




# Chapter 6

## Tile-Based Video Modes



**T**his chapter explains the Game Boy Advance's tile-based graphics modes, with coverage of tile images, tile maps, scrolling backgrounds, and rotating backgrounds, as well as a tutorial on creating tiles, and converting them to a C array. Two complete programs are included in this chapter to demonstrate how to use scrolling and rotating backgrounds: the TileMode0 program and the RotateMode2 program. Here are the key topics covered:

- Introduction to tile-based video modes
- Creating a scrolling background
- Creating a rotating background

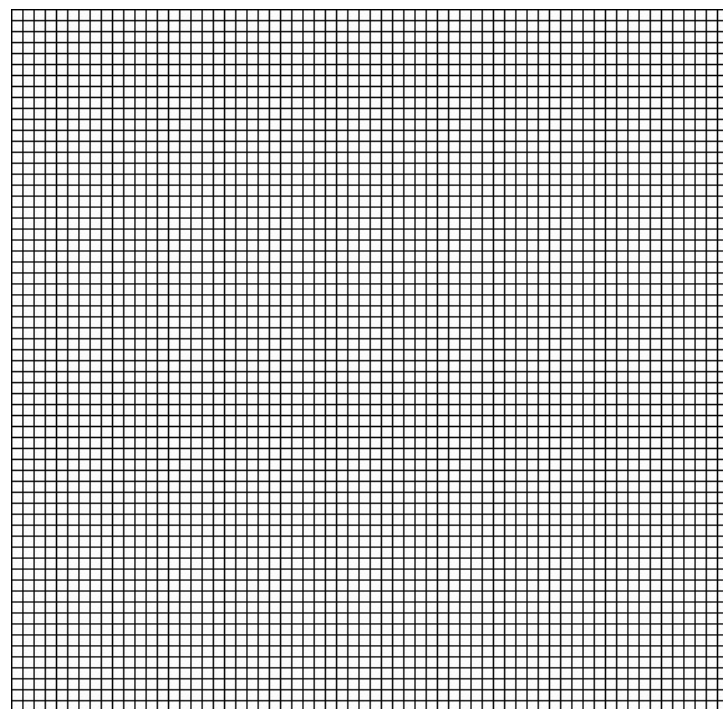
## Introduction to Tile-Based Video Modes

The Game Boy Advance offers three tile-based (also called text-based, or character) video modes that support a tiled screen comprising 8 x 8 tiles. A full screen is therefore made up of 30 tiles across and 20 tiles down. The maximum size of the background tile map is 1024 x 1024 pixels (when a rotation map is being used, 512 x 512 otherwise), or rather, 128 tiles across and 128 tiles down. As you might imagine, this provides the capability for storing a sizable level in a single tile map. Table 6.1 shows the properties of the tile-based video modes.

**Table 6.1 Tiled Video Modes**

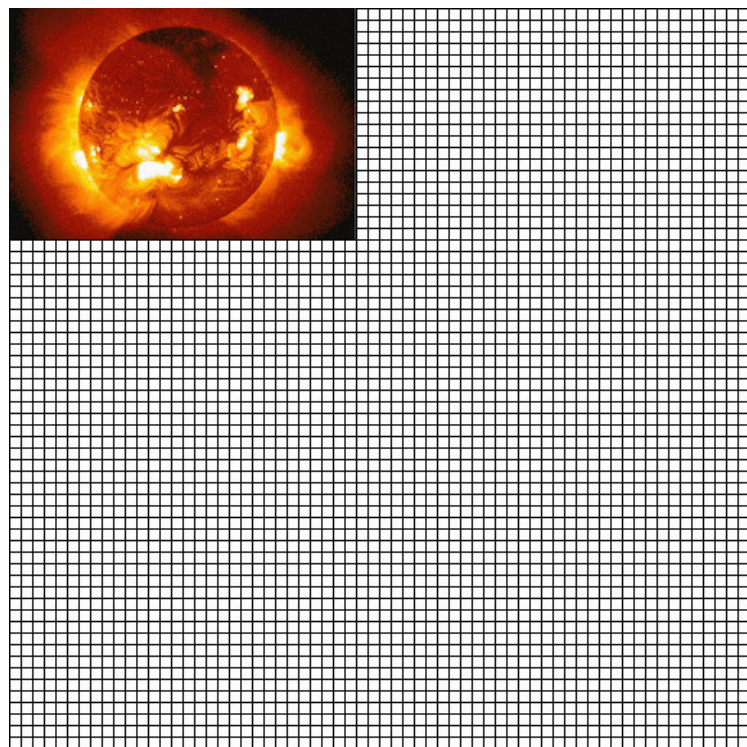
Mode	Backgrounds	Rotation/Scaling
0	0, 1, 2, 3	No
1	0, 1, 2	Yes (2)
2	2, 3	Yes (2, 3)

For instance, six rows or six columns of tiles can be stored in the 128 x 128 tile map and scrolled horizontally or vertically, adjusting the position of the "screen" to the next row or column upon reaching the edge of the previous one. See Figure 6.1.



*Figure 6.1*  
The maximum size of a (non-rotation) tile map is 512 x 512 pixels, or 64 x 64 tiles.

As you can see from Figure 6.2, you can create a large tile map for a game indeed, because the image in the upper-left corner represents one full screen of tiles!



*Figure 6.2*

*A single screen uses only a small portion of the maximum number of tiles.*

## Backgrounds

Since the tiled "text" backgrounds (0 and 1) support hardware scrolling of the background, you can simply plug in all the tiles you need for a game level (which would obviously not be made up of a single picture as shown in Figure 6.2). A typical tile-based game will have hundreds of tiles, many of which make up larger tiles (such as 16 x 16, 32 x 32, and 64 x 64 or larger). Displaying a larger "tile" really just involves displaying the smaller tiles that make up that large tile. It all breaks down to the least common denominator, which is the 8 x 8 pixel tile.

