Chapter 2

Two-Dimensional Graphics

With this chapter, we begin our study of computer graphics by looking at the twodimensional case. Things are simpler and a lot easier to visualize in 2D than in 3D, but most of the ideas that are covered in this chapter will also be very relevant to 3D.

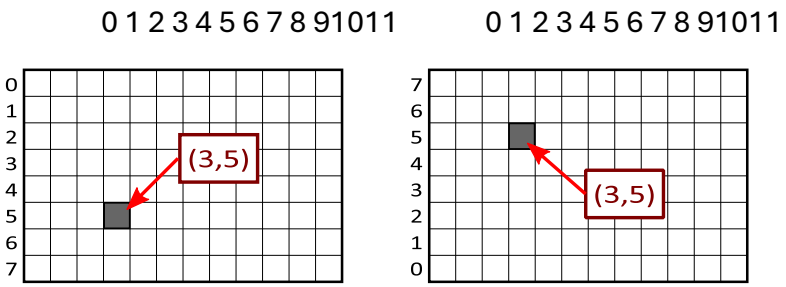
The chapter begins with four sections that examine 2D graphics in a general way, without tying it to a particular programming language or graphics API. The coding examples in these sections are written in pseudocode that should make sense to anyone with enough programming background to be reading this book. In the next three sections, we will take quick looks at 2D graphics in three particular languages: Java with Graphics2D, JavaScript with HTML <canvas> graphics, and SVG. We will see how these languages use many of the general ideas from earlier in the chapter.

2.1 Pixels, Coordinates, and Colors

To create a two-dimensional image, each point in the image is assigned a color. A point in 2D can be identified by a pair of numerical coordinates. Colors can also be specified numerically. However, the assignment of numbers to points or colors is somewhat arbitrary. So we need to spend some time studying coordinate systems, which associate numbers to points, and color models, which associate numbers to colors.

2.1.1 Pixel Coordinates

A digital image is made up of rows and columns of pixels. A pixel in such an image can be specified by saying which column and which row contains it. In terms of coordinates, a pixel can be identified by a pair of integers giving the column number and the row number. For example, the pixel with coordinates (3,5) would lie in column number 3 and row number 5. Conventionally, columns are numbered from left to right, starting with zero. Most graphics systems, including the ones we will study in this chapter, number rows from top to bottom, starting from zero. Some, including OpenGL, number the rows from bottom to top instead.



12-by-8 pixel grids, shown with row and column numbers.

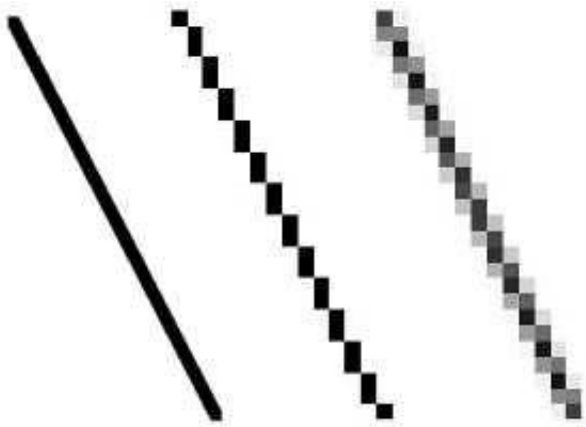
On the left, rows are numbered from top to bottom, on the right, they are numberd bottom to top.

Note in particular that the pixel that is identified by a pair of coordinates (*x*,*y*) depends on the choice of coordinate system. You always need to know what coordinate system is in use before you know what point you are talking about.

Row and column numbers identify a pixel, not a point. A pixel contains many points; mathematically, it contains an infinite number of points. The goal of computer graphics is not really to color pixels—it is to create and manipulate images. In some ideal sense, an image should be defined by specifying a color for each point, not just for each pixel. Pixels are an approximation. If we imagine that there is a true, ideal image that we want to display, then any image that we display by coloring pixels is an approximation. This has many implications.

Suppose, for example, that we want to draw a line segment. A mathematical line has no thickness and would be invisible. So we really want to draw a thick line segment, with some specified width. Let's say that the line should be one pixel wide. The problem is that, unless the line is horizontal or vertical, we can't actually draw the line by coloring pixels. A diagonal geometric line will cover some pixels only partially. It is not possible to make part of a pixel black and part of it white. When you try to draw a line with black and white pixels only, the result is a jagged staircase effect. This effect is an example of something called "aliasing." Aliasing can also be seen in the outlines of characters drawn on the screen and in diagonal or curved boundaries between any two regions of different color. (The term aliasing likely comes from the fact that ideal images are naturally described in real-number coordinates. When you try to represent the image using pixels, many real-number coordinates will map to the same integer pixel coordinates; they can all be considered as different names or "aliases" for the same pixel.)

Antialiasing is a term for techniques that are designed to mitigate the effects of aliasing. The idea is that when a pixel is only partially covered by a shape, the color of the pixel should be a mixture of the color of the shape and the color of the background. When drawing a black line on a white background, the color of a partially covered pixel would be gray, with the shade of gray depending on the fraction of the pixel that is covered by the line. (In practice, calculating this area exactly for each pixel would be too difficult, so some approximate method is used.) Here, for example, is a geometric line, shown on the left, along with two approximations of that line made by coloring pixels. The lines are greatly magnified so that you can see the individual pixels. The line on the right is drawn using antialiasing, while the one in the middle is not:



Note that antialiasing does not give a perfect image, but it can reduce the “jaggies” that are caused by aliasing (at least when it is viewed on a normal scale).

There are other issues involved in mapping real-number coordinates to pixels. For example, which point in a pixel should correspond to integer-valued coordinates such as (3,5)? The center of the pixel? One of the corners of the pixel? In general, we think of the numbers as referring to the top-left corner of the pixel. Another way of thinking about this is to say that integer coordinates refer to the lines between pixels, rather than to the pixels themselves. But that still doesn’t determine exactly which pixels are affected when a geometric shape is drawn. For example, here are two lines drawn using HTML canvas graphics, shown greatly magnified. The lines were specified to be colored black with a one-pixel line width:

The top line was drawn from the point (100,100) to the point (120,100). In canvas graphics, integer coordinates correspond to the lines between pixels, but when a one-pixel line is drawn, it extends one-half pixel on either side of the infinitely thin geometric line. So for the top line, the line as it is drawn lies half in one row of pixels and half in another row. The graphics system, which uses antialiasing, rendered the line by coloring both rows of pixels gray. The bottom line was drawn from the point (100.5,100.5) to (120.5,100.5). In this case, the line lies exactly along one line of pixels, which gets colored black. The gray pixels at the ends of the bottom line have to do with the fact that the line only extends halfway into the pixels at its endpoints. Other graphics systems might render the same lines differently.

The interactive demo *c2/pixel-magnifier.html* lets you experiment with pixels and antialiasing. Interactive demos can be found on the web pages in the on-line version of this book. If you have downloaded the web site, you can also find the demos in the folder named *demos*. (Note that in any of the interactive demos that accompany this book, you can click the question mark icon in the upper left for more information about how to use it.)

* * *

All this is complicated further by the fact that pixels aren't what they used to be. Pixels today are smaller! The resolution of a display device can be measured in terms of the number of pixels per inch on the display, a quantity referred to as PPI (pixels per inch) or sometimes DPI (dots per inch). Early screens tended to have resolutions of somewhere close to 72 PPI. At that resolution, and at a typical viewing distance, individual pixels are clearly visible. For a while, it seemed like most displays had about 100 pixels per inch, but high resolution displays today can have 200, 300 or even 400 pixels per inch. At the highest resolutions, individual pixels can no longer be distinguished.

The fact that pixels come in such a range of sizes is a problem if we use coordinate systems based on pixels. An image created assuming that there are 100 pixels per inch will look tiny on a 400 PPI display. A one-pixel-wide line looks good at 100 PPI, but at 400 PPI, a one-pixel-wide line is probably too thin.

In fact, in many graphics systems, "pixel" doesn't really refer to the size of a physical pixel. Instead, it is just another unit of measure, which is set by the system to be something appropriate. (On a desktop system, a pixel is usually about one one-hundredth of an inch. On a smart phone, which is usually viewed from a closer distance, the value might be closer to 1/160 inch. Furthermore, the meaning of a pixel as a unit of measure can change when, for example, the user applies a magnification to a web page.)

Pixels cause problems that have not been completely solved. Fortunately, they are less of a problem for vector graphics, which is mostly what we will use in this book. For vector graphics, pixels only become an issue during rasterization, the step in which a vector image is converted into pixels for display. The vector image itself can be created using any convenient coordinate system. It represents an idealized, resolution-independent image. A rasterized image is an approximation of that ideal image, but how to do the approximation can be left to the display hardware.

2.1.2 Real-number Coordinate Systems

When doing 2D graphics, you are given a rectangle in which you want to draw some graphics primitives. Primitives are specified using some coordinate system on the rectangle. It should be possible to select a coordinate system that is appropriate for the application. For example, if the rectangle represents a floor plan for a 15 foot by 12 foot room, then you might want to use a coordinate system in which the unit of measure is one foot and the coordinates range from 0 to 15 in the horizontal direction and 0 to 12 in the vertical direction. The unit of measure in this case is feet rather than pixels, and one foot can correspond to many pixels in the image. The coordinates for a pixel will, in general, be real numbers rather than integers. In fact, it's better to forget about pixels and just think about points in the image. A point will have a pair of coordinates given by real numbers.

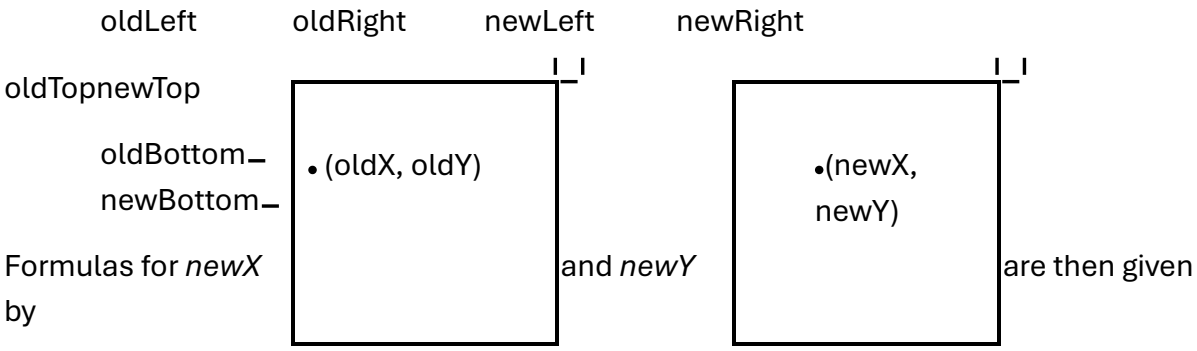
To specify the coordinate system on a rectangle, you just have to specify the horizontal coordinates for the left and right edges of the rectangle and the vertical coordinates for the top and bottom. Let's call these values *left*, *right*, *top*, and *bottom*. Often, they are thought of as *xmin*, *xmax*, *ymin*, and *ymax*, but there is no reason to assume that, for example, *top* is less than *bottom*. We might want a coordinate system in which the vertical coordinate increases from bottom to top instead of from top to bottom. In that case, *top* will correspond to the maximum *y*-value instead of the minimum value.

To allow programmers to specify the coordinate system that they would like to use, it would be good to have a subroutine such as

setCoordinateSystem(left,right,bottom,top)

The graphics system would then be responsible for automatically transforming the coordinates from the specified coordinate system into pixel coordinates. Such a subroutine might not be available, so it's useful to see how the transformation is done by hand. Let's consider the general case. Given coordinates for a point in one coordinate system, we want to find the coordinates for the same point in a second coordinate system. (Remember that a coordinate system is just a way of assigning numbers to points. It's the points that are real!) Suppose that the horizontal and vertical limits are *oldLeft*, *oldRight*, *oldTop*, and *oldBottom* for the first coordinate system, and are *newLeft*, *newRight*, *newTop*, and *newBottom* for the second. Suppose that a point has coordinates (*oldX,oldY*) in the first coordinate system. We want to find the coordinates

(*newX,newY*) of the point in the second coordinate system



Formulas for *newX* and *newY* are then given by

newX = newLeft +

((oldX - oldLeft) / (oldRight - oldLeft)) * (newRight - newLeft)) newY = newTop +

((oldY - oldTop) / (oldBottom - oldTop)) * (newBottom - newTop)

The logic here is that *oldX* is located at a certain fraction of the distance from *oldLeft* to *oldRight*. That fraction is given by

((oldX - oldLeft) / (oldRight - oldLeft))

The formula for *newX* just says that *newX* should lie at the same fraction of the distance from *newLeft* to *newRight*. You can also check the formulas by testing that they work when *oldX* is equal to *oldLeft* or to *oldRight*, and when *oldY* is equal to *oldBottom* or to *oldTop*.

As an example, suppose that we want to transform some real-number coordinate system with limits *left*, *right*, *top*, and *bottom* into pixel coordinates that range from 0 at left to 800 at the right and from 0 at the top 600 at the bottom. In that case, *newLeft* and *newTop* are zero, and the formulas become simply

newX = ((oldX - left) / (right - left)) * 800 newY = ((oldY - top) / (bottom - top)) * 600

Of course, this gives *newX* and *newY* as real numbers, and they will have to be rounded or truncated to integer values if we need integer coordinates for pixels. The reverse transformation—going from pixel coordinates to real number coordinates—is also useful. For example, if the image is displayed on a computer screen, and you want to react to mouse clicks on the image, you will probably get the mouse coordinates in terms of integer pixel coordinates, but you will want to transform those pixel coordinates into your own chosen coordinate system.

In practice, though, you won't usually have to do the transformations yourself, since most graphics APIs provide some higher level way to specify transforms. We will talk more about this in Section 2.3.

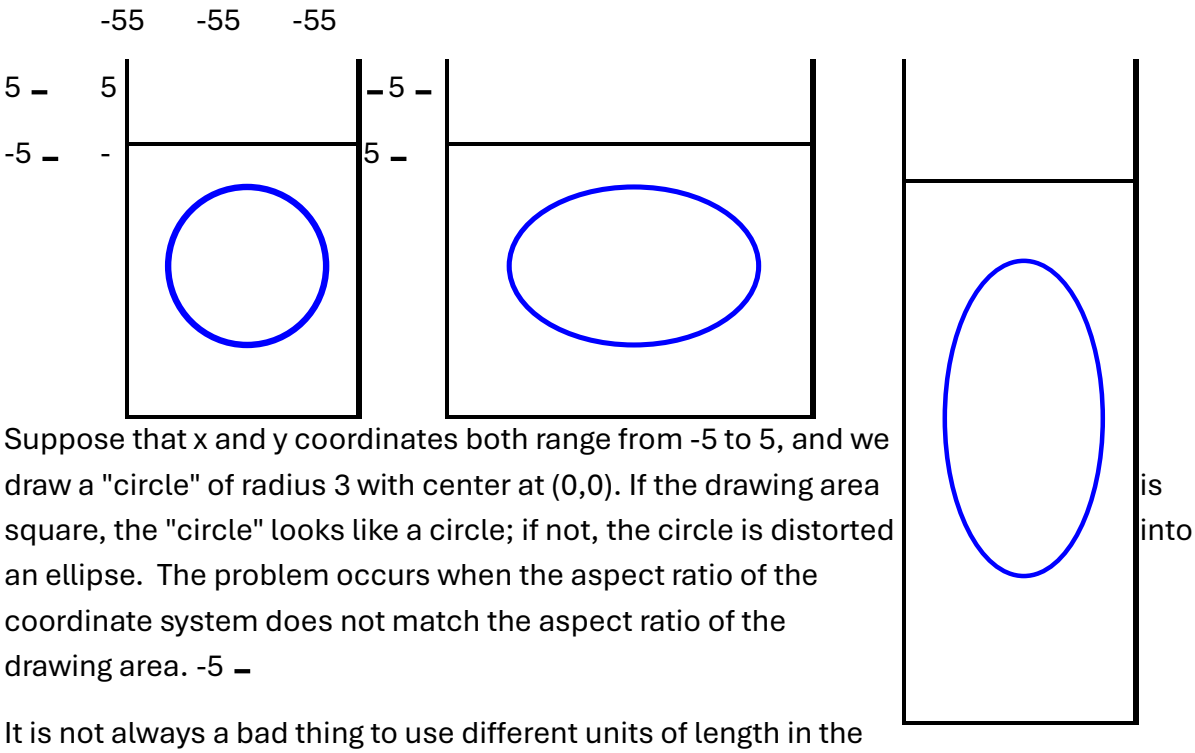
2.1.3 Aspect Ratio

The aspect ratio of a rectangle is the ratio of its width to its height. For example an aspect ratio of 2:1 means that a rectangle is twice as wide as it is tall, and an aspect ratio of 4:3 means that the width is 4/3 times the height. Although aspect ratios are often written in the form *width*:*height*, I will use the term to refer to the fraction *width/height*. A square has aspect ratio equal to 1. A rectangle with aspect ratio 5/4 and height 600 has a width equal to 600*(5/4), or 750.

A coordinate system also has an aspect ratio. If the horizontal and vertical limits for the coordinate system are *left*, *right*, *bottom*, and *top*, as above, then the aspect ratio is the absolute value of

(right - left) / (top - bottom)

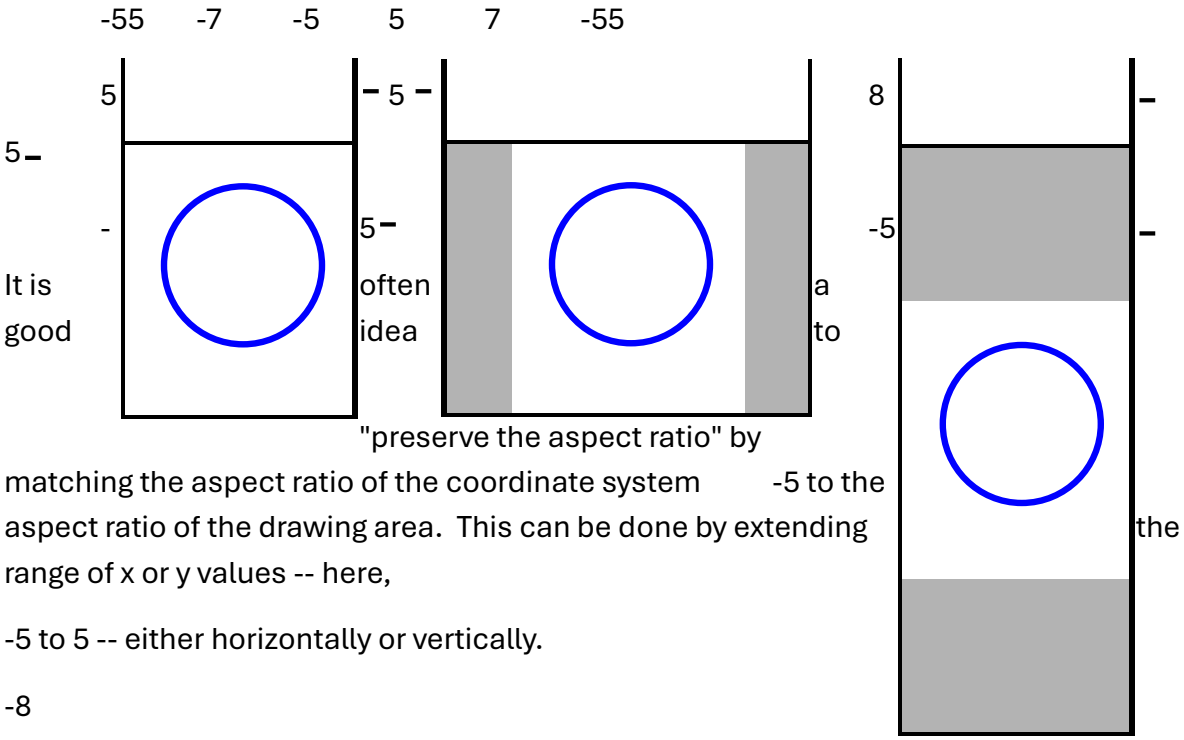
If the coordinate system is used on a rectangle with the same aspect ratio, then when viewed in that rectangle, one unit in the horizontal direction will have the same apparent length as a unit in the vertical direction. If the aspect ratios don't match, then there will be some distortion. For example, the shape defined by the equation $x^2+y^2=9$ should be a circle, but that will only be true if the aspect ratio of the (x,y) coordinate system matches the aspect ratio of the drawing area.



Suppose that x and y coordinates both range from -5 to 5, and we draw a "circle" of radius 3 with center at (0,0). If the drawing area is square, the "circle" looks like a circle; if not, the circle is distorted into an ellipse. The problem occurs when the aspect ratio of the coordinate system does not match the aspect ratio of the drawing area. -5

It is not always a bad thing to use different units of length in the vertical and horizontal directions. However, suppose that you want to use coordinates with limits *left*, *right*, *bottom*, and *top*, and that you do want to preserve the aspect ratio.

In that case, depending on the shape of the display rectangle, you might have to adjust the values either of *left* and *right* or of *bottom* and *top* to make the aspect ratios match:



It is often a good idea to "preserve the aspect ratio" by matching the aspect ratio of the coordinate system to the aspect ratio of the drawing area. This can be done by extending the range of x or y values -- here, -5 to the range -5 to 5 -- either horizontally or vertically.

We will look more deeply into geometric transforms later in the chapter, and at that time, we'll see some program code for setting up coordinate systems.

2.1.4 Color Models

We are talking about the most basic foundations of computer graphics. One of those is coordinate systems. The other is color. Color is actually a surprisingly complex topic. We will look at some parts of the topic that are most relevant to computer graphics applications.

The colors on a computer screen are produced as combinations of red, green, and blue light. Different colors are produced by varying the intensity of each type of light. A color can be specified by three numbers giving the intensity of red, green, and blue in the color. Intensity can be specified as a number in the range zero, for minimum intensity, to one, for maximum intensity. This method of specifying color is called the RGB color model, where RGB stands for Red/Green/Blue. For example, in the RGB color model, the number triple (1, 0.5, 0.5) represents the color obtained by setting red to full intensity, while green and blue are set to half intensity. The red, green, and blue values for a color are called the color components of that color in the RGB color model.

