从零开始实现GameBoy模拟器 #8 绘制背景和窗口

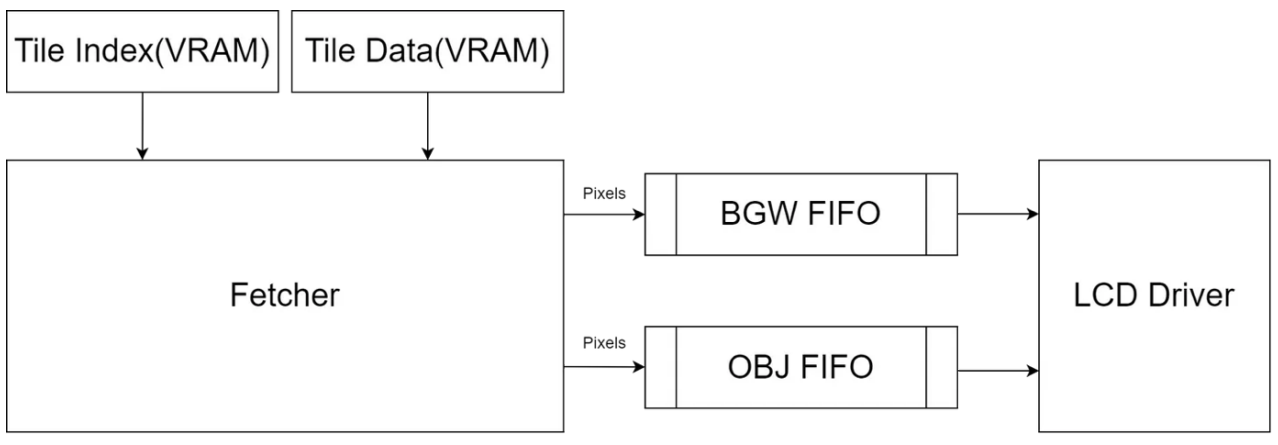
銀葉吉祥
浙江大学 软件工程硕士

欢迎来到从零开始实现GameBoy模拟器第八章。本章将完善PPU的绘制功能，以绘制背景和窗口的图块。至于精灵图块的绘制，我们将留到下一章完成。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-08，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。

绘制单元架构

当GameBoy PPU处于0x03 DRAWING模式时，其会进行实际的绘制操作。下图展示了GameBoy PPU的绘制单元架构：



绘制流程由多个组件合作完成，包括：用于获取图块数据并解码像素的fetcher、用于显示像素的LCD驱动，以及用于将像素从fetcher传递给LCD驱动的两个FIFO队列，分别用于传输背景/窗口像素和精灵对象像素。

Fetcher

fetcher并没有非常贴切的中文翻译，因此我们这里直接使用英文原名称呼。fetcher是GameBoy的PPU中用于绘制的硬件电路单元，当PPU处于绘制模式时，fetcher会从左往右生成屏幕每一行的像素，并且将这些像素发送到像素队列中，供LCD驱动读取。fetcher以图块为处理单位，当PPU进入绘制模式时，其会解析与当前扫描行最左侧像素相交的图块，然后解析该图块右侧的图块，循环往复，直到当前扫描行覆盖的所有图块都被处理完毕。fetcher中有一个内部寄存器（下文称之为FETCH_X），用于记录下一个需要获取的像素的X坐标，该坐标从0开始增加。由于每个图块的宽度为8，因此在每一次解析图块时，fetcher都会产生8个连续的像素并压入队列中，并将FETCH_X的值增加8。

在解析每个图块时，fetcher都会执行以下五个步骤：

1. 确定图块数据地址。
2. 读取图块数据的第一个字节。
3. 读取图块数据的第二个字节。
4. 等待。
5. 将像素压入渲染队列。

以上的五个步骤每个消耗2个时钟周期，因此也可以认为fetcher每2个时钟周期更新一次。在5个步骤中，1~4步在执行后都会立即跳转到下一个步骤，而第5步只有在成功压入像素以后才会重新跳转到第1步。在第5步中，fetcher将会检查BGW队列中像素的个数，如果像素个数小于8，则其可以成功将新的像素压入队列，否则就无限循环等待，直到队列中像素个数小于8，像素被成功压入。