On the other hand, there are the two scale and rotate backgrounds (2 and 3). These backgrounds support only 8-bit color and vary in resolution from 128 to 1,024 pixels across. The usual palette located at 0 x 5000000 contains 256 color values, each a 16-bit number (which you learned about in the last chapter). I should also point out that the tile-based modes support 16-color palettes, and when using 16-color palettes, there are 16 separate, individual palettes available. Due to the smaller memory footprint of a 4-bit color (one-fourth the size of a 16-bit color), images that use 16-color palettes are also smaller. I will be sticking to 8-bit and 16-bit (actually, 15-bit, as you have already learned) colors for

simplicity. As Table 6.2 shows, backgrounds 0 and 1 are text backgrounds, while backgrounds 2 and 3 support rotation and scaling.

**Table 6.2 Backgrounds**

Background	Max Resolution	Rotation/Scaling
0	512 x 512	No
1	512 x 512	No
2	128 to 1,024	Yes
3	128 to 1,024	Yes

## Background Scrolling

The real advantage to tiled modes that I have yet to emphasize is the fact that these backgrounds can be layered on top of each other and that there is a priority involved in the layering, somewhat like a Z-buffer (if you lean toward the 3D realm). If you use video mode 0, with four text backgrounds (i.e., no scaling or rotation), then you can have four levels of parallax scrolling in your game, without any extra coding on your part (as far as writing the scrolling code or parallax layer transparency code, because that is all handled by the hardware). Most games that feature parallax are side scrollers, because it makes more sense to have scenery in the distance, with layers of terrain or objects closer to the player seeming to scroll by at a faster rate.

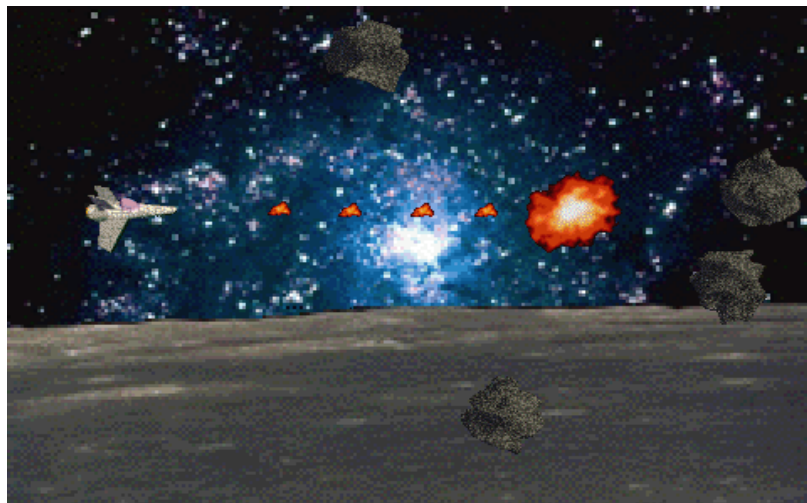
Mode 0 is great for this because all four backgrounds are hardware rendered. You do not need to write your own parallax scrolling routine. Now, you might be wondering, what is parallax scrolling? It's a concept that has been around for decades and is somewhat taken for granted today because it is so prevalent (kind of like a PC with a 3D card, something that was once uncommon). Parallax scrolling involves multiple layers, with closer layers scrolling faster than the distant layers. Figure 6.3 shows a fictional game scene with a starry background scrolling by slowly and a moonscape scrolling by more rapidly, with sprites transparently displayed on top of the two layers.

## Tiles and Sprites

One of the primary advantages to using a tiled mode is that there is more memory available for hardware sprites in these modes, whereas in bitmap modes (3, 4, and 5) only half of the



VRAM is available for sprites. How you use that memory, based on sprite size, is up to you, and I will cover this subject in the next chapter. One thing is a given, though, that there are a maximum of 128 sprites available. If you want to write a scrolling shoot-'em-up, for instance, you would almost certainly want to use a tiled mode, if not for the hardware background scrolling, then certainly for the large number of supported sprites. Of course, you may use any combination of sizes for the sprites in your game. Although I am not covering sprites in this chapter, I hope this has piqued your interest, because sprites are covered in the next chapter, and it is a fun subject, building on the subjects covered in this chapter.



*Figure 6.3*  
*The foreground and background layers are scrolling at a different rate of speed. The spaceship and asteroids are sprites drawn over the backgrounds.*

## The Tile Data and Tile Map

The tile map is stored in the same location as the video buffer (in the bitmap video modes), an array of numbers that point to the tile images. In the text backgrounds (0 and 1) the tile map comprises 16-bit numbers, while the rotation backgrounds (2 and 3) store 8-bit numbers in the tile map. This is an important distinction that you should carefully remember because it can be a source of frustration when writing code, particularly when you switch to another background.

The GBA uses several registers to determine where the bitmaps are stored for the tiles displayed on the screen, which differs for each background. As you learned, the tile modes support two or more backgrounds each. The tile data itself can be stored anywhere in VRAM (video memory) as long as it is on a 16 KB boundary, which starts at 0x6000000 and goes through 0x600FFFF. When you are working with tile-based modes, video memory is divided

into four logical *character base blocks*, which are made up of 32 smaller *screen base blocks*, as shown in Figure 6.4.

The tile map (which defines where the tiles are positioned) must begin at screen base boundary 31 at the very end of video memory.

