts. People who work in paint and print media are trained using the subtractive primaries appropriate to their medium. Understanding the complementary nature of CMY and RGB is important when communicating with these communities.

Transparency

When drawing on a screen we always draw in back to front order. By drawing in this order anything in front overlays or obscures whatever is behind. The result is consistent with how we see things in the real world. Sometimes, however, what is in front is partially transparent. In such cases what we see is light from the object in front as well as some of the light from objects behind. There are many situations where transparency is a helpful technique. Many graphics toolkits allow control of the transparency or opacity. Transparency and opacity are complements of each other. Some systems define their controls in terms of transparency and some in terms of opacity.

Transparency is the fraction of the light that should show through from behind the object being drawn. Transparency of 0 shows none of what is behind. The front object is completely opaque. Transparency of 1.0 or 100% means that the object being drawn is completely transparent and all of the light from behind shows through. Opacity is 1.0 minus transparency or 100% minus transparency.

To compute transparent color for a particular pixel we have **B** or the color of that pixel in the background, **O** or the color of the pixel for the object being drawn and **T**, which is the transparency of the object being drawn. Based on these three values we can compute a new pixel color **N=B*T+O*(1.0-T)**. Using this formula, if **T** is 0.0 there will be no background color, only object color. If **T** is 1.0 then there will be no object color. A transparency of 0.5 will mix object and background colors equally.

Many systems specify color as RGBA where A is for alpha channel. The alpha channel is the opacity information. An alpha value of 0 is completely transparent and a value of 1 (or sometimes 255) is completely opaque.

Drawing Models

The basic drawing model for all images is a rectangular array of pixels with each pixel having a color. This is how display screens, printers and image files are all structured. All drawing models must eventually convert their representations into pixel colors. There are three common ways to represent drawings. They are pixels, strokes or regions. Many systems and applications move back and forth among these representations depending upon the need.

Pixels

When displaying on a screen there is always a *frame buffer*. This is a piece of memory that contains a rectangular array of pixel values. These pixel values are

use to determine the color of each spot on the screen. Changing the pixel values changes the screen. On good displays this change occurs within $1/60^{th}$ of a second. Slower update rates introduce flicker in the display and cause eye fatigue.

There are two basic forms for frame buffers. A full color frame buffer stores 24 bits (one byte each for red, green, blue) for each pixel. This allows colors to be defined independently for each pixel. This provides for the most realistic images, but is more expensive in terms of space and time to update the frame buffer. Some frame buffers use *indexed color*. In an indexed color representation each pixel has 8 bits (or some times fewer). The pixel value is not a color but an index into a table of colors (see figure 2.9). The *color lookup table* is an array of 256 entries, each of which has 24 bits of color information stored. Indexed color representations take 1/3 as many bytes to store (1 byte per pixel rather than 3) and are three times faster in copying material from one part of the buffer to another, as when a window is dragged across the screen. However, indexed color representations can only have 256 colors visible at a time. The GIF image format uses indexed color in its representation. For many applications this is fine, however, for realistic images this is insufficient. There are some indexed color schemes that go beyond 256 elements in the table.

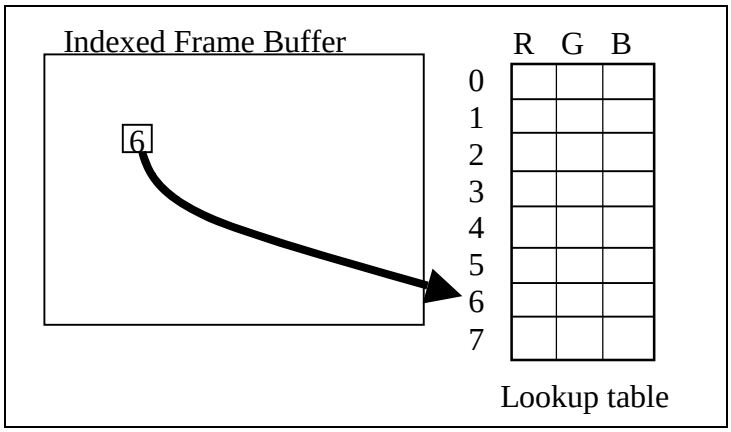

Figure 2.9 – Indexed-color images

The quality of a drawn image is determined both by its *spatial resolution* and its *color resolution*. Spatial resolution is measured in the number of dots per inch or pixels per centimeter. The higher the spatial resolution, the more crisp and smooth the image looks. However, there are limits to useful spatial resolution. At some point the spatial resolution of the retina limits the perceived spatial

