从零开始实现GameBoy模拟器 #9 绘制精灵

銀葉吉祥
浙江大学 软件工程硕士

目录 收起

欢迎来到从零开始实现GameBoy模拟器第九章。在本章中，我们将进一步完善PPU，实现精灵绘制功能，从而让我们的模拟器能够绘制完整的游戏画面。那么，让我们开始吧！

> 本章节代码已经上传至项目仓库，名称为LunaGB-09，已经下载项目代码的读者可以更新代码库来查阅最新代码。同时，欢迎大家加入GameBoy模拟器开发交流群：33854734
> 2024.2.21更新：修改PPU::lcd_draw_pixel，对当前的draw_x值进行判断，避免绘制越界。
> 2024.3.13更新：修改了DMA传输部分的表述。

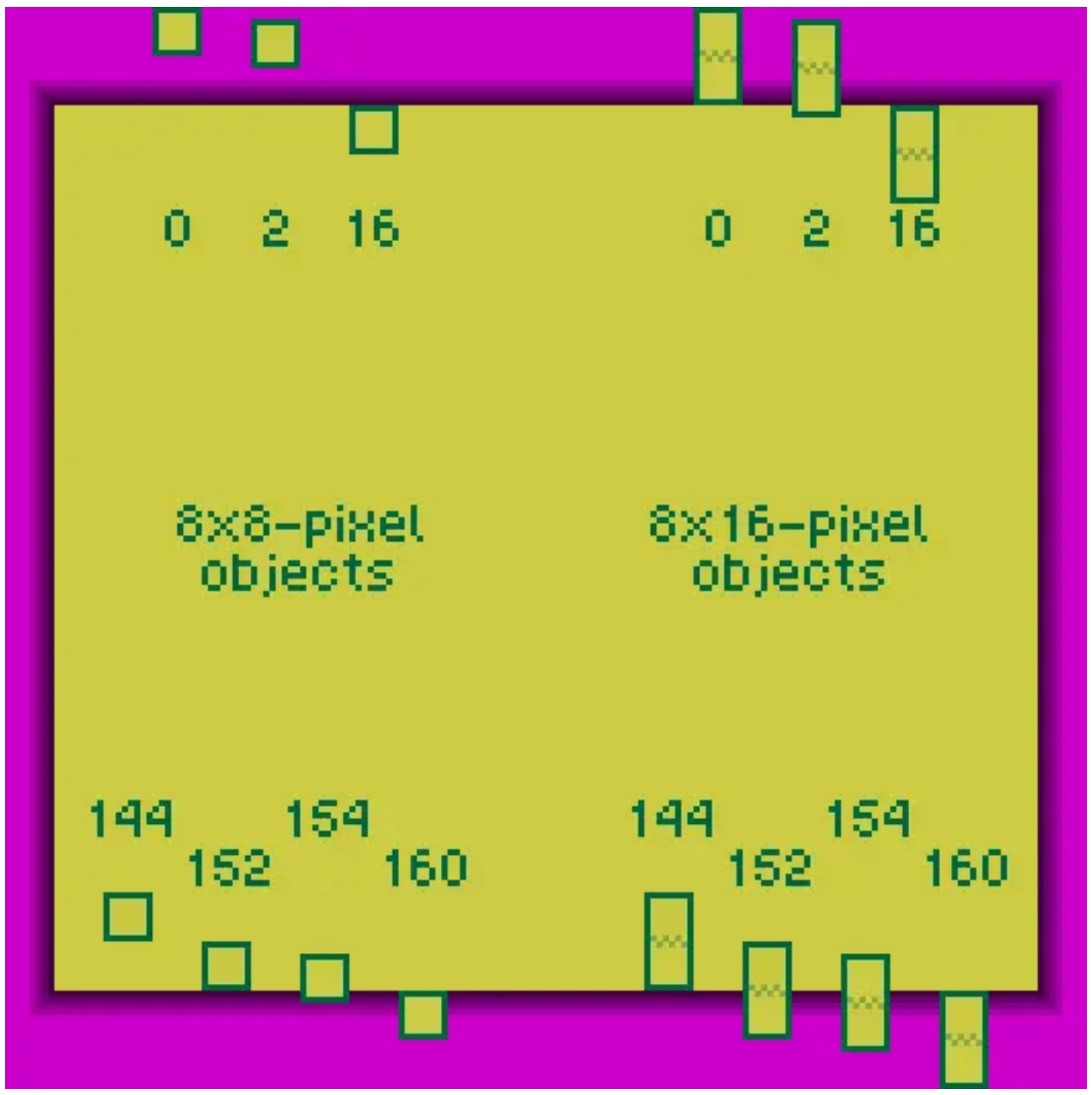
对象属性内存（OAM）

在实际进行精灵图块的绘制前，我们首先需要了解精灵绘制的原理。精灵和背景/窗口一样是由图块构成的，一个精灵可以由一个图块构成，因此具有8x8像素的尺寸，也可以由在图块内存中连续存储的两个图块构成，这两个图块上下拼接，使得精灵具有8x16像素的尺寸。精灵的尺寸由LCDC.2全局控制，因此一个游戏中通常只会出现8x8的精灵或者8x16的精灵。每一个精灵可以被放置在以屏幕左上角为原点的屏幕坐标系中的任意位置，并且精灵支持水平翻转和垂直翻转。GameBoy内置两个用于精灵着色的调色板，每一个精灵可以选择其中一个调色板用于精灵像素的着色。

