
Programming 2D
Scrolling Games
Updated for
PureBasic 4.61 & 5.0
John P. Logsdon
Derlidio C. L. Siqueira

Programming 2D Scrolling Games

Updated for PureBasic 4.61 & 5.0

Programming 2D Scrolling Games
Authors: John P. Logsdon ("Krylar")
Derlidio Siqueira ("PJoe")
Graphics and Cover Art: Ric Lumb ("Putty")
Game Music: Steve Harrison ("Fash")
Editing: Lorelei J. Logsdon ("Soeth")

Visit me on the web

www.JohnPLogsdon.com

PART 1: PUREBASIC BASICS

Chapter 1: Welcome to PureBasic

This book is designed to get you started programming in one of the most powerful basic-like languages available today. Taking you from fundamental programming concepts to advanced techniques, ***Programming 2D Scrolling Games*** will have you designing and developing your own games in no time.

What is PureBasic 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.

I've since used many development languages on the market that were written with the hobbyist in mind. PureBasic is such a language. But don't let that hobbyist tag fool you! Most people would find that a language such as PureBasic is much more robust and powerful than their own hand-coded routines in a language such as C/C++. Also, it's far simpler to master.

PureBasic 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. PureBasic uses one of the simplest languages as its base, BASIC. However, where BASIC is an interpreted language (meaning that as the program runs, the computer translates each line into machine language before executing it), PureBasic compiles the code directly to machine language before executing any lines. This means that a program created with PureBasic will run without unnecessary steps that can slow it down.

Something equally important about PureBasic is that it's a REAL programming language. I have seen a number of products that are known as "Click and Play" game development systems, but PureBasic 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 PureBasic 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 PureBasic?

There are many languages out there that you could choose from, so why pick PureBasic? The simple answer is that PureBasic will get you developing your games and applications 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, PureBasic will allow you to prototype games quickly and easily without drastic speed loss and inherent restrictions of a click-n-play type system.

Finally, PureBasic has game development as part of its function. C and C++ are used in a lot of game development projects, but they were not designed with programming games in mind. PureBasic was. Therefore, when you start out with PureBasic you are in a language that supports your goal of game development.

What Will I Need to Run PureBasic?

In order to run the PureBasic Integrated Development Environment (IDE), you'll need to have a system running Microsoft Windows. While PureBasic also has versions available for Linux, Amiga, and Mac, this book will only be focused on the Windows versions.

This book is based on the commercial version of PureBasic. Some of the example programs may not work with the demo version of PureBasic. Also, make sure you have the latest version. At the time of this writing, I am using PureBasic Version 4.61.

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

Section two, "PB 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 "PureBasic" a lot. To make for easier reading, you'll often see me refer to PureBasic as simply "PB."

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

```
Result = OpenScreen(800,600,16,"My Game")
```

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

```
↵ and →
```

Such as:

```
If Shields < 100 and Armor > 100 and ↵
   → RepairAbility < 10

  Gosub DestroyShip
EndIf
```

The "↵" symbol means that the line is continued on the next line. Depending on the interface you're using to read this book (iPad, for example), and your selection of font and font size, you will see the lines of code break in various locations. In the actual PureBasic development environment you will need to type the line as one full line because PureBasic 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. The goal of these two symbols is to help you know which lines stay together. Unfortunately, it won't be an exact science since there are so many combinations of devices and layouts that I could not possibly account for, so please be sure to use care when entering code from the text. Also, always keep in mind that you can download the source code, which will not have this issue since it is already in PureBasic's required format.

Where can I get the source?
You can download the source at
https://www.mediafire.com/folder/aajnae2bi4ta4/Pure_Basic

What if there are errors in the book or code?
While both myself and the editors have tried to catch all the errors, it's likely the case that something slipped past all of our testing.

I am no longer supporting this book, which is why it has been released for free.

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 PB.

The developer of the PureBasic language, Fantaisie Software, 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 PureBasic looks like:

```
If InitSprite() = 0 Or OpenScreen(640, 480, 16, "Test") = 0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
Else
   ClearScreen(RGB(0,0,0))
   StartDrawing(ScreenOutput())
      DrawText(280,240,"Hello, World!")
    StopDrawing()

    FlipBuffers()

    Delay(5000)
EndIf

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, which CAN be used inside of PureBasic) that are much more cryptic when compared to the easily-read PureBasic language.

Object Code

When you have completed a project, you must request that PureBasic 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.

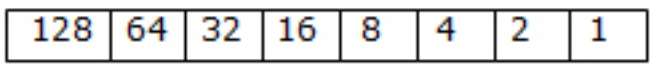
Object Code is the native language of your computer's processor. It's nearly impossible to read since it is purely numerical, which is why we develop in languages such as PureBasic and allow the compilers to do the conversions for us.

Bits and Bytes

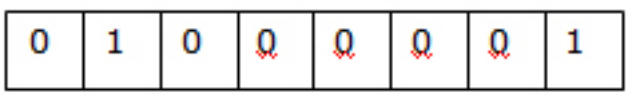
Before going much further, let's touch on the topic of bits and bytes as you'll need to know what these are for some of the information coming up.

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.



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:



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 255, giving you 256 total states. Keep in mind that a computer counts from 0, not 1. So if you take all the bit

values and set them to 0, you'd have a value of 0. If you set them all to 1, you'd have a value of 255. If you count from 0 to 255, keeping 0 inclusive (as in "0…1…2…3…etc.), you would count 256 when finished. So, again, you have 256 elements in each byte, but the maximum actual value is 255 because of the 0 start.

Screen Resolutions and Bit-Depth

In order for PB 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 4-bit, 8-bit, 16-bit, 24-bit, and 32-bit.

4-Bit Color: As you may recall in our discussion on bits and bytes, 4 bits can only contain a number up to 16. So in 4-bit mode, we have 16 colors to work with. In this day and age, that's pretty pointless, but if your OS supports it you can always push your personal boundaries and try to make something work at such a low color value.

8-Bit Color: 8 bits can only contain a number up to 255, which, starting from 0, is 256 states. This means that each pixel drawn can be 1 of 256 colors (0-255). 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-Bit Color: 24-bit, also known as *True Color*, gives us the ability to use one of 16,777,215 colors per pixel. That's a TON of color choices… more than the human eye can distinguish, actually. It's argued that there's no real point in going any higher in color on video cads and printers since we won't be able to distinguish the subtleties anyway.

32-Bit Color: 32-bit is really just an extension of 24-bit. It has the same number of colors because the first three bytes (the 24-bit component) are for Red, Green, and Blue. But the addition of 8 bits gives two advantages: 1) It keeps the memory "byte aligned," meaning that since Intel-based chips move data along the bus at 32-bits per move, 32-bit color moves the data without adjustment. 2) While the original idea was that the 4th byte (bits 25-32) were simply for speed and thus discarded upon hitting video memory, it was decided to put that 4th byte to use. Now that 4th byte is useful in "alpha channeling," or "masking." This means that we can specify how we want to *merge* a color of a pixel when it overlays another pixel. So instead of overwriting a blue pixel with a red pixel, for example, we can display a pixel at that location that is a merger of the two colors thus giving the effect that one pixel is crossing over another, which gives the illusion of depth.

Speed Impact of Higher Resolutions and Bit-Depths

The higher the resolution and bit-depth, the slower your game will run (except for 32-bit over 24-bit. 32-bit does run faster than 24-bit due to the architecture). The reason for the speed differences comes down to how many pixels must be displayed per screen and how many bits each pixel contains, and also how data is moved back and forth using proper alignments...meaning that a computer will more rapidly move a 32-bit value than it will a 24-bit value because of the architecture of a computer. 24-bit values require that the computer do offset-computations where a 32-bit value does not require the same calculation since the machine is built to work with such values.

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 very speedy. You almost have to work at slowing the things down. But, trust me, you can if you really try.

DirectX, Peripheral Cards and Drivers

PureBasic uses a proprietary graphics engine that sits on top of 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 PureBasic 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. PureBasic 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, PureBasic lets you get to work on your project without having to understand all the fundamentals of Windows and DirectX programming.

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.

To move up with the times since the original writing of this book, multiple monitors are the way to go. I have five (yes, 5!) monitors at my desktop now. It's taken a while to build this system up, of course, but at this point I can't imagine life as a developer with only one monitor. If you're still stuck in single-monitor land, you don't know what you're missing! But, yes, I still have the toys on my desk.

Chapter 3: Getting Started with PureBasic

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.

OpenConsole

First let's go ahead and see what the OpenConsole command is like. Type in the following code *exactly as shown*, save the file and then ask PB to run it by pressing F5 on your keyboard.

```
OpenConsole()
Print("Hello, World!")
Delay(10000)
End
```

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

```
OpenConsole()
```

This command instructs PB to open up a DOS-like window, also known as a Console Window. It's the area of the Operating system that is non-graphical, giving the user a more direct textual approach to development.

Why would anyone want to use this? Well, a number of programs do not need the heavy (or even light) graphics and visual appeal usually ingrained in Windows applications. For example, you may have a program that just runs through a bunch of files and lets you know how many total characters there are in them. Why go through days of building a graphical interface when you can easily snag that information with a few lines of code?

To see a console application at work, you can go to your Windows START-RUN area and type in "ping www.purebasic.com" and you'll see the data spit out in white on a black background with no graphics present.

```
Print("Hello, World!")
```

The Print command tells PureBasic that you wish to display some information to the user. This is a very straightforward command that accepts the text you wish to display as an argument contained in the quotes.

```
Delay(10000)
```

By using Delay we effectively pause the application for a set number of milliseconds. This will give us time to read the output of the application before it quits. You would likely want to use a method of waiting for a key press before exiting, but we're going to keep it simple for now. Note that I used the value of 10,000 as the argument. Since the argument is in milliseconds, the Delay command will hold up the application for 10 seconds.

```
End
```

While PB is smart enough to end on its own, once the application is finished, this command is not really necessary. But I find that it is always good practice to include as it is always wise for the programmer to rely on his/her ability to make sure loose ends are tied up, not something that should be easily relinquished to the language one uses.

OpenWindow

If you're looking to open up a standard Windows application window, you'll be calling PB's OpenWindow command. Again, to see this in action, type in the following code exactly as you see it and then run it. Don't forget that the ↵ and → are not to be typed in, they're just to denote that the text is all part of a single line.

```
OpenWindow(0,200,200,200,100, ,"Hello, World Test Application", ↵
→ #PB_Window_SystemMenu | #PB_Window_TitleBar)

StartDrawing(WindowOutput(0))
    DrawText(0,0,"Hello, World!")
StopDrawing()

Delay(5000)

End
```

There is a lot of information in that code. Here's the breakdown:

```
OpenWindow(0,200,200,200,100, ,"Hello, World Test Application", ↵
→ #PB_Window_SystemMenu | #PB_Window_TitleBar)
```

The **OpenWindow** command has a number of arguments available to it. To get the full effect of these, please highlight that command in your PB IDE window and press the F1 key.

In this example, I'm telling PB to identify this window as "0". That's the first argument, which is used so you can set a numeric identifier for each window you create. You can also use #PB_Any to have PureBasic assign a unique value for you, which, as we'll see in later chapters can be quite useful.

The next two arguments tell PB where you want the top-left edge of the window to start, as default. In this example, I'm instructing PB to start the window at position 200,200 of the screen.

Next I tell PB I want the window to be 200 pixels wide by 100 pixels high in its internal area.

Then I sent along the name that we want to display in the title bar for this application.

And finally I send two *flags* to the function on how I want the window to operate. In this instance I want to make sure there is a System Menu, an "X" in the upper-right to allow the user to close the window (though in this example I'm not checking for mouse clicks so it really won't allow the "X" close option), and also I want to make sure that the title bar is displayed. The title bar display is the default anyway, but I wanted to show that multiple flags can be used by using the "|" operator (which is the OR operator).

Again, make sure to hit F1 after highlighting the command in your IDE to see the full options.

```
StartDrawing(WindowOutput(0))
    DrawText(0,0,"Hello, World!")
StopDrawing()
```

The StartDrawing command informs PB where you want it to draw various items, such as text, circles, pixels, etc. This command accepts an argument that allows you to specify where, exactly, something should be drawn. In our example, we're telling the command to use our current window by passing the 0 in WindowOutput. Remember that the first argument we passed to OpenWindow was 0, so we'll need to use that here.

DrawText is similar to the Print command we used back in the Console example. Its purpose is to put text up on a window or game window screen. In a little while we'll be covering the use of various fonts and colors we can use with this command. The 0,0 at the beginning of the call is the x,y coordinate to display our text. We will discuss coordinates shortly.

StopDrawing is the command that informs PB you're done drawing to the selected surface.

```
  Delay(5000)
End
```

Here, again, we delay the execution of the program so you can see things happening. This time I've set it to 5 seconds.

We've already touched on the End command, but again it's just to tie things up and tell PB you're done with the program.

OpenScreen
Our final form of screen control is the one that you would most commonly use for full screen games. Sample code:

```
If InitSprite() = 0 Or OpenScreen(640, 480, 16, "Test") = 0 ↵
→ MessageRequester("Error!", "Unable to Initialize Environment", ↵
→ #PB_MessageRequester_Ok)
  End
Else
    ClearScreen(RGB(0,0,0))
    StartDrawing(ScreenOutput())
        DrawText(280,240,"Hello, World!")
    StopDrawing()

    FlipBuffers()

    Delay(5000)
EndIf

End
```

As with the last two examples, let's break this down.

```
If InitSprite() = 0 Or OpenScreen(640, 480, 16, "Test") = 0
```

The InitSprite command let's PB know that you want it to use the game graphics commands known as "sprites." We'll be discussing sprite commands in full detail later in the book, for now though just know that when performing any commands regarding to full screen games, open through OpenScreen command, you must be sure that the user environment is capable of dealing with them, and InitSprite checks this out for you.

A video card with enough video memory and proper drivers (i.e. DirectX) must be installed on the user's system. InitSprite will do that checking and tell us if the system is OK for game-specific needs. There are some other steps we should consider to tell something to the user

when his/her system is not capable of running our program, but we'll get to that later. For now just keep in mind that for full screen games InitSprite must be at the beginning of your program, before any other graphic commands are called.

Also notice that we are using an IF here and comparing the value returned by InitSprite to zero(0). If the value returned is 0, then InitSprite was *not* able to setup the system properly and therefore we need to let the user know there was a problem and then exit.

The OpenScreen command is where we'll be setting the width, height, bit-depth, and title of our application. Very similar to the OpenWindow command we discussed before, but with a number of differences. First, note that there are no flags usable in this command, and also that there is no starting X and Y location for the window. This is because OpenScreen is used for full screen, non-windowed applications, most notably games.

Another thing that OpenScreen does is set up the environment to have multiple buffers. Buffers are areas of memory (most commonly video memory) that your program writes graphics too. Once the buffer is completed, your program shows the completed buffer to the user and then begins writing the new graphics to the next buffer.

Note that we also check the return value of the OpenScreen command to make sure it's not zero(0). If it is, we display an error message and exit the application.

There are some commands in PureBasic (not all of them) that, after performing the action they are meant for, return a value to inform us if they were successful or not. Generally, if the returned value is NOT zero(0) it means that the command has performed OK. In programmer-speak, when a non-zero value is returned from a command, we say that the command has returned a "True," or "Success!" On the other hand, if the returned value is 0 (zero), then it means that the command could not perform its assigned task. In that case, we use to say that the command has returned a "False," or "Failure." It's up to the programmer to handle these True and False answers from PB commands accordingly. Despite the importance of these returned values, we're not going to deal with these worries for now. Why will we dare to neglect such precious information? Because the code becomes more complicated for beginners to understand. There are some very basic concepts that must be introduced first. Once we have introduced all the concepts you must be aware in order to deal with the command results, then we'll start to handle them in the way they deserve to be handled! In other words, we'll teach you how to cook, that's the deal. But first, we must teach you how to setup some fire, and how to choose from the various models of pans, and which spices to put in the sauce...

Whenever we move from buffer to buffer, most often we will want to make sure there's nothing on that buffer. So we clear it.

```
ClearScreen(RGB(0,0,0))
```

The ClearScreen function will clear our screen with a particular color. Note that we use the RGB command to get the final color. There are three arguments for the RGB command: Red, Green, and Blue. By placing a 0 in each argument, we have set the color to clear the screen with to black.

```
StartDrawing(ScreenOutput())
    DrawText(280,240,"Hello, World!")
StopDrawing()
```

Just as in our last example, this grouping of commands will draw our desired text to the screen. You should see thought that the argument inside of the StartDrawing command has changed. The ScreenOutput command returns the screen device used for 2D drawing operations.

```
FlipBuffers()
```

Now that we've drawn our text to the buffer, we need to display that buffer to the user. The FlipBuffers command does just this. This will be discussed in much greater detail in later chapters.

And we again finish with the Delay and End commands.

I know that I've pointed a lot to later chapters for some definitions. I apologize for that, but to try and cover these pieces in detail now would be premature and confusing. We'll get to them soon enough, though. I promise!

Chapter 4: The Basics of PureBasic

Variables, What are they?

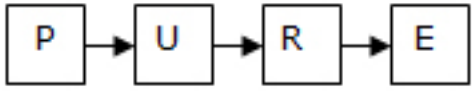
An important part of any PureBasic program is the ability to have various forms of data. It can be numerical, character, memory, graphical, etc. All data can be entered into your program manually, but this doesn't allow for the dynamic nature of most applications.

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 PureBasic:

- String (also known as Scalars)
- Fixed String
- Byte
- Ascii
- Character
- Word
- Unicode
- Long
- Integer
- Float
- Quad
- Double
- Pointer/Handle

Strings: A String is simply a collection of characters. For example, "PureBasic" is a string of 9 characters. Why is it called a String? The idea is that the "P" is tied to the "u" and that is tied to the "r," and so on. So if you were to take all those letters and "string" them together, you would get the word "PureBasic." Consider the following:



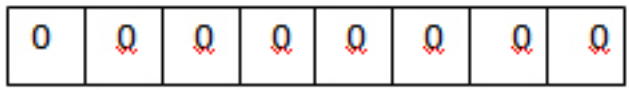
Individually, just like letters in the alphabet, these are simply characters. But those arrows demonstrate the way PB will look at a string. Each letter points to the next and treats them as a word.

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

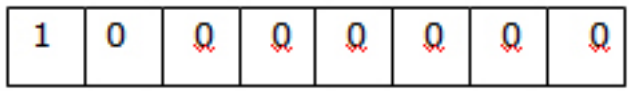
description of that planet, the ship they're flying, the ship's description, etc.

Byte: A collection of 8 bits. As you may recall in Chapter 2, a byte may hold any number from 0 to 255 (a total of 256 elements). PureBasic byte-type variables, though, are signed variables. This means that instead of using the whole set of 8 bits for holding values between 0 and 255, they'll use only 7 bits for this purpose and save the very last one to serve as a sign. If the sign is off (0) then the value is positive, if it is on (1) then it is negative. Because of this, the possible values contained within a byte variable in PureBasic are –128 to +127.

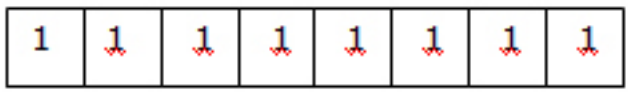
Why is it –128 to +127 and not –127 to +127? This has to do with the sign bit. Consider the following:



This layout gives us the value of 0. 0 is neither positive nor negative, so the sign bit is not enabled. But what happens if we decide to enable the sign bit?



Now we have an issue. Since 0 cannot be negative, the CPU will look at this as both a sign bit action *and* as 128. So we have –128. But what if we did this:



Doing the above would give us –127 because the sign bit is not the only bit enabled. It is only when it is solely enabled that the CPU will consider it both negative and 128.

Often times a programmer will use Byte values for numbers that are certain to be very small. Since it's always wise to use as little memory as possible, programmers look for every opportunity to use a Byte variable size. An example usage for a Byte value in a game would be ship types. Let's say that you have 18 different ships that a user may purchase throughout the many levels of your game. Since 18 is definitely never going to go beyond the positive limit of 127, you may as well use a Byte value to store the ship numbers.

Ascii: Similar to a byte, but the values it holds are from 0 to +255, which allows you to use the extended Ascii character set.

Word: A collection of 16 bits, or 2 bytes. A Word variable is signed, like the Byte value discussed above, and therefore follows the same rules as the Byte value. But the range of numbers it can hold is -32768 to +32767. Again, note that the negative value is one higher than the positive. To understand this, please review the Byte description above.

I have used Word values on a number of things in a game, and have found it probably one of the more common variable types in things I've done. One example is my character's position in the game world. It's not often that you'll have a game map be so large that it will go beyond 32,767 world units. It's possible, certainly, but it would make for a massive map. I tend to keep my maps less than that though, so I most often use a Word value (or a Float) to store my map locations.

Unicode: Similar to Ascii, but much larger. This type can hold values from 0 to +65535. You would usually use this type for multiple language support where diverse characters are required.

Long: The default variable type for PureBasic. A Long is also known as a "Double Word." This is because it consists of 4 bytes, or 32 bits, so it is exactly two times larger in bits than a Word is. And, like a Word and a Byte, a Long variable is signed. Its range is -2147483648 to +2147483647.

High scores is the easiest example here. When you are running up your player's score, use a Long. It'll hold very high numbers. If you ever foresee going beyond the maximum possible value in a Long, go with a Float.

Integer: Depending on your operating system, the Integer may support 4 bytes (32-bit OS) or 8-bytes (64-bit OS). If 4 bytes, the Integer may contain values from -2147483648 to + 2147483647; if 8 bytes, the Integer can hold -9223372036854775808 to +9223372036854775807

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 0.00001 would take quite a while.

I tend to use Floats when I want to move my sprites at really slow increments, mostly for smoothing movement.

Quad: This type is similar to the 64-bit Integer, so it's 8 bytes and can hold values from -9223372036854775808 to +9223372036854775807.

Double: A Double is essentially a 64-bit float. It has 8 bytes, but since it (like the Float) supports scientific notation, it is essentially limitless.

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 PureBasic programs.

- Local: Variables created this way will be available for reading and manipulation only by a predefined portion of the program, and will be alive only for the time needed to execute the code that portion contains.

- Protected: This type of variable acts as the Local type, but with a twist. It allows the programmer to name a variable within a portion of code using the same name of another variable already declared as Global, without causing conflicts.

- 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 "l" for Long, "s" for String, "f" for Float, etc. So, instead of using *TotalScore*, I would likely use *lTotalScore*. I now know, by just a glance that this variable is a Long and it's used to hold the total score of the player. As you will see shortly, though, PB variables are defined with the type of variable at the end of the variable name. So, you could also just keep using that throughout your code to keep track of the type of variable you're working with.

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
- bCounterValue
- fShipAcceleration
- PlanetDescription
- BrakingSpeed
- ShieldPower
- WeaponType.b
- JumpSpeed.w
- ShipName.s

This is also known as UpperCamelCase because most of the variables start with an uppercase letter and then each word has another uppercase letter. The idea of "camel" here is that the letters represent the bumps on the back of a camel. So, "PlanetDescription" has two bumps.

With lowerCamelCase you would start the first word in the variable with a lowercase letter. For example: planetDescription.

Why have upper and lower camel case options? That really depends on you, the developer. You may want to have all of your standard variables as lowerCamelCase and all of your globar variables as UpperCamelCase so that you can differentiate them. Different development houses have different naming conventions. The main point is to try to make a rule and be consistent so that you can know at-a-glance what you're looking at.

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

```
TopSpeed.b = 5
```

That one line sets up a variable named *TopSpeed*, as a Byte 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:

```
TopSpeed.b = TopSpeed.b + 2
```

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

That describes how a Byte variable is setup, but what about the others? The only difference is the variable name and the type of data assigned. For example:

```
TopSpeed.f = 5.5
```

Notice the ".f" after the variable name. This tells PB that you want this value to be of type float. You only have to put the ".f" definition on the variable name when you declare the variable. After this definition, PB will remember its type.

```
TopSpeed.f = 5.5
TopSpeed.f = TopSpeed.f + 0.5
```

PureBasic will now alter *TopSpeed* to hold the value of 6.0.

```
PlayerName.s = "Krylar"
```

The above example creates a variable of type String. The .s at the end of the variable instructs PB that the data held will be character data, non-numeric.

PureBasic also allows the string variables to be created using the $ sign, as is common in BASIC-like languages. However, do note that the following variable names will be considered different variables to PB:

```
PlayerName.s = "Krylar"
PlayerName$ = "Derlidio"
```

To PureBasic, *PlayerName$* includes the "$" as part of the variable name. This is why both of these definitions are valid, and will both contain different information.

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 develop* your 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 when.

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 PureBasic 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 OpenScreen version of the "Hello, World!" program and comment it. Notice that the semi-colon (;) is used at the beginning of each comment line. PureBasic 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.pb
 ; Author: Krylar
 ; Current Version: 1.0
 ; Last Updated: 01/01/01
;*****************************************************
 ; Description:
 ;   Simply puts up "Hello, World!" and delays 5 seconds
;*****************************************************
 ; Update History:
 ;   12/01/00: Started project
 ;   01/01/01: Moved the text to the top of the screen
;*****************************************************

 ; Initialize the sprite  and a 640x480, 16-bit screen
 If InitSprite() = 0 Or OpenScreen(640,480,16,"Input Test") = 0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
   End
 EndIf

 ; make sure our output buffer is cleared to black
 ClearScreen(RGB(0,0,0))

 StartDrawing(ScreenOutput())              ; Tell PB to start drawing
     DrawText(280, 240, "Hello, World!")  ; draw out our text
 StopDrawing()                             ; Tell PB we're done drawing

 ; Flip the buffers to show our changes to the user
 FlipBuffers()

 Delay(5000) ; delay the program for 5 seconds

 End ; Tell PB we're finished with the program
```

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

```
Delay(5000) ; delay the program for 5 seconds
```

This method is fine, too. Actually, I will usually use both methods in my code, as you can see from the above example and from many to come.
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 PureBasic completely ignores all comments when it compiles your code. Thus, to PB, 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. The above example is far too heavily commented for what it does, for example. I only used it as an example so you could see many facets of commenting.

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:

```
Value.b = 1 + 2
```

Value 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:

```
Value.b = 1 + 2 * 10 / 5 – 3
```

You may think that PB 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 PB 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 PB 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 PB will handle the above calculation:

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

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

```
Value.b = (1+2) * 10 / 5 – 3
```

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

The order of precedence is as follows:

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

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 PureBasic you would use the Pow command to do exponent math.

```
Value.f = Pow(2,4)
```

This command takes the first argument and increases it by the power of the second argument. So the example call above would result in the following:

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

Cartesian Coordinates

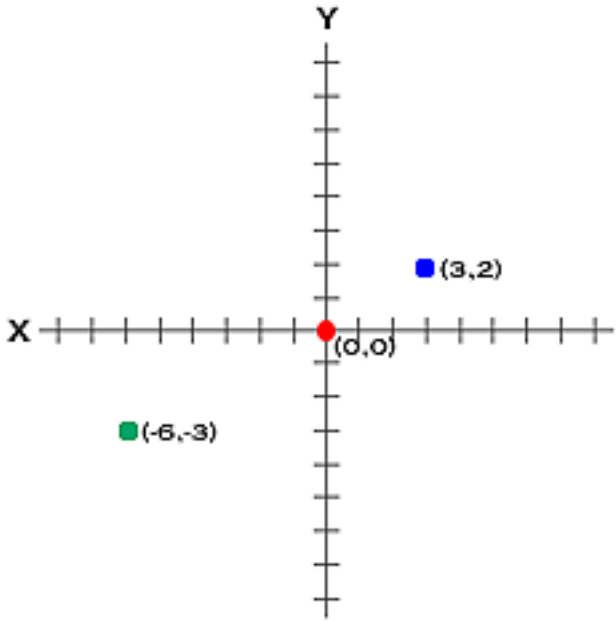
While the object of this book is not to teach mathematical concepts, the Cartesian Coordinate system is something you'll need to understand to grasp how PureBasic handles things. If you already know about this system, feel free to skip ahead to the next section.

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.

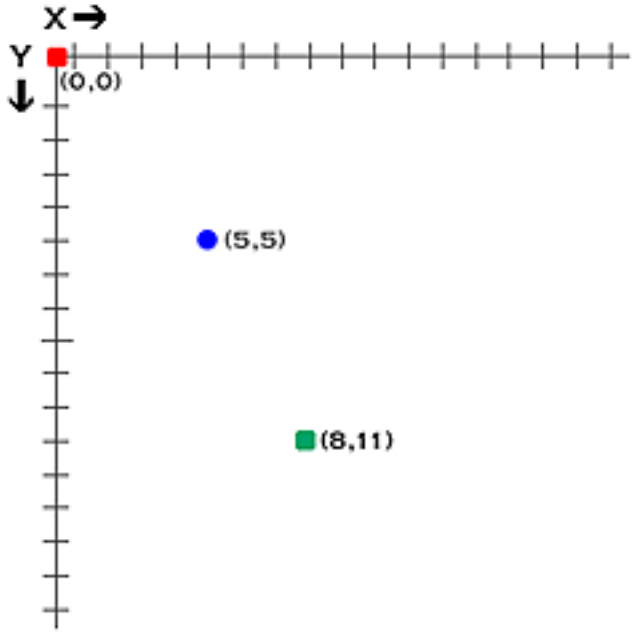
In figure 4.1, you can see what a Cartesian graph looks like. The dot at (0,0) represents the position in the graph known as the origin. The origin is the starting point of all other positions. Anything to the right of the origin on the X-axis (horizontal) is a positive number. Likewise, anything to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

PB 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 PB is the upper left corner of the monitor. Refer to figure 4.2 to see what PB does when handling Cartesian coordinates.



(Figure 4.2)

As you can see there are no negative values to worry about when drawing in PB. 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…Else…EndIf

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

Take, for example, deciding what you're going 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 in real life, you unconsciously go through the IF…ELSE process. It happens so fast (most of the time) that we're just not always aware that it's happening. Any time you make decisions in your code, however, you will have to consciously develop each IF…ELSE process to make sure your code will be handling decisions in the way you expect it to.

The format looks like this:

```
If Condition is True
    …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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16, ↵
   →"Input Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)  ; assign our black color to a variable
ClearScreen(ClearColor)  ; clear the output buffer to black
```

```
; Display text to the screen asking the user for input
StartDrawing(ScreenOutput())
    DrawText(0,0,"Enter a 1 or some other number:");
StopDrawing()

FlipBuffers()    ; flip the buffers to show the user the request

; Initialize a string variable and set it to be blank
Answer.s = ""

; Now tell PB to wait until a key is pressed before going any further
While Answer.s = ""
   ; Call PB's ExamineKeyboard to see if there's any keyboard activity
   ExamineKeyboard()
   ; If there is activity, assign key pressed to our Answer.s variable
   Answer.s = KeyboardInkey()
Wend

ClearScreen(ClearColor)   ; clear the output buffer to black

; Start drawing to the output buffer
StartDrawing(ScreenOutput())
    ; check to see if the user input the number 1 or not
    If Answer.s = "1"
      ; if so, then write out text about that
      DrawText(0,0,"You entered a 1!  Press any key to exit.")
    Else
      ; if not, then just tell them they didn't enter a 1
      DrawText(0,0,"You did NOT enter a 1! Press any key to exit.")
    EndIf
StopDrawing()

FlipBuffers()     ; flip the buffer to show the user the result

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End   ; end the program
```

Notice specifically how we can control what our program does by using the IF…ELSE 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 *ShieldsOn* 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
   ; Write "Shields On!" at the top of the screen
   DrawText(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.

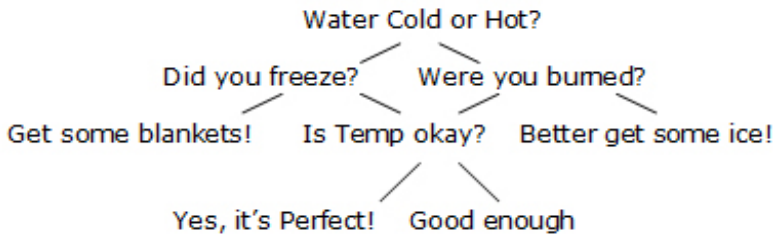
Also note that PureBasic does not allow the use of a THEN portion of the IF… ELSE…ENDIF construct. Some BASIC languages offer, and some require, the use of THEN, but PB does not.

At the end of every IF… ELSE…ENDIF construct, though, you *must* put the ENDIF. This is the only way that PureBasic 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.

You may be wondering why I added a variable in this code called *ClearColor* and I assigned the **RGB** value of 0,0,0 to it. As you progress into games you will be using that **ClearScreen** almost every drawing iteration. So if you call the **RGB** function every time, that slows things down. Not substantially, but as programmers we always try to be efficient where possible. Since I know that I'm going to always clear my screen to **RGB(0,0,0)**, I don't need to call that every time I call **ClearScreen**. I just call it once at the top of my program and then use the resultant value in my **ClearScreen** calls. If you think in terms of 30 frames per second, that means I'll be clearing my screen 30 times a second. That translates to ***not*** calling **RGB** 30 times a second! Every time you relieve your program from having to make unnecessary calls, you can squeeze out just a little more processing power.

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… ELSE…ENDIF constructs. Here is the code:

```
; if the water is hot
If WaterHot  = 1
   ; see if the user got burned
   If WereYouBurned  = 1
      DrawText(0,0,"Better get some ice!")
   Else
      ; is the temp okay?
      If SoTempIsGood  = 1
         DrawText(0,0,"Good enough!")
      Else
         DrawText(0,0,"Yes, it's perfect!")
      EndIf ; end of If SoTempIsGood
   EndIf ; end of If WereYouBurned
Else
   ; did the user freeze?
   If DidYouFreeze  = 1
      DrawText(0,0,"Better get a blanket!")
   Else
      ; is the temp okay?
      If SoTempIsGood  = 1
         DrawText(0,0,"Good enough!")
      Else
         DrawText(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… ELSE…ENDIF section, but 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? Hopefully now you understand the need for decision trees and nested IF's.

ElseIf Statement

One way to help avoid too much nesting is to use an ELSEIF. As opposed to creating a completely new IF block, ELSEIF allows you to keep within the main IF block while still giving the ability to check *else* conditions.

Here is and example that uses the ELSEIF layout instead of a bunch of embedded IF statements.

```
If KeyValue = LeftArrow
   DrawText(0,0,"You hit the left arrow!")
ElseIf KeyValue = RightArrow
    DrawText(0,0,"You hit the right arrow!")
ElseIf KeyValue = UpArrow
     DrawText(0,0,"You hit the up arrow!")
ElseIf KeyValue = DownArrow
     DrawText(0,0,"You hit the down arrow!")
Else
      DrawText(0,0,"You did not hit an arrow key!")
Endif
```

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 PB 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
  If PlayerHit  = 1 And PlayerCrouched  = 1
     ; otherwise, just take away 1 point
    PlayerShields  = PlayerShields  – 1
  EndIf
EndIf
```

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

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 PureBasic 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? Use 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
        DrawText(0,0,"You hit the left arrow!")
    Case RightArrow
        DrawText(0,0,"You hit the right arrow!")
    Case UpArrow
        DrawText(0,0,"You hit the up arrow!")
    Case DownArrow
        DrawText(0,0,"You hit the down arrow!")
     Default
        DrawText(0,0,"You did not hit an arrow key!")
EndSelect
```

Now, compare that with the IF method:

```
If KeyValue = LeftArrow
   DrawText(0,0,"You hit the left arrow!")
Else
   If KeyValue = RightArrow
     DrawText(0,0,"You hit the right arrow!")
   Else
     If KeyValue = UpArrow
       DrawText(0,0,"You hit the up arrow!")
     Else
       If KeyValue = DownArrow
          DrawText(0,0,"You hit the down arrow!")
       Else
          DrawText(0,0,"You did not hit an arrow key!")
       EndIf
     Endif
   Endif
Endif
```

Or, compare using the ELSEIF method:

```
If KeyValue = LeftArrow
   DrawText(0,0,"You hit the left arrow!")
```

```
ElseIf KeyValue = RightArrow
   DrawText(0,0,"You hit the right arrow!")
ElseIf KeyValue = UpArrow
    DrawText(0,0,"You hit the up arrow!")
ElseIf KeyValue = DownArrow
    DrawText(0,0,"You hit the down arrow!")
Else
     DrawText(0,0,"You did not hit an arrow key!")
Endif
```

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

Aside from clarity in your coding, the SELECT command allow for better optimization when a single value is to be tested several times. Optimization is always a good thing!

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 real 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 PureBasic 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 four types of loops available to us in PureBasic:

- For…Next
- ForEach…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.

```
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
DisplaySprite(Asteroid_Image,0,0)
```

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

```
For Images  = 0 To 9
    DisplaySprite(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. Remember, 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 "PureBasic" 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 PureBasic 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 a variable called *TextY* and we'll add 16 to it each time the loop iterates. This will tell PureBasic where we want each line of text displayed.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Input Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

ClearScreen(ClearColor)  ; clear the output buffer to black

TextY = 0        ; initialize our Y position to 0

; Display text to the screen asking the user for input
StartDrawing(ScreenOutput())

    ; Use a FOR...NEXT loop to put up our text 10 times
    For Rows.b = 0 To 9

        ; Write out the text
        DrawText(0,TextY,"PureBasic")

        ; update the next Y position to draw to
        TextY = TextY + 16
    Next

    ; put a little message on the screen for the user
    DrawText(0,400,"Press any key to exit.")
StopDrawing()

FlipBuffers()    ; flip the buffers to show the user the request

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End   ; end the program
```

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
    DrawSprite(Asteroid_Image,0,0)
Next
```

Notice that our initial value is greater than our goal value. If you guessed that PB 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 PureBasic 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:

```
TextY = 0
For Images  = 5 To 1 Step -1
    DrawText(0,TextY,Str(Images ))
    TextY = TextY + 10
Next
```

That "Step –1" piece will inform PureBasic 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.

```
TextY = 0
For Images  = 0 To 100 Step 10
    DrawText(0,TextY,Str(Images ))
    TextY = TextY + 10
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.

```
#Value = 10
TextY = 0

For Images  = 0 To 100 Step #Value
     DrawText(0,TextY,Str(Images ))
     TextY = TextY + 10
Next
```

Placing a "#" before the name makes the area of memory set aside a constant. This means that you can't change the value assigned to it (well, you could, but it'd be rather tricky and a bit more involved than we're going to get).

You cannot use a variable in conjunction with the STEP command. 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
TextY = 0

While Images  <= 9
      DrawText(0,TextY,Str(Images ))
      Images  = Images  + 1
      TextY = TextY + 10
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 that I've seen in games 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, PureBasic!" until the user presses a key.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Input Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; check the keyboard to see if any key has been hit
ExamineKeyboard()

; While the user has NOT hit a key
While  KeyboardReleased(#PB_Key_All) = 0
     ClearScreen(ClearColor)  ; clear the output buffer to black

     ; write out our text
     StartDrawing(ScreenOutput())
        DrawText(270,240,"Hello, PureBasic!")
     StopDrawing()

     FlipBuffers() ; flip the buffers to show the user

     Delay(100)    ; wait for 100 milliseconds

     ClearScreen(clearColor)  ; clear the output buffer to black

     FlipBuffers() ; flip the buffers to show the user

     Delay(100)    ; wait for 100 milliseconds

     ; check the keyboard to see if any key has been hit
     ExamineKeyboard()
Wend

End   ; end of program
```

See how the WHILE…WEND loop continues to roll, unaffected, until the user presses a key? 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.

