Chapter 2. Designing and Developing a Program

The previous chapter characterizes programming as a craft that uses the medium of a computer programming language to create an artifact called a program. A programmer is thus a craftsperson who creates programs that, when executed on an appropriate data set, can be used to solve specific problems or, more generally, to explore, express, or entertain.

Because a fundamental part of being a craftsperson is understanding the materials and dynamics of the medium itself, this chapter begins by considering what it is that Processing programs are made from. Processing provides all the basic elements found in most computer programming languages, but it is geared principally toward the generation, combination, and manipulation of digital images to produce dynamic and interactive visual experiences. Thus, an important part of beginning to acquire an understanding of our medium is the exploration of the kinds of digital image "objects" that form the basis not only of programming in Processing but also of all forms of graphical input and output that operate throughout computing in general. The three primary categories of digital image concepts are basic geometric elements, digital photos, and typographical onscreen text.

In this chapter we will illustrate these three digital image concepts and introduce how they are implemented in Processing. We will also begin to work with the notion of algorithms as a way to design programs.

2.1. Example: Our First Processing Sketch

Imagery is an important component of communication. For example, the first photographs of the Earth taken from space were images that truly transformed the way people thought about this planet. Indeed, these images are still quite compelling ones, even today. Now, suppose we are participating in a project — scientific, educational, artistic, or otherwise — that is centered on the use of satellite-based technologies to pinpoint locations on the Earth from space, perhaps to raise awareness of global ecology or to explain the application of GPS tracking technology. Our goal is to create some sort of visual creation in which the key image is that of the Earth as viewed from space that might be useful in a conference presentation, information kiosk, web site, and so on.

Now, one might wonder why we want to create a computer *program* to do this instead of simply using an application software package in which one can combine photos, text, and geometric elements. As this exercise will illustrate, one reason is that this enables you to create works that are far more *dynamic* than is at all possible with application software in which the design options that one can use are limited. Our goal is to create an onscreen presentation that *changes* — even *randomly* — each time it is viewed.

In the Processing programming environment, programs are called *sketches*. This term highlights the visual nature of programming in Processing and also suggests the widely practiced concept of beginning a

design process by “sketching out ideas." Professional animators often begin with hand-drawn sketches on paper so that when they begin their software modeling of the various characters and objects needed for their animation, they can use these sketches as a guide. The concepts, characteristics, and activities inherent in such a notion of “sketching” can be strongly recommended for computer programming as well, perhaps especially when it involves visual elements. But sketching out an idea, whether verbally or visually, can also be communicative and collaborative, allowing others to share in one’s idea. When “sketching” in Processing's programming environment, you will definitely find yourself exploring ideas *verbally* as you study the language of Processing and *visually* as you learn about the image elements that its statements produce.

By definition, a sketch is “sketchy,” in the sense that it is understood to be incomplete, open to change. Articulating an idea too completely and with too much detail can prematurely limit one’s creativity and greatly discourage helpful input from others. A sketch of an idea is also understood to be fairly “disposable,” keeping open the important option of putting this particular version of the idea aside and starting over on a new one.   Thus, such a notion of “sketching” helps to maintain a productive balance between planning and discovery, as well as between individual creativity and collaboration.

For our example, such sketching might produce something like that shown in Figure 2-1. It is important to remember that this is only a “sketch,” only a *provisional* design idea, one that will serve to get us started in this process. As in other fields of design, our initial sketch of our idea will evolve as we undertake this implementation. However, what we learn from this initial implementation can inform our next design and implementation. This kind of *incremental and iterative approach to design* is often celebrated as a highly productive one, in a wide range of fields.

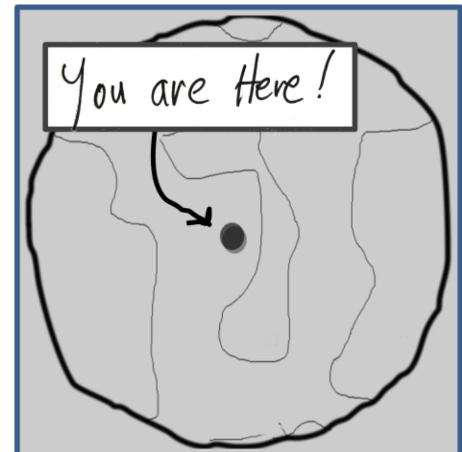


Figure 2-1. A sketch of our initial design

This sketch shows several graphical elements that will be needed and that are examples of the three categories of digital image concepts we listed earlier:

- an image of the Earth
- the text “You are Here” drawn near the top of the display window;
- two simple geometric objects: points marking locations on the Earth’s surface and lines connecting the text with the points;

In this chapter we will use this sketch as an initial design and proceed to examine how the various graphical elements can be implemented. The output that will be produced will be orchestrated by program code written in the Processing language. Near the end of the chapter we will see how Processing programs are capable not only of producing static images as implied by the initial sketch and supported by most image processing applications, but also dynamic images and, in later chapters, animated images and interactive user experiences.

2.2. Working with Pixels and the Coordinate System

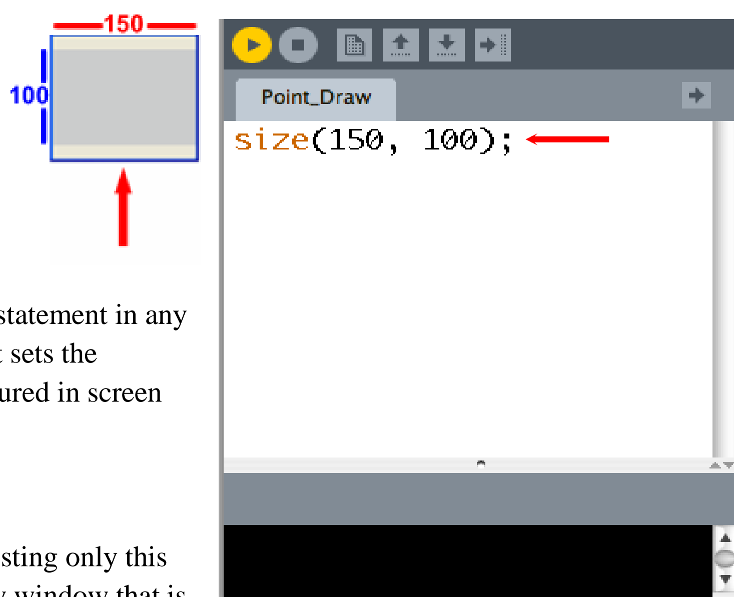
The simplest graphical elements shown in our sketch are the points. Before seeing how these can be drawn by a program written in the Processing language, however, we must first look at the particulars and the dynamics of the Processing programming environment. It is important to do this first because this environment is the actual medium in which we will be seeking to implement the sketch. We will first discuss pixels and the canvas before moving on to points.

2.2.1. Screen Pixels

All visual information presented on a computer screen is comprised of tiny squares of colored light called "pixels." The term "pixel" is short for "picture element" and means, in general terms, "a single colored piece of the picture." A digital camera stores a photographic image as a file containing a grid of individual color codes, each of which might be called a "photo pixel." Everything we see on a computer screen is comprised of single-colored squares of light, which we will call *screen pixels*. The "screen resolution" of a computer monitor describes how many pixels comprise a full-screen image. For example, a monitor set to 1280 × 1024 presents a full-screen image that is comprised of 1280 columns and 1024 rows of screen pixels.

2.2.2. The Blank Canvas: Processing's Display Window

Almost any application software that involves working with digital images makes use of some sort of blank background object that functions as a "container" into which digital image objects can be placed and also as a "frame" defining the only portion of the assembled image elements that will be visible on the screen. For example, in the case of presentation software, this is called the "slide background." In some multimedia authoring software, it is called the "stage." In software dedicated primarily to digital images, it is commonly called the "canvas."