和背景/窗口图块一样，精灵图块的像素数据本身并不保存精灵本身的位置、颜色等属性。在上一章节中，我们知道背景/窗口图块使用一个32x32字节的索引矩阵来表示背景/窗口每一个图块在图块像素数据区域中的位置。精灵对象的属性也使用类似的方法保存，其保存在地址为0xFE00~0xFE9F的一块被称为对象属性内存（Object Attribute Memory, OAM）的特殊内存区域中。这一块总计160字节的内存区域既不在WRAM中，也不在VRAM中，而是直接嵌入在PPU中，并通过总线映射供程序读写。在OAM中，每一个精灵对象使用四个字节来存储其属性信息，因此PPU最多可以同时存储40个精灵的属性，并绘制这些精灵。**由于OAM位于PPU内部，其只有当PPU处于VBLANK和HBLANK模式下才可以被程序直接读写。**

对于每个精灵对象来说，其在OAM中的四个字节从低地址到高地址的作用如下：

- 字节0：Y坐标。OAM中存储的是精灵的实际Y坐标+16像素后的值，因此允许精灵的Y坐标最低为-16，即完全在屏幕顶部的上边。由于精灵的高度可以是16个像素，因此GameBoy在设计的时候使用的是+16，而非+8像素。精灵的坐标原点在精灵图像的左上角。下图展示了不同高度的精灵在不同的Y坐标值时在屏幕上的位置。



来源：https://gbdev.io/pandocs/OAM.html

- 字节1：X坐标。OAM中存储的是精灵的实际X坐标+8像素后的值，因此允许精灵的X坐标最小为-8，即完全在屏幕左侧的左边。
- 字节2：图块索引。该字节存储了精灵绘制所使用的图块在图块像素数据内存区域中的位置，以索引值0~255表示。对于8x16像素的精灵，该字节的最低位会被忽略，并在读取上半部分图块时设置为0，在读取下半部分图块时设置为1。
- 字节3：对象属性标志位。每一位的作用如下表所示：

7	6	5	4	3	2	1	0
优先级	垂直翻转	水平翻转	调色板	-	-	-	-

- 优先级：该值确定了背景像素相对于精灵像素的绘制优先级。当该值为0时，非00**色值**的精灵像素会覆盖背景像素；当该值为1时，**颜色不为全白（00）**的背景像素会覆盖精灵像素。
- 水平翻转：当该值为1时，精灵图块会被水平翻转后绘制。
- 垂直翻转：当该值为1时，精灵图块会被垂直翻转后绘制。
- 调色板：选择绘制该精灵时使用的调色板。0：第一个精灵调色板；1：第二个精灵调色板。

DMA传输

虽然程序允许直接读写OAM区域数据，但是更建议的方式是先把需要写入OAM的数据保存在RAM中，然后调用DMA传输（DMA transfer）功能让PPU将数据自动从RAM中写入OAM中。DMA是Direct Memory Access的缩写，该功能可以在无需CPU介入的情况下，由DMA控制器控制，将一段数据从一个地址拷贝至另一个地址，从而实现高效的数据传输。在GameBoy上，DMA控制器内嵌在PPU中，可以将128字节的数据自动从RAM（0x8000~0xDFFF）中的任意指定位置拷贝至OAM区域中。

使用DMA传输代替CPU传输的主要优点就是快。在GameBoy上，一段由CPU驱动的内存拷贝代码至少包括以下几个指令：

```
copy:
ld A, (BC)  // BC存储拷贝源地址
ld (HL+), A // HL存储拷贝目标地址
inc BC      // BC地址增加1
inc D       // D用于记录当前拷贝的字节数
ld A, d8    // d8为希望拷贝的总数
cp D        // 判断当前是否已经拷贝了足够的数据
jr nz copy
```

可以看到，使用CPU进行内存拷贝，需要13个机器周期才能完成一个字节的拷贝，而DMA控制器可以在**每个机器周期拷贝一个字节的数据**，相比CPU拷贝快了整整13倍！

在单色GameBoy中，由于DMA工作期间需要占用总线，而GameBoy的所有部件共用一条总线，

因此在DMA传输完成前，CPU无法访问WRAM、ROM、CRAM等外部存储芯片中的数据，只能访问HRAM的数据，因为HRAM的内存直接嵌入在CPU中，不需要通过总线进行访问。在彩色GameBoy中，由于卡带和WRAM使用两条独立的总线，因此CPU在DMA工作期间除了可以访问WRAM之外，也可以访问卡带的ROM和CRAM数据。该特性在开发GameBoy游戏时需要特别留意，但是对我们实现模拟器并不影响，因为能够发行的游戏卡带已经保证了这些规则会被正确遵守。