Again, the most common use I've seen 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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Input Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; set the ExitCondition to 0
ExitCondition.b = 0

; While the ExitCondition is 0
While ExitCondition.b = 0
    ClearScreen(ClearColor)  ; clear the output buffer to black

    ; write out our text
    StartDrawing(ScreenOutput())
        DrawText(270,240,"Hello, PureBasic!")
    StopDrawing()

    FlipBuffers() ; flip the buffers to show the user

    Delay(100)    ; wait for 100 milliseconds

    ClearScreen(ClearColor)  ; clear the output buffer to black

    FlipBuffers() ; flip the buffers to show the user

    Delay(100)    ; wait for 100 milliseconds

    ; check the keyboard to see if any key has been hit
    ExamineKeyboard()

    If KeyboardInkey()
      ClearScreen(ClearColor)  ; clear the output buffer to black

      ; put up the question to the user
      StartDrawing(ScreenOutput())
          DrawText(0,0,"Do you really want to quit (Y/N)?")
      StopDrawing()

      FlipBuffers()  ; show the question to the user
```

```
        ; set the Answers.s variable to be a blank
        Answer.s = ""

        ; Wait for a keypress before going any further
        While Answer.s = ""
            ExamineKeyboard()
            Answer.s = KeyboardInkey()
        Wend

        ; if the answer is "Y" (or "y"), then set the ExitCondition to 1
        If Answer.s = "Y" Or Answer.s = "y"
          ExitCondition.b = 1
        EndIf
      EndIf

Wend  ; end of While loop

End   ; end of program
```

Please note that the KeyboardInkey command will not process certain keys, such as *Alt*, *Ctrl*, *Shift*, etc. For these you'll need the KeyboardPushed command, which we will be working with later.

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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Input Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; set the ExitCondition to 0
ExitCondition.b = 0

; Repeat until the ExitCondition is 1
Repeat
    ClearScreen(ClearColor)  ; clear the output buffer to black

    ; write out our text
    StartDrawing(ScreenOutput())
        DrawText(270,240,"Hello, PureBasic!")
```

```
    StopDrawing()

    FlipBuffers() ; flip the buffers to show the user

    Delay(100)    ; wait for 100 milliseconds

    ClearScreen(ClearColor)  ; clear the output buffer to black

    FlipBuffers() ; flip the buffers to show the user

    Delay(100)    ; wait for 100 milliseconds

    ; check the keyboard to see if any key has been hit
    ExamineKeyboard()
    If KeyboardInkey()
      ClearScreen(ClearColor)  ; clear the output buffer to black

      ; put up the question to the user
      StartDrawing(ScreenOutput())
          DrawText(0,0,"Do you really want to quit (Y/N)?")
      StopDrawing()

      FlipBuffers()  ; show the question to the user

      ; set the Answers.s variable to be a blank
      Answer.s = ""

      ; Wait for a keypress before going any further
      While Answer.s = ""
            ExamineKeyboard()
            Answer.s = KeyboardInkey()
      Wend

      ; if the answer is "Y" (or "y"), then set the ExitCondition to 1
      If Answer.s = "Y" Or Answer.s = "y"
        ExitCondition.b = 1
      EndIf
    EndIf

Until ExitCondition.b = 1    ; End of Repeat loop

End   ; end of program
```

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 BREAK command. BREAK breaks out of a loop and places the execution at the point directly after the loop. You can also put an argument after BREAK to inform it how many loops (assuming there are nested loops) you wish it to break from.

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 "Name1.s," "Name2.s," etc., or we could use an array.

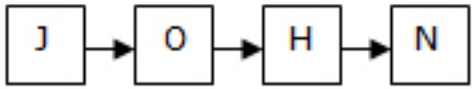
So, we could use the individual strings and have:

```
Name1.s ="John"
Name2.s ="Lorelei"
Name3.s ="Fred"
Name4.s ="Betty"
Name5.s ="George"
```

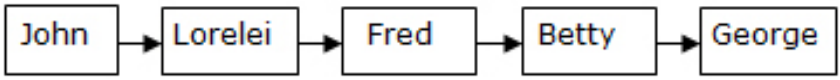
This format would setup the individual strings and we would have to recall the variable name in full when referencing a particular player. If, however, we used an array we would only need to know the array name and the location of the player within the array. Here is an example of what that would look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

```
NameArray.s(0) = "John"
NameArray.s(1) = "Lorelei"
NameArray.s(2) = "Fred"
NameArray.s(3) = "Betty"
NameArray.s(4) = "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 PureBasic 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 PB 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.s(4)
```

That's it. In that one statement, we've told PB to reserve enough memory to hold five pieces of data of type string. From here PB will carve out a memory chunk for us and get it ready to hold any string data we want to store in there. Why five elements when we have the number 4 in there? Because PureBasic starts counting from 0, remember. So if you count 0…1…2…3…4 what you will actually have is 5 elements. An easy way to remember this is knowing that whatever number you place in the array definition, PB will reserve that number + 1. Thus, in our example, it would be 4+1, or 5, elements.

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.s(0) = “John”
NameArray.s(1) = “Lorelei”
NameArray.s(2) = “Fred”
NameArray.s(3) = “Betty”
NameArray.s(4) = “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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Array Test")=0
    MessageRequester("Error!", "Unable to Initialize Environment", ↵
    → #PB_MessageRequester_OK)
  End
EndIf

; Dimension our NameArray
Dim NameArray.s(4)

; Assign our values to the array
NameArray.s(0) = "John"
NameArray.s(1) = "Lorelei"
NameArray.s(2) = "Fred"
NameArray.s(3) = "Betty"
NameArray.s(4) = "George"

TextY = 0   ; initialize the starting Y position
StartDrawing(ScreenOutput())
    ; Loop through the array and print out the values
    For Names = 0 To 4
        DrawText(0,TextY,NameArray(Names))
        TextY = TextY + 16
    Next

    ; display a message so the user knows how to exit
    DrawText(0,400,"Press any key to exit")
StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
```

```
Until KeyboardReleased(#PB_Key_All)

End  ; end the program
```

Arrays are not limited to string values, of course. You can also set them up as Byte, Word, Long, Float, or as a structure. We have not touched on structures yet, but will get into much detail on them soon.

You treat Byte, Word, Long, and Float types exactly as you do String, with the only exception being the definition.

```
Dim NameArray.s(4)       ; creates an array of strings
Dim WeaponArray.b(4)     ; creates an array of integers
Dim MissionArray.w(4)    ; creates an array of integers
Dim ScoreArray.l(4)      ; creates an array of integers
Dim PrecisionArray.f(4)  ; creates an array of floats
```

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

```
Dim Name.s(4)       ; creates an array of strings
Dim Weapon.b(4)     ; creates an array of integers
Dim Mission.w(4)    ; creates an array of integers
Dim Score.l(4)      ; creates an array of integers
Dim Precision.f(4)  ; 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:

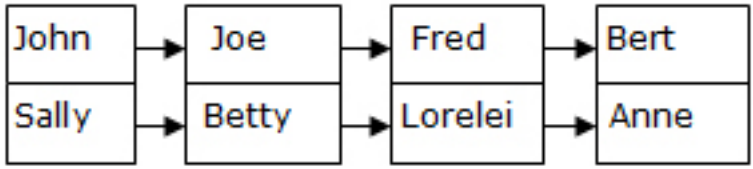
John -> Joe -> Fred -> Bert
Sally-> Betty -> Lorelei -> Anne
(Figure 6.1)

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 PureBasic sees the array in memory. PB 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 PB, it looks like this:

PureBasic 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 PureBasic is already prepared to help you handle most of these difficulties.

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

```
Dim NameArray.s(1,3)
```

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

```
NameArray.s(0,0) = "John"
NameArray.s(1,0) = "Sally"
```

This means that "John" will now sit in row 0, column 0, and that "Sally" will be in row 1, column 0. Remember that PB counts from 0, not 1.

Accessing the array is a bit trickier 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 0 and then move on to row 1. 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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Array Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

; Dimension our NameArray
Dim NameArray.s(1,3)

; Assign our values to the array
NameArray(0,0) = "John"
NameArray(0,1) = "Joe"
NameArray(0,2) = "Fred"
NameArray(0,3) = "George"
NameArray(1,0) = "Sally"
NameArray(1,1) = "Betty"
NameArray(1,2) = "Lorelei"
NameArray(1,3) = "Anne"

TextY = 0   ; initialize the starting Y position
StartDrawing(ScreenOutput())
   ; Loop through the array and print out the values
   For NamesRow = 0 To 1
      For NamesColumn = 0 To 3
         DrawText(0,TextY,NameArray(NamesRow,NamesColumn))
         TextY = TextY + 16
      Next
   Next

   ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to exit")
StopDrawing()

FlipBuffers()  ; show the output to the user
```

```
; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End  ; end the program
```

If you type in that program and run it, you'll see all the names listed starting with the first row. Try altering the *NameRow* 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.s(4,1,2)
```

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 as a 3D array, being X,Y, and Z as the three values. There are many ways to visualize this concept.

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.s(4)
…load in ship name data and print…
Dim NameArray.s(2)
…load in animal name data and print…
Dim NameArray.s(4)
…load in ship name data and print…
Dim NameArray.s(2)
```

```
…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 PureBasic will keep track of array information.

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

```
SizeOfArray = 4
Dim NameArray.s(SizeOfArray)
…load in ship name data and print…

SizeOfArray = SizeOfArray - 2

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

Loading Data Values into an Array

There is a neat little ability in PureBasic 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 PB'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 PB everything on the line is to be taken as information for later processing.
- Restore: Tells PB where in the program it should start reading data values. It's based on a label that you create.
- Read: This command tells PB 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:

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

The first thing to note is:

```
DataSection
```

This line informs PureBasic that what follows will be data used for your program.

```
NameData:
```

This is the label of the section I'm using for the data being stored. You don't have to call it *NameData*. You can call it pretty much whatever you want, just remember to be mildly descriptive so it's not ambiguous.

This label will be used with the Restore command. Make sure you put a colon (":") at the end of the name. If you don't have the colon in there, PureBasic will not know what the intention of the line is and your Restore command will not be able to locate the label, and you'll get an error during compile.

Then we have our group of DATA statements:

```
Data.s "John","Joe","Mark","George"
Data.s "Sally","Betty","Lorelei","Anne"
Data.s "Fido","Spot","Killer","Tank"
Data.s "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.

```
EndDataSection
```

And finally, in order to let PB know when you're finished you use the EndDataSection keyword.

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. The following code with read the DATA elements into the array:

```
; Dimension our NameArray
Dim NameArray.s(3,3)

; Go to the front of the data lines for the NameData
Restore NameData

; loop through the data and READ to the array
For NameType  = 0 To 3
   For Names  = 0 To 3
      Read.s NameArray.s(NameType ,Names )
   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 colon (":") 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. Note that the Read command now requires that you pass the data type you are looking to read. To use the Read command, use the following layout:

```
DataSection
   NameData:
      Data.s "John","Joe","Mark","George","Sally","Betty","Lorelei", "Anne"
EndDataSection
```

…which is the same thing as this:

```
DataSection
   NameData:
      Data.s "John","Joe","Mark","George"
      Data.s "Sally","Betty","Lorelei", "Anne"
EndDataSection
```

The formatting is for the programmer's benefit, not PB'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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Array Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

; Dimension our NameArray
Dim NameArray.s(3,3)

; Go to the front of the data lines for the NameData
Restore NameData

; loop through the data and READ to the array
For NameType = 0 To 3
   For Names = 0 To 3
      Read.s NameArray(NameType,Names)
   Next
Next

; Set up the vertical control variable
 TextY = 0

 StartDrawing(ScreenOutput())
    ; Loop through the array we READ and print it out
    For NameType = 0 To 3
       For Names = 0 To 3
          DrawText(0,TextY,NameArray(NameType,Names))
          TextY = TextY + 16
       Next
    Next

   ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to exit")
 StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)
```

```
End  ; end the program

; Here is our data area
DataSection
   NameData:
      Data.s "John","Joe","Mark","George"
      Data.s "Sally","Betty","Lorelei","Anne"
      Data.s "Fido","Spot","Killer","Tank"
      Data.s "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
EndDataSection
```

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

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Array Test")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

; Dimension our Arrays
Dim NameArray.s(1,3)
Dim ShipNameArray.s(3)

; Go to the front of the data lines for the NameData
Restore NameData

; loop through the data and READ to the array
For NameType = 0 To 1
   For Names = 0 To 3
      Read.s NameArray(NameType,Names)
   Next
Next

; Go to the front of the data lines for the ShipNameData
Restore ShipNameData

; loop through the data and READ to the array
For Names = 0 To 3
   Read.s ShipNameArray(Names)
Next

; Set up the vertical control variable
 TextY = 0

 StartDrawing(ScreenOutput())
    ; Put up a header for the list
```

```
    DrawText(0,TextY,"Names:")
    TextY = TextY + 16

    ; Loop through the Name array we READ and print it out
    For NameType = 0 To 1
       For Names = 0 To 3
          DrawText(0,TextY,NameArray(NameType,Names))
          TextY = TextY + 16
       Next
    Next

    ; put one additional space in between lists
    TextY = TextY + 16

    ; put up a header for the 2nd list
    DrawText(0,TextY,"Ship Names:")
    TextY = TextY + 16

    ; Loop through the Ship Name array we READ and print it out
    For Names = 0 To 3
       DrawText(0,TextY,ShipNameArray(Names))
       TextY = TextY + 16
    Next

   ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to exit")
StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End  ; end the program

; Here is our data area
DataSection
   NameData:
      Data.s "John","Joe","Mark","George"
      Data.s "Sally","Betty","Lorelei","Anne"

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

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.

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.s "John","Joe","Mark","George"
 Data.s "Sally","Betty","Lorelei","Anne"
 Data.s "Fido","Spot","Killer","Tank"
 Data.s "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
 Data.s "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… ELSE…ENDIF commands. Here's the example code:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"Array Test")=0
    MessageRequester("Error!", "Unable to Initialize Environment", ↵
    → #PB_MessageRequester_OK)
   End
EndIf

; Set up the vertical control variable
TextY = 0

; set up our flag value for seeing if we're done or not
FinishedListing = 0

Restore NameData

StartDrawing(ScreenOutput())
    ; while we're NOT Finished
    While FinishedListing = 0
        ; read a Name from the data segment
```

```
        Read.s Name.s
        ; if that Name = STOP, then we're done
        If Name = "STOP"
          FinishedListing = 1
        Else
          ; otherwise, show the Name we read
          DrawText(0,TextY,Name)
          TextY = TextY + 16
        EndIf
    Wend

    ; display a message so the user knows how to exit
    DrawText(0,400,"Press any key to exit")
StopDrawing()

FlipBuffers() ;show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End  ; end the program

; Here is our data area
DataSection
   NameData:
      Data.s "John","Joe","Mark","George"
      Data.s "Sally","Betty","Lorelei","Anne"
      Data.s "Fido","Spot","Killer","Tank"
      Data.s "Millennium Hawk","Tea Fighter","Zap-Wing","Dead Star"
      Data.s "STOP"
EndDataSection
```

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? PB 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, PureBasic doesn't care.

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 structure, and we'll touch again on how to load variable length values using the DATA commands.

Chapter 7: Understanding/Using Structures

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 a Byte, 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 Structure, 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. Structures only use what's needed and nothing more.

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

```
Structure PersonalInfo
    Name.s          ; name of the person
    Age.b           ; age of the person
    GPAPercent.f    ; Grade Point Average of the person
EndStructure
```

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

Note that we don't assign any values during the building of our Structure. This is because our format is merely a blueprint for the data that can be held by *PersonalInfo*. To actually store data, we must first initialize the Structure as a Variable, an Array, or a List.

Arrays of Structures

Since we just finished discussing Arrays, let's start with that method.

```
Dim People.PersonalInfo(99)
```

The above line will create an Array of 100 elements (remember, we count from 0, not 1!) for our Structure. This is exactly like creating any other Array, with the exception that we're using the already defined structure name as part of the declaration.

Obviously we won't be able to use the same type of notation for assigning values that we use with a Byte Array, so how do we do it? Consider the following:

```
People(0)\Name = "John"
People(0)\Age = 36
People(0)\GPAPercent = 3.75
People(1)\Name = "Lorelei"
People(1)\Age = 36
People(1)\GPAPercent = 4
```

As you can see, with the addition of the "\" and the name of the variable within the structure, we can assign the varied values with ease! Retrieving the values is just as simple:

```
StudentName = People(0)\Name
StudentAge = People(0)\Age
StudentGPA = People(0)\GPAPercent
```

See how easy that is?

One thing you may have noticed is that I'm not bothering to put the variable type at the end of each data field. This is because PB already knows what type field is since it's in the blueprint. The following *used to be* an error:

```
People(0)\Name.s = "John"
```

But it appears in version 4.61 that PB will accept the field with or without the variable type and not complain either way.

Now let's take a look at how powerful it can be in a game situation.

The following code will create a Structure that houses information about different space ships. We'll be using Data statements in this example, which we learned about in the last chapter, and we'll be creating two ship types with some differing information.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   →#PB_MessageRequester_OK)
   End
EndIf

; setup our structure
```

```
Structure Ships
        Name.s              ; name of this ship
        LaserPower.b          ; 1-20 points per hit
        Armor.b             ; 75-125 points depending on the ship
        ShieldPower.b         ; 50-100 points added on to Armor
        TopSpeed.b            ; 2 - 4 depending on ship type
EndStructure

; dimension our Fighters
Dim Fighter.Ships(1)

; go to the ShipNameData section
Restore ShipNameData

; use a standard array looping style
For i = 0 To 1
    ; read the data
    Read.s Fighter.Ships(i)\Name
 Next

; go to the ShipSpecsData section
Restore ShipSpecsData

; use a standard array looping style
For i = 0 To 1
   ; read the data
   Read.b Fighter.Ships(i)\LaserPower
   Read.b Fighter.Ships(i)\Armor
   Read.b Fighter.Ships(i)\ShieldPower
   Read.b Fighter.Ships(i)\TopSpeed
Next

; Set up the vertical control variable
TextY = 0

StartDrawing(ScreenOutput())
   ; Step through the Ships array and print
   For i = 0 To 1
      shipText.s = "Ship Name: " + Fighter.Ships(i)\Name
      DrawText(0,TextY,shipText.s)
      TextY = TextY + 16

      shipText.s = "Laser Power: " + Str(Fighter.Ships(i)\LaserPower)
      DrawText(0,TextY,shipText.s)
      TextY = TextY + 16

      shipText.s = "Armor: " + Str(Fighter.Ships(i)\Armor)
      DrawText(0,TextY,shipText.s)
      TextY = TextY + 16

      shipText.s = "Shield Power: " + Str(Fighter.Ships(i)\ShieldPower)
```

```
      DrawText(0,TextY,shipText.s)
      TextY = TextY + 16

      shipText.s = "Top Speed: " + Str(Fighter.Ships(i)\TopSpeed)
      DrawText(0,TextY,shipText.s)
      TextY = TextY + 32
   Next

    ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to exit")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End  ; end the program

; Here is our data area
DataSection
   ShipNameData:
      Data.s "Kliazian Raptor", "Weltic Cruiser"

   ShipSpecsData:
      Data.b 15,125,50,3
      Data.b 20,100,75,4
EndDataSection
```

Most of this should already be familiar with you, so I won't step through it all. But the pieces that are a bit new are the Structure creation and how I'm calling the Read.

```
; setup our structure
Structure Ships
        Name.s              ; name of this ship
        LaserPower.b        ; 1-20 points per hit
        Armor.b             ; 75-125 points depending on the ship
        ShieldPower.b       ; 50-100 points added on to Armor
        TopSpeed.b          ; 2 - 4 depending on ship type
EndStructure
```

A quick glance at the above shows that we're setting up pretty basic information on a ship. If we were to really build this ship out we'd need to think about additional things, such as: cargo space, number of

missiles, weapons allowed, weapons available, turning speed, braking speed, landing bays (for smaller ships to be housed), and so on. This list can actually get rather large and have many different item types, which is why the use of a Structure is ideal!

```
; use a standard array looping style
For i = 0 To 1
    ; read the data
    Read Fighter.Ships(i)\LaserPower
    Read Fighter.Ships(i)\Armor
    Read Fighter.Ships(i)\ShieldPower
    Read Fighter.Ships(i)\TopSpeed
 Next
```

We're doing the exact same thing with the Read here that we did in the last chapter, except that here we store the values directly into the Structure. It's quite easy, isn't it?

Arrays within Structures

One of the things you'll likely want to do is have the ability to incorporate an Array *inside* of a Structure. There are many reasons for this, but for a quick example let's just say that we want to keep track of the number of missiles our ship currently has, and also what their classification is. Certainly we could do this with just adding two additional fields, but play along with me so you can learn this method.

Taking the last example, let's edit the Structure a bit:

```
; setup our structure
Structure Ships
        Name.s          ; name of this ship
        LaserPower.b    ; 1-20 points per hit
        Armor.b         ; 75-125 points depending on the ship
        ShieldPower.b   ; 50-100 points added on to Armor
        TopSpeed.b      ; 2 - 4 depending on ship type
        Missiles.b[2]   ; Missile array of 2 elements
EndStructure
```

Notice that the only difference from our original example is the *Missiles.b[2]* line. This line tells PureBasic that we want to create an Array within the Structure that can hold 2 elements.

Now you may be thinking that the Array will hold 3 elements, as described in the last chapter, but Arrays inside of a Structure do not behave exactly the same as those defined outside of one. This is due to this type of an Array being a *Static* array (denoted by the [ ] and the lack of a DIM statement). In Structures a static Array doesn't behave like the normal BASIC array (defined using Dim). This has to do with the

handling of advanced API (Applications Programmer Interface) porting to the C/C++ language. Huh? I know that sounds really advanced. Basically note that PureBasic is a very powerful language that allows you to interface with other languages and, because of this, therefore PB has to make sure it stays compatible with the needs of those other languages and their formatting.

What this really means to us thought is that a Structure (static) Array of [2] will allocate an Array from 0 to 1, where a DIM (non-static) Array of (2) will allocate an array from 0 to 2.

To read in the elements from our DATA statement is a snap. Just do the following:

```
; go to the ShipNameData section
Restore ShipNameData

Read Fighter.Ships(0)\Missiles[0]
Read Fighter.Ships(0)\Missiles[1]

Read Fighter.Ships(1)\Missiles[0]
Read Fighter.Ships(1)\Missiles[1]
```

Check out the example code in the Chapter 7 directory under the file name "ex7-2ArrayOfStructuresArray.pb" to see this in action.

Basic Structure Lists

For a more dynamic approach to using a Structure, we will turn to the concept of *Lists*. A List is similar to an Array in that each element is in a procession of elements in memory. The primary difference is that a List is dynamic. You can add and delete elements in a List on the fly, which means that you can control precisely how many elements are allocated.

One way to visualize this is email. When you receive a piece of email, the item is placed in a list of emails among all the other emails you've received. Pretend you have 10 emails in your queue. You read item number 5 and then delete it. The list just changed dynamically. You still have the remaining 9 emails in your mailbox, and if another one arrives, you'll have 10 again. This is similar to how a List works in PB.

For an example of this let's pretend that you are launching missiles from your ship. You are allowed to launch a maximum of 10 missiles at a time. Each missile can have a speed anywhere from 20-100 units per second, and can travel anywhere between 500-2500 units before it fizzles out.

The first step will be to create a Structure for our missiles.

```
Structure Missiles
        Speed.w             ; Missile Speed (5-20)
        MaxDistance.w       ; How far can it go (500-5000)
        CurrentDistance.w   ; How far have we gone?
EndStructure
```

In this Structure we're including how fast the missile will be able to travel, how far it can travel, and then putting in a field that let's us keep track of how far it has already gone.

```
NewList Missile.Missiles()
```

The NewList command instructs PB to declare a new List. In this instance, I elected to name this List *Missile* since each element will contain data on a single missile. You may call this whatever you would like. Note that NewList will automatically make the List global, so there is no need to do any sort of scope on the definition. Also, whenever referencing the Structure from this point on, be sure to do it using the List name (*Missile*) and not the Structure name (*Missiles)*.

After you've declared a List, you will need a way to add elements to it. Consider the following snippet:

```
If AddElement(Missile()) <> 0
   Missile()\Speed = Random(15) + 5
   Missile()\MaxDistance = Random(4500) + 500
   Missile()\CurrentDistance = 0
Else
   MessageRequester("Error!", "Can't allocate memory for new element", ↵
    → #PB_MessageRequester_OK)
EndIf
```

The first line informs PB to allocate a portion of memory for a single element of the *Missile()* List. As long as the value returned by AddElement is NOT zero(0), then we can continue on. If, however, PB returns zero it means that it could not allocate the needed memory for the element.

Next we start assigning values to our List. This is done by using the List name following by the () identifier, a "\" character, and then the field within the Structure we wish to populate. If you were to put ***Missile\Speed = 5*** you would receive an error. You must include the () after the List name.

A quick note about the Random command: I'm using this command to put in a random value every time the player presses a key. This command returns a value between 0 and the number you put as the

argument. If you want to ensure Random returns a number between 5 and 20, say, you will need to do a little addition against the value Random returns. In the example of the *Speed* field I'm doing just that. I ask Random turn return a value between 0 and 15, and then add 5 to whatever value is returned. So if Random returns 0, 0+5 = 5, and if it returns 10, 10+5 = 15. You'll find that in games development you'll almost always have uses for random values, so keep this command at the ready.

Finally, if you look at the *Else* portion of the code you will see the MessageRequester command. This command is useful in reporting errors to players of your game. In this instance, I'm just informing the user that the program was unable to successfully allocate memory using the AddElement command. If you enter this command in your PB IDE and press F1 while it is highlighted, you will see all the options available with it.

One of the things you'll want to be able to know is how many elements are in a list at any given time. This data is acquired using the CountList command (**NOTE:** *use the* ***ListSize*** *command in PureBasic 5 as CountList has been deprecated*). The following line will place the number of elements currently in the list into the variable *Value*.

```
Value.w = CountList(Missile())
```

Next we need a way to run through all of our elements, display some information to the user, and update the elements as well.

```
ForEach Missile()
    MissileText.s = "Missile Speed: " + Str(Missile()\Speed) + ↵
  → ", Max Distance: " + Str(Missile()\MaxDistance) + ↵
  → ", Current Distance: " + Str(Missile()\CurrentDistance)
    DrawText(0,TextY,MissileText.s)
    TextY = TextY + 16

    ; now add the speed of the missile to its current distance
    Missile()\CurrentDistance = Missile()\CurrentDistance +  Missile()\Speed

    ; if the distance is passed the maximum distance it can travel
    If Missile()\CurrentDistance > Missile()\MaxDistance
        ; delete the missile
        DeleteElement(Missile())
    EndIf
Next
```

The ForEach command acts similarly to a For command, except that it's specifically reserved for use with Lists. It starts at the beginning of the List (known as the *head*) and traverses the list all the way until the end

(or the *tail*). Any call you make to the List in this loop will affect only the element that ForEach is currently pointing at.

At the top you'll see a very large assignment of data to *MissileText*. All I'm doing here is putting together a string of information so the player can see the information on each launched missile. Then I draw it up and increment the Y counter so the lines don't overwrite each other.

Immediately following I update the position of the current missile by adding its cruising speed to its current location. Then I check to see if it has passed it's maximum traveling distance. If it has, I want to remove this missile from the List. This is accomplished using the **DeleteElement** command. If you call that command with the *Missile()* argument, the element will be removed from your List.

Here is the complete source to our basic List Structure example:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
   End
EndIf

ClearColor = RGB(0,0,0)

; setup our structure
Structure Missiles
        Speed.w          ; Missile Speed (5-20)
        MaxDistance.w    ; How far can it go (500-5000)
        CurrentDistance.w  ; How far have we gone?
EndStructure

; setup a new list for the missiles
NewList Missile.Missiles()

; set the ExitCondition to 0
ExitCondition = 0

; While the ExitCondition is 0
While ExitCondition = 0

  ClearScreen(ClearColor)        ; clear the screen to black

  ; Set up the vertical control variable
  TextY = 0

  StartDrawing(ScreenOutput())

      ; Step through the missile list and print info
```

```
    ForEach Missile()
      MissileText.s = "Missile Speed: " + Str(Missile()\Speed) + ↵
→ ", Max Distance: " + Str(Missile()\MaxDistance) + ↵
→ ", Current Distance: " + Str(Missile()\CurrentDistance)
      DrawText(0,TextY,MissileText)
      TextY = TextY + 16

      ; now add the speed of the missile to its current distance
      Missile()\CurrentDistance = Missile()\CurrentDistance + Missile()\Speed
      ; if the distance is passed the maximum distance it can travel
      If Missile()\CurrentDistance > Missile()\MaxDistance
        ; delete the missile
        DeleteElement(Missile())
      EndIf
    Next

    ; display a message so the user knows how to fire/exit
    DrawText(0,400,"Press Spacebar to Fire -- ESC to Exit")

StopDrawing()

; see if any key activity has happened
ExamineKeyboard()

; to make sure that we don't just roll-up the missiles
; too quickly.  Make sure the player has to let go of
; the fire key before firing again
If KeyboardReleased(#PB_Key_Space)
  ; if the key was released, reset our Fired flag
  Fired = 0
EndIf

; If the player hits the fire key
If KeyboardPushed(#PB_Key_Space) And Fired = 0
  ; make sure we don't have more than 20 missiles out already
  If CountList(Missile()) < 19
    ; add the element of a new missile and populate it
    If AddElement(Missile()) <> 0
      Missile()\Speed = Random(15) + 5
      Missile()\MaxDistance = Random(4500) + 500
      Missile()\CurrentDistance = 0
    Else
      MessageRequester("Error!", "Unable to allocate memory for ↵
→ new element", #PB_MessageRequester_Ok)
    EndIf
  EndIf
  ; set our flag to show that the missile has been fired
  ; this is so we can make sure the player releases the key
  Fired = 1
EndIf
```

```
  ; if the player hits ESC, set our ExitCondition and quit
  If KeyboardPushed(#PB_Key_Escape)
    ExitCondition = 1
  EndIf

  FlipBuffers()  ; show the output to the user

Wend

End  ; end the program
```

Advanced Operations – Extending Structures

One of the coolest things about Structures is the ability to extend their abilities without having to constantly change their core.

Imagine that you have a Structure that you're using for a ship, but you start thinking that while there are some commonalities between all ships, there are also some major differences. For example, a freighter is going to be more concerned with cargo space than it will be with weapons. A fighter ship will be more concerned with speed and dexterity than it will be with cargo space. And so on. But they both need to have an engine, braking abilities, communications, some form of protection (weapons, shields, armor), etc. So to create two entirely separate Structures with essentially the same data is redundant.

Extending a Structure requires using the Extends command, as follows:

```
Structure Ships
        EngineType.w        ; 0=slow, 1=medium, 2=Fast
        WeaponsType.w       ; 0=basic, 1=advanced
        Shields.w           ; 200-400
        Armor.w             ; 200-400
EndStructure

Structure Freighters Extends Ships
        Name.s              ; name of the freigther
        Cargo.l             ; 20000 - 50000 units
EndStructure
```

Here we have a Structure defined that has all of our basic ship elements. Then we have a Structure specifically for freighter-type ships called, amazingly, *Freighters*. What you'll note in the *Freighters* definition though is that it contains "Extends *Ships*." This instructs PB to take all of the fields from the *Ships* Structure and duplicate them inside o of *Freighters*. Now all you have to do is create a List for *Freighters* and you'll be able to reference all the fields in both Structures!

Here is the full code for the example:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
   End
EndIf

ClearColor = RGB(0,0,0)

; setup our structure
Structure Ships
        EngineType.w   ; 0=slow, 1=medium, 2=Fast
        WeaponsType.w  ; 0=basic, 1=advanced
        Shields.w      ; 200-400
        Armor.w        ; 200-400
EndStructure

; setup our freighter structure and inherit all of the structure
; elements from Ships
Structure Freighters Extends Ships
        Name.s         ; name of the freigther
        Cargo.l        ; 20000 - 50000 units
EndStructure

; setup our fighter structure and inherit all of the structure
; elements from Ships
Structure Fighters Extends Ships
        Name.s              ; name of the figther
        SpeedBooster.w  ; 0=level 1 (moderate), 1 = level 2 (Fast)
        Dexterity.w         ; 0=level 1 (moderate), 1 = level 2 (Fast)
EndStructure

; setup a new list for the Freighters
NewList Freighter.Freighters()

; setup a new list for the Fighters
NewList Fighter.Fighters()

;Now let's just populate two Freighters by hand
AddElement(Freighter())
Freighter()\Name = "Big Freighter"
Freighter()\Cargo = 50000
Freighter()\EngineType = 0
Freighter()\WeaponsType = 0
Freighter()\Shields= 200
Freighter()\Armor = 300

AddElement(Freighter())
```

```
Freighter()\Name = "Medium Freighter"
Freighter()\Cargo = 35000
Freighter()\EngineType = 1
Freighter()\WeaponsType = 0
Freighter()\Shields= 250
Freighter()\Armor = 200

;And now let's populate two Fighters by hand
AddElement(Fighter())
Fighter()\Name = "Fast Fighter"
Fighter()\SpeedBooster = 1
Fighter()\Dexterity = 1
Fighter()\EngineType = 2
Fighter()\WeaponsType = 1
Fighter()\Shields= 300
Fighter()\Armor = 200

AddElement(Fighter())
Fighter()\Name = "Mid Fighter"
Fighter()\SpeedBooster = 0
Fighter()\Dexterity = 1
Fighter()\EngineType = 2
Fighter()\WeaponsType = 1
Fighter()\Shields= 400
Fighter()\Armor = 400

ClearScreen(ClearColor)         ; clear the screen to black

; Set up the vertical control variable
TextY = 0

StartDrawing(ScreenOutput())

    DrawText(0,TextY,"FREIGHTERS")
    TextY = TextY + 32

    ; Step through the Freighter list and print info
    ForEach Freighter()

        MissileText.s = "Name: " + Freighter()\Name
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Cargo: " + Str(Freighter()\Cargo)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Engine Type: " + Str(Freighter()\EngineType)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16
```

```
        MissileText.s = "Weapons Type: " + Str(Freighter()\WeaponsType)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Shields: " + Str(Freighter()\Shields)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Armor: " + Str(Freighter()\Armor)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 32
    Next

    ; display a message so the user knows how to fire/exit
    DrawText(0,400,"Press any key to see the Fighters")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

ClearScreen(ClearColor)         ; clear the screen to black

; Set up the vertical control variable
TextY = 0

StartDrawing(ScreenOutput())

    DrawText(0,TextY,"FIGHTERS")
    TextY = TextY + 32

    ; Step through the Fighter list and print info
    ForEach Fighter()

        MissileText.s = "Name: " + Fighter()\Name
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Speed Booster: " + Str(Fighter()\SpeedBooster)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Dexterity: " + Str(Fighter()\Dexterity)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Engine Type: " + Str(Fighter()\EngineType)
```

```
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Weapons Type: " + Str(Fighter()\WeaponsType)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Shields: " + Str(Fighter()\Shields)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Armor: " + Str(Fighter()\Armor)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 32
    Next

    ; display a message so the user knows how to fire/exit
    DrawText(0,400,"Press any key to Exit")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End  ; end the program
```

Advanced Structure Operations – Pointers

One of the most interesting topics in programming is that of pointers. Pointers are pieces of memory setup to *point* at other pieces of memory. The user treats them much like a variable, but the value contained within is a memory location to another variable or data structure.

In the previous section we demonstrated how to build a basic List. What we're going to do now is alter the method used to incorporate Pointers. We get POWER! Remember that with power comes responsibility, though, so make sure you're careful.

It's much more likely that you'll end up with errors using the following method than with the *Basic List* method described prior. This is because the previous method handled all of the memory control in the background. Here you'll be able to directly address the memory locations, so you will want to be careful about each step in the process.

```
Structure Ships
        Name.s              ; Name of the ship
```

```
    WeaponsType.w     ; 0=basic, 1=advanced
    Shields.w         ; 200-400
    Armor.w           ; 200-400
EndStructure
```

Our next move is to define a pointer that we can use to reference the *Ships* Structure completely.

```
Define.Ships *Ship
```

The only thing really new here are the lines containing the * sign. Firstly, note that an * before a variable name denotes that it is a pointer.

```
NewList Fighter.Ships()
```

We still have to initialize our List so don't neglect to do that!

Now that we've defined our pointer and our List, we can add to it.

```
AddElement(Fighter())
If Fighter()
   Fighter()\Name = "Fast Fighter"
   Fighter()\WeaponsType = 1
   Fighter()\Shields= 300
   Fighter()\Armor = 200
EndIf

AddElement(Fighter())
If Fighter()
   Fighter()\Name = "Mid Fighter"
   Fighter()\WeaponsType = 0
   Fighter()\Shields= 400
   Fighter()\Armor = 400
EndIf
```

Yes, we still call good old AddElement, but this time we want the return value, which will be the memory location where this element starts, to be assigned to our *Ship* pointer. If the pointer has a zero(0) value, then AddElement was not successful in allocating the memory. From here we assign values to the List just as before, except we use the *Ship* pointer as our index controller.

To traverse the List, we must first make sure that the pointer is pointing at the proper element. Which, in our case will be the first element:

```
*Ship = FirstElement(Fighter())
```

Then we start by making sure that *Ship* is pointing at something tangible:

```
While *Ship <> 0
```

And, if so, we move in and print the values out.

```
MissileText.s = "Name: " + *Ship\Name
DrawText(0,TextY,MissileText.s)
TextY = TextY + 16
…etc…
```

In order to get to the next element in the list, we must use the *Next* pointer that we setup in our original Structure.

```
*Ship =  NextElement(Fighter())
```

This will then take us back the top of the *While* loop where we can see if the *Ship* pointer is still pointing at something valid or not.

You can also traverse the list in reverse. You should try changing the following code to start at the last element **(LastElement)** in the List and write out the List backwards while using **PreviousElement**.

```
; Initialize the sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or OpenScreen(640,480,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_Ok)
  End
EndIf

ClearColor = RGB(0,0,0)

; setup our structure
Structure Ships
        Name.s          ; Name of the ship
        WeaponsType.w   ; 0=basic, 1=advanced
        Shields.w       ; 200-400
        Armor.w         ; 200-400
EndStructure

; Define a pointer to the Structure
Define.Ships *Ship
```

```
; setup a new list for the Freighters
NewList Fighter.Ships()

;And now let's populate two Fighters by hand, using pointers
AddElement(Fighter())
If Fighter()
  Fighter()\Name = "Fast Fighter"
  Fighter()\WeaponsType = 1
  Fighter()\Shields= 300
  Fighter()\Armor = 200
EndIf

AddElement(Fighter())
If Fighter()
  Fighter()\Name = "Mid Fighter"
  Fighter()\WeaponsType = 0
  Fighter()\Shields= 400
  Fighter()\Armor = 400
EndIf

ClearScreen(ClearColor)          ; clear the screen to black

; Set up the vertical control variable
TextY = 0

StartDrawing(ScreenOutput())
    ; point to the first element in the Fighter List
    *Ship = FirstElement(Fighter())

    ; While it's a valid element
    While *Ship <> 0
        MissileText.s = "Name: " + *Ship\Name
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Weapons Type: " + Str(*Ship\WeaponsType)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Shields: " + Str(*Ship\Shields)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 16

        MissileText.s = "Armor: " + Str(*Ship\Armor)
        DrawText(0,TextY,MissileText)
        TextY = TextY + 32

        ; now point to the next element
        *Ship = NextElement(Fighter())
```

```
    Wend

    ; display a message so the user knows how exit
    DrawText(0,400,"Press any key to Exit")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End  ; end the program
```

Other List Commands

There are a few additional commands that we should briefly touch on with Lists.

```
ClearList(ListName())
```

This command will literally wipe the List out. It's used to delete the entire List, instead of going through them one-by-one.

```
FirstElement(ListName())
```

Although we've already seen this one in action in our previous example, its role is to return the location of the very first element in the List.

And of course, its counterpart:

```
LastElement(ListName())
```

…which returns the location of the very last element in the List. If you only have a single element, both FirstElement and LastElement will return the same value.

Next we have InsertElement. This is a very useful command as it will allow us to push an element in between two existing elements, or even right at the front of the List, if so needed. Maybe a high score would be a good example:

```
FirstElement(ListName())
InsertElement(ListName())
```

```
ListName()\Score = 139820
```

If you want to find the numeric value of an element in your list (0…1…2…3), you would use the ListIndex command.

```
Element = ListIndex(ListName())
```

To see how this works, I would recommend adding the ListIndex command into one of the above examples, as part of the information printout.

Please note that ListIndex does not return a memory location, but rather a positional location within the List. So if you have 5 elements, it will number them 0-4 for you. This is helpful when used in conjunction with the SelectElement command.

```
SelectElement(ListName(),2)
```

This will set your List location to be at the 2nd element so you can process that element directly. You'll find this quite handy in many situations, and it will save you time in your code by helping you avoid unnecessary loops.

