7

HIGH-LEVEL FEATURES

This chapter covers the high-level graphics features of the DMG—the background, the window, and sprites—and explains how to leverage them. By the end of the chapter, you'll have all the knowledge you need to display custom graphics on the LCD.

You'll begin by learning about the parameters and functionalities of the background, including scrolling and wrapping. Building on that knowledge, you'll learn about frame pacing, a fundamental technique for producing smooth animations.

Then, we'll take a closer look at the window, comparing its parameters and functionalities to those of the background. Finally, you'll learn how to use sprites and gain an understanding of their behavior and limitations.

Background

The *background* is a large scrollable tilemap that forms the visual backdrop for a DMG game. You configure the background by specifying the palette, selecting the tilemap, and setting a few other parameters. There are also a couple of useful built-in functionalities to be aware of.

Palette

The first element you need to set up is the background palette, as demonstrated in the *bg* sample. The relevant code from *sample.rgbasm* is shown in Listing 7-1.

```
    ld a, %11100100
    ld [rBGP], a
```

Listing 7-1: Setting the background palette

The background has only one palette, which is set through the `rBGP` I/O register. In this code snippet, we place white at color index 0 (bits 0 and 1), light gray at index 1 (bits 2 and 3), dark gray at index 2 (bits 4 and 5), and black at index 3 (bits 6 and 7). This is just one possible palette setup; you can put the colors in any order you like (for example, `%00011011`), or even reuse the same color multiple times (for example, `%00001111`). For a refresher on the available colors, refer back to Table 5-1.

LCD Parameters

Now that we've set up the palette, we need to set the LCD control register, `rLCDC`, which will enable the background and set both the addressing mode (`$8000` or `$8800`) and the tilemap address (`$9800` or `$9C00`).

In Chapter 6, you learned that the `rLCDC` register is a collection of flags that control the behavior of the DMG's graphics system. The *hardware.rgbinc* file defines the possible flag values, as shown in Listing 7-2.

```
DEF LCDCF_OFF     EQU %00000000 ; LCD Control Operation
DEF LCDCF_ON      EQU %10000000 ; LCD Control Operation
DEF LCDCF_WIN9800 EQU %00000000 ; Window Tile Map Display Select
DEF LCDCF_WIN9C00 EQU %01000000 ; Window Tile Map Display Select
DEF LCDCF_WINOFF  EQU %00000000 ; Window Display
DEF LCDCF_WINON   EQU %00100000 ; Window Display
DEF LCDCF_BG8800  EQU %00000000 ; BG & Window Tile Data Select
DEF LCDCF_BG8000  EQU %00010000 ; BG & Window Tile Data Select
DEF LCDCF_BG9800  EQU %00000000 ; BG Tile Map Display Select
DEF LCDCF_BG9C00  EQU %00001000 ; BG Tile Map Display Select
DEF LCDCF_OBJ8    EQU %00000000 ; OBJ Construction
DEF LCDCF_OBJ16   EQU %00000100 ; OBJ Construction
DEF LCDCF_OBJOFF  EQU %00000000 ; OBJ Display
DEF LCDCF_OBJON   EQU %00000010 ; OBJ Display
DEF LCDCF_BGOFF   EQU %00000000 ; BG Display
DEF LCDCF_BGON    EQU %00000001 ; BG Display
```

Listing 7-2: The `rLCDC` flags

Three flags, or bits, are relevant to the background. Table 7-1 summarizes the possible values of these flags and what they mean.

Table 7-1: Background Flag Values

Bit	Value	Meaning
0	LCDCF_BGOFF	The background is disabled.
	LCDCF_BGON	The background is enabled.
3	LCDCF_BG9800	The tilemap is located at $9800.
	LCDCF_BG9C00	The tilemap is located at $9C00.
4	LCDCF_BG8000	The $8000 addressing mode is used.
	LCDCF_BG8800	The $8800 addressing mode is used.

Listing 7-3 shows how the LCD control register is set in the *bg* sample.

```
    ld a, LCDCF_ON | LCDCF_BG8800 | LCDCF_BG9800 | LCDCF_BGON
    ld [rLCDC], a
```

Listing 7-3: Background setup

We need to enable the background before we can use it, so we raise bit 0 using `LCDCF_BGON`. The tilemap is at address `$9800` in VRAM and the tileset was organized for the `$8800` addressing mode (see Figure 6-1), so the flags for bits 3 and 4 are set accordingly. The `LCDF_ON` flag ensures that the LCD is on.