Char Base Block 0	Screen Base Block 0	0x6000000
	Screen Base Block 1	0x6000800
	Screen Base Block 2	0x6001000
	Screen Base Block 3	0x6001800
	Screen Base Block 4	0x6002000
	Screen Base Block 5	0x6002800
	Screen Base Block 6	0x6003000
	Screen Base Block 7	0x6003800
Char Base Block 1	Screen Base Block 8	0x6004000
	Screen Base Block 9	0x6004800
	Screen Base Block 10	0x6005000
	Screen Base Block 11	0x6005800
	Screen Base Block 12	0x6006000
	Screen Base Block 13	0x6006800
	Screen Base Block 14	0x6007000
	Screen Base Block 15	0x6007800
Char Base Block 2	Screen Base Block 16	0x6008000
	Screen Base Block 17	0x6008800
	Screen Base Block 18	0x6009000
	Screen Base Block 19	0x6009800
	Screen Base Block 20	0x600A000
	Screen Base Block 21	0x600A800
	Screen Base Block 22	0x600B000
	Screen Base Block 23	0x600B800
Char Base Block 3	Screen Base Block 24	0x600C000
	Screen Base Block 25	0x600C800
	Screen Base Block 26	0x600D000
	Screen Base Block 27	0x600D800
	Screen Base Block 28	0x600E000
	Screen Base Block 29	0x600E800
	Screen Base Block 30	0x600F000
	Screen Base Block 31	0x600F800

*Figure 6.4  
Tile-based video memory  
is divided into logical  
blocks at 16 KB boundaries.*

## Creating a Scrolling Background

To demonstrate a tile-based scrolling background, I will walk you through a project called TileMode0, which you will write from scratch. Figure 6.5 shows the program running in the IDE.

For this program I cheated a little in order to make it easier to explain, at least for this first program in the chapter. What I mean by cheating is that I just created a single 256 x 256 bitmap image with all the tiles positioned already. In other words, this *does* use a tile map with tile images, but they are already set up, without the need for a map editor or for manual placement. Figure 6.6 shows the bitmap image.

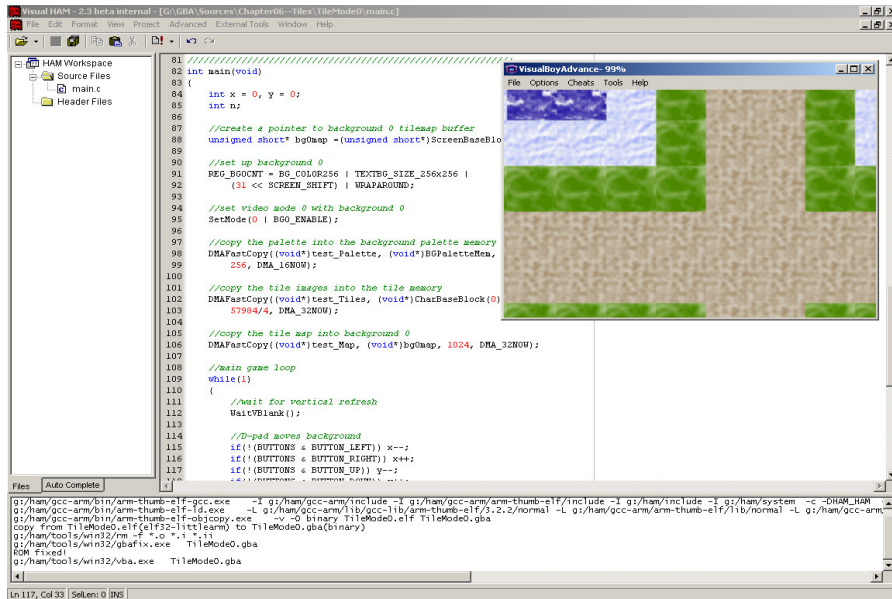


Figure 6.5  
The TileMode0 program demonstrates a tiled scrolling background.

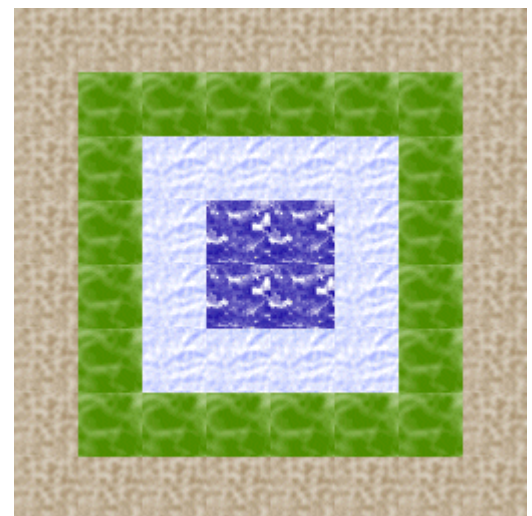


Figure 6.6  
The bitmap file used as the source image for the tiles.

Now, there are only four different tiles in this image, so it's very wasteful to duplicate them throughout the image. The whole point of tiling is to create a single tile set and use it for the whole map. But like I said, this is a learning experience so I wanted to make it easier to understand. The next program, where I explain how to create a rotating background, will use a modified version of this tile set with just four tiles referenced in the map file (as shown in Figure 6.7).

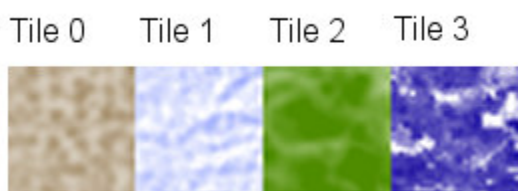


Figure 6.7  
There are really only four tiles needed for the tile map.