0xFF46 - DMA

该寄存器用于启动DMA传输。将任意值写入该寄存器都会启动一个新的DMA传输，写入的值指定了传输所需的源数据的读取地址除以0x100后的值，因此写入0x80会从0x8000~0x809F中读取数据，写入0xDF会从0xDF00~0xDF9F中读取数据，诸如此类。DMA每次运行都固定消耗160个机器周期，在每一个机器周期中会传输一个字节的数据，因此总共160字节，正好等于OAM的总尺寸。

一旦DMA传输被启动，则其在完成前都不可以被打断，并且程序也没有任何硬件方法（例如中断、或者轮询某个寄存器的值是否改变）来检测DMA传输是否完成。在实践中，由于DMA的时间开销是固定的，大部分程序都是简单等待160个机器周期后就认为DMA已经完成，并开始执行接下来的操作。

0xFF48, 0xFF49 - OBP0, OBP1

这两个寄存器用于存储精灵的调色板数据。PPU内置了两个精灵调色板，每一个精灵对象可以使用对象属性标志位的位4来指定其使用调色板0还是调色板1。精灵的调色板着色规则与背景/窗口调色板一致，但是色值00固定为透明色，并在混色时不予考虑，因此精灵调色板最低的2位数据会被忽略。

实现DMA传输

在了解了DMA的原理以后，首先就让我们开始实现DMA传输功能吧。首先我们需要在PPU中加入记录DMA相关状态的变量：

```
/*...*/
// PPU internal state.

bool dma_active;
u8 dma_offset;
u8 dma_start_delay;

//! The number of cycles used for this scan line.
u32 line_cycles;
/*...*/
```

然后在PPU::init中初始化这些变量：

```
void PPU::init()
{
    /*...*/
    set_mode(PPUMode::oam_scan);
    dma_active = false;
    dma_offset = 0;
    dma_start_delay = 0;
    line_cycles = 0;
    /*...*/
}
```

其中，dma_active用于表示当前是否有一个正在进行中的DMA传输，dma_offset保存当前tick需要传输的值的下标（从0到160），dma_start_delay用于在DMA真正开始前延迟若干个机器周期，因为我们需要在写入0xFF46后的下一个机器周期中才开始真正的数据传输。

为了存储OAM数据，我们需要在Emulator类中加入OAM内存区域：

```
byte_t vram[8_kb];
byte_t wram[8_kb];
byte_t oam[160];
byte_t hram[128];
```

然后在Emulator::init中初始化内存数据：

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
```

```
    /*...*/
    memzero(wram, 8_kb);
    memzero(vram, 8_kb);
    memzero(oam, 160);
    memzero(hram, 128);
    int_flags = 0;
    /*...*/
}
```

最后修改Emulator::bus_read和Emulator::bus_write，加入OAM区域的读写支持：

```
u8 Emulator::bus_read(u16 addr)
{
    /*...*/
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        return wram[addr - 0xC000];
    }
    if(addr >= 0xFE00 && addr <= 0xFE9F)
    {
        return oam[addr - 0xFE00];
    }
    if(addr >= 0xFF01 && addr <= 0xFF02)
    {
        return serial.bus_read(addr);
    }
    /*...*/
}
void Emulator::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr <= 0xDFFF)
    {
        // Working RAM.
        wram[addr - 0xC000] = data;
        return;
    }
    if(addr >= 0xFE00 && addr <= 0xFE9F)
    {
        oam[addr - 0xFE00] = data;
        return;
    }
    if(addr >= 0xFF01 && addr <= 0xFF02)
    {
        serial.bus_write(addr, data);
        return;
    }
    /*...*/
}
```

接着，我们需要修改PPU::bus_write，在程序写入0xFF46地址时启动DMA传输：

```
void PPU::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr == 0xFF44) return; // read only.
    if(addr == 0xFF46)
    {
        // Enable DMA transfer.
        dma_active = true;
        dma_offset = 0;
        dma_start_delay = 1;
    }
    ((u8*)(&lcdc))[addr - 0xFF40] = data;
}
```

我们在PPU类中定义一个新的函数tick_dma来表示DMA传输在每个机器周期中的更新：

```
void tick_dma(Emulator* emu);
```

然后在PPU.cpp中实现该函数：

```
void PPU::tick_dma(Emulator* emu)
{
```

```
    if(!dma_active) return;
    if(dma_start_delay)
    {
        --dma_start_delay;
        return;
    }
    emu->oam[dma_offset] = emu->bus_read((((u16)dma) * 0x100) + dma_offset);
    ++dma_offset;
    dma_active = dma_offset < 0xA0;
}
```

tick_dma的代码非常直观：首先，整个逻辑只有在dma_active为true的时候才会运行，而当0xFF46被写入时，我们会将dma_active设置为true；其次，当dma_active为true，但是dma_start_delay不为0时，我们会将dma_start_delay减一，并在该机器周期中不执行任何操作，从而实现等待N个时间周期才开始传输数据的逻辑；最后，当dma_start_delay为0时，我们会执行实际的数据传输。数据传输在每个机器周期执行一次，传输一个字节的数据，而要传输的字节使用dma_offset变量表示，并且dma_offset会在每个机器周期增加1。当dma_offset超过160以后，我们通过将dma_active设置为false来自动停止DMA传输。

最后，我们将tick_dma接入PPU::tick中：

