This is part 5 of an article series on emulation development in JavaScript; ten parts are currently available, and others are expected to follow.

- *Part 1: The CPU*
- *Part 2: Memory*
- *Part 3: GPU Timings*
- *Part 4: Graphics*
- *Part 5: Integration*
- *Part 6: Input*
- *Part 7: Sprites*
- *Part 8: Interrupts*
- *Part 9: Memory Banking*
- *Part 10: Timers*

In part 4, the GameBoy's graphics subsystem was explored in detail, and an emulation put together. Without a set of register mappings for the GPU to be dealt with in software, the graphics subsystem cannot be used by the emulator; once these registers have been made available, the emulator is essentially ready for basic use.

With the additions detailed below to add the GPU registers, and a basic interface for the control of the emulator, the result is as follows.

Reset | Run

Figure 1: jsGB implementation with graphics

GPU registers

The graphics unit of the GameBoy has a series of registers which are mapped into memory, in the I/O space of the memory map. In order to get a working emulation with a background image, the following registers will be needed by the GPU (other registers are also available to the GPU, and will be explored in later parts of this series).

Address	Register	Status
0xFF40	LCD and GPU control	Read/write

0xFF42	Scroll-Y	Read/write
0xFF43	Scroll-X	Read/write
0xFF44	Current scan line	Read only
0xFF47	Background palette	Write only

Table 1: Basic GPU registers

The background palette register has previously been explored, and consists of four 2-bit palette entries. The scroll registers and scanline counter are full-byte values; this leaves the LCD control register, which is made up of 8 separate flags controlling the sections of the GPU.

Bit	Function	When 0	When 1
0	Background: on/off	Off	On
1	Sprites: on/off	Off	On
2	Sprites: size (pixels)	8x8	8x16
3	Background: tile map	#0	#1
4	Background: tile set	#0	#1
5	Window: on/off	Off	On
6	Window: tile map	#0	#1
7	Display: on/off	Off	On

Table 2: GPU control register

In the above table, the additional features of the GPU appear: a "window" layer which can appear above the background, and sprite objects which can be moved against the background and window. These additional features will be covered as the need for them arises; in the meantime, the background flags are most important for basic rendering functions. In particular, it can be seen here how the background tile map and tile set can be changed, simply by flipping bits in the register `0xFF40`.

Implementation: GPU registers

Armed with the conceptual GPU register layout, an emulation can be implemented simply by adding handlers for these addresses to the MMU. This can either be done by hard-coding the GPU updates into the MMU, or defining a range of registers wherein the GPU will be called from the MMU, for more specialised handling to be done from there. In the interests of modularity, the latter approach has been taken here.

MMU.js: Zero-page I/O: GPU

```
    rb: function(addr)
    {
        switch(addr & 0xF000)
        {
            ...
            case 0xF000:
                switch(addr & 0x0F00)
                {
                    ...
                    // Zero-page
                    case 0xF00:
                        if(addr >= 0xFF80)
                        {
```

```
                        return MMU._zram[addr & 0x7F];
                    }
                    else
                    {
                        // I/O control handling
                        switch(addr & 0x00F0)
                        {
                            // GPU (64 registers)
                            case 0x40: case 0x50: case 0x60: case 0x70:
                                return GPU.rb(addr);
                        }
                        return 0;
                    }
                }
            }
        },

        wb: function(addr, val)
        {
            switch(addr & 0xF000)
            {
                ...
                case 0xF000:
                    switch(addr & 0x0F00)
                    {
                        ...
                        // Zero-page
                        case 0xF00:
                            if(addr >= 0xFF80)
                            {
                                MMU._zram[addr & 0x7F] = val;
                            }
                            else
                            {
                                // I/O
                                switch(addr & 0x00F0)
                                {
                                    // GPU
                                    case 0x40: case 0x50: case 0x60: case 0x70:
                                        GPU.wb(addr, val);
                                        break;
                                }
                            }
                            break;
                    }
                    break;
            }
        }
```