> 事实上，在某些情况下（例如从绘制背景切换到绘制窗口时，以及需要绘制覆盖背景/窗口像素的精灵时），步骤5的耗时会增加，导致了DRAWING阶段的耗时在172~289个时钟周期不等。由于是否实现精确的绘制时间开销并不会对大部分游戏产生影响（游戏不允许在绘制阶段修改显存、OAM和寄存器的任何数据，因此游戏程序总是会等待PPU进入HBLANK模式以后才会进行数据更新），因此本文出于方便实现的考虑，简化了绘制的计时流程，没有实现完全精确的计时。读者可以参考这篇文章了解如何实现精确的绘制计时。

fetcher的内部电路可以同步处理背景/窗口图块和精灵图块，因此上述五个步骤均同时处理背景/窗口图块和精灵图块，并分别将像素压入背景/窗口队列和对象队列中。fetcher保证对于每一个压入背景/窗口队列的像素，总会有一个对应的精灵图块的像素同时被压入对象队列，因此LCD驱动总是可以一次性从两个队列中分别取出一个像素进行混合。

下面我们将针对上述的每一个步骤进行详细说明。

确定图块数据地址

在该阶段中，fetcher会根据LY、SCX、SCY、WX、WY等寄存器的值以及FETCH_X的值，确定下一个像素所在的背景或窗口图块，以及该图块在地图/窗口图块数据中的位置（以索引值0-255表示），然后再加上背景/窗口图块数据的起始地址，确定这一次处理的图块数据的地址。同时，fetcher也会计算并存储图块的第一个像素在屏幕上的X坐标，当SCX的值不为8的倍数时，每一行的第一个图块的部分像素的X坐标会变成负值，在这种情况下，fetcher会按照正常逻辑解析像素数据，但是会在第五步中将所有不在屏幕范围内的像素丢弃。

在确定背景/窗口图块像素在屏幕上的X坐标范围后，fetcher在该阶段中还会检查所有在OAM_SCAN中被标记的，与当前扫描行相交的精灵对象，并找出其中与当前绘制的背景图块在屏幕上重叠的精灵对象（最多3个）。对于每个找到的重叠精灵对象，fetcher都会根据精灵图块的起始地址和该对象在精灵图块序列中的位置确定对应的图块数据地址。

在该步骤结束时，fetcher会保存一个背景/窗口图块的数据地址，以及0至3个精灵图块的数据地址。

读取图块数据

在确定了图块数据地址以后，fetcher会开始从该地址中读取图块数据。从上一章对图块的介绍中可知，图块的每一行使用两个字节的数据表示，因此fetcher会在接下来的4个时钟周期中分别读取每个图块第一个字节和第二个字节的数据。根据上一步骤中解析的数据地址数量，fetcher在该阶段中会读取一个背景/窗口图块的数据，以及0~3个精灵图块的数据。

等待

该阶段fetcher不执行任何操作，其只是简单等待两个时钟周期，然后进入下一阶段。

将像素压入渲染队列

在该阶段中，fetcher会将读取到的图块数据翻译成实际的像素数据，并将其压入渲染队列。

当处理背景/窗口像素时，fetcher会读取阶段一中计算的图块的第一个像素在屏幕上的X坐标，进而求得图块的所有8个像素在屏幕上的X坐标。对于所有X坐标小于0的像素，fetcher会直接丢弃，但是对于X坐标大于等于160（屏幕X分辨率）的像素，其并不会丢弃，而是将它们按照正常逻辑压入队列中。由于LCD驱动从左到右将像素绘制到屏幕上，当其读取160个像素以后，便不会继续从队列中读取像素，队列中多余的像素将在下一次进入DRAWING模式时被自动清空。

由于窗口的X坐标也可以是任意在有效范围的值，在绘制背景时，可能会出现部分背景像素被窗口像素覆盖的情况。由于fetcher在一次操作中只能读取一个背景/窗口图块的数据，因此当fetcher在该阶段检测到被窗口覆盖的像素时，其会立即中断该像素及其之后像素的入队操作，并且将fetcher的FETCH_X值设置为窗口的第一个像素的X坐标。这样一来，在下一次fetcher处理图块时，其就会从窗口的第一个图块开始依次处理，完成窗口的绘制操作。

在处理背景/窗口像素的同时，fetcher也会在该阶段同时处理与这些背景/窗口像素重叠的精灵图块像素。对于任意一个背景像素，如果其没有与任何精灵像素重叠，则fetcher会生成一个空白像素（色值为00的透明像素），并压入像素队列中，因此背景像素和精灵像素队列中的元素个数是

