Learn to Program 2D Games in

Publisher: IDigiCon Ltd
Ashfield House
Ashfield Road
Balby
Doncaster
South Yorkshire, DN4 8QD
United Kingdom
Tel: +44 (0) 1302 310800,
Fax: +44 (0) 1302 314001

Learn to Program 2D Games in Blitz Basic
Author: John P. Logsdon (AKA. "Krylar")
Edited By: Lorelei J. Logsdon (AKA. "Beulah")
Graphics and Cover Art: Troy Robinson (AKA. "Pongo")
BlitzPlay Library: John Arnold (AKA. "SurreaL")
ISBN: 1-894752-17-1

Acknowledgements

Oh boy, where to start. There are so many people that I would like to thank, but that would probably take up a book in and of itself…and I'm doubtful IDigiCon would agree to publish it. ☺

Well, first and foremost thanks to my wife and son, Lorelei (Beulah) and Jake (Wyke), for all the support through this. I know I've spent a lot more time on typing than hanging out with you guys! Thanks to my Mom for hooking me up with my first and second Commodore computers (C-64 rules!). Thanks to my Dad for getting me started in the IBM-Compatible world and buying me compilers.

Thanks to the communities at BlitzCoder (http://www.blitzcoder.com) and the folks in the #BlitzCoder and #BlitzBasic IRC chat rooms. You are always helpful, encouraging, and patient…not to mention a fun diversion from coding mayhem!

Thanks goes big-time to Troy Robinson (Pongo) for providing all the art used in the examples and the cool cover art. If it wasn't for Troy's great skill and willingness to help, you would have been subjected to my horrific attempts at art…and that would be bad.

Thanks to John Arnold (SurreaL) for all his help with network coding. Without his help, this book would have taken quite a bit longer to complete and would have turned my brain to mush.

George Bray and the good people at IDigiCon have been simply wonderful throughout this project. These folks are top-notch! I would highly recommend working with them if you ever get the chance…it's definitely a pleasure. Thanks!

And last, but certainly not least, to Mark Sibly for being an all around good guy and for creating the best programming language I've ever used!

PART 1: BLITZ BASIC BASICS

Chapter 1: Welcome to Blitz Basic

This book is designed to get you started programming in one of the coolest languages available today. Taking you from fundamental programming concepts to advanced techniques, ***Learn to Program 2D Games in Blitz Basic*** will have you designing and developing your own games in no time.

What is Blitz Basic and who is this Book for?

For years I had struggled in trying to learn the techniques that the professional game developers used in their creations. I searched the Internet and read numerous books, but while many of them certainly provided terrific information, most were far over my head. Slowly, through much persistence, I began to understand a lot of what went into game development from a developer's standpoint.

I've also had the very fortunate experience of being around some of the best and brightest developers in the Game Industry, by having worked in the capacity of Producer and Executive Producer at various online game companies.

So with a ton of theory in my pocket, I started using my C programming skills to get my games underway. Then the dreaded DirectX interface got in the way. It's not that DirectX is super-complicated or anything, but when you're developing as a hobby you don't want to spend months learning how to use a tool that will only help you get to the first ring of development. That's where Blitz Basic came into the picture.

Blitz Basic was developed with the intent of allowing both beginning and advanced game developers to get their creations going without the need to learn or use a ton of low-level coding techniques. Blitz Basic uses one of the simplest languages as its base, BASIC. But don't let that fool you! Blitz Basic is not to be compared with BASIC in any way other than the choice of command syntax used. Where BASIC is an interpreted language (meaning that as the program runs, the computer translates each line into machine language before executing it), Blitz Basic compiles the code directly to machine language before executing any lines. This means that a program created with Blitz Basic will run without unnecessary steps that can slow it down.

Something equally important is that Blitz Basic is a REAL programming language. I have seen a number of products that are known as "Click and Play" game development systems, but Blitz requires that you use your imagination and coding-prowess to make your dreams into reality on the

computer. Coding-prowess is what I'll be focusing on in this book, although I will touch on imagination and game-play as well.

If you've never programmed before, you've come to the right place. This book starts with the fundamentals of programming while integrating the Blitz Basic commands needed to create your future games. You will be guided into stronger elements that will all be used in examples to help you gain full understanding of needed topics.

Why Learn Blitz Basic?

There are many languages out there that you could choose from, so why pick Blitz Basic? The simple answer is that Blitz will get you developing your game quickly. But it's also easier to learn than most languages; you don't need to learn the underlying Microsoft DirectX components, and you don't have to code the majority of image processing, collision, input, multi-player, or sound routines that you would normally have to.

If you're a seasoned game developer, Blitz Basic will allow you to prototype games quickly and easily without drastic speed loss of a click-n-play type system.

Finally, because Blitz Basic is a game programming language. C and C++ are used in a lot of game development projects, but they were not designed solely for programming games. Blitz was designed specifically with game development in mind. Therefore, when you start out with Blitz you are in a language that supports your goal of game development.

What Will I Need to Run Blitz Basic?

In order to run the Blitz Basic Integrated Development Environment (IDE), you'll need to have a system running Microsoft Windows and have Microsoft DirectX version 7.0 or greater installed.

This book is based on the commercial version of Blitz2D or Blitz3D. Many of the example programs will not work with the demo version of Blitz Basic. Also, make sure you have the latest version!

The Major Sections of this Book

In order to cover most needs while trying to maintain a non-exponential learning curve, I have broken this book up into sections.

The first section, "Blitz Basic Basics" is focused on the fundamentals of programming and the use of Blitz Basic. Here is where you will learn how to create simple applications that will help hone your development skills.

Section two, "BB Game Tools," is where we'll start putting images on the screens and moving them around. Using knowledge gained in section one, we will also work on animation, collisions, and timing functions.

"Advanced Topics" will be the focus of section three. That's where we'll get into a few tricks that will help build your programming expertise.

Conventions Used in this Book

Up until now, you've seen me using the full title "Blitz Basic" a lot. There are two accepted abbreviations in the Blitz Basic community that I'm aware of and they make for easier reading. So, instead of the full title, you'll often see me refer to Blitz Basic as either "Blitz" or simply "BB."

Throughout this text you will see boxes that are filled with bold text. These are "code boxes" because the text inside is actual Blitz code. Here is an example:

```
Graphics 640,480
```

You will also notice the following special characters on some lines in the code:

```
↲ and →
```

Such as:

```
If Ship.Shields < 100 and Ship.Armor > 100 and ↲
   → Ship.RepairAbility < 10

  Delete Ship
End If
```

The "↲" symbol means that the line is continued on the next line. This is so I can fit code in properly for the book. In the actual Blitz Basic development environment you will need to type the line as one full line because Blitz will not allow multiple line entries. Note that the next line will include the "→" symbol to further denote that the line is meant to be entered in as part of the previous line.

What's on the CD?

The CD contains all of the source code in these chapters, along with their respective sounds and images. Also the full source code and images to the Network Space Game demo. There are a number of other source code files, libraries, demos, etc. on this disk as well!

Chapter 2: Fundamentals of Programming

What is a Program?

A program is simply a set of instructions that the computer executes in some sequence. There are many types of programs that you are already familiar with, including Netscape, Microsoft Windows, America Online, and so on.

In order to create these programs, teams of developers (or programmers) write thousands of lines of code using languages such as C, C++, Visual Basic, etc. Typically a developer is responsible for a certain section of the project and codes exclusively on that section. The code developed is then shared with other developers that can incorporate it with their code. In a sense, this is what's happening with Blitz.

The developer of the Blitz Basic language, Mark Sibly, has programmed the graphics, sounds, input, multi-player, and many other routines that you, the game developer, can incorporate into your project.

Here is an example program to give you an idea of what code in Blitz Basic looks like:

```
Graphics 800,600

Text 0,0,"This is some sample code!"

WaitKey()

End
```

Notice that most of the text is very English-like. This is how most programming languages are these days. There are still some languages (such as Assembler) that are much more cryptic when compared to the easily-read Blitz Basic language.

Object Code

When you have completed a project, you must request that Blitz Basic translate the code in your project to something the computer can understand. This process is known as "compiling." What this process does is basically take your English-like commands and turn them into Object Code, which is also known as Machine Code.

Object Code is the native language of your computer's processor. It's nearly impossible to read since it is purely numerical, which is why we develop in languages such as Blitz Basic and allow the compilers to do the conversions for us.

Bits and Bytes

Before going much further, let's touch on the topic of bits and bytes as you'll need to know what these are for some of the information coming up.

A bit is the smallest unit of storage in a computer. Since computers actually read only 0's and 1's, each is measured as a bit. For example, the letter "A" consists of 8 bits (or eight 0's and 1's) that, when combined, total the numeric value of 65.

A byte is a combination of 8 bits. So, in order to get that letter "A," we must use a byte value. Each bit in a byte has a value assigned to it based on its position in the byte.

128	64	32	16	8	4	2	1

Now, starting from the right side you'll note that each number increases by a factor of itself. 1+1=2, 2+2=4, 4+4=8, etc. Each of the little squares in that diagram represents an element of the byte, or a bit. In actuality, those boxes would contain either a 0 or a 1, not the number shown in that diagram. But referring to the diagram, the byte total would accumulate the represented number if the bit contained a value of "1." Here's an example:

0	1	0	0	0	0	0	1

Since the first and seventh bits are flipped on, we know to take the byte values of "1" and "64" (as per the previous diagram) and add them together, thus making this byte value a total of 65. If all the bits are set to 1's, you would have to add all the values up in a byte by element and you would get the byte value of 256.

Screen Resolutions and Bit-Depth

In order for BB to start up a program, it must know what screen resolution you're going to use, and its bit-depth. Screen resolutions come in all shapes and sizes, and which ones are available to you is based on the quality of your video card.

You may have heard people use the terms "640x480," "800x600," and "1024x768." Those are a few of the many resolutions available. Basically, the first number describes the number of pixels that go across the screen (the

width). The second number describes the number of pixels going from the top to the bottom of the screen (the height). So, "640x480" simply means that there are 640 pixels going across and 480 going from top to bottom.

The biggest advantage of having your game use a higher resolution is that the images displayed are crisp and you can fit more on the screen. The biggest disadvantage is that it makes it a speed hog. I'll get into why there is a slow down in the next section.

Bit-Depth is the number of bits used to display the color of each pixel. You can choose from 8-bit, 16-bit, 24-bit, and 32-bit.

8-Bit Color: As you may recall in our discussion on bits and bytes, 8 bits can only contain a number up to 256. This means that each pixel drawn can be 1 of 256 colors. Sounds very limiting, huh? It is, but keep in mind that a lot of games were made using this bit-depth. Look at most any game made between 1987 and 1997 and you'll see 256 colors in action. 8-bit caused most games to work with palettes. Palettes allowed the artist to re-assign color values to the various 256 spots. This made it possible to have various shades of the same color, which made color transitions much more pleasing to the eye. Unfortunately, it also meant that the artist would lose a color for each shade created. As you might imagine, it was quite the challenge to handle art development for this environment. This depth also made it so the programmer would have to write code to handle the various palettes created.

16-Bit Color: In the late 90's, 16-Bit color on the PC became a way to produce better quality graphics. This is because the artists were no longer held to the 256 color limitation. With 16-bits the artist can use up to 65,535 colors per pixel. At that level of colors, palettes pretty much got tossed out the window. Artists started creating much more stunning graphical elements. This was a huge step in the game industry because it allowed for more realistic environments. The challenge, as we'll see in a bit, was that use of 16-bit greatly affected the speed of games.

24/32-Bit Color: Over the last couple of years 24-bit and 32-bit color depths have hit the market. 24-bit offers the ability to use one of 16,777,215 colors per pixel, and 32-bit allows one of 4,294,967,295 colors per pixel. That's a TON of color choices…more than the human eye can distinguish, actually.

So, how do you choose which bit-depth to use in BB? The best thing to do is to let BB make this determination for you, since it will select the quickest bit-depth for the resolution you have selected.

Speed Impact of Higher Resolutions and Bit-Depths

The higher the resolution and bit-depth, the slower your game will run. The reason for this comes down to how many pixels must be displayed per screen and how many bits each pixel contains.

Let's use the case of 640x480 with a bit-depth of 8. Since 8-bits is 1 byte, we are in effect saying that we need to draw 640 bytes x 480 bytes for every screen we render. To put that into perspective, we have to use 307,200 bytes for each rendered screen. That's A LOT of bytes. If we increase that bit-depth to 16, then we have to draw 2 bytes for each pixel, thus increasing our total byte use to 614,400. Now granted, the pictures are a bunch prettier, but that's double the bytes required for each render.

To make this even more impressive, let's say our video mode is 1024x768 with a 32-bit depth. The math is 1024x768x4 (since 32-bits is 4 bytes). The total bytes per render equals 3,145,728!

If you've ever heard the term "Frames Per Second," you'll start getting why this is so important. Commonly known as FPS, it's the number of frames of animation your game can show every second. This is important because the human eye requires a minimum number of frames per second to be fooled into believing that an image is actually "moving." If the FPS is too low, the eye will pick up the choppy effect and will not be fooled.

Screen resolution and bit-depth affects this number because of the number of bytes required to make a single frame of animation. 640x480x16 will take twice the amount of time to accomplish this than 640x480x8. 1024x768x32 will take quite a bit longer than 640x480x8! So the higher the resolution and the higher the bit-depth, the slower your FPS…and that's BEFORE you get into other elements that impact FPS such as Artificial Intelligence and various graphical effects.

The good news is that today's video cards are getting faster and faster. Some of the higher-end cards can play at extremely high resolutions and bit-depths while maintaining a minimum FPS of over 30. Actually, many of today's cards perform better at higher bit- depths! It's really the older machines and older video cards that you have to be careful with.

DirectX, Peripheral Cards and Drivers

DirectX: The current underlying system for Blitz Basic is a Software Development Kit (SDK) created by Microsoft that's called DirectX. DirectX is simply a set of routines that work within the Microsoft Windows environment to handle graphics, sounds, input devices, etc. It was written in such a way that peripheral manufacturers could easily support powerful multimedia enhancements by just providing updated drivers.

Some of you may be wondering why you wouldn't just use a programming language other than Blitz Basic to interact with DirectX. The primary reason is that DirectX can be somewhat cryptic, especially for newer users. You would need to understand Windows programming architecture and understand the fundamentals of COM (Component Object Model) programming to really utilize the power of DirectX directly. Blitz Basic allows you to focus on creating your game or application in a simple to use, easy to learn language that is extremely fast and powerful. In a nutshell, Blitz Basic lets you get to work on your project without having to understand all the fundamentals of Windows and DirectX programming.

Future versions of Blitz Basic may support openGL or another graphics SDK that is more focused on 3D development. OpenGL has the advantage of supporting many different operating systems, such as Windows, MacOS, Linux, and UNIX.

Peripheral Cards and Drivers: Peripherals are basically anything that you add to your computer that has some type of interaction with you/your computer. Examples are: video cards, a mouse, a joystick, a keyboard, etc.

With so many brands of peripherals on the market, developers were having a difficult time programming their games to support the functions of each one. DirectX helped address this problem by requiring the various manufacturers to conform to the DirectX model—assuming the manufacturer wanted to get Microsoft DirectX certified.

In order to stay up on the latest DirectX versions, the manufacturers have to constantly update the drivers for each peripheral based on direction from Microsoft's DirectX developers. Drivers are simply a set of interface programs that DirectX uses to communicate with the peripheral. You should always ensure that you have the latest drivers for your peripherals, and you should make sure to inform the players of your games that they should install their latest drivers as well.

Creative and Technical Design Documents

One of the most important things to consider when beginning any development project is design. Designing is just the process of making sure you have a road map of where you want to be at the end of the development cycle. Without a design you'll basically be playing it by ear in your development. For small projects, this is usually not so bad, but the larger the project becomes the more likely you'll have a lot to re-do if you don't plan properly.

So how do you go about designing? Depending on the scope of your project, a design may only be a couple of quick sketches and a few lines that help to

remind you what to look for as you develop. But larger projects require more detail and typically are separated into "Creative" and "Technical" design documents.

I have seen creative design documents that are over 1,000 pages long! They've included the main story line, profiles for each character, weapon details, game level/map details, NPC (non-player characters) details, etc. The technical design documents are usually smaller, ranging from 30-250 pages.

Don't be too concerned here, though. Keep in mind that these documents are for games that have millions of dollars backing them. The biggest design document I've written for personal use was about 50 pages long and the technical document was about 20.

When working on your creative design document, you'll want to focus on a number of questions, such as:

1) What is the game about? If I had to sum it up in ONE sentence, what would I say?
2) What type of game is it? First-person shooter, role-playing game, strategy, etc.
3) What are the primary features? Cool graphics, game-play, multi-player, etc.
4) Who is the main character…or are there many to choose from, and what do they look like, etc.?
5) Where is the game set? Is it ancient Rome, a distant galaxy, a cloud molecule, etc.?
6) Who are the bad guys, and why are they bad guys?
7) What do all the bad guys look like, and what are their names, etc.?
8) What is the ultimate goal of the player and what are the main obstacles stopping that player from attaining that goal?
9) What will the player's interface look like (also called the HUD "heads-up display")?

There are many more questions you could ask yourself, but this should get you started on seeing what creative design is all about.

Now, you may just want to re-create a game that has already been done. If so, you probably won't need to deal with a creative design since you've played the game so much that it's ingrained in your mind. But either way, you'll probably want to write up the technical design document.

Technical design documents are simply a list of technical issues that you'll likely face when developing your game, and the steps you plan to take in tackling these issues. A simple example of this may be the desire to have

different explosion types based on the weapon being used by the player. This is a simple example because you can just check which type of weapon was fired and then tell your program to display the respective explosion upon contact.

A more complex example would be unit movement. Let's say that you have a bunch of units in your army and you need to move them from point A to point B. To make matters worse, your maps include obstacles such as water, trees, and buildings. You may think that this is a simple task, but it's pretty complex because you have to remember that you're just displaying little graphical images…they don't know there are trees in the way! With this you would either write down "To Be Resolved" in your document, or you'd go and study up on path-finding algorithms such as A*. Don't be too concerned here…there are a lot of libraries that have already been written to help you handle these types of issues.

Good Coding Style and Commenting

Everyone has his/her own style with how to do things, but some styles are based more on being different than being clear. If you ever have the notion to allow other developers to use/modify your code and/or work on a team with you, I would highly recommend that you adopt a style that is accessible.

Commenting code is the most important, yet most overlooked, aspect of development. I can think of nothing worse than seeing pages and pages of code without a single comment as to what the code does. This makes for a seriously difficult time in maintaining or upgrading and should be avoided at all costs. I've fallen for this trap and have found myself confused at my own code after not seeing it for months.

To make matters worse, commenting is EASY. All you have to do is write a quick line that describes what a section of code is for.

A Place to Work

Okay, you may think this part is goofy but it's probably the most important part of your development project. Game developers are notoriously lazy. You need to find a place where you can focus on your game designing and development that feels comfortable and fits your mood.

To give an example, my office is full of gamer junk. There are toys all over the place and there's a killer sound system that keeps the music going so I can't hear anything else going on in the house that may distract me. I don't play with the toys (most of the time), but they set the tone that I'm a game-developing junkie and that keeps me in the mood to create! Another cool part of this is that when I face development roadblocks, I don't easily give up.

Since I'm in a comfortable development place (my happy place!), I'm already in the right mindset to tackle tough issues.

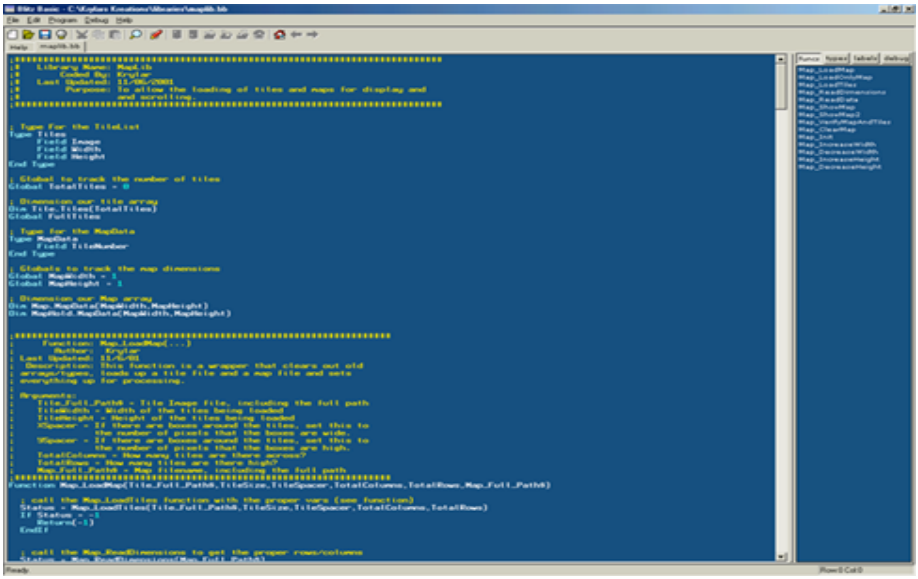
Again, I know this sounds goofy, but if you don't make sure you're set in this department you'll soon find yourself slowly drifting away from your efforts.

Chapter 3: Getting Started with Blitz Basic

The Blitz Basic IDE Main Screen

The place where you'll be doing most of your programming is inside the Blitz Basic Integrated Development Environment (IDE). The IDE is merely an editor that has enhanced features to help you program effectively.

A very helpful aspect of the IDE is that it will show different code pieces in different colors. Blitz Basic keywords (commands that you use in Blitz Basic) default to a yellow color, for example. There are also different colors for text you wish to display, comments that you put in your code, numbers, and so on. It makes it much easier from a glance to see what's going on in your code with the various color schemes. Here is a screenshot of the Blitz Basic IDE. Note that this is the newer Blitz3D version of the IDE, which still has all the full 2D commands but also allows for robust 3D development.



(Figure 3.1)

The first thing to notice in the above is that the code window is pretty large. This is because your code will often extend far to the right of the page and you'll want to see as much of it as possible. If your code extends beyond the right side of the code window, you will see a scroll bar that will allow left and right scrolling.

The window on the right hand side contains one of four things, depending on your selection.

- **Functions:** Lists all of the functions in the currently opened window. Functions are simply smaller programs that do a particular thing.
- **Types:** Lists all of the Types in the currently opened window. Types are sets of related data that can have many elements.
- **Labels:** List of all the Data statement names (and any inserted label areas) in the program.
- **Debug:** Helps you track the values stored in your variables so you can more easily find problems.

At the top of the window there are a group of icons that have specific uses. Here is the list breakdown (these images will appear slightly different in the Blitz2D version, but they should be easily recognizable):

 - Create a new file.

 - Open an existing file.

 - Save your currently opened file.

 - Close your currently opened file.

 - Cut the currently highlighted text.

 - Copy the currently highlighted text.

 - Paste the previously cut/copied text.

 - Search the opened file for a particular word.

 - Compile and Run the currently opened file.

 - Pause the currently running program.

 - Resume the previously paused program.

 - Execute the next line in the program and then stop. If the next statement is a function call, then run the rest of the program before stopping.

 - Execute the next line in the program and then stop. If the next statement is a function call, then run to the first line of the called function and stop.

 - Execute until exiting a function call. Then stop.

 - Completely stop the program and return control to the IDE.

 - Go to the main page of the Blitz Basic IDE.

 - Go to the previously viewed page in the IDE Help System.

 - Go to the page you just clicked previous from, in the IDE Help System.

You can also click on the main menu items: File, Edit, Program, Debug, and Help to access the various Blitz Basic IDE tools.

What Every Blitz Basic Program Must Contain

To setup Blitz Basic's graphics system, you must use the GRAPHICS command. This command takes up to 4 arguments, depending on how you wish to handle bit-depth. The first two arguments are the screen resolution (width and height); the last 2 arguments are the optional bit-depth and the window type. Here's what the command looks like:

```
Graphics 640,480,16,2
```

This line tells BB to start your game using 640x480 resolution and 16-bit color, and use a desktop window instead of full screen. For standard full-screen games, it's best just to do this:

```
Graphics 640,480
```

The GRAPHICS command is the only command that your program *must* contain. If you leave out this command, your program won't even start.

The Good Old "Hello, World!" Program

Almost every programming book I've ever seen starts out with a program that simply puts "Hello, World!" on the screen. Typically I dare to be different, but in this case I'm going to keep with the norm. Type in the following code *exactly as shown*, save the file and then ask BB to run it by clicking on the little Rocket () icon.

```
Graphics 640,480

Text 280,240,"Hello, World!"

Waitkey()
```

End

Now let's break this down so you can see what's going on.

Graphics 640, 480

We've already discussed this line in detail above. We're just telling BB to setup our video mode to be 640 pixels wide by 480 pixels high.

Text 280,240,"Hello, World!"

The **TEXT** command is used to display information to the user. The first argument sent to this command (in this case, 280) is the X, or horizontal position to start displaying the text. The second argument (240) is the Y, or vertical position to display the text at. Finally, the third argument is the actual text, enclosed in quotes that we want to display.

WaitKey()

This command tells BB to do nothing until the user hits a key. This command is very useful when starting to program as it allows you to see results without having to worry about the screen closing on you.

End

Although this command is not necessary, it's a good idea to include it. Basically, the END command tells BB that you have finished with the graphics system and that you want to return complete control to the Operating System. If you don't include this command, BB will put up a dialog box that says, "Program has ended," but it won't hurt anything.

Chapter 4: The Basics of Blitz Basic

Variables, What are they?

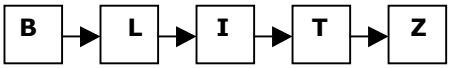
An important part of any Blitz Basic program is the ability to have various forms of data. It can be numerical, character, memory, graphical, etc. All data can be entered into your program manually, but this doesn't allow for the dynamic nature of most applications.

So how can we store a value that we can update at any time? We do so by using variables.

A variable is simply an area of your computer's memory that has been set aside for holding values that you wish to hold and manipulate. Variables are created real-time by the developer. The amount of memory they consume depends on the type of variable required to hold the proposed value. Here is a list of variable types in Blitz Basic:

- **String (also known as Scalars)**
- **Integer**
- **Float**
- **Pointer/Handle**

Strings: A String is simply a collection of characters. For example, "Blitz Basic" is a string of 11 characters. Why is it called a String? The idea is that the "B" is tied to the "l" and that is tied to the "i," and so on. So if you were to take all those letters and "string" them together, you would get the words "Blitz Basic." Consider the following:



Individually, just like letters in the alphabet, these are simply characters. But those arrows demonstrate the way Blitz will look at a string. Each letter points to the next and treats them as you would a word.

Strings are used for any textual information that you will deal with. Examples would be the player's name, the planet they set up on, the description of that planet, the ship they're flying, the ship's description, etc.

Integers: The basic number variable. Any time you want to hold whole numbers (all numbers that are not fractions are whole numbers - whether negative, positive or zero), you would use an Integer variable. Maybe you want to keep track of a player's score, or their ship's current speed, missile count, shields, etc. Integers are the way to go.

Floats: A float value is important when you are looking for more precision in your calculations. The term "Float" means "Floating Point," and it's simply referencing that the decimal point can float (or move) from one position to another in a value.

For example, you may have the value 10.75. If you multiply that value by 10, you'd get 107.50. Notice that the decimal point "floated," or moved, over one space to the right.

Floats are particularly useful when making precision movements from one screen location to another. They allow for smooth movement because they can have such tiny adjustments in values. Also, let's say you have a big space freighter that takes a while to reach top speed. If top speed is 5, counting by 1 isn't going to take long at all, but counting by .00001 would take quite a while.

Note that Blitz Basic currently supports a precision up to 6 to the right of the decimal.

Pointers/Handles: A memory value that holds the position of another value. For example, when you load an image, you will have an image handle that you use from that point on to reference that image. So, in essence, you are *pointing* to that image when using image functions.

Defining Variables

There are a few ways that variables can be created:

- **Global:** This type of variable will be available for reading and manipulation by ALL of your Blitz Basic programs.

- **Local:** This type of declaration will make this variable available for reading and manipulation by ONLY the section it is declared in. This is the default for any defined variable.

- **Argument:** This is a variable type that is used with functions, which we will discuss in a later chapter.

A very important issue when using variables is creating a name that is meaningful. You'll often see variable names such as "a" or "xs" or some other seemingly random grouping of letters. To the developer of the program, these may have a significant meaning; but to the world that's going to modify this code, it's gibberish.

When you are creating a variable name, think about what the variable does and then use something descriptive to define it. For example, let's say that

you need to keep a list of the player's current total score. Why not name the variable "TotalScore"? It makes sense immediately what the variable is for, and it's not overly verbose.

Sometimes I will use a little descriptor at the beginning of most variables so I can instantly see what kind of variable it is. I'll use the letter "i" for Integer, "s" for String, and "f" for Float. So, instead of using "TotalScore," I would likely use "iTotalScore." I now know, by just a glance that this variable is an Integer and it's used to hold the total score of the player.

One last thing on naming conventions… Notice that I also capitalize the first letter in each word. Again, this is just to make things more clear. Typically you don't need to do this, but it's good practice. Think of a variable that is to hold the passing scale of a student in a class. Without capitalization, the variable would be "passscale." You could easily miss an "s" in that. With capitalization, it becomes clearer: PassScale.

Here are examples of good variable names:

- sPlayerName
- iCounterValue
- fShipAcceleration
- PlanetDescription
- BrakingSpeed
- ShieldPower

The first step in using a variable is to declare it. In Blitz Basic, this simply means that you put up the variable name and assign it a value. Assignments are done using the "=" symbol. Here's an example:

```
iTopSpeed = 5
```

That one line sets up the variable iTopSpeed as an Integer and assigns it the number 5. From here we could easily adjust that value by doing a mathematical function on it. Let's say that our ship's top speed just increased by 2 because we got a really cool new engine installed. We could do the following:

```
iTopSpeed = iTopSpeed + 2
```

That's the equivalent of saying iTopSpeed = 5 + 2, because remember that our top speed value was originally assigned 5.

That describes how Integer variables are setup, but what about Strings and Floats? The only difference is the variable name and the type of data assigned. For example:

```
fTopSpeed# = 5.5
```

Notice the # sign at the end of the variable name. This tells Blitz that you want this value to be of type float. You only have to put the # symbol on the variable name when you declare the variable. Blitz will remember its type throughout the program.

```
fTopSpeed# = 5.5
fTopSpeed = fTopSpeed + .5
```

Blitz will now alter fTopSpeed to hold the value of 6.0. Since you don't have to use the # throughout, it's a good idea to start that variable with the "f" so you don't confuse this variable type with an Integer. Sometimes I just use the # throughout and don't use the "f" notation.

```
sPlayerName$ = "Krylar"
```

The above example creates a variable of type String. The $ at the end of the variable instructs Blitz that the data held will be character data, non-numeric. Again, it's unnecessary to keep putting the $ throughout your code, but you may want to do that anyway, or adopt the "s" at the beginning of the variable declaration.

One last thing to note: Integer types CAN be created using the % at the end of their respective name, but it's not necessary as Integer is the default variable type for Blitz.

```
iValue1 = 1
iValue1% = 1
```

Those two variable declarations are identical to Blitz Basic. Some folks prefer using the %, others don't. There is a benefit to using the %, #, and $ throughout the code, and that is that Blitz Basic will check each occurrence for valid data that way.

Commenting Your Code

Everyone has a style for commenting code, and you will likely build your own method as well, but here are a few things to think about when commenting:

- Make comments as clear and concise as possible. Brevity is important, but only if the comment clearly conveys the purpose of the code.
- Try to comment as you code, not as an afterthought. Commenting as you code ensures that you'll have a fresh perspective on what the code is doing. It can also help you pinpoint bugs easier since you'll need to clearly describe the code piece.
- As you update your code, also update your comments. Comments are only as good as the code they describe. If the code evolves and the comments don't, then the comments quickly become irrelevant.
- If there are multiple people working on the code, make sure you put an identifier in the comment to denote who changed the piece of code and updated the comment.
- It is sometimes best to date subsequent changes on applications released with source code. This is so other developers can know what has changed and where.

You may decide to never share your source code with others, but this doesn't mean that you should avoid commenting. One day you will likely end up revamping your own code and you'll be just as lost as anyone else looking at your non-commented code.

Even though Blitz Basic is a simple language, algorithms can still become quickly cryptic. Worse even is that you often will find yourself hacking your own code to make it do what you want. This is typical for most programmers, but when you come back a year later to update this code you'll be completely confused at what you were thinking about if you don't clearly comment it.

To help you understand this, I'm going to take our "Hello, World!" program and comment it. Notice that the semi-colon (;) is used at the beginning of each comment line. Blitz Basic will consider anything after the semi-colon and up until the end of the line a comment, instead of code. I put the asterisks (*) in simply to make the sections more pronounced in the program definition.

Compare the first "Hello, World" program to the following one. Granted that this is a very simple program that needs little explanation, but you can immediately see what the purpose of the program is, when it was updated, what was updated, and a piece by piece breakdown of what is being done.

```
;***********************************************
; Title: Hello World!
; Files: helloworld.bb
; Author: Krylar
; Current Version: 1.0
; Last Updated: 01/01/01
```

```
;************************************************
; Description:
;          Simply puts up "Hello, World!" and waits for a
; keypress.
;************************************************
; Update History:
;         12/01/00: Started project
;         01/01/01: Moved the text to the top of the screen
;************************************************

; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; Updated 01/01/01: Write "Hello, World!" at 280,0
Text 280,0,"Hello, World!"

Waitkey(); Wait for the user to press a key
End  ; Tell Blitz Basic that we're finished
```

Some people prefer to put their comments directly after the commands, as follows:

```
Graphics 640,480    ;Initialize to 640x480 resolution
```

This method is fine, too. Actually, I will usually use both methods in my code, as you will see later on. ***Note: I will not be including the top comment section in all of the examples due to space limitations.***

Regardless of the number of comments in your code, your final application file size and speed will not be affected. This is because Blitz Basic completely ignores all comments when it compiles your code. Thus, to Blitz, it's as if they're not even in there. That said, however, there is such a thing as commenting too much. You don't need to be overly verbose as long as you're clear. If you find that you're putting in a paragraph to describe a single line, you probably need to rethink what you're trying to do.

Simple Arithmetic

Math is an essential element of most any game you'll develop, so you'll need a way to perform calculations. Later we'll get into the advanced calculations that you can do to get various effects working, but for now let's just look at simple arithmetic.

Addition, subtraction, multiplication, and division are handled by the symbols +, -, *, and /, respectively. For example:

```
iValue = 1 + 2
```

iValue would be equal to 3. That's simple, no? If you replace that + symbol with any of the other symbols (-, *, or /) you'll get a different result, but it's still easy.

But look what happens when we have calculations like this:

```
iValue = 1 + 2 * 10 / 5 – 3
```

You may think that BB will tackle the problem like this:

```
1+ 2 = 3
3 * 10 = 30
30 / 5 = 6
6 – 3 = 3
```

But it won't. This is because BB will use *precedence* when calculating this value. Precedence simply means the order in which an equation is calculated. Like standard math, equations are calculated in BB by handling first multiplication and division, then addition and subtraction. Some of you math whizzes may know that exponents and parenthesis, etc. will take precedence even over that… We'll get there - don't worry.

So, here's how BB will handle the above calculation:

```
2 * 10 = 20
20 / 5 = 4
1 + 4 = 5
5 – 3 = 2
```

So, what if you *were* looking to get "3" as your answer? You'd have to use parenthesis to change the precedence of the calculation. Here's what your calculation would look like:

```
iValue = (1+2) * 10 / 5 – 3
```

The addition of the parenthesis will make it so the addition will occur before the multiplication, thus resulting in "3" instead of "2."

This is a very important concept to grasp because you can literally change the outcome of an equation by a misplaced parenthesis or by not including parenthesis where they are needed. So be cautious of this.

Another area that we'll touch on quickly is exponent math. An exponent is a number that is multiplied by itself a set number of times. In Blitz this is represented by using the ^ symbol.

```
2^4
```

…is the equivalent of

2 * 2 * 2 * 2, which equals 16.

An exponent will be handled before multiplication and division, but after parenthesis. So the order of precedence is as follows:

```
( ), ^, *, /, +, -
```

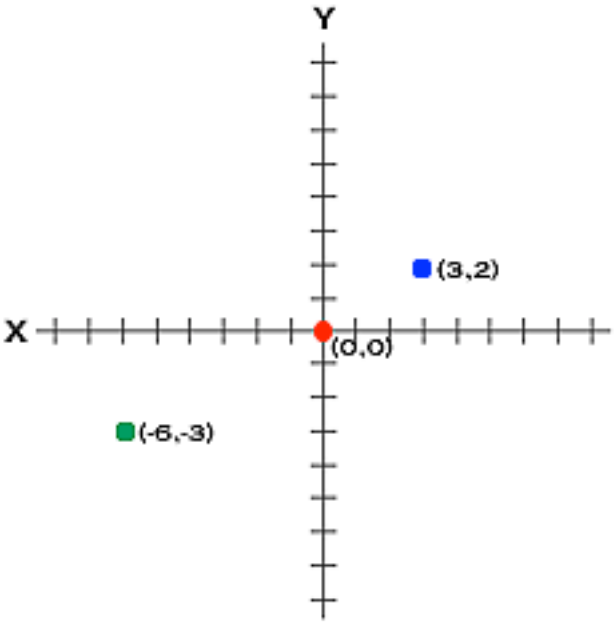
Cartesian Coordinates

While the object of this book is not to teach mathematical concepts, the Cartesian Coordinate system is something you'll need to understand to grasp how Blitz Basic handles things. If you already know about this system, feel free to skip ahead to the next section.

The Cartesian Coordinate system is just a way to show points on a two-dimensional graph. Each point has a horizontal, often referred to as X, and a vertical, often referred to as Y, value. These values describe the location that a point will have on the graph. You may hear people using terms such as "x, y coordinates" when regarding two-dimensional (2D) games. They are simply referring to the pixel's horizontal and vertical position on the screen.

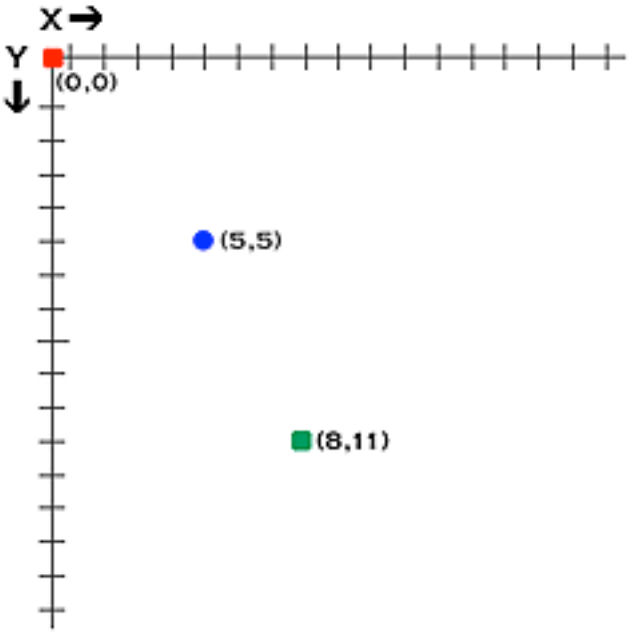
In figure 4.1, you can see what a Cartesian graph looks like. The dot at (0,0) represents the position in the graph known as the origin. The origin is the starting point of all other positions. Anything to the right of the origin on the X-axis (horizontal) is a positive number. Likewise, anything to the left of the origin on the X-axis is a negative number. On the Y-axis (vertical), anything above the origin is positive and anything below is negative.

Note that the dot in the upper-right has a position of 3,2. This means that the X position is 3 spaces to the right of the origin, and that the Y position is 2 spaces up from the origin. The lower-left dot (-6,-3) demonstrates a negative position on the graph.



(Figure 4.1)

Blitz uses the Cartesian system for drawing pixels, text, and images to the screen, but the placement of the origin does not allow for negative X and Y positions. The origin used by Blitz is the upper left corner of the monitor. Refer to figure 4.1 to see what Blitz does when handling Cartesian coordinates.



(Figure 4.2)

As you can see there are no negative values to worry about when drawing in Blitz. You would still have to worry about negative values when comparing two locations, of course, but that's easily accomplished with simple subtraction.

Chapter 5: Program Control Statements

While it would be nice to simply have five lines of code to create a full game that meets all your expectations, that's not going to happen anytime soon. The reality is you'll probably be looking at thousands of lines of code. This being the case, we'll need a way to execute only the pertinent lines at the appropriate times. To do this requires the use of program control statements.

If…Then…Else…EndIf

Even if you've never done any programming in your past, you're already familiar with the concepts of IF…THEN…ELSE. Why? Because you use this in every day decisions that you make.

Take, for example, simply deciding what to have for dinner. IF I cook dinner THEN I will need to prepare all the food and clean up afterwards, ELSE I'll have a messy kitchen. IF I go out to dinner THEN it will cost me some hard-earned cash. Any time you make any decision, you go through the IF…THEN…ELSE process, and it's no different in your code.

The format looks like this:

If *Condition is True* **Then**
  …process commands…
Else
  …process commands…
EndIf

Let's write a little program that asks the user to enter some number. If it turns out to be the number 1, say so. Otherwise, tell the user it's not the number 1.

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; Ask the user to enter a number
sAnswer$ = Input$("Enter a 1 or some other number: ")

; Did the user respond with a "1"?
If sAnswer$ = "1" Then
  ; Write out "You entered a 1!"
  Text 0,20,"You entered a 1!"
Else
  ; Write out "You did NOT enter a 1!"
```

```
  Text 0,20,"You did NOT enter a 1!"
EndIf  ; end of  If Answer$

Waitkey()  ; Wait for the user to press a key
End  ; Tell Blitz Basic that we're finished
```

Notice how we can control what our program does by using evaluations? This is very powerful since we are in a constant state of evaluation during a game. Think of the following evaluations:

- Is the player running or walking?
- Did the player fall?
- Is the player jumping?
- Is the player being stopped because of a wall?
- Is the player firing a weapon?
- Was the player hit by an enemy's attack?
- Does the player have any "lives" remaining?
- Did the player make the high-score list?
- Did the player meet the objectives of this level?

This is a tiny list of the questions you'll need to answer during the course of your game. The larger the game, of course, the more questions you'll be asking.

It's important to note that you don't need to use an ELSE if it's not needed in your evaluation. For example, if you wanted to print "Shields On!" if the variable *iShieldsOn* was equal to 1, you would do the following (note that this little snippet of code will not run on its own):

```
; if ShieldsOn is equal to 1
If ShieldsOn = 1 Then

   ; Write "Shields On!" at the top of the screen
  Text 0,0,"Shields On!"
EndIf ; end of If ShieldsOn = 1
```

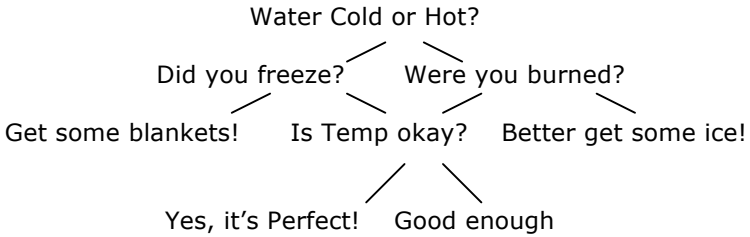
You don't need the ELSE here because the text will only be displayed if the variable is equal to 1. Now, if you wanted it displayed when the shields are off, you would use an ELSE for that.

Also note that Blitz Basic does not require that you use the THEN portion of the IF…THEN…ELSE…ENDIF construct. Some BASIC languages do require this and many people find it as good practice to use it, but it is not required. So, taking out THEN would have no effect on the outcome of the above code.

At the end of every IF…THEN…ELSE…ENDIF construct, though, you *must* put the ENDIF. This is the only way that Blitz Basic knows you've completed this particular "*decision block*." A *decision block* is a term used to describe a set of instructions acted upon when a particular decision has been made. If you removed the ENDIF from the above example, you would get an error when you tried to run the program.

Nested IF Statements

Sometimes decisions will need to be made as part of other decisions. This is sometimes called a "*decision tree."* If you've ever done a flowchart, you are already aware of what a decision tree looks like from a flowcharting perspective.



At first this may look kind of confusing, but spend a few seconds studying it and it should become clear. We're simply asking a bunch of questions, and based upon the response, another pertinent question is answered.

But how would we represent that in our code? We'd have to use nested IF…THEN…ELSE…ENDIF constructs. Here is the code:

```
; if the water is hot
If WaterHot = 1
   ; see if the user got burned
   If WereYouBurned = 1
      Text 0,0,"Better get some ice!"
   Else
      ; is the temp okay?
      If SoTempIsGood = 1
         Text 0,0,"Good enough!"
      Else
         Text 0,0,"Yes, it's perfect!"
      EndIf ; end of If SoTempIsGood
   EndIf ; end of If WereYouBurned
Else
   ; did the user freeze?
   If DidYouFreeze = 1
```

```
      Text 0,0,"Better get a blanket!"
   Else
      ; is the temp okay?
      If SoTempIsGood = 1
         Text 0,0,"Good enough!"
      Else
         Text 0,0,"Yes, it's perfect!"
      EndIf ; end of If SoTempIsGood
   EndIf ; end of If DidYouFreeze
EndIf ; end of If WaterHot
```

I know that's a lot to digest your first time around, but study that carefully and compare it to the decision tree above. If you take it line-by-line you should be able to see how it works pretty easily.

We talked about the various evaluations a bit in the IF…THEN…ELSE…ENDIF section, how do those relate to nested IF's? Here's a breakdown of some on that same list with additional questions, to give you a taste:

- Is the player running or walking?
  - Does the player have on Rocket shoes?
    - Which model?
  - Is the player on a conveyor belt?
- Did the player fall?
  - Was the player injured from the fall?
    - Did the player land on something sharp?
    - Is the player still healthy enough to continue?
      - Will the player's speed be affected?
      - Will the player's jumping ability be affected?

See how quickly you can get into many areas of evaluations? And also how one evaluation can spring up many others? Now you understand the need for decision trees and nested IF's.

Not, And, and Or Statements

There will certainly be occasions where you'll want to compare two or more values on the same IF line. Imagine you wanted to know if the player has been hit while jumping. You could do a nested IF, of course, but it's not necessary. Instead you can ask Blitz if *both* cases are true on one line.

```
; if the player has been hit and is jumping
If PlayerHit = 1 And PlayerJumping = 1
  ; take away 2 points from the shields
  PlayerShields = PlayerShields – 2
```

```
Else
  ; otherwise, just take away 1 point
  PlayerShields = PlayerShields – 1
EndIf
```

Let's look at the functionality of each of these.

NOT: This statement asks Blitz to evaluate the condition with reversed logic. In other words, respond that the condition has been met when the condition is NOT true. There are many reasons to do this, but one of the more common is continuing to process commands while the user has NOT pressed the Escape key (we'll touch on this in the loops section).

AND: This checks to see if two or more conditions have been met. The main thing to note is that ALL of the conditions must be met when using AND in order for Blitz to return a positive result. Something to think about when using the AND is to always use the most common check first in the list. In our above example we first checked to see if the player was hit before bothering to see if he was jumping. If the player wasn't hit we don't want to waste time checking for the jump, right? Since the AND requires *all* conditions to be true, if the player was not hit, then the rest of the statement is ignored… which saves time.

OR: The OR statement allows you to check if one OR another statement is true. What if you needed to check whether a player was hit by shrapnel OR an explosion? You could use nested IF statements, of course, or you could use OR.