And, as we previously mentioned, there are the two keywords that help you traverse the lists.

```
NextElement(ListName())
PreviousElement(ListName())
```

Finally, if you need the ability to set your List location to *before* the first element in the list:

```
ResetList(ListName())
```

That's all you need to do. Now you can start off using your NextElement command and get your data out!

Chapter 8: Working with Memory

While PureBasic handles the majority of memory needs for you, sometimes you'll want to access memory buffers more directly. PB has included tools for creating and managing these memory buffers, and that's what we're going to discuss in this chapter.

Memory buffers 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. PureBasic doesn't handle all of the processing like it does with Structures, for example, 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. But it can also be quite challenging if you're not careful in your planning and use of these powerful tools.

Creating and Freeing Memory Buffers

Whenever you create a buffer it's important to know that you are allocating byte-sized chunks. If, for example, you plan to store a long into a bank, you would have to create a buffer that is four bytes long. If you created a buffer that is only 1 byte wide and tried to store a long value, you would be allowed to do that but that could be very problematic. The reason is that you've only been granted ONE byte of memory, so the remaining 3 bytes are overwriting an area of memory that hasn't been reserved for you. These types of things can be very dangerous and cause all sorts of program crashes and problems. So make sure that you're being very careful when working with memory buffers.

Creating a buffer is easy. Just use the AllocateMemory command. Here is the layout:

MemoryBuffer* = ***AllocateMemory(**NumberOfBytes****)***

MemoryBuffer, in this example, is a unique *pointer* (hence the preceding *) that you will use in reference to this particular buffer for processing. It can be any name you choose, except for any PureBasic 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 PureBasic was able to allocate the memory requested. If the *MemoryBuffer* contains 0 (zero), then PB was unsuccessful in creating the buffer and you should take appropriate steps.

You also want to be sure to release the memory associated to your buffer 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 FreeMemory command:

FreeMemory(*MemoryBuffer**)***

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

```
  ; allocate 500 bytes of memory
*MemoryID = AllocateMemory(500)

; if we can't, display an error and exit
If *MemoryID = 0
   MessageRequester("Error!", "Unable to Allocate Memory",  ↵
   → #PB_MessageRequester_OK)
   End
EndIf
```

This piece of code attempts to allocate 500 bytes of memory. If it can't, it will display a message to the user stating that it failed, and will then exit. Again, it's very wise to make sure that AllocateMemory was successful in allocating the memory requested.

Poke and Peek

So after we've allocated memory, 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 five POKE/PEEK types that you have access to:

- **PokeB / PeekB**
- PokeW / PeekW
- PokeL / PeekL
- PokeF / PeekF
- PokeS / PeekS

The POKE commands are all formatted identically, with the exception of PokeS which we'll touch in a second. Let's use PokeB as an example:

PokeB(*MemoryID,115**)***

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

Value ***= PeekB(****MemoryID**)***

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.

Strings are handled a bit differently. Since a string may be any number of bytes, PB needs to add a mark to it stating where the string ends. This mark is a byte value of '0' that is automatically added after the last character in the string. Whenever you retrieve a string (Peek), PB will keep snagging memory until that '0' is hit. Let's look at the PokeS and PeekS commands:

PokeS(*MemoryID,"Hello"****)***
… or…
PokeS(*MemoryID,"Hello",5****)***

Notice that you can include the string length as an argument in the call. This is completely optional, but you may find it useful for various reasons. If you put a 5 as the length to POKE, PB actually will use 6 bytes because it has to incorporate that '0' as the string's terminator. Likewise, PeekS will read in the length + 1.

Value.s = ***PeekS(****MemoryID****)***
… or …
Value.s = ***PeekS(****MemoryID,5****)***

If you stored a value that is 5 bytes long and tried to read back 10 bytes, PB will only read the 5 bytes. This is because when PB hits byte number 6, it will find the '0' terminator. Once that's found, the reading stops.

Here is a list of the possible POKE/PEEK types and their respective sizes:

Type	Size
Ascii	1 Byte
Byte	1 Byte
Char	1 Byte in Ascii, 2 Bytes in Unicode
Double	8 Bytes
Float	4 Bytes
Integer	4 Bytes on 32-bit processors, 8 Bytes on 64-bit processors
Long	4 Bytes
Quad	8 Bytes
String	Unlimited - requires a null character (0) for termination
Unicode	2 Bytes
Word	2 Bytes

Here is a little piece of code that demonstrates a simple two-byte buffer (or a Word) using POKE and PEEK:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
```

```
EndIf

; Allocate 2 bytes of memory, WORD VALUE
*MemoryID = AllocateMemory(2)

If *MemoryID = 0
  MessageRequester("Error!", "Unable to Allocate Memory",
#PB_MessageRequester_Ok)
  End
EndIf

; store our value, using Poke
PokeW(*MemoryID,12123)

; Read back our value, using Peek
Value.w = PeekW(*MemoryID)

StartDrawing(ScreenOutput())

   ; Show the user the value
   DrawText(0,0,"The value in our memory buffer is: " + Str(Value))

    ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to exit")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

; free the allocated memory
FreeMemory(*MemoryID)

End  ; end the program
```

Resizing Allocated Memory

So what if you have a buffer all set up, but you find that you need to expand its size? Maybe you are dynamically allocating buffer memory for each new map that you load in. You could easily just free the current buffer and re-create it, or you could use the ReAllocateMemory command. Here is the format:

ReAllocateMemory(**MemoryID,10****)***

Resizing allocated memory is as simple as creating it, with the only difference being that you already know the pointer name of the buffer!

It is still important that you check that PB was able to resize successfully, as you do with AllocateMemory, so don't leave out that step. Again, here's a little snippet that creates a bank and then resizes it.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
   End
EndIf

; Allocate 2 bytes of memory, WORD VALUE
*MemoryID = AllocateMemory(2)

If *MemoryID = 0
   MessageRequester("Error!", "Unable to Allocate Memory",
#PB_MessageRequester_Ok)
   End
EndIf

; store our value, using Poke
PokeW(*MemoryID,12123)

; Read back our value, using Peek
Value.w = PeekW(*MemoryID)

StartDrawing(ScreenOutput())

    ; Show the user the value
    DrawText(0,0,"The value in our memory buffer is: " + Str(Value))

     ; display a message so the user knows how to exit
    DrawText(0,400,"Press any key to continue")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
   ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)


; Let's reallocate the memory to be a Long
ReAllocateMemory(*MemoryID,4)

If *MemoryID = 0
```

```
  MessageRequester("Error!", "Unable to ReAllocate Memory",
#PB_MessageRequester_Ok)
  End
EndIf

; store our value, using Poke - now a long
PokeL(*MemoryID,50000)
; Read back our value, using Peek
Value2.l = PeekL(*MemoryID)

StartDrawing(ScreenOutput())

   ; Show the user the value
   DrawText(0,0,"The new value in our memory buffer is: " + Str(Value2))

    ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to continue")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

; free the allocated memory
FreeMemory(*MemoryID)

End  ; end the program
```

So what happens to the data if you resize a buffer 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 Memory Buffers

Copying data between two buffers is a snap. All you need to do is use the CopyMemory command, which looks as follows:

CopyMemory(**SourceMemoryID,*DestinationMemoryID,Length****)***

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

```
CopyMemory(*SourceMemoryID,*DestinationMemoryID,100)
```

If we reference *DestinationMemoryID* it will contain the identical information that *SourceMemoryID* contains.

Now here's something that's a bit interesting. What if we have a 2-byte piece of memory (a Word) value, but we only copied the 1st byte? You'll get only the value that the first byte holds, that's what will happen. Enter the following code to see an example of this:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
   End
EndIf

; Allocate 2 bytes of memory, WORD VALUE
*SourceMemoryID = AllocateMemory(2)
*DestinationMemoryID = AllocateMemory(2)

If *SourceMemoryID = 0 Or *DestinationMemoryID = 0
   MessageRequester("Error!", "Unable to Allocate Memory",
#PB_MessageRequester_Ok)
   End
EndIf

; store our value, using Poke
PokeW(*SourceMemoryID,12123)

; copy only 1 byte from the source
CopyMemory(*SourceMemoryID,*DestinationMemoryID,1)

; Read back our values, using Peek
Value.w = PeekW(*SourceMemoryID)
Value2.w = PeekW(*DestinationMemoryID)

StartDrawing(ScreenOutput())

    ; Show the user the value
    DrawText(0,0,"The value in our source buffer is: " + Str(Value))
    DrawText(0,16,"The value in our destination buffer is: " + Str(Value2))

     ; display a message so the user knows how to exit
    DrawText(0,400,"Press any key to continue")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
   ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)
```

```
; free the allocated memory
FreeMemory(*SourceMemoryID)
FreeMemory(*DestinationMemoryID)

End  ; end the program
```

See how the first value is 12123, but the second is only 91? This is because of how much the value of 2 bytes adds up to versus how much the single byte is worth. Here is the Word layout:

```
101111101011011
```

If you go through and add that up using binary, you'll get a value 12123. Now, here's that same value with the bytes split:

```
1011111 | 01011011
```

Computers read bytes from right-to-left, so when we ask PureBasic to copy only 1 byte in a 2-byte field, it will copy the right-most byte. If we had a 4-byte field and asked PB to copy two of them, it would copy the two right-most bytes. This means our resultant copy is:

```
01011011
```

Which, if you add it up, results in 91.

Comparing Memory

Since you will certainly have pieces of memory you will want to compare (say for password checks or the like), PureBasic provides you with a method of handling this.

Result = ***CompareMemory(*****MemoryID1,*MemoryID2,2****)***

This will compare the contents in *MemoryID1* to the contents in *MemoryID2*, but only for the number of bytes requested, which in this case is 2. If the comparison is equal, the CompareMemory command will return a 1; if not, it will return a 0.

Here is an example based off of our previous CopyMemory example:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
```

```
  End
EndIf

; Allocate 2 bytes of memory, WORD VALUE
*MemoryID1 = AllocateMemory(2)
*MemoryID2 = AllocateMemory(2)

If *MemoryID1 = 0 Or *MemoryID2 = 0
  MessageRequester("Error!", "Unable to Allocate Memory",
#PB_MessageRequester_Ok)
  End
EndIf

; store our value, using Poke
PokeW(*MemoryID1,12123)

; copy only 1 byte from the source
CopyMemory(*MemoryID1,*MemoryID2,1)

; Read back our values, using Peek
Value.w = PeekW(*MemoryID1)
Value2.w = PeekW(*MemoryID2)

; now compare the first 2 bytes of the memory buffers
Result  = CompareMemory(*MemoryID1,*MemoryID2,2)

; now compare the first byte of the memory buffers
Result2 = CompareMemory(*MemoryID1,*MemoryID2,1)

StartDrawing(ScreenOutput())

   ; Show the user the value
   DrawText(0,0,"The value in our source buffer is: " + Str(Value))
   DrawText(0,16,"The value in our destination buffer is: " + Str(Value2))
   DrawText(0,32,"Comparing the first 2 bytes: " + Str(Result))
   DrawText(0,48,"Comparing the first byte only: " + Str(Result2))

    ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to continue")

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

; free the allocated memory
FreeMemory(*MemoryID1)
```

```
FreeMemory(*MemoryID2)

End  ; end the program
```

String-Specific Commands

Strings have a few specific commands associated to them only. Where we use CopyMemory and CompareMemory with our other types, we would use CopyMemoryString and CompareMemoryString with strings.

First we'll look at the CopyMemoryString command:

Result = ***CopyMemoryString(****"Hello",@*PointerToMemory****)***

CopyMemoryString is really just an extended version of PokeS. Where PokeS does a 'standard' string copy, CopyMemoryString allows you to copy data in a 'streamed' manner. You pass the command the address to a memory pointer, and then you may copy data to that whenever you'd like, streaming it in instead of the standard one-time copy. With the streaming ability you can adjust the location in the memory buffer by adjusting the pointer. If you want to go back 6 spots in the memory buffer, do:

```
*PointerToMemory - 6
```

So if you had the string "Hi There!" and your pointer was sitting just beyond the "!" in memory, you could change that string to say "Hi Buddy!" using the following code:

```
*PointerToMemory – 6
CopyMemoryString("Buddy!")
```

Next, we look at the CompareMemoryString command:

Result = ***CompareMemoryString(*****String1,*String2****)***
… or …

Result = ***CompareMemoryString(*****String1,*String2, 0, 6****)***

This command allows a simple one-to-one compare of two strings in memory. Alternately, you can control how it compares the two strings by adjusting the third argument (the mode), and how many characters to compare by adjusting the fourth argument (length).

There are two modes available:

- 0 – Compares the strings with case-sensitivity on. This means that 'A' will not equal 'a' in the comparison.

- 1 – Compares the strings without regard to case. This means that 'A' *will* equal 'a' in the comparison.

The result returned is handled a bit differently than with the standard CompareMemory command. Here is the breakdown of the returns:

- 0 – This result means that the strings are equal.
- 1 – Means that String 1 is greater than String 2
- -1 – Means that String1 is less than String 2

Here is an example piece of code that is built off of our previous example:

```
; Initialize the sprite, keyboard, and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Allocate 20 bytes of memory
*MemoryID1 = AllocateMemory(20)
*MemoryID2 = AllocateMemory(20)

; set a pointer to our second memory buffer
*Pointer = *MemoryID2

If *MemoryID1 = 0 Or *MemoryID2 = 0
  MessageRequester("Error!", "Unable to Allocate Memory", ↵
  → #PB_MessageRequester_Ok)
  End
EndIf

; store our value, using Poke
PokeS(*MemoryID1,"Hi there!")

; copy the string to our 2nd memory buffer
CopyMemoryString(*MemoryID1,@*Pointer)

; Read back our values, using Peek
Value.s = PeekS(*MemoryID1)
Value2.s = PeekS(*MemoryID2)

; now compare the two strings
Result = CompareMemoryString(*MemoryID1,*MemoryID2,1,6)

; and get the length of our second one
Length = MemoryStringLength(*MemoryID2)
```

```
If StartDrawing(ScreenOutput())
   ; Show the user the value
   DrawText(0,0,"The value in our source buffer is: " + Value)
   DrawText(0,16,"The value in our destination buffer is: " + Value2)
   DrawText(0,32,"Comparing the two strings: " + Str(Result))
   DrawText(0,48,"Length of memory string 2 is: " + Str(Length))

    ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to continue")
Else
  MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
   → #PB_MessageRequester_Ok)
  End
EndIf

StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

ClearScreen(ClearColor)

*Pointer - 6
CopyMemoryString("Buddy!")
Value.s = PeekS(*MemoryID1)
Value2.s = PeekS(*MemoryID2)

; now compare the two strings
Result = CompareMemoryString(*MemoryID1,*MemoryID2,1,6)

; and get the length of our second one
Length = MemoryStringLength(*MemoryID2)

If StartDrawing(ScreenOutput())
   ; Show the user the value
   DrawText(0,0,"The value in our source buffer is: " + Value)
   DrawText(0,16,"The value in our destination buffer is: " + Value2)
   DrawText(0,32,"Comparing the two strings: " + Str(Result))
   DrawText(0,48,"Length of memory string 2 is: " + Str(Length))

    ; display a message so the user knows how to exit
   DrawText(0,400,"Press any key to continue")
Else
  MessageRequester("Error!", "Unable to Draw to ScreenOutput()", ↵
   → #PB_MessageRequester_Ok)
  End
EndIf
```

```
StopDrawing()

FlipBuffers()  ; show the output to the user

; wait for any key to be pressed
Repeat
   ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

; free the allocated memory
FreeMemory(*MemoryID1)
FreeMemory(*MemoryID2)

End  ; end the program
```

Chapter 9: Procedures 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 Procedures.

A procedure 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 PureBasic. When you click on it to run, the computer takes over and loads up PB. Upon completion, the computer gives control back to you. Procedures act similarly, with the exception that they return control back to the code that called them.

Procedures 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 procedure that checks where on your ship was hit, what the current armor was at the time of the hit and how much power the hit delivered. From here the procedure processes all the data, does a calculation and let's you know what the total damage was.

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

Declaring a Procedure

In order to use a procedure, you must first declare it before it is called. Some people prefer to put all their procedures first in their source code, but most prefer either putting them under after their main loop or in a separate file altogether. I'm going to assume that you're one of the second lot, since there are more of those.

PureBasic has two commands for our procedures: Declare and Procedure. Here is the layout of both:

Declare.<ReturnType> *ProcedureName (Arguments)*

Procedure.<ReturnType> *ProcedureName(Arguments)*

The Declare command is used near the top of your code, before you call the procedure from your main code. This is done so the compiler will be

aware of the procedure and how the procedure is supposed to be called. The Procedure command is used at the top of the actual procedure you are coding, and it basically replicates what you put in the Declare command.

It's a better method to either put your procedures above your main code or to put them in a separate file and include them in your main code. I've found that most people tend to do the later method. In the following examples I will be using the Declare statement for clarity, but as we get into building our own libraries I will be using this statement only as necessary.

What you call the procedure 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 procedure in other programs.

Examples of bad procedure names:

- Procedure a()
- Procedure MoveIt(It)
- Procedure 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 procedure 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 procedure.

Examples of good names:

- Procedure FireLaser(Direction)
- Procedure CheckCollisions(Image1, Image2)
- Procedure DisplayScore(CurrentScore)

Notice that each of these procedures 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.

If you're going to create a library, however, you may consider taking it a step further.

Examples of good names for a library of, say, map routines:

- Procedure Map_Move(Direction)
- Procedure Map_CheckCollisions(Image1, Image2)
- Procedure Map_Load (CurrentScore)

Notice that I preface the procedure name with the word *Map*. This is because all of the procedures within this particular library will be specifically for map manipulation. If we share our library with someone else, we should be sure that our library names won't conflict with their standard procedure names. They may already have a *CheckCollisions* procedure, as it's a common name, but since they're opting to use our map library, it's not likely that they'll have a *Map_CheckCollisions* procedure.

Procedures most commonly take some data, do some form of manipulation on the data, and return to you the result of the manipulation. You can specify the type of value you want returned following the procedure name with a period '.' and then the return type. The return types are as follows:

- b - Returns a Byte.
- s - Returns a String.
- w - Returns a Word.
- l – Returns a Long.
- f – Returns a Float.

So, if you had a procedure that added to floats, you may declare the procedure as follows:

```
Declare AddFloats.f(Number1.f,Number2.f)
```

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

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

I won't do this on all my procedures, just ones that warrant it. Some smaller procedures end up having more comments than code, so I tend to be much less verbose with those. Also, there is no point in putting all of this data with every single procedure in your library. In other words, if you've created a bunch of map files, you know you're the author of them all, so just put in the necessary bits. No args or returns or comments? Don't bother putting them in.

Now, it's certainly not necessary that you use this format or that you comment your procedures at all. But I would highly recommend that you do to at least some degree. 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.

Passing Arguments and Returning Results

First thing I should qualify is what exactly a procedure *argument* is. An *argument* is a piece of data that you send to a procedure for processing. For example, if you wanted to add two numbers together, you would *send* the numbers to the procedure. The procedure would then add the two numbers and return the resultant value.

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

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

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

Declare.b AddNumbers(iNumber1.b, iNumber2.b)
Declare.s ConcatString(String1.s, String2.s)
Declare.f AvgNumbers(float1.f, float2.f, float3.f)
Declare.w GetArray(Location.w)
Declare SetupShips()
Declare.w GetShip(ShipID.b)
Declare WaitKey()

; make a dummy array and fill it with values
Global Dim ArrayValues.l(4)
ArrayValues(0) = 100
ArrayValues(1) = 200
ArrayValues(2) = 300
ArrayValues(3) = 400
ArrayValues(4) = 500

; define a type for Ships
Structure Ships
   ShipID.b   ; what's it's ID?
   ShipName.s ; the name?
   Speed.b    ; top speed (3-7)
EndStructure
```

```
Global NewList Ship.Ships()

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

; call AddNumbers procedure and place the returned-value
; into the variable "byteValue"
byteValue.b = AddNumbers(10,20)

; call ConcatString procedure and place the returned-value
; into the variable "stringValue"
stringValue.s = ConcatString("How","dy")

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

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

; call GetShip procedure and place the returned-value
; into the variable "ShipLocation"
ShipLocation.w = GetShip(Random(9))
; now make sure we have that element selected
SelectElement(Ship(),ShipLocation)

; Set up the vertical control variable
TextY = 0

StartDrawing(ScreenOutput())
; display returned values
DrawText(0,TextY,"byteValue = " + Str(byteValue))
TextY = TextY + 16

DrawText(0,TextY,"stringValue = " + StringValue)
TextY = TextY + 16

DrawText(0,TextY,"floatValue = " + Str(floatValue.f))
TextY = TextY + 16

DrawText(0,TextY,"arrayValue = "+ Str(arrayValue))
TextY = TextY + 32

DrawText(0,TextY,"** Ship Info **")
TextY = TextY + 16

DrawText(0,TextY,"ID = " + Str(Ship()\ShipID))
TextY = TextY + 16
```

```
DrawText(0,TextY,"Name = " + Ship()\ShipName)
TextY = TextY + 16

DrawText(0,TextY,"Speed = " + Str(Ship()\Speed))
StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey ()

; end the program
End


;************************************************
;           P R O C E D U R E S
;************************************************
;   Author: John Logsdon
;   Last Upd: 8/22/2012

;************************************************
;  Procedure: AddNumbers()
;    Purpose: add two numbers and return the result
;       Args: Two numbers
;    Returns: Byte - Sum of the two numbers sent
;************************************************
Procedure.b AddNumbers(iNumber1.b, iNumber2.b)
   iSum.b = iNumber1 + iNumber2
   ProcedureReturn(iSum)
EndProcedure

;************************************************
;  Procedure: ConcatString$()
;    Purpose: concatonates two strings
;       Args: Two strings
;    Returns: the resultant string
;************************************************
Procedure.s ConcatString(String1.s, String2.s)
   ProcedureReturn(String1.s + String2.s)
EndProcedure

;************************************************
;   Procedure: AvgNumbers()
;    Purpose: Find the average of 3 floats
;       Args: Two floats
;    Returns: Sum of the two numbers sent
;************************************************
Procedure.f AvgNumbers(float1.f, float2.f, float3.f)
   avg.f = (float1 + float2 + float3) / 3
```

```
    ProcedureReturn(avg)
EndProcedure

;*************************************************
;   Procedure: GetArray()
;   Purpose: Get a particular array value
;      Args: location in the array
;   Returns: The array value
;*************************************************
Procedure.w GetArray(Location.w)
    ProcedureReturn(ArrayValues(Location))
EndProcedure

;*************************************************
;   Procedure: SetupShips()
;   Purpose: adds a few ships to the Structure
;*************************************************
Procedure SetupShips()
    For i = 0 To 9
        If AddElement(Ship()) <> 0
            Ship()\ShipID = i
            Ship()\ShipName = "Ship" + Str(i)
            Ship()\Speed = Random(4) + 3
        EndIf
    Next
EndProcedure

;*************************************************
;  Procedure: GetShip()
;   Purpose: locates a ship and returns its index
;      Args: Ship's ID
;   Returns: the Index entry for the Ship
;*************************************************
Procedure.w GetShip(ShipID.b)
    ForEach Ship()
        If Ship()\ShipID = ShipID.b
            Break
        EndIf
    Next
    ProcedureReturn(ListIndex(Ship()))
EndProcedure

;*************************************************
;  Procedure: WaitKey()
;   Purpose: Wait for a keypress
;*************************************************
Procedure WaitKey()
  Repeat
    ExamineKeyboard()
  Until KeyboardReleased(#PB_Key_All)
```

```
EndProcedure
```

If you study that code in detail it should be pretty clear how to handle each return case. And, yes, I know that the comments are way overdone, was just doing that to show you how it can be handled for different procedures.

Including Files

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

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

```
IncludeFile "ImageProcessing.PB"
```

Now that piece of code will be inserted into your code at the position that you called it.

But what if we had a number of libraries that you wanted to include in our project, and those libraries all rely on yet another library?

If we use the IncludeFile command, PureBasic will complain because it will see a bunch of procedures and variables being declared more than once. What we need is a command that will include the procedures and supporting code one time, but will make it accessible to all of our code for calling. Fortunately, PB has a command just for this: XIncludeFile.

```
XIncludeFile "ImageProcessing.PB"
```

For most cases you can just use XIncludeFile, but there may be that rare occurrence where you will want to make sure your procedures are *not* available to all your code, hence the IncludeFile command.

As an example, let's say that you have two libraries, *Ships.pb* and *Planets.pb*. Now let's say that each of those libraries relies on yet another library called *Starfield.pb*. If you did the following:

[example piece of code for *Ships.pb*]

```
IncludeFile "Starfield.pb"
```

[example piece of code for *Planets.pb*]

```
IncludeFile "Starfield.pb"
```

[example piece of code for your main program]

```
IncludeFile "Ships.pb"
IncludeFile "Planets.pb"
```

PureBasic will give you an error during compilation. However, if you did the following:

[example piece of code for *Ships.pb*]

```
XIncludeFile "Starfield.pb"
```

[example piece of code for *Planets.pb*]

```
XIncludeFile "Starfield.pb"
```

[example piece of code for your main program]

```
IncludeFile "Ships.pb"
IncludeFile "Planets.pb"
```

PureBasic will see that the Starfield.pb file is already included and will not re-include it, so there won't be any errors during compilation.

Libraries

When you put a bunch of related procedures in a file, you can officially call that file a *library*. This is because it is now a "library of procedures." Pretty spiffy, no?

Here is a sample of what you can do. First, let's split off a couple of the procedures from our last example and put them in a file of their own. I'll name this file "myprocedures.pb" for fun. Here is what the contents of that file will look like:

```
;*************************************************
; Sample library just to show how to set one up.
;*************************************************
;    Author: John Logsdon
;  Last Upd: 8/22/2012

;*************************************************
```

```
;  Procedure: AddNumbers()
;    Purpose: add two numbers and return the result
;       Args: Two numbers
;    Returns: Byte - Sum of the two numbers sent
;-----------------------------------------------
Procedure.b AddNumbers(iNumber1.b, iNumber2.b)
   iSum.b = iNumber1 + iNumber2
   ProcedureReturn(iSum)
EndProcedure

;-----------------------------------------------
;   Procedure: AvgNumbers()
;    Purpose: Find the average of 3 floats
;       Args: Two floats
;    Returns: Sum of the two numbers sent
;-----------------------------------------------
Procedure.f AvgNumbers(float1.f, float2.f, float3.f)
   avg.f = (float1 + float2 + float3) / 3
   ProcedureReturn(avg)
EndProcedure

;-----------------------------------------------
;  Procedure: WaitKey()
;    Purpose: Wait for a keypress
;-----------------------------------------------
Procedure WaitKey()
  Repeat
    ExamineKeyboard()
  Until KeyboardReleased(#PB_Key_All)
EndProcedure
```

And now, here is the main source file that shows how to call it:

```
; include the myprocedures library
XIncludeFile "myprocedures.pb"

If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

 ; call AddNumbers procedure and place the returned-value
; into the variable "byteValue"
byteValue.b = AddNumbers(10,20)

; call AvgNumbers procedure and place the returned-value
; into the variable "floatValue"
floatValue.f = AvgNumbers(1.495,3.772,11.1935)
```

```
; Set up the vertical control variable
TextY = 0

StartDrawing(ScreenOutput())
; display returned values

DrawText(0,TextY,"byteValue = " + Str(byteValue))
TextY = TextY + 16

DrawText(0,TextY,"floatValue = " + Str(floatValue.f))
StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey ()

; end the program
End
```

IncludeFile/ XIncludeFile 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 procedures are available for you to use in piecing the final code together. And if your team member commented the procedure tops well enough, you'll also know how to call each procedure, what its purpose is, and what you can expect it to return.

Last thing I want to touch on is the IncludePath command. Often times you may have a number of libraries that you wish to include in various projects. As opposed to copying all of these libraries into each of your individual development directories, you can store them all in a central area and just let PB know where that area is.

```
IncludePath "libraries\gamelibs"
XIncludeFile "map.pb"
XIncludeFile "sprites.pb"
```

You could certainly just do the following:

```
XIncludeFile "libraries\gamelibs\map.pb"
XIncludeFile "libraries\gamelibs\sprites.pb"
```

But if you have a bunch of different library files all in one directory, it just makes it cleaner to use the IncludePath command.

Note that from now on I will have a little library called "generic.pb" that will contain the "WaitKey" procedure I setup. I will place this in a directory called "libraries" and will include it with most every example. This will save on code within the examples and will also help get you used to using libraries. You should expand this library with any generic procedures you end up writing.

Chapter 10: Working with Files

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 are they different? 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 use 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. But it'll make your player much happier.

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 advanced users can easily get past this. So maybe your file also contains encryption and compression to further protect the data.

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 a File

PureBasic 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, PureBasic handles this in one command. The CreateFile

command literally creates a file for writing. The command layout is as follows:

FilePtr = ***CreateFile (#File,FileName.s)***

Be careful when using this command because if you already have a file with the name you pass to the CreateFile command, PB will overwrite that file. Also, note that you must include the full path to the file. If you don't, PB will use the same directory that your program currently resides in to create the file.

The *#File* argument may contain any number you select as an identifier for this file. If the number is too high, you will get an error. If the number conflicts with a previously opened file, PB will close the previously opened file and use the file you just requested. In order to avoid this complication, you can replace that argument with the PureBasic constant *#PB_Any*. If CreateFile receives this constant as the *#File* argument, it will find an open value and return it to your *FilePtr*. Then you can just reference the *FilePtr* value when doing operations on your file.

If you specify the #File argument, then the result will just tell you if the file was successfully created or not. Any further access to the file must be made using the same handle you have specified as the #File parameter. On the other hand, if you use #PB_Any as the argument, then the result will not just tell you about success or failure. Instead, it will return you the handle you'll have to use for accessing the file. For both cases, a return value of 0 (zero) means the file could not be created, in which case you must avoid trying to access the file.

Writing to a File

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, most of which are for binary files only. Here they are and what each is for:

- WriteAsciiCharacter: Writes an ASCII character (1-byte)
- WriteByte: Writes a single byte
- WriteCharacter: Writes a character number (1-byte ASCII, 2-bytes Unicode)
- WriteData: Writes the contents of a specific memory buffer
- WriteDouble: Writes a double number (8-bytes)
- WriteFloat: Writes a floating-point value (4-bytes)
- WriteInteger: Writes an integer number (4-bytes on 32-bit, 8-bytes on 64-bit)
- WriteLong: Writes a long number (4-bytes)
- WriteQuad: Writes a quad number (8-bytes)
- WriteString: Writes a character string
- WriteStringFormat: Writes a Byte Order Mark
- WriteStringN: Writes a character string and includes a line feed

- WriteUnicodeCharacter: Writes a unicode character (2-bytes)
- WriteWord: Writes a word number (2-bytes)

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

WriteByte(*FilePtr, Value.b****)***
WriteWord(*FilePtr, Value.w****)***
WriteStringN(*FilePtr, Value.s****)***
etc...

The WriteData command, however, uses the following format:

WriteData(*FilePtr, *MemoryBufferID, LengthToWrite****)***

When using the WriteString/WriteStringN commands you are essentially telling PureBasic that you want your file to have easily readable text. This could be so you can allow people to customize the game via the file. You can still store out numeric data, of course, using the same method that you did with the DrawText command. So, for example, if you wanted to write out a line that had the contents of a numeric variable, you would do this:

```
WriteString(FilePtr , “Here is the value” + Str(Value))
```

After finishing up with a file, we should be sure to close the file using the CloseFile command. If you don’t properly close a file it could become corrupted, so take care to do this.

Let’s put together a small program that writes out two lines to a text file.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

; include our generic library
XIncludeFile "../libraries/generic.pb"

; create the file
FilePtr = CreateFile(#PB_Any,"filetest.txt")

If FilePtr = 0
  MessageRequester("Error!", "Unable to Create File", #PB_MessageRequester_Ok)
  End
```

```
EndIf

; write our strings out to the file
WriteStringN(FilePtr,"Hello, PureBasic!")
WriteStringN(FilePtr,"Testing...testing...1...2...3!")

; close the file
CloseFile(FilePtr)

; put up a little message to the user that we're done
If StartDrawing(ScreenOutput())
  DrawText(0,0,"File created and written to.  Press any key to exit")
Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
    → #PB_MessageRequester_Ok)
   End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey ()

; end the program
End
```

Reading From a File

In order to read from a file, we must first open that file. When opening a file, PureBasic will return a non-zero *Result* if successful, and a zero if not. Again, you may use the *#PB_Any* option in place of *#File*, which cause PB to store the file handle to the *Result*. Here is the format of the ReadFile command:

FilePtr = ***ReadFile(****#File,FileName.s****)***

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

- ReadAsciiCharacter: Reads an ASCII character (1-byte)
- ReadByte: Reads a single byte
- ReadCharacter: Reads a character number (1-byte ASCII, 2-bytes Unicode)
- ReadData: Reads the contents from a file into a specific memory buffer
- ReadDouble: Reads a double number (8-bytes)
- ReadFloat: Reads a floating-point value (4-bytes)

- ReadInteger: Reads an integer number (4-bytes on 32-bit, 8-bytes on 64-bit)
- ReadLong: Reads a long number (4-bytes)
- ReadQuad: Reads a quad number (8-bytes)
- ReadString Reads a character string to a file until it finds an end-of-line character
- ReadStringFormat: Reads a Byte Order Mark
- ReadUnicodeCharacter: Reads a unicode character (2-bytes)
- ReadWord: Reads a word number (2-bytes)
-

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:

Value.b = ***ReadByte(****FilePtr****)***
Value.w = ***ReadWord(****FilePtr****)***
Value.l = ***ReadLong(****FilePtr****)***

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

Length = ***ReadData(****FilePtr****,*****MemoryBufferID,LengthToRead****)***

The *Length* result will contain the actual number of bytes read.

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

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

; include our generic library
XIncludeFile "../libraries/generic.pb"

; open the file for reading
FilePtr = ReadFile(#PB_Any,"filetest.txt")

If FilePtr = 0
  MessageRequester("Error!", "Unable to Read File", #PB_MessageRequester_Ok)
  End
EndIf

; read in our strings from the file
String1.s = ReadString(FilePtr)
String2.s = ReadString(FilePtr)
```

```
; close the file
CloseFile(FilePtr)

; show the reads
If StartDrawing(ScreenOutput())
  DrawText(0,0,"String1: " + String1)
  DrawText(0,16,"String2: " + String2)
  DrawText(0,400,"Press any key to exit")
Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
    → #PB_MessageRequester_Ok)
   End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey ()

; end the program
End
```

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

This command will create a file if it does not already exist, otherwise it will just open it. But it will allow you to read and write to the file real-time. In fact, you have to use this command to alter any existing file because CreateFile 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.

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

FilePosition = ***Loc()***

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

FileSeek(*FilePosition****)***

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.

Result = ***Eof(#****File****)***

This command will return non-zero if the end of the file has been hit or a zero if not.

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 sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

; include our generic library
XIncludeFile "../libraries/generic.pb"

ClearColor = RGB(0,0,0)

; open the file for reading
FilePtr = OpenFile(#PB_Any,"filetest.txt")

If FilePtr = 0
  MessageRequester("Error!", "Unable to Read File", #PB_MessageRequester_Ok)
  End
EndIf

; read in our 1st from the file
String1.s = ReadString(FilePtr)

; save the current file position
FilePosition.w = Loc(FilePtr)

; read in the 2nd string
String2.s = ReadString(FilePtr)

; go back to the beginning of the 2nd string
FileSeek(FilePtr,FilePosition)
```

```
; write a new string
WriteStringN(FilePtr,"Testing...testing...4...5...6!")

; close the file
CloseFile(FilePtr)

; show the reads
If StartDrawing(ScreenOutput())
  DrawText(0,0,"String1: " + String1)
  DrawText(0,16,"String2: " + String2)
  DrawText(0,400,"Press any key to see the changes...")
Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()", ↵
    → #PB_MessageRequester_Ok)
   End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey()

ClearScreen(ClearColor)

; open the file for reading
FilePtr = OpenFile(#PB_Any,"filetest.txt")

If FilePtr = 0
   MessageRequester("Error!", "Unable to Read File", ↵
    → #PB_MessageRequester_Ok)
   End
EndIf

; read in our newly changed strings
String1.s = ReadString(FilePtr)
String2.s = ReadString(FilePtr)

; close the file
CloseFile(FilePtr)

; show the altered reads
If StartDrawing(ScreenOutput())
  DrawText(0,0,"String1: " + String1)
  DrawText(0,16,"String2: " + String2)
  DrawText(0,400,"Press any key to exit")
Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
```

```
    → #PB_MessageRequester_Ok)
  End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey ()

; end the program
End
```

Do note that the line of text we used to overwrite our previous line is the same length. If our new data had been shorter, we would still have leftover data in the file.