```
void PPU::tick(Emulator* emu)
{
    if((emu->clock_cycles % 4) == 0)
    {
        tick_dma(emu);
    }
    if(!enabled()) return;
    /*...*/
}
```

实现精灵绘制

在正确实现DMA以后，程序就能将精灵的属性加载到OAM区域，供PPU绘制精灵了。根据上一章的描述可知，精灵的绘制和背景/窗口的绘制同步进行，且逐行进行。在每一行开始绘制的OAM_SCAN模式中，PPU会从头到尾扫描OAM区域中的40个精灵对象，确定与当前扫描行相交的最多10个精灵对象，并将它们按照X轴坐标从小到大的顺序排序保存在一个内部缓存中。对于X轴坐标相等的精灵对象，则会按照其在OAM中出现的顺序排序，从而确定一个整体的绘制顺序（X轴坐标越小，则精灵对象越先绘制）。而在DRAWING模式中，PPU会在每一次处理背景/窗口图块时，同步处理与这些背景/窗口图块有重合的精灵图块，并将对应的像素随着背景/窗口像素1:1提交到精灵像素队列中。在一次处理背景/窗口图块时，PPU最多可以处理与背景/窗口图块重叠，且互相重叠的最多三个精灵图块，并绘制优先级最高的，且色值非0的精灵图块。

首先让我们先在PPU.hpp中定义OAM中每个对象的属性结构体：

```
struct OAMEntry
{
    //! The Y position of the sprite plus 16.
    u8 y;
    //! The X position of the sprite plus 8.
    u8 x;
    //! The tile index that stores the sprite.
    u8 tile;
    //! Attribute flags.
    u8 flags;

    //! 0 : OBP0, 1 : OBP1.
    u8 dmg_palette() const { return (flags >> 4) & 0x01; }
    bool x_flip() const { return bit_test(&flags, 5); }
    bool y_flip() const { return bit_test(&flags, 6); }
    bool priority() const { return bit_test(&flags, 7); }
};
```

在结构体中，我们定义了一些辅助函数，以判断标志位的每个位的状态。然后我们需要定义用于表示精灵队列中像素的结构体：

```
struct ObjectPixel
{
    //! The color index.
    u8 color;
    //! The palette used for this pixel.
    u8 palette;
    //! Holds flag 7 of the OAM entry.
    bool bg_priority;
```

```
};
```

相比背景/窗口像素，精灵像素需要额外携带一个优先级信息，以确定该精灵是否在背景的上层被渲染。接着我们在PPU类中添加精灵渲染相关的变量：

```
//! The FIFO queue for objects (sprites).
RingDeque<ObjectPixel> obj_queue;
//! The loaded sprite data during OAM scan stage, sorted by their X position.
Vector<OAMEntry> sprites;
//! The sprites used in the current fetch.
OAMEntry fetched_sprites[3];
u8 num_fetched_sprites;
//! The fetched sprite data in LCDFetchState::data0 and LCDFetchState::data1 st
u8 sprite_fetched_data[6];
```

各个变量的用处如下：

- obj_queue与之前的bgw_queue类似，用于将精灵像素提交给LCD驱动。
- sprites数组用于存储在OAM_SCAN阶段扫描到的，与当前扫描行相交的精灵对象信息。
- fetched_sprites和num_fetched_sprites用于在DRAWING阶段存储与当前背景/窗口图块的像素相交的最多3个精灵对象信息。
- sprite_fetched_data用于存储在fetcher的步骤2（data0）、步骤3（data1）阶段获取的精灵像素信息。由于我们最多可能需要同时处理3个精灵对象，因此需要存储最多2x3字节的像素信息。

然后在PPU中添加一些精灵绘制相关的辅助函数：

```
bool obj_enable() const { return bit_test(&lcdc, 1); }
u8 obj_height() const
{
    return bit_test(&lcdc, 2) ? 16 : 8;
}
```

这两个函数的含义可以参考上一章对LCDC寄存器的说明，此处不再赘述。

扫描精灵对象

在PPU绘制每一行像素前，我们需要在OAM_SCAN阶段扫描所有的精灵对象，并找出与当前扫描行相交的对象。修改PPU::tick_oam_scan如下：

```
void PPU::tick_oam_scan(Emulator* emu)
{
    if(line_cycles >= 80)
    {
        set_mode(PPUMode::drawing);
        fetch_window = false;
        fetch_state = PPUFetchState::tile;
        fetch_x = 0;
        push_x = 0;
        draw_x = 0;
    }
    // Can be any tick between 0 and 79.
    // The real PPU finishes OAM scanning in 80 cycles, but we can do it in one
    if(line_cycles == 1)
    {
        sprites.clear();
        sprites.reserve(10);
        u8 sprite_height = obj_height();
        // Scan all 40 entries.
        for(u8 i = 0; i < 40; ++i)
        {
            if(sprites.size() >= 10)
            {
                // We can hold at most 10 sprites per line.
                break;
            }
            OAMEntry* entry = (OAMEntry*)(emu->oam) + i;
            // Check if this sprite is in this scanline.
            if(entry->y <= ly + 16 && entry->y + sprite_height > ly + 16)
            {
                auto iter = sprites.begin();
                while(iter != sprites.end())
                {
                    if(iter->x > entry->x) break;
```

