Chapter 7

Rounding Up Sprites

This chapter is an extension of the previous two chapters, which introduced the concept of bitmapped and tiled backgrounds and provided an overview of the six video modes available on the GBA. This chapter takes it a significant step further by building upon the concepts presented in those two chapters and adding additional material, most notably of which is coverage of sprites. Until now, all the graphics programming you have been doing has been directly on backgrounds in one or another of the six video modes. Some of the video modes were rather easy to draw upon, while others were significantly more difficult to get a pixel lit.

The point of this chapter is to refine that base knowledge and develop a sprite handler that incorporates all the code needed to deal with all the video modes and backgrounds built into the GBA, while at the same time providing significant coverage of the hardware sprite blitter. By the time you have finished this chapter, you will have a solid understanding of the most important aspect of Game Boy Advance programming: sprites.

Ready? Okay, let's go! This is the first "real-world" chapter that is more than a deluge of information—it actually gets into some of the fun factor involved in writing GBA programs. Until now, there was so much prerequisite information needed that it wasn't possible to write even a simple game—well, at least not a sprite-based game, which is the point after all. Here are the major topics presented in this chapter:

- Let's get serious: programming sprites
- Drawing a single sprite
- Creating a sprite handler
- Sprite special effects

Let's Get Serious: Programming Sprites

What is a sprite, you ask? A sprite is a small, easily moved object that has a defined shape (usually withing a rectangular image), and is the focus of the action in a 2D game (of which there is a majority on the GBA). A sprite can be the player's space ship, role-playing character, baseball player, as well as a baseball, bird, missile, explosion, alien creature, or even a vigilante's car. On most PC games, the game code itself must draw these sprites pixel by pixel; left to right, top to bottom. This is known as a software sprite. Console hardware, on the other hand, has traditionally provided hardware that can draw entire sprite bitmaps in a single call or instruction. We call these hardware sprites.

Sprites have been a part of video games since the earliest days. Indeed, you can consider the ball and paddles of Pong to be sprites. Not all video game machines have had hardware support for sprites. The term *software sprite* has become common on systems like the PC where bitmaps reign supreme. The GBA has significant hardware support for sprites. In this chapter you will learn about object attribute memory (OAM) and examine a sprite handler to make sprites more manageable. Look at the definition of a sprite. We need to be able to specify a position and image for each of the sprites we want to display. The GBA also gives us a number of options that can be applied to the sprites. Setting these properties is the purpose of the OAM.

The GBA has built-in hardware support for up to 128 sprites. Each of the 128 sprites has the following attributes:

Tile Index. This specifies the image tile (or tiles) that holds the image of the sprite.

Size. Sprites can be several different sizes using from one to 64 "tiles" for the image. We'll talk more about tiles and how they relate to sprites in the next section.

Position. This specifies the horizontal and vertical position of the sprite on the screen.

Priority. This defines in which of four layers the sprite will drawn, allowing the sprite to show in front of or behind other graphics.

Palette Information. The tile graphics can use either 4 or 8 bits per pixel. One of 16 palettes must be chosen for 4 bit-per-pixel graphics.

Mosaic Effect. Sprites can have a mosaic effect applied.

Flip. Sprites can be flipped horizontally, vertically, or both.

Rotation and Scaling. Sprites can be rotated and scaled. Attributes specify which of 32 rotation and scaling parameters will be used.

These attributes are packed into 6 bytes of memory per sprite, but these structures are spaced 8 bytes apart. The extra 2 bytes of each chunk of memory are used for defining the sprite rotation and scaling parameters.

Moving Images the Simple Way

It is very easy to create moving objects on the screen using sprites, so let's look at each of these steps in detail.. The basic steps are as follows:

1. Create the sprite graphics.
2. Initialize the OAM.
3. Enable sprites in REG.DISPCNT.
4. Set the sprite attributes in OAM during VBlank for each sprite you want to display.
5. Any time an object moves or changes graphics, update the attributes in the OAM.

Creating Sprite Graphics

The image for a sprite is made up of one or more *tiles*. Each tile is an 8 x 8 bitmap of either 4 or 8 bits per pixel. The typical sprite is often larger than this and is seldom a solid rectangle.

Software sprites—moving objects drawn into a bitmap by software—handle the issue of transparency in a couple different ways.

One way to encode transparency is to have a *mask*—typically a 1 bit-per-pixel bitmap showing where there is solid sprite and where there is transparency. This is exactly like a crude alpha channel (where a color is made up of three channels (red, green, and blue) along with the alpha or transparency channel). One can then use this mask to erase what was in the bitmap and to combine the new graphics into the picture. This can be quite expensive in software.

Another way to encode transparency is to have a specific color that represents transparent pixels. This keeps one from wasting space on a separate mask bitmap but still requires a comparison for every pixel—again very expensive.

The GBA sprite hardware uses a variant of this second method. Since sprites always use color palettes (of either 16 or 256 colors), color index zero is set aside for transparency. This is true for every graphics mode on the GBA that uses palettes—the first color entry in the palette specifies the transparent pixels of the image. Therefore, when you are creating game graphics in a graphic editor, be sure to modify the palette so that the transparent color is in the first position.

Creating Tiles

The easiest way to create the data for tiles is to draw the images in a larger bitmap and then use a utility program to chop the bitmap up into the tile data. We'll use the same graphics converter we used before, gfx2gba for this purpose, along with another tool called pcx2sprite that is particularly suitable for single sprites. There is an ideal width to use for creating sprite tiles. This width is different depending on the color depth you are using. We'll see why this is important in a little bit.

For This Color Depth	**Use This Width**
16 colors	256 pixels
256 colors	128 pixels

I find it easiest to work on these graphics by zooming in on them. A zoom factor of 4 x to 6 x works really well for me. Turn on the Grid option and set the grid to 8 pixels for width and height. Each square of the grid shows the boundaries of one tile. Using the ideal bitmap widths you will have 32 tiles per row for 16-color tiles or 16 tiles per row for 256-color tiles

Converting Tiles

The pcx2sprite program has no parameters, and it is convenient because you can just drag a .pcx file over the program file in Windows Explorer to convert the file (note that only 256 colors are supported). The other tool that is still needed for backgrounds is gfx2gba. The parameters most commonly used with gfx2gba are as follows:

-t8	Sets the size of a tile to 8 x 8 pixels.
-c32k	Use hicolor for mode 3 (not needed for tiled modes). The default is 256 even if the source files are 16 color files.
-pFilename	Sets the name for the palette file.
-fsrc	Output will be in source code format.

Bitmap graphics modes 3, 4, and 5 use the first half of the Sprite Tile VRAM for part of their buffers. This limits the sprite tiles to the last 16 KB of VRAM—512 16-color tiles or 256 256-color tiles. In both of these cases the first usable sprite tile is index 512.

Using Sprites

The mechanics of using sprites are fairly simple but can cause some annoying problems. Some examples: The data stored in the OAM is packed with various single and multibit quantities. The OAM needs to be updated during the vertical refresh period. Sprites of the same display priority are sorted by OAM position (lowest sprite number has priority).

Because of these issues, most games use a separate buffer typically known as *shadow memory*. The shadow has the identical bit layout as the OAM but is located in RAM where it can be modified without affecting the display. The sprite attribute array is then copied into the OAM at the start of the vertical refresh. Note that you don't need to bother with the sprite images after having initially copied them to the data portion of OAM..

Larger Sprites

Let's face it. Sprites that are 8 x 8 pixels just aren't very large. To make recognizable characters and animations, you'll want to use multiple tiles per sprite. Here's where the natural size of the source bitmaps comes into play. As noted before, the natural size is different for 16- and 256-color sprites. That's because the 256-color tiles use twice as much memory as the 16-color tiles.

The 256-color tiles each take 64 bytes of memory. The first 256-color tile uses the memory from 0x06010000 to 0x0601003f. This is the same memory that tiles 0 and 1 take up in 16-color tiles. The second 256-color tile starts at 0x06010040—the same address as tile index 2 of 16-color tiles. Any guesses what index value we use for this second 256-color tile? Right, index 2. The formula for converting a tile index to memory addresses remains the same in the two modes, but the 256-color tiles only use the even indexes. This also means that there are only 512 tiles in this mode.

Linear Tile Layouts

The tile arrangements that you've seen so far are the default and are known as the 2D tile arrangement. This layout is very convenient for the tile artists since they can simply draw the larger sprites in a bitmap and the conversion tools directly give us usable sprite orderings. For many games this is fine because 1,024 (or 512) tiles are often enough for all the sprites in one level of a game.

There are many times when this is not the case. Games with a lot of large characters or long animation sequences will not be able to fit all their graphics for a level in the 32 KB provided for the sprite tiles. This means you need to dynamically load graphics data from your game ROM (or EWRAM) during gameplay.

Dealing with the 2D tile layout while dynamically loading sprite tiles is inefficient and likely to cause severe brain damage to the programmer.

You can flip one bit in REG.DISPCNT and change the layout of tiles for all sprites.

```
REG.DISPCNT |= DC_SPRITESEQ;
```

This bit sets the sprite's tiles to a sequential layout. This means that instead of adding 32 to get the index of the first tile on the next row, this tile immediately follows the last tile on the previous row. This mode keeps all the graphics data for a single sprite image in contiguous memory allowing a single DMA transfer to move an entire image. There are tools that will convert a bitmaps into tiles arranged in this manner. The gfx2gba utility will do this for you using its "tiling" option. Another tool that comes with the HAM SDK, called the "Bitmap ReSizer" (see Start, Programs, HAM Development Kit, Tools, Bitmap ReOrganizer), that will convert a tiled bitmap into a linear format, and is a windowed program, rather than a command-line program. Of course, this program works only with bitmaps, and does not convert them. You still need to use gfx2gba to convert the image.