A Quick Binary Example

Just so you can see how binary files look and such, here is a quick example that does creates a file, writes binary data to it, reads the data back in and display the result.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

; include our generic library
XIncludeFile "../libraries/generic.pb"

ClearColor = RGB(0,0,0)

; open the file for reading
FilePtr = CreateFile(#PB_Any,"filetest.bin")

If FilePtr = 0
  MessageRequester("Error!", "Unable to Create File", #PB_MessageRequester_Ok)
  End
EndIf

; create a couple of buffer for the Write/Read Data commands
*MemoryBuffer1 = AllocateMemory(15)
*MemoryBuffer2 = AllocateMemory(15)
PokeS(*MemoryBuffer1,"Five Thousand!")
```

```
; write out our data
WriteByte(FilePtr,50)
WriteWord(FilePtr,500)
WriteLong(FilePtr,5000)
WriteFloat(FilePtr,5000.5)
WriteData(FilePtr,*MemoryBuffer1,15)

; close the file
CloseFile(FilePtr)

; show the reads
If StartDrawing(ScreenOutput())
  DrawText(0,0,"File created!")
  DrawText(0,400,"Press any key to read the data...")
Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
    → #PB_MessageRequester_Ok)
   End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey()

ClearScreen(ClearColor)

; open the file for reading
FilePtr = OpenFile(#PB_Any,"filetest.bin")

If FilePtr = 0
   MessageRequester("Error!", "Unable to Read File", ↵
    → #PB_MessageRequester_Ok)
   End
EndIf

; read in the binary data from the test file
value1.b = ReadByte(FilePtr)
value2.w = ReadWord(FilePtr)
value3.l = ReadLong(FilePtr)
value4.f = ReadFloat(FilePtr)
ReadData(FilePtr,*MemoryBuffer2,15)
value5.s = PeekS(*MemoryBuffer2,15)

; close the file
CloseFile(FilePtr)

; show the altered reads
```

```
If StartDrawing(ScreenOutput())
  DrawText(0,0,"Byte: " + Str(value1))
  DrawText(0,16,"Word: " + Str(value2))
  DrawText(0,32,"Long: " + Str(value3))
  DrawText(0,48,"Float: " + Str(value4.f))
  DrawText(0,64,"Data: " + value5)
  DrawText(0,400,"Press any key to exit")
Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
    → #PB_MessageRequester_Ok)
   End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey ()

; end the program
End
```

You should use a text editor to go and try to read the file "filetest.bin" so you can see how it looks. Some of the text will be readable, but most is not. This type of file will help stop most people from breaking into your files, but it certainly shouldn't be considered enough to stop even the most casual of hackers out there. For that you will need to have encryption algorithms and various other tricks, which, unfortunately, is beyond the scope of this writing. There are many books and Internet sites available that discuss how to incorporate these though, and there are also a number of libraries available to help you protect your data as well.

Miscellaneous File Commands

There are a few remaining commands that you should be aware of: Lof, IsFile, and UseFile.

The Lof command will return, in bytes, the actual size of a file. The format of this command is:

Result = ***Lof()***

IsFile checks the validity of an opened file, and makes sure it's ready to use.

Result = ***IsFile(****#File****)***

If the *Result* is non-zero then the file is ready to use.

Finally, the UseFile command allows the programmer to set the internal PB pointer, for the text file commands, to an opened file.

UseFile(*#File****)***

While you may have multiple files open at any given time, PB only references one at a time. UseFile makes it so you can control which one is to be referenced.

PART 2: PB 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 turned red also (unless that was your plan). So being able to know what the current color is will be a key factor in color control and manipulation.

In order to tell PB what color you want to use, you have to understand the concept of mixing three key values: R, G and B. These are the short names for Red, Green and Blue, respectively. In the same way a painter can mix basic ink colors in order to get different pallets, so you can mix the RGB of your pixels to accomplish the same. Unfortunately, computers mix colors a little bit different, but I'm pretty sure you'll get the hang of it soon enough.

The command for specifying the color you want to use is shown below:

FrontColor(*Red,Green,Blue****)***

This will set all further draws for pixels, lines, boxes, circles, and even fonts (characters) to use the color you requested. The values passed for each argument must be between 0 and 255. Sending 255,255,255 to the FrontColor 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.

Many of the drawing commands have an option for setting the color right in their argument list. But if you elect not to use the optional color argument, PB will take whatever you last assigned in the FrontColor command and use that. The optional color you can pass to the drawing

commands will not be separated in R, G and B, but mixed into a 24-bit value, making it a single parameter in the command call. In order to obtain a mixed value, you should use the RGB command:

MixedValue = RGB(Red,Green,Blue)

Once you have drawn a screen, you can get the color of a particular point on that screen by using the Point command. The format of this command is:

ColorValue = Point(*X, Y****)***

Point returns the 24-bit mixed color value of the pixel found at the X, Y location. Note that it will return the 32-bit information if the drawing mode is set to one of the alpha channel modes. If it's not set, then the alpha channel is set to 0.

There are three commands in PureBasic that you can use to extract the R, G and B values from a given 24-bit mixed value, and the names of these commands are all suggestive:

- Red() – gets the Red component of a 24-bit color mixed value.
- Green() – gets the Green component of a 24-bit color mixed value.
- Blue() – gets the Blue component of a 24-bit color mixed value.

So, if you get the color of a point on the screen and want to know the amount of Red that was used to draw that point, you can do this:

```
ColorValue = Point(X, Y)
RedAmount = Red(ColorValue)
```

The same is valid for the other components:

```
ColorValue = Point(X, Y)
RedAmount = Red(ColorValue)
GreenAmount = Green(ColorValue)
BlueAmount = Blue(ColorValue)
```

Another point to note is that the ClearScreen command can be used to clear the screen to a particular color, using the RGB command, as we've seen in prior chapters:

ClearScreen(RGB(*Red,Green,Blue****))***

Dealing with Pixels

A pixel (a condensed word meaning *picture element*) is just 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, is all made using pixels.

In order to draw a pixel to the screen, you would use the PureBasic Plot command. This command takes the current color and draws a pixel at the corresponding X, Y position on the screen. Alternately, using the method below, you can include the color directly into the Plot command call. To do this, it's easiest to use the RGB command. The following code will randomize colors and draw pixels over most of the screen until you press a key to exit.

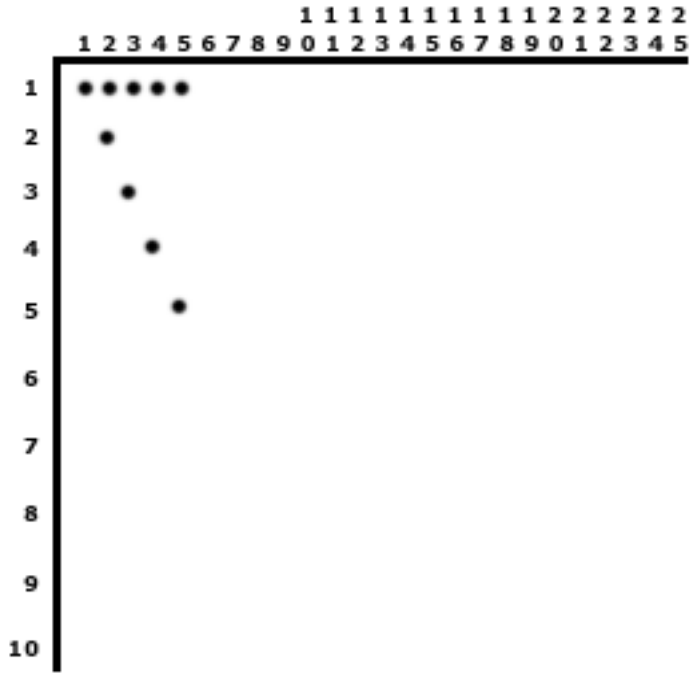
```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

Repeat
 If StartDrawing(ScreenOutput())
   For y=0 To 449
     For x=0 To 639
       ; choose random colors
       r = Random(255)
       g = Random(255)
       b = Random(255)
       ; draw out the pixel in the selected color
       Plot(x,y,RGB(r,g,b))
     Next
   Next
   DrawText(0,460,"Press any key to exit...")
  Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
  → #PB_MessageRequester_Ok)
   End
  EndIf

  StopDrawing()

  ; show the output to the users
  FlipBuffers()

  ; see if the user has pressed a key
  ExamineKeyboard()
Until KeyboardPushed(#PB_Key_All)

; end the program
End
```

Enter that in and you'll see a ton of pixels start filling up your screen.

Drawing Lines

A line is basically just a bunch of plotted points, or pixels. The problem is that 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 PB's built-in Line and LineXY commands. 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 PureBasic Line and LineXY commands handle it for us. Here's the format of the Line command:

Line(*StartX, StartY, Width, Height, [optional color])*

To see the Line command in action, enter in the following code:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

Repeat
 ClearScreen(ClearColor)
 If StartDrawing(ScreenOutput())
   For lines = 0 To 1000
     ; choose random colors
     r = Random(255)
     g = Random(255)
     b = Random(255)
     ; draw out the line in the selected color, at random places and sizes
     Line(Random(639),Random(449),Random(150),Random(150),RGB(r,g,b))
   Next
   DrawText(0,460,"Press any key to exit...")
  Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
  → #PB_MessageRequester_Ok)
   End
  EndIf

  StopDrawing()

  ; show the output to the users
  FlipBuffers()

  ; see if the user has pressed a key
  ExamineKeyboard()
  Delay(1)
Until KeyboardPushed(#PB_Key_All)
```

```
; end the program
End
```

The Line command is used to draw from a single known point to a distance on both the X and Y planes that may be variable. If you wanted to use a line as a missile, say, you would always know that the line is 5 pixels long moving horizontally across the screen. And the starting X point would increase a certain amount each frame (to give the appearance of movement). So, your call to the Line command may look like this:

```
Line(MissileX,MissileY, 5,0)
```

Each loop iteration you would update the MissileX value and the line would just draw from its new starting point. You won't have to worry about where it ends because PB will take care of that for you since you gave it the length of 5.

If you use the LineXY command, however, you will have to tell PB not only where you want the starting point to be, but also specifically where you want the ending point to be. This can give you better control over your lines, certainly, but may be a little more tedious. Here is the same snippet for that missile movement using LineXY.

```
LineXY(MissileX,MissileY, MissileX+5,MissileY)
```

And here is our prior full example using LineXY instead of Line.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

Repeat
 ClearScreen(ClearColor)
 If StartDrawing(ScreenOutput())
   For lines = 0 To 1000
      ; choose random colors
      r = Random(255)
      g = Random(255)
      b = Random(255)
```

```
      ; draw out the line in the selected color, at random places and sizes
      LineXY(Random(639),Random(449),Random(150),Random(150),RGB(r,g,b))
    Next
    DrawText(0,460,"Press any key to exit...")
  Else
    MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
   → #PB_MessageRequester_Ok)
    End
  EndIf

  StopDrawing()

  ; show the output to the users
  FlipBuffers()

  ; see if the user has pressed a key
  ExamineKeyboard()
  Delay(1)
Until KeyboardPushed(#PB_Key_All)

; end the program
End
```

Rectangles

To put up rectangles in PB, you use the Box command. You can make the rectangles as large or small as you want as well. Here is the layout for this command:

Box(*StartX, StartY, Width, Height, [Optional Color]****)***

The following code will draw a bunch of unfilled and filled rectangles all over the screen.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

Repeat
 ClearScreen(ClearColor)
 If StartDrawing(ScreenOutput())
   For lines = 0 To 1000
     ; choose random colors
     r = Random(255)
```

```
      g = Random(255)
      b = Random(255)
      ; draw out the line in the selected color, at random places and sizes
      Box (Random(639),Random(449),Random(150), Random(150),RGB(r,g,b))
    Next
    DrawText(0,460,"Press any key to exit...")
  Else
    MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
  →#PB_MessageRequester_Ok)
    End
  EndIf

  StopDrawing()

  ; show the output to the users
  FlipBuffers()

  ; see if the user has pressed a key
  ExamineKeyboard()
  Delay(1)
Until KeyboardPushed(#PB_Key_All)

; end the program
End
```

If you want to have the boxes be unfilled, add the DrawingMode command under the StartDrawing command, as follows:

```
If StartDrawing(ScreenOutput())
  DrawingMode(4)                ; ADD THIS LINE
  For lines = 0 To 1000
```

The DrawingMode command allows you to control the way things are drawn in PB. There are 4 primary options, but they can all be combined in various ways. The options are:

- 0 (#PB_2DDrawing_Default) – This is the default. Text has backgrounds, shapes (boxes, circles, etc.) are filled.
- 1 (#PB_2DDrawing_Transparent) – The background behind the text is transparent.
- 2 (#PB_2DDrawing_XOr) – Enable the XOR'd mode (all graphics XOR'd with current background).
- 4 (#PB_2DDrawing_Outlined) – Enable outlined shapes. Circles, boxes, etc. will be drawn unfilled.

To combine these, you would use the *Logical OR* symbol, "|"

```
DrawingMode(2 | 4)  ; Enable XOR and make shapes unfilled
```

Circles and Ellipses

The final primitives we're going to discuss are the circle and the ellipse. The circle has a starting point at its center and we need to provide the radius to evenly expand it outward from that point. The ellipse, on the other hand, allows us to control the radius on both the X and Y planes, thus allowing variance in height and width.

Here are the layouts for both commands:

Circle(*X, Y, Radius, [Optional Color]****)***
Ellipse(*X, Y, RadiusX, RadiusY, [Optional Color]****)***

Using those layouts, go ahead and alter the code in the box example, incorporating first the circle to see how it works, and then the ellipse.

Chapter 12: Working with Sprites

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 PureBasic. 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 code a game that nobody even gives a second glance at because the artwork is 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 Sprites

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 PB command LoadSprite. Pretty intuitive, no? Here's the layout for the command:

Result **= LoadSprite(***SpriteNumber, FileName [,Optional Mode]***)**

You can either assign a unique number that you choose, using the *SpriteNumber* argument, in order to keep track of the sprite, or you can let PB give you a number by passing #PB_ANY. If you decide to allow PB to create a number, that number will be returned in the *Result* value. If you manually assign a number, just be sure that *Result* does not equate to zero. If it does, that means an error has occurred.

The second argument is the name of the file that contains your sprite image.

Finally, you have the option to include instructions for how you'd like PB to load in the file. There are a number of options:

- 0 – Default. Sprite resides in video memory (assuming there is room).
- #PB_Sprite_Memory – Sprite is loaded into main PC memory (for SpecialFX).
- #PB_Sprite_Alpha – Sprite is 8 bits grayscale and will be used by DisplayAlphaSprite() or DisplayShadowSprite().

- #PB_Sprite_Texture – Sprite is created with 3D support - useful for the CreateSprite3D() command.
- #PB_Sprite_AlphaBlending - Sprite is created with per-pixel alpha-channel support. The image format (PNG/TIFF only for now) has to support it. #PB_Sprite_Texture also needs to be specified, and the sprite has to be displayed using DisplaySprite3D.

The default file type that PB supports is BMP. If you wish to use another kind of file, you will need to call on the ImagePlugin library. Here are the most commonly used graphics types, and their calls:

- UseJPEGImageDecoder() – Sets up for allowing JPG/JPEG files.
- UseJPEG2000ImageDecoder() – Sets up for allowing JPG/JPEG 2000 files.
- UsePNGImageDecoder() – Sets up to allow PNG files.
- UseTIFFImageDecoder() – Sets up to allow TIFF files.
- UseTGAImageDecoder() – Sets up to allow TGA files.

After loading the sprite, you will want to draw it to the screen using the DisplaySprite command. DisplaySprite takes the *SpriteNumber* and draws the associated sprite at the specified X, Y coordinates.

DisplaySprite(*SpriteNumber, X, Y)*

If you specify an invalid *SpriteNumber*, PB will break out with an error. So, be safe and check that LoadSprite 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 BMP, but make sure you change the LoadSprite line in the code below):

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite()=0 Or InitKeyboard()=0 Or OpenScreen(640,480,16,"App Title")=0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

; include our generic library
XIncludeFile "../libraries/generic.pb"

; We're going to use PNG as our file, so call the plugin
UsePNGImageDecoder()

; Now load up the sprite
Result = LoadSprite(0,"test.png",0)

; make sure it was successful, or end the program
```

```
If Result = 0
  MessageRequester("Error!","Unable to load test.png",#PB_MessageRequester_Ok)
  End
EndIf

; display the sprite
DisplaySprite(0,200,50)

; put up a little message to the user that we're done
If StartDrawing(ScreenOutput())
 DrawText(0,460,"Press any key to exit")
Else
  MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
  → #PB_MessageRequester_Ok)
  End
EndIf

StopDrawing()

; show the output to the users
FlipBuffers()

; wait for a keypress
WaitKey()

; end the program
End
```

Note that the DisplaySprite command is called *outside* of the StartDrawing(…) piece. StartDrawing can be used to draw to Windows, Printers, Screens, Sprites, etc. It doesn't require the use DirectX, but you'll need DirectX to draw to screens or Sprites, of course. It's also used with the GUI part of PB. For example, to display images on windows and things like that you'll use the StartDrawing command. So, while it may seem strange that you call DisplaySprite outside of the StartDrawing command, when you take into account that they're really two different beasts, it should be more understandable.

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 PureBasic 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 PB will draw all of the pixels of the image with the exception of a color (called a *Mask*) that YOU select by

calling the TransparentSpriteColor command. The format of this command is:

TransparentSpriteColor(*SpriteNumber,* ***RGB****(RedColor, GreenColor, BlueColor)****)***

The color value is best set using the RGB command. The default color is black. So the default *mask* is black.

The DisplaySprite command does not pay attention to transparency, so it will draw everything, regardless of color.

In order to use the transparency, you will need to call the DisplayTransparentSprite command, which is called identically to DisplaySprite.

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 (everyone does!). You could make your player's ship by drawing it at all the different angles by hand, which is going to be the cleanest method for your images, but also the most time-consuming. Or you could draw it facing straight up and then use your graphics program to rotate each frame and then place the frames together. Or maybe draw it facing straight up, have PB load it and do all the rotations for you.

Let’s make a function that uses the last method, but using our main character, “Migz.” It will require the use of a couple of the 3D commands, but only because there are no default 2D commands for rotation in the core PB commands.

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.

We'll use a combination of RotateSprite3D, DisplaySprite3D, and GrabSprite commands. There are support commands that we’ll also need, like SpriteWidth and SpriteHeight.

This will be our most ambitious piece of code thus far, so make sure you take the time to study it. Also, note that there are more elegant ways of handling things like this, not mention many varying ways of doing it. Our method will be quick and useful, but not necessarily elegant.

Here’s the main code piece, we’ll show the procedure in a moment:

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitSprite3D() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(640,480,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Make it use 10-deg points, change this number around to see differences,
; but be sure it's equally divisible into 360 or you may see blowups.
; Try 10, 36, 72, 180, 360
Rotations = 36

; setup an array to hold our ship images
Global Dim Migz(Rotations)

; declard our graphics procedure
Declare Rotate_2D_To_Array(ImageToUse,Rotations)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprite with 3D capability
SpriteID = LoadSprite(#PB_Any,"migz.png",#PB_Sprite_Texture)

; set the appropriate mask so we have transparencies
TransparentSpriteColor(SpriteID,RGB(255,0,255))

; call the library procedure that rotates and copies the images
Rotate_2D_To_Array(SpriteID,Rotations)

; setup a little var for rotations
CurrentAngle = 0

Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ; just show our sprites spinning
  For y = 0 To 10
    For X = 0 To 14
      DisplayTransparentSprite(Migz(CurrentAngle), X * SpriteWidth(SpriteID), ↵
      → Y * SpriteHeight(SpriteID))
    Next
  Next

  ; simple tracking of the array position for visual effect
  CurrentAngle = CurrentAngle + 1
  If CurrentAngle >= Rotations
```

```
    CurrentAngle = 0
  EndIf

  StartDrawing(ScreenOutput())
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Any)

End
```

Before moving on to the rotation procedure there are a few things we should look at in the above code.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitSprite3D() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(640,480,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf
```

We have a new addition to our initializations, InitSprite3D. This call informs PureBasic that we will want to use the 3D command set. If PB can't initialize it on your system, then it will cause a system halt. Most systems these days should support the 3D suite of commands, but there are still some that do not.

```
; setup an array to hold our migz images
Global Dim Migz(Rotations)

; declard our graphics procedure
Declare Rotate_2D_To_Array(ImageToUse,Rotations)
```

We've dimensioned our array to hold the plethora of rotations that we'll need. Notice that we needed to put the Global keyword in front of the Dim keyword. This is because we're going to be using the array outside of its default scope.

Also, we've now setup the Declare for the procedure in charge of rotating our Migz image around.

Now, for loading the sprite:

```
; load in the sprite with 3D capability
SpriteID = LoadSprite(#PB_Any,"migz.png",#PB_Sprite_Texture)

; set the appropriate mask so we have transparencies
TransparentSpriteColor(SpriteID,255,0,255)
```

In order to load in a sprite with 3D support, we have to pass along the optional #PB_Sprite_Texture argument, and make sure we set up the transparency color too. You'll notice that I've used the "#PB_Any" value here to have PureBasic return a unique sprite identifier, called SpriteID. Now I just need to be sure to use that variable for each call that requires it.

Okay, now let's take a look at the library code that actually rotates the images and stores them into an array.

```
;************************************************
;  Procedure: Rotate_2D_To_Array(...)
;     Author: John Logsdon
;   Last Upd: 9/02/2012
;    Purpose: Rotates a 2D image and places their
;             respective handles into an array
;       Args: The image to use, the # of rotations
;    Returns: n/a - but does fill the array
;   Comments: I grab from -5,-5 to +5, +5 of the 3D
;             image because when an image is rotated,
;             it changes size accordingly.I know there
;             are mathematical ways to properly adjust
;             for this, but I'm just using a
;             quick-n-dirty method here.
;************************************************
Procedure Rotate_2D_To_Array(ImageToUse,Rotations)
  ; find the actual rotation amount
  ; i.e.: 360/36 = 10.  So 10-degrees is the rotation distance
  RotationAmount = 360 / Rotations
  ; start our initial angle at 0
  Angle = 0

  ; where to display the object for grabbing
  DisplayX = 50
  DisplayY = 50

  ; get the width and height plus some buffering so we don't have
  ; to call it every iteration of the loop
  Width=SpriteWidth(ImageToUse) + 10
  Height=SpriteHeight(ImageToUse) + 10

  ; Create a 3D sprite
  Sprite3D_ID = CreateSprite3D(#PB_Any,ImageToUse)
```

```
  ; since we're only going to use this to rotate and place in 2D pieces,
  ; set the quality up
  Sprite3DQuality(1)

  ; run through the rotations
  For i = 0 To Rotations - 1
    ; Set a number in our Migz array
    Migz(i) = i

    ClearScreen(ClearColor)

    ; start up the 3D drawing piece
    Start3D()
      ; rotate the sprite, but then leave back to it's original position
      ; for next call
      RotateSprite3D(Sprite3D_ID,Angle,#PB_Relative)
      ; draw the sprite using 3D
      DisplaySprite3D(Sprite3D_ID,DisplayX,DisplayY)
    Stop3D()
    ; grab the sprite to 2D
    GrabSprite(Migz(i),DisplayX - 5,DisplayY - 5,Width,Height,0)
    ; Update the angle
    Angle = Angle + RotationAmount
  Next
EndProcedure
```

SpriteWidth and SpriteHeight are used so we know how much of a space to grab using the GrabSprite command. Note that I add 10 both the width and height value because a diagonal image is larger than a non-diagonal one. Here is a visual for that:

See how the image on the right has little bits of it cut off? This is because the image changes slightly in size as it rotates. To compensate for this, I just add 5 pixels to all four sides so nothing gets cut off.

```
; Create a 3D sprite
Sprite3d_ID = CreateSprite3D(#PB_Any,0)
```

CreateSprite3D makes a 3D location for storing 2D sprites. It's just a rectangle that will accept a texture, which is a normal 2D sprite. The main thing to be aware of is that this will want a square sprite (8x8,

16x16, 32x32, 64x64, etc.). You can try to use a non-square sprite, but not all hardware cards will support it.

```
; since we're only going to use this to rotate and place in 2D pieces, set the quality up
Sprite3DQuality(1)
```

What we're doing here is telling PB that we want the sprite to have *bilinear filtering*, which means that the pixels will be averaged together to give a more realistic look. In actuality this makes the image look a little blurry, but that makes it look less pixilated. Less pixilated is a good thing for games, and it also makes for rotations to be more smooth and clean.

```
; start up the 3D drawing piece
Start3D()
```

Just like we have to use StartDrawing to draw up text and standard primitives, PB asks that we also call the Start3D and Stop3D commands to let it know when we're working with 3D.

```
; rotate sprite, put back to it's original position for next call
RotateSprite3D(Sprite3d_ID,Angle,0)
```

This is the command that does the actual rotation. It takes the image that we passed along, rotates it a certain amount, and then tells PB that the next time it gets called it should reset it back to what it was originally. We could tell the command to just keep rotating from where it was, but I don't like doing that. I don't like it because it tends to make the image look less clean than if it uses the original as the axis.

```
; draw the sprite using 3D
DisplaySprite3D(Sprite3d_ID,DisplayX,DisplayY)
```

This is just the 3D counterpart to DisplaySprite.

```
; grab the sprite to 2D
GrabSprite(Migz(i),DisplayX - 5,DisplayY - 5,Width,Height,0)
```

GrabSprite takes the image from video memory and slaps the handle to it into the array.

Type all that in and give it a go. You should see a bunch of Migz's (or whatever image you choose to use) spinning all over the screen.

Writing directly to a sprite

Every time you load in a new sprite, grab a sprite, create a sprite, etc. you are really creating a sprite *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 sprite. For example, let's take our Migz image and then draw some lines all over it. Why? Cause it's fun.

```
; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or OpenScreen(640,480,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprite with 3D capability
SpriteID = LoadSprite(#PB_Any,"Migz.png")

; set the appropriate mask so we have transparencies
TransparentSpriteColor(SpriteID,RGB(255,0,255))

; set the drawing element to be our sprite
StartDrawing(SpriteOutput(SpriteID))
  ; draw a couple of lines on it
  Line(0,0,31,31,RGB(255,255,0))
  Line(0,31,31,-31,RGB(255,255,0))
StopDrawing()

Repeat
  ; clear the screen
  ClearScreen(ClearColor)
  ; show our sprite
  DisplayTransparentSprite(SpriteID,300,200)
  ; put up text for exiting
  StartDrawing(ScreenOutput())
    DrawText(0,460,"Press any key To quit")
  StopDrawing()
  FlipBuffers()
  ExamineKeyboard()
Until KeyboardReleased(#PB_Any)

End
```

The key component here is the StartDrawing line. Here is where we set the drawing buffer to be our Migz character. So instead of using the full screen buffer for our drawing, we just use the sprite instead.

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.

Chapter 13: Handling Animation

In the previous chapter we talked about sprites and their uses. Now let's have some fun with them.

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 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 tells the video card to point to a new place in video memory when doing its refresh, it's instant.

I'm going to move a ball from one side of the screen to the next, bouncing it off the edges and such.

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprite
SpriteID = LoadSprite(#PB_Any,"ball.png")

; get the width and height of the sprites
Sprite_Width  = SpriteWidth(SpriteID)
Sprite_Height = SpriteHeight(SpriteID)
```

```
XDir = 1   ; 0=left, 1=right
YDir = 1   ; 0=up,   1=down
Speed = 2  ; how fast do we go (in pixels)

Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ; show our sprite
  DisplaySprite(SpriteID,X,Y)

  ; if we're moving left, subtract speed
  If XDir = 0
    X = X - Speed
  Else
    ; otherwise add speed
    X = X + Speed
  EndIf

  ; if we hit the right edge, start moving left
  If X > #ScreenWidth - Sprite_Width
    XDir = 0
  EndIf
  ; if we hit the left edge, start moving right
  If X < 1
    XDir = 1
  EndIf

  ; if we're moving up, subtract speed
  If YDir = 0
    Y = Y - Speed
  Else
    ; otherwise add speed
    Y = Y + Speed
  EndIf

  ; if we hit the bottom edge, start moving up
  If Y > #ScreenHeight - Sprite_Height
    YDir = 0
  EndIf
  ; if we hit the top edge, start moving down
  If Y < 1
    YDir = 1
  EndIf

  ; put up text for exiting
  StartDrawing(ScreenOutput())
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()
```

```
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

I've set up two variables at the very top of the code to assign the screen's width and height values. 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 as constants, then PB will incorporate them at compile time. Then you only need to change these values in ONE spot when doing updates and testing. It's safer, faster, and is less likely to introduce bugs in your code upon making such changes.

What makes this example work is that we keep adding the *Speed* to the *X* and *Y* values for the placement of the ball until we hit a wall. Depending on which wall we hit, we reverse the addition to subtraction (or subtraction to addition, as the case may be) for that plane and it gives the appearance that the ball is bouncing off the walls.

If you get rid of the ClearScreen(…) 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 Structure and a List that will keep track of a bunch of bouncing balls. We will 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 Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()
```

```
; load in the sprite
SpriteID = LoadSprite(#PB_Any,"ball.png")

; get the width and height of the sprites
Sprite_Width  = SpriteWidth(SpriteID)
Sprite_Height = SpriteHeight(SpriteID)

Structure Balls
   X.w     ; track the X position of the ball
   Y.w     ; track the Y position of the ball
   XDir.b  ; track the X direction of the ball (0=left,1=right)
   YDir.b  ; track the Y direction of the ball (0=up,  1=down)
   Speed.b ; track the speed of the ball
EndStructure

; setup a list of balls
NewList Ball.Balls()

; create a bunch of instances of balls
For BCounter = 0 To 50
   If AddElement(Ball()) <> 0
     Ball()\X = Random(#ScreenWidth - Sprite_Width)
     Ball()\Y = Random(#ScreenHeight - Sprite_Height)
     Ball()\XDir = Random(1)
     Ball()\YDir = Random(1)
     Ball()\Speed = Random(4) + 1
   Else
     MessageRequester("Error!", "Unable to allocate memory for new element", ↵
     → #PB_MessageRequester_Ok)
     End
   EndIf
Next

Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ForEach Ball()

    ; show our sprite (using transparency)
    DisplayTransparentSprite(SpriteID,Ball()\X,Ball()\Y)

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; First let's work with the X-Axis
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; if we're moving left, subtract speed
    If Ball()\XDir = 0
      Ball()\X = Ball()\X - Ball()\Speed
    Else
```

```
      ; we must be moving right, add speed
      Ball()\X = Ball()\X + Ball()\Speed
    EndIf

    ; if we hit the right edge, start moving left
    If Ball()\X > #ScreenWidth - Sprite_Width
      Ball()\XDir = 0
    EndIf
    ; if we hit the left edge, start moving right
    If Ball()\X < 1
      Ball()\XDir = 1
    EndIf

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; Now let's work with the Y-Axis
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; if we're moving up, subtract speed
    If Ball()\YDir = 0
      Ball()\Y = Ball()\Y - Ball()\Speed
    Else
      ; we must be moving down, add speed
      Ball()\Y = Ball()\Y + Ball()\Speed
    EndIf

    ; if we hit the bottom edge, start moving up
    If Ball()\Y > #ScreenHeight - Sprite_Height
      Ball()\YDir = 0
    EndIf
    ; if we hit the top edge, start moving down
    If Ball()\Y < 1
      Ball()\YDir = 1
    EndIf
  Next

  ; put up text for exiting
  StartDrawing(ScreenOutput())
   DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

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?

In addition to moving stuff around the screen, you'll 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. You'll either need to create two image files with the two circles like above, or you can make your own image file with whatever animation layout you would like. Here's the code:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprite
LoadSprite(0,"ballframe1.png")
LoadSprite(1,"ballframe2.png")

; get the width and height of the sprites
Sprite_Width  = SpriteWidth(0)
Sprite_Height = SpriteHeight(0)

Structure Balls
   X.w     ; track the X position of the ball
   Y.w     ; track the Y position of the ball
   XDir.b  ; track the X direction of the ball (0=left,1=right)
```

```
    YDir.b  ; track the Y direction of the ball (0=up,  1=down)
    Speed.b ; track the speed of the ball
    Frame.b ; track the frame we're using in the animation
EndStructure

; setup a list of balls
NewList Ball.Balls()

; create a bunch of instances of balls
For BCounter = 0 To 50
    If AddElement(Ball()) <> 0
      Ball()\X = Random(#ScreenWidth - Sprite_Width)
      Ball()\Y = Random(#ScreenHeight - Sprite_Height)
      Ball()\XDir = Random(1)
      Ball()\YDir = Random(1)
      Ball()\Speed = Random(4) + 1
      Ball()\Frame = Random(1)
    Else
      MessageRequester("Error!", "Unable to allocate memory for new element", ↵
      → #PB_MessageRequester_Ok)
      End
    EndIf
Next

Repeat
   ; clear the screen
   ClearScreen(ClearColor)

   ForEach Ball()

      ; show our sprite (using transparency)
      DisplayTransparentSprite(Ball()\Frame,Ball()\X,Ball()\Y)

      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; First let's work with the X-Axis
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; if we're moving left, subtract speed
      If Ball()\XDir = 0
        Ball()\X = Ball()\X - Ball()\Speed
      Else
        ; we must be moving right, add speed
        Ball()\X = Ball()\X + Ball()\Speed
      EndIf

      ; if we hit the right edge, start moving left
      If Ball()\X > #ScreenWidth - Sprite_Width
        Ball()\XDir = 0
      EndIf
      ; if we hit the left edge, start moving right
      If Ball()\X < 1
        Ball()\XDir = 1
```

```
    EndIf

    ;.....................................
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; Now let's work with the Y-Axis
    ;.....................................
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; if we're moving up, subtract speed
    If Ball()\YDir = 0
      Ball()\Y = Ball()\Y - Ball()\Speed
    Else
      ; we must be moving down, add speed
      Ball()\Y = Ball()\Y + Ball()\Speed
    EndIf

    ; if we hit the bottom edge, start moving up
    If Ball()\Y > #ScreenHeight - Sprite_Height
      Ball()\YDir = 0
    EndIf
    ; if we hit the top edge, start moving down
    If Ball()\Y < 1
      Ball()\YDir = 1
    EndIf

    ;.....................................
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; Finally let's update the frame
    ;.....................................
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;if we're on Frame 0, move to 1
    If Ball()\Frame = 0
      Ball()\Frame = 1
    Else
      ; otherwise move to 0
      Ball()\Frame = 0
    EndIf
  Next

  ; put up text for exiting
  StartDrawing(ScreenOutput())
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

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 FlipBuffers 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, PureBasic provides some timing commands that we can use to control animation speeds. You'll need to add a few fields to your structure in order to make this effective though. Let's take our *Balls* Structure and add to it:

```
Structure Balls
   X.w              ; track the X position of the ball
   Y.w              ; track the Y position of the ball
   XDir.b           ; track the X direction of the ball (0=left,1=right)
   YDir.b           ; track the Y direction of the ball (0=up,  1=down)
   Speed.b          ; track the speed of the ball
   Frame.b          ; track the frame we're using in the animation
   FrameTimer.b     ; time between frame changes?
   LastChanged.l    ; last frame change time?
EndStructure
```

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 *FrameTimer*, and we'll need to use the ElapsedMilliseconds command to get the current time after each animated frame. Then we'll need to include the following two lines when we're creating the ball instances:

```
    Ball()\FrameTimer = Random(100) + 30
    Ball()\LastChanged = ElapsedMilliseconds()
```

And then alter the section of code that handles the frame changing as follows:

```
      ; has sufficient time elapsed to change the frame on this ball?
  If Current_Time > Ball()\LastChanged + Ball()\FrameTimer
    ;if we're on Frame 0, move to 1
```

```
      If Ball()\Frame = 0
        Ball()\Frame = 1
      Else
        ; otherwise move to 0
        Ball()\Frame = 0
      EndIf
      ; make sure to reset the LastChanged time!
      Ball()\LastChanged = Current_Time
    EndIf
```

Now, here's the full source for our example:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprite
LoadSprite(0,"ballframe1.png")
LoadSprite(1,"ballframe2.png")

; get the width and height of the sprites
Sprite_Width  = SpriteWidth(0)
Sprite_Height = SpriteHeight(0)

Structure Balls
   X.w          ; track the X position of the ball
   Y.w          ; track the Y position of the ball
   XDir.b       ; track the X direction of the ball (0=left,1=right)
   YDir.b       ; track the Y direction of the ball (0=up,  1=down)
   Speed.b      ; track the speed of the ball
   Frame.b      ; track the frame we're using in the animation
   FrameTimer.b  ; time between frame changes?
   LastChanged.l ; last frame change time?
EndStructure

; setup a list of balls
NewList Ball.Balls()
```

```
; create a bunch of instances of balls
For BCounter = 0 To 50
   If AddElement(Ball()) <> 0
      Ball()\X = Random(#ScreenWidth - Sprite_Width)
      Ball()\Y = Random(#ScreenHeight - Sprite_Height)
      Ball()\XDir = Random(1)
      Ball()\YDir = Random(1)
      Ball()\Speed = Random(4) + 1
      Ball()\Frame = Random(1)
      Ball()\FrameTimer = Random(100) + 30
      Ball()\LastChanged = ElapsedMilliseconds()
   Else
      MessageRequester("Error!", "Unable to allocate memory for new element", ↵
   → #PB_MessageRequester_Ok)
      End
   EndIf
Next

Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ; get the current time for processing
  Current_Time = ElapsedMilliseconds()

  ForEach Ball()
    ; show our sprite (using transparency)
    DisplayTransparentSprite(Ball()\Frame,Ball()\X,Ball()\Y)

    ;;;;; First let's work with the X-Axis ;;;;;
    ; if we're moving left, subtract speed
    If Ball()\XDir = 0
      Ball()\X = Ball()\X - Ball()\Speed
    Else
      ; we must be moving right, add speed
      Ball()\X = Ball()\X + Ball()\Speed
    EndIf

    ; if we hit the right edge, start moving left
    If Ball()\X > #ScreenWidth - Sprite_Width
      Ball()\XDir = 0
    EndIf
    ; if we hit the left edge, start moving right
    If Ball()\X < 1
      Ball()\XDir = 1
    EndIf


    ;;;;; Now let's work with the Y-Axis ;;;;
    ; if we're moving up, subtract speed
```

```
    If Ball()\YDir = 0
      Ball()\Y = Ball()\Y - Ball()\Speed
    Else
      ; we must be moving down, add speed
      Ball()\Y = Ball()\Y + Ball()\Speed
    EndIf

    ; if we hit the bottom edge, start moving up
    If Ball()\Y > #ScreenHeight - Sprite_Height
      Ball()\YDir = 0
    EndIf
    ; if we hit the top edge, start moving down
    If Ball()\Y < 1
      Ball()\YDir = 1
    EndIf

    ;;;;; Finally let's update the frame  ;;;;;;
    ; has sufficient time elapsed to change the frame on this ball?
    If Current_Time > Ball()\LastChanged + Ball()\FrameTimer
      ;if we're on Frame 0, move to 1
      If Ball()\Frame = 0
        Ball()\Frame = 1
      Else
        ; otherwise move to 0
        Ball()\Frame = 0
      EndIf
      ; make sure to reset the LastChanged time!
      Ball()\LastChanged = Current_Time
    EndIf
  Next

  ; put up text for exiting
  StartDrawing(ScreenOutput())
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

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!

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:

(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, PureBasic is really drawing out a square. It's just ignoring the black pixels because the default *mask* is black (assuming you're using the DisplayTransparentSprite 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 we'll have a collision. This is because of how computers handle images. They are squares regardless of the shape the non-black (or mask color) pixels are. Consider:

(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, PB 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 SpriteCollision command is used to check if two images are overlapping on their respective bounding-boxes. Here is the format of this command:

SpriteCollision (*Sprite1,X,Y,Sprite2,X,Y****)***

Taking the bouncing ball example again, we can run through the *Balls* structure and check each element for overlap, as follows:

```
; first let's save the current instance of BALL()
CurrentBall = Ball()
; then let's store the X & Y values for the current ball
CurrentBallX = Ball()\X
CurrentBallY = Ball()\Y

; run through the list of balls
ForEach Ball()
  ; make sure we're not on the current ball
  If Ball() <> CurrentBall
```