1:1对应的关系。

LCD驱动

在fetcher将像素压入队列后，LCD驱动负责从队列中取出背景/窗口和精灵像素，根据两个像素的色值和调色板值混合，确定最终的像素颜色，并显示到屏幕上。LCD驱动在每一个时钟周期可以处理一个像素，像素从左到右被取出和显示。当背景/窗口队列中的像素数量小于8时，LCD驱动将会暂停绘制，直到队列中有足够多的像素可以被绘制（>=8）。当LCD驱动绘制完一整行的160个像素时，其会向PPU发送一个绘制结束指令，PPU在接收到该指令以后会结束DRAWING模式，并进入HBLANK模式，从而完成一行像素的绘制。

0xFF42, 0xFF43 - SCY, SCX

SCY（scroll Y）和SCX（scroll X）寄存器标识屏幕左上角像素在地图上的位置，从而控制地图在屏幕中显示的区域。当SCX的值不为8的倍数时，屏幕上每一行的第一个图块的部分像素将会被屏幕左侧边缘遮挡。在这种情况下，我们需要在fetcher的阶段5中进行判断，并丢弃这些位于屏幕之外的像素。

0xFF4A, 0xFF4B - WX, WY

WX（window X），WY（window Y）寄存器标识窗口左上角像素在屏幕上的位置，这两个寄存器的值只有当窗口启用（LCDC.0 == 1 && LCDC.5 == 1）时才生效。WX寄存器存储的是窗口左上角像素X坐标加7的值，因此当WX==7，WY==0时，窗口的左上角像素将位于屏幕的左上角，因此窗口将完全覆盖背景像素。

在绘制窗口时，fetcher将会使用单独的变量（window line counter）记录当前绘制的是窗口的第几行像素，该变量从0开始，并且在每一次LY增加的时候，如果下一扫描行与窗口相交（LY>=WY），则会与LY同步加1。该特性导致程序即使在窗口绘制到一半时修改了WY的值，只要下一扫描行仍然与窗口相交，则窗口的绘制结果会类似于没有修改WY的值，从而避免窗口在垂直方向上的移动导致窗口画面出现错位。

0xFF47 - BGP

BGP（background palette）寄存器存储了用于绘制窗口和背景的调色板数据。该寄存器具有8位，每2位用于控制一个色值的实际显示颜色，具体来说：

- BGP.0和BGP.1控制色值0的实际显示颜色。
- BGP.2和BGP.3控制色值1的实际显示颜色。
- BGP.4和BGP.5控制色值2的实际显示颜色。
- BGP.6和BGP.7控制色值3的实际显示颜色。

每一个色值的显示颜色都使用2个bit来指定，因此一共可以表示四种色值颜色，分别为：

- 0：白色
- 1：浅灰色
- 2：深灰色
- 3：黑色

由于初代GameBoy使用的是绿色面板，黑色像素的LCD屏幕，因此颜色0表示像素不显示，此时画面为绿色；颜色3表示像素完全显示，此时画面为黑色。而当CGB运行单色GameBoy卡带时，会将0显示为白色，3显示为黑色。

为了方便起见，我们可以在PPU.cpp中定义一个函数来执行调色板颜色的映射：

```
inline u8 apply_palette(u8 color, u8 palette)
{
    switch(color)
    {
        case 0: color = palette & 0x03; break;
        case 1: color = ((palette >> 2) & 0x03); break;
        case 2: color = ((palette >> 4) & 0x03); break;
        case 3: color = ((palette >> 6) & 0x03); break;
        default: lupanic(); break;
    }
    return color;
}
```

实现PPU

在了解了背景和窗口绘制原理后，就让我们来编写代码实现窗口和背景绘制逻辑吧！首先我们在PPU.hpp中添加一些新的类型：

```
enum class PPUFetchState : u8
{
    tile,
    data0,
    data1,
    idle,
    push
};
struct BGWPixel
{
    //! The color index.
    u8 color;
    //! The palette used for this pixel.
    u8 palette;
};
```

PPUFetchState用于表示当前fetcher的状态，与上文提到的五个阶段一一对应。BGWPixel表示一个用于提交到背景/窗口队列的像素，这个像素的信息包括像素的色值和其使用的调色板参数，LCD驱动会使用这两个参数求得最终需要显示的颜色。