Light is made up of waves with a variety of wavelengths. A pure color is one for which all the light has the same wavelength, but in general, a color can contain many wavelengths— mathematically, an infinite number. How then can we represent all colors by combining just red, green, and blue light? In fact, we can't quite do that.

You might have heard that combinations of the three basic, or "primary," colors are sufficient to represent all colors, because the human eye has three kinds of color sensors that detect red, green, and blue light. However, that is only an approximation. The eye does contain three kinds of color sensors. The sensors are called "cone cells." However, cone cells do not respond exclusively to red, green, and blue light. Each kind of cone cell responds, to a varying degree, to wavelengths of light in a wide range. A given mix of wavelengths will stimulate each type of cell to a certain degree, and the intensity of stimulation determines the color that we see. A different mixture of wavelengths that stimulates each type of cone cell to the same extent will be perceived as the same color. So a perceived color can, in fact, be specified by three numbers giving the intensity of stimulation of the three types of cone cell. However, it is not possible to produce all possible patterns of stimulation by combining just three basic colors, no matter how those colors are chosen. This is just a fact about the way our eyes actually work; it might have been different. Three basic colors can produce a reasonably large fraction of the set of perceivable colors, but there are colors that you can see in the world that you will never see on your computer screen. (This whole discussion only applies to people who actually have three kinds of cone cell. Color blindness, where someone is missing one or more kinds of cone cell, is surprisingly common.)

The range of colors that can be produced by a device such as a computer screen is called the color gamut of that device. Different computer screens can have different color gamuts, and the same RGB values can produce somewhat different colors on different screens. The color gamut of a color printer is noticeably different—and probably smaller—than the color gamut of a screen, which explains why a printed image probably doesn't look exactly the same as it did on the screen. (Printers, by the way, make colors differently from the way a screen does it. Whereas a screen combines light to make a color, a printer combines inks or dyes. Because of this difference, colors meant for printers are often expressed using a different set of basic colors. A common color model for printer colors is CMYK, using the colors cyan, magenta, yellow, and black.)

In any case, the most common color model for computer graphics is RGB. RGB colors are most often represented using 8 bits per color component, a total of 24 bits to

represent a color. This representation is sometimes called “24-bit color.” An 8-bit number can represent 2^8, or 256, different values, which we can take to be the positive integers from 0 to 255. A color is then specified as a triple of integers (r,g,b) in that range.

This representation works well because 256 shades of red, green, and blue are about as many as the eye can distinguish. In applications where images are processed by computing with color components, it is common to use additional bits per color component to avoid visual effects that might occur due to rounding errors in the computations. Such applications might use a 16-bit integer or even a 32-bit floating point value for each color component. On the other hand, sometimes fewer bits are used. For example, one common color scheme uses 5 bits for the red and blue components and 6 bits for the green component, for a total of 16 bits for a color. (Green gets an extra bit because the eye is more sensitive to green light than to red or blue.) This “16-bit color” saves memory compared to 24-bit color and was more common when memory was more expensive.

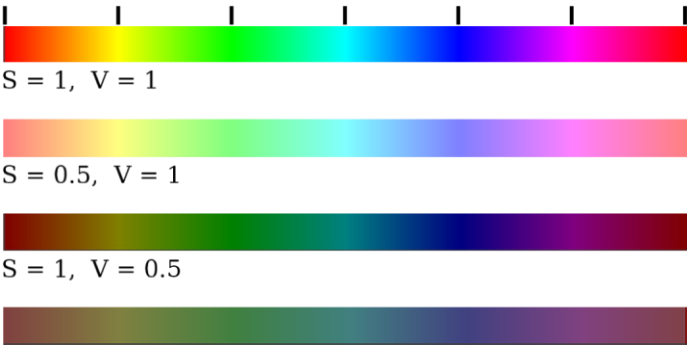
There are many other color models besides RGB. RGB is sometimes criticized as being unintuitive. For example, it’s not obvious to most people that yellow is made of a combination of red and green. The closely related color models HSV and HSL describe the same set of colors as RGB, but attempt to do it in a more intuitive way. (HSV is sometimes called HSB, with the “B” standing for “brightness.” HSV and HSB are exactly the same model.)

The “H” in these models stands for “hue,” a basic spectral color. As H increases, the color changes from red to yellow to green to cyan to blue to magenta, and then back to red. The value of H is often taken to range from 0 to 360, since the colors can be thought of as arranged around a circle with red at both 0 and 360 degrees.

The “S” in HSV and HSL stands for “saturation,” and is taken to range from 0 to 1. A saturation of 0 gives a shade of gray (the shade depending on the value of V or L). A saturation of 1 gives a “pure color,” and decreasing the saturation is like adding more gray to the color. “V” stands for “value,” and “L” stands for “lightness.” They determine how bright or dark the color is. The main difference is that in the HSV model, the pure spectral colors occur for V=1, while in HSL, they occur for L=0.5.

Let’s look at some colors in the HSV color model. The illustration below shows colors with

a full range of H-values, for S and V equal to 1 and to 0.5. Note that for S=V=1, you get bright, pure colors. S=0.5 gives paler, less saturated colors. V=0.5 gives darker colors.



It's probably easier to understand color models by looking at some actual colors and how they are represented. The interactive demo *c2/rgb-hsv.html* lets you experiment with the RGB and HSV color models.

* * *

Often, a fourth component is added to color models. The fourth component is called alpha, and color models that use it are referred to by names such as RGBA and HSLA. Alpha is not a color as such. It is usually used to represent transparency. A color with maximal alpha value is fully opaque; that is, it is not at all transparent. A color with alpha equal to zero is completely transparent and therefore invisible. Intermediate values give translucent, or partly transparent, colors. Transparency determines what happens when you draw with one color (the foreground color) on top of another color (the background color). If the foreground color is fully opaque, it simply replaces the background color. If the foreground color is partly transparent, then it is blended with the background color. Assuming that the alpha component ranges from 0 to 1, the color that you get can be computed as new color = (alpha)*(foreground_color) + (1 - alpha)*(background_color)

This computation is done separately for the red, blue, and green color components. This is called alpha blending. The effect is like viewing the background through colored glass; the color of the glass adds a tint to the background color. This type of blending is not the only possible use of the alpha component, but it is the most common.

An RGBA color model with 8 bits per component uses a total of 32 bits to represent a color. This is a convenient number because integer values are often represented using 32-bit values. A 32-bit integer value can be interpreted as a 32-bit RGBA color. How the color components are arranged within a 32-bit integer is somewhat arbitrary. The most common layout is to store the alpha component in the eight high-order bits, followed by

red, green, and blue. (This should probably be called ARGB color.) However, other layouts are also in use.

2.2 Shapes

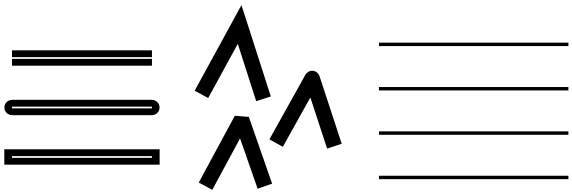
We have been talking about low-level graphics concepts like pixels and coordinates, but fortunately we don't usually have to work on the lowest levels. Most graphics systems let you work with higher-level shapes, such as triangles and circles, rather than individual pixels. And a lot of the hard work with coordinates is done using transforms rather than by working with coordinates directly. In this section and the next, we will look at some of the higher-level capabilities that are typically provided by 2D graphics APIs.

2.2.1 Basic Shapes

In a graphics API, there will be certain basic shapes that can be drawn with one command, whereas more complex shapes will require multiple commands. Exactly what qualifies as a basic shape varies from one API to another. In the WebGL API, for example, the only basic shapes are points, lines, and triangles. In this subsection, I consider lines, rectangles, and ovals to be basic.

By "line," I really mean line segment, that is a straight line segment connecting two given points in the plane. A simple one-pixel-wide line segment, without antialiasing, is the most basic shape. It can be drawn by coloring pixels that lie along the infinitely thin geometric line segment. An algorithm for drawing the line has to decide exactly which pixels to color. One of the first computer graphics algorithms, Bresenham's algorithm for line drawing, implements a very efficient procedure for doing so. I won't discuss such low-level details here, but it's worth looking them up if you want to start learning about what graphics hardware actually has to do on a low level. In any case, lines are typically more complicated. Antialiasing is one complication. Line width is another. A wide line might actually be drawn as a rectangle.

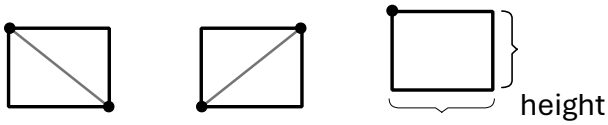
Lines can have other attributes, or properties, that affect their appearance. One question is, what should happen at the end of a wide line? Appearance might be improved by adding a rounded "cap" on the ends of the line. A square cap—that is, extending the line by half of the line width—might also make sense. Another question is, when two lines meet as part of a larger shape, how should the lines be joined? And many graphics systems support lines that are patterns of dashes and dots. This illustration shows some of the possibilities:



On the left are three wide lines with no cap, a round cap, and a square cap. The geometric line segment is shown as a dotted line. (The no-cap style is called “butt.”) To the right are four lines with different patterns of dots and dashes. In the middle are three different styles of line joins: mitered, rounded, and beveled.

* * *

The basic rectangular shape has sides that are vertical and horizontal. (A tilted rectangle generally has to be made by applying a rotation.) Such a rectangle can be specified with two points, (x1,y1) and (x2,y2), that give the endpoints of one of the diagonals of the rectangle. Alternatively, the width and the height can be given, along with a single base point, (x,y). In that case, the width and height have to be positive, or the rectangle is empty. The base point (x,y) will be the upper left corner of the rectangle if y increases from top to bottom, and it will be the lower left corner of the rectangle if y increases from bottom to top.

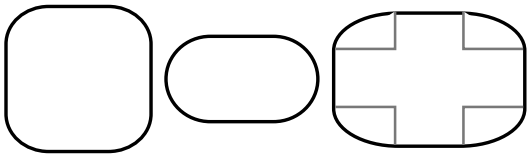


Suppose that you are given points (x1,y1) and (x2,y2), and that you want to draw the rectangle that they determine. And suppose that the only rectangle-drawing command that you have available is one that requires a point (x,y), a width, and a height. For that command, x must be the smaller of x1 and x2, and the width can be computed as the absolute value of x1 minus x2. And similarly for y and the height. In pseudocode,

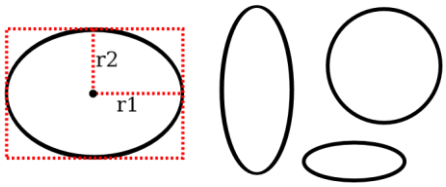
DrawRectangle from points (x1,y1) and (x2,y2): x = min(x1, x2) y = min(y1, y2) width = abs(x1 - x2) height = abs(y1 - y2)

DrawRectangle(x, y, width, height)

A common variation on rectangles is to allow rounded corners. For a “round rect,” the corners are replaced by elliptical arcs. The degree of rounding can be specified by giving the horizontal radius and vertical radius of the ellipse. Here are some examples of round rects. For the shape at the right, the two radii of the ellipse are shown:



My final basic shape is the oval. (An oval is also called an ellipse.) An oval is a closed curve that has two radii. For a basic oval, we assume that the radii are vertical and horizontal. An oval with this property can be specified by giving the rectangle that just contains it. Or it can be specified by giving its center point and the lengths of its vertical radius and its horizontal radius. In this illustration, the oval on the left is shown with its containing rectangle and with its center point and radii:



The oval on the right is a circle. A circle is just an oval in which the two radii have the same length.

If ovals are not available as basic shapes, they can be approximated by drawing a large number of line segments. The number of lines that is needed for a good approximation depends on the size of the oval. It's useful to know how to do this. Suppose that an oval has center point (x,y), horizontal radius r1, and vertical radius r2. Mathematically, the points on the oval are given by

(x + r1*cos(angle), y + r2*sin(angle))

where *angle* takes on values from 0 to 360 if angles are measured in degrees or from 0 to 2*π* if they are measured in radians. Here *sin* and *cos* are the standard sine and cosine functions. To get an approximation for an oval, we can use this formula to generate some number of points and then connect those points with line segments. In pseudocode, assuming that angles are measured in radians and that *pi* represents the mathematical constant *π*,

Draw Oval with center (x,y), horizontal radius r1, and vertical radius r2:

for i = 0 to numberOfLines:

angle1 = i * (2*pi/numberOfLines) angle2 = (i+1) * (2*pi/numberOfLines) a1 = x + r1*cos(angle1) b1 = y + r2*sin(angle1) a2 = x + r1*cos(angle2) b2 = y + r2*sin(angle2)

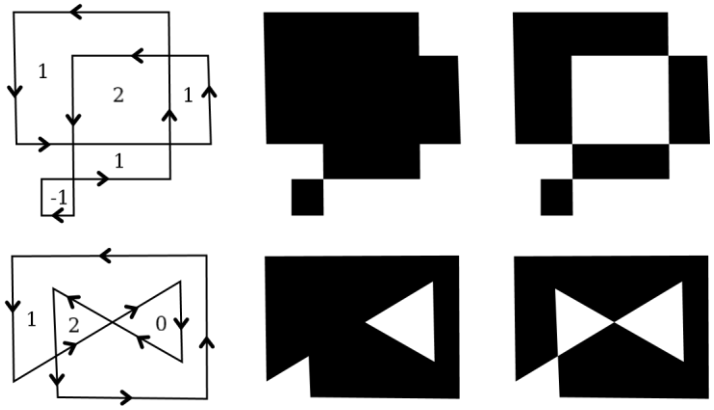
Draw Line from (x1,y1) to (x2,y2)

For a circle, of course, you would just have r1 = r2. This is the first time we have used the sine and cosine functions, but it won't be the last. These functions play an important role in computer graphics because of their association with circles, circular motion, and rotation. We will meet them again when we talk about transforms in the next section.

2.2.2 Stroke and Fill

There are two ways to make a shape visible in a drawing. You can stroke it. Or, if it is a closed shape such as a rectangle or an oval, you can fill it. Stroking a line is like dragging a pen along the line. Stroking a rectangle or oval is like dragging a pen along its boundary. Filling a shape means coloring all the points that are contained inside that shape. It's possible to both stroke and fill the same shape; in that case, the interior of the shape and the outline of the shape can have a different appearance.

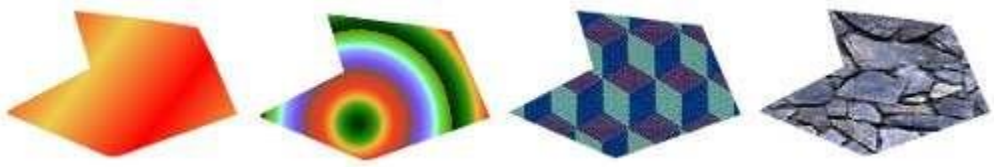
When a shape intersects itself, like the two shapes in the illustration below, it's not entirely clear what should count as the interior of the shape. In fact, there are at least two different rules for filling such a shape. Both are based on something called the winding number. The winding number of a shape about a point is, roughly, how many times the shape winds around the point in the positive direction, which I take here to be counterclockwise. Winding number can be negative when the winding is in the opposite direction. In the illustration, the shapes on the left are traced in the direction shown, and the winding number about each region is shown as a number inside the region.



The shapes are also shown filled using the two fill rules. For the shapes in the center, the fill rule is to color any region that has a non-zero winding number. For the shapes shown on the right, the rule is to color any region whose winding number is odd; regions with even winding number are not filled.

There is still the question of what a shape should be filled *with*. Of course, it can be filled with a color, but other types of fill are possible, including patterns and gradients. A

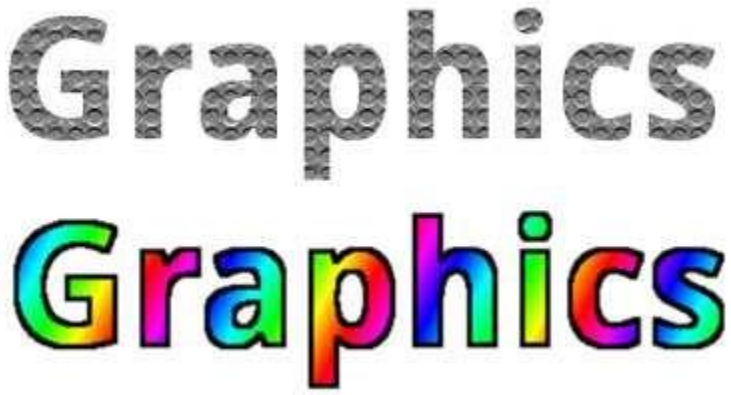
pattern is an image, usually a small image. When used to fill a shape, a pattern can be repeated horizontally and vertically as necessary to cover the entire shape. A gradient is similar in that it is a way for color to vary from point to point, but instead of taking the colors from an image, they are computed. There are a lot of variations to the basic idea, but there is always a line segment along which the color varies. The color is specified at the endpoints of the line segment, and possibly at additional points; between those points, the color is interpolated. The color can also be extrapolated to other points on the line that contains the line segment but lying outside the line segment; this can be done either by repeating the pattern from the line segment or by simply extending the color from the nearest endpoint. For a linear gradient, the color is constant along lines perpendicular to the basic line segment, so you get lines of solid color going in that direction. In a radial gradient, the color is constant along circles centered at one of the endpoints of the line segment. And that doesn't exhaust the possibilities. To give you an idea what patterns and gradients can look like, here is a shape, filled with two gradients and two patterns:



The first shape is filled with a simple linear gradient defined by just two colors, while the second shape uses a radial gradient.

Patterns and gradients are not necessarily restricted to filling shapes. Stroking a shape is, after all, the same as filling a band of pixels along the boundary of the shape, and that can be done with a gradient or a pattern, instead of with a solid color.

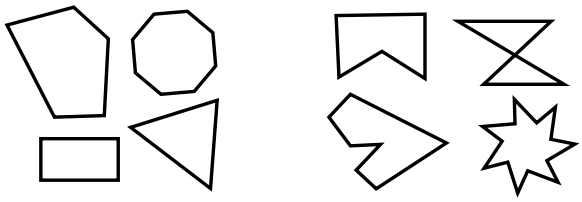
Finally, I will mention that a string of text can be considered to be a shape for the purpose of drawing it. The boundary of the shape is the outline of the characters. The text is drawn by filling that shape. In some graphics systems, it is also possible to stroke the outline of the shape that defines the text. In the following illustration, the string "Graphics" is shown, on top, filled with a pattern and, below that, filled with a gradient and stroked with solid black:



2.2.3 Polygons, Curves, and Paths

It is impossible for a graphics API to include every possible shape as a basic shape, but there is usually some way to create more complex shapes. For example, consider polygons. A polygon is a closed shape consisting of a sequence of line segments. Each line segment is joined to the next at its endpoint, and the last line segment connects back to the first. The endpoints are called the vertices of the polygon, and a polygon can be defined by listing its vertices.

In a regular polygon, all the sides are the same length and all the angles between sides are equal. Squares and equilateral triangles are examples of regular polygons. A convex polygon has the property that whenever two points are inside or on the polygon, then the entire line segment between those points is also inside or on the polygon. Intuitively, a convex polygon has no "indentations" along its boundary. (Concavity can be a property of any shape, not just of polygons.)



olygons olygons

Sometimes, polygons are required to be "simple," meaning that the polygon has no selfintersections. That is, all the vertices are different, and a side can only intersect another side at its endpoints. And polygons are usually required to be "planar," meaning that all the vertices lie in the same plane. (Of course, in 2D graphics, *everything* lies in the same plane, so this is not an issue. However, it does become an issue in 3D.)

How then should we draw polygons? That is, what capabilities would we like to have in a graphics API for drawing them. One possibility is to have commands for stroking and for

filling polygons, where the vertices of the polygon are given as an array of points or as an array of x-coordinates plus an array of y-coordinates. In fact, that is sometimes done; for example, the Java graphics API includes such commands. Another, more flexible, approach is to introduce the idea of a "path." Java, SVG, and the HTML canvas API all support this idea. A path is a general shape that can include both line segments and curved segments. Segments can, but don't have to be, connected to other segments at their endpoints. A path is created by giving a series of commands that tell, essentially, how a pen would be moved to draw the path. While a path is being created, there is a point that represents the pen's current location. There will be a command for moving the pen without drawing, and commands for drawing various kinds of segments. For drawing polygons, we need commands such as

- createPath() — start a new, empty path
- moveTo(x,y) — move the pen to the point (x,y), without adding a segment to the path; that is, without drawing anything
- lineTo(x,y) — add a line segment to the path that starts at the current pen location and ends at the point (x,y), and move the pen to (x,y)
- closePath() — add a line segment from the current pen location back to the starting point, unless the pen is already there, producing a closed path.

(For closePath, I need to define "starting point." A path can be made up of "subpaths" A subpath consists of a series of connected segments. A moveTo always starts a new subpath. A closePath ends the current segment and implicitly starts a new one. So "starting point" means the position of the pen after the most recent moveTo or closePath.)

Suppose that we want a path that represents the triangle with vertices at (100,100), (300,100), and (200, 200). We can do that with the commands

createPath() moveTo(100, 100) lineTo(300, 100) lineTo(200, 200) closePath()

The closePath command at the end could be replaced by lineTo(100,100), to move the pen back to the first vertex.

A path represents an abstract geometric object. Creating one does not make it visible on the screen. Once we have a path, to make it visible we need additional commands for stroking and filling the path.