```
      ; do a bounding box collision check
      If SpriteCollision(SpriteID,Ball()\X,Ball()\Y,SpriteID, ↵
      → CurrentBallX,CurrentBallY)
        ; we have an overlap, so add one to our variable
        Hits = Hits + 1
      EndIf
    EndIf
  Next
  ; set the Ball() pointer back to the current ball
  ChangeCurrentElement(Ball(),CurrentBall)
```

You'll want to take note that we are using a ForEach within another ForEach, and that both are using the same structure. So be sure to save the current position and then reset that position before ending the outside loop. If you don't, things will get very odd.

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. Here's the full code:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprite
SpriteID = LoadSprite(#PB_Any,"ball.png")

; get the width and height of the sprites
Sprite_Width  = SpriteWidth(SpriteID)
Sprite_Height = SpriteHeight(SpriteID)

Structure Balls
   X.w    ; track the X position of the ball
   Y.w    ; track the Y position of the ball
   XDir.b  ; track the X direction of the ball (0=left,1=right)
```

```
    YDir.b  ; track the Y direction of the ball (0=up,  1=down)
    Speed.b ; track the speed of the ball
EndStructure

; setup a list of balls
NewList Ball.Balls()

; create a bunch of instances of balls
For BCounter = 0 To 10
    If AddElement(Ball()) <> 0
      Ball()\X = Random(#ScreenWidth - Sprite_Width)
      Ball()\Y = Random(#ScreenHeight - Sprite_Height)
      Ball()\XDir = Random(1)
      Ball()\YDir = Random(1)
      Ball()\Speed = Random(2) + 1
    Else
      MessageRequester("Error!", "Unable to allocate memory for new element", ↲
      → #PB_MessageRequester_Ok)
      End
    EndIf
Next

Hits.w = 0

Repeat
   ; clear the screen
   ClearScreen(ClearColor)

   ForEach Ball()

     ; show our sprite (using transparency)
     DisplayTransparentSprite(SpriteID,Ball()\X,Ball()\Y)

     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     ; First let's work with the X-Axis
     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     ; if we're moving left, subtract speed
     If Ball()\XDir = 0
       Ball()\X = Ball()\X - Ball()\Speed
     Else
       ; we must be moving right, add speed
       Ball()\X = Ball()\X + Ball()\Speed
     EndIf

     ; if we hit the right edge, start moving left
     If Ball()\X > #ScreenWidth - Sprite_Width
       Ball()\XDir = 0
     EndIf
     ; if we hit the left edge, start moving right
     If Ball()\X < 1
```

```
    Ball()\XDir = 1

  EndIf

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ; Now let's work with the Y-Axis
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ; if we're moving up, subtract speed
  If Ball()\YDir = 0
    Ball()\Y = Ball()\Y - Ball()\Speed
  Else
    ; we must be moving down, add speed
    Ball()\Y = Ball()\Y + Ball()\Speed
  EndIf

  ; if we hit the bottom edge, start moving up
  If Ball()\Y > #ScreenHeight - Sprite_Height
    Ball()\YDir = 0
  EndIf
  ; if we hit the top edge, start moving down
  If Ball()\Y < 1
    Ball()\YDir = 1
  EndIf

  ; first let's save the current instance of BALL()
  CurrentBall = Ball()
  ; then let's store the X & Y values for the current ball
  CurrentBallX = Ball()\X
  CurrentBallY = Ball()\Y

  ; run through the list of balls
  ForEach Ball()
    ; make sure we're not on the current ball
    If Ball() <> CurrentBall
      ; do a bounding box collision check
      If SpriteCollision(SpriteID,Ball()\X,Ball()\Y,SpriteID, ↵
      → CurrentBallX,CurrentBallY)
        ; we have an overlap, so add one to our variable
        Hits = Hits + 1
      EndIf
    EndIf
  Next
  ; set the Ball() pointer back to the current ball
  ChangeCurrentElement(Ball(),CurrentBall)
Next

; put up text for exiting
StartDrawing(ScreenOutput())
 DrawText(0,0,"Hits: " + Str(Hits))
 DrawText(0,460,"Press any key To quit")
StopDrawing()
```

```
  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

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 PureBasic 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 lot! Fortunately PB first checks to see if the images even overlap before doing this, so there aren't any unnecessary checks. This is a great bonus because if you were coding this in, say, C or ActionScript, you would have to do this check on your own first. Not that it's incredibly difficult, but it's just cool that PB handles it for you!

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 PB. Just use the SpritePixelCollision command. It's called exactly the same as SpriteCollision, so if you just change SpriteCollision to SpritePixelCollision you'll be all set.

For fun, let's take our ball example and make it so there are only four balls on the screen. When two of them hit on a pixel-perfect check, we'll pause to show them and then reset the screen and go again. Also, we'll use red, green, blue, and yellow for our colors to track which ones have collided the most.

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
```

```
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprites
LoadSprite(0,"ball_blue.png")
LoadSprite(1,"ball_red.png")
LoadSprite(2,"ball_green.png")
LoadSprite(3,"ball_yellow.png")

; get the width and height of the sprites
Sprite_Width  = SpriteWidth(0)
Sprite_Height = SpriteHeight(0)

Structure Balls
   Color.b ; 0=blue,1=red,2=green,3=yellow
   X.w     ; track the X position of the ball
   Y.w     ; track the Y position of the ball
   XDir.b  ; track the X direction of the ball (0=left,1=right)
   YDir.b  ; track the Y direction of the ball (0=up,  1=down)
   Speed.b ; track the speed of the ball
EndStructure

; setup a list of balls
NewList Ball.Balls()

; create a bunch of instances of balls
For BCounter = 0 To 3
   If AddElement(Ball()) <> 0
      Ball()\Color = BCounter
      Ball()\X = Random(#ScreenWidth - Sprite_Width)
      Ball()\Y = Random(#ScreenHeight - Sprite_Height)
      Ball()\XDir = Random(1)
      Ball()\YDir = Random(1)
      Ball()\Speed = 3
   Else
      MessageRequester("Error!", "Unable to allocate memory for new element", ↵
      → #PB_MessageRequester_Ok)
      End
   EndIf
Next

; set up some variable to track hits
```

```
Hit.b = 0
Blue.w = 0
Red.w = 0
Green.w = 0
Yellow.w = 0

Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ForEach Ball()
    ;;;;; First let's work with the X-Axis ;;;;;
    ; if we're moving left, subtract speed
    If Ball()\XDir = 0
      Ball()\X = Ball()\X - Ball()\Speed
    Else
      ; we must be moving right, add speed
      Ball()\X = Ball()\X + Ball()\Speed
    EndIf

    ; if we hit the right edge, start moving left
    If Ball()\X > #ScreenWidth - Sprite_Width
      Ball()\XDir = 0
    EndIf
    ; if we hit the left edge, start moving right
    If Ball()\X < 1
      Ball()\XDir = 1
    EndIf

    ;;;;; Now let's work with the X-Axis ;;;;;
    ; if we're moving up, subtract speed
    If Ball()\YDir = 0
      Ball()\Y = Ball()\Y - Ball()\Speed
    Else
      ; we must be moving down, add speed
      Ball()\Y = Ball()\Y + Ball()\Speed
    EndIf

    ; if we hit the bottom edge, start moving up
    If Ball()\Y > #ScreenHeight - Sprite_Height
      Ball()\YDir = 0
    EndIf
    ; if we hit the top edge, start moving down
    If Ball()\Y < 1
      Ball()\YDir = 1
    EndIf

    ; first let's save the current instance of BALL()
    CurrentBall = Ball()
    ; then let's store the color, and X & Y values for the current ball
    CurrentBallC = Ball()\Color
```

```
  CurrentBallX = Ball()\X
  CurrentBallY = Ball()\Y

  ; run through the list of balls
  ForEach Ball()
    ; make sure we're not on the current ball
    If Ball() <> CurrentBall
      ; do a pixel-perfect collision check
      If SpritePixelCollision(Ball()\Color,Ball()\X,Ball()\Y, ↵
      → CurrentBallC,CurrentBallX,CurrentBallY)
        ; see which ones need adding to, and add to them!
        Select Ball()\Color
            Case 0
                Blue = Blue + 1
            Case 1
                Red = Red + 1
            Case 2
                Green = Green + 1
            Case 3
                Yellow = Yellow + 1
        EndSelect
        Select CurrentBallC
            Case 0
                Blue = Blue + 1
            Case 1
                Red = Red + 1
            Case 2
                Green = Green + 1
            Case 3
                Yellow = Yellow + 1
        EndSelect
        ; set a flag to say we've had a collision
        Hit = 1
        ; break out of this loop
        Break
      EndIf
    EndIf
  Next
  ChangeCurrentElement(Ball(),CurrentBall)

  ; show our sprite (using transparency)
  DisplayTransparentSprite(Ball()\Color,Ball()\X,Ball()\Y)
Next

; put up text for exiting
StartDrawing(ScreenOutput())
  BackColor(RGB(0,0,0))
  FrontColor(RGB(0,0,255))
  DrawText(0,0, "Blue: " + Str(Blue))

  FrontColor(RGB(255,0,0))
```

```
    DrawText(100,0,"Red: " + Str(Red))

    FrontColor(RGB(0,255,0))
    DrawText(200,0,"Green: " + Str(Green))

    FrontColor(RGB(255,255,0))
    DrawText(300,0,"Yellow: " + Str(Yellow))

    FrontColor(RGB(255,0,255))
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ; if we had a hit, delay to show the hit and then reset everything
  If Hit = 1
    Delay(500)
    ForEach Ball()
      Ball()\X = Random(#ScreenWidth - Sprite_Width)
      Ball()\Y = Random(#ScreenHeight - Sprite_Height)
      Ball()\XDir = Random(1)
      Ball()\YDir = Random(1)
    Next
    ; be sure to reset the hit flag!
    Hit = 0
  EndIf

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

The important parts in this program are that we're loading in four different balls and assigning them to the *Balls* structure; we're using counters (one per color) to keep track of which colors have hit; we've instituted a *Hit* flag to let us know when to pause and reset; we're now using the SpritePixelCollision command for checks; and we've moved the DisplayTransparentSprite command to the bottom of the loop so we can see the collisions as they happen.

You may see a point where two of the balls are said to be colliding, but to your eye they're just a little bit apart. This is due color-smoothing (also known as antialiasing). To make circles and lines look less jagged, paint programs use dimmer and dimmer colors on the edges until they fade out. Because of this, you may have a situation where two of these colors overlap and cause collision. Since these colors aren't exactly black (or whatever your mask color is) the system will register a collision. Even having an RGB value of 0,1,0 would not be black and would thus be considered collision-worthy.

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 KeyboardInkey 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 KeyboardPushed and KeyboardReleased commands come in. All these function do is return a TRUE or FALSE response when asked if a particular key is being held down or released. In order for you to send it a particular key to check, you need to know the key's *scancode*. This is a code that the computer recognizes the key by. Generally what I do is find the default PB constant (which can be found in the KeyboardPushed command Help area in the default PureBasic IDE) and use it as my argument.

The following code checks to see if either the left or right arrow has been hit. If so, it displays a value accordingly:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, mouse, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  KeyBeingHit.s = "None"
  If ExamineKeyboard()
    If KeyboardPushed(#PB_Key_Left)
      KeyBeingHit.s = "Left Arrow"
    EndIf
```

```
    If KeyboardPushed(#PB_Key_Right)
      KeyBeingHit.s = "Right Arrow"
    EndIf
  EndIf

  ; put up text for information
  StartDrawing(ScreenOutput())
    DrawText(0,0,"Use the Left and Right arrows!")
    DrawText(0,200,"Key = " + KeyBeingHit.s)
    DrawText(0,460,"Press ESCAPE key To quit")
  StopDrawing()

  FlipBuffers()

Until KeyboardReleased(#PB_Key_Escape)

End
```

For each check, we need to first examine our keyboard:

```
  If ExamineKeyboard()
```

If that responds that something has happened, then we can check the keys accordingly:

```
    If KeyboardPushed(#PB_Key_Left)
      KeyBeingHit.s = "Left Arrow"
    EndIf

    If KeyboardPushed(#PB_Key_Right)
      KeyBeingHit.s = "Right Arrow"
    EndIf
```

As you can see, I just pass in the scan codes for the left and right arrow keys and set the variable as needed.

You can replace these #PB values with any in the scan code list and change the way this program functions.

KeyboardPushed has no delays associated to it, so you will have an immediate response to your key presses.

Using the Mouse

The next device to touch on is the mouse. There are a number of different commands that can be used with the mouse, but the mouse command set itself is not daunting.

Our primary concerns are to examine the state of the mouse, see if there is any movement, where our mouse cursor is on the screen, what buttons have been pressed/released, and whether or not the mouse wheel (if the user has one) has been used. Fortunately, PB has all of these commands tidied up for us!

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, mouse, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitMouse() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

LastClicked.s = "None"
HeldDown.s = "None"

; load in the sprite
LoadSprite(0,"mouse.bmp")

MouseLocate(320,240)

; Keep going until the user hits a key
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  If ExamineMouse()
    X = MouseX()
    Y = MouseY()
    X_Movement = MouseDeltaX()
    If X_Movement < 0
      X_MouseMovement.s = "Left"
    ElseIf X_Movement > 0
      X_MouseMovement.s = "Right"
    Else
      X_MouseMovement.s = ""
    EndIf

    Y_Movement = MouseDeltaY()
    If Y_Movement < 0
      Y_MouseMovement.s = "Up"
    ElseIf Y_Movement > 0
      Y_MouseMovement.s = "Down"
    Else
```

```
      Y_MouseMovement.s = ""
    EndIf

    Wheel_Movement = MouseWheel()
    If Wheel_Movement < 0
      Wheel_MouseMovement.s = "Down"
    ElseIf Wheel_Movement > 0
      Wheel_MouseMovement.s = "Up"
    Else
      Wheel_MouseMovement.s = ""
    EndIf
    MB_Data.s = ""
    For MButton = 1 To 3
       If MouseButton(MButton)
         If MB_Data.s = ""
           MB_Data.s = Str(MButton)
         Else
           MB_Data.s = MB_Data.s + ", " + Str(MButton)
         EndIf
       EndIf
    Next
  EndIf

  ; show our mouse
  DisplaySprite(0,X,Y)

  ; put up text for information
  StartDrawing(ScreenOutput())
    DrawText(0,0,"Current Mouse Position: " + Str(X) + "," + Str(Y))
    DrawText(0,20,"X Movement = " + X_MouseMovement.s + ↵
   → ", Distance = " + Str(X_Movement))
    DrawText(0,40,"Y Movement = " + Y_MouseMovement.s + ↵
   → ", Distance = " + Str(Y_Movement))
    DrawText(0,60,"Wheel Movement = " + Wheel_MouseMovement.s + ↵
   → ", Distance = " + Str(Wheel_Movement))
    DrawText(0,80,"Buttons Pressed = " + MB_Data.s )
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

Obviously we must first use the InitMouse command at the top of our program so as to allow PB to set us up for mouse usage, or tell us we can't use it.

As with most of the commands in PureBasic, the first thing we call to check on status is Examine.

```
If ExamineMouse()
```

In this case we'll be calling the ExamineMouse command. This will check the status of the mouse, see if there were any updates since the last call, and prepare us for utilizing those updates.

Next we will want to keep track of the X and Y coordinates of the mouse:

```
X = MouseX()
Y = MouseY()
```

Both MouseX and MouseY will return a pixel position value for the mouse's hotspot, which is usually the tip of the mouse pointer.

Now we check to see if the mouse has moved since our last inspection:

```
X_Movement = MouseDeltaX()
If X_Movement < 0
  X_MouseMovement.s = "Left"
Else
  If X_Movement > 0
    X_MouseMovement.s = "Right"
  Else
    X_MouseMovement.s = ""
  EndIf
EndIf
```

MouseDeltaX will return a negative if the mouse moved left, a 0 if no movement on the X-axis, and a positive if the mouse was moved to the right. MouseDeltaY does the same thing, but for the Y-Axis. Negative = Moved up, 0 = Not moved, Positive = Moved down.

To check the movement of the wheel, we'll use the following:

```
Wheel_Movement = MouseWheel()
If Wheel_Movement < 0
  Wheel_MouseMovement.s = "Down"
Else
  If Wheel_Movement > 0
    Wheel_MouseMovement.s = "Up"
  Else
    Wheel_MouseMovement.s = ""
```

```
    EndIf
  EndIf
```

The MouseWheel command acts similarly to MouseDeltaY, but the values are reversed. Negative = Moved down, 0 = Not Moved, Positive = Moved up.

Now let's check the buttons:

```
MB_Data.s = ""
For MButton = 1 To 3
  If MouseButton(MButton)
    If MB_Data.s = ""
      MB_Data.s = Str(MButton)
    Else
      MB_Data.s = MB_Data.s + ", " + Str(MButton)
    EndIf
  EndIf
Next
```

Notice here that I'm running through and checking only three buttons. Some mice have more than three, so you may want to account for that. I'm just doing a simple For…Next loop and calling MouseButton on each value (1…2…3). If the value is non-zero, we have a click! If it *is* zero, then we don't.

Displaying a Custom Mouse Cursor

When you go from using the PureBasic IDE window for your game to using the Full-Screen, 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...PB 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 commands.

In our above example, I have already included the mouse pointer graphic. Here is the code piece that loads it in (look at the top of the source code):

```
; load in the sprite
MouseImageID = LoadSprite(#PB_Any,"mouse.bmp")
```

And then after all the mouse checks are completed, here is what to do to draw the mouse image:

```
; show our mouse
DisplaySprite(MouseImageID,X,Y)
```

That's it. What that will do is update your screen each frame with the mouse. You probably want this to be the last DisplaySprite command called in your game loop to be sure it stays on top of all the other images.

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 DisplaySprite. 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. First, of course, is the InitJoystick command. We don't want to go messing about with the joystick until we're sure that PureBasic is able to locate it.

To keep with our previous examples, let's create a program that keeps track of all our movements and button clicks:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, joystick, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitJoystick() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
   MessageRequester("Error!", "Unable to Initialize Environment", ↵
```

```
    → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

LastButton.s = "None"
OtherButton.s = "None"

; Keep going until the user hits a key
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  Button.s = ""
  If ExamineJoystick(0)
    ; check to see if there has been any movement on X
    X_Movement = JoystickAxisX(0)
    If X_Movement < 0
      X_JoystickMovement.s = "Left"
    Else
      If X_Movement > 0
        X_JoystickMovement.s = "Right"
      Else
        X_JoystickMovement.s = ""
      EndIf
    EndIf

    ; check to see if there has been any movement on Y
    Y_Movement = JoystickAxisY(0)
    If Y_Movement < 0
      Y_JoystickMovement.s = "Up"
    Else
      If Y_Movement > 0
        Y_JoystickMovement.s = "Down"
      Else
        Y_JoystickMovement.s = ""
      EndIf
    EndIf

    ; check to see if any of the buttons have been pressed
    For Buttons = 1 To 10
       If JoystickButton(0,Buttons)
         If Button.s = ""
           Button.s = Str(Buttons)
         Else
           Button.s = Button.s + ", " + Str(Buttons)
         EndIf
       EndIf
    Next
```

```
  EndIf

  ; put up text for exiting
  StartDrawing(ScreenOutput())
    DrawText(0,0,"Buttons Pressed: " + Button.s)
    DrawText(0,20,"X Movement = " + X_JoystickMovement.s )
    DrawText(0,40,"Y Movement = " + Y_JoystickMovement.s )
    DrawText(0,460,"Press any key To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

End
```

Again, the first thing to check during each iteration of our main loop is if there has been any joystick activity at all:

```
If ExamineJoystick()
```

If there has been, then we process the various options. There are few options to check on the Joystick: X, Y movement and button presses. That's it!

```
  ; check to see if there has been any movement on X
  X_Movement = JoystickAxisX()
  If X_Movement < 0
    X_JoystickMovement.s = "Left"
  Else
    If X_Movement > 0
      X_JoystickMovement.s = "Right"
    Else
      X_JoystickMovement.s = ""
    EndIf
  EndIf
```

Unlike our mouse checks, the JoystickAxisX and JoystickAxisY will provide predefined values to let us know what direction, if any, the joystick has been moved since the last check. For JoystickAxisX the values are: -1 = Moved left, 0 = Not moved, 1 = Moved right. For JoystickAxisY the values are: -1 = Moved up, 0 = Not moved, 1 = Moved down. Simple enough, eh?

Now we check on our buttons:

```
; check to see if any of the buttons have been pressed
For Buttons = 1 To 3
   If JoystickButton(Buttons)
     If Button.s = ""
       Button.s = Str(Buttons)
     Else
       Button.s = Button.s + ", " + Str(Buttons)
     EndIf
   EndIf
Next
```

Just like in the mouse example, I'm just checking for three of the joystick buttons here. By calling JoystickButton with the associated button to check, PB will respond if we have had a click or not. The PureBasic documentation states it can handle standard joysticks that have up to eight buttons.

If you'd like to expand this joystick demo a little bit, see if you can add in the mouse image and move it around with the joystick. It's a relatively simple task, but there is one primary thing you'll have to watch out for: there is no way to know where the X/Y position of the joystick is, because there's no such thing. So you'll have to somehow add and subtract to manually keep track of the two positions. Now go to it! ☺

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 initialize the sound environment. Just as you have done with the keyboard, mouse, joystick, etc., PureBasic requires that you first initialize the sound system. This ensures that your system is ready to play all the sounds you need it to. As with all the other initialization commands, you simply call the InitSound command to accomplish this. If the command returns a NON-ZERO value then you're all set. Otherwise, you should slap up a message and exit the program gracefully.

After the system is prepared to play sounds, you'll need a sound to load in. In our sample directory for this chapter, and under the "sounds" directory from there, I have placed a file called "explosion1.wav" for our use.

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

; Load in our .WAV file
SoundID = LoadSound(#PB_Any,"sounds/explosion1.wav")

; if the file loaded okay...
If SoundID
  StartDrawing(ScreenOutput())
    DrawText(0,0,"Playing an explosion sound...This program will exit ↵
  → automatically.")
  StopDrawing()
  FlipBuffers()
```

```
  ; play the sound
  PlaySound(SoundID)
Else
  MessageRequester("Error!", "Unable to load sound file",
#PB_MessageRequester_Ok)
  End
EndIf

; give a short delay so the sound can finish
Delay(5000)
End
```

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, PB will respond with a runtime error.

At the time of this writing, the only industry standard sound types supported are WAV, FLAC, OGG, by using a sound plug-in. As other types are supported, the simple inclusion of the plug-in for the new type will allow you to load and manipulate the sound. 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 OGG for in-game music.

In our above example, we load in a WAV file and then play it.

```
; Load in our .WAV file
SoundID = LoadSound(#PB_Any,"sounds/explosion1.wav")
```

This piece does the actual loading of the file. Note that the first parameter passed to the procedure is the numerical identifier we will be using for this sound. As with most PureBasic procedures, you can use the #PB_Any constant to have PB assign a valid numeric value. Then you can verify that the value loaded into *SoundPointer* is valid, or non-zero, before continuing on. Keeping with

```
  ; play the sound
  PlaySound(SoundID)
```

Now we simply play the sound. Assuming the sound loaded properly, you should hear an explosion when PB processes this command. There are a couple of options you have when using PlaySound. Firstly, if you want to play the sound only once, you may either call the command as shown above, or you can specify the optional *Mode* to use when playing.

```
; play the sound
PlaySound(SoundID,0)
```

This will act identically to the above snippet of code. However, if you want to have the explosion play over and over again, you would use a 1 (or use #PB_Sound_Loop) as the *Mode*, as follows:

```
; play the sound
PlaySound(SoundID,1)
```

Now our explosion will loop!

Another option is to play the sound in multichannel mode. Instead of stopping the sound that was just played, multichannel will use the same sound a play it on a different channel. This allows us to play the same sound on multiple channels at once. We could then use SoundVolume, SoundFrequency, SoundPan, and StopSound on the specific channel that the sound is playing on.

How do we get that channel, though? When setting up the PlaySound, it will return a result. If we pass along the #PB_Sound_MultiChannel flag during the call to play, the resultant value that is returned will be the channel for that sound.

```
; play the sound
SoundChannelID = PlaySound(SoundID,#PB_Sound_MultiChannel)
```

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 SoundFrequency.

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.

SoundVolume can be anywhere from 0 to 100—the higher the value, the louder the sound.

SoundPan is used to move the sound from one speaker to another. If you have a value of 0, which is the default, the sound will be played in the center, or equally loud on both speakers. If you go negative, you can have the sound play more loudly on the left side and more quietly on the right, up to a maximum of –100. The opposite is true for using positive numbers, up to 100.

SoundFrequency goes from 1000hz to 100000hz. 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 80000hz 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.

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Load in our .WAV file
SoundID = LoadSound(#PB_Any,"sounds/explosion1.wav")

; if our file loaded okay...
If SoundID
  ; initialize the Volume to half and set
  Sound1Volume = 50
  SoundVolume(SoundID,Sound1Volume)

  ; initialize the pan to center and set
  Sound1Pan = 0
  SoundPan(SoundID,Sound1Pan)

  ; initialize the pitch to highest and set
  Sound1Frequency = 44000
  SoundFrequency(SoundID,Sound1Pitch)
Else
  MessageRequester("Error!", "Unable to load sound file", ↵
   → #PB_MessageRequester_Ok)
  End
```

```
EndIf


; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ExamineKeyboard()
  ; if the user hits the space bar, play
  ; the sound
  If KeyboardPushed(#PB_Key_Space)
    PlaySound(SoundID)
  EndIf

  ; if the user hits the up arrow, increase
  ; the volume
  If KeyboardPushed(#PB_Key_Up)
    Sound1Volume = Sound1Volume + 1
    If Sound1Volume > 100
      Sound1Volume = 100
    EndIf
    SoundVolume(SoundID,Sound1Volume)
  EndIf

  ; if the user hits the down arrow, decrease
  ; the volume
  If KeyboardPushed(#PB_Key_Down)
    Sound1Volume = Sound1Volume - 1
    If Sound1Volume < 0
      Sound1Volume = 0
    EndIf
    SoundVolume(SoundID,Sound1Volume)
  EndIf

  ; if the user hits the right arrow,
  ; move the pan a bit to the right
  If KeyboardPushed(#PB_Key_Right)
    Sound1Pan = Sound1Pan + 1
    If Sound1Pan > 100
      Sound1Pan = 100
    EndIf
    SoundPan(SoundID,Sound1Pan)
  EndIf

  ; if the user hits the left arrow,
  ; move the pan a bit to the left
  If KeyboardPushed(#PB_Key_Left)
    Sound1Pan = Sound1Pan - 1
    If Sound1Pan < -100
      Sound1Pan = -100
```

```
    EndIf
    SoundPan(SoundID,Sound1Pan)
  EndIf


  ; if the user hits the right-control,
  ; raise the pitch slightly
  If KeyboardPushed(#PB_Key_RightControl)
    Sound1Frequency = Sound1Frequency + 1000
    If Sound1Frequency > 100000
      Sound1Frequency = 100000
    EndIf
    SoundFrequency(SoundID,Sound1Frequency )
  EndIf

  ; if the user hits the left-control,
  ; lower the pitch slightly
  If KeyboardPushed(#PB_Key_LeftControl)
    Sound1Frequency = Sound1Frequency - 1000
    If Sound1Frequency < 1000
      Sound1Frequency = 1000
    EndIf
    SoundFrequency(SoundID,Sound1Frequency)
  EndIf

  ; put up some text to explain usage
 ; put up text for information
 StartDrawing(ScreenOutput())
   DrawText(0,0,"Up Arrow = Increase Volume, Down Arrow = Decrease Volume")
   DrawText(0,16,"Left Arrow = Pan Left, Right Arrow = Pan Right")
   DrawText(0,32,"Left-Control = Decrease Pitch, Right-Control = Increase Pitch")
   DrawText(0,48,"Spacebar = Play Explosion sound")

   DrawText(0,100,"SOUND INFORMATION:")
   DrawText(0,116,"Sound Volume = " + Str(Sound1Volume))
   DrawText(0,132,"Sound Pan = " + Str(Sound1Pan))
   DrawText(0,148,"Sound Frequency = " + Str(Sound1Frequency))

   DrawText(0,400,"Press ESCAPE to quit")
  StopDrawing()

  FlipBuffers()

Until KeyboardReleased(#PB_Key_Escape)

End
```

Play around with that demo a bit (making sure that you have the "explosion1.wav" file in the proper location for loading) and you'll get a feel for the power of the sound commands.

Let's take a look at the specifics though:

```
; Load in our .WAV file
SoundID = LoadSound(#PB_Any,"sounds/explosion1.wav")

; if our file loaded okay...
If SoundID
  ; initialize the Volume to half and set
  Sound1Volume = 50
  SoundVolume(SoundID,Sound1Volume)

  ; initialize the pan to center and set
  Sound1Pan = 0
  SoundPan(SoundID,Sound1Pan)

  ; initialize the pitch to highest and set
  Sound1Frequency = 44000
  SoundFrequency(SoundID,Sound1Pitch)
```

After verifying that the WAV has properly loaded, we can do some basic setups of volume, pan, and frequency. We first setup a variable for each and then call the corresponding command to get it all set. Then as the user plays around in the program, we can adjust the values dynamically.

```
  ; if the user hits the right arrow,
  ; move the pan a bit to the right
  If KeyboardPushed(#PB_Key_Right)
    Sound1Pan = Sound1Pan + 1
    If Sound1Pan > 100
      Sound1Pan = 100
    EndIf
    SoundPan(SoundID,Sound1Pan)
  EndIf
```

Using the pan as an example, if the user presses the right arrow key, he'll hear the explosion *move* to the right speaker. This is extremely useful in making a game seem more realistic.

Imagine running along with your soldier through enemy territory when all the sudden you hear and explosion. If the panning is handled correctly, then you'll know that the explosion was off to your right, which will help you know to run to the left. Also, if the developer was thoughtful enough to control the volume and frequency of the sound based on a relative distance from you you'll have a better grasp of how far to the left you have to run!

Multiple Sounds Playing Simultaneously

Obviously having only one explosion able to go off at any given time is too much of a hindrance for any game developer worthy of their craft. So how do we handle multiple explosions then?

The method we'll use will call LoadSound to load the same sound in multiple times, using differing identifiers for each. This works fine, but you may want to control the number of allowable explosions to play simultaneously so you can maintain some level of control. In the following example, we will load up five explosion sounds for you to play with:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or ↵
   → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
  → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Load in our .WAV files
For ExplosionSound = 0 To 4
   If LoadSound(ExplosionSound,"sounds\explosion1.wav")
      ; set the Volume
      Volume = Random(75) + 10
      SoundVolume(ExplosionSound,Volume)

      ; set the pan
      Pan = Random(200)
      If Pan > 100
         Pan = 100 - Pan
      EndIf
      SoundPan(ExplosionSound,Pan)

      ; set the frequency (pitch)
      Frequency = Random(50000) + 10000
      SoundFrequency(ExplosionSound,Frequency)
   Else
      MessageRequester("Error!", "Unable to load sound file",
#PB_MessageRequester_Ok)
      End
   EndIf
Next
```

```
; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ExamineKeyboard()
  ; if the user hits the appropriate key, play the appropriate sound
  If KeyboardPushed(#PB_Key_1)
    PlaySound(0)
  EndIf

  If KeyboardPushed(#PB_Key_2)
    PlaySound(1)
  EndIf

  If KeyboardPushed(#PB_Key_3)
    PlaySound(2)
  EndIf

  If KeyboardPushed(#PB_Key_4)
    PlaySound(3)
  EndIf

  If KeyboardPushed(#PB_Key_5)
    PlaySound(4)
  EndIf

  ; put up text for information
  StartDrawing(ScreenOutput())
   DrawText(0,0,"Use keys 1 - 5 to play the various explosions")

   DrawText(0,400,"Press ESCAPE to quit")
  StopDrawing()

  FlipBuffers()

Until KeyboardReleased(#PB_Key_Escape)

End
```

Notice how we're able to have a number of sounds playing at the same time? And all this done from loading right from the same WAV file. But this isn't the best way to handle things, is it? Nope. A better way is to load a sound once and then just use it repeatedly as it's needed.

To do this, we will first need to learn how to load sounds into memory.

Loading Sounds into Memory

While you can certainly get by with using LoadSound and pull the sound file straight from the disk, another method you may consider is including

the file in memory and then "catching" it to a sound channel. To do this, you'll use the CatchSound command. Consider the following:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
    → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Load in our .WAV files from the DataSection
For ExplosionSound = 0 To 4
  If CatchSound(ExplosionSound,?ExplosionSound)
    ; set the Volume
    Volume = Random(75) + 10
    SoundVolume(ExplosionSound,Volume)

    ; set the pan
    Pan = Random(200)
    If Pan > 100
      Pan = 100 - Pan
    EndIf
    SoundPan(ExplosionSound,Pan)

    ; set the frequency (pitch)
    Frequency = Random(50000) + 10000
    SoundFrequency(ExplosionSound,Frequency)
  Else
    MessageRequester("Error!", "Unable to load sound file",
#PB_MessageRequester_Ok)
    End
  EndIf
Next

; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ExamineKeyboard()
  ; if the user hits the appropriate key, play the appropriate sound
  If KeyboardPushed(#PB_Key_1)
    PlaySound(0)
  EndIf
```

```
  If KeyboardPushed(#PB_Key_2)
    PlaySound(1)
  EndIf

  If KeyboardPushed(#PB_Key_3)
    PlaySound(2)
  EndIf

  If KeyboardPushed(#PB_Key_4)
    PlaySound(3)
  EndIf

  If KeyboardPushed(#PB_Key_5)
    PlaySound(4)
  EndIf

  ; put up text for information
  StartDrawing(ScreenOutput())
    DrawText(0,0,"Use keys 1 - 5 to play the various explosions")
    DrawText(0,400,"Press ESCAPE to quit")
  StopDrawing()

  FlipBuffers()

Until KeyboardReleased(#PB_Key_Escape)

End

DataSection
  ExplosionSound:
    IncludeBinary "sounds\explosion1.wav"
EndDataSection
```

The two pertinent pieces here are the *DataSection* and the use of the CatchSound command.

```
DataSection
  ExplosionSound:
    IncludeBinary "sounds\explosion1.wav"
EndDataSection
```

Here is where we actually load in the WAV file for processing. At this point PB has no idea that this is a sound file. It's just loading a binary file into memory.

```
    If CatchSound(ExplosionSound,?ExplosionSound)
```

This is where PB learns what type of tile it is and then loads it in. Note the use of the *?ExplosionSound* argument. The "*?*" signifies a pointer to a label, which, of course, is *ExplosionSound*. ☺

CatchSound behaves exactly as LoadSound does, but instead of using a file to load in the sound, it uses memory. This can be quite useful for when you decide to pack all of your sounds into one file, or tack them on to the end of your executable. You'll be able to load the portion of the packed file directly into memory and then use CatchSound to actually load the sound in.

But, again, we are kind of stuck here because if you press the number "1" over and over again, you'll hear that explosion sound cut off and restart. That's not what we want ultimately though. We want to have multiple sounds playing, overlapping each other, even if they're the same sound.

Overlaying Multiple Sounds

In order to get multiple sounds to overlap we're going to have to be a little sneaky. Actually, with most things in game development you end up having to be a little sneaky…it's the nature of the beast!

In this case our sneakiness will take us into the realm of using binary loading of a WAV file within a data section, the CatchSound command that we just touched on, the use of a structure with a list, and some varied sound commands for effect.

Here is the code:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; setup our structure for the sounds
Structure SoundController
  ID.l
  Length.l
  StartTime.l
EndStructure
```

```
; make sure we have a list for the sounds
Global NewList Sounds.SoundController()

;*************************************************
;  Procedure: MyPlaySound()
;     Author: Derlidio Siqueira & John Logsdon
;   Last Upd: 2/07/2005
;    Purpose: Adds a sound to the Sounds list and
;             randomizes it's volume, pan, and frequency values
;       Args: n/a
;    Returns: n/a
;   Comments: None
;*************************************************
Procedure MyPlaySound()

  ; first grab the sound out of memory
  ID = CatchSound(#PB_Any,?ExplosionSound)

  ; then set the approximate time for the sound
  Length.l = 5000

  ; if a valid ID is found
  If ID
    ; add the sound to the list
    AddElement(Sounds())

    ; then populate the list
    Sounds()\ID = ID
    Sounds()\Length = Length.l
    Sounds()\StartTime = ElapsedMilliseconds()

    ; set up some randomness for effect
    Volume = Random(50) + 50
    SoundVolume(ID,Volume)
    Pan = Random(200) - 100
    SoundPan(ID,Pan)
    Frequency = Random(30000) + 30000
    SoundFrequency(ID,Frequency)

    ; play the sound
    PlaySound(ID)

  EndIf

EndProcedure

;*************************************************
;  Procedure: MyReleaseSound()
;     Author: Derlidio Siqueira
;   Last Upd: 2/07/2005
```

```
  ;    Purpose: Removes a sound from the Sounds list
  ;       Args: n/a
  ;    Returns: n/a
  ;   Comments: This will kill a sound regardless if
  ;             it is still playing or not. It kills
  ;             the sound based on the time value
  ;             given in MyPlaySound()
  ;-***************************************************
  Procedure MyReleaseSound()
    ; run through each of the sounds
    ForEach(Sounds())
      ; if a particular one has expired (time-wise)
      If ElapsedMilliseconds() - Sounds()\StartTime > Sounds()\Length
        ; stop the sound, remove the sound, and delete the element
        ; from the list
        StopSound(Sounds()\ID)
        FreeSound(Sounds()\ID)
        DeleteElement(Sounds())
      EndIf
    Next
  EndProcedure

; Keep going until the user hits ESCAPE
Repeat
    ; clear the screen
    ClearScreen(ClearColor)

    ExamineKeyboard()
    ; if the user hits the spacebar, play the sound
    If KeyboardReleased(#PB_Key_Space)
      MyPlaySound()
    EndIf

    ; put up text for information
    StartDrawing(ScreenOutput())
      DrawText(0,0,"Press the spacebar as much as you want to make explosions!")
      DrawText(0,400,"Press ESCAPE to quit")
    StopDrawing()

    FlipBuffers()
    MyReleaseSound()
Until KeyboardReleased(#PB_Key_Escape)

End

DataSection
; where we load in the explosion sound
ExplosionSound: IncludeBinary "sounds\explosion1.wav"
EndDataSection
```

Let's examine the structure:

```
; setup our structure for the sounds
Structure SoundController
  ID.l
  Length.l
  StartTime.l
EndStructure

; make sure we have a list for the sounds
Global NewList Sounds.SoundController()
```

All we're really storing here is an ID value that will be returned by PureBasic when we go to catch the sound, the length that the sound will play in milliseconds (which is an estimate you'll need to make for each sound as PB has no built-in functionality for this), and the time, in milliseconds, when the sound began playing. And finally we have to create a list in order to make this structure useful to us. Remember that since we're using this list outside of its normal scope, we need to make it Global.

Now we have to actually get the sound into the list:

```
Procedure MyPlaySound()

   ; first grab the sound out of memory
   ID = CatchSound(#PB_Any,?ExplosionSound)

   ; then set the approximate time for the sound
   Length.l = 5000
```

We use #PB_Any to inform PB that we want it to handle the creation of an unused value for our sound ID, and we use CatchSound to grab the sound from memory. Then we set an approximate (or specific, if you really want to) value to allow the sound to play.

Now we load up the list:

```
   ; if a valid ID is found
   If ID
     ; add the sound to the list
     AddElement(Sounds())

     ; then populate the list
     Sounds()\ID = ID
     Sounds()\Length = Length.l
     Sounds()\StartTime = ElapsedMilliseconds()
```

First make sure you verify that an ID was successfully created. If you compare the ID returned and it's a zero (0), then you don't have a sound to play with and you'll get errors!

Call the AddElement command in order to open up a new slot in the sounds list. Then populate them with the values we've already retrieved, making sure to get the current time for the *StartTime* structure element by calling ElapsedMilliseconds.

Since we still have the ID of the sound that was just created, let's have a little fun and mess around with the volume, pan, and frequency.

```
; set up some randomness for effect
Volume = Random(50) + 50
SoundVolume(ID,Volume)
Pan = Random(200) - 100
SoundPan(ID,Pan)
Frequency = Random(30000) + 30000
SoundFrequency(ID,Frequency)
```

And then, finally, we play the sound!

```
; play the sound
PlaySound(ID)
```

Now that our playing of the sound is running fine, we must turn our attention to removing the sound when it's done playing. To do that we must first run through the entire list of sounds and see what has expired.

```
Procedure MyReleaseSound()

  ; run through each of the sounds
  ForEach(Sounds())
    ; if a particular one has expired (time-wise)
    If ElapsedMilliseconds() - Sounds()\StartTime > Sounds()\Length
```

This little bit of code will take the time that the sound started, subtract that from the number of milliseconds since the computer was powered up, and then see if the resultant value is greater than the length we prescribed this sound to play.

If so, then we stop the sound, free it from memory, and release the list element that contained the ID, duration, and start timer information on that sound.

```
; stop the sound, remove the sound, and delete the element from the list
StopSound(Sounds()\ID)
FreeSound(Sounds()\ID)
DeleteElement(Sounds())
```

And that's all there is to it!

Keep this in mind as it will be very useful for all your game sounds, and there will be many!

Playing Music

Music files are not entirely different than sound files. Actually, PB treats them the same way. For example, if you were to convert the explosion WAV file into an OGG file (and call on the appropriate plug-in) it would play it just fine. After all, a WAV file can be used for music. The problem is that WAV files tend to be rather large, whereas OGG files are compressed. Compression is good, assuming it's done well. OGG files sound excellent, they're relatively small, and they're royalty free to use in your games (at least at the time of this writing). So they're a good choice for developers.

The following example demonstrates how similar the setup is for playing music as it is for how we played the explosion sound:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; make sure we use the appropriate plug-in!
UseOGGSoundDecoder()

; Load in our .OGG file
MusicID = LoadSound(#PB_Any,"music\song.ogg")

If MusicID
  ; play the sound
  PlaySound(MusicID)
Else
```

```
  MessageRequester("Error!", "Unable to load sound file",
#PB_MessageRequester_Ok)
  End
EndIf

Repeat
  ClearScreen(ClearColor)
  StartDrawing(ScreenOutput())
    DrawText(0,0,"Playing the tune...press any key to exit")
  StopDrawing()
  FlipBuffers()
  ExamineKeyboard()
Until KeyboardReleased(#PB_Key_All)

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!

Music Modules

Many people use modules (also known as "mods") to make their music. The good news is that PureBasic supports the use of mods, so many people may find that a useful thing to know.