The Processing programming environment is based upon a similar idea, where all visual output is placed within what is called the "display window." It is here that "views" for any software models of image elements in your Processing program will be rendered in the form of screen pixels. Many times, the first interesting statement in any program written in Processing is one that sets the dimensions of the display window, measured in screen pixels; for example:

```
size(150, 100);
```

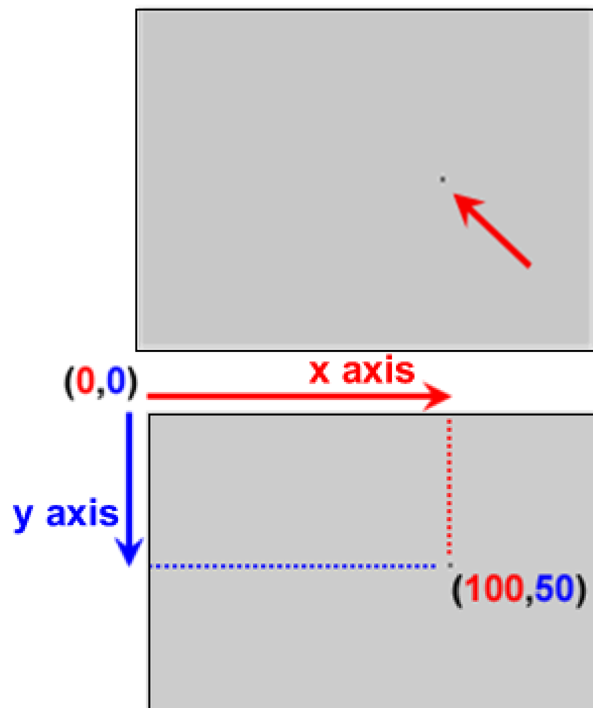
To illustrate, a Processing program consisting only this statement in its body will create a display window that is 150 screen pixels in *width* and 100 screen pixels in height when the program is run. In other words, the

display window is comprised of 150 columns and 100 rows of screen pixels for a total of 15,000 screen pixels. At any given time, each screen pixel will be assigned a single color of light. This set of colors that are defined for all of the screen pixels that comprise the display window is sometimes understood as constituting a single visual “frame.” When we undertake to create animations, we will take special note of “frame rate,” a setting that determines how many times per second it is that the colors of all of the pixels in the display window are updated.

2.2.3. Points and Coordinates

By default, all screen pixels in the Processing display window are gray. However, adding the following statement will instruct Processing to draw a **point** on the screen, represented in the form of a single **black** screen pixel at location **(100,50)**.

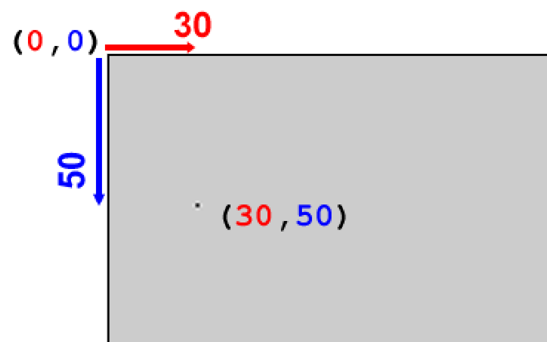
```
size(150, 100);
point(100, 50);
```



In computer imaging, pixels are addressed in a way that is similar to the way that points are described in geometry — with a pair of *x* and *y* coordinates. However, in a computer image, the pixel in the *upper-left corner* is considered to be at location (0, 0), with the ***x* axis** increasingly horizontally to the *right*, and the ***y* axis** increasing in a *downward* direction. The (**x, y**) location of this screen pixel within the Processing display window is, therefore, (100**,** 50).

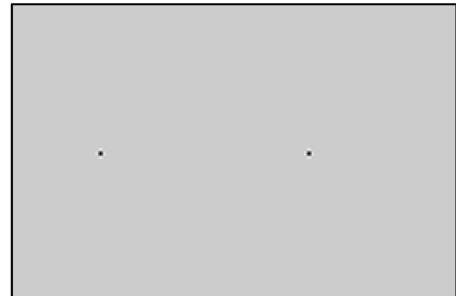
Similarly, if we wanted to draw the point in a different location, we could edit this statement to use different parameters:

```
size(150, 100);
point(30, 50);
```



And if we include *both* versions of these instructions to draw a point, then two points are rendered at the specified locations within the display window.

```
size(150, 100);
point(30, 50);
point(100, 50);
```



Our Processing program now consists of three "statements" — one that defines the dimensions of the output window and two that draw points at the locations specified by their respective arguments. Notice the semicolon at the ends of the statements; ***all statements in Processing must end with a semicolon (`;`).***

> **NOTE:** *You are strongly urged to open Processing and actually run the examples as you work through this chapter to experience for yourself what they do.*

2.2.4. Calling Methods

The particular statements used in our programs so far are examples of "calls" to `size()` and `point()`, two of the many methods that are pre-defined as part of the Processing programming language. This is not unlike the functions that one uses in a typical spreadsheet program, where any arguments that the function needs to undertake its predefined actions are supplied within a pair of parentheses, separated by commas. Each of the method calls that we have seen has the following basic form:

Method Call

```
methodName(argumentList);
```

where

- *`methodName`* is the predefined name (also called an *identifier*) of a particular method,
- *`argumentList`* is a list (possibly empty) of items called *arguments* expected by the method that are in a specified order, enclosed in parentheses, and separated by commas

For example, the general form of the `size` method is:

```
size(windowWidth, windowHeight);
```

where

windowWidth is the width of the display window in screen pixels, and
windowHeight is its height, also measured in screen pixels.

The general form of the `point` method is:

```
point(xPosition, yPosition);
```

where

xPosition is the *x* coordinate of the screen pixel where the point should be centered, and
yPosition is the *y* coordinate of this same screen pixel.

2.3. Drawing Geometric Shapes

In addition to the points in our hand-drawn sketch in Section 2.3, the lines connecting the text with a location on the Earth's surface are also graphical elements. Processing provides a number of methods for drawing geometric elements. In this section we will consider lines, rectangles and ellipses.

2.3.1. Lines

Processing provides a method named `line` for drawing a line segment between two points. The general form of a call to this method is

```
line(x1, y1, x2, y2);
```

where

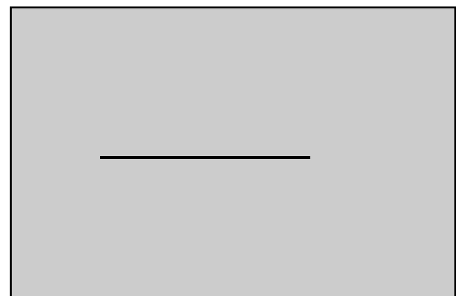
x1 and *y1* are the *x* and *y* coordinates of the first screen pixel, and
x2 and *y2* are the *x* and *y* coordinates of the second point.

To illustrate this, consider the two points that are rendered by the current version of our program:

```
size(150, 100);
point(30, 50);
point(100, 50);
```

If we edit our program to use the arguments from the two statements that call the `point` method and use them in a `line` statement instead, then our program will now instead render a line segment between these two screen pixels:

```
size(150, 100);
line(30, 50,  100, 50);
```



Note that the points and lines are drawn in black because *black is the default "stroke color*" used to render points, lines, and the edges of geometric shapes in the Processing environment. Later in this chapter, we will learn how to set the stroke color to other hues.

2.3.2. Rectangles

Processing also provides methods for rendering a number of basic geometric shapes. One of these is the `rect` method for drawing rectangles. The basic form of a call to this method is:

```
rect(x, y, rectWidth, rectHeight);
```

where

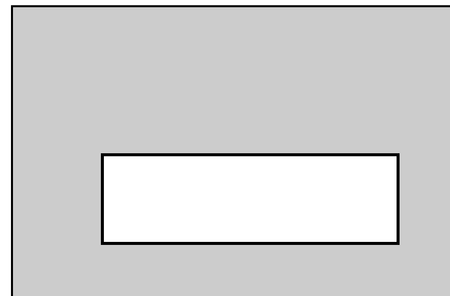
x and *y* are the *x* and *y* coordinates, respectively, of the screen pixel that is to be the top left corner of the rectangle,

rectWidth is the defined width of the rectangle in screen pixels, and

rectHeight is the defined height of the rectangle in screen pixels.

For example, suppose that instead of a line having the screen pixel (30, 50) as one endpoint, we wish to draw a rectangle with this point as its upper-left corner. We use the *x* and *y* coordinates — 30, 50 — of this point as the first two arguments in a call to the `rect` method and the width and height of the rectangle as the last two arguments. Thus, the following call to `rect` will render within the display window a rectangle that has its upper-left corner located at screen pixel (30, 50), is 100 screen pixels wide, and is 50 screen pixels high:

```
size(150, 100);
rect(30, 50, 100, 50);
```



The rectangle's interior is white because *white is the default "fill color"* used to render geometric shapes in Processing. Later in this chapter, we will learn how to set the fill color to other hues as well.

2.3.3. Ellipses

The basic form of the `ellipse` method for rendering ovals and circles is:

```
ellipse(x, y, ellipseWidth, ellipseHeight);
```

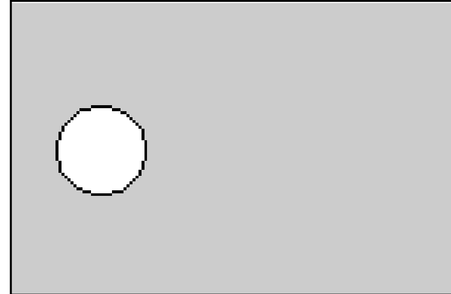
where

x and *y* are the *x* and *y* coordinates of the screen pixel defined as the center of the ellipse,

ellipseWidth is the width of the ellipse in screen pixels, and

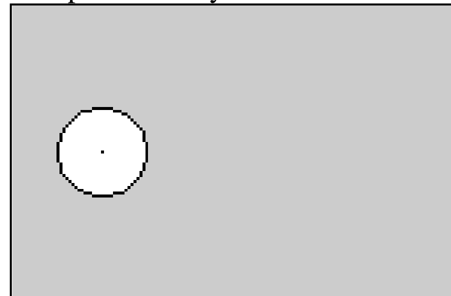
`ellipseHeight` is the height of the ellipse in screen pixels.

We can render a circle using the `ellipse` method simply by specifying that the width and height of the ellipse should be the *same*. For example, if we replace the call to `rect` in the preceding version of our program with the following call to the `ellipse` method, a circle with center at pixel (30, 50) and radius 30 will be rendered in the display window:



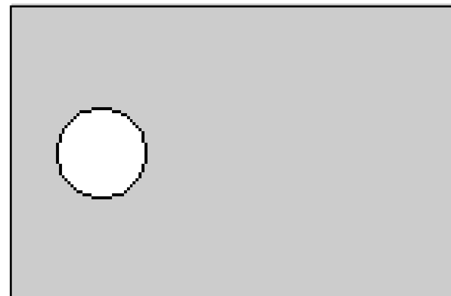
```
size(150, 100);
ellipse(30, 50, 30, 30);
```

There are several details worth noting here. First, notice that the *x* and *y* coordinates 30, 50 play a different role than in our previous call to the `rect` method. There they specified the screen pixel (30, 50) as the *rectangle's upper-left corner*, but in our call to the `ellipse` method they designate the *center of the ellipse* (in this case, a circle). To see this, we might add a call to the `point` method to draw the point with these coordinates. As we have learned, this will cause the screen pixel at location (30, 50) to be assigned the color black, the current stroke color, and in this case helps to visually illustrate that this screen pixel is the center of this circle.



```
size(150, 100);
ellipse(30, 50, 30, 30);
point(30, 50);
```

Something that needs to be underscored here is that *the order of the statements in a program is important! By default, statements are executed sequentially, from top to bottom, when the program is run. Consequently, changing the order of statements in a program will usually produce very different results.* For example, if we interchange the last two statements in our current program, we would *not* see a black screen pixel at location (30, 50), identifying the center of the circle.



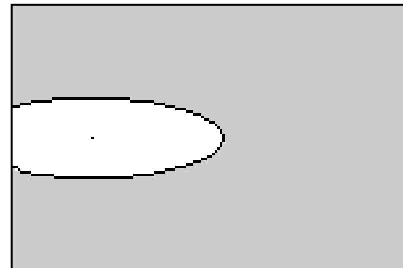
```
size(150, 100);
point(30, 50);
ellipse(30, 50, 30, 30);
```

Why did the point disappear? The reason is that like most image software, Processing interprets the sequence in which image objects are drawn as also defining the "*stacking order*" of those image objects in the display window. When an image object is rendered, it is interpreted to be visually *on top of* (or, alternatively, *in front of*) all other image objects already rendered.

This means, in particular, that a color assigned to a screen pixel in rendering this object will override the color that was assigned to that same screen pixel when rendering an image object interpreted as visually *beneath* (or, alternatively, *in back of*) this object. In this example, the fill color of the circle is by default a completely opaque white, so when this circle is rendered, the screen pixel at location (30, 50) is assigned *white*, not black. The effect is that the 100% opaque white used to fill the circle rendered by the call to the `ellipse` method visually "covers up" the black point rendered by the earlier call to the `point` method. Later in this chapter, we will learn more about the stacking order and how to define fill colors having degrees of *transparency* as opposed to their default 100% opaqueness.

However, while we're still discussing ellipses, let's also try drawing a *non*-circular ellipse at screen pixel location (30, 50) with width 100 and height 30 and also move the call to the `point` method back to its original location *after* the call to the `ellipse` method.

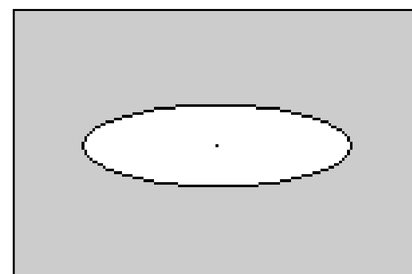
```
size(150, 100);
ellipse(30, 50,  100, 30);
point(30, 50);
```



The ellipse is not fully visible because it is too wide to fit within the display window. More specifically, it is geometrically impossible for an ellipse with center at the screen pixel (30, 50) and with a width and height of 100 and 30 screen pixels, respectively, to be rendered fully within the display window.

One way that we could ensure that this ellipse is rendered fully within the display window is by placing its defined center at the center of the display window itself. Given that the current dimensions of our display window are 150 screen pixels and 100 screen pixels, respectively, and that screen pixel location (0, 0) is defined to be the one located in the upper-left corner of the window, we might reasonably estimate the center of the display window to be at screen pixel (75, 50) and use these coordinates to revise our calls to the `ellipse` and `point` methods, which will then produce the following rendering:

```
size(150, 100);
ellipse(75, 50,  100, 30);
point(75, 50);
```



2.3.4. Example Revisited

Because we have now actually acquired the capability to write Processing statements to render a number of geometric elements, we can use them to produce a Processing program that is an approximation, sometimes called a *prototype*, of the original sketch in Section 2.1.

The first step in constructing a program is to design a finite set of operations, called an *algorithm*, that, when properly implemented, will produce the effect we desire. Producing an appropriate algorithm is part of the design process; it focuses on the basic operations and the sequence in which they need to be

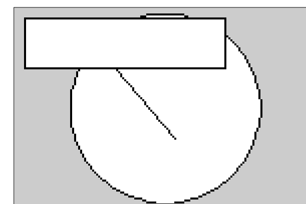
executed rather than on the actual syntax of the programming language. This allows us to design the structure of the program without being bothered by programming details. Our goal in this first iteration is to design a program that displays an approximation of the sketch shown in Figure 2-1. We can specify an appropriate algorithm for this goal using the following textual, enumerated format:

1. Create an display window that is 150 x 100 pixels.
2. Draw a 95 x 95 pixel ellipse in the middle of the display window.
3. Draw a 100 x 25 pixel rectangle on the upper left of the display window.
4. Draw a point somewhere on the surface of the ellipse.
5. Draw a line that connects the bottom of the rectangle and the point.

This algorithm comprises a finite set of relatively simple operations, all of which can be easily implemented using the programming tools presented in the previous sections. Step 2 draws a circle that can stand in as a kind of "placeholder" for the Earth photo we will eventually insert and step 3 draws a rectangle that can stand in for the text message that we will eventually add to the window. The algorithm also sequences these operations in a manner that will achieve our goal. As we learned in the previous sections, the sequence is often critical in getting programs to work properly. For example, we would not want to reorder steps 2 and 3 because then the ellipse would cover up the rectangle.

We can now implement this algorithm in a programming language. If the algorithm is well-designed, this step should be a relatively simple matter of faithfully translating the algorithm into Processing statements. We may have to fill in some relatively minor details as we go along, for example, exactly where on the ellipse should we draw the point? Here is a Processing program that implements this algorithm, using one Processing statement for each step in the algorithm.

```
size(150, 100);
ellipse(75, 50, 95, 95);
rect(5, 5, 100, 25);
point(80, 65);
line(41, 30, 80, 65);
```



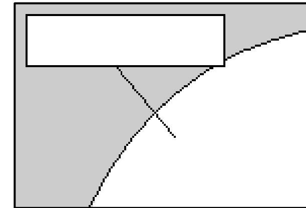
The output rendered by this program is shown on the right. Note that the point and the line are indistinguishable in the output because the line contains the point and they are both drawn with the same "thickness." We will address this issue in the next section

It should be evident that we have managed to take an incremental but substantive step in the direction of implementing our initial provisional paper sketch in the form of a first approximation of it as a Processing program. However, we have also made a few discoveries as we experimented with the various methods provided by Processing for creating geometric elements. *Reflecting upon such discoveries and considering how they might potentially be useful is an important part of the design process*. For example, we inadvertently discovered that it is possible to render an ellipse that is only partially visible within the display window. This might raise the question of whether it might be more visually compelling to have only part of the earth visible in the window to suggest a view of it from space.

We can actually explore this design idea by adjusting the arguments in the method calls in our current program. First, we can re-locate our circle to *beyond* the bottom-right corner of the window. The screen

pixel at that corner has coordinates (150, 100) and we can add various amounts to these values until we find a center that is to our liking. We might also find that a larger circle looks better. For example, we might arrive at the following changes to the program code, which produces the rendering shown:

```
size(150, 100);
ellipse(175, 160,  300, 300);
rect(5, 5,  100, 25);
point(80, 65);
line(41, 30, 80, 65);
```



We could also use this rendering as the basis for a revision of our earlier hand-drawn sketch, shown in Figure 2-2. Remember, *program design, like most processes of design, is* ***incremental****,* ***iterative****, and* ***inventive***! This new hand-drawn sketch and the algorithm shown above, along with a small change required for step 2 to accommodate our new sketch, can now lead us into our next attempt at implementing our sketch.

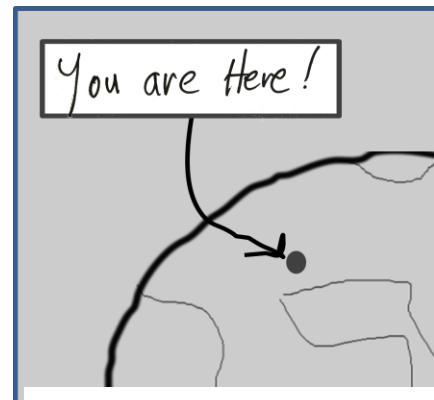


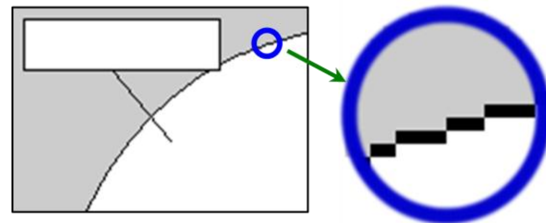
Figure 2-2. A modified sketch of our design

2.4. Attributes of Geometric Figures

The preceding Processing program and the result rendered is an improvement over what we had before, but there are a couple of issues that we will address in this section. The three diagonal line segments appear jagged as does the circular arc. Also, the line segments seem to visually obscure the three points being rendered by the calls to the `point` method. In this section we will address these issues using drawing attributes that configure how Processing renders geometric figures.

2.4.1. Smoothing

If we zoom in on the screen pixels in our rendering, we can better observe what is causing this jaggedness. At this magnification, we can see that the problem arises from the difficulty of trying to draw smooth diagonal lines or curves using block-shaped screen pixels.[1] The result is a kind of "stair-stepping" that is visible even in

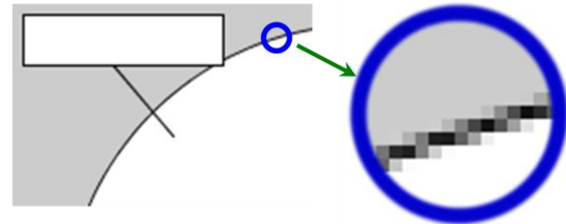


[1] Computer monitors and HDTV screens have *square* pixels. In contrast, standard television screens have *rectangular* pixels. Both square and rectangular pixels present the problem of jagged lines and curves. This causes many headaches for persons working in digital video, because they have to bear in mind whether their video is being rendered for a television screen or for an HDTV/computer screen.

the case of the small-sized screen pixels of a high-resolution computer monitor. The unsightly visual phenomenon is often referred to informally as “jaggies” but the more formal term for it is *aliasing*.[2]

Fortunately, Processing provides the `smooth` method to initiate the use of “anti-aliasing” when rendering the display window. To illustrate, inserting into our program a call to the `smooth` method — which is, incidentally, an example of a method that does not take any arguments — produces a rendering with much smoother diagonal and curved lines:

```
size(150, 100);
smooth();
ellipse(175, 160,  300, 300);
rect(5, 5,  100, 25);
point(80, 65);
line(41, 30, 80, 65);
```



If we once again zoom in on the screen pixels in our rendering, we can better understand why and how this "smoothing" took place. Smoothing — i.e., anti-aliasing — smoothes edges by softening the harsh contrast between the stroke color of that edge and the colors of the areas that border on this edge. This is accomplished by changing some screen pixels where this edge and the bordering fields meet to *intermediate* colors — i.e., shades between the color of the edge and the color of the bordering field. In this example, some of the black, white, and medium gray pixels are replaced with appropriate intermediate shades of gray, thereby softening the transitions from black to medium gray and from black to white.

Although anti-aliasing is a wonderful option to have available when renderings involve diagonal and curved lines, it also creates more work for the processor. It is good to be aware of program elements that are known to be more “processor-intensive” so that, in the event that their accumulated usage starts to visibly affect the speed or quality of the rendering, we can explore the possibility of not using some of them. For this reason, Processing provides a `noSmooth` method as a companion to the `smooth` method, enabling the programmer to toggle anti-aliasing on or off as needed for the visual quality of the rendering. No-smooth mode is the default mode.

2.4.2. Stroke Weights

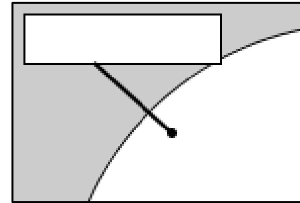
Let's address now the problem of how the points were obscured by the rendered line segments. As we have seen, Processing’s default stroke thickness, known as the *stroke weight*, is one screen pixel. This means that by default, it renders each point as a single screen pixel and each line as having a thickness of one screen pixel. This same stroke weight is also used when rendering ellipses and other geometric figures.

---

[2] This might seem like an odd term for this phenomenon. However, it is helpful if one considers how the word “alias” connotes the idea of a discrepancy between the reality of something and the way it is being represented.

However, Processing provides the **`strokeWeight`** method to change the stroke weight. The only argument that needs to be supplied to `strokeWeight` is an integer specifying the number of screen pixels to be used.[3] For example, if we insert a call to this method with stroke weights of 5 and 2 screen pixels is just prior to our calls to the `point` and `line` methods, then the point will be rendered with a diameter of 5 screen pixels and the line with a diameter of 2 screen pixels:

```
size(150, 100);
smooth();

ellipse(175, 160,  300, 300);
rect(5, 5,  100, 25);

strokeWeight(5);
point(80, 65);

strokeWeight(2);
line(41, 30, 80, 65);
```



Processing continues to draw the ellipse and the rectangle using the default stroke weight of 1 screen pixel since they are drawn before changing the stroke weight.

2.5. Modeling Color

Thus far all of the geometric elements — points, lines, rectangles, ellipses — have been rendered in black with white interiors for rectangles and ellipses. These rather drab renderings of our sketch formulated at the beginning of the chapter could certainly be greatly enhanced if we could add color — a colorful image on a black background with lines and points rendered in a distinguishable color. As we have mentioned, Processing does indeed support color: a scale of gray values ranging from black to white, called *grayscale color*, as well as a gamut of colors using combinations of red, green and blue, called *RGB color*.

2.5.1. Grayscale Color

Computer systems identify individual shades of color through the use of numeric codes. For example, the range of numeric codes used in modeling one of the more common "**grayscale**" palettes runs from **0**, which is assigned to **black**, through **255**, which is assigned to **white**, with 254 intermediate shades of gray between black and white, numbered from 1 to 254. Processing nicely allows you to reference these monochrome shades simply by providing the corresponding grayscale value from 0 to 255.

[3] Processing also provides a `strokeCap` method that specifies the way it draws line endings and a `stokeJoin` method for corners. The Processing web reference library http://processing.org/reference/ describes these methods and gives patterns for their use.

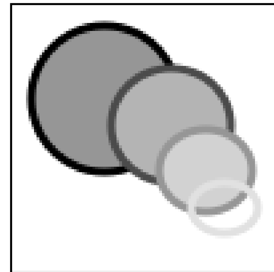
Recall from Chapter 1 that a *byte* is an 8-digit binary number, each binary digit of which is called a *bit*. In the case of this particular monochrome palette, one 8-digit byte is used for numbering all of the shades; accordingly, it is often called "8-bit grayscale." This also explains why the values in this scale run from 0 to 255. As we learned, with 8 binary digits, one can count from 00000000 (equivalent to 0 in decimal) up to 11111111 (equivalent to 255 in decimal).

The numeric codes assigned to shades of color in computer systems are not arbitrary. Rather, they are comprised of measurements of the amount of certain properties of light. For example, one way of thinking of grayscale is as a range of quantities of white light, where "black" is no white light at all and "white" is the maximum amount of white light defined by the scale.

Processing provides a variety of methods that make use of numeric color codes as arguments: the **`background`** method sets the background color of the display window; the **`stroke`** method sets the stroke color used when rendering points, line segments, and edges of shapes; and the **`fill`** method sets the color used to fill the interior area of shapes that falls within its edges. Processing also provides the **`noStroke`** and **`noFill`** methods that enable you to define any subsequent strokes or fills to be completely transparent. (Like the `smooth` method, these last two methods take no arguments.)

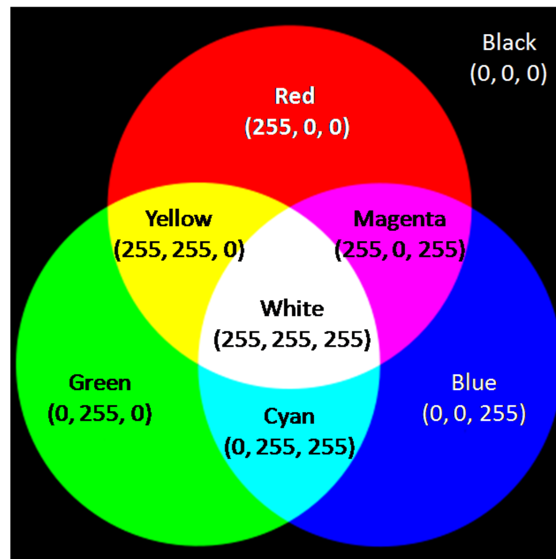
For example, the following code uses a selection of these methods to render an image using various shades of gray. The background color is set to white with `background(255);` and the line colors and the ellipse fill colors set to diminishing shades of gray using the `stroke` method. The call to the **`noFill`** method instructs Processing to leave the fourth ellipse unfilled; note how this allows the figures behind it to show through.

```
smooth();
background(255);
strokeWeight(3);

stroke(0);
fill(150);
ellipse(35, 35, 55, 55);
stroke(75);
fill(180);
ellipse(60, 45, 45, 43);

stroke(150);
fill(210);
ellipse(73, 62, 35, 31);

stroke(225);
noFill();
ellipse(80, 76, 25, 19);
```



2.5.2. RGB Color

The main numeric coding scheme for colors used by computer systems is called "RGB." This acronym refers to the fact that each shade of color is defined as a set of three numbers, each between 0 and 255, that describe a quantity of each of the three primary colors of light: red light, green light, and blue light, respectively. When these three quantities of red light, green light and blue light are mixed, the color of light produced is the particular shade of color that is associated with this trio of numbers. For example, the trio of RGB numbers for the three primary colors of light are (255, 0, 0) for pure red light, (0, 255, 0) for pure green light, and (0, 0, 255) for pure blue light. Similarly, the trio of RGB numbers for the three secondary colors of light are (0, 255, 255) for pure cyan light, (255, 0, 255) for pure magenta light, and (255, 255, 0) for pure yellow light. The way that colored light behaves when mixed is often



mapped out in the form of a color wheel like the one shown below.[4]

RGB color is sometimes called "**24-bit color**" because an 8-digit binary number is used to describe the amount of red light, another for the amount of green light, and another for the amount of blue light, for a total of 24 binary digits. This allows for 255 × 255 × 255 = 16, 581,375 different combinations of amounts of red, green, and blue light. Thus, using RGB, literally millions of different shades of light colors are possible.[5]

---

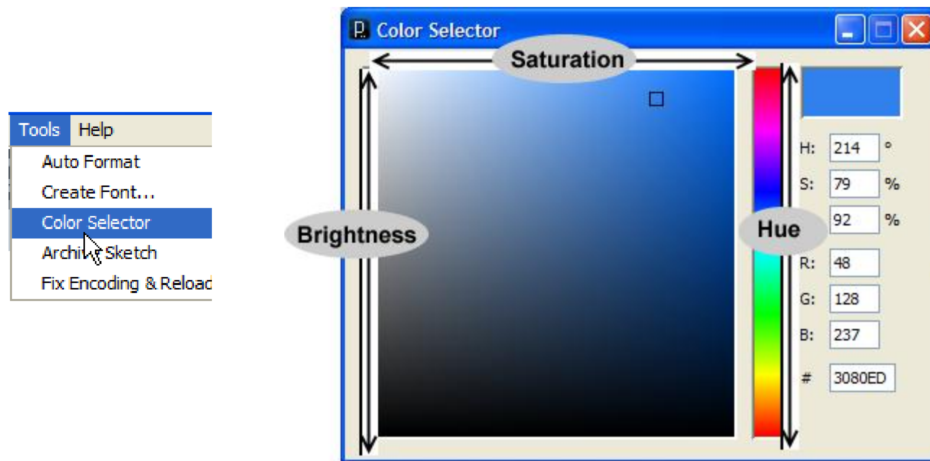
[4] This is often identified as an *additive* color model, underscoring the fact that what are being mixed are colors of light emitted from a light source, as opposed to the colored light that reflects off of colored pigments, which can be mapped using a *subtractive* color model such as the familiar red-yellow-blue color wheel that can be used for pigments, inks, and paints. Computer printers actually use what are the three main *secondary* colors of the RGB scheme — cyan, magenta, and yellow — as the primary colors of printer ink. Accordingly, this subtractive color scheme is identified by the acronym CMYK, where "K" refers to black (since "B" already refers to blue).

[5] It is not fully 16,581,375 different colors because some of these different combinations of red, green, and blue light produce colors that the human eye perceives as identical.

The RGB color scheme also describes the way digital scanners, cameras, and video cameras generate bitmap images using light-sensitive computer chips that sample the amounts of red, green, and blue light at various points within the frame of the illuminated image presented through the lens. For each sample, an image pixel is generated when the RGB code describing the quantities of red, green, and blue light detected at that point is stored in the resultant bitmap file. The RGB color model also fits fairly well to scientific models of the colors of light and the physiology of the rods and cones of the human eye.

However, humans tend to perceive and talk about colors primarily in terms of particular *shades* of color, rather than in terms of recipes for producing colors. This is one of the reasons why color models like the "HSB" numeric scheme are also used to describe colors within computer software systems. Briefly, the first term referred to by this acronym, "hue," refers to what we think of as the particular shade of color — i.e, a shade of red, yellow, blue, green, etc. The second term, "saturation" refers to how intense or vivid the color is, where less and less saturated versions of a particular hue will seem to be less and less "pure" and start to look increasingly pale until it eventually appears white. The third term, "brightness," refers to how light or dark this version of the hue is, where descending brightness values correspond to darker and darker versions of this hue until it appears black.

To help with selecting shades of color and identifying their corresponding numeric codes, the Processing programming environment provides a **Color Selector** option on the **Tools** menu that is similar to the sort of "color picker" that is found in many application software programs.



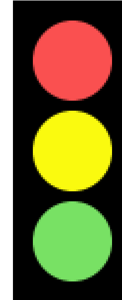
Processing's Color Selector tool provides a numeric description of each shade of color in both the RGB and HSB. In fact, it is often easier to pick a color by first using more of an HSB approach by using the colorful vertical bar on the right side of tool first to pick a hue, then clicking up or down within the main square panel to pick brighter or darker versions of this hue, and clicking to the right or left within this panel to select more or less saturated versions of this hue. Then, when you've found the shade of color you like, you can use the displayed RGB values in a method call.[6]

---

[6] Processing also provides the `colorMode` method that allows you to choose whether to use the RGB or HSB numbering scheme.

The following stoplight example demonstrates the use of RGB color using the same `background`, `stroke`, and `fill` method calls of the previous section but with RGB arguments. It has a black background and circles filled with shades of red, yellow and green.

```
size(60, 150);
background(0);
smooth();
noStroke();

fill(250, 80, 80);
ellipse(30, 30, 40, 40);
fill(250, 250, 15);
ellipse(30, 75, 40, 40);
fill(120, 225, 100);
ellipse(30, 120, 40, 40);
```

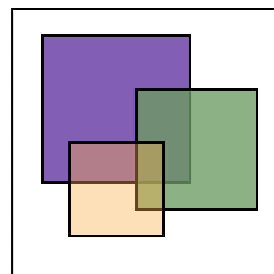


2.5.3. Transparency

In the context of high-end image editing software, it is common to make multiple *layers* of image elements, with the option of making a given layer semi-transparent so that the any layer below it will be partially revealed. Processing provides the option of making use of one of the more common schemes of transparency, one that is usually referred to by the acronym "**RGBA"** because it adds to the RGB color coding scheme the capability for specifying transparency through the use of an "**alpha**" value between **0** for full transparency and **255** for full opacity. Thus, RGBA is a 32-bit color scheme because of the addition of a fourth 8-digit binary number used to describe alpha transparency.

The following example illustrates transparency in rendering three overlapping squares. The smallest square sets its alpha value to 100, which means that the other two squares below it in the stacking order are not completely obscured. The middle-sized square has an alpha value of 200, which is not as transparent at the small square, but still allows the larger square to show through.

```
background(255);
fill(130, 95, 180);
rect(10, 10, 55, 55);

fill(108, 155, 100, 200);
rect(45, 30, 45, 45);

fill(250, 175, 75, 100);
rect(20, 50, 35, 35);
```



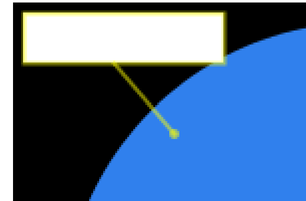
2.5.4. Revisiting the Example

As we noted earlier, we can enhance our Processing program in Section 2.3.4 considerably by adding color. We can set the display window's background color to black (0, 0, 0) to suggest the deepness of space and the stroke color of our line segments, points, and rectangle edges to yellow (255, 255, 0) to make them stand out against the black background. Also, we can use the RGB values (48, 128, 237) to fill our circle with the shade of blue shown in the earlier illustration of color picker. We might also like

the points and lines to be rendered fully opaque but the lines at approximately 50% transparency which we can achieve by adding an alpha value of 128 to the current call made to the `stroke` method that sets the stroke color to yellow (255, 255, 0, 128).

Because these upgrades require only minor changes to the algorithm we developed in Section 2.3.4, we will not include an updated algorithm here. The structure of the algorithm remains unchanged, but each step can now specify the desired attributes and color. The following code incorporates these changes and the rendering shows that we have achieved what we wanted.

```
size(150, 100);
smooth();
background(0, 0, 0);

fill(48, 128, 237);
noStroke();
ellipse(175, 160, 300, 300);

fill(255, 255, 255);
stroke(255, 255, 0, 128);
strokeWeight(2);
rect(5, 5, 100, 25);

strokeWeight(5);
point(80, 65);

strokeWeight(2);
line(50, 30, 80, 65);
```



Alternatively, we could have simply left off the alpha value in the call to the `stroke` method

```
stroke(255, 255, 0);
```

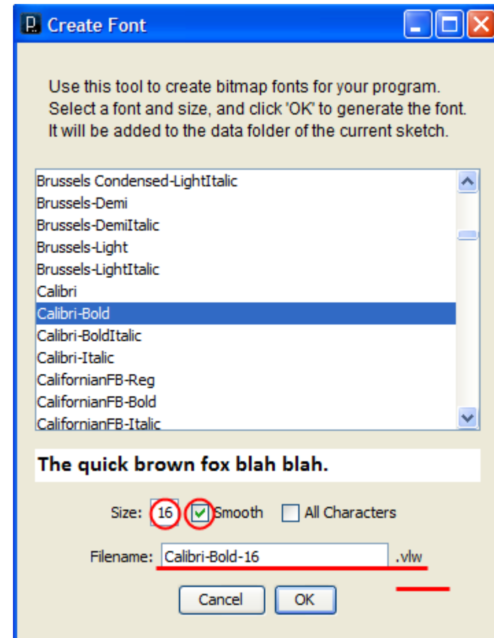
and the default 100% opacity of standard RGB would be used.

2.6. Working with Fonts

We have been using a rectangle in the upper left corner of the display window as a place holder for a caption, but our paper sketch has a text caption there. To add this caption, we need to look at how text can be rendered in the display window. Processing supports typesetting of text with *fonts*, which are complete sets of characters designed in a particular typeface (e.g., Calibri), size (e.g., 12 points) and weight (e.g., bold).

2.6.1. Creating Fonts

The first step required for rendering text in Processing is to select a font definition. Processing's font definition files have names that end in a "**.vlw**" extension. The easiest way to work with such files and font definitions is to use the "**Create Font**" option that Processing provide on the **Tools** menu. It allows you select a particular font, set certain font attributes such as its size and whether or not to use smoothing, and specify the filename that will be used to reference this font definition. (Note that the .vlw extension is added automatically.) For example, to use 16-pixel Calibri Bold as our font, we would simply select Calibri-Black in the list of fonts, specify its size as 16, opt for smoothing, and accept the default filename that Processing recommends for the font definition file that it will create for this font: `Calibri-Bold-16.vlw`.



In general, the best fonts for onscreen display are *sans-serif* fonts — those that do not have the small horizontal and vertical lines called *serifs* added. Also, fonts with uniform stroke weight throughout any given character will usually perform best for onscreen display. Thus, although the popular "Times Roman" font with its serifs and non-uniform stroke weight performs well in print, is not usually a good font for onscreen display. In contrast, the also popular sans-serif "Calibri" font with uniform stroke weight performs superbly in onscreen display.

2.6.2. Loading and Using Fonts

Once this font file has been created, we must load the font definition that it contains so that it will be available to use in rendering onscreen text in the current Processing program. Processing requires two methods for this:

- The **`loadFont`** method loads the font definition file specified by a filename given in double quotes. It places a copy of this file in a subfolder named data that will be created in the folder that contains our Processing program (a .pde file). We can check the contents of this data file by selecting the "Show Sketch Folder" option on the Sketch menu. It will open the folder that contains your current Processing program (with a .pde extension) and a data folder (if one is needed).
- Once a font has been loaded, the **`textFont`** method activates the font definition, which means that it will be used for any subsequent renderings of text in the display window (unless we change to a different font).

At this early stage, we will simply "nest" the call to the `loadFont` method inside the argument list for the `textFont` method; for example,

```
textFont( loadFont("Calibri-Bold-16.vlw") );
```

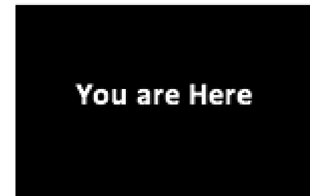
Don't worry too much about the unusual syntax of this “nesting” ; focus instead on the main ideas. In basic terms, the above statement would first call the `loadFont` method to load the font definition from the `Calibri-Bold-16.vlw` file. Then, the call to the `textFont` method with activate this font definition so it can be used in the program.

Now that we have a selected, loaded, and activated a font, we can use the `text` method to render text in this font. There are three arguments that this method needs: the string of characters that will comprise this text (enclosed in double quotes) and the *x* and *y* coordinates of the screen pixel that specifies the location of the upper-left corner of the text's location in the display window.; for example,

```
text("You are Here", 30, 50);
```

Also, thinking of font characters as more like filled geometric *shapes* than collections of points, curves, and lines helps one remember that the `fill` method — not the `stroke` method — is used to change the color of the font. The following example demonstrates this and shows the text that was rendered:

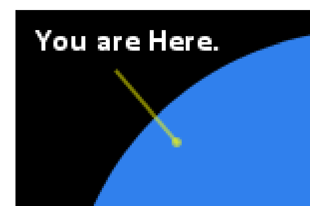
```
size(150, 100);
background(0, 0, 0);

textFont( loadFont("Calibri-Bold-16.vlw") );
fill(255, 255, 255);

text("You are Here", 30, 50);
```



2.6.3. Revisiting the Example

The rectangle we have been using as a placeholder for our text caption now seems superfluous, so we can remove it. We simply delete the calls to the `rect` method and the call to the `strokeWeight` method that precedes it. However, perhaps the resulting rendering without the white rectangle might suggest that we’d like to have our text be white, so we leave the `fill(255,255,255)` in place. With these changes, the code of our program and the rendering it produces are now the following:

```
size(150, 100);
smooth();
background(0, 0, 0);

fill(48, 128, 237);
noStroke();
ellipse(175, 160, 300, 300);

fill(255, 255, 255);
textFont( loadFont("Calibri-Bold-16.vlw") );
text("You are Here.", 10, 20);
stroke(255, 255, 0, 128);
strokeWeight(2);
rect(5, 5, 100, 25);

strokeWeight(5);
point(80, 65);
```



```
strokeWeight(2);
line(50, 30, 80, 65);
```

2.7. Working with Images

At the beginning of this chapter we stated that our goal was to develop a Processing program that combines a photo of Earth with geometric elements and a text caption in a way that seems both visually compelling and purposeful. We have used a variety of geometric elements and have created a text caption, so all that remains is to incorporate an image into our program.

2.7.1. Loading and Rendering Bitmap Images

To begin our study of how Processing enables us to do this, consider a bitmap image file named `blueMarble200px.png` that contains a 200 × 200 pixel version of the famous "blue marble" image, taken by the crew of Apollo 17. (For a detailed discussion of the issues involved working with images, see Appendix A).

The initial steps involved in working with bitmap images in Processing are quite similar to those for working with text fonts desribed in the preceding section.

- First, we must make the image file available for use in our Processing program by adding it to the folder that contains our Processing program (a .pde file). To simplify this step, Processing provides the "Add File" option on its Sketch menu. After selecting this option, we then browse to locate the image file we want to use in this program is stored and select it. To check that a copy of this file was placed in our sketch folder, we can use the "Show Sketch Folder" option on the same menu.

- Once the image file has been copied into our sketch folder, it is must then be loaded into our program with the **`loadImage`** method. Like the `loadFont` method, the `loadImage` method only has one argument: a file name, enclosed in double-quotes:

  ```
  loadImage("blueMarble200px.png");
  ```

Once our program has read in this bitmap image pixel data, it can be used to render an image on the screen in the form of screen pixels. For this we use the **`image`** method, the basic form of which contains three arguments: the *already-loaded* image pixel data for the image to be rendered, followed by the *x* and *y* coordinates of the screen pixel where the upper-left corner of this image will be located when it is rendered. Similar to what we did with fonts, one of the ways that we can specify this first argument to the `image` method is by "nesting" the call to the `loadImage` method inside the argument list of the `image` method:

```
image( loadImage("earth25px.png"), 0, 0);
```

The image-pixel data loaded from the bitmap file

(x, y) of the screen pixel where upper-left corner of image will be

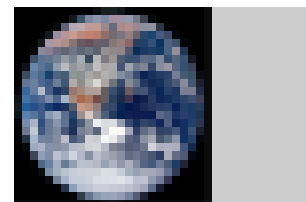
Again, for now, don't worry about the unusual syntax of this nesting of method calls and focus instead on the main ideas. In basic terms, the above statement will first peform the call to the `loadImage` method to read in the image pixel data from the `blueMarble200px.png` file. Then, the call to the `image` method uses this image pixel data to render it within the display window in the form of screen pixels, beginning with the screen pixel at location (0, 0). Thus, if our current `setup()` method consisted only of the following two statements, the display window would be rendered as shown:

```
size(150, 100);
image( loadImage("blueMarble200px.png"), 0, 0 );
```

The reason that some of the image of the earth is missing is that by default, the `image` method renders the image at 100% magnification: i.e., each image pixel is rendered in the form of a single screen pixel with an onscreen color that corresponds to the numeric RGB code assigned to that image pixel. Since we have set the dimensions of the display window to 150 × 100 screen pixels, only an image made of 150 × 100 image pixels will fit. For our image, the display window is not large enough to assign a screen pixel to each of the 200 × 200 image pixels of the bitmap image in `blueMarble200px.png`, so only the upper left portion of the image is displayed.

If we did want to have the entire image displayed, we could use an alternate form of the `image` method that has an additional pair of arguments: the width and height to be used when rendering this image. This has the effect of rendering the image at something *other* than 100% magnification. For example, the following modification of the earlier statements specifies that we wish to render the image using 100 × 100 screen pixels:[7]

```
size(150, 100);
image( loadImage("blueMarble200px.png"),
                    0,  0, 100, 100 );
```



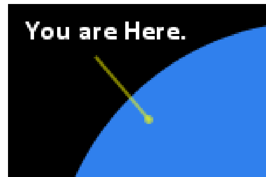
A form of "downsampling" takes place to reduce this 200 × 200 matrix of image pixel data to a 100 × 100 matrix of screen pixels. It should be noted, however, that although *reducing* the size of a bitmap image as we did here usually preserves good visual quality, *enlarging* a bitmap image quickly reduces its visual quality. For example, if we had used an image made of 25 × 25 image pixels, the resulting screen image would be quite fuzzy because each of the image

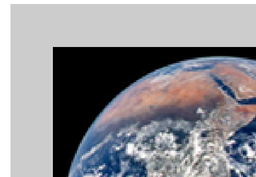
[7] Note that we have split the second statement across two lines. Splitting statements across lines is allowed in Processing.

pixels would be assigned a 4 × 4 block of screen pixels, all 16 of which are rendered in the same color, the one corresponding to the RGB code that was assigned to that image pixel.[8]

2.7.2. Revisiting the Example

Let's now use what we've learned about how Processing handles bitmap image files to get the Earth image inserted into our program. In the last prototype of our program, the statements served to produce the rendering shown below at the left. After some experimentation, it appears that adding the following statement near the beginning of our program will produce a rendering shown below at the right that positions the earth image at a comparable position within the display window.

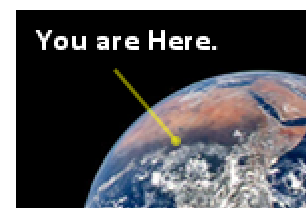




```
image( loadImage("blueMarble200px.png"), 25, 25 );
```

As with the last iteration, we will not include an updated algorithm because it simply needs to replace the "place holder" ellipse and rectangle with the desired text and image. Let's replace the three statements we used to render our blue circle with this statement. After doing so, here is the current version of our program and the rendering it produces.

```
size(150, 100);
smooth();
background(0, 0, 0);

image( loadImage("blueMarble200px.png"), 25, 25 );

fill(255, 255, 255);
textFont( loadFont("Calibri-Bold-16.vlw") );
text("You are Here.", 10, 20);

stroke(255, 255, 0, 128);
strokeWeight(5);
point(80, 65);

strokeWeight(2);
line(50, 30, 80, 65);
```



2.8. Documentation

As we look at our code, we can see that it is not just a bunch of individual statements that we just happened to discover will generate the result we want. Rather, our program is the result of thoughtful

---

[8] When an onscreen image has a "blocky" appearance like this, it is often called a "pixellated" image, because it looks almost as if the screen pixels that comprise this image themselves have become larger, to the point of being easily and undesirably visible to the human eye.

development and can be understood to consist of a carefully developed sequence of several major actions, each of which involves the rendering of one of the visual elements of our sketch. For this reason it is good programming practice to insert *comments* in a program to identify the basic actions and objects described by the each of the groups of statements — the *algorithm*, the step-by-step procedure we used in designing our program.

Comments within a program are merely for explanation and documentation; they are *not* interpreted as statements to be performed when the program is executed. They have one of the following forms in Processing.

Comments

```
/*
   Multi-line comments

*/
```

or

```
// Single-line comment
```

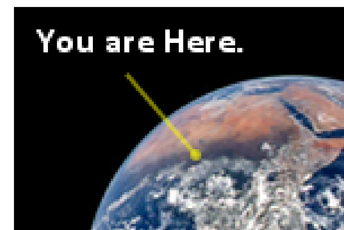
- For multi-line comments, Processing ignores the text between the `/*` and the `*/`.
- For single-line comments, Processing ignores the rest of the text on the line after the `//`.