## Converting the Graphics

Before you can use the test.bmp file in a tile-based scroller, you'll need to convert it to a C array, just like you did with the sample programs in the previous chapter. This is very easy to do using the gfx2gba utility that is included with HAM. Before you can use it, you'll need to create a path to the program. Assuming you took my advice and installed HAM to the root under \HAM, then you can set up a HAM path by typing the following command into the command prompt (opened by selecting Start, Run, and typing "cmd" into the Run dialog box):

```
\ham\startham.bat
```

This batch file is included with the HAM SDK (which includes the GCC compiler, ARM assembler, and related tools). There is another option for starting a Command Prompt with support for the HAM tools, and that is a menu item in the Start menu that is provided by the HAM installer (that is, version 2.7 or later). Simply open Start, Program, HAM Development Kit, and click the option titled "HAM shell". That will open a Command Prompt that automatically runs startham.bat to set up the environment for running command-line tools.

Assuming you have copied the test.bmp file off the CD-ROM from \Sources\Chapter06\TileMode0 to your current project folder (where you plan to store the upcoming TileMode0 program), you can type this command:

```
gfx2gba -fsrc -m -ptest.pal -t8 test.bmp
```

Another, probably more convenient, method is to simply include the path to the utility when you run gfx2gba, like this (unless you used the "HAM shell" option, which I recommend):

```
\ham\tools\win32\gfx2gba -fsrc -m -ptest.pal -t8 test.bmp
```

The -m parameter tells the program to create a map file, while the -t8 parameter specifies a tile size of 8 x 8 pixels (the standard size supported by the GBA, which I wouldn't recommend changing, unless you are writing your own tile engine).

If gfx2gba was able to convert the file properly, you should see output that looks like this:

```
=====  
gfx2gba Converter v0.14  
-----
```

(C) 2001-2002 [TRiNiTY]

=====

Reading: test.bmp (256x256 pixel, 256 colors)

Number of tiles before optimization: 1024

Number of tiles after optimization: 0906

Saving tiled bitmap data to: test.raw.c ... ok

Saving map data to: test.map.c ... ok

Saving masterpalette to...: test.pal.c ... ok

Total files read & converted.: 1

Colors used before converting: 108

Colors used after converting.: 108

Colors saved.....: 0

If you pore over this output, you may notice something interesting, two lines that tell you how many tiles were created before and after optimization:

Number of tiles before optimization: 1024

Number of tiles after optimization: 0906

Doing a little math, you can determine that there are 32 tiles across and 32 tiles down in this map, resulting in 1,024 total tiles (at 8 x 8 pixels each), based on the source 256 x 256 pixel image. But gfx2gba is a smart program and was able to optimize the tiles somewhat—not completely, or else it would have seen that there are a more limited number of tiles, but it does try to help. The test.bmp file that I created has four different "large" tiles, each of which is 32 x 32 pixels in size—meaning there are 16 of the 8 x 8 tiles in each one of my large tiles. At most, then, there should be only 64 tiles, rather than 906 tiles, but I don't particularly care because this is a first-time demo. I'll create an optimized tile map for the rotation program later.

## Fast Blitting with DMA

One of the things that I have employed in this program to make it as fast as possible is a technique called DMA fast copy, which uses a special feature of the GBA to copy data from

one memory buffer to another—extremely fast. Basically, there's a custom chip on the GBA that handles memory—copying, moving, setting, clearing portions of memory, as well as normal accessing of data in memory by the CPU. Anytime the DMA chip is used, the CPU is temporarily suspended (only a matter of microseconds), until the DMA process is finished. This prevents the CPU from doing anything until a memory access is finished, otherwise problems could occur. Not only that, but in most computer systems and consoles, there is just one memory controller, and it can work on only one thing at a time. The newer memory architectures such as RDRAM and DDR found on PCs use two or more DMA controllers, meaning that memory can be accessed by two or more processes at the same time (or by the same process to access memory twice as fast). When the DMA chip is employed to write memory, it can't be used to read from memory at the same time, and vice versa. Therefore, the CPU is given a wait state while DMA activities are occurring.

DMA is a powerful aid to a GBA program, because it essentially replaces much source code with a single DMA call (or rather, three calls, as you will see shortly). Let's not forget also that DMA is a hardware process, where a software blitter is compiled and run by the CPU as machine instructions. You can't begin to compare a hardware process with a software process, because anything that is hard-coded into the silicon will blow away a series of machine instructions. For instance, the ARM7 CPU is a reduced instruction set computer (RISC) architecture, meaning that it has a small set of multipurpose instructions built in. More complex instructions must be built using what might be called building block instructions. Without getting into assembler language at this point (which is reserved for Chapter 12, "Optimizing Code with Assembly Language"), you could write a fast memory copy routine in assembler, and it would be much faster than a C routine. However, DMA will blow them both away when it comes to memory copies—and that is essentially what you need with a full-screen blitter. In fact, you don't even have to accommodate transparency in your code, because the GBA treats palette entry 0 as the transparent color. This could be useful for doing multilayer parallax backgrounds.

