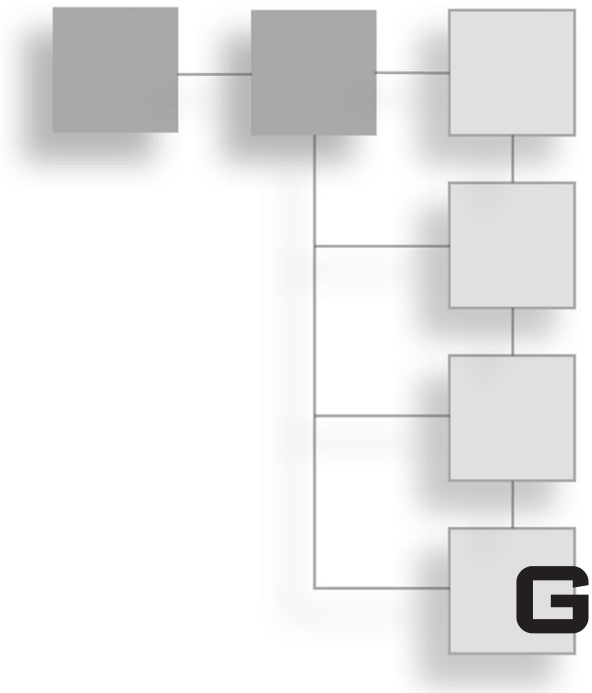
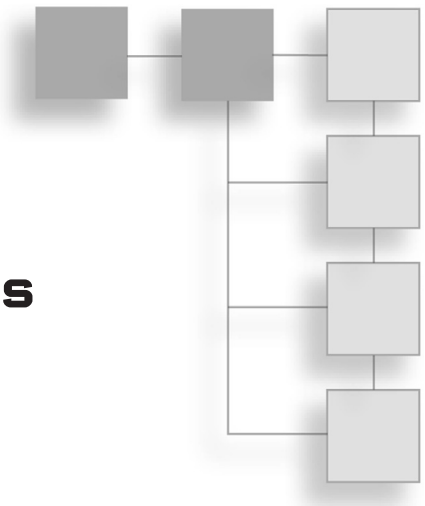


PART II

GETTING GRAPHICAL

CHAPTER 5



BEGINNING GRAPHICS

Hey, welcome back! Today, we're gonna start using graphics in our program. This chapter will be a huge jump for you; it teaches you how to initialize the graphical window and how to perform image loads. It also shows you how to display and move your images on the screen.

Anyway, get ready. This chapter is simple, but it's packed with some serious stuff.

Creating the Graphics Window

A graphics window is a little bit different from the text windows we have been using thus far. Unlike the programs we have been running to this point, which could only display text, graphical windows can also display graphics, such as images and pictures. They can also change colors of text.

Every BlitzPlus graphical program contains a line of code that initializes the window. This process basically sets up the window for later use. To set up a graphical window, call the function `Graphics`. `Graphics` is declared as follows:

```
Graphics width, height, color depth, [mode]
```

Table 5.1 details each parameter.

Table 5.1 Graphics Parameters

Parameter	Meaning
`width`	The width of the window in pixels
`height`	The height of the window in pixels
`color depth`	The colors per pixel (in bits)
`[mode]`	The mode of the window: 0 = auto, 1 = full-screen mode, 2 = windowed mode, 3 = scaled-window mode

What Is Initialization?

I use the term *initialization* a lot in this chapter, and you might wonder what it means. To initialize a window is to set the window up, so, when you initialize the graphics in BlitzPlus, you are setting it up. After initialization, you will be able to use graphics in the program.

Width and Height

Let's discuss each parameter in depth. Take a look at `width` and `height`—they affect your program in a huge way, but only a few modes are commonly used. These modes are shown in the following list. You might be wondering why we only use these modes, and there certainly is a reason.

- 640×480
- 800×600
- 1024×768
- 1280×1024
- 1600×1200

If you were to take a ruler to your computer monitor and measure the height and width, you would always come out with a bigger width than height. But the cool part is, the numbers you come up with are always proportional to one another. For example, my monitor is 14.66 inches wide and 11 inches tall. If you divide 14.66 by 11, you get 1.33. This means that my computer monitor's width is 1.33 times its height. This proportion works for all monitors and most televisions as well. Try it out!

Because the monitor's width is longer than its height, all of the pixel values on the monitor must change. If you were to draw a box that was an exact square, it would end up looking like a rectangle on the monitor (its width would be longer than its height). To combat

this problem, resolutions make the height pixels larger than the width pixels. The pixels are stretched out a bit, and the square actually looks like a square. Refer to Figure 5.1 to see the monitor's proportion.

Color Depth

note

Take note that setting the color depth only makes a difference in full-screen mode. In windowed mode, the color depth of your game is limited to the color depth of the player's desktop; in full-screen mode, the color depth can be set to any one of the color depths from Table 5.2. To see your desktop's color depth, right-click on your desktop and select Properties. Then find the Settings tab. Your color depth is under Color Quality.

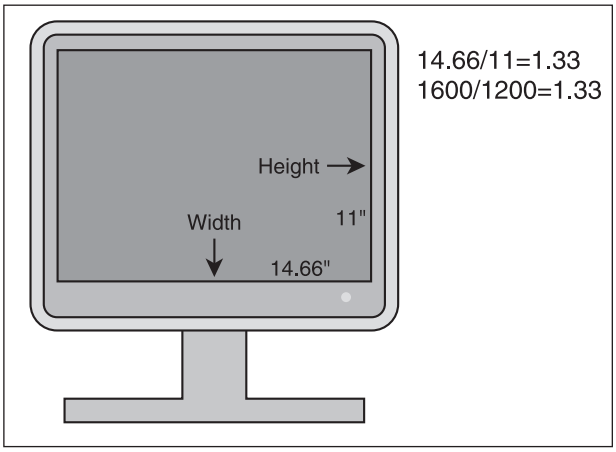


Figure 5.1 The monitor's proportion.

The next variable is color depth. The color depth is actually the number of colors that each pixel can be, and is numbered in bits. See Table 5.2 for the common color depths and their respective color counts.

Table 5.2 Color Depth

Color Depth (Bits)	Colors
8	256
16	65536
24	16,777,216
32	4,294,967,296

note

To determine how many colors each color depth provides, simply raise 2 to the power of the color depth. For example, if you want to find out how many colors a color depth of 8 gives, multiply 2 by itself 8 times ($2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$) or find 2 to the 8th power (2^8).

note

Although these are the only color depths used commonly today, other depths have been used in the past. For example, some very old games might have run in a color depth mode of 1, which provides only two colors—black and white.

caution

Make sure you know which bit depth you should be using before you select it. If you use a color depth of 8, for example, but the colors in your game need at least a color depth of 16, the colors in your game won't show up.