As you look back at our program, you will notice that it has a "modular" structure in that the statements can be understood in terms of groups of basic actions and elements needed to produce the result we wanted. This modularity is a natural outgrowth of taking an intentionally *incremental* approach to programming. Rather than thinking of a program as just a long sequence of statements, we thought both in terms of the *overall* course of action that we wanted our program to accomplish and in terms of single steps that needed to happen in our program. Likewise, instead of thinking of our program as involving various facts and data, we thought both in terms of the individual "objects" and elements we needed as well as in terms of the overall relationship among them. Thus, like most design processes, programming involves some *analysis* — a word that literally means stretching something complex apart so that the component elements can be better examined. But it also requires thinking in terms of *synthesis* — the placing together of things that were previously separate. In each case, by thinking not in terms of isolated statements and facts but, rather, in terms of larger units of actions and elements, we arrived at code that was much more modular.

Such modularity in code also makes it easier to follow and also to refine. When writing a paper or giving a speech, having your remarks proceed in some kind of recognizable order — chronological order, for example — can greatly assist your reader or audience in understanding and remaining attentive to your remarks. For example, when describing some sort of process, chronological order is often used. When discussing a kind of network of relationships between multiple individual items, a spatial arrangement of visual aids can be very effective. Similarly, there is a rhetoric of programming. You are working in *language*, after all. When you are writing a term paper, periodically creating outlines of the work you

have done — or the work you have yet to do — can greatly assist you in recognizing and fine tuning the logic in the work you are doing. Sometimes, such an outline might suggest itself at the outset; at other times, the work may need to evolve more before such an order starts to emerge.

2.9. Revisiting our Example: One More Enhancement



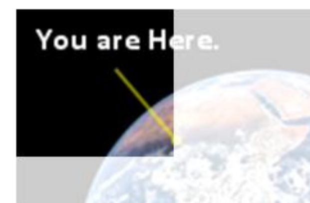
We have now come to the end of the chapter, having succeeded in creating a program that achieves the goals we set for ourselves in Sections 2.1 and 2.3.4 (see Figure 2-1 and Figure 2-2). However, as illustrative and instructive as this exercise has been thus far, it currently renders a visual result that we could have just as well created using application software. The real power of taking a programmatic approach — i.e., using a programming language — in regard to visual computing is in the ability to create works that are *dynamic* — i.e., that change automatically .

For example, our program draws outer space as a simple black background. What if we wanted to conjure up the image of deep space by adding a few stars randomly scattered around this background? Each time we run the program, we would like to get a *different* pattern of stars. Our "static poster" then becomes the kind of "dynamic poster" that we described in the opening description in Section 2.1.

To draw stars with random sizes and locations, we need to make use of a standard programming language construct that is used to simulate the behavior of natural systems in science, human systems in social science, computer and video games, to mention but a few: a *random number generator* that produces a sequence of numbers that follow no pattern — i.e., that appear to be random. In Processing this is provided by a method named **`random`**.

The `random` method can be called with either one or two numeric arguments. Calling it with two arguments will return some number that is greater than or equal to the first number and less than the second number. For example, `random(1,5)` will return a number between one and five. When only one argument will return a real number that is greater than or equal to zero but less than the value supplied in the call. For example, `random(3);` will return a number that is greater than or equal to zero but less than three. The second use of the method will be sufficient for our purposes here.

In our case, we would like to generate a seemingly random location of a point in space. One simple way to do this is to use the `random` method twice to randomly generate an *x* coordinate and then a *y* coordinate for the screen pixel to be used as the location of the star. In doing this we must make sure that these coordinates fall within an area of the display window that we perceive as within the outer-space portion of the image. For example, we could ensure that the *x* and *y* coordinates of the screen pixel are ones that always fall within the upper-left part of the display window.



Since the dimensions of our display window are 150 × 100, we could place stars at random locations whose x and y coordinates fall between 0 and 75. Thus, we can use the following code to place a single white star whose

diameter is some random value from 0 to 3 in a location that falls in the upper-left of the image:

```
stroke(255);
strokeWeight(random(3));
point(random(75), random(75));
```

This will not place stars over all the outer-space part of the image, and could place them on part of the Earth part of the image, but it is a reasonable approximation of what we would like. We can copy and paste this code four times to produce four random stars.

This upgrade requires a small addition to the algorithm that we designed in Section 2.3.4 and has served us so well throughout the chapter. We now need to add four additional steps to the end of the algorithm, each specifying the inclusion of a star. With these changes, the algorithm is as follows:

1. Create a display window that is 150 x 100 pixels.
2. Draw a 95 x 95 pixel ellipse in the middle of the display window.
3. Draw a 100 x 25 pixel rectangle on the upper left of the display window.
4. Draw a point somewhere on the surface of the ellipse.
5. Draw a line that connects the bottom of the rectangle and the point.
6. Draw a point of random width (0 to 3 pixels) in a random location (with x-y coordinates between 0 and 75).
7. Draw a second random point using the same characteristics.
8. Draw a third random point using the same characteristics.
9. Draw a fourth random point using the same characteristics.

The last four steps in this algorithm are rather repetitive and only produce four random stars. In future chapters, we will present ways to accomplish this goal more effectively. For now, we will have to live with this as a rough approximation of our goal.

The final version of our program, after implementing the changes to the algorithm and adding appropriate documentation, is shown below. Note that we have added opening documentation to describe what the program does, who wrote it, when it was written, and so on, and comments within the program identifying what each section of the code does.

```
/**
 * This program produces a "dynamic" poster that displays
 * a "You are Here" message connected to rondomly selected
 * points on an image of the Earth.
 *
 * @author nyhl, jnyhoff, kvlinden, snelesen
 * @version Fall, 2009
 */


// Create a display window with a black
background.
size(150, 100);
smooth();
background(0);
```

```
// Add an Earth photo.
image(loadImage("blueMarble200px.jpg"), 25, 25);

// Create the text caption.
fill(255, 255, 255);
textFont( loadFont("Calibri-Bold-16.vlw") );
text("You are Here.", 10, 20);

// Draw the point.
stroke(255, 255, 0, 128);
strokeWeight(5);
point(80, 65);

// Draw a line from the text caption to the point.
strokeWeight(2);
line(50, 30, 80, 65);

// Add some randomly sized white stars to the background.
stroke(255);
strokeWeight(random(3));
point(random(75), random(75));
strokeWeight(random(3));
point(random(75), random(75));
strokeWeight(random(3));
point(random(75), random(75));
strokeWeight(random(3));
point(random(75), random(75));
```

Each time we run this program, we get the same graphical elements from before but a different display of stars in the background.

2.10. Incremental/Iterative Development

This chapter begins with a communicative goal of developing a motivational poster. We embodied this goal in an initial design shown as a sketch in Section 2.1 and then again in an improved sketch shown in Section 2.3.4. Based on the sketch, we identified the key graphical elements required to implement that sketch and then introduced the basic tools provided by Processing for implementing these graphical elements.

Throughout the development process, our approach has been both *iterative* and *incremental*. The approach is iterative in that it takes numerous passes over the problem producing a prototype on each pass and it is incremental in that it cumulatively implements one graphical element at a time culminating in the final product. This approach, which has much akin with agile software development, strikes a balance between careful pre-planning on the one hand and exploratory development on the other. We will call this approach Incremental/Iterative Development (IID). IID comprises the following basic phases:

1. **Analysis** – This phase articulates the general goals that motivate and drive our work. It generally asks “what” questions – What are our customers asking for? What do we want to accomplish? In the chapter example, our goal was to produce a motivational poster based on an image of the Earth from outer space.

2. **Design** – This phase takes the goal or goals articulated in the analysis phase and creates a plan for achieving them. It generally asks "how" questions – How can we achieve our goals? The design of graphical output for Processing applications usually includes a sketch and a prioritized list of the graphical elements in that sketch. It will also include specifications of the data structures and algorithms required to produce the desired sketch. In the chapter example, we started with the sketch and graphical objects listed in Section 2.1.
3. **Implementation** – This phase faithfully translates the design into a working prototype program.
4. **Testing** – This phase reviews the quality of the current prototype. The impressions or feedback that we get generally drive future iterations of this process.

In this chapter, Section 2.1 articulated the analysis (step 1) and the preliminary design (step 2). Sections 2.2 - 2.7 discussed a sequence of iterations (steps 1-4) each adding important features to the prototype. The final prototype was discussed in Section 2.9. As it turned out, the analysis remained roughly constant throughout the chapter, but the design changed from time to time and the prototype implementation changed in each section.

It is important to view process models such as this one as guidelines rather than hard rules. Process models help organize the effort on a project but can stifle creativity if followed too strictly. This particular model, with its iteration and incremental development provides a nice balance between order and discovery. All future chapters in this text will adopt this model of development.