There are four DMA registers that you can use—or rather three, as the first one (# 0) is reserved. I'm just going to use the last register, although you could use DMA register 1, 2, or 3 just as well. Following is a list of the defines that you will need to use DMA fast copy for a background, as you will see in the upcoming TileMode0 program. The important defines here are the last two: DMA\_32NOW and DMA\_16NOW. These include options for copying 16-bit memory (such as external work RAM) and 32-bit memory (such as internal work RAM). For instance, the palette for a background is stored in a 16-bit memory address, while the tiles are stored in 32-bit memory.

```

#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4
#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8
#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC
#define DMA_ENABLE 0x80000000
#define DMA_TIMING_IMMEDIATE 0x00000000
#define DMA_16 0x00000000
#define DMA_32 0x04000000
#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32)
#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_16)

```

In order to perform a DMA fast copy, you simply set the three DMA registers to a value, and that triggers the process to start. Here is the DMAFastCopy function:

```

void DMAFastCopy(void* source,void* dest,unsigned int count,unsigned int mode)
{
    if (mode == DMA_16NOW || mode == DMA_32NOW)
    {
        REG_DMA3SAD = (unsigned int)source;
        REG_DMA3DAD = (unsigned int)dest;
        REG_DMA3CNT = count | mode;
    }
}

```

Note how the function first makes sure that the two standard copy modes have been passed to it. Although there are other options that you could use, I am simply adding in this small level of error checking to keep the function from overwriting memory somewhere if an invalid option is passed to it. There are other time options, for instance, other than immediate. For instance, you can have DMA start the copy after a specified number of CPU clocks. I personally don't find utility in such features, because a fast copy should run immediately.

## TileMode0 Source Code

This program has a lot of defines due to the various background mode and DMA settings used, but after you get past all the defines, the source code for the program is extremely short. It literally takes just a single line of code each to set up the palette, tiles, and map.



The most surprising thing about using backgrounds on the GBA is that you don't actually have to do any blitting code yourself; it is all done in the hardware, which is really bizarre if you are used to doing everything the hard way on a PC (for instance, using DirectX). On the GBA, once you have set the values into the appropriate locations in memory for the background settings, tile images, and tile map, the GBA handles the rest, including scrolling. In fact, to scroll the background, all you have to do is plug an X and Y value into the appropriate registers and—presto!—instant scrolling.

Now fire up Visual HAM and create a new project called TileMode0, or you may load this project off the CD-ROM from \Sources\Chapter06\TileMode0. You will need to have the test.map.c, test.pal.c, and test.raw.c files handy. If you skipped over the previous section on converting the tile graphics, you may want to go over that topic now, or simply copy the files off the CD-ROM. It's an invaluable lesson in creating tile maps, so I encourage you to go through process of converting the graphics rather than just using my premade files. Simply copy those files into the same folder where you created the new TileMode0 program, because this program includes those files. Since I spent so much time explaining how DMA works and how to initialize the background, and so on, I'm going to glaze over some of the other essentials for a scrolling background demo at this point and defer those explanations for the rotation example in the next section. Here is the source code for the TileMode0 program:

```
////////////////////////////////////  
// Programming The Game Boy Advance  
// Chapter 6: Tile-Based Video Modes  
// TileMode0 Project  
// main.c source code file  
////////////////////////////////////  
  
#define MULTIBOOT int __gba_multiboot;  
MULTIBOOT  
  
//include the sample tileset/map  
#include "test.pal.c"  
#include "test.raw.c"  
#include "test.map.c"
```

```

//function prototype
void DMAFastCopy(void*, void*, unsigned int, unsigned int);

//defines needed by DMAFastCopy
#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4
#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8
#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC
#define DMA_ENABLE 0x80000000
#define DMA_TIMING_IMMEDIATE 0x00000000
#define DMA_16 0x00000000
#define DMA_32 0x04000000
#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32)
#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_16)

//scrolling registers for background 0
#define REG_BG0HOF5 *(volatile unsigned short*)0x4000010
#define REG_BG0VOFS *(volatile unsigned short*)0x4000012

//background setup registers and data
#define REG_BG0CNT *(volatile unsigned short*)0x4000008
#define REG_BG1CNT *(volatile unsigned short*)0x400000A
#define REG_BG2CNT *(volatile unsigned short*)0x400000C
#define REG_BG3CNT *(volatile unsigned short*)0x400000E
#define BG_COLOR256 0x80
#define CHAR_SHIFT 2
#define SCREEN_SHIFT 8
#define WRAPAROUND 0x1

//background tile bitmap sizes
#define TEXTBG_SIZE_256x256 0x0
#define TEXTBG_SIZE_256x512 0x8000
#define TEXTBG_SIZE_512x256 0x4000
#define TEXTBG_SIZE_512x512 0xC000

```

```

//background memory offset macros
#define CharBaseBlock(n) (((n)*0x4000)+0x6000000)
#define ScreenBaseBlock(n) (((n)*0x800)+0x6000000)

//background mode identifiers
#define BG0_ENABLE 0x100
#define BG1_ENABLE 0x200
#define BG2_ENABLE 0x400
#define BG3_ENABLE 0x800

//video identifiers
#define REG_DISPCNT *(unsigned int*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)
#define SetMode(mode) REG_DISPCNT = (mode)

//vertical refresh register
#define REG_DISPSTAT *(volatile unsigned short*)0x4000004

//button identifiers
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTONS (*(volatile unsigned int*)0x04000130)

//wait for vertical refresh
void WaitVBlank(void)
{
    while((REG_DISPSTAT & 1));
}

////////////////////////////////////
// Function: main()
// Entry point for the program

```

```

////////////////////////////////////
int main(void)
{
    int x = 0, y = 0;
    int n;

    //create a pointer to background 0 tilemap buffer
    unsigned short* bg0map =(unsigned short*)ScreenBaseBlock(31);

    //set up background 0
    REG_BG0CNT = BG_COLOR256 | TEXTBG_SIZE_256x256 |
        (31 << SCREEN_SHIFT) | WRAPAROUND;

    //set video mode 0 with background 0
    SetMode(0 | BG0_ENABLE);

    //copy the palette into the background palette memory
    DMAFastCopy((void*)test_Palette, (void*)BGPaletteMem,
        256, DMA_16NOW);

    //copy the tile images into the tile memory
    DMAFastCopy((void*)test_Tiles, (void*)CharBaseBlock(0),
        57984/4, DMA_32NOW);

    //copy the tile map into background 0
    DMAFastCopy((void*)test_Map, (void*)bg0map, 512, DMA_32NOW);

    //main game loop
    while(1)
    {
        //wait for vertical refresh
        WaitVBlank();

        //D-pad moves background