If you aren't quite sure which color depth to select, BlitzPlus can automatically select the best color depth for you. To have Blitz do this, just omit the color depth or set it to 0. Basically, what this means is, if you know what color depth you need, pick it yourself; if not, let BlitzPlus pick for you.

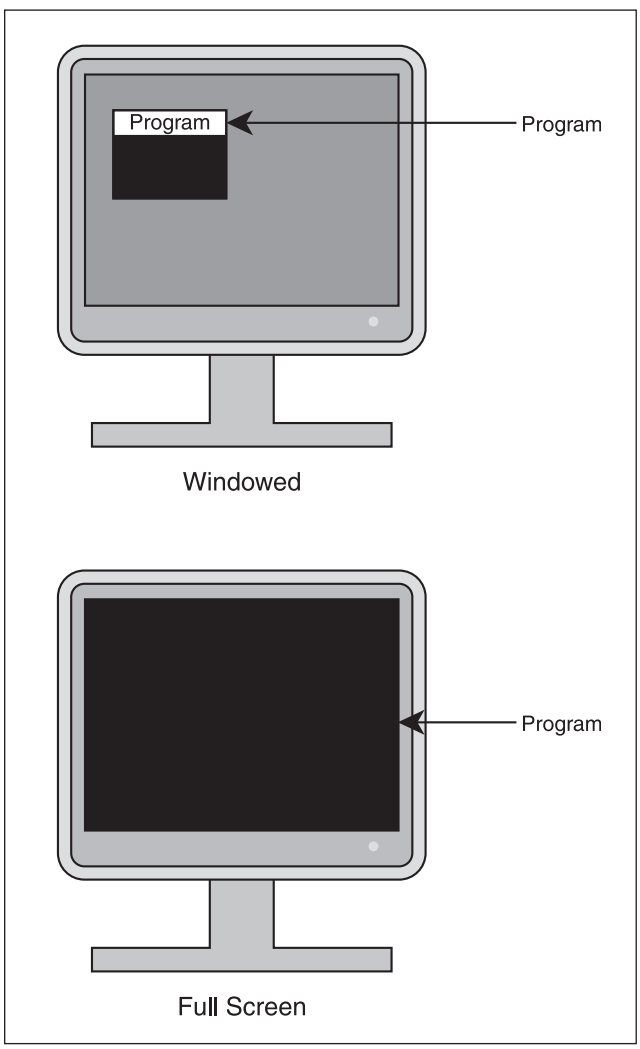


Figure 5.2 Full-screen and windowed modes.

[Mode]

The final variable in the `Graphics` function is the `[mode]` variable. `[Mode]` can be one of four choices—0, 1, 2, or 3. The `[mode]` variable determines how the program window behaves.

0 is `[mode]`'s default value; if you leave `[mode]` blank, it is automatically set to 0. When your program runs in auto mode, it runs windowed in debug mode and full screen otherwise. Figure 5.2 shows the difference in full-screen and windowed modes.

What Is Debug Mode?

I refer to debug mode a lot, and you might want to know what it means. When writing a game, you often come across hidden bugs that are extremely hard to find. Debugging allows you to step through a program line-by-line to discover where your program goes wrong. Debugging offers another reason for using functions—discovering bugs in a program where most of the code is located in functions separate from the main code is much easier than finding bugs in a program where all the code is thrown together in the main function.

When you are planning on debugging a game, you work in debug mode. This allows you to see the line you are debugging and find out what value each variable contains. When you have finished your game, you turn debug off and distribute the actual game. To turn debug mode on and off, check or uncheck Program>Debug Enabled. See Figure 5.3 to see how to enable Debug Mode.

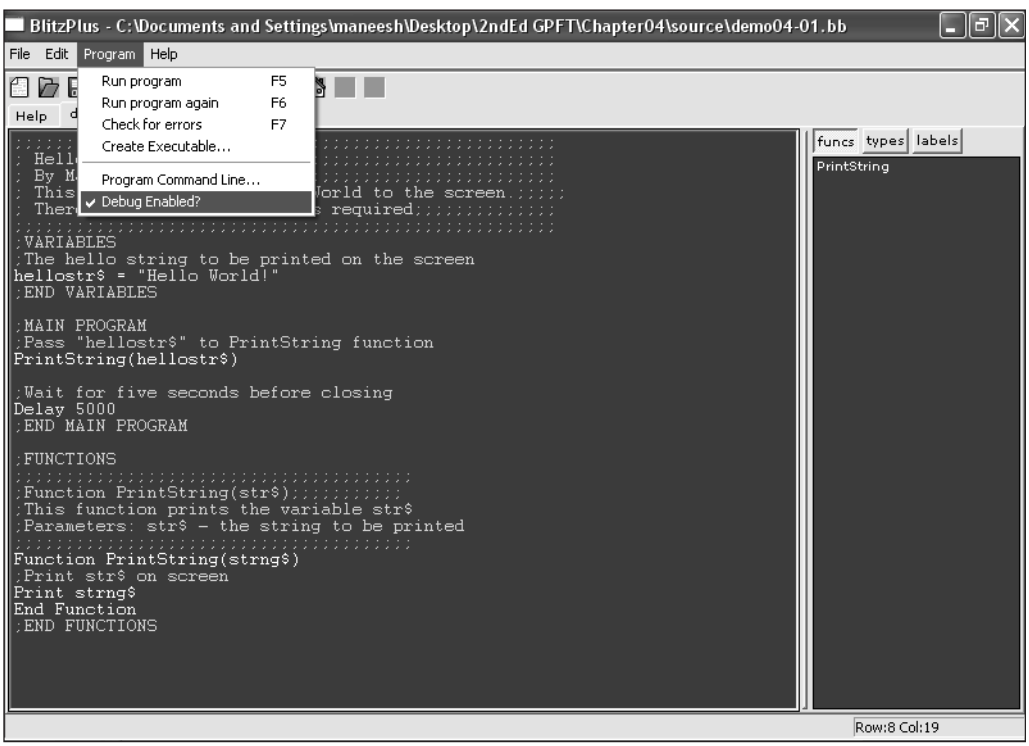


Figure 5.3 Debug mode.

Table 5.3 details each of `[mode]`'s possible values. Selecting 1 for the `[mode]` variable causes your game to run full screen. A game in full-screen mode takes up the entire screen; there are no other windows or programs on the screen. Of course, the other programs are

running, they are just hidden under the game. Full-screen mode tends to make the game run faster, but it takes over most of the player's computer screen. Figure 5.4 is a screenshot of a full-screen game.

Table 5.3 [mode]'s Values

Value	Mode Name	Meaning
0	auto	Runs in windowed mode when in debug mode and full screen when not.
1	full screen	Game takes up the full screen—no other programs can be seen.
2	windowed	Game runs as a regular windows program.
3	scaled windowed	Game runs as a regular windows program but also allows resizing, minimizing, and so on.