Listing 7-4 shows the fully updated `InitSample` function from the *sample.rgbasm* file.

```
InitSample:
    ; init the palette
    ld a, %11100100
    ld [rBGP], a

    LoadGraphicsDataIntoVRAM

    ; set the graphics parameters and turn LCD back on
    ld a, LCDCF_ON | LCDCF_BG8800 | LCDCF_BG9800 | LCDCF_BGON
    ld [rLCDC], a

    ret
```

Listing 7-4: The `InitSample` function for the background (bg) sample

First, the background palette is set. Then, the graphics data, which consists of tiles and a tilemap, is loaded into VRAM. Finally, the remaining background parameters are set through the LCD control register so that the background displays properly on the LCD.

Figure 7-1 shows what you should see when you compile the code for this sample and run it in BGB.

Figure 7-1: The background sample

Notice that we see only a part of the tilemap because the tilemap is 256×256 pixels but the LCD can only display 160×144 pixels. We'll use one of the background's key functionalities in the next section to reveal the rest of the tilemap.

Scrolling and Wrapping

The background has two key built-in functionalities: scrolling (controlled by the programmer) and wrapping (automatic).

You can control the scrolling of the background using the `rSCX` and `rSCY` registers. The `rSCX` register controls the horizontal scrolling, and `rSCY` controls vertical scrolling.

The `rSCX` and `rSCY` registers store the coordinates of the screen relative to the top-left corner of the tilemap. In this book, I'll use the term *abscissa* and *ordinate* to refer to the x-coordinate and y-coordinate, respectively. Figure 7-2 shows how background scrolling works.

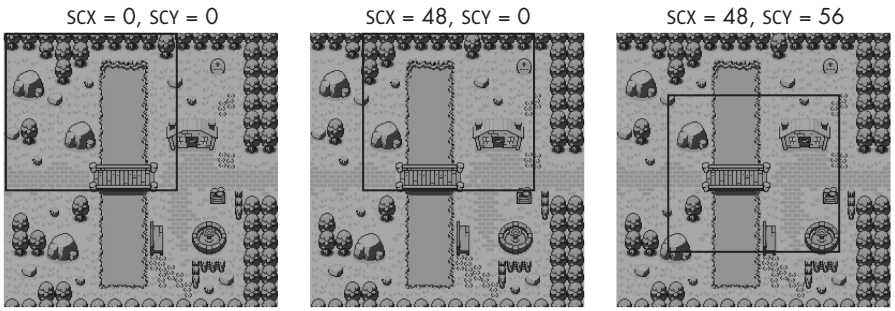

Figure 7-2: Three background scrolling positions and their corresponding coordinates

The square area represents the screen. The default position on the left is what we got previously, in Figure 7-1. The image in the center shows the

background scrolled horizontally, and the right-hand image shows the background scrolled in both directions.

When you scroll far enough in one direction, the boundary of the background's tilemap ends up displayed on the screen. That's when the wrapping kicks in. Figure 7-3 illustrates how wrapping works.

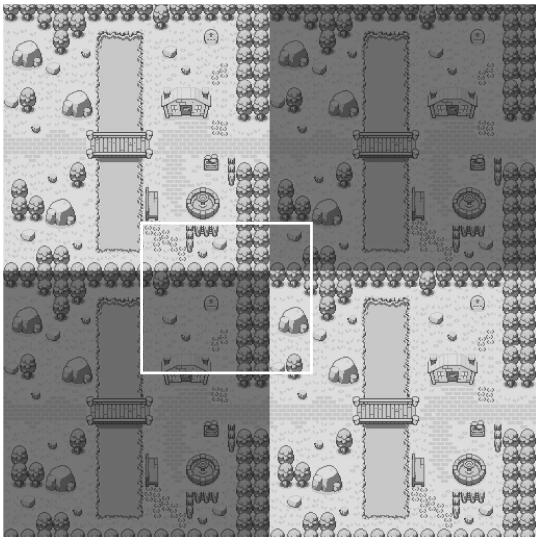

Figure 7-3: Background wrapping

The background's tilemap repeats in every direction. I've highlighted the repeat tilemaps to make the wrapping effect stand out, but in real life, the colors would not change.

To experiment with wrapping in BGB, you can update the `UpdateSample` function. In the sample, the scrolling values are zero by default, as shown in Listing 7-5.

```
UpdateSample:
    ld a, 0
    ld [rSCX], a
    ld a, 0
    ld [rSCY], a
    ret
```

Listing 7-5: The `UpdateSample` function

Try using values such as 176 and 184 for `rSCX` and `rSCY`, respectively. There's nothing special about those two values, but they allow you to see both scrolling and wrapping in effect, as in Figure 7-3.

While the background has only a few configurable parameters and functionalities, it's an essential part of the DMG graphics system, and most, if not all, released games make use of it.

Frame Pacing

In this section, you'll learn about an issue in the update function that you must address before configuring the other high-level graphics features. I'll illustrate the issue using the *scrolling_anim_broken* sample, then show you how to fix it using a technique called *frame pacing*, which synchronizes the update function's call with the DMG's frame mechanism to produce smooth animations.