```



```

    if(!(BUTTONS & BUTTON_LEFT)) x--;
    if(!(BUTTONS & BUTTON_RIGHT)) x++;
    if(!(BUTTONS & BUTTON_UP)) y--;
    if(!(BUTTONS & BUTTON_DOWN)) y++;

    //use hardware background scrolling
    REG_BG0VOFS = y ;
    REG_BG0HOFS = x ;

    //wait for vertical refresh
    WaitVBlank();

    for(n = 0; n < 4000; n++);
}
return 0;
}

////////////////////////////////////
// Function: DMAFastCopy
// Fast memory copy function built into hardware
////////////////////////////////////
void DMAFastCopy(void* source, void* dest, unsigned int count,
    unsigned int mode)
{
    if (mode == DMA_16NOW || mode == DMA_32NOW)
    {
        REG_DMA3SAD = (unsigned int)source;
        REG_DMA3DAD = (unsigned int)dest;
        REG_DMA3CNT = count | mode;
    }
}

```

## Creating a Rotating Background

Rotation backgrounds (2 and 3) are similar to text backgrounds in that they are made up of tiles in video modes 0, 1, or 2 (and behave differently in video modes 3, 4, or 5). But the so-called rotation backgrounds (obviously) support special features, such as rotation and scaling. I have written a sample program called RotateMode2 to demonstrate how to work with backgrounds 2 and 3. This program in particular uses background 2 and also runs in video mode 2—which, as you'll recall, supports the two rotation backgrounds, 2 and 3. For starters, let's create a new project in Visual HAM called RotateMode2. As usual, delete the default code that is inserted into main.c. I'll get into the source code as soon as I have finished explaining the tiles and map used in this program. The RotateMode2 program is shown in Figure 6.8.

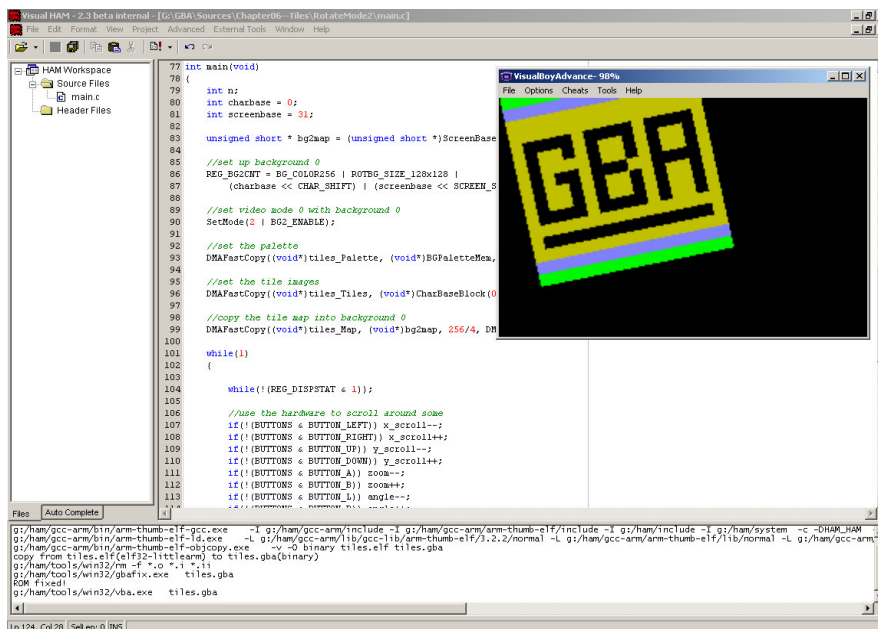


Figure 6.8  
The RotateMode2 program demonstrates how to rotate a background.

## Converting the Tile Image

This program uses a simple bitmap file to hold the five tiles used in the RotateMode2 program and is shown in Figure 6.9.



Figure 6.9  
The simple tiles used in the RotateMode2 program.

To convert this program to a C array, like you did with the previous program in this chapter, you'll run `gfx2gba` with the following options:

```
\ham\tools\win32\gfx2gba -fsrc -m -t8 -rs -ptiles.pal tiles.bmp
```

These options specify a map file (-m), a tile size of 8 x 8 (-t8), and output for rotate/scale backgrounds (-rs).

## Creating the Tile Map

Following is a listing of the tile map used in the `RotateMode2` program. I scrapped the map generated by `gfx2gba` and created this one manually. First, this map is easier to read because it's not in hexadecimal, but rather it just shows simple decimal numbers. Second, this is a small map, so it's easy to see what the map looks like before actually running the program. You can also edit this map to see how your changes look when run. This map is stored in a file called `tilemap.h` and is included by the main program.

```
//16x16 tile map
const unsigned char tiles_Map[256] = {
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 4, 4, 4, 4, 3, 4, 4, 4, 4, 3, 4, 4, 4, 4, 3,
3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 3, 3, 3, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 3, 4, 4, 3, 4, 4, 4, 3, 3, 4, 4, 4, 4, 3,
3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3, 4, 3, 3, 4, 3,
3, 4, 4, 4, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};
```