resolution of an image. Remember that the relationship between image spatial resolution and retina resolution depends upon the distance between the image and the eye. An image that is half as far away must have twice the spatial resolution to appear to have the same quality. Perceived color resolution is not dependent upon distance.

Various systems exploit the spatial/color tradeoff. This is particularly true when printing. Printers generally have the ability to place ink on a spot or not. They generally do not have the ability to vary the intensity or transparency of the ink. However, it is much easier for a printer to print finer spots than the eye can see. Printers use this ability to provide the perception of color resolution. By printing spots of various sizes they control how much ink is shown vs. how much background paper. Computer screens generally have limited spatial resolution both because of display technology and memory limitations. However, such screens easily produce all of the colors that a human can see. Such screens can compensate for lack of spatial resolution by using shades of color to produce more smoothly appearing shapes. This is one of the approaches used by Microsoft's ClearType technology to produce more readable text without the cost of higher resolution hardware.

Stroke Drawing Models

All drawing on a computer is a process of determining the color for each of the pixels in a displayed or printed image. However, with the exception of pointillist painters such as Georges Seurat, most of us do not want to draw one pixel at a time. Much of the drawing that we do on a computer is composed of simple geometric shapes or strokes, as in figure 2.10. These are lines, circles, ellipses and curves. These shapes have the advantage of being simple to specify and somewhat resolution independent. A line has four parameters (X1,Y1, X2,Y2), which are the coordinates of its end points. A circle has a center and a radius. Ellipses have bounding rectangles. Text string drawing is also generally included in the stroke model. Most drawing interfaces provided by graphical toolkits are based on the stroke model. In addition to the geometry of strokes there are other properties that must be specified such as color, width and patterns such as dashes, dots or cross-hatching. The Graphics object is predominantly stroke-based. The hardware/software implementation under the Graphics object handles the conversion from strokes to pixels.

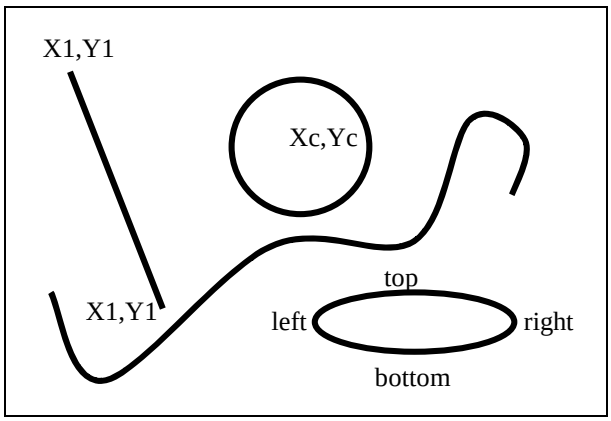

Figure 2.10 – Stroke drawing

Region Models

Strokes, however, are a geometric myth. A theoretic line is not useful to us because it is infinitely thin. We need a line that has thickness so that we can see it and we need to vary that thickness to create various weights of lines. The reality is that all strokes actually define regions of pixels. It is convenient to think about a line being one pixel wide but this causes many problems. The first is when printing. A line one pixel wide may appear well on the screen, but when printed one pixel wide on a 600 dots per inch printer will appear very faint and thin. As shown in figure 2.11, a line is actually a polygonal region. Representing the line as a polygonal region explicitly accounts for its width. The region-defined line is now resolution independent. If a device has more dots per inch, then more pixels will lie inside of the line's region and the edges of the line will be smoother. The same occurs for all of the other stroke representations. They are most accurately represented by converting them to regions. However, we retain the stroke model because specifying the end points and width of a line is much simpler than specifying its polygonal region. We leave it up to the underlying graphics software to make the conversion from stroke representations to regions.

