This is part 7 of an article series on emulation development in JavaScript; ten parts are currently available, and others are expected to follow.

Previously in this series, the emulator was extended to enable keypad input, which meant that a game of tic-tac-toe could be played. The problem left by this was that the game had to be played blind: there was no indication of where the next move would be made, nor of to where on the game a keypress would move you. Traditionally, two-dimensional gaming consoles have solved this issue through the use of sprites: movable object blocks that can be placed independently of the background, and which contain data separate to that of the background.

The GameBoy is no exception in this regard: it provides for sprites to be placed above or below the background, and multiple sprites to be on screen at the same time. Once this has been implemented in the emulator, the tic-tac-toe game runs as below.

Reset | Run

Figure 1: jsGB implementation with sprites

Introduction: GameBoy sprites

GameBoy sprites are graphic tiles, just like those used for the background: this means that each sprite is 8x8 pixels. As stated above, a sprite can be placed anywhere on the screen, including halfway or all the way off-screen, and it can be placed above or below the background. What this means technically is that sprites below the background show through where the background has colour value 0.

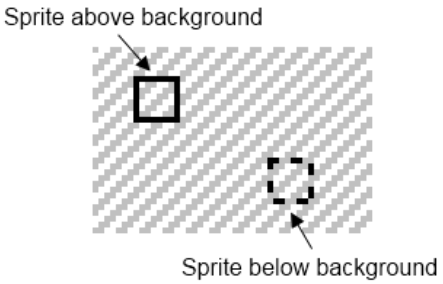

Figure 2: Sprite priorities

In the above figure, the sprite above the background shows the background through the middle of it, since these pixels in the sprite are set to colour 0; in the same way, the background lets through the sprites below it where the background colour is 0. In order to simulate this in an emulator, the simplest procedure would be to render the sprites below the background, then the background itself, and finally the sprites above it. However, this is a somewhat naive algorithm, since it duplicates the sprite rendering process; it's simpler instead to draw the background first, then work out whether a given pixel in the sprite should appear based on its priority and the background colour at that position.

Pseudocode for sprite rendering

```
For each row in sprite
    If this row is on screen
        For each pixel in row
            If this pixel is on screen
                If this pixel is transparent
                    * Do nothing
                Else
                    If the sprite has priority
                        Draw pixel
                    Else if this pixel in the background is 0
                        Draw pixel
                    Else
                        * Do nothing
                    End If
                End If
            End If
        End For
    End If
End For
```

One additional complication to the GameBoy sprite system is that a sprite can be "flipped" horizontally or vertically by the hardware, at the time it's rendered; this saves space in the game, since (for example) a spaceship flying backwards can be represented by the same sprite as forward motion, with the appropriate flip applied.

Sprite data: Object Attribute Memory

The GameBoy can hold information about 40 sprites, in a dedicated region of memory called Object Attribute Memory (OAM). Each of the 40 sprites has four bytes of data in the OAM associated with it, as detailed below.

Byte	Description
0	Y-coordinate of top-left corner (Value stored is Y-coordinate minus 16)

<table>
<tr><td>1</td><td>X-coordinate of top-left corner
(Value stored is X-coordinate minus 8)</td></tr>
<tr><td>2</td><td>Data tile number</td></tr>
<tr><td>3</td><td>Options
<table>
<tr><th>Bit</th><th>Description</th><th>When 0</th><th>When 1</th></tr>
<tr><td>7</td><td>Sprite/background priority</td><td>Above background</td><td>Below background (except colour 0)</td></tr>
<tr><td>6</td><td>Y-flip</td><td>Normal</td><td>Vertically flipped</td></tr>
<tr><td>5</td><td>X-flip</td><td>Normal</td><td>Horizontally flipped</td></tr>
<tr><td>4</td><td>Palette</td><td>OBJ palette #0</td><td>OBJ palette #1</td></tr>
</table></td></tr>
</table>

Table 1: OAM data for a sprite

In order to more easily access this information when it comes to rendering a scanline, it's useful to build a structure to hold the sprite data, which is filled in based on the contents of the OAM. When data is written to the OAM, the MMU in consort with the graphics emulation can update this structure for later use. An implementation of this would be as follows.