```
                    ++iter;
                }
                sprites.insert(iter, *entry);
            }
        }
    }
}
```

在实际的PPU工作中，OAM_SCAN阶段共消耗80个时钟周期，并且是随着时间的流逝逐步进行，但是出于便于实现的考虑，我们可以选择其中任意一个时钟周期（例如上面代码中用的是1），并在一个时钟周期中完成扫描的整个流程。由于游戏程序不会在OAM_SCAN阶段修改OAM中保存的值（OAM_SCAN阶段程序无法访问OAM区域，使用DMA传输修改会导致PPU直接不绘制该扫描行的精灵像素，因此没有游戏这么做），因此我们可以认为该简化不会对最终的模拟结果产生影响。

在执行OAM_SCAN的实际逻辑时，上述代码从头到尾扫描每个OAM中的精灵对象，并判断精灵对象的Y坐标范围是否与当前扫描行（LY）相交。当精灵对象的Y坐标范围与LY相交时，我们需要按照精灵在X轴出现的顺序来将其插入sprites数组中。精灵在sprites数组中出现的顺序决定了精灵绘制的顺序，当精灵重叠时，PPU使用以下规则判断精灵的绘制顺序：

1. 当精灵的X坐标不相等时，X坐标较小的精灵先绘制，X坐标较大的精灵后绘制。
2. 当精灵的X坐标相等时，在OAM中索引较小（位置较前）的精灵先绘制，索引较大的精灵后绘制。

反应在上述插入代码上，就是我们需要将新的精灵插入到**第一个X坐标比它大的精灵的前面**，或者如果没有X坐标更大的精灵，就插入到数组的尾部，从而完成精灵的排序。

由于PPU最多只能支持一行绘制10个精灵，因此在遍历精灵对象时，我们会检查当前sprites的数组长度是否大于等于10，并在满足条件时直接不再继续搜索。

获取精灵对象图块

在将与当前行相交的精灵对象存储在sprites之后，我们需要修改fetcher，以便其可以处理精灵图块，并在产生背景/窗口像素时同步产生精灵像素。首先我们需要修改PPU::fetcher_get_tile函数，添加获取精灵图块的逻辑代码：

```
void PPU::fetcher_get_tile(Emulator* emu)
{
    if(bg_window_enable())
    {
        if(fetch_window)
        {
            fetcher_get_window_tile(emu);
        }
        else
        {
            fetcher_get_background_tile(emu);
        }
    }
    else
    {
        tile_x_begin = fetch_x;
    }
    if(obj_enable())
    {
        fetcher_get_sprite_tile(emu);
    }
    fetch_state = PPUFetchState::data0;
    fetch_x += 8;
}
```

我们使用单独的fetcher_get_sprite_tile函数来获取精灵像素。由于精灵像素的获取需要使用tile_x_begin来确定当前背景/窗口图块的屏幕X坐标范围，因此即使我们关闭了背景/窗口渲染，也需要及时更新tile_x_begin变量，来保证精灵像素能够被正确渲染。接着，我们在PPU类中声明fetcher_get_sprite_tile函数：

```
void fetcher_get_sprite_tile(Emulator* emu);
```

然后在PPU.cpp中实现：

```
void PPU::fetcher_get_sprite_tile(Emulator* emu)
{
    num_fetched_sprites = 0;
```

```
    // Load this sprite tile.
    for(u8 i = 0; i < (u8)sprites.size(); ++i)
    {
        i32 sp_x = (i32)sprites[i].x - 8;
        // If the first or last pixel of the sprite row falls in this fetch
        if(((sp_x >= tile_x_begin) && (sp_x < (tile_x_begin + 8))) ||
            ((sp_x + 7 >= tile_x_begin) && (sp_x + 7 < (tile_x_begin + 8))))
        {
            fetched_sprites[num_fetched_sprites] = sprites[i];
            ++num_fetched_sprites;
        }
        // We can handle at most 3 sprites in one fetch.
        if(num_fetched_sprites >= 3)
        {
            break;
        }
    }
}
```

在fetcher_get_sprite_tile时，我们会扫描OAM_SCAN阶段存储在sprites中的每一个精灵对象，通过判断精灵对象的X坐标范围是否与背景图块的X坐标范围有重合，来判断我们是否需要在加载当前背景/窗口图块时同时加载精灵图块。由于在一次图块处理时，我们最多只能同时加载3个精灵图块，因此代码会判断当前需要加载的图块是否超过3个，并在条件满足时停止继续扫描。

获取精灵对象像素数据

修改fetcher_get_data如下：

```
void PPU::fetcher_get_data(Emulator* emu, u8 data_index)
{
    if(bg_window_enable())
    {
        bgw_fetched_data[data_index] = emu->bus_read(bgw_data_area() + bgw_data
    }
    if(obj_enable())
    {
        fetcher_get_sprite_data(emu, data_index);
    }
    if(data_index == 0) fetch_state = PPUFetchState::data1;
    else fetch_state = PPUFetchState::idle;
}
```

我们使用单独的fetcher_get_sprite_data函数来获取精灵像素，在PPU类中声明如下：