Earlier in this section, we saw how to approximate an oval by drawing, in effect, a polygon with a large number of sides. In that example, I drew each side as a separate

line segment, so we really had a bunch of separate lines rather than a polygon. There is no way to fill such a thing. It would be better to approximate the oval with a polygonal path. For an oval with center (x,y) and radii r1 and r2:

createPath()

moveTo(x + r1, y) for i = 1 to numberOfPoints-1 angle = i * (2*pi/numberOfLines) lineTo(x + r1*cos(angle), y + r2*sin(angle))

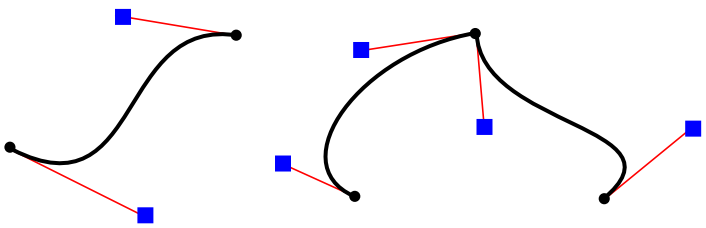
closePath()

Using this path, we could draw a filled oval as well as stroke it. Even if we just want to draw the outline of a polygon, it's still better to create the polygon as a path rather than to draw the line segments as separate sides. With a path, the computer knows that the sides are part of single shape. This makes it possible to control the appearance of the "join" between consecutive sides, as noted earlier in this section.

* * *

I noted above that a path can contain other kinds of segments besides lines. For example, it might be possible to include an arc of a circle as a segment. Another type of curve is a Bezier curve. Bezier curves can be used to create very general curved shapes. They are fairly intuitive, so that they are often used in programs that allow users to design curves interactively. Mathematically, Bezier curves are defined by parametric polynomial equations, but you don't need to understand what that means to use them. There are two kinds of Bezier curve in common use, cubic Bezier curves and quadratic Bezier curves; they are defined by cubic and quadratic polynomials respectively. When the general term "Bezier curve" is used, it usually refers to cubic Bezier curves.

A cubic Bezier curve segment is defined by the two endpoints of the segment together with two control points. To understand how it works, it's best to think about how a pen would draw the curve segment. The pen starts at the first endpoint, headed in the direction of the first control point. The distance of the control point from the endpoint controls the speed of the pen as it starts drawing the curve. The second control point controls the direction and speed of the pen as it gets to the second endpoint of the curve. There is a unique cubic curve that satisfies these conditions.



The illustration above shows three cubic Bezier curve segments. The two curve segments on the right are connected at an endpoint to form a longer curve. The curves are drawn as thick black lines. The endpoints are shown as black dots and the control points as blue squares, with a thin red line connecting each control point to the corresponding endpoint. (Ordinarily, only the curve would be drawn, except in an interface that lets the user edit the curve by hand.) Note that at an endpoint, the curve segment is tangent to the line that connects the endpoint to the control point. Note also that there can be a sharp point or corner where two curve segments meet. However, one segment will merge smoothly into the next if control points are properly chosen.

This will all be easier to understand with some hands-on experience. The interactive demo *c2/cubic-bezier.html* lets you edit cubic Bezier curve segments by dragging their endpoints and control points.

When a cubic Bezier curve segment is added to a path, the path's current pen location acts as the first endpoint of the segment. The command for adding the segment to the path must specify the two control points and the second endpoint. A typical command might look like cubicCurveTo(cx1, cy1, cx2, cy2, x, y)

This would add a curve from the current location to point (x,y), using (cx1,cy1) and (cx2,cy2) as the control points. That is, the pen leaves the current location heading towards (cx1,cy1), and it ends at the point (x,y), arriving there from the direction of (cx2,cy2).

Quadratic Bezier curve segments are similar to the cubic version, but in the quadratic case, there is only one control point for the segment. The curve leaves the first endpoint heading in the direction of the control point, and it arrives at the second endpoint coming from the direction of the control point. The curve in this case will be an arc of a parabola.

Again, this is easier to understand this with some hands-on experience. Try the interactive demo *c2/quadratic-bezier.html*.

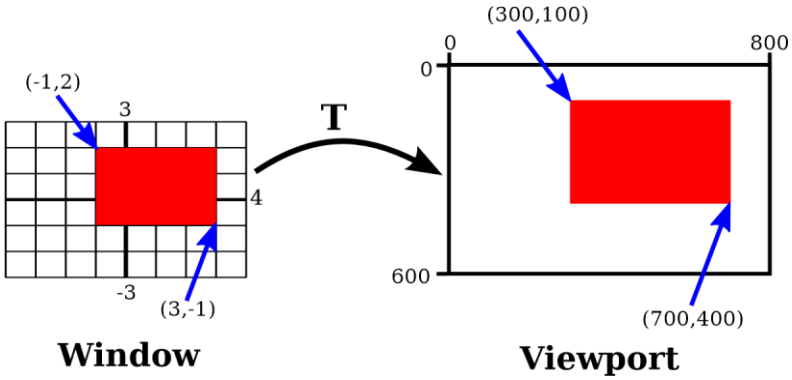
2.3 Transforms

In Section 2.1, we discussed coordinate systems and how it is possible to transform coordinates from one coordinate system to another. In this section, we'll look at that idea a little more closely, and also look at how geometric transformations can be used to place graphics objects into a coordinate system.

2.3.1 Viewing and Modeling

In a typical application, we have a rectangle made of pixels, with its natural pixel coordinates, where an image will be displayed. This rectangle will be called the viewport. We also have a set of geometric objects that are defined in a possibly different coordinate system, generally one that uses real-number coordinates rather than integers. These objects make up the "scene" or "world" that we want to view, and the coordinates that we use to define the scene are called world coordinates.

For 2D graphics, the world lies in a plane. It's not possible to show a picture of the entire infinite plane. We need to pick some rectangular area in the plane to display in the image. Let's call that rectangular area the window, or view window. A coordinate transform is used to map the window to the viewport.

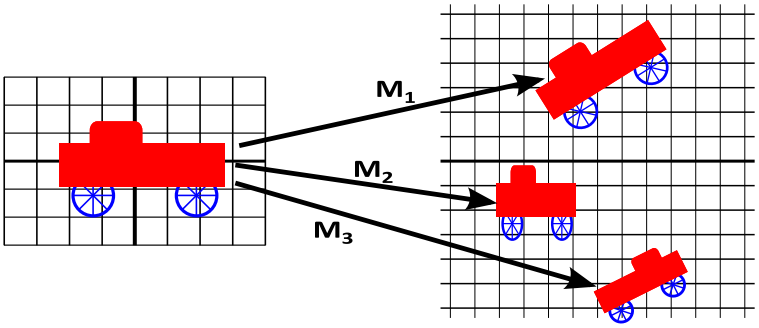


In this illustration, **T** represents the coordinate transformation. **T** is a function that takes world coordinates (*x*,*y*) in some window and maps them to pixel coordinates **T**(*x*,*y*) in the viewport. (I've drawn the viewport and window with different sizes to emphasize that they are not the same thing, even though they show the same objects, but in fact they don't even exist in the same space, so it doesn't really make sense to compare their sizes.) In this example, as you can check,

T(x,y) = (800*(x+4)/8, 600*(3-y)/6)

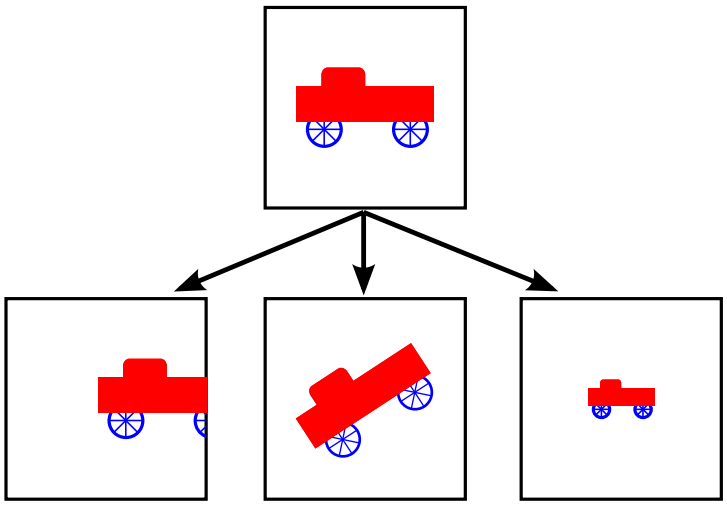
Look at the rectangle with corners at (-1,2) and (3,-1) in the window. When this rectangle is displayed in the viewport, it is displayed as the rectangle with corners **T**(-1,2) and **T**(3,-1). In this example, **T**(-1,2) = (300,100) and **T**(3,-1) = (700,400).

We use coordinate transformations in this way because it allows us to choose a world coordinate system that is natural for describing the scene that we want to display, and it is easier to do that than to work directly with viewport coordinates. Along the same lines, suppose that we want to define some complex object, and suppose that there will be several copies of that object in our scene. Or maybe we are making an animation, and we would like the object to have different positions in different frames. We would like to choose some convenient coordinate system and use it to define the object once and for all. The coordinates that we use to define an object are called object coordinates for the object. When we want to place the object into a scene, we need to transform the object coordinates that we used to define the object into the world coordinate system that we are using for the scene. The transformation that we need is called a modeling transformation. This picture illustrates an object defined in its own object coordinate system and then mapped by three different modeling transformations into the world coordinate system:



Remember that in order to view the scene, there will be another transformation that maps the object from a view window in world coordinates into the viewport.

Now, keep in mind that the choice of a view window tells which part of the scene is shown in the image. Moving, resizing, or even rotating the window will give a different view of the scene. Suppose we make several images of the same car:



What happened between making the top image in this illustration and making the image on the bottom left? In fact, there are two possibilities: Either the car was moved to the *right*, or the view window that defines the scene was moved to the *left*. This is important, so be sure you understand it. (Try it with your cell phone camera. Aim it at some objects, take a step to the left, and notice what happens to the objects in the camera's viewfinder: They move to the right in the picture!) Similarly, what happens between the top picture and the middle picture on the bottom? Either the car rotated *counterclockwise*, or the window was rotated *clockwise*. (Again, try it with a camera—you might want to take two actual photos so that you can compare them.) Finally, the change from the top picture to the one on the bottom right could happen because the car got *smaller* or because the window got *larger*. (On your camera, a bigger window means that you are seeing a larger field of view, and you can get that by applying a zoom to the camera or by backing up away from the objects that you are viewing.)

There is an important general idea here. When we modify the view window, we change the coordinate system that is applied to the viewport. But in fact, this is the same as leaving that coordinate system in place and moving the objects in the scene instead. Except that to get the same effect in the final image, you have to apply the opposite transformation to the objects (for example, moving the window to the *left* is equivalent to moving the objects to the *right*). So, there is no essential distinction between transforming the window and transforming the object. Mathematically, you specify a geometric primitive by giving coordinates in some natural coordinate system, and the computer applies a sequence of transformations to those coordinates to produce, in the end, the coordinates that are used to actually draw the primitive in the image. You will think of some of those transformations as modeling transforms and some as coordinate transforms, but to the computer, it's all the same.

The on-line version of this section includes the live demo *c2/transform-equivalence-2d.html* that can help you to understand the equivalence between modeling transformations and viewport transformations. Read the help text in the demo for more information.

We will return to this idea several times later in the book, but in any case, you can see that geometric transforms are a central concept in computer graphics. Let's look at some basic types of transformation in more detail. The transforms we will use in 2D graphics can be written in the form

x1 = a*x + b*y + e y1 = c*x + d*y + f

where (*x*,*y*) represents the coordinates of some point before the transformation is applied, and (*x1*,*y1*) are the transformed coordinates. The transform is defined by the six constants *a*, *b*, *c*, *d*, *e*, and *f*. Note that this can be written as a function **T**, where

T(x,y) = (a*x + b*y + e, c*x + d*y + f)

A transformation of this form is called an affine transform. An affine transform has the property that, when it is applied to two parallel lines, the transformed lines will also be parallel. Also, if you follow one affine transform by another affine transform, the result is again an affine transform.

2.3.2 Translation

A translation transform simply moves every point by a certain amount horizontally and a certain amount vertically. If (*x*,*y*) is the original point and (*x1*,*y1*) is the transformed point, then the formula for a translation is

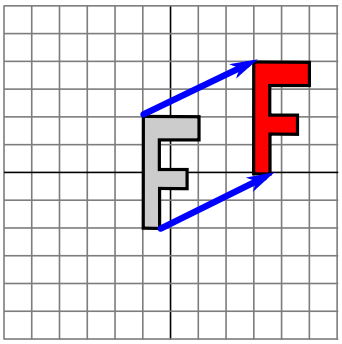
x1 = x + e y1 = y + f

where *e* is the number of units by which the point is moved horizontally and *f* is the amount by which it is moved vertically. (Thus for a translation, $a = d = 1$, and $b = c = 0$ in the general formula for an affine transform.) A 2D graphics system will typically have a function such as

translate(e, f)

to apply a translate transformation. The translation would apply to everything that is drawn **after** the command is given. That is, for all subsequent drawing operations, *e* would be added to the x-coordinate and *f* would be added to the y-coordinate. Let's look at an example. Suppose that you draw an "F" using coordinates in which the "F" is centered at (0,0). If you say *translate*(4,2) **before** drawing the "F", then every point of the

“F” will be moved horizontally by 4 units and vertically by 2 units before the coordinates are actually used, so that after the translation, the “F” will be centered at (4,2):



The light gray “F” in this picture shows what would be drawn without the translation; the dark red “F” shows the same “F” drawn after applying a translation by (4,2). The top arrow shows that the upper left corner of the “F” has been moved over 4 units and up 2 units. Every point in the “F” is subjected to the same displacement. Note that in my examples, I am assuming that the y-coordinate increases from bottom to top. That is, the y-axis points up.

Remember that when you give the command *translate*(*e*,*f*), the translation applies to **all** the drawing that you do after that, not just to the next shape that you draw. If you apply another transformation after the translation, the second transform will not replace the translation. It will be combined with the translation, so that subsequent drawing will be affected by the combined transformation. For example, if you combine *translate*(4,2) with *translate*(-1,5), the result is the same as a single translation, *translate*(3,7). This is an important point, and there will be a lot more to say about it later.

Also remember that you don’t compute coordinate transformations yourself. You just specify the original coordinates for the object (that is, the object coordinates), and you specify the transform or transforms that are to be applied. The computer takes care of applying the transformation to the coordinates. You don’t even need to know the equations that are used for the transformation; you just need to understand what it does geometrically.

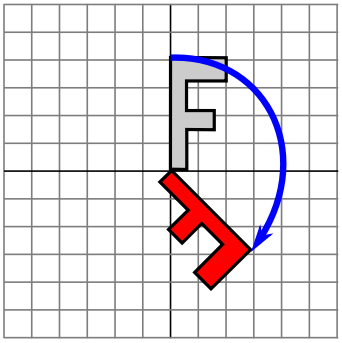
2.3.3 Rotation

A rotation transform, for our purposes here, rotates each point about the origin, (0,0). Every point is rotated through the same angle, called the angle of rotation. For this purpose, angles can be measured either in degrees or in radians. (The 2D graphics APIs for Java and JavaScript that we will look at later in this chapter use radians, but OpenGL

and SVG use degrees.) A rotation with a positive angle rotates objects in the direction from the positive x-axis towards the positive y-axis. This is counterclockwise in a coordinate system where the y-axis points up, as it does in my examples here, but it is clockwise in the usual pixel coordinates, where the y-axis points down rather than up. Although it is not obvious, when rotation through an angle of *r* radians about the origin is applied to the point (*x*,*y*), then the resulting point (*x1*,*y1*) is given by

x1 = cos(r) * x - sin(r) * y y1 = sin(r) * x + cos(r) * y

That is, in the general formula for an affine transform, *e* = *f* = 0, *a* = *d* = cos(*r*), *b* = -sin(*r*), and *c* = sin(*r*). Here is a picture that illustrates a rotation about the origin by the angle negative 135 degrees:



Again, the light gray "F" is the original shape, and the dark red "F" is the shape that results if you apply the rotation. The arrow shows how the upper left corner of the original "F" has been moved.

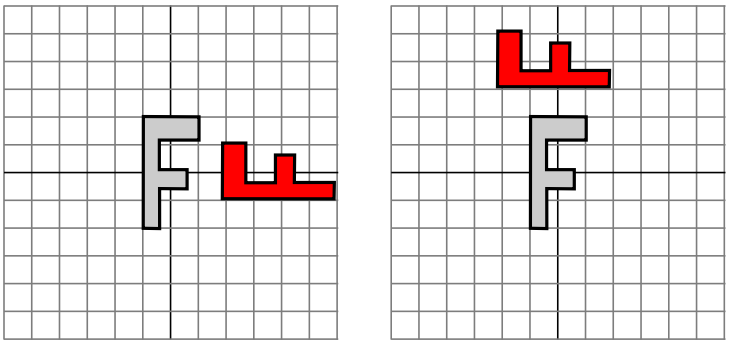
A 2D graphics API would typically have a command *rotate*(*r*) to apply a rotation. The command is used **before** drawing the objects to which the rotation applies.

2.3.4 Combining Transformations

We are now in a position to see what can happen when you combine two transformations. Suppose that before drawing some object, you say

translate(4,0) rotate(90)

Assume that angles are measured in degrees. The translation will then apply to all subsequent drawing. But, because of the rotation command, the things that you draw after the translation are **rotated** objects. That is, the translation applies to objects that have **already** been rotated. An example is shown on the left in the illustration below, where the light gray "F" is the original shape, and red "F" shows the result of applying the two transforms to the original. The original "F" was first rotated through a 90 degree angle, and then moved 4 units to the right.



Note that transforms are applied to objects in the reverse of the order in which they are given in the code (because the first transform in the code is applied to an object that has already been affected by the second transform). And note that the order in which the transforms are applied is important. If we reverse the order in which the two transforms are applied in this example, by saying

rotate(90) translate(4,0)

then the result is as shown on the right in the above illustration. In that picture, the original "F" is first moved 4 units to the right and the resulting shape is then rotated through an angle of 90 degrees about the origin to give the shape that actually appears on the screen.

For another example of applying several transformations, suppose that we want to rotate a shape through an angle *r* about a point (*p*,*q*) instead of about the point (0,0). We can do this by first moving the point (*p*,*q*) to the origin, using *translate*(-*p*,-*q*). Then we can do a standard rotation about the origin by calling *rotate*(*r*). Finally, we can move the origin back to the point (*p*,*q*) by applying *translate*(*p*,*q*). Keeping in mind that we have to write the code for the transformations in the reverse order, we need to say

translate(p,q) rotate(r) translate(-p,-q)

before drawing the shape. (In fact, some graphics APIs let us accomplish this transform with a single command such as *rotate*(*r*,*p*,*q*). This would apply a rotation through the angle *r* about the point (*p*,*q*).)

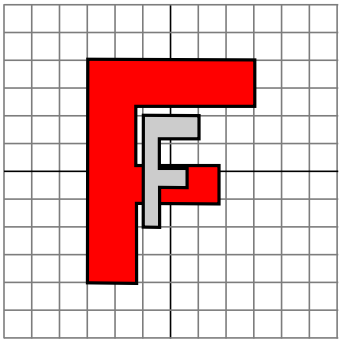
2.3.5 Scaling

A scaling transform can be used to make objects bigger or smaller. Mathematically, a scaling transform simply multiplies each x-coordinate by a given amount and each y-coordinate by a given amount. That is, if a point (*x*,*y*) is scaled by a factor of *a* in the x

direction and by a factor of *d* in the y direction, then the resulting point (*x1*,*y1*) is given by

x1 = a * x y1 = d * y

If you apply this transform to a shape that is centered at the origin, it will stretch the shape by a factor of *a* horizontally and *d* vertically. Here is an example, in which the original light gray "F" is scaled by a factor of 3 horizontally and 2 vertically to give the final dark red "F":



The common case where the horizontal and vertical scaling factors are the same is called uniform scaling. Uniform scaling stretches or shrinks a shape without distorting it.

When scaling is applied to a shape that is not centered at (0,0), then in addition to being stretched or shrunk, the shape will be moved away from 0 or towards 0. In fact, the true description of a scaling operation is that it pushes every point away from (0,0) or pulls every point towards (0,0). If you want to scale about a point other than (0,0), you can use a sequence of three transforms, similar to what was done in the case of rotation.

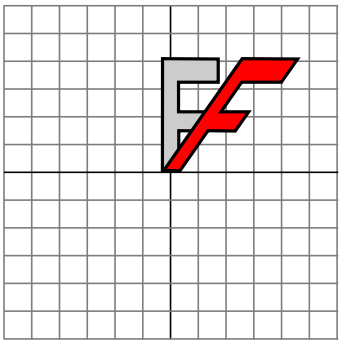
A 2D graphics API can provide a function *scale*(*a*,*d*) for applying scaling transformations. As usual, the transform applies to all *x* and *y* coordinates in subsequent drawing operations. Note that negative scaling factors are allowed and will result in reflecting the shape as well as possibly stretching or shrinking it. For example, *scale*(1,-1) will reflect objects vertically, through the *x*-axis.

It is a fact that every affine transform can be created by combining translations, rotations about the origin, and scalings about the origin. I won't try to prove that, but *c2/transforms2d.html* is an interactive demo that will let you experiment with translations, rotations, and scalings, and with the transformations that can be made by combining them.

I also note that a transform that is made from translations and rotations, with no scaling, will preserve length and angles in the objects to which it is applied. It will also preserve aspect ratios of rectangles. Transforms with this property are called "Euclidean." If you also allow **uniform** scaling, the resulting transformation will preserve angles and aspect ratio, but not lengths.

2.3.6 Shear

We will look at one more type of basic transform, a shearing transform. Although shears can in fact be built up out of rotations and scalings if necessary, it is not really obvious how to do so. A shear will "tilt" objects. A horizontal shear will tilt things towards the left (for negative shear) or right (for positive shear). A vertical shear tilts them up or down. Here is an example of horizontal shear:



A horizontal shear does not move the x-axis. Every other horizontal line is moved to the left or to the right by an amount that is proportional to the y-value along that line. When a horizontal shear is applied to a point (*x*,*y*), the resulting point (*x1*,*y1*) is given by

x1 = x + b * y y1 = y

for some constant shearing factor *b*. Similarly, a vertical shear with shearing factor *c* is given by the equations

x1 = x y1 = c * x + y

Shear is occasionally called "skew," but skew is usually specified as an angle rather than as a shear factor.