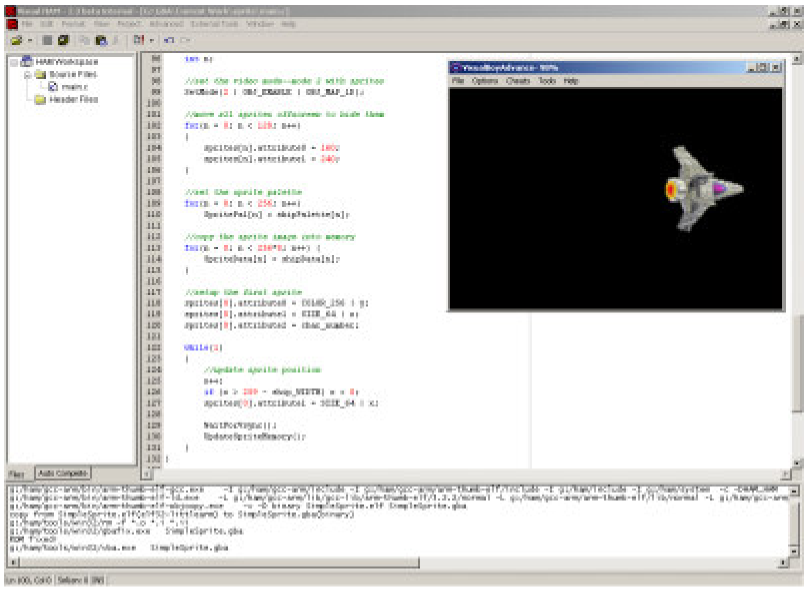
Drawing a Single Sprite

Now that you've had a little theory, how about delving into some real code for drawing sprites on the screen? The first sample program in this chapter is called SimpleSprite and is shown in Figure 7.1. As you can see from the figure, the SimpleSprite program draws a spaceship sprite and moves it across the screen, warping to the left when it hits the right side of the screen.

This program has just enough code to get you going, without a lot of complicated extraneous stuff because I want you to first grasp how to convert a sprite file and then display it on the screen. In later sections, I'll get into special effects like rotation, scaling, and transparency, as well as how to handle multiple sprites. In fact, you will be writing a sprite handler before this chapter is over.

Converting the Sprite

The first thing you need in a sprite-based program is an image of a sprite, which can be anything: a spaceship, car, soldier, lemming, ghost, hero, monster. Basically, this is the key to the game, your graphics! Figure 7.2 shows an image of a spaceship stored in ship.pcx. You can find this file in \Sources\Chapter07\SimpleSprite, along with the source files for this project.



```
    int n;

    //set the video mode--mode 2 with sprites
    SetMode(2 | OBJ_ENABLE | OBJ_MAP_1D);

    //move all sprites offscreen to hide them
    for(n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = shipPalette[n];

    //copy the sprite image into memory
    for(n = 0; n < 256*8; n++) {
        SpriteData[n] = shipData[n];
    }

    //setup the first sprite
    sprites[0].attribute0 = COLOR_256 | y;
    sprites[0].attribute1 = SIZE_64 | x;
    sprites[0].attribute2 = char_number;

    while(1)
    {
        //update sprite position
        x++;
        if (x > 239 - ship_WIDTH) x = 0;
        sprites[0].attribute1 = SIZE_64 | x;

        WaitForVsync();
        UpdateSpriteMemory();
    }
}
```

```
g:/ham/gcc-arm/bin/arm-thumb-elf-gcc.exe   -I g:/ham/gcc-arm/include -I g:/ham/gcc-arm/arm-thumb-elf/include -I g:/ham/include -I g:/ham/system  -c -DHAM_HAM
g:/ham/gcc-arm/bin/arm-thumb-elf-ld.exe   -L g:/ham/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.2.2/normal -L g:/ham/gcc-arm/arm-thumb-elf/lib/normal -L g:/ham/gcc-arm
g:/ham/gcc-arm/bin/arm-thumb-elf-objcopy.exe   -v -O binary SimpleSprite.elf SimpleSprite.gba
copy from SimpleSprite.elf(elf32-littlearm) to SimpleSprite.gba(binary)
g:/ham/tools/win32/rm -f *.o *.i *.ii
g:/ham/tools/win32/gbafix.exe   SimpleSprite.gba
ROM fixed!
g:/ham/tools/win32/vba.exe   SimpleSprite.gba
```

Figure 7.1

The SimpleSprite program moves a single sprite across the screen.

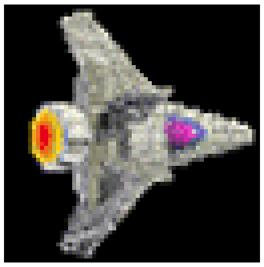


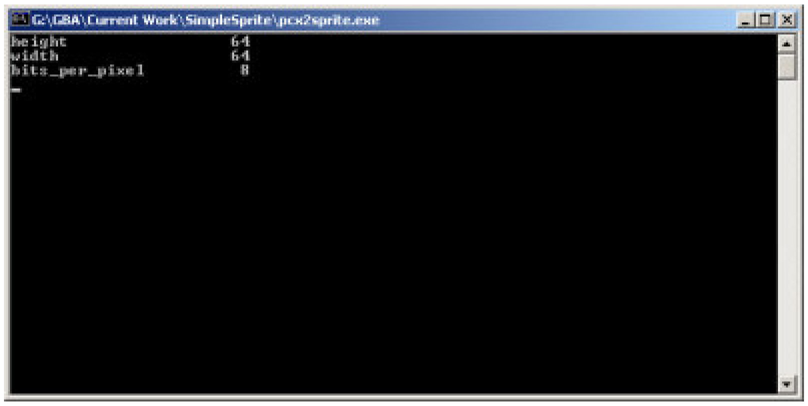
Figure 7.2

The ship.pcx file used as the sprite in the SimpleSprite program.

Also included in the folder for this project is a program called pcx2sprite.exe. This program is really handy for quick conversion of sprites because it doesn't require a command-line interface. You simply drag a .pcx file over the program file in Windows Explorer, and it converts it to a C source code file—which is similar to the files produced by gfx2gba, but pcx2sprite puts the palette and bitmap inside the same file.

You can download pcx2sprite, pcx2gba, and many other utilities, from the Pern Project Web site at http://www.thepernproject.com, operated by Jason Rogers (a.k.a. Dovoto).

Open Windows Explorer, browse to \Sources\Chapter07\SimpleSprite, and locate ship.pcx. Now drag this file over the pcx2sprite.exe file to convert it. The file must be a 256-color .pcx image, otherwise pcx2sprite will output an error message. The output is shown in Figure 7.3.



```
height              64
width               64
bits_per_pixel       8
_
```

Figure 7.3

Output from the pcx2sprite program.

After that is done, you'll have a file called ship.h. If you open the file, you'll see something like this:

```
/**********************************************\
*       ship.h                                 *
*          by dovotos pcx->gba program         *
/**********************************************/
#define  ship_WIDTH   64
```

```
#define  ship_HEIGHT  64


const u16 shipData[] = {
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x1301, 0x0000, 0x0000, 0x0000, 0xFB50, 0x0000, 0x0000,
0x1900, 0xE9AD, 0x0000, 0x0000,
...
```

Only the top few lines are shown from a file that is a few hundred lines long, but this gives you an idea of what the ship.h file looks like after conversion. Since the compiler complains about Dovoto's funky header at the top, I always just delete the header.

The SimpleSprite Source Code

The SimpleSprite program has just a single source listing with all the defines and stuff you need to compile the program in one place. Create a new project in Visual HAM, name it SimpleSprite, and delete the default code in main.c, replaced with the following code listing:

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// SimpleSprite Project
// main.c source code file
/////////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;
MULTIBOOT


typedef unsigned short u16;


#include "ship.h"


//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)
```

```
//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)
#define REG_VCOUNT *(volatile unsigned short*)0x4000006
#define REG_DISPSTAT *(volatile unsigned short *)0x4000004

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)

//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)

//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)

//misc sprite constants
#define OBJ_MAP_2D          0x0
#define OBJ_MAP_1D          0x40
#define OBJ_ENABLE          0x1000

//attribute0 stuff
#define ROTATION_FLAG      0x100
#define SIZE_DOUBLE        0x200
#define MODE_NORMAL        0x0
#define MODE_TRANSPARENT   0x400
#define MODE_WINDOWED       0x800
#define MOSAIC              0x1000
#define COLOR_16            0x0000
#define COLOR_256           0x2000
#define SQUARE              0x0
#define TALL                0x4000
#define WIDE                0x8000

//attribute1 stuff
```

```
#define ROTDATA(n)          ((n) << 9)
#define HORIZONTAL_FLIP     0x1000
#define VERTICAL_FLIP       0x2000
#define SIZE_8              0x0
#define SIZE_16             0x4000
#define SIZE_32             0x8000
#define SIZE_64             0xC000

//attribute2 stuff
#define PRIORITY(n)        ((n) << 10)
#define PALETTE(n)        ((n) << 12)

//sprite structs
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;

//create an array of 128 sprites equal to OAM
Sprite sprites[128];

//function prototypes
void WaitForVsync(void);
void UpdateSpriteMemory(void);