然后我们在PPU类型中添加以下变量：

```
//! The FIFO queue for background/window pixels.
RingDeque<BGWPixel> bgw_queue;
//! true when we are fetching window tiles.
//! false when we are fetching background tiles.
bool fetch_window;
//! The window line counter.
u8 window_line;
//! The current fetch state.
PPUFetchState fetch_state;
//! The X position of the next pixel to fetch in screen coordinates.
//! Advanced by 8, since we fetch 8 pixels at one time.
u8 fetch_x;
//! The fetched background/window tile data offset from PPUFetchState::tile ste
//! Add this with bgw_data_area() to get the final address.
u16 bgw_data_addr_offset;
//! The x position of the first pixel in fetched tile, in screen coordinates.
//! May be negative if scroll_x is not times of 8.
i16 tile_x_begin;
//! The fetched background/window data in PPUFetchState::data0 and PPUFetchStat
u8 bgw_fetched_data[2];
//! The X position of the next pixel to push to the bgw FIFO.
u8 push_x;
//! The X position of the next pixel to draw to the back buffer in screen coord
//! If draw_x >= PPU_XRES then all pixels are drawn, so we can start HBLANK.
u8 draw_x;
```

各个变量的解释如下：

- bgw_queue表示用于存放背景/窗口像素的队列。RingDeque是一个双端队列容器，其使用ring buffer来保存元素，并支持在队列两端压入和弹出元素，适合用于作为FIFO缓存队列使用。
- fetch_window用于判断当前需要处理的是窗口像素还是背景像素。在默认情况下，fetcher将处理背景像素，直到在阶段5中遇到一个被窗口覆盖的像素。当fetcher检查到像素被窗口覆盖时，其会将模式切换为处理窗口像素，并且将fetch_x重置为窗口的第一个像素。
- window_line用于记录当前正在绘制的窗口行，从0开始依次增加。
- fetch_state是fetcher当前的状态。与PPUState类似，在每一次fetcher更新时，其会根据当前状态执行对应操作，并更新fetch_state至下一状态。
- fetch_x用于记录fetcher下一个需要处理的像素在屏幕上的位置，fetcher在步骤一中根据该值确定下一个要读取的图块的索引值。
- bgw_data_addr_offset用于记录步骤一中获取的背景/窗口像素行的数据地址相对于背景/窗口像素区域起始地址的偏移。
- tile_x_begin用于记录步骤一中获取的图块的第一个像素的屏幕X坐标，由于确定图块覆盖的像素范围。
- bgw_fetched_data用于记录步骤二和步骤三中获取的背景图块数据。
- push_x用于记录fetcher下一个将要压入队列的像素的屏幕X坐标。
- draw_x用于记录LCD驱动下一个将要从队列中取出并绘制的像素的屏幕X坐标。

同时，我们还需要在PPU类中添加一些辅助函数，以便后续编写代码时更加直观：

```
bool bg_window_enable() const { return bit_test(&lcdc, 0); };
bool window_enable() const { return bit_test(&lcdc, 5); }
u16 bg_map_area() const
{
    return bit_test(&lcdc, 3) ? 0x9C00 : 0x9800;
```

```
}
u16 window_map_area() const
{
    return bit_test(&lcdc, 6) ? 0x9C00 : 0x9800;
}
u32 bgw_data_area() const
{
    return bit_test(&lcdc, 4) ? 0x8000 : 0x8800;
}
bool window_visible() const
{
    return window_enable() && wx <= 166 && wy < PPU_YRES;
}
bool is_pixel_window(u8 screen_x, u8 screen_y) const
{
    return window_visible() && (screen_x + 7 >= wx) && (screen_y >= wy);
}
```

这些辅助函数的含义可以参考上一章对于LCDC函数的解释。在进入DRAWING模式时，我们需要对其中部分变量进行初始化操作，如下所示：

```
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;
        bgw_queue.clear();
    }
}
```

同时，在修改LY的值的时候，我们需要判断当前扫描行是否与窗口相交，并在相交的时候增加window_line的值，如下所示：

```
void PPU::increase_ly(Emulator* emu)
{
    if(window_visible() && ly >= wy &&
       (u16)ly < (u16)(wy + PPU_YRES))
    {
        ++window_line;
    }
    ++ly;
    if(ly == lyc)
    {
        set_lyc_flag();
        if(lyc_int_enabled())
        {
            emu->int_flags |= INT_LCD_STAT;
        }
    }
    else
    {
        reset_lyc_flag();
    }
}
```