The Synchronization Issue

For the purpose of this example, we'll add a simple animation to the background (shown in Listing 7-6).

```
UpdateSample:
    ld a, [rSCX]
    inc a
    ld [rSCX], a

    ret
```

Listing 7-6: A simple background animation

To create this basic animation, the code increments the value of `rSCX` each time the `UpdateSample` function is called. From this, you would expect that the background will scroll smoothly to the left, one pixel at a time, as each frame is drawn. But the actual result, shown in Figure 7-4, is quite different.

Figure 7-4: Broken background animation

The problem is that the animation is not synchronized with the DMG frame mechanism, described in Chapter 5. Currently, the `UpdateSample` function is called from the main loop as many times as the DMG CPU speed will allow. That's well over a thousand times per frame, but it should be

called only once per frame. The result is that each line of the screen scrolls independently of the other lines, causing the visual mess in Figure 7-4.

The Pacing Solution

To fix the pacing issue, we'll wait for the next vblank at the start of the `UpdateSample` function. This ensures that `UpdateSample` executes only per frame because, by definition, there is only one vblank per frame.

In Listing 6-5, we waited for the vblank when we disabled the LCD in the main function. To accomplish this, we waited for the current line number register, `rLY`, to equal 144, which meant we had entered the vblank period. That technique is good enough for initialization, but a better way to wait for the vblank in the main loop is to use `halt` and the vblank interrupt. The `halt`/interrupt method is superior because it has the potential to prolong the device's battery life. When using `halt`, the DMG enters low power mode. On the other hand, the DMG will be fully working during the loop of the `rLY` method.

Before implementing the `halt`/interrupt method, we need to define what an interrupt is. For the purposes of this book, an *interrupt* is a function that gets called when a particular event occurs. When the event arises, the CPU will stop, or interrupt, any work it's currently doing, execute a function associated with the event (the interrupt function), and then go back to its previous work. In the sample, the event is the start of the vblank, and the associated interrupt function address for that event is `$0040`. Four other event types trigger interrupts, and each has its own fixed function address; I'll cover these later in the book.

Now that we know what an interrupt is, let's use the vblank interrupt in the sample.

Enabling the vblank Interrupt

The first step is to enable the vblank interrupt. We do this during the initialization of the *scrolling_anim_fixed* sample.

Listing 7-7 shows the relevant code from `InitSample`.

```
ld a, IEF_VBLANK
ld [rIE], a
ei
```

Listing 7-7: Enabling the vblank interrupt

We perform two actions here. First, the code sets the `rIE` (interrupt enable) register. This register contains flags to tell the DMG which interrupts are enabled. For now, only the vblank interrupt is necessary, so we raise a single flag, `IEF_VBLANK`. Second, we enable the interrupt functionality. As you saw in Chapter 4, we disabled the interrupts in the template by using `di` (disable interrupts) in the entry point so they wouldn't get in the way during the sample initialization. Here, we use `ei` (enable interrupts) to re-enable the interrupts.

Defining the Interrupt Function

The second step of using the vblank interrupt is to define the interrupt function, as shown in Listing 7-8.

```
section "vblank_interrupt", rom0[$0040]
    reti
```

Listing 7-8: The vblank interrupt function

We define a section at `$0040`, because this is the address of the function called when the vblank event occurs. The function contains a single instruction, `reti`, which returns from the function (like a regular `ret`) with the additional effect of enabling interrupts.

We need to use `reti` and not `ret` here because the DMG disables the interrupts (like `di` would) before executing any interrupt function. This prevents interrupting the interrupt function, which could lead to a bunch of issues, like CPU hangs. Because we want to detect subsequent vblank events, we need to use `reti` to enable interrupts again after the function returns.

Waiting for the vblank Interrupt

The last step is to wait for the interrupt to occur. This is done with the previously mentioned `halt` instruction, which puts the DMG into low power mode. In this mode, no instructions are executed, and the DMG wakes up only when an interrupt triggers. In the sample, the only enabled interrupt is the vblank interrupt. This means that each time the `halt` instruction completes, a new vblank period has just started.

We can add `halt` to the `UpdateSample` function, as shown in Listing 7-9.

```
UpdateSample:
    halt

    ld a, [rSCX]
    inc a
    ld [rSCX], a

    ret
```

Listing 7-9: Using `halt` to wait for the vblank

Now, each time the main loop calls `UpdateSample`, the DMG waits for the next vblank. The result is that `UpdateSample` is called only once per frame.

If you compile and run the sample again in BGB, you should now see that the scrolling animation is fluid because we have properly synchronized the update function with the DMG display mechanism.

Frame Pacing Mechanics

We now have the proper frame pacing in place, allowing for smooth animations. Let's quickly recap the three steps we implemented to fix the problem and the timing of each step.