2.3.7 Window-to-Viewport

The last transformation that is applied to an object before it is displayed in an image is the window-to-viewport transformation, which maps the rectangular view window in the xy-plane that contains the scene to the rectangular grid of pixels where the image will be displayed. I'll assume here that the view window is not rotated; that it, its sides

are parallel to the xand y-axes. In that case, the window-to-viewport transformation can be expressed in terms of translation and scaling transforms. Let's look at the typical case where the viewport has pixel coordinates ranging from 0 on the left to *width* on the right and from 0 at the top to *height* at the bottom. And assume that the limits on the view window are *left*, *right*, *bottom*, and *top*. In that case, the window-to-viewport transformation can be programmed as:

scale(width / (right-left), height / (bottom-top)); translate(-left, -top)

These should be the last transforms that are applied to a point. Since transforms are applied to points in the reverse of the order in which they are specified in the program, they should be the first transforms that are specified in the program. To see how this works, consider a point (*x*,*y*) in the view window. (This point comes from some object in the scene. Several modeling transforms might have already been applied to the object to produce the point (*x*,*y*), and that point is now ready for its final transformation into viewport coordinates.) The coordinates (*x*,*y*) are first translated by (-*left*,-*top*) to give (*x*-*left*,*y*-*top*). These coordinates are then multiplied by the scaling factors shown above, giving the final coordinates

x1 = width / (right-left) * (x-left) y1 = height / (bottom-top) * (y-top)

Note that the point (*left*,*top*) is mapped to (0,0), while the point (*right*,*bottom*) is mapped to (*width*,*height*), which is just what we want.

There is still the question of aspect ratio. As noted in Subsection 2.1.3, if we want to force the aspect ratio of the window to match the aspect ratio of the viewport, it might be necessary to adjust the limits on the window. Here is pseudocode for a subroutine that will do that, again assuming that the top-left corner of the viewport has pixel coordinates (0,0):

subroutine applyWindowToViewportTransformation (left, right, // horizontal limits on view window bottom, top, // vertical limits on view window width, height, // width and height of viewport preserveAspect // should window be forced to match viewport aspect?)

if preserveAspect :

// Adjust the limits to match the aspect ratio of the drawing area. displayAspect = abs(height / width); windowAspect = abs((top-bottom) / (right-left)); if displayAspect > windowAspect : // Expand the viewport vertically. excess = (top-bottom) * (displayAspect/windowAspect - 1)

top = top + excess/2 bottom = bottom - excess/2

else if displayAspect < windowAspect : // Expand the viewport horizontally. excess = (right-left) * (windowAspect/displayAspect - 1)

right = right + excess/2 left = left - excess/2

scale(width / (right-left), height / (bottom-top)) translate(-left, -top)

2.3.8 Matrices and Vectors

The transforms that are used in computer graphics can be represented as matrices, and the points on which they operate are represented as vectors. Recall that a matrix, from the point of view of a computer scientist, is a two-dimensional array of numbers, while a vector is a onedimensional array. Matrices and vectors are studied in the field of mathematics called linear algebra. Linear algebra is fundamental to computer graphics. In fact, matrix and vector math is built into GPUs. You won't need to know a great deal about linear algebra for this textbook, but a few basic ideas are essential.

The vectors that we need are lists of two, three, or four numbers. They are often written as (x,y), (x,y,z), and (x,y,z,w). A matrix with N rows and M columns is called an "N-by-M matrix." For the most part, the matrices that we need are N-by-N matrices, where N is 2, 3, or 4. That is, they have 2, 3, or 4 rows and columns, and the number of rows is equal to the number of columns.

If A and B are two N-by-N matrices, then they can be multiplied to give a product matrix $C = AB$. If A is an N-by-N matrix, and v is a vector of length N, then v can be multiplied by A to give another vector $w = Av$. The function that takes v to Av is a transformation; it transforms any given vector of size N into another vector of size N. A transformation of this form is called a linear transformation.

Now, suppose that A and B are N-by-N matrices and v is a vector of length N. Then, we can form two different products: $A(Bv)$ and $(AB)v$. It is a central fact that these two operations have the same effect. That is, we can multiply v by B and then multiply the result by A, or we can multiply the matrices A and B to get the matrix product AB and then multiply v by AB. The result is the same.

Rotation and scaling, as it turns out, are linear transformations. That is, the operation of rotating (x,y) through an angle d about the origin can be done by multiplying (x,y) by a 2-by-2 matrix. Let's call that matrix R_d. Similarly, scaling by a factor a in the horizontal direction and b in the vertical direction can be given as a matrix $S_{a,b}$. If we want to apply

a scaling followed by a rotation to the point $v = (x,y)$, we can compute **either** $R_d(S_{a,b}v)$ **or** $(R_dS_{a,b})v$.

So what? Well, suppose that we want to apply the same two operations, scale then rotate, to thousands of points, as we typically do when transforming objects for computer graphics. The point is that we could compute the product matrix $R_dS_{a,b}$ once and for all, and then apply the combined transform to each point with a single multiplication. This means that if a program says

rotate(d) scale(a,b) .

. // draw a complex object

.

the computer doesn't have to keep track of two separate operations. It combines the operations into a single matrix and just keeps track of that. Even if you apply, say, 50 transformations to the object, the computer can just combine them all into one matrix. By using matrix algebra, multiple transformations can be handled as efficiently as a single transformation!

This is really nice, but there is a gaping problem: **Translation is not a linear transformation.** To bring translation into this framework, we do something that looks a little strange at first: Instead of representing a point in 2D as a pair of numbers (x,y), we represent it as the triple of numbers $(x,y,1)$. That is, we add a one as the third coordinate. It then turns out that we can then represent rotation, scaling, and translation—and hence any affine transformation—on 2D space as multiplication by a 3-by-3 matrix. The matrices that we need have a bottom row containing (0,0,1). Multiplying $(x,y,1)$ by such a matrix gives a new vector $(x1,y1,1)$. We ignore the extra coordinate and consider this to be a transformation of (x,y) into $(x1,y1)$. For the record, the 3-by-3 matrices for translation ($T_{a,b}$), scaling ($S_{a,b}$), and rotation (R_d) in 2D are

$$\mathbf{T}_{a,b} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{S}_{a,b} = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{R}_d = \begin{pmatrix} \cos(d) & -\sin(d) & 0 \\ \sin(d) & \cos(d) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

away from this discussion is that a sequence of transformations can be combined into a single transformation. The computer only needs to keep track of a single matrix, which we can call the “current matrix” or “current transformation.” To implement transform commands such as *translate*(a,b) or *rotate*(d), the computer simply multiplies the current matrix by the matrix that represents the transform.

2.4 Hierarchical Modeling

In this section, we look at how complex scenes can be built from very simple shapes. The key is hierarchical structure. That is, a complex object can be made up of simpler objects, which can in turn be made up of even simpler objects, and so on until it bottoms out with simple geometric primitives that can be drawn directly. This is called hierarchical modeling. We will see that the transforms that were studied in the previous section play an important role in hierarchical modeling.

Hierarchical structure is the key to dealing with complexity in many areas of computer science (and in the rest of reality), so it be no surprise that it plays an important role in computer graphics.

2.4.1 Building Complex Objects

A major motivation for introducing a new coordinate system is that it should be possible to use the coordinate system that is most natural to the scene that you want to draw. We can extend this idea to individual objects in a scene: When drawing an object, use the coordinate system that is most natural for the object.

Usually, we want an object in its natural coordinates to be centered at the origin, (0,0), or at least to use the origin as a convenient reference point. Then, to place it in the scene, we can use a scaling transform, followed by a rotation, followed by a translation to set its size, orientation, and position in the scene. Recall that transformations used in this way are called modeling transformations. The transforms are often applied in the order scale, then rotate, then translate, because scaling and rotation leave the reference point, (0,0), fixed. Once the object has been scaled and rotated, it's easy to use a translation to move the reference point to any desired point in the scene. (Of course, in a particular case, you might not need all three operations.) Remember that in the code, the transformations are specified in the opposite order from the order in which they are applied to the object and that the transformations are specified before drawing the object. So in the code, the translation would come first, followed by the rotation and then the scaling. Modeling transforms are not always composed in this order, but it is the most common usage.

The modeling transformations that are used to place an object in the scene should not affect other objects in the scene. To limit their application to just the one object, we can save the current transformation before starting work on the object and restore it afterwards. How this is done differs from one graphics API to another, but let's suppose here that there are subroutines *saveTransform*() and *restoreTransform*() for performing those tasks. That is, *saveTransform* will make a copy of the modeling transformation

that is currently in effect and store that copy. It does not change the current transformation; it merely saves a copy. Later, when *restoreTransform* is called, it will retrieve that copy and will replace the current modeling transform with the retrieved transform. Typical code for drawing an object will then have the form:

saveTransform()

translate(dx,dy) // move object into position rotate(r) // set the orientation of the object scale(sx,sy) // set the size of the object

.

. // draw the object, using its natural coordinates .

restoreTransform()

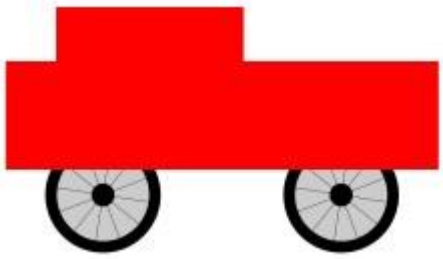
Note that we don't know and don't need to know what the saved transform does. Perhaps it is simply the so-called identity transform, which is a transform that doesn't modify the coordinates to which it is applied. Or there might already be another transform in place, such as a coordinate transform that affects the scene as a whole. The modeling transform for the object is effectively applied in addition to any other transform that was specified previously. The modeling transform moves the object from its natural coordinates into its proper place in the scene. Then on top of that, a coordinate transform that is applied to the scene as a whole would carry the object along with it.

Now let's extend this idea. Suppose that the object that we want to draw is itself a complex entity, made up of a number of smaller objects. Think, for example, of a potted flower made up of pot, stem, leaves, and bloom. We would like to be able to draw the smaller component objects in their own natural coordinate systems, just as we do the main object. For example, we would like to specify the bloom in a coordinate system in which the center of the bloom is at (0,0). But this is easy: We draw each small component object, such as the bloom, in its own coordinate system, and use a modeling transformation to move the sub-object into position **within the main object**. We are composing the complex object in its own natural coordinate system as if it were a complete scene.

On top of that, we can apply **another** modeling transformation to the complex object as a whole, to move it into the actual scene; the sub-objects of the complex object are carried along with it. That is, the overall transformation that applies to a sub-object consists of a modeling transformation that places the sub-object into the complex object, followed by the transformation that places the complex object into the scene.

In fact, we can build objects that are made up of smaller objects which in turn are made up of even smaller objects, to any level. For example, we could draw the bloom's petals in their own coordinate systems, then apply modeling transformations to place the petals into the natural coordinate system for the bloom. There will be another transformation that moves the bloom into position on the stem, and yet another transformation that places the entire potted flower into the scene. This is hierarchical modeling.

Let's look at a little example. Suppose that we want to draw a simple 2D image of a cart with two wheels.



This cart is used as one part of a complex scene in an example below. The body of the cart can be drawn as a pair of rectangles. For the wheels, suppose that we have written a subroutine drawWheel()

that draws a wheel. This subroutine draws the wheel in its own natural coordinate system. In this coordinate system, the wheel is centered at (0,0) and has radius 1.

In the cart's coordinate system, I found it convenient to use the midpoint of the base of the large rectangle as the reference point. I assume that the positive direction of the *y*-axis points upward, which is the common convention in mathematics. The rectangular body of the cart has width 6 and height 2, so the coordinates of the lower left corner of the rectangle are (-3,0), and we can draw it with a command such as *fillRectangle*(-3,0,6,2). The top of the cart is a smaller red rectangle, which can be drawn in a similar way. To complete the cart, we need to add two wheels to the object. To make the size of the wheels fit the cart, they need to be scaled. To place them in the correct positions relative to body of the cart, one wheel must be translated to the left and the other wheel, to the right. When I coded this example, I had to play around with the numbers to get the right sizes and positions for the wheels, and I found that the wheels looked better if I also moved them down a bit. Using the usual techniques of hierarchical modeling, we save the current transform before drawing each wheel, and we restore it after drawing the wheel. This restricts the effect of the modeling transformation for the

wheel to that wheel alone, so that it does not affect any other part of the cart. Here is pseudocode for a subroutine that draws the cart in its own coordinate system:

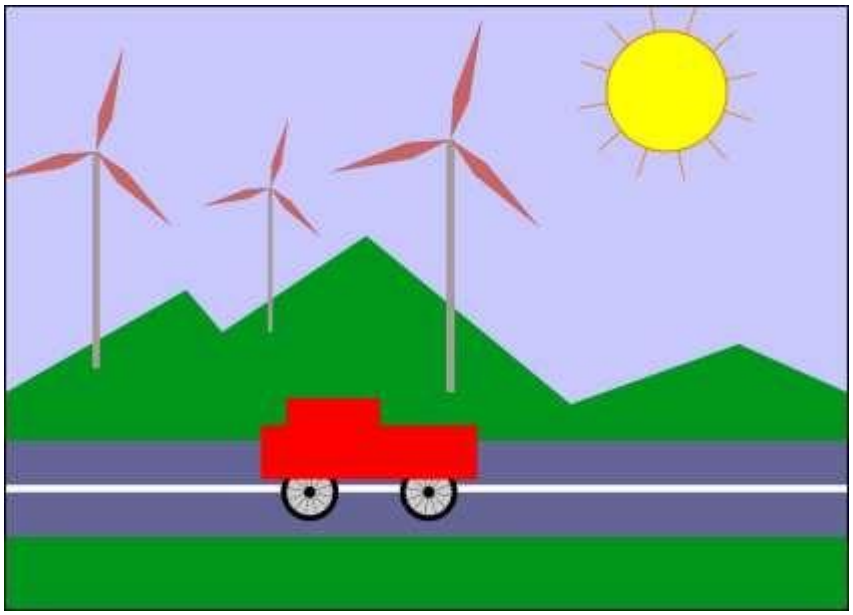
```
subroutine drawCart() :
        saveTransform()        // save the current transform
translate(-1.65,-0.1) // center of first wheel will be at (-1.65,-0.1)
            scale(0.8,0.8)          // scale to reduce radius from 1 to 0.8
            drawWheel()             // draw the first wheel
            restoreTransform()      // restore the saved transform
            saveTransform()         // save it again
            translate(1.5,-0.1)     // center of second wheel will be at (1.5,-0.1)
            scale(0.8,0.8)          // scale to reduce radius from 1 to 0.8
            drawWheel()             // draw the second wheel
            restoreTransform()      // restore the transform
setDrawingColor(RED) // use red color to draw the rectangles fillRectangle(-3, 0, 6, 2)
        // draw the body of the cart fillRectangle(-2.3, 1, 2.6, 1) // draw the top of the cart
```

It's important to note that the same subroutine is used to draw both wheels. The reason that two wheels appear in the picture in different positions is that different modeling transformations are in effect for the two subroutine calls.

Once we have this cart-drawing subroutine, we can use it to add a cart to a scene. When we do this, we apply another modeling transformation to the cart as a whole. Indeed, we could add several carts to the scene, if we wanted, by calling the *drawCart* subroutine several times with different modeling transformations.

You should notice the analogy here: Building up a complex scene out of objects is similar to building up a complex program out of subroutines. In both cases, you can work on pieces of the problem separately, you can compose a solution to a big problem from solutions to smaller problems, and once you have solved a problem, you can reuse that solution in several places.

The demo *c2/cart-and-windmills.html* uses the cart in an animated scene. Here's one of the frames from that demo:



You can probably guess how hierarchical modeling is used to draw the three windmills in this example. There is a *drawWindmill* method that draws a windmill in its own coordinate system. Each of the windmills in the scene is then produced by applying a different modeling transform to the standard windmill. Furthermore, the windmill is itself a complex object that is constructed from several sub-objects using various modeling transformations.

* * *

It might not be so easy to see how different parts of the scene can be animated. In fact, animation is just another aspect of modeling. A computer animation consists of a sequence of frames. Each frame is a separate image, with small changes from one frame to the next. From our point of view, each frame is a separate scene and has to be drawn separately. The same object can appear in many frames. To animate the object, we can simply apply a different modeling transformation to the object in each frame. The parameters used in the transformation can be computed from the current time or from the frame number. To make a cart move from left to right, for example, we might apply a modeling transformation translate(frameNumber * 0.1, 0)

to the cart, where *frameNumber* is the frame number. In each frame, the cart will be 0.1 units farther to the right than in the previous frame. (In fact, in the actual program, the translation that is applied to the cart is translate(-3 + 13*(frameNumber % 300) / 300.0, 0)

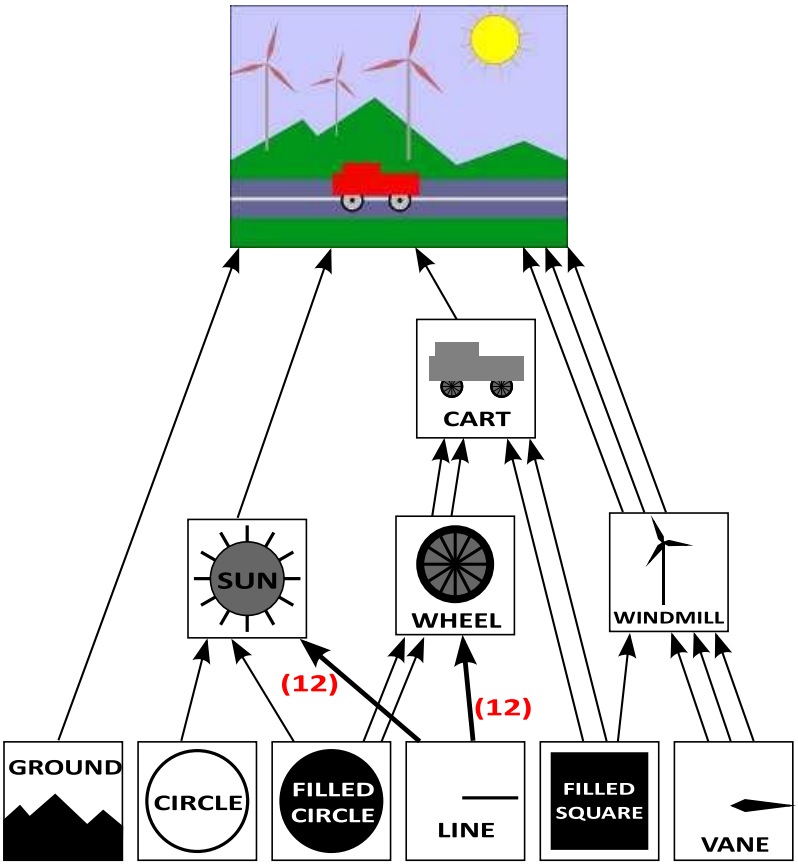
which moves the reference point of the cart from -3 to 13 along the horizontal axis every 300 frames. In the coordinate system that is used for the scene, the x-coordinate ranges from 0 to 7, so this puts the cart outside the scene for much of the loop.)

The really neat thing is that this type of animation works with hierarchical modeling. For example, the *drawWindmill* method doesn't just draw a windmill—it draws an *animated* windmill, with turning vanes. That just means that the rotation applied to the vanes depends on the frame number. When a modeling transformation is applied to the windmill, the rotating vanes are scaled and moved as part of the object as a whole. This is an example of hierarchical modeling. The vanes are sub-objects of the windmill. The rotation of the vanes is part of the modeling transformation that places the vanes into the windmill object. Then a further modeling transformation is applied to the windmill object to place it in the scene.

The file *java2d/HierarchicalModeling2D.java* contains the complete source code for a Java version of this example. The next section of this book covers graphics programming in Java. Once you are familiar with that, you should take a look at the source code, especially the *paintComponent*() method, which draws the entire scene. The same example, using the same scene graph API, is implemented in JavaScript in *canvas2d/HierarchicalModel2D.html*.

2.4.2 Scene Graphs

Logically, the components of a complex scene form a structure. In this structure, each object is associated with the sub-objects that it contains. If the scene is hierarchical, then the structure is hierarchical. This structure is known as a scene graph. A scene graph is a tree-like structure, with the root representing the entire scene, the children of the root representing the top-level objects in the scene, and so on. We can visualize the scene graph for our sample scene:



In this drawing, a single object can have several connections to one or more parent objects. Each connection represents one occurrence of the object in its parent object. For example, the “filled square” object occurs as a sub-object in the cart and in the windmill. It is used twice in the cart and once in the windmill. (The cart contains two red rectangles, which are created as squares with a non-uniform scaling; the pole of the windmill is made as a scaled square.) The “filled circle” is used in the sun and is used twice in the wheel. The “line” is used 12 times in the sun and 12 times in the wheel; I’ve drawn one thick arrow, marked with a 12, to represent 12 connections. The wheel, in turn, is used twice in the cart. (My diagram leaves out, for lack of space, two occurrences of the filled square in the scene: It is used to make the road and the line down the middle of the road.)

Each arrow in the picture can be associated with a modeling transformation that places the sub-object into its parent object. When an object contains several copies of a sub-object, each arrow connecting the sub-object to the object will have a different associated modeling transformation. The object is the same for each copy; only the transformation differs.

Although the scene graph exists conceptually, in some applications it exists only implicitly. For example, the Java version of the program that was mentioned above draws the image "procedurally," that is, by calling subroutines. There is no data structure to represent the scene graph. Instead, the scene graph is implicit in the sequence of subroutine calls that draw the scene. Each node in the graph is a subroutine, and each arrow is a subroutine call. The various objects are drawn using different modeling transformations. As discussed in Subsection 2.3.8, the computer only keeps track of a "current transformation" that represents all the transforms that are applied to an object. When an object is drawn by a subroutine, the program saves the current transformation before calling the subroutine. After the subroutine returns, the saved transformation is restored. Inside the subroutine, the object is drawn in its own coordinate system, possibly calling other subroutines to draw sub-objects with their own modeling transformations. Those extra transformations will have no effect outside of the subroutine, since the transform that is in effect before the subroutine is called will be restored after the subroutine returns.