## RotateMode2 Source Code

The source code for RotateMode2 follows. There are some new defines that you have not seen yet, notably the values and memory addresses needed by the rotation backgrounds, such as the following:

```
#define REG_BG2X      *(volatile unsigned int*)0x4000028
#define REG_BG2Y      *(volatile unsigned int*)0x400002C
#define REG_BG2PA     *(volatile unsigned short *)0x4000020
#define REG_BG2PB     *(volatile unsigned short *)0x4000022
#define REG_BG2PC     *(volatile unsigned short *)0x4000024
#define REG_BG2PD     *(volatile unsigned short *)0x4000026
```

which are used when rotating, scaling, and translating the background. The new rotation background defines are also needed:

```
#define ROTBG_SIZE_128x128  0x0
#define ROTBG_SIZE_256x256  0x4000
#define ROTBG_SIZE_512x512  0x8000
#define ROTBG_SIZE_1024x1024 0xC000
```

We'll be using the `ROTBG_SIZE_128x128` define to set up the background. The key to this program, and to rotating backgrounds, is the RotateBackground function:

```
void RotateBackground(int ang, int cx, int cy, int zoom)
{
    center_y = (cy * zoom) >> 8;
    center_x = (cx * zoom) >> 8;

    DX = (x_scroll - center_y * SIN[ang] - center_x * COS[ang]);
    DY = (y_scroll - center_y * COS[ang] + center_x * SIN[ang]);

    PA = (COS[ang] * zoom) >> 8;
    PB = (SIN[ang] * zoom) >> 8;
    PC = (-SIN[ang] * zoom) >> 8;
    PD = (COS[ang] * zoom) >> 8;
}
```



Unfortunately, as you can see from this function, the GBA doesn't support hardware translation of the background in order to rotate it, as you must perform the rotation with your own code. The GBA does have registers set aside to actually do the pixel-by-pixel rotation, so at least that more difficult aspect is handled by the hardware.

The only prerequisite for this program is an external file called rotation.h, which must be in the same folder as the main program file. I won't list the file here because it's too long, and the listing is filled with long hexadecimal numbers. This file is necessary in order to perform the background rotation, as it contains the precalculated values for every one of the 360 degrees of rotation for both sine and cosine! Simply grab this file off the CD-ROM and put it in the RotateMode2 project folder. I have a different solution to sine and cosine in the next chapter that pre-calculates the sine and cosine at the start of the program, but this adds a short delay to the program's startup. For this reason, I leave you with these two solutions and let you decide which is better for your purposes.

Now let's get on with the full source code for this bad boy.

```
////////////////////////////////////  
// Programming The Game Boy Advance  
// Chapter 6: Tile-Based Video Modes  
// RotateMode2 Project  
// main.c source code file  
////////////////////////////////////  
  
#define MULTIBOOT int __gba_multiboot;  
MULTIBOOT  
  
#include "rotation.h"  
#include "tiles.pal.c"  
#include "tiles.raw.c"  
#include "tilemap.h"  
  
//prototypes  
void DMAFastCopy(void*, void*, unsigned int, unsigned int);  
void RotateBackground(int, int, int, int);
```

```

//defines needed by DMAFastCopy
#define REG_DMA3SAD *(volatile unsigned int*)0x40000D4
#define REG_DMA3DAD *(volatile unsigned int*)0x40000D8
#define REG_DMA3CNT *(volatile unsigned int*)0x40000DC
#define DMA_ENABLE 0x80000000
#define DMA_TIMING_IMMEDIATE 0x00000000
#define DMA_16 0x00000000
#define DMA_32 0x04000000
#define DMA_32NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_32)
#define DMA_16NOW (DMA_ENABLE | DMA_TIMING_IMMEDIATE | DMA_16)

//background movement/rotation registers
#define REG_BG2X *(volatile unsigned int*)0x4000028
#define REG_BG2Y *(volatile unsigned int*)0x400002C
#define REG_BG2PA *(volatile unsigned short *)0x4000020
#define REG_BG2PB *(volatile unsigned short *)0x4000022
#define REG_BG2PC *(volatile unsigned short *)0x4000024
#define REG_BG2PD *(volatile unsigned short *)0x4000026

//background 2 stuff
#define REG_BG2CNT *(volatile unsigned short *)0x400000C
#define BG2_ENABLE 0x400
#define BG_COLOR256 0x80

//background constants
#define ROTBG_SIZE_128x128 0x0
#define ROTBG_SIZE_256x256 0x4000
#define ROTBG_SIZE_512x512 0x8000
#define ROTBG_SIZE_1024x1024 0xC000
#define CHAR_SHIFT 2
#define SCREEN_SHIFT 8
#define WRAPAROUND 0x1
#define BG_MOSAIC_ENABLE 0x40

```

```

//video-related memory
#define REG_DISPCNT      *(volatile unsigned int*)0x4000000
#define BGPaletteMem    ((unsigned short *)0x5000000)
#define REG_DISPSTAT    *(volatile unsigned short *)0x4000004

#define BUTTON_A        1
#define BUTTON_B        2
#define BUTTON_RIGHT    16
#define BUTTON_LEFT     32
#define BUTTON_UP       64
#define BUTTON_DOWN     128
#define BUTTON_R        256
#define BUTTON_L        512
#define BUTTONS         (*(volatile unsigned int*)0x04000130)

#define CharBaseBlock(n)    (((n)*0x4000)+0x6000000)
#define ScreenBaseBlock(n) (((n)*0x800)+0x6000000)
#define SetMode(mode) REG_DISPCNT = (mode)

//some variables needed to rotate the background
int x_scroll=0,y_scroll=0;
int DX=0,DY=0;
int PA,PB,PC,PD;
int zoom = 2;
int angle = 0;
int center_y,center_x;

////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////
int main(void)
{