```
; was the player hit?
If PlayerHit = 1

   ; was it just by shrapnel or the effect the explosion?
   If ByShrapnel = 1 Or ByExplosion = 1
      ; just take 3 damage points off the player's shields
      PlayerShields = PlayerShields – 3
   Else
      ; must have been a direct hit
      ; take the appropriate damage off
      PlayerShields = PlayerShields – ProjectileDamage
   EndIf ; end of If ByShrapnel …

EndIf ; end of If PlayerHit
```

The SELECT Statement

What if you have a bunch of things to check, but you don't want to have a bunch of IF statements to check it with? You may allow the user to hit different keys in your game, each having a different purpose. You have left arrow, right arrow, up arrow, down arrow, spacebar, etc. Doing an IF statement for each of these may start to make your code look a little sloppy. So what do you do?

Enter the SELECT statement. In a nutshell, SELECT allows you to check one variable for a lot of different values. Here is an example:

```
Select KeyValue
    Case LeftArrow
        Text 0,0,"You hit the left arrow!"
    Case RightArrow
        Text 0,0,"You hit the right arrow!"
    Case UpArrow
        Text 0,0,"You hit the up arrow!"
    Case DownArrow
        Text 0,0,"You hit the down arrow!"
End Select
```

Now, compare that with the IF method:

```
If KeyValue = LeftArrow Then
   Text 0,0,"You hit the left arrow!"
Endif

If KeyValue = RightArrow Then
   Text 0,0,"You hit the right arrow!"
Endif

If KeyValue = UpArrow Then
   Text 0,0,"You hit the up arrow!"
Endif

If KeyValue = DownArrow Then
   Text 0,0,"You hit the down arrow!"
Endif
```

There's not an amazing difference in size, but you can see where the SELECT command could come in handy where one variable can have a multitude of values.

Loop Basics

There is a lot of repetitive action in video games. The game "Asteroids" is a prime example because it's the same thing over and over. The only difference from level to level is that there are more rocks and more UFO's. Other than that, it's essentially the same game throughout.

Due to this repetition in games, and programming in general, we need a way to do things multiple times without having too much code.

Imagine that you wanted to draw 50 asteroids on the screen, and imagine that drawing each asteroid would take one line of code. So, it's easy to deduce that you would have 50 lines of code. Now, take that a step further and say that you also have 30 laser shots flying out of your ship toward those asteroids. Now you've gone up to 80 lines of code. Each time a new asteroid appears or a laser shot is fired, so increases your lines of code. There has to be a more efficient way of handling this, right? Right, it's done by using *loops*.

A loop handles this because it is a means of telling Blitz Basic to do something over and over until a certain condition is met, which is precisely the kind of thing we're looking for.

There are three types of loops available to us in Blitz Basic:

- **For…Next**
- **While…Wend**
- **Repeat…Until/Forever**

Each of these loop types has its merits, so let's run through them one-by-one and discuss.

For…Next Loops

This type of loop can be considered as a "counter" loop. Meaning that it is given an initial value to start at, and then counts up until it reaches another value, and then it stops. As this loop continues counting, it will process any instructions repeatedly until it meets its destined value.

Here is the layout of a FOR…NEXT loop:

```
For Variable = InitialValue To EndingValue
    …process commands…
Next
```

Using our asteroid scenario, let's look at some pseudo-code to demonstrate the use of FOR…NEXT. First we'll look at the method of drawing ten asteroids without looping.

```
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
DrawImage Asteroid_Image,0,0
```

Now let's do the same thing using the FOR…NEXT loop:

```
For Images = 0 To 9
    DrawImage Asteroid_Image,0,0
Next
```

See how much smaller the latter is? You would really see a big difference if you had to draw 50 or 100 asteroids, wouldn't you?

If you're really observant, you'll notice that we didn't start from 1 and go to 10 in our FOR loop. We could have easily done this and it would have worked fine, but you should start getting used to the fact that computers count from 0, not 1. Where you go 1…2…3…4…5, a computer goes 0…1…2…3…4. You'll often see code that has counter offsets beginning at 0, so you should probably start getting used to that now.

Now, let's do something fun to really hone this in. Let's create a little program that lists the name "Blitz Basic" down the screen ten times. No, this isn't an amazing use of this powerful tool, but it helps get the idea across.

Since we don't want the text to overwrite the other pieces of text, we'll need to make sure that the Y-axis is spaced appropriately. The standard font used in Blitz Basic requires us to put a distance of about 16 pixels between the lines to ensure we don't overlap. To do this, we're going to keep an integer variable called "iTextY" and we'll add 16 to it each time the loop iterates. Doing this will tell Blitz Basic where we want each line of text displayed. Note that you could also use the PRINT command, and we'll touch on that a little bit later, but for most examples I'll be using the TEXT command.

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; initialize our vertical text-placement variable
iTextY% = 0

; loop 10 times and write "Blitz Basic!" on the screen
For iRows = 0 To 9
    Text 0,iTextY,"Blitz Basic!"

    ; increase our vertical offset to avoid overlap
    iTextY = iTextY + 16
Next  ; end of For iRows = 0 To 9

; Wait for the user to press a key
Waitkey()

; Tell Blitz Basic that we're finished
End
```

You can set your initial and goal values to virtually anything. Your initial value may be a negative or positive number, or zero. If you use an initial value that's greater than the ending value, however, you're going to run into a problem. Consider the following code:

```
For Images = 10 To 9
    DrawImage Asteroid_Image,0,0
Next
```

Notice that our initial value is greater than our goal value. If you guessed that Blitz would bypass this loop, you guessed correctly! But what if you wanted to count from a higher number to a lesser number? Maybe you need to count down from 5 to 1 because you've got a racing game and you want to convey when the racers can start.

You would do this by using the STEP command. STEP informs Blitz how you want the loop variable to be adjusted before evaluation. The following code demonstrates a countdown from 5 to 1, displaying the counter value as it goes. Note the use of the STEP command in this example:

```
For Images = 5 To 1 Step -1
    Text 0,0,Images
Next
```

That "Step –1" piece will inform Blitz to subtract 1 from the counter variable *Images* until it reaches the goal value of 1.

You can use any value to step with, also. Let's say that you want to count to 100 by 10's.

```
For Images = 0 To 100 Step 10
    Text 0,0,Images
Next
```

Pretty simple, eh?

You can also use a constant as the STEP increment/decrement value. To do this, you would set up a constant and assign it a value. Then instead of putting a number after the STEP command, you would put the constant name.

```
Const Value = 10

For Images = 0 To 100 Step Value
    Text 0,0,Images
Next
```

You cannot use a variable in conjunction with the STEP command though. Only constant values will work.

While…Wend Loops

Where a FOR…NEXT loop processes based on a count from one value to another, a WHILE…WEND loop can offer another option. This type of loop can simply repeat a set of instructions *WHILE* a certain condition is true. Yes, you can make this a count if you'd like, but it's not a requirement.

The functional layout of this loop is as follows (note that WEND simply means "While End," thus signifying the end of the loop):

While *Condition Is True*
    … process commands…
Wend

Here is a piece of code to demonstrate how you could use the WHILE…WEND combination to provide the same functionality as a FOR…NEXT loop.

```
Images = 0
```

```
While Images < 9
      Text 0,0,Images
      Images = Images + 1
Wend
```

So, what will this piece of code do? It will count from 0 to 9 and put that number on the screen. I personally prefer the use of the FOR…NEXT loop in these situations though, as it is tailored specifically for counting between two values.

The most common use of the WHILE…WEND loop is as the main game control loop, which we'll get into later. But for now let's look at a simple example that will blink "Hello, Blitz Basic!" until the user presses the Escape key.

```
; initialize Blitz Basic at 640x480
Graphics 640,480

; While the user has NOT hit Escape
While Not KeyHit (1)
        ; clear the screen
       Cls
        ; write out our text
       Text 270,240,"Hello, Blitz Basic!"
        ; wait for 250 milliseconds
       Delay(250)

        ; clear the screen
       Cls
        ; wait for 250 milliseconds
       Delay(250)
Wend  ; end of While Not KeyHit (1)
End   ; end of program
```

See how the WHILE…WEND loop continues to roll, unaffected, until the user presses the Escape key (which, by the way, has a *Scan Code* of 1)? This is very important because it gives us a method where we can more dynamically control a piece of code. There is still an end-goal in mind with this type of loop, of course, but it has no pre-determined end. It ends when the user wants it to end.

The most common use of this loop type is the main game loop. Programmers typically do all of their initializations (loading graphics, sounds, etc.) and then drop into a While…Wend loop for the rest of the game. Most games allow you

to exit by pressing Escape or some other quick key, which makes this loop type perfect for controlling the action while waiting for the user to quit. Even games that have the *"Are you sure you want to quit?"* box come up, likely use this loop method. But instead of making the exit based on a key press, it's based on a value. Here's an example:

```
; initialize Blitz Basic at 640x480
Graphics 640,480

; Initialize our "GameInProcess" Variable to 1 (TRUE)
iGameInProcess = True

; While iGameInProcess is True
While iGameInProcess = TRUE
        ; clear the screen
       Cls
        ; write out our text
       Text 270,240,"Hello, Blitz Basic!"
        ; wait for 250 milliseconds
       Delay(250)

        ; clear the screen
       Cls
        ; wait for 250 milliseconds
       Delay(250)

       ; if the user hits Escape
       If KeyHit (1)
          Cls
          ; Ask the user if he/she really wants to quit
          QuitAnswer$ = Input$("Really Quit (Y/N)?")

          ; if the user enters a "Y"
          If QuitAnswer$ = "Y" Or QuitAnswer$ = "y"
             ; set our condition flag to False
             iGameInProcess = False
          EndIf ; end of If QuitAnswer$
      EndIf ; end of If KeyHit (1)

Wend  ; end of While Not KeyHit (1)

; end of program
End
```

Also note the use of TRUE and FALSE here. These are two Blitz keywords that return a "1" if a condition is met, or a "0" if not. You may prefer to compare off 0 and 1, but you have the option of TRUE and FALSE as well.

Repeat...Until/Forever

Good news on this one, it's *almost* identical to WHILE…WEND. The only differences are the syntax used and the fact that the loop is guaranteed to process at least once. To quickly help you understand, I will take the last program we used and convert it to the REPEAT…UNTIL format.

```
; initialize Blitz Basic at 640x480
Graphics 640,480

; Initialize our "GameInProcess" Variable to 1 (TRUE)
GameInProcess = True

Repeat
        ; clear the screen
       Cls
        ; write out our text
       Text 270,240,"Hello, Blitz Basic!"
        ; wait for 250 milliseconds
       Delay(250)

        ; clear the screen
       Cls
        ; wait for 250 milliseconds
       Delay(250)

       ; if the user hits Escape
       If KeyHit (1)
          Cls
          ; Ask the user if he/she really wants to quit
          QuitAnswer$ = Input$("Really Quit (Y/N)?")

          ; if the user enters a "Y"
          If QuitAnswer$ = "Y" Or QuitAnswer$ = "y"
             ; set our condition flag to False
             GameInProcess = False
          EndIf ; end of If QuitAnswer$
       EndIf ; end of If KeyHit (1)

; Until GameInProcess is no longer True
Until Not GameInProcess
```

```
; end of program
End
```

Notice that only two code lines changed. "While" was replaced with "Repeat" and "Wend" was replaced with "Until" plus the condition we're checking for.

You can see how the commands in a WHILE…WEND could be bypassed completely if the statement evaluated by WHILE is false. In REPEAT…UNTIL, however, the statement is not evaluated until the end of the loop, so all of the commands will be processed once before the evaluation. The point is that any time you need a set of commands to be processed no less than one time, use the REPEAT…UNTIL loop format over WHILE…WEND.

If you decided to use the REPEAT…FOREVER combination, however, the loop would never stop. This means that you would not need to have an end condition to check for, but you will need to have a way to get out of the loop or the computer will be locked. Getting out of the loop requires the use of the EXIT command. EXIT breaks out of a loop and places the execution at the point directly after the loop.

Chapter 6: Understanding/Using Arrays

When we were talking about the different variable types in chapter 4, we got into a bit of detail with the String type. This is the variable that "strings" characters together to form a word. ARRAYS can be envisioned similarly. As a matter of fact, as you'll soon see, a string *is* an ARRAY!

What Arrays Look Like

In order to define what an array actually looks like, we need to take an example. Let's pretend that we had the names of five players, and we wanted to store them all in memory. We could either set up five individual variables named "sName1$," "sName2$," etc., or we could use an array.

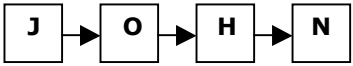
So, we could use the individual strings and have:

```
sName$1 ="John"
sName$2 ="Lorelei"
sName$3 ="Mark"
sName$4 ="Betty"
sName$5 ="George"
```

This format would setup the individual strings and we would have to recall the variable name in full when referencing a particular player. If, however, we used an array we would only need to know the array name and the location in the array of the player. Here is an example of what that would look like:

```
NameArray$(1) = "John"
NameArray$(2) = "Lorelei"
NameArray$(3) = "Mark"
NameArray$(4) = "Betty"
NameArray$(5) = "George"
```

But that's not much different than the string method, is it? Remember what a string looked like in memory? Here's a refresher:



That's exactly what an array looks like too, except that it takes the full piece of data and places it side-by-side, as follows:

So, really, the data inside of the above example is broken down further into arrays. Thus, as strings are "characters strung together," arrays are "data strung together."

Okay, but what's the real benefit? As we move on through the various topics, you'll begin seeing a ton of uses for arrays, but to give an example: Imagine that you have a list of high scores in a file. You have 100 different scores in there and you want to load it up and display it to the user. Well, you can either go line-by-line creating 100 variables, or you can create a single array that has the potential of holding 100 scores. Also, you can easily read each line from the file using a FOR…NEXT loop that keeps track of where you are in your array during assigning and reading of values.

Initializing an Array (the DIM command)

The first thing you need to do when using an array is let Blitz know what type of array you want and how much data it's to contain. The second thing to note is that all arrays are automatically defined as GLOBAL. This means that arrays, regardless of where they are defined in your program, may be manipulated and read by all of your Blitz code.

To initialize an array, we use the DIM command. DIM is short for "dimension," and it refers to the size of the array. Think of it as you would the dimensions of a room. It's just a size indicator.

Keeping with our five-name example, here's how we could define our array:

```
Dim NameArray$(5)
```

That's it. In that one statement, we've told Blitz to reserve enough memory to hold five pieces of data of type string. From here Blitz will carve out a memory chunk for us and get it ready to hold any string data we want to store in there.

As you've already seen, it's easy to add names to our array. We just note the location in the array and assign the value.

```
NameArray$(1) = "John"
NameArray$(2) = "Lorelei"
NameArray$(3) = "Mark"
NameArray$(4) = "Betty"
NameArray$(5) = "George"
```

To print these out we would probably want to use a FOR…NEXT loop because we know the beginning value to start at and we know the ending value as well. It's a defined size, and FOR…NEXT loops are perfect for that scenario.

Here is an example that will print all of the contents of our array out on separate lines. Note the use of the vertical control variable again. This is to ensure that the lines don't overwrite each other.

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; Dimension our NameArray
Dim NameArray$(5)

; Assign our values to the array
NameArray$(1) = "John"
NameArray$(2) = "Lorelei"
NameArray$(3) = "Mark"
NameArray$(4) = "Betty"
NameArray$(5) = "George"

; Loop through the array and print out the values
iTextY% = 0
For iNames = 1 To 5
    Text 0,iTextY,NameArray$(iNames)
    iTextY = iTextY + 16
Next

; wait for a keypress
WaitKey ()

; end the program
End
```

Arrays are not limited to string values, of course. You can also set them up as *integers*, *floats* or *TYPES*. We have not touched on TYPES yet, but will get into much detail on them soon.

You treat integer and float types exactly as you do strings, with the only exception being the definition.

```
Dim NameArray$(5)     ; creates an array of strings
Dim ScoreArray%(5)    ; creates an array of integers
Dim PrecisionArray#(5) ; creates an array of floats
```

Note that you don't have to use the word "array" in our array definition. This is a practice that I sometimes use to keep straight what's what in my coding. The following would work just as effectively.

```
Dim Names$(5)     ; creates an array of strings
Dim Scores%(5)    ; creates an array of integers
Dim Precisions#(5) ; creates an array of floats
```

Multidimensional Arrays

I know that "multidimensional array" sounds like something out of a science fiction novel, but it's really just an array that has more than one dimension. Think of it this way, if someone asked you only for the length of a rectangle, they are asking for a single dimension. If they ask for the length and the width, however, then they are asking for multiple dimensions.

Likewise, arrays can be linear or multidimensional. We've already described a linear array, where everything moves along as *item1->item2->item3* and so on. But in a multidimensional array we would see something that conceptually looks like this:

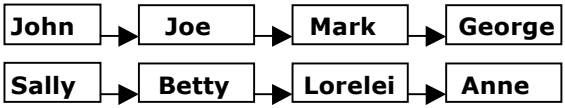
John -> Joe -> Mark -> George
Sally-> Betty -> Lorelei -> Anne
(Figure 6.1)

Here you have seemingly two lists. The first is a list of male names, and the second is a list of female names. Now we could have two separate arrays for this, but there's no need to. We can simply make an array with two dimensions. The first dimension is all the male names, and the second is all the female names.

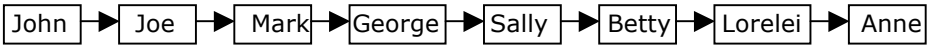
You can also imagine this as rows and columns if that makes it easier. In our example, we have two rows of names and each consists of four columns. Thus, as you would say a room is 9x12 when asked for dimensions, you could say our array is 2x4.

From a non-conceptual point of view, however, this is not how Blitz sees the array in memory. Blitz sees a multidimensional array as just a larger single-dimensioned array. The multidimensional components are for the programmer, not the language. The reason for this is because it's easier for the programmer to keep track of row/column than it is to keep track of a bunch of columns that have a bunch of set-based data.

To the programmer it looks like this:



To Blitz, it looks like this:



Blitz handles the details for you (as do many languages that offer multidimensional array support), so you can have an easier method of wrapping your mind around your data. As your data needs grow with your game development concepts, so too will the complexity of how you piece that data together. Fortunately Blitz is already prepared to help you handle most of these difficulties.

So, how do we declare this type of array? As follows:

```
Dim NameArray$(2,4)
```

To add to that array, we tell Blitz the row and column to place an entry into.

```
NameArray$(1,1) = "John"
NameArray$(2,1) = "Sally"
```

This means that "John" will now sit in row 1, column 1, and that "Sally" will be in row 2, column 1.

Accessing the array is a little more tricky because we'll need to use a nested FOR…NEXT loop. We need to do this because we must first grab all the items from row 1 and then move on to row 2. Here is a program that demonstrates the entire concept. Pay close attention to the FOR…NEXT loops so you can see how we handle the rows and columns individually.

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; Dimension our NameArray
 Dim NameArray$(2,4)

 ; Assign our values to the array
 NameArray$(1,1) = "John"
 NameArray$(1,2) = "Joe"
```

```
NameArray$(1,3) = "Mark"
NameArray$(1,4) = "George"
NameArray$(2,1) = "Sally"
NameArray$(2,2) = "Betty"
NameArray$(2,3) = "Lorelei"
NameArray$(2,4) = "Anne"

; Set up the vertical control variable
iTextY% = 0

; loop from 1 to 2 (rows)
For iNameRow = 1 To 2
   ; loop from 1 to 4 (columns)
   For iNameColumn = 1 To 4
      Text 0,iTextY,NameArray$(iNameRow,iNameColumn)
      iTextY = iTextY + 16
   Next
Next

; wait for a keypress
WaitKey ()

; end the program
End
```

If you type in that program and run it, you'll see all the names listed starting with the first row. Try altering the *iNameRow* loop to print out the female names first (hint: you'll need to use that STEP command!).

You're not limited to two dimensions on your arrays either. If you want to move on to three dimensions, you can do so by declaring your array as follows:

```
Dim NameArray$(5,2,3)
```

The statement creates an array that is 5 elements deep, 2 high, and 3 wide. This is just one way to look at it, you may decide to conceptualize it in different ways that make more sense to you.

Since you already know how to access single-dimensioned arrays and two-dimensioned arrays, you should be able to use that knowledge to figure out how to access the three-dimensional arrays. Take the above code for 2D arrays and play around with it until you get the 3D arrays working properly.

It's not that difficult and it's a good way for you to get used to the dynamic coding issues that arise in game creation.

Re-dimensioning Arrays

You may find it necessary to change the dimension of your array while the program is running. In other words, you don't want the program to stop so you can manually change the dimension of the array, you want the program to change the dimension of the array on its own.

Let's assume you knew you would have five names for ships and three names for animals, and you didn't want to have two arrays to cover the gamut, you would simply do the following:

```
Dim NameArray$(5)
...load in ship name data and print...
Dim NameArray$(3)
...load in animal name data and print...
Dim NameArray$(5)
...load in ship name data and print...
Dim NameArray$(3)
...load in animal name data and print...
```

Yes, I showed those twice to demonstrate that you can go back and forth all you want and Blitz will keep track of array information.

You can also use variables to dynamically control the size of the re-dimensioning, as follows:

```
SizeOfArray = 5
Dim NameArray$(SizeOfArray)
...load in ship name data and print...

SizeOfArray = SizeOfArray - 2

Dim NameArray$(SizeOfArray)
...load in animal name data and print...
```

Loading Data Values into an Array

There is a neat little ability in Blitz that allows you to put all of your data in one location, in a readable format, that you can then "load" from. It's done by using Blitz's DATA statement and its support constructs.

While you can certainly use a disk file to hold all of your data, you may not wish to for various reasons. Maybe you don't want someone tampering with key values that your game needs to run correctly, for example. Depending on the game, I will generally use disk files for most of my processing, but I will rely on DATA statements to help keep some of the more secretive stuff secure. It's not a guarantee of security, mind you, but it's more secure than an opened disk file. And even if both the data values and the file are encrypted, it's still a safer method.

So why use disk files at all? I find disk files easier to deal with and less messy. Small pieces of data in DATA statements are fine, but larger pieces can quickly become confusing because there's so much going on. So if you keep the data to a minimum, it's a great resource.

There are a few commands you'll need to be aware of when using this tool:

- **Data:** This is the command that tells Blitz everything on the line is to be taken as information for later processing.
- **Restore:** Tells Blitz where in the program it should start reading data values. It's based on a label that you create.
- **Read:** This command tells Blitz to read an individual element from the list of data entries.

The following piece of code shows you how to create and populate a data area:

```
.NameData
 Data "John","Joe","Mark","George"
 Data "Sally","Betty","Lorelei","Anne"
 Data "Fido","Spot","Killer","Tank"
 Data "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
```

The first thing to note is:

```
.NameData
```

That line is the label that you will use with the RESTORE command. It's properly formatted by putting a period (".") in front of the name. If you don't have the period in there, Blitz will not know what the intention of the line is and your RESTORE command will not be able to locate the label.

Secondly, our group of DATA statements:

```
 Data "John","Joe","Mark","George"
```

```
Data "Sally","Betty","Lorelei","Anne"
Data "Fido","Spot","Killer","Tank"
Data "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
```

You can imagine this as you would an array. There are four rows of data, each consisting of four columns. So, in essence, we've just drawn a two-dimensional array of names. This is good because we want to read these values into an array anyway, so their formatting makes it easy for us to wrap our minds around.

In order to read these values into our array, we'll need to first RESTORE them and then use the READ command in conjunction with an array. Just like printing out the array values in our last section, code needs to be written that reads the proper value into the proper array location.

```
; Dimension our NameArray
 Dim NameArray$(4,4)

 ; Go to the front of the data lines
 Restore NameData

 ; loop from 1 to 4 (rows)
 For iNameType = 1 To 4
    ; loop from 1 to 4 (columns)
    For iNames = 1 To 4
       Read NameArray$(iNameType,iNames)
    Next
 Next
```

First off, we created an array of 4x4 because we have four rows by four columns. Secondly, we use RESTORE to go to the front of the *NameData* data set. You should note that there is no period (".") at the front of the label in a RESTORE call.

Our next step is to loop through all the rows and columns, using READ as we go to fill in our array. Each call to READ will grab *one* element from the DATA values. The READ command doesn't care if you put all of the elements in your DATA values on one line or on multiple lines. To use the READ command, use the following layout:

```
.NameData
 Data "John","Joe","Mark","George","Sally","Betty","Lorelei", "Anne"
```

…which is the same thing as this:

```
.NameData
 Data "John","Joe","Mark","George"
 Data "Sally","Betty","Lorelei","Anne"
```

The formatting is for the programmer's benefit, not Blitz's. As you can see, though, it's much easier to understand the second list than the first because of the grouping component.

The following piece of code is an altered version of our array printout code. It uses DATA statements to provide the array with the proper values.

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; Dimension our NameArray
Dim NameArray$(4,4)

; read in the data
; Go to the front of the data lines
 Restore NameData

 ; loop from 1 to 4 (rows)
For iNameType = 1 To 4
   ; loop from 1 to 4 (columns)
   For iNames = 1 To 4
      Read NameArray$(iNameType,iNames)
   Next
Next

; print out the data
; Set up the vertical control variable
iTextY% = 0

 ; loop from 1 to 4 (rows)
For iNameType = 1 To 4
   ; loop from 1 to 4 (columns)
   For iNames = 1 To 4
      Text 0,iTextY,NameArray$(iNameType,iNames)
      iTextY = iTextY + 16
   Next
Next
```

```
; wait for a keypress
WaitKey ()

; end the program
End

.NameData
Data "John","Joe","Mark","George"
Data "Sally","Betty","Lorelei","Anne"
Data "Fido","Spot","Killer","Tank"
Data "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
```

So what if you had different types of data that you wanted to read in to different arrays? You would use different labels. Study the following piece of code and note the use of multiple labels.

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; Dimension our NameArray
Dim NameArray$(2,4)

; Dimension our ShipNameArray
Dim ShipNameArray$(4)

; read in the data for NameArray
; Go to the front of the data lines
Restore NameData

; loop from 1 to 2 (rows)
For iNameType = 1 To 2
   ; loop from 1 to 4 (columns)
   For iNames = 1 To 4
      Read NameArray$(iNameType,iNames)
   Next
Next

; read in the data for ShipNameArray
; Go to the front of the data lines
Restore ShipNameData

; loop from 1 to 4 (columns)
For iNames = 1 To 4
```

```
   Read ShipNameArray$(iNames)
 Next

; print out the data
; Set up the vertical control variable
 iTextY% = 0

 ; print out the NameArray first
 ; loop from 1 to 2 (rows)
 For iNameType = 1 To 2
    ; loop from 1 to 4 (columns)
    For iNames = 1 To 4
       Text 0,iTextY,NameArray$(iNameType,iNames)
       iTextY = iTextY + 16
    Next
 Next

 ; now print out the ShipNameArray
 ; loop from 1 to 4 (columns)
 For iNames = 1 To 4
    Text 0,iTextY,ShipNameArray$(iNames)
    iTextY = iTextY + 16
 Next

 ; wait for a keypress
 WaitKey ()

 ; end the program
 End

.NameData
 Data "John","Joe","Mark","George"
 Data "Sally","Betty","Lorelei","Anne"

.ShipNameData
 Data "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
```

When studying that piece of code, pay special attention to the fact that *NameArray* is a two-dimensional array and *ShipNameArray* is singly dimensioned. The purpose of this was to demonstrate the use of the various dimensions when reading in values via DATA commands.

One other thing to note is the "Label" button on the upper right corner of the Blitz IDE. Click on the "Label" button after you've entered the above code and you'll see two items listed in there. One should say "*NameData*" and the

other, "*ShipNameData*." Now, click on either of those terms and the Blitz IDE will take you right to the position of those statements. Pretty cool, huh?

Variable Length Data Statements

In the next chapter we will read in data sets that have varied sizes, and ones that can be changed on the fly without having to hunt through our code making all the related changes. This means that we won't waste time remembering all of the places our arrays can be affected.

For now, however, let's just print out a list of values in a data statement, change it and using the same code base, print them again. The focus here is to change nothing other than the actual data statements.

The first step is to decide on a value that we can use as our closing value. Sticking with our name convention, let's say the final value is simply "STOP." So, when we create our data set, we'll just need to put one line that has the word "STOP" in it, as follows:

```
.NameData
 Data "John","Joe","Mark","George"
 Data "Sally","Betty","Lorelei","Anne"
 Data "Fido","Spot","Killer","Tank"
 Data "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
 Data "STOP"
```

Now, all we need to do is check each value against the word "STOP." If the value is found, then we're all finished! To handle this process, we'll want to call on the WHILE…WEND and IF…THEN…ELSE…ENDIF commands. Here's the example code:

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

; print out the data
 ; Set up the vertical control variable
 iTextY% = 0

 ; set up our boolean value for seeing if we're done or not
 bFinished = False

 ; while we're NOT finished
 While Not bFinished
     Read Name$
     If Name$ = "STOP"
```

```
      bFinished = True
    Else
      Text 0,iTextY,Name$
      iTextY = iTextY + 16
    EndIf
Wend

; wait for a keypress
WaitKey ()

; end the program
End

.NameData
Data "John","Joe","Mark","George"
Data "Sally","Betty","Lorelei","Anne"
Data "Fido","Spot","Killer","Tank"
Data "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
Data "STOP"
```

Play around with this a bit by adding data values. You can put them anywhere you want as long as you end with a "STOP." What happens if you don't have "STOP" as your last item? Blitz will toss up an error saying that it's run out of data to process. This isn't a big deal in your testing, but it will be to the people playing your game, so be careful. Also, you don't need to use the word "STOP." I chose that word because it seemed applicable. You could use "–1" or "Blibbledeeebloob" if you wanted to, Blitz doesn't care.

The last thing I'd like to touch on here is that it doesn't matter to Blitz what type of values you put in. They can be strings, floats, or integers, just like any variables. Let's alter the above example to add a couple of values so you can see the other types in action. Do note that you'll need to read in the non-strings using the appropriate variable type to get the results you want.

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; print out the data
; Set up the vertical control variable
iTextY% = 0

; set up our boolean value for seeing if we're done or not
bFinished = False
```

```
; while we're NOT finished
While Not bFinished
    Read Name$
    If Name$ = "STOP"
      bFinished = True
    Else
      Read Age%
      Read SomePercent#
      Text 0,iTextY,Name$ + ", " + Age% + ", " + SomePercent#
      iTextY = iTextY + 16
    EndIf
Wend

; wait for a keypress
WaitKey ()

; end the program
End

.NameData
 Data "John",33,97.75
 Data "Sally",44,99.375
 Data "Fido",7,47.125
 Data "STOP"
```

Here we have the first value in each data statement as a string. This is important because we're using a string ("STOP") as our end value. So we need to make sure we check on that before reading in additional values, otherwise we'll get that Blitz has run out of data.

Before moving on you've probably noticed that the good old TEXT started looking rather odd there. Here's the code:

```
Text 0,iTextY,Name$ + ", " + Age% + ", " + GPAPercent#
```

What's with all the plus (+) signs and quotes? In order to display variable values, we need to inform Blitz that's what we're looking to do. If we enclosed the variables inside the quotes, Blitz would print them out literally. For example, the following code…

```
Text 0,iTextY,"Name$, Age%, GPAPercent#"
```

Would print out:

Name$, Age%, SomePercent#

Exactly like that. Now, that's not what we want; we want the actual values contained within the variables. So we need to somehow put those variables *outside* the quotes. To handle this, Blitz is configured to treat anything with a plus sign that is outside of quotes as a variable. The things in the quotes are literal text values that Blitz will print. To further this example, replace that line in the program with the following line:

```
Text 0,iTextY,"Name: " + Name$+ ", Age: " + Age% + ", ↲
    → Percent: " + GPAPercent#
```

Hopefully this is all starting to come together for you. In the next chapter we'll be learning about another powerful data construct called a TYPE and we'll touch again on how to load variable length values using the DATA commands.

Chapter 7: Understanding/Using Types

We'll often work with various data sets containing a bunch of related items, but are all different types. Taking our previous example of getting an individual's personal information, let's say we wanted to know the name, age and grade point average (GPA) of a person. The name is a string, the age is an integer, and the GPA is a float.

While we could use an array for that, the data can become more confusing as the list of info we want on each person grows. Using a TYPE, however, gives us a more dynamic tool for building data sets with varied information. This is key because game data has to be dynamic! Another key point is that arrays take chunks of memory whether they use them or not. TYPES only use what's needed and nothing more.

So what does a TYPE look like? Here's a little snippet of code that defines our personal information values:

```
Type PersonalInfo
   Field Name$          ; name of the person
   Field Age%           ; age of the person
   Field GPAPercent#    ; Grade Point Average of the person
End Type
```

The first line defines the name of the TYPE, which in this case is *PersonalInfo*. Then we have a group of fields that build the actual variables in the TYPE. Finally, we have to let Blitz know that we are done configuring the TYPE, so we place the command END TYPE.

Note that we don't assign any values during the building of our TYPE. This is because our format is merely a blueprint for the data that can be held by *PersonalInfo*. To actually store data, we need to create an "instance" of *PersonalInfo*.

```
People.PersonalInfo = New PersonalInfo
```

The above line creates an "instance" of *PersonalInfo*. Specifically used for this is the NEW command, which tells Blitz to create a "new instance" of our TYPE. All that "instance" means is that there is now a valid piece of memory set aside that we can store our data in. Think of it as a standard house blueprint. You build a house from the blueprint and you have a *real* instance of the blueprint. At the very beginning of that line you'll see that I have an identifier

called "People." This is the "instance identifier" I am using for this example. You can call it whatever you want, as long as it's not a Blitz keyword. Just make sure that you follow it with a period and then the TYPE name. You'll need this identifier when you reference the values for this TYPE, so make it something applicable to your situation.

But there is something very important we need to think about here. What if Blitz was unable to build this instance? What if there wasn't enough memory available, for example? If there isn't enough memory or there was some error in securing an instance of a type, Blitz will return a NULL (empty value). We can check for that with an IF statement, as follows:

```
; attempt to create an instance of PersonalInfo
People.PersonalInfo = New PersonalInfo

; if the attempt returned a non-Null, the process
If People.PersonalInfo <> Null
   …process our data…
; else, fail.
Else
   …inform user of error…
EndIf
```

This type of checking cannot be stressed enough because you may well run into instances where users are crashing and you can't figure out why. If you have this type of checking in place, you'll be much safer in pinpointing memory allocation errors.

After we have built and allocated our TYPE, we then need to populate the values. For a simple example, I will do this by hand.

```
People\Name$ = "John"
People\Age% = 33
People\GPAPercent# = 97.75
```

The identifier is the first piece of the line in an assignment or access of the type's instance. In this case, that identifier is *People*. The second element is the backslash ("\"). This symbol tells Blitz that you want to access the Field contained in the TYPE that follows the "\" symbol. So taking our first line, we are instructing Blitz to access the *Name$* field in the TYPE identified by *People*. Then we assign the value "John" to that.

To access these instances for print, we run through the list until we hit the end, printing along the way.

```
; while we're NOT finished
 For People.PersonalInfo = Each PersonalInfo
    Text 0,iTextY,People\Name$+ ", " + People\Age% + ↲
         → ", "  + People\GPAPercent#
 Next
```

The FOR…NEXT loop used here looks a bit different than what we've studied previously. This loop has a specific purpose in dealing with types. The reason for this is that a TYPE can have an indeterminate ending point. So the developer of Blitz decided to make a loop type that could track each instance of a TYPE while helping the developer avoid the trickiness of dealing with this issue directly. In a nutshell, this FOR…NEXT loop goes through EACH of the instances assigned to the TYPE until there are no more instances.

Printing the values in each instance requires that we use the *identifier\field* components again. Looking at the source you'll see where we used *People\Name$*. This will access the value in the variable *Name$* and display that value.

Finally, we will want to release the memory that was gobbled up when we created the instance. To do this we DELETE each instance. Just as we printed each instance, use a FOR…NEXT loop to run through EACH instance to DELETE. Here's the snippet of code:

```
For People.PersonalInfo = Each PersonalInfo
   Delete People
Next
```

Now, here is the piece in its entirety. Enter this in and run it so you can see how easy types are to work with.

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a data Type
 Type PersonalInfo
    Field Name$          ; name of the person
    Field Age%           ; age of the person
    Field GPAPercent#  ; Grade Point Average of the person
 End Type

 ; add an entry
 People.PersonalInfo = New PersonalInfo
```

```
If People.PersonalInfo <> Null
  People\Name$ = "John"
  People\Age% = 33
  People\GPAPercent# = 97.75
EndIf

; Set up the vertical control variable
iTextY% = 0

; Step through Each instance of the type and print
For People.PersonalInfo = Each PersonalInfo
   Text 0,iTextY,People\Name$+ ", " + People\Age% + ↲
        → ", " + People\GPAPercent#
Next

; Step through Each instance of the type and delete
For People.PersonalInfo = Each PersonalInfo
   Delete People
Next

; wait for a keypress
WaitKey ()

; end the program
End
```

So, why do we need an identifier anyway? Well, we may have different types of information that we're tracking that require the same values. Let's use an example of space ships. You have your good guy ship and all of your bad guy ships, plus you've got freighters and destroyers. They all have a name, laser power, shield power, armor, missile compliment, speed, etc. So why would you want to create a different TYPE for each one? Just use one and use different identifiers.

```
Type Ships
      Field Name$          ; name of this ship
      Field LaserPower%  ; 1-20 points per hit
      Field Armor%         ; 100-150 points
      Field ShieldPower% ; 150-300 points-adds to Armor
      Field Missiles%       ; 5-50 depending on ship type
      Field TopSpeed#      ; 2.00–3.50 based on ship type
End Type
```

Now, instead of having to create three (or many more depending on the number of ship types you have) complete TYPE constructs, you can use three separate identifiers.

```
PlayerFighter.Ships = New Ships
BadguyFighter.Ships = New Ships
Freighter.Ships = New Ships
Destroyer.Ships = New Ships
```

Each one can now be accessed directly and assigned values that will only be applicable to the identifier that they are associated with. That saves some major headache, believe me!

Loading Data Statements into Types

Now you may be thinking that this would be a great way to handle using variable length DATA statements. Well, you're right!
Since you're already familiar with the DATA statements, I'm not going to delve into another description of those. Instead, I'm going to give you a piece of code that will print out a bunch of ship specifications from a data set of values.

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a data Type
Type Ships
        Field Name$           ; name of this ship
        Field LaserPower%  ; 1-20 points per hit
        Field Armor%          ; 100-150 points
        Field ShieldPower% ; 150-300 points adds to Armor
        Field Missiles%        ; 5-50 depending on ship type
        Field TopSpeed#       ; 2.00–3.50 based on ship type
End Type

 ; go to the ShipData section
 Restore ShipData

 ; set our data reading flag to true
 bData = True

 ; while we still have data (not "STOP")
 While bData
    ; read the first data element of the line
    Read Name$
```

```
   ; if it = "STOP" then we are finished
   If Name$ = "STOP"
     bData = False
   Else
     ; read the rest of the data
     Read LaserPower%
     Read Armor%
     Read ShieldPower%
     Read Missiles%
     Read TopSpeed#

     ; add an entry
     Fighter.Ships = New Ships
     If Fighter.Ships <> Null
        Fighter\Name$ = Name$
        Fighter\LaserPower% = LaserPower%
        Fighter\Armor% = Armor%
        Fighter\ShieldPower% = ShieldPower%
        Fighter\Missiles% = Missiles%
        Fighter\TopSpeed# = TopSpeed#
     Else
        Text 0,0,"ERROR: Could not create an instance of ↲
              → PersonalInfo!"
       bData = False
     EndIf
   EndIf
Wend

; Set up the vertical control variable
iTextY% = 0

; Step through Each instance of the type and print
For Fighter.Ships = Each Ships
   Text 0,iTextY,"Ship Name: " + Fighter\Name$
   iTextY% = iTextY + 16
   Text 0,iTextY,"Laser Power: " + Fighter\LaserPower%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Armor: " + Fighter\Armor%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Shield Power: " + Fighter\ShieldPower%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Missiles: " + Fighter\Missiles%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Top Speed: " + Fighter\TopSpeed#
   iTextY% = iTextY + 32
```

```
Next

; Step through Each instance of the type and delete
For Fighter.Ships = Each Ships
   Delete Fighter
Next

; wait for a keypress
WaitKey ()

; end the program
End

.ShipData
Data "Kliazian Raptor",15,125,200,25,2.50
Data "Weltic Cruiser",20,150,220,45,2.00
Data "STOP"
```

I know that's a lot of code, but nothing that should be a surprise to you at this point. Just give it a good study, play with it a bit and you'll be an expert at it in no time. Add a few more lines of ships to see the flexibility that TYPE gives you. Note that, unlike the array method, you don't need to change anything but the actual data set.

To help you really get this concept down, try to add another data set of Freighters. Remember that you'll need a new label for the DATA command, you'll need to RESTORE it, and you'll need to create a new *identifier* for the *Ships* TYPE. This also means that you'll need to setup another FOR…NEXT loop to handle the new instances.

Arrays within Types

You can also create an array within a TYPE, but it is not configured the same as a stand-alone array. Instead of using the DIM command, you would create a field and place a set of brackets ([]) with a dimension value inside. Here's an example of what I'm describing:

```
Type Ships
      Field Name$          ; name of this ship
      Field LaserPower%  ; 1-20 points per hit
      Field Armor%         ; 100-150 points
      Field ShieldPower% ; 150-300 points adds to Armor
      Field Missiles%[2]   ; 5-50 depending on ship type
      Field TopSpeed#      ; 2.00–3.50 based on ship type
End Type
```

Note the field *Missiles* to see the format. The *Missiles* field is now a single dimensioned array. Adding to this Field is done exactly as you assign any value to an array field. The only exception is that you have to use the *identifier* and the backslash ("\") combination.

```
Fighter\Missiles%[1] = 10
Fighter\Missiles%[2] = 20
```

So now the first element of the array contains "10" and the second element contains "20."

Printing the values is equally simple.

```
Text 0,iTextY,"Missiles: " + Fighter\Missiles%[1]
Text 0,iTextY,"Secondary: " + Fighter\Missiles%[2]
```

Again, you just reference the Array position that you want to print from.

You do not need to do anything special when using the DELETE *identifier* to free up memory, Blitz will handle the Array clearing for you.

Array of Types

Another cool feature of Blitz is the ability to have an array of TYPE. You dimension an array as you normally would, but instead of using a variable descriptor, you would use a TYPE descriptor.

```
; setup a data Type
Type Ships
	Field Name$		; name of this ship
	Field LaserPower%	; 1-20 points per hit
	Field Armor%		; 100-150 points
	Field ShieldPower%	; 150-300 points adds to Armor
	Field Missiles%[2]	; 5-50 depending on ship type
	Field TopSpeed#		; 2.00–3.50 based on ship type
End Type

; Dimension our array of type Ships
Dim Fighter.Ships(2)
```

You now have a single dimensioned array of the TYPE *Ships*. You will still need to create individual instances of *Ship*, but you can now access them in array

fashion. To create the instances, you will need to use the array style in conjunction with the TYPE style.

```
Fighter.Ships(1) = New Ships
Fighter.Ships(2) = New Ships
```

As you can see, the array position is still required, but so is the use of the NEW command. You can use a FOR…NEXT loop in array fashion to access the individual elements.

Here is our previous code example using the array of TYPES method:

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a data Type
  Type Ships
        Field Name$           ; name of this ship
        Field LaserPower%  ; 1-20 points per hit
        Field Armor%         ; 100-150 points
        Field ShieldPower% ; 150-300 points adds to Armor
        Field Missiles%[2]    ; 5-50 depending on ship type
        Field TopSpeed#       ; 2.00–3.50 based on ship type
  End Type

 Dim Fighter.Ships(2)

 ; go to the ShipData section
 Restore ShipData

; use a standard array looping style
 For i = 1 To 2
    ; read the data
    Read Name$
    Read LaserPower%
    Read Armor%
    Read ShieldPower%
    Read Missiles%
    Read SecondaryMissiles%
    Read TopSpeed#

    Fighter.Ships(i) = New Ships
    Fighter.Ships(i)\Name$ = Name$
    Fighter.Ships(i)\LaserPower% = LaserPower%
```

```
   Fighter.Ships(i)\Armor% = Armor%
   Fighter.Ships(i)\ShieldPower% = ShieldPower%
   Fighter.Ships(i)\Missiles%[1] = Missiles%
   Fighter.Ships(i)\Missiles%[2] = SecondaryMissiles%
   Fighter.Ships(i)\TopSpeed# = TopSpeed#
Next

; Set up the vertical control variable
iTextY% = 0

; Step through the ships array and print
For i = 1 To 2
   Text 0,iTextY,"Ship Name: " + Fighter.Ships(i)\Name$
   iTextY% = iTextY + 16
   Text 0,iTextY,"Laser Power: " + Fighter.Ships(i)\LaserPower%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Armor: " + Fighter.Ships(i)\Armor%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Shield Power: " + Fighter.Ships(i)\ShieldPower%
   iTextY% = iTextY + 16
   Text 0,iTextY,"Missiles: " + Fighter.Ships(i)\Missiles%[1]
   iTextY% = iTextY + 16
   Text 0,iTextY,"Secondary: " + Fighter.Ships(i)\Missiles%[2]
   iTextY% = iTextY + 16
   Text 0,iTextY,"Top Speed: " + Fighter.Ships(i)\TopSpeed#
   iTextY% = iTextY + 32
Next

; wait for a keypress
WaitKey ()

; end the program
End

.ShipData
Data "Kliazian Raptor",15,125,200,25,12,2.50
Data "Weltic Cruiser",20,150,220,45,27,2.00
```

You can immediately see that we're no longer using a WHILE…WEND loop, and we're not checking for a value of "STOP" anywhere. No need to in this instance because, as you are already aware, arrays are of fixed length. I also made sure to keep the array within the TYPE intact, so you could see that it would have no additional impact. Thus, what we have is an ARRAY of TYPE that contains an array. How's that for scary?

Types within Types

Another tricky, but useful ability, is to have a TYPE embedded in another TYPE. This can benefit you in a number of ways, but the one way I find most compelling is called *inheritance*.

Think of our *Ship* TYPE. While it's certainly good that we can have a different *identifier* for our fighters, freighters, and destroyers, because they all share some components, a destroyer will likely have things that fighters and freighters don't. Now we can certainly just create a TYPE for each different ship class, but since we know that all ships have propulsion, weapons, armor, shielding, etc., it's nice to have a basic set that they share. But where a destroyer may have landing bays, a fighter doesn't, so we'll also need to have the ability to expand the destroyer's information above the basic set.

So what we really want to do is *inherit* the basic set of ship components and then add to it. To do this, we'll need to use a TYPE within a TYPE. Here's the definition of our *Ships* TYPE.

```
Type Ships
      Field Name$           ; name of this ship
      Field LaserPower%     ; 1-20 points per hit
      Field Armor%          ; 100-150 points
      Field ShieldPower%    ;150-300 points adds to Armor
      Field Missiles%       ; 5-50 depending on ship type
      Field TopSpeed#       ; 2.00–3.50 based on ship type
 End Type
```

Now, let's *inherit* that basic set and extend it with our *Destroyer* TYPE.

```
Type Destroyers
    Field ShipBasics.Ships  ; take all basic ship information
    Field LandingBays%      ; add number of landing bays
    Field FighterCompliment%   ; add fighter ships onboard
    Field IonCannonPower%  ; add power of the Ion Cannon
 End Type
```

If you look at the first Field, you'll notice that we have an *identifier* called *ShipBasics* that references the TYPE *Ships*. This essentially translates into the following (from a conceptual point of view):

```
Type Destroyers
    Field Name$           ;name of this ship
    Field LaserPower%     ; 1-20 points per hit
    Field Armor%          ; 100-150 points
```

```
    Field ShieldPower%    ; 150-300 points adds to Armor
    Field Missiles%       ; 5-50 depending on ship type
    Field TopSpeed#       ; 2.00–3.50 based on ship type
    Field LandingBays%    ; add number of landing bays
    Field FighterCompliment% ; add fighter ships on board
    Field IonCannonPower%  ; add power of the Ion Cannon
End Type
```

The primary difference is that any changes we make to the *Ships* TYPE will automatically be incorporated into the *Destroyers* TYPE at the time of compilation. Otherwise we would have to go through each ship class and add/delete/modify the entries directly. No big deal if you have 2 classes, but it gets tougher if you add tons more. And remember that this can be applied to anything. As you add in weapon types, robots, shield types, engine types, planets, moons, space stations, etc., you'll find this concept more and more useful.

We now essentially have a TYPE, called *Destroyers*, that has three elements that other ship classes don't necessarily have. So when you create *instances* of *Destroyers*, you will have specific information to deal with. *Fighters*, however, won't have landing bays or compliment issues to worry about.

If you recall from the last section, we need to be concerned with asking Blitz to reserve memory for each of our TYPE *instances*. In this case we'll need to reserve not only the *Destroyers* TYPE, but also the *inherited Ships* TYPE.

```
Destroyer.Destroyers = New Destroyers
If Destroyer.Destroyers <> Null
   Destroyer\ShipBasics.Ships = New Ships
   If Destroyer\ShipBasics.Ships <> Null
     ...process commands...
   Else
     ...notify of error...
   EndIf
Else
     ...notify of error...
EndIf
```

The first thing we do is *call* the NEW command to reserve an *instance* of *Destroyers*. If the return was not NULL, then we attempt to reserve an *instance* of *Ships*. If the return is not NULL, then we begin processing. Otherwise, we have an error and we need to inform the user.

A key thing to catch here is the format that we use when accessing the *ShipBasics.Ships identifier*. Since it is part of *Destroyers*, we use the format:

Primary_Identifier\Inherited_Identifier.Type

This is the case when creating the *instance*. When accessing/modifying the individual elements, we don't need to specify the *inherited* TYPE. This same principal goes for printing out the values. Here is a line of code that demonstrates the access/modification of a TYPE within a TYPE**.**

```
Destroyer\ShipBasics\Name$ = Name$
```

So, here we use the format:

Primary_Identifier\Inherited_Identifier\Element_Name = Value

The *Destroyers* specific elements (non-inherited) are accessed normally, as follows:

```
Destroyer\LandingBays% = LandingBays%
```

Finally, we'll want to free up the memory allocated by these *instances*, so we'll use DELETE in conjunction with the FOR…NEXT loop. It is important that you release the memory for the *inherited* TYPE first. If you release the *primary* first, Blitz will no longer have a link to the *inherited* TYPE.