The first step is to wait for the next vblank using the `halt` instruction. This step's duration is whatever time was left of the 16.74 ms of the previous frame.

The next step is to perform any operation that requires VRAM access or affects graphics directly before the end of the vblank. This step can take up to 1.08 ms, which is the duration of the vblank. As discussed in "Copying Assets to VRAM" on page 87, VRAM access is restricted during the LCD refresh, so it's critical to carry out any VRAM updates during the vblank. For example, the PPU will ignore a tile update, which is a write to VRAM, if it's executed outside the vblank. This will likely cause graphical artifacts, as the intended graphics data will not be in VRAM. Likewise, while operations such as scrolling are allowed at any time, they might cause visual glitches if not executed during the vblank (if the `rSCX` register is updated outside the vblank, the screen will end up split in two halves that use different scrolling values).

The final step of the update function is to perform any remaining program operations, such as checking input or running the sample logic. This step can take 15.66 ms (the duration of the LCD refresh) plus whatever time remains from the vblank's 1.08 ms.

Window

In this section we'll look at the *window*, which can display part of a tilemap in a rectangular area of the screen superimposed on the main background. You'll learn about the window's similarities and differences compared with the background as well as its main pitfalls.

Comparing the Window and the Background

The window and background share a few parameters. The first one is the palette. We set the palette for the background in the `InitSample` function, as shown in Listing 7-10.

```
ld a, %11100100
ld [rBGP], a
```

Listing 7-10: Setting the background and window palette

This sets the palette for the window too.

The second shared parameter is the tile addressing mode. The addressing mode is set in the LCD control register, `rLCDC`, using either the `LCDCF_BG8000` flag or the `LCDCF_BG8800` flag. These flag names are a bit misleading, because it looks like they apply only to the background, but the addressing mode setting actually applies to both the background and the window.

Earlier, we set the background addressing mode to `LCDCF_BG8800`, as shown in Listing 7-11.

```
ld a, LCDCF_ON | LCDCF_BG8800 | LCDCF_BG9800 | LCDCF_BGON
ld [rLCDC], a
```

Listing 7-11: Setting the background and window tile addressing mode

The window will also use the `$8800` addressing mode.

There are three differences between the window and the background. The first difference is that they don't need to point to the same tilemap. Remember that there can be up to two tilemaps in VRAM. We can set up the window and background to point to the same tilemap or use a separate tilemap for each.

The second difference is that the window does not wrap. The window is just a square area that moves around over the background (although there are some limitations and particularities to be aware of, which I'll explain in more detail in the next section).

The final difference is that the window has its own enable flag (bit 5) in the LCD control register. For example, in Listing 7-11 we enabled the background with the `LCDCF_BGON` flag, but to enable the window, we would need to add the `LCDCF_WINON` flag too.

There's a catch, though: Disabling the background (by clearing bit 0 in `rLCDC`) will also disable the window. That means we can enable the window if and only if we enable the background as well. The possible combinations are background on/window on, background on/window off, and background off/window off. There is no way to get a background off/window on combination.

Table 7-2 summarizes the LCD flags that affect the window.

Table 7-2: Window Flag Values

Bit	Value	Shared	Meaning
0	`LCDCF_BGOFF`	Yes	The background and window are disabled.
	`LCDCF_BGON`	Yes	The background and window are enabled.
4	`LCDCF_BG8000`	Yes	The `$8000` addressing mode is used.
	`LCDCF_BG8800`	Yes	The `$8800` addressing mode is used.
5	`LCDCF_WINOFF`	No	The window is disabled.
	`LCDCF_WINON`	No	The window is enabled.
6	`LCDCF_WIN9800`	No	The window tilemap is located at `$9800`.
	`LCDCF_WIN9C00`	No	The window tilemap is located at `$9C00`.

These flags are used in the `rLCDC` register to control the window parameters alongside the flags for controlling the other high-level graphics features.

Moving the Window

Moving the window is similar to scrolling the background. We use the `rWX` and `rWY` registers to set the position of the top-left corner of the window.

In the `rWY` register, 0 represents the top line of the LCD. A value of 144 or more will put the window beyond the bottom of the screen, making it invisible.

The `rWX` register is a bit peculiar. First, the leftmost column of the LCD is at abscissa 7, not 0, as one would expect. This means that the coordinate for the top-left corner of the LCD is (7, 0). Second, there is a hardware bug that will very likely cause the window to display improperly for certain values of `rWX`: namely, anything in the 0–6 range and the value 166. Entering a value in the 0–6 range would technically bring the window slightly beyond the left edge of the LCD. A value of 166 would display only a single column of pixels from the window on the right side of the LCD. The bug will cause the window to either display at the wrong location, lack some columns of pixels, or flicker on the LCD. It's best to stay away from using the buggy values of `rWX` altogether.