Figure 5.4 KONG in full-screen mode.

Setting `[mode]` to 2 forces your game to run like a normal windows program. This means that your program has a toolbar and can be moved around just like a normal program, as in Figure 5.5. However, you cannot resize your window.

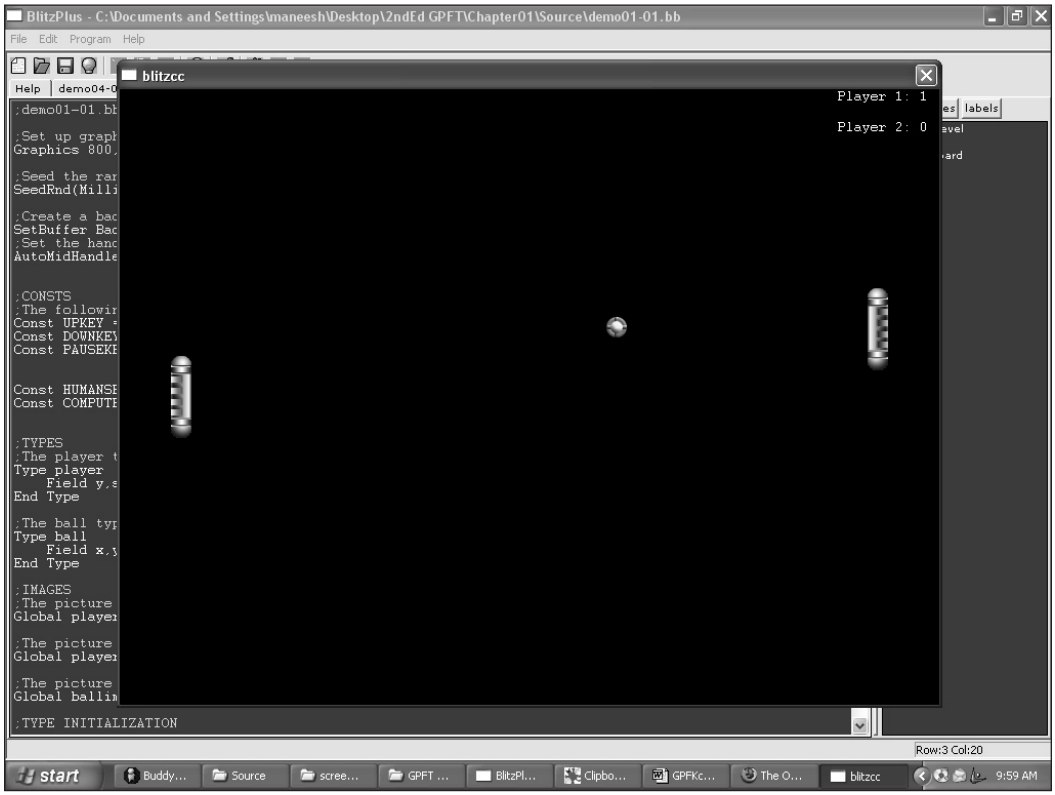


Figure 5.5 KONG in windowed mode.

If `[mode]` is set to 3, your program acts just like it would if it were set to 2, but you are able to resize, minimize, and maximize the window to your liking. However, this advantage comes at a price—a drastic decrease in speed often occurs as a result of scaled window mode. See Figure 5.6 for an example of what a scaled window could look like.

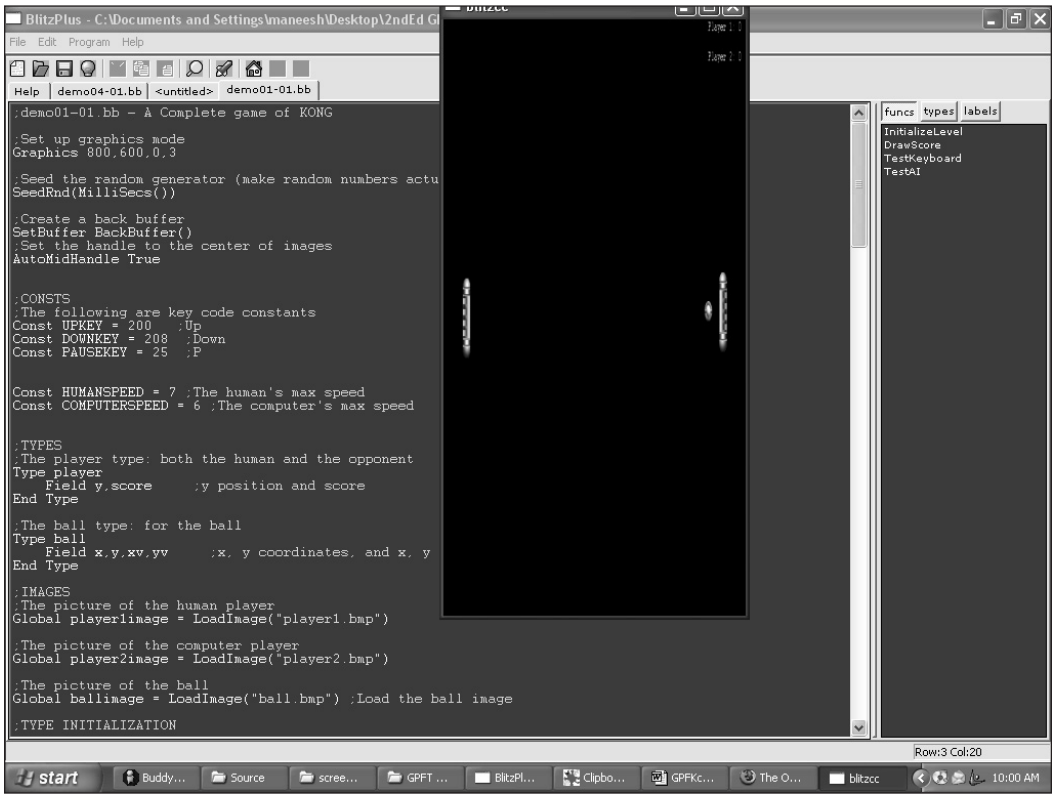


Figure 5.6 KONG in scaled windowed mode.

Images

Whew, that was one big graphics call! Let's get into more specialized graphics stuff. This section explains how to load an image, how to draw it onscreen, and the like. Are you ready?

LoadImage

The first call we will be using is `LoadImage`. This function loads the image of your choice into your program's memory. You must load an image before you can display it or manipulate it in your program. `LoadImage` is defined as this:

```
LoadImage(filename$)
```

Table 5.4 examines each parameter. To load an image, just substitute the file name of the image for `filename$` (make sure the file name is in quotes), and assign it to a variable, like this:

```
Global playerimage = LoadImage("playerimage.bmp")
```

Table 5.4 LoadImage's Parameter

Parameter	Description
`filename$`	The path of the image

Why .bmp?