```
For Destroyer.Destroyers = Each Destroyers
     Delete Destroyer\ShipBasics.Ships
     Delete Destroyer
Next
```

The following example shows this process in action. It's recommended that you study this code thoroughly and then begin to make adjustments. Playing around with the examples will help you to become more comfortable with the various concepts and their implementation. Here's the code in its full form:

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a Ships data Type
 Type Ships
    Field Name$           ; name of this ship
    Field LaserPower%   ; 1-20 points per hit
```

```
    Field Armor%          ; 100-150 points
    Field ShieldPower%  ; 150-300 points adds to Armor
    Field Missiles%       ; 5-50 depending on ship type
    Field TopSpeed#       ; 2.00-3.50 based on ship type
End Type

; setup a Destroyers data Type
Type Destroyers
    Field ShipBasics.Ships  ;Inherit basic ship info
    Field LandingBays%     ; add number of landing bays
    Field FighterCompliment%  ; add fighter ships onboard
    Field IonCannonPower%     ; power of the Ion Cannon
End Type

; go to the ShipData section
Restore ShipData

; set our data reading flag to true
bData = True

; while we still have data (not "STOP")
While bData
   ; read the data
   Read Name$
   If Name$ <> "STOP"
      Read LaserPower%
      Read Armor%
      Read ShieldPower%
      Read Missiles%
      Read SecondaryMissiles%
      Read TopSpeed#
      Read LandingBays%
      Read FighterCompliment%
      Read IonCannonPower%

      ; create an instance of Destroyers
     Destroyer.Destroyers = New Destroyers
      ; if successful
      If Destroyer.Destroyers <> Null
        ; create an instance of Ships
        Destroyer\ShipBasics.Ships = New Ships
        ; if successful, add to the instance
        If Destroyer\ShipBasics.Ships <> Null
           Destroyer\ShipBasics\Name$ = Name$
           Destroyer\ShipBasics\LaserPower% = LaserPower%
```

```
            Destroyer\ShipBasics\Armor% = Armor%
            Destroyer\ShipBasics\ShieldPower% =ShieldPower%
            Destroyer\ShipBasics\Missiles% = Missiles%
            Destroyer\ShipBasics\TopSpeed# = TopSpeed#
            Destroyer\LandingBays% = LandingBays%
            Destroyer\FighterCompliment% = FighterCompliment%
            Destroyer\IonCannonPower% = IonCannonPower%
          Else
            Text 0,0,"Error: Could not create the Ships instance!"
            bData = False
          EndIf
        Else
          Text 0,0,"Error: Could not create the Destroyer instance!"
          bData = False
        EndIf
      Else
        bData = False
      EndIf
    Wend

    ; Set up the vertical control variable
    iTextY% = 0

    ; Step through and print
    For Fighter.Destroyers = Each Destroyers
      Text 0,iTextY,"Ship Name: " + Destroyer\ShipBasics\Name$
      iTextY% = iTextY + 16
      Text 0,iTextY,"Laser Power: " + ↲
          → Destroyer\ShipBasics\LaserPower%
      iTextY% = iTextY + 16
      Text 0,iTextY,"Armor: " + Destroyer\ShipBasics\Armor%
      iTextY% = iTextY + 16
      Text 0,iTextY,"Shield Power: " + ↲
          → Destroyer\ShipBasics\ShieldPower%
      iTextY% = iTextY + 16
      Text 0,iTextY,"Missiles: " + Destroyer\ShipBasics\Missiles%
      iTextY% = iTextY + 16
      Text 0,iTextY,"Top Speed: " + Destroyer\ShipBasics\TopSpeed#
      iTextY% = iTextY + 16
      Text 0,iTextY,"Landing Bays: " + Destroyer\LandingBays%
      iTextY% = iTextY + 16
      Text 0,iTextY,"Fighter Compliment: " + ↲
          → Destroyer\FighterCompliment%
      iTextY% = iTextY + 16
      Text 0,iTextY,"Ion Cannon Power: " + ↲
```

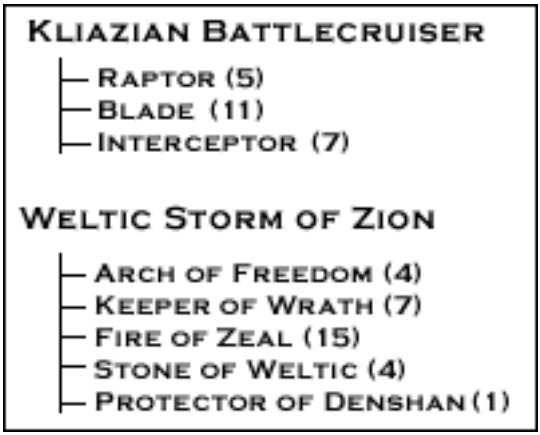
```
        → Destroyer\IonCannonPower%
    iTextY% = iTextY + 32
Next

; delete the instances
For Destroyer.Destroyers = Each Destroyers
    Delete Destroyer\ShipBasics.Ships
    Delete Destroyer
Next

; wait for a keypress
WaitKey ()

; end the program
End

.ShipData
Data "Kliazian Raptor",15,125,200,25,12,2.50,20,100,500
Data "Weltic Cruiser",20,150,220,45,27,2.00,25,200,750
Data "STOP"
```

Parent-Child Data Lists

Another great benefit of using TYPES, is the ability to tie a bunch of data into one data point.

Suppose you have created a Destroyer that has a bunch of Fighters aboard. There are various classes of fighters, each with differing weapons and so on. You'll want to be able to keep track of them all, so you'll need a way to have one Destroyer—the *parent*—and multiple Fighters—known as *children*—all together in one list that can be accessed and manipulated.

We already have had a single *instance* of the TYPE within a TYPE, but this will require multiple *instances* of a secondary (*child*) TYPE within a primary (*parent*) TYPE.



(Figure 7.1)

Figure 7.1 shows that the two destroyers have different compliments of fighters. Each *parent* ("Kliazian Battlecruiser" and "Weltic Storm of Zion") has a number of fighters. But there are different amounts in each, and all are of differing classes.

This is important to understand because you may have a game that allows the player to own a destroyer, and maybe they can fill it with small fighters to a certain compliment. But if the player doesn't have enough money to carry the full compliment, he'll still want to be able to carry a subset. Additionally, the player may not be able to afford all the top-class fighters, so you'll need to have the ability to house a variety of classes of fighters.

The code for doing this may look a little daunting at first, but it's simply another set of blocks on the wall we've been building. Before showing the entire source, let's talk about the primary pieces.

```
; setup a Ships data Type
 Type Ships
    Field Name$           ; name of this ship
    Field LaserPower%   ; 1-20 points per hit
    Field Armor%          ; 100-150 points
    Field ShieldPower%  ;150-300 points adds to Armor
    Field Missiles%       ; 5-50 depending on ship type
    Field TopSpeed#       ; 2.00–3.50 based on ship type
 End Type

 ; setup the Fighters data type
 Type Fighters
```

```
    Field ParentID%         ; add in the Parent ID
    Field ShipBasics.Ships  ; Inherit basic ship info
End Type

; setup a Destroyers data Type
Type Destroyers
    Field ID%               ; add in an ID for this Destroyer
    Field ShipBasics.Ships  ; Inherit basic ship info
    Field LandingBays%      ; add number of landing bays
    Field FighterCompliment%  ; add fighter ships onboard
    Field IonCannonPower%     ; power of the Ion Cannon
End Type
```

The primary differences between this TYPE setup area and our previous code examples are the addition of the *Fighters* TYPE, and the addition of a new Field. *Destroyers* now has a Field called ID, which allows us to keep a unique value for each *instance* created. *Fighter* contains a Field called *ParentID* which will correspond to the *Destroyers* with which it belongs. Any *Fighters* with a *ParentID* of 1, for example, would be "owned" by the *Destroyers* with the *ID* of 1.

Where we have to load *Fighters* separately from *Destroyers*, the *only* difference in loading the actual values is that each has to have a corresponding *ID* value. Other than that, the process is identical.

For the *Destroyers*:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; READ IN DESTROYER DATA
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; go to the DestroyerData section
Restore DestroyerData

; set our data reading flag to true
bData = True

; while we still have data (not "STOP")
While bData
   ; read the data
   Read Name$
   If Name$ <> "STOP"
     Read ID%
     Read LaserPower%
```

```
      Read Armor%
      Read ShieldPower%
      Read Missiles%
      Read SecondaryMissiles%
      Read TopSpeed#
      Read LandingBays%
      Read FighterCompliment%
      Read IonCannonPower%

      ; create an instance of Destroyers
      Destroyer.Destroyers = New Destroyers
      ; if successful
      If Destroyer.Destroyers <> Null
        ; create an instance of Ships
        Destroyer\ShipBasics.Ships = New Ships
        ; if successful, add to the instance
        If Destroyer\ShipBasics.Ships <> Null
          Destroyer\ID% = ID%
          Destroyer\ShipBasics\Name$ = Name$
          Destroyer\ShipBasics\LaserPower% = LaserPower%
          Destroyer\ShipBasics\Armor% = Armor%
          Destroyer\ShipBasics\ShieldPower% =ShieldPower%
          Destroyer\ShipBasics\Missiles% = Missiles%
          Destroyer\ShipBasics\TopSpeed# = TopSpeed#
          Destroyer\LandingBays% = LandingBays%
          Destroyer\FighterCompliment% = FighterCompliment%
          Destroyer\IonCannonPower% = IonCannonPower%
        Else
          Text 0,0,"Error: Could not create the Ships instance!"
          bData = False
        EndIf
      Else
        Text 0,0,"Error: Could not create the Destroyer instance!"
        bData = False
      EndIf
    Else
      bData = False
    EndIf
  Wend
```

And for our *Fighters*:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; READ IN FIGHTER DATA
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; go to the DestroyerData section
Restore FighterData

; set our data reading flag to true
bData = True

; while we still have data (not "STOP")
While bData
   ; read the data
   Read Name$
   If Name$ <> "STOP"
      Read ParentID%
      Read LaserPower%
      Read Armor%
      Read ShieldPower%
      Read Missiles%
      Read SecondaryMissiles%
      Read TopSpeed#

      ; create an instance of Fighters
      Fighter.Fighters = New Fighters
      ; if successful
      If Fighter.Fighters <> Null
         ; create an instance of Ships
         Fighter\ShipBasics.Ships = New Ships
         ; if successful, add to the instance
         If Fighter\ShipBasics.Ships <> Null
            Fighter\ParentID% = ParentID%
            Fighter\ShipBasics\Name$ = Name$
            Fighter\ShipBasics\LaserPower% = LaserPower%
            Fighter\ShipBasics\Armor% = Armor%
            Fighter\ShipBasics\ShieldPower% = ShieldPower%
            Fighter\ShipBasics\Missiles% = Missiles%
            Fighter\ShipBasics\TopSpeed# = TopSpeed#
         Else
            Text 0,0,"Error: Could not create the Ships instance!"
            bData = False
         EndIf
      Else
         Text 0,0,"Error: Could not create the Destroyer instance!"
         bData = False
      EndIf
   Else
```

```
      bData = False
    EndIf
 Wend
```

Notice that the actual loading process has not changed, but only the elements that must be loaded.

To print out the values will require that we use a nested FOR…NEXT loop. Also, during that loop, we'll need to compare the *Destroyer\ID* to the *Fighter\ParentID* each iteration. This is to see if the *Fighter* belongs to the *Destroyer*.

```
; Set up the vertical control variable
 iTextY% = 0

 ; Step through and print
 For Destroyer.Destroyers = Each Destroyers
    Text 0,iTextY,"Ship Name: " + ↲
         → Destroyer\ShipBasics\Name$
    iTextY% = iTextY + 16

    ; now loop through each of the fighters and print out
    ; all of the ones belonging to this destroyer
    For Fighter.Fighters = Each Fighters
       If Fighter\ParentID% = Destroyer\ID%
         Text 50,iTextY,"Fighter Name: " + Fighter\ShipBasics\Name$
         iTextY% = iTextY + 16
       EndIf
    Next

    iTextY% = iTextY + 32
Next
```

In the full code I also put in a WAITKEY statement and an IF…THEN…ELSE…ENDIF, simply to make the text better formatted and visible. It has no direct bearing on the TYPES themselves, but I didn't want you to be confused at the additional code that will be in this section in the full listing.

To delete all the TYPES, we first DELETE the *Fighters* and then the *Destroyers*. You don't have to do it this way since the two are *only* related by a Field element (there is not direct memory tie between them). But it's a good idea to pretend there is a direct connection because you'll be creating and releasing *Destroyers* and *Fighters* on the fly and you don't want to have

Fighters out there taking up memory that don't have a corresponding *Destroyer* around.

```
; delete the instances
 For Destroyer.Destroyers = Each Destroyers
    ; now loop through each of the fighters and delete
    ; all of the ones belonging to this destroyer
    For Fighter.Fighters = Each Fighters
       If Fighter\ParentID% = Destroyer\ID%
          Delete Fighter\ShipBasics.Ships
          Delete Fighter
       EndIf
    Next

    Delete Destroyer\ShipBasics.Ships
    Delete Destroyer
 Next
```

And that's all there is to it. Now, here's the entire code listing. You should again enter this in and play around with the values to gain familiarity with these concepts.

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a Ships data Type
 Type Ships
     Field Name$           ; name of this ship
     Field LaserPower%     ; 1-20 points per hit
     Field Armor%          ; 100-150 points
     Field ShieldPower%    ; 150-300 points adds to Armor
     Field Missiles%       ; 5-50 depending on ship type
     Field TopSpeed#       ; 2.00–3.50 based on ship type
 End Type

 ; setup the Fighters data type
 Type Fighters
     Field ParentID%         ; add in the Parent ID
     Field ShipBasics.Ships  ; Inherit basic ship info
 End Type

 ; setup a Destroyers data Type
 Type Destroyers
     Field ID%               ; add in an ID for this Destroyer
```

```
    Field ShipBasics.Ships   ; Inherit basic ship info
    Field LandingBays%       ; add number of landing bays
    Field FighterCompliment% ;add fighter ships onboard
    Field IonCannonPower%    ; power of the Ion Cannon
End Type

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; READ IN DESTROYER DATA
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; go to the DestroyerData section
Restore DestroyerData

; set our data reading flag to true
bData = True

; while we still have data (not "STOP")
While bData
   ; read the data
   Read Name$
   If Name$ <> "STOP"
      Read ID%
      Read LaserPower%
      Read Armor%
      Read ShieldPower%
      Read Missiles%
      Read SecondaryMissiles%
      Read TopSpeed#
      Read LandingBays%
      Read FighterCompliment%
      Read IonCannonPower%

      ; create an instance of Destroyers
      Destroyer.Destroyers = New Destroyers
      ; if successful
      If Destroyer.Destroyers <> Null
         ; create an instance of Ships
         Destroyer\ShipBasics.Ships = New Ships
         ; if successful, add to the instance
         If Destroyer\ShipBasics.Ships <> Null
            Destroyer\ID% = ID%
            Destroyer\ShipBasics\Name$ = Name$
            Destroyer\ShipBasics\LaserPower% = LaserPower%
            Destroyer\ShipBasics\Armor% = Armor%
            Destroyer\ShipBasics\ShieldPower% = ShieldPower%
```

```
        Destroyer\ShipBasics\Missiles% = Missiles%
        Destroyer\ShipBasics\TopSpeed# = TopSpeed#
        Destroyer\LandingBays% = LandingBays%
        Destroyer\FighterCompliment% = FighterCompliment%
        Destroyer\IonCannonPower% = IonCannonPower%
      Else
        Text 0,0,"Error: Could not create the Ships instance!"
        bData = False
      EndIf
    Else
      Text 0,0,"Error: Could not create the Destroyer instance!"
      bData = False
    EndIf
  Else
    bData = False
  EndIf

Wend

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; READ IN FIGHTER DATA
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; go to the DestroyerData section
Restore FighterData

; set our data reading flag to true
bData = True

; while we still have data (not "STOP")
While bData
  ; read the data
  Read Name$
  If Name$ <> "STOP"
    Read ParentID%
    Read LaserPower%
    Read Armor%
    Read ShieldPower%
    Read Missiles%
    Read SecondaryMissiles%
    Read TopSpeed#

    ; create an instance of Fighters
    Fighter.Fighters = New Fighters
    ; if successful
```

```
      If Fighter.Fighters <> Null
        ; create an instance of Ships
        Fighter\ShipBasics.Ships = New Ships
        ; if successful, add to the instance
        If Fighter\ShipBasics.Ships <> Null
           Fighter\ParentID% = ParentID%
           Fighter\ShipBasics\Name$ = Name$
           Fighter\ShipBasics\LaserPower% = LaserPower%
           Fighter\ShipBasics\Armor% = Armor%
           Fighter\ShipBasics\ShieldPower% = ShieldPower%
           Fighter\ShipBasics\Missiles% = Missiles%
           Fighter\ShipBasics\TopSpeed# = TopSpeed#
        Else
           Text 0,0,"Error: Could not create the Ships instance!"
          bData = False
        EndIf
      Else
        Text 0,0,"Error: Could not create the Destroyer instance!"
        bData = False
      EndIf
    Else
      bData = False
    EndIf
Wend

; Set up the vertical control variable
iTextY% = 0

; Step through and print
For Destroyer.Destroyers = Each Destroyers
    Text 0,iTextY,"Ship Name: " + Destroyer\ShipBasics\Name$
    iTextY% = iTextY + 16

    ; now loop through each of the fighters and print out
    ; all of the ones belonging to this destroyer
    For Fighter.Fighters = Each Fighters
       If Fighter\ParentID% = Destroyer\ID%
         Text 50,iTextY,"Fighter Name: " + Fighter\ShipBasics\Name$
         iTextY% = iTextY + 16
       EndIf
    Next

    iTextY% = iTextY + 32

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
    ; next part is just for ease of viewing
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; if it's the first destroyer, wait for key then show next
    If Destroyer\ID% = 1
      Text 0,450, "press a key to see the next ship"
      WaitKey ()
      Cls
      iTextY = 0
    ; if it's the second destroyer, wait for key then exit
    Else
      Text 0,450, "press a key to exit"
    EndIf
  Next

; delete the instances
For Destroyer.Destroyers = Each Destroyers

    ; now loop through each of the fighters and delete
    ; all of the ones belonging to this destroyer
    For Fighter.Fighters = Each Fighters
        If Fighter\ParentID% = Destroyer\ID%
            Delete Fighter\ShipBasics.Ships
            Delete Fighter
        EndIf
    Next

    Delete Destroyer\ShipBasics.Ships
    Delete Destroyer
Next

; wait for a keypress
WaitKey ()

; end the program
End

.DestroyerData
 Data "Kliazian Battlecruiser", 1,15,125,200,25,12,2.50, 20,100,500
 Data "Weltic Storm of Zion",2,20,150,220,45,27,2.00, 25,200,750
 Data "STOP"

.FighterData
 Data "Raptor",1,15,125,200,25,12,2.50
 Data "Raptor2",1,15,125,200,25,12,2.50
 Data "Raptor3",1,15,125,200,25,12,2.50
```

```
Data "Blade",1,20,150,220,45,27,2.00
Data "Blade2",1,20,150,220,45,27,2.00
Data "Blade3",1,20,150,220,45,27,2.00
Data "Blade4",1,20,150,220,45,27,2.00
Data "Blade5",1,20,150,220,45,27,2.00
Data "Blade6",1,20,150,220,45,27,2.00
Data "Interceptor",1,20,150,220,45,27,2.00
Data "Interceptor2",1,20,150,220,45,27,2.00
Data "Interceptor3",1,20,150,220,45,27,2.00
Data "Interceptor4",1,20,150,220,45,27,2.00
Data "Interceptor5",1,20,150,220,45,27,2.00
Data "Interceptor6",1,20,150,220,45,27,2.00
Data "Arch of Freedom",2,20,150,220,45,27,2.00
Data "Arch of Freedom2",2,20,150,220,45,27,2.00
Data "Arch of Freedom3",2,20,150,220,45,27,2.00
Data "Arch of Freedom4",2,20,150,220,45,27,2.00
Data "Keeper of Wrath",2,20,150,220,45,27,2.00
Data "Keeper of Wrath2",2,20,150,220,45,27,2.00
Data "Keeper of Wrath3",2,20,150,220,45,27,2.00
Data "Keeper of Wrath4",2,20,150,220,45,27,2.00
Data "Keeper of Wrath5",2,20,150,220,45,27,2.00
Data "Keeper of Wrath6",2,20,150,220,45,27,2.00
Data "Keeper of Wrath7",2,20,150,220,45,27,2.00
Data "Fire of Zeal",2,20,150,220,45,27,2.00
Data "Fire of Zeal2",2,20,150,220,45,27,2.00
Data "Fire of Zeal3",2,20,150,220,45,27,2.00
Data "Fire of Zeal4",2,20,150,220,45,27,2.00
Data "Fire of Zeal5",2,20,150,220,45,27,2.00
Data "Stone of Weltic",2,20,150,220,45,27,2.00
Data "Stone of Weltic2",2,20,150,220,45,27,2.00
Data "Stone of Weltic3",2,20,150,220,45,27,2.00
Data "Stone of Weltic4",2,20,150,220,45,27,2.00
Data "Protector of Denshan",2,20,150,220,45,27,2.00
Data "STOP"
```

Chapter 8: Data Banks

For more direct access to your computer's memory, Blitz has included a tool for creating and managing *Data Banks*. A *Data Bank* is an area of memory reserved for manipulation as you, the developer, see fit. You can write any values you want to these banks, you just have to make sure that there is enough memory.

Banks can be used in various situations. They can be completely controlled on a size basis, and they tend to be quite quick. Their biggest drawback is that they are user-controlled. Blitz doesn't handle all of the processing like it does with TYPES and arrays. You have to handle the processing in your own style. In some ways this is good because you have the freedom to code the memory chunks as you see fit (maybe to help prevent hacking, for example, you plop various banks all over the place with cryptic naming conventions and such). But it can also be quite challenging if you're not careful in your planning and use of these powerful tools.

Creating and Freeing Data Banks

Whenever you create a bank it's important to know that you are allocating byte-sized chunks. If, for example, you plan to store an integer into a bank, you would have to create a bank that is four bytes long. If you created a bank that is only 3 bytes wide and tried to store an integer value, Blitz would generate an error at run-time when you tried to read that value back.

Creating a bank is easy. Just use the CREATEBANK command. Here is the layout:

BankHandle = ***CreateBank (****NumberOfBytes****)***

BankHandle is a unique identifier that you will use in reference to this particular bank for any bank processing. It can be any name you choose, except for any Blitz reserved name. However, as with all variable declarations, you'd be wise to make the name something relevant to its purpose.

It's also a good idea to make sure that Blitz was able to reserve the bank requested. If the *BankHandle* contains 0 (zero), then Blitz was unsuccessful in creating the bank and you should take appropriate steps.

You also want to be sure to release the memory associated to your bank when you have finished using it. This is very important because you could end up with tons of memory being allocated but never freed. If you neglect to free

the memory you may eventually run out of memory in your game. Here is the format of the FREEBANK command:

FreeBank *BankHandle*

Here is a snippet that attempts to create a bank and verifies if it was successful or not:

```
; create a bank of 500 bytes
MyBank = CreateBank (500)

; verify that the create was successful.
; If not, notify the user and exit.
If MyBank = 0
   RunTimeError "Error Creating Data Bank –MyBank-"
   End
EndIf
```

Notice the command RUNTIMEERROR in that example. In previous chapters we didn't really focus on informing the user if there is a problem, but since we have moved into the realm of Data Banks it's become more important. All the RUNTIMEERROR command does is to display a window on the screen with a message that you want displayed and waits for the user to click the "OK" button. You still have to handle any processing after that. It's a smart thing to include in all of your *failed* checks, whether for ARRAYS, TYPES, data loading, or anything that can *fail* in your program.

Poke and Peek

So after we've created a bank, how do we use it? There are a number of commands in the set, but most of them are similar in function. To place a value into a bank you would use POKE, and to read you would use PEEK. There are currently four POKE/PEEK types that you have access to:

- **PokeByte / PeekByte**
- **PokeShort / PeekShort**
- **PokeInt / PeekInt**
- **PokeFloat / PeekFloat**

The POKE commands are all formatted identically. Let's use POKEINT as an example:

PokeInt *BankName,Offset,Value*

Likewise, all of the PEEK commands share the same format:

Value **=** ***PeekInt*** *BankName,Offset*

Really, the only difference is the type of *Value* being used. This is an important distinction, of course, but at least you won't have to fiddle around wondering if one command differs from the other simply due to its data type.

Now, *where* you POKE and PEEK values to/from is an issue you'll need to deal with. This is type-dependent not because the commands differ, but because the value sizes do. Here is a list of the possible type sizes:

- **Byte =** 1 Byte
- **Short =** 2 Bytes
- **Int =** 4 Bytes
- **Float =** 8 Bytes

Why is this important to note? When you go to POKE/PEEK a value, you have to inform Blitz of where you want to POKE/PEEK from. If you have two integer values, for example, the offset of the first would be zero and the offset of the second would be four. You're used to the ARRAY convention of zero holding the first integer, one holding the second, two holding the third, and so on. Data Banks don't work that way though, because recall that *you* have to handle the specific processing. This means that you need to control where the offsets are, and to do that you need to know the types and sizes.

Here is a little piece of code that demonstrates a simple two-integer bank using POKE and PEEK**:**

```
; initialize our graphics
Graphics 640,480

; create a databank 8 bytes long
MyBank = CreateBank (8)

; verify the bank was created succesffully, error out if not.
If MyBank = 0
  RuntimeError "Not enough memory for the bank!"
  End
EndIf

; poke two integer values (pay attention to the offset!)
PokeInt MyBank,0,100
PokeInt MyBank,4,200

; peek those values back out (pay attention to the offset!)
```

```
IntValue1 = PeekInt(MyBank,0)
IntValue2 = PeekInt(MyBank,4)

; write out the values for the user to see
Text 0,0,IntValue1
Text 0,16,IntValue2

; free the memory allocated by CreateBank
FreeBank MyBank

WaitKey ; wait for a key press
End ; end the program
```

Note that the *offsets* are spaced by 4 bytes. That's because we're using an integer value. If we were to use a *short*, the POKES would look like this:

```
PokeShort MyBank,0,100
PokeShort MyBank,2,200
```

For floats, they would look like this:

```
PokeFloat MyBank,0,100.175
PokeFloat MyBank,8,200.25
```

And for bytes, it would simply be:

```
PokeByte MyBank,0,100
PokeByte MyBank,1,200
```

Now you're not likely going to want to do the manual calculation for each size and write out a ton of POKE/PEEK statements, so you'll need a way to have Blitz determine the offsets for you. No problem, just use a multiplier of the current offset with the value size. Here's an example:

```
; initialize our graphics
Graphics 640,480

Const IntSize = 4
Const MyBankSize = 20

; create a databank 8 bytes long
MyBank = CreateBank (MyBankSize * IntSize)
```

```
; verify the bank was created succesffully, error out if not.
If MyBank = 0
  RuntimeError "Not enough memory for the bank!"
  End
EndIf

; go through and put in all the values
For i = 0 To MyBankSize - 1
   PokeByte MyBank, i * IntSize, i
Next

; declare a variable to track the Y position of our text output
iTextY% = 0

; now got through and read them back out and display them
For i = 0 To MyBankSize - 1
   ; just incorporate the PeekInt directly into the Text call
   Text 0,iTextY,"Bank: "+i+" contains the value: "+ ↲
        → PeekInt(MyBank,i * IntSize)

   ; increment our Y position for the text output
   iTextY = iTextY + 16
Next

; free the memory allocated by CreateBank
FreeBank MyBank

; wait for a key press
WaitKey

; end the program
End
```

You may have caught that in our FOR…NEXT loops we went from *0 to MyBankSize – 1*. We use the –1 because recall that Blitz is counting from 0. Since bank size is 20, that means that Blitz will actually hit a final value of 21 before satisfying this loop. That also means we'll get an error. In order to make sure that we don't go past the allocated bank memory, and knowing that Blitz starts counting at zero, not 1, we use the –1 in the loop.

Resizing, Copying, and Finding Current Size Information

So what if you have a bank all set up, but you find that you need to expand its size? Maybe you are dynamically allocating bank memory for each new

map that you load in. You could easily just FREE the current bank and re-create it, or you could use the RESIZEBANK command. Here is the format:

ResizeBank *BankHandle,NewSize*

Resizing is as simple as creating, with the only difference being that you already know the name of the bank! It is still important that you check that Blitz was able to resize successfully, as you do with CREATEBANK, so don't leave out that step. Again, here's a little snippet that creates a bank and then resizes it.

```
; create a bank of 500 bytes
 MyBank = CreateBank (500)

 ; verify that the create was successful.
 ; If not, notify the user and exit.
 If MyBank = 0
    RunTimeError "Error Creating Data Bank –MyBank-"
    End
 EndIf

 ; Resize the bank to 750 bytes
 ResizeBank (750)

 ; verify the resize was successful.
 ; If not, notify the user and exit.
 If MyBank = 0
    RunTimeError "Error Resizing Data Bank –MyBank-"
    End
 EndIf
```

So what happens to the data if you resize a bank to a smaller size? The data is gone. Anything past the resize point will not be contained, but anything up to the resize point will remain.

Copying data between two banks, and even from one position in a bank to another position in the same bank, is also supported. To use this feature, you'll need the COPYBANK command, which is formatted as follows:

CopyBank *SourceBankHandle,Offset,DestBankHandle,Offset,* ↲
→ *NumberOfBytes*

If we had a bank of 100 bytes and wanted to copy it to another bank of 100 bytes, we could use this code:

```
CopyBank MyBank,0,NewBank,0,100
```

And then when we reference *NewBank* it will contain the identical information that *MyBank* contains (assuming no subsequent alterations to *MyBank*). If we wanted to duplicate the first 50 bytes in *MyBank* over the last 50 bytes, we would do this:

```
CopyBank MyBank,0,MyBank,50,50
```

Keep in mind that the *offset* and the *NumberOfBytes* values are byte-sized, so you'll need to ensure that you're setting these according to your data type.

Finally, if you ever want to know what the current size of your bank is, just call the BANKSIZE command. The format is:

Size = ***BankSize (****BankHandle****)***

Again, note that if you have a bank of 10 integers, the size returned from BANKSIZE will be 10*4 (because an integer is 4 bytes).

Chapter 9: Functions and Libraries

As you get deeper and deeper into game development, you'll soon find that you're replicating a lot of work. Maybe you've already written code that handles the various input devices (mouse, joystick, keyboard, etc.). Why write those processes all over again? Also, your code is going to get bigger and bigger as you continue developing. How will you maintain all those pages effectively? Enter functions.

A function is a piece of code that you can call to perform a particular activity and then, when it's completed, return control back to the calling code. Think of it as clicking on one of your computer's applications…say Blitz Basic. When you click on it to run, the computer takes over and loads up BB. Upon completion, the computer gives control back to you. Functions act similarly, with the exception that they return control back to the code that called them.

Functions also have the capability to process information sent to them and return information back based on the processing that was done.

For example, let's say that each time a laser blast smacked your ship you need to know what the resulting damage was. Well, maybe you have a function that checks where on your ship you were hit, what the current armor was at the time of the hit and how much power the hit delivered. From here the function processes all the data, does a calculation and let's you know what the total damage was.

There are literally thousands of uses for functions.

Another extremely important use is to maintain fluency in your coding. In other words, with functions you can break up the code into manageable chunks, with each "chunk" having a declarative name that clearly identifies its purpose.

Declaring a Function

In order to use a function, you must first declare it. In order to do this you would use the Blitz command FUNCTION. Here is the layout:

Function *FunctionName<ReturnType>(Optional Arguments)*

What you call the function is completely up to you, but the more descriptive your name for it, the easier it will be to use it and recall its purpose. This is important if you ever plan on using this function in other programs.

Examples of bad function names:

- **Function** a()
- **Function** MoveIt(It)
- **Function** Sideways()

When you look at these three example names, the only one that remotely makes sense is *MoveIt*. The only problem is that you won't easily be able to integrate this function into another program because the *It* portion of *MoveIt* is likely specific to the current program. Now, that's not necessarily a bad thing, as long as it is not your goal to reuse this function.

Examples of good names:

- **Function** FireLaser(Direction)
- **Function** CheckCollisions(Image1, Image2)
- **Function** DisplayScore(CurrentScore)

Notice that each of these functions is clearly named. If you ever want to see if two images collide, simply call on *CheckCollision* and it'll tell you. Want to display the score? *DisplayScore* does the trick.

To declare the function, you use the FUNCTION command with a selected name followed by the return type and then (), either including arguments or not as the case may be. The return types are as follows:

- **%** - Returns an integer value. This is the default.
- **$** - Returns a string.
- **#** - Returns a float.
- **.TypeName** – Returns a value of TYPE.

Another thing that I do when declaring my functions is to place a detailed comment above it with pertinent information. Such as:

```
;************************************************
;  Function: FunctionName()
;         By: Author
;  Last Upd: Date
;   Purpose: The purpose of this function
;      Args:  Describe what's to be sent to this fnction
;   Returns:  Describe what the function will return
; Comments:  Place any additional comments here
;************************************************
```

Now, it's certainly not necessary that you use this format or that you comment your functions at all. But I would highly recommend that you do, as I would recommend that you comment your code thoroughly regardless of functions. Eventually, you'll revisit your work (or someone else will) and you'll be very glad to know what you were thinking at the time you were coding.

When you've completed processing, you simply closeout the function with the END FUNCTION command. This command just lets Blitz know that all the code within this function is complete. If you leave this line out, Blitz will assume that more lines are coming and will either give an error or include erroneous data.

Passing Arguments and Returning Results

First thing I should qualify is what exactly a function *argument* is. An *argument* is a piece of data that you send to a function for processing. For example, if you wanted to add two numbers together, you would *send* the numbers to the function. The function would then add the two numbers and RETURN the resultant value. An argument can be anything from a number or string to an array or TYPE. And functions can return mostly anything as well.

One of the major limitations of a function is that it can only return one value, at least without the use of trickery. So, while you can send many arguments, only one value can come back.

Here is an example program that has a bunch of functions, all with different return types:

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; setup the random second generator
SeedRnd MilliSecs()

; make a dummy array and fill it with values
Dim ArrayValues(5)
ArrayValues(1) = 100
ArrayValues(2) = 200
ArrayValues(3) = 300
ArrayValues(4) = 400
ArrayValues(5) = 500

; define a type for Ships
Type Ships
   Field ShipID     ; what's it's ID?
```

```
   Field ShipName$  ; the name?
   Field X,Y        ; where's it's coords?
End Type

; make Ship.Ships a global var
Global Ship.Ships = First Ships

; add a few ships.  Note that this function does NOT
; return a value!
SetupShips()

; call AddNumbers function and place the returned-value
; into the variable "intValue"
intValue = AddNumbers(10,20)

; call ConcatString function and place the returned-value
; into the variable "StringValue"
StringValue$ = ConcatString$("How","dy")

; call AvgNumbers function and place the returned-value
; into the variable "floatValue"
floatValue# = AvgNumbers#(1.495,3.772,11.1935)

; call GetArray function and place the returned-value
; into the variable "arrayValue"
arrayValue = GetArray(3)

; call GetShip function and place the returned-value
; into the variable "ShipValue.Ships"
ShipValue.Ships = GetShip.Ships(Rand(1,10))

; display returned values
Print "intValue = " + intValue
Print "StringValue = " + StringValue$
Print "floatValue = " + floatValue#
Print "arrayValue = "+ arrayValue
Print
Print "** Ship Info **"
Print "---------------"
Print "ID = " + ShipValue\ShipID
Print "Name = " + ShipValue\ShipName$
Print "X = " + ShipValue\X + ", Y = " + ShipValue\Y

; wait for a keypress
WaitKey ()
```

```
; end the program
End

;*********************************************
;  Function: AddNumbers()
;    Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: add two numbers and return the result
;      Args: Two numbers
;   Returns: Sum of the two numbers sent
;  Comments: None
;*********************************************
Function AddNumbers(iNumber1, iNumber2)
   iSum = iNumber1 + iNumber2
   Return(iSum)
End Function

;*********************************************
;  Function: ConcatString$()
;    Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: concatonates two strings
;      Args: Two strings
;   Returns: the resultant string
;  Comments: None
;*********************************************
Function ConcatString$(String1$, String2$)
   Return String1$ + String2$
End Function

;*********************************************
;  Function: AvgNumbers#()
;    Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: Find the average of 3 floats
;      Args: Two numbers
;   Returns: Sum of the two numbers sent
;  Comments: None
;*********************************************
Function AvgNumbers#(float1#, float2#, float3#)
   avg# = (float1 + float2 + float3) / 3
   Return avg#
End Function
```

```
;*********************************************
;  Function: GetArray()
;     Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: Get a particular array value
;      Args: location in the array
;   Returns: The array value
;  Comments: None
;*********************************************
Function GetArray(Location%)
   Return ArrayValues(Location)
End Function

;*********************************************
;  Function: GetShip.Ships()
;     Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: locates a ship and returns it
;      Args: Ship's ID
;   Returns: the Type entry for the Ship
;  Comments: None
;*********************************************
Function GetShip.Ships(ShipID%)
   For Ship.Ships = Each Ships
      If Ship\ShipID = ShipID%
        Exit
      EndIf
   Next
   Return Ship.Ships
End Function

;*********************************************
;  Function: SetupShips()
;     Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: adds a few ships to the Type
;      Args: N/A
;   Returns: N/A
;  Comments: None
;*********************************************
Function SetupShips()
   For i = 1 To 10
      Ship.Ships = New Ships
      Ship\ShipID = i
      Ship\ShipName = "Ship" + i
```

```
      Ship\X = Rand(0,200)
      Ship\Y = Rand(0,300)
   Next
End Function

;*********************************************
;  Function: DeleteShips()
;     Author: John Logsdon
;  Last Upd: 5/4/02
;   Purpose: frees all the Ships Type memory
;      Args: N/A
;   Returns: N/A
;  Comments: None
;*********************************************
Function DeleteShips()
   For Ship.Ships = Each Ships
      Delete Ship
   Next
End Function
```

If you study that code in detail it should be pretty clear how to handle each return case.

You can place your functions at the top of your code, at the bottom (after the END), or even in another file—just remain consistent with your approach. I tend to place all application specific functions in my main BB file, and any functions that I plan to reuse in the future I place in separate files.

Using INCLUDE

Whenever you create a file that contains functions you will want to reuse, you'll have to have a way to let Blitz know that you want to include them in your main code. The relevant command is appropriately named INCLUDE. INCLUDE opens a particular file and squishes it in with another file.

Let's say you have a file called *ShipFighter.BB* and you have a bunch of functions in a file called *ImageProcessing.BB*. Instead of manually cutting and pasting, you need a way to just tell Blitz that when it compiles to include *ImageProcessing.BB*. All you would do is place the following line somewhere (preferably at the top of your code) in the *ShipFighter.BB* file:

```
Include "ImageProcessing.BB"
```

Now, from that moment on you will be able to use all of the functions in that file. When you put a bunch of related functions in a file, you can officially call

that file a *library*. This is because it is now a "library of functions." Pretty spiffy, no?

You should give this a try by using the above *AddNumbers* function. Cut that piece of code out and save it to another file. Then use the INCLUDE command to tell Blitz that you want it considered during compilation. Compile and run and you'll see the exact same result.

INCLUDE also makes it nice when others are working on projects with you because each of you may have an area of expertise or responsibility. For example, if you have a team member that's focusing on the menu system, you'll not likely care about all the intricacies of the code for the system, but you will care about what functions are available for you to use in piecing the final code together. Simply open the file and look at the *funcs* list on the right in BB. And if your team member commented the function tops well enough, you'll also know how to call each function, what its purpose is, and what you can expect it to return.

One last thing to think about when making function libraries is the names you use for your variables. Take care not to duplicate a variable name in your main code that already exists in the library. To help avoid this, I will often call my functions and variables starting with a prefix specific to the library. For example, if the library is called "MapLib.bb," I may call a variable *Map_TileSize* instead of just *TileSize.* Using a similar practice will help you keep your libraries out of trouble.

Chapter 10: Basic File Manipulation

The ability to save and load information from files will be extremely useful to you as your game development prowess grows. You'll have more and more data to process. Everything from map files to story lines to player save files to debugging information. If you look at almost any commercial quality game available today, you'll see that there are tons of files that make up the game's directories. One day your games will be like this too, so you may as well get used to files early on in your development career.

Files come in all shapes and sizes. There are binary files, full of what appears to be gibberish (it's not gibberish, mind you…just looks that way). There are also text files, which you can open in any editor and read clearly. Some files are enormous, containing all the necessary data to make up a full game level, while others contain only one or two lines.

So why the differences? The answer to that comes in the design phase. Let's say, for example, that your game allows a user to select the video mode to use when playing. You could require that the user select this every time she plays, but that would be annoying. Why not instead have a tiny file that is updated upon the change of the graphics mode selection, and then each time your game loads it reads that file and sets the mode accordingly? This file may only be one byte in length. Sounds like a waste of a file, but your computer doesn't care and since it's only read in at the beginning of your game, it's not going to impact performance one iota.

Taking another example, let's say that you have a file that is used to keep track of where the player is in the game. This file contains all the "secrets" to your game, such as hidden objects, opened and closed paths, keys for doors, etc. Well, if you make this a straight text file, then any player can simply open it up and have a look at what to do to pass the level. So in this case you may decide to use a binary file (which looks more like gibberish). This will stop the common user from finding out your secrets, but more power users can easily get past this. So maybe your file also contains encryption and compression to further protect.

As you can see, the choice is yours on how you want to configure your files, so let's start talking about the basics of file manipulation.

Creating and Writing Files

Blitz offers a number of file manipulation commands, but typically the best place to start in describing file processing is by writing to a file. In order to write to a file, you must first have a file to write to. Fortunately, Blitz handles

this in one command. The WRITEFILE command literally opens a file for writing. The command layout is as follows:

FileHandle = ***WriteFile (****FullPath/Filename****)***

Be careful when using this command because if you already have a file with the name you pass to the WRITEFILE command, Blitz will overwrite that file. Also, note that you must include the full path to the file. If you don't, Blitz will use the same directory that your program currently resides in to create the file. Finally, verify that the returned *FileHandle* is a non-zero value. If it is zero that means Blitz could not open the file for writing and any future attempts at the file will be unsuccessful.

You can use the FILETYPE command to see if the file exists prior to using WRITEFILE. FILETYPE takes the argument of a valid path and filename. If the file is not found, this command returns a 0. If it is found, it will return a 1. If, however, it turns out to be a directory name, it will instead return a 2.

Our next step is to determine the type of value we want to write out. There are currently 7 different write options to choose from. Here they are and what each is for:

- **WriteByte:** Writes a single byte to a file.
- **WriteShort:** Writes a short-integer (16-bits) to a file.
- **WriteInt:** Writes an integer (32-bits) to a file.
- **WriteFloat:** Writes a floating-point value to a file.
- **WriteString:** Writes a character string to a file in binary mode.
- **WriteLine:** Writes a character string to a file in text mode.
- **WriteBytes:** Writes data from a **DataBank** to a file.

With the exception of WRITEBYTES, all of these commands have the same format:

WriteByte(*FileHandle,Value)*
WriteShort(*FileHandle,Value)*
WriteInt(*FileHandle,Value)*
WriteFloat(*FileHandle,Value)*
WriteString(*FileHandle,Value)*
WriteLine(*FileHandle,Value)*

The WRITEBYTES command, however, uses the following format:

WriteBytes *BankHandle,FileHandle,Offset,Count*

Let's put together a small program that writes out two lines to a text file. First, we'll use the text mode version:

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a variable to hold our file name so we
 ; can use it throughout our code.
 FileName$ = "filetest.txt"

 ; open a file for writing, thus overwriting the existing file
 hFile = WriteFile (FileName$)

 ; check to see if the file was successfully created
 If hFile <> 0
   ; write out text to the file
   WriteLine(hFile,"Hello, Blitz Basic!")
   WriteLine(hFile,"Testing...testing...1...2....3")

   ; close the file
   CloseFile(hFile)

   ; notify of our success
   Print "Data successfully written to <" + FileName$ + ">"
 Else
   ; explain that there was an error
   Print "Could not open file: <" + FileName$ + ">"
 EndIf

 WaitKey () ; wait for the user to press a key
 End  ; end the program
```

In order to see what that output would look like in binary format, change the WRITELINE commands to WRITESTRING.

It's always a good idea to close your files when you are finished with them. You don't want to have a file left open because it can get garbled. So, make sure to call CLOSEFILE at the appropriate time in your code.

Reading From a File

In order to read from a file, we must first open that file. When opening a file Blitz will return a *FileHandle* if successful, or 0 if not. Here is the format of the READFILE command:

FileHandle = ***ReadFile (****FilePath\FileName****)***

From here you simply use one of the *Read* commands below:

- **ReadByte:** Reads a single byte to a file.
- **ReadShort:** Reads a short-integer (16-bits) to a file.
- **ReadInt:** Reads an integer (32-bits) to a file.
- **ReadFloat:** Reads a floating-point value to a file.
- **ReadString:** Reads a character string to a file in binary mode.
- **ReadLine:** Reads a character string to a file in text mode.
- **ReadBytes:** Reads data from a **DataBank** to a file.

The idea here is that whatever you used to write the value out, you in return use the read equivalent. Also, it's important to note that the formatting is a little different as well. Here is the basic layout:

ReturnValue = ***ReadByte(****FileHandle****)***
ReturnValue = ***ReadShort(****FileHandle****)***
ReturnValue = ***ReadInt(****FileHandle****)***
ReturnValue = ***ReadFloat(****FileHandle****)***
ReturnValue = ***ReadString(****FileHandle****)***
ReturnValue = ***ReadLine(****FileHandle****)***

…and for READBANK (notice it's *almost* identical to its counterpart):

ReadBank *BankHandle,FileHandle,Offset,Count*

Now let's read in and display the values in the file that we just created with the sample code above:

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; setup a variable to hold our file name so we
 ; can use it throughout our code.
 FileName$ = "filetest.txt"

 ; open a file for reading
 hFile = ReadFile (FileName$)

 ; check to see if the file was successfully created
 If hFile <> 0
   ; read in the two lines we wrote out
   Line1$ = ReadLine(hFile)
   Line2$ = ReadLine(hFile)
```

```
  ; close the file
  CloseFile(hFile)

  ; notify of our success and print out the values
  Print "Data read from <" + FileName$ + ">"
  Print "Line 1: " + Line1$
  Print "Line 2: " + Line2$
Else
  ; explain that there was an error
  Print "Could not open file: <" + FileName$ + ">"
EndIf

WaitKey () ; wait for the user to press a key
End ; end the program
```

What if you want to open a file for reading *and* writing? In other words, you don't want to have to use WRITEFILE to do all your work and then close and use READFILE to read everything, etc. You would use the OPENFILE command.

This command will *not* create a file—you must use WRITEFILE for that—but it will allow you to read and write to an already existing file without having to create it and use various other commands first. In fact, you have to use this command to alter any existing file because WRITEFILE will overwrite an existing file.

Moving Around Inside of Files

Whenever you read or write a file an internal file "pointer" moves around to keep your position within that file. Imagine the pointer literally. It's just a piece of memory that holds (*points* at) a specific location in the file. Each time you read or write a character, the pointer increases to point to the position beyond its current position. This is an important concept to grasp because you'll undoubtedly have a need to move around inside your files in order to update them dynamically.

Blitz offers a few commands to help you keep track of this pointer, and to move it around accordingly. The first command is called FILEPOS and its job is to simply tell you the current position that the pointer is at in your file. Here's the format:

FilePosValue = ***FilePos (****FileHandle****)***

The second command is called SEEKFILE and it allows you to position the pointer wherever you want in the file, as long as it's a valid position.

SeekFile (*FileHandle,Offset****)***

Finally, you can use the EOF command to see if you've reached the end of the file. This is so you don't overrun the pointer.

EOFValue = ***EOF (****FileHandle****)***

This command will return a 1 if the end of the file has been hit, a 0 if not, and a –1 if there was an error.

Now that we have this stuff under our belts, let's take our previous file, open it and change the second line to say something else:

```
; Initialize Blitz Basic to 640x480 resolution
Graphics 640,480

; setup a variable to hold our file name so we
; can use it throughout our code.
FileName$ = "filetest.txt"

; open a file for reading
hFile = OpenFile(FileName$)

; check to see if the file was successfully created
If hFile <> 0
  ; read in the two lines we wrote out
  Line1$ = ReadLine(hFile)
  fPos = FilePos (hFile)
  Line2$ = ReadLine(hFile)

  ; Show our original values
  Print "Original Text:
  Print "Line 1: " + Line1$
  Print "Line 2: " + Line2$

  Print ""

  ; wait for the user to hit a key
  Print "...Press A key to change text in file..."
  WaitKey ()

  ; reset the pointer in the file and write the new line
```

```
  SeekFile (hFile,fPos)
  WriteLine(hFile,"The new line 2!")

  ; reset the pointer to the beginning of the file
  SeekFile (hFile,0)

  ; read in the new lines
  Line1$ = ReadLine(hFile)
  Line2$ = ReadLine(hFile)

  Print ""

  ; output the updated text
  Print "Updated Text:"
  Print "Line 1: " + Line1$
  Print "Line 2: " + Line2$
  Print ""
  Print "...BUT there is now a line 3!"
  Line3$ = ReadLine(hFile)
  Print "Line 3: " + Line3$


  ; close the file
  CloseFile(hFile)

Else
  ; explain that there was an error
  Text 0,0,"Could not open file: <" + FileName$ + ">"
EndIf

WaitKey () ; wait for the user to press a key
End ; end the program
```

You should now go and open that file with a text editor because what you'll see is that while the outputted text on your screen was accurate, the text in the file looks a little funny. You should see three lines of text with the last being "…1…2…3". This is because we are only overwriting from the beginning of the second line for the duration of the output string. So there is leftover data in that file. You'll need to be careful about this and devise ways to handle these circumstances accordingly.