In practice, you can safely move the window anywhere between coordinates (7, 0) and (165, 143). At (7, 0), the window completely covers the background. For any abscissa beyond 167 and any ordinate beyond 143, the window will be outside the bounds of the LCD screen and thus invisible. For any coordinates between (7, 0) and (165, 143), the window will partially cover the background, moving up and to the left from the bottom-right corner.

Figure 7-5 shows these three cases, using the background and window from the simple sample in Figures 5-3 and 5-4.

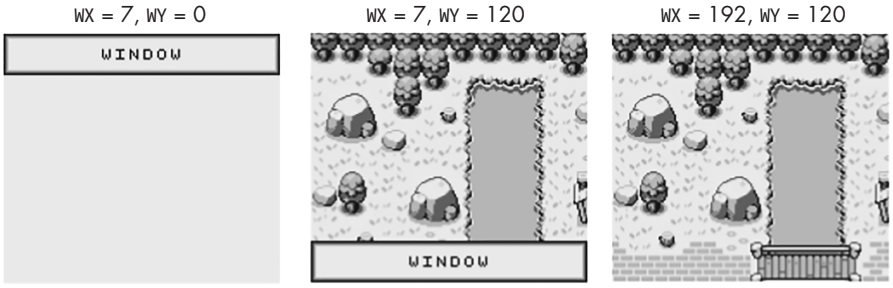

Figure 7-5: Window location examples

On the left, the window is at the origin and covers the whole background. In the center, the window is at its actual position in the sample. It covers only the portion of the background at the bottom of the LCD. Finally, on the right, the window is positioned beyond the bounds of the LCD display (to the right) and is not visible at all.

The window uses a tilemap for its graphics, which is 256×256 pixels, but the window itself has a fixed size of 160×144 pixels (the same dimensions as the LCD screen). If you consider the tilemap surface in Figure 7-6, the light area would potentially display on the LCD, while the dark area would never display.

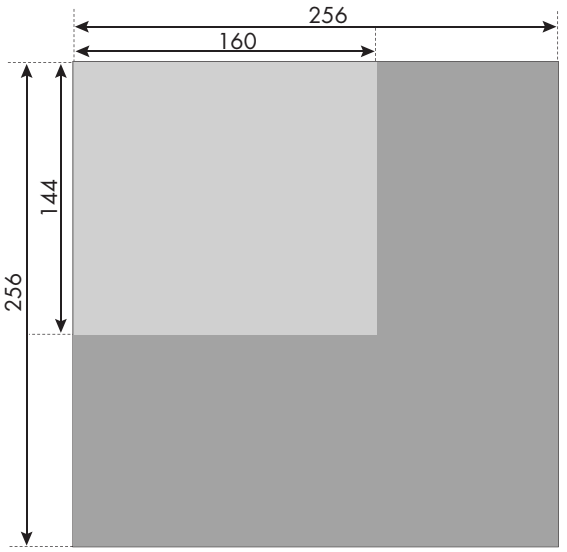

Figure 7-6: Unused tilemap parts

Keep the light area in mind when designing tilemaps for use with the window. The dark area won't necessarily be wasted, because the background can use it. This would most likely require using advanced graphics tricks, though, which I'll cover later in this book.

Displaying the Window Over the Background

To wrap up this section, let's explore the *window* sample that displays the window over the background.

We'll start by adding the necessary graphics data to the ROM, as shown in Listing 7-12.

```
section "graphics_data", rom0[GRAPHICS_DATA_ADDRESS_START]
incbin "tileset.chr"
incbin "background.tlm"
incbin "window.tlm"
```

Listing 7-12: Adding the window tilemap

The *window.tlm* tilemap contains graphics for the window and is now at the end of the `"graphics_data"` section.

Next, we update the computation of this section's starting address, `GRAPHICS_DATA_ADDRESS_START`, to reflect the fact that two tilemaps are now used. Listing 7-13 shows the revised calculation.

```
def TILEMAPS_COUNT              equ (2)
def BYTES_PER_TILEMAP           equ (1024)
def TILEMAPS_BYTE_SIZE          equ (TILEMAPS_COUNT * BYTES_PER_TILEMAP)

def GRAPHICS_DATA_SIZE          equ (TILES_BYTE_SIZE + TILEMAPS_BYTE_SIZE)
```

```
def GRAPHICS_DATA_ADDRESS_END       equ ($8000)
def GRAPHICS_DATA_ADDRESS_START \
    equ (GRAPHICS_DATA_ADDRESS_END - GRAPHICS_DATA_SIZE)
```

Listing 7-13: Updating the constants to position the graphics assets

Finally, we initialize the window coordinates and set the tilemap address, as shown in Listing 7-14.

```
    ; place the window at the bottom of the LCD
    ld a, 7
    ld [rWX], a
    ld a, 120
    ld [rWY], a

    ; set the graphics parameters and turn the LCD back on
    ld a, LCDCF_ON | LCDCF_WINON | LCDCF_WIN9C00 | \
          LCDCF_BG8800 | LCDCF_BG9800 | LCDCF_BGON
    ld [rLCDC], a
```

Listing 7-14: Positioning the window and setting the `rLCDC` flags

We place the window at coordinates (7, 120) by setting `rWX` and `rWY`. Then, we add two flags, `LCDCF_WINON` and `LCDCF_WIN9C00`, to the LCD control register. The first flag enables the window, and the second flag makes the window point to the second tilemap (at `$9C00` in VRAM).

Figure 7-7 shows the results after compiling and running the sample in BGB.

Figure 7-7: The window *sample*

If you're feeling adventurous, try changing the `rWX` value to one of the unreliable values discussed earlier. You'll be able to see the hardware bug in action.

Sprites

Sprites are used to represent moving elements, such as characters or projectiles, which would be difficult, if not impossible, to depict with the background and window. In this section, we'll look at how they're used and what the limitations are.

Palettes

The first thing you need to set up for sprites are the palettes. Whereas the background and window have only one palette, sprites come with two palette options: `rOBP0` and `rOBP1`.

Each sprite can use only one palette at a time, but it's easy to swap palettes. Usually, we use the second palette for palette switching effects. For example, a character taking damage can briefly switch to the second palette to reflect its weakened status.

For sprites, the color at index 0 in a palette is the *transparent color*. If a sprite has pixels of transparent color, the background, window, and even other sprites will show through the transparent parts. This is a handy feature, because it lets you use sprites to represent non-square objects. For example, you could depict a round ball using the transparent color for pixels outside the ball graphics. The only downside to this feature is that it means sprites have only three visible colors. We'll see the transparent color in action in the samples in this section.

LCD Parameters

The LCD control register has two flags that affect sprites. The first one is the enable flag, which turns the sprites on and off. Unlike the window, which can be enabled only if the background is also enabled, sprites are totally independent of the background and window.

The second is the size flag, which controls whether sprites are 8×8 pixels (one tile) or 8×16 pixels (two tiles, aligned vertically). This parameter affects all sprites: They are either all 8×8 or all 8×16.

Table 7-3 summarizes the sprite-related flags.

Table 7-3: Sprite Flag Values

Bit	Value	Meaning
0	`LCDCF_OBJOFF`	The sprites are disabled.
	`LCDCF_OBJON`	The sprites are enabled.
3	`LCDCF_OBJ8`	The sprites are 8×8 (one tile).
	`LCDCF_OBJ16`	The sprites are 8×16 (two tiles).

Note that there is no tile addressing mode flag for sprites; sprites always use the `$8000` tile addressing mode.

Object Attribute Memory

Each sprite has four attributes: an abscissa, an ordinate, a tile index, and control flags. Each attribute takes 1 byte in memory, so each sprite takes 4 bytes. The sprite attributes are set in a 160-byte dedicated memory area called the *object attribute memory (OAM)*. The OAM extends from `\$FE00` to `$FE9F`.

The PPU accesses the OAM when drawing the sprites to the LCD. Like the VRAM, the OAM is mostly inaccessible during the LCD refresh, so for now we'll set the sprites' attributes during the vblank. We'll look at an alternative technique for doing this in Chapter 12.

The first two attributes are the abscissa, X, and the ordinate, Y. Both attributes refer to the coordinates of the top-left corner of the sprite. X is equal to the horizontal position of the sprite plus 8. Y is equal to the vertical position of the sprite plus 16. That means when it comes to positioning sprites, the top-left corner of the LCD is actually at coordinates (8, 16). This allows the sprites to enter the left and upper sides of the LCD gradually. When a sprite's X or Y value is 0, it's outside the bounds of the LCD display and thus completely invisible. The same goes when X is above 168 (160 + 8) or Y is above 160 (144 + 16).

The third attribute to configure for sprites is the tile index. As mentioned previously, sprites always use the `$8000` tile addressing mode. So, the indices go from 0 to 255 and can refer to any tiles in blocks 0 and 1. When using 8×16 sprites (two tiles), this attribute is the index of the first tile, which renders the top of the sprite. We cannot specify the second tile's index; it's always the value of the index attribute plus one. The second tile renders the bottom of the sprite.

The fourth and last attribute is a byte containing flags to control the sprite rendering. Table 7-4 summarizes these flags, which use only 4 bits (bits 0 to 3 are unused).

Table 7-4: OAM Flags

Bit	Description
4	Selects the palette number of the sprite
5	Flips the sprite horizontally when the bit is set
6	Flips the sprite vertically when the bit is set
7	Draws the background and window colors 1 to 3 over the sprite when the bit is set

Bit 4 selects one of the two sprite palettes. When the bit is zero, the first sprite palette is used; otherwise, the second palette is used.