```
void fetcher_get_sprite_data(Emulator* emu, u8 data_index);
```

接着在PPU.cpp中实现：

```
void PPU::fetcher_get_sprite_data(Emulator* emu, u8 data_index)
{
    u8 sprite_height = obj_height();
    for(u8 i = 0; i < num_fetched_sprites; ++i)
    {
        u8 ty = (u8)(ly + 16 - fetched_sprites[i].y);
        if(fetched_sprites[i].y_flip())
        {
            // Flip y in tile.
            ty = (sprite_height - 1) - ty;
        }
        u8 tile = fetched_sprites[i].tile;
        if(sprite_height == 16)
        {
            tile &= 0xFE; // Clear the last 1 bit if in double tile mode.
        }
        sprite_fetched_data[(i * 2) + data_index] = emu->bus_read(0x8000 + (ti
    }
}
```

在获取精灵图块的数据时，我们需要通过判断当前扫描行与精灵的Y坐标的相对关系，来确定要获取的行在图块数据中的偏移值。同时，由于精灵具有垂直翻转（flip Y）功能，我们需要在代码中对该标志位进行判断，并在垂直翻转标志位为1时翻转需要获取的行在图块中的Y坐标值。在确定了需要读取的图块和对应的行后，我们通过与读取背景/窗口图块类似的代码，通过bus_read函数读取图块对应行的数据。在图块尺寸为8x16的模式下，由于图块的数据保存在内存中相邻的

两个图块中，且上方（Y坐标更小）的图块先存储，因此我们只需要直接通过Y坐标值偏移，就可以在直接读取相邻的第二个图块的数据。

将精灵像素压入队列

修改fetcher_push_pixels函数如下：

```
void PPU::fetcher_push_pixels()
{
    bool pushed = false;
    if(bgw_queue.size() < 8)
    {
        u8 push_begin = push_x;
        fetcher_push_bgw_pixels();
        u8 push_end = push_x;
        fetcher_push_sprite_pixels(push_begin, push_end);
        pushed = true;
    }
    if(pushed)
    {
        fetch_state = PPUFetchState::tile;
    }
}
```

我们使用单独的fetcher_push_sprite_pixels函数来将精灵对象压入队列。由于精灵像素和背景/窗口像素的提交必须完全对应，因此我们使用push_begin和push_end变量来记录在fetcher_push_bgw_pixels中提交的背景/窗口像素的屏幕空间X坐标范围， 并作为参数输入fetcher_push_sprite_pixels中。虽然背景/窗口像素队列和精灵像素队列是两个独立的队列，但是由于这两个队列的像素数量严格一致，因此我们只需要对背景/窗口像素队列中像素的数量进行判断，就可以确定当前是否需要压入新的像素。在实际的硬件上，fetcher和LCD驱动也是通过仅判断背景/窗口队列中像素的数量来确定当前队列的负载情况的。

接着我们在PPU类中添加fetcher_push_sprite_pixels的声明：

```
void fetcher_push_sprite_pixels(u8 push_begin, u8 push_end);
```

然后在PPU.cpp中实现该函数：

```
void PPU::fetcher_push_sprite_pixels(u8 push_begin, u8 push_end)
{
    for(u32 i = push_begin; i < push_end; ++i)
    {
        ObjectPixel pixel;
        // The default value is one transparent color.
        pixel.color = 0;
        pixel.palette = 0;
        pixel.bg_priority = true;
        if(obj_enable())
        {
            for(u8 s = 0; s < num_fetched_sprites; ++s)
            {
                i32 spx = (i32)fetched_sprites[s].x - 8;
                i32 offset = (i32)(i) - spx;
                if(offset < 0 || offset > 7)
                {
                    // This sprite does not cover this pixel.
                    continue;
                }
                u8 b1 = sprite_fetched_data[s * 2];
                u8 b2 = sprite_fetched_data[s * 2 + 1];
                u8 b = 7 - (u8)offset;
                if(fetched_sprites[s].x_flip())
                {
                   b = (u8)offset;
                }
                u8 lo = (!!(b1 & (1 << b)));
                u8 hi = (!!(b2 & (1 << b))) << 1;
                u8 color = hi | lo;
                if(color == 0)
                {
                    // If this sprite is transparent, we look for the next spri
                    continue;
                }
                // Use this pixel.
                pixel.color = color;
                pixel.palette = fetched_sprites[s].dmg_palette() ? obp1 : obp0;
```