Unfortunately, the demo version of BlitzPlus only allows you to use bitmap files for image processing. This means that you can't just open some image off your computer and use it, unless it has a .bmp extension. However, there is a simple way around this problem. Just take the jpeg, gif, or png file, and open it in Microsoft Paint or in Paint Shop Pro (which is included on the CD). Then choose Save As and convert the image to a bitmap!

note

Check out what I set the file name variable to. Making the file name just the name of the file (without adding any path info) works only if the image is in the same directory as the game. If not, you might need to include your drive information. It might look something like this:

```
Playerimage = LoadImage("c:\windows\desktop\playerimage.bmp")
```

Even so, I suggest you keep all of your images in the same folder as the game because if you ever decide to distribute your game, the game won't work on other computers unless the user puts the images in the exact same folder as yours.

I usually name my image variables in such a way that I can easily see that they are images. This means I begin my image names with its actual job (*player* in playerimage.bmp) and suffix it with image.

The name that you assign to the loaded image is called a *handle*. Basically, a handle is just an identifier that refers to an image in memory, like in Figure 5.7.

`LoadImage()`, by default, searches directly in the same folder as the location of the BlitzBasic file. If you want to load an image from another directory, you must provide the full path to the image.

Okay, now that we've got this `LoadImage` stuff down, its time to actually draw it!

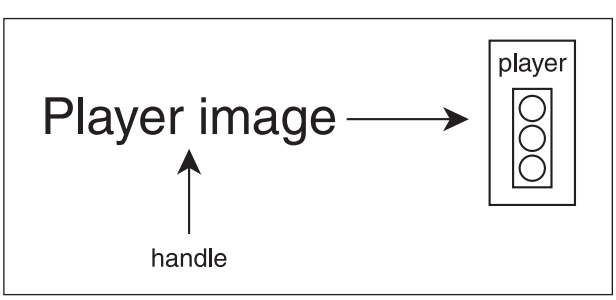


Figure 5.7 A handle to an image in memory.

DrawImage

It is pretty easy to guess what this function does: it draws images! Table 5.5 examines each parameter. Let's start with the declaration.

```
DrawImage handle,x,y,[frame]
```

`DrawImage` has a couple of parameters, so let's move on to a discussion of the handle variables.

Table 5.5 DrawImage's Parameters

Name	Description
`handle`	The variable that holds the image
`x`	The drawn image's x coordinate
`y`	The drawn image's y coordinate
`[frame]`	Advanced, leave as 0

Handle

This is a pretty easy-to-understand parameter. Remember when you loaded an image like this?

```
playerimage = LoadImage("player.bmp")
```

Well, the handle is `playerimage`. So, when you're sending parameters to `DrawImage`, use the same image handle that you loaded earlier as the `DrawImage` handle parameter.

X and Y

The x and y parameters work just like most x and y coordinates in BlitzPlus. Using `DrawImage`, your selected image is drawn at the x and y coordinates, as shown in Figure 5.8. Its top-left corner is located at the given x and y values. However, there is a way to center the image so that the image's center is located at x,y.

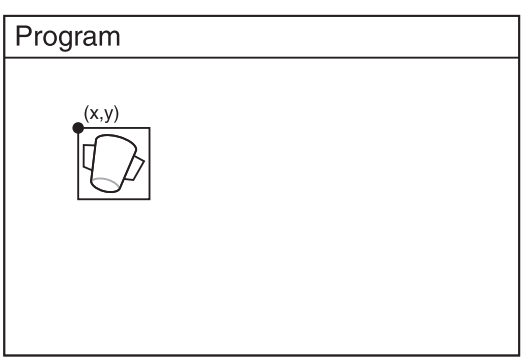


Figure 5.8 The image at x,y.

Very often, you will want to center the image. This is most useful when rotating images because rotating images around the top-left corner looks bad (not to mention trippy) due to the fact that you would expect images to rotate around their centers. Check out demo05-01.bb to see how an image looks when it is rotated around the top-left corner.

Although actual rotation is a more advanced technique and is explained in a later chapter, I am using it to illustrate the use of placing the x and y values in the center of the image. The actual function is called `AutoMidHandle` and is declared like this:

```
AutoMidHandle true|false
```

note

> What does "|" mean? | means or. When I say `AutoMidHandle true|false`, I mean `AutoMidHandle` can use either `true` *or* `false`.

To use this function and place the x and y values in the center of the image, call `AutoMidHandle` with the parameter `true`, like this:

```
AutoMidHandle true
```

Easy, huh? And to set the x and y location back to the top left, just call `AutoMidHandle`, like this:

```
AutoMidHandle false
```

It is a good idea to use `AutoMidHandle` because it helps you understand exactly where the images are located. Because your access point is directly in the center of the image, you won't need to worry about the image's width and height as much as if the access point was in the top left.

Table 5.6 details the parameters, and Figure 5.9 shows how demo05-02.bb, which uses `AutoMidHandle true`, works. Look at the difference in Figures 5.8 and 5.9. In Figure 5.8, you can see how the x and y coordinates are located at the top-left corner of the image. In Figure 5.9, the x and y coordinates are in the center of the image. Try running demo05-02.bb and watch how it rotates from the center instead of from the left corner, as in demo05-01.bb.

Table 5.6 AutoMidHandle's Parameters

Name	Description
`true`	Places the x and y coordinates in the center of the image.
`false`	Places the coordinates at the top left of the image.

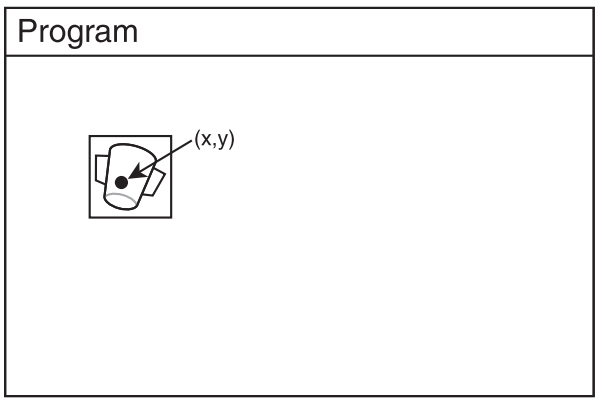


Figure 5.9 The image at x,y with AutoMidHandle set to true.

Make absolutely sure that you place `AutoMidHandle True` before you load the image, otherwise the function won't work.

By the way, there is another function called `MidHandle` that is a lot like `AutoMidHandle`, except that it doesn't set the x and y coordinates to the center of all of the images. It only sets the x and y coordinates to the center of an image you choose. It is declared like this:

```
MidHandle image
```

The image handle you pass it will be reset to the center of the image. Use this if you only want one image handle to be in the center of the image, rather than all of them.