PART 2: BB GAME TOOLS

Chapter 11: Colors and Drawing Primitives

When I used to hear terms like "primitives" I had no idea what people were talking about. Well, it's not as bad as you might think. Basically, a primitive is something that is a building block for more advanced graphics. For example, in order to draw a line, you must use pixels. To draw a box you use lines. To draw a ship you use a bunch of things, like cubes, spheres, cylinders, and cones...and those can be made using triangles.

But since you'll want all of these primitives to have varied colors, so they're not too bland, you'll also want to use colors.

Getting and Setting Colors

Colors will be changed constantly in your game. You'll have specific text types that will show up brighter than other text. You'll have pixel effects that need to have a variety of colors to have deeper impact. I'm sure you can think of a million reasons for using colors in your game.

Because of this, you need to be familiar with not only how to set colors, but also how to remember the current color before doing changes. It wouldn't look very good to have a pixel turn red and therefore all of your text turns red (unless that was your plan). So being able to know what the current colors are is a key factor in color control and manipulation. That said, let's start with getting the current colors.

There are three commands in Blitz that grab the current color scheme.

- **ColorRed()** – gets the Red component of the RGB scheme
- **ColorGreen()** – gets the Green component of the RGB scheme
- **ColorBlue()** – gets the Blue component of the RGB scheme

Each of these returns a numeric value between 0 and 255 for the component.

You'll need to grab all three components each time in order to have the accurate color to reset to. In order to reset the color, you have to know how to set the current RGB color to whatever you want. The color is set with the COLOR command, which has the following format:

Color(*Red,Green,Blue****)***

The values passed for each argument must be between 0 and 255. Sending 255,255,255 to the COLOR command would set the color to white. Sending

0,0,0 would set it to black. You can combine these numbers in any fashion that you see fit in order to make whatever colors you want.

Another point to note is that the CLSCOLOR command can be used to clear the screen to a particular color. It is called identically as the COLOR command.

If you wanted to get the color of a particular point on the screen, you could use the GETCOLOR command. What this function does is reads the color value of a single point and sets that as the current drawing color. The format of this command is:

GetColor *X, Y*

It does not return a value, but rather sets the current drawing color to whatever color was found at the X, Y location.

Dealing with Pixels

A pixel (a condensed word meaning *picture element*) is simply a dot on the screen. When you combine a bunch of these dots, you can make most any image come to life. Everything you see on the screen, from the letters to icons to even the mouse cursor, are all made using pixels.

In order to draw a pixel to the screen, you must call one of the Blitz pixel drawing commands.

- **Plot**: Draws a single pixel on the current buffer in the current color
- **WritePixel**: Quickly draws a single pixel to any buffer in any color
- **WritePixelFast**: Very fast pixel writing to any buffer in any color, but must be used in conjunction with the **LockBuffer**/**UnlockBuffer** commands.

The PLOT command takes the current color and draws a pixel at the corresponding X, Y position on the screen. The following code will randomly plot pixels on the screen until you press the Escape key.

```
; setup our screen constants
Const ScreenWidth = 640
Const ScreenHeight = 480

; Initialize Blitz Basic
Graphics ScreenWidth, ScreenHeight

; set the seed for the random number generator
SeedRnd MilliSecs ()
```

```
; save our current colors
Red = ColorRed ()
Green = ColorGreen ()
Blue = ColorBlue ()

; while the user doesn't hit <ESC>, which is '1'
While Not KeyHit (1)

  ; reset the color so our text doesn't change colors
  Color Red, Green, Blue

  ; print out our heading
  Text 0,0,"Plot Pixel Demo.  Hit <ESC> to Quit."

  ; randomize a color
  Color Rnd (0,255), Rnd(0,255), Rnd(0,255)

  ; plot a pixel in a random location
  Plot Rnd (0,ScreenWidth), Rnd(0,ScreenHeight)
Wend
End  ; end the program
```

Enter that in and you'll see a ton of pixels start filling up your screen. Try commenting out the COLOR command above the Text line and you'll get to see the text change colors too.

Also, you should note the use of the SEEDRND and the RND commands. These are functions that create random numbers. SEEDRND accepts a value, preferably something unique like the computer's clock (hence the use of MILLISECS), which RND then uses to create other random numbers.

RND accepts a low number and a high number and randomly picks something in-between, though it is inclusive. Here are the formats for both of these commands:

SeedRnd *Value*
RandomNumber = ***Rnd (****LowestNumber, HighestNumber****)*

The WRITEPIXEL command is more robust, but it must have an encoded RGB command sent along with it. One of the ways to handle this is to use the READPIXEL command to grab the RGB values of a pixel and then write the pixel with that color. Here is the format of WRITEPIXEL:

WritePixel *X,Y,RGB Value,Buffer*

While the *Buffer* argument is optional, the *RGB* argument is not. Here is the pixel-plotting demo using the WRITEPIXEL/READPIXEL combination. Note that it's all one color because I plot a single pixel, read its value and then display based on its value.

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

 ; Initialize Blitz Basic
 Graphics ScreenWidth, ScreenHeight

 ; set the seed for the random number generator
 SeedRnd MilliSecs ()

 ; draw our source pixel for color collection
 Plot 0,0

 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)

   ; print out our heading
   Text 0,0,"WritePixel Demo.  Hit <ESC> to Quit."

   ; write a pixel in a random location
   WritePixel (Rnd (0,ScreenWidth),Rnd(0,ScreenHeight), ↲
              → ReadPixel(0,0))
Wend
End  ; end the program
```

To speed things up even more we can use WRITEPIXELFAST and READPIXELFAST. These two commands are super-optimized, but they do require that you lock the buffer you'll be using them on before using them. Be *very* careful when using these locking commands though because you can cause lots of problems by not using them correctly.

The process is as follows:

1) LOCK the buffer
2) Write and/or Read Pixels
3) UNLOCK the buffer

That's all there is to it. It's much faster, but it's still not fast enough to use for real-time effects. You should always use images where you can, but we'll touch on that later.

Here's the modified code:

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

 ; Initialize Blitz Basic
 Graphics ScreenWidth, ScreenHeight

 ; set the seed for the random number generator
 SeedRnd MilliSecs ()

 Plot 0,0
 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)

   ; print out our heading
   Text 0,0,"WritePixelFast Demo.  Hit <ESC> to Quit."

   ; lock the front buffer
   LockBuffer FrontBuffer ()

   ; write a pixel in a random location
   WritePixelFast (Rnd (0,ScreenWidth), ↲
                  → Rnd (0,ScreenHeight), ReadPixelFast(0,0))

   ; unlock the front buffer
   UnlockBuffer FrontBuffer ()
Wend
End  ; end the program
```

You can also copy pixels using the COPYPIXEL and COPYPIXELFAST commands. The same holds true for locking buffers with the COPYPIXELFAST command, but COPYPIXEL does not require this. Here is the format for these commands:

CopyPixel (*SrcX,SrcY,SrcBuffer,DestX,DestY,DestBuffer****)***
CopyPixelFast (*SrcX,SrcY,SrcBuffer,DestX,DestY,DestBuffer****)***

The *DestBuffer* is an optional argument, but you will have to provide the *SrcBuffer (SourceBuffer)*. In our current circumstance, using the FRONTBUFFER command here works fine. When we get into animation techniques and more advanced buffer discussions that will change.

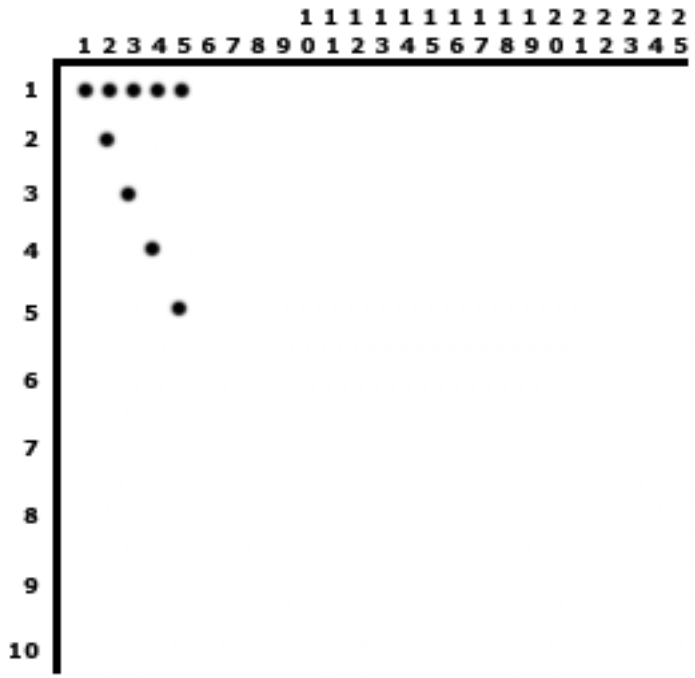
Here is the sample code using COPYPIXEL:

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

 ; Initialize Blitz Basic
 Graphics ScreenWidth, ScreenHeight

 ; set the seed for the random number generator
 SeedRnd MilliSecs ()

 Plot 0,0
 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)

   ; print out our heading
   Text 0,0,"CopyPixel Demo.  Hit <ESC> to Quit."

   ; copy the source pixel to a random location
   CopyPixel (0,0,FrontBuffer (),Rnd (0,ScreenWidth), ↲
            → Rnd (0,ScreenHeight))
Wend
End  ; end the program
```

Drawing Lines

A line is basically just a bunch of plotted points, but if you attempt to manually plot the points necessary to make a line you'll notice a lot of weird things.

Firstly, it'll probably be quite a bit slower than just using Blitz's built-in LINE command. Secondly, you'll need to compensate for the fact that moving along an X-axis that has a tighter ratio of pixels will make all of your Y-axis pixels appear to jump. To understand this more clearly, look at the following graphic:



(Figure 11.1)

See how the first line contains five dots that are all tightly lined up, yet the diagonal dots have a rather large gap between them? This is exactly the kind of thing you can expect to deal with when trying to implement your own line drawing function. The reason this occurs, as the above graph shows, is because there are fewer graphing points on the Y-axis then there are on the X-axis.

You'll be dealing with resolutions such as 640x480 and 1024x768. In all cases the number of pixels wide will be different than the number high.

There are many algorithms for dealing with this issue, such as the famous Brensenham algorithm, but discussing those topics is beyond the scope of this book. Search the web for Brensenham and you'll likely find many references.

Fortunately, we don't have to deal with this issue since the Blitz LINE command handles it for us. Here's the format of the LINE command:

Line *StartX, StartY, EndX, EndY*

To see the LINE command in action, replace the PLOT command in our *Plot Pixel* example with the following line:

```
Line Rnd (0,Screenwidth), Rnd(0,ScreenHeight), ↲
        → Rnd (0,ScreenWidth), Rnd(0,ScreenHeight)
```

Rectangles

To put up rectangles in Blitz, you use the RECT command. This command will allow you to specify whether the rectangle is solid (filled-in with the current color) or hollow (only outlined in the current color). You can make the rectangles as large or small as you want as well. Here is the layout for this command:

Rect (*StartX,StartY,Width,Height,SolidFlag****)***

Setting the *SolidFlag* to zero will make the rectangle hollow, whereas setting it to 1 will make it filled. The following code will draw a bunch of unfilled and filled rectangles all over the screen.

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

 ; Initialize Blitz Basic
 Graphics ScreenWidth, ScreenHeight

 ; set the seed for the random number generator
 SeedRnd MilliSecs ()

 ; save our current colors
 Red = ColorRed ()
 Green = ColorGreen ()
 Blue = ColorBlue ()

 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)

   ; reset the color so our text doesn't change colors
   Color Red, Green, Blue

   ; print out our heading
   Text 0,0,"Rectangle Demo.  Hit <ESC> to Quit."
```

```
  ; randomize a color
  Color Rnd (0,255), Rnd(0,255), Rnd(0,255)

  ; draw a rectangle in a random location
  Rect (Rnd (0,ScreenWidth), Rnd(0,ScreenHeight), ↲
      → Rnd (0,ScreenWidth), Rnd(0,ScreenHeight), ↲
      → Rnd (0,1))
Wend

End ; end the program
```

You can also copy rectangles by using the COPYRECT command. The format of this command is as follows:

CopyRect *SrcX,SrcY,Width,Height,DestX,DestY,SrcBuff,DestBuff*

The *SourceBuff(er)* and *DestBuff(er)* arguments are optional, but they give you the capability to copy from one buffer to another. Try modifying the above code to draw a single rectangle and then copy from it to the screen. Use the *CopyPixel* demo for ideas on how to do this.

Ovals

The final primitive we're going to discuss is the oval. Ovals can be used to make a lot of circular shaped objects, such as buttons, bullets, etc. To create an oval, use the OVAL command. Here is the format:

Oval(*StartX, StartY, Width, Height, SolidFlag****)***

Notice that this command is setup identically to the RECT command. Thus, if you change the RECT command in the previous code example to OVAL, you'll see ovals in action. Give it a try!

```
Oval(Rnd (0,ScreenWidth), Rnd(0,ScreenHeight), ↲
    → Rnd (0,ScreenWidth), Rnd(0,ScreenHeight),Rnd(0,1))
```

Chapter 12: Displaying Images

Now I know you've been waiting to get to this part of the book, but keep in mind that everything in Section 1 is extremely important for you to understand in order to make games with Blitz. You'll be spending the majority of your development time working with algorithms, only using images to convey your game's premise. So make sure you understand all that's gone on up to now!

From the player's point of view, graphics are the life of the game. Whether stunningly beautiful or ruggedly crude, the images you display will set the tone for your game. You don't have to be an amazing artist to create amazing games either. I would say that there are a good number of games that have great game play, but not so great artwork. But you should do the best artwork you can, or consider working with an artist that has decent skills. The worst thing you could do is make a game that nobody even gives a second glance to because the artwork is really poor. Be as picky with your art as you are with your code…and be *very* picky with your code.

Basic Loading and Displaying of Images

Let's start out with loading an image and displaying it. No animation at this point, we just want to load something in and draw it up on the screen.

To load the image, we'll need to use the Blitz command LOADIMAGE. Pretty intuitive, no? At the time of this writing, Blitz only supports the loading of *.BMP, .JPG,* and *.PNG* files. If you use some other file type, Blitz will return that it could not load the file. Here is the format of the LOADIMAGE command:

ImageHandle **= LoadImage (***ImageName***)**

The *ImageHandle* is a value that Blitz will use to keep track of where the image is in memory. You will need to use this value whenever you wish to display the image or to free the image's memory. If *ImageHandle* contains a zero, it means that Blitz was unable to load the image, so make sure you process accordingly. *ImageName* is the name of the image, with its full path.

After loading the image, you will want to draw it to the screen using the DRAWIMAGE command. DRAWIMAGE takes the *ImageHandle* and draws the associated image at the specified X, Y coordinates.

DrawImage *ImageHandle, X, Y*

If you specify an invalid *ImageHandle*, Blitz will break out with an error. So, be safe and check that LOADIMAGE was successful when it attempted to load the image. Here is a piece of code that loads and draws up an image (you can either create a *.PNG* file called "test.png" in the same directory that you are running this program, or just use the example program in the appropriate chapter folder. Alternately, you can use .JPG or .BMP, but make sure you change the LOADIMAGE line in the code below):

```
; Initialize Blitz Basic to 640x480 resolution
 Graphics 640,480

 ; Load in an image
 ImageHandle = LoadImage ("test.png")

 ; verify that the image loaded properly
 If ImageHandle <> 0
   ; draw the image
   DrawImage ImageHandle,0,0
 Else
   ; tell the user that we couldn't load the image
   Text 0,0,"Could not load the image!"
 EndIf

WaitKey () ; wait for a keypress
End ; end the program
```

Using this method you can load up any supported image type and display it at any X, Y coordinate you want, but keep in mind that if you display it at a coordinate beyond the range of your screen resolution, you won't see the image. This is because Blitz will automatically *clip* anything outside of the visual field. The term "clip" is used to mean *not* drawing anything off the visual field.

Another thing to think about when drawing images is *transparency*. Transparency simply means that Blitz will draw all of the pixels of the image with the exception of a color (called a *Mask*) that YOU select by calling the MASKIMAGE command. The format of this command is:

MaskImage *ImageHandle,RedColor,GreenColor,BlueColor*

Each color value can be between 0 and 255, and the default is 0 for all three…which is black. So the default *mask* is black.

If you wanted to draw all the pixels, meaning using NO transparency, you could use the DRAWBLOCK command. It's used identically to DRAWIMAGE

but it doesn't take into account any transparency factors. Thus, you will see a black (or whatever color) box around your images.

Rotating an Image to Make Multiple Frames

One of the things that a lot of games do is to have a 2D graphic that rotates around its mid-point. For example, let's say that you're making a 2D space game (why not, everyone does!). You could make your player's ship by drawing it at all the different angles by hand (ouch!). Or you could draw it facing straight up and then use your graphics program to rotate each frame and then place the frames together (not too bad). Or maybe draw it facing straight up, have BB load it and do all the rotations for you (ah...that's better).

Now this method isn't going to work in all situations. If, for example, you had the same image that either changed sizes real-time or had different light-sources depending on the angles (and that light-source was not dynamic), you couldn't use this method. But I've used this for a number of demos and games without hesitation.

Just like the previous example, you're going to need to load the image you want to rotate and you're going to need a variable to point to it. From there, you'll use a combination of the COPYIMAGE command and the ROTATEIMAGE command. You may also need to use the MASKIMAGE command if you have a mask color that is not black. Also, you should set BB's AUTOMIDHANDLE to TRUE. This will ensure that your images are rotated at their center point. If you don't use this, your images will rotate around the top-left corner. Take out the AUTOMIDHANDLE TRUE section to see what I mean.

Remember that the mask color is used to tell Blitz NOT to draw any pixels (dots on the image) of that color. This is so your image will appear transparent. You wouldn't want to have this cool looking ship drawn with a big black rectangle all around it, so you'll need to tell BB to not draw certain colors. Again, the default for transparency is black (or Red=0,Green=0, Blue=0), but a lot of people use a really ugly color (like bright magenta) to ensure they won't accidentally use the transparent color as part of the actual art.

Even though the image I'll be loading will use black as the mask, the following source code will use the MASKIMAGE function for demonstration purposes. Here's the code to demonstrate this rotation:

```
; Initialize BB to run at 800x600
Graphics 800,600

;how many rotations do we want?
```

```
Const iNumRotations=36

;Holds the rotated images
Dim ShipFramesImage(iNumRotations)

Global iCurrentFrame = 0    ;current frame displayed

; tell BB to handle the centering of our images
AutoMidHandle True

; Load the player image and point to it with "TempImage"
TempImage = LoadImage ("ship.png")

; See if the image was loaded successfully
; (the "TempImage" will be 0 if not)
If TempImage = 0 Then
  Text 100,100,"Invalid Image!"
 Else
  ; set it's mask (transparent color)
  MaskImage TempImage,0,0,0

  ; now run through the loop and rotate the image
  For iLoop=0 To iNumRotations-1
      ; first copy the original image into the current frame
      ShipFramesImage(iLoop)=CopyImage ( TempImage)
      ; rotate frame the appropriate number of degrees
       RotateImage ShipFramesImage(iLoop), ↲
                  → iLoop*360/iNumRotations
  Next
EndIf

; loop until the player hits the ESC key
While Not KeyHit (1)

 ; clear the screen
 Cls

 ; display current Ship Frame in the center of the screen
 DrawImage (ShipFramesImage(iCurrentFrame),400,300)

 ; change the current frame (spin it to the right)
 iCurrentFrame = iCurrentFrame + 1

 ; if the current frame counter goes past our number of
 ; allowable rotations, reset it to 0
```

```
  If iCurrentFrame > iNumRotations - 1
    iCurrentFrame = 0
  EndIf

  ; delay for 50 milliseconds so we can see the animation
  Delay 50
Wend
End ; end the program
```

So this basically loads in a ship and spins it around on the center of the screen. Mess around a bit with the delay speeds and the iNumRotations variables to see the weird things you can do.

Grabbing Images from Memory

You may have a situation where you'll need to grab a piece of the screen and save it off to its own image which you can then manipulate and so on. For example, let's say that you have the ability to grab an engine component and move it to your inventory. You can either make it so it instantly appears, or you can make it so you determine where in your inventory it will appear by moving it and clicking the inventory location.

While you can certainly do this type of thing by just referencing the original image, let's instead use the CREATEIMAGE and GRABIMAGE commands to see how this is done.

In the following example, I'm going to plop a ball up on the screen and move it around. Every couple of seconds I'll plop up a new ball that has been copied from the original. After a few more seconds the ball will be deleted. I'll use FREEIMAGE to remove the image itself and since I'll also be using a TYPE, I'll delete the node from that as well.

Note that the use of FREEIMAGE is extremely important. If you don't use this command, you'll begin eating up tons of memory and that will eventually cause problems.

```
; setup our screen constants
Const ScreenWidth = 640
Const ScreenHeight = 480

; Initialize Blitz Basic
Graphics ScreenWidth, ScreenHeight

; setup a type to hold the elements
```

```
Type Balls
    Field Image          ; holds the copied image
    Field X              ; x location of the ball
    Field Y              ; y location of the ball
    Field TimePlaced     ; when was it placed on the screen?
    Field TimeToHold     ; how long before we remove it?
End Type

; load up the ball
BallImage = LoadImage ("ball.png")

; set some base values for X,Y for the original ball
BallX  = 50
BallY = 100

; get the size values of the image
BallWidth = ImageWidth(BallImage)
BallHeight = ImageHeight(BallImage)

; Setup timer for creating new elements
CreateElementTime = 1000
; setup another timer to track the current time
NewElementTimer = MilliSecs ()

; while the user doesn't hit <ESC>, which is '1'
While Not KeyHit (1)
 Cls
 ; print out our Bank info
 Text 0,0,"Hit <ESC> to Quit."

 ; go through the list and display each ball
 For Ball.Balls = Each Balls
     ; if the ball's time has expired, free and remove it
     If MilliSecs () > Ball\TimePlaced + Ball\TimeToHold
        FreeImage Ball\Image
        Delete Ball
     Else
        ; otherwise draw the image
        DrawImage (Ball\Image,Ball\X,Ball\Y)
     EndIf
 Next

 ; draw the image at the current BallX,BallY position
 DrawImage (BallImage,BallX,BallY)
```

```
  ; if CreateElementTimer is triggered, create new element
  If MilliSecs () > NewElementTimer + CreateElementTime
     Ball.Balls = New Balls
     If Ball.Balls <> Null
        ; create the image buffer
        Ball\Image = CreateImage (BallWidth,BallHeight)
        ; now grab the image from the current screen
        GrabImage Ball\Image,BallX,BallY
        ; setup the rest of the values
        Ball\X = Rnd (0,600)
        Ball\Y = Rnd (0,400)
        Ball\TimePlaced = MilliSecs ()
        Ball\TimeToHold = Rnd (500,5000)
     EndIf
     ; make sure to reset the NewElementTimer!
     NewElementTimer = MilliSecs ()
  EndIf
  ; delay a bit to avoid flicker as much as possible
  Delay(30)
Wend
End ; end the program
```

Whenever you use CREATEIMAGE, Blitz allocates a chunk of memory. Then when GRABIMAGE is called, it takes the pixels from the screen and moves them into the newly created image buffer. Be careful to use GRABIMAGE before you do a CLS (or overwrite the image you're grabbing) or you will have unexpected results.

Image Buffers

Every time you load in a new image, grab an image, create an image, etc. you are really creating an image *buffer*. This is just a piece of memory that holds an image. That's really all there is to it.

You can manipulate this buffer, drawing to it at will, by setting the current buffer to be that image. For example, we loaded an image into *BallImage* in our previous example. If we wanted to draw something to that image, say a line, all we would need to do is the following:

```
; set the buffer that Blitz will draw to
SetBuffer ImageBuffer(BallImage)

; draw a couple of white lines to make an X
Color 255,255,255
Line 10,10,20,20
```

```
Line 20,10,10,20

; reset the buffer to be our drawing buffer
SetBuffer FrontBuffer ()
```

If you neglect to reset your buffer, you'll have some very interesting results, so keep this in mind with directly accessing buffers.

There are a ton of reasons for manipulating buffers directly. For example, what if you wanted to show damage on your ship each time a bullet hit it? You could draw a dark pixel to the spot that it was hit. And maybe over time the spot fades back to its original color because you are doing repairs. You could also inscribe the player's name on the hood of a car or maybe allow the player to place specific designs to customize their character. Again, the number of options here is limitless, so get used to playing with buffers directly.

Quick and Dirty Animation

Now let's have some fun. I'm going to move the ball from one side of the screen to the next, bouncing it off the edges and such.

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

 ; Initialize Blitz Basic
 Graphics ScreenWidth, ScreenHeight

 ; load up the ball
 BallImage = LoadImage ("ball.png")

 ; set some base values for X,Y and speeds
 BallX = 0
 BallY = 0
 BallXIncrement = 3
 BallYIncrement = 2

 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)
   ; clear the screen
   Cls
   ; print out our Bank info
   Text 0,0,"Bouncing Ball Demo.  Hit <ESC> to Quit."
```

```
  ; add/subtract current XIncrement to the ball's X position
  BallX = BallX + BallXIncrement

  ; if the ball minus its width is greater than the screen
  If BallX > ScreenWidth - ImageWidth(BallImage)
    ; start subtracting during each loop
    BallXIncrement = -3
  EndIf
  ; if the ball is less then 0 on the screen
  If BallX < 0
    ; start adding during each loop
    BallXIncrement = 3
  EndIf

  ; add/subtract current YIncrement to the ball's Y position
  BallY = BallY + BallYIncrement
  ; if the ball minus its Height is greater than the screen
  If BallY > ScreenHeight - ImageHeight(BallImage)
    ; start subtracting during each loop
    BallYIncrement = -2
  EndIf
  ; if the ball is less then 0 on the screen
  If BallY < 0
    ; start subtracting during each loop
    BallYIncrement = 2
  EndIf

  ; draw the image at the current BallX,BallY position
  DrawImage (BallImage,BallX,BallY)

  ; delay a little bit so we can see the ball
  Delay(30)
Wend

End ; end the program
```

The first thing to notice is that I've set up two constants at the very top of the code to assign the screen's width and height values to. I do this because in most games you will be gauging how to handle animations based on the screen's dimensions. If you hard code these numbers then you'll need to change them all throughout your program, which can be quite a hassle. If you set them up as CONST then Blitz will incorporate them at compile time and you'll see no speed degradation. Plus you only need to change these values in ONE spot when doing updates and testing.

What makes this example work is the IF…THEN…ENDIF sections. Since, in each loop iteration, we are adding a value to BallX and BallY, we can control if that value added is positive or negative. Since a negative number added to a positive number is the equivalent of subtraction, we can reverse the course of the ball by simply changing the sign of the corresponding increment.

Chapter 13: Animation Techniques

In the previous chapter we animated a ball around on a screen, but you probably noticed that the ball flickered and jumped a lot. This is because of the type of animation I elected to use was Blit animation. This chapter discusses two forms of animation and describes how to use them.

Screen Blit Animation

This form of animation actually allows you to write to the screen's buffer real-time. This used to be the way that games were made, especially at 320x200 resolutions, but it's not as common today. The idea is that the monitor is refreshed a certain number of times per second. If we catch the monitor halfway through its refresh phase, we'll see flickering in our images because you'll see half the screen drawn the first time and the second half the second time. This looks bad.

Blitz provides a command called VWAIT. This command tells your program not to do anything until the Vertical Sync is on. In other words, when the monitor has completed refreshing the screen and is going back up top to start its refresh again, Blitz will know this and will tell your program to quickly draw everything.

Our Ball example can be updated with this command easily. Simply put the following piece of code above the CLS:

VWait

Now it will wait until the monitor is about to refresh before clearing the screen and re-drawing the ball.

Page Flip Animation

Page flipping is utilized most often in today's games because it's a way to ensure you're not going to get flicker. The concept is to have a piece of memory set aside (preferably video memory, for speed reasons) that is laid out exactly like your primary video memory (or screen buffer). So, you'd have a primary (front) buffer and a secondary (back) buffer. The back buffer has a duplicate layout of the primary buffer. The primary buffer is also known as the front buffer.

The idea is that while your front buffer is displayed to the user, you get busy drawing on the back buffer. This way you are not drawing anything to the main screen while the user watches. When you have completed your drawing

you FLIP the two pages. So, now your front buffer becomes your back buffer and your back buffer becomes your front buffer. Since this simply tells the video card to point to a new place in video memory when doing its refresh, it's instant. Plus Blitz makes sure the Vertical Retrace is accounted for when doing the FLIP.

To further clarify, imagine a film reel. Each frame is slightly different than the last one and they run through the projector pretty quickly. As they run through the projector, the viewer sees animation. Consider page-flipping as nearly the same thing, with the exception that instead of running a bunch of frames through a projection piece you instead create a new frame and display it on that projection piece.

There are three functions that you'll need to use in order to do page flip animation. They are the SETBUFFER, BACKBUFFER, and the FLIP commands. The SETBUFFER command tells Blitz where to draw. BACKBUFFER returns a pointer to the back buffer we discussed above. The FLIP command tells Blitz that we're ready to swap our front and back buffers.

Here is the bouncing ball code with the update of using page-flip animation:

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

 ; Initialize Blitz Basic
 Graphics ScreenWidth, ScreenHeight

 ; Setup the Backbuffer for page flipping
 SetBuffer BackBuffer ()

 ; load up the ball
 BallImage = LoadImage ("ball.png")

 ; set some base values for X,Y and speeds
 BallX = 0
 BallY = 0
 BallXIncrement = 3
 BallYIncrement = 2

 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)
   ; clear the screen
   Cls
   ; print out our Bank info
```

```
  Text 0,0,"Bouncing Ball Demo.  Hit <ESC> to Quit."

  ; add/subtract current XIncrement to the ball's X position
  BallX = BallX + BallXIncrement

  ; if the ball minus its width is greater than the screen
  If BallX > ScreenWidth - ImageWidth(BallImage)
    ; start subtracting during each loop
    BallXIncrement = -3
  EndIf
  ; if the ball is less then 0 on the screen
  If BallX < 0
    ; start adding during each loop
    BallXIncrement = 3
  EndIf

  ; add/subtract current YIncrement to the ball's Y position
  BallY = BallY + BallYIncrement
  ; if the ball minus its Height is greater than the screen
  If BallY > ScreenHeight - ImageHeight(BallImage)
    ; start subtracting during each loop
    BallYIncrement = -2
  EndIf
  ; if the ball is less then 0 on the screen
  If BallY < 0
    ; start subtracting during each loop
    BallYIncrement = 2
  EndIf

  ; draw the image at the current BallX,BallY position
  DrawImage (BallImage,BallX,BallY)

  ; now flip the buffers
  Flip
 Wend

End ;end the program
```

Notice how much more smoothly the animation looks here. It's fast and it doesn't flicker! This is the type of animation that you should consider using for your games.

All we did differently was add the FLIP and SETBUFFER BACKBUFFER() commands. From there Blitz is smart enough to know that we want to use page-flip animation. If you remove the FLIP command, you'll see a blank

screen until you hit Escape. This is because all of the drawing is done on the back buffer, but it's never switched over to show the user what has been changed.

If you get rid of the CLS command you'll see a bunch of artifacts all over the screen. This command is important because it will first clear the back buffer and then draw everything in its updated position.

For fun let's add in a TYPE value that will keep track of a bunch of bouncing balls. We'll create them on the fly in a function, randomizing their positions and speeds as we go, and then animate them all over the screen.

```
; setup our screen constants
Const ScreenWidth = 640
Const ScreenHeight = 480

; Initialize Blitz Basic
Graphics ScreenWidth, ScreenHeight

; Setup the Backbuffer for page flipping
SetBuffer BackBuffer ()

; Initialize the random number generator
SeedRnd MilliSecs ()

; load up the ball
Global BallImage = LoadImage ("ball.png")

; Setup our Ball type
Type Balls
   Field X          ; x position of the ball
   Field Y          ; y position of the ball
   Field XIncrement   ; XIncrement of the current ball
   Field YIncrement   ; YIncrement of the current ball
End Type

; create a global pointer to the Balls Type
Global Ball.Balls

; call the function to create the balls
Status = CreateBalls(50)

; if the function failed, end the program
If Status = -1
  RuntimeError("Memory not allocated in <CreateBalls>")
```

```
  End
EndIf

; while the user doesn't hit <ESC>, which is '1'
While Not KeyHit (1)
 ; clear the screen
 Cls
 ; print out our Bank info
 Text 0,0,"Bouncing Ball Demo.  Hit <ESC> to Quit."

 ; run through all the instances of the type
 For Ball.Balls = Each Balls
    ; add/subtract current XIncrement to ball's X position
    Ball\X = Ball\X + Ball\XIncrement

    ; if the ball minus its width is greater than the screen
    If Ball\X > ScreenWidth - ImageWidth(BallImage)
      ; start subtracting during each loop
      Ball\XIncrement = 0 - Ball\XIncrement
    EndIf

    ; if the ball is less then 0 on the screen
    If Ball\X < 0
      ; start adding during each loop
      Ball\XIncrement = Abs(Ball\XIncrement)
    EndIf

    ; add/subtract current YIncrement to ball's Y position
    Ball\Y = Ball\Y + Ball\YIncrement
    ; if the ball minus its Height is greater than the screen
    If Ball\Y > ScreenHeight - ImageHeight(BallImage)
      ; start subtracting during each loop
      Ball\YIncrement = 0 - Ball\YIncrement
    EndIf
    ; if the ball is less then 0 on the screen
    If Ball\Y < 0
      ; start subtracting during each loop
      Ball\YIncrement = Abs(Ball\YIncrement)
    EndIf

    ; draw the image at the current BallX,BallY position
    DrawImage (BallImage,Ball\X,Ball\Y)
 Next

 ; now flip the buffers
```

```
  Flip
 Wend

End ; end the program
 ;*********************************************
 ;  Function: CreateBalls()
;    Purpose: To create a bunch of random objects
 ;       Args:  how many objects to create
 ;    Returns:  0=success, -1=failure
;*********************************************
 Function CreateBalls(NumberToCreate)
   ; loop through and create new instances
   For Number = 0 To NumberToCreate
      Ball.Balls = New Balls
      ; if we got the memory, assign the random values
      If Ball.Balls <> Null
        Ball\X = Rnd(0,ScreenWidth - ImageWidth(BallImage))
        Ball\Y = Rnd(0,ScreenHeight - ImageHeight(BallImage))
        Ball\XIncrement = Rnd(1,10)
        Ball\YIncrement = Rnd(1,10)
      Else
        ; otherwise, return that we have an error
        Return -1
      EndIf
   Next

   Return 0 ; return that all went well
 End Function
```

See if you can make it so the balls all start out going in different directions. A hint on that is that you'll need some of them starting out having negative X/Y increments, so you'll want to randomize that option maybe using an IF statement.

Animating Images

It's pretty likely that you'll want to have an image animate in and of itself. For example, what fun is it to have a character running across the screen without its arms and feet moving about? Or how about a car that doesn't have spinning tires?

So in addition to moving stuff around the screen, you also need to think about the various frames of the image as it changes. For example, take the following image in Figure 13.1:


(Figure 13.1)

This is a *very* basic image that demonstrates what I'm talking about. If you show these images in a successive display, your mind will perceive a spinning wheel.

In the following example we take the bouncing ball demo and put in the spinning ball. Either create a 64x32 image and make it look like the above image, or use the image off the CD. Don't put any spaces between the two objects, though, because the LOADANIMIMAGE function will read the *exact* amount of pixels you pass to the function. Here's the code:

```
; setup our screen constants
Const ScreenWidth = 640
Const ScreenHeight = 480

; Initialize Blitz Basic
Graphics ScreenWidth, ScreenHeight

; Setup the Backbuffer for page flipping
SetBuffer BackBuffer()

; Initialize the random number generator
SeedRnd MilliSecs()

; load up the ball
Global BallImage = LoadAnimImage ("balls.png",32,32,0,2)

; Setup our Ball type
Type Balls
   Field X          ; x position of the ball
   Field Y          ; y position of the ball
   Field XIncrement   ; XIncrement of the current ball
   Field YIncrement   ; YIncrement of the current ball
   Field CurrentFrame ; What frame is showing?
End Type

; create a global pointer to the Balls Type
Global Ball.Balls

; call the function to create the balls
Status = CreateBalls(50)
```

```
; if the function failed, end the program
If Status = -1
  RuntimeError("Memory not allocated in <CreateBalls>")
  End
EndIf

; while the user doesn't hit <ESC>, which is '1'
While Not KeyHit (1)
 ; clear the screen
 Cls
 ; print out our Bank info
 Text 0,0,"Bouncing Ball Demo.  Hit <ESC> to Quit."

 ; run through all the instances of the type
 For Ball.Balls = Each Balls
    ; add/subtract current XIncrement to ball's X position
    Ball\X = Ball\X + Ball\XIncrement

    ; if the ball minus its width is greater than the screen
    If Ball\X > ScreenWidth - ImageWidth(BallImage)
      ; start subtracting during each loop
      Ball\XIncrement = 0 - Ball\XIncrement
    EndIf

    ; if the ball is less then 0 on the screen
    If Ball\X < 0
      ; start adding during each loop
      Ball\XIncrement = Abs(Ball\XIncrement)
    EndIf

    ; add/subtract current YIncrement to ball's Y position
    Ball\Y = Ball\Y + Ball\YIncrement
    ; if the ball minus its Height is greater than the screen
    If Ball\Y > ScreenHeight - ImageHeight(BallImage)
      ; start subtracting during each loop
      Ball\YIncrement = 0 - Ball\YIncrement
    EndIf
    ; if the ball is less then 0 on the screen
    If Ball\Y < 0
      ; start subtracting during each loop
      Ball\YIncrement = Abs(Ball\YIncrement)
    EndIf

    ; change the current frame that we're drawing
```

```
    Ball\CurrentFrame = Ball\CurrentFrame + 1
    If Ball\CurrentFrame > 1
      Ball\CurrentFrame = 0
    EndIf

    ; draw the image at the current BallX,BallY position
    ; using the current frame
    DrawImage (BallImage,Ball\X,Ball\Y, Ball\CurrentFrame)
  Next

  Flip  ; flip the buffers
Wend

End ; end the program

;***********************************************
;  Function: CreateBalls()
;   Purpose: To create a bunch of random objects
;         Args:  how many objects to create
;      Returns:  0=success, -1=failure
;***********************************************
Function CreateBalls(NumberToCreate)
   ; loop through and create new instances
   For Number = 0 To NumberToCreate
      Ball.Balls = New Balls
      ; if we got the memory, assign the random values
      If Ball.Balls <> Null
        Ball\X = Rnd (0,ScreenWidth - ImageWidth(BallImage))
        Ball\Y = Rnd (0,ScreenHeight - ImageHeight(BallImage))
        Ball\XIncrement = Rnd (1,10)
        Ball\YIncrement = Rnd (1,10)
        Ball\CurrentFrame = Rnd (0,1)
      Else
        ; otherwise, return that we have an error
        Return -1
      EndIf
   Next

   Return 0 ; return that all went well
End Function
```

After you've run this you may be surprised to see that the images animate so fast that you can't even see the animation! A quick way to fix this so you can see the animation in action is to put the following line *after* the FLIP command:

```
Delay(50)
```

That will slow down the processing enough to allow everything to be visible. But there is a big problem with this method: it not only slows down your animation but your *entire* game!

Animation Timing

So how do you control the speed at which an image animates? I don't mean how do you control how fast it moves, but literally how long it is between one frame of the image and the next. This is a key issue because you may have many things animating at different rates on your screen, and you'll need a way to keep track of them all.

Fortunately, Blitz provides some timing commands that we can use to control animation speeds. You'll need to add a few *fields* to your TYPE in order to make this effective though. Let's take our *Balls* TYPE and add to it:

```
; Setup our Ball type
 Type Balls
    Field X          ; x position of the ball
    Field Y          ; y position of the ball
    Field XIncrement   ; XIncrement of the current ball
    Field YIncrement   ; YIncrement of the current ball
    Field CurrentFrame ; What frame is showing?
    Field FrameChangeTimer ; time between frame changes?
    Field LastChangedTime     ; last frame change time?
 End Type
```

You can use whatever field names you want, of course, but these seem to convey the point clearly to me so I'll stick with them.

Next we'll need to assign a value to the *FrameChangeTimer*, and we'll need to use the MILLISECS command to get the current time after each animated frame. In our *CreateBall* function, then, we'll need to include the following two lines:

```
Ball\FrameChangeTimer = Rnd (30,100)
Ball\LastChangedTime = MilliSecs ()
```

And then alter the section of code that handles the frame changing as follows:

```
; check to see if the current milliseconds are greater than
```

```
; the last change value plus the amount of time to change
If MilliSecs () > Ball\LastChangedTime + Ball\FrameChangeTimer
        ; change the current frame that we're drawing
        Ball\CurrentFrame = Ball\CurrentFrame + 1

        If Ball\CurrentFrame > 1
           Ball\CurrentFrame = 0
        EndIf

        ; make sure to reset the LastChangedTime!
        Ball\LastChangedTime = MilliSecs ()
EndIf
```

If you make these changes, you'll notice some of the balls spinning faster than others. This is exactly the kind of thing you'll need for your games!

Animation Efficiency

Every time an image is displayed, it covers up something in the background. To give the appearance that an image is moving non-destructively over the background, you must somehow replace the background after each frame. The image moving across the screen non-destructively, by the way, is called a *Sprite*.

You can think of a mouse cursor as a sprite, to help picture this effect. When a mouse cursor moves over the icons on your main screen, it does not erase those icons. This is because the image is being restored each frame.

In Blitz, there are two ways to handle this sprite effect. The first way is to use the GRABIMAGE and DRAWIMAGE commands to snag each chunk from the background *before* drawing the mouse image, then to replace that chunk after the loop. Here are the steps:

- Replace the "grabbed" background piece before rendering, using DRAWIMAGE.
- Grab the new background piece using GRABIMAGE (Note: You will first need to use CREATEIMAGE in order to setup the proper size requirements for the GRABIMAGE command.)
- Render the screen and draw the mouse in its new position

It is a little tricky to do this, but more than tricky this can be quite slow since you will be reading from and writing to the video's memory a lot. Reading from video memory is the problem...it's very slow! According to my tests and the comments of most Blitz coders, re-drawing the screen during each rendering cycle is much quicker than doing many reads. If you follow the

grab-chunk, draw an image, replace-chunk method, you'll likely be tempted to do this with all of your images. This will take your speedy little game down to a crawl. Instead, either clear the screen or just redraw everything each frame. It’s much faster this way, and tremendously less of a headache.

Chapter 14: Collision Detection

This particular topic has been the nemesis of many a hobbyist game programmer. The concept of determining when two objects overlap seems to be, on the surface, a simple thing to check. In practice, though, this can be quite a difficult accomplishment.

Bounding Box Collisions

The problem arises in that a graphical object is a square. Sure, it may *look* like a circle, but computers don't display images as circles…rather they display them as squares. Take another look at the *balls* image, as an example (Figure14.1):

(Figure 14.1)

See how you can separate those two images into squares? They touch only on the left-hand side, but when you go to display the image, Blitz is really drawing out a square. It's just ignoring the black pixels because the default *mask* is black (unless you use the DrawBlock command).

(Figure 14.2)

Here we have two circles that are not touching. Thus there is no collision. However, all we have to do is overlap the two black edges and it could be considered a collision. This is because of how computers handle images. They are squares regardless of the shape the non-black (or other mask color) pixels are. The following graphic demonstrates this concept:

(Figure 14.3)

Those two images are overlapping since the squares that contain them are touching. This is why in some games you'll see an explosion before a missile hits a ship, for example.

The following two graphics show something else interesting in regards to this method of collision detection. The first graphic shows a cheesy little rocket with an even cheesier little bullet sitting off to the right of it. There is no collision here, of course:

(Figure 14.4)

However, I'll now move the bullet to sit right next to the rocket:

(Figure 14.5)

It's not actually hitting the rocket, but since the two boxes overlap, Blitz will respond that a collision has taken place.

This type of collision detection is known as the "bounding box" method. It's used because it's fast. It's in no way accurate, but it is fast. And there are times where this method is the best choice, such as if you have a game made of blocks that smack into each other or something.

The IMAGESOVERLAP command is used to check if two images are overlapping on their respective bounding-boxes. Here is the format of this command:

ImagesOverlap (*Image1,X,Y,Image2,X,Y****)***

Taking the bouncing ball example again, we can run through the *Balls* TYPE and check each element for overlap, as follows:

```
; run through all the instances of the type
  For Ball.Balls = Each Balls
      ; go through again using a different pointer (BallTest)
      For BallTest.Balls = Each Balls
```

```
            ; make sure it isn't our current Ball element...
            If BallTest <> Ball
               ; check for images overlapping
               ; add to a counter if so
               If   ImagesOverlap (BallImage, Ball\X, Ball\Y, ↲
                                  → BallImage, BallTest\X, ↲
                                  → BallTest\Y)
                   CollisionCounter = CollisionCounter + 1
               EndIf
            EndIf
      Next
 Next
```

And that's all there is to it. Add this snippet into the bouncing ball code and add this line under the first TEXT line:

```
Text 0,16, "Number of Collisions: " + CollisionCounter
```

Now each time the images overlap, you'll see the collision counter move up. Keep in mind that as the images pass over each other they will continue to increment until they are no longer overlapping.

Pixel-Perfect Collision Detection

To get an accurate collision, you must use a more accurate method of detection. The benefit is obvious, no more hits happening that don't really "hit" the image. The drawback is that Blitz will have to check each non-transparent pixel to see if the pixel overlaps with a pixel of another image, but it needs to first check each pixel to see if it's even transparent! This can slow things down a bit. Fortunately Blitz first checks to see if the images even overlap before doing this, so there aren't any unnecessary checks.

What we want is a collision to be triggered only when the non-transparent pixels actually touch, right? Right! So, the following graphic would demonstrate a collision we'd be happy with:

(Figure 14.6)

That little bullet is actually touching the rocket now! I've got great news for you too…it's easy to accomplish this in Blitz. Just use the IMAGESCOLLIDE

command. It's nearly identical to the IMAGESOVERLAP command in how it's called, but there are two important argument additions. Here's the format:

ImagesCollide (*Image1,X,Y,Frame,Image2,X,Y,Frame****)***

The added arguments are for the frame of the object. This is important to note because as the image changes frames, so will the pixels to compare against. If you don't take into account the current frame to check on, you'll see hits when there aren't really hits. This is because the frame being displayed may not actually be the frame being checked for collision. So, be safe and make sure you're checking the displayed frame when doing pixel-perfect detection.

The code looks as follows:

```
; run through all the instances of the type
  For Ball.Balls = Each Balls
       ; go through again using a different pointer (BallTest)
       For BallTest.Balls = Each Balls
            ; make sure it isn't our current Ball element...
            If BallTest <> Ball
               ; check for images colliding (pixel perfect),
               ; add to a counter if so
               If   ImagesCollide (BallImage, Ball\X, Ball\Y, ↲
                                  → Ball\CurrentFrame, ↲
                                  → BallImage, BallTest\X, ↲
                                  → BallTest\Y, ↲
                                  → BallTest\CurrentFrame)
                    CollisionCounter = CollisionCounter + 1
               EndIf
            EndIf
       Next
  Next
```

Since we are already tracking the current frame, we just add that into the IMAGESCOLLIDE call and voilà, we're done.

Chapter 15: Handling Input

There are a number of devices that your players *can* use with your game, but you have control of which ones you'll support.

Using the Keyboard

When moving a ship around, firing, etc., you'll want to use a keyboard routine that keeps track of when a key is held down. While you can certainly use the KEYHIT command for checking on one-time hit keys such as Escape, you'll need something a little more robust for real-time stuff.

This is where the KEYDOWN command comes in. All this function does is return a TRUE or FALSE response when asked if a particular key is being held down. In order for you to send it a particular key to check, you need to know the *scancode*. This is a code that the computer recognizes the key by. Generally what I do is find the code (which can be found in the command reference section of the Blitz IDE) and make a corresponding constant with a recognizable name. Like this:

```
Const Key_ArrowPad_Left = 203
Const Key_ArrowPad_Rigth = 205
```

And so on. There are a number of libraries out there with all these values in them.