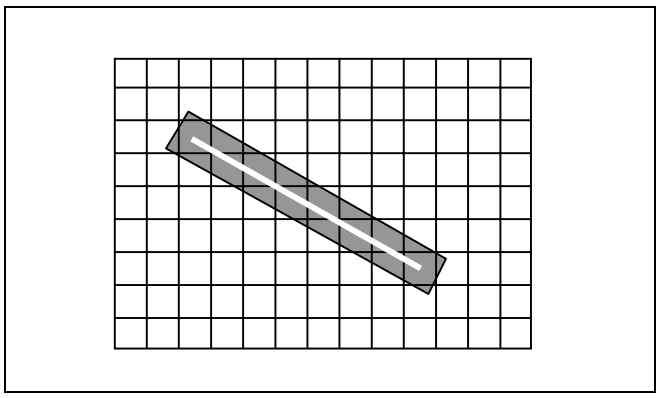

Figure 2.11 – Region representation of a line

The limitations of the stroke model became very clear when the first laser printers were produced. The very high resolutions of laser printers made simple strokes appear ugly. Similar problems occurred with text. To accommodate this, PostScript[2], which is the language of many laser printers, uses a region model for all of its drawing. PostScript also supports drawing images directly using the pixel model.

Regions are represented by their borders, and in most systems the borders are connected segments of straight lines and cubic (polynomials of degree 3) curves. We will discuss the geometry of cubic curves more in chapter 12. As shown in figure 2.12 a wide variety of shapes including circles, text characters and blobs can be represented in this form.

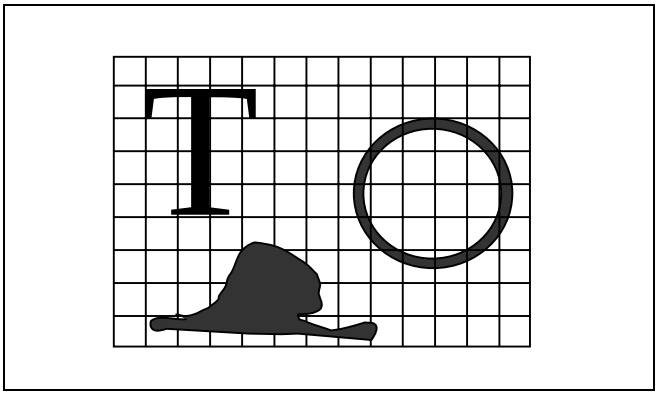

Figure 2.12 – Region shapes

All region models must eventually be reduced to pixels. This gives rise to "the jaggies" or *aliasing*. If we take our line region from figure 2.11 and convert it to pixels by coloring all pixels that are more than 50% covered by the line region, "the jaggies" appear as in figure 2.13 because pixels do not match up cleanly with region boundaries. The term aliasing comes from signal processing where signals that are sampled at too low of a resolution take on the appearance of completely different signals. In our case the signal is a shape and it starts to look funny.

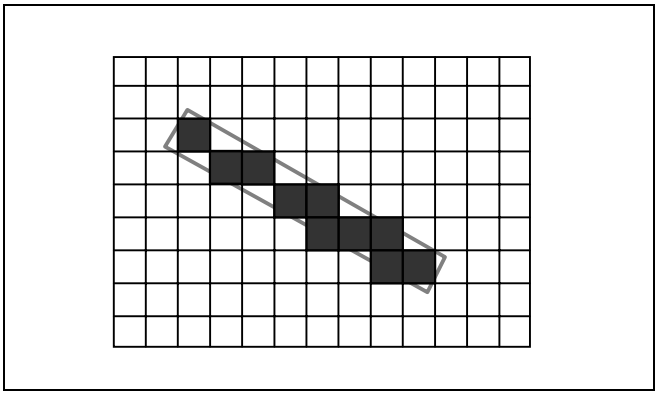

Figure 2.13 – Aliasing of a region

The ultimate solution is to use more pixels so that the jagged edges become so fine that the eye can no longer distinguish them. This is the approach that quality printers use. Unless a person gets very close or uses a magnifying glass the edges appear smooth because the spatial resolution is higher than the

perceived resolution. The alternative solution used in many on-screen drawing applications is called *antialiasing*. It works by using shades of gray or shades of color to approximate the partially included pixels. The amount of color is modified according to how much of the pixel is covered by the region. We use the percent of pixel coverage as an opacity value. The result is as shown in figure 2.14. At a distance the shaded pixels make the edges appear smoother. The antialiased version of the line looks straighter and more like a line rather than a blobby bunch of boxes even though the spatial resolution is the same for both.