Bit 5 and 6 flip (mirror) the sprite horizontally and vertically, respectively. This is a handy option to save on the number of tiles used for characters, because it avoids using separate tiles for the character facing left and right. Figure 7-8 shows an 8×16 sprite flipped in four different ways.

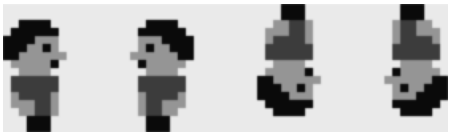

Figure 7-8: Flipping a sprite in different ways

From left to right, the four orientations are no flip, horizontal flip, vertical flip, and both flips. Combining both flips is equivalent to a 180-degree rotation.

By default, the sprites display over the background and window. Sprites are opaque except when pixels use color index 0, which lets the background and window show through. When bit 7 is set, the background and window display over the sprites. Figure 7-9 illustrates how this works.

Figure 7-9: The draw priority of a sprite

The house is part of the background and has two windows with color index 0, so what's *behind* shows through them. On the left, no sprites are visible, just the house. In the middle, a character sprite has bit 7 of its flags cleared and thus appears in front of the house. On the right, the sprite has bit 7 of its flags set and thus appears behind the house but is partially visible through the window.

Limitations

There are three limitations to sprites that you should be aware of. First is the number of sprites you can store. Because each sprite takes 4 bytes, there can be only 40 sprites in OAM at any given time. We'll see how to partially compensate for this limitation in Chapter 12.

Second, at most 10 sprites can display per LCD line. The PPU identifies the first 10 sprites in OAM that are on each line and ignores any additional sprites on the line. This limitation affects several commercial games, resulting in sprites flickering on the screen.

Finally, you can't make a single sprite invisible through flags. Instead, you have to move that sprite off the screen. The issue isn't too difficult to address, but it means that when you need to make a sprite invisible temporarily, you must save its position.

Also, keep in mind that it's better to move the sprite above or below the screen (using an ordinate equal to 0 or higher than 168) rather than to the sides (using an abscissa equal to 0 or higher than 168), because sprites that aren't visible on the LCD display will still count toward the maximum number of sprites allowed per line (10).

A Sprite Sample

Let's get some practice using sprites. You can follow along using the *sprites* sample.

First, we'll set the parameters for all sprites, starting with the two sprite palettes, as shown in Listing 7-15.

```
    ld a, %11100100
    ld [rBGP], a
    ld [rOBP0], a
    ld a, %00011011
    ld [rOBP1], a
```

Listing 7-15: Configuring the palettes

We set the first palette to the same value as the background palette. The second palette is set to a different value, but it is not actually referenced by any sprite in the sample.

Next, we set the `rLCDC` flags to enable and configure the sprites, as shown in Listing 7-16.

```
    ld a, LCDCF_ON | LCDCF_WIN9C00 | LCDCF_WINON | LCDCF_BG8800 | \
          LCDCF_BG9800 | LCDCF_OBJ16 | LCDCF_OBJON | LCDCF_BGON
```

Listing 7-16: Setting the flags

The `LCDCF_OBJON` flag enables the sprites, and the `LCDCF_OBJ16` flag makes them 8×16 pixels.

We'll set the sprite attributes inside the `UpdateSample` function. Although it's not really necessary in this sample (because the sprite attributes won't change over time), in a real-world situation you'd update the sprites frequently, and the update function is the right place to do this. Because the OAM is mostly inaccessible outside the vblank, we need to set the sprite attributes after `halt`, within the duration of the vblank (1.08 ms).

We use simple load instructions to set the attributes: one instruction to load the attribute value in `a` and another to load `a` into an OAM attribute. These two instructions have been abstracted by a new macro, `copy`, shown in Listing 7-17. The macro is defined in *utils.rgbinc*.

```
; copy \2 into \1 through (a)
; example: copy [$FF84], 10
macro copy
    ld a, \2
    ld \1, a
endm
```

Listing 7-17: The copy macro

This macro covers the cases when two loads are necessary to move a value from a source to a destination, using `a` as an intermediary. We use it to set up two sprites over a few lines of code, as illustrated in Listing 7-18.

```
    ; set up the first sprite
def SPRITE_0_ADDRESS equ (_OAMRAM)
    copy [SPRITE_0_ADDRESS + OAMA_X], 16
    copy [SPRITE_0_ADDRESS + OAMA_Y], 32
    copy [SPRITE_0_ADDRESS + OAMA_TILEID], 16
    copy [SPRITE_0_ADDRESS + OAMA_FLAGS], OAMF_PAL0

    ; set up the second sprite
def SPRITE_1_ADDRESS equ (_OAMRAM + sizeof_OAM_ATTRS)
    copy [SPRITE_1_ADDRESS + OAMA_Y], 80
    copy [SPRITE_1_ADDRESS + OAMA_X], 80
    copy [SPRITE_1_ADDRESS + OAMA_TILEID], 0
    copy [SPRITE_1_ADDRESS + OAMA_FLAGS], OAMF_PAL0 | OAMF_XFLIP
```