It is also possible for a scene graph to be represented by an actual data structure in the program. In an object-oriented approach, the graphical objects in the scene are represented by program objects. There are many ways to build an object-oriented scene graph API. For a simple example implemented in Java, you can take a look at *java2d/SceneGraphAPI2D.java*. This program draws the same animated scene as the previous example, but it represents the scene with an object-oriented data structure rather than procedurally. The same scene graph API is implemented in JavaScript in the live demo *c2/cart-and-windmills.html*, and you might take a look at its source code after you read about HTML canvas graphics in Section 2.6.

In the example program, both in Java and in JavaScript, a node in the scene graph is represented by an object belonging to a class named SceneGraphNode. SceneGraphNode is an abstract class, and actual nodes in the scene graph are defined by subclasses of that class. For example, there is a subclass named CompoundObject to represent a complex graphical object that is made up of sub-objects. A variable, *obj*, of type CompoundObject includes a method *obj.add*(*subobj*) for adding a sub-object to the compound object.

When implementing a scene graph as a data structure made up of objects, a decision has to be made about how to handle transforms. One option is to allow transformations to be associated with any node in the scene graph. In this case, however, I decided to use special nodes to represent transforms as objects of type TransformedObject. A TransformedObject is a SceneGraphNode that contains a link to another

SceneGraphNode and also contains a modeling transformation that is to be applied to that object. The modeling transformation is given in terms of scaling, rotation, and translation amounts that are instance variables in the object. It is worth noting that these are always applied in the order scale, then rotate, then translate, no matter what order the instance variables are set in the code. If you want to do a translation followed by a rotation, you will need two TransformedObjects to implement it, since a translation plus a rotation in the same TransformedObject would be applied in the order rotate-then-translate. It is also worth noting that the setter methods for the scaling, rotation, and translation have a return value that is equal to the object. This makes it possible to chain calls to the methods into a single statement such as transformedObject.setScale(5,2).setTranslation(3.5,0);

and even say things like

world.add(new TransformedObject(windmill).setScale(0.4,0.4).setTranslation(2.2,1.3));

This type of chaining can make for more compact code and can eliminate the need for a lot of extra temporary variables.

Another decision has to be made about how to handle color. One possibility would be to make a ColoredObject class similar to TransformedObject. However, in this case I just added a *setColor*() method to the main ScreenGraphNode class. A color that is set on a compound object is inherited by any sub-objects, unless a different color is set on the sub-object. In other words, a color on a compound object acts as a default color for its sub-objects, but color can be overridden on the sub-objects.

In addition to compound objects and transformed objects, we need scene graph nodes to represent the basic graphical objects that occupy the bottom level of the scene graph. These are the nodes that do the actual drawing in the end.

For those who are familiar with data structures, I will note that a scene graph is actually an example of a "directed acyclic graph" or "dag." The process of drawing the scene involves a traversal of this dag. The term "acyclic" means that there can't be cycles in the graph. For a scene graph, this is the obvious requirement that an object cannot be a sub-object, either directly or indirectly, of itself.

2.4.3 The Transform Stack

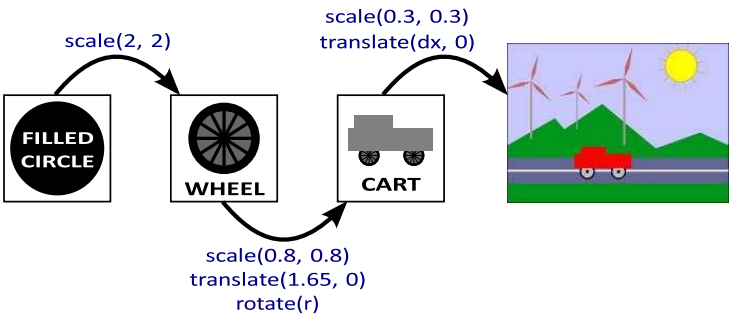
Suppose that you write a subroutine to draw an object. At the beginning of the subroutine, you use a routine such as *saveTransform*() to save a copy of the current transform. At the end of the subroutine, you call *restoreTransform*() to reset the current

transform back to the value that was saved. Now, in order for this to work correctly for hierarchical graphics, these routines must actually use a stack of transforms. (Recall that a stack is simply a list where items can be added, or "pushed," onto one end of the list and removed, or "popped," from the same end.) The problem is that when drawing a complex object, one subroutine can call other subroutines. This means that several drawing subroutines can be active at the same time, each with its own saved transform. When a transform is saved after another transform has already been saved, the system needs to remember both transforms. When *restoreTransform*() is called, it is the most recently saved transform that should be restored.

A stack has exactly the structure that is needed to implement these operations. Before you start drawing an object, you would push the current transform onto the stack. After drawing the object, you would pop the transform from the stack. Between those two operations, if the object is hierarchical, the transforms for its sub-objects will have been pushed onto and popped from the stack as needed.

Some graphics APIs come with transform stacks already defined. For example, the original OpenGL API includes the functions *glPushMatrix*() and *glPopMatrix*() for using a stack of transformation matrices that is built into OpenGL. The Java Graphics2D API does not include a built-in stack of transforms, but it does have methods for getting and setting the current transform, and the get and set methods can be used with an explicit stack data structure to implement the necessary operations. When we turn to the HTML canvas API for 2D graphics, we'll see that it includes functions named *save*() and *restore*() that are actually *push* and *pop* operations on a stack. These functions are essential to implementing hierarchical graphics for an HTML canvas.

Let's try to bring this all together by considering how it applies to a simple object in a complex scene: one of the filled circles that is part of the front wheel on the cart in our example scene. Here, I have rearranged part of the scene graph for that scene, and I've added labels to show the modeling transformations that are applied to each object:



The rotation amount for the wheel and the translation amount for the cart are shown as variables, since they are different in different frames of the animation. When the computer starts drawing the scene, the modeling transform that is in effect is the identity transform, that is, no transform at all. As it prepares to draw the cart, it saves a copy of the current transform (the identity) by pushing it onto the stack. It then modifies the current transform by multiplying it by the modeling transforms for the cart, *scale*(0.3,0.3) and *translate*(dx,0). When it comes to drawing the wheel, it again pushes the current transform (the modeling transform for the cart as a whole) onto the stack, and it modifies the current transform to take the wheel's modeling transforms into account. Similarly, when it comes to the filled circle, it saves the modeling transform for the wheel, and then applies the modeling transform for the circle.

When, finally, the circle is actually drawn in the scene, it is transformed by the combined transform. That transform places the circle directly into the scene, but it has been composed from the transform that places the circle into the wheel, the one that places the wheel into the cart, and the one that places the cart into the scene. After drawing the circle, the computer replaces the current transform with one it pops from the stack. That will be the modeling transform for the wheel as a whole, and that transform will be used for any further parts of the wheel that have to be drawn. When the wheel is done, the transform for the cart is popped. And when the cart is done, the original transform, the identity, is popped. When the computer goes onto the next object in the scene, it starts the whole process again, with the identity transform as the starting point.

This might sound complicated, but I should emphasize that it is something that the computer does for you. Your responsibility is simply to design the individual objects, in their own natural coordinate system. As part of that, you specify the modeling transformations that are applied to the sub-objects of that object. You construct the scene as a whole in a similar way. The computer will then put everything together for you, taking into account the many layers of hierarchical structure. You only have to deal with one component of the structure at a time. That's the power of hierarchical design; that's how it helps you deal with complexity.

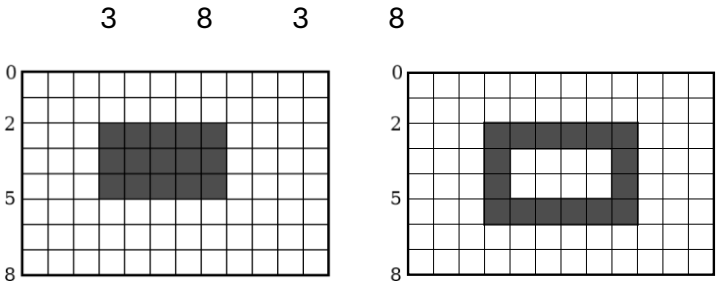
2.5 Java Graphics2D

In the rest of this chapter, we look at specific implementations of two-dimensional graphics. There are a few new ideas here, but mostly you will see how the general concepts that we have covered are used in several real graphics systems.

In this section, our focus is on the Java programming language. Java remains one of the most popular programming languages. Its standard desktop version includes a sophisticated 2D graphics API, which is our topic here. Before reading this section, you should already know the basics of Java programming. But even if you don't, you should be able to follow most of the discussion of the graphics API itself. (See Section A.1 in Appendix A for a very basic introduction to Java.)

The graphics API that is discussed here is part of Swing, an API for graphical user interface programming that is included as part of the standard distribution of Java. Many Java programs are now written using an alternative API called JavaFX, which is not part of the standard distribution. JavaFX is not discussed in this textbook. Its graphics API is, in fact, quite similar to the API for HTML canvas graphics, which is discussed in Section 2.6.

The original version of Java had a much smaller graphics API. It was tightly focused on pixels, and it used only integer coordinates. The API had subroutines for stroking and filling a variety of basic shapes, including lines, rectangles, ovals, and polygons (although Java uses the term *draw* instead of *stroke*). Its specification of the meaning of drawing operations was very precise on the pixel level. Integer coordinates are defined to refer to the lines between pixels. For example, a 12-by-8 pixel grid has *x*-coordinates from 0 to 12 and *y*-coordinates from 0 to 8, as shown below. The lines between pixels are numbered, not the pixels.



The command *fillRect*(3,2,5,3) fills the rectangle with upper left corner at (3,2), with width 5, and with height 3, as shown on the left above. The command *drawRect*(3,2,5,3) conceptually drags a "pen" around the outline of this rectangle. However, the pen is a 1-pixel square, and it is the upper left corner of the pen that moves along the outline. As

the pen moves along the right edge of the rectangle, the pixels to the *right* of that edge are colored; as the pen moves along the bottom edge, the pixels below the edge are colored. The result is as shown on the right above. My point here is not to belabor the details, but to point out that having a precise specification of the meaning of graphical operations gives you very fine control over what happens on the pixel level.

Java's original graphics did not support things like real-number coordinates, transforms, antialiasing, or gradients. Just a few years after Java was first introduced, a new graphics API was added that does support all of these. It is that more advanced API that we will look at here.

2.5.1 Graphics2D

Java is an object-oriented language. Its API is defined as a large set of classes, The actual drawing operations in the original graphics API were mostly contained in the class named Graphics. In the newer Swing API, drawing operations are methods in a class named Graphics2D, which is a subclass of Graphics, so that all the original drawing operations are still available. (A class in Java is contained in a collection of classes known as a "package." Graphics and Graphics2D, for example, are in the package named *java.awt*. Classes that define shapes and transforms are in a package named *java.awt.geom*.)

A graphics system needs a place to draw. In Java, the drawing surface is often an object of the class JPanel, which represents a rectangular area on the screen. The JPanel class has a method named *paintComponent*() to draw its content. To create a drawing surface, you can create a subclass of JPanel and provide a definition for its *paintComponent*() method. All drawing should be done inside *paintComponent*(); when it is necessary to change the contents of the drawing, you can call the panel's *repaint*() method to trigger a call to *paintComponent*(). The *paintComponent*() method has a parameter of type Graphics, but the parameter that is passed to the method is actually an object of type Graphics2D, and it can be type-cast to Graphics2D to obtain access to the more advanced graphics capabilities. So, the definition of the *paintComponent*() method usually looks something like this:

protected void paintComponent(Graphics g) { Graphics2D g2; g2 = (Graphics2D)g; // Type-cast the parameter to Graphics2D.

.

. // Draw using g2.

.

}

In the rest of this section, I will assume that *g2* is a variable of type Graphics2D, and I will discuss some of the things that you can do with it. As a first example, I note that Graphics2D supports antialiasing, but it is not turned on by default. It can be enabled in a graphics context *g2* with the rather intimidating command

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
```

For simple examples of graphics in complete Java programs, you can look at the sample programs *java2d/GraphicsStarter.java* and *java2d/AnimationStarter.java*. They provide very minimal frameworks for drawing static and animated images, respectively, using Graphics2D. The program *java2d/EventsStarter.java* is a similar framework for working with mouse and key events in a graphics program. You can use these programs as the basis for some experimentation if you want to explore Java graphics.

2.5.2 Shapes

Drawing with the original Graphics class is done using integer coordinates, with the measurement given in pixels. This works well in the standard coordinate system, but is not appropriate when real-number coordinates are used, since the unit of measure in such a coordinate system will not be equal to a pixel. We need to be able to specify shapes using real numbers. The Java package *java.awt.geom* provides support for shapes defined using real number coordinates. For example, the class Line2D in that package represents line segments whose endpoints are given as pairs of real numbers.

Now, Java has two real number types: double and float. The double type can represent a larger range of numbers than float, with a greater number of significant digits, and double is the more commonly used type. In fact, doubles are simply easier to use in Java. However, float values generally have enough accuracy for graphics applications, and they have the advantage of taking up less space in memory. Furthermore, computer graphics hardware often uses float values internally.

So, given these considerations, the *java.awt.geom* package actually provides two versions of each shape, one using coordinates of type float and one using coordinates of type double. This is done in a rather strange way. Taking Line2D as an example, the class Line2D itself is an abstract class. It has two subclasses, one that represents lines using float coordinates and one using double coordinates. The strangest part is that these subclasses are defined as nested classes inside Line2D: Line2D.Float and

Line2D.Double. This means that you can declare a variable of type Line2D, but to create an object, you need to use Line2D.Double or Line2D.Float:

Line2D line1, line2; line1 = new Line2D.Double(1,2,5,7); // Line from (1.0,2.0) to (5.0,7.0) line2 = new Line2D.Float(2.7F,3.1F,1.5F,7.1F); // (2.7,3.1) to (1.5,7.1)

Note that when using constants of type float in Java, you have to add "F" as a suffix to the value. This is one reason why doubles are easier in Java. For simplicity, you might want to stick to using Line2D.Double. However, Line2D.Float might give slightly better performance.

* * *

Let's take a look at some of the other classes from *java.awt.geom*. The abstract class Point2D—with its concrete subclasses Point2D.Double and Point2D.Float—represents a point in two dimensions, specified by two real number coordinates. A point is not a shape; you can't fill or stroke it. A point can be constructed from two real numbers ("new Point2D.Double(1.2,3.7)"). If *p* is a variable of type Point2D, you can use *p.getX*() and *p.getY* () to retrieve its coordinates, and you can use *p.setX*(*x*), *p.setY* (*y*), or *p.setLocation*(*x*,*y*) to set its coordinates. If *pd* is a variable of type Point2D.Double, you can also refer directly to the coordinates as *pd.x* and *pd.y* (and similarly for Point2D.Float). Other classes in *java.awt.geom* offer a similar variety of ways to manipulate their properties, and I won't try to list them all here.

There is a variety of classes that represent geometric shapes, including Line2D, Rectangle2D, RoundRectangle2D, Ellipse2D, Arc2D, and Path2D. All of these are abstract classes, and each of them contains a pair of subclasses such as Rectangle2D.Double and Rectangle2D.Float. Some shapes, such as rectangles, have interiors that can be filled; such shapes also have outlines that can be stroked. Some shapes, such as lines, are purely one-dimensional and can only be stroked. Aside from lines, rectangles are probably the simplest shapes. A Rectangle2D has a corner point (*x*,*y*), a *width*, and a *height*, and can be constructed from that data ("new Rectangle2D.Double(x,y,w,h)"). The corner point (*x*,*y*) specifies the minimum *x*and *y*-values in the rectangle. For the usual pixel coordinate system, (*x*,*y*) is the upper left corner. However, in a coordinate system in which the minimum value of *y* is at the bottom, (*x*,*y*) would be the lower left corner. The sides of the rectangle are parallel to the coordinate axes. A variable *r* of type Rectangle2D.Double has public instance variables *r.x*, *r.y*, *r.width*, and *r.height*. If the width or the height is less than or equal to zero, nothing will be drawn when the rectangle is filled or stroked. A common task is to define a rectangle from two corner points (*x1*,*y1*) and (*x2*,*y2*). This can be accomplished by

creating a rectangle with height and width equal to zero and then *adding* the second point to the rectangle. Adding a point to a rectangle causes the rectangle to grow just enough to include that point:

Rectangle2D.Double r = new Rectangle2D.Double(x1,y1,0,0); r.add(x2,y2);

The classes Line2D, Ellipse2D, RoundRectangle2D and Arc2D create other basic shapes and work similarly to Rectangle2D. You can check the Java API documentation for details.

The Path2D class is more interesting. It represents general paths made up of segments that can be lines and Bezier curves. Paths are created using methods similar to the *moveTo* and *lineTo* subroutines that were discussed in Subsection 2.2.3. To create a path, you start by constructing an object of type Path2D.Double (or Path2D.Float):

Path2D.Double p = new Path2D.Double();

The path *p* is empty when it is first created. You construct the path by moving an imaginary "pen" along the path that you want to create. The method *p.moveTo*(*x*,*y*) moves the pen to the point (*x*,*y*) without drawing anything. It is used to specify the initial point of the path or the starting point of a new piece of the path. The method *p.lineTo*(*x*,*y*) draws a line from the current pen position to (*x*,*y*), leaving the pen at (*x*,*y*). The method *p.close*() can be used to close the path (or the current piece of the path) by drawing a line back to its starting point. For example, the following code creates a triangle with vertices at (0,5), (2,-3), and (-4,1):

Path2D.Double p = new Path2D.Double(); p.moveTo(0,5);

p.lineTo(2,-3);

p.lineTo(-4,1);

p.close();

You can also add Bezier curve segments to a Path2D. Bezier curves were discussed in Subsection 2.2.3. You can add a cubic Bezier curve to a Path2D *p* with the method

p.curveTo(cx1, cy1, cx2, cy2, x, y);

This adds a curve segment that starts at the current pen position and ends at (*x*,*y*), using (*cx1*,*cy1*) and (*cx2*,*cy2*) as the two control points for the curve. The method for adding a quadratic Bezier curve segment to a path is *quadTo*. It requires only a single control point:

p.quadTo(cx, cy, x, y);

When a path intersects itself, its interior is determined by looking at the winding number, as discussed in Subsection 2.2.2. There are two possible rules for determining whether a point is interior: asking whether the winding number of the curve about that point is non-zero, or asking whether it is odd. You can set the winding rule used by a Path2D *p* with

p.setWindingRule(Path2D.WINDNON ZERO);

p.setWindingRule(Path2D.WIND_EVEN ODD);

The default is WINĐ NƟN ZERO.

Finally, I will note that it is possible to draw a copy of an image into a graphics context. The image could be loaded from a file or created by the program. I discuss the second possibility later in this section. An image is represented by an object of type Image. In fact, I will assume here that the object is of type BufferedImage, which is a subclass of Image. If *img* is such an object, then g2.drawImage(img, x, y, null);

will draw the image with its upper left corner at the point (*x*,*y*). (The fourth parameter is hard to explain, but it should be specified as *null* for BufferedImages.) This draws the image at its natural width and height, but a different width and height can be specified in the method:

g2.drawImage(img, x, y, width, height, null);

There is also a method for drawing a string of text. The method specifies the string and the basepoint of the string. (The basepoint is the lower left corner of the string, ignoring

“descenders” like the tail on the letter “g”.) For example, g2.drawString("Hello World", 100, 50);

Images and strings are subject to transforms in the same way as other shapes. Transforms are the only way to get rotated text and images. As an example, here is what can happen when you apply a rotation to some text and an image:

The Space Station

2.5.3 Stroke and Fill

Once you have an object that represents a shape, you can fill the shape or stroke it. The Graphics2D class defines methods for doing this. The method for stroking a shape is called *draw*:

g2.fill(shape); g2.draw(shape);

Here, *g2* is of type Graphics2D, and shape can be of type Path2D, Line2D, Rectangle2D or any of the other shape classes. These are often used on a newly created object, when that object represents a shape that will only be drawn once. For example g2.draw(new Line2D.Double(-5, -5, 5, 5));

Of course, it is also possible to create shape objects and reuse them many times.

The "pen" that is used for stroking a shape is usually represented by an object of type BasicStroke. The default stroke has line width equal to 1. That's one unit in the current coordinate system, not one pixel. To get a line with a different width, you can install a new stroke with g2.setStroke(new BasicStroke(width));

The *width* in the constructor is of type float. It is possible to add parameters to the constructor to control the shape of a stroke at its endpoints and where two segments meet.

(See Subsection 2.2.1.) For example,

g2.setStroke(new BasicStroke(5.0F,

BasicStroke.CAP_ROUND, BasicStroke.JOIN_BEVEL));

It is also possible to make strokes out of dashes and dots, but I won't discuss how to do it here.

* * *

Stroking or filling a shape means setting the colors of certain pixels. In Java, the rule that is used for coloring those pixels is called a “paint.” Paints can be solid colors, gradients, or patterns. Like most things in Java, paints are represented by objects. If *paint* is such an object, then g2.setPaint(paint);

will set *paint* to be used in the graphics context *g2* for subsequent drawing operations, until the next time the paint is changed. (There is also an older method, *g2.setColor*(*c*), that works only for colors and is equivalent to calling *g2.setPaint*(*c*).)

Solid colors are represented by objects of type Color. A color is represented internally as an RGBA color. An opaque color, with maximal alpha component, can be created using the constructor

new Color(r, g, b);

where *r*, *g*, and *b* are integers in the range 0 to 255 that give the red, green, and blue components of the color. To get a translucent color, you can add an alpha component, also in the range 0 to 255:

new Color(r, b, g, a);

There is also a function, *Color.getHSBColor*(*h,s,b*), that creates a color from values in the HSB color model (which is another name for HSV). In this case, the hue, saturation, and brightness color components must be given as values of type float. And there are constants to represent about a dozen common colors, such as *Color.WHITE*, *Color.RED*, and *Color.YELLOW*. For example, here is how I might draw a square with a black outline and a light blue interior:

Rectangle2D square = new Rectangle2D.Double(-2,-2,4,4); g2.setPaint(new Color(200,200,255)); g2.fill(square);

g2.setStroke(new BasicStroke(0.1F)); g2.setPaint(Color.BLACK); g2.draw(square);

Beyond solid colors, Java has the class GradientPaint, to represent simple linear gradients, and TexturePaint to represent pattern fills. (Image patterns used in a similar way in 3D graphics are called textures.) Gradients and patterns were discussed in Subsection 2.2.2. For these paints, the color that is applied to a pixel depends on the coordinates of the pixel.

To create a TexturePaint, you need a BufferedImage object to specify the image that it will use as a pattern. You also have to say how coordinates in the image will map to drawing coordinates in the display. You do this by specifying a rectangle that will hold one copy of the image. So the constructor takes the form:

new TexturePaint(image, rect);

where *image* is the BufferedImage and *rect* is a Rectangle2D. Outside that specified rectangle, the image is repeated horizontally and vertically. The constructor for a GradientPaint takes the form

new GradientPaint(x1, y1, color1, x2, y2, color2, cyclic)

Here, *x1*, *y1*, *x2*, and *y2* are values of type float; *color1* and *color2* are of type Color; and *cyclic* is boolean. The gradient color will vary along the line segment from the point (*x1*,*y1*) to the point (*x2*,*y2*). The color is *color1* at the first endpoint and is *color2* at the second endpoint. Color is constant along lines perpendicular to that line segment. The boolean parameter *cyclic* says whether or not the color pattern repeats. As an example, here is a command that will install a GradientPaint into a graphics context:

g2.setPaint(new GradientPaint(0,0, Color.BLACK, 200,100, Color.RED, true));

You should, by the way, note that the current paint is used for strokes as well as for fills.

The sample Java program *java2d/PaintDemo.java* displays a polygon filled with a GradientPaint or a TexturePaint and lets you adjust their properties. The image files *java2d/QueenOfHearts.png* and *java2d/TinySmiley.png* are part of that program, and they must be in the same location as the compiled class files that make up that program when it is run.

2.5.4 Transforms

Java implements geometric transformations as methods in the Graphics2D class. For example, if *g2* is a Graphics2D, then calling *g2.translate*(1,3) will apply a translation by (1,3) to objects that are drawn after the method is called. The methods that are available correspond to the transform functions discussed in Section 2.3:

- g2.scale(sx,sy) — scales by a horizontal scale factor *sx* and a vertical scale factor *sy*.
- g2.rotate(r) — rotates by the angle *r* about the origin, where the angle is measured in radians. A positive angle rotates the positive x-axis in the direction of the positive y-axis.
- g2.rotate(r,x,y) — rotates by the angle *r* about the point (*x*,*y*).
- g2.translate(dx,dy) — translates by *dx* horizontally and *dy* vertically.

- g2.shear(sx,sy) — applies a horizontal shear amount *sx* and a vertical shear amount *sy*. (Usually, one of the shear amounts is 0, giving a pure horizontal or vertical shear.)

A transform in Java is represented as an object of the class AffineTransform. You can create a general affine transform with the constructor

AffineTransform trns = new AffineTransform(a,b,c,d,e,f);

The transform *trns* will transform a point (*x,y*) to the point (*x1,y1*) given by

x1 = a*x + c*y + e y1 = b*x + d*y + f;

You can apply the transform *trns* to a graphics context *g2* by calling *g2.transform*(*trns*).

The graphics context *g2* includes the current affine transform, which is the composition of all the transforms that have been applied. Commands such as *g2.rotate* and *g2.transform* modify the current transform. You can get a copy of the current transform by calling *g2.getTransform*(), which returns an AffineTransform object. You can set the current transform using *g2.setTransform*(*trns*). This replaces the current transform in *g2* with the AffineTransform *trns*. (Note that *g2.setTransform*(*trns*) is different from *g2.transform*(*trns*); the first command **replaces** the current transform in *g2*, while the second **modifies** the current transform by composing it with *trns*.)

The *getTransform* and *setTransform* methods can be used to implement hierarchical modeling. The idea, as discussed in Section 2.4, is that before drawing an object, you should save the current transform. After drawing the object, restore the saved transform. Any additional modeling transformations that are applied while drawing the object and its sub-objects will have no effect outside the object. In Java, this looks like

AffineTransform savedTransform = g2.getTransform(); drawObject();

g2.setTransform(savedTransform);

For hierarchical graphics, we really need a stack of transforms. However, if the hierarchy is implemented using subroutines, then the above code would be part of a subroutine, and the value of the local variable *savedTransform* would be stored on the subroutine call stack. Effectively, we would be using the subroutine call stack to implement the stack of saved transforms.

In addition to modeling transformations, transforms are used to set up the window-toviewport transformation that establishes the coordinate system that will be used for drawing. This is usually done in Java just after the graphics context has been created,

before any drawing operations. It can be done with a Java version of the *applyWindowToViewportTransformation* function from Subsection 2.3.7. See the sample program *java2d/GraphicsStarter.java* for an example.

* * *

I will mention one more use for AffineTransform objects: Sometimes, you do need to explicitly transform coordinates. For example, given object coordinates (*x*,*y*), I might need to know where they will actually end up on the screen, in pixel coordinates. That is, I would like to transform (*x*,*y*) by the current transform to get the corresponding pixel coordinates. The AffineTransform class has a method for applying the affine transform to a point. It works with objects of type Point2D. Here is an example:

AffineTransform trns = g2.getTransform();

Point2D.Double originalPoint = new Point2D.Double(x,y); Point2D.Double transformedPoint = new Point2D.Double(); trns.transform(originalPoint, transformedPoint);

// transformedPoint now contains the pixel coords corresponding to (x,y) int pixelX = (int)transformedPoint.x; int pixelY = (int)transformedPoint.y;

One way I have used this is when working with strings. Often when displaying a string in a transformed coordinate system, I want to transform the basepoint of a string, but not the string itself. That is, I want the transformation to affect the location of the string but not its size or rotation. To accomplish this, I use the above technique to obtain the pixel coordinates for the transformed basepoint, and then draw the string at those coordinates, using an original, untransformed graphics context.

The reverse operation is also sometimes necessary. That is, given pixel coordinates (*px*,*py*), find the point (*x*,*y*) that is transformed to (*px*,*py*) by a given affine transform. For example, when implementing mouse interaction, you will generally know the pixel coordinates of the mouse, but you will want to find the corresponding point in your own chosen coordinate system. For that, you need an inverse transform. The inverse of an affine transform **T** is another transform that performs the opposite transformation. That is, if **T**(*x*,*y*) = (*px*,*py*), and if **R** is the inverse transform, then **R**(*px*,*py*) = (*x*,*y*). In Java, the inverse transform of an AffineTransform *trns* can be obtained with

AffineTransform inverse = trns.createInverse();

(A final note: The older drawing methods from Graphics, such as *drawLine*, use integer coordinates. It's important to note that any shapes drawn using these older methods

are subject to the same transformation as shapes such as Line2D that are specified with real number coordinates. For example, drawing a line with *g.drawLine*(1,2,5,7) will have the same effect as drawing a Line2D that has endpoints (1.0,2.0) and (5.0,7.0). In fact, all drawing is affected by the transformation of coordinates.)

2.5.5 BufferedImage and Pixels

In some graphics applications, it is useful to be able to work with images that are not visible on the screen. That is, you need what I call an off-screen canvas. You also need a way to quickly copy the off-screen canvas onto the screen. For example, it can be useful to store a copy of the on-screen image in an off-screen canvas. The canvas is the official copy of the image. Changes to the image are made to the canvas, then copied to the screen. One reason to do this is that you can then draw extra stuff on top of the screen image without changing the official copy. For example, you might draw a box around a selected region in the on-screen image. You can do this without damaging the official copy in the off-screen canvas. To remove the box from the screen, you just have to copy the off-screen canvas image onto the screen.

In Java, an off-screen image can be implemented as an object of type BufferedImage. A BufferedImage represents a region in memory where you can draw, in exactly the same way that you can draw to the screen. That is, you can obtain a graphics context *g2* of type Graphics2D that you can use for drawing on the image. A BufferedImage is an Image, and you can draw it onto the screen—or into any other graphics context—like any other Image, that is, by using the *drawImage* method of the graphics context where you want to display the image. In a typical setup, there are variables

```
BufferedImage OSC; // The off-screen canvas.
      Graphics2D OSG;     // graphics context for drawing to the canvas
```

The objects are created using, for example,

```
OSC = new BufferedImage( 640, 480, BufferedImage.TYPE_INT_RGB ); OSG = OSC.createGraphics();
```

The constructor for BufferedImage specifies the width and height of the image along with its type. The type tells what colors can be represented in the image and how they are stored. Here, the type is TYPE INT RGB, which means the image uses regular RGB colors with 8 bits for each color component. The three color components for a pixel are packed into a single integer value.

In a program that uses a BufferedImage to store a copy of the on-screen image, the *paintComponent* method generally has the form

protected void paintComponent(Graphics g) {

g.drawImage(OSC, 0, 0, null); Graphics2D g2 = (Graphics2D)g.create(); .

. // Draw extra stuff on top of the image.

.

}

A sample program that uses this technique is *java2d/JavaPixelManipulation.java*. In that program, the user can draw lines, rectangles, and ovals by dragging the mouse. As the mouse moves, the shape is drawn between the starting point of the mouse and its current location. As the mouse moves, parts of the existing image can be repeatedly covered and uncovered, without changing the existing image. In fact, the image is in an off-screen canvas, and the shape that the user is drawing is actually drawn by *paintComponent* over the contents of the canvas. The shape is not drawn to the official image in the canvas until the user releases the mouse and ends the drag operation.

But my main reason for writing the program was to illustrate pixel manipulation, that is, computing with the color components of individual pixels. The BufferedImage class has methods for reading and setting the color of individual pixels. An image consists of rows and columns of pixels. If *OSC* is a BufferedImage, then int color = OSC.getRGB(x,y)

gets the integer that represents the color of the pixel in column number *x* and row number *y*. Each color component is stored in an 8-bit field in the integer *color* value. The individual color components can be extracted for processing using Java's bit manipulation operators:

int red = (color >> 16) & 255; int green = (color >> 8) & 255; int blue = color & 255;

Similarly, given red, green, and blue color component values in the range 0 to 255, we can combine those component values into a single integer and use it to set the color of a pixel in the image:

int color = (red << 16) | (green << 8) | blue;

OSC.setRGB(x,y,color);

There are also methods for reading and setting the colors of an entire rectangular region of pixels.

Pixel operations are used to implement two features of the sample program. First, there is a “Smudge” tool. When the user drags with this tool, it’s like smearing wet paint. When the user first clicks the mouse, the color components from a small square of pixels surrounding the mouse position are copied into arrays. As the user moves the mouse, color from the arrays is blended into the color of the pixels near the mouse position, while those colors are blended into the colors in the arrays. Here is a small rectangle that has been “smudged”:

The second use of pixel manipulation is in implementing “filters.” A filter, in this program, is an operation that modifies an image by replacing the color of each pixel with a weighted average of the colors of a 3-by-3 square of pixels. A “Blur” filter for example, uses equal weights for all pixels in the average, so the color of a pixel is changed to the simple average of the colors of that pixel and its neighbors. Using different weights for each pixel can produce some striking effects.

The pixel manipulation in the sample program produces effects that can’t be achieved with pure vector graphics. I encourage you to learn more by looking at the source code. You might also take a look at the live demos in the next section, which implement the same effects using HTML canvas graphics.

2.6 HTML Canvas Graphics

Most modern web browsers support a 2D graphics API that can be used to create images on a web page. The API is implemented using JavaScript, the client-side programming language for the web. I won't cover the JavaScript language in this section. To understand the material presented here, you don't need to know much about it. Even if you know nothing about it at all, you can learn something about its 2D graphics API and see how it is similar to, and how it differs from, the Java API presented in the previous section. (For a short introduction to JavaScript, see Section A.3 in Appendix A.)

2.6.1 The 2D Graphics Context

The visible content of a web page is made up of "elements" such as headlines and paragraphs. The content is specified using the HTML language. A "canvas" is an HTML element. It appears on the page as a blank rectangular area which can be used as a drawing surface by what I am calling the "HTML canvas" graphics API. In the source code of a web page, a canvas element is created with code of the form

```
<canvas width="800" height="600" id="theCanvas"></canvas>
```

The *width* and *height* give the size of the drawing area, in pixels. The *id* is an identifier that can be used to refer to the canvas in JavaScript.

To draw on a canvas, you need a graphics context. A graphics context is an object that contains functions for drawing shapes. It also contains variables that record the current graphics state, including things like the current drawing color, transform, and font. Here, I will generally use *graphics* as the name of the variable that refers to the graphics context, but the variable name is, of course, up to the programmer. This graphics context plays the same role in the canvas API that a variable of type Graphics2D plays in Java. A typical starting point is

```
canvas = document.getElementById("theCanvas"); graphics = canvas.getContext("2d");
```

The first line gets a reference to the canvas element on the web page, using its *id*. The second line creates the graphics context for that canvas element. (This code will produce an error in a web browser that doesn't support canvas, so you might add some error checking such as putting these commands inside a try..catch statement.)

Typically, you will store the canvas graphics context in a global variable and use the same graphics context throughout your program. This is in contrast to Java, where you typically get a new Graphics2D context each time the *paintComponent*() method is

called, and that new context is in its initial state with default color and stroke properties and with no applied transform. When a graphics context is global, changes made to the state in one function call will carry over to subsequent function calls, unless you do something to limit their effect. This can actually lead to a fairly common type of bug: For example, if you apply a 30-degree rotation in a function, those rotations will **accumulate** each time the function is called, unless you do something to undo the previous rotation before the function is called again.

The rest of this section will be mostly concerned with describing what you can do with a canvas graphics context. But here, for the record, is the complete source code for a very minimal web page that uses canvas graphics:

```
<!DOCTYPE html>

<html>

<head>

<title>Canvas Graphics</title>

<script> let canvas;   // DOM object corresponding to the canvas let graphics; // 2D graphics context for drawing on the canvas

function draw() {

// draw on the canvas, using the graphics context graphics.fillText("Hello World", 10, 20);

}

function init() { canvas = document.getElementById("theCanvas"); graphics = canvas.getContext("2d");

draw(); // draw something on the canvas

} window.onload = init;

</script>

</head> <body>

<canvas id="theCanvas" width="640" height="480"></canvas>

</body>

</html>
```

For a more complete, though still minimal, example, you can look at the sample page *canvas2d/GraphicsStarter.html*. (You should look at the page in a browser, but you should also read the source code.) This example shows how to draw some basic shapes using canvas graphics, and you can use it as a basis for your own experimentation. There are also three more advanced "starter" examples: *canvas2d/GraphicsPlusStarter.html* adds some utility functions for drawing shapes and setting up a coordinate system; *canvas2d/AnimationStarter.html* adds animation and includes a simple hierarchical modeling example; and *canvas2d/EventsStarter.html* shows how to respond to keyboard and mouse events.

2.6.2 Shapes

The default coordinate system on a canvas is the usual: The unit of measure is one pixel; (0,0) is at the upper left corner; the *x*-coordinate increases to the right; and the *y*-coordinate increases downward. The range of *x* and *y* values is given by the *width* and *height* properties of the <canvas> element. The term "pixel" here for the unit of measure is not really correct. Probably, I should say something like "one nominal pixel." The unit of measure is one pixel at typical desktop resolution with no magnification. If you apply a magnification to a browser window, the unit of measure gets stretched. And on a high-resolution screen, one unit in the default coordinate system might correspond to several actual pixels on the display device.

The canvas API supports only a very limited set of basic shapes. In fact, the only basic shapes are rectangles and text. Other shapes must be created as paths. Shapes can be stroked and filled. That includes text: When you stroke a string of text, a pen is dragged along the outlines of the characters; when you fill a string, the insides of the characters are filled. It only really makes sense to stroke text when the characters are rather large. Here are the functions for drawing rectangles and text, where *graphics* refers to the object that represents the graphics context:

- graphics.fillRect(x,y,w,h) — draws a filled rectangle with corner at (*x,y*), with width *w* and with height *h*. If the width or the height is less than or equal to zero, nothing is drawn.
- graphics.strokeRect(x,y,w,h) — strokes the outline of the same rectangle.
- graphics.clearRect(x,y,w,h) — clears the rectangle by filling it with fully transparent pixels, allowing the background of the canvas to show. The background is determined by the properties of the web page on which the canvas appears. It might be a background color, an image, or even another canvas.

- graphics.fillText(str,x,y) — fills the characters in the string *str*. The left end of the baseline of the string is positioned at the point (*x,y*).
- graphics.strokeText(str,x,y) — strokes the outlines of the characters in the string.

A path can be created using functions in the graphics context. The context keeps track of a "current path." In the current version of the API, paths are not represented by objects, and there is no way to work with more than one path at a time or to keep a copy of a path for later reuse. Paths can contain lines, Bezier curves, and circular arcs. Here are the most common functions for working with paths:

- graphics.beginPath() — start a new path. Any previous path is discarded, and the current path in the graphics context is now empty. Note that the graphics context also keeps track of the current point, the last point in the current path. After calling *graphics.beginPath*(), the current point is undefined.
- graphics.moveTo(x,y) — move the current point to (*x,y*), without adding anything to the path. This can be used for the starting point of the path or to start a new, disconnected segment of the path.
- graphics.lineTo(x,y) — add the line segment starting at current point and ending at (*x,y*) to the path, and move the current point to (*x,y*).
- graphics.bezierCurveTo(cx1,cy1,c2x,cy2,x,y) — add a cubic Bezier curve to the path. The curve starts at the current point and ends at (*x,y*). The points (*cx1,cy1*) and (*cx2,cy2*) are the two control points for the curve. (Bezier curves and their control points were discussed in Subsection 2.2.3.)
- graphics.quadraticCurveTo(cx,cy,x,y) — adds a quadratic Bezier curve from the current point to (*x,y*), with control point (*cx,cy*).
- graphics.arc(x,y,r,startAngle,endAngle) — adds an arc of the circle with center (*x,y*) and radius *r*. The next two parameters give the starting and ending angle of the arc. They are measured in radians. The arc extends in the positive direction from the start angle to the end angle. (The positive rotation direction is from the positive x-axis towards the positive y-axis; this is clockwise in the default coordinate system.) An optional fifth parameter can be set to *true* to get an arc that extends in the negative direction. After drawing the arc, the current point is at the end of the arc. If there is a current point before *graphics.arc* is called, then before the arc is drawn, a line is added to the path that extends from the current point to the starting point of the arc. (Recall that immediately after *graphics.beginPath*(), there is no current point.)

- graphics.closePath() — adds to the path a line from the current point back to the starting point of the current segment of the curve. (Recall that you start a new segment of the curve every time you use *moveTo*.)

Creating a curve with these commands does not draw anything. To get something visible to appear in the image, you must fill or stroke the path.

The commands *graphics.fill*() and *graphics.stroke*() are used to fill and to stroke the current path. If you fill a path that has not been closed, the fill algorithm acts as though a final line segment had been added to close the path. When you stroke a shape, it's the center of the virtual pen that moves along the path. So, for high-precision canvas drawing, it's common to use paths that pass through the centers of pixels rather than through their corners. For example, to draw a line that extends from the pixel with coordinates (100,200) to the pixel with coordinates (300,200), you would actually stroke the geometric line with endpoints (100.5,200.5) and (100.5,300.5). We should look at some examples. It takes four steps to draw a line:

```
graphics.beginPath();          // start a new path

graphics.moveTo(100.5,200.5); // starting point of the new path

graphics.lineTo(300.5,200.5); // add a line to the point (300.5,200.5) graphics.stroke();
    // draw the line
```

Remember that the line remains as part of the current path until the next time you call *graphics.beginPath*(). Here's how to draw a filled, regular octagon centered at (200,400) and with radius 100:

```
graphics.beginPath();

graphics.moveTo(300,400);

for (let i = 1; i < 8; i++) {

let angle = (2*Math.PI)/8 * i; let x = 200 + 100*Math.cos(angle); let y = 400 +
100*Math.sin(angle);

graphics.lineTo(x,y);

}

graphics.closePath(); graphics.fill();
```

The function *graphics.arc*() can be used to draw a circle, with a start angle of 0 and an end angle of *2*Math.PI*. Here's a filled circle with radius 100, centered at 200,300:

graphics.beginPath();

graphics.arc(200, 300, 100, 0, 2*Math.PI); graphics.fill();

To draw just the outline of the circle, use *graphics.stroke*() in place of *graphics.fill*(). You can apply both operations to the same path. If you look at the details of *graphics.arc*(), you can see how to draw a wedge of a circle:

graphics.beginPath(); graphics.moveTo(200,300); // Move current point to center of the circle. graphics.arc(200,300,100,0,Math.PI/4); // Arc, plus line from current point. graphics.lineTo(200,300); // Line from end of arc back to center of circle. graphics.fill(); // Fill the wedge.

There is no way to draw an oval that is not a circle, except by using transforms. We will cover that later in this section. But JavaScript has the interesting property that it is possible to add new functions and properties to an existing object. The sample program *canvas2d/GraphicsPlusStarter.html* shows how to add functions to a graphics context for drawing lines, ovals, and other shapes that are not built into the API.

2.6.3 Stroke and Fill

Attributes such as line width that affect the visual appearance of strokes and fills are stored as properties of the graphics context. For example, the value of *graphics.lineWidth* is a number that represents the width that will be used for strokes. (The width is given in pixels for the default coordinate system, but it is subject to transforms.) You can change the line width by assigning a value to this property:

graphics.lineWidth = 2.5; // Change the current width.

The change affects subsequent strokes. You can also read the current value:

saveWidth = graphics.lineWidth; // Save current width.

The property *graphics.lineCap* controls the appearance of the endpoints of a stroke. It can be set to “round”, “square”, or “butt”. The quotation marks are part of the value. For example, graphics.lineCap = "round";

Similarly, *graphics.lineJoin* controls the appearance of the point where one segment of a stroke joins another segment; its possible values are “round”, “bevel”, or “miter”. (Line endpoints and joins were discussed in Subsection 2.2.1.)

Note that the values for *graphics.lineCap* and *graphics.lineJoin* are strings. This is a somewhat unusual aspect of the API. Several other properties of the graphics context

take values that are strings, including the properties that control the colors used for drawing and the font that is used for drawing text.

Color is controlled by the values of the properties *graphics.fillStyle* and *graphics.strokeStyle*. The graphics context maintains separate styles for filling and for stroking. A solid color for stroking or filling is specified as a string. Valid color strings are ones that can be used in CSS, the language that is used to specify colors and other style properties of elements on web pages. Many solid colors can be specified by their names, such as “red”, “black”, and “beige”. An RGB color can be specified as a string of the form “rgb(r,g,b)”, where the parentheses contain three numbers in the range 0 to 255 giving the red, green, and blue components of the color. Hexadecimal color codes are also supported, in the form “#XXYYZZ” where XX, YY, and ZZ are two-digit hexadecimal numbers giving the RGB color components. For example,

graphics.fillStyle = "rgb(200,200,255)"; // light blue graphics.strokeStyle = "#0070A0"; // a darker, greenish blue

The style can actually be more complicated than a simple solid color: Gradients and patterns are also supported. As an example, a gradient can be created with a series of steps such as

let lineargradient = graphics.createLinearGradient(420,420,550,200); lineargradient.addColorStop(0,"red"); lineargradient.addColorStop(0.5,"yellow"); lineargradient.addColorStop(1,"green"); graphics.fillStyle = lineargradient; // Use a gradient fill!

The first line creates a linear gradient that will vary in color along the line segment from the point (420,420) to the point (550,200). Colors for the gradient are specified by the *addColorStop* function: the first parameter gives the fraction of the distance from the initial point to the final point where that color is applied, and the second is a string that specifies the color itself. A color stop at 0 specifies the color at the initial point; a color stop at 1 specifies the color at the final point. Once a gradient has been created, it can be used both as a fill style and as a stroke style in the graphics context.

Finally, I note that the font that is used for drawing text is the value of the property *graphics.font*. The value is a string that could be used to specify a font in CSS. As such, it can be fairly complicated, but the simplest versions include a font-size (such as *20px* or *150%*) and a font-family (such as *serif*, *sans-serif*, *monospace*, or the name of any font that is accessible to the web page). You can add *italic* or *bold* or both to the front of the string. Some examples:

```
graphics.font = "2cm monospace"; // the size is in centimeters

graphics.font = "bold 18px sans-serif";

    graphics.font = "italic 150% serif";   // size is 150% of the usual size
```

The default is “10px sans-serif,” which is usually too small. Note that text, like all drawing, is subject to coordinate transforms. Applying a scaling operation changes the size of the text, and a negative scaling factor can produce mirror-image text.

2.6.4 Transforms

A graphics context has three basic functions for modifying the current transform by scaling, rotation, and translation. There are also functions that will compose the current transform with an arbitrary transform and for completely replacing the current transform:

- graphics.scale(sx,sy) — scale by *sx* in the *x*-direction and *sy* in the *y*-direction.
- graphics.rotate(angle) — rotate by *angle* radians about the origin. A positive rotation is clockwise in the default coordinate system.
- graphics.translate(tx,ty) — translate by *tx* in the *x*-direction and *ty* in the *y*direction.
- graphics.transform(a,b,c,d,e,f) — apply the affine transform x1 = a*x + c*y + e, and y1 = b*x + d*y + f.
- graphics.setTransform(a,b,c,d,e,f) — discard the current transformation, and set the current transformation to be x1 = a*x + c*y + e, and y1 = b*x + d*y + f.

Note that there is no shear transform, but you can apply a shear as a general transform. For example, for a horizontal shear with shear factor 0.5, use graphics.transform(1, 0, 0.5, 1, 0, 0)

To implement hierarchical modeling, as discussed in Section 2.4, you need to be able to save the current transformation so that you can restore it later. Unfortunately, no way is provided to read the current transformation from a canvas graphics context. However, the graphics context itself keeps a stack of transformations and provides methods for pushing and popping the current transformation. In fact, these methods do more than save and restore the current transformation. They actually save and restore almost the entire state of the graphics context, including properties such as current colors, line width, and font (but not the current path):

- graphics.save() — push a copy of the current state of the graphics context, including the current transformation, onto the stack.
- graphics.restore() — remove the top item from the stack, containing a saved state of the graphics context, and restore the graphics context to that state.

Using these methods, the basic setup for drawing an object with a modeling transform becomes:

```
graphics.save();         // save a copy of the current state
```

graphics.translate(a,b); // apply modeling transformations graphics.rotate(r); graphics.scale(sx,sy); .

. // Draw the object!

.

```
graphics.restore();     // restore the saved state
```

Note that if drawing the object includes any changes to attributes such as drawing color, those changes will be also undone by the call to *graphics.restore*(). In hierarchical graphics, this is usually what you want, and it eliminates the need to have extra statements for saving and restoring things like color.

To draw a hierarchical model, you need to traverse a scene graph, either procedurally or as a data structure. It's pretty much the same as in Java. In fact, you should see that the basic concepts that you learned about transformations and modeling carry over to the canvas graphics API. Those concepts apply very widely and even carry over to 3D graphics APIs, with just a little added complexity. The sample web page *canvas2d/HierarchicalModel2D.html* implements hierarchical modeling using the 2D canvas API.

* * *

Now that we know how to do transformations, we can see how to draw an oval using the canvas API. Suppose that we want an oval with center at (*x,y*), with horizontal radius *r1* and with vertical radius *r2*. The idea is to draw a circle of radius 1 with center at (0,0), then transform it. The circle needs to be scaled by a factor of *r1* horizontally and *r2* vertically. It should then be translated to move its center from (0,0) to (*x,y*). We can use *graphics.save*() and *graphics.restore*() to make sure that the transformations only affect the circle. Recalling that the order of transforms in the code is the opposite of the order in which they are applied to objects, this becomes:

graphics.save(); graphics.translate(x, y); graphics.scale(r1, r2);

graphics.beginPath();

graphics.arc(0, 0, 1, 0, Math.PI); // a circle of radius 1 graphics.restore();
graphics.stroke();

Note that the current path is **not** affected by the calls to *graphics.save*() and *graphics.restore*(). So, in the example, the oval-shaped path is not discarded when *graphics.restore*() is called. When *graphics.stroke*() is called at the end, it is the oval-shaped path that is stroked. On the other hand, the line width that is used for the stroke is not affected by the scale transform that was applied to the oval. Note that if the order of the last two commands were reversed, then the line width would be subject to the scaling.

There is an interesting point here about transforms and paths. In the HTML canvas API, the points that are used to create a path are transformed by the current transformation before they are saved. That is, they are saved in pixel coordinates. Later, when the path is stroked or filled, the current transform has no effect on the path (although it can affect, for example, the line width when the path is stroked). In particular, you can't make a path and then apply different transformations. For example, you can't make an oval-shaped path, and then use it to draw several ovals in different positions. Every time you draw the oval, it will be in the same place, even if different translation transforms are applied to the graphics context.

The situation is different in Java, where the coordinates that are stored in the path are the actual numbers that are used to specify the path, that is, the object coordinates. When the path is stroked or filled, the transformation that is in effect at that time is applied to the path. The path can be reused many times to draw copies with different transformations. This comment is offered as an example of how APIs that look very similar can have subtle differences.

2.6.5 Auxiliary Canvases

In Subsection 2.5.5, we looked at the sample program *java2d/JavaPixelManipulation.java*, which uses a BufferedImage both to implement an off-screen canvas and to allow direct manipulation of the colors of individual pixels. The same ideas can be applied in HTML canvas graphics, although the way it's done is a little different. The sample web application *canvas2d/SimplePaintProgram.html* does pretty much the same thing as the Java program (except for the image filters).

The on-line version of this section has a live demo version of the program that has the same functionality. You can try it out to see how the various drawing tools work. Don't forget to try the "Smudge" tool! (It has to be applied to shapes that you have already drawn.)

For JavaScript, a web page is represented as a data structure, defined by a standard called the DOM, or Document Object model. For an off-screen canvas, we can use a <canvas> that is not part of that data structure and therefore is not part of the page. In JavaScript, a <canvas> can be created with the function call *document.createElement*("canvas"). There is a way to add this kind of dynamically created canvas to the DOM for the web page, but it can be used as an off-screen canvas without doing so. To use it, you have to set its width and height properties, and you need a graphics context for drawing on it. Here, for example, is some code that creates a 640-by-480 canvas, gets a graphics context for the canvas, and fills the whole canvas with white:

```
OSC = document.createElement("canvas"); // off-screen canvas

OSC.width = 640;      // Size of OSC must be set explicitly. OSC.height = 480;

OSG = OSC.getContext("2d"); // Graphics context for drawing on OSC.

OSG.fillStyle = "white"; // Use the context to fill OSC with white.
OSG.fillRect(0,0,OSC.width,OSC.height);
```

The sample program lets the user drag the mouse on the canvas to draw some shapes. The off-screen canvas holds the official copy of the picture, but it is not seen by the user. There is also an on-screen canvas that the user sees. The off-screen canvas is copied to the on-screen canvas whenever the picture is modified. While the user is dragging the mouse to draw a line, oval, or rectangle, the new shape is actually drawn on-screen, over the contents of the off-screen canvas. It is only added to the off-screen canvas when the user finishes the drag operation. For the other tools, changes are made directly to the off-screen canvas, and the result is then copied to the screen. This is an exact imitation of the Java program.

(The demo version mentioned above actually uses a somewhat different technique to accomplish the same thing. It uses two on-screen canvases, one located exactly on top of the other. The lower canvas holds the actual image. The upper canvas is completely transparent, except when the user is drawing a line, oval, or rectangle. While the user is dragging the mouse to draw such a shape, the new shape is drawn on the upper canvas, where it hides the part of the lower canvas that is beneath the shape. When the user

releases the mouse, the shape is added to the lower canvas and the upper canvas is cleared to make it completely transparent again. Again, the other tools operate directly on the lower canvas.)

2.6.6 Pixel Manipulation

The "Smudge" tool in the sample program and demo is implemented by computing with the color component values of pixels in the image. The implementation requires some way to read the colors of pixels in a canvas. That can be done with the function *graphics.getPixelData*(*x,y,w,h*), where *graphics* is a 2D graphics context for the canvas. The function reads the colors of a rectangle of pixels, where (*x,y*) is the upper left corner of the rectangle, *w* is its width, and *h* is its height. The parameters are always expressed in pixel coordinates. Consider, for example colors = graphics.getImageData(0,0,20,10)

This returns the color data for a 20-by-10 rectangle in the upper left corner of the canvas. The return value, *colors*, is an object with properties *colors.width*, *colors.height*, and *colors.data*. The *width* and *height* give the number of rows and columns of pixels in the returned data. (According to the documentation, on a high-resolution screen, they might not be the same as the width and height in the function call. The data can be for real, physical pixels on the display device, not the "nominal" pixels that are used in the pixel coordinate system on the canvas. There might be several device pixels for each nominal pixel. I'm not sure whether this can really happen in practice.)

The value of *colors.data* is an array, with four array elements for each pixel. The four elements contain the red, blue, green, and alpha color components of the pixel, given as integers in the range 0 to 255. For a pixel that lies outside the canvas, the four component values will all be zero. The array is a value of type *Uint8ClampedArray* whose elements are 8-bit unsigned integers limited to the range 0 to 255. This is one of JavaScript's typed array datatypes, which can only hold values of a specific numerical type. As an example, suppose that you just want to read the RGB color of one pixel, at coordinates (*x,y*). You can set pixel = graphics.getImageData(x,y,1,1);

Then the RGB color components for the pixel are R = *pixel.data*[0], G = *pixel.data*[1], and B = *pixel.data*[2].

The function *graphics.putImageData*(*imageData*,*x*,*y*) is used to copy the colors from an image data object into a canvas, placing it into a rectangle in the canvas with upper left corner at (*x,y*). The *imageData* object can be one that was returned by a call to *graphics.getImageData*, possibly with its color data modified. Or you can create a blank image data object by calling *graphics.createImageData*(*w,h*) and fill it with data.

Let's consider the "Smudge" tool in the sample program. When the user clicks the mouse with this tool, I use *OSG.getImageData* to get the color data from a 9-by-9 square of pixels surrounding the mouse location. *OSG* is the graphics context for the canvas that contains the image. Since I want to do real-number arithmetic with color values, I copy the color components into another typed array, one of type Float32Array, which can hold 32-bit floating point numbers. Here is the function that I call to do this:

```
function grabSmudgeData(x, y) { // (x,y) gives mouse location let colors =
OSG.getImageData(x-5,y-5,9,9); if (smudgeColorArray == null) {

// Make image data & array the first time this function is called.

smudgeImageData = OSG.createImageData(9,9); smudgeColorArray = new
Float32Array(colors.data.length);

} for (let i = 0; i < colors.data.length; i++) {

// Copy the color component data into the Float32Array.

smudgeColorArray[i] = colors.data[i]; }

}
```

The floating point array, *smudgeColorArray*, will be used for computing new color values for the image as the mouse moves. The color values from this array will be copied into the image data object, *smudgeImageData*, which will then be used to put the color values into the image. This is done in another function, which is called for each point that is visited as the user drags the Smudge tool over the canvas:

```
function swapSmudgeData(x, y) { // (x,y) is new mouse location let colors =
OSG.getImageData(x-5,y-5,9,9); // get color data from image for (let i = 0; i <
smudgeColorArray.length; i += 4) {

// The color data for one pixel is in the next four array locations. if
(smudgeColorArray[i+3] && colors.data[i+3]) {

// alpha-components are non-zero; both pixels are in the canvas; // (getImageData()
gets 0 for the alpha value at pixel coordinates // that are not actually part of the canvas).

for (let j = i; j < i+3; j++) { // compute new RGB values

let newSmudge = smudgeColorArray[j]*0.8 + colors.data[j]*0.2; let newImage =
smudgeColorArray[j]*0.2 + colors.data[j]*0.8; smudgeImageData.data[j] = newImage;
smudgeColorArray[j] = newSmudge;
```

```
} smudgeImageData.data[i+3] = 255; // alpha component

} else {

// one of the alpha components is zero; set the output

// color to all zeros, "transparent black", which will have // no effect on the color of the pixel in the canvas.

for (let j = i; j <= i+3; j++) { smudgeImageData.data[j] = 0;

}

}

}

OSG.putImageData(smudgeImageData,x-5,y-5); // copy new colors into canvas }
```

In this function, a new color is computed for each pixel in a 9-by-9 square of pixels around the mouse location. The color is replaced by a weighted average of the current color of the pixel and the color of the corresponding pixel in the *smudgeColorArray*. At the same time, the color in *smudgeColorArray* is replaced by a similar weighted average.

It would be worthwhile to try to understand this example to see how pixel-by-pixel processing of color data can be done. See the source code of the example for more details.

2.6.7 Images

For another example of pixel manipulation, we can look at image filters that modify an image by replacing the color of each pixel with a weighted average of the color of that pixel and the 8 pixels that surround it. Depending on the weighting factors that are used, the result can be as simple as a slightly blurred version of the image, or it can be something more interesting.

The on-line version of this section includes an interactive demo that lets you apply several different image filters to a variety of images.

The filtering operation in the demo uses the image data functions *getImageData*, *createImageData*, and *putImageData* that were discussed above. Color data from the entire image is obtained with a call to *getImageData*. The results of the averaging computation are placed in a new image data object, and the resulting image data is copied back to the image using *putImageData*.

The remaining question is, where do the original images come from, and how do they get onto the canvas in the first place? An image on a web page is specified by an element in the web page source such as

<img src="pic.jpg" width="400" height="300" id="mypic">

The *src* attribute specifies the URL from which the image is loaded. The optional *id* can be used to reference the image in JavaScript. In the script, image = document.getElementById("mypic");

gets a reference to the object that represents the image in the document structure. Once you have such an object, you can use it to draw the image on a canvas. If *graphics* is a graphics context for the canvas, then graphics.drawImage(image, x, y);

draws the image with its upper left corner at (*x,y*). Both the point (*x,y*) and the image itself are transformed by any transformation in effect in the graphics context. This will draw the image using its natural width and height (scaled by the transformation, if any). You can also specify the width and height of the rectangle in which the image is drawn:

graphics.drawImage(image, x, y, width, height);

With this version of *drawImage*, the image is scaled to fit the specified rectangle.

Now, suppose that the image you want to draw onto the canvas is not part of the web page? In that case, it is possible to load the image dynamically. This is much like making an off-screen canvas, but you are making an "off-screen image." Use the *document* object to create an *img* element:

newImage = document.createElement("img");

An *img* element needs a *src* attribute that specifies the URL from which it is to be loaded. For example, newImage.src = "pic2.jpg";

As soon as you assign a value to the *src* attribute, the browser starts loading the image. The loading is done asynchronously; that is, the computer continues to execute the script without waiting for the load to complete. This means that you can't simply draw the image on the line after the above assignment statement: The image is very likely not done loading at that time. You want to draw the image after it has finished loading. For that to happen, you need to assign a function to the image's *onload* property before setting the *src*. That function will be called when the image has been fully loaded. Putting this together, here is a simple JavaScript function for loading an image from a specified URL and drawing it on a canvas after it has loaded:

```
function loadAndDraw( imageURL, x, y ) { let image = document.createElement("img");
image.onload = doneLoading; image.src = imageURL; function doneLoading() {
graphics.drawImage(image, x, y);

}

}
```

A similar technique is used to load the images in the filter demo.

There is one last mystery to clear up. When discussing the use of an off-screen canvas in the *SimplePaintProgram* example earlier in this section, I noted that the contents of the off-screen canvas have to be copied to the main canvas, but I didn't say how that can be done. In fact, it is done using *drawImage*. In addition to drawing an image onto a canvas, *drawImage* can be used to draw the contents of one canvas into another canvas. In the sample program, the command graphics.drawImage(OSC, 0, 0);

is used to draw the off-screen canvas to the main canvas. Here, *graphics* is a graphics context for drawing on the main canvas, and *OSC* is the object that represents the off-screen canvas.

2.7 SVG: A Scene Description Language

We finish this chapter with a look at one more 2D graphics system: SVG, or Scalable Vector Graphics. So far, we have been considering graphics programming APIs. SVG, on the other hand is a scene description language rather than a programming language. Where a programming language creates a scene by generating its contents procedurally, a scene description language specifies a scene "declaratively," by listing its content. Since SVG is a vector graphics language, the content of a scene includes shapes, attributes such as color and line width, and geometric transforms. Most of this should be familiar to you, but it should be interesting to see it in a new context.

SVG is an XML language, which means it has a very strict and somewhat verbose syntax. This can make it a little annoying to write, but on the other hand, it makes it possible to read and understand SVG documents even if you are not familiar with the syntax. It's possible that SVG originally stood for "Simple" Vector Graphics, but it is by no means a simple language at this point. I will cover only a part of it here, and there are many parts of the language and many options that I will not mention. My goal is to introduce the idea of a scene description language and to show how such a language can use the same basic ideas that are used in the rest of this chapter.

SVG can be used as a file format for storing vector graphics images, in much the same way that PNG and JPEG are file formats for storing pixel-based images. That means that you can open an SVG file with almost any web browser to view the image. An SVG image can be included in a web page by using it as the *src* of an <img> element. That's how the SVG examples in the web version of this section are displayed. Since SVG documents are written in plain text, you can create SVG images using a regular text editor, and you can read the source

for an SVG image by opening it in a text editor or by viewing the source of the image when it is displayed in a web browser.

2.7.1 SVG Document Structure

An SVG file, like any XML document, starts with some standard code that almost no one memorizes. It should just be copied into a new document. Here is some code that can be copied as a starting point for SVG documents of the type discussed in this section (which, remember use only a subset of the full SVG specification):

```
<?xml version="1.0"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"

"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="4in" height="4in" viewBox="0 0 400
400" preserveAspectRatio="xMidYMid">

<!-- The scene description goes here! -->

</svg>
```

The first three lines say that this is an XML SVG document. The rest of the document is an <svg> element that acts as a container for the entire scene description. You'll need to know a little about XML syntax. First, an XML "element" in its general form looks like this:

```
<elementname attrib1="value1" attrib2="value2"> ...content...

</elementname>
```

The element starts with a "start tag," which begins with a "<" followed by an identifier that is the name of the tag, and ending with a ">". The start tag can include "attributes," which have the form *name="value"*. The *name* is an identifier; the *value* is a string. The value must be enclosed in single or double quotation marks. The element ends with an "end tag," which has an element name that matches the element name in the start tag and has the form </*elementname*>. Element names and attribute names are case-sensitive. Between the start and end tags comes the "content" of the element. The content can consist of text and nested elements. If an element has no content, you can replace the ">" at the end of the start tag with "/>", and leave out the end tag. This is called a "self-closing tag." For example,

```
<circle cx="5" cy="5" r="4" fill="red"/>
```

This is an actual SVG element that specifies a circle. It's easy to forget the "/" at the end of a self-closing tag, but it has to be there to have a legal XML document.

Looking back at the SVG document, the five lines starting with <svg are just a long start tag. You can use the tag as shown, and customize the values of the *width*, *height*, *viewBox*, and *preserveAspectRatio* attributes. The next line is a comment; comments in XML start with "<!--" and end with "-->".

The *width* and *height* attributes of the <svg> tag specify a natural or preferred size for the image. It can be forced into a different size, for example if it is used in an <img> element on a web page that specifies a different width and height. The size can be specified using units of measure such as *in* for inches, *cm* for centimeters, and *px*, for pixels, with 90 pixels to the inch. If no unit of measure is specified, pixels are used. There cannot be any space between the number and the unit of measure.

The *viewBox* attribute sets up the coordinate system that will be used for drawing the image. It is what I called the view window in Subsection 2.3.1. The value for viewBox is a list of four numbers, giving the minimum *x*-value, the minimum *y-value*, the width, and the height of the view window. The width and the height must be positive, so *x* increases from left-to-right, and *y* increases from top-to-bottom. The four numbers in the list can be separated either by spaces or by commas; this is typical for lists of numbers in SVG.

Finally, the *preserveAspectRatio* attribute tells what happens when the aspect ratio of the viewBox does not match the aspect ratio of the rectangle in which the image is displayed. The default value, "xMidYMid", will extend the limts on the viewBox either horizontally or vertically to preserve the aspect ratio, and the viewBox will appear in the center of the display rectangle. If you would like your image to stretch to fill the display rectangle, ignoring the aspect ratio, set the value of *preserveAspectRatio* to "none". (The aspect ratio issue was discussed in Subsection 2.3.7.)

Let's look at a complete SVG document that draws a few simple shapes. Here's the document. You could probably figure out what it draws even without knowing any more about SVG:

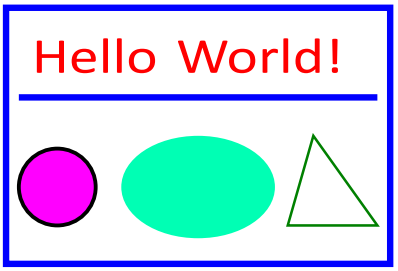
```
<?xml version="1.0"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"

"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="300px" height="200px" viewBox="0
0 3 2"

preserveAspectRatio="xMidYMid">

<rect x="0" y="0" width="3" height="2" stroke="blue" fill="none" stroke-width="0.05"/>

<text x="0.2" y="0.5" font-size="0.4" fill="red">Hello World!</text>
```

<line x1="0.1" y1="0.7" x2="2.9" y2="0.7" stroke-width="0.05" stroke="blue"/>

<ellipse cx="1.5" cy="1.4" rx=".6" ry=".4" fill="rgb(0,255,180)"/>

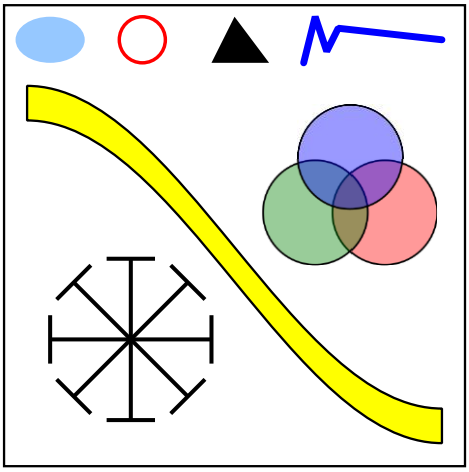
<circle cx="0.4" cy="1.4" r="0.3" fill="magenta" stroke="black" stroke-width="0.03"/>

<polygon points="2.2,1.7 2.4,1 2.9,1.7" fill="none" stroke="green" stroke-width="0.02"/>
</svg> and here's the image that is produced by this example:



In the drawing coordinate system for this example, *x* ranges from 0 to 3, and *y* ranges from 0 to 2. All values used for drawing, including stroke width and font size, are given in terms of this coordinate system. Remember that you can use any coordinate system that you find convenient! Note, by the way, that parts of the image that are not covered by the shapes that are drawn will be transparent.

Here's another example, with a larger variety of shapes. The source code for this example has a lot of comments. It uses features that we will discuss in the remainer of this section.



You can take a look at the source code, *svg/svg-starter.svg*. (For example, open it in a text editor, or open it in a web browser and use the browser's "view source" command.)

2.7.2 Shapes, Styles, and Transforms

In SVG, a basic shape is specified by an element in which the tag name gives the shape, and attributes give the properties of the shape. There are attributes to specify the geometry, such as the endpoints of a line or the radius of a circle. Other attributes specify style properties, such as fill color and line width. (The style properties are what I

call attributes elsewhere in this book; in this section, I am using the term “attribute” in its XML sense.) And there is a *transform* attribute that can be used to apply a geometric transform to the shape.

For a detailed example, consider the *rect* element, which specifies a rectangle. The geometry of the rectangle is given by attributes named *x*, *y*, *width* and *height* in the usual way. The default value for *x* and *y* is zero; that is, they are optional, and leaving them out is the same as setting their value to zero. The *width* and the *height* are required attributes. Their values must be non-negative. For example, the element

<rect width="3" height="2"/> specifies a rectangle with corner at (0,0), width 3, and height 2, while

<rect x="100" y="200" height="480" width="640"/> gives a rectangle with corner at (100,200), width 640, and height 480. (Note, by the way, that the attributes in an XML element can be given in any order.) The *rect* element also has optional attributes *rx* and *ry* that can be used to make “roundRects,” with their corners replaced by elliptical arcs. The values of *rx* and *ry* give the horizontal and vertical radii of the elliptical arcs.

Style attributes can be added to say how the shape should be stroked and filled. The default is to use a black fill and no stroke. (More precisely, as we will see later, the default is for a shape to inherit the values of style attributes from its environment. Black fill and no stroke is the initial environment.) Here are some common style attributes:

- fill — specifies how to fill the shape. The value can be “none” to indicate that the shape is not filled. It can be a color, in the same format as the CSS colors that are used in the HTML canvas API. For example, it can be a common color name such as “black” or “red”, or an RGB color such as “rgb(255,200,180)”. There are also gradient and pattern fills, though I will not discuss them here.

- stroke — specifies how to stroke the shape, with the same possible values as “fill”.

- stroke-opacity and fill-opacity — are numbers between 0.0 and 1.0 that specify the opacity of the stroke and fill. Values less than 1.0 give a translucent stroke or fill. The default value, 1.0, means fully opaque.

- stroke-width — is a number that sets the line width to use for the stroke. Note that the line width is subject to transforms. The default value is ”1”, which is fine if the coordinate system is using pixels as the unit of measure, but often too wide in custom coordinate systems.

- stroke-linecap — determines the appearance of the endpoints of a stroke. The value can be “square”, “round”, or “butt”. The default is “butt”. (See Subsection 2.2.1 for a discussion of line caps and joins.)

- stroke-linejoin — determines the appearance of points where two segments of a stroke meet. The values can be “miter”, “round”, or “bevel”. The default is “miter”.

As an example that uses many of these options, let’s make a square that is rounded rather than pointed at the corners, with size 1, centered at the origin, and using a translucent red fill and a gray stroke:

<rect x="-0.5" y="-0.5" width="1" height="1" rx="0.1" ry="0.1" fill="red" fill-opacity="0.5"

stroke="gray" stroke-width="0.05" stroke-linejoin="round"/>

and a simple outline of a rectangle with no fill:

<rect width="200" height="100" stroke="black" fill="none"/>

* * *

The *transform* attribute can be used to apply a transform or a series of transforms to a shape. As an example, we can make a rectangle tilted 30 degrees from the horizontal:

<rect width="100" height="50" transform="rotate(30)"/>

The value “rotate(30)” represents a rotation of 30 degrees (not radians!) about the origin, (0,0). The positive direction of rotation, as usual, rotates the positive x-axis in the direction of the positive y-axis. You can specify a different center of rotation by adding arguments to *rotate*.

For example, to rotate the same rectangle about its center

<rect width="100" height="50" transform="rotate(30,50,25)"/>

Translation and scaling work as you probably expect, with transform values of the form “translate(*dx,dy*)” and “scale(*sx,sy*)”. There are also shear transforms, but they go by the names *skewX* and *skewY*, and the argument is a skew angle rather than a shear amount. For example, the transform “skewX(45)” tilts the y-axis by 45 degrees and is equivalent to an x-shear with shear factor 1. (The function that tilts the y-axis is called *skewX* because it modifies, or skews, the x-coordinates of points while leaving their y-coordinates unchanged.) For example, we can use *skewX* to tilt a rectangle and make it into a parallelogram:

<rect width="100" height="50" transform="skewX(-30)"/>

I used an angle of -30 degrees to make the rectangle tilt to the right in the usual pixel coordinate system.

The value of the *transform* attribute can be a list of transforms, separated by spaces or commas. The transforms are applied to the object, as usual, in the opposite of the order in which they are listed. So,

<rect width="100" height="50" transform="translate(0,50) rotate(45) skewX(-30)"/>

would first skew the rectangle into a parallelogram, then rotate the parallelogram by 45 degrees about the origin, then translate it by 50 units in the y-direction.

* * *

In addition to rectangles, SVG has lines, circles, ellipses, and text as basic shapes. Here are some details. A <line> element represents a line segement and has geometric attributes *x1*, *y1*, *x2*, and *y2* to specify the coordinates of the endpoints of the line segment. These four attributes have zero as default value, which makes it easier to specify horizontal and vertical lines. For example,

<line x1="100" x2="300" stroke="black"/>

Without the *stroke* attribute, you wouldn't see the line, since the default value for *stoke* is "none".

For a <circle> element, the geometric attributes are *cx*, *cy*, and *r* giving the coordinates of the center of the circle and the radius. The center coordinates have default values equal to zero. For an <ellipse> element, the attributes are *cx*, *cy*, *rx*, and *ry*, where *rx* and *ry* give the radii of the ellipse in the x- and y-directions.

A <text> element is a little different. It has attributes *x* and *y*, with default values zero, to specify the location of the basepoint of the text. However, the text itself is given as the content of the element rather than as an attribute. That is, the element is divided into a start tag and an end tag, and the text that will appear in the drawing comes between the start and end tags.

For example,

<text x="10" y="30">This text will appear in the image</text>

The usual stroke and fill attributes apply to text, but text has additional style attributes. The *font-family* attribute specifies the font itself. Its value can be one of the generic font names "serif", "sans-serif", "monospace", or the name of a specific font that is available on the system. The *font-size* can be a number giving the (approximate) height of the characters in the coordinate system. (Font size is subject to coordinate and modeling transforms like any other length.) You can get bold and italic text by setting *font-weight* equal to "bold" and *font-style* equal to "italic". Here is an example that uses all of these options, and applies some additional styles and a transform for good measure:

<text x="10" y="30" font-family="sans-serif" font-size="50" font-style="italic" font-weight="bold" stroke="black" stroke-width="1" fill="rgb(255,200,0)" transform="rotate(20)">Hello World</text>

2.7.3 Polygons and Paths

SVG has some nice features for making more complex shapes. The <polygon> element makes it easy to create a polygon from a list of coordinate pairs. For example,

<polygon points="0,0 100,0 100,75 50,100 0,75"/> creates a five-sided polygon with vertices at (0,0), (100,0), (100,75), (50,100), and (0,75). Every pair of numbers in the *points* attribute specifies a vertex. The numbers can be separated by either spaces or commas. I've used a mixture of spaces and commas here to make it clear how the numbers pair up. Of course, you can add the usual style attributes for stroke and fill to the polygon element. A <polyline> is similar to a <polygon>, except that it leaves out the last line from the final vertex back to the starting vertex. The difference only shows up when a polyline is stroked; a polyline is filled as if the missing side were added.

The <path> element is much more interesting. In fact, all of the other basic shapes, except text, could be made using path elements. A path can consist of line segments, Bezier curves, and elliptical arcs (although I won't discuss elliptical arcs here). The syntax for specifying a path is very succinct, and it has some features that we have not seen before. A path element has an attribute named *d* that contains the data for the path. The data consists of one or more commands, where each command consists of a single letter followed by any data necessary for the command. The moveTo, lineTo, cubic Bezier, and quadratic Bezier commands that you are already familiar with are coded by the letters M, L, C, and Q. The command for closing a path segment is Z, and it requires no data. For example the path data "M 10 20 L 100 200" would draw a line segment from the point (10,20) to the point (100,200). You can combine several connected line segments into one L command. For example, the <polygon> example given above could be created using the <path> element

<path d="M 0,0 L 100,0 100,75 50,100 0,75 Z"/>

The Z at the end of the data closes the path by adding the final side to the polygon. (Note that, as usual, you can use either commas or spaces in the data.)

The C command takes six numbers as data, to specify the two control points and the final endpoint of the cubic Bezier curve segment. You can also give a multiple of six values to get a connected sequence of curve segements. Similarly, the Q command uses four data values to specify the control point and final endpoint of the quadratic Bezier curve segment. The large, curvy, yellow shape shown in the picture earlier in this section was created as a path with two line segments and two Bezier curve segments:

<path d="M 20,70 C 150,70 250,350 380,350 L 380,380 C 250,380 150,100 20,100 Z" fill="yellow" stroke-width="2" stroke="black"/>

SVG paths add flexibility by defining "relative" versions of the path commands, where the data for the command is given relative to the current position. A relative move command, for example, instead of telling *where* to move, tells *how far* to move from the

current position. The names of the relative versions of the path commands are lower case letters instead of upper case. “M 10,20” means to move to the point with coordinates (10,20), while “m 10,20” means to move 10 units horizontally and 20 units vertically from the current position. Similarly, if the current position is (*x,y*), then the command “l 3,5”, where the first character is a lower case L, draws a line from (*x,y*) to (*x*+3,*y*+*5*).

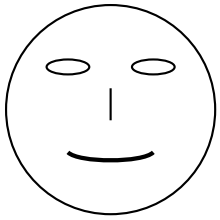
2.7.4 Hierarchical Models

SVG would not be a very interesting language if it could only work with individual simple shapes. For complex scenes, we want to be able to do hierarchical modeling, where objects can be constructed from sub-objects, and a transform can be applied to an entire complex object. We need a way to group objects so that they can be treated as a unit. For that, SVG has the <g> element. The content of a <g> element is a list of shape elements, which can be simple shapes or nested <g> elements.

You can add style and *transform* attributes to a <g> element. The main point of grouping is that a group can be treated as a single object. A *transform* attribute in a <g> will transform the entire group as a whole. A style attribute, such as *fill* or *font-family*, on a <g> element will set a default value for the group, replacing the current default. Here is an example:

```
<g fill="none" stroke="black" stroke-width="2" transform="scale(1,-1)">

<circle r="98"/>

<ellipse cx="40" cy="40" rx="20" ry="7"/>

<ellipse cx="-40" cy="40" rx="20" ry="7"/>

<line y1="20" y2="-10"/>

<path d="M -40,-40 C -30,-50 30,-50 40,-40" stroke-width="4"/> </g>
```

The nested shapes use fill=“none” stroke=“black” stroke-width=“2” for the default values of the attributes. The default can be overridden by specifying a different value for the element, as is done for the stroke-width of the <path> element in this example. Setting transform=“scale(1,−1)” for the group flips the entire image vertically. I do this only because I am more comfortable working in a coordinate system in which y increases from bottom-to-top rather than top-to-bottom. Here is the simple line drawing of a face that is produced by this group:



Now, suppose that we want to include multiple copies of an object in a scene. It shouldn't be necessary to repeat the code for drawing the object. It would be nice to have something like reusable subroutines. In fact, SVG has something very similar: You can define reusable objects inside a <defs> element. An object that is defined inside <defs> is not added to the scene, but copies of the object can be added to the scene with a single command. For this to work, the object must have an *id* attribute to identify it. For example, we could define an object that looks like a plus sign:

<defs>

<g id="plus" stroke="black">

<line x1="-20" y1="0" x2="20" y2="0"/>

<line x1="0" y1="-20" x2="0" y2="20"/>

</g> </defs>

A <use> element can then be used to add a copy of the plus sign object to the scene. The syntax is

<use xlink:href="#plus"/>

The value of the *xlink:href* attribute must be the *id* of the object, with a "#" character added at the beginning. (Don't forget the #. If you leave it out, the <use> element will simply be ignored.) You can add a *transform* attribute to the <use> element to apply a transformation to the copy of the object. You can also apply style attributes, which will be used as default values for the attributes in the copy. For example, we can draw several plus signs with different transforms and stroke widths:

<use xlink:href="#plus" transform="translate(50,20)" stroke-width="5"/>

<use xlink:href="#plus" transform="translate(0,30) rotate(45)"/>

Note that we can't change the color of the plus sign, since it already specifies its own stroke color.

An object that has been defined in the <defs> section can also be used as a sub-object in other object definitions. This makes it possible to create a hierarchy with multiple levels. Here is an example from *svg/svg-hierarchy.svg* that defines a "wheel" object, then uses two copies of the wheel as sub-objects in a "cart" object:

<defs>

<!-- Define an object that represents a wheel centered at (0,0) and with radius 1. The wheel is made out of several filled circles, with thin rectangles for the spokes. -->

<g id="wheel">

<circle cx="0" cy="0" r="1" fill="black"/>

<circle cx="0" cy="0" r="0.8" fill="lightGray"/>

<rect x="-0.9" y="-0.05" width="1.8" height=".1" fill="black"/> <rect x="-0.9" y="-0.05" width="1.8" height=".1" fill="black" transform="rotate(120)"/>

<rect x="-0.9" y="-0.05" width="1.8" height=".1" fill="black" transform="rotate(240)"/>

<circle cx="0" cy="0" r="0.2" fill="black"/>

</g>

<!-- Define an object that represents a cart made out of two wheels, with two rectangles for the body of the cart. -->

<g id="cart">

<use xlink:href="#wheel" transform="translate(-1.5,-0.1) scale(0.8,0.8)"/>

<use xlink:href="#wheel" transform="translate(1.5,-0.1) scale(0.8,0.8)"/>

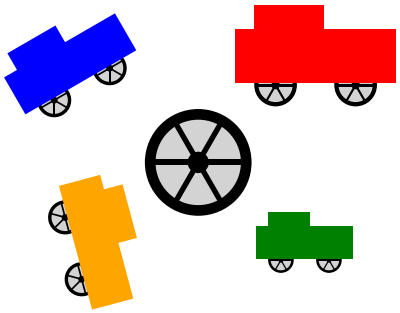
<rect x="-3" y="0" width="6" height="2"/>

<rect x="-2.3" y="1.9" width="2.6" height="1"/>

</g>

</defs>

The SVG file goes on to add one copy of the wheel and four copies of the cart to the image. The four carts have different colors and transforms. Here is the image:



2.7.5 Animation

SVG has a number of advanced features that I won't discuss here, but I do want to mention one: animation. It is possible to animate almost any property of an SVG object, including geometry, style, and transforms. The syntax for animation is itself fairly complex, and I will only do a few examples. But I will tell you enough to produce a fairly complex hierarchical animation like the "cart-and-windmills" example that was discussed and used as a demo in Subsection 2.4.1. An SVG version of that animation

can be found in *svg/cart-and-windmills.svg*. (But note that some web browsers do not implement SVG animations correctly or at all.)

Many attributes of a shape element can be animated by adding an <animate> element to the content of the shape element. Here is an example that makes a rectangle move across the image from left to right:

<rect x="0" y="210" width="40" height="40">

<animate attributeName="x" from="0" to="430" dur="7s" repeatCount="indefinite"/>

</rect>

Note that the <animate> is nested inside the <rect>. The *attributeName* attribute tells which attribute of the <rect> is being animated, in this case, *x*. The *from* and *to* attributes say that *x* will take on values from 0 to 430. The *dur* attribute is the "duration", that is, how long the animation lasts; the value "7s" means "7 seconds." The attribute *repeatCount*="indefinite" means that after the animation completes, it will start over, and it will repeat indefinitely, that is, as long as the image is displayed. If the *repeatCount* attribute is omitted, then after the animation runs once, the rectangle will jump back to its original position and remain there. If *repeatCount* is replaced by *fill*="freeze", then after the animation runs, the rectangle will be frozen in its final position, instead of jumping back to the starting position. The animation begins when the image first loads. If you want the animation to start at a later time, you can add a *begin* attribute whose value gives the time when the animation should start, as a number of seconds after the image loads.

What if we want the rectangle to move back and forth between its initial and final position? For that, we need something called keyframe animation, which is an important idea in its own right. The *from* and *to* attributes allow you to specify values only for the beginning and end of the animation. In a keyframe animation, values are specified at additional times in the middle of the animation. For a keyframe animation in SVG, the *from* and *to* attributes are replaced by *keyTimes* and *values*. Here is our moving rectangle example, modified to use keyframes:

<rect x="0" y="210" width="40" height="40">

<animate attributeName="x"

keyTimes="0;0.5;1" values="0;430;0" dur="7s" repeatCount="indefinite"/>

</rect>

The *keyTimes* attribute is a list of numbers, separated by semicolons. The numbers are in the range 0 to 1, and should be in increasing order. The first number should be 0 and the last number should be 1. A number specifies a time during the animation, as a fraction of the complete animation. For example, 0.5 is a point half-way through the

animation, and 0.75 is three-quarters of the way. The *values* attribute is a list of values, with one value for each key time. In this case, the value for *x* is 0 at the start of the animation, 430 half-way through the animation, and 0 again at the end of the animation. Between the key times, the value for *x* is obtained by interpolating between the values specified for the key times. The result in this case is that the rectangle moves from left to right during the first half of the animation and then back from right to left in the second half.

Transforms can also be animated, but you need to use the <animateTransform> tag instead of <animate>, and you need to add a *type* attribute to specify which transform you are animating, such as “rotate” or “translate”. Here, for example, is a transform animation applied to a group:

<g transform="scale(0,0)">

<animateTransform attributeName="transform" type="scale" from="0,0" to="0.4,0.7"

begin="3s" dur="15s" fill="freeze"/>

<rect x="-15" y="0" width="30" height="40" fill="rgb(150,100,0)"/>

<polygon points="-60,40 60,40 0,200" fill="green"/> </g>

The animation shows a growing “tree” made from a green triangle and a brown rectangle. In the animation, the transform goes from *scale*(0,0) to *scale*(0.4,0.7). The animation starts 3 seconds after the image loads and lasts 15 seconds. At the end of the animation, the tree freezes at its final scale. The *transform* attribute on the <g> element specifies the scale that is in effect until the animation starts. (A scale factor of 0 collapses the object to size zero, so that it is invisible.) You can find this example, along with a moving rectangle and a keyframe animation, in the sample file *svg/first-svg-animation.svg*.

You can create animated objects in the <defs> section of an SVG file, and you can apply animation to <use> elements. This makes it possible to create hierarchical animations. A simple example can be found in the sample file *svg/hierarchical-animation.svg*.

The example shows a rotating hexagon with a rotating square at each vertex of the hexagon. The hexagon is constructed from six copies of one object, with a different rotation applied to each copy. (A copy of the basic object is shown in the image to the right of the hexagon.) The square is defined as an animated object with its own rotation. It is used as a sub-object in the hexagon. The rotation that is applied to the hexagon applies to the square, on top of its own built-in rotation. That’s what makes this an example of hierarchical animation.

If you look back at the cart-and-windmills example now, you can probably see how to do the animation. Don’t forget to check out the source code, which is surprisingly short!