[Frame]

Okay, this command is very advanced. `[Frame]` allows you to draw images that are animated. It is too advanced right now, but we will be going over using animated images very soon!

CreateImage

This function is pretty cool. It allows you to create an image that looks like whatever you want, and use it just like a loaded image. For example, say you wanted to create an image with 100 dots on it. First, call the `CreateImage` function, which has a declaration like this:

```
CreateImage(width, height, [frames])
```

`Width` and `height` explain how big the image is; `[frame]` is used with animated images and should be set to 0 for now. To create the image, call `CreateImage` like this:

```
dotfieldimage = CreateImage(100,100,0)
```

Okay, you now have the handle to the image. Now, you have to populate the field with dots. The following is the full source for the program, which can also be found on the CD as demo05-03.bb:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; demo05-03.bb;;;;;;;;;;;;;;;;;;;;
; By Maneesh Sethi;;;;;;;;;;;;;;;;
; Creates an image and displays it!
; No input parameters required;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;INITIALIZATION

;Set up the graphics
Graphics 800,600


;Seed the Random Generator
SeedRnd MilliSecs()

;CONSTANTS
;The length of each block
Const LENGTH = 100

;The height of each block
Const HEIGHT = 100

;The amount of dots in each block
Const DOTS = 100
;END CONSTANTS



;IMAGES
;Create the dotfield image
dotfieldimage = CreateImage(LENGTH,HEIGHT)
;END IMAGES

;For each dot, draw a random dot at a random location
For loop = 0 To DOTS ;For every star
;draw only on created image
SetBuffer ImageBuffer(dotfieldimage)

;Plot the dot
Plot Rnd(LENGTH), Rnd(HEIGHT)

Next
```

```
;Set buffer back to normal
SetBuffer BackBuffer()
;END INITIALIZATION

;MAIN LOOP

;Tile the image until the user quits (presses ESC)

Cls
TileImage dotfieldimage
Flip

WaitKey


;END MAIN LOOP
```

Figure 5.10 shows the dot field.

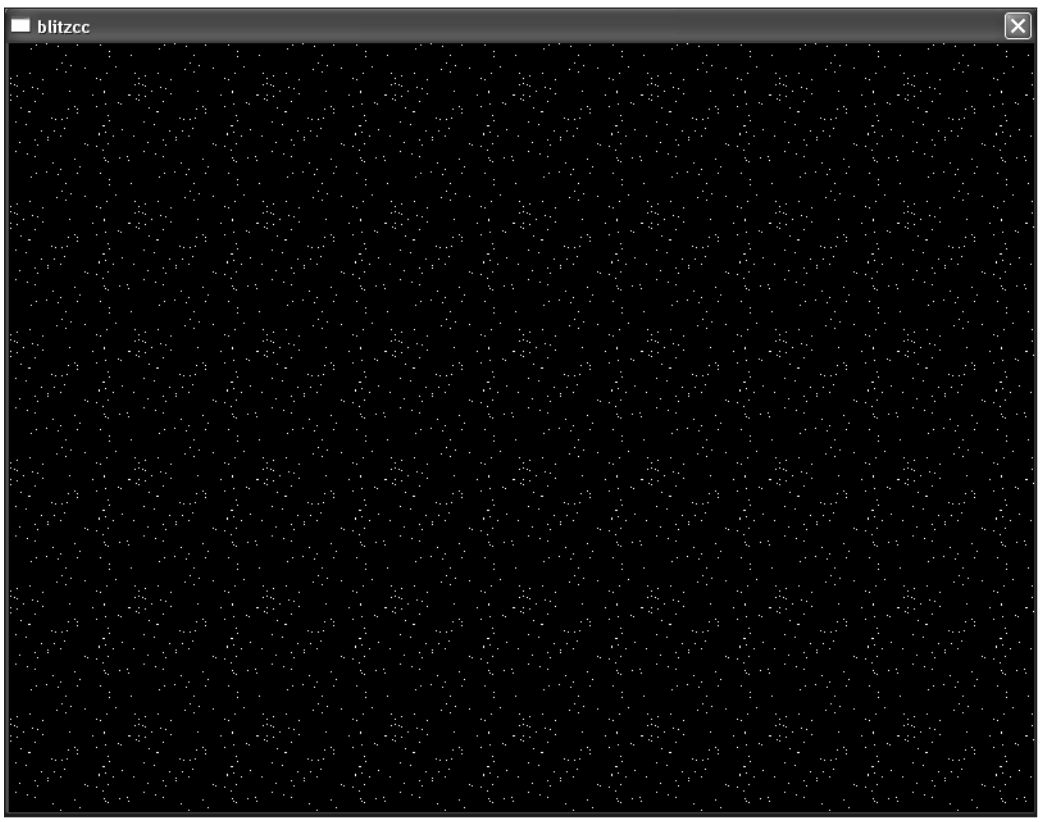


Figure 5.10 The dot field.

There are a few new functions introduced in this program, and I'll go over them now. The first new function is `ImageBuffer()`.

`ImageBuffer()` acts a lot like `BackBuffer()`. You will learn how `BackBuffer()` allows you to draw on the back buffer instead of the front buffer, so that you can flip the buffers and create animation. Well, `ImageBuffer()` is just like `BackBuffer()`, but instead of drawing on a buffer, you are drawing on an image. `ImageBuffer()` is declared as this:

```
ImageBuffer(handle, [frame])
```

where `handle` is the handle of the selected image and `[frame]` is the chosen frame to draw on (leave as 0 for now). Drawing on an image buffer is a lot like Figure 5.11. As you can see, calling `SetBuffer ImageBuffer(dotfieldimage)` allows you to extract the image from the program and only draw on that. Then, when you finish, you call the `SetBuffer` function again. In this program, I used `SetBuffer FrontBuffer()`, only because there is no page flipping; however, in most games use `SetBuffer BackBuffer()`. Table 5.7 details `ImageBuffer's` parameters.

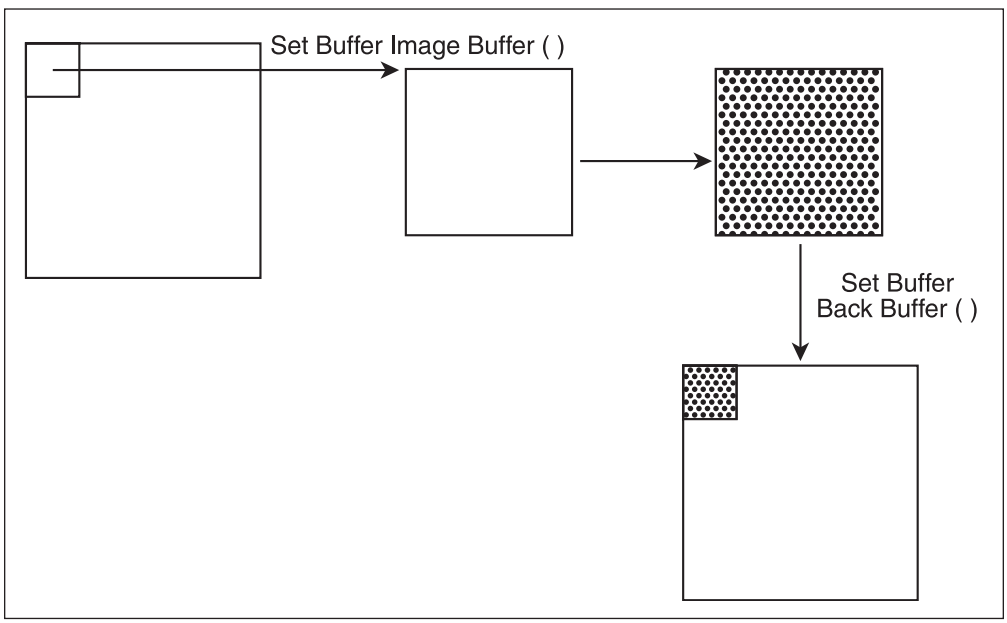


Figure 5.11 `SetBuffer ImageBuffer()`.

Table 5.7 ImageBuffer's Parameters

Name	Description
`handle`	The handle of the selected image
`[frame]`	The chosen frame to draw on; leave as 0 for now

The next function introduced is `TileImage()`. `TileImage()` is declared like this:

```
TileImage handle, [x], [y], [frame]
```

`TileImage` works like this: it takes an image you give it and it places copies of it all across the programming board. Think of it like a chess board—there are only two images on a chessboard, black and white. But these two images are tiled over and over until the entire board is filled with black and white tiles. See Figure 5.12 for a visual aid to tiling, and Table 5.8 for a list of each parameter.

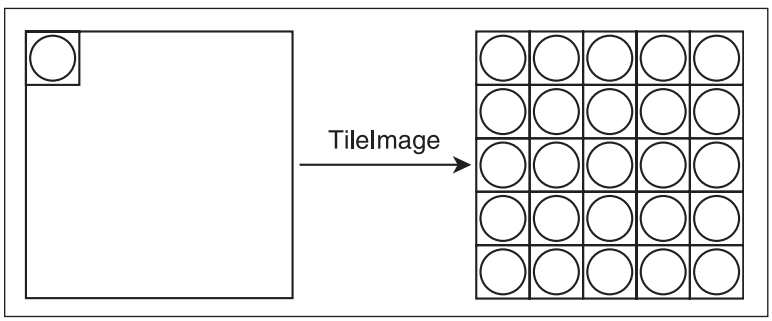


Figure 5.12 The `TileImage` function.

Table 5.8 TileImage's Parameters

Name	Description
`handle`	The image you wish to tile
`[x]`	The starting x coordinate of the tiled image; 0 by default
`[y]`	The starting y coordinate of the tiled image; 0 by default
`[frame]`	The chosen frame to tile; 0 by default

To tile an image, call `TileImage` with the handle of an image you wish to tile. BlitzPlus will take care of the rest. By the way, in later chapters, you will learn how to move the tiled field up and down to simulate movement.

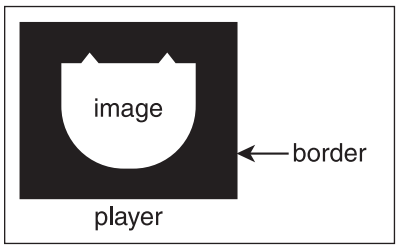
The last part of the program calls the function `WaitKey`. This function simply pauses the program until a key is pressed.

MaskImage

All right, the next function I want to go over is called `MaskImage()`. `MaskImage()` is defined like this.

```
MaskImage handle, red, green, blue
```

`MaskImage()` allows you to define a color of your image as transparent. What does that mean? Let me show you.



When you draw or create an image, you always have a border that is not part of the image. See Figure 5.13 for a description of the border. As you can see, the outer part of the image is not used, and should be discarded. You don't want the border to be shown, like in Figure 5.14, do you?

Figure 5.13 An unmasked image.

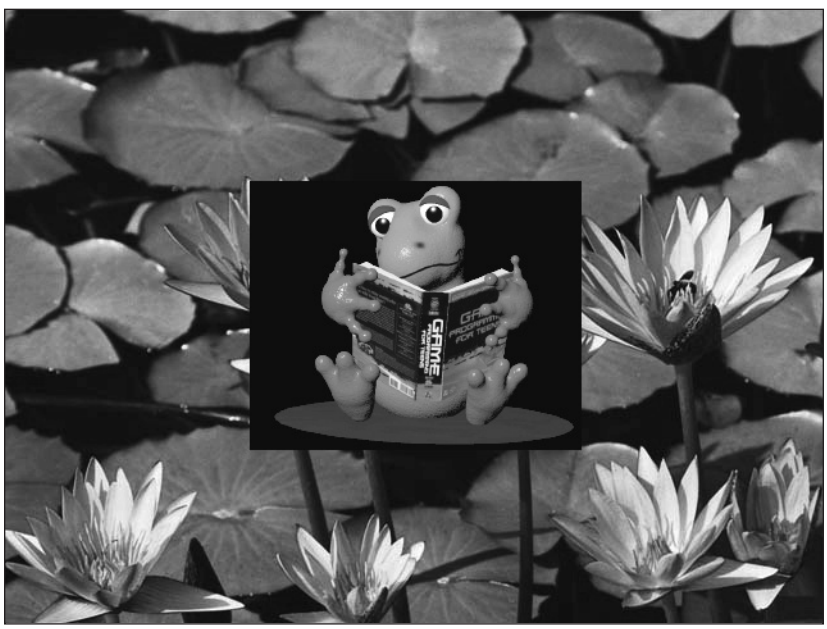


Figure 5.14 A drawn image with a border.

note

Because black is automatically masked by default, the image in Figure 5.14 does not have a purely black border. I added a tiny amount of blue to the image so that the background wouldn't be masked. The RGB value of this image's background is 0,0,10.

Calling `MaskImage()` can get rid of that border for you. Table 5.9 explains each parameter. Because the RGB value of this background is 0,0,10, call the `MaskImage()` function with the correct parameters.

Table 5.9 MaskImage Parameters

Name	Description
`handle`	The image you wish to mask
`red`	The red value of the mask
`green`	The green value of the mask
`blue`	The blue value of the mask

The full program is detailed next:

```
;;;;;;;;;;;;;;;;;;;;;;
;demo05-05.bb
;By Maneesh Sethi
;Demonstrates the use of masking
;No Input Parameters required
;;;;;;;;;;;;;;;;;;;;;;
;Initialize graphics
Graphics 640,480