//////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
//////////////////////////////////////////////////////////////
int main(void) {
    signed short x = 10, y = 40;
    signed short xdir = 1, ydir = 1;
    int char_number = 0;
    int n;

    //set the video mode--mode 2 with sprites
      SetMode(2 | OBJ_ENABLE | OBJ_MAP_1D);

    //move all sprites offscreen to hide them
    for(n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = shipPalette[n];

    //copy the sprite image into memory
    for(n = 0; n < 256*8; n++) {
        SpriteData[n] = shipData[n];
    }

    //setup the first sprite
    sprites[0].attribute0 = COLOR_256 | y;
    sprites[0].attribute1 = SIZE_64 | x;
    sprites[0].attribute2 = char_number;

    while(1)
    {
        //update sprite x position
        x += xdir;
        if (x > 239 - ship_WIDTH) x = 0;
```

```
        //update sprite y position
        y += ydir;
        if (y > 159 - ship_HEIGHT)
        {
            y = 159 - ship_HEIGHT;
            ydir = -1;
        }
        if (y < 1)
        {
            y = 1;
            ydir = 1;
        }

        //update sprite attributes with new x,y position
      sprites[0].attribute0 = COLOR_256 | y;
        sprites[0].attribute1 = SIZE_64 | x;

        //wait for vertical retrace
        WaitForVsync();

        //display the sprite
        UpdateSpriteMemory();
      }
}

/////////////////////////////////////////////////////////////
// Function: WaitForVsync
// Waits for the vertical retrace
/////////////////////////////////////////////////////////////
void WaitForVsync(void)
{
    while((REG_DISPSTAT & 1));
}
```

```
////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128*4; n++)
        SpriteMem[n] = temp[n];
}
```

I am basically going to follow a learn-by-doing philosophy in this chapter (and others) because a brief glance at a snippet of code often explains more than a dozen pages of commentary.

Creating a Sprite Handler

Now that you've seen what might be called crude sprite code running, I'd like to show you a few tricks that will make sprite handling more manageable. Writing a game entirely with GBA sprite code is possible, and there's nothing wrong with doing it that way. Many have done that without any trouble. But I prefer to keep track of sprites with a handler, which is basically a struct with basic sprite values stored inside, such as x, y, size, xdir, ydir, and so on. Over the next few sections I'll improve the basic sprite handler so it incorporates more features in time (such as rotation, scaling, and transparency).

What Does the Sprite Handler Do?

The sprite handler basically keeps track of the sprites in the program so you don't have too many global variables floating around. Here's what the struct looks like at first revision:

```
typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;
}SpriteHandler;
```

Actually using the struct involves creating an array of structs:

```
SpriteHandler mysprites[128];
```

Now, with this code in place, you have much more control over the sprites in your program, and you can manage them in large quantity without an exponential increase in the amount of code. In fact, you can simply process all the sprites in a for loop.

The BounceSprite Source Code

Let's put the sprite handler to use, shall we? This section includes a program called BounceSprite, which draws a background image and displays several sprites on the screen, bouncing them around inside the dimensions of the screen. The output is shown in Figure 7.4.

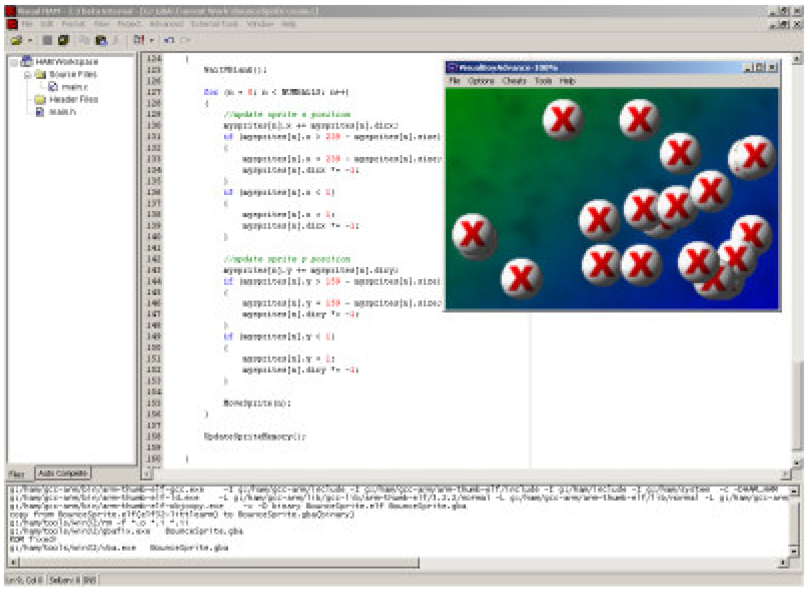
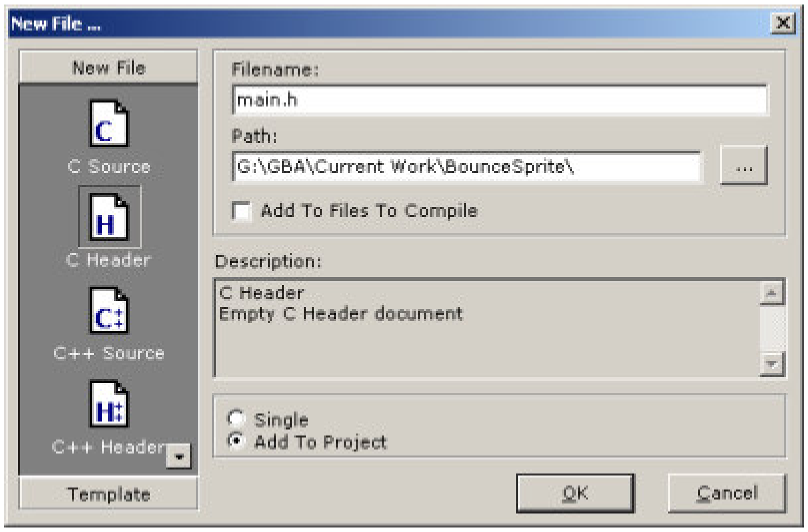


Figure 7.4

The BounceSprite program bounces 10 ball sprites around on the screen.

I have divided this program into two separate source files to make it easier to follow. While the SimpleSprite program was somewhat short, the BounceSprite program is a little more involved because of the new handler code. It also does quite a bit more than the SimpleSprite program, as there are now 10 sprites on the screen, moving independently.

Go ahead and create a new project in Visual HAM called BounceSprite, or you may copy the project off the CD-ROM, located in \Sources\Chapter07\BounceSprite. If you are typing in the program, you'll want to add a new file to the project. Select File, New, New File to bring up the New File dialog box, as shown in Figure 7.5.



Filename: main.h

Path: G:\GBA\Current Work\BounceSprite\

Add To Files To Compile

Description: C Header
Empty C Header document

Single [✓ Add To Project]

Figure 7.5

The New File dialog box is used to add new source files to the project.

Select the C Header file type at the left, then type "main.h" for the new file name, and be sure to select the option Add To Project before closing the dialog box. The new main.h file should now be in your BounceSprite project.

The Header File

Type the following code into the main.h file:

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// BounceSprite Project
// main.h header file
/////////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include "ball.h"
#include "bg.raw.c"

//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)
```

```
//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)

//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)

//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;

//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)

//misc sprite constants
#define OBJ_MAP_2D 0x0
#define OBJ_MAP_1D 0x40
#define OBJ_ENABLE 0x1000
#define BG2_ENABLE0x400

//attribute0 stuff
#define ROTATION_FLAG        0x100
#define SIZE_DOUBLE          0x200
#define MODE_NORMAL          0x0
```

```
#define MODE_TRANSPARENT     0x400
#define MODE_WINDOWED        0x800
#define MOSAIC               0x1000
#define COLOR_256            0x2000
#define SQUARE               0x0
#define TALL                 0x4000
#define WIDE                 0x8000

//attribute1 stuff
#define SIZE_8               0x0
#define SIZE_16              0x4000
#define SIZE_32              0x8000
#define SIZE_64              0xC000

//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;

//create an array of 128 sprites equal to OAM
Sprite sprites[128];

typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;

}SpriteHandler;
```

```
SpriteHandler mysprites[128];


#endif
```

The Main Source File

The main source code file for BounceSprite is listed next. Since most of the building blocks are now stored in main.h, this code listing is much more manageable than it would have otherwise been. Note the **#define NUMBALLS 10** definition. You may change that to another number if you wish, to see how the program performs with differing numbers of sprites. Take care, however, because this program is using large sprites, so there are not a full 128 slots available in memory.

```
/////////////////////////////////////////////////////////////

// Programming The Game Boy Advance

// Chapter 7: Rounding Up Sprites

// BounceSprite Project

// main.c source code file

/////////////////////////////////////////////////////////////


#define MULTIBOOT int __gba_multiboot;

MULTIBOOT


#include "main.h"


#define NUMBALLS 10


/////////////////////////////////////////////////////////////

// Function: HideSprites

// Moves all sprites off the screen

/////////////////////////////////////////////////////////////

void HideSprites()

{

    int n;

    for (n = 0; n < 128; n++)

    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}

////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}
```