PureBasic uses a module plugin (ModPlug XMMS) for all its mod uses. Thus, to keep up to date on this mod's requirements and limitations, please refer to your PB documentation (Help file), looking specifically under the LoadModule command, or go to the ModPlug website: http://modplug-xmms.sourceforge.net/

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard, sound, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or InitSound() = 0 Or  InitModule() = 0 ↵
   → Or OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Thanks to Derlidio for donating this MOD!
```

```
; Load up he module, give it the id of 1
ModID = LoadModule(#PB_Any,"sounds\music.it")

; play the module
PlayModule(ModID)

Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ExamineKeyboard()

  ; put up text for information
  StartDrawing(ScreenOutput())
    DrawText(0,0, "Now playing: Derlidio's 'Get Ready!' mod :)")
    DrawText(0,400,"Press ESCAPE to quit")
  StopDrawing()

  FlipBuffers()

Until KeyboardReleased(#PB_Key_Escape)

; close down the module system
StopModule(ModID)

End
```

If you're completely unfamiliar with mods, you may consider checking out the following site:

- http://www.modarchive.com/

You'll be able to find tons of songs, useful information, and even some links to nice mod software packages.

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 computer 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, but it will take some work on your part.

Frames per Second (FPS) Tracking

You will need is a way to track how fast your FPS is in your game. Then you have 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 your game 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:

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
```

```
#ScreenHeight = 480

; Initialize the sprite and keyboard, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title", ↵
    → #PB_Screen_NoSynchronization) = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; setup our timer and init it's value
FPSTimer = ElapsedMilliseconds()

; setup our tracking variables
FPS = 0
FPSCounter = 0

; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
 ClearScreen(ClearColor)

 If StartDrawing(ScreenOutput())
   For lines = 0 To 500
     ; choose random colors
     r = Random(254)
     g = Random(254)
     b = Random(254)
     ; draw out the line in the selected color, at random places and sizes
     Line(Random(639),Random(449),Random(150),Random(150),RGB(r,g,b))
   Next
   ; show the current FPS
   DrawText(0,460,"FPS = " + Str(FPS) + "  --  Press ESC to exit...")
 Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
   → #PB_MessageRequester_Ok)
   End
 EndIf

 StopDrawing()

 ; if the current time is 1000 milliseconds (1 second)
 ; past the starting timer
 If ElapsedMilliseconds() > FPSTimer + 1000
   ; set the FPS var to the FPS counter
   FPS = FPSCounter
   ; reset the counter
   FPSCounter = 0
```

```
    ; reset the timer
    FPSTimer = ElapsedMilliseconds()
  Else
    ; otherwise, ad 1 to the counter variable
    FPSCounter = FPSCounter + 1
  EndIf

  ExamineKeyboard()

  ; to make it so PB doesn't enable syncing, use FlipBuffers with a 0
  FlipBuffers()

Until KeyboardReleased(#PB_Key_Escape)

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 330FPS. Change the number of lines in the FOR…NEXT loop and see how the FPS changes. Also, remove the #PB_Screen_NoSynchronization argument from OpenScreen function and you'll see it get locked to your screen's refresh rate. When I remove that argument (or set it to 1), I get 60FPS with an occasional 59FPS.

So why not just use synchronization? Because not all machines will be able to play your game fast enough to use that high of a rate. It could very well be that many machines will be playing your game at well over 60FPS, so this may be a halfway decent solution. But what if the monitor's refresh rate is 72 for a particular computer? Then you'll be running it on that machine at 72FPS. But, again, if the machine that you're game is running on is very slow, your end-user may only see 30FPS. So we need a better solution.

Another 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 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 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.

What we need to 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 = ElapsedMilliseconds()

ClearColor = RGB(0,0,0)

; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
 ClearScreen(ClearColor)

  ;what's the difference in time since our last check?
   ElapsedTime = ElapsedMilliseconds() - 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=ElapsedMilliseconds()
   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.

```
Main_Timer = ElapsedMilliseconds()
```

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 an i3 chip is the equivalent to 1 full second on an i7 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=ElapsedMilliseconds()
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. But used in combination with the OpenScreen that doesn't use the #PB_NoSynchronization option in PureBasic, and you may have something good going!

Locking in at Real Time

I have seen a number of games using the Real Time method. I have tried it myself with great success as well.

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.

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize the sprite and keyboard, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; setup our timers and init their value
FPSTimer = ElapsedMilliseconds()
Start_Time.f = ElapsedMilliseconds()
```

```
; setup our FPS tracking variables
FPS = 0
FPSCounter = 0

; setup our real time tracking variables
XUnit.f = 0.250
YUnit.f = 0.125

; setup our starting points
X.f = 0
Y.f = 0

; Keep going until the user hits ESCAPE
Repeat
  ; clear the screen
 ClearScreen(ClearColor)
 ; get the current time
 Current_Time.f = ElapsedMilliseconds()
 ; determine how much time has passed
 Time_Passed.f = Current_Time - Start_Time
 ; reset the start time
 Start_Time = Current_Time
 ; Add X's current value to the number of units it is to move
 ; multiplied the amount of time that has passed
 X = X + (XUnit * Time_Passed)
 ; if the resultant value goes beyond the screen, reset to 0
 If X > #ScreenWidth
   X = 0
 EndIf

 ; Add Y's current value to the number of units it is to move
 ; multiplied the amount of time that has passed
 Y = Y + (YUnit * Time_Passed)
 ; if the resultant value goes beyond the screen, reset to 0
 If Y > #ScreenHeight
   Y = 0
 EndIf


 If StartDrawing(ScreenOutput())
   ; draw out the box
   Box(X,Y,20,20,RGB(255,0,255))

   ; show the current FPS
   DrawText(0,460,"FPS = " + Str(FPS) + "  --  Press ESC To exit...")
 Else
   MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
   → #PB_MessageRequester_Ok)
   End
 EndIf
```

```
  StopDrawing()

  ExamineKeyboard()

  FlipBuffers()

  ; if the current time is 1000 milliseconds (1 second)
  ; past the starting timer
  If ElapsedMilliseconds() > FPSTimer + 1000
    ; set the FPS var to the FPS counter
    FPS = FPSCounter
    ; reset the counter
    FPSCounter = 0
    ; reset the timer
    FPSTimer = ElapsedMilliseconds()
  Else
    ; otherwise, ad 1 to the counter variable
    FPSCounter = FPSCounter + 1
  EndIf

Until KeyboardReleased(#PB_Key_Escape)

End
```

Now, if you use that method and set up a field in a structure to handle all these units, 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.

```
; setup our Screen Width and Height here for easier tracking
#ScreenWidth = 640
#ScreenHeight = 480

; Initialize the sprite and keyboard, and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)
```

```
; setup our structure for the boxes
Structure BoxStructure
  X.f : Y.f
  Width.l : Height.l
  r.l : g.l : b.l
  XUnits.f : YUnits.f
EndStructure

; make sure we have a list for the boxes
Global NewList Boxes.BoxStructure()

;************************************************
;  Procedure: CreateBoxes()
;     Author: John Logsdon & Derlidio Siqueira
;    Purpose: Creates a new box with random info
;       Args: Amount - Number of boxes to create
;************************************************
Procedure CreateBoxes(Amount.l)
  For i = 0 To (Amount - 1)
    If AddElement(Boxes()) <> 0
      Boxes()\X = 0
      Boxes()\Y = 0
      Boxes()\r = Random(254)
      Boxes()\g = Random(254)
      Boxes()\b = Random(254)
      Boxes()\Width = Random(25) + 5
      Boxes()\Height = Random(25) + 5
      Boxes()\XUnits = ((Random(1000) * 0.375)/1000.0 + 0.050)
      Boxes()\YUnits = ((Random(1000) * 0.375)/1000.0 + 0.050)
    Else
      MessageRequester("Error!", "Unable to add element", ↵
        → #PB_MessageRequester_Ok)
      End
    EndIf
  Next
EndProcedure

;************************************************
;  Procedure: UpdateBoxes()
;     Author: John Logsdon & Derlidio Siqueira
;    Purpose: Moves our boxes around the screen
;       Args: Modifier - The time since last call
;************************************************
Procedure UpdateBoxes(Modifier.f)
  ForEach(Boxes())
    ; Add X's current value to the number of units it is to move
    ; multiplied the amount of time that has passed
    Boxes()\X = Boxes()\X + (Boxes()\XUnits * Modifier)
    ; if the resultant value goes beyond the screen, reset to 0
    If Boxes()\X > #ScreenWidth
      Boxes()\X = 0
```

```
    EndIf

    ; Add Y's current value to the number of units it is to move
    ; multiplied the amount of time that has passed
    Boxes()\Y = Boxes()\Y + (Boxes()\YUnits * Modifier)
    ; if the resultant value goes beyond the screen, reset to 0
     If Boxes()\Y > #ScreenHeight
        Boxes()\Y = 0
     EndIf

    If StartDrawing(ScreenOutput())
      ; draw the box
      Box(Boxes()\X,Boxes()\Y,Boxes()\Width,Boxes()\Height,RGB(Boxes()\r, ↵
    → Boxes()\g,Boxes()\b))
    Else
      MessageRequester("Error!", "Unable to Draw to ScreenOutput()",  ↵
       →  #PB_MessageRequester_Ok)
      End
    EndIf

    StopDrawing()
  Next
EndProcedure

; setup our timers and init their value
FPSTimer = ElapsedMilliseconds()
Start_Time.f = ElapsedMilliseconds()

; setup our FPS tracking variables
FPS = 0
FPSCounter = 0

; create some boxes
CreateBoxes(100)

; Keep going until the user hits ESCAPE
Repeat
   ; clear the screen
  ClearScreen(ClearColor)
  ; get the current time
  Current_Time.f = ElapsedMilliseconds()
  ; determine how much time has passed
  Time_Passed.f = Current_Time.f - Start_Time.f
  ; reset the start time
  Start_Time.f = Current_Time.f

  ; update and draw the boxes to the screen
  UpdateBoxes(Time_Passed.f)

  If StartDrawing(ScreenOutput())
```

```
    ; show the current FPS
    DrawText(0,460,"FPS = " + Str(FPS) + "  --  Press ESC to exit...")
  Else
    MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
    → #PB_MessageRequester_Ok)
    End
  EndIf

  StopDrawing()

  ExamineKeyboard()

  FlipBuffers()

  ; if the current time is 1000 milliseconds (1 second)
  ; past the starting timer
  If ElapsedMilliseconds() > FPSTimer + 1000
    ; set the FPS var to the FPS counter
    FPS = FPSCounter
    ; reset the counter
    FPSCounter = 0
    ; reset the timer
    FPSTimer = ElapsedMilliseconds()
  Else
    ; otherwise, ad 1 to the counter variable
    FPSCounter = FPSCounter + 1
  EndIf

Until KeyboardReleased(#PB_Key_Escape)

End
```

There are a couple of things in this code that I want to touch on.

```
    Boxes()\XUnits = ((Random(1000) * 0.375)/1000.0 + 0.050)
    Boxes()\YUnits = ((Random(1000) * 0.375)/1000.0 + 0.050)
```

You may be wondering what is going on here. The issue is that the Random command will not return a precision decimal value. If a *float* is used as the destination for the return, the value will be made a decimal (like 500.0), but only whole values are returned. We need precision, so we use a little math.

Here is how it looks broken down:

```
    Value.f = Random(1000)
```

That, obviously, will return a value between 0.0 and 1000.0. Let's imagine that we were returned the value of 425.0.

```
Value.f = Value.f * 0.375
```

...or...

```
Value.f = 425.0 * 0.375  = 159.375
```

So now we have a value resultant value of 159.375. That's far too high for the speeds we want. So we have to get rid of the values to the left of the decimal. Since we multiplied by a three decimal value, we will need to divide the resultant value by 1,000. If we had use a two decimal value in our multiplication (i.e. 425.0 * 0.25), then we would divide the result by 100…and so on.

```
Value.f = Value.f/1000
```

... or ...

```
Value.f = 159.375/1000 = 0.159375
```

And from here we want to add any differential (mostly to ensure we get a non-zero value). In this case, let's use 0.050.

```
Value.f = Value.f + 0.050
```

... or ...

```
Value.f = 0.159375 +0.050 = 0.209375
```

Which means that for every time-interval that passes, we will be moving this particular object 0.209375 pixels. As you can see, this means that roughly every 10 time-intervals, this object will move one pixel over. Now that's control!

If you use a combination of the techniques discussed in this chapter, you should be able to get a really good handle on speed control across various computers. But the main thing to do is test, test, test: have your friends test on all their differing machines until you find a solution that works best for your style of game.

PART 3:
Migz Callo: Laser Blazer

Chapter 18: Game Design

One thing that's almost always overlooked is game design. People typically start out just doing a little bit of this and a little bit of that and before long they have a game shaping up. Of course when they go back to look at their game, it's often in need of a revamp. This usually occurs because there wasn't a real strong idea of where the game was going.

Now in the case of copying another game, most of the work is already done for you.

For example, if you were to make an Asteroids clone, do you really need a full design document for that? Probably not, but if you're planning to really expand on the original game, you may want to at least consider some asset lists and balancing issues.

Another benefit of making a design document is that it often helps you see some items that will be affect later but what you do now. Imagine spending a few days wrestling with collision concepts on a predefined sprite size. You get it all figured out—finally—and you're all happy. And then you get that next piece of art that has a totally different size and it doesn't want to play nicely with your collision routines. If you had spent the time designing the collision routines more openly you'd still be okay, but at the time of coding it wasn't something you had to worry about. Fortunately, you don't really need to worry about collision routines anyway since PB already handles them…it was just an example!

So, with that, here is a simple game design for the demo game:

Title: *Migz Callo: Laser Blaser*
Game design: John "Krylar" Logsdon
Art design: Ric "Putty" Lumb
Music: Steve "Fash" Harrison
Game type: 2D Side-Scroller, futuristic shoot'em-up

Background Story

Top of his class at Rignally's School for Robotic Decimation, Migz Callo was the natural selection for dealing with the machine-overtake at Cyclopticbots, Inc.

Cyclopticbots had spent the better part of two cycles developing robots that would have highly advanced decision-making abilities, while retaining their natural metallic charm. Unfortunately, the plan went awry. The robots made less than thoughtful choices on most occasions, and were prone to laziness and irritability. Though not nearly as intelligent as their designers had hoped, the robots were good with guns, and they seemed more than willing to use them. Within weeks of the first batch, the ill-tempered robots had taken up their weapons and driven all the scientists and support personnel away.

Finally, after many failed attempts of Cyclopticbot's own security forces, the decision was made to acquire the best and brightest from Rignally's.

Most graduates were placed on a team for their first runs, but not the academically acclaimed "Laser Blazer." No, Migz Callo had been special from day one. His ability to bolt up and down ladders faster than any of his peers alone made him somewhat of a notable figure. But it was his quick-draw speed with the laser that really turned heads. Where most could drag a weapon at 50% the speed of a robot, Migz—more often than not—matched or bettered the bots. Even veterans were amazed at the pull-speed the young man had.

So when the vets were approached to take on the Cyclopticbot's job, they tipped their hats toward the Laser Blazer…most in the hopes of seeing the young hotshot fail.

Migz took the job, got on the ship, beamed to the clearest door in the station, strapped on his gear, adjusted his goggles, and donned his trademarked mischievous grin.

Now's the time to do the job he's been trained for: Total Robotic Decimation.

Game Features

Here is a brief list of what we're going for in this game. Keep in mind that this is a demo game!

- Three demo levels to nuke robots
- Challenging and addictive game play
- Cool futuristic space station artwork, robots, and good 'ol Migz himself
- Awesome soundtrack by Steve "Fash" Harrison
- If you own the "Programming 2D Scrolling Games" book, you'll also have the full source-code and chapters explaining how the game was made
- Full 2D level designer (no source) supporting the game's 32x32 tileset or any player-made tilesets of the same size

Art Asset List

All transparencies are Magenta (255,0,255) so PB will know which colors to process.

- Cheezy little space backdrop (snagged from NSG space game)

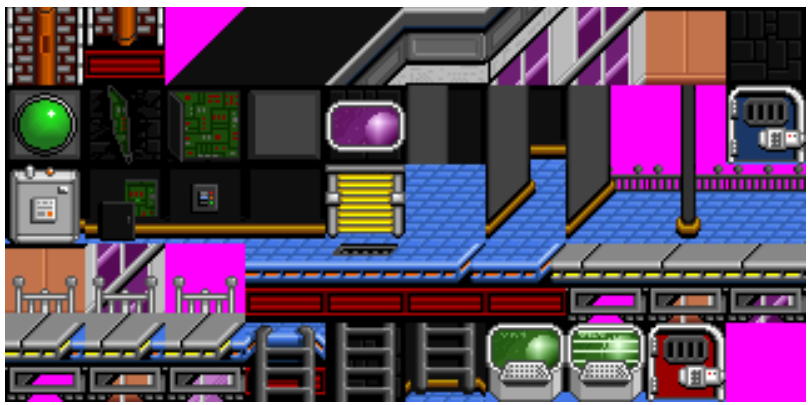
- User Interface for game-play. Just a basic wrapper for the game window, maybe with the main character thrown in and the game name too.

- Splash screen ("Migz Callo: Laser Blazer")

- Story Screen – Just something to give the basic idea of the game

- How to play – This is a tips and tricks page

- Tile images (32x32) -- Futuristic Looking
    - Walls (including edge-of-map versions for corners). Cracked pieces, power switches, etc.
    - Ceiling (for top of map)
    - Floors (with edge for depth)
    - Floor grill (with edge for depth)
    - Computer Wall Tiles
    - Door (one entry, one exit)
    - Ladder (need bottom, center, and top pieces)

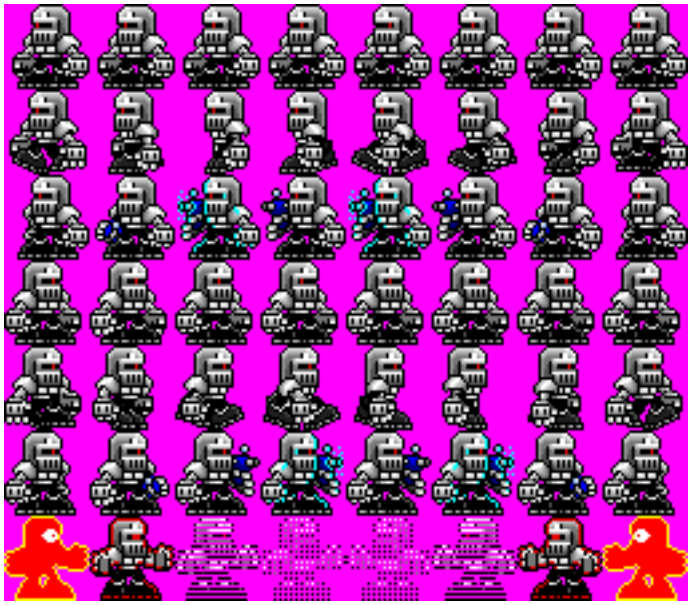


- Player Sprite – Space Spy (32x32)

- I see the player as wearing a baggy black spacesuit, blaster in hand, a pair of goggles, and a headband that his disheveled hair pokes out and hangs over.
- 8 Frames walking left
- 8 Frames walking right
- 8 Frames climbing
- Frames standing still, facing left, drawing and firing gun
- Frames standing still, facing right, drawing and firing gun
- Frame of the player being hit
- Frames player disintegrating
- Frames of the player fidgeting (cause the keys aren't moving)
- Frames of the player sleeping

- NPC Sprite Robot (32x32)
    - Silver, one red-light eye that sweeps back and forth
    - 8 Frames walking left
    - 8 Frames walking right
    - Frames standing still, facing left, drawing and firing gun
    - Frames standing still, facing right, drawing and firing gun
    - Frame of the robot getting hit

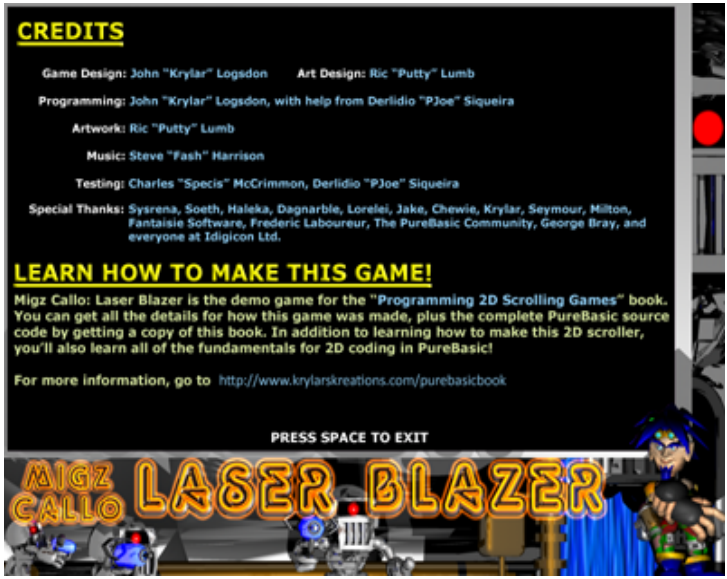
- Frames robot disintegrating



- Blue laser bolt (8x8)
- Red laser bolt (8x8)
- Blue mini-laser explosion (8x8) for laser hitting target and walls
- Red mini-laser explosion (8x8) for laser hitting target and walls

- HealthPak (16x16)

- Credits Screen (names and duties to be provided)



Sound Asset List

- Laser firing (will take single sound and change frequency for robot's laser)
    - Laser.wav – Player frequency 30000
    - Laser.wav – Robot frequency 12000
- HealthPak pickup
    - Powerup.wav – set volume to 60
- Player hit
    - Playerhit.wav – set volume to 75
- Player disintegration
    - Playerdeath.wav
- Robot hit
    - Robothit.wav – set volume to 75
- Robot disintegration
    - Robotdeath.wav – set volume to 60
- Start of level
    - Startlevel.wav
- Exit level
    - Exitlevel.wav – set volume to 75
- Intruder Alert
    - Intruderalert.wav
- Player snoring
    - Snore.wav – set volume to 90

Music Asset List

Just need one futuristic spy tune that is able to loop nicely. Should be upbeat and have that side-scroller action feel.

- Spy.ogg – set loop to true

Map Asset List

- Level 1: Not too complex a level layout, but challenging enough. Few robots so the player can get the feel.
  - Migzlevel1.dat
- Level 2: More robots, more healthpaks, and a different layout, but the same width/height ratio. This should be challenging.
  - Migzlevel2.dat
- Level 3: Lots of robots and healthpaks. This one should be really tough. The width/height ratio should be changed…making it more high than wide this time.
  - Migzlevel3.dat

Technical List

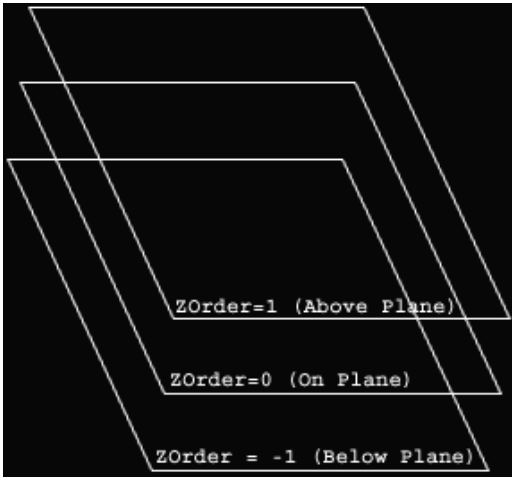
- Will need the ability to scroll a two-dimensional map of any size, including tile-sizes. Note that this game will use 32x32 tiles though.
- Player and robots must be animated appropriately depending on direction, firing status, hit and death status, etc.
- Player must be able to climb down/up ladders as needed to get to the robots.
- Robots should give the appearance of thought, even though it may be minor. They should have a timeout where they select a random action and follow-through with it. It is to give the player the feeling that the robots are patrolling.
- If a player gets within a certain distance of a robot, the robot (at its next decision cycle) should turn toward the player and start firing.
- The player must move with and independently of the map. So if the player is at the edge of the map, the map should stop scrolling while the player continues walking. If the player hits the center of the view port, walking away from the map's edge, the map should begin to scroll with the player.
- The player should pickup health when hitting a healthpak and the players/robots should lose health when hit. Duh.
- When all the robots in a level are wiped out the exit door should become active.
- Sounds should play at appropriate levels upon called actions (firing, hits, deaths, etc.) and should be layered (not stomp on each other or cut off each other).
- All level assets should refresh upon player death or new level entry.
- Display a little health bar over the player and robots heads to keep track of, well, health.

Chapter 19: Z-Ordering

What is Z-Ordering?

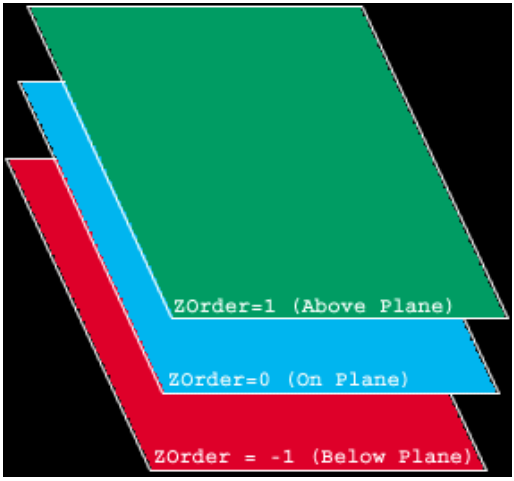
The term “Z-Ordering” can mean something different depending on context. In the realm of 3D graphics, I've seen it described as "...to derive closed-form relations for the difference between the node indices, which can be used to browse the tree in constant time." That's NOT the way we'll be using it 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 19.1)

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 19.2)

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.

Another example would be our side-scrolling game. We have to choose the order in which Migz will be shown. So we do this:

- Draw the backdrop
- Draw the Tiles
- Draw the Healthpaks
- Draw the Robots
- Draw the Lasers/Explosions
- Draw Migz

So the Migz is on top of the lasers, and the lasers are on top of the robots, etc. Since we can use transparency in our images, it also can give 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 structure (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 a 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 Screen Width and Height here for easier tracking
#ScreenWidth  = 640
#ScreenHeight = 480

; Initialize sprite and keyboard systems and a 640x480, 16-bit screen
If InitSprite() = 0 Or InitKeyboard() = 0 Or ↵
    → OpenScreen(#ScreenWidth,#ScreenHeight,16,"App Title") = 0
  MessageRequester("Error!", "Unable to Initialize Environment", ↵
   → #PB_MessageRequester_OK)
  End
EndIf

ClearColor = RGB(0,0,0)

; Use the PNG decoder for image loads.
UsePNGImageDecoder()

; load in the sprites
Tile1_Image = LoadSprite(#PB_Any,"tile1.png")
Tile2_Image = LoadSprite(#PB_Any,"tile2.png")
Robot_Image = LoadSprite(#PB_Any,"robot.png")
; set the appropriate mask so we have transparencies
TransparentSpriteColor(Robot_Image,RGB(255,0,255))

Structure Sprites
 Image.l
 X.l
 Y.l
 Direction.l
 ZOrder.l
EndStructure

; setup a list for the Images
NewList Sprite.Sprites()

; Tile 1
AddElement(Sprite())
```

```
Sprite()\Image = Tile1_Image
Sprite()\X = 125
Sprite()\Y = 100
Sprite()\Direction = -1
Sprite()\ZOrder = -1

; Tile 2
AddElement(Sprite())
Sprite()\Image = Tile2_Image
Sprite()\X = 100
Sprite()\Y = 100
Sprite()\Direction = 0
Sprite()\ZOrder = 0

; Robot
AddElement(Sprite())
Sprite()\Image = Robot_Image
Sprite()\X = 75
Sprite()\Y = 100
Sprite()\Direction = 1
Sprite()\ZOrder = 1


Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ; 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 sprites
    ForEach(Sprite())
      ; see if the current sprite is on this plane
      If Sprite()\ZOrder = Planes
        ; if so, move it based on its direction
        Select Sprite()\Direction
          Case -1
            Sprite()\X = Sprite()\X - 1
            If Sprite()\X < 75
              Sprite()\Direction = 1
            EndIf
          Case 1
            Sprite()\X = Sprite()\X + 1
            If Sprite()\X > 125
              Sprite()\Direction = -1
            EndIf
        EndSelect
        ; display the sprite
        DisplayTransparentSprite(Sprite()\Image,Sprite()\X,Sprite()\Y)
```

```
      EndIf

    Next
  Next

  ; put up text for exiting
  StartDrawing(ScreenOutput())
    DrawText(0,460,"Press Space to change the ZOrder, or ESC To quit")
  StopDrawing()

  FlipBuffers()

  ExamineKeyboard()

  ; if the spacebar gets pressed
  If KeyboardReleased(#PB_Key_Space)
    ; run through the sprites
    ForEach(Sprite())
      ; and alter their ZOrder and Direction
      Select Sprite()\ZOrder
        Case -1
            Sprite()\ZOrder = 0
            Sprite()\Direction = 0
        Case 0
            Sprite()\ZOrder = 1
            Sprite()\Direction = 1
        Case 1
            Sprite()\ZOrder = -1
            Sprite()\Direction = -1
      EndSelect
    Next
  EndIf
Until KeyboardReleased(#PB_Key_Escape)

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 20: 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 20.1)

Other times, the artist puts a block around each image to keep them visually separated:

(Figure 20.2)

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 GrabSprite(…) command in PB, 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 20.3)

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.pb." 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 for keeping track of the map and collisions too. Here they are:

```
;*****************************************************************
;* Begin Map Control Defines
;*****************************************************************
; Structure For the TileList

Structure Map_Tiles
  Image.l
  X.l
  Y.l
  Width.l
  Height.l
EndStructure

Global Map_TotalTiles.l
Map_TotalTiles = 0

Global Map_TilesImages.l

; Dimension our tile array
Global Dim Tile.Map_Tiles(Map_TotalTiles)
Global Map_FullTiles.l
Map_FullTiles = 0

; Structure for the Map_Data
Structure Map_Data
 TileNumber.l
EndStructure

; Globals to track the map dimensions
Global Map_Width
Map_Width = 1
Global Map_Height
Map_Height = 1
Global Map_X_Start
```

```
Map_X_Start = 0
Global Map_Y_Start
Map_Y_Start = 0
Global Map_X_DisplayOffset
Map_X_DisplayOffset = 0
Global Map_Y_DisplayOffset
Map_Y_DisplayOffset = 0
Global Map_ScrollWidth
Map_ScrollWidth = 0
Global Map_ScrollHeight
Map_ScrollHeight = 0

; Dimension our Map array
Global Dim Map.Map_Data(Map_Width,Map_Height)
Global Dim MapHold.Map_Data(Map_Width,Map_Height)

;************************************************************
;* End Map Control Defines
;************************************************************

;************************************************************
;* Begin Collision Control Defines
;************************************************************

; create the Player collision array for bounding boxes on the player
Global Dim Map_PlayerCollisionArray(16)

; setup the walls structure for knowing where all the walls are for collisions
Structure Map_Walls
 TileNumber.l
 X.l
 Y.l
EndStructure

; setup a list for the map walls
Global NewList Wall.Map_Walls()

; setup the walls structure for knowing where all the walls are for collisions
Structure Map_Ladders
  TileNumber.l
  X.l
  Y.l
EndStructure

; setup a list for the map walls
Global NewList Ladder.Map_Ladders()


;************************************************************
;* End Collision Control Defines
;************************************************************
```

We will have to load the actual image tiles each time a new map is loaded. In order to do this, we will have to open the image file that has all of our tiles in it, then run through and load up each into the Tile array structure.

The following source is fully commented so study it carefully!

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     Procedure: Map_LoadTiles(...)
;       Author:  Krylar
;  Description: This function loads in 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.s = Tile Image file, including the full path
;    TileWidth = Width of the tiles being loaded
;    TileHeight = Height of the tiles being loaded
;    TileSpacer = If there are boxes around the tiles, set this to
;            the number of pixels that the boxes are wide and high.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_LoadTiles(Tile_Full_Path.s,TileWidth,TileHeight,TileSpacer.l)

  Dim Tile.Map_Tiles(0)

  ; first off let's load the full image containing all the tiles
  ; into a temporary space
  Temp_Image.l = LoadSprite(#PB_Any,Tile_Full_Path.s)

  ImageSizeX.l = SpriteWidth(Temp_Image.l)
  ImageSizeY.l = SpriteHeight(Temp_Image.l)

  TileColumns.l = ImageSizeX.l / TileWidth
  TileRows.l = ImageSizeY.l / TileHeight

  Dim Tile.Map_Tiles(TileRows.l * TileColumns.l)

  If Temp_Image.l = 0
    ProcedureReturn(-1)
  EndIf

  ; get the current buffer so we can restore to it
  DisplaySprite(Temp_Image,0,0)

  ; setup our basic vars.  The X and Y values will be whatever the XSpacer
  ; and YSpacer values are. If they are 0, then it's assumed that there is
  ; no space between the tiles. If you look at the sample images included
  ; with this demo you'll see a white box around each image...that's why
  ; I have these spacers...we don't wanna load the boxes, just the images.
```

```
  X.l = TileSpacer.l
  Y.l = TileSpacer.l

  ; keep track of the number of images
  ImageNumber.l = 0

  ; run through the total number of rows (minus 1, of course)
  For Rows.l = 0 To TileRows.l -1
    ; and run through all the columns per row (minus 1 again)
    For Columns.l = 0 To TileColumns.l -1
      ; create a new TileList element and assign it the current
      ; ImageNumber.  Then populate the TileList element with
      ; the TileWidth, TileHeight, and the actual Tile_Image
      Tile(ImageNumber)\Width = TileWidth
      Tile(ImageNumber)\Height = TileHeight

      ; then we grab the image based off the size setup in CreateImage(...)
      ; from our X,Y location
      Tile(ImageNumber)\Image=GrabSprite(#PB_Any,X,Y,TileWidth,TileHeight)

      ; now we add X to the TileWidth and the XSpacer to get the new X position.
      ; So, 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.l = X.l + TileWidth + TileSpacer.l

      ; increment our tile counter (for the array positioning)
      ImageNumber.l = ImageNumber.l + 1
    Next
    ; we've finished with that row, so reset X back to the spacer position
    X.l = TileSpacer.l

    ; now we add Y to the TileHeight and the YSpacer to get the new Y position.
    ; So, if our current Y = 2, and the TileHeight = 32 and the YSpacer = 2, we'd
    ; 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 (or the 36th pixel
    ; from the top).
    Y.l = Y.l + TileHeight + TileSpacer.l
  Next

  ; free the image from memory, so we don't hold memory for no reason
  FreeSprite(Temp_Image)

  ; reset our global tile tracker
  Map_TotalTiles.l = ImageNumber.l

  ProcedureReturn(0)
EndProcedure
```

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. Continuing down this thought process should show you that when you call on the Tile structure and ask for the corresponding image, you'll see that image appear in the appropriate X,Y coordinates on your screen.

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:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Procedure: Map_ReadTextDimensions(...)
;       Author:  Krylar
;  Description: This function loads in the actual dimensions of
;  the map file and stores the values in Map_Width and Map_Height.
;  This function is for use with TEXT files only.
; Arguments:
;    Map_Full_Path.s = Map file, including the full path
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_ReadTextMapDimensions(Map_Full_Path.s)

 ; first thing we do is open the file using a Pointer Variable (which
 ; is "FilePtr" in this case)
 FilePtr = ReadFile(#PB_Any,Map_Full_Path.s)

 ; if the file is not found, return -1
 If FilePtr = 0
   ProcedureReturn(-1)
 EndIf

 ; Then we read the first line of the file...it *should* contain the
 ; width/height data.  If it doesn't, somebody goofed up on the layout
 ; of the file
 MapDimensions.s = ReadString(FilePtr)

 ; we need to set up some variables to parse the line.  We could just
 ; set this up to have the x,y on two lines and save 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
  ; let's first put the current position in the string into Temp$
  ; this is just 1 character from the string cause we use the Mid$(...)
  ; command to yank out that value
  Temp.s = Mid(MapDimensions.s,Offset+1,1)

  ; if that character is not a comma we haven't gone past the length
  ; of the string
  If Temp.s <> "," And (Offset <= Len(MapDimensions.s))
   ; then it must be a number, so we save it in the HoldString$
   HoldString.s = HoldString.s + Temp.s
   ; otherwise, it's either gone too far or it's the comma seperator
  Else
   ; if we've gone too far, then we know that the Map_Height is done
   If Offset > Len(MapDimensions.s)
    ; so we convert the current HoldString$ to an Int and assign it
    Map_Height = Val(HoldString.s)
    ; and make sure the While loop breaks
    EndOfString = 1
   ; it must be a comma, so that means our Map_Width value is loaded
   Else
    ; convert the current HoldString$ to an int and assign it
    Map_Width = Val(HoldString.s)
    ; reset the HoldString$ to blank so we can start loading Map_Height
    HoldString.s = ""
   EndIf
```

```
   EndIf
   ; increase the string position offset by 1 (to move to the next character)
   Offset = Offset + 1
  Wend

  ; we're done, so close the file!
  CloseFile(FilePtr)

  Dim Map.Map_Data(Map_Width,Map_Height)

  Map_Init(Map_Width,Map_Height)

  ProcedureReturn(0)
EndProcedure
```

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:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     Procedure: Map_LoadTextMap(...)
;       Author:  Krylar
;  Description: This function loads in the actual map information.
;  This function is for use with TEXT files only.
;
; Arguments:
;    Map_Full_Path$ = Map file, including the full path
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_LoadTextMap(Map_Full_Path.s)

  ; first thing we do is open the file using a Pointer Variable (which
  ; is "FilePtr" in this case)
  FilePtr = ReadFile(#PB_Any,Map_Full_Path.s);
  ; 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 Map_Width/Map_Height, so you can DIM
  ; the Map_Data array appropriately
  MapDimensions.s = ReadString(FilePtr);

  ; set up our vars for array placements.  The X will be for columns, and
  ; the Y will be for rows.  The EndOfFile just let's us keep track of how
  ; far into the file we've gone.
```

```
X.l=0
Y.l=0
EndOfFile.l = 0

; do this until we reach the end of the file
While EndOfFile = 0

  ; read a line of data from the file and put it in the MapLine$ string
  MapLine.s = ReadString(FilePtr);

  ; make sure that the length of the line is more than 1 character
  If Len(MapLine.s) > 1

    ; set up our EndOfString var to be 0...this will help us keep
    ; track of the current MapLine$ string position
    EndOfString.l = 0

    ; This Offset var will let us keep track of our current position
    ; in the MapLine$ string
    Offset.l = 0

    ; until we reach the end of the MapLine$ string
    While EndOfString.l = 0

      ; let's first put the current position in the string into Temp$
      ; this is just 1 character from the string cause we use the Mid$(...)
      ; command to yank out that value
      Temp.s = Mid(MapLine.s,Offset.l + 1,1)

      ; if that character is not a comma we haven't gone past the length
      ; of the string
      If Temp.s <> "," And (Offset.l <= Len(MapLine.s))
        ; then it must be a number, so we save it in the HoldString$
        HoldString.s = HoldString.s + Temp.s
        ; otherwise, it's either gone too far or it's the comma seperator
      Else
        ; if we've gone too far, then we know that this line is done
        If Offset.l > Len(MapLine.s)
          ; fill in the number of the HoldString$ by converting it to an Int
          Map(X.l,Y.l)\TileNumber = Val(HoldString.s)
          ; reset the HoldString$ to a blank
          HoldString.s = ""
          ; exit the loop for *this* line (so we can read the next one)
          EndOfString.l = 1
          ; it must be a comma, so that means this column's value is loaded
        Else
          ; fill in the number of the HoldString$ by converting it to an Int
          Map(X.l,Y.l)\TileNumber = Val(HoldString.s)
          ; reset the HoldString$ to a blank
          HoldString.s = ""
          ; Increase the X (or Column) location for the Array
```

```
        X.l = X.l + 1
      EndIf
    EndIf
    ; add one to our string position offset
    Offset.l = Offset.l + 1
  Wend

  ; set X back to 0 (so we're back to column 0)
  X.l = 0
  ; add 1 to Y (so we move down 1 row in the array)
  Y.l = Y.l + 1
 ; if we've gone past the length of the file
 Else
  ; then tell the loop to stop cause we're done!
  EndOfFile.l = 1
 EndIf
Wend

; make sure to close the file!
CloseFile(FilePtr)

ProcedureReturn(0)
EndProcedure
```

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 the ***K-2D MapMaker*** system. ***K-2D MapMaker*** will load/store binary map files and such, and of course let you create and edit maps with varying tile sets (though it is limited to 32x32 tiles). It's not overly fancy, but it's good enough to start you working on your own maps (and even creating additional ones for Migz to play on!).

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:

```
;.....................................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Procedure: Map_LoadBinaryMap(...)
;       Author:  Krylar
;  Description: This function loads a binary map
;
; Arguments:
;       Map_Full_Path$: Name to load
;.....................................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_LoadBinaryMap(Map_Full_Path.s)

  ; first thing we do is open the file using a Pointer Variable (which
  ; is "FilePtr" in this case)
  FilePtr.l = ReadFile(#PB_Any,Map_Full_Path.s);

  If FilePtr.l = 0
    ProcedureReturn(-1)
  EndIf

  Dim Map.Map_Data(0,0)

  ; read in the map dimensions
  Map_Width = ReadLong(FilePtr);
  Map_Height = ReadLong(FilePtr);

  Dim Map.Map_Data(Map_Width,Map_Height)

  ; set up our vars for array placements.  The X will be for columns, and
  ; the Y will be for rows.  The EndOfFile just let's us keep track of how
  ; far into the file we've gone.
  X.l = 0
  Y.l = 0
  EndOfFile.l = 0

  ; for all of the rows (minus 1)
  For Rows.l = 0 To Map_Height - 1
    ; and for all the columns (minus 1)
    For Columns.l = 0 To Map_Width - 1
      ; read in the encrypted value
      Map(X.l,Y.l)\TileNumber = ReadLong(FilePtr)

      ; add 1 to X (so we move 1 column to the right in the array)
      X.l = X.l + 1
```

```
    Next
    ; set X back to 0 (so we're back to column 0)
    X.l = 0
    ; add 1 to Y (so we move down 1 row in the array)
    Y.l = Y.l + 1
  Next

  ; make sure to close the file!
  CloseFile(FilePtr)

  ProcedureReturn(0)
EndProcedure
```

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.