MMU.js: OAM access

```
    rb: function(addr)
    {
        switch(addr & 0xF000)
        {
            ...
            case 0xF000:
                switch(addr & 0x0F00)
                {
                    ...
                    // OAM
                    case 0xE00:
                        return (addr < 0xFEA0) ? GPU._oam[addr & 0xFF] : 0;
                }
        }
    },

    wb: function(addr)
    {
        switch(addr & 0xF000)
        {
            ...
            case 0xF000:
                switch(addr & 0x0F00)
                {
                    ...
                    // OAM
                    case 0xE00:
                        if(addr < 0xFEA0) GPU._oam[addr & 0xFF] = val;
                        GPU.buildobjdata(addr - 0xFE00, val);
                        break;
                }
        }
    }
```

GPU.js: Sprite structure

```
    _oam: [],
    _objdata: [],

    reset: function()
    {
        // In addition to previous reset code:
        for(var i=0, n=0; i < 40; i++, n+=4)
        {
            GPU._oam[n + 0] = 0;
            GPU._oam[n + 1] = 0;
            GPU._oam[n + 2] = 0;
            GPU._oam[n + 3] = 0;
            GPU._objdata[i] = {
                'y': -16, 'x': -8,
                'tile': 0, 'palette': 0,
                'xflip': 0, 'yflip': 0, 'prio': 0, 'num': i
            };
        }
    },

    buildobjdata: function(addr, val)
    {
        var obj = addr >> 2;
        if(obj < 40)
        {
            switch(addr & 3)
            {
                // Y-coordinate
                case 0: GPU._objdata[obj].y = val-16; break;

                // X-coordinate
                case 1: GPU._objdata[obj].x = val-8; break;

                // Data tile
                case 2: GPU._objdata[obj].tile = val; break;

                // Options
                case 3:
                    GPU._objdata[obj].palette = (val & 0x10) ? 1 : 0;
                    GPU._objdata[obj].xflip   = (val & 0x20) ? 1 : 0;
                    GPU._objdata[obj].yflip   = (val & 0x40) ? 1 : 0;
                    GPU._objdata[obj].prio    = (val & 0x80) ? 1 : 0;
                    break;
            }
        }
    }
```

Sprite palettes

As hinted above, the GPU offers a choice of two palettes for the sprites: each of the 40 sprites can use one of the two palettes, as specified in its OAM entry. These object palettes are stored in the GPU, in addition to the background palette, and can be changed through I/O registers in much the same manner as the palette for the background.

GPU.js: Sprite palette handling

```
    _pal: {
        bg: [],
        obj0: [],
        obj1: []
    },

    wb: function(addr)
    {
        switch(addr)
        {
            // ...
            // Background palette
            case 0xFF47:
                for(var i = 0; i < 4; i++)
                {
                    switch((val >> (i * 2)) & 3)
                    {
                        case 0: GPU._pal.bg[i] = [255,255,255,255]; break;
                        case 1: GPU._pal.bg[i] = [192,192,192,255]; break;
                        case 2: GPU._pal.bg[i] = [ 96, 96, 96,255]; break;
                        case 3: GPU._pal.bg[i] = [  0,  0,  0,255]; break;
                    }
                }
                break;

            // Object palettes
            case 0xFF48:
                for(var i = 0; i < 4; i++)
                {
                    switch((val >> (i * 2)) & 3)
                    {
                        case 0: GPU._pal.obj0[i] = [255,255,255,255]; break;
                        case 1: GPU._pal.obj0[i] = [192,192,192,255]; break;
                        case 2: GPU._pal.obj0[i] = [ 96, 96, 96,255]; break;
                        case 3: GPU._pal.obj0[i] = [  0,  0,  0,255]; break;
                    }
                }
                break;

            case 0xFF49:
                for(var i = 0; i < 4; i++)
                {
                    switch((val >> (i * 2)) & 3)
                    {
                        case 0: GPU._pal.obj1[i] = [255,255,255,255]; break;
                        case 1: GPU._pal.obj1[i] = [192,192,192,255]; break;
                        case 2: GPU._pal.obj1[i] = [ 96, 96, 96,255]; break;
                        case 3: GPU._pal.obj1[i] = [  0,  0,  0,255]; break;
                    }
                }
                break;
        }
    }
```

Rendering sprites

The GameBoy graphics system renders each line of the screen as it's encountered: this includes not only the background, but the sprites below and above it. In other words, rendering of the

sprites must be added to the scanline renderer, as a process that occurs after drawing the background. Just as with the background, there's a switch to enable sprites within the LCDC register, and this must be added to the I/O handling for the GPU.

Since a sprite can be anywhere on the screen, including positioned somewhere off-screen, the renderer has to check which sprites are positioned within the current scanline. The simplest algorithm for this is to check the position of each one, and render the appropriate line of the sprite if it falls within the bounds of the scanline. The sprite data can be retrieved in the same way as it is for the background, through the pre-calculated tile set. An example of these things brought together is as follows.

GPU.js: Rendering a scanline with sprites

```
    renderscan: function()
    {
        // Scanline data, for use by sprite renderer
        var scanrow = [];

        // Render background if it's switched on
        if(GPU._switchbg)
        {
            var mapoffs = GPU._bgmap ? 0x1C00 : 0x1800;
            mapoffs += ((GPU._line + GPU._scy) & 255) >> 3;
            var lineoffs = (GPU._scx >> 3);
            var y = (GPU._line + GPU._scy) & 7;
            var x = GPU._scx & 7;
            var canvasoffs = GPU._line * 160 * 4;
            var colour;
            var tile = GPU._vram[mapoffs + lineoffs];

            // If the tile data set in use is #1, the
            // indices are signed; calculate a real tile offset
            if(GPU._bgtile == 1 && tile < 128) tile += 256;

            for(var i = 0; i < 160; i++)
            {
                // Re-map the tile pixel through the palette
                colour = GPU._pal.bg[GPU._tileset[tile][y][x]];

                // Plot the pixel to canvas
                GPU._scrn.data[canvasoffs+0] = colour[0];
                GPU._scrn.data[canvasoffs+1] = colour[1];
                GPU._scrn.data[canvasoffs+2] = colour[2];
                GPU._scrn.data[canvasoffs+3] = colour[3];
                canvasoffs += 4;

                // Store the pixel for later checking
                scanrow[i] = GPU._tileset[tile][y][x];

                // When this tile ends, read another
                x++;
                if(x == 8)
                {
                    x = 0;
                    lineoffs = (lineoffs + 1) & 31;
                    tile = GPU._vram[mapoffs + lineoffs];
                    if(GPU._bgtile == 1 && tile < 128) tile += 256;
                }
            }
```

```
        }
    }

    // Render sprites if they're switched on
    if(GPU._switchobj)
    {
        for(var i = 0; i < 40; i++)
        {
            var obj = GPU._objdata[i];

            // Check if this sprite falls on this scanline
            if(obj.y <= GPU._line && (obj.y + 8) > GPU._line)
            {
                // Palette to use for this sprite
                var pal = obj.pal ? GPU._pal.obj1 : GPU._pal.obj0;

                // Where to render on the canvas
                var canvasoffs = (GPU._line * 160 + obj.x) * 4;

                // Data for this line of the sprite
                var tilerow;

                // If the sprite is Y-flipped,
                // use the opposite side of the tile
                if(obj.yflip)
                {
                    tilerow = GPU._tileset[obj.tile]
                                          [7 - (GPU._line - obj.y)];
                }
                else
                {
                    tilerow = GPU._tileset[obj.tile]
                                          [GPU._line - obj.y];
                }

                var colour;
                var x;

                for(var x = 0; x < 8; x++)
                {
                    // If this pixel is still on-screen, AND
                    // if it's not colour 0 (transparent), AND
                    // if this sprite has priority OR shows under the bg
                    // then render the pixel
                    if((obj.x + x) >= 0 && (obj.x + x) < 160 &&
                       tilerow[x] &&
                       (obj.prio || !scanrow[obj.x + x]))
                    {
                        // If the sprite is X-flipped,
                        // write pixels in reverse order
                        colour = pal[tilerow[obj.xflip ? (7-x) : x]];

                        GPU._scrn.data[canvasoffs+0] = colour[0];
                        GPU._scrn.data[canvasoffs+1] = colour[1];
                        GPU._scrn.data[canvasoffs+2] = colour[2];
                        GPU._scrn.data[canvasoffs+3] = colour[3];

                                                  ff+= 4
```

```
                        canvasoffs += 4;
                    }
                }
            }
        }
    },

    rb: function(addr)
    {
        switch(addr)
        {
            // LCD Control
            case 0xFF40:
                return (GPU._switchbg  ? 0x01 : 0x00) |
                       (GPU._switchobj ? 0x02 : 0x00) |
                       (GPU._bgmap     ? 0x08 : 0x00) |
                       (GPU._bgtile    ? 0x10 : 0x00) |
                       (GPU._switchlcd ? 0x80 : 0x00);

            // ...
        }
    },

    wb: function(addr, val)
    {
        switch(addr)
        {
            // LCD Control
            case 0xFF40:
                GPU._switchbg  = (val & 0x01) ? 1 : 0;
                GPU._switchobj = (val & 0x02) ? 1 : 0;
                GPU._bgmap     = (val & 0x08) ? 1 : 0;
                GPU._bgtile    = (val & 0x10) ? 1 : 0;
                GPU._switchlcd = (val & 0x80) ? 1 : 0;
                break;

            // ...
        }
    }
```

Coming up

With sprites in place, basic games like the tic-tac-toe running in Figure 1 can work in full. Many games, however, will not run without something else: a method of determining when the screen can be redrawn. Almost every game will perform a "refresh" of the screen data while the screen is in vertical blanking, since changes to the screen won't show up until the next time the GPU comes to draw a frame.

Basic games and demos sometimes do this by checking whether the GPU has hit line #144 in its redrawing process, but this takes up a lot of processing power in repeated looping. The more common method is for the game to be informed when an event has occurred: this message is referred to as an interrupt. In the next part, I'll take a look at the vertical blanking interrupt in particular, and how it can be simulated to provide this message passing process to an emulated game.

Imran Nazar <tf@imrannazar.com>, Oct 2010.

Article dated: 10th Oct 2010