Now I know that all I have to do is remember "Key_ArrowPad_Right," not 205. This makes it easier to read code that way, and since the naming convention is consistent I don't have to look up the name for the up arrow since I know it will be "Key_ArrowPad_Up."

The following code shows a function that I called *CheckKeys*. All it does is checks to see if the left or right arrow is being held down and then spins an image around accordingly by adjusting the CurrentFrame:

```
Function CheckKeys()
  If KeyDown (Key_ArrowPad_Left)
    ; change the current frame (spin it to the left)
    iCurrentFrame = iCurrentFrame - 1

    ; if the current frame counter is less than 0
    ; reset it to the number of rotations - 1
    If iCurrentFrame < 0
      iCurrentFrame = iNumRotations - 1
    EndIf
```

```
  EndIf

  If KeyDown (Key_ArrowPad_Right)
    ; change the current frame (spin it to the right)
    iCurrentFrame = iCurrentFrame + 1

    ; if the current frame counter goes past our number of
    ; allowable rotations, reset it to 0
    If iCurrentFrame > iNumRotations - 1
      iCurrentFrame = 0
    EndIf
  EndIf
End Function
```

KEYDOWN has no delays associated to it, so you will have an immediate response to your key presses. However, values will be placed in a buffer as processed, so if you want an immediate stop to processing if a key is let up, you can use the FLUSHKEYS command. This command clears the keyboard buffer.

Using the Mouse

The next device I'll touch on is the mouse. Since the mouse has way fewer codes to worry about, it's a little easier to check on. However, you may still need a little trickery to get things done.

Let's say that we wanted to duplicate our keyboard routine using the mouse. If the mouse is being run to the left, spin the image left. If to the right, spin it right. This sounds like a simple little thing to do, and it is if you know a tiny trick.

First let's look at the problem. If you check the mouse coordinates using the MOUSEX and MOUSEY commands, you'll find that they will be locked to the screen resolution. This is cool if you are going to do button clicks and such on a screen. But since we want to spin the ship left as the mouse moves left, we'll need to see if the current mouse position is different from the previous check. Since you'll eventually get a MOUSEX position of 0, you'll find after a while that the image will stop spinning.

To solve this we use the MOUSEMOVE command to re-center our mouse after each check. This will ensure that we will always either be equal to, greater than, or less than our last check.

The following code shows a function called *CheckMouse*. All it does is checks to see if the mouse is moving left or right and spins the ship accordingly. Note that the section before *Function CheckMouse()* would really be located at the top of the program, it's here for reference only:

```
; center the mouse on the screen
MoveMouse ScreenWidth/2, ScreenHeight/2

;Save our mouse position for X
Global iCurrentMouseX=MouseX ()

Function CheckMouse()
 If MouseX () < iCurrentMouseX Then
   ; change the current frame (spin it to the left)
   iCurrentFrame = iCurrentFrame - 1

   ; if the current frame counter is less than 0
   ; reset it to the number of rotations - 1
   If iCurrentFrame < 0
     iCurrentFrame = iNumRotations - 1
   EndIf
 EndIf

 If MouseX () > iCurrentMouseX Then
   ; change the current frame (spin it to the right)
   iCurrentFrame = iCurrentFrame + 1

   ; if the current frame counter goes past our number of
   ; allowable rotations, reset it to 0
   If iCurrentFrame > iNumRotations - 1
     iCurrentFrame = 0
   EndIf
 EndIf

 ; reset the mouse to be at the center of the screen
 MoveMouse ScreenWidth/2, ScreenHeight/2

 ; reset iCurrentMouseX to be at the center of the screen
 iCurrentMouseX = MouseX ()
End Function
```

So we've seen that you can use MOUSEX and MOUSEY commands to find the current location of the mouse, and you can use MOUSEMOVE to set the location of the mouse. Now how do you go about getting the mouse clicks and such?

The good news is that the commands are very similar to the keyboard commands. There is a MOUSEHIT command which checks how many times a button was clicked since the last check, and there is a MOUSEDOWN command which returns if a button is currently being held down or not. Here are the formats for both commands:

TimesPressed **= *MouseHit (****ButtonToCheck****)***
IsDown? = ***MouseDown (****ButtonToCheck****)***

The following code will display two text lines that will tell you if a button has been clicked or is being held down, and where its current position is. Note that you'll need to run this program in *debug* mode to see the mouse cursor, since the mouse cursor is turned off in full-screen mode.

```
; setup our screen constants
Const ScreenWidth = 640
Const ScreenHeight = 480

Graphics ScreenWidth, ScreenHeight
SetBuffer BackBuffer ()

LastClicked$ = "None"
HeldDown$ = "None"

; while the user doesn't hit <ESC>, which is '1'
While Not KeyHit (1)
  ; clear the screen
  Cls
  ; print out our mouse position info
  Text 0,0,"Current Mouse Position: " + MouseX () + ", " + MouseY ()

  ; now check to see if any of the buttons have been clicked
  If MouseHit (1)
    LastClicked$ = "Button 1"
  EndIf
  If MouseHit (2)
    LastClicked$ = "Button 2"
  EndIf
  If MouseHit (3)
    LastClicked$ = "Button 3"
  EndIf

  Text 0,16, "Last Button Clicked: " + LastClicked$
```

```
 ; check to see if any of the buttons are being held down
 If MouseDown (1) Or MouseDown(2) Or MouseDown(3)
    HeldDown$ = ""
    If MouseDown (1)
       HeldDown$ = "Button 1 - "
    EndIf
    If MouseDown (2)
       HeldDown$ = HeldDown$ + "Button 2 - "
    EndIf
    If MouseDown (3)
       HeldDown$ = HeldDown$ + "Button 3"
    EndIf
 Else
    HeldDown$ = "None"
 EndIf

 Text 0,32, "Button Held Down: " + HeldDown$

  Flip ; flip the buffers
 Wend
End
```

Another method of seeing which button was recently *hit* (not held down) is the GETMOUSE command. This command returns the number of the button that was clicked assuming there was a click. So you don't have to request for a particular button check.

As with the keyboard commands, to flush the mouse buffer you would use the FLUSHMOUSE command. This will ensure no leftover mouse clicks are kept if you don't want them.

Another option you have, as with the WAITKEY command for the keyboard, is the WAITMOUSE command. This command will halt the execution of the program until a mouse button is clicked.

Displaying a Custom Mouse Cursor

When you go from using the Blitz IDE debug window for your game to using the Full-Screen method, you'll soon see that your mouse image is no longer visible. You can still keep track of the mouse position and click, but there's no image the user can reference for position.

The problem is that where Windows has the mouse functionality built in for displaying, saving the background and restoring the background...Blitz does not. See, where the mouse cursor appears to be a graphic image that

magically moves over the background, there's a lot more going on underneath the hood.

If you were to draw a graphic image and move it around without first saving what's under that image (or better, just redrawing what's behind it), you'd get a bunch of "chunks" ripped out of your background. So, somehow we have to save the data directly behind the mouse and redisplay that before we redraw the mouse in its new position.

So how do you do it then? It's easy! You use page-flip animation. Draw all of the images each rendering cycle and just treat the mouse like any other image. You may still want to keep track of your old positions so you can see if the mouse has even moved, but that's simply a case of using the MOUSEX and MOUSEY functions.

Here's a tiny snippet that would show what you would do:

```
DrawImage MouseImage,MouseX (),MouseY ()
```

That's it. What that will do is update your screen each frame with the mouse. You probably want this to be the last DRAWIMAGE command called in your game loop, right before you call FLIP.

If you want to change the image used as the mouse cursor, it's a snap. Since the mouse is basically being portrayed like any other image, all you have to do is change the image that you send to DRAWIMAGE. So whatever image you use, regardless of its size, color, orientation, etc., will be drawn here. This is also cool because if you keep track of the frame as we did in the Animation section, you can have a neat animated mouse cursor.

Using the Joystick

As with the other input devices, there are a number of joystick commands that can be used.

To keep with our previous examples, let's create a function that rotates an image based on joystick movement. The JOYXDIR command tells us if the X direction of the Joystick is -1 (left), 0 (centered), or 1 (right), so it's perfect for our need. We could use the JOYYDIR command of course, and replacing the JOYXDIR command in the following code would offer the same results.

```
Function CheckJoystick()
  ; call the JoyXDir () function to see if
  ; the stick is being pushed left
  If JoyXDir () < 0 Then
```

```
    ; change the current frame (spin it to the left)
    iCurrentFrame = iCurrentFrame - 1

    ; if the current frame counter is less than 0
    ; reset it to the number of rotations - 1
    If iCurrentFrame < 0
      iCurrentFrame = iNumRotations - 1
    EndIf
  EndIf

  ; call the JoyXDir () function to see if
  ; the stick is being pushed right
  If JoyXDir () > 0 Then
    ; change the current frame (spin it to the right)
    iCurrentFrame = iCurrentFrame + 1

    ; if the current frame counter goes past our number of
    ; allowable rotations, reset it to 0
    If iCurrentFrame > iNumRotations - 1
      iCurrentFrame = 0
    EndIf
  EndIf
 End Function
```

This demonstrates that you can see in which direction the joystick is being held. But what if you want to have something move faster the farther the joystick was pushed in a particular direction? In kind, this would mean that the object would move more slowly if the joystick was barely being pushed.

You would use the JOYX and JOYY commands. These commands return a floating-point value between –1 and 1. So if you were barely pressing to the left, you may get a return value of -.0025, for example. This is important because you can use this as a value to control the speed of acceleration or a turn or braking, etc.

The following piece of code demonstrates the positional information contained in these commands. It will only work if you have a joystick attached, obviously. The good news is that I've also incorporated a command, JOYTYPE, that demonstrates how to see if a joystick is attached and what type of joystick it is.

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480
```

```
Graphics ScreenWidth, ScreenHeight
SetBuffer BackBuffer ()

; check the type of joystick we have
JoyStickType = JoyType ()

Select JoyStickType
   Case 0
      JSType$ = "No Joystick Attached"
   Case 1
      JSType$ = "Digital Joystick"
   Case 2
      JSType$ = "Analog Joystick"
End Select

; while the user doesn't hit <ESC>, which is '1'
While Not KeyHit (1)
  ; clear the screen
  Cls
  ; show the type of joystick attached
  Text 0,0,"Current X Location of the joystick: " + JSType$

  ; print out our precisioned joystick positional info
  Text 0,16,"Current X Location of the joystick: " + JoyX ()
  Text 0,32,"Current Y Location of the joystick: " + JoyY ()

  ; check to see which way the joystick is being pressed
  ; first do the left/right
  If JoyXDir () < 0
    Joystick_LR_Direction$ = "Going Left"
  EndIf
  If JoyXDir () = 0
    Joystick_LR_Direction$ = "Centered"
  EndIf
  If JoyXDir () > 0
    Joystick_LR_Direction$ = "Going Right"
  EndIf

  ; now handle up/down
  If JoyYDir () < 0
    Joystick_UD_Direction$ = "Going Up"
  EndIf
  If JoyYDir () = 0
    Joystick_UD_Direction$ = "Centered"
```

```
  EndIf
  If JoyYDir () > 0
    Joystick_UD_Direction$ = "Going Down"
  EndIf

  ; print out the direction information
  Text 0,48,"Direction Pressed (L/R Axis) " + Joystick_LR_Direction$
  Text 0,60,"Direction Pressed (U/D Axis) " + Joystick_UD_Direction$
   Flip
Wend
End
```

In order to handle Joystick button hits, we use either the JOYHIT command or the GETJOY command. You can also use the WAITJOY command to halt the program until the user hits a joystick button.

Just as the KEYHIT command, JOYHIT requires that you pass it the particular button to check for. GETJOY on the other hand, returns whatever button was pressed. Similar to our mouse button example, here is one for the joystick:

```
; setup our screen constants
 Const ScreenWidth = 640
 Const ScreenHeight = 480

Graphics ScreenWidth, ScreenHeight
SetBuffer BackBuffer ()

 LastButton$ = "None"
 OtherButton$ = "None"

 ; while the user doesn't hit <ESC>, which is '1'
 While Not KeyHit (1)
  ; clear the screen
  Cls

  ; check to see if any of the buttons have been pressed
  If JoyHit (1)
    LastButton$ = "Button 1"
  EndIf
  If JoyHit (2)
    LastButton$ = "Button 2"
  EndIf
  If JoyHit (3)
    LastButton$ = "Button 3"
```

```
  EndIf
  If JoyHit (4)
    LastButton$ = "Button 4"
  EndIf

  Text 0,0, "Last Defined Button Pressed: " + LastButton$

  ; There may be more buttons that we could get info from,
  ; so let's use GetJoy () to find if they've been pressed
  ButtonPressed = GetJoy ()

  ; we should only process them if they are beyond the 4
  ; buttons we've already defined using JoyHit ()
  If ButtonPressed > 4
    OtherButton$ = "Button " + ButtonPressed
  EndIf

  Text 0,16, "Last Undefined Button Pressed: " + OtherButton$
  Flip
Wend
 End
```

Finally, you can use the FLUSHJOY command to clear out the joystick buffer, just as you did with the keyboard and the mouse.

Chapter 16: Sounds and Music

What kind of game has no explosions, weird sounds, ambience, or a soundtrack? A boring one, if you ask me. You have to have sounds!

But there are a number of things to think about when incorporating sounds and music. First off, it would be pretty lame to have one sound cut off as soon as another one plays. Secondly, if all the sounds are so loud that they're basically bleeding together, that's no good. Also, shouldn't the user have the ability to turn stuff down or off altogether? We'll get into these issues in this chapter.

Loading Sounds

The first thing you'll need to do when working with sounds is to load them in. You do this by using the LOADSOUND command, which has the following layout:

SoundHandle ***= LoadSound (****"path/SoundFile.wav"****)***

It's important to note that with this command (and all file-loading commands) you must include the appropriate path to the files you're attempting to load in. If you neglect to do this, BB will respond with a runtime error.

At the time of this writing, the only industry standard sound types supported are WAV, MIDI, OGG, and MP3. I have found that OGG files are very clean audio files that are substantially smaller in size than WAV and it's a royalty-free format.

Most people use WAV files for quick sounds (explosions, gunfire, engine sounds, button clicks, etc.) and reserve MP3/OGG/MIDI for in-game music.

Playing and Manipulating Sounds

There are a few commands that you can use to help make your sounds a bit more realistic in games. They are: SOUNDVOLUME, SOUNDPAN, and SOUNDPITCH, and they have the following layouts:

SoundVolume *SoundHandle,Volume*
SoundPan *SoundHandle,PanValue*
SoundPitch *SoundHandle,PitchValue*

Using these creatively, you can give your player the feeling that there are explosions far in the distance. Or maybe if they've been hit, they'll know they were shot from off to their left because you've panned the sound fully to the

left speaker. Plus, since sounds farther away tend to have a deeper pitch than sounds close by, you can control the pitch of the sounds accordingly.

Sound volume can be anywhere from 0 to 1.00. The number is controlled via floating point to allow you complete control over the exact volume you want.

Sound pitch goes from 0hz to 44000hz. Drop it down to 10000hz and you'll hear a very deep and slow sound. And, depending on the hertz amount you saved your sound, if you pop it up to 44000hz you'll hear a really high-pitched, quick sound.

Here is a little demo that allows the manipulation of an explosion sound. You may replace the WAV filename with any file name that you wish, just make sure it's in the appropriate directory.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Sound Volume, Pan, and Pitch Demo
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Graphics 640,480
SetBuffer BackBuffer ()

; include our Keyboard library
Include "..\libraries\keyconstants.bb"

; Load in our .WAV file
Sound1 = LoadSound ("sounds/explosion1.wav")

; initialize the Volume to half and set
Sound1Volume# = .5
SoundVolume Sound1,Sound1Volume#

; initialize the pan to center and set
Sound1Pan# = 0
SoundPan Sound1,Sound1Pan#

; initialize the pitch to highest and set
Sound1Pitch = 44000
SoundPitch Sound1,Sound1Pitch

; while the user hasn't pressed the ESC key
While Not KeyHit (Key_Escape)

   ; clear the screen
   Cls
```

```
; if the user hits the space bar, play
; the sound
If KeyHit (Key_Space)
  PlaySound(Sound1)
EndIf

; if the user hits the up arrow, increase
; the volume
If KeyDown (Key_ArrowPad_Up)
  Sound1Volume# = Sound1Volume# + .01
  If Sound1Volume# > 1
    Sound1Volume# = 1
  EndIf
  SoundVolume Sound1,Sound1Volume#
EndIf

; if the user hits the down arrow, decrease
; the volume
If KeyDown (Key_ArrowPad_Down)
  Sound1Volume# = Sound1Volume# - .01
  If Sound1Volume# < 0
    Sound1Volume# = 0
  EndIf
  SoundVolume Sound1,Sound1Volume#
EndIf

; if the user hits the right arrow,
; move the pan a bit to the right
If KeyDown (Key_ArrowPad_Right)
  Sound1Pan# = Sound1Pan# + .01
  If Sound1Pan# > 1
    Sound1Pan# = 1
  EndIf
  SoundPan Sound1,Sound1Pan#
EndIf

; if the user hits the left arrow,
; move the pan a bit to the left
If KeyDown (Key_ArrowPad_Left)
  Sound1Pan# = Sound1Pan# - .01
  If Sound1Pan# < -1
    Sound1Pan# = -1
  EndIf
  SoundPan Sound1,Sound1Pan#
```

```
  EndIf


  ; if the user hits the right-control,
  ; raise the pitch slightly
  If KeyDown (Key_RightControl)
    Sound1Pitch = Sound1Pitch + 100
    If Sound1Pitch > 44000
      Sound1Pitch = 44000
    EndIf
    SoundPitch Sound1,Sound1Pitch
  EndIf

  ; if the user hits the left-control,
  ; lower the pitch slightly
  If KeyDown (Key_LeftControl)
    Sound1Pitch = Sound1Pitch - 100
    If Sound1Pitch < 0
      Sound1Pitch = 0
    EndIf
    SoundPitch Sound1,Sound1Pitch
  EndIf

  ; put up some text to explain usage
  Text 0,0,"Up Arrow = Increase Volume, Down Arrow = ↲
       → Decrease Volume"
  Text 0,16,"Left Arrow = Pan Left, Right Arrow = Pan Right"
  Text 0,32,"Left-Control = Decrease Pitch, ↲
       → Right-Control = Increase Pitch"
  Text 0,48,"Spacebar = Play Explosion sound"

  ; put up more text to show current sound values
  Text 0,100,"SOUND INFORMATION:"
  Text 0,116,"Sound Volume = " + Sound1Volume#
  Text 0,132,"Sound Pan = " + Sound1Pan#
  Text 0,148,"Sound Pitch = " + Sound1Pitch

 Flip
Wend
End
```

Playing Music

Music files are not entirely different than sound files. Actually, most of the commands are interchangeable. Change that WAV file to an MP3/OGG/XMS/MIDI file and you'll see what I mean.

There are an additional couple of commands that you can use specifically for music, though. These commands are PLAYMUSIC and PLAYCDTRACK, and their layouts are as follows:

MusicHandle = ***PlayMusic(****"path/filename"****)***
MusicHandle = ***PlayCDTrack(****Track, [Mode]****)***

PLAYMUSIC acts similar to PLAYSOUND, but with one big exception. PLAYMUSIC only plays the music file once. When it's done playing, you must physically reload the music file to play it again. There is a way around this using *Channels*, though, which we'll get into shortly. PLAYMUSIC currently supports WAV, MP3, OGG, XMS, and MIDI files.

PLAYCDTRACK actually snags a valid track off of the CD to play. There are a few options (or play *modes*) that you can use with this command. The following would play track #1 once, and only once:

```
MusicHandle = PlayCDTrack (1,1)
```

This next line would loop track #2:

```
MusicHandle = PlayCDTrack (2,2)
```

And, finally, this line would start at track #3 and play all the tracks until the end of the CD:

```
MusicHandle = PlayCDTrack (3,3)
```

Channels

To really get the most out of sounds and music, you should become familiar with the concept of channels. Imagine that you have ten radios surrounding you. Each one has a different song playing. In order for this to be possible, each radio must be tuned to a different station, right? So, picture each radio as being a *channel* and you'll start getting the idea. On channel 1 you have classical music, channel 2 has rock, and so on. If you want to turn down the classical music, you just turn the volume down on channel 1 (or radio 1).

Now take this concept and put it in with our sounds and music. On channel 1 we have our game music, channel 2 is carrying our big explosion sound, channel 3 is the engine sound for our ship, and so on. To decrease the volume of the music, we'd snag Channel 1 and turn it down. To pan our explosion to the right, grab that channel and pan it.

There is also an added benefit to channels in that they work on values real-time. So if we're changing the pitch of an explosion with channel commands, the currently playing explosion will be affected immediately. Our previous examples only showed changes for the next time the sound was played.

Setting up channels is a snap. Basically, the sound is loaded in using LOADSOUND. Then, when using PLAYSOUND, PLAYMUSIC or PLAYCDTRACK, we just assign a channel handle for manipulation. Here is an idea of it:

```
MusicHandle = LoadSound ("music\rocktoids.ogg")
MusicChannel = PlaySound(MusicHandle)
```

And that's all there is to it. From that point on we can use the following *channel* commands using the handle *MusicChannel* (or whatever you elect to call it).

StopChannel *ChannelHandle*
PauseChannel *ChannelHandle*
ResumeChannel *ChannelHandle*
ChannelVolume *ChannelHandle, ChannelVolume*
ChannelPan *ChannelHandle, PanValue*
ChannelPitch *ChannelHandle, PitchValue*
ReturnValue = ***ChannelPlaying(****ChannelHandle****)***

STOPCHANNEL literally *stops* the music/sound and frees the channel. If you use this command, you'll need to use PLAYSOUND to get it going again. This is a good thing to use if you plan on starting the sound from the beginning. So, if you're half way through a song and you want to start it over again, you would first stop it and then play it again. Make sure to reset the handle appropriately or you'll get an error.

PAUSECHANNEL on the other hand allows you to keep the channel alive, but stop the music at its current position. RESUMECHANNEL will begin playing from the point the music was paused.

CHANNELVOLUME, CHANNELPAN, and CHANNELPITCH should be pretty obvious at this point, but keep in mind that as long as the channel is being played these can be used to affect the sound/music real-time. If the player shoots a missile and it hits a tank, say, as the explosion is going, you could

make it so if the player begins to turn away from the sound it would begin panning in the appropriate direction. Pretty cool, no?

Finally, in this bunch is the CHANNELPLAYING command. This command returns a 1 if the channel is playing, or 0 if not. This can be handy in controlling volume levels for various sounds or just to make sure you don't start a song over an already playing song. This function will return a 1 even if the channel is paused, though, because in effect the handle is still alive.

The following code demonstrates the channel commands in action. You'll need your own OGG/MP3/WAV file plugged in (or just use one from the CD) to make this work though!

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Music Demo
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; setup our Graphics
Graphics 640,480

; set up for page flipping
SetBuffer BackBuffer ()

; include our Keyboard library
Include "..\libraries\keyconstants.bb"

; Load in our music file
MusicHandle = LoadSound ("music\rocktoids.ogg")

; set this to Loop
LoopSound MusicHandle

; Start up the sound and assign a channel
MusicChannel = PlaySound(MusicHandle)

; initialize the Volume to half and set
MusicVolume# = .5
ChannelVolume MusicChannel,MusicVolume#

; initialize the pan to center and set
MusicPan# = 0
ChannelPan MusicChannel,MusicPan#

; initialize the pitch to highest and set
MusicPitch = 44000
```

```
ChannelPitch MusicChannel,MusicPitch

; variables to toggle music on/pause/off
MusicPause = 0
MusicStop = 0
MusicStatus$ = "Playing"

ChannelPlayingValue = ChannelPlaying (MusicChannel)

; while the user hasn't pressed the ESC key
While Not KeyHit (Key_Escape)

   ; clear the screen
   Cls

   ; if the user hits the space bar, either
   ; Pause the channel, or Resume it
   If KeyHit (Key_Space)
     If MusicPause = 0
       PauseChannel MusicChannel
       MusicPause = 1
       MusicStatus$ = "Paused"
       ; set the ChannelPlaying value
       ChannelPlayingValue = ChannelPlaying (MusicChannel)
     Else
       ResumeChannel MusicChannel
       MusicPause = 0
       MusicStatus$ = "Playing"
       ; set the ChannelPlaying value
       ChannelPlayingValue = ChannelPlaying (MusicChannel)
     EndIf
   EndIf

   ; if the user hits the enter, either
   ; Stop the channel or restart the channel
   If KeyHit (Key_Enter)
     If MusicStop = 0
       StopChannel MusicChannel
       MusicStop = 1
       MusicStatus$ = "Stopped"

       ; set the ChannelPlaying value
       ChannelPlayingValue = 0
     Else
       ; reset all the values for the channel
```

```
    MusicChannel = PlaySound(MusicHandle)

    ; initialize the Volume to half and set
    MusicVolume# = .5
    ChannelVolume MusicChannel,MusicVolume#

    ; initialize the pan to center and set
    MusicPan# = 0
    ChannelPan MusicChannel,MusicPan#

    ; initialize the pitch to highest and set
    MusicPitch = 44000
    ChannelPitch MusicChannel,MusicPitch

    MusicStop = 0
    MusicStatus$ = "Playing"

    ; set the ChannelPlaying value
    ChannelPlayingValue = ↲
          → ChannelPlaying (MusicChannel)
  EndIf
EndIf

; if the user hits the up arrow, increase
; the volume
If KeyDown (Key_ArrowPad_Up)
  MusicVolume# = MusicVolume# + .01
  If MusicVolume# > 1
    MusicVolume# = 1
  EndIf
  ChannelVolume MusicChannel,MusicVolume#
EndIf

; if the user hits the down arrow, decrease
; the volume
If KeyDown (Key_ArrowPad_Down)
  MusicVolume# = MusicVolume# - .01
  If MusicVolume# < 0
    MusicVolume# = 0
  EndIf
  ChannelVolume MusicChannel,MusicVolume#
EndIf

; if the user hits the right arrow,
; move the pan a bit to the right
```

```
If KeyDown (Key_ArrowPad_Right)
  MusicPan# = MusicPan# + .01
  If MusicPan# > 1
    MusicPan# = 1
  EndIf
  ChannelPan MusicChannel,MusicPan#
EndIf

; if the user hits the left arrow,
; move the pan a bit to the left
If KeyDown (Key_ArrowPad_Left)
  MusicPan# = MusicPan# - .01
  If MusicPan# < -1
    MusicPan# = -1
  EndIf
  ChannelPan MusicChannel,MusicPan#
EndIf

; if the user hits the right-control,
; raise the pitch slightly
If KeyDown (Key_RightControl)
  MusicPitch = MusicPitch + 100
  If MusicPitch > 44000
    MusicPitch = 44000
  EndIf
  ChannelPitch MusicChannel,MusicPitch
EndIf

; if the user hits the left-control,
; lower the pitch slightly
If KeyDown (Key_LeftControl)
  MusicPitch= MusicPitch - 100
  If MusicPitch< 0
    MusicPitch = 0
  EndIf
  ChannelPitch MusicChannel,MusicPitch
EndIf

; put up some text to explain usage
Text 0,0,"Up Arrow = Increase Volume, Down Arrow = ↲
     → Decrease Volume"
Text 0,16,"Left Arrow = Pan Left, Right Arrow = Pan Right"
Text 0,32,"Left-Control = Decrease Pitch, ↲
      → Right-Control = Increase Pitch"
Text 0,48,"Spacebar = Toggle Music Pause/Resume"
```

```
   Text 0,60,"Enter = Toggle Music Stop/Restart"

   ; put up more text to show current sound values
   Text 0,100,"MUSIC INFORMATION:"
   Text 0,116,"Volume = " + MusicVolume#
   Text 0,132,"Pan = " + MusicPan#
   Text 0,148,"Pitch = " + MusicPitch
   Text 0,172,"Music is: " + MusicStatus$
   Text 0,200,"ChannelPlaying Value = " + ChannelPlayingValue

  Flip
Wend
End
```

This should get you started in the use of sounds and music. From here you should play around with tying sounds into events, such as when a person clicks the mouse button making a bullet sound. If a dot hits the wall, make a bouncing sound or something. There are a bunch of things you can do, so roll up your sleeves and get to work!

Chapter 17: Timers

A big problem in making games is keeping the frame rate consistent among many machines.

Let's say that you have a slow computer. You create your game on this computer and get it to run nicely. Next you release your game for others to play, but they come back to you saying that it runs way too fast. What you'll find out is that the game will run as fast as the processor will allow. Another issue may be that the game runs too slowly on machines that are not as powerful as your computer. The good news is that there are ways to combat this issue. The bad news is that it takes some work on your part.

Frames per Second (FPS) Tracking

The first thing you'll need is a way to track how fast your FPS is in your game. You'll then want to find a way to lock it to a certain rate regardless of the machine the game is running on.

Frames per Second means how many times Blitz draws a scene and displays it to the user every second. If you have a super fast computer that runs your game at 120FPS, you may assume that it's going to be over 30FPS on slower machines…and you may well be right. The problem is twofold here, though. Firstly, why would you want to waste over 60-70FPS when most monitors can't display frames that fast? You're actually only displaying half of the frames to the player, so they are missing 50% of the action. Secondly, you're going to have a different play experience on each computer that people play on. That's not good. The experience should be as consistent as possible.

That said, let's discuss how to show the current FPS for your game.

1) Setup a variable that keeps track of the starting frame time
2) While the current time is not greater than the starting frame time plus 1000 (which translates as "while the current time is not 1 second greater than the starting frame time"), increment a counter by 1.
3) When 1 second has passed:
   a) Set the FPS tracker to the counter
   b) Reset the counter to 0
   c) Reset the starting frame timer to the current time
4) Go back to step 2

Here is a very small program that demonstrates this in action:

```
; set up our constants for screen dimensions
```

```
Const ScreenWidth = 800
Const ScreenHeight = 600

Graphics ScreenWidth,ScreenHeight
SetBuffer BackBuffer ()

; create our FPS start timer and set it to
; the current time
FPS_Timer = MilliSecs ()

; setup FPS tracking variables
FPS = 0
FPS_Counter = 0

; while the user has not hit Escape
While Not KeyDown (1)

 ; clear the screen
 Cls

 ; draw a bunch of random lines in random colors
 For i = 0 To 500
    Color(Rnd (0,255),Rnd(0,255),Rnd(0,255))
    Line(Rnd (0,ScreenWidth),Rnd(32,ScreenHeight), ↲
       → Rnd (0,ScreenWidth),Rnd(32,ScreenHeight))
 Next

 ; If the current time is 1000 millisecs (1 second)
 ; passed the starting timer
 If MilliSecs () > FPS_Timer + 1000
   ; set the FPS variable to the counter
   FPS = FPS_Counter
   ; reset the counter to 0
   FPS_Counter = 0
   ; reset the starting timer to the current time
   FPS_Timer = MilliSecs ()
 Else
   ; otherwise, add 1 to the counter variable
   FPS_Counter = FPS_Counter + 1
 EndIf

 ; reset our color to white and display the FPS
 Color(255,255,255)
 Text 0,0,"FPS = " + FPS
```

```
 Flip
Wend
End
```

That little program will draw a bunch of lines on the screen using random colors and locations. Depending on the speed of your computer you will see either really high FPS or really low. My computer ran that test at about 50FPS. Change the number of lines in the FOR…NEXT loop and see how the FPS changes.

The biggest problem with this example is that it uses a FOR…NEXT loop that basically forces the computer to run through all of the processes regardless of speed. This is important to note because you'll want to be careful with using these types of things. While you'll certainly need loops to process all of your enemies, tiles, etc., be careful to control how often they're used.

The "WaitTimer" Timer

Blitz includes a method whereby you can create a timer of a certain amount and then ask Blitz to wait for that timer to ping before continuing on. The two functions you need to use this method are CREATETIMER and WAITTIMER.

CREATETIMER is used to get Blitz to grab a value that will lock your PC to 60FPS (or whatever value you choose). WAITTIMER is called after each FLIP to halt execution if that timer value has not been met.

The following snippet demonstrates this in action. Note that the time passed is *not* the frames per second. This just shows you how consistent the timing is per frame. If you alter the CREATETIMER value you'll see that the time passed values change accordingly.

```
; set up our constants for screen dimensions
Const ScreenWidth = 800
Const ScreenHeight = 600

Graphics ScreenWidth,ScreenHeight
SetBuffer BackBuffer ()

; create our FPS_Timer using CreateTimer,
; and set it to 60 FPS
FPS_Timer=CreateTimer (60)

; create a start time var
StartTime = MilliSecs ()
```

```
; while the user has not hit Escape
While Not KeyDown (1)

 ; clear the screen
 Cls

 ; see how much time has passed since the last flip
 TimePassed = MilliSecs () - StartTime

 ; reset the start time
 StartTime = MilliSecs ()

 ; show the TimePassed value
 Text 0,0,"Time Passed = " + TimePassed

 Flip

 ; call the WaitTimer function to keep
 ; the app locked at 60FPS
 WaitTimer (FPS_Timer)
Wend
End
```

This function is really for limiting the speed of a game on faster machines. On slower machines, this isn't always affective because the processing of an individual frame may slow to the point where the WAITTIMER command just makes things even slower. So, if you specify a minimum system requirement that you know the WAITTIMER works great on, then your players won't have any arguments.

The Rolling Timer

Another way to handle keeping the game moving decently on all machines is to move all the objects based on the individual speeds of the machines. I know that sounds obvious, but here's the point: on a fast machine you'll want all the objects to move, say, only every 2 frames. Now, to the human eye, this will be undetectable. On a slow machine you'll want the objects to move multiple times each frame.

So what we'll do is find a decent speed that we like, determine how much time has elapsed each frame, then make a calculation to move our objects multiple times before redisplaying (which *can* have a jumpy effect if it's a really slow machine).

Here is a piece of code that shows this:

```
; Initialize our main timer
Main_Timer = Millisecs ()

While Not KeyHit (1)
    ;what's the difference in time since our last check?
   ElapsedTime = Millisecs () - Main_Timer

   ; slowing down! - clamp update to 40 FPS
   ; (1000/40=25 millisecs)
   If ElapsedTime > 25
      ClampValue = Elapsedtime / 25
      For i=1 To ClampValue
           ; update Objects here
      Next

      ; add appropriate offset to Main_Timer controller
     Main_Timer = Main_Timer + ClampValue * 25
    Else
     ; Update Objects here as normal and
     ; reset Main_Timer to the current time
    Main_Timer=Millisecs ()
   EndIf
Wend
```

That code does nothing on its own, it's meant to be incorporated into a larger project, but let's go through it to see what's happening.

ElapsedTime = Millisecs () - Main_Timer

That piece grabs the current time and subtracts the initial time from it. That way we'll know how many milliseconds have passed since the initialization.

If ElapsedTime > 25

Next we want to see if the difference is greater than 25 milliseconds. We do this because if you take 1000 milliseconds (1 second) and divide it by 40 (what we want our FPS to be), you'll get 25. So, the idea is that we want updates done and displayed every 25 milliseconds. Since a millisecond is a specific unit of measure, all computers will share the same value for it. One full second on a 386 is the equivalent to 1 full second on a 2Ghz machine.

```
ClampValue = Elapsedtime / 25
```

The next thing we need to do is divide how much time has elapsed by the value of 25. This is because slower machines will likely be way past the 25-millisecond mark on your renderings. So, let's say that a slow computer is hitting 75 milliseconds per frame. Since 75 divided by 25 is 3, we'll want to make 3 updates before our next frame.

```
For i=1 To ClampValue
    ; update Objects here
 Next
```

The above code does exactly this. Since each time you update an object it moves X, Y (and maybe Z) values, a slower machine will need to make more than one of these updates per frame.

```
Main_Timer = Main_Timer + ClampValue * 25
```

Now we need to multiply our *ClampValue* by that 25 and add it to our current *Main_Timer* value so we bring up the timer values accordingly.

```
Else
     ; Update Objects here as normal and
     ; reset Main_Timer to the current time
    Main_Timer=Millisecs ()
EndIf
```

If the elapsed time doesn't go past 25 then your game is running faster than 40FPS and we just want to update the objects as we always do and then reset the timer.

Now you may be thinking that this will look really bad. Each frame instead of 1 or 2 pixel moves per object, it could be 3-6 pixels. If the machine is *really* slow, it will look choppy. But it isn't that bad on machines that are off by 25-50ms, and it's better that you control the movement of the objects than allowing it to be controlled by the frames themselves. If you're running at 20FPS and not controlling things, for example, you're going to see a ship take twice as long to cross from point A to point B than on machine running at 40FPS. With the rolling timer method, they will pass between the points at the same speed, albeit a little choppier.

The biggest problem with this method is that it doesn't cap the FPS. That means that you could literally be displaying far more images than your monitor can handle. That brings us to another, more reliable method:

Locking in at Real Time

I have seen a number of people talking about the Real Time method as of late, and I have tried it myself with great success.

The idea is that you want to define how many units per second an object is allowed to move. The number of units is defined by you, as well as what exactly a unit is sized at. For example, you may decide that a missile can only move at 20 pixels per second. That being the case, 20 pixels = 1 unit for missiles. Ship A may move at 15 pixels per second while ship B moves at 17 pixels per second. Therefore, Ship A's units are sized at 15 pixels and Ship B's at 17 pixels.

Now, since we are likely updating by milliseconds, not full seconds, we'll want the accuracy given to us by floating point numbers. If we went with integers there would be some drastic jumps on the screen by your objects.

Below is a little demo that moves a box across the screen and does so by using the Real Time method. Study the code carefully to see how it works.

```
; set up our constants for screen dimensions
Const ScreenWidth = 800
Const ScreenHeight = 600

Graphics ScreenWidth,ScreenHeight
SetBuffer BackBuffer ()

; create a start time var
StartTime# = MilliSecs ()

; Declare our unit measurements
XUnit# = .250
YUnit# = .125

; Declard our starting points
X# = 0
Y# = 0

; while the user has not hit Escape
While Not KeyHit (1)

 ; get the current millisecs
```

```
EndTime# = MilliSecs ()

; have we moved at least 1 millisec further ahead?
If EndTime# > StartTime#
  ; clear the screen
  Cls

  ; assign the TimePassed
  TimePassed# = EndTime# - StartTime#

  ; reset the StartTime to the EndTime
 StartTime# = EndTime#

  ; Add X's current value to the number of units its to
  ; move multiplied by how much time has passed
  X# = X# + (XUnit# * TimePassed#)

  ; Add Y's current value to the number of units its to
  ; move multiplied by how much time has passed
  Y# = Y# + (YUnit# * TimePassed#)

  ; if X is greater than the screen width move X to 0
  If X# > ScreenWidth
    X# = 0
  EndIf

  ; if Y is greater than the screen height move Y to 0
  If Y# > ScreenHeight
    Y# = 0
  EndIf

  ; move a rectangle across screen using specified units
  Rect (X#,Y#,20,20)

  ; calculate the FPS by dividing 1 (second) by the
  ; number of milliseconds passed, and then multiply
  ; that value by 1000ms (1 second)
  FPS# = (1.0 / TimePassed#) * 1000

  ; display it without decimal on the screen
  Text 0,0,"FPS = " + Int(FPS)

  Flip
 EndIf
Wend
```

```
End
```

Now, if you use that method and setup field in a TYPE called *Units* (for example), you'll be able to control how many units each object on the screen is moved. That way you'll have objects moving at all different speeds!

Here's an example that does just that. It will display 100 boxes of varying sizes and move them around at varying speeds. Each distance moved on both the X and Y values will be determined by a random value generated for X units per second and Y units per second. Study this closely to see how it works.

```
; set up our constants for screen dimensions
Const ScreenWidth = 800
Const ScreenHeight = 600

Graphics ScreenWidth,ScreenHeight
SetBuffer BackBuffer ()

; create a start time var
StartTime# = MilliSecs ()

; create a type to hold our boxes
Type Boxes
   Field X#
   Field Y#
   Field Width
   Field Height
   Field XUnits#
   Field YUnits#
End Type

; create a bunch of boxes
CreateBoxes(100)

; while the user has not hit Escape
While Not KeyHit (1)

 ; get the current millisecs
 EndTime# = MilliSecs ()

 ; have we moved at least 1 millisec further ahead?
 If EndTime# > StartTime#
```

```
   ; clear the screen
   Cls

   ; assign the TimePassed
   TimePassed# = EndTime# - StartTime#

   ; reset the StartTime to the EndTime
   StartTime# = EndTime#

   ; call our UpdateBoxes functions with the Time that's
   ; passed since the last call
   UpdateBoxes(TimePassed#)

   ; now calculate the FPS (Frames Per Second)
   FPS# = (1.0 / TimePassed#) * 1000

   ; display it without decimal on the screen
   Text 0,0,"FPS = " + Int(FPS)

   Flip
  EndIf
Wend
End

; ***********************************************
; This function simply creates a bunch of boxes with
; random starting points, widths, and units/sec values.
;
; Arguments:
;      Quantity = how many boxes to create
;***********************************************
Function CreateBoxes(Quantity)
   ; run through create random boxes with random speeds
   For i = 0 To Quantity
        Box.Boxes = New Boxes
        Box\X# = Rnd#(0,ScreenWidth)
        Box\Y# = Rnd#(0,ScreenHeight)
        Box\Width = Rnd (5,30)
        Box\Height = Rnd (5,30)
        Box\XUnits# = Rnd#(.0125,.500)
        Box\YUnits# = Rnd#(.0125,.500)
   Next
End Function

;***********************************************
```

```
; This function updates the X,Y positions of each box based
; on the number of  XUnits and YUnits and the speed value
; (based off the elapsed time) sent.
;
; Arguments:
;      Speed# = Time in milliseconds since last update
;****************************************** Function
UpdateBoxes(Speed#)
   ; run through all the boxes
   For Box.Boxes = Each Boxes
       ; update X position based on the speed and units/sec
       Box\X# = Box\X# + (Box\XUnits# * Speed#)
       If Box\X# > ScreenWidth
          Box\X# = 0
       EndIf

       ; update Y position based on the speed and units/sec
       Box\Y# = Box\Y# + (Box\YUnits# * Speed#)
       If Box\Y# > ScreenHeight
          Box\Y# = 0
       EndIf

       ; draw out the box
       Rect (Box\X#,Box\Y#,Box\Width,Box\Height)
   Next
End Function
```

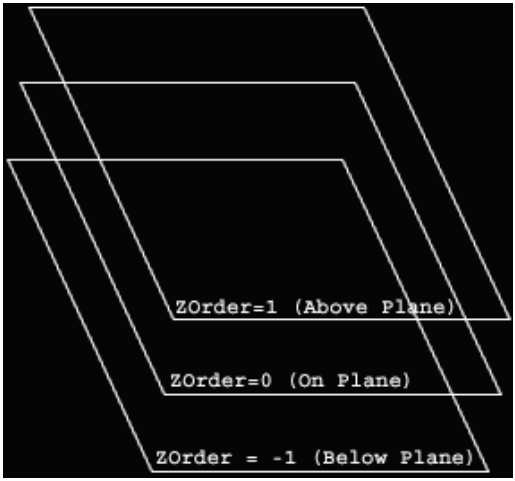
PART 3: ADVANCED TOPICS

Chapter 18: Z-Ordering

What is Z-Ordering?

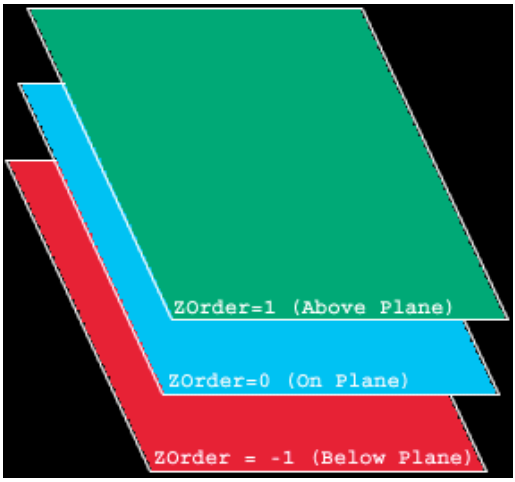
The term "Z-Ordering" can mean something different depending on context. In the realm of 3D graphics, I've seen it described as "...to derive closed-form relations for the difference between the node indices, which can be used to browse the tree in constant time." That's NOT the way we'll be using it in this book.

Since we'll be applying *Z-Ordering* to a 2D graphics plane, I'll define the term as "the drawing of graphical elements in order of their height in respect to graphics plane." Look at the image below and you'll see that we have three planes of drawing. Each overlaps the other.



(Figure 18.1)

These are the planes of drawing. Anything that you draw on plane -1 will be "under" anything drawn on plane 0 or plane 1, as anything on plane 0 will be "under" anything drawn on plane 1. To further demonstrate, I've filled in that image here:



(Figure 18.2)

Note that the only plane completely visible is plane 1.

Why Use Z-Ordering?

There are lots of reasons you'll want to use this concept. If you take the standard space game, for example, you would probably want your ship to look as if it's hovering over a planet. Or maybe you'll have a neat little code that lets you hide behind planets (in multiplayer maybe?). How are you going to visually do that? The only way is to control the order in which you draw the planet and the ship.

Let's say that you're doing a football game. You'll need to have a way to show the ball in front of the players, or behind...heck, you'll need a way to show the players to the front or back of other players too or it'll look REALLY weird. *Z-Ordering* handles that.

Another example would be a side-scrolling game. Imagine you have this little character running along a grassy tile set, and behind him is a line of trees. Up ahead is a waterfall...but wait...it's flowing IN FRONT of the grassy tile set! Your character gets up to that point and nearly disappears through the waterfall... he's *behind* the waterfall! In order to do this...we'd draw in this order:

- Draw the trees
- Draw the grass
- Draw the player
- Draw the waterfall

So the grass is on top of the trees, and the player is on top of the grass, and the waterfall is on top of the player. Since we can use transparency in our images, it gives us the illusion of depth!

How to Implement Z-Ordering

One way is to define certain images ahead of time to have a certain plane. This works great in the case of the side-scrolling game. You define either an array or TYPE (depending on your preference) that contains all of the Plane 0 elements. Then others that contains all the Plane -2, Plane -1, Plane 1, Plane 2, etc. (as many you want, really). Next you start at the lowest Plane and draw it, then move up to the next, draw it, and then continue that cycle until they're all drawn.

Sometimes you'll want to be more dynamic (as in the case of the football game). In this case, you'll need to have the Z-Order for each Image dynamic. This way, depending on comparisons in your game, you'll be able to change the value of Z-Order in your "Update" phase and watch the change take effect in your "Render" phase.

Here is a piece of code that grabs three images and moves them over each other based on Z-Order. With only 3 images, this is *really* unnecessary, but it gets the point across of *how* to use this technique.