```
;.....................................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     Procedure: Map_SaveBinaryMap(...)
;       Author:  Krylar
;  Description: This function saves a map in binary format
;
; Arguments:
;         SaveFileName$: Name to save it as
;.....................................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_SaveBinaryMap(SaveFileName.s)

  ; first open the file for writing.  This *will* overwrite the existing file.
  FilePtr = CreateFile(#PB_Any,SaveFileName.s)

  ; write the Map_Width and Map_Height
  WriteLong(FilePtr,Map_Width)
  WriteLong(FilePtr,Map_Height)

  ; for all of the rows (minus 1)
  For Rows.l = 0 To Map_Height - 1
    ; and for all the columns (minus 1)
    For Columns.l = 0 To Map_Width - 1
      ; get the tilenumber
      TileNumber.l = Map(Columns.l,Rows.l)\TileNumber
      ; write it to the file
      WriteLong(FilePtr,TileNumber.l)
    Next
  Next
```

```
 ; close the file!
 CloseFile(FilePtr)

 ProcedureReturn(0)
EndProcedure
```

Showing a Loaded Map

After calling this function you should have all the data loaded into your *MapData* array of Structure. Now it's just a matter of calling the *Map_ShowMap* function. This function runs through the *MapData* array structure, grabs the image number to show, and then calls PB's DisplayTransparentSprite function with the *TileList* array element's image for that image number.

This code can be used for non-scrolling maps as well as full scrolling maps. Don’t worry though we’ll get into that in the next chapter!

Here's the code:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Procedure: Map_ShowMap(...)
;      Author:  Krylar
;  Description: This function moves and displays a map starting at
; a user-defined top-left corner at whatever width/height the user
; wants.  It will move N/S/E/W directions and do so at whatever
; distance (speed) the user needs.
;
; Arguments:
;    XOffset    = Where to start the left edge
;    YOffset    = Where to start the top edge
;    Direction  = 1-North, 2-South, 3-East, 4-West
;    Distance   = How far to move the map per call
;    ShowWidth  = How many columns to show (tiles)
;    ShowHeight = How many rows to show (tiles)
;    TileWidth  = How Wide are the tiles for the map
;    TileHeight = How High are the tiles for the map
;    ShowBoxes  = 0-No, 1-Show collision Points
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_ShowMap(XOffset.l,YOffset.l,Direction.l,Distance.l, ↵
      → ShowWidth.l, ShowHeight.l,TileWidth, TileHeight,ShowBoxes.l)
 RowPosition = 0
 ColumnPosition = 0

 ; Which way are we scrolling the map?
 Select Direction
  Case 1 ; North
   ; First let's make sure that we're not already on the edge
   If Map_Y_Start > 0
    ; Start counting up to zero for our offset
```

```
    Map_Y_DisplayOffset =  Map_Y_DisplayOffset + Distance
    ; if we hit or go past zero, subtract the amount over 0 from the
    ;TileHeight the user passed along
    ; and decrement our Map Y starting position
    If Map_Y_DisplayOffset > 0
      Map_Y_DisplayOffset = -(TileHeight) + Distance
      Map_Y_Start = Map_Y_Start - 1
    EndIf
    ; We want to show exactly the ShowHeight
    Map_ScrollHeight = Map_Y_Start + ShowHeight
  Else
    ; since we're already on Zero (top edge), just keep scrolling until the
    ; offset hits the edge too.
    If Map_Y_DisplayOffset < 0
      Map_Y_DisplayOffset = Map_Y_DisplayOffset + Distance
    EndIf
    ; We want to show one over the ShowWidth here so we don't end up
    ; with blinking edges
    Map_ScrollHeight = Map_Y_Start + ShowHeight + 1
  EndIf

Case 2 ; South
  ; Verify that we've not gone too far down
  If Map_Y_Start < Map_Height - ShowHeight - 1 And Map_Y_Start >= 0
    ; Count down to the AdjustedTileSize (see above)
    Map_Y_DisplayOffset = Map_Y_DisplayOffset - Distance
    ; if we hit or go past the negative size, add that amount to the Tile size
    ; sent by the user and make it negative.
    If Map_Y_DisplayOffset < -(TileHeight)
      Map_Y_DisplayOffset = 0 - Distance
      ; Increment our Map position
      Map_Y_Start = Map_Y_Start + 1
      ; Do another sanity check to make sure we didn't run over. If we did, drop
      ; the Map position back 1, set the offset so it's on the edge (scrolling stops)
      If Map_Y_Start >= Map_Height - ShowHeight
        Map_Y_Start = Map_Y_Start - 1
      EndIf
    EndIf
    ; We want to show exactly the ShowHeight
    Map_ScrollHeight = Map_Y_Start + ShowHeight
  Else
    If Map_Y_Start > 0
      ; We must already be at the point where the right edge is showing, so
      ; just scroll until the offset hits the edge too.
      Map_Y_DisplayOffset = Map_Y_DisplayOffset - Distance
      ; if we hit or go past the negative size, add that amount
      ; to the Tile size sent by the user and make it negative.
      If Map_Y_DisplayOffset < -(TileHeight)
        Map_Y_DisplayOffset = -(TileHeight)
      EndIf
    EndIf
```

```
  EndIf

Case 3 ; West (Left Arrow key)
  ; First let's make sure that we're not already on the edge
  If Map_X_Start > 0
    ; Start counting up to zero for our offset
    Map_X_DisplayOffset =  Map_X_DisplayOffset + Distance
    ; if we hit or go past zero, subtract the amount over 0 from the TileWidth the
    ; user passed along and decrement our Map X starting position
    If Map_X_DisplayOffset > 0
      Map_X_DisplayOffset = -(TileWidth) + Distance
      Map_X_Start = Map_X_Start - 1
    EndIf
    ; We want to show exactly the ShowWidth
    Map_ScrollWidth = Map_X_Start + ShowWidth
  Else
    ; since we're already on Zero (left edge), just keep scrolling until the offset
    ; hits the edge too.
    If Map_X_DisplayOffset < 0
      Map_X_DisplayOffset = Map_X_DisplayOffset + Distance
    EndIf
    ; We want to show one over the ShowWidth here so we don't end up
    ; with blinking edges
    Map_ScrollWidth = Map_X_Start + ShowWidth + 1
  EndIf

Case 4 ; East (Right Arrow Key)
  ; Verify that we've not gone too far right
  If Map_X_Start < Map_Width - ShowWidth - 1 And Map_X_Start >= 0
    ; Count down to the AdjustedTileSize (see above)
    Map_X_DisplayOffset = Map_X_DisplayOffset - Distance
    ; if we hit or go past the negative size, add that amount
    ; to the Tile size sent by the user and make it negative.
    If Map_X_DisplayOffset < -(TileWidth)
      Map_X_DisplayOffset = 0 - Distance
      ; Increment our Map position
      Map_X_Start = Map_X_Start + 1
      ; Do another sanity check to make sure we didn't run over. If we did, drop
      ; the Map position back 1, set the offset so it's on the edge (scrolling stops)
      If Map_X_Start >= Map_Width - ShowWidth
        Map_X_Start = Map_X_Start - 1
      EndIf
    EndIf
    ; We want to show exactly the ShowWidth
    Map_ScrollWidth = Map_X_Start + ShowWidth
  Else
    If Map_X_Start > 0
      ; We must already be at the point where the right edge is showing, so
      ; just scroll until the offset hits the edge too.
      Map_X_DisplayOffset = Map_X_DisplayOffset - Distance
      ; if we hit or go past the negative size, add that amount to the Tile size sent
```

```
      ; by the user and make it negative.
      If Map_X_DisplayOffset < -(TileWidth)
        Map_X_DisplayOffset = -(TileWidth)
      EndIf
    EndIf
  EndIf

  ; we're not moving the map, but the map still needs to be display. Check where
  ; we are on the map and display the proper width/height to avoid blinking.
Default
  If Map_X_Start < Map_Width - ShowWidth
    Map_ScrollWidth = Map_X_Start + ShowWidth
  Else
    Map_ScrollWidth = Map_X_Start + ShowWidth - 1
  EndIf

  If Map_Y_Start < Map_Height - ShowHeight
    Map_ScrollHeight = Map_Y_Start + ShowHeight
  Else
    Map_ScrollHeight = Map_Y_Start + ShowHeight - 1
  EndIf
  If Distance = -1
    Map_X_DisplayOffset = 0
    Map_Y_DisplayOffset = 0
  EndIf
EndSelect

; hook up our X, Y values to draw the images at the proper locations
X = Map_X_DisplayOffset + XOffset
Y = Map_Y_DisplayOffset + YOffset

; and then run through the Arrays and Draw stuff out!
For Rows = Map_Y_Start To Map_ScrollHeight
  For Columns = Map_X_Start To Map_ScrollWidth
    TileNumber = Map(Columns,Rows)\TileNumber
    ; if the Tile isn't a -1, then draw it
    If TileNumber <> -1 And TileNumber <> 49
      DisplayTransparentSprite(Tile(TileNumber)\Image,X,Y)
    EndIf

    ; if the user wants to see the collision boxes, draw them here
    If ShowBoxes.l = 1
      ForEach(Wall())
        If Wall()\TileNumber = TileNumber
          If StartDrawing(ScreenOutput())
            DrawingMode(4)
            Box(X,Y,TileWidth,TileHeight,RGB(0,255,255))
            DrawText(X + 10,Y + 10,Str(TileNumber))
            StopDrawing()
            Break
          Else
```

```
            MessageRequester("Error!", "Unable to Draw to ScreenOutput()", ↵
      → #PB_MessageRequester_Ok)
          EndIf
        EndIf
      Next
      ForEach(Ladder())
        If Ladder()\TileNumber = TileNumber
          If StartDrawing(ScreenOutput())
            DrawingMode(4)
            Box(X,Y,TileWidth,TileHeight,RGB(0,255,0))
            DrawText(X + 10,Y + 10,Str(TileNumber))
            StopDrawing()
            Break
          Else
            MessageRequester("Error!", "Unable to Draw to ScreenOutput()",↵
      → #PB_MessageRequester_Ok)
          EndIf
        EndIf
      Next

    EndIf
    ; increase our X position for tile drawing
    X = X + TileWidth
  Next
  ; reset the X position and increase the Y position
  X = Map_X_DisplayOffset + XOffset
  Y = Y + TileHeight
 Next
EndProcedure
```

By changing the values in *MapColumnStart* and *MapRowStart* you can scroll the map.

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 21: Moving Sprites on Scrolling Maps

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 having 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.

We'll be using the first method for the Migz game, so that's what I'll cover in this 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 SpritePixelCollision(…) 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.

Here is a visual idea of what we'll be doing. In figure 1 we have Migz surrounded by a bounding box:

(Figure 21.1)

Look carefully at the above graphic. We have Migz with a rectangle overlaying him. 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 21.2)

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 21.3)

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.

```
; create the Player collision array for bounding boxes on the player
Dim Map_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.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Procedure: Map_SetupPlayerBoundingBoxes()
; Author: Krylar
;
; sets up where the x1,y1,x2,y2 values are for the 4 boxes
; that make up the collision points on the player
;
; Arguments:
;       Top, Bottom, Left, Right values of X1, Y1, X2, Y2 - Respectively
;
; Returns: n/a
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure Map_SetupPlayerBoundingBoxes(TopX1.l,TopY1.l,TopX2.l,TopY2.l, ↵
    → BottomX1.l,BottomY1.l,BottomX2.l,BottomY2.l, ↵
    → LeftX1.l,LeftY1.l,LeftX2.l,LeftY2.l, ↵
    → RightX1.l,RightY1.l,RightX2.l,RightY2.l)
  ; setup the top-box
  Map_PlayerCollisionArray(0) = TopX1.l
  Map_PlayerCollisionArray(1) = TopY1.l
  Map_PlayerCollisionArray(2) = TopX2.l
  Map_PlayerCollisionArray(3) = TopY2.l
  ; setup the bottom-box
  Map_PlayerCollisionArray(4) = BottomX1.l
  Map_PlayerCollisionArray(5) = BottomY1.l
  Map_PlayerCollisionArray(6) = BottomX2.l
  Map_PlayerCollisionArray(7) = BottomY2.l
  ; setup the Left-box
  Map_PlayerCollisionArray(8) = LeftX1.l
  Map_PlayerCollisionArray(9) = LeftY1.l
  Map_PlayerCollisionArray(10) = LeftX2.l
  Map_PlayerCollisionArray(11) = LeftY2.l
  ; setup the Right-box
  Map_PlayerCollisionArray(12) = RightX1.l
  Map_PlayerCollisionArray(13) = RightY1.l
  Map_PlayerCollisionArray(14) = RightX2.l
```

```
  Map_PlayerCollisionArray(15) = RightY2.l
EndProcedure
```

For our game, I'm going to call the above function as follows:

```
; Initialize the Player's bounding boxes
Map_SetupPlayerBoundingBoxes(7,0,24,1, 7,30,24,31, 7,0,8,31, 23,0,24,31)
```

Next we'll want to run through our map and setup the walls. First we'll setup a structure to hold them all.

```
Structure Map_Walls
 TileNumber.l
 X.l
 Y.l
EndStructure

; setup a list for the map walls
Global NewList Wall.Map_Walls()
```

And we're also going to setup one for our ladders since we'll want Migz to animate differently when he's climbing.

```
Structure Map_Ladders
  TileNumber.l
  X.l
  Y.l
EndStructure

; setup a list for the map walls
Global NewList Ladder.Map_Ladders()
```

We'll call our *Map_AddWall* and *Map_AddLadder* procedures to actually populate the structures. To do this you need to know the actual numbers for your walls and ladders, of course.

```
Procedure Map_AddWall(TileToAdd.l,TileWidth,TileHeight)
  X.l = 0
  Y.l = 0

  ; run through all of the map
  For Rows.l = 0 To Map_Height - 1
    For Columns.l = 0 To Map_Width - 1
      TileNumber.l = Map(Columns.l,Rows.l)\TileNumber
      ; see if the tile number is one we want designated as a wall
```

```
    If TileNumber.l = TileToAdd.l
      ; Add it to the Wall Type
      AddElement(Wall())
      Wall()\TileNumber = TileNumber.l
      Wall()\X = X.l
      Wall()\Y = Y.l
    EndIf
    X.l = X.l + TileWidth
  Next
  ; reset X to the beginning of the next row
  X.l = 0
  ; increase our Y drawing position by the Tile's Height
  Y.l = Y.l + TileHeight
 Next
EndProcedure


Procedure Map_AddLadder(TileToAdd.l,TileWidth,TileHeight)
 X.l = 0
 Y.l = 0

 ; run through all of the map
 For Rows.l = 0 To Map_Height - 1
  For Columns.l = 0 To Map_Width - 1
    TileNumber.l = Map(Columns.l,Rows.l)\TileNumber
    ; see if the tile number is one we want designated as a ladder
    If TileNumber.l = TileToAdd.l
      ; Add it to the ladder Type
      AddElement(Ladder())
      Ladder()\TileNumber = TileNumber.l
      Ladder()\X = X.l
      Ladder()\Y = Y.l
    EndIf
    X.l = X.l + TileWidth
  Next
  ; reset X to the beginning of the next row
  X.l = 0
  ; increase our Y drawing position by the Tile's Height
  Y.l = Y.l + TileHeight
 Next
EndProcedure
```

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.

I wrote up a little function that calls *Map_AddWall* and *Map_AddLadder* with all the pertinent numbers so I could keep the function as encapsulated as possible.

```
Procedure SetupCollisionPoints(TileWidth,TileHeight)
  ; First let's set up the walls
  Map_AddWall(0,TileWidth,TileHeight)
  Map_AddWall(1,TileWidth,TileHeight)
  Map_AddWall(2,TileWidth,TileHeight)
  Map_AddWall(3,TileWidth,TileHeight)
  Map_AddWall(4,TileWidth,TileHeight)
  Map_AddWall(5,TileWidth,TileHeight)
  Map_AddWall(6,TileWidth,TileHeight)
  Map_AddWall(7,TileWidth,TileHeight)
  Map_AddWall(8,TileWidth,TileHeight)
  Map_AddWall(9,TileWidth,TileHeight)
  Map_AddWall(10,TileWidth,TileHeight)
  Map_AddWall(11,TileWidth,TileHeight)
  Map_AddWall(12,TileWidth,TileHeight)
  Map_AddWall(13,TileWidth,TileHeight)
  Map_AddWall(14,TileWidth,TileHeight)
  Map_AddWall(15,TileWidth,TileHeight)
  Map_AddWall(16,TileWidth,TileHeight)
  Map_AddWall(17,TileWidth,TileHeight)
  Map_AddWall(18,TileWidth,TileHeight)
  Map_AddWall(33,TileWidth,TileHeight)
  Map_AddWall(34,TileWidth,TileHeight)
  Map_AddWall(35,TileWidth,TileHeight)
  Map_AddWall(36,TileWidth,TileHeight)
  Map_AddWall(37,TileWidth,TileHeight)
  Map_AddWall(38,TileWidth,TileHeight)
  Map_AddWall(39,TileWidth,TileHeight)
  Map_AddWall(40,TileWidth,TileHeight)
  Map_AddWall(41,TileWidth,TileHeight)
  Map_AddWall(42,TileWidth,TileHeight)
  Map_AddWall(49,TileWidth,TileHeight)

  ; Then we'll add in our ladders
  Map_AddLadder(43,TileWidth,TileHeight)
  Map_AddLadder(44,TileWidth,TileHeight)

EndProcedure
```

Now that we have our walls and ladders 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.

```
Procedure Map_CheckWallCollision(X.l,Y.l,TileWidth,TileHeight)

  ; do one calculation here for each absolute box position
  ; so we don't do them every itteration of our loop below
```

```
BoxTopX1.l    = X.l + Map_PlayerCollisionArray(0)
BoxTopY1.l    = Y.l + Map_PlayerCollisionArray(1)
BoxTopX2.l    = X.l + Map_PlayerCollisionArray(2)
BoxTopY2.l    = Y.l + Map_PlayerCollisionArray(3)
BoxBottomX1.l = X.l + Map_PlayerCollisionArray(4)
BoxBottomY1.l = Y.l + Map_PlayerCollisionArray(5)
BoxBottomX2.l = X.l + Map_PlayerCollisionArray(6)
BoxBottomY2.l = Y.l + Map_PlayerCollisionArray(7)
BoxLeftX1.l   = X.l + Map_PlayerCollisionArray(8)
BoxLeftY1.l   = Y.l + Map_PlayerCollisionArray(9)
BoxLeftX2.l   = X.l + Map_PlayerCollisionArray(10)
BoxLeftY2.l   = Y.l + Map_PlayerCollisionArray(11)
BoxRightX1.l  = X.l + Map_PlayerCollisionArray(12)
BoxRightY1.l  = Y.l + Map_PlayerCollisionArray(13)
BoxRightX2.l  = X.l + Map_PlayerCollisionArray(14)
BoxRightY2.l  = Y.l + Map_PlayerCollisionArray(15)

; run through all of the walls
ForEach(Wall())
  XCollision.l = 0
  YCollision.l = 0

  ; grab the Type values to speed things up a bit
  WallX.l = Wall()\X
  WallY.l = Wall()\Y

  ; calculate the WallWidth and Height to speed things up
  WallWidth.l = WallX.l + TileWidth
  WallHeight.l = WallY.l + TileHeight

  ; check the top bounding box
  If BoxTopX1.l >= WallX.l And BoxTopX2.l <= WallWidth.l
    XCollision.l = 1
  EndIf
  If BoxTopY1.l >= WallY.l And BoxTopY2.l <= WallHeight.l
    YCollision.l = 1
  EndIf

  ; check the bottom bounding box
  If BoxBottomX1.l >= WallX.l And BoxBottomX2.l <= WallWidth.l
    XCollision.l = 1
  EndIf
  If BoxBottomY1.l >= WallY.l And BoxBottomY2.l <= WallHeight.l
    YCollision.l = 1
  EndIf

  ; check the left bounding box
  If BoxLeftX1.l >= WallX.l And BoxLeftX2.l <= WallWidth.l
    XCollision.l = 1
  EndIf
```

```
    If BoxLeftY1.l >= WallY.l And BoxLeftY2.l <= WallHeight.l

      YCollision.l = 1
    EndIf

    ; check the right bounding box
    If BoxRightX1.l >= WallX.l And BoxRightX2.l <= WallWidth.l
      XCollision.l = 1
    EndIf
    If BoxRightY1.l >= WallY.l And BoxRightY2.l <= WallHeight.l
      YCollision.l = 1
    EndIf

    ; if there is a collision on both the X and Y axis, return 1
    If XCollision.l = 1 And YCollision.l = 1
      ProcedureReturn(1)
    EndIf
  Next

  ; no hit so return 0
  ProcedureReturn(0)
EndProcedure
```

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.

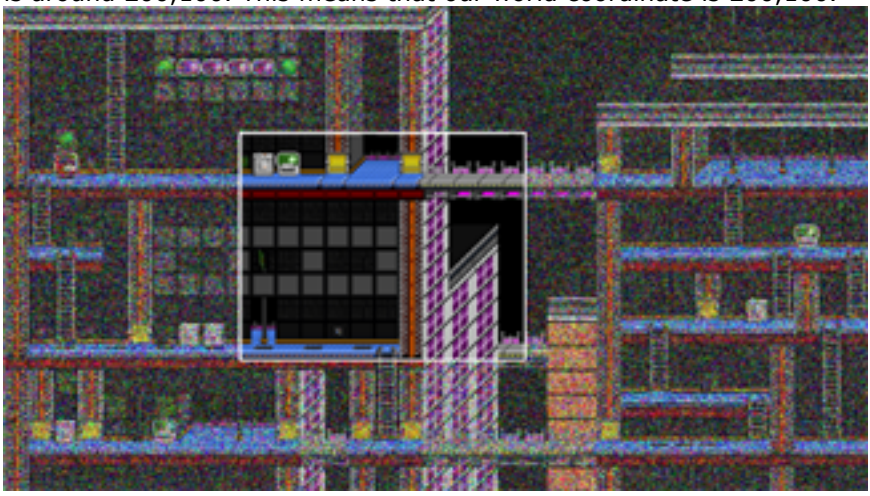
So instead, we use two coordinates. To determine which tiles of the map, which bad guys or NPC (non-player characters), etc., are drawn,

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 (Theory)

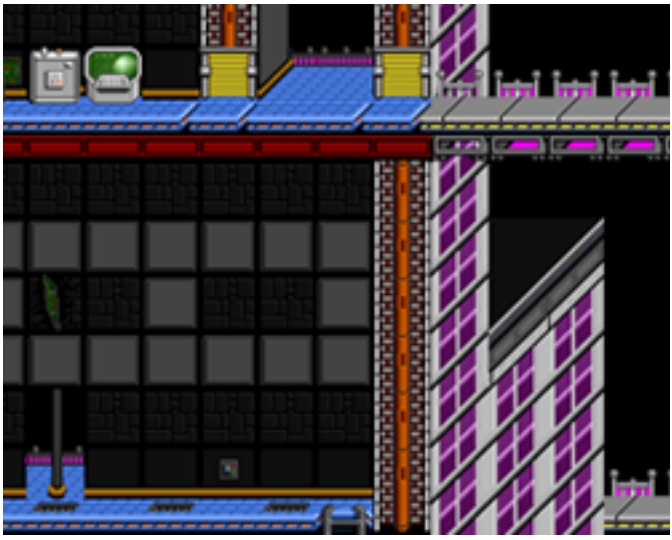
I once had tons of trouble understanding the concept behind how a map scrolls, but then I read an explanation that made it all fall together. I'll try to re-tell that here!

The white box in Figure 20.4 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 around 200,100. This means that our world coordinate is 200,100.



(Figure 21.4)

So the actual view port will show this:



(Figure 21.5)

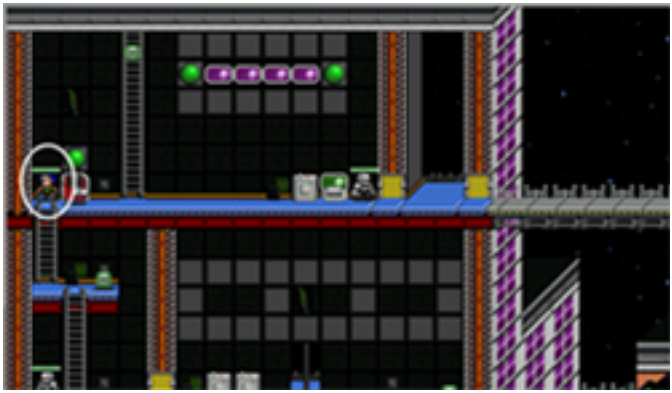
Now all you have to do is imagine moving that white box around pixel-by-pixel (or 4 pixels or whatever), redrawing the view port data from the new world coordinates, and bam…you have the concept of 2D map scrolling under your hat! Hopefully that will also help you to better understand the screen and world coordinate concept.

Edge-Independent Scrolling

Edge Independent Scrolling is a term that I made up (I think). The object is to scroll the map only 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.

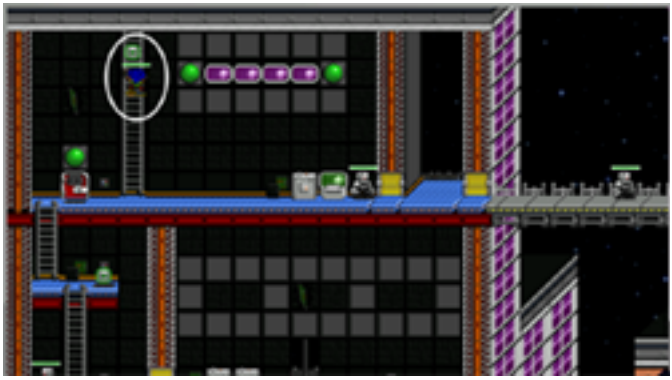
(Figure 21.6)

Here 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 left?



(Figure 21.7)

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 21.8)

See how the player is now in the upper-left 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

The following code pieces are brand new, but they need a little description and theory talk.

First off, there is more than one way to scroll a map. The method I'm going to present is pretty simple (in my opinion) and it runs at a nice speed.

The basic theory goes something like this:

1) We have a tile size, in this case it's 32x32.

2) We want to scroll the map at a certain speed. I use 4 pixels per step. You can easily change this to whatever you want, but this code will assume that you are using a value that's evenly divisible into 32. Feel free to alter the code to your heart's content, of course.

3) Each step that the player makes will change an offset value based on the distance of each step. So, if we just launched the game, the XOffset value would be a whopping 0. One step to the right and it becomes -4. Why MINUS? Because we want that left-most tile to start displaying at -4 now. This will make the tile display partially outside of the view port.

4) We display all the tiles in the view port + 1. Huh? In other words, if we only display the exact number of tiles that the view port will hold, we'll see the right-most column be nothing but empty space until we clear the left most tile completely. We don't want that. So where the view port may hold exactly 20 tiles, we want to DRAW 21. Don't worry, PureBasic will automagically clip the tiles.

5) Update the WorldX and WorldY coordinates of the player to keep track where he is. This is important because we're going to be checking the WorldX, WorldY values for collisions. We don't check the ScreenX and ScreenY for this because they don't change.

6) If we register a collision, no biggie, just reset the World value back to what it was before we moved a step.

That's pretty much it.

Here's the Move Player code for review:

```
Procedure MovePlayer(Direction.l,Distance.l,TileWidth,TileHeight)

  MoveMapX.l = 0
  MoveMapY.l = 0
  Select Direction.l
    ; Standing still
    Case 0
```

```
  ; If we're at an edge, let the sprite walk independently of the scroll
  If PlayerWorldX <= ScreenX Or PlayerWorldX > Map_Pixel_Width - ScreenX
    If PlayerWorldX <= ScreenX
      DrawPlayerX = PlayerWorldX
    Else
      DrawPlayerX = ViewPortWidth - (Map_Width * TileWidth - PlayerWorldX)
    EndIf
  Else
    ; We'renot be at an edge, so we want to keep the sprite in the middle
    ; and let the program know we're good to scroll
    DrawPlayerX = ScreenX
    MoveMapX.l = 1
  EndIf

  ; If we're at an edge, let the sprite walk independently of the scroll
  If PlayerWorldY <= ScreenY Or PlayerWorldY > Map_Pixel_Height - ScreenY
    If PlayerWorldY <= ScreenY
      DrawPlayerY = PlayerWorldY
    Else
      DrawPlayerY = ViewPortHeight - (Map_Height * TileHeight - PlayerWorldY)
    EndIf
  Else
    ; We must not be at an edge, so we want to keep the sprite in the middle
    ; and let the program know we're good to scroll
    DrawPlayerY = ScreenY
    MoveMapY.l = 1
  EndIf

  ; Going Up
Case 1
  ; save the current World value
  OldY.l = PlayerWorldY
  ; calculate the new value based on the distance
  PlayerWorldY = PlayerWorldY - Distance.l
  ; make sure the new value is >= 0
  If PlayerWorldY < 0
    PlayerWorldY = 0
  EndIf

  ; See if there's a collision
  If Map_CheckWallCollision(PlayerWorldX,PlayerWorldY,TileWidth, ↵
    → TileHeight) = 1
    ; If so, reset the PlayerWorldY value
    PlayerWorldY = OldY.l
  Else
    ; Otherwise, see where we are on the map
    ; If we're at an edge, let the sprite walk independently of the scroll
    If PlayerWorldY < ScreenY Or PlayerWorldY >= Map_Pixel_Height - ScreenY
      If PlayerWorldY <= ScreenY
        DrawPlayerY = PlayerWorldY
      Else
```

```
      DrawPlayerY = ViewPortHeight - (Map_Height * TileHeight - PlayerWorldY)
    EndIf
   Else
    ; We must not be at an edge, so we want to keep the sprite in the middle
    ; and let the program know we're good to scroll
    DrawPlayerY = ScreenY
    MoveMapY.l = 1
   EndIf
  EndIf

; Going Down
Case 2
  OldY.l = PlayerWorldY
  PlayerWorldY = PlayerWorldY + Distance.l
  If PlayerWorldY > Map_Height * TileHeight - TileHeight
    PlayerWorldY = Map_Height * TileHeight - TileHeight
  EndIf
  ; See if there's a collision
  If Map_CheckWallCollision(PlayerWorldX,PlayerWorldY,TileWidth, ↵
    → TileHeight) = 1
    ; If so, reset the PlayerWorldY value
    PlayerWorldY = OldY.l
  Else
    ; Otherwise, see where we are on the map
    ; If we're at an edge, let the sprite walk independently of the scroll
    If PlayerWorldY <= ScreenY Or PlayerWorldY > Map_Pixel_Height - ScreenY
      If PlayerWorldY <= ScreenY
        DrawPlayerY = PlayerWorldY
      Else
        DrawPlayerY = ViewPortHeight - (Map_Height * TileHeight - PlayerWorldY)
      EndIf
    Else
      ; We must not be at an edge, so we want to keep the sprite in the middle
      ; and let the program know we're good to scroll
      DrawPlayerY = ScreenY
      MoveMapY.l = 1
    EndIf
  EndIf

; Going Left
Case 3
  OldX.l = PlayerWorldX
  PlayerWorldX = PlayerWorldX - Distance.l
  If PlayerWorldX < 0
    PlayerWorldX = 0
  EndIf
  ; See if there's a collision
  If Map_CheckWallCollision(PlayerWorldX,PlayerWorldY,TileWidth, ↵
    → TileHeight) = 1
    ; If so, reset the PlayerWorldX value
```

```
      PlayerWorldX = OldX.l
    Else
      ; Otherwise, see where we are on the map
      ; If we're at an edge, let the sprite walk independently of the scroll
      If PlayerWorldX < ScreenX Or PlayerWorldX >= Map_Pixel_Width - ScreenX
        If PlayerWorldX < ScreenX
          DrawPlayerX = PlayerWorldX
        Else
          DrawPlayerX = ViewPortWidth - (Map_Width * TileWidth - PlayerWorldX)
        EndIf
      Else
        ; We must not be at an edge, so we want to keep the sprite in the middle
        ; and let the program know we're good to scroll
        DrawPlayerX = ScreenX
        MoveMapX.l = 1
      EndIf
    EndIf

  ; Going Right
  Case 4
    OldX.l = PlayerWorldX
    PlayerWorldX = PlayerWorldX + Distance.l
    If PlayerWorldX > Map_Width * TileWidth - TileWidth
      PlayerWorldX = Map_Width * TileWidth - TileWidth
    EndIf
    ; See if there's a collision
    If Map_CheckWallCollision(PlayerWorldX,PlayerWorldY,TileWidth, ↵
      → TileHeight) = 1
      ; If so, reset the PlayerWorldX value
      PlayerWorldX = OldX.l
    Else
      ; Otherwise, see where we are on the map
      ; If we're at an edge, let the sprite walk independently of the scroll
      If PlayerWorldX <= ScreenX Or PlayerWorldX > Map_Pixel_Width - ScreenX
        If PlayerWorldX <= ScreenX
          DrawPlayerX = PlayerWorldX
        Else
          DrawPlayerX = ViewPortWidth - (Map_Width * TileWidth - PlayerWorldX)
        EndIf
      Else
        ; We must not be at an edge, so we want to keep the sprite in the middle
        ; and let the program know we're good to scroll
        DrawPlayerX = ScreenX
        MoveMapX.l = 1
      EndIf
    EndIf
EndSelect

; if either X or Y is scrollable, return 1 to let the caller know
If MoveMapX.l = 1 Or MoveMapY.l = 1
  ProcedureReturn(1)
```

```
  Else

    ProcedureReturn(0)
  EndIf
EndProcedure
```

Note that the above code is in our main program, not the "maplib.pb" one. This is because how we move the player will be up to *us*, not up to a map module.

Here is the set of calls made that setup scrolling:

```
; move the player and see if we should scroll the map or not
MoveMap.l = MovePlayer(Direction.l,#MoveDistance,#TileWidth,#TileHeight)
; if the player is standing still
If Direction.l = 0
  ; show the map, but don't scroll it
  Map_ShowMap(0,0,0,0,#MapColumns,#MapRows,#TileWidth,#TileHeight, ↵
    → ShowCollisionPoints)
Else
  ; if we want to scroll the map, do that here
  If MoveMap.l = 1
    Map_ShowMap(0,0,Direction.l,#MoveDistance,#MapColumns,#MapRows, ↵
    → #TileWidth,#TileHeight, ShowCollisionPoints)
  Else
    ; otherwise, just show it
    Map_ShowMap(0,0,0,0,#MapColumns,#MapRows,#TileWidth,#TileHeight, ↵
    → ShowCollisionPoints)
  EndIf
EndIf
```

You can see that I'm showing the map in all instances. If I didn't, we'd see sporadic blank screens with our little space hero on them…a sort of purgatory, if you will. But it's not always scrolling so we need to make sure to layout the *Map_ShowMap* function to handle that, which we did in Chapter 20.

More on Coordinate Systems

We touched on the basics of *Screen* and *World* coordinates, but in order to effectively use them in our game, we'll need to hone in a bit more on them.

Screen Vs. World

In order to get a better understanding of how these systems differ, I will use a few screen shots from the Migz Callo game:

(Figure 21.9)

Here we see the upper-left position of the map. The top-left is literally 0,0 in the map array. Migz is sitting at position 3,5 in the map array, but his pixel position is 96,160. In order to determine Migz' pixel position, I just multiplied by our tile size, which is 32. So, X = 3 * 32 = 96, Y = 5 * 32 = 160. Thus, the top-left pixel of Migz' sprite is at 96,160.

In our game, we've set our view port to show 24 columns of tiles and 15 rows.

```
; determine how many columns/rows to draw per refresh
#MapColumns = 24        ; how many columns?
#MapRows = 15           ; how many rows?
```

Again, since our tiles are 32x32, this means that the view port will show 24x32 = 768 pixels on the X coordinate and 15x32 = 480 pixels on the Y coordinate. But, our maps can be quite a bit larger than that.

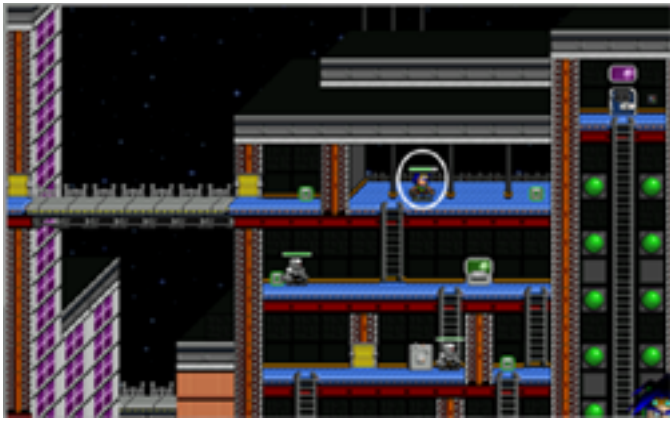
For example, the first level of the demo game is 40 columns wide and 20 rows high. Translation:

40 x 32 = 1280pixels
20 x 32 = 640pixels

…and we could easily go much larger than that.

This is where screen and world coordinates come into play though.

Imagine that we move Migz toward the right side of the map:


(Figure 21.10)

Okay, so now he stands at around 25 on the X coordinate. The problem is that 25x32 = 800. We're only able to show 768pixles in our view port. So, in essence, Migz shouldn't be showing up at all! So, why is he? And why is he showing up in the center of the view port?

Glad you asked. This is because when Migz hits that wondrous middle-of-the-view-port location, we no longer update his *Screen Coordinate*. We only update his *World Coordinate*. So, in the *World* he is at position 800, but on the *Screen* (where we actually draw the little fellow) he is at position 384. Using this method allows us to keep the map scrolling around while making sure our character never disappears from it.

But we also keep an eye on the *World Coordinate* because once Migz has passed a certain point in the *World* (the entire map), we want him to resume walking independently.

Robots, HealthPaks, and Lasers…oh my!

The robots and lasers can't be handled the same as Migz because they're not the focus points of the game. They have to appear to be in their own little space on the map, independent of Migz entirely.

Yet, if we don't take a few precautions, we'll end up with lasers going faster or slower than they should, robots floating up and down the screen, and healthpaks being ever out of reach.

Fortunately, this is pretty easy to deal with.

The next logical question we should ask is why the robots, lasers, and healthpaks have both *Screen* and *World* coordinates.

The reason for this is pretty much the same as the reason that Migz has both: a robot that walks around in *WorldX* of 955 can't be drawn using DisplayTransparentSprite at that location because it won't be seen. So

we have to somehow translate that robot's *World* coordinate to one that will be visible to Migz when the robot is within range.

So while the robot's *World* coordinates are applicable only to the robot, its *Screen* coordinates are updated by Migz' movements.

First we check to see if the map is even moving. If it's not we just let the robot's move whatever distance they normally do; if it is then we need to make sure to compensate for Migz' movement as well.

```
If MoveMap  = 1
  Select Direction
    Case 1
      Robot()\ScreenY = Robot()\ScreenY + #MoveDistance
    Case 2
      Robot()\ScreenY = Robot()\ScreenY - #MoveDistance
    Case 3
      Robot()\ScreenX = Robot()\ScreenX + #MoveDistance
    Case 4
      Robot()\ScreenX = Robot()\ScreenX - #MoveDistance
  EndSelect
EndIf
```

What this code does is check which *direction* Migz is moving and then just does an adjustment on the robot's *Screen* X, Y position using the distance that we've set the *MoveDistance* constant to. That's it!

The code for the lasers and healthpaks has the same layout.

For fun, comment those bits out of the code and watch all the joyous results you end up getting!

Lot's in this chapter, I know. Note that this chapter is the biggest piece of this entire game, if not the entire book. So really take the time to get this stuff down. Hopefully the comments and a little study will help you through that though, especially since you already have the knowledge of previous chapters to guide you.

Chapter 22: Simple AI

Artificial Intelligence demands a book on its own, but there are little things that one can do in a game that will make things at least appear to be alive and somewhat independent.

Robots Doing Stuff

In most side-scrollers the NPC's (non-player characters…bad guys, typically) are somewhat dumb. They tend to walk from point A to point B and the player has to jump over them or shoot them. I didn't want to go that route as I thought it a bit too simple, even for the likes of our demo game.

So, instead I set about making the robots evolve just mildly above that level of intellect. Not much, mind you, but enough to make things more interesting.

The first thing I did was to give each robot an *ActionTimer*. The object of this timer is to inform the robot that, when the timer goes off, it is to make a decision as to what to do next. The list of things it can decide is as follows:

- Stand still, facing left
- Stand still, facing right
- Walk left until the *ActionTimer* clicks or until hitting a minimum X point
- Walk right until the *ActionTimer* clicks or until hitting a maximum X point

This happens to be immensely effortless to accomplish.

```
; pick a random direction and go
Robot()\Direction = Random(3) + 1
```

All we do is tell PB to choose a random number from 1 to 4 and assign it over whatever currently sits in the *Direction* structure entity. During each loop we merely process that information for each robot and move them accordingly.

First, take a look at the code for when the robot is just standing sill, facing left.

```
Select Robot()\Direction
    Case 1   ; still, facing left
            ; if he's been hit or is dead
            If Robot()\Hit > 0 Or Robot()\Dead > 0
```

```
            ; if dead
            If Robot()\Dead > 0
              ; play the death animation until he's completely bye bye
              Robot()\Frame = Robot()\Frame - 1
              If Robot()\Frame < 52
                Robot()\Dead = -1
              EndIf
              ; otherwise, just show the hit
            Else
              Robot()\Frame = 55
              Robot()\Hit = Robot()\Hit - 1
            EndIf
          Else ; if he's neither hit nor dead, update the frames as usual
            Robot()\Frame = Robot()\Frame - 1
            If Robot()\Frame < 0
              Robot()\Frame = 7
            EndIf
            If Robot()\Frame > 7
              Robot()\Frame = 0
            EndIf
          EndIf
```

Notice that we first check to see if he's been hit or has died before doing anything else. This is key because it would look odd to have a walking robot that's, um, deactivated.

Now here is the case for when the robot is *walking* to the right:

```
Case 4   ; Right
      If Robot()\Hit > 0 Or Robot()\Dead > 0
        If Robot()\Dead > 0
          Robot()\Frame = Robot()\Frame + 1
          If Robot()\Frame > 51
            Robot()\Dead = -1
          EndIf
        Else
          Robot()\Frame = 48
          Robot()\Hit = Robot()\Hit - 1
        EndIf
      Else
        Robot()\Frame = Robot()\Frame + 1
        If Robot()\Frame < 32
          Robot()\Frame = 32
        EndIf
        If Robot()\Frame > 39
          Robot()\Frame = 32
        EndIf
        OldX = Robot()\WorldX
        Robot()\WorldX = Robot()\WorldX + #MoveDistance
        If Robot()\WorldX > Robot()\MaximumX
```

```
        Robot()\WorldX = OldX
        Robot()\Direction = 2
      Else
        Robot()\ScreenX = Robot()\ScreenX + #MoveDistance
      EndIf
    EndIf
```

Following along with that, you'll see that the only major change from walking and standing is that you have to update the WorldX position and check it against the maximum allowable X (or minimum, if walking left). Don't forget to the update the ScreenX position too as this variable keeps track of where to draw the robot on the screen in relation to the player.

Now, I know that doesn't seem like much, but when you're playing the game you will get the feeling of independent thought of these silly little bots. Stupid? Certainly. But, they're not just standing around playing with their capacitors either.

There are things that can keep a robot from doing stuff.

- The robot gets shot. This can halt its firing of its weapon, or make it freeze in place momentarily.
- Migz gets nuked. The robots are smart enough to know there's no sense in firing lasers at a dead guy.

Robots Firing

If Migz happens to be within range, the robot will take it that its proper decision would be to face Migz and shoot at him.

Step one is for the robot to see if Migz is even on the same Y-plane.

```
If Robot()\WorldY >= PlayerWorldY And Robot()\WorldY <= PlayerWorldY + 25
```

If so, then the robot wants to see if it should even both firing at the spaceman, since it knows that lasers beams on travel so far in this wacky environment.

```
If Robot()\WorldX > PlayerWorldX And Robot()\WorldX - PlayerWorldX < 300
```

That will check to see if Migz is within 300 units to the left of the robot. If so, it's shootin' time.

```
If Robot()\Firing = 1
  ; see which way he's facing
  Select Robot()\Direction
```

```
  Case 1
    ; and update the frames accordingly
    Robot()\Frame = Robot()\Frame - 1
    If Robot()\Frame < 16
      Robot()\Frame = 7
      Robot()\Firing = 0
      Robot()\ActionTimer = Current_Time
      Robot()\ActionTimeout = 500
    EndIf
    ; when the robot hits the middle frame, add in the laser
    If Robot()\Frame = 19
      AddLaser(1,Robot()\ScreenX,Robot()\ScreenY+11,Robot()\WorldX, ↵
    → Robot()\WorldY,1, #TileWidth, #TileHeight)
    EndIf
  Case 2
    Robot()\Frame = Robot()\Frame + 1
    If Robot()\Frame > 47
      Robot()\Frame = 24
      Robot()\Firing = 0
      Robot()\ActionTimer = Current_Time
      Robot()\ActionTimeout = 500
    EndIf
    ; when the robot hits the middle frame, add in the laser
    If Robot()\Frame = 44
      AddLaser(1,Robot()\ScreenX+32,Robot()\ScreenY+11,Robot()\WorldX+32, ↵
    → Robot()\WorldY,2, #TileWidth,#TileHeight)
    EndIf
EndSelect
```

If you study that little bit, something interesting should pop out at you. We don't actually add in the laser shot until a certain frame in the animation is hit. Additionally, we add a bit to the X coordinate of where the laser is fired from, if we are facing right. Why?

We add the laser animation to a particular frame because it would look odd for the laser to fire any other time. If you look at the robot firing frames, you'll see that one of them sits directly between two flashes. That's the frame we want to add the laser on because it will look the most realistic.

(Figure 22.1)

We move the laser over on the X coordinate a bit when the robot faces right because otherwise it will look like it's firing from his backside instead of from the weapon's nozzle.

Migz Gets Bored

One of the things that I've seen time and again in little games such as ours is the main character getting fidgety. So you, the player, put Migz in a nice safe spot and decide to go grab something to drink. When you come back Migz appears to have fallen asleep! If you keep your hands off the keys for a bit, you'll actually see Migz give you a couple of curious glances, a couple of yawns, and then he'll finally fall asleep. Face it, spacemen get bored easily when neglected.

So how's this done? We use timers, flags, and some nice animated frames.

Here's the set of frames for when Migz gets fidgety and starts to yawn and tire.

(Figure 22.2)

Obviously we want him to fidget a bit before yawning, and then we want him to yawn a couple of times before falling asleep. Here's the code for that:

```
; make sure the player's not fidgeting
If Current_Time > Fidget_Timer + Fidget_Interval And PlayerFidgeting = 0
  ; up the fidget level a bit...player's getting bored!
  Fidget_Level = Fidget_Level + 1
  If Fidget_Level > 7
    Fidget_Level = 7
  EndIf
  ; do a little fidget move
  If Fidget_Level > 0 And Fidget_Level < 3
    Fidget_Speed = 250
    PlayerFrame = 7
    Fidget_Interval = 2000
  EndIf
  ; yawn a bit
  If Fidget_Level > 2 And Fidget_Level < 5
    Fidget_Speed = 500
    PlayerFrame = 0
    Fidget_Timer = Current_Time
    Fidget_Interval = 2000
  EndIf
  ; yawn a smidgeon longer
```

```
  If Fidget_Level > 4 And Fidget_Level < 7
    Fidget_Speed = 1250
    PlayerFrame = 0
    Fidget_Interval = 2000
  EndIf
  ; okay, that's it...you're asleep
  If Fidget_Level >= 7
    Fidget_Speed = 250
    PlayerFrame = 111
    Fidget_Interval = 100
    PlayerSleeping = 1
  EndIf
  ; player is definitely fidgeting
  PlayerFidgeting = 1
EndIf
```

Okay, so we've set some fidget levels, but now we need to animate them.

```
; The player character is fidgeting
If PlayerFidgeting = 1
  ; just do a little fidget, crack the knuckles
  If Fidget_Level > 0 And Fidget_Level < 3
    ; assuming it's the right time of course
    If Current_Time > Frame_Timer + Fidget_Speed
      PlayerFrame = PlayerFrame - 1
      If PlayerFrame < 3
        PlayerFrame = 7
        PlayerFidgeting = 0
        ; reset the frame timer
        Fidget_Timer = Current_Time
      EndIf
      ; reset the frame timer
      Frame_Timer = Current_Time
    EndIf
  EndIf

  ; same as above
  If Fidget_Level > 2 And Fidget_Level < 5
    If Current_Time > Frame_Timer + Fidget_Speed
      PlayerFrame = 0
      ; reset the frame timer
      Fidget_Timer = Current_Time
      ; reset the frame timer
      Frame_Timer = Current_Time
      PlayerFidgeting = 0
    EndIf
  EndIf

  ; same as above
```

```
 If Fidget_Level > 4 And Fidget_Level < 7
  If Current_Time > Frame_Timer + Fidget_Speed
   PlayerFrame = PlayerFrame + 1
   If PlayerFrame > 1
    PlayerFrame = 0
    PlayerFidgeting = 0
    ; reset the frame timer
    Fidget_Timer = Current_Time
   EndIf
   ; reset the frame timer
   Frame_Timer = Current_Time
  EndIf
 EndIf
```

Migz Falls Asleep

Okay, so he's been fidgety and he's been yawning. The next logical progression would be to let the little fellow sleep:

(Figure 22.3)

Check out that not only do we put him in sleep mode, we also add a little snoring sound effect just for kicks.

```
; The player character is sleeping
If PlayerSleeping = 1
 If Current_Time > Snore_Timer + Snore_Interval
  MyPlaySound(7)
  Snore_Timer = Current_Time
 EndIf

 If Current_Time > Frame_Timer + Fidget_Speed
  PlayerFrame = PlayerFrame + 1
  If PlayerFrame > 103
   PlayerFrame = 96
  EndIf
  ; reset the frame timer
  Frame_Timer = Current_Time
 EndIf
EndIf
```

Now, again, there are many things that you *can* do using AI in your games, including path-finding, sneaking up, hiding, playing dead, running away, chasing a player, etc. But most of these things are a bit beyond the scope of this game and book as most side-scrollers are very simple in this realm.

However, don't let that stop you from pursuing these things. Path-finding is extremely important in many games these days, and you'll need to know how to go about using it effectively (or getting a tool that will aid you) in order to accomplish it fairly.

With a little effort you should be able to take the code from Migz Callo and make those robots smarter. Add the ability to climb the ladders (which will require some artwork) or chase after Migz when he tries to run away. Or maybe set up a portal system and a panic call for the robots. When a robot is near its end it tries to get to a portal to jump to a different spot on the level. It runs away, as it were. These things should be relatively easy to accomplish if you use your imagination and a bit of what we've already learned here.

Chapter 23: Putting It All Together

Well, we've gone from knowing next to nothing of making a 2D scrolling—or even how to program at all—to being able to build up a nice little fun futuristic world for Migz and his pals to play about in.

We've got the art, the sounds, the music, and even the map editor. And, of course, we've got the full source code for the game, which you should certainly modify to including things like jumping, falling, chasing, crouching, crawling, and anything else you can think up.

So now we have to put it all together.

The main loop

The key to making any game work properly is to have a nice main loop that it runs from. If you put everything into one long loop, without separating into functions where you can, you'll end up finding it more and more complicated to update and fix bugs in your game.

Here is our main loop for Migz:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAIN GAME LOOP
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Repeat
  ; clear the screen
  ClearScreen(ClearColor)

  ; draw the backdrop image
  DisplaySprite(BackDrop_Image,2,2)

  ; Get the current time so we only call the ElapsedMilliseconds() once per loop
  Current_Time = ElapsedMilliseconds()

  ; check for any movement keys being pressed (assuming we're not already firing!)
  If PlayerFiring = 0 And StartingLevel = 0
    ; see what direction we're moving, or 0 if nothing's being hit
    Direction.l = CheckMovementKeys()
  EndIf

  ; check the keyboard here to see if the firing/activation key has been hit too
  ExamineKeyboard()

  ; see if the user wants a screen shot or not
  TakeScreenie = 0
  If KeyboardPushed(#PB_Key_F10)
    ; if so set the flag for use after the flip
    TakeScreenie = 1
```

```
EndIf

; Check for firing/activation
If KeyboardPushed(#PB_Key_Space) And PlayerHitAnimation = 0 And ↵
    → PlayerDead = 0  And StartingLevel = 0
  ; if there are still Robots, then we must be shooting
  If ListSize(Robot()) > 0
    ; Make sure we're not on a ladder
    MTile = Map_CheckOnLadder(PlayerWorldX,PlayerWorldY, ↵
    → #TileWidth,#TileHeight)
    ; We're not on a ladder...
    If MTile = 0
      ; has enough time lapsed since our last firing?
      If Current_Time > Firing_Timer + Firing_Delay
        ; set the player firing flag to 1
        PlayerFiring = 1
        ; force the direction to 0 so no running or anything can happen
        Direction = 0
        ; set the start firing frame based on which way the player is facing
        If PlayerFacing = 3
          PlayerFrame = 31
        Else
          PlayerFrame = 64
        EndIf
        ; reset our firing time
        Firing_Timer = Current_Time
      EndIf
    EndIf
  Else ; the robots must all be dead...why shoot when there's no robots?

    ; See if we're at the exit point
    If PlayerWorldX >= PlayerExitX-5 And PlayerWorldY >= PlayerExitY-5 And ↵
    → PlayerWorldX <= PlayerExitX+37 And PlayerWorldY <= PlayerExitY+37
      ; if so, we're done with this level, so go to the next one
      Current_Level = Current_Level + 1
      If Current_Level > 3
        Current_Level = 1
      EndIf
      ; play the exiting level sound
      MyPlaySound(9)
      ; start up a new level!
      StartLevel(Current_Level,#TileWidth,#TileHeight)
    EndIf
  EndIf
EndIf

; Check to see what frame we should display in our next player draw
AnimatePlayer(Current_Time,#Animation_Speed,Direction,#TileWidth,#TileHeight)

; move the player and see if we should scroll the map or not
```

```
MoveMap.l = MovePlayer(Direction.l,#MoveDistance,#TileWidth,#TileHeight)
; if the player is standing still
If Direction.l = 0
  ; show the map, but don't scroll it
  Map_ShowMap(0,0,0,0,#MapColumns,#MapRows,#TileWidth,#TileHeight, ↵
    → ShowCollisionPoints)
Else
  ; if we want to scroll the map, do that here
  If MoveMap.l = 1
    Map_ShowMap(0,0,Direction.l,#MoveDistance,#MapColumns,#MapRows, ↵
    → #TileWidth,#TileHeight,ShowCollisionPoints)
  Else
    ; otherwise, just show it
    Map_ShowMap(0,0,0,0,#MapColumns,#MapRows,#TileWidth,#TileHeight, ↵
    → ShowCollisionPoints)
  EndIf
EndIf

; Move the health paks
MoveHealthPaks(Current_Time,MoveMap.l,Direction.l,#TileWidth,#TileHeight)

; Move the robots
MoveRobots(Current_Time,MoveMap.l,Direction.l,#TileWidth,#TileHeight)

; Move the lasers
MoveLasers(Current_Time,MoveMap.l,Direction.l,#TileWidth,#TileHeight)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Drawing the player sprite
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Make sure that the player's not dead and that we're not at the starting screen
If PlayerDead > -1 And StartingLevel = 0
   ; Then draw the player
   DisplayTransparentSprite(Player(PlayerFrame)\Image,DrawPlayerX
,DrawPlayerY)
   ; next we want to draw the player's healthbar
   If StartDrawing(ScreenOutput())
      ; draw an unfilled box first to hold the bar
      DrawingMode(4)
      Box(DrawPlayerX,DrawPlayerY-5,32,3,RGB(255,255,255))
      ; now we go back to filling the boxes we draw
      DrawingMode(0)
      ; fill it with gray
      Box(DrawPlayerX+1,DrawPlayerY-4,30,1,RGB(155,155,155))
      ; Now choose a color and distance to fill the box, based on the health of player
      If PlayerHealthBar > 15
         Box(DrawPlayerX+1,DrawPlayerY-4,PlayerHealthBar,1,RGB(0,255,0))
      EndIf
      If PlayerHealthBar > 9 And PlayerHealthBar <= 15
         Box(DrawPlayerX+1,DrawPlayerY-4,PlayerHealthBar,1,RGB(255,255,0))
      EndIf
```

```
    If PlayerHealthBar <= 9
      Box(DrawPlayerX+1,DrawPlayerY-4,PlayerHealthBar,1,RGB(255,0,0))
    EndIf
  Else
    MessageRequester("Error!", "Unable to Draw to ScreenOutput()", ↵
    → #PB_MessageRequester_Ok)
  EndIf
  StopDrawing()
EndIf

; if the player wants to see collision info, draw the bounding box here
If ShowCollisionPoints = 1
  Map_ShowPlayerBoundingBoxes(DrawPlayerX,DrawPlayerY)
EndIf

; Show our overlay image that wraps around the map viewport
DisplayTransparentSprite(HUD_Image,0,0)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Special messages
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
If StartDrawing(ScreenOutput())
  ; if the player has died
  If PlayerDead = -1
    ; show a little message and a countdown. Then restart the level.
    If Current_Time > Restart_Timer + 1000
      RestartTime = RestartTime - 1
      If RestartTime = 0
        StartLevel(Current_Level,#TileHeight,#TileWidth)
        StartingLevel = 0
        MyPlaySound(10)
      EndIf
      Restart_Timer = Current_Time
    Else
     DrawText(ScreenCenterX-150,ScreenCenterY,"Putting the jumper cables ↵
    → on Migz and restarting level in " + Str(RestartTime))
    EndIf
  EndIf
  ; if the level is just starting fresh (not a restart), show new level information
  If StartingLevel = 1
    If Current_Time > Start_Timer + 1000
      StartingLevelTime = StartingLevelTime - 1
      If StartingLevelTime = 0
        StartingLevel = 0
        MyPlaySound(10)
      EndIf
      Start_Timer = Current_Time
    EndIf
    DrawText(ScreenCenterX-80,ScreenCenterY-40,"Level " + ↵
    → Str(Current_Level) + "...starting in " + Str(StartingLevelTime))
    DrawText(ScreenCenterX-150,ScreenCenterY,"Robots take " + Str(RHits) + ↵
```

```
      → " hits to destroy in this level.")
      DrawText(ScreenCenterX-65,ScreenCenterY+40,"Good luck, Migz!")
    EndIf
  Else
    MessageRequester("Error!", "Unable to Draw to
ScreenOutput()",#PB_MessageRequester_Ok)
  EndIf
  StopDrawing()

  ; nuke any sounds that have outlived their welcome
  MyRemoveSound()

  ; flip the screen so the user can see what happened
  FlipBuffers()

  ; if the player asked for a screenshot, handle that here
  If TakeScreenie = 1
    ; snag the full screen
    Screenie = GrabSprite(#PB_Any,0,0,800,600)
    ; update the filename so multiple screenies can be snagged per game
    Screenshot.s = "Migzscreenie" + Str(ScreenNum) + ".bmp"
    ; save out the screenshot
    SaveSprite(Screenie,Screenshot.s)
    ; reset the flag
    TakeScreenie = 0
    ; increase the image for next time
    ScreenNum = ScreenNum + 1
    ; wait a full second so we can show that something happened and so we can
    ; make sure we don't get like 20 F-10 presses!
    Delay(1000)
  EndIf
  ; handle the case where a player hits ALT-TAB and loses game focus
  ; if you don't do this, your game will crash on ALT-TAB
  If IsScreenActive() = 0
    Repeat
      FlipBuffers()
    Until IsScreenActive() = 1
  EndIf
Until KeyboardReleased(#PB_Key_Escape)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; END OF MAIN GAME LOOP
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

I know you must be thinking that is a lot of code, and it is. Could it be made even smaller? Sure it could, but in the effort to make things as clear as possible, I opted to go with a bit more verbose code to make it more understandable.

If we took and added all those function calls in there, though, it'd be enormous. So, all in all, that's actually quite tidy!

Did you notice there were a number of places where I called RGB with repeating values? What could you do to make that more efficient? Could you maybe move them to the top of the file and assign them to variables instead? Give it a shot and see if you can get it working properly.

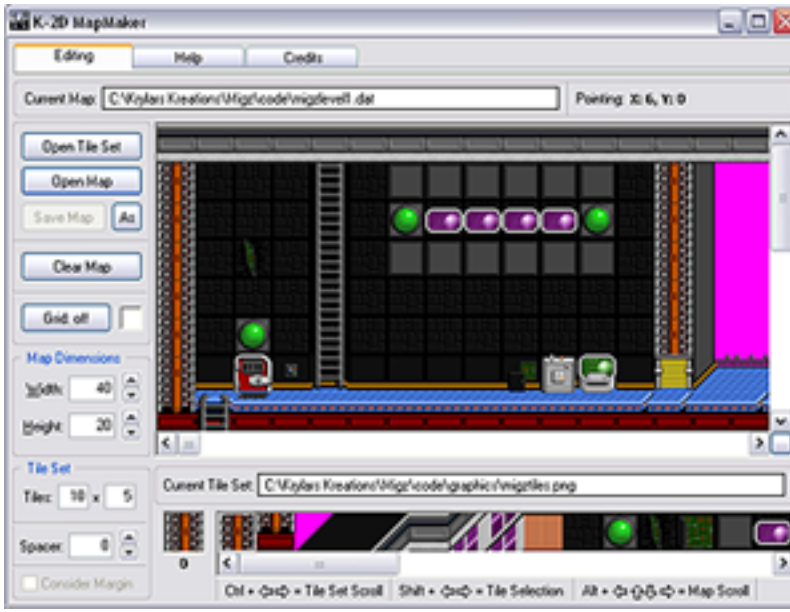
Making a level for Migz

There are a number of things that go into making a level for Migz. First thing is to note the limitations of our game:

- This game, again in order to keep it simple enough to fit the scope of this text, was built to allow only four types of movement: left, right, climb ladder, descend ladder. There is no jumping or falling or walking up into the page kind of things.
- The tiles are 32x32. Now the game engine can handle different sizes than that, but the supplied *K-2D MapMaker* (in its current form) cannot. So, in effect, we are limited to 32x32 unless a different editor that supports a similar file format is created with differing tile sizes.
- We have to place the robots by hand, and set the minimum and maximum walk points.
- We have to place the healthpaks by hand.
- Migz has to be placed somewhere on the upper-left of the map in order to start up the engine correctly. You can certainly go into the Map code and remove this limitation, if you'd like.

As you build out a new level, pay special attention to making sure that Migz will be able to walk only where you want him to, that there aren't too many robots, and that there are enough healthpaks to make the level possible to complete. Also, make sure that all of the tiles line up properly and nothing looks cut off. The time you spend building your levels will reflect either greatly or poorly on your final product, so be sure to take the time to do it right.

When you open the ***K-2D MapMaker*** you will first want to load in a tile set to use for your maps. Open up "migztiles.png" from the graphics directory of the game. Then open up the map "level1.dat."



(Figure 23.1)

If you do a "Save as" and call your file "level4.dat" or something, you can immediately make alterations and feel confident that you won't mess anything up.

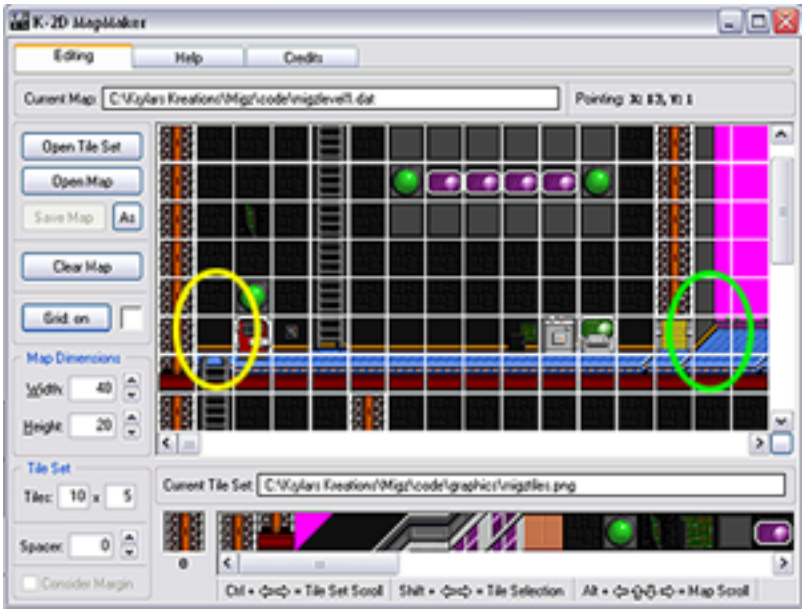
- You can select the tile at the bottom of the screen to place in the main map area. Do make sure to maximize this window to get the most map area on the screen at once.
- You can control the spacer values between tiles in the event that your tile set has spacers.
- You can adjust the width and the height of the map to make small or very large worlds.
- You can put the grid on (in whatever color you want) in order to see exact placement of tiles before you click.
- …and you can of course scroll all around your map and all that fine stuff.

Play with the width and height settings in the *K-2D MapMaker* as well. Build a huge map, if you want to. It's tougher, but it's doable!

Placing robots and healthpaks

To place items on the map, you should flip the grid on. This will help make sure that you're putting items in the proper spots.

Let's take an example of adding a robot to the above map and setting the robot's minimum and maximum allowable travel points.



(Figure 23.2)

The green circle above we'll use for two things. First, we'll say that this is where we want the robot to start, and second we'll say that this is the farthest position to the right that the robot is allowed to walk. The yellow circle represents the farthest point to the left the robot may walk.

If you place your mouse inside of the grid-square in the green circle, you'll see the following information near the top of the screen:

(Figure 23.3)

That tells you right away what the map array location will be for this robot's starting point. Also, the "14" tells you how far on the X the robot will be allowed to travel. If you decide to improve the robots' abilities by allowing them to move up and down ladders and such too, then you would also want to start using that Y value. But, keeping to our example game, we'll just stick with X.

Now place your mouse in the yellow circle and you'll get:

(Figure 23.4)

This is obviously the minimum X that the robot will be allowed to travel to.

Now that we have that information, we just call on the *AddRobot* procedure and we're done:

```
AddRobot(14,6,1,14,TileWidth,TileHeight)
```

Here we're just telling the *AddRobot* procedure to create a robot at position 14,6 in our map array, and then tell it to allow the robot to walk anywhere between position 1 and position 14. And we put in the tile size information too, in case someone creates a game with different tile sizes.

The only difference in adding healthpaks is that there is no minimum X or maximum X as healthpaks can't move on their own. So if we were to place a healthpak at the same position as the robot, we'd just do this:

```
AddHealthPak(14,6,TileWidth,TileHeight)
```

Easy stuff, no?

Code for starting a level

All this is well and good, but you'll need to have some code to start up the levels. You want to make sure that you're loading in the proper tiles, the proper level data, setting up your robots as they should be setup, adding in healthpaks, and placing your character on the appropriate spot as well.

Here is the code used to set up level one in the demo game, broken down. If you want to see the full code, it's on your disk in the migzdemo.pb file:

Step 1: Load in our tiles

```
; Load the tiles for the Map
Map_LoadTiles("graphics\migztiles.png",TileWidth , TileHeight ,0)
```

Step 2: Reset the map to make sure all arrays (map, collision, etc.) are all wiped out!

```
; Reset the map
Map_ResetMap(0,0)
```

Step 3: Load in the actual map data and fill up the map array information

```
; Load the actual map
Map_LoadBinaryMap("migzlevel1.dat")
```

Step 4: Set all the wall collision points in the map. If you neglect this, Migz will be able to walk through walls!

```
; Setup the wall objects so we know where we can expect to collide
SetupCollisionPoints(TileWidth ,TileHeight )
```

Step 5: Setup the spots on the player that will be checked for collisions. You can adjust these as you see fit based upon your level requirements.

```
; Initialize the Player's bounding boxes
Map_SetupPlayerBoundingBoxes(7,0,24,1, 7,30,24,31, 7,0,8,31, 23,0,24,31)
```

Step 6: Make certain that we have the appropriate widths and heights, as most maps are differently sized.

```
; reset the pixel width's and height's accordingly
Map_Pixel_Width= Map_Width * TileWidth
Map_Pixel_Height = Map_Height * TileHeight
```

Step 7: Get rid of all the previous robots. We don't want robots showing up in the middle of walls and stuff because the last map they were on was laid out differently.

```
; nuke all previous robots
DeleteRobots()
```

Step 8: Make sure there are no lingering lasers either.

```
; nuke all previous lasers
DeleteLasers();
```

Step 9: Wouldn't it be frustrating to be within a fraction of your life and see a healthpak hiding in the center of a wall and you can't get to it? Make sure to delete legacy paks!

```
; nuke all previous healthpaks
DeleteHealthPaks()
```

Step 10: Put the player in the appropriate spot on the new level.

```
; reset the player information
PlayerWorldX = 2 * TileWidth  ; start point X
PlayerWorldY = 6 * TileHeight ; start point Y
```

Step 11: Set the exit point on the map, so the player can get on to the next level. What you choose as your exit point is entirely up to you. In

the Migz demo, I opted to use the blue door tile, but you may do whatever you wish.

```
PlayerExitX = 37 * TileWidth  ; exit point X
PlayerExitY = 3 * TileHeight  ; exit point Y
```

Step 12: Reset all the basic information on the player. This stuff should just be cut-n-paste.

```
DrawPlayerX = ScreenCenterX
DrawPlayerY = ScreenCenterY
PlayerFrame = 0
PlayerDead = 0
PlayerHit = 0
PlayerHitAnimation = 0
PlayerFacing  = 4
PlayerArmor = 100
PlayerHealthBar = 30
Fidget_Level = 0
Fidget_Timer = Current_Time
PlayerFidgeting = 0
Fidget_Interval = 5000
Fidget_Speed = 250
PlayerSleeping = 0
PlayerFiring = 0
LaserSpeed = 8
RobotArmorHit = 7
PlayerArmorHit = 10
AllowableDistance = 30
RHits = 3
```

Step 13: You've picked various spots for your robots to hang out, and have selected their minimum/maximum X travel points, so add them here.

```
AddRobot(15,6,1,26,TileWidth,TileHeight)
AddRobot(26,6,1,26,TileWidth,TileHeight)
AddRobot(33,6,28,34,TileWidth,TileHeight)
AddRobot(6,13,1,15,TileWidth,TileHeight)
AddRobot(2,17,1,30,TileWidth,TileHeight)
AddRobot(22,17,1,30,TileWidth,TileHeight)
AddRobot(35,17,32,38,TileWidth,TileHeight)
AddRobot(28,15,25,30,TileWidth,TileHeight)
AddRobot(30,12,25,31,TileWidth,TileHeight)

AddRobot(25,9,25,34,TileWidth,TileHeight)
```

Step 14: You've picked various spots for your healthpaks to be, so add them here.

```
AddHealthPak(4,1,TileWidth,TileHeight)
```

```
AddHealthPak(26,6,TileWidth,TileHeight)
AddHealthPak(3,9,TileWidth,TileHeight)
AddHealthPak(25,9,TileWidth,TileHeight)
AddHealthPak(31,12,TileWidth,TileHeight)
AddHealthPak(15,13,TileWidth,TileHeight)
AddHealthPak(2,17,TileWidth,TileHeight)
AddHealthPak(29,17,TileWidth,TileHeight)
AddHealthPak(34,6,TileWidth,TileHeight)
AddHealthPak(33,12,TileWidth,TileHeight)
```

And that is how you use the code to make a level!

The Libraries

In the original version of this book, I had placed most everything the in the migzdemo.pb file, and it was a right mess. In this version of the book I have split out a number of things to make the code more maintainable. If you check out the "libs" directory, you will see a bunch of .pb files in there that make up the brunt of this game. Feel free to alter the code in these files as you see fit.

Conclusion

It's been a long journey to get to this point, and there has been a ton of stuff to learn. The more interesting news is that you're just getting started. There is so much more to learn and grow in that you'll soon find this text as pedantic as 3rd grade math. But that's okay. We all have to start somewhere, right?

But now what do you do?

Well, now you take the fundamentals from this game and expand upon them. Find other ways to accomplish similar tasks. Add more detailed AI to the game. Study up on game physics and add in jumping and falling. The only limit to what you can do in the world of 2D games here is your imagination and your ability to figure out the puzzles that come along with each challenge.

Often times you'll be able to find someone that has faced a similar problem to something you face now, and you may get an answer right away to your problem. But I caution you to first try to figure it out on your own. This is the best way to grow in this field, and for all you know your solution may be better than the one you got from another person!

It's been a fun journey. I want to thank you for reading this book and I wish you the very best with your game development!

Appendix

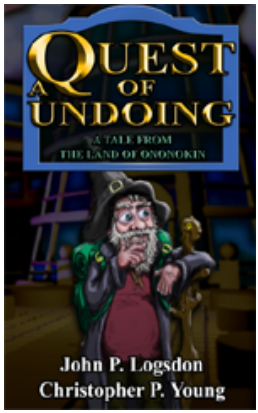
License

I hope you enjoyed this book and I hope that it's helped you out in some way. Please note that I will no longer be supporting this book, but you may feel free to update it as you see fit, as long as you stick with the rules of the Creative Commons License.

My Other Work

If you found this book helpful, please consider checking out my other projects. Word of mouth and reviews are the most helpful thing to any author, so I would greatly appreciate your support.

My lighthearted fantasy novels (Rated: PG-13)



A Quest of Undoing
ISBN-13: 978-1482625295
Amazon: http://www.amazon.com/Quest-Undoing-John-Logsdon-ebook/dp/B00F1EREBS

Whizzfiddle the wizard has to finish this quest in 30 days or he'll lose his guild membership status, meaning he'll have to actually work for a living. At the perpetual age of 650, this is really no longer an option.

He has to find a quest, get it contracted, and finish it to the letter. Since he's already done nearly everything there is as far as questing goes--it has to be interesting.

The last questing party to come in to Gilly's were the oddest bunch he'd seen in a long time, and being a wizard in the land of Ononokin, that is saying something. Seeing that it was a quest of undoing, Whizzfiddle signed on straight away.

Now he must constantly struggle to keep the troop on-track while simultaneously fighting to ward off his former apprentice's attempts to foil the mission...and he can't help wondering if it's even worth the effort.

The Journey Home
ISBN-13: 978-1494204013
Amazon: http://www.amazon.com/Journey-Home-John-Logsdon-ebook/dp/B00H8TBMR6

Paulie Vergen is your average, every day vampire. He's short, overweight, and balding, and he's a bit lacking in the realm of self-confidence.

Life was what it was for Paulie until the day that a newly infected werewolf landed on his doorstep. That day launched the one (and only, to this point) adventure that Paulie had ever had...and it was a doozy. Since new werewolves suffer amnesia, Paulie dubs the werewolf 'Mr. Biscuits' by night and 'Burt Biscuits' by day (you know, because werewolves are only werewolves at night and all).

Paulie has to help Burt get to Yezan, the land of the werewolves, in order to find help in getting Burt's memories back.

But there's a problem (well, multiple ones really). Stelan Bumache, the notable assassin who works for King Larkin in Yezan, has been hired to kill the newly infected Burt Biscuits.

Can Paulie and Burt successfully navigate the treacherous path from Viq to Yezan while avoiding the attempts of Stelan Bumache to destroy them both?

My kids science fiction series

Nanoagents: Induction
ISBN-13: 978-1492386957
Amazon: http://www.amazon.com/Nanoagents-Induction-John-Logsdon-ebook/dp/B00FEN5W9M

Seth Brennan is a typical, run-of-the-mill kid genius who is leading the school's computer club into the upcoming science fair.

Seth gets an email from Xabigan Industries while working on ideas for a project. The email promises a new piece of technology that will revolutionize the way people use the Internet. And it includes free shipping!

The problem is that a lot of computer-savvy kids have gone missing over the last few months, so Seth wants to be careful with this new piece of tech. Unfortunately, his friend and next-door neighbor, Chen, also got the email from Xabigan Industries and he's never careful with anything.

When Seth gets home from school that night he learns that Chen has disappeared!

Could Xabigan Industries be behind these disappearances? Could this new technology have some sinister application that nobody knows about? Seth has to connect the tech, outwit a couple of robots, foil the plans of a malevolent alien, and try to save Chen before it's too late!

Nanoagents: Lightcycle
ISBN-13: 978-1492878759
Amazon: http://www.amazon.com/Nanoagents-Lightcycle-John-Logsdon-ebook/dp/B00FNX7TVW

A suspected terrorist, Fedir Goraya, has built a motorcycle that is capable of housing a nuclear bomb, and it can be driven by remote control. It's called the Lightcycle.

The Nanoagents team is called in to track information on the suspect, which is not easy because his access point inside the Internet is on another grid and he's encrypted his data stream. Plus, the agents are just getting used to working inside the 'net.

Ajita has to spoof Goraya's access page, Cheryl and Bits need that intel to hack the rest of the way so they can snag the schematics on the bike and to setup a block against the detonation codes for the bomb, and Jaden and Rez need the bike's layout so they can build a piece of tech that Seth and Chen can use to get inside the Lightcycle.

And this has to all be done without Goraya suspecting a thing!

Can the agents get control of the Lightcycle before Goraya sets it in motion?

The Nanoagents have to pull together, work as a team, and leverage each other's strengths in order to save a small city in the Ukraine from certain destruction!

Nanoagents: Moon Base
ISBN-13: 978-1492978930
Amazon: http://www.amazon.com/Nanoagents-Moon-Base-John-Logsdon-ebook/dp/B00H8R8YFQ

A secret research facility has gone dark and the Nanoagents are sent to find out what happened. The problem is that the facility is on the moon!

The team works with NASA to determine a way to get to the moon. Once there they learn that a group of aliens has taken the Moon Base crew hostage. Seth, Ajita, Jaden, Cheryl, and Chen have to figure out how to free the hostages and take back Moon Base.

That's all good and well, except...

The aliens (known as Klakzaskians) that have taken over the base are not the ones running the show, they're just following orders. The alien in-charge is a Zbrakni, just like the alien named "Xabigan" that Seth had run into a few months earlier. His name is "Civugan" and he has learned a way to make the Klakzaskians do whatever he wants, and one of the things he wants to do is conquer Earth!

Can the Nanoagents get the Klakzaskians on their side? Will they be able to work with these new aliens to take Civugan down and get back control of Moon Base?

The Nanoagents will have to rely on each other if they're going to free the hostages and save the Earth from certain doom.

Visit John on the web

www.JohnPLogsdon.com