```
                pixel.bg_priority = fetched_sprites[s].priority();
                break;
            }
        }
        obj_queue.push_back(pixel);
    }
}
```

该函数有以下几个实现要点：

1. 该函数根据push_begin和push_end来执行push_end-push_begin次循环，在每一次循环中，该函数一定会将一个像素压入精灵像素队列，以保证两个队列的像素数量一致。
2. 如果obj_enable()返回false，即PPU的精灵对象绘制没有打开，则该函数默认会压入一个空像素。空像素的色值为0，意味着该像素会被LCD驱动认为是透明像素并被丢弃。
3. 在obj_enable()返回true的情况下，函数会依次查询在阶段1、2、3中读取的最多3个精灵图块像素行的数据，以确定最终呈现的精灵像素颜色。由于每一个精灵图块的X坐标都可以不一样，因此对于每一个屏幕像素，我们都需要首先确定该像素是否被当前的精灵图块覆盖，然后确定该精灵图块在该像素上的色值是否不为0。如果以上两个条件都符合，我们就会使用该图块在该像素的颜色来作为最终像素颜色，否则我们就会继续查找下一个精灵图块，并重复上述步骤。由于精灵对象具有水平翻转功能，在读取像素颜色时，我们需要考虑精灵对象的水平翻转标志位是否为1，并在标志位为1时将需要读取的像素位进行翻转，以读取正确的像素色值。如果所有的精灵图块都不影响本像素的颜色（不相交或者色值均为0），则我们会压入一个空像素。

绘制精灵像素

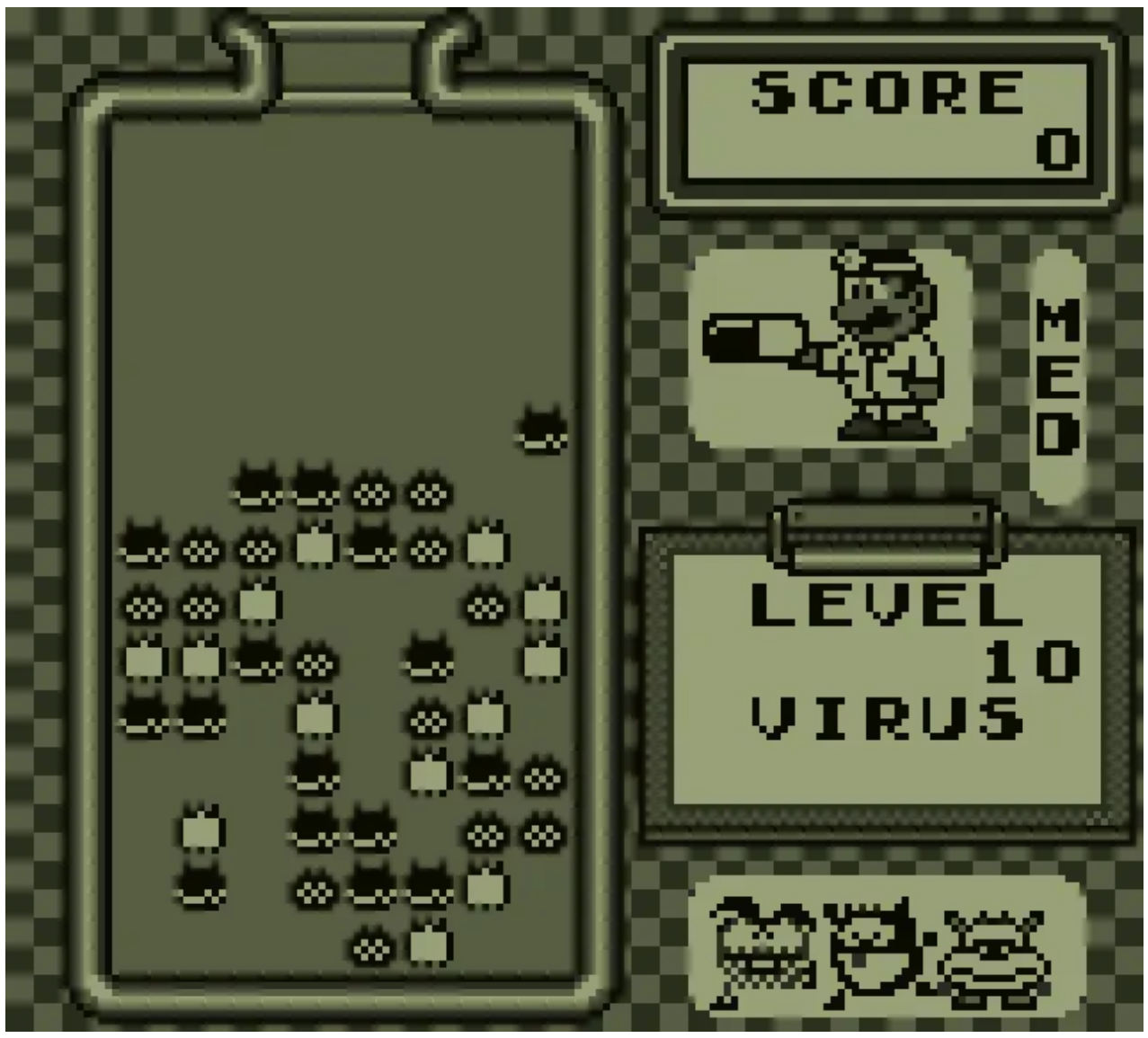
最后就是修改我们的LCD驱动代码，在绘制实际像素之前混合背景/窗口像素和精灵像素了。修改lcd_draw_pixel函数如下：