```
; setup our graphics mode
 Graphics 640,480

 ; set the Backbuffer as the drawing surface
 ; for page flipping
 SetBuffer BackBuffer ()

 ; load up our dummy images
 Grass_Image = LoadImage ("grass.png")
 Ship_Image = LoadImage ("ship.png")
 Swashbuckler_Image = LoadImage ("swashbuckler.png")
 MaskImage Swashbuckler_Image,255,0,255

 ; create a type that has which image to use, where
 ; to draw it, a direction to show the ZOrdering in
 ; action, and the ZOrder of the image
 Type Images
    Field ImageToUse
    Field X,Y
    Field iDir
    Field ZOrder
 End Type
```

```
; create an instance of the Images Object, starting it
; moving from right to left, and starting it as being
; "behind" the baseline plane.  Make this one the "wall"
; (or big blue square)
NewImage.Images = New Images
NewImage\ImageToUse = 0
NewImage\X = 125
NewImage\Y = 100
NewImage\iDir = -1
NewImage\ZOrder = -1

; create an instance of the Images Object, no movement,
; And starting it as being ON the baseline plane.
; Make this one the "Knight" image
NewImage.Images = New Images
NewImage\ImageToUse = 1
NewImage\X = 100
NewImage\Y = 100
NewImage\iDir = 0
NewImage\ZOrder = 0

; create an instance of the Images Object, moving from
; left to right, and starting it as being "In Front of"
; the baseline plane. Make this one the the little rock
; image
NewImage.Images = New Images
NewImage\ImageToUse = 2
NewImage\X = 75
NewImage\Y = 100
NewImage\iDir = 1
NewImage\ZOrder = 1

; quit only if the user hits ESC
While Not KeyHit (1)

 ; clear the screen
 Cls

 ; Use this FOR...NEXT loop to cover each ZOrder plane of
 ; drawing.  So, if we're at -1, that will be the bottom.
 ; 0 will be the middle (or baseline), and 1 will be the top.
 For Planes = -1 To 1

    ; run through each of our images
```

```
For NewImage.Images = Each Images

   ; see if the current image is on the Plane we're on
   If NewImage\ZOrder = Planes

      ; if so, see what image we need to draw
      Select NewImage\ImageToUse

         ; it's the wall image
         Case 0
            DrawImage (Grass_Image, NewImage\X, ↲
                  → NewImage\Y)

            ; this is just for looks, but it moves the
            ; wall back and forth
            Select NewImage\iDir
               Case 1
                  NewImage\X = NewImage\X + 1
                  If NewImage\X > 125
                     NewImage\iDir = -1
                  EndIf
               Case -1
                  NewImage\X = NewImage\X - 1
                  If NewImage\X < 75
                     NewImage\iDir = 1
                  EndIf
            End Select

         ; it's the Swashbuckler image
         Case 1
            DrawImage (Swashbuckler_Image, ↲
                  → NewImage\X,NewImage\Y)

         ; it's the rock image
         Case 2
            DrawImage (Ship_Image,NewImage\X, ↲
                  → NewImage\Y)

            ; this is just for looks, but it moves the
            ; rock back and forth
            Select NewImage\iDir
               Case 1
                  NewImage\X = NewImage\X + 1
                  If NewImage\X > 125
                     NewImage\iDir = -1
```

```
                    EndIf
                Case -1
                    NewImage\X = NewImage\X - 1
                    If NewImage\X < 75
                       NewImage\iDir = 1
                    EndIf
            End Select
        End Select
      EndIf
    Next
  Next

  ; a little text telling the user how to change the ordering
  Text 0,300,"Press the spacebar to change the ZOrder"

  ; if the user hits the spacebar
  If KeyHit (57)
    ; run through the image list again
    For NewImage.Images = Each Images

      ; determine the ZOrder value, change it accordingly
      ; -1 will change to 1, 0 to -1, and 1 to 0...again, this
      ; is just so the user can see it in action
      Select NewImage\ZOrder
          Case -1
              NewImage\ZOrder = 1
          Case 0
              NewImage\ZOrder = -1
          Case 1
              NewImage\ZOrder = 0
      End Select
    Next
  EndIf

  ; flip the pages so the user will see the animation's effect
  Flip
Wend
End
```

For fun you could hook up a function that puts a bunch of these images up in Random X, Y locations, with Random Z-Orders. Should be a pretty simple thing to do and it could help you get your hands dirty in this coding practice.

Chapter 19: Loading Map Files

The method described here is just one of many, and it is a simple method. But it should be good enough to get you started. After seeing how this works, I would recommend that you expand upon this and make it much more robust.

Loading Tiles

Before doing anything with the map, we need to have something to display. Generally folks put a bunch of fixed-sized tiles (though they can be varied in size if your code permits) in a single image file. Sometimes all of the tiles run together, such as shown here:

(Figure 19.1)

Other times, the artist puts a block around each image to keep them visually separated:

(Figure 19.2)

This is an important distinction because you don't want to end up loading the blocks with the images, you just want the actual images. Because of this, you'll not only need to know how tall and wide each image is, but also how much space is between each of your images.

For example, let's say that you have images that are 32x32. That is, they are 32 pixels wide by 32 pixels high. And let's say that you've put a 1x1 box around each image. When you go to use the GRABIMAGE command in Blitz, you don’t want to grab from 0,0 (the top left of the image), rather you should grab from the inside-top-left edge of the box at 1,1. See below for an example:

(Figure 19.3)

In an effort to make this entire process easier on the caller, I've set up a group of functions for a map loading/displaying library. I named it "maplib.bb." You can call it whatever you want.

At the very top of the map library, I have set a number of global variables and arrays. Here they are:

```
; Type For the TileList
Type Tiles
    Field Image
    Field Width
    Field Height
End Type

; Global to track the number of tiles
Global TotalTiles = 1

; Dimension our tile array
Dim Tile.Tiles(TotalTiles)

; Type for the MapData
Type MapData
    Field TileNumber
End Type

; Globals to track the map dimensions
Global MapWidth = 1
Global MapHeight = 1

; Dimension our Map array
Dim Map.MapData(MapWidth,MapHeight)
```

We will have to load the actual image tiles each time a new map is loaded.

The following source is fully commented so study it carefully!

```
;***********************************************
;    Function: Map_LoadTiles(...)
; Description: This function loads the actual tiles to be used
; with the map.  It takes into account any boxes that may
;have  been placed around each image by allowing the
;caller to specify the box widths and heights.
;
;Arguments:
;   Tile_Full_Path$ = Tile Image file, including the full path
;   TileSize = Height and Width of the tiles being loaded
;   TileSpacer = boxes around the tiles, set this to the
;              number of pixels that the boxes are wide/high.
;***********************************************
Function Map_LoadTiles(Tile_Full_Path$,TileSize, TileSpacer)

 For TileCounter = 0 To TotalTiles - 1
      If Tile.Tiles(TileCounter) <> Null
          FreeImage Tile(TileCounter)\Image
          Delete Tile(TileCounter)
      EndIf
 Next

 ; first off let's load the full image containing all the tiles
 ; into a temporary space
 Temp_Image=LoadImage (Tile_Full_Path$)

 ImageSizeX = ImageWidth(Temp_Image)
 ImageSizeY = ImageHeight(Temp_Image)

 TileColumns = ImageSizeX / TileSize
 TileRows = ImageSizeY / TileSize

 Dim Tile.Tiles(TileRows * TileColumns )

 If Temp_Image = 0
   Return(-1)
 EndIf

 ; get the current buffer so we can restore to it
 CurrentBuffer = GraphicsBuffer()

 ; next make that image the current buffer
 SetBuffer ImageBuffer(Temp_Image)

 ; The X and Y values will be whatever the XSpacer
```

```
; and YSpacer values are. If 0, then it's assumed there is
; no space between the tiles.
X = TileSpacer
Y = TileSpacer

; keep track of the number of images
ImageNumber = 0

; run through the total number of rows
For Rows=0 To TileRows  -1
   ; and run through all the columns per row
   For Columns=0 To TileColumns  -1
        ; create a new TileList element and assign it the
        ; current ImageNumber.  Then populate the TileList
        ; element with TileWidth, TileHeight, and the actual
        ; Tile_Image
        Tile.Tiles(ImageNumber) = New Tiles
        Tile(ImageNumber)\Width = TileSize
        Tile(ImageNumber)\Height = TileSize

        ;we first use CreateImage (...) to   make sure
        ; BB allocates enough space in the TileList field
        ;Tile_Image
        Tile(ImageNumber)\Image= CreateImage (TileSize,TileSize)

        ; then we grab the image based off the size setup in
        ; CreateImage (...) from our X,Y location
       GrabImage (Tile(ImageNumber)\Image,X,Y)

        ; now we add X to the TileWidth and the XSpacer to
        ; get the new X position. If our current X = 2, and
        ; the TileWidth = 32 and the XSpacer = 2, we'd
        ; have X = 2 + 32 + 2, or 36.  This means that the
        ; next time we call GrabImage (...) it will start
        ; grabbing from the X position 36 (or the 36th pixel
        ; from the left).
        X=X + TileSize + TileSpacer

        ; increment our tile counter (array positioning)
        ImageNumber = ImageNumber + 1
   Next
   ; finished that row, reset X back to the spacer position
   X = TileSpacer

   ; add Y to TileHeight and YSpacer to get new Y position.
```

```
    ; if our current Y = 2, and TileHeight = 32 and YSpacer =
    ; 2, we have Y = 2 + 32 + 2, or 36.  This means that the
    ; next time we call  GrabImage (...) it will start grabbing
    ; from the Y position 36 (the 36th pixel from the top).
    Y = Y + TileSize + TileSpacer
  Next

  ; restore the buffer
  SetBuffer CurrentBuffer

  ; free the image from memory
  FreeImage Temp_Image

  ; reset our global tile tracker
  TotalTiles = ImageNumber

  Return(0)
End Function
```

There is a lot to that function, but if you go over it a few times it should become clear how it works.

Text-Based Map File Format

There are many ways to layout a map file. Some use numbers separated by spaces or commas, others use various methods of encryption, some just go straight across with the numbers and parse appropriately. Additionally, some map generators and files take into consideration Z-Ordering.

For ease of understanding, I've decided to go with numbers separated by commas.

The first line of my map file will have two numbers: The width, or columns on the map, and the height, or rows on the map. Immediately following that will be the appropriate number of rows of data mixed with the appropriate number of columns. Here is an example:

```
3,2
10,1,5
4,15,6
```

This map says that there are 3 columns and 2 rows. The first row contains 3 images and they are: Image 10, Image 1, and Image 5. The second row's images are: Image 4, Image 15, and Image 6. If you recall, when we loaded in the Tiles each tile was assigned a place in the array. So, when we go to call

DRAWIMAGE we can pass the *Tile(ImageNumber)* and it will display the appropriate tile!

Loading Map Dimensions

In order to load this data in, we must first determine the number of elements we'll need to store within our *MapData* array. In order to make this easier on the caller I have created two functions to handle reading in the map data. The first is called *Map_ReadDimensions* and its sole purpose is to open a map data file, read in the first line, and parse that line so it knows how many columns and rows are in the map.

Take a look at the function and study the comments closely:

```
;*****************************************
;    Function: Map_ReadTextDimensions(...)
;  Description: This function loads in the actual
;  dimensions of the map file and stores the values in
;  MapWidth and MapHeight. This function is for use with
;  TEXT files only.
; Arguments:
;   Map_Full_Path$ = Map file, including the full path
;*****************************************
Function Map_ReadTextMapDimensions(Map_Full_Path$)

 ;  open the file using a Pointer Variable
 File=ReadFile (Map_Full_Path$);

 ;  read the first line of the file...it *should* contain the
 ; width/height data.  If not, the layout is goofed up
 MapDimensions$ = ReadLine$(File);

 ; set up some variables to parse the line.  We could just
 ; have x,y on two lines makes some trouble from
 ; a coding perspective, but this method makes the map
 ; creation more intuitive for the user...and that's our job ;)
 EndOfString = 0
 Offset = 0

 ; while we haven't reached the end of the string
 While EndOfString = 0
        ; put the current position into Temp$
        ; just 1 character cause we use the Mid$(...)
        ; command to yank out that value
       Temp$ = Mid$(MapDimensions$,Offset+1,1)
```

```
        ; if not a comma we're still in the string
        If Temp$ <> "," And (Offset <= ↲
                              → Len(MapDimensions$))
           ; must be a number, save it in HoldString$
           HoldString$ = HoldString$ + Temp$
        ;it's either gone too far or it's a comma
        Else
           ; gone too far, must be the MapHeight is done
           If Offset > Len(MapDimensions$)
              ; convert HoldString$ to Int and assign
              MapHeight = Int(HoldString$)
              ; and make sure the While loop breaks
              EndOfString = 1
           ; must be a comma, MapWidth value is loaded
           Else
              ; convert HoldString$ to int and assign
              MapWidth = Int(HoldString$)
              ; reset HoldString$ to blank
              HoldString$ = ""
           EndIf
        EndIf
        ; increase string position offset by 1
        Offset = Offset + 1
  Wend
  ; we're done, so close the file!
  CloseFile(File)

 ; redimension the map with the new data
 Dim Map.MapData(MapWidth,MapHeight)

 ; call the function that resets the array values
 Map_Init(MapWidth,MapHeight)

  Return(0)
End Function
```

Now you may consider this overkill for just determining the columns and rows...and, frankly, you may be right. But my goal is to make it brain-dead simple for the person calling these functions to use the map and the function, so I don't mind overkill in my code.

Loading the Map Data

The second function is called *Map_ReadData* and its job is to take the information it knows for the number of columns and rows (which it gets from

Map_ReadDimensions) and load in all of the image numbers into the *MapData* array. Again, study this carefully:

```
;*********************************************
;    Function: Map_LoadTextMap(...)
; Description: This function loads in the actual map info.
; This function is for use with TEXT files only.
;
; Arguments:
;   Map_Full_Path$ = Map file, including the full path
;*********************************************
Function Map_LoadTextMap(Map_Full_Path$)

 ; open the file using a Pointer Variable
 File=ReadFile (Map_Full_Path$);

 ; read the Map dimensions line, but don't do anything with
 ; it...this is just to move to the map data line.  Use the
 ; Map_ReadTextDimensions(...) function for getting the
 ; actual MapWidth/MapHeight, so you can DIM the

 ; MapData array appropriately
 MapDimensions$ = ReadLine$(File);

 ; vars for array placements.  The X will be for columns, and
 ; the Y will be for rows.  EndOfFile keeps track of how
 ; far into the file we've gone.
 X=0
 Y=0
 EndOfFile = 0

 ; do this until we reach the end of the file
 While EndOfFile = 0

     ; read a line of dataand put it in the MapLine$ string
     MapLine$ = ReadLine$(File);

     ; make sure the length is more than 1 character
     If Len(MapLine$) > 1

       ; Init EndOfString var to 0...this will help us keep
       ; track of the current MapLine$ string position
       EndOfString = 0
       ; var to track current position in MapLine$
       Offset = 0
```

```
  ; until we reach the end of the MapLine$ string
  While EndOfString = 0
        ; put current position into Temp$
       Temp$ = Mid$(MapLine$,Offset+1,1)
       ; if not a comma we're still in the string
       If Temp$ <> "," And (Offset <= Len(MapLine$))
          ; must be a number, save it in HoldString$
          HoldString$ = HoldString$ + Temp$
       ; otherwise, it's too far or it's a comma
       Else
          ; if too far, this line is done
          If Offset > Len(MapLine$)
             ; make a new element in Map(x,y) array
             Map.MapData(X,Y) = New MapData
             ; Assign number in HoldString$
             Map(X,Y)\TileNumber = Int(HoldString$)
             ; reset the HoldString$ to a blank
             HoldString$ = ""
             ; exit the loop for *this* line
             EndOfString = 1
          ; must be a comma, column's value is loaded
          Else
             ; make a new element in Map(x,y) array
             Map.MapData(X,Y) = New MapData
             ; Assign number in HoldString$
             Map(X,Y)\TileNumber = Int(HoldString$)
             ; reset the HoldString$ to a blank
             HoldString$ = ""
            ; Increase X (or Column) location for Array
            X=X+1
          EndIf
       EndIf
       ; add one to our string position offset
       Offset = Offset + 1
  Wend
   ; set X back to 0 (so we're back to column 0)
   X=0
   ; add 1 to Y (so we move down 1 row in the array)
   Y=Y+1
; if we've gone past the length of the file
Else
   ; then tell the loop to stop cause we're done!
   EndOfFile = 1
EndIf
```

```
  Wend
  ; make sure to close the file!
  CloseFile(File)

  Return(0)
End Function
```

Binary-Based Map Files

Binary map files are a little different than text-based because you don't have to deal with delimiters (i.e. commas). But you'll need to have some way to create them other than just a standard text editor. Most people create a Map Creator to do this type of thing.

A Map Creator is just a visual editor that allows you to place tiles and such in a "mapping" fashion and then store that map file. I've provided a simple map creator on the disk under "MapMaker" that will load/store binary map files and such. It's not overly fancy, but it's good enough to start you working on your own maps. And it comes with the full source code so you can pick it apart and do what you want with it.

Loading Binary Maps

We don't need to have two separate functions for reading dimensions and such because we're only reading one integer at a time from the map. This means that we just have to know what our map layout is. I'm using the following format:

```
MapWidth
MapHeight
Tile1
Tile2
Tile3
Etc...
```

That's it. So all we need to do is read in the first two integers, assign them to the width and height for the map, and then just run through and fill in the *Map* array. Here's the code:

```
;*****************************************
;    Function: Map_LoadBinaryMap(...)
; Description: This function loads a binary map
;
; Arguments:
;      Map_Full_Path$: Name to load
```

```
;*****************************************
Function Map_LoadBinaryMap(Map_Full_Path$)

 ;  open the file using a Pointer Variable
 FilePtr=ReadFile (Map_Full_Path$);

 If FilePtr = -1
    Return -1
 EndIf

 ; for all of the rows (minus 1)
 ; delete the curren array info
 For Rows = 0 To MapHeight - 1
     ; and for all the columns (minus 1)
     For Columns = 0 To MapWidth - 1
          Delete Map.MapData(X,Y)
     Next
 Next

 ; read in the map dimensions
 MapWidth = ReadInt(FilePtr);
 MapHeight = ReadInt(FilePtr);

 ;redimension the map accordingly
 Dim Map.MapData(MapWidth,MapHeight)

 ; vars for array placements.  X will be for columns, and
 ; Y will be for rows.  EndOfFile keeps track of how
 ; far into the file we've gone.
 X=0
 Y=0
 EndOfFile = 0

 ; for all of the rows (minus 1)
 For Rows = 0 To MapHeight - 1
      ; and for all the columns (minus 1)
      For Columns = 0 To MapWidth - 1
          Map.MapData(X,Y) = New MapData
          ; read in the encrypted value
          Map(X,Y)\TileNumber = ReadInt(FilePtr)
          ; add 1 to X (move 1 column right in  the array)
          X=X+1
      Next
      ; set X back to 0 (so we're back to column 0)
      X=0
```

```
      ; add 1 to Y (so we move down 1 row in the array)
      Y=Y+1
  Next

  ; make sure to close the file!
  CloseFile(FilePtr)

  Return(0)
End Function
```

Now our array has the appropriate tile numbers in there and they're ready to be displayed.

Saving Binary Maps

To save a binary map is even easier. Simply write out the width and height and then run through the array, writing out each tile number as you go. Here's the code for saving binary maps.

```
;*********************************************
;    Function: Map_SaveBinaryMap(...)
;  Description: This function saves a map in binary format
;
; Arguments:
;       SaveFileName$: Name to save it as
;*********************************************
Function Map_SaveBinaryMap(SaveFileName$)

   ; open for writing.  This *will* overwrite the existing file.
   FilePtr = WriteFile (SaveFileName$)

   ; write the MapWidth and MapHeight
   WriteInt(FilePtr,MapWidth)
   WriteInt(FilePtr,MapHeight)



   ; for all of the rows (minus 1)
   For Rows = 0 To MapHeight - 1
        ; and for all the columns (minus 1)
        For Columns = 0 To MapWidth - 1
            ; get the tilenumber
            TileNumber = Map(Columns,Rows)\TileNumber
            ; write it to the file
            WriteInt(FilePtr,TileNumber)
```

```
      Next
   Next
   ; close the file!
   CloseFile(FilePtr)

   Return 0
End Function
```

Showing a Loaded Map

After calling this function you should have all the data loaded into your *MapData* array of TYPE. Now it's just a matter of calling the *Map_ShowMap* function. This function runs through the *MapData* array TYPE, grabs the image number to show, and then calls BB's DRAWIMAGE function with the *TileList* array type's image for that image number.

This code has other features in it that will allow for different scrolling techniques. We'll get into that in the next chapter!

Here's the code:

```
;*********************************************
;    Function: Map_ShowMap(...)
; Description: Displays a map starting at a certain
; X,Y offset, at a certain point within the map, and shows
; a certain number of rows and columns.
;
; Arguments:
;   XOffset = The X position to begin displaying the tiles.
;   YOffset = The Y position to begin displaying the tiles.
;   MapCollumnStart = Column of the map to display from
;   MapRowStart = Row of the map to display from
;   ShowWidth = How many columns to show
;   ShowHeight = How many rows to show
;   TileDimensions = how high and wide the tiles are...it's
;              considered square: 32x32, 16x16, 64x64, etc.
;*********************************************
Function Map_ShowMap(XOffset,YOffset,MapColumnStart, ↲
                     → MapRowStart,ShowWidth, ↲
                     → ShowHeight,TileDimensions)
   ; initialize X and Y to the arguments.  It's easier to
   ; type X,Y in code, but it's easier for the caller to
   ; understand what's expected by using XOffset,YOffset.
   ; Personal preference...
   RowPosition = 0
```

```
ColumnPosition = 0

; scrolling to the top edge, the offset will be less than 0
If MapRowStart <= 0
   ; so we want to get the screen coords of it here.
   YOffset = YOffset + (TileDimensions * Abs(MapRowStart))
   MapRowStart = 0
EndIf

; if it's the bottom edge
If (MapRowStart + (ShowHeight - 1)) >= MapHeight
   ; change the number of rows to show
   ShowHeight = MapHeight - MapRowStart
   If ShowHeight < 1
      ShowHeight = 1
   EndIf
EndIf

; scrolling to the left edge, the offset will be less than 0
If MapColumnStart <= 0
   ; so we want to get the screen coords of it here.
   XOffset = XOffset + (TileDimensions * Abs(MapColumnStart))
   MapColumnStart = 0
EndIf

; if it's the right edge
If (MapColumnStart + (ShowWidth - 1)) >= MapWidth
   ; change the number of columns to show
   ShowWidth = MapWidth - MapColumnStart
   If ShowWidth < 1
      ShowWidth = 1
   EndIf
EndIf

; initialize our X and Y values to the current offsets
X=XOffset
Y=YOffset

; for all of the rows (minus 1)
For Rows = MapRowStart To (ShowHeight+ ↲
          → MapRowStart) -1
    ; and for all the columns (minus 1)
    For Columns = MapColumnStart To  (ShowWidth + ↲
                → MapColumnStart) -1
        TileNumber = Map(Columns,Rows)\TileNumber
```

```
            If TileNumber <> -1
               DrawImage Tile(TileNumber)\Image,X,Y
            EndIf
            ; add the TileSize here
            X = X + TileDimensions
        Next
        ; reset X to the beginning of the next row
        X = XOffset
        ; increase our Y drawing position by the Tile's size
        Y = Y + TileDimensions
    Next
End Function
```

By changing the values in *MapColumnStart* and *MapRowStart* you can scroll the map.

Calling the Functions

From the caller's perspective, this library is easy to use. Just call that *Map_LoadMap* function with the appropriate arguments and you're set for loading. Then call *Map_ShowMap* with the appropriate arguments to display. If you need to change maps, just call *Map_LoadMap* again and it will clear out the old images and map data and load in the new.

The caller just needs to create a map file in the format described above, draw up some tiles, setup the *TileList* and *MapData* TYPES, and then put the *Map* function callers in at some point where they want to show the map. Here's an idea of it:

```
; initialize our graphics
Graphics 800,600

; setup our constants
Const TileSize = 32
Const MapDisplayColumns = 22
Const MapDisplayRows = 14

; include our map library
Include "../libraries/maplib.bb"

; Load images and a map file
Map_LoadTiles("alltiles.bmp",TileSize,0)
Map_LoadBinaryMap("map.dat")

; show the map
```

```
Map_ShowMap(0,0,0,0,MapDisplayColumns, ↲
            → MapDisplayRows,TileSize)

WaitKey () ; wait for a keypress
End
```

Try setting the map up to scroll. To do this you'll need to check for key presses on all the arrow keys and update the *MapColumnsStart* and *MapRowsStart* accordingly. If you use page-flip animation and a while loop, you'll see some map scrolling in action.

Hopefully this will give you some insight on a very simple map file. I also hope that you'll expand on this method and go for a much more heavyweight version!

Chapter 20: Moving Sprites on Tiled / Scrolling Backgrounds

When you're moving your player's image (often referred to as a *sprite*) around on your tiled map, you'll certainly going to want some objects to block paths. Maybe the player can't cross the water, or there are walls in the way, etc. Whatever you choose, there must be a way to prohibit the player from crossing certain boundaries.

Additionally, if you have a large map, you should be able to have the character move around the entire map. You need to be a little careful here because of visual aesthetics. If you let the player run to the edge of the screen before you begin scrolling, the player won't have the advantage of seeing what's coming up.

Most side-scrolling games handle this by making the player's character sit in the dead center of the screen until that player hits an edge of the map. From this point, there are a couple of ways to handle what happens.

1) Instead of continuing to scroll the map, the player's sprite will now have the ability to move away from the center of the screen until such time that the screen can again scroll.
2) There is a static backdrop so the map doesn't look odd having its edge sitting in the center of the screen.

Don't worry, I'll show examples of these different methods in this chapter. Before jumping into side-scrolling techniques, let's look at a single screen application.

***NOTE**: All of the routines in here rely on the map functions discussed in the previous chapter.*

Player hits a wall

There are a number of ways to handle collision checks on walls and other impassable objects. Here are a few:

1) Array-based checks: Since each tile is an element of an array (at least in the examples provided in this book), you can find out whether or not your player's next move will cause him/her to spill over to a tile that is a wall.
2) Pixel-based checks: You can use the IMAGESCOLLIDE command to see if your player is literally hitting a wall.
3) Box-based checks: You can set up specific depths that your character can overlap a wall before a collision is triggered.

I like the third option for 2D because it allows the most flexibility, so that's what I'm going to demonstrate here. Keep in mind that I'm talking about the player overlaying the tiles and such, not bullet collisions. That's a totally different thing altogether.

Here is a visual idea of what we'll be doing. In figure 1 we have our Swashbuckler surrounded by a bounding box:

(Figure 20.1)

Look carefully at the above graphic. We have a swashbuckler with a box around his mid-section. This box is not really in the sprite graphic, of course, it's just to give you an idea of what a bounding box is. In reality, all we really want to know for the bounding box is what the X1, Y1, X2, Y2 values are for the box. We're not even going to compare the actual sprite image to the wall image at all!

What we're going to do is find out where the player's sprite is on the screen and from there use basic math and IF…THEN…ENDIF statements to determine if there is an overlap. So again, the box shown in that graphic is only to convey the concept.

By using this method of collision detection, we have much more control over how much to overlap our walls. Here's an idea of what that would look like:


(Figure 20.2)

See how the sprite overlays the wall up to the top of the bounding-box shown in Figure 20.1? If we instead went with a method that checked the entire player, we would get something more like this:

(Figure 20.3)

Since the top of the character hits the bottom of the wall, there would be a collision, and that doesn't look nearly as nice.

In order to handle this type of collision checking effectively, we'll need four total boxes for the character: top, bottom, left, and right. I'm going to use an array to do this.

```
Dim PlayerCollisionArray(16)
```

Our sprite image is 32x32, which is why the numbers you see are 32 or less. Now you could certainly compare outside of the image space if you wanted to.

I've also set up a little function that initializes the bounding boxes for the sprite. You can alter the little numbers as you see fit.

```
;***********************************************
; FUNCTION: SetupPlayerBoundingBoxes()
;
; sets up where the x1,y1,x2,y2 values are for the 4 boxes
; that make up the collision points on the player
;
; Returns: n/a
;***********************************************
Function SetupPlayerBoundingBoxes()
 ; setup the top-box
 PlayerCollisionArray(0) = 2
 PlayerCollisionArray(1) = 15
 PlayerCollisionArray(2) = 30
 PlayerCollisionArray(3) = 20
 ; setup the bottom-box
 PlayerCollisionArray(4) = 2
 PlayerCollisionArray(5) = 30
 PlayerCollisionArray(6) = 30
 PlayerCollisionArray(7) = 32
```

```
 ; setup the Left-box
 PlayerCollisionArray(8) = 4
 PlayerCollisionArray(9) = 0
 PlayerCollisionArray(10) = 6
 PlayerCollisionArray(11) = 32
 ; setup the Right-box
 PlayerCollisionArray(12) = 26
 PlayerCollisionArray(13) = 0
 PlayerCollisionArray(14) = 28
 PlayerCollisionArray(15) = 32
End Function
```

Next we'll want to run through our map and setup the walls. First we'll setup a TYPE to hold them all.

```
; setup the walls type
Type Walls
   Field TileNumber
   Field X
   Field Y
End Type

; make it global
Global Wall.Walls = First Walls
```

Next we'll call our *SetupWalls* function that actually populates the TYPE. To do this you need to know the actual numbers for your walls. Then modify the following little function to include those numbers.

```
;************************************************
; FUNCTION: SetupWalls()
; sets up the Walls type to know which tiles are walls
; Returns: n/a
;************************************************
Function SetupWalls()
   X = 0
   Y = 0
   For Rows = 0 To MapHeight - 1
      ; and for all the columns (minus 1)
      For Columns = 0 To MapWidth - 1
         TileNumber = Map(Columns,Rows)\TileNumber
         Select TileNumber
            ; include all the wall numbers here!
            Case  0,1,2,3,5,6,7,8,10,11,12,13,15,16,17,18,20,21,22,23
```

```
                Wall.Walls = New Walls
                Wall\TileNumber = TileNumber
                Wall\X = X
                Wall\Y = Y
            End Select
            X = X + 32
        Next
        ; reset X to the beginning of the next row
        X = 0
        ; increase our Y drawing position by the Tile's Height
        Y = Y + 32
    Next
End Function
```

It's important that you keep the X and Y values updating accordingly in this function, as these values will designate where your walls are on the map.

Now that we have our walls in place and accounted for, we just need a function that compares where the player's sprite is in relation to them. If any of the areas overlap, we simply stop the player's movement. The following function returns a value of 1 if an overlap is detected, and a value of 0 if there is no overlap.

```
;*********************************************
; FUNCTION: CheckWallCollisions(...)
;
; Checks an area for collision based on bounding boxes
;
; Arguments:
;   X,Y: The current world X,Y position
;
; Returns:
;         0 = no hit
;         1 = collision with wall
;*********************************************
Function CheckWallCollision(X,Y)

    ; do one calculation here for each absolute box position
    ; so we don't do them every itteration of our loop below
    BoxTopX1 = X + PlayerCollisionArray(0)
    BoxTopY1 = Y + PlayerCollisionArray(1)
    BoxTopX2 = X + PlayerCollisionArray(2)
    BoxTopY2 = Y + PlayerCollisionArray(3)
    BoxBottomX1 = X + PlayerCollisionArray(4)
```

```
BoxBottomY1 = Y + PlayerCollisionArray(5)
BoxBottomX2 = X + PlayerCollisionArray(6)
BoxBottomY2 = Y + PlayerCollisionArray(7)
BoxLeftX1 = X + PlayerCollisionArray(8)
BoxLeftY1 = Y + PlayerCollisionArray(9)
BoxLeftX2 = X + PlayerCollisionArray(10)
BoxLeftY2 = Y + PlayerCollisionArray(11)
BoxRightX1 = X + PlayerCollisionArray(12)
BoxRightY1 = Y + PlayerCollisionArray(13)
BoxRightX2 = X + PlayerCollisionArray(14)
BoxRightY2 = Y + PlayerCollisionArray(15)

; run through all of the walls
For Wall.Walls = Each Walls
   ; set our collision values = 0
   XCollision = 0
   YCollision = 0

   ; grab the Type values to speed things up a bit
   WallX = Wall\X
   WallY = Wall\Y
   ; calculate WallWidth and Height to speed things up
   WallWidth = WallX + TileSize
   WallHeight = WallY + TileSize

   ; check the top bounding box
   If BoxTopX1 >= WallX And BoxTopX2 <= WallWidth
     XCollision = 1
   EndIf
   If BoxTopY1 >= WallY And BoxTopY2 <= WallHeight
     YCollision = 1
   EndIf

   ; check the bottom bounding box
   If BoxBottomX1 >= WallX And BoxBottomX2 <= WallWidth
      XCollision = 1
   EndIf
   If BoxBottomY1 >= WallY And BoxBottomY2 <= WallHeight
     YCollision = 1
   EndIf

   ; check the left bounding box
   If BoxLeftX1 >= WallX And BoxLeftX2 <= WallWidth
      XCollision = 1
   EndIf
```

```
    If BoxLeftY1 >= WallY And BoxLeftY2 <= WallHeight
       YCollision = 1
    EndIf

    ; check the right bounding box
    If BoxRightX1 >= WallX And BoxRightX2 <= WallWidth
       XCollision = 1
    EndIf

    If BoxRightY1 >= WallY And BoxRightY2 <= WallHeight
       YCollision = 1
    EndIf

    ; if there is a collision on both the X and Y axis, return 1
    If XCollision = 1 And YCollision = 1
      Return 1
    EndIf
  Next

 ; no hit so return 0
 Return 0
End Function
```

To make this a faster routine you could put all of the collisions on one line using parenthesis and *one* IF statement. I separated it out here to make it easier to read and follow.

Single Screen Games

A single screen game is a game that doesn't scroll. The map stays mostly as is and the player can only move around within its confines. "PacMan" is the perfect example. The screen doesn't scroll, and it doesn't change until that map/level is cleared.

We already know how to load and display a map screen, but how do we move the character and make the appropriate wall checks? What we do is first save the player's current X and Y locations. Then we update the X and Y's by a certain pixel distance. Then we see if there is a collision under the new position. If there is, we revert back to the old X, Y values. If not, we make the updated X, Y values our player's new position and display the sprite accordingly. Here's the code:

```
;*********************************************
; FUNCTION: MovePlayer(...)
;
```

```
; This function moves the player's character by a certain
; number of pixels.
;
; Arguments:
;   Direction: 1=up,2=down,3=left,4=right
;   Distance: how many pixels to move?
;*******************************************
Function MovePlayer(Direction,Distance)
    ; player is moving up
    If Direction = 1
      ; save the current World value
      OldY = WorldY
      ; calculate the new value based on the distance
      WorldY = WorldY - Distance
      ; see if there's a collision at the new position
      HitWall = CheckWallCollision(WorldX,WorldY)
      ; if so, reset the World value to what it was prior
      ; to the collision
      If HitWall = 1
        WorldY = OldY
      EndIf
    EndIf

    ; player is moving down
    If Direction = 2
      OldY = WorldY
      WorldY = WorldY + Distance
      HitWall = CheckWallCollision(WorldX,WorldY)
      If HitWall = 1
        WorldY = OldY
      EndIf
    EndIf

    ; player is moving left
    If Direction = 3
      OldX = WorldX
      WorldX = WorldX - Distance
      HitWall = CheckWallCollision(WorldX,WorldY)
      If HitWall = 1
         WorldX = OldX
      EndIf
    EndIf

    ; player is moving right
    If Direction = 4
```

```
      OldX = WorldX
      WorldX = WorldX + Distance
      HitWall = CheckWallCollision(WorldX,WorldY)
      If HitWall = 1
         WorldX = OldX
      EndIf
   EndIf
End Function
```

From here it's just a case of loading in the map and the character and then processing any input.

Screen and World Coordinates

The concept of screen and world coordinates can be tricky, so let's cover that first.

Screen coordinates are where on the screen the player (or some other object) will be displayed.

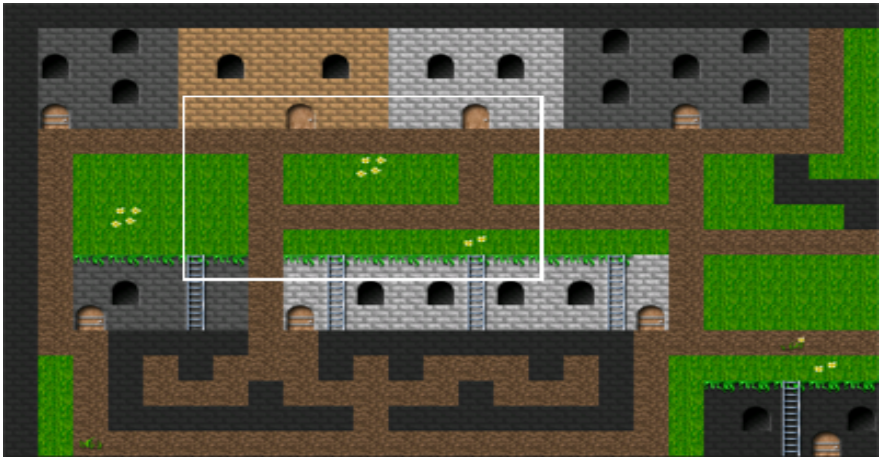
World coordinates denote the X, Y position the player is in the world. So while the player may be sitting in the center of the screen, he may be near the bottom right of a big map. This is important because the map location of the player will indicate what tiles are shown, where the enemies or traps are, etc.

Imagine that we have a map that is 100 tiles wide by 100 tiles tall. Each tile is 32x32, so in essence we have a map that is 3200 pixels by 3200 pixels, right? Now the section of the screen we're going to use to display the map (known as a "view port") is 20 wide by 15 high. Taking our 32x32 images, this means that we'll only be seeing 640 pixels on the X-axis and 480 pixels on the Y-axis. Knowing this, we need a way in which to display the rest of the pixels as we move across the map. Since we are starting off at pixel 0,0, we know that one move to the right would put us at pixel 1,0. But if we move the player along with the pixel movement, the player's sprite will soon leave the screen.

So instead, we use two coordinates. To determine which tiles of the map, which bad guys or NPC (non-player characters), etc., are drawn, we use the world coordinates. To actually draw the player and the currently visible map tiles we use the screen coordinates. There is a little bit of calculation involved to get this all to work. First, though, let's talk about the concept behind scrolling a map.

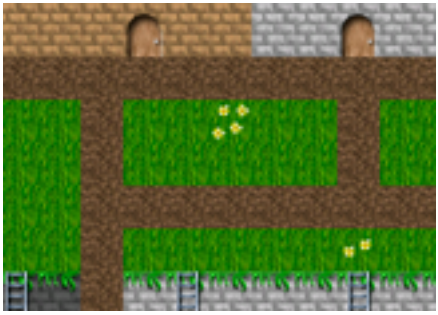
Scrolling a Map

I once had tons of trouble understanding the concept behind how a map scrolls, but then I read an explanation that made it all fall together. I'll try to re-tell that here!



(Figure 20.1)

The white box in Figure 20.1 is to denote the view port area. This is the area we are going to actually display to the user. The upper left corner of the map is 0,0, but the upper left corner of the view port (white box) is 83,60. This means that our world coordinate is 83,60. Let's say that our view port's top left position is 10,10. This way we can have menus, title bars, heads-up-display (also known as HUD), chat, etc. So the actual view port will show this:



(Figure 20.2)

Now all you have to do is imagine moving that white box around pixel-by-pixel (or 4 pixels or whatever), redrawing the view port data from the new world coordinates, and bam…you have the concept of 2D map scrolling under your hat! Hopefully that will also help you to better understand the screen and world coordinate concept.

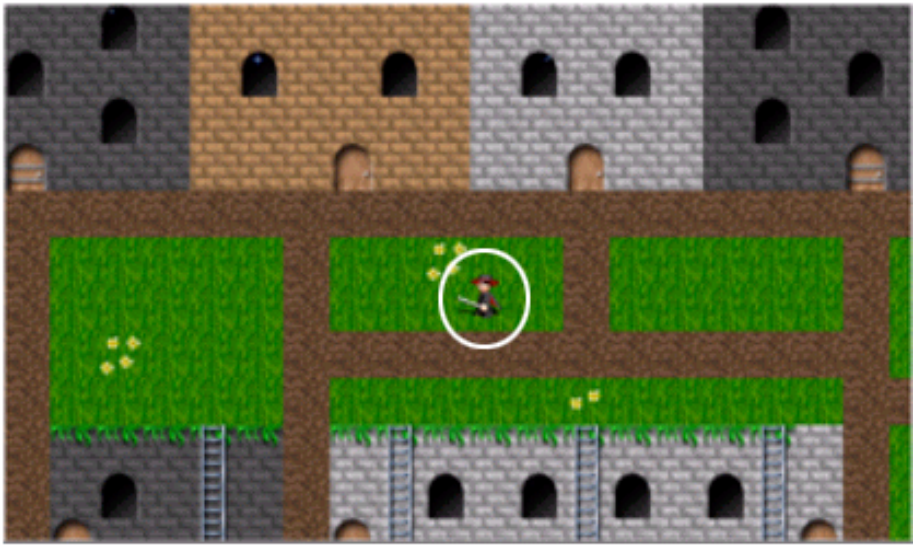
Scrolling Types

Most of you have played side-scrolling games. You run your character around in a level as it scrolls all over the place. The techniques behind doing this used to be somewhat complex, but with Blitz handling the basic tile drawing and such it's now really quite simple.

There are different types of scrolling maps. You've got the static screen type that stays static until the player hits a door, and then a quick scroll occurs. This isn't exactly what I'd call a side-scrolling game, but it can certainly be cool nonetheless. I won't be getting into this type here, but rest assured that the functions provided could make this type of game a snap to code with just a little playing around.

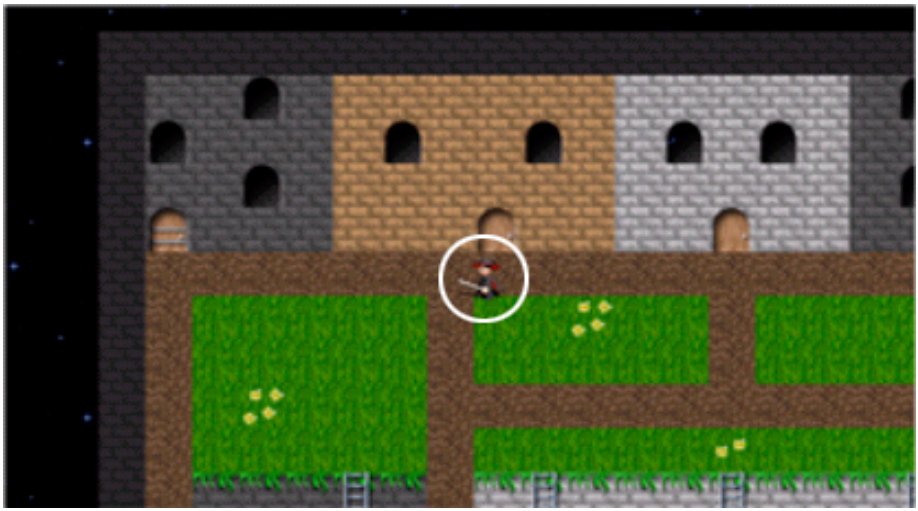
The second type is where the character stays in the center while the map scrolls around him. When an edge is reached, the character still stays in the center but the map scrolls in towards him. For the sake of a naming convention, I'm going to call this type "To the Edge Scrolling."

Here are a few images to demonstrate this concept:



(Figure 20.3)

Here the player sits in the center of the screen and there are no visible edges.



(Figure 20.4)

The player is still in the center of the screen, but we can now see the edges. Notice also that there is a star field in the background. This makes it appear that our world is floating in space. Okay, I admit a "Space Swashbuckler" is a little goofy, but it's simply to demonstrate a concept. ☺

(Figure 20.5)

Finally, in figure 20.5, we have our player colliding with the wall and the building. The star field is more than half the screen now. But the player is still in the center of the map screen. So, in essence we've moved the map literally around the player. The player's position relative to the map screen has not changed, but relative to the actual map it has.

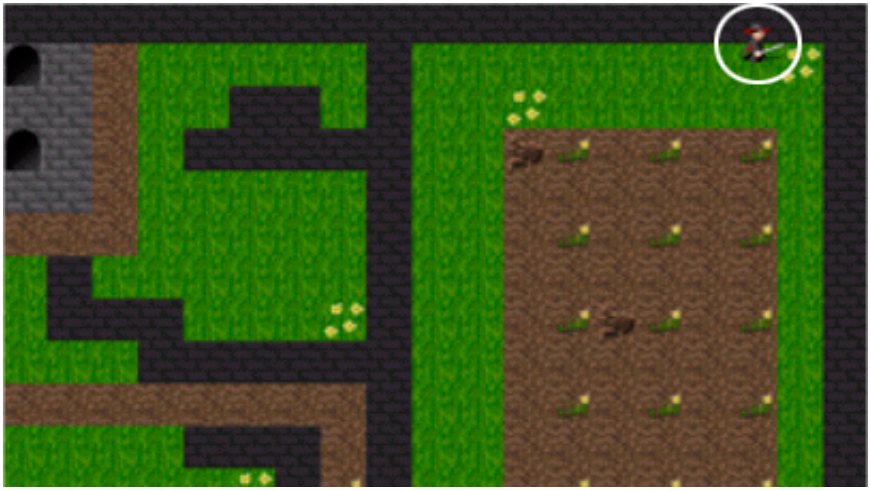
The third type is very similar to the second, but there is one major difference: the map only scrolls until the player reaches a point in the world where the edge is at the side of the view port. From here the player moves independently of the map. This type I'll call "Edge Independent Scrolling."


(Figure 20.6)

Again we see our player is sitting in the center of the view port, but we can see the edge of the map. So what happens when the player moves to the right?

(Figure 20.7)

Notice that the map has not changed, but the player's position has. The map edges used to move all the way to the player, but now when the player reaches an edge, the map stays put and the player will move instead. Here is another view to demonstrate this:



(Figure 20.8)

See how the player is now in the upper-right and not in the center? This is the concept that I'm trying to get across.

When the player gets back to a point that he's crossing the center of the screen, from either the X-axis or the Y-axis, the map will scroll accordingly.

Scrolling Code
Both of these methods will require some changes to the *MovePlayer* code shown above, and we'll also be adding in a new function called *MoveMap*.

In the "To the Edge" scrolling style the player stays in the center all the time, so we need to bring the walls and other tiles to him. This also means that we need to update the map's origin each move and check for collisions. Before we brought the player to the wall to collide, but now we have to bring the walls to the player.

In the "Edge Independent" style the player stays in the center until an edge hits the view port and then the player moves independently. So this is a combination of the static screen and "To the Edge" types. We have to sometimes bring the walls to the player and other times bring the player to the walls.

Fortunately our code will work in either case. Here's what the new *MovePlayer* code looks like:

```
;***********************************************
; FUNCTION: MovePlayer(...)
;
; This function moves the player's character by a certain
; number of pixels.
;
; Arguments:
;   Direction: 1=up,2=down,3=left,4=right
;   Distance: how many pixels to move?
;***********************************************
Function MovePlayer(Direction,Distance)

    ; player is moving up
    If Direction = 1
      ; save the current World value
      OldY = WorldY
      ; calculate the new value based on the distance
      WorldY = WorldY - Distance
      ; make sure the new value is >= 0
      If WorldY < 0
        WorldY = 0
      EndIf
      ; see if there's a collision at the new position
```

```
    HitWall = CheckWallCollision(WorldX,WorldY)
    ; if so, reset the World value to what it was prior
    ; to the collision
    If HitWall = 1
      WorldY = OldY
    Else
      ; otherwise scroll the map
      MoveMap(Direction,Distance)
    EndIf
  EndIf

  ; player is moving down
  If Direction = 2
    OldY = WorldY
    WorldY = WorldY + Distance
    HitWall = CheckWallCollision(WorldX,WorldY)
    If HitWall = 1
      WorldY = OldY
    Else
      MoveMap(Direction,Distance)
    EndIf
  EndIf

  ; player is moving left
  If Direction = 3
    OldX = WorldX
    WorldX = WorldX - Distance
    If WorldX < 0
      WorldX = 0
    EndIf
    HitWall = CheckWallCollision(WorldX,WorldY)
    If HitWall = 1
      WorldX = OldX
    Else
      MoveMap(Direction,Distance)
    EndIf
  EndIf

  ; player is moving right
  If Direction = 4
    OldX = WorldX
    WorldX = WorldX + Distance
    HitWall = CheckWallCollision(WorldX,WorldY)
    If HitWall = 1
      WorldX = OldX
```

```
      Else
        MoveMap(Direction,Distance)
      EndIf
    EndIf
End Function
```

The primary difference here is that if we haven't hit a wall, we move the map a step. What that does is draws the map from the new origin, and moves the player's world coordinates accordingly. Without doing this important step, we won't have working collisions.

Next the *MoveMap* code handles actually changing the place in the *Map* array that we are at. Since we are moving the map until there is a collision, we don't have to worry too much about player/tile placements. Here's the code:

```
;***********************************************
; FUNCTION: MoveMap(...)
;
; This function moves map array around appropriately, and
; sets up the values for the next call to ShowMap.
;
; Arguments:
;   Direction: 1=up,2=down,3=left,4=right
;   Distance: how many pixels to move?
;***********************************************
Function MoveMap(Direction, Distance)

    ; player is moving up
    If Direction = 1
      ; see if the world coordinates will force the player
      ; to move independtly
      If MapY > LowestMapY And ScreenY = ScreenCenterY
        ; if not update the map position based on distance
        MoveMapY = MoveMapY + Distance
        ; if the Move position >= 0 then move the overall
        ; position value
        If MoveMapY >= 0
          MapY = MapY - 1
          MoveMapY = -TileSize
        EndIf

      Else  ; the player is moving independently

        ; if the player is in the upper-left of the map,
```

```
      ; just set the screen coordinate to equal the
      ; world coordinate.
      If WorldY <= ScreenCenterY
         ScreenY = WorldY
         ; if the player has pasted the center point away
         ; from the edge, make sure that the player's new
         ; center point is the center of the screen.
         If ScreenY >= ScreenCenterY
            ScreenY = ScreenCenterY
         EndIf
      EndIf

      ; if the player is in the lower-right of the map,
      ; set the screen coordinate to be the width of the
      ; map in pixels minus the world coordinate,
      ; subtracted from the center of the screen.  Then take
      ; that value and add it to the actual screen center
     If WorldY >= FullMapHeight - ScreenCenterY
        ScreenY = (ScreenCenterY - (FullMapHeight - ↲
                  → WorldY )) + ScreenCenterY
         ; if the player has pasted the center point away
         ; from the edge, make sure that the player's new
         ; center point is the center of the screen.
         If ScreenY <= ScreenCenterY
            ScreenY = ScreenCenterY
         EndIf
      EndIf

      ; now move the map position based on distance
      MoveMapY = MoveMapY + Distance
      If MoveMapY > -TileSize
         MoveMapY = -TileSize
      EndIf
   EndIf
EndIf

; player moving down
If Direction = 2
   If MapY < HighestMapY And ScreenY = ScreenCenterY
      MoveMapY = MoveMapY - Distance
      If MoveMapY <= -(TileSize * 2)
         MapY = MapY + 1
         MoveMapY = -TileSize
      EndIf
   Else
```

```
      If WorldY <= ScreenCenterY
         ScreenY = WorldY
         If ScreenY >= ScreenCenterY
            ScreenY = ScreenCenterY
         EndIf
      EndIf
      If WorldY >= FullMapHeight - ScreenCenterY
         ScreenY = (ScreenCenterY - (FullMapHeight - ↲
                  → WorldY )) + ScreenCenterY
         If ScreenY <= ScreenCenterY
            ScreenY = ScreenCenterY
         EndIf
      EndIf
      MoveMapY = MoveMapY - Distance
      If MoveMapY < -TileSize
         MoveMapY = -TileSize
      EndIf
   EndIf
EndIf

; player moving left
If Direction = 3
   If MapX > LowestMapX And ScreenX = ScreenCenterX
      MoveMapX = MoveMapX + Distance
      If MoveMapX >= 0
         MapX = MapX - 1
         MoveMapX = -TileSize
      EndIf
   Else
      If WorldX <= ScreenCenterX
        ScreenX = WorldX
        If ScreenX >= ScreenCenterX
           ScreenX = ScreenCenterX
        EndIf
      EndIf
      If WorldX >= FullMapWidth - ScreenCenterX
         ScreenX = (ScreenCenterX - (FullMapWidth - ↲
                  → WorldX)) + ScreenCenterX
         If ScreenX <= ScreenCenterX
            ScreenX = ScreenCenterX
         EndIf
      EndIf
      MoveMapX = MoveMapX + Distance
      If MoveMapX > -TileSize
         MoveMapX = -TileSize
```

```
        EndIf
      EndIf
   EndIf

   ; player moving right
   If Direction = 4
      If MapX < HighestMapX And ScreenX = ScreenCenterX
         MoveMapX = MoveMapX - Distance
         If MoveMapX <= -(TileSize * 2)
            MapX = MapX + 1
            MoveMapX = -TileSize
         EndIf
      Else
         If WorldX <= ScreenCenterX
            ScreenX = WorldX
            If ScreenX >= ScreenCenterX
               ScreenX = ScreenCenterX
            EndIf
         EndIf
         If WorldX >= FullMapWidth - ScreenCenterX
            ScreenX = (ScreenCenterX - (FullMapWidth - ↲
                  → WorldX)) + ScreenCenterX
             If ScreenX <= ScreenCenterX
                ScreenX = ScreenCenterX
             EndIf
         EndIf
         MoveMapX = MoveMapX - Distance
         If MoveMapX < -TileSize
            MoveMapX = -TileSize
         EndIf
      EndIf
   EndIf
End Function
```

So how do we control whether or not the map scrolls all the way to the edge or if it only scrolls to a certain point before making it edge independent? We setup the edges ahead of time. In other words, we tell the code where we want the edges to stop.

These styles will require you to tweak until you like the outcome. This is due to the amount of pixels you move per step, your view port size, etc. The following demonstrates how I tweaked it to work well with the view port in the scrolling demos.