Listing 7-18: Defining the sprites

The code defines two constants, `SPRITE_0_ADDRESS` and `SPRITE_1_ADDRESS`, that hold the addresses of the first two sprites in OAM. The attributes are filled in for each sprite. The first sprite is a front-facing character positioned in the upper-left corner of the screen. The second sprite is a right-facing character, but we flip it horizontally to make it face left instead. It is located close to the center of the screen.

If you compile and run the sample, the two sprites should display as expected. However, you will likely get some visual corruption, as shown in Figure 7-10. (If it isn't visible right away, try running the sample a few more times.)

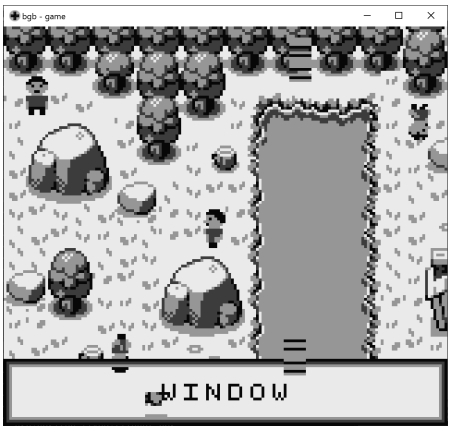

Figure 7-10: Visual corruption

Some graphics tiles appear randomly over the background and window. The issue is that the OAM, the memory that contains the sprite attributes, is not initialized by default. That means the 38 sprites that we're not using have attributes set to random values, causing them to display unexpectedly all over the screen.

To fix this visual corruption, we must initialize the OAM to position all sprites outside the screen by default. We do this with the `InitOAM` macro (Listing 7-19), which is included in the *sample.rgbasm* file in the *sprites_fixed* sample.

```
macro InitOAM
    ld c, OAM_COUNT
    ld hl, _OAMRAM + OAMA_Y
    ld de, sizeof_OAM_ATTRS
    .init_oam\@
        ld [hl], 0
        add hl, de
        dec c
        jr nz, .init_oam\@
endm
```

Listing 7-19: The `InitOAM` macro

This macro is a loop iterating over the OAM, setting the Y attribute of all 40 sprites to 0. It starts by initializing the registers used in the macro. First, `c` is set to `OAM_COUNT` (40). This acts as the loop index. Next, `hl` is set to point to the Y attribute of the first sprite in the OAM. Finally, `de` is set to `sizeof_OAM_ATTRS`, which is the number of bytes for the attributes of a single sprite (4).

The loop spans the remainder of the macro. For each iteration, the byte pointed to by `hl` (the Y attribute of one of the sprites in OAM) is set to zero. Then, `hl` is incremented by 4 so that it points to the next sprite's Y attribute. The register `de` holds the value 4, because the only 16-bit `add` instruction takes registers as parameters, not literals. Finally, the code jumps back to the start of the loop until all iterations are done.

Once the loop is completed, all the sprites' Y attributes are equal to zero, moving them outside the LCD display area and removing the graphical artifacts from Figure 7-10. You don't need to initialize any of the other attributes (X, tile index, and control flags), because you can initialize them before the sprites actually get used.

WARNING

As a rule of thumb, it's best to assume that memory is not initialized. BGB seems to do the right thing here by having random values in the OAM, just as the real hardware would, but some emulators might mistakenly zero-clear some of the memory areas, including the OAM. If you make the wrong assumptions about the state of the memory, you might find that your program works fine on an emulator but not on real hardware.

Finally, we use the `InitOAM` macro in the `InitSample` function, just before `LoadGraphicsDataIntoVRAM` (see Listing 7-20).

```
    ; init graphics data
    InitOAM
    LoadGraphicsDataIntoVRAM
```

Listing 7-20: Invoking the `InitOAM` macro

Now when you compile and run the sample, it should look like Figure 7-11.

Figure 7-11: A sprite sample

All the visual corruption is gone.

Summary

In this chapter, you learned how to leverage the high-level graphics features of the DMG. You should now be able to initialize and scroll the background as well as enable and move the window around the screen. Finally, you learned how to set up the OAM to display sprites.

Combined with what you learned in Chapter 6, you should now have a strong understanding of how to display custom graphics on the console. There is one more topic left with regard to sprites: the DMA transfer, which is a different method for copying data to the OAM. This and other advanced graphics topics will be covered in Chapter 12.

Now we're ready to switch our focus to the input. The next chapter introduces the joypad, which allows us to interact with the DMG.