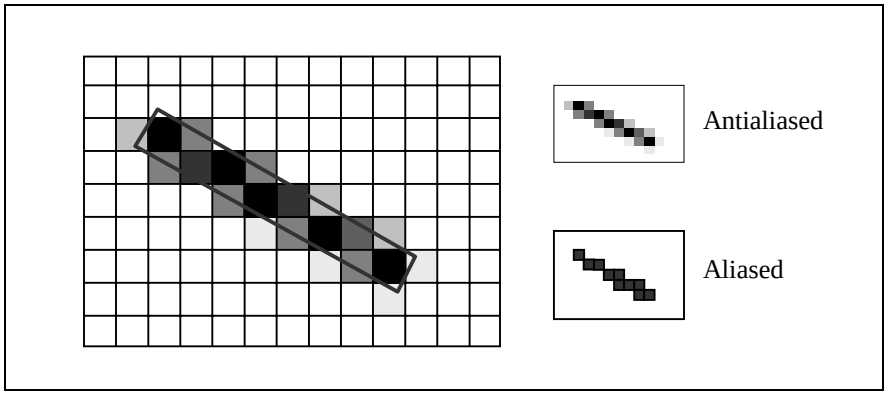

Figure 2.14 – Antialiased regions

Draw Methods

Using pixels, strokes or regions a Graphics object must provide a set of methods that the view can use for drawing. Most Graphics class designs separate their drawing methods into path drawing, region drawing, image drawing and text.

Paths

The simplest path drawing is the line. Lines have two end points (X1,Y1) and (X2,Y2). However, there are three basic styles for drawing lines. Older systems such as the Microsoft Foundation Classes use two calls to draw a line: MoveTo(X1,Y1) and LineTo(X2,Y2). This model is based on the old pen plotter interfaces where a MoveTo() would move the plotter head to a new location with the pen up and a LineTo() would move the plotter head with the pen down. Newer APIs, such as Java and C#, use a single drawLine(X1,Y1, X2,Y2) method. A third approach is to provide a super class of data objects called shapes. There is then a single draw() method that takes a shape as its only argument. Shapes contain all of

the geometric information for drawing. The purpose of the shape approach is to provide a simple way for saving geometry information in the model of a drawing application.

There is more to drawing a line than its geometry. There are properties of the line such as its color, thickness, transparency, style (such as dashed or dotted) and a variety of others. One approach would be to add all of these parameters to the drawLine() method. However, this would be very painful to use because we usually draw many lines in the same style. There are two ways in which various graphics APIs specify drawing properties.

In the *current settings* approach the Graphics object stores attributes for various drawing settings. Such graphics objects provide set and get methods such as setColor() and getColor(). When a line is drawn, only its geometry is specified as parameters to the method. The remaining information is drawn from the current settings of the Graphics object. Figure 2.15 shows how to draw red and blue lines using Java's Graphics object.

```
public void paint (Graphics g)
{
    g.setColor(Color.BLUE);
    g.drawLine(10,12,35,14);
    g.setColor(new Color(1.0,0.5,0.5)) // red with medium saturation
    g.drawLine(45,14,20,13);
}
```

Figure 2.15 – Drawing lines using current settings in Java/Graphics

An alternative approach to non-geometric information is the concept of a *pen*. A pen is an object that contains all of the non-geometric information about how to draw a path shape. All path drawing methods such as drawLine() have a pen object as a parameter. By packaging all of the information into a single object there is only one additional parameter. The code is also easier to read because it is clear from the parameters what is actually being drawn. Figure 2.16 shows an example of drawing lines in C# using the pen approach.

```
protected override void onPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
```

```
        Pen bluePen = new Pen(Color.Blue,1); // color and width
        g.drawLine(bluePen, 10, 12, 35, 14);
        Pen redPen = new Pen(Color.FromArgb(255,128,128),1); // color and width
        g.drawLIne(redPen, 45, 14, 20, 13);
}
```