```
; find out the full width, taking  into account tile size
; information.  NOTE: This may ; need to be tweaked to your
; visual liking
Global FullMapWidth=(MapWidth * TileSize) + TileSize / 2
Global FullMapHeight=(MapHeight * TileSize) + TileSize / 2
```

For "To the Edge" scrolling, put in numbers that are far from the edges. Since the walls will stop the player before ever getting to an "out of bounds" situation, we don't have to worry about how far out we go. Here is some setup code for "To the Edge" scrolling:

```
; The array values for X and Y will determine how far left,
; right,up,down to scroll past the edges.  This gives you
; control over where to stop the edge detection. -20 and
; Widths/Heights + 20 will bring the map all the way into
; the player, for example

; the lowest the map can move on the X axis?
Const LowestMapX = -20
; what's the lowest the map can move on the Y axis?
Const LowestMapY = -20
; what's the highest X?
Global HighestMapX = (MapWidth - MapColumns) + 20
; what's the highest Y?
Global HighestMapY = (MapHeight - MapRows) + 20
```

Obviously if we really got to –20 our code would error out, but that's why the *Map_ShowMap* function has that extra array index check with absolute values.

The "Edge Independent" method needs to be setup quite differently.

```
; The array values for X and Y will determine how far left,
; right,up,down to scroll past the edges.  This gives you
; control over where to stop the edge detection. -1 and
; MapWidths/Rows + 3 will scroll to the edge and then allow
; the player to move independtly without the screen scroll.

; what's the lowest the map can move on the X axis?
Const LowestMapX = -1
; what's the lowest the map can move on the Y axis?
Const LowestMapY = -1
; what's the highest X?
Global HighestMapX = (MapWidth - MapColumns) + 3
; what's the highest Y?
```

Global HighestMapY = (MapHeight - MapRows) + 3

The *HighestMapX* and *HighestMapY* values may need to be tweaked a bit to fit within your dimensions. This should be pretty simple to do though.

Note that these must all be called *after* you've loaded your maps. If you try to initialize these ahead of time, you'll have problems.

Chapter 21: Creating a *MapMaker* Program

Since you will certainly want to have a way to create your maps visually, I thought it would make sense to include a chapter on creating a program that handles this sort of thing. Note that this chapter will focus more on theory than code. There is full source code to everything talked about here on the CD though, so don't worry!

Features
The first thing to consider when developing your Map Making system is what features to have. Will you allow multiple tile sets, Z-Ordering, backdrops, etc.?

This topic could span many chapters if we discussed all that we could include in a *MapMaker*, so I'm going to stick with the basics and let you take it and expand from there.

The *MapMaker* we'll be creating will have the ability to load and save maps, user-controlled spacing between tiles to account for different types of tile sets, a grid display option, and we'll make it so the user can use the keyboard for great scrolling speed…both for the map and the tiles. So, yes, it's basic stuff, but it's good enough to get us started!

Dimensions of Tiles and Map Area
Don't let this important step go by untouched. You have to know what tile sizes you'll be supporting for a number of reasons. One is that wherever the user is to select tiles from, those tiles must be laid out correctly. Second is that the map area must be able to hold tiles without ugly overlapping. Lastly, for calculation purposes and array placement. If you don't know what size tiles you're working with, your array placements will be way off.

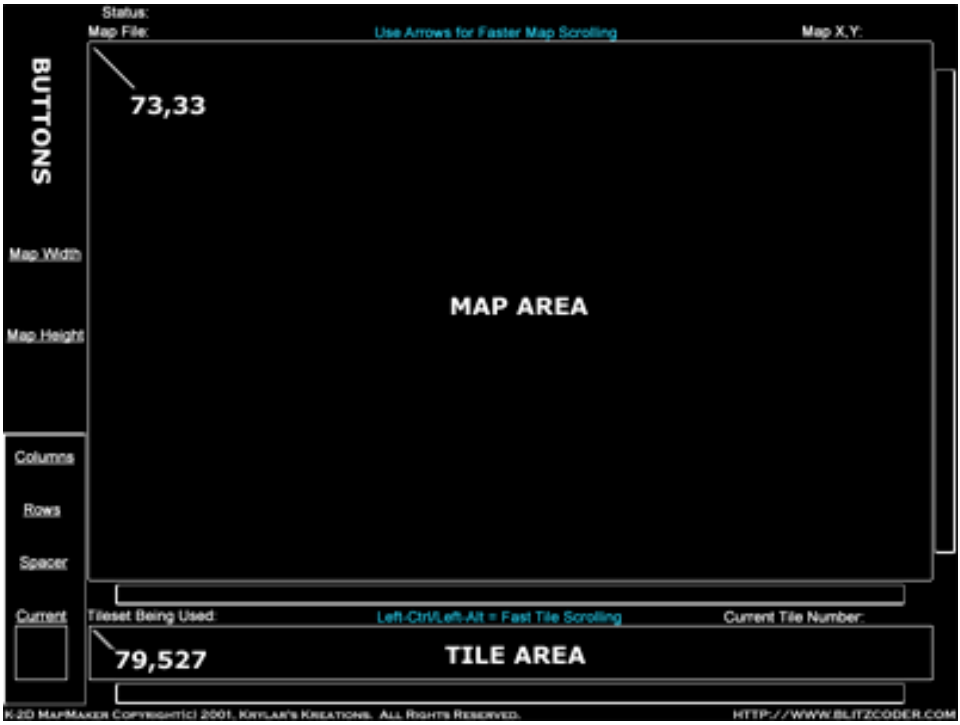
I'm going to be using tile dimensions of 32x32 in this section.

The image in figure 21.1 shows the basic layout that I'll be using in the *MapMaker*. Notice the big square in the center named "map area." This will show where we'll be displaying the actual map the user is creating. The upper-left corner of that area is positioned at 73, 33. This is *very* important to know because not only will our tiles begin displaying here, we'll need this data when dealing with the placement of the tiles in the array.

On the left side of the window is the "buttons" area. Here we will display the various options that the user has while manipulating maps.

At the bottom is the "tiles area." This area will display a number of the tiles from the user's tile set. This area, like the map area, will be scrollable. Note again that we have the upper left corner of this area at 79, 527. This is only for tile placement visually and has no impact on the array data.

The rest of the screen has additional information for the current X, Y mouse location, the columns and rows of the tile set, the spacer used between tiles, the current tile number, the map width and height info, etc.



(Figure 21.1)

Note also that there are a few rectangles shown to the right and bottom of the map area, and one under the tile area. We'll use these as location indicators. They will have appropriately facing arrows by them and a little blue box in them. This blue box will move to the left and right as you scroll the maps and/or tiles. You can compare this to the scrolling indicator you see when scrolling a document or a web site.

Handling Buttons

Since we will need to open a tile set, we may as well discuss how to give the user a button that does this.

I have created a little library called "*buttons.bb*" that handles buttons. The concept behind this library was to make it so the developer could display a graphical button at a particular place on the screen and await a press from the user. The image associated to the button *must* have two frames, one for the pushed-in look and the other for the popped-out look. Here is an example:

(Figure 21.2)

There is a function that will place this image on the screen, keep track of it, and return user-specified information if that button has been clicked. It's the *Buttons_Create* function. Here is the format:

Buttons_Create(*Set, Image, X, Y, Type, ReturnValue, DelayAmount****)***

Set: This argument details the *set* of buttons that this button will belong to. This is important because you may have numerous sets of buttons.

Image: This is the actual image you will be using for this button. You must make sure to include the *full path* for this image or you will experience problems.

X, Y: These control the upper-left corner of button placement on the screen.

ReturnValue: The value you specify to be returned whenever a user clicks this button.

DelayAmount: How long to delay between the two images (for that pressed-in look).

Now, after creating a button you will need a way to show that button to the user. This is handled with the *Buttons_Show* function. All you do is call that function when you want your buttons displayed. Make sure you pass the *set* you wish displayed as well.

The *Buttons_Check* will see if any buttons in a certain set have been clicked. Without this function you would just have a bunch of useless buttons on the screen.

Finally, make sure to delete the buttons using the *Buttons_Delete* function. The last thing you want is a bunch of buttons taking up memory unnecessarily.

Here is a little piece of code that demonstrates the use of the buttons library:

```
Include "buttons.bb"

Graphics 800,600

SetBuffer BackBuffer()

SetupButtons()
OkayValue$ = "Not Pressed"
Set1Selected$ = "Plus"
Set2Selected$ = "Yes"

While Not KeyHit(1)
  Cls
  ; draw the buttons!
  Buttons_Show("Cancel")
  Buttons_Show("Okay")
  Buttons_Show("Set1")
  Buttons_Show("Set2")

  If MouseHit(1)

    ReturnValue$ = Buttons_Check$("Cancel") ; check the buttons
    If ReturnValue$ = "Cancel"
      Cancels = Cancels + 1
    EndIf

    ReturnValue$ = Buttons_Check$("Okay") ; check the buttons
    Select ReturnValue$
        Case "Okay0"
            OkayValue$ = "Not Pressed"
        Case "Okay1"
            OkayValue$ = "Pressed"
    End Select

    ReturnValue$ = Buttons_Check$("Set1") ; check the buttons
    Select ReturnValue$
        Case "Plus"
            Set1Selected$ = "Plus"
        Case "Minus"
            Set1Selected$ = "Minus"
    End Select

    ReturnValue$ = Buttons_Check$("Set2") ; check the buttons
    Select ReturnValue$
```

```
        Case "Yes"
            Set2Selected$ = "Yes"
        Case "No"
            Set2Selected$ = "No"
        Case "Ok"
            Set2Selected$ = "Ok"
    End Select


  EndIf
  Text 0,500,"X = " + MouseX() + ", Y = " + MouseY()
  Text 0,520,"Cancel hits = " + Cancels
  Text 0,540,"Okay = " + OkayValue$
  Text 0,560,"Set 1 Selected = " + Set1Selected$
  Text 0,580,"Set 2 Selected = " + Set2Selected$
  Flip
Wend

Buttons_Delete("Cancel")
Buttons_Delete("Okay")
Buttons_Delete("Set1")
Buttons_Delete("Set2")

End

; setup the buttons using the "Buttons_Create" function
Function SetupButtons()
   Buttons_Create("Cancel","dircancel.png",0,10,0,"Cancel",250)

   Buttons_Create("Okay","dirok.png",0,50,1,"Okay",25)

   Buttons_Create("Set1","plus.png",200,10,2,"Plus",25)
   Buttons_Create("Set1","minus.png",200,50,2,"Minus",25)

   Buttons_Create("Set2","yesbuttons.png",400,10,2,"Yes",25)
   Buttons_Create("Set2","nobuttons.png",400,50,2,"No",25)
   Buttons_Create("Set2","dirok.png",400,100,2,"Ok",25)
End Function
```

Directory Control

Getting directory information isn't all that difficult, but making it visually appealing can be a little challenging. I created a library to help get you started though. I've called it "*directories.bb*" and you can find it on the CD.

The object behind this library was to provide the user a visual means of selecting both tile sets and map sets for loading in and processing. This means that I needed to put together a couple of images, such as a little window, some arrows, "ok" and "cancel" buttons, etc. No big deal, but you'll probably want to tailor these images to suit your needs.

Next I needed to figure out a way to find all the disk drives available on the computer. I set up a TYPE as follows:

```
Type DirDrives
   Field DriveNumber
   Field DriveName$
End Type
```

This TYPE will hold the physical number of the drive and the name of that drive, which we will manipulate a little. To populate this TYPE, I used the following code:

```
; This is the type that contains all of the files in a directory
 ; run through all the drives to see which drives are available
 For DriveLoop = 2 To 26
    DriveLetter$=Chr$(DriveLoop+64)+":"
    Dir = ReadDir(DriveLetter$)
    If Dir
      Drives.DirDrives = New DirDrives
      Drives\DriveName$ = DriveLetter$ + "\"
      Drives\DriveNumber = TotalDrives
      CloseDir Dir
      ; we still need to update the TotalFiles piece because the
      ; DIR is treated like a file for listing.  You may wanna change
      ; this if you remove the DIR from the file list.
      TotalDrives = TotalDrives + 1
    EndIf
 Next
```

I started off with *DriveLoop* set to 2 to avoid calling on the floppy drives. You may change this value to include the floppy drives if you wish. Study the commands on that listing to see what it's doing in detail.

The next step was to use the Blitz READDIR command to open a directory for reading. After the directory is opened for reading, the use of NEXTFILE and FILETYPE help to determine the rest of the file information to build the directory with. Here is the code:

```
; Open the current directory for reading
DirNumber=ReadDir(Directory$)

; set a flag so we can keep track of the files being listed
Files_Being_Listed = 1

; while we're still listing files, do all this stuff
While (Files_Being_Listed = 1)
      ; grab the NextFile in the directory
      File$ = NextFile$(DirNumber)
      ; if the filename doesn't equal "" (or blank)
      If File$ <> ""
         ; see what type of file it is
         If FileType(File$) = 2
            ; if it's a regular old file, fill in the
            ; DirFiles Type here
            If File$ <> "."
               DriveFound = 0
               For Drives.DirDrives = Each DirDrives
                   If Left$(Drives\DriveName$,2) = File$
                      DriveFound = 1
                   EndIf
               Next

               If DriveFound = 0
                  Files.DirFiles = New Dirfiles
                  Files\FileNumber = TotalFiles
                  Files\TypeOfFile = 2
                  Files\Filename$ = File$
                  TotalFiles = TotalFiles + 1
               EndIf
            EndIf
         EndIf
      Else
         ; if the filename returned DOES equal "" (blank), then
         ; we are done listing files, so set the var to break the loop
         Files_Being_Listed = 0
      EndIf
Wend
```

There's a lot to that, so study it carefully. The main thing to note is that we are filling up the *DirFiles* TYPE. Here is what that TYPE looks like:

```
Type DirFiles
```

```
    Field FileNumber
    Field TypeOfFile
    Field Filename$
End Type
```

So you can see that we can now just display our file listing by running through that TYPE and placing the text in the appropriate window. This is done using the *Directory_Show* function. Also, we need to check if the user is selecting files, scrolling, canceling, etc., so we use the *Directory_CheckMenu* function for that.

Here is a little code that will load up a directory, let you select from it and then display what you selected. You will need to have all of the graphical assets in to use this, so it's best to grab them from the disk.

```
Include "buttons.bb"
Include "directories.bb"

Graphics 800,600

SetBuffer BackBuffer()

; use a mouse pointer for full-screen mode
Global Mouse_Image = LoadImage("mouseptr.bmp")

Cls
; call the directory display function with an "*" for ALL FILES
FileSelected$=Directory_Main$("*",ScreenWidth,ScreenHeight, ↲
                                → "Open Map File")

Cls
; show the user what they selected
Text 0,0,"You selected the file: " + FileSelected$
Flip

WaitKey()
End
```

Tile Set Placement and Size Restraints

When we load in tiles, we certainly won't be able to fit them all on the screen…at least not in most cases. So, we need to have a way to scroll them. But that opens a whole new can of worms. If we scroll them, then how will the program know which one we're selecting?

There are a couple of ways to handle this. You can do real-time calculations to see where you are in the array, you can use *world* coordinate systems, or you can use offsets. I like the last method because it's clean and easy to use. So let's study that method.

First we start by showing the tiles. To do this we need to know the upper-left edge of the location to display from. We're in luck because if you refer back to Figure 21.1, you can see that the tile area has an upper-left edge of 79,527. This is great because now we know that all we have to do is set the default for X to 79 and the default for Y to 527.

Now we have to determine how many tiles we can display in the tile area at once. Since we know our tiles are 32x32, we now just need to determine how wide the tile area is and we can divide that by 32 accordingly. The width of that area happens to be 777. If we take our left edge of 79 and subtract it from the width (777), we'll get 704. This is important because 704 is divisible *evenly* by 32. 704 / 32 = 22. This means that we can display 22 tiles in the tile area if we use no spacing. I drop this to 21 to save room for spacing.

Using Offsets

We need to be a little careful when displaying these tiles because we want to ensure that our FOR…NEXT loop starts at the appropriate tile to begin from. This is where the offsets come into play.

We only need to track a single offset for the tile display because we are only displaying a single row. In other words, we need only track the column we are on, not the column and the row. To do this, I created a global variable called *TileColumn*. Any time the player hits the tile scrolling arrows or holds down the corresponding tile scrolling keys, I update the value of this variable. But we also have to make sure that we don't go too high or too low on our updates, so here is a little snippet that demonstrates how we handle this:

```
Case "TileLeft"
      TileColumn = TileColumn - 1
      If TileColumn <= 0
         TileColumn = 0
       EndIf

Case "TileRight"
      TileColumn = TileColumn + 1
      If (TileColumn + TileDisplaySize) >= TotalTiles
         TileColumn = TotalTiles - TileDisplaySize
          If TileColumn < 0
             TileColumn = 0
          EndIf
```

```
    EndIf
```

That is taken directly from a SELECT statement that is used in conjunction with our button return values described previously. The *TileDisplaySize* value is a global value that I set to 21 (the number we determined would fit nicely in the tile area). The *TotalTiles* value is calculated for us by our *Map_LoadTiles* routine we discussed in Chapter 19.

So here we're just doing checks to ensure that we're not going to overrun our array, but we also take care of the problem of where to stop scrolling the tiles. If we don't take the *TileDisplaySize* into account, the last tile will scroll all the way to the left edge of the tile area. That looks rather lame, in my opinion.

Showing the Tile Set

Well, now that we have the information for the tile locations, how do we go about scrolling them in the tile area? The following code takes care of this displaying for us.

```
Function ShowTiles(XOffset,YOffset,TileStart,ShowWidth)

 ; save our color values
 Red = ColorRed()
 Blue = ColorBlue()
 Green = ColorGreen()

 ; reset our TYPE so we can hit the proper offset location
 For TileSelection.TileSelections = Each TileSelections
     Delete TileSelection
 Next

 ; initialize X and Y to the arguments.  It's easier to
 ; type X,Y in code, but it's easier for the caller to
 ; understand what's expected by using XOffset,YOffset.
 ; Personal preference
 X=XOffset
 Y=YOffset

 ; verify that we will not overrun the total tiles
 If (TileStart + (ShowWidth - 1)) >= TotalTiles
   TileStart = TotalTiles - ShowWidth
   If TileStart < 0
     TileStart = 0
     ShowWidth = TotalTiles
   EndIf
```

```
EndIf

; run through the tiles from the current tile to
; the size we can display
For Columns = TileStart To (ShowWidth+TileStart)-1
    ; draw the tile
    DrawImage Tile(Columns)\Image,X,Y
    ; set our color to white and drow a box around the tile
    Color(255,255,255)
    Rect(X,Y,Tile(Columns)\Width,Tile(Columns)\Height,0)

    ; re-add this tile to the TileSelections TYPE
    TileSelection.TileSelections = New TileSelections
    TileSelection\TileNumber = Columns
    TileSelection\X = X
    TileSelection\Y = Y
    TileSelection\Width = Tile(Columns)\Width
    TileSelection\Height = Tile(Columns)\Height
    TileEndX = X + Tile(Columns)\Width

    ; increment our drawing position
    X = X + Tile(Columns)\Width + 1
Next

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; left/right map indicator display section
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; first determine the total width of the tiles
TotalWidth# = TotalTiles - ShowWidth

; next make sure that there is more than 0 tiles
If TotalWidth - (ShowWidth - 1) <> 0
   ; if so, calculate the distance the cursor should move
   CursorDistance# = (ShowWidth*TileSize)/TotalWidth#
   ; including the distance indicator and offset
   CursorOffset# = (CursorDistance# * TileStart) + 98
   ; if it goes too far, keep it in the box
   If CursorOffset# > 743
      CursorOffset# = 743
   EndIf
Else
   ; just leave it to the left-edge if there are 0 tiles
   CursorOffset# = 98
EndIf
```

```
  ; set the color and draw the indicator
  Color 0,0,255
  Rect CursorOffset#,572,9,9,1
  Color 255,255,255
  Rect CursorOffset#,572,9,9,0

  ; reset the color
  Color(Red,Green,Blue)
End Function
```

That will display the tiles. But notice also that we made sure that the program knows when we've scrolled over so we get the appropriate tile during our selections. That *also* gives the appearance of scrolling the tiles because of the offset, which is passed in by the caller.

It's important to note that I'm resetting the *TileSelections* TYPE every time we go into that function. I do this because it's an easier way for handling the selections and scrolling. But it's not a very efficient way for doing real-time scrolling, so stick with the previous methods discussed for that topic. Feel free to alter this code to use a more speedy method if you'd like, but it seems to me that this should more than suffice for a map creation tool.

Selecting a Tile from the Tile Set

To select a tile, we first see if the mouse is even inside the tile area by using the MOUSEX, MOUSEY commands. After finding this out, we run through the tile list and see if we're inside the current X, Y position of a tile. If so, we set the *CurrentTileSelected* to be *that* tile. Here's the code:

```
If X >= TileStartX And X <= TileEndX And Y >= TileStartY And ↲
                        → Y <= TileStartY + TileSize
   For TileSelection.TileSelections = Each TileSelections
       If X >= TileSelection\X And X <= TileSelection\X +
TileSelection\Width
          CurrentTileSelected = TileSelection\TileNumber
       EndIf
   Next
EndIf
```

Setting Map Tiles

The process for scrolling the map here is nearly identical to the process used for the tile array, with the only difference being that we need two offsets. One offset will be for our columns, like the tile sets, and the other will be for rows.

Also like the tile sets, I am using a TYPE that refreshes on each move as opposed to using the real-time calculation method. Don't confuse this with the array that holds the actual map data. These are just the little square selection spots that we can click on. They hold no permanent data at all. Again, feel free to change this if you see fit to do so. Here's what the TYPE for the map looks like:

```
Type MapSelections
    Field MapXPosition    ; the X offset
    Field MapYPosition    ; the Y offset
    Field X               ; left edge
    Field Y               ; right edge
    Field Width           ; width of this square
    Field Height          ; height of this square
End Type
```

And, similar to the tile sets, the refresh code looks like this:

```
Function
RefreshMapType(MapStartX,MapStartY,MapX,MapY,Columns,Rows)
   ; delete the current TYPE entries
   For MapSelection.MapSelections = Each MapSelections
       Delete MapSelection
   Next

   X = MapStartX
   Y = MapStartY
   MapXValue = MapX
   MapYValue = MapY
   For Row = 0 To Rows -1
        For Column = 0 To Columns -1
            MapSelection.MapSelections = New MapSelections
            MapSelection\MapXPosition = MapXValue
            MapSelection\MapYPosition = MapYValue
            MapSelection\X = X
            MapSelection\Y = Y
            MapSelection\Width = TileSize
            MapSelection\Height = TileSize
            X = X + TileSize
            MapXValue = MapXValue + 1
       Next
       MapEndX = X + TileSize
       X = MapStartX
```

```
        Y = Y + TileSize
        MapXValue = MapX
        MapYValue = MapYValue + 1
  Next
  MapEndY = Y + TileSize
End Function
```

Now since I'm not doing the calculations real-time, but rather just doing offsetting, the tile placement calculation is already handled! So when we do a check on the mouse click, we just assign the value as follows:

```
For MapSelection.MapSelections = Each MapSelections
     If X >= MapSelection\X + 1 And ↲
     → X <= (MapSelection\X - 1) + MapSelection\Width And ↲
     → Y >= MapSelection\Y + 1 And ↲
     → Y <= (MapSelection\Y - 1) + MapSelection\Height
              Map(MapSelection\MapXPosition, ↲
                 → MapSelection\MapYPosition)\ ↲
                 → TileNumber = CurrentTileSelected
    EndIf
Next
```

The X and Y values listed here are passed in as calls to the MOUSEX and MOUSEY functions. Those values are then compared to see if they fall within the map window. If so, they are compared to the X, Y value of the *MapSelections* TYPE. If the X, Y is found to be within one of the *MapSelections* elements, then we can assign the *CurrentTileSelected* value to the *Map* array.

The Map Array

This array is a two-dimensional array that holds integer values. Each value represents a tile. When we loaded in our tiles, these values were assigned. When we selected tiles for drawing, the tile was displayed and the value for that tile was used to update the array. Here is how the array was originally dimensioned:

```
; Type for the MapData
Type MapData
    Field TileNumber
End Type

; Globals to track the map dimensions
Global MapWidth = 1
```

```
Global MapHeight = 1

; Dimension our Map array
Dim Map.MapData(MapWidth,MapHeight)
```

The first item is a TYPE. This is where you could change things to have Z-Ordering elements, animated tile sets, etc. The example only shows a field for the *TileNumber*, but this is easily expandable.

The two global values, *MapWidth* and *MapHeight*, are used throughout the "maplib.bb" library. They keep track of any changes in the width and height of the map and make sure that all the functions are aware of any updates. The value is initialized to 1, 1 in the library, but is quickly altered by the calling program to fit the actual desired dimensions.

Re-dimensioning the Map *array*

As the map size increases, the array is re-dimensioned on the fly. We back up the old array first so we don't lose everything on the map. To do this, we need to have another array to hold this data.

```
Dim MapHold.MapData(MapWidth,MapHeight)
```

This is an exact duplicate of the *Map* array in scope, but to make the content exact requires some processing on our part. The following code demonstrates the entire process of backing up the *Map* array, re-dimensioning it, and restoring the backup to the new positions. This code only demonstrates the increasing of the width. There is also a function to increase the height…and there are functions to decrease both as well.

```
Function Map_IncreaseWidth(Amount)
   ; dimension the temporary array
   Dim MapHold.MapData(MapWidth,MapHeight)

   CurrentWidth = MapWidth
   CurrentHeight = MapHeight

   ; for all of the rows (minus 1)
   For Rows = 0 To CurrentHeight-1
        ; and for all the columns (minus 1)
        For Columns = 0 To CurrentWidth-1
             ; create a new instance of the MapHold array
             MapHold.MapData(Columns,Rows) = New MapData
             ; backup the Map data
```

```
            MapHold(Columns,Rows)\TileNumber = ↲
                  → Map(Columns,Rows)\TileNumber
        Next
    Next

    ;increase MapWidth value by the amount sent by the caller
    MapWidth = MapWidth + Amount
    ; re-dimension the map array
    Dim Map.MapData(MapWidth,MapHeight)

    ; for all of the rows (minus 1)
    For Rows = 0 To CurrentHeight-1
        ; and for all the columns (minus 1)
        For Columns = 0 To CurrentWidth-1
            ; create a new instance of the Map array
            Map.MapData(Columns,Rows) = New MapData
            ; restore the Map data
            Map(Columns,Rows)\TileNumber = ↲
               → MapHold(Columns,Rows)\TileNumber
        Next
    Next

    ; run through the map and find anything that has a NULL value
    ; and set that value to -1

    ; for all of the rows (minus 1)
    For Rows = 0 To MapHeight-1
        ; and for all the columns (minus 1)
        For Columns = 0 To MapWidth-1
            If Map.MapData(Columns,Rows) = Null
               Map.MapData(Columns,Rows) = New MapData
               Map(Columns,Rows)\TileNumber = -1
            EndIf
        Next
    Next
End Function
```

Note the very last section of that code. What this does is sets unused tile pieces to the value of –1. This is the value I selected to represent a space on the map that is blank.

Drawing the Grid

To make a grid appear over the tiles on the map is actually quite simple. All you need to do is set the appropriate color (make sure to save the color

first!), and use the RECT command to draw the grid all over the map at the proper width/height ratio. The code is so small it hardly needs detailed explanation.

```
Function DrawGrid(X,Y,Width,Height,Columns,Rows)
 ; save the colors
 Red = ColorRed()
 Blue = ColorBlue()
 Green = ColorGreen()

 ; set the color to white
 Color(255,255,255)

 ; run through the columns and rows
 For R = 0 To Rows -1
      For C = 0 To Columns -1
           ; draw the square
           Rect(X+(C*Width),Y+(R*Height),Width,Height,0)
      Next
 Next

 ; reset the color
 Color(Red,Green,Blue)
End Function
```

The MapMaker *Code*

As stated earlier, the full code for the *MapMaker* is included on the CD. I recommend that you take a look into the code and make upgrades and additions to this tool as you see fit. I have taken a step from the norm here and placed the *MapMaker* in its own directory, so look for it separate from the other demos in this chapter.

Chapter 22: Homing Objects

A number of games out there have this cool effect of allowing one object to home in on another object.

Take a space game, for example. You have your ship firing a dumb missile at the enemy. If the enemy turns, the missile doesn't pursue. It just goes straight ahead about its business until it's no longer in range. Next you fire a homing missile, after acquiring a target of course, and that missile locks on to the target and chases it. Once it hits the target it connects and blows up.

This sounds like a very simple thing to do, right? You just grab the current coordinates of the missile and the current coordinates of the target, compare the two, and move the missile in the direction that brings it closest to the target. That's close, and will work perfectly fine if your missiles are completely circular with no marks that demonstrate its nose. But if you have a traditionally shaped missile, then you're going to have a problem.

The problem is that you not only have to know which way to move the missile, but you also have to know which way to face the missile. It would look rather dumb to have a missile pointing due north and tracking due west, don't you think?

Making A Thinking Missile

The method I'll be using is to treat the missile as its own entity, having its own thrust and angular information. Next I will make it "smart" enough to know which way it must turn and provide thrust in order to get closer to the target. So, in a nutshell, we're going to use some *very* basic artificial intelligence.

The missile must *think* the following things:

1) Where is my target in relation to me?
2) If I go straight ahead by one unit (based on thrust and direction), how far will I be from my target?
3) If I turn left by one unit, and move forward by one unit, how far will I be from my target?
4) If I turn right by one unit, and move forward by one unit, how far will I be from my target?
5) Which of these three projections (straight, left, right) would bring me closer to my target?
6) I will turn in that direction and move forward by one unit

7) Have I hit the target? If I have then I must complete my mission. If I have not, then I must go back to step 1 and continue movement.

Determining the Distance from Object to Target

The following code can be used to see how far two points are from each other, as is needed in Step 1. You need to provide the object X, Y coordinates and the target X, Y coordinates, and the function will return the proper distance.

```
;*********************************************
; Function: GetDistance#(...)
; Last Upd: 11/15/01
;  Purpose: Finds the distance between two points
;      Args: the X,Y locations of both points
;  Returns: The distance
;*********************************************
Function GetDistance#(XSource#,YSource#,XTarget#,YTarget#)
   ; find the difference between the source and target
   XDist# = XTarget - XSource
   YDist# = YTarget - YSource

   ; use a little math
   TotalDist# = Sqr#((XDist * XDist) + (YDist * YDist))

   ; return the value
   Return (TotalDist)
End Function
```

It's important to explain that the SQR command isn't the speediest thing on the planet. You could pre-compute a look-up table of all the possible distances based on your world to speed things up, but this would take quite a large table to do so. Alternately, you can remove the SQR altogether, thus having the following line for *TotalDist#*:

```
; use a little math
TotalDist# = (XDist * XDist) + (YDist * YDist)
```

This will not provide as accurate a result and you will have to adjust your smoothness (which we'll discuss shortly) accordingly, but it does save on processes.

Turning the Missile

Next we need to go to Steps 2-4 to see which direction would be closest to the target after a single unit movement. The following function will return the

angle that it feels is the best to take in order to most quickly reach your target:

```
;*********************************************
; Function: AngleToTarget(...)
;  Purpose: Calculates the best direction to turn based on
;           distance from the current position to a target
;     Args: X,Y of homing Object
;           X,Y of Target Object
;           Current direction of homing object
;           Number of rotation the homing object can have
;           Smoothness factor to send along to avoid shaking
;  Returns: Best Angle for heading toward target
;*********************************************
Function AngleToTarget(X#, Y#, XTarget#, YTarget#, ↲
                       → Speed#, ShipDir, Rotations, ↲
                       → Smoothness#)

   ; Calculate what the next x,y position of the ship would be
   ; if it kept moving in its current direction
   StraightX# = X# + (xSinTable#(ShipDir) * Speed#)
   StraightY# = Y# + (yCosTable#(ShipDir) * Speed#)

   ; Calculate what the next x,y position of the ship would be
   ; if it turned one unit left and moved forward from there
   LeftShipDir = ShipDir - 1
   If LeftShipDir < 0
      LeftShipDir = Rotations - 1
   EndIf
   LeftX# = X# + (xSinTable#(LeftShipDir) * Speed#)
   LeftY# = Y# + (yCosTable#(LeftShipDir) * Speed#)

   ; Calculate what the next x,y position of the ship would be
   ; if it turned one unit right and moved forward from there
   RightShipDir = ShipDir + 1
   If RightShipDir > Rotations - 1
      RightShipDir = 1
   EndIf
   RightX# = X# + (xSinTable#(RightShipDir) * Speed#)
   RightY# = Y# + (yCosTable#(RightShipDir) * Speed#)

   ; using our above calculated projections, let's see what
   ; the distance is between the target and each projection
   StraightDist# = GetDistance#(StraightX,StraightY, ↲
                                → XTarget,YTarget)
```

```
   LeftDist# = GetDistance#(LeftX,LeftY,XTarget,YTarget)
   RightDist# = GetDistance#(RightX,RightY,XTarget, YTarget)

   ; if the Left distance is less than the Straight and the
   ; Right distances, the the best direction to turn would
   ; be left.  That will bring us closer to the target.
   If LeftDist < StraightDist And LeftDist < RightDist
      ; see if there is enough of an angle to warrant
      ; changing the dir...if this was left out the ship
      ; will shake madly when it gets close to the target
      If StraightDist - LeftDist > Smoothness
        ; change the ship dir accordingly
        ShipDir = LeftShipDir
      EndIf
   EndIf

   ; if the Right distance is less than the Straight and the
   ; Left distances, the the best direction to turn would
   ; be Right.
   If RightDist < StraightDist And RightDist < LeftDist
      ; see if there is enough of an angle to warrant
      ; changing the dir...if this was left out the ship
      ; will shake madly when it gets close to the target
      If StraightDist - RightDist > Smoothness
         ; change the ship dir accordingly
         ShipDir = RightShipDir
      EndIf
   EndIf

   ; return the appropriate direction
   Return(ShipDir)
End Function
```

If you study that code carefully you'll see that it uses a pre-calculated table for Sine/Cosine values, and that it also uses the previous *GetDistance* function to determine which of the new positions would be closest. Based on its findings, this function will return the appropriate direction value.

Smoothness

When the object gets closer to the target it will have an increased chance of alternating between straight and left/right as being closest. What will occur is the object begins quickly alternating between the two angles very fast, thus giving the appearance of shaking. It's not very pleasing to the eye, unless of

course you want to pretend that your missiles get really excited when they're about to blow up! This happens because the calculations of the three angles become tighter and tighter as they close in on the target. Look at the following for an idea of this:

Distance	Straight Calculation	Left Calculation	Right Calculation	Best Dir
1.000	.9999	.9998	1.111	Left
.9998	.9996	.9998	1.000	Straight
.9996	.9995	.9994	.9999	Left
.9994	.9992	.9993	.9998	Straight

…and so on. Those calculations are in no way accurate. They are simply to show a point that without smoothness each frame will alternate between, in this example, straight and left. That makes it look like the object is shaking.

How we can handle this is to see if there is enough difference between the current angle and the best angle to warrant a change. If, for example, we had set the smoothness to *.1* in our table, the ship would have continued turning left. This is because the difference between *.9998* and *.9996* is far less than *.1*. This smoothing variable is configurable and should be adjusted to taste, but keep in mind that the higher the smoothness value the tougher it will be to actually make contact with your target as the turns lose accuracy by that factor of smoothness.

The Demo Code

On the disk there is code that demonstrates this method. It takes a single image that is pointing due north, loads it in and creates an array of rotations for it. It then pre-computes the Sine/Cosine tables to make sure our angles are correct. Finally, it places the image object on the screen and chases the mouse cursor around. You must be in debug mode to see the mouse cursor in this example.

Study that code carefully and you should get the hang of it pretty easily. Please keep in mind that if you decide *not* to use the SQR command, you will need to adjust the smoothness accordingly.

Chapter 23: WayPoint Path-finding

Though there are many path-finding methods that are available, let's use the information gathered from the previous chapter and get into *WayPoint*. A *WayPoint* is an X, Y position that a player or NPC (non-player character) is to go to. Consider it a series of destinations. When each destination is achieved, you move to the next one.

You can also imagine it like a homing missile sees its target. The homing missile leaves the ship, and it heads towards the enemy. The enemy is a destination (*WayPoint*). Now, imagine that you have some really cool missile that releases tiny missiles when it gets close to its intended victim. After the missile does that, it heads toward the next enemy to do the same. Each enemy, then, would be a *WayPoint*.

Setting Things Up

In our *Homing Missiles* demo, we talked about how we can adjust the position of an object and then thrust it forward one unit until it got to the target. Now we need to:

1) Put together a list of targets
2) Use the homing algorithm to get to the first target
3) When the target is reached, do whatever processing you want
4) Set the next target as active and go back to step 2

That's it! It's really pretty simple.

Creating Way Points

You can handle this a number of ways, but I found the easiest is to create a little program that allows you to click the mouse in a specific location. Then that location is saved into the elements of a TYPE. Then I save those elements to a file and re-load them at will. You'll clearly want something more robust than the little example in this chapter, but it'll show you how to put it together.

Here is the TYPE definition:

```
Type WayPoints
    Field SET          ; What SET of Waypoints is this for?
    Field Current      ; 0=idle, 1 =Current Waypoint
    Field X#           ; What's the X for this Waypoint?
    Field Y#           ; What's the Y for this Waypoint?
End Type
```

Since we may have different objects going along differing paths, we'll want to denote different sets of *WayPoints*. The *SET* field is for this purpose. This field can contain any number definition for the set that you're creating. To use this set later, refer to that number and it will only grab data for that set.

We use the *Current* field to say whether or not the *WayPoint* is what we're after. You want to ensure that there are not two *WayPoints* set as current…the code will not like that. You could of course write an algorithm that determines which *WayPoint* is closest of the two and act on that one if you wish.

The following is the code for adding a *WayPoint*:

```
;*********************************************
; Function: AddWayPoint(...)
; Purpose: Adds another node to the waypoints lists
; Args: the X,Y location of point, and the SET # of its SET
;  Returns: none
; Comments: none
;*********************************************
Function AddWayPoint(X#,Y#,SET)
   WayPoint.WayPoints = New WayPoints
   If WayPoint.WayPoints <> Null
      WayPoint\Current = 0
      WayPoint\X# = X#
      WayPoint\Y# = Y#
      WayPoint\SET = SET
   EndIf
End Function
```

Moving from WayPoint to WayPoint

Next we want to be able to go to the next *WayPoint* in the list. This requires that we set the *Current* field in the current *WayPoint* to 0, and set the *Current* field in the next *WayPoint* to 1. Essentially, we want the next *WayPoint* to become the current *WayPoint*.

Here's the code for that:

```
;*********************************************
; Function: GotoNextWayPoint()
;  Purpose: Setup the next waypoint to be active
;  Args: SET of the current WayPoint set
;  Returns: none
```

```
; Comments: none
;*********************************************
Function GotoNextWayPoint(SET)

   IsCurrent = 0
   ; run through the list of waypoints and set the
   ; current waypoint to 0, and then find the next
   ; waypoint and set it to the current one.
   For WayPoint.WayPoints = Each WayPoints
       ; dSET we find the current one?
       If WayPoint\SET = SET And WayPoint\Current = 1
          WayPoint\Current = 0
          ; this flag will allow us to know when we are past
          ; the current one, but have hit the next one
          IsCurrent = 1
       Else
          If WayPoint\SET = SET And IsCurrent = 1
             WayPoint\Current = 1
             IsCurrent = 0
             Exit  ; leave the loop
          EndIf
       EndIf
   Next

   ; if we shut off the current one but dSET not find
   ; another one, then reset the set.
   If IsCurrent = 1
      ResetWayPointPosition(SET)
   EndIf
End Function
```

The next thing we do is notify our *WayPoint-following-object* where the X, Y position of the next *WayPoint* is. While you could have one function that resets the positions on global X, Y variables, I elected to instead split the functions up. I did this for cleaner coding practice, but you can certainly incorporate these into one function as you see fit. Also, note that incorporating them into one function may give a little speed boost as well because you'll only have one function to call instead of two.

```
;*********************************************
; Function: GetNextWayPointX#()
;  Purpose: Determine the X coord for the next waypoint
;     Args: SET of the current WayPoint set
;  Returns: X value of the next waypoint
```

```
; Comments: none
;*********************************************
Function GetNextWayPointX#(SET)
   For WayPoint.WayPoints = Each WayPoints
        If WayPoint\SET = SET And WayPoint\Current = 1
           Return(WayPoint\X)
        EndIf
   Next
   Return(0)
End Function

;*********************************************
; Function: GetNextWayPointY#()
;  Purpose: Determine the Y coord for the next waypoint
;     Args: SET of the current WayPoint set
;  Returns: Y value of the next waypoint
; Comments: none
;*********************************************
Function GetNextWayPointY#(SET)
   For WayPoint.WayPoints = Each WayPoints
        If WayPoint\SET = SET And WayPoint\Current = 1
           Return(WayPoint\Y)
        EndIf
   Next
   Return(0)
End Function
```

Saving and Loading WayPoints

To save and load the *WayPoints* is a snap. Run through the list and write out the values to the file for saving. For loading, you first clear the list, and then read in the values and add them in to the new list.

```
;*********************************************
; Function: SaveWayPoints()
;  Purpose: Setup the next waypoint to be active
;     Args: SET of the current WayPoint set
;  Returns: none
; Comments: none
;*********************************************
Function SaveWayPoints(Filename$,SET)
   WayPoint.WayPoints = First WayPoints
   ; make sure there are WayPoints at all
   If WayPoint.WayPoints <> Null
      ; open the file
```

```
      OutputFile = WriteFile (Filename$)
      ; run through and write out the X,Y coords
      ; for each WayPoint in the current set.
      For WayPoint.WayPoints = Each WayPoints
           If WayPoint\SET = SET
              WriteInt(OutputFile,WayPoint\X)
              WriteInt(OutputFile,WayPoint\Y)
           EndIf
      Next
      ; close the file
      CloseFile(OutputFile)
   EndIf
End Function

;*********************************************
; Function: LoadWayPoints()
;  Purpose: Setup the next waypoint to be active
;     Args: none
;  Returns: none
; Comments: none
;*********************************************
Function LoadWayPoints(Filename$, SET)
   ; first remove all the WayPoints in this set
   ; so we don't have more than we should
   For WayPoint.WayPoints = Each WayPoints
        If WayPoint\SET = SET
           Delete WayPoint
        EndIf
   Next

   ; now open the file and read in the new set
   InputFile = ReadFile (Filename$)
   While Not Eof(InputFile)
          WayPoint.WayPoints = New WayPoints
          WayPoint\SET = SET
          WayPoint\Current = 0
          WayPoint\X = ReadInt(InputFile)
          WayPoint\Y = ReadInt(InputFile)
   Wend
   CloseFile(InputFile)

   ; set it up to have the Current as the first item in the set
   ResetWayPointPosition(SET)
End Function
```

Showing WayPoints

Another thing I decided to do was to have the ability to show the *WayPoints*. I did this because I wanted to know where the little object was heading. This is not necessary, of course, but it can be useful to see why an object may not be going in proper directions.

```
;***********************************************
; Function: ShowWayPoints(...)
;  Purpose: Display a bunch of boxes to represent the waypoints
;     Args: SET of the set to display
;  Returns: none
; Comments: none
;***********************************************
Function ShowWayPoints(SET)
   ; save the current color
   Red = ColorRed ()
   Green = ColorGreen ()
   Blue = ColorBlue ()

   ; run through the list and see if there is
   ; a current WayPoint
   IsCurrent = 0
   For WayPoint.WayPoints = Each WayPoints
        If WayPoint\SET = SET
           If WayPoint\Current = 1
              IsCurrent = 1
              Exit ; exit the loop
           EndIf
        EndIf
   Next

   ; if not, reset it
   If IsCurrent = 0
      ResetWayPointPosition(SET)
   EndIf

   ; now run through each one and draw a box
   ; to denote the location of the point.  Only
   ; draw the points in the set SET
   For WayPoint.WayPoints = Each WayPoints
        If WayPoint\SET = SET
            If WayPoint\Current = 1
              Color(255,0,0)
           Else
              Color(125,125,125)
```

```
            EndIf
            Rect Int(WayPoint\X)-5,Int(WayPoint\Y)-5,10,10
         EndIf
   Next

   ; set the color back to what it was
   Color(Red,Green,Blue)
End Function
```

Now all that does is draw tiny little boxes wherever a *WayPoint* was placed. It draws the current *WayPoint* in red, so you know where the object is headed, and the rest are drawn in light gray.

Resetting the Waypoint Position

It's important to reset the *Current* Field for each *SET* before using it because you may end up with either no current Waypoint or a lost *Waypoint*. To do this, use the following code:

```
;*********************************************
; Function: ResetWayPointPostion(...)
;  Purpose: Setup the next waypoint to be active
;     Args: SET of the set to initialize at
;  Returns: none
; Comments: none
;*********************************************
Function ResetWayPointPosition(SET)

   ; first make all the "Current" set values = 0
   For WayPoint.WayPoints = Each WayPoints
         If WayPoint\SET = SET And WayPoint\Current = 1
            WayPoint\Current = 0
         EndIf
   Next

   ; then go back to the top of the list
   WayPoint.WayPoints = First WayPoints

   ; and grab the first of the SET and make it current
   For WayPoint.WayPoints = Each WayPoints
         If WayPoint\SET = SET
            WayPoint\Current = 1
            Exit ;exit the loop
         EndIf
   Next
```

```
End Function
```

Where to go from here

If you plan to utilize *WayPoints* in your game, I would recommend that you create a way to add/delete/alter them visually. As you would likely create a map-making program to make your scrolling maps with, you should also consider doing the same (or adding this functionality into your map-maker) with *WayPoints*.

Chapter 24: Particles and Explosions

What fun would a game be without cool special effects? No particles, no explosions! We can't have that…so let's get into it.

Particle Effects

Lots of games use these effects to improve the quality of explosions, or magical spells, or engine exhaust, etc. There are super fast ways of handling these, and horrifically slow ways. The way presented here will be one that's quick enough for most uses but is more focused on helping you understand the concept.

What is a particle? A particle is an image that has a defined beginning and location, defined velocity and direction, and a defined lifespan. Typically particles come in various sizes and have varied speeds and colors. They move along their little courses, slowly fading from brightened colors to more and more dimmed ones until they finally expire.

Many people use individual pixels for this effect, but pixel drawing is really slow so that's not recommended. Some use images that they pre-render at each phase on the lifeline to have exact dimming and so on. I'm going to use the Blitz RECT command.

Particle Setup

The first thing we want to do is setup a TYPE that will store each of our particles.

```
Type Particles
   Field dX#,dY#              ; screen x,y coords
   Field dSpeedX#,dSpeedY#    ; speed of each particle
   Field Red,Green,Blue       ; the rgb color for each
   Field StartTime#           ; millisec that it began
   Field FadeSpeed#           ; how often to fade
   Field LifeSpan             ; how long it will live
End Type
```

So you can see how we can bring to "life" a particle, and we can control a bunch of information on it. You can certainly add more to this than just that information of course, but this is enough to provide an example.

Launching Particles

In order to launch a particle, we just need to populate the *Particles* TYPE. You can control in detail how to do this, but I've written a little function for demonstration purposes:

```
Function LaunchParticles(X#,Y#,Amount,Dir)
   ; run through the amount of particles requested
   For i = 0 To Amount - 1
      ; create a new instance of the particle
      Particle.Particles = New Particles
      Particle\dX = X
      Particle\dY = Y

      ; use the Sin/Cos functions multiplied by a random
      ; value to set the speed of the particle
      Particle\dSpeedX# = -Sin(Dir) * Rnd#(.5,1.5)
      Particle\dSpeedY# = Cos(Dir) * Rnd#(.5,1.5)

      ; randomize the color of this particle
      Particle\Red = Rand(180,255)
      Particle\Green = Rand(180,255)
      Particle\Blue = Rand(180,255)

      ; set the particle's start
      Particle\StartTime = MilliSecs()
      ; determine a fade speed.
      Particle\FadeSpeed = Rnd#(.00001,1)

      ; determine a lifespan for this particle
      Particle\LifeSpan = Rand(1500,5000)
   Next
End Function
```

There are a number of calls to the RND and RAND commands in that function. This is for demo purposes only. You should do your best to avoid these calls where you can because they do eat up speed. However, you must keep in mind that in order to have a particle appear to have its own "life" some randomness is necessary.

Updating Particles

After a particle has been launched, we have to follow it through its full life span and update it accordingly. It will dim over time and eventually disappear. We need to track this so we know when it has finally dimmed completely. Then we delete it to avoid taking up unnecessary memory.

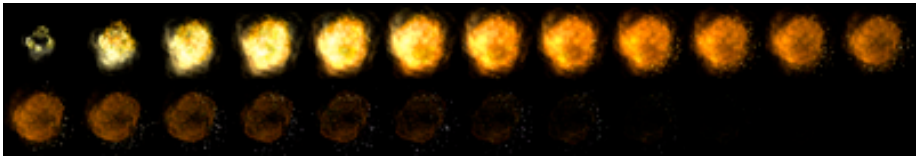
```
Function UpdateParticles()

   ; get the current time in milliseconds
   CurrentT = MilliSecs()

   ; run through all of the living particles
   For Particle.Particles = Each Particles
         CurrentColor = 0
         ; if the current time is greater than the time this
         ; particle started and the speed we've set to fade it
         If CurrentT > Particle\StartTime + Particle\FadeSpeed
            ; reset the start time for this particle
            Particle\StartTime = MilliSecs()

            ; start dimming the colors...change the decrease
            ; amount to further control dimming speed
            Particle\Red = Particle\Red - 1
            If Particle\Red <= 0
               Particle\Red = 0
            EndIf

            Particle\Green = Particle\Green - 1
            If Particle\Green <= 0
               Particle\Green = 0
            EndIf

            Particle\Blue = Particle\Blue - 1
            If Particle\Blue <= 0
               Particle\Blue = 0
            EndIf
          EndIf

          ; update particle's X,Y location based on speed info
          Particle\dX# = Particle\dX# - Particle\dSpeedX#
          Particle\dY# = Particle\dY# - Particle\dSpeedY#

          ; if particle has lived out its life, delete it
         If CurrentLifeTime > Particle\StartTime + Particle\LifeSpan
            Delete Particle
         Else
            ; if the particle's colors are all 0 and it's still alive,
            ; just kill it cause it's wasting processor time for nothing.
            ; you don't have to go all the way to 0 either...go only to
            ; the point you can't see it.
```

```
            If Particle\Red = 0 And Particle\Green = 0 And ↲
                                    → Particle\Blue=0
               Delete Particle
            Else
               ; set the color of the particle and draw it out
               Color Particle\Red,Particle\Green,Particle\Blue
               Rect Particle\dX,Particle\dY,3,3
            EndIf
         EndIf
      Next
End Function
```

Explosions

There are a number of ways to handle explosions. You could use animated pre-rendered images, or particle effects, or you could use pre-rendered images *and* particle effects, and I'm sure there are a number of other ways.

I'm going to first just show how to use basic images. The following explosion graphic sequence was pre-rendered:



(Figure 24.1)

Notice how the graphic starts out condensed and bright and slowly loosens up. Drawn in succession, and with controlled timing, this is a very effective-looking explosion.

As with particles, we need to setup a TYPE for our explosions so we can track many of them. Here's the TYPE layout I'll be using:

```
; Explosion type
Type Explosions
   Field dX#,dY#      ;screen x,y coords of the explosion
   Field Frame        ;current frame in the animation
   Field TotalFrames  ;total frames of animation
   Field StartTime    ;starting time of the frame
   Field Speed        ;speed of animation per frame
End Type
```

And since we have to populate that TYPE, I've created a function similar to the *LaunchParticles* function.

```
Function LaunchExplosion(dX#, dY#)
      ; create a new Explosion instance
      Explosion.Explosions = New Explosions

      ; Fill in the type values
      Explosion\dX = dX
      Explosion\dY = dY
      Explosion\Frame = 0
      Explosion\TotalFrames = 23
      Explosion\StartTime = MilliSecs()
      Explosion\Speed = 50
End Function
```

When that is called you can see that we'll have an image that has 24 frames (0 to 23) and it will change frames every 30 milliseconds. And, of course, it will be drawn at whatever X, Y coordinate we sent along.

Now we just need to have a way to update the explosion accordingly, as we did with *UpdateParticles*.

```
Function UpdateExplosions()
 CurrentT = MilliSecs()
 ; go through all of the explosions
 For Explosion.Explosions = Each Explosions
      ; if it's the proper animation time, change frames
      If CurrentT > Explosion\StartTime + Explosion\Speed
         Explosion\StartTime=MilliSecs()
         Explosion\Frame = Explosion\Frame + 1
         ; if we're passed the total frames, delete the instance
         If Explosion\Frame >= Explosion\TotalFrames
            Delete Explosion
         Else
             DrawImage Explosion_Image,Explosion\dX, ↲
                       → Explosion\dY,Explosion\Frame
         EndIf
      Else
         DrawImage Explosion_Image,Explosion\dX, ↲
                   →  Explosion\dY,Explosion\Frame
      EndIf
 Next
```

```
End Function
```

So as soon as a frame time has expired, that function will change frames and draw the new frame. It will draw the frame each FLIP regardless, so we don't have flicker, but it will only *change* frames at the proper time.

Explosions and Particles

Mixing the two is incredibly easy. But we will need to tweak the particle code a little bit to get the proper effect.

The particle TYPE will remain as is, but the *LaunchParticles* function will need to be updated so the particles can go in all different directions. Consider the following code:

```
Function LaunchParticles(X#,Y#,Amount)
   ; run through the amount of particles requested
   For i = 0 To Amount - 1
        ; create a new instance of the particle
        Particle.Particles = New Particles
        Particle\dX = X
        Particle\dY = Y
        Dir = Rand(0,359)
        ; use the Sin/Cos functions multiplied by a random
        ; value to set the speed of the particle
        Particle\dSpeedX# = -Sin(Dir) * Rnd#(.2,2)
        Particle\dSpeedY# = Cos(Dir) * Rnd#(.2,2)

        ; randomize the color of this particle
        Particle\Red = Rand(218,255)
        Particle\Green = Rand(167,253)
        Particle\Blue = Rand(9,197)

        ; set the particle's start
         Particle\StartTime = MilliSecs()
         ; determine a fade speed.
         Particle\FadeSpeed = Rnd#(.001,.5)

         ; determine a lifespan for this particle
         Particle\LifeSpan = Rand(500,1000)
   Next
End Function
```

Pretty much all of the TYPE settings changed, but that's because I was looking for a particular effect. This is what you'll likely need to do for each of your cases as well. Also note that I no longer use the direction argument!

I've made it so the direction goes in a full circle. I've increased the particle speed and fade speed, decreased the life span, and picked colors that were closer to the explosion graphic.

The *LaunchExplosion* function has one new line added in as well:

```
Function LaunchExplosion(dX#, dY#)
        ; create a new Explosion instance
        Explosion.Explosions = New Explosions
        ; Fill in the type values
        Explosion\dX = dX
        Explosion\dY = dY
        Explosion\Frame = 0
        Explosion\TotalFrames = 23
        Explosion\StartTime = MilliSecs()
        Explosion\Speed = 50

        ; launch some particles for added effect
        LaunchParticles(dX,dY,250)
End Function
```

Now I'm launching out 250 random particles each time an explosion is triggered. They all go in different speeds and directions, and have different colors that were selected from the explosion graphic.

Finally, when displaying these in our main loop, I thought it looked better to draw the particles OVER the explosions.

```
; go through and update the explosions
 UpdateExplosions()

 ; then update particles so they are on top of the explosions
 UpdateParticles()
```

But if you like it the other way, just move the *UpdateParticles* above *UpdateExplosions* and you'll be all set.

Images as Particles

The final thing I'd like to touch on is the use of images as particles. This gives you a little more control over the actual particles being displayed and what their color/life-expectancy will be.

I'm going to put the onus on you to figure this one out in coding, but it's really a snap. All you need to do is draw up an image of n-frames (say 4 frames). The first image is white, for example, the second is medium-gray, the third dark gray, and the final very dark gray. Now every time you go to dim the particle, you really are just changing the frame you use to draw it!

Chapter 25: Multi-Player Programming

One of the hottest topics in game development today is multi-player game programming. Unfortunately, it's also considered one of the more complicated. This is because there are so many variables the developer has to take into account when coding network games.

Terminology

When doing this type of development, there are some terms you'll need to understand. Otherwise this will all look very odd. There are many terms that can be added in here, but I'm only going to add in the ones that I'll be using throughout this chapter.

Lag: Also known as "latency," this term defines how long it takes for a piece of information to get from one machine to the next. If you've ever played an online game where items seem to jump from one point to another all of the sudden, you've experienced the effects of lag.

Packets: Think of these as little envelopes. They carry the necessary data from point A to point B. They come in all shapes and sizes, but the best packets are as small as possible while being packed with information.

Ping: *Packet Internet Groper.* This is the name given for when a one computer checks to see if and how well another computer is connected to the Internet. Often this term is used in conjunction with *Ping Time*.

Ping Time: The time from sending a packet to someone and receiving an acknowledgement.

IP Address: This is the computer's identifier on the Internet.

DirectPlay: This is the Microsoft version of network gaming code you can use to make games. It's probably one of the easiest to implement, but it lacks the speed and depth of other methods.

TCP: *Transmission Control Protocol*. This is a commonly used protocol for *Lock-Step*, or *Synchronous* (see below) games because it guarantees that each packet sent will be delivered. The problem with this protocol is that it has a lot of overhead. The IP part of the wording stands for *Internet Protocol* and is basically the header information that will be part of every packet (including the UDP packets shown next).

UDP: *User Datagram Protocol*. This is probably the most used protocol for real-time, speed intensive applications, because it allows the developer to design in the necessary control processes independently. It's fast too. But the problem is that this protocol does not guarantee delivery of packets, so the onus falls on the developer to handle this.

BlitzPlay: This is the network game created completely in Blitz Basic and uses only UDP packets to communicate.

Lock-Step/Synchronous Games: This type of game requires that all players have exactly the same information before the game can continue playing. So if you make a move, all other 7 players have to know about that move before anything else can happen. On an internal network, this isn't such a bad thing and it usually runs pretty well. On the Internet, however, this can be a terrible experience for your players.

Asynchronous Games: This method allows players to act independently, regardless of someone else's packet or lag problems. This can be very tricky though because players may start to see effects of this method, such as *Warping*. Also, you'll need to figure in things such as *Decisions & Determinations*, and *Dead Reckoning*.

Warping: When a player has lag, the messages may take longer than they should to broadcast. If this happens, you may receive a bunch of messages all at once instead of semi-evenly spaced. Depending on how the game is developed, this means that you may all of a sudden see the player blink from one position to the next, or you may see the player animate really quickly from one position to the next. Thus, it's as if the player has "warped" or "jumped to light speed."

Decisions & Determinations: Okay, I made that term up. Basically this is where you have to decide what to do when a player warps. Here's the point. Player A is lagging badly. Player B shoots at Player A and on his screen Player B sees a hit. He's happy. In Player A's world, however, he sees that Player B is moving erratically and not firing at all. All of the sudden, Player A sees about 10 bullets fly out of Player B, and Player B's ship zooms by. Player B sees Player A's ship warp up half the screen and wonders if Player A had been damaged after all. Here's where you, the developer, have to make some determinations. Was there a hit or not? How do you reconcile that with the players?

Dead Reckoning: Is a method by which each computer knows specific information about each entity in the game. Based on this information, each computer can update the positional information of any entity regardless of packet receipt. This cuts down on the amount of network traffic (or amount

of information passed between the players), and it's always good to keep this traffic low where possible. There should be a minimum packet transfer of once every 5 seconds, with quicker updates upon entity change specifics.

Think of an enemy ship that is moving up your screen at 2 pixels per page-flip. If the remote player doesn't change the speed or turn, you can surmise where that ship will be in the next two seconds, right? This is the concept of Dead Reckoning. It's most effective when applied to entities that stay their course, such as non-homing projectiles, but it can be used for nearly any entity type with decent success.

BlitzPlay

Before going any further, I need to point out that all of the examples in this chapter will be done using the BlitzPlay system by John Arnold (also know as SurreaL). It's clean, fast, easy to use, and the support is top-notch!

There are currently two versions of BlitzPlay: BP-Lite and BP-Pro. The BP-Lite version is included on the CD and the examples in this book use it exclusively. The BP-Pro version offers a number of enhancements over the BP-Lite version that are definitely worth looking into, such as:

- Guaranteed packet options
- Connection Tracking
- Constant GameTime tracking (to keep all the connections on the same timer)
- Cubic Splines. Probably the best looking dead-reckoning algorithm I've seen.
- …and more

I know this looks like a major plug for Blitzplay, and it is! I wouldn't recommend it, though, unless I believed in the capabilities of this product. It's been a joy to use and it has saved me countless hours. You can check out the BlitzPlay website at "http://www.blitzcoder.com/blitzplay".

Configuring Packets

One of the most important things you'll need to do is determine how to setup each packet you're going to send to the host. The object is to make the packets as small as possible with each send, and only sending when you need to update others.

Let's say that we have a ship in space moving along. We certainly would like to know that ship's X, Y coordinates, but we also need to know which direction the ship is facing so it'll face the same way on everyone's screen. Let's say that the X, Y values are of type float and the ship's direction is of type integer, and their values are: X=5000.000000, Y=5000.000000, and Dir=27.

Now, we *could* send these across individually but that would be three sends every time we have a positional update. That's not too efficient. A better way would be to combine the three elements as one packet. But we have a couple of problems. The first is that in order to combine these elements in a fashion that we'll be able to distinguish them on arrival, we'll need to convert them to strings.

You may be thinking about doing this:

```
StringX$ = X#       ; convert X float coordinate to a string
StringY$ = Y#       ; convert Y float coordinate to a string
StringDir$ = Dir%   ; convert Dir integer to a string
```

And you well could do that! But, again, the efficiency is not that great because of the following:

StringX$* and *StringY$ will both now contain 11 bytes of data. The size of "5000.000000" is 11 bytes. The integer will be 2 bytes here because it's on "27," but if you had an integer value of "1000," it would be 4 bytes. Thus, using this method on our test data will make our full packet size for positional updates 24 bytes (not including UDP overhead and BlitzPlay header information). That's not too bad, but we can do better.

BlitzPlay comes with a few commands that will convert our floats and integers into compacted strings on one end, and then allow us to convert them back on the other end. The commands and their usage are:

StringValue$ **= *BP_IntToStr(****IntVariable,ResultantSize*)
*StringValue****$*** **= *BP_FloatToStr(****FloatVariable*)
IntValue% **= *BP_StrToInt(****String$*)
FloatValue# **= *BP_StrToFloat(****String$*)

These routines will compress our integers and floats down pretty well for us. A float becomes a total of 4 bytes long (regardless of the actual size of the float), and an integer will be between 1 and 4 bytes, depending on the user's call. Negative integers require 4 bytes (or a workaround) because of the sign requirements. 1 byte can be used for 0-255, 2 bytes for 0-65,535, 3 bytes for 0-16,777,215, and 4 bytes for 0-4,294,967,295 (or negative numbers).

So, if we use these routines, our 24 bytes can drop down to 10 bytes! Now you may be thinking that's not a big deal, but when you consider that you're sending about 10 packets per second with positional updates, then you're saving 140 bytes per second…and that's a pretty substantial savings.

Host vs. Client

Whenever you setup a networked game you need to figure out how you want the messages routed, who's going to control the flow of information, and so on. Using BP, you'll only need to worry whether a player is a host or a client.

If the player is a client, then his machine sends its information to the host and gets updates from the host about other players. If the player is the host, on the other hand, then the machine will need to keep track of all the attached clients and broadcast to all of the clients where everyone is in the game, and what everyone is doing.
To setup a machine as a host in BP, you would call the following:

*Success = **BP_HostSession(**HostName$, MaxPlayers%, GameType%, MyPort%, TimeoutPeriod%**)***

The *HostName$* is whatever you want to call it. Could be anything from "Garntwab" to "BP_Host." It's completely up to you. Just note that whatever computer is the host, this name will be what's used for that player in the game!

The *MaxPlayers%* variable allows you to control how many players will be allowed in before you start blocking people.

The *GameType%* variable is also user defined. The purpose of this variable is whatever you want it to be. In other words, maybe your game says a *GameType* of 0 is "Death Match" and a *GameType* of 1 is "Cooperative."

The *MyPort%* value is the communications port that you will be using on the machine. This can be any number between 0-65535, but some ISP's don't support the use of some of the higher numbers, and some are reserved. I usually stick between 1001 and 4000.

The *TimeoutPeriod%* is how long you, the host, are willing to wait (in seconds) for users to send across their data. If the user goes beyond this timeout period, the user will be dropped from the game. BlitzPlay will send out "keep alive" packets to test connections, so you don't have to worry about that.

The *Success* return is either a TRUE or FALSE value depending on whether or not *BP_HostSession* was successful. Don't ignore this value as the game depends on a TRUE to continue!

Now that we have our session's host up, we need a way to join the session. The following BP command handles that for us:

Success = ***BP_JoinSession(****ClientName$, MyPort%, HostIP%, HostPort%****)***

The *ClientName$* value is what you'll be known as to the host. So, if your alias is "Gleep" then you'll want that as your client name.

The *MyPort%* value is the communications port that you will be using on the machine. Again, I stick with numbers between 1001 and 4000.

The *HostIP%* value is the IP Address of the host machine. This value needs to be in integer format, so you'll want to convert the IP accordingly. You can use the *BP_ConvertIP* command to do this using *BP_ConvertIP("127.0.0.1")*, for example.

The *HostPort%* value is the port on the host's machine that the host is using.

The *Success* return will contain one of five results:
- **BP_NOREPLY** = No reply from the BlitzPlay Host
- **BP_IAMBANNED** = This IP address has been banned
- **BP_GAMEISFULL** = Player maximum has been reached. Will block connection.
- **BP_PORTNOTAVAILABLE** = Local port is not available
- **BP_SUCCESS** = Success! Player is in.

Again, pay close attention to this value, as it will tell you if you're in the game or not!

Sending Packets

Clearly you're going to need a way to send information back and forth to the clients and host. The command in BP for this is:

BP_UPDMessage(*Target%,Type%,Data$****)***

The *Target%* value is the network ID that you are sending to. I'll show you how to get this ID shortly.

The *Type%* value refers to what kind of packet this is.

The *Data$* value refers to the actual data in the packet (remember the string that we created from X, Y, Dir above).

Finding the Target machine

As a client, the target is going to be for "Broadcasting" in most all of your sends. Setting *Target%* to 0 (zero) will make sends workout just fine. Now

you can send to individual players, just replace that Target% with the network ID of the player.

Packet Types
Each packet sent will have to describe its purpose. This is done by sending along an integer value between 0 and 255 with each message sent. Be warned though that numbers 230-255 are reserved for BP internal messages! The following are the pre-defined BP types:

- **255** – Acknowledgment (ACK) reply
- **254** – Connection info for a new player
- **253** – User has left the game
- **252** – New player has successfully connected
- **251** – The Host has disconnected
- **250** – An "Are you still there?" packet (also called a heartbeat)
- **249** – Someone got kicked out of the game
- **230-248** – Reserved for future use by BP

This means that you can have the 0-229 packet types (which is a tremendous amount of types, I assure you!) to do what you want with. But make sure you design these out so you don't get confused.

Here are examples that are used in the network space game demo on the CD:

- **1** – Positional update. Contains X, Y, ShipDir, and Thruster values.
- **2** – Speed/Shield update. Contains current speed and shield status.
- **3** – Bullet Fired. Contains bullet type, direction, and player ID it will hit.
- **255** – Piggy-backed on BP's 255 to add a player to the NetPlayers list.
- **254** – Piggy-backed on BP's 254 so I can remove a player that's been disconnected.

Now your client will know what type to send over to the host whenever an update occurs, and only the host will know what the message means and what to with it upon arrival.

Making a Connection
So, taking what we know, let's create a simple program that will connect and send information back and forth. You don't need separate machines to do this, so don't worry. Simply use the BP variable *BP_LocalHost* in the *BP_JoinSession* command as the IP address. Then run the Blitz Basic IDE twice and launch one copy as the host and the other as the client.

Here is the code. It is a bit involved, but that's only because it's showing you a lot of functionality that you can control here. You will need to have

BPLite.BB in the same directory as this code for this to work. This code comes right out of the BPLite demo.

```
; bring in the BlitzPlay library
Include "bplite.bb"

; setup random number generations
SeedRnd MilliSecs()

; set a flag for client/host
Global client

; create a type for holding the text
; passed back and forth
Type Info
        Field txt$
End Type

; initialize the graphics
Graphics 600,400,16,2

; ask if the player is to be the host or the client
Cls
Text 0,0,"Would you like to:"
Text 20,FontHeight()*2,"[H]ost"
Text 10,FontHeight()*3,"or"
Text 20,FontHeight()*4,"[J]oin"
; clear the key buffer
FlushKeys()

; let them answer by just ONE keypress
Repeat
 If KeyHit(35) Then
        client = Fasle
        Exit
 End If
 If KeyHit(36) Then
        client = True
        Exit
 End If
Forever

; clear the key buffer
FlushKeys()
```

```
; if this is a client
If client Then
  ; join the session at BP_LocalHost, using port 1001
  reason = BP_JoinSession ("client", Rand(1002,2000), ↲
                              → BP_LocalHost,1001)
  ; if the connection was successful, get all the information
  ; from the host about the game. And send a couple of
  ; messages too...just for demonstration.
  If reason = BP_SUCCESS Then
     Print "Connected."
     Print "Sending messages.."
     BP_UDPMessage (BP_Host_ID, 1, "[ target = BP_Host")
     Print "[msg sent]"
     BP_UDPMessage (0, 2, "[ target = broadcast")
     Print "[msg sent]"
     Print
     Print "=-Game Info-="
     Print "Game Type: " + BP_GameType
     Print "BP_My_ID: " + BP_My_ID
     Print "BP_Host_ID: " + BP_Host_ID
     Print "BP_NumPlayers: " + BP_NumPlayers
     Print "BP_MaxPlayers: " + BP_MaxPlayers
     Print
     Print "Press a key to exit."
     Text 0,300,"Current Players in game = " + ↲
                 → BP_GetNumberOfPlayers()
     Text 0,320,"Current Game Type = " + BP_GetGameType()

     Text 230,300,"Packets Sent = " + BP_GetPacketsSent()
     Text 230,320,"Max Players = " + BP_GetMaxPlayers()
     Text 230,340,"Host ID = " + BP_GetHostID()
     Text 230,360,"Host IP = " + BP_GetHostIP()
     Text 230,380,"Host Port = " + BP_GetHostPort()

     Text 450,300,"Packets Recvd = " + ↲
                    → BP_GetPacketsReceived()
     Text 450,320,"My ID = " + BP_GetMyID()
     Text 450,340,"My IP = " + BP_GetMyIP()
     Text 450,360,"My Port = " + BP_GetMyPort()
     Text 450,380,"My Name = " + BP_GetMyName$()
      ; wait for a keypress
     WaitKey
  Else
      ; otherwise tell the user that the session didn't work
      Print "Session not joined. Reason code #" + reason
```

```
    WaitKey
  End If

; the user wants to host
Else
   ; initialize the host
   BP_HostSession ("BP_Host",10,1,1001,10)

    ; setup for limiting the frame rate
    tmr = CreateTimer(70)

    ; create an information record
    i.Info = New Info
    i\txt = "Session opened."

    ; NOW set the back buffer
    SetBuffer BackBuffer()

    While Not KeyHit(1)
          ;Framelimiting
           WaitTimer(tmr)

          ;Checking network messages
            BP_UpdateNetwork()

          ; process those messages
            For msg.MsgInfo = Each MsgInfo
                ; add the message to the display list
                 i.Info = New Info
                Select msg\msgType
                        Case 255
                              nInfo.NetInfo = ↲
                                → BP_FindID(msg\msgFrom)
                              i\txt = "ID #" + msg\msgFrom + ↲
                                  → " has joined the game"
                          Case 254
                                  i\txt = "ID #" + ↲
                                  → msg\msgFrom + ↲
                                  → " has left the game"
                          Default
                                  i\txt = "[Type : " + ↲
                                    → msg\msgType + ↲
                                    → " ] " + "[From : " + ↲
                                    → BP_GetPlayerName ↲
                                    → (msg\msgFrom) + ↲
```

```
                                    → " ] " + msg\msgData

            End Select
            ; then delete the message
            Delete msg
    Next

; since you're the host you can change the options
; on the fly too!

; sets the game type
If KeyHit(2)  ; key 1
    BP_SetGameType(1)
EndIf
If KeyHit(3)  ; key 2
   BP_SetGameType(2)
 EndIf

 ; sets the debug packets
 If KeyHit(4)  ; key 3
    BP_SimulatePacketLoss(0)
 EndIf
 If KeyHit(5)  ; key 4
    BP_SimulatePacketLoss(2)
 EndIf

 ; sets the timeout period
 If KeyHit(6)  ; key 5
    BP_SetTimeoutPeriod(10000)
 EndIf
 If KeyHit(7)  ; key 6
    BP_SetTimeoutPeriod(20000)
 EndIf

 ; sets up the logfile
 If KeyHit(8)  ; key 7
    BP_StartLogFile("test.log")
 EndIf
 If KeyHit(9)  ; key 8
    BP_StopLogFile()
 EndIf

 ;Now show the incomming messages to the host
 textcounter = 0
 counter = 0
```

```
        For i.Info = Each Info
            counter = counter + 1
           ; if we have more than 15 listed, delete the first
            If counter > 15 Then
              Temp.Info = First Info
             Delete Temp
            Else
              Text 0,FontHeight()*textcounter, i\txt
              textcounter = textcounter + 1
            End If
        Next

       ; throw out some status text
       Text 0,300,"Current Players in game = " + ↲
                → BP_GetNumberOfPlayers()
       Text 0,320,"Current Game Type = " + BP_GetGameType()
       Text 0,360,"Current Timeout = " + ↲
                →  BP_GetTimeoutPeriod()
       If BP_GetLogFileName$() = ""
          Text 0,380,"Log File = NONE"
       Else
          Text 0,380,"Log File = " + BP_GetLogFileName$()
       EndIf

       Text 230,300,"Packets Sent = " + BP_GetPacketsSent()
       Text 230,320,"Max Players = " + BP_GetMaxPlayers()
       Text 230,340,"Host ID = " + BP_GetHostID()
       Text 230,360,"Host IP = " + BP_GetHostIP()
       Text 230,380,"Host Port = " + BP_GetHostPort()

       Text 450,300,"Packets Recvd = " + ↲
                   → BP_GetPacketsReceived()
       Text 450,320,"My ID = " + BP_GetMyID()
       Text 450,340,"My IP = " + BP_GetMyIP()
       Text 450,360,"My Port = " + BP_GetMyPort()
       Text 450,380,"My Name = " + BP_GetMyName$()

        ; flip the pages and clear the screen
        Flip
        Cls
    Wend
End If

; Make sure to shut down BlitzPlay properly!
BP_EndSession ()
```

```
End
```

If you want to try that demo on different machines, change the following line:

```
reason = BP_JoinSession ("client", Rand(1002,2000), ↲
                              → BP_LocalHost,1001)
```

To:

```
reason = BP_JoinSession ("client", Rand(1002,2000), ↲
                           → BP_ConvertIP("##.##.##.##"),1001)
```

Where the "#" signs are the IP address, for example:

```
reason = BP_JoinSession ("client", Rand(1002,2000), ↲
                           → BP_ConvertIP("55.55.125.123"), 1001)
```

I have no idea whose IP address that is, if anyone's…so you may not want to try and actually connect to that one. It's just an example!

There are a bunch of commands that you can use to see what's going on in your game. To see a full list of the BlitzPlay commands, go to http://www.blitzcoder.com/blitzplay.

Chapter 26: The Network Space Game Demo

On the CD there is a demo game that demonstrates the multiplayer development method, but it does more than that! It also uses a number of things found in this book. It uses Z-Ordering, collision detection, TYPES and arrays, input, sound, animation techniques, homing missiles, objects, and of course the multiplayer code using BlitzPlay.

Designing the Game

I've always been a huge fan of space games. I don't know why, I guess it has a lot to do with all the space shows that were prevalent in my youth. Also, I always enjoyed a good game of Subspace when it was around. So I figured it'd be fun to write a quick little space game to demonstrate a number of techniques we've discussed throughout this book.

Story

Human life had all but ended in the Wyke-Beulah sector, due to the horrible onslaught of the Pongo Empire—a Specis of SurreaL, yet warlike savages controlled by the Lords Sharpe and Shadow. With only a few surviving clans remaining, battle for home worlds is tough and ruthless, and the pilots are rough and toothless. The only ship class remaining is the Sibly-1k, which sports a Bray-9 Shield casing, and fires Snarty-11 laser pulses.

Setting and Point of View

This is a 2D top-down space game with a parallaxing star field backdrop. The player has one ship, which is always set in the center of the screen. There are 9 planets laid out in a square so players can keep track of their positions more easily.

Player's Purpose

The purpose is to blow up other players, while avoiding being blown up. There is no ultimate goal really…it's a demo.

Art Requirements

Ship: 32x32, rotated within the file. Total of 36 frames to allow for 10-degree incremental spinning.
Planet: 128x128, animated within the file. Total of 32 frames for smooth spinning appearance.
Laser: 9x9, not rotated within the file. Will do the rotations on this at load time.

Explosion: Large (128x128) animated within the file. Small (32x32) animated within the file. Total of 24 frames to show the full range of explosion to dissipation.
Exhaust: 3x3, faded within the image. Total of 4 frames.
Star Fields: 200x200. There are two fields used. One is brighter than the other, but they are the same layout.

Statistical Settings
Top Speed: Standard engine top speed runs at 3 world units per frame. Boost speeds move that to 8.
Thrust Speed: Standard thrust is at .025 per frame. Boost thrust runs at .125.
Turning Speed: Ships may turn every 40ms. This helps the player control angle.
Firing Speed: The player's ship will release a laser pulse every 350ms. We don't want the bullets coming out too fast for a number of reasons, with the primary being packet send control.
Firing Distance: Standard distance is 100 frames on the side of the player who fired, and 150 frames in the worlds of all other players. The additional 50 frames are to compensate for network latency.
Laser Speed: Lasers move at the firing player's top speed plus 3 world units per frame. Thus you can't outrun a laser pulse without using boosts.
Shield Power: Shields are defaulted to 200 points and regenerate at 5 points every 1 second.
Laser Power: A laser impact causes a 10-point drain from the current shield strength.
Player Status: 0 = Player is dead. 1 = Player is alive.

Network Objectives
Send updates to the host for broadcasting, thus keeping all players in as best sync as possible, and doing it in the fastest possible way (UDP). Keeping track of all player kill and death tallies is also important, as is positional information, shield strength per player, and laser firings. Finally, players must be able to communicate via chat.

Packet ID Specifications
ID 1: Positional update. This packet will contain the player's World X, Y coordinates, plus the direction the player is facing, and whether or not their thruster is on.

ID 2: Speed and Shield update. This packet will contain the player's current speed and the power level of their shields.

ID 3: Bullet launch packet. This packet contains the type of bullet launched, the direction it was facing, and the pre-determined ID of the player it will collide with (if any).

ID 22: Player was killed. Includes the information on who was killed and who did the killing. Allows the system to send a system message about it (i.e. "Joe was killed by Betty"), and launches a more encompassing explosion where the player was killed.

ID 23: Actual ID of the player that killed and the player who was killed, so each client can have the proper updated info.

ID 24: Tells us that the player is alive once again. Contains the ID of the player to reset as alive.

ID 25: Updated kill/death values. Contains the updated information on the kill/death tallies of all the players. Is used for a new player joining.

ID 50: Requests the actual name (alias) of another player.

ID 51: Contains the response from the request in packet ID 50.

ID 99: Contains an incoming chat message.

ID 100: Player has "warped zones" packet. Basically, if a player passes a World X, Y coordinate that is outside of the playing area, that player will be wrapped to the opposite side of the map. This I've decided to call "warping zones." This packet informs everyone that a player has, in fact, done this.

ID 199: Host "targeting" option. If a player is acting in a not-so-nice fashion, the host may use the command "/target=<player>". This will cause ALL laser pulses to fly only at the player targeted. It will increase the speed of the laser pulses to 6 and make their distances 10,000 frames before fizzling. Additionally, the targeted player will be unable to fire. Using "/target=" without a name will shut this off.

ID 254: A player has left the game. Simply contains the ID of a player who has disconnected.

ID 255: A player has joined the game. Contains the ID of a player that has just hopped in. Here's where we setup a lot about the player.

Network Update Frequency

This one took a little tweaking. I opted to go for standard positional updates every 150ms. With the "smoothing" algorithms, this could probably be

expanded to twice that, thus lowering the network traffic. Speed and shield updates go out every 500ms, as they are less important than the positional information.

Chat goes out whenever the user hits the <enter> key. This is because there's no way that a user is going to type multiple lines faster than 100ms.

Handling "Smoothing" (Dead Reckoning) of Ships

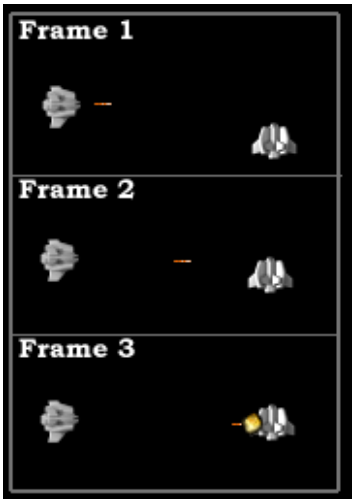
For the ships I used a "Homing object" method (as discussed in Chapter 22). Every positional update contains the latest World X, Y coordinate of a player. All I do is take that value and put it as the Target X, Y. Then I determine the best angle to compute the Sin/Cosine speed values on to bring the ship closer to that point, and move the ship accordingly.

I had originally had this so it would just update the World X, Y and display the ship at the new location. The problem with this is that the ships would blink all over the place. With the method I'm using here, the ship now appears to "move" to the new location from its previous location.

Note though that this only happens if the remote player is within the viewing player's visual field. Otherwise, I don't waste time smoothing, and instead just update the World X, Y.

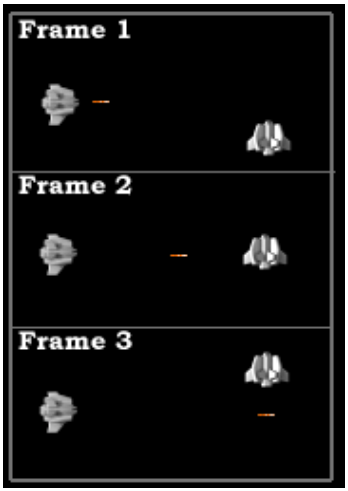
Laser Determinations (Dead Reckoning)

I decided on a less traditional method of handling laser-to-ship hits. When a laser is fired, the player that fired it may see a collision and think that they've damaged another player. The problem is that by the time the laser packet gets to the other player, many milliseconds have passed and that player may no longer be in the position he was when the laser was fired. Visually, here's what you may see as the player firing a laser:



(Figure 26.1)

Once that bullet gets across the network to the host, and then is broadcast out to the player whom you are firing at. This is what that player will see:



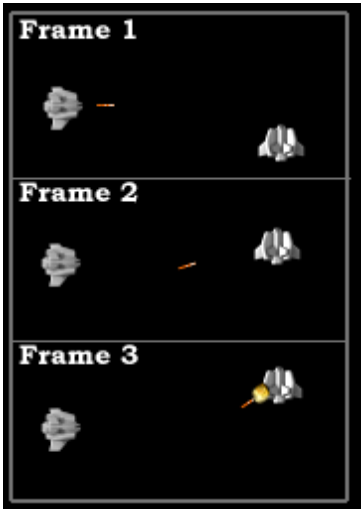
(Figure 26.2)

So while you may feel ready to run out and tell everyone about your shooting prowess, you may want to think twice. It's rather like taking the speed of the wind into account when firing a rifle at long distances.

In order to compensate for this lag-induced problem, I have cheated a little in the code. What I do is run the bullet the distance it can travel immediately upon its being fired and check to see if the bullet will come close enough for a lock. If the bullet is within a certain radius on the firing player's screen of another player, then I tag that bullet as a "Will Collide" bullet, and set the value of that bullet as the ID of the player in the game that was determined to have been hit. Now I can send off the bullet and begin animating it on the firing player's screen.

When the bullet arrives at all the player's machines, each player will check to see if it was the target of that bullet. If so, the bullet will become a homing object to their ship. If not, the bullet will just go straight or home in on another object in the visual display.

The effect looks like this on the targeted player's screen:



(Figure 26.3)

This *nearly* guarantees that if you hit a player on your screen, that player will get hit. If the packet is lost in transmission, the player won't be hit, and if the player is able to boost away fast enough, he will also avoid being hit in his world. There are many ways to handle this, but I wanted to use one that incorporated things we've already covered.

Here is the code that does this determination:

```
Function CheckCollision(dX#,dY#,iDir)
  BSpeedModifier# = 3
```

```
; determine player's speed and create bullet speed based on that
BSpeedX# = ((xSinTable#(iDir)) * (Player\dTopSpeed + ↲
          → BSpeedModifier#))
BSpeedY# = ((yCosTable#(iDir)) * (Player\dTopSpeed + ↲
          → BSpeedModifier#))

; run through each player in the world
For NetPlayer.NetPlayers = Each NetPlayers
    BWorldX# = dX
    BWorldY# = dY
    ; run the bullet out as far as it would normall go
    For BDist = 0 To 100
        ; updating the position all the while
        BWorldX# = BWorldX# - BSpeedX#
        BWorldY# = BWorldY# - BSpeedY#

        ; if the bullet's x,y are close enough to the player
        If BWorldX >= NetPlayer\dWorldX - 40 And
           BWorldX <= NetPlayer\dWorldX + 40 And
           BWorldY >= NetPlayer\dWorldY - 40 And
           BWorldY <= NetPlayer\dWorldY + 40
           ; return the player ID because he's gonna get hit!
          Return (NetPlayer\NetID)
        EndIf
   Next
Next

; no hit, return 0 since there are no 0 netID's
Return 0
End Function
```

Determining What to Display to the Player

The main thing I wanted to do here was avoid unnecessary calls to drawing functions. If a ship is halfway across the galaxy from me, why should I try to draw that ship?

In order to make appropriate determinations, I decided to only draw things that were within a certain range of my ship. I could easily expand my determinations per item drawn by increasing or decreasing the amount accordingly. For example, planets are much larger than ships and lasers, so I will allow them to be drawn a bit further out.

Since I know the World coordinates of all things in the game, I just see where everything is in relation to me. If an object falls within a certain range, I

translate that object's World coordinates into Screen coordinates and draw the image. Here's the idea of it:

```
; if the planet is in the visual area, draw it
If Planet\dWorldX >  Player\dWorldX - ScreenCenterX - 105 ↲
  → And Planet\dWorldY >  Player\dWorldY - ScreenCenterY - 105 ↲
  → And Planet\dWorldX < Player\dWorldX + ScreenCenterX + 105 ↲
  → And Planet\dWorldY < Player\dWorldY + ScreenCenterY + 105

  PlanetX = (Planet\dWorldX - Player\dWorldX) + ScreenCenterX
  PlanetY = (Planet\dWorldY - Player\dWorldY) + ScreenCenterY
  DrawImage Planet\Image, PlanetX, PlanetY, Planet\iFrame

  ; draw the planet name over it
  Color(0,119,158)
  Text PlanetX-100,PlanetY-80,Planet\Name$
EndIf
```

The conversion from World to Screen is done by taking the Object's World X and subtracting the Player's World X from it, then by adding the center of the Screen to that. Do this also with the Y coordinates and you'll have the Screen coordinates.

Animating the Planets

This is a snap. All I did was determine a speed that was fast enough to avoid choppy spinning, but slow enough so the people on the planet wouldn't be flung off of it ☺. Then whenever the timer is triggered for the planet, I update the frame and display it.

Handling the Ship Exhaust

The little exhaust effect is actually a mix between using particle effects and animated images. I "launch" particles at random velocities and angles from the back of the ship as the thrust is applied. Each of these particles continues moving on its path at its predetermined speed for a particular span (called a "particle life-span").

Each particle has a decay time that leads to its eventual demise. When a decay point is hit, I change the image to the next dimmest on in the list. This is done until I'm out of images, at which point that particle is deleted.

Here is the code used to "launch a particle:"

```
Function LaunchParticles(X#,Y#,Amount,Dir)
   ; for however many were requested
```

```
   For i = 0 To Amount - 1
      ; create new instances of particles and assign
      ; positional values, but throw them out in random
      ; directions with random speeds an lifespans
      Particle.Particles = New Particles
      Particle\dWorldX = X
      Particle\dWorldY = Y
      DirOffset = Rand(-1,1)
      DirOffset = DirOffset + Dir
      If DirOffset > iNumRotations - 1
        DirOffset = DirOffset - iNumRotations
      EndIf
      If DirOffset < 0
        DirOffset = DirOffset + iNumrotations
      EndIf
      Particle\dSpeedX# = xSinTable#(DirOffset) * Rnd#(.60,1)
      Particle\dSpeedY# = yCosTable#(DirOffset) * Rnd#(.60,1)
      Particle\StartTime = GlobalTimer
      Particle\FadeSpeed = .0001
      Particle\LifeSpan = Rand(1500,2000)
      Particle\Frame = 0
      Particle\Image = Exhaust_Image
      Particle\dWorldX# = Particle\dWorldX# - ↲
                          → (Particle\dSpeedX# * ShipSize)
      Particle\dWorldY# = Particle\dWorldY# - ↲
                          → (Particle\dSpeedY# * ShipSize)
   Next
End Function
```

Next, we need a way to continue the movement of that particle and to slowly fade it over time before removing it. Here's the function that handles that:

```
Function UpdateParticles()
   ; run through all of the living particles
   For Particle.Particles = Each Particles
      If GlobalTimer > Particle\StartTime + Particle\FadeSpeed
         ; reset the start time for this particle
         Particle\StartTime = GlobalTimer
         Particle\Frame = Particle\Frame + 1
      EndIf
      ; update particle's X,Y location based on speed info
      Particle\dWorldX# = Particle\dWorldX# - Particle\dSpeedX#
      Particle\dWorldY# = Particle\dWorldY# - Particle\dSpeedY#
```

```
    ; if particle has lived out its life, delete it
    If CurrentLifeTime > Particle\StartTime + Particle\LifeSpan
       Delete Particle
    Else
      ; if we've gone past the frame limit, then delete it
      If Particle\Frame > 3
        Delete Particle
      Else
        ; otherwise draw the particle
        If Particle\dWorldX > Player\dWorldX - ScreenCenterX - 64 ↲
         → And Particle\dWorldY > Player\dWorldY - ↲
        → ScreenCenterY - 64 ↲
        → And Particle\dWorldX < Player\dWorldX + ↲
        → ScreenCenterX + 64 ↲
        → And Particle\dWorldY < Player\dWorldY + ↲
        → ScreenCenterY + 64
          ParticleX# = (Particle\dWorldX - Player\dWorldX) ↲
                    → + ScreenCenterX
          ParticleY# = (Particle\dWorldY - Player\dWorldY) ↲
                    → + ScreenCenterY
          DrawBlock Particle\Image, ParticleX, ParticleY, ↲
                    → Particle\Frame
        EndIf
      EndIf
    EndIf
  Next
End Function
```

Displaying the Mini-Map (Radar)

I found this one a bit tricky at first, but it quickly became a snap. The problem is translating a ton of space (no pun intended) into an itty-bitty space (pun intended).

What I needed was to figure out the algorithm for determining the radar plane. After determining this value, I can then use the same algorithm I used for determining what to display to the user on the screen and apply that to the little map. I would, of course, use smaller graphical elements to do so.

The default *scan-size* for the radar is 3,000. This means that I want a visual depiction of all the tracked objects within 3,000 World units of my player. What this actually translates into, however, are 3,000 World units to the left, to the right, up and down. So in actuality, I will be tracking double that amount: 3,000 to the left and 3,000 to the right. Likewise for up and down. Thus, I'm really tracking 6,000 total square World units.

Next I need to know the size of the display field for my radar area. In the case of this game, that little radar window is 100x100.

Now that I have these two values, I'm set to pull in the data of whatever is within a full 6,000 World elements by using this IF statement:

```
If Planet\dWorldX > Player\dWorldX - ScanSize ↲
→ And Planet\dWorldY > Player\dWorldY - ScanSize ↲
→ And Planet\dWorldX < Player\dWorldX + ScanSize ↲
→ And Planet\dWorldY < Player\dWorldY + ScanSize

  ; ...do stuff...
EndIf
```

The *ScanSize* value here will contain *3,000*. So if you look at that statement, you'll see that it checks both positive 3,000 and negative 3,000…totaling our 6,000. If anything (in this case a planet) falls within that distance of the player, it's included in the display.

But how do we translate that into mini-map coordinates?

By using the following algorithm:

*ScanDivisor = (ScanSize * 2) / RadarSize*

…or…

*ScanDivisor = (3000 * 2) / 100*
ScanDivisor = 6000/100
ScanDivisor = 60

Now we just do our normal World-to-Screen translations and divide it by the *ScanDivisor* (in this case, 60). Setup our base X, Y locations to begin the drawing from, do the calculation with the appropriate divisor value and then draw up an oval for the map image.

```
MiniMapX = ScreenWidth - 150
MiniMapY = 10
PlanetX# = ((Planet\dWorldX - Player\dWorldX) + ↲
          → ScanSize) / ScanDivisor
PlanetY# = ((Planet\dWorldY - Player\dWorldY) + ↲
          → ScanSize) / ScanDivisor
; now just draw up a little oval in blue on the mini-map radar
```

```
Color 0,255,255
Oval MiniMapX+PlanetX,MiniMapY+PlanetY,3,3,1
```

The Demo Code

The following files are on the CD under Network Space Game. They all-encompass the full demo.

Spacegamenet.bb: The main application for the network space game.

Defines.bb: Contains all of the global variables and types that are game-specific.

Chat.bb: Contains all of the chat functionality for the game.

Network.bb: Contains the functions to setup the network for the game and to process the various packets.

SinCosTables.bb: Handles the pre-computing of the Sine and Cosine tables.

Waypointlib.bb: Holds all of the angle and distance determination function for the game's smoothing and dead-reckoning purposes.

Keyconstants.bb: Holds English-like names for the various key scan codes.

BPLite.bb: The actual BlitzPlay Lite source code.

You may feel free to use the code however you'd like. Modify it to your heart's content! Maybe try taking this source code and add in sounds and music? Or take a crack at adding your own ships and planets!

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z