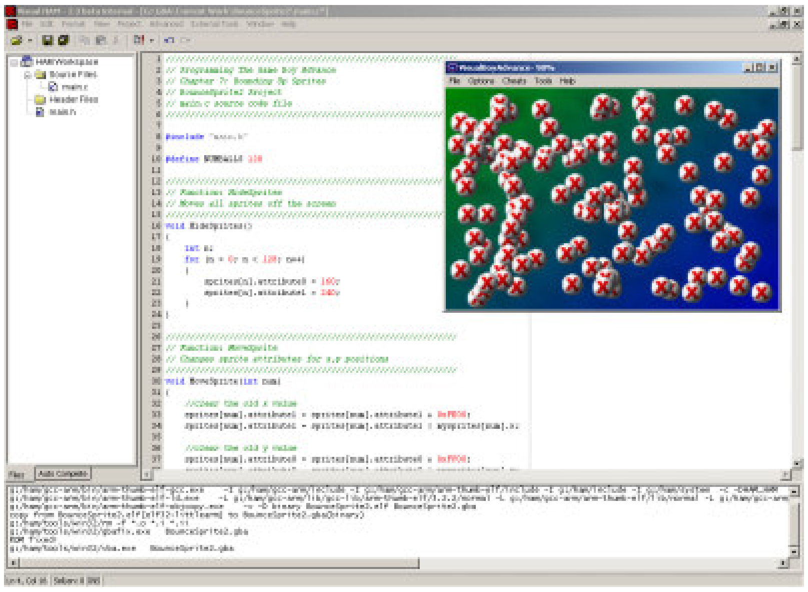
```
//////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
//////////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int color, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = color | y;

    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }

    sprites[num].attribute1 = sprite_size | x;
}

//////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
//////////////////////////////////////////////////////////////
```

```
void WaitVBlank(void)
{
    while(*ScanlineCounter < 160);
}

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

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = ballPalette[n];

    //load ball sprite
    for(n = 0; n < 512; n++)
        SpriteData3[n] = ballData[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image (512)
    for (n = 0; n < NUMBALLS; n++)
    {
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH,
            COLOR_256, 512);
        while (mysprites[n].dirx == 0)
            mysprites[n].dirx = rand() % 6 - 3;
        while (mysprites[n].diry == 0)
            mysprites[n].diry = rand() % 6 - 3;
    }

    //main loop
    while(1)
    {
        //keep the screen civil
        WaitVBlank();

        for (n = 0; n < NUMBALLS; n++)
        {
            //update sprite x position
            mysprites[n].x += mysprites[n].dirx;
            if (mysprites[n].x > 239 - mysprites[n].size)
            {
                mysprites[n].x = 239 - mysprites[n].size;
                mysprites[n].dirx *= -1;
            }
            if (mysprites[n].x < 1)
            {
                mysprites[n].x = 1;
                mysprites[n].dirx *= -1;
            }

            //update sprite y position
            mysprites[n].y += mysprites[n].diry;
            if (mysprites[n].y > 159 - mysprites[n].size)
            {
                mysprites[n].y = 159 - mysprites[n].size;
                mysprites[n].diry *= -1;
            }
            if (mysprites[n].y < 1)
            {
                mysprites[n].y = 1;
                mysprites[n].diry *= -1;
            }

            //update the sprite properties
            MoveSprite(n);
        }
        //copy all sprites into object attribute memory
        UpdateSpriteMemory();
    }
}
```

Resizing the Ball Sprite

This sprite handler is not fully featured, but it does allow you to change the sprite size on the fly without modifying the source code in any way. The reason this is possible is because the pcx2sprite program supplies the image width and height in the converted source file (such as ball.h). The InitSprite function has a parameter that specifies the dimensions of the sprite. For all practical purposes, you will be using square sprites, with the same width and height, so it makes sense to use a single parameter, size, for the dimensions. If your particular situation calls for rectangular sprite images, then you may modify the program to use a width and height. This is but a stepping stone on the way to delivering sprites to the screen, however.

As Figure 7.6 shows, we're on the right track, but these GBA sprites are capable of animation, not to mention special effects like scaling and rotation. This minor change is also a demonstration of sprite performance, as the BounceSprite2 program increases the NUMBALLS to 128 but is otherwise identical to the BounceSprite program. There is an additional need to grab frames out of an image to use for individual sprites, so what we need is a function for loading sprites, in the traditional sense. Let's get into some special effects now, and I'll cover tiled sprite images at the same time.



```
//////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Bounding Up Sprites
// BounceSprite2 Project
// main.c source code file
//////////////////////////////////////////////////////////

#include "main.h"

#define NUMBALLS 128

//////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

//////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
//////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
```

Figure 7.6

Resizing the ball image to 16 x 16 requires absolutely no change in the source code, because image dimensions are stored in the converted file by pcx2sprite.

Sprite Special Effects

The GBA is a highly optimized sprite-blitting machine and provides some extra features in addition to transparency—which, in case you didn't notice, was in use with the BounceSprite program, and this was all done automatically by the GBA hardware. The traditional use of transparency is to show only solid pixels in the sprite, thus allowing transparent pixels to show what is behind the sprite (on the background, for instance). The GBA takes care of this for you—something that requires quite a heaping of theory and code in other platforms! There is another form of transparency—or rather, translucency—and that is called *alpha blending*.

Implementing Alpha Blending

Alpha blending is a technique whereby one image is translucent, allowing the images behind it to show through, while still remaining visible itself. The effect is extremely useful not only for sprite special effects but also for displaying dialogs or other images over the game screen while still showing the game in the background. One such example is an options screen that might appear when you press the Start button, providing options such as restart, save, load, and quit. Displaying a menu in a translucent dialog has a very nice effect on the screen, with the appearance of being less invasive.

There are really only a few things that you must do to enable alpha blending of foreground sprites. First, you'll need two new registers, **REG_BLDMOD** and **REG_COLEV**:

```
//transparency registers

#define REG_BLDMOD     *(unsigned short*)0x4000050
```

```
#define REG_COLEV        *(unsigned short*)0x4000052
```

As usual, these are pointers to memory addresses where the hardware defines these features. Put into use, translucency (alpha blending) can be turned on with the following code:

```
//set transparency level
REG_BLDMOD = (1 << 4) | (1 << 10);
REG_COLEV = (8) + (8 << 8);
```

When you set these two registers as shown, any sprites that have transparency enabled will appear so, while those without the option will be displayed normally. Here are the two definitions of the options used to set up a sprite for transparency:

```
#define MODE_NORMAL          0x0
#define MODE_TRANSPARENT     0x400
```

Where do you use these definitions? The object attribute memory (OAM) sprite struct has four attributes that are used to specify various options for each sprite. One such attribute is attribute0, which takes care of the color mode, rotation factor, the vertical (y) position, as well as the transparency of the sprite. It is unfortunate that so much is crammed into each attribute; perhaps you will figure out a way to rewrite the struct with individual attributes for each setting? It would require a lot of tinkering to figure out all the bits but may be worth the attempt. I will stick with the standard way of modifying sprites. I realize it is confusing that *transparent* is used to mean alpha blending, as well as a transparent sprite color. But I think we can get away with the terminology when dealing with the GBA because the traditional use of *transparency* is handled by the GBA hardware already, so you really aren't going to be dealing with that aspect at all (I was tempted to say "very often," but really, the GBA does this entirely for you). Here is how you would set up attribute0 to enable transparency:

```
sprites[num].attribute0 = COLOR_256 | MODE_TRANSPARENT | y;
```

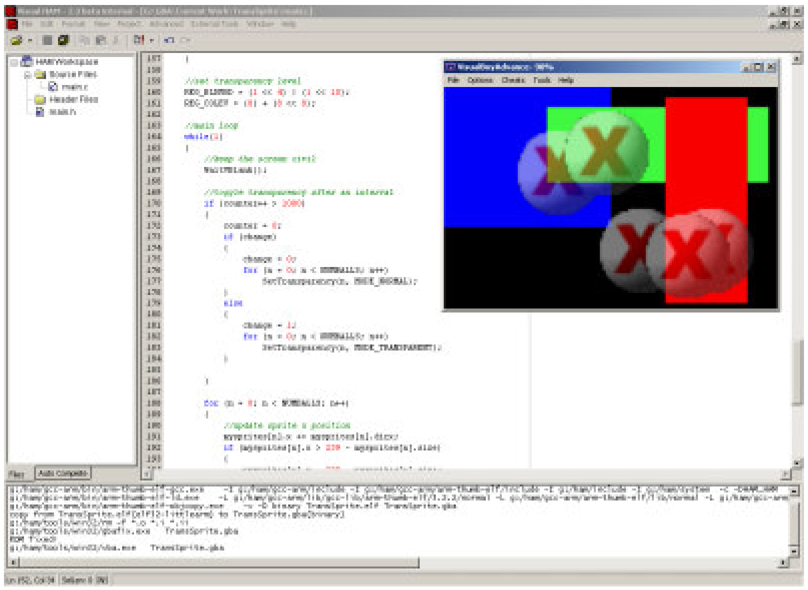
Blitting Transparent Sprites

I've written a sample program called TransSprite, which I'd like to walk you through. It's a short program, like all the other sample programs in this chapter, so it takes but a few minutes to type it in. As usual, create a new project in Visual HAM. Name the new project TransSprite. The program is located on the CD-ROM under \Sources\Chapter07\TransSprite. Figure 7.7 shows the TransSprite program running. In this particular screen shot, the sprites are all transparent (or rather, translucent, or alpha blended).

If you watch the TransSprite program run for a few seconds, you'll see the sprites alternate from solid to transparent. Figure 7.8 shows the two variations of the program side by side.

In this case, the VisualBoyAdvance screen is shown actual size (whereas I normally run it at 2 x).

Well, it seems as if you have enough to go on already to implement alpha blending of sprites. How about a sample program, just to put into good use what you have learned? It's great being able to get instant feedback on some new technology—one of the joys of programming (and the reason why there is a field called *computer science*).



```
    }

    //set transparency level
    REG_BLDMOD = (1 << 4) | (1 << 10);
    REG_COLEV = (8) + (8 << 8);

    //main loop
    while(1)
    {
        //keep the screen civil
        WaitVBlank();

        //toggle transparency after an interval
        if (counter++ > 1000)
        {
            counter = 0;
            if (change)
            {
                change = 0;
                for (n = 0; n < NUMBALLS; n++)
                    SetTransparency(n, MODE_NORMAL);
            }
            else
            {
                change = 1;
                for (n = 0; n < NUMBALLS; n++)
                    SetTransparency(n, MODE_TRANSPARENT);
            }

        }

        for (n = 0; n < NUMBALLS; n++)
        {
            //update sprite x position
            mysprites[n].x += mysprites[n].dirx;
            if (mysprites[n].x > 239 - mysprites[n].size)
            {
```

Figure 7.7

The TransSprite program demonstrates how to turn on alpha blending, which allows sprites to be drawn transparently.

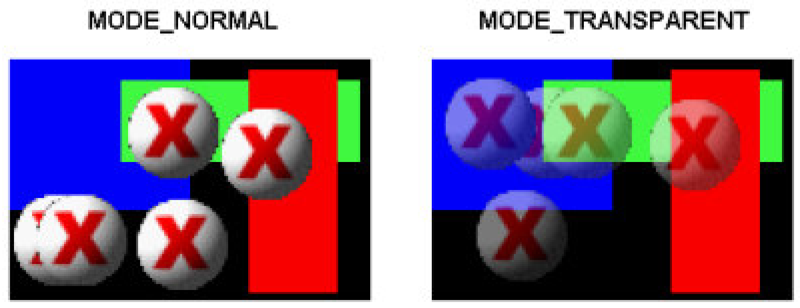


Figure 7.8

The TransSprite program alternates the sprites from MODE_NORMAL to MODE_TRANSPARENT.

Now let's create a project for this program. In Visual HAM, open the File menu and select New, New Project. Name the project TransSprite. As you did earlier with the previous project, add a new header file called main.h.

The TransSprite Header File

The header file for the TransSprite program is called main.h and contains all the includes, defines, arrays, variables, and GBA registers needed by the main program and is extremely welcome because none of these statements ever change while the program is running, so it's better to hide them away. As long as you know what all of these statements are for, and how they were derived, that's the whole point of the lesson. So let's hide them away in main.h.

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// TransSprite Project
// main.h header file
/////////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include "ball.h"
#include "bg.raw.c"

//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)
```

```
//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)

//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;

//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)

//transparency stuff
#define REG_BLDMOD     *(unsigned short*)0x4000050
#define REG_COLEV      *(unsigned short*)0x4000052

//misc sprite constants
#define OBJ_MAP_2D            0x0
#define OBJ_MAP_1D            0x40
#define OBJ_ENABLE            0x1000
#define BG2_ENABLE            0x400

//attribute0 stuff
#define ROTATION_FLAG         0x100
#define SIZE_DOUBLE           0x200
#define MODE_NORMAL           0x0
#define MODE_TRANSPARENT      0x400
#define MODE_WINDOWED         0x800
#define MOSAIC                0x1000
#define COLOR_256             0x2000
#define SQUARE                0x0
#define TALL                  0x4000
#define WIDE                  0x8000

//attribute1 stuff
#define SIZE_8                0x0
```

```
#define SIZE_16              0x4000
#define SIZE_32              0x8000
#define SIZE_64              0xC000

//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;

//create an array of 128 sprites equal to OAM
Sprite sprites[128];

typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;
    int colormode;
    int trans;

}SpriteHandler;

SpriteHandler mysprites[128];

#endif
```

Did you notice the two new elements in the SpriteHandler struct that I snuck in? The new elements are **colormode** and **trans** and are provided to allow each sprite to have separate and distinct properties from all others. You'll be adding more items to the struct in later projects as well, so don't get too comfortable with the sprite handler just yet.

The TransSprite Source File

The main source code file for TransSprite may be the most lengthy code listing of the chapter so far, but it is not inefficient by any means. Given what this program does, it is quite small compared to the amount of code needed to implement alpha blending on another platform. Now here is the code listing for the main source code file:

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// TransSprite Project
// main.c source code file
/////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"

#define NUMBALLS 5

/////////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
/////////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

/////////////////////////////////////////////////////////////
```

```
// Function: MoveSprite
// Changes sprite attributes for x,y positions
////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}

////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}
```

The InitSprite () function that follows is where the sprite attribute is modified to enable alpha blending of the sprite. I have highlighted the key line in bold text.

```
////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
////////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int tileIndex)
```

```
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;
    mysprites[num].colormode = COLOR_256;
    mysprites[num].trans = MODE_TRANSPARENT;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = COLOR_256 | MODE_TRANSPARENT | y;

    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
        }

    sprites[num].attribute1 = sprite_size | x;
}
```

Another new function that is used in this program is SetTrans. This function allows you to selectively toggle the transparency flag of any sprite at runtime.

```
//////////////////////////////////////////////////////////////
// Function: SetTransparency
// Changes the transparency of a sprite,
// MODE_NORMAL or MODE_TRANSPARENT
//////////////////////////////////////////////////////////////
```

```
void SetTrans(int num, int trans)
{
    mysprites[num].trans = trans;
    sprites[num].attribute0 = mysprites[num].colormode |
        mysprites[num].trans | mysprites[num].y;
}

/////////////////////////////////////////////////////////////
// Function: SetColorMode
// Changes the color mode of the sprite
// COLOR_16 or COLOR_256
/////////////////////////////////////////////////////////////
void SetColorMode(int num, int colormode)
{
    mysprites[num].colormode = colormode;
    sprites[num].attribute0 = mysprites[num].colormode |
        mysprites[num].trans | mysprites[num].y;
}

/////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(*ScanlineCounter < 160);
}
```

The main function follows. Most of this code should look familiar to you after going through the previous sample programs, but there is some new code here that is needed to support alpha blending. In addition to the code for bouncing the sprites around on the screen is a section that toggles transparency on and off every so often.

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

```
//////////////////////////////////////////////////////////////
int main()
{
    int n;
    int counter = 0;
    int change = 0;

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = ballPalette[n];

    //load ball sprite
    for(n = 0; n < ball_WIDTH * ball_HEIGHT / 2; n++)
        SpriteData3[n] = ballData[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image (512)
    for (n = 0; n < NUMBALLS; n++)
    {
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH, 512);
        while (mysprites[n].dirx == 0)
            mysprites[n].dirx = rand() % 6 - 3;
        while (mysprites[n].diry == 0)
            mysprites[n].diry = rand() % 6 - 3;
    }
```

```
//set transparency level
REG_BLDMOD = (1 << 4) | (1 << 10);
REG_COLEV = (8) + (8 << 8);

//main loop
while(1)
{
    //keep the screen civil
    WaitVBlank();

    //toggle transparency after an interval
    if (counter++ > 1000)
    {
        counter = 0;
        if (change)
        {
            change = 0;
            for (n = 0; n < NUMBALLS; n++)
                SetTrans(n, MODE_NORMAL);
        }
        else
        {
            change = 1;
            for (n = 0; n < NUMBALLS; n++)
                SetTrans(n, MODE_TRANSPARENT);
        }
    }

    for (n = 0; n < NUMBALLS; n++)
    {
        //update sprite x position
        mysprites[n].x += mysprites[n].dirx;
        if (mysprites[n].x > 239 - mysprites[n].size)
```

```
            {
                mysprites[n].x = 239 - mysprites[n].size;
                mysprites[n].dirx *= -1;
            }
            if (mysprites[n].x < 1)
            {
                mysprites[n].x = 1;
                mysprites[n].dirx *= -1;
            }

            //update sprite y position
            mysprites[n].y += mysprites[n].diry;
            if (mysprites[n].y > 159 - mysprites[n].size)
            {
                mysprites[n].y = 159 - mysprites[n].size;
                mysprites[n].diry *= -1;
            }
            if (mysprites[n].y < 1)
            {
                mysprites[n].y = 1;
                mysprites[n].diry *= -1;
            }

            //update the sprite properties
            MoveSprite(n);
        }
        //copy all sprites into object attribute memory
      UpdateSpriteMemory();
      }
}
```

Well, that's the end of TransSprite. Go ahead and run the program, and I'm sure you will agree it is fascinating to watch the sprites moving around with alpha blending enabled. There are so many things you can do with this—you are but limited by your imagination!

Rotation and Scaling

Another fascinating special effect that really makes sprites fun is the ability to rotate and scale them in real time. Is there really any need to draw prerotated sprites anymore when support for rotating a sprite is built into the GBA hardware? The process isn't perfect, because the GBA doesn't have a floating-point processor, so all rotation must be done with fixed-point math. But that can be solved easily enough with a precalculated array of sine and cosine values. This is a refinement over the SIN and COS arrays that you saw in the previous chapter, as there is no longer any need for a source file containing these radian values since they're just computed at the start of the program. This does cause a slight delay at the start of the program, but you could deal with that by displaying a splash screen and using the calculations as a sort of delay, so the player doesn't notice that an actual computational delay is taking place (and I'm talking about only a few short seconds). However, if your game sprites need to display shadows, or if you want more precision in the game's graphics, you will want to pre-rotate all sprite images. Some objects, however, where precision is not as important, such as with an asteroid or a missile, rotating in the game should work fine.

In order to use rotation and scaling, you must define a new struct that fills in the missing rotational elements of the OAM struct used previously. The new struct, RotData, points to the same address in OAM and might be thought of as a union struct. However, note the use of filler elements in the struct, followed by pa, pb, pc, and pd. These are new attributes that describe the sprite's behavior and are used for rotation and scaling.

```
typedef struct tagRotData

{

    u16 filler1;

    u16 pa;

    u16 filler2;

    u16 pb;    u16 filler3;

    u16 pc;    u16 filler4;

    u16 pd;

}RotData,*pRotData;
```

The declaration of a new pointer is needed to use this struct, and as you'll note, it points to the sprites struct array (defined earlier in the program).

```
pRotData rotData = (pRotData)sprites;
```

The only thing that needs explanation is the use of sine and cosine to actually rotate the sprites on the screen. As I mentioned in the previous chapter regarding background rotation, the GBA supports the rendering of a rotated sprite, but you must provide the trigonometry for the actual rotation. That is accomplished with fixed-point (integer) math, which is a type of virtual floating-point emulation that is extremely fast. Calculations must be done with radians, so the usual 360 degrees must be converted to radians.

```
//math values needed for rotation

#define PI 3.14159265

#define RADIAN(n) (((float)n) / (float)180 * PI)
```

Here are the two SIN and COS arrays used to hold the precomputed angle of rotation values:

```
//precomputed sine and cosine arrays

signed int SIN[360];

signed int COS[360];
```

And here is the loop that creates the SIN and COS arrays of precomputed values. This is a somewhat time-consuming process that ties up the CPU for a few seconds, so I suggest displaying a splash or title screen before running this code.

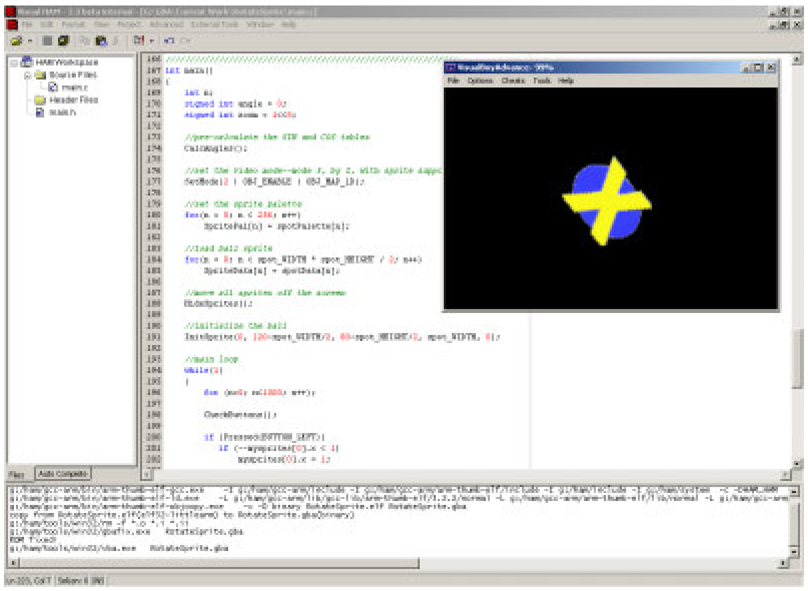
```
for(n = 0; n < 360; n++)

{

    SIN[n] = (signed int)(sin(RADIAN(n)) * 256);

    COS[n] = (signed int)(cos(RADIAN(n)) * 256);

}
```

The RotateSprite Program

Now that you have some of the basics down for rotating sprites, it's time to write a sample program to demonstrate how it all works. I realize that I have skimmed over the material and that you may be wondering how it all works. Without listing the actual code beforehand and then explaining it, I think it makes more sense to just type in the actual program and see how it works *firsthand*. Figure 7.9 shows the output from the RotateSprite program.

The RotateSprite program clears out a lot of the code from the previous program in order to help you understand exactly what is going on just with the rotation and scaling of the sprite. Therefore, there is no background image. What I have done differently with this program is provide a means to control the sprite using the GBA's buttons. The LEFT, RIGHT, UP, and DOWN buttons will move the sprite on the screen; the A and B buttons rotate the sprite; while the L and R buttons change the scale of the sprite.

Create a new project in Visual HAM and call it RotateSprite. You may also load the project off the CD-ROM, located in \Sources\Chapter07\RotateSprite. The RotateSprite.gba file is the binary that you may run directly in VisualBoyAdvance (or another GBA emulator, if you wish).



```
//////////////////////////////////////////////////////////////
int main()
{
    int n;
    signed int angle = 0;
    signed int zoom = 1<<8;

    //pre-calculate the SIN and COS tables
    CalcAngles();

    //set the video mode--mode 2, bg 2, with sprite support
    SetMode(2 | OBJ_ENABLE | OBJ_MAP_1D);

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = spotPalette[n];

    //load ball sprite
    for(n = 0; n < spot_WIDTH * spot_HEIGHT / 2; n++)
        SpriteData[n] = spotData[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the ball
    InitSprite(0, 120-spot_WIDTH/2, 80-spot_HEIGHT/2, spot_WIDTH, 0);

    //main loop
    while(1)
    {
        for (n=0; n<1000; n++);

        CheckButtons();

        if (Pressed(BUTTON_LEFT))
            if (--mysprites[0].x < 1)
                mysprites[0].x = 1;
```

Figure 7.9

The RotateSprite program demonstrates how to rotate and scale a sprite from player input.

The RotateSprite Header File

The header file takes care of all the defines, includes, and so on and is to be typed into a new file called main.h. If you need help adding a new file to the project, refer to one of the previous projects in this chapter for a summary.

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// RotateSprite Project
// main.h header file
//////////////////////////////////////////////////////////////


#ifndef _MAIN_H
#define _MAIN_H


typedef unsigned short u16;


#include <stdlib.h>
#include <math.h>
#include "spot.h"
#include "bg.raw.c"


//macro to change the video mode
```

```
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)

//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)

//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;

//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)

//transparency stuff
#define REG_BLDMOD     *(unsigned short*)0x4000050
#define REG_COLEV      *(unsigned short*)0x4000052

//misc sprite constants
#define OBJ_MAP_2D            0x0
#define OBJ_MAP_1D            0x40
#define OBJ_ENABLE            0x1000
#define BG2_ENABLE            0x400
```

```
//attribute0 stuff
#define ROTATION_FLAG          0x100
#define SIZE_DOUBLE            0x200
#define MODE_NORMAL            0x0
#define MODE_TRANSPARENT       0x400
#define MODE_WINDOWED          0x800
#define MOSAIC                 0x1000
#define COLOR_16               0x0000
#define COLOR_256              0x2000
#define SQUARE                 0x0
#define TALL                   0x4000
#define WIDE                   0x8000

//attribute1 stuff
#define ROTDATA(n)             ((n) << 9)
#define HORIZONTAL_FLIP        0x1000
#define VERTICAL_FLIP          0x2000
#define SIZE_8                 0x0
#define SIZE_16                0x4000
#define SIZE_32                0x8000
#define SIZE_64                0xC000

//Attribute2 stuff
#define PRIORITY(n)            ((n) << 10)
#define PALETTE(n)             ((n) << 12)

//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;

typedef struct tagRotData
{
    u16 filler1;
    u16 pa;
    u16 filler2;
    u16 pb;
    u16 filler3;
    u16 pc;
    u16 filler4;
    u16 pd;
}RotData,*pRotData;

//create an array of 128 sprites equal to OAM
Sprite sprites[128];
pRotData rotData = (pRotData)sprites;

typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;
    int colormode;
    int trans;
    signed int rotate;
    signed int scale;

}SpriteHandler;

SpriteHandler mysprites[128];

//define the buttons
```

```
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512

//create pointer to the button interface in memory
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;

//keep track of the status of each button
int buttons[10];

//math values needed for rotation
#define PI 3.14159265
#define RADIAN(n) (((float)n)/(float)180 * PI)

//precomputed sine and cosine arrays
signed int SIN[360];
signed int COS[360];

#endif
```

The RotateSprite Source File

Here is the code listing for the main.c file of RotateSprite.

```
//////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// RotateSprite Project
// main.c source code file
//////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "main.h"

//////////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

//////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
//////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}

/////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
/////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}
```

The InitSprite function has been changed again, this time providing support for rotation and scaling. I have highlighted key lines in bold text.

```
//////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
//////////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;
    mysprites[num].rotate = ROTATION_FLAG;
    mysprites[num].scale = 1 << 8;
    mysprites[num].angle = 0;
    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = y |
        COLOR_256 |
        ROTATION_FLAG;

    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }

    sprites[num].attribute1 = x |
        sprite_size |
        ROTDATA(tileIndex);
}

/////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(*ScanlineCounter < 160);
}
```

Here is the CalcAngles function, which generates the precomputed sine and cosine values for rotation:

```
/////////////////////////////////////////////////////////////
// Function: CalcAngles
// Pre-calculates the sine and cosine tables
/////////////////////////////////////////////////////////////
void CalcAngles(void)
{
    int n;
    for(n = 0; n < 360; n++)
    {
        SIN[n] = (signed int)(sin(RADIAN(n)) * 256);
        COS[n] = (signed int)(cos(RADIAN(n)) * 256);
    }
}
```

I didn't go over the RotateSprite function earlier, so now a short summary is called for. This function uses the precomputed SIN and COS arrays as if they were sin() and cos() functions, with the usual rotation algorithm. Since the SIN and COS arrays are filled with fixed-point integer values, with the decimal fixed between bits 8 and 9 (where the first 8 bits represent the whole number, and the second 8 bits the fractional number), this code runs quite fast compared to floating-point code. After the new values have been calculated, they are plugged into the rotData structure for that particular sprite.

```
/////////////////////////////////////////////////////////////
// Function: RotateSprite
// Rotates and scales a hardware sprite
/////////////////////////////////////////////////////////////
void RotateSprite(int rotDataIndex, int angle,
    signed int xscale, signed int yscale)
{
    signed int pa,pb,pc,pd;

    //use the pre-calculated fixed-point arrays
    pa = ((xscale) * COS[angle])>>8;    pb = ((yscale) * SIN[angle])>>8;
    pc = ((xscale) * -SIN[angle])>>8;
    pd = ((yscale) * COS[angle])>>8;

    //update the rotation array entry
    rotData[rotDataIndex].pa = pa;    rotData[rotDataIndex].pb = pb;
    rotData[rotDataIndex].pc = pc;
    rotData[rotDataIndex].pd = pd;
}

/////////////////////////////////////////////////////////////////
// Function: CheckButtons
// Polls the status of all the buttons
/////////////////////////////////////////////////////////////////
void CheckButtons()
{
    //store the status of the buttons in an array
    buttons[0] = !((*BUTTONS) & BUTTON_A);
    buttons = !((*BUTTONS) & BUTTON_B);
    buttons = !((*BUTTONS) & BUTTON_LEFT);
    buttons = !((*BUTTONS) & BUTTON_RIGHT);
    buttons = !((*BUTTONS) & BUTTON_UP);
    buttons[5] = !((*BUTTONS) & BUTTON_DOWN);
    buttons[6] = !((*BUTTONS) & BUTTON_START);
    buttons[7] = !((*BUTTONS) & BUTTON_SELECT);
    buttons[8] = !((*BUTTONS) & BUTTON_L);
    buttons[9] = !((*BUTTONS) & BUTTON_R);
}

/////////////////////////////////////////////////////////////////
// Function: Pressed
// Returns the status of a button
/////////////////////////////////////////////////////////////////
int Pressed(int button)
{
    switch(button)
    {
        case BUTTON_A: return buttons[0];
        case BUTTON_B: return buttons;
        case BUTTON_LEFT: return buttons;
```

```
            case BUTTON_RIGHT: return buttons;
            case BUTTON_UP: return buttons;
            case BUTTON_DOWN: return buttons[5];
            case BUTTON_START: return buttons[6];
            case BUTTON_SELECT: return buttons[7];
            case BUTTON_L: return buttons[8];
            case BUTTON_R: return buttons[9];
        }
        return 0;
    }
```

The main function of the RotateSprite program is shown next. This function handles setting up the screen, calling the CalcAngles function to generate the SIN and COS lookup tables, and loads the sprite into OAM. After initialization, the program goes into a loop to handle button input and move the sprites on the screen.

```
////////////////////////////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////////////////////////////
int main()
{
    int n;

    //pre-calculate the SIN and COS tables
    CalcAngles();

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(2 | OBJ_ENABLE | OBJ_MAP_1D);

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = spotPalette[n];

    //load ball sprite
    for(n = 0; n < spot_WIDTH * spot_HEIGHT / 2; n++)
        SpriteData[n] = spotData[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the sprite at the center of the screen
    InitSprite(0, 120-spot_WIDTH/2, 80-spot_HEIGHT/2, spot_WIDTH, 0);

    //main loop
    while(1)
    {
        //comment out when running on real hardware
        for (n=0; n<1000; n++);
        //grab the button status
        CheckButtons();

        //control sprite using buttons
        if (Pressed(BUTTON_LEFT))
           if (--mysprites[0].x < 1)
               mysprites[0].x = 1;

        if (Pressed(BUTTON_RIGHT))
            if (++mysprites[0].x > 239-spot_WIDTH)
                mysprites[0].x = 239-spot_WIDTH;

        if (Pressed(BUTTON_UP))
            if (--mysprites[0].y < 1)
                mysprites[0].y = 1;

        if (Pressed(BUTTON_DOWN))
            if (++mysprites[0].y > 159-spot_HEIGHT)
                mysprites[0].y = 159-spot_HEIGHT;

        //buttons A and B change the angle
```

```
        if (Pressed(BUTTON_A))
            if (--mysprites[0].angle < 0)
                mysprites[0].angle = 359;

        if (Pressed(BUTTON_B))
            if (++mysprites[0].angle > 359)
                mysprites[0].angle = 0;

        //buttons L and R change the scale
        if (Pressed(BUTTON_L))
            mysprites[0].scale--;

        if (Pressed(BUTTON_R))
            mysprites[0].scale++;

        //update sprite position
        MoveSprite(0);

        //rotate and scale the sprite
        RotateSprite(0, mysprites[0].angle,
            mysprites[0].scale, mysprites[0].scale);

        //wait for vertical refresh before updating sprites
        WaitVBlank();

        //copy all sprites into object attribute memory
        //this is only possible during vertical refresh
        UpdateSpriteMemory();
    }
}
```

Animated Sprites

The only other special effect (or feature) of note aside from scaling, rotation, and alpha blending, would have to be animation. Many games on the GBA use rather static sprites, but the graphically rich games always include animated sprites. From a programming perspective, animated sprites require a lot more memory than static sprites, because every frame of animation is another small bitmap image that must be kept in memory. The easiest way to animate a sprite is to copy a particular frame of an animation sequence into OAM so that it is rendered during the next screen refresh. From the perspective of OAM, there is just one sprite image, but your program copies a new version of the sprite bitmap into sprite display memory.

The AnimSprite Program

The AnimSprite program (shown in Figure 7.10) is very similar to the BounceSprite program, and even uses the same sphere image, only this version of the sphere includes many frames of animation so that it appears to be rotating in 3D.

The AnimSprite Header

Here is the header for the AnimSprite program, which should be typed into a file called main.h. If you haven't created the AnimSprite project yet, go ahead and create the project now, and add a new file, as you have done with other sample programs in this chapter.

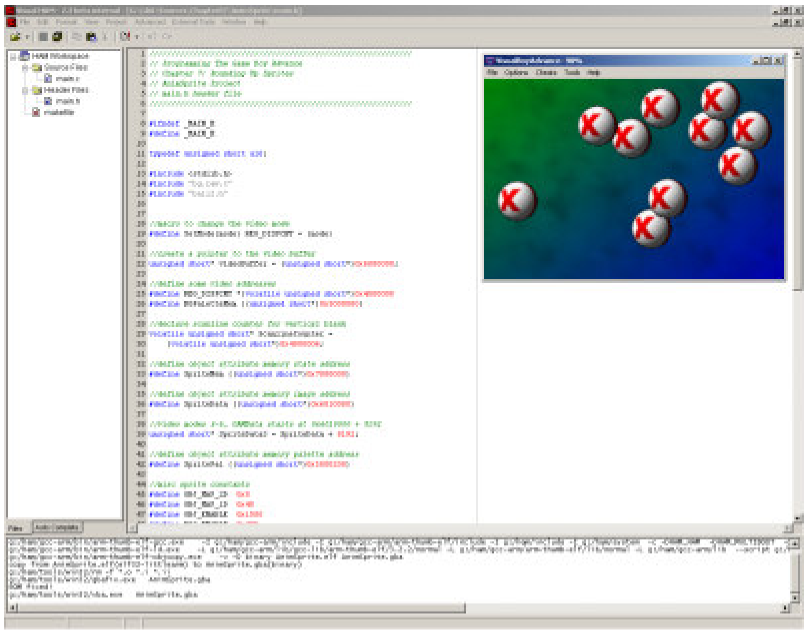


Figure 7.10

The AnimSprite program demonstrates how to draw animated sprites.

```
////////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// AnimSprite Project
// main.h header file
//////////////////////////////////////////////////////////////

#ifndef _MAIN_H
#define _MAIN_H

typedef unsigned short u16;

#include <stdlib.h>
#include "bg.raw.c"
#include "ball2.h"


//macro to change the video mode
#define SetMode(mode) REG_DISPCNT = (mode)

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//define some video addresses
#define REG_DISPCNT *(volatile unsigned short*)0x4000000
#define BGPaletteMem ((unsigned short*)0x5000000)

//declare scanline counter for vertical blank
volatile unsigned short* ScanlineCounter =
    (volatile unsigned short*)0x4000006;

//define object attribute memory state address
#define SpriteMem ((unsigned short*)0x7000000)

//define object attribute memory image address
#define SpriteData ((unsigned short*)0x6010000)
```

```
//video modes 3-5, OAMData starts at 0x6010000 + 8192
unsigned short* SpriteData3 = SpriteData + 8192;

//define object attribute memory palette address
#define SpritePal ((unsigned short*)0x5000200)

//misc sprite constants
#define OBJ_MAP_2D              0x0
#define OBJ_MAP_1D              0x40
#define OBJ_ENABLE              0x1000
#define BG2_ENABLE              0x400

//attribute0 stuff
#define ROTATION_FLAG           0x100
#define SIZE_DOUBLE             0x200
#define MODE_NORMAL             0x0
#define MODE_TRANSPARENT        0x400
#define MODE_WINDOWED           0x800
#define MOSAIC                  0x1000
#define COLOR_256               0x2000
#define SQUARE                  0x0
#define TALL                    0x4000
#define WIDE                    0x8000

//attribute1 stuff
#define SIZE_8                  0x0
#define SIZE_16                 0x4000
#define SIZE_32                 0x8000
#define SIZE_64                 0xC000

//an entry for object attribute memory (OAM)
typedef struct tagSprite
{
    unsigned short attribute0;
    unsigned short attribute1;
    unsigned short attribute2;
    unsigned short attribute3;
}Sprite,*pSprite;

//create an array of 128 sprites equal to OAM
Sprite sprites[128];

typedef struct tagSpriteHandler
{
    int alive;
    int x, y;
    int dirx, diry;
    int size;

}SpriteHandler;

SpriteHandler mysprites[128];

#endif
```

The AnimSprite Source Code

Okay, now for the last source code listing of the chapter, the code for the main AnimSprite program. The code is similar to the BounceSprite program, so it should be familiar to you. Assuming you have already created the AnimSprite project, replace the default code in the main.c file with the following code listing.