GPU.js: Register handling

```
        rb: function(addr)
        {
            switch(addr)
            {
                // LCD Control
                case 0xFF40:
                    return (GPU._switchbg  ? 0x01 : 0x00) |
```

```
                    (GPU._bgmap     ? 0x08 : 0x00) |
                    (GPU._bgtile    ? 0x10 : 0x00) |
                    (GPU._switchlcd ? 0x80 : 0x00);

            // Scroll Y
            case 0xFF42:
                return GPU._scy;

            // Scroll X
            case 0xFF43:
                return GPU._scx;

            // Current scanline
            case 0xFF44:
                return GPU._line;
        }
    },

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

            // Scroll Y
            case 0xFF42:
                GPU._scy = val;
                break;

            // Scroll X
            case 0xFF43:
                GPU._scx = val;
                break;

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

Running one frame

At present, the dispatch loop for the emulator's CPU runs forever, without pause. The most basic interface for an emulator allows for the simulation to be reset or paused; in order to allow for this, a known amount of time must be used as the base unit of the emulator interface. There are three possible units of time that can be used for this:

- **Instruction**: Providing the opportunity to pause after every CPU instruction. This causes a great deal of overhead, since the dispatch function must be called for each step made by the CPU; at 4.19MHz, many steps must be made for an appreciable amount to happen.
- **Scanline**: Pausing after the rendering of each line by the GPU. This produces less of an overhead, but the dispatcher must still be called a few thousand times a second; in addition, the emulation can be paused in a state where the canvas display doesn't correspond to the current scanline.
- **Frame**: Allowing for the emulation to stop after a whole frame is emulated, rendered and pushed to the canvas. This provides the best compromise of timing accuracy and optimal speed, while ensuring that the emulated canvas is consistent with the GPU state.

Since a frame is made of 144 scanlines and a 10-line vertical blank, and each scanline takes 456 clock cycles to run, the length of a frame is 70224 clocks. In conjunction with an emulator-level reset function, which initialises each subsystem at the start of the emulation, the emulator itself can be run, and a rudimentary interface provided.

index.html: Emulator interface

```
<canvas id="screen" width="160" height="144"></canvas>
<a id="reset">Reset</a> | <a id="run">Run</a>
```

jsGB.js: Reset and dispatch

```
jsGB = {
    reset: function()
    {
        GPU.reset();
        MMU.reset();
        Z80.reset();

        MMU.load('test.gb');
    },

    frame: function()
    {
        var fclk = Z80._clock.t + 70224;
        do
        {
            Z80._map[MMU.rb(Z80._r.pc++)]();
            Z80._r.pc &= 65535;
            Z80._clock.m += Z80._r.m;
            Z80._clock.t += Z80._r.t;
            GPU.step();
        } while(Z80._clock.t < fclk);
    },

    _interval: null,

    run: function()
    {
        if(!jsGB._interval)
        {
            jsGB._interval = setTimeout(jsGB.frame, 1);
            document.getElementById('run').innerHTML = 'Pause';
```

```
        }
        else
        {
            clearInterval(jsGB._interval);
            jsGB._interval = null;
            document.getElementById('run').innerHTML = 'Run';
        }
    }
};

window.onload = function()
{
    document.getElementById('reset').onclick = jsGB.reset;
    document.getElementById('run').onclick = jsGB.run;
    jsGB.reset();
};
```

Testing

Previously shown in Figure 1 is the result of bringing this code together: the emulator is capable of loading and running a graphics-based demo. In this case, the test ROM being loaded is a scrolling test written by Doug Lanford: the background displayed will scroll when one of the directional keypad buttons is pressed. In this particular case, with the keypad un-emulated, a static background is displayed.

In the next part, this piece of the jigsaw will be put in place: a keypad simulation which can provide the appropriate inputs to the emulated program. I'll also be looking at how the keypad works, and how the inputs are mapped into memory.

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

Article dated: 5th Sep 2010