Figure 2.16 – Drawing lines using pen objects in C#

In many cases one wants to draw a series of lines that are connected together. This can be done with multiple calls to drawLine() and in many cases this is perfectly acceptable. However, when one considers that lines with width greater than one pixel are actually regions, there are problems that arise when joining lines together. To deal with this problem most APIs provide a polyline drawing method that takes an array of points rather than just two and connects them all together smoothly. Figure 2.17 shows a pair of lines drawn independently and as a polyline. The polyline on top is joined smoothly because the software knows that they are joined when drawn in a single call. When drawn using separate drawLine() calls, the software does not know they should be joined and simply squares off the end of each line.

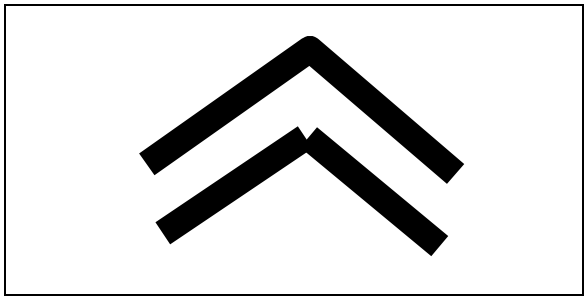

Figure 2.17 – Polylines vs. separate lines

Lines, of course are not the only path shapes that can be drawn. Virtually all graphics systems provide a drawRectangle() method. Rectangles are specified either by their bounds (top, left, bottom, right) or by the upper left corner and size (top, left, width, height).

Almost all systems will draw ellipses whose axes are aligned with the X and Y axes of the Graphics object. Ellipse geometry is specified by a bounding rectangle. Circles are drawn as a square ellipse. Most systems also provide for elliptical arcs or portions of an ellipse. Arc geometry is specified by providing the bounding rectangle of the ellipse plus the starting angle (generally in degrees) with zero being vertical and then specifying the angle of the arc. Positive arc

angles go clockwise from the start angle and negative angles go counter clockwise.

Most systems also provide a mechanism for drawing curves using cubic polynomials. However, cubic polynomials are a painful user interface for specifying curve geometry. Chapter 12 will provide a more extensive treatment of how curves are specified and drawn. The simplest way to specify a curve is to provide a list of points through which the curve must path. This can be thought of as a smooth polyline. The geometry information to draw such curves is identical to polylines. In addition, polylines can be closed to form polygons and curves can also be closed to form blobby paths as shown in figure 2.18. Closed paths implicitly connect their first and last point.

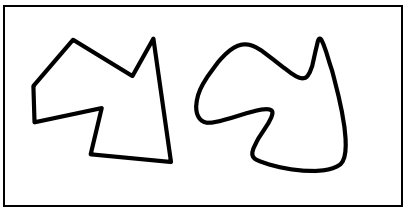

Figure 2.18 – Polygons and curves

Regions

Drawings are not simply made up of lines. The next most important class of drawing primitives is the filled region. Any closed shape such as a rectangle, ellipse, polygon, or closed curve can define a filled shape. For all of these shapes, most drawing systems provide draw methods and fill methods. For example drawRect() will draw the lines around the border of the rectangle. The corresponding fillRect() method will fill in the area of the rectangle. Figure 2.19 shows the code in C# to draw a variety of shapes. Figure 2.20 shows the results. Following C#'s approach to bundling properties, all of the property information for a filled shape is gathered into a Brush object. The equivalent Java/Graphics code uses methods to set the current fill properties. Note that filling a shape and then drawing a shape with a different color creates a bordered version of the shape.

```
Graphics g = e.Graphics;
Brush grayBrush = new SolidBrush(Color.Gray);
Pen blackWidePen = new Pen(Color.Black,6);
Pen blackSkinnyPen = new Pen(Color.Black,2);
Brush lightGrayBrush = new SolidBrush(Color.LightGray);
g.FillRectangle(grayBrush,10,10,600,200);
g.DrawRectangle(blackWidePen,10,10,600,200);
Brush whiteBrush = new SolidBrush(Color.White);
g.FillEllipse(whiteBrush,10,10,600,200);
g.FillPie(lightGrayBrush,10,10,600,200,30,-60);
g.DrawPie(blackSkinnyPen,10,10,600,200,30,-60);
g.DrawEllipse(blackWidePen,10,10,600,200);
g.DrawLine(blackWidePen,310,40,310,180);
```

Figure 2.19 – C# code for drawing shapes

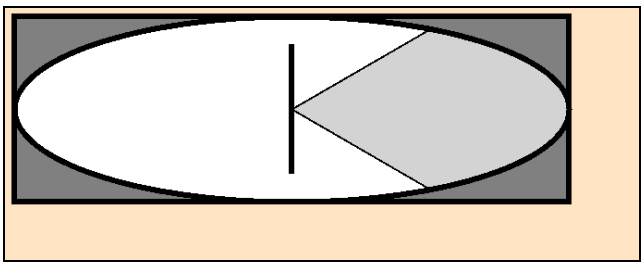

Figure 2.20 – Sample drawing

There are a variety of ways in which a region can be filled. These include various colors, patterns, transparency, gradients in shading and others. These non-geometric attributes for filling regions are handled similarly to the information for paths.

Image Drawing

The last picture drawing methods are those that draw images. These have become increasingly important in enhancing the visual appeal of graphical applications. Images are easy to create using a variety of painting tools. Many image representations use RGBA for their colors, which means that pixels can be transparent or translucent. The end result is that artistic effects are very simple to add by using previously painted images. Much of the visual appeal of many web pages comes from the effective use of images. The HTML drawing model is quite primitive, but images make up the difference quite nicely.

Modern interactive toolkits provide some easy mechanisms for loading images from files or across the web into some Image object. The Graphics object then provides methods for drawing those images. The simplest drawing methods take an image and the X, Y position for the upper left corner of the image. Many systems allow the programmer to specify a rectangle on the Graphics object where the drawing is to appear and possibly a rectangle in the image from which pixels are to be drawn. A scaling and stretching of the image is performed to make the two rectangles conform.

Java has complicated their image drawing model somewhat in order to support web programming. Because images are slow to load over the Internet, Java provides a facility for drawing images asynchronously as the information arrives. This is nice for web applet development because it allows the images to gradually appear rather than making the user wait for the entire image. However, Java applies this model to all image drawing. Therefore every image to be drawn must be managed asynchronously whether it is necessary or not. Reading a good Java book is essential to correct image drawing.

Text Drawing

Drawing text is special because of the variety of information to be specified and the ways in which drawing systems handle how text is to be drawn. There are three basic parameters to drawing text: the string, the font and the geometry. For text that uses all one font, the drawing is quite easy. Most Graphics objects provide a drawString(String, X, Y) method. This will draw the specified string of text at the specified *anchor point* using the current font setting.

Specifying the font usually involves the size of the text, the font family and the style information. The size of text is usually specified in points. A point is 1/72 of an inch. A font size of 12 pt. with normal spacing will produce 6 lines of text per inch. Points are a term held over from the printing press industry. With many different output devices of many resolutions the scale-independent point measure dominates over pixels in specifying text size.

Font family

When a user selects a font, they are actually selecting a font family. This defines how the characters are actually drawn. On most systems a font family is an array of character shapes indexed by character code (ASCII or UNICODE). On older systems the character shape was defined as a 2D array of bits with a 1 signifying where a pixel should be drawn and a 0 signifying not to draw. This strategy was very efficient and produced good results for the low-resolution

displays then in use. However, with the advent of laser printing, large fonts and high-resolution screens, the pixel approach was not adequate. Modern font systems such as ClearType or PostScript use regions bounded by lines and cubic curves. The region representation is scalable to any resolution with good clear results. Changing fonts allows for many styles of script to be implemented. It also allows for the script of many different languages and writing systems as discussed in chapter 10 on internationalization.

Font families are divided into mono-spaced and proportionally spaced, and also into serifed and sans serif. In a mono-spaced font every character is exactly the same width. This mimics the old typewriter and text terminal technology. Mono-spaced fonts are also easier to specify alignment by adding spaces. This paragraph is set using a proportionally spaced font where each character has a different width. Proportional spacing is easier to read and more efficient in its use of screen space.

This paragraph is set using a serifed. Font. At the bottom of each character there are serifs or “little feet”. The effect of this is to create the illusion of a continuous line across the page at the baseline of the text. The macula of our eye (where character recognition occurs) is very small. The tracking of our eyes as we read is handled by the low-resolution periphery of our retina. The serif aids in following along close lines of text. However, with larger headings or in posters the cleaner look of a sans-serif font without the clutter is frequently preferred.

Graphics objects generally only provide two style options: bold and italic. If a font family only contains normal font characters, bold and italic characters can be automatically created from the region shapes by scaling and skewing the shape. Many font families, however, provide separate character shapes for bold and italic. Many user applications provide other styles such as superscript, subscript, underline, shadowed or strike-through. Generally these are not provided in the Graphics object interface because they are easy to produce by combining features already present.

Text metrics

Drawing text requires that the programmer pay attention to the size of the text in order to place it correctly. Figure 2.21 shows the basic text measures. The *baseline* is the basic horizontal location for all of the text in the string. The *ascent* is the distance from the baseline to the top of the tallest character in the font. The *descent* is the distance from the baseline to the lowest character. The *leading* is the spacing between lines. It draws its name from the lead spacers placed between lines of type on old printing presses.

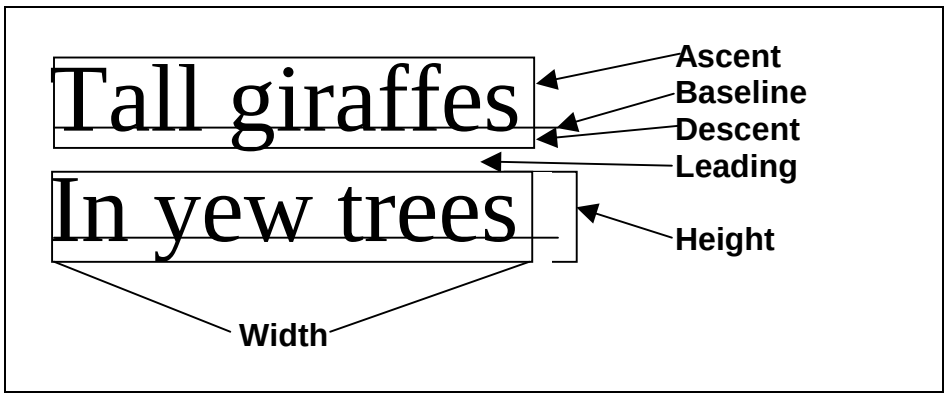

Figure 2.21 – Text metrics

There is some variation among graphics systems as to how these measures are used. Traditionally the anchor point for a piece of text to be drawn was specified at the left end of the baseline and the height was the ascent plus the descent. Leading was up to the programmer and was adjusted by positioning the Y coordinate of the next line's anchor point. Other systems place the anchor point at the top left corner of the text's bounding rectangle and define the height to be ascent+descent+leading. This simplifies a lot of text placement chores. Single spaced text simply starts at some top point and proceed by adding the height to position the anchor point of the next string. Careful reading of the API of your system is required to get this right.

With mono-spaced fonts the width of a string is simply the number of characters times the character width of the font. With proportionally spaced fonts the width is generally the sum of the widths of the characters in the string. For most drawing systems a method is provided to calculate the width of a string. There are, however, some subtle details in the sizing of text. As I am typing this paragraph it is being displayed on my screen using a "Times New Roman" font from Microsoft. When this is printed for you to read it will probably use a version of "Times" font prepared by the vendor of the printer on which this page was printed. These fonts will be very similar but not necessarily identical. For the right-justification of this paragraph to work correctly, the size measurement of the strings must be exact, not just sort of, kind of, maybe similar. To accommodate this, every Graphics object provides access to some form of FontMetrics. A FontMetrics object provides information about exact sizing of strings or characters. It is the Graphics object that knows where the text will ultimately be

drawn and thus can return size information that is appropriate to that output destination.

One variation on proportional spacing is *kerning*. Kerning tries to overlap characters along the X-axis to save space and to produce a more pleasing visual spacing. On the left of figure 2.22 there are two examples of kerning. The top-left example is the normal kerning provided by Adobe Photoshop. The bottom-left shows even more extreme kerning that appreciably shortens the string. The string on the right is not kerned. It uses the standard proportional font spacing that has each character using its own region on the X-axis. Note that the text on the right seems to be uncomfortably spread out even though there is technically zero space between characters. This is because our eyes see space independent of axis alignment. For interactive use we generally do not use kerning because it complicates character spacing and also interactive character selection with a mouse. The font metrics are also more complicated because the appropriate spacing is based on character pairs rather than individual characters.

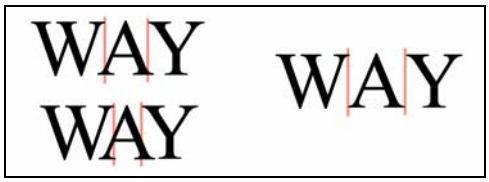

Figure 2.22 - Kerning

Text placement

There are a variety of ways to specify where a string of text should be drawn. Most of them are based on an anchor point, which is the X, Y position specified in the Graphics method that draws a string. As mentioned earlier some systems place the anchor point on the baseline of the string and others at the top of the string. Many systems allow an alignment to be specified, generally left, center or right. These alignments indicate whether the left, center or right side of the string should be aligned with the anchor point. This simplifies laying out multiple lines of text with a given alignment.

There is a problem, however, with text that is justified on both margins. The standard approach is to calculate the size of each word as a separate string, compute the amount of space on the line and then distribute the space evenly between the words. Some graphics systems, such as the one associated with C#, allow text to be specified in a bounding rectangle. By specifying a rectangle, the

underlying graphics system can provide automatic word wrapping and justification within that rectangle.

Multi-font drawing

Drawing in multiple fonts and multiple sizes, such as in figure 2.23 is generally done using multiple calls to the drawString() method. Each section of text that has its own font setting is dealt with as a separate draw so that the font information can be specified. This becomes complicated with automatic word wrap or justification drawing. To do the word wrap and justification the text must be presented all at once. However, a simple text string does not have embedded font information. Some systems have started to provide drawing facilities that support the basic formatting specifications of HTML[3]. This makes multi-font drawing quite easy. However, HTML drawing is far from universal among graphics systems.

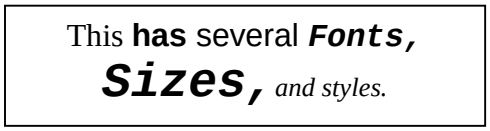

Figure 2.23 – Multi-font drawing

Summary of Drawing

In the model-view-controller architecture it is the responsibility of the View to translate some portion of the model into a presentation on the screen. In interactive settings there are many different events that can create a requirement for the view to be redrawn. Rather than deal with each one individually, every widget (component, view, control, face or other name) implements a redraw() method (paint, onPaint, refresh) that will completely redraw the view from the model. This method is called whenever the windowing system is notified of a need for redrawing.

Drawing generally occurs in a rectangular window and the view assumes that it has complete use of the entire rectangle. All drawing is done through a Graphics object (canvas, surface, pane, device context, etc). The Graphics object insulates the view from the underlying hardware or other drawing medium as well as all other visible windows. The Graphics object generally provides mechanisms for drawing lines, polylines, ellipses, arcs, curves, polygons, curved boundary regions and images. Text is also drawn through the Graphics object. The drawing

of text is also managed by font objects and font metrics to deal with placement and sizing.

---

[1] Kendel, E. R., Schwartz, J. H. and Jessell, T. M, *Principles of Neural Science*, McGraw Hill (2000).

[2] Adobe Systems, *PostScript Language Reference*, Addison-Wesley (1999).

[3] http://www.w3.org/MarkUp/