```

```

int n;
int charbase = 0;
int screenbase = 31;

unsigned short * bg2map = (unsigned short *)ScreenBaseBlock(screenbase);

//set up background 0
REG_BG2CNT = BG_COLOR256 | ROTBG_SIZE_128x128 |
    (charbase << CHAR_SHIFT) | (screenbase << SCREEN_SHIFT);

//set video mode 0 with background 0
SetMode(2 | BG2_ENABLE);

//set the palette
DMAFastCopy((void*)tiles_Palette, (void*)BGPaletteMem, 256, DMA_16NOW);

//set the tile images
DMAFastCopy((void*)tiles_Tiles, (void*)CharBaseBlock(0), 256/4, DMA_32NOW);

//copy the tile map into background 0
DMAFastCopy((void*)tiles_Map, (void*)bg2map, 256/4, DMA_32NOW);

while(1)
{
    while(!(REG_DISPSTAT & 1));

    //use the hardware to scroll around some
    if(!(BUTTONS & BUTTON_LEFT)) x_scroll--;
    if(!(BUTTONS & BUTTON_RIGHT)) x_scroll++;
    if(!(BUTTONS & BUTTON_UP)) y_scroll--;
    if(!(BUTTONS & BUTTON_DOWN)) y_scroll++;
    if(!(BUTTONS & BUTTON_A)) zoom--;
    if(!(BUTTONS & BUTTON_B)) zoom++;
    if(!(BUTTONS & BUTTON_L)) angle--;

```



```

    if(!(BUTTONS & BUTTON_R)) angle++;

    if(angle > 359)
        angle = 0;
    if(angle < 0)
        angle = 359;

    //rotate the background
    RotateBackground(angle, 64, 64, zoom);

    while((REG_DISPSTAT & 1));

    //update the background
    REG_BG2X = DX;
    REG_BG2Y = DY;
    REG_BG2PA = PA;
    REG_BG2PB = PB;
    REG_BG2PC = PC;
    REG_BG2PD = PD;

    while((REG_DISPSTAT & 1));
    for(n = 0; n < 100000; n++);

}
}

////////////////////////////////////
// Function: RotateBackground
// Helper function to rotate a background
////////////////////////////////////
void RotateBackground(int ang, int cx, int cy, int zoom)
{
    center_y = (cy * zoom) >> 8;
    center_x = (cx * zoom) >> 8;
}

```

```

    DX = (x_scroll - center_y * SIN[ang] - center_x * COS[ang]);
    DY = (y_scroll - center_y * COS[ang] + center_x * SIN[ang]);


    PA = (COS[ang] * zoom) >> 8;
    PB = (SIN[ang] * zoom) >> 8;
    PC = (-SIN[ang] * zoom) >> 8;
    PD = (COS[ang] * zoom) >> 8;
}

////////////////////////////////////
// Function: DMAFastCopy
// Fast memory copy function built into hardware
////////////////////////////////////
void DMAFastCopy(void* source, void* dest, unsigned int count,
    unsigned int mode)
{
    if (mode == DMA_16NOW || mode == DMA_32NOW)
    {
        REG_DMA3SAD = (unsigned int)source;
        REG_DMA3DAD = (unsigned int)dest;
        REG_DMA3CNT = count | mode;
    }
}

```

## Summary

This has been one of the most challenging chapters of the book so far and was much more involved than the relatively simple video modes covered in the last chapter. However, now that you have conquered this difficult subject, you are on the downhill stretch of mastering the GBA, because you have now overcome the two most difficult subjects: bitmap and tile video modes. Get ready for even more graphics, as the next chapter finally covers the fascinating subject of sprite programming. Chapter 7 will involve even more of the subjects



covered in Chapters 5 and 6, giving you plenty of opportunity to practice using scrolling backgrounds and the like.

## Challenges

The following challenges will help to reinforce the material you have learned in this chapter.

**Challenge 1:** The TileMode0 program used a 256 x 256 tile map and also tiled image. Modify the tiles and the source code, changing the tile map to 512 x 512, and note the differences in how fast the program runs.

**Challenge 2:** The RotateMode2 program uses a 128 x 128 tile map and corresponding tile image. However, the GBA supports rotation backgrounds of up to 1,024 x 1,024 in size! Modify the program to make use of this greater resolution.

## Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. What are the three video modes 0, 1, and 2 called?
  - A. Tiled backgrounds
  - B. Bitmap backgrounds
  - C. Cascading backgrounds
  - D. Provocative backgrounds
2. How many backgrounds are available on the GBA, regardless of the video mode?
  - A. 3
  - B. 2
  - C. 4
  - D. 1
3. Which video mode features four tiled backgrounds?
  - A. Mode 4
  - B. Mode 0

- C. Mode 7
  - D. Mode 2
4. Which backgrounds are supported by video mode 2?
- A. 1 and 2
  - B. 1, 2, and 3
  - C. 4, 5, and 6
  - D. 2 and 3
5. Which video mode uses the two rotation backgrounds?
- A. Mode 0
  - B. Mode 2
  - C. Mode 3
  - D. Mode 1
6. What three backgrounds are supported by video mode 1?
- A. 0, 1, and 2
  - B. 1, 2, and 3
  - C. 2 and 3
  - D. 3, 4, and 5
7. Which of the three mode 1 backgrounds is considered a rotation background?
- A. 3
  - B. 1
  - C. 2
  - D. 0
8. What are the following registers used for: REG\_BG2PA, REG\_BG2PB, REG\_BG2PC, and REG\_BG2PD?
- A. Background scrolling
  - B. Background transparency
  - C. Background hosiery
  - D. Background rotation
9. How many registers are required to perform a DMA fast memory copy?
- A. 4
  - B. 2



- C. 3
- D. 1

10. True/False: Does the GBA support hardware scrolling of the background?

- A. True
- B. False