 ;Load Background
lilliesimage = LoadImage("lillies.bmp")
;Draw background
DrawImage lilliesimage,0,0


;Load the frog
frogimage = LoadImage("frog.bmp")
;Center the frog
MidHandle frogimage
;Mask the Frog Image
MaskImage frogimage,0,0,10
;Draw it in the center
DrawImage frogimage,320,240

Flip
;Wait for user to press a button
WaitKey
```

Figure 5.15 is a picture of this program. Beautiful, isn't it? It looks as if the frog is actually part of the image! On the CD, demo05-04.bb is a program without masking, and demo05-05.bb is the same program with masking.

Figure 5.15 An image drawn with a mask.

One thing to note: an RGB value of 0,0,0 is the default. 0,0,0 is the color of black. This means that if your image is drawn with a black border, it will automatically be masked. In other words, try to make all your images have a black background so you don't need to worry about masking images.

You might have noticed the command `Flip` at the end of the program. By default, BlitzPlus draws its information on the back buffer. By using `Flip`, you move the information from the buffer to the screen. We will learn more about this in later chapters.

Colors

Before I end this chapter, I want to teach you how to work with color. Of course, color is an integral part of any program. When using page flipping (which is explained in the next chapter), color takes on an even greater importance.

You need to know some functions before you move on to the next chapter. These functions are `Color`, `Cls`, and `ClsColor`. You also need to understand RGB values.

RGB

When working with color, you will often encounter RGB (red, green, blue) values. These numbers allow you to pick any one of 16 million different colors. That's a lot, huh?

Why 16 Million?

When you are using RGB values, you usually pick a number between 0 and 255 for each color. What does this have to do with the amount of colors? Well, if you multiply 256 by itself three times because there are three colors (256 × 256 × 256), you get 16.7 million. This means that you have all 16.7 million values to choose from.

When color is used in functions, there are usually three fields for you to enter your choices—red, green, and blue. For each field, you can pick a number between 0 and 255 (256 choices total). The higher the number, the more of that color there will be. For example, if you set the red value to 255 and the green and blue values to zero (255,0,0), you will have a perfectly red color. 0,0,0 is black, and 255,255,255 is white.

Now, you may be wondering how you are supposed to find the exact values for the color you want. Well, there are two ways. You can use guess and check (by putting in guesses for the red, green, and blue fields) or you can use a program, such as Microsoft Paint.

Open Microsoft Paint by going to Start Menu>All Programs>Accessories>Paint. See Figure 5.16 for a visual image of Microsoft Paint and how to open it (the background is Paint, the foreground is the Start menu [your menu will probably be a little different]). Nowchoose Colors>Edit Colors. A window will pop up. Click where it says Define Custom Colors. Figure 5.17 shows you the custom colors box.

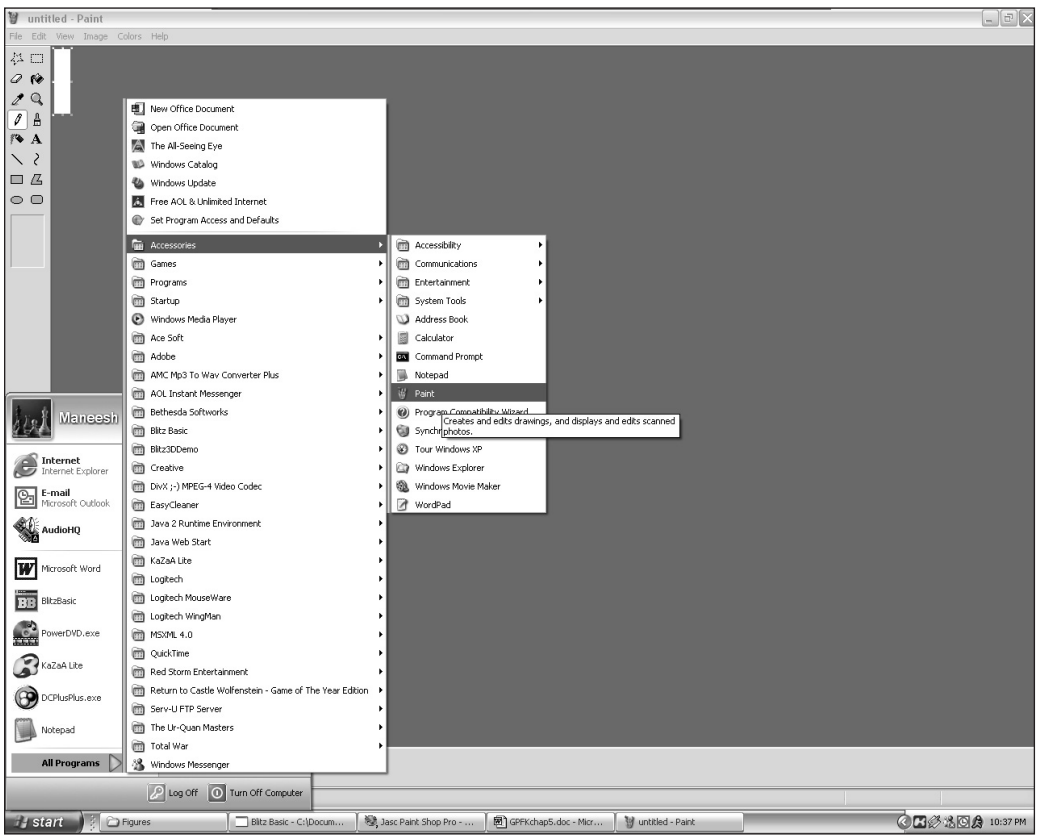


Figure 5.16 Opening Microsoft Paint.

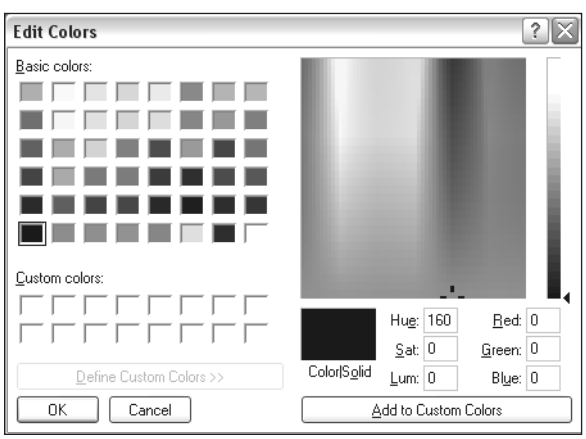


Figure 5.17 Defining custom colors.

Now choose your color, and it should tell you the RGB value on the bottom. If it doesn't work at first, move the scrollbar on the far right, and then proceed to pick your color.

That's pretty much all there is to RGB. You're ready to use color in your programs now.

Color

`Color` is kind of a fun function. It defines what the default color of the program is. When you draw something, be it lines, shapes, or text (not images), it will be drawn with the defined color.

What can you do with `Color`? If you want to make the text anything other than white, just use this. Or maybe you want to draw a green triangle. Just set the color to green and draw it! You can change the color at any time.

note

The default color of any BlitzPlus program (before you call `Color`) is white (RGB 255,255,255).

You can start with the function declaration.