```
void PPU::lcd_draw_pixel()
{
    // The LCD driver is drived by BGW queue only, it works when at least 8 pix
    if(bgw_queue.size() >= 8)
    {
        if (draw_x >= PPU_XRES) return;
        BGWPixel bgw_pixel = bgw_queue.front();
        bgw_queue.pop_front();
        ObjectPixel obj_pixel = obj_queue.front();
        obj_queue.pop_front();
        // Calculate background color.
        u8 bg_color = apply_palette(bgw_pixel.color, bgw_pixel.palette);
        // Draw object if:
        // 1. Color index is not 0 (transparent) and:
        // 2. Background priority is not greater than object priority, or the l
        bool draw_obj = obj_pixel.color && (!obj_pixel.bg_priority || bg_color
        // Calculate obj color.
        u8 obj_color = apply_palette(obj_pixel.color, obj_pixel.palette & 0xFC)
        // Selects the final color.
        u8 color = draw_obj ? obj_color : bg_color;
        // Output pixel.
        switch(color)
        {
            case 0: set_pixel(draw_x, ly, 153, 161, 120, 255); break;
            case 1: set_pixel(draw_x, ly, 87, 93, 67, 255); break;
            case 2: set_pixel(draw_x, ly, 42, 46, 32, 255); break;
            case 3: set_pixel(draw_x, ly, 10, 10, 2, 255); break;
        }
        ++draw_x;
    }
}
```

该函数的实现有以下几个要点：

1. 该函数只有在背景/窗口队列中像素数量大于等于8时才起作用，该特性为GameBoy的LCD驱动的硬件特性，即LCD驱动只有在背景/窗口队列中像素数量大于等于8时才开始工作。该特性保证了在LCD驱动绘制像素时总能获取到足够数量的有效像素。在每一次LCD驱动工作时，其都会分别从背景/窗口队列和精灵队列中取出一个像素进行处理。
2. draw_obj代码判断对于当前像素，我们需要使用背景/窗口像素的颜色还是精灵像素的颜色。可以看到GameBoy的PPU并没有真正意义上的像素混合，其所有的像素混合操作都是像素选择操作，即要么完全使用A像素，要么完全使用B像素。根据文档可知，当精灵像素色值为0，或者其背景优先级标志位为1，且对应的背景像素最终颜色不为0时，我们需要绘制背景像素，否则就绘制精灵像素。
3. 最后，根据draw_obj的结果，我们选择背景/窗口像素或者精灵像素，并应用调色板算出最终颜色后进行显示。

编译并运行上述代码，并加载《马里奥医生》游戏卡带，在等待一段时间进入演示模式后，可以看到画面的所有元素都已经能够正常显示和更新了：



PPU单元测试

为了测试我们的PPU是否正确实现所有的特性，我们可以下载一个专门用于测试GameBoy PPU的卡带，地址如下：

https://github.com/mattcurrie/dmg-acid2
github.com/mattcurrie/dmg-acid2

在运行dmg-acid2.gb后，如果PPU的所有功能正确实现，则其展示的画面应当如下所示：

如果读者实现的PPU渲染结果与上图有差异，可以根据dmg-acid2的GitHub页面中对每个画面组件以及渲染错误原因的介绍来排查PPU实现上的问题，从而确保PPU正确实现。

以上就是本章节的全部内容了，至此我们就完成了PPU部分的所有功能的实现。在下一章中，我们将实现GameBoy的按键输入功能，并扩展卡带的读写逻辑，从而支持MBC1、MBC2、MBC3卡带，让我们的模拟器能够畅玩市面上大部分的GameBoy游戏ROM文件。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #10 按键输入、MBC1卡带
13 赞同 · 0 评论 文章

编辑于 2024-03-13 11:41 · IP 属地上海

游戏机模拟器 Game Boy（GB）

欢迎参与讨论

2 条评论 默认 最新

KeDouJia
02-23 · IP 属地湖北 回复 喜欢

w4ngzhen
大佬牛逼，大佬高产！
02-20 · IP 属地四川 回复 喜欢

文章被以下专栏收录

吉祥的游戏制作笔记
游戏制作中的技术、艺术和设计灵感记录

推荐阅读

横跨了几代人的经典！PSP安卓模拟器深度教程：模拟器系
PSP是一代经典掌机，那时数码产品还未普及，PSP不仅有掌机的游戏的功能，还承担了MP4，MP3和电子书的功能，甚至能在PSP上聊QQ，最终PSP的全球销量为7000多万台，其魅力可见一斑 该平台上
乌托邦游戏 发表于乌托邦游戏

永恒经典的索尼掌机！PSP模拟器深度教程PC篇：模拟器系
PSP是一代经典掌机，如今在手机、PC上已经能比较完美的模拟器了，相对于手机模拟的便捷性，因为PC主机的性能更强大，PC模拟可以带来更好的画质和运行帧数，一些原先在PSP模糊锯齿严重的游
乌托邦游戏 发表于乌托邦游戏

玩模拟器的掌机神器-RG
想要原汁原味玩各种老游戏当然是烧原装主机，上模拟屏。原装主机如果运气好，也许可以买到箱说全三码合一的二十多年的全新珍藏版，不过除了价格贵以外还很占地方，如果书房没有多余地方放第...
韦易笑

电脑上可以用手机模拟器，你知道吗？
Anonymous

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ···