Lesson 15: Graphics

The purpose of this lesson is to prepare you with concepts and tools for writing interesting graphical programs. This lesson will cover the basic concepts of 2-D computer graphics in more depth than was covered in “Lesson 6: Turtle Graphics”. This lesson shows how to draw still images. A later lesson will handle animation and mouse and keyboard input.

Although the examples in this lesson use the `cpif.graphics` module that came with this book, the concepts learned in this lesson are generally useful in many different graphics systems.

Introducing Computer Graphics

This section introduces some basic principles of computer graphics.

Pixels

Your computer screen displays text and pictures. Each letter and each image is made of lots of tiny colored dots called *pixels*. These pixels are evenly spaced in a rectangular grid over the surface of the computer's display screen. Creating computer graphics is a matter of making the right pixels turn the right color at the right time.

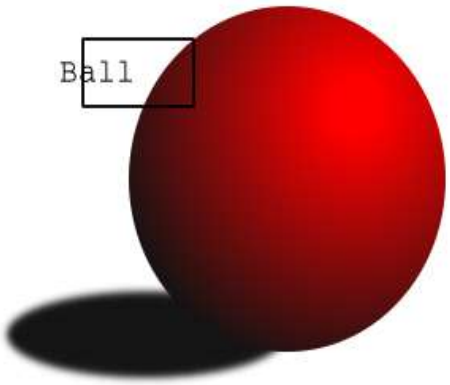

Illustration 1Computer graphics image

Illustration 2Image expanded to show pixels

Coordinates

Coordinates are numbers used to identify points in 2 or 3-dimensional space. In this book we will concentrate on 2-dimensional space; in other words, a flat plane.

A point on a plane is identified by two numbers. In the rectangular coordinate system that we will use, the two numbers are the distance from a vertical measuring line to the point, and the distance from a horizontal measuring line to the point. By convention, the first number is often called *x*

and the second number is called y. The horizontal measuring line is called the *x-axis;* the vertical measuring line is called the *y-axis*.

Coordinates in math textbooks

You may be used to the rectangular coordinate system you were first taught in school. If you haven't studied that yet, don't worry. I'll briefly explain it here. Coordinates are commonly written as two numbers inside of parenthesis, separated by a comma, like this: (2.5, 3.4). The first number, 2.5, is the x value, that is, the horizontal location of a point. The second number, 3.4, is the y value, that is, the vertical location of a point.

In math textbooks the origin (0, 0) is at the center. The x value increases to the right, and the y value increases going up. In the illustration below, the thickest horizontal line is the x-axis and the thickest vertical line is the y-axis. The point (2.5, 3.4) is plotted like this:

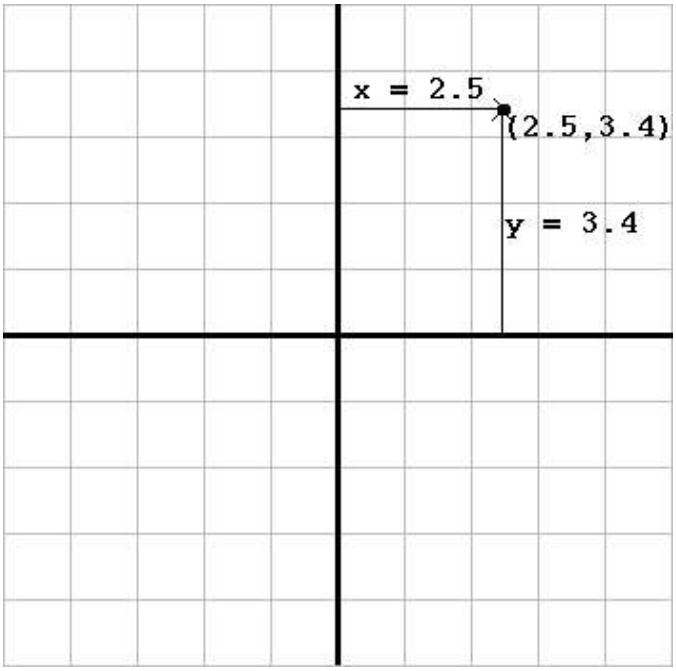

Illustration 3Rectangular coordinate system used in math textbooks

Pixel Coordinates

Pixel coordinates as commonly used in computer graphics are similar to the rectangular coordinates used in mathematics textbooks, with a few differences. First, pixel coordinates identify individual pixels, so they are composed of integers, not fractional numbers. If a computer graphics function accepts fractional floating point numbers as coordinates, internally those get rounded to integers anyway. Second, the origin or (0, 0) coordinate is commonly at the upper-left hand corner, not the center. Third, in pixel coordinates the y coordinate values increase going down.

The illustration below shows a computer graphics image with some pixel locations highlighted.

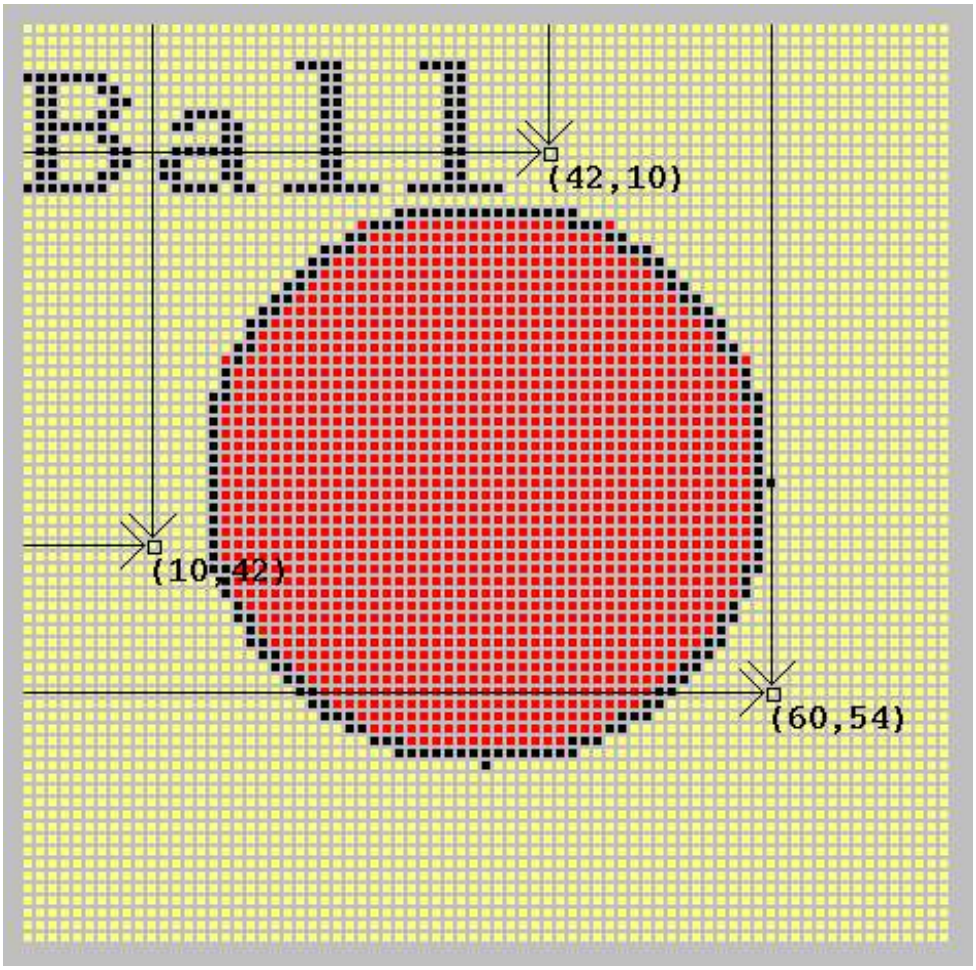

Illustration 5Pixel coordinates

The careful reader may have noticed that the pixel at coordinate (10, 42) is actually the 11th pixel from the left margin, and the pixel at (42, 10) is actually the 11th pixel down from the top margin. That is not a mistake. Counting pixels, like counting locations in a Python list, begins at zero.

Colors

Every pixel has a color. Colors are usually described in computer graphics using three numbers that specify intensities of red, green, and blue. Computer monitors don't actually display colors in their real shades, but rather as varying intensities of red, blue, and green dots. Because the dots are so close together our eyes see blended colors instead of little dots.

The graphics programming library used in this book gives two choices for specifying colors. You may use common English names for colors, in the form of lowercase strings. (Both the American

and British spellings of 'gray' and 'grey' are accepted.) Or you can use tuples of red, green, and blue numbers. Each number is a color intensity between 0 and 255, inclusive.

For example:

`'black'` and `(0, 0, 0)` are two ways of specifying the color black.

`'red'` and `(255, 0, 0)` are two ways of specifying the color red.

`'green'` and `(0, 255, 0)` are two ways of specifying the color green.

`'blue'` and `(0, 0, 255)` are two ways of specifying the color blue.

`'white'` and `(255, 255, 255)` are two ways of specifying the color white.

The tuple `(200, 100, 50)` results in a sort of reddish-brown color.

You'll get to try your own computer graphics colors soon.

Drawing with DrawingWindow

The `cpif` software package that comes with this book includes a *class* called DrawingWindow that provides a simple way for you to create your own computer graphics and animation. (See “Introducing Python Classes” on p. 104. You can read most of this section without knowing the details about classes.)

Let's create a DrawingWindow and draw a line on it. Run these commands in the Python Shell window:

```
>>> from cpif.graphics import DrawingWindow
>>> dw = DrawingWindow(100, 100)
```

A small, blank window will appear.

Now let's draw a line. You'll probably need to click the mouse on the title bar of the Python Shell window to restore its keyboard focus before typing this next command.

```
>>> id = dw.line(10, 10, 90, 90, 'black')
```

A diagonal black line appears:

Illustration 6A line in a DrawingWindow

When you are tired of looking at the line, you can tell the the drawing window to delete it.

```
>>> dw.delete(id)
```

When you are done with the window, tell it to close itself.

```
>>> dw.close()
```

Once you have closed your drawing window you cannot draw with it any longer.

Let's go back over the commands you have just used.

```
>>> from cpif.graphics import DrawingWindow
```

This imports the `DrawingWindow` class from the graphics module in the `cpif` package.

```
>>> dw = DrawingWindow(100, 100)
```

This creates a `DrawingWindow` object with a drawing area 100 pixels wide and 100 pixels tall, and gives the new object the name dw.

```
>>> id = dw.line(10, 10, 90, 90, 'black')
```

This creates a black line in the window starting at pixel coordinate (10, 10) and ending at (90, 90). The number returned by the `line()` method identifies the object you just drew. Most of the `DrawingWindow` methods return an ID for the object you just drew in the window. You can use this ID later to delete the object, using the `delete()` method, like this:

```
>>> dw.delete(id)
```

You can get help on the line method, or any other method in the `DrawingWindow` class, using the help function, like this:

```
>>> help(DrawingWindow.line)
```

Please ignore the first parameter, called `self`. (I explain *self* when I talk about Python classes. As far as you are concerned, when you draw a line the first parameter is `x1`.)

I encourage you to run `help()` on new methods and functions so you can learn more about them. In this case, `help()` told us that there is an optional keyword parameter called `width`. What do you suppose that does?

So much for drawing a line. Let's try drawing some other things. But first, some good advice:

Before creating a new DrawingWindow, make sure you get rid of the old one.

Close the old DrawingWindow by using the following command, if you have not done so already:

```
>>> dw.close()
```

Or you can close by clicking the close button (usually containing a symbol that looks like an X) in the upper-right corner of the window frame. But if you close the window by clicking its close button, make sure you do not call methods on **`dw`** (the old DrawingWindow object) or you will get errors. You can re-use the **`dw`** variable to create a new DrawingWindow, as in the next example.

Drawing Text in the Drawing Window

Type the following commands in the Python Shell window:

```
>>> dw = DrawingWindow(75, 75, background='yellow')
>>> text = dw.drawText(0, 0, 'Ball')
```

```
>>> circle = dw.circle(37, 37, 18, 'black', fill='red')
```

This displays a drawing that looks like this:

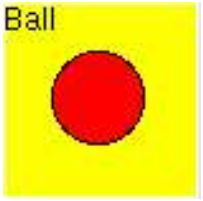

Illustration 7Ball in DrawingWindow

Hey, that looks familiar!

Notice that you passed an extra keyword parameter, `background='yellow'`, to the `DrawingWindow` class to make the background yellow.

You used the `drawText()` method to draw the word *Ball* on the window. The first two parameters to `drawText()` give the x and y location of the text. The location (0, 0) is the upper-left corner of the window. This indicates that the upper left corner of the text should be at the top-left corner of the screen.

You can get the width and the height of the text, in pixels, by passing the ID returned by drawText() to the getTextSize() method, like this:

```
>>> dw.getTextSize(text)
(7, 13)
```

Your numbers might be different, depending on what font your computer uses.

Getting help on the circle method

Type the following command to get an explanation of the `circle()` method:

```
>>> help(dw.circle)
```

Notice that this is the same as calling `help(DrawingWindow.circle)`. (Remember to ignore the `self` parameter for now.)

We put the center of the circle at (37, 37), which is the center of the window. 37 is roughly half of 75, which is the width and height of the window. The next parameter we gave to `circle()`, 18, is the radius, or distance in pixels from the center of the circle to the outer rim.

There are several more drawing methods in `DrawingWindow`. To get the whole list of methods yourself, you can type this command:

```
>>> help(DrawingWindow)
```

You will see a lot of drawing methods. You might want to try some of these yourself.

Drawing Colored Boxes

As I promised earlier, here is your chance to try out the various colors that Python can make with your computer.

I'll show you a simple program to demonstrate colors. First it will draw boxes, then we'll change it to draw colored boxes, with the color of each box depending on it's position.

Go to the Python Shell window and select “File”, then “Open”. Open the **`colors1.py`** file in your **`samples`** directory. Here is the code:

```
# colors1.py
# Draw boxes - in the next version they will be colored

from cpif.graphics import DrawingWindow
dw = DrawingWindow(256, 256)

for y in range(0, 256, 16):
    for x in range(0, 256, 16):
        dw.box(x, y, x+16, y+16, 'black')

dw.run()

# end-of-file
```

Press F5 to run the program. You should see a window that looks something like this:

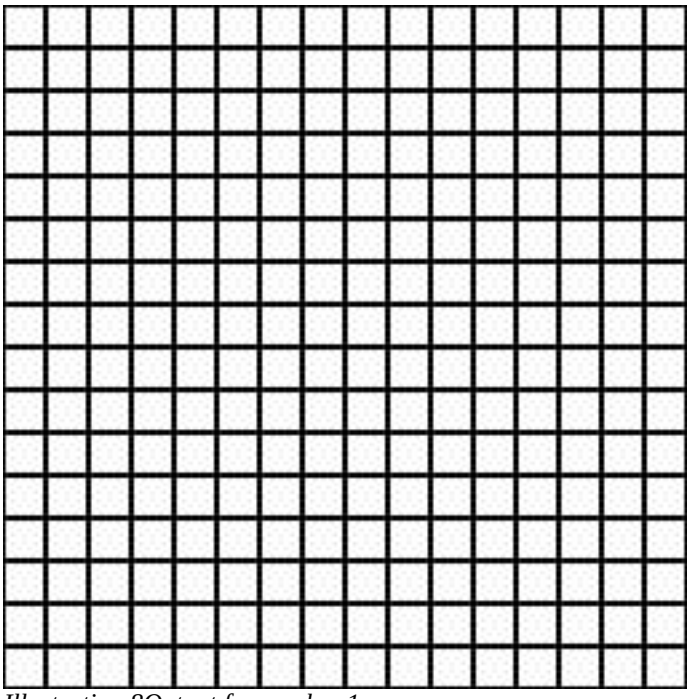

Illustration 8Output from colors1.py

This doesn't look very colorful yet, I admit. To make this more interesting, let's calculate a fill

color for each box that this program draws.

We can specify a color either as a string containing a color name, such as `'black'`, or as a tuple of red, green, and blue amounts, like `(128, 64, 255)`. In the **`colors1.py`** program I chose a window size of 256x256 pixels. It draws 16 rows of 16 boxes. These numbers were chosen carefully. The x and y value for each box is always a number between 0 and 255, inclusive. It so happens that valid color numbers are also numbers between 0 and 255, inclusive. So we can use the coordinate of each box to generate a color tuple.

Now we'll modify the **`colors1.py`** program by finding this line:

```
        dw.box(x, y, x+15, y+15, 'black')
```

and changing it into two lines that look like this:

```
        color = (y, x, y)
        dw.box(x, y, x+15, y+15, 'black', fill=color)
```

Note that we added a new parameter to the call to the `box()` method, called **`fill`**. This indicates that we want the box to be filled in, and tells Python what color we want it to use for filling.

Open **`colors2.py`** in your **`samples`** directory. Here is the code, with the changes highlighted:

```
# colors2.py
# Draw colored boxes

from cpif.graphics import DrawingWindow
dw = DrawingWindow(256, 256)

for y in range(0, 256, 16):
    for x in range(0, 256, 16):
        color = (x, y, x)
        dw.box(x, y, x+16, y+16, 'black', fill=color)

dw.run()

# end-of-file
```

Now press F5 to run the program. You should see something that looks like this:

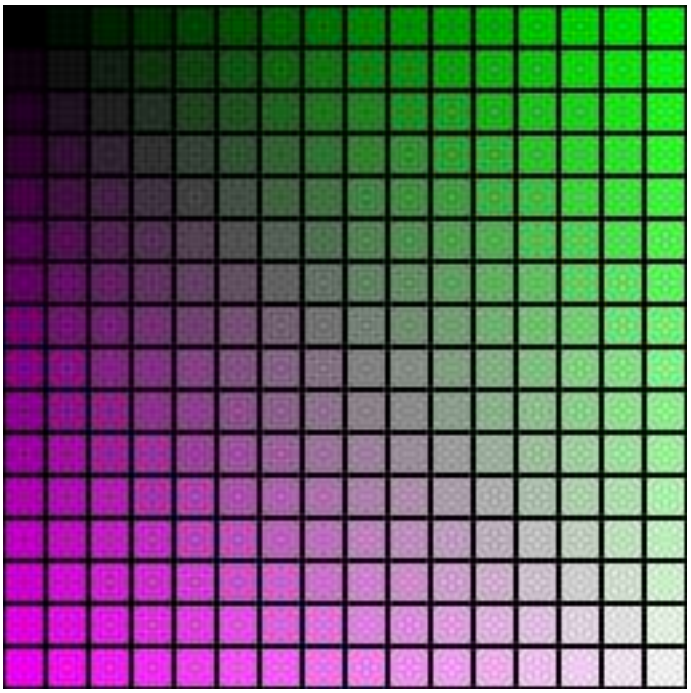

Illustration 9Output from colors2.py

The colors on the screen may look a little different than the colors in the book. You should have solid green in the upper right box, and violet in the lower left box, and varying shades of green and violet in between.

If you'd like to try other color variations, just change this line:

```
color = (y, x, y)
```

to this:

```
color = (y, y, x)
```

or this:

```
color = (255, x, y)
```

and press F5 to see what happens!

How does it work?

The line

```
for y in range(0, 256, 16):
```

causes the name **y** to have the values from 0 up to but not including 256, going up by 16. Thus **y** will equal 0, 16, 36, etc. when the program runs.

Inside of the first `for` statement is another `for` statement:

```
    for x in range(0, 256, 16):
```

This causes values of **x** to be 0, 16, 32, etc., in the same manner as the **y** values.

Inside of both `for` statements are the statements to calculate a color and draw a box based on the current **x** and **y** values. When the `for` statements are run, **x** and **y** will have these values:

(0, 0), (0, 16), (0, 32), ... (0, 240)

(16, 0), (16, 16), (16, 32), ... (16, 240)

(32, 0), (32, 16), (32, 32), ... (32, 240)

...

(240, 0), (240, 16), (240, 32), ... (240, 240)

Thus boxes will be drawn at each of those locations, and colors will be calculated using each of those values.

Looking a little closer at this statement

```
        color = (y, x, y)
```

we see that for the top row of boxes, y=0 and x goes from 0 to 240, so the upper right box will have a fill color of (0, 240, 0). Since the color numbers are (red, green, blue), color (0, 240, 0) is strong green with no red or blue. That's why the box in the upper right corner was green.

Drawing Images

You can use the DrawingWindow class to display images. By images, I mean files that contain picture data. DrawingWindow can display picture files whose names end in **`.gif`** or .bmp. Files that end in **`.gif`** ('GIF files") are typically used for putting pictures in web pages, and files that end in **`.bmp`** ("Bitmap files") are in the format that is created by the Paint program that comes with Microsoft Windows. Many other programs can write pictures in **`.bmp`** format as well.

In order to draw an image on a DrawingWindow, you need to know:

1. The full name of the image file
2. The location in the window where you wish to place the upper-left corner of the image

The `cpif` software includes some sample image files for you to use. One of those sample images is **`PythonPowered.gif`**, which I downloaded from the web site at:

http://starship.python.net/~just/pythonpowered/

This image is free for anyone to use who wants to promote Python. Always check the web site license conditions before downloading and using someone else's pictures in your programs.

To get the full name of the image file, use the `cpif.images()` function to get the name of the sample images directory, and the `os.path.join()` function to connect the name of the directory with the name of the image, like this:

```
>>> import os
>>> import cpif
>>> image_file = os.path.join(cpif.images(), 'PythonPowered.gif')
```

To show the image in a DrawingWindow, use the `drawImage()` method, like this:

```
>>> from cpif.graphics import DrawingWindow
>>> dw = DrawingWindow(200, 200)
>>> image_id = dw.drawImage(0, 0, image_file)
```

This displays an image in the upper-left corner of the window that looks like this:

Centering an Image

It looks a little strange to have the image in the upper-left corner of the window. Let's center the image.

To center an image (or any other object) in a window you need to know the size of the window and the size of the image. DrawingWindow can tell us those things:

```
>>> window_width, window_height = dw.getSize()
>>> print window_width, window_height
200 200
>>> image_width, image_height = dw.getImageSize(image_id)
>>> print image_width, image_height
110 44
```

To center the image we need to move the upper-left corner of the image to a place where the center of the image is also the center of the window. How do we calculate the coordinates of this spot? Let's think about it. If the image were really small, say zero pixels by zero pixels, then that spot would be in the center of the window, which is at `(window_width/2, window_height/2)`. To correct for the size of the image, we need to move the corner of the image half the width of the image to the left, and half of the height of the image up. Moving left means subtracting, and moving up also means subtracting. So the formula to center an image is:

```
>>> x = (window_width/2) - (image_width/2)
>>> y = (window_height/2) - (image_height/2)
```

Where `(x, y)` are the coordinates at which we want to put the upper left corner of the image.

Try putting the image at that location:

```
>>> dw.moveTo(image_id, x, y)
```

You will see the image move to the center of the window.

Quiz 15

1. Create a new Python program that draws a house. Use the `polygon()` and `box()` methods of DrawingWindow to draw at least the roof and main wall of the house. Save the program in a new file called house.py.

   Hint: try `help(DrawingWindow.polygon)` and `help(DrawingWindow.box)` to get more information about those methods. You might consider using graph paper to plan out the coordinates of the corners of your boxes and polygons.

2. Write a program called **`randcirc.py`** that draws 500 random circles on a 640 by 480 pixel window. Make each circle a random size between 2 and 10, filled with a random color chosen from a list of `['red', 'blue', 'green']`.

   To generate random numbers, use the `random` module. Use `random.choice(list)` to pick a color from a list. Use `random.randrange(start, stop)` to pick a number from `start` up to but not including `stop`.