```
Color red, green, blue
```

See Table 5.10 for the parameters. You will most likely just put in the red, green, and blue values to get your color.

Table 5.10 Color's Parameters

Name	Description
red	The color's red value
green	The color's green value
blue	The color's blue value

Now let's write a program that uses this function. This program will draw a bunch of ellipses with random sizes and colors.

```
;;;;;;;;;;;;;;;;;;;;;;
;demo05-06.bb
;By Maneesh Sethi
;Demonstrates the Color function, draws ellipses
;No Input Parameters required
;;;;;;;;;;;;;;;;;;;;;;
Graphics 800,600

;Seed random generator
SeedRnd (MilliSecs())


;Max width of ellipse
Const MAXWIDTH = 200
 ;Max Height of ellipse
```

```
Const MAXHEIGHT = 200

;Main Loop
While Not KeyDown(1)

;Clear the screen
Cls

;Set the color to a random value
Color Rand(0,255), Rand(0,255), Rand(0,255)


;Draw a random oval
Oval Rand(0,800),Rand(0,600),Rand(0,MAXWIDTH),Rand(0,MAXHEIGHT), Rand(0,1)

;Slow down!
Delay 50
Flip
Wend
```

Pretty cool, huh? Figure 5.18 shows a screenshot from the program. Let's look a little closer. The program first sets the graphics mode and seeds the random generator. Then it defines the maximum width and height of each ellipse. Feel free to change the values.

Next, the game enters the main loop. It first sets the color to a random value, using the line

```
Color Rand(0,255), Rand(0,255), Rand(0,255)
```

This allows the next line to draw an ellipse with the random color. The ellipse function (notice that it is actually called `Oval`—I just like the word ellipse) is defined like this:

```
Oval x,y,width,height[,solid]
```

Take a look at Table 5.11 for each parameter.

Table 5.11 Oval's Parameters

Parameter	Description
`x`	The x coordinate of the ellipse
`y`	The y coordinate of the ellipse
`width`	The width in pixels of the ellipse
`height`	The height in pixels of the ellipse
`[solid]`	Default value is 0; set to 1 if you prefer the ellipse to be filled. Otherwise, the inner region will be transparent

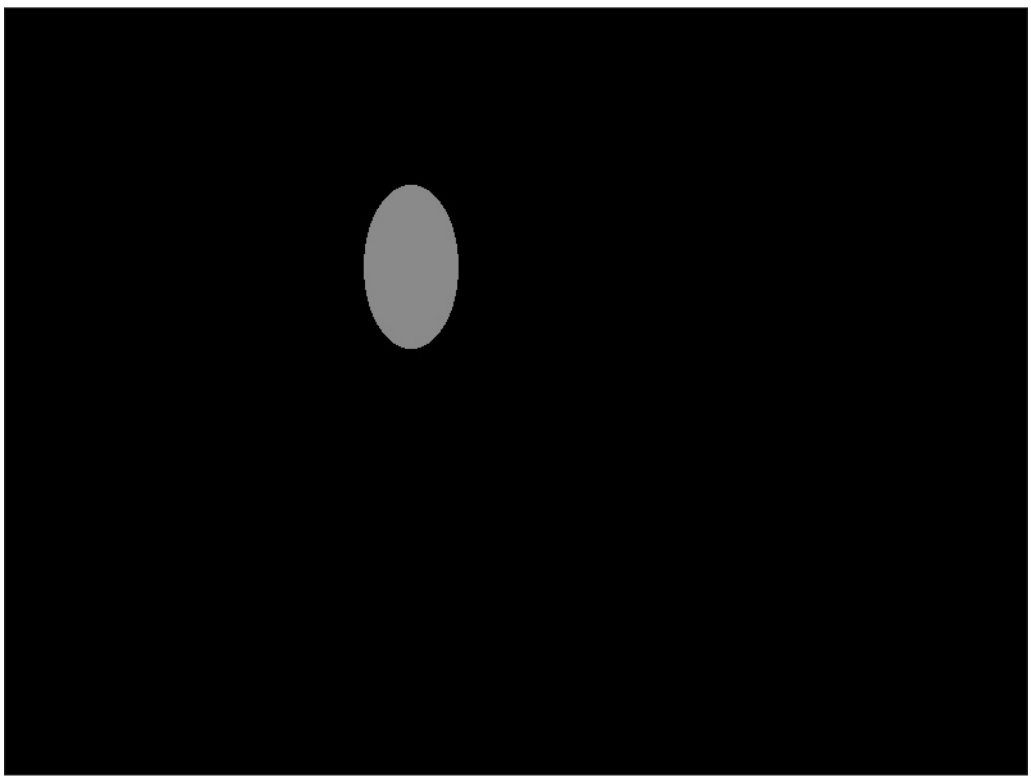


Figure 5.18 The demo05-06.bb program.

Well, that's pretty much it for the `Color` function. Next up—the `Cls` and the `ClsColor` functions.

Cls and ClsColor

We are almost done with this chapter! Before I send you packing, though, I want you to have a bit of basis for the next chapter.

The function `Cls`'s action is pretty simple. All it does is clear the screen. The next chapter goes over it in more depth. The `ClsColor` function works with `Cls` to allow you to change the background of your program.

`ClsColor` is defined like this:

```
ClsColor red,green,blue
```

See Table 5.12 for a description of each parameter.

Table 5.12 ClsColor's Parameters

Name	Description
`red`	The color's red value
`green`	The color's green value
`blue`	The color's blue value

`ClsColor`'s job is to change the background color. This means that you can leave the default black behind and make the background anything you want it to be. To use this function, call `ClsColor` with the red, green, and blue values you want, and then call Cls to actually clear the screen with the background color.

Let's try a program. Demo05-07.bb makes a bunch of colors appear on the screen (along with some advice you should follow). Try it out!

Summary

Okay, you now have a working knowledge of graphics in video games. In this chapter, we learned about a lot of functions: `Graphics`, `LoadImage()`, `DrawImage()`, `CreateImage()`, `ImageBuffer()`, and `MaskImage()`. Believe me, you will find many uses for all of these functions in your games.

This chapter studied the topics of:

- Creating a graphics window
- Loading, drawing, and using images
- Using colors

Next up, we learn about page flipping and basic input. The following chapter is important because you learn about basic animation.