

Chapter 22: Advanced Graphics and Animation

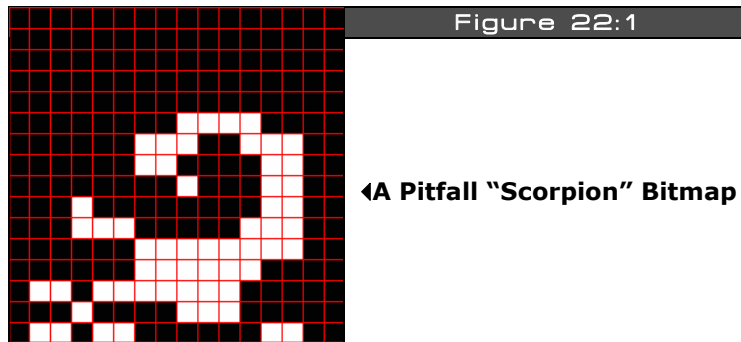
In this chapter we are going to discuss the problems associated with video game asset management along with animation and scrolling techniques. That is, given so much art, sound, music, and so forth, how does one simplify the process of getting the media into the HYDRA and doing something with it? This is the number one problem in writing games since without assets you can't make much of a game. Up to this point in the book we have manually entered in bitmaps, palettes, and data by hand, but this isn't going to cut it forever. Therefore to advance in our discussions we need tools, which are what we are going to develop in this chapter. Then with the tool chain in hand we are going to talk about scrolling techniques as well as general character animation for games and how that's done. So the primary topics in this chapter are:

- ▶ Using tool chains to build and import graphics
- ▶ Tile map scrolling
- ▶ Vector scrolling
- ▶ Parallax scrolling
- ▶ Framed animation

22.1 Using Tool Chains to Build and Import Graphics

Video game development is very demanding when it comes to asset generation and management. Thus far, we have more or less entered bitmap data in binary data statements that were hand drawn on graph paper or generated on the fly using a text editor. This obviously isn't going to cut it moving forward, so we need some simple tools to help us out or I am going to lose all my hair! If you recall in the last chapter on sound programming there was simply no way around building a tool to convert audio data into Spin code, it had to be done. And we are at that same turning point in the development of our graphics technology. So with that in mind, we need to develop a couple simple tools that allow us to draw imagery on the PC using a graphics painting program like *Photoshop* or *Paint Shop Pro* and then convert the bitmap data to Spin code that is somehow compatible with our current engines and technology. Secondly, we want to take advantage of the features of the tile engine, such as scrolling. To accomplish this we need to use an external tool to generate game field maps and then export them out and once again convert them to Spin-compliant code somehow. Alas, lots to do, very complicated, and details count, so let's get started!

22.2 Bitmap to Spin Conversion



The HEL graphics engine introduced a couple chapters ago is a tile-based engine with 4 colors per tile, where tiles are 16×16 pixels each represented by 2 bits per color, which index into the 4-BYTE color palette assigned to that tile. Thus, each tile can have 4 individual colors. The color palettes are in the form of a LONG each where BYTE 0 is Color 0, BYTE 1 is Color 1, and so forth. As an example, take a look at Figure 22:1, it depicts a “scorpion” bitmap from the classic 80’s game Pitfall (or at least it’s supposed to!) which I drew in Paint Shop Pro/Photoshop. Now, if I wanted to add this bitmap to a game or demo then it would have to be converted to our graphics tile bitmap format as shown below:

```

' a pitfall scorpion tile
scorpion_bitmap LONG %%0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_0_1_1_1_1_0_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_1_1_1_0_0_0_1_1_1_0_0_0_0_0_0_0
LONG %%0_0_1_1_0_0_0_1_0_0_0_0_0_0_0_0_0_0
LONG %%0_0_1_1_0_0_0_0_0_0_0_0_0_0_1_0_0_0
LONG %%0_0_1_1_1_0_0_0_0_0_0_1_1_1_0_0_0_0
LONG %%0_0_1_1_1_1_1_1_1_1_0_0_0_0_0_0_0_0
LONG %%0_0_0_0_1_1_1_1_1_1_0_0_0_0_0_0_0_0
LONG %%0_0_0_0_1_1_1_1_1_1_1_0_1_1_0_0_0_0
LONG %%0_0_0_0_1_1_1_0_0_0_0_0_1_0_0_0_0_0
LONG %%0_0_1_1_0_0_0_0_0_0_0_1_1_0_1_1_0_0

```


with templated graphics. Let's discuss the process of templating graphics assets using the *Pitfall* art as an example to work with.



Figure 22:2

◀A Screen Shot of Pitfall

Figure 22:2 shows a screen shot of Pitfall running on my Atari 2600. Based on this screen shot, and playing the game for many years, I was able to draw all the graphics in the game (roughly) and then place them into a template. Figure 22:3 shows the templated graphics.

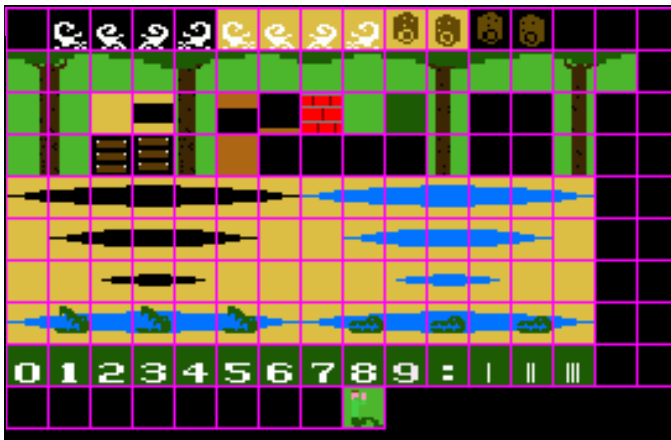



Figure 22:3

◀The Pitfall Templated Graphics

If course, what you see in Figure 22:3 is only a subset of the graphics from the game; I would be here for ever trying to see every little sprite in the game and then recreating it. Of course, you could always download the ROM of the game (if you own it) and pull the

graphics from it as long as it's only for experimenting like we are. In any case, the actual bitmap shown in Figure 22:3 is named **PITFALL_TILES_WORK_03.BMP** and is located in the **\SOURCE** directory as well as in the **\MEDIA\GRAPHICS** directory on the CD. You can open it up and view it in your favorite paint program. The first thing you will notice about the art is that there is a 1-pixel border around each tile, this border is exactly 1-pixel wide and common practice when tiling graphics. However, some programs do not like the 1-pixel border and you must remove it. I suggest that as a convention you draw all your graphics on a "work" bitmap where they are just all over the place since its your "art" workspace. Next, you cut up your tiles into 16×16 blocks (or whatever the size is your engine uses) and tile them into another file with a 1-pixel border. Finally, you create yet another file where the border is removed, and slide all the bitmaps together. Figure 22:4 shows the same bitmaps from Figure 22:3, but with the border removed (and a few extra tiles).



WARNING

All bitmap files must have a width and height that are both an even multiple of 8 for all tools discussed to work correctly. Thus, when you make your templated graphics resize the canvas around your bitmap data to make sure this rule is followed.



I personally prefer to work with a 1-pixel border to separate my tiles, but some third-party tools you might want to use only work with bitmaps that have no border (Mappy is one of them and we need to use it). As another example of templated art I have taken some of the royalty free tiles from **Ari Feldman's "Sprite Lib"** and tiled them as well. Figure 22:5(left) shows the tiled graphics with border while Figure 22:5(right) shows without border. The point is that you need to create your art, tile the art, and put it in a couple formats to make it easy for tools to read. The actual filenames of the two Sprite Lib based tile sets are

SL_BLOCKS_02.BMP (with border) and **SL_BLOCKS_03.BMP** (without border); both files are on the CD in the **\SOURCE** directory as well as in **\MEDIA\GRAPHICS\SPRITELIB** located on the CD.



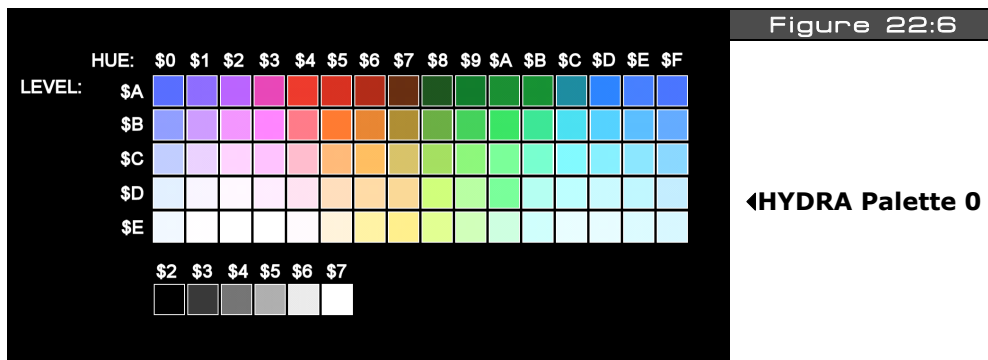
Before moving on to the tool itself and its use, there are a couple more details we need to cover. First, we haven't mentioned anything about color! We know that the Propeller only generates so many colors, so how can a 24-bit Truecolor image be represented by the HYDRA? Well, it can't, so we have to approximate colors and match or map them to the HYDRA palette. This means that bitmaps converted to the HYDRA might not look as good or vibrant as on the PC, but this is ok and standard for game graphics. To defend against graphics degradation typically what an artist does is work in 24-bit color and now and then reduce the color space to the target system's graphics palette. Every paint tool has the ability to reduce color space, so you can work with a nice 24-bit image then load the "HYDRA color palette" and see how it looks. If it looks terrible, for example, you are losing a lot of browns, then you know you need to lay off the browns and so forth. The question is, "What is the HYDRA/Propeller's palette?" It depends on the TV you hook the system up to, thus what had to be done is 10-20 TVs with "general" settings had to be connected to the HYDRA and then a complete palette rendered on the screen, then this was digitized and painstakingly ordered, labeled and numbered. The results are two palettes which I call "Palette 0" (Figure 22:6) and "Palette 1" (Figure 22:7) both of which contain 86 colors (including black). Palette 0 looks best on most TVs without adjusting the tint, but Palette 1 looks better on older "composite computer monitors" so you can take your pick.

Since this book is black and white you won't be able to see the colors from the palette, but you can look at the original bitmaps in files **HEL_GRAPHICS_PALETTE_0.BMP** and **HEL_GRAPHICS_PALETTE_1.BMP** which are located on the CD in the directory **\MEDIA\GRAPHICS**. Additionally, there are "color only" versions of these bitmaps with no

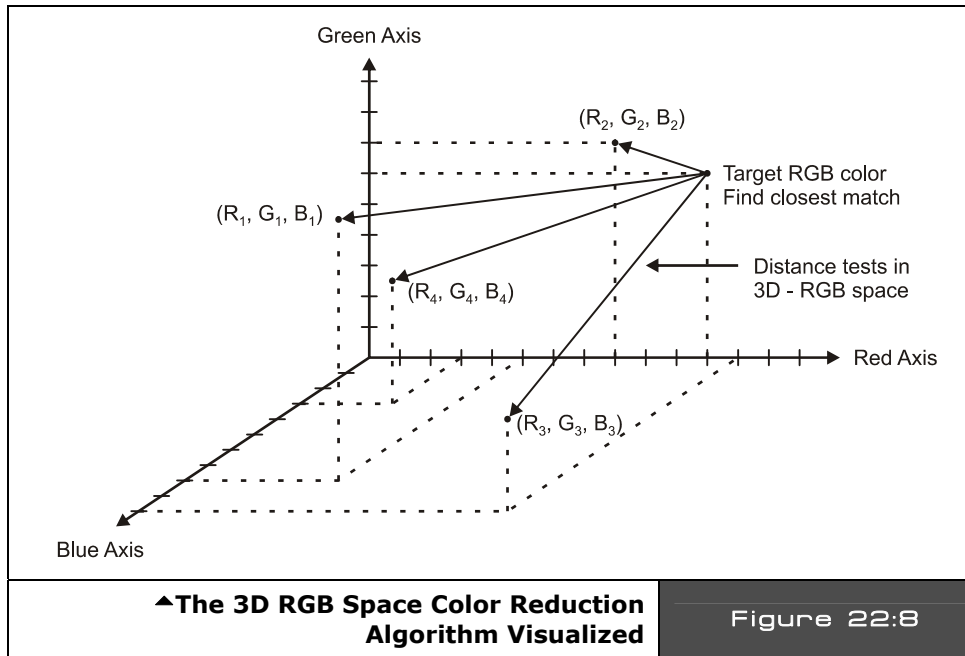
text, annotation, or other colors, so you can use them to color reduce your art with, that is use the “generate palette from bitmap function of your paint program.” Please use the following bitmaps to generate a palette to test match your art within your painting tool:

HEL_GRAPHICS_PALETTE_0b.BMP
HEL_GRAPHICS_PALETTE_1b.BMP

For example, you might draw some underwater art in 24-bit color in Photoshop and you want to see how well it will map to the HYDRA palette. So load in either palette “color only” image **HEL_GRAPHICS_PALETTE_0b.BMP** or **HEL_GRAPHICS_PALETTE_1b.BMP** then generate a palette from it and assign it to your art, instantly you will see the art change and the colors reduce and you can focus on problem areas. When you are satisfied with your art then you can leave it in 24-bit color mode or 256 color mode and save your artwork in templated format. You will at least need the borderless version for the other tools in our tool chain, but I suggest making both a 1-pixel border version as well as a borderless version.



22.2.1 3D RGB Space Color Reduction Algorithm



Before moving on to the actual invocation of the tool and its parameters and use, let's take a brief aside to discuss how color matching is performed. As mentioned, the HYDRA only has 86 colors in its palette, so whatever art you want to map to the HYDRA must be mapped to these 86 colors. Moreover, for each tile there can only be 4 colors, thus the tool has to generate a histogram to select the 4 most predominant colors in each tile to build the palette for the tile. Considering all that, the setup is as follows: there is a table which contains all 86 colors the HYDRA can generate in 24-bit RGB format (8 bits for each color). Then the bitmap to be converted into Spin code first needs its colors analyzed and mapped, here are the steps to the algorithm that does it:

Step 1: For all pixels in the source bitmap create a histogram table of each individual color and how many times it occurs.

Step 2: Sort the color histogram.

Step 3: Select the top 4 occurring colors, If BLACK occurs at all in image, it's so important it must be part of the final palette, remove one of the top 4 colors and insert black.

Step 4: The “target” palette for the tile bitmap in question now contains 4 RGB colors, next step is to match each one of these RGB colors to its “best match” in the HYDRA color palette.

The color matching process then gets complex. The problem simply put is given a 3-tuple number that represents an RGB color, compare it to 86 other 3-tuples and find the 3-tuple that is the closest match. In other words, we can assume that the RGB color we are looking for is a 3D point in RGB space, then each of the 86 colors that make up the HYDRA palette are points in this space. All we need to do is find one of the 86 colors that is “closest” to the target color and that’s the color we use to match. Figure 22:8 depicts this graphically (only a few colors are shown with generic numbers assigned). So the idea is you simply run an iterative algorithm that computes the distance from your source desired color to all the HYDRA colors and try and find the best match which is simply the “closest” color in 3D RGB space. For example, here’s a snippet of code from the **BMP2SPIN.EXE** tool that illustrates the matching algorithm:

```
// for each color in HYDRA color map test if its closer to target color in 3D
// colorspace
for (index2 = 0; index2 < num hydra colors; index2++)
{
    // extract test r,g,b we are testing against for match?
    r test = hydra color map[ index2 ].peRed;
    g test = hydra color map[ index2 ].peGreen;
    b test = hydra color map[ index2 ].peBlue;

    // compare test rgb to target rgb and if its closer update best match
    int rgb dist = (r target - r test)*(r target - r test) +
                  (g target - g test)*(g target - g test) +
                  (b target - b test)*(b target - b test);

    if (rgb dist < col dist)
    {
        // update distance and index
        col dist      = rgb dist;
        best col match = index2;
    } // end if
} // end for index2
```

You might be asking, “How does the conversion work for 8-bit palettized bitmaps?” The tool works in almost the same way, but instead of each pixel having its own RGB value, each pixel is an index into the color lookup table which is in RGB format. There is one more step of indirection to look up a pixel’s color then the matching process is the same. Hence, you are free to use either palettized 8-bit graphics or full Truecolor bitmaps.

22.2.2 Using the BMP2SPIN.EXE Tool

Now that you have seen most of the conceptual aspects as well as the practical ones of building a tool, let’s go ahead and take a look at the tool itself. The executable is once again called **BMP2SPIN.EXE**, it has very little error handling, so if you try to break it, you won’t have to try hard, so make sure to give it valid arguments. The tool takes as input a Windows

BMP file in 8-bit palettized or 24-bit RGB format. The bitmap must not be compressed. The usage of the tool is outlined below:

Usage:

BMP2SPIN.exe inputfilename [flags] > outputfilename

Where:

flags = -B -TW[1...255] -TH[1..255] -W[1..256] -H[1..256] -C[1..256] -XYx,y -M -FX -FY -V -I -P[0|1] -?

- B = Enable 1 pixel border for templated bitmaps, default no border
- TWxx = Width of tile set on x-axis, default = 1
- THxx = Height of tile set on y-axis, default = 1
- Wxx = Width of tile, default = 16 pixels
- Hxx = Height of tile, default = 16 pixels
- Cxx = Total count of tiles to be converted, if omitted assumed to be tw*th
- XYxx,yy = Coordinate of single tile to pull from bitmap, upper left (0,0) overrides -Cxx
- M = Enables mask write as well after each tile
- FX = Flips the output bitmap on the X axis (needed for hel engine 4.0)
- FY = Flips the output bitmap on the Y axis
- Px = Selects the color matching palette 0 or 1, 0 is default, 1 skews color hue 15 degrees approximately
- I = Enables interactive mode
- V = Verbose flag for debugging purposes
- ? = Prints help

22.2.2.1 Extracting a Single Bitmap

The flags are self-explanatory, so let's focus on using the tool in two different ways that are the most common. The first way you might want to use the tool is to extract a single bitmap from a larger bitmap full of templated graphics. The tool is mostly set up with defaults, so you don't have to give it many flags, but at very minimum you need to tell the tool the template width and height of your tiles. For example, in the Pitfall art there are 16×10 tiles in the template, these values get passed as "-TW16" and "-TH10" respectively. Also, we need to mirror the output bitmaps on the X axis, so we need the "-FX" flag. Next, to extract a single bitmap tile, we need to tell the tool the tile coordinates with the "-XYxx,yy" flag which doesn't tolerate spaces, so you must type everything right next to each other. Lastly, we need to decide if we want to extract from a tile template with a 1-pixel border or not, let's assume there is a border, so we need the "-B" flag. Given all that, let's "rip" tile x=1, y=0 from the Pitfall tile set which is the scorpion if you refer to the tile bitmap artwork. The **\SOURCE**

As you can see the tool is fairly thorough. It begins by outputting some header information to let you know what is what as well as some “getter” code if you want to import the file as a whole into your programs. The two PUB functions *tile_bitmaps* and *tile_palette_map* always return the starting address of the data. The header is followed by the bitmap(s) data and at the end is the palette. In this case, there will be only one bitmap and one palette. If you want to do a little detective work then follow along: notice that if you blur your eyes, you can indeed see the scorpion. It’s made up of two colors: 0 and 1. If you look at the single palette entry, you will notice that the rightmost or low BYTE is \$02 which is BLACK and represents color index 0. Additionally, BYTE 1 which represents color 1 is \$07 which if you look at the HYDRA palette is indeed WHITE. Thus, the bitmap was scanned and converted correctly and a billion calculations were done to give us this! Of course, we didn’t indicate which palette to use, so the default Palette 0 was used. Lastly, to get the actual file output into a .txt or .Spin file simply redirect the console output like this:

```
BMP2SPIN pitfall_tiles_work_03.bmp -B -TW16 -TH10 -FX -XY1,0 > TEST.SPIN
```

...then all the text would go to **TEST.SPIN** which you could edit, or import directly into your main game program like this:

```
OBJ
bitmap_data: "test.spin" ' import external file and use getter functions.
```

Additionally, if you wanted to use this tile as a sprite, you would need its “mask.” To get mask output as well after each tile, simply add the “-M” flag like this:

```
BMP2SPIN pitfall_tiles_work_03.bmp -B -TW16 -TH10 -FX -XY1,0 -M > TEST.SPIN
```



NOTE

Flags can be in any order, but the source bitmap filename must *always* be the *first* parameter.

22.2.2.2 Extracting Multiple Bitmaps

So far so good, now let’s try to use the tool to extract multiple bitmaps. We are going to use almost the same flags, but instead of specifying an x,y tile to extract, we are going to specify a “count” of tiles to extract. The extraction process will proceed from left to right, top to bottom in the tile template and extract however many tiles we indicate. Also, for fun, let’s use no border, and let’s request Palette 1 for the match. Here’s the command line for the multiple bitmap extraction:

```
BMP2SPIN pitfall_tiles_work_04.bmp -TW15 -TH11 -FX -C165 -P1
```