```
/////////////////////////////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 7: Rounding Up Sprites
// AnimSprite Project
// main.c source code file
/////////////////////////////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
```

```
MULTIBOOT

#include "main.h"

#define NUMBALLS 10

//////////////////////////////////////////////////////////////
// Function: HideSprites
// Moves all sprites off the screen
//////////////////////////////////////////////////////////////
void HideSprites()
{
    int n;
    for (n = 0; n < 128; n++)
    {
        sprites[n].attribute0 = 160;
        sprites[n].attribute1 = 240;
    }
}

//////////////////////////////////////////////////////////////
// Function: MoveSprite
// Changes sprite attributes for x,y positions
//////////////////////////////////////////////////////////////
void MoveSprite(int num)
{
    //clear the old x value
    sprites[num].attribute1 = sprites[num].attribute1 & 0xFE00;
    sprites[num].attribute1 = sprites[num].attribute1 | mysprites[num].x;

    //clear the old y value
    sprites[num].attribute0 = sprites[num].attribute0 & 0xFF00;
    sprites[num].attribute0 = sprites[num].attribute0 | mysprites[num].y;
}
```

```
//////////////////////////////////////////////////////////////
// Function: UpdateSpriteMemory
// Copies the sprite array into OAM memory
//////////////////////////////////////////////////////////////
void UpdateSpriteMemory(void)
{
    int n;
    unsigned short* temp;
    temp = (unsigned short*)sprites;

    for(n = 0; n < 128 * 4; n++)
        SpriteMem[n] = temp[n];
}

///////////////////////////////////////////////////////////////
// Function: InitSprite
// Initializes a sprite within the sprite handler array
///////////////////////////////////////////////////////////////
void InitSprite(int num, int x, int y, int size, int color, int tileIndex)
{
    unsigned int sprite_size = 0;

    mysprites[num].alive = 1;
    mysprites[num].size = size;
    mysprites[num].x = x;
    mysprites[num].y = y;

    //in modes 3-5, tiles start at 512, modes 0-2 start at 0
    sprites[num].attribute2 = tileIndex;

    //initialize
    sprites[num].attribute0 = color | y;
    switch (size)
    {
        case 8: sprite_size = SIZE_8; break;
        case 16: sprite_size = SIZE_16; break;
        case 32: sprite_size = SIZE_32; break;
        case 64: sprite_size = SIZE_64; break;
    }

      sprites[num].attribute1 = sprite_size | x;
}

/////////////////////////////////////////////////////////////
// Function: WaitVBlank
// Checks the scanline counter for the vertical blank period
/////////////////////////////////////////////////////////////
void WaitVBlank(void)
{
    while(!(*ScanlineCounter));
    while((*ScanlineCounter));
}

void UpdateBall(index)
{
    u16 n;

    //load ball sprite
    for(n = 0; n < 512; n++)
        SpriteData3[n] = ballData[(512*index)+n];
}

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

```
int main()
{
    int n;

    //set the video mode--mode 3, bg 2, with sprite support
    SetMode(3 | OBJ_ENABLE | OBJ_MAP_1D | BG2_ENABLE);

    //draw the background
    for(n=0; n < 38400; n++)
        videoBuffer[n] = bg_Bitmap[n];

    //set the sprite palette
    for(n = 0; n < 256; n++)
        SpritePal[n] = ballPalette[n];

    //move all sprites off the screen
    HideSprites();

    //initialize the balls--note all sprites use the same image
    for (n = 0; n < NUMBALLS; n++)
    {
        InitSprite(n, rand() % 230, rand() % 150, ball_WIDTH,
            COLOR_256, 512);

        while (mysprites[n].dirx == 0)
            mysprites[n].dirx = rand() % 6 - 3;
        while (mysprites[n].diry == 0)
            mysprites[n].diry = rand() % 6 - 3;
    }

    int ball_index=0;

    //main loop
    while(1)
```

```
{
    if(++ball_index > 31) ball_index=0;
    UpdateBall(ball_index);

    for (n = 0; n < NUMBALLS; n++)
    {
        //update sprite x position
        mysprites[n].x += mysprites[n].dirx;
        if (mysprites[n].x > 239 - mysprites[n].size)
        {
            mysprites[n].x = 239 - mysprites[n].size;
            mysprites[n].dirx *= -1;
        }
        if (mysprites[n].x < 1)
        {
            mysprites[n].x = 1;
            mysprites[n].dirx *= -1;
        }

        //update sprite y position
        mysprites[n].y += mysprites[n].diry;
        if (mysprites[n].y > 159 - mysprites[n].size)
        {
            mysprites[n].y = 159 - mysprites[n].size;
            mysprites[n].diry *= -1;
        }
        if (mysprites[n].y < 1)
        {
            mysprites[n].y = 1;
            mysprites[n].diry *= -1;
        }

        //update the sprite properties
        MoveSprite(n);
        }

        //keep the screen civil
        WaitVBlank();

        //copy all sprites into object attribute memory
      UpdateSpriteMemory();
      }
}
```

Well, that sums up sprite rotation and animation! The only thing you need to worry about regarding the different video modes and backgrounds is that some backgrounds are not capable of being rotated or scaled, so if you come to a dead end and your own rotation code doesn't seem to be working, you might want to check the capabilities of the video mode and background you are using as a first attempt to get the program working. Another thing to remember is that the tile index for the sprite data is different for bitmap-based video modes (3–5) than for the tile-based modes (0–2). The reason for this is that bitmapped backgrounds require more video memory, and that encroaches on the sprite memory, so you must copy your sprite images and data into a higher position in sprite memory, depending on the video mode. Refer to the sample programs in this chapter for details on how to program the different modes when working with sprites.

Summary

Sprites are, without exception, the most important aspect of programming games on the GBA. This chapter has provided not only an overview and sample code for using hardware sprites, including how to convert source artwork into source code format, but this chapter has also delved into special effects. You have learned how to display and move sprites around on the screen using tile-based and bitmap-based video modes, as well as how to rotate and scale sprites in real time. This chapter also provided an explanation of sprite translucency using a process called alpha blending, as well as an example program that shows how to draw animated sprites on the screen.

Challenges

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

Challenge 1: Two of the programs in this chapter featured no backgrounds, in order to make the programs easier to understand. Modify either the SimpleSprite or RotateSprite program, giving it a tile-based background.

Challenge 2: Test the BounceSprite2 program with various sprite sizes (8 x 8, 16 x 16, 32 x 32, and 64 x 64) to see how many sprites are available when using each of these sizes. Simply reduce NUMBALLS until all the balls are moving to find the value in each case.

Challenge 3: The TransSprite program looks pretty neat, don't you think? Well, you can do better, I'm sure! Modify the program so it uses two different types of sprites, and make each sprite animated. You can do this by simply loading two frames for each sprite and storing them consecutively in OAM. Simply determine how large each sprite is and index that far into OAM for each new sprite.

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 Appendix D.

1. How many hardware sprites does the GBA support as an upper limit, regardless of video mode?
   A. 64
   B. 128
   C. 256
   D. 384

2. What is the maximum sprite size supported by the GBA?
   A. 16 x 16
   B. 32 x 32
   C. 64 x 64
   D. 128 x 128

3. What video modes support sprite rotation and scaling?
   A. Mode 2
   B. Mode 3
   C. Mode 4
   D. Any mode

4. What is the name of the define used to enable transparency in a sprite?
   A. MODE_TRANSPARENT
   B. MODE_ALPHABLEND
   C. MODE_TRANSLUCENT
   D. MODE_WINDOWED

5. What trigonometric functions are used to calculate degrees of rotation?
   A. cos and arctan
   B. sin and tan
   C. cosine and tangent
   D. sin and cos

6. True/False: Does the ARM7 processor have built-in support for floating-point numbers?
   A. True
   B. False

7. What special effects in all does the GBA provide for hardware sprites?
   A. Rotation and scaling
   B. Blitting, rotation, scaling, and alpha blending
   C. Blitting and scaling
   D. Blitting, rotation, and transparency

8. Where is the object attribute memory (OAM) image address located, where the actual sprites are stored?
   A. 0x6000000
   B. 0x7000000
   C. 0x6010000
   D. 0x4000052

9. What two programs were used in this chapter to convert sprite images into C source listings?
   A. gfx2gba and pcx2sprite
   B. bmp2gba and gif2gba
   C. pcx2gba and jpg2sprite
   D. pdf2sprite and doc2pdf

10. Which sprite attribute is used to control special effects, such as rotation, scaling, and transparency?
   A. Attribute0
   B. Attribute1
   C. Attribute2
   D. Attribute3