同样，当我们进入完成一帧的渲染的时候，我们需要同时把LY和window_line计数器归零：

```
void PPU::tick_vblank(Emulator* emu)
{
    if(line_cycles >= PPU_CYCLES_PER_LINE)
    {
        increase_ly(emu);
        if(ly >= PPU_LINES_PER_FRAME)
        {
            // move to next frame.
            set_mode(PPUMode::oam_scan);
            ly = 0;
            window_line = 0;
            if(oam_int_enabled())
            {
                emu->int_flags |= INT_LCD_STAT;
```

```
            }
        }
        line_cycles = 0;
    }
}
```

然后我们修改PPU::tick_drawing，添加绘制部分的主要逻辑：

```
void PPU::tick_drawing(Emulator* emu)
{
    // The fetcher is ticked once per 2 cycles.
    if((line_cycles % 2) == 0)
    {
        switch(fetch_state)
        {
            case PPUFetchState::tile:
                fetcher_get_tile(emu); break;
            case PPUFetchState::data0:
                fetcher_get_data(emu, 0); break;
            case PPUFetchState::data1:
                fetcher_get_data(emu, 1); break;
            case PPUFetchState::idle:
                fetch_state = PPUFetchState::push; break;
            case PPUFetchState::push:
                fetcher_push_pixels(); break;
            default: lupanic(); break;
        }
        if(draw_x >= PPU_XRES)
        {
            luassert(line_cycles >= 252 && line_cycles <= 369);
            set_mode(PPUMode::hblank);
            if(hblank_int_enabled())
            {
                emu->int_flags |= INT_LCD_STAT;
            }
        }
    }
    // LCD driver is ticked once per cycle.
    lcd_draw_pixel();
}
```

我们添加了四个单独的函数用于执行fetcher的四个不同阶段的任务，这些函数都会每2个时钟周期执行一次，同时lcd_draw_pixel用于模拟LCD驱动绘制像素的过程，该函数每个时钟周期执行一次。现在，在PPU中添加这些函数的声明：

```
void fetcher_get_tile(Emulator* emu);
void fetcher_get_data(Emulator* emu, u8 data_index);
void fetcher_push_pixels();
void lcd_draw_pixel();
```

然后在PPU.cpp中实现以上函数：

```
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);
        }
    }
    fetch_state = PPUFetchState::data0;
    fetch_x += 8;
}
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(data_index == 0) fetch_state = PPUFetchState::data1;
    else fetch_state = PPUFetchState::idle;
```

```
}
void PPU::fetcher_push_pixels()
{
    bool pushed = false;
    if(bgw_queue.size() < 8)
    {
        fetcher_push_bgw_pixels();
        pushed = true;
    }
    if(pushed)
    {
        fetch_state = PPUFetchState::tile;
    }
}
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();
        // Calculate background color.
        u8 bg_color = apply_palette(bgw_pixel.color, bgw_pixel.palette);
        // Output pixel.
        switch(bg_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;
    }
}
```

在fetcher_get_tile函数中，当LCDC.0为1时，我们需要获取背景或者窗口图块的数据地址。在这里我们通过fetch_window变量来判断当前需要处理背景图块还是窗口图块，该变量在每一行刚开始绘制的时候设置为false，并在fetcher遇到第一个窗口像素时设置为true，并一直保持到这一行绘制结果。接着我们调用fetcher_get_window_tile和fetcher_get_background_tile两个子函数来分别处理背景图块和窗口图块的地址获取，这两个函数会在下面专门的章节中讲解。最后，我们会将fetch_x的值加8，因为fetcher一次可以处理一个图块的8个像素，然后将fetcher的模式设置为data0。

在fetcher_get_data中，我们根据bgw_data_addr_offset变量中存储的地址加上背景/窗口图块数据的起始地址来获得我们要获取的实际数据地址，然后读取地址中的数据，保存在bgw_fetched_data变量中。fetcher_get_data会被调用两次，分别用于获取第一个字节和第二个字节的数据。在数据读取完毕以后，fetcher_get_data会根据当前读取的是第一个字节还是第二个字节，分别将fetcher的模式设置为data1和idle。

在fetcher_push_pixels中，我们需要在像素队列有足够空间（像素数量小于8）的情况下解码获取的像素数据，并将像素压入队列中。我们使用单独的函数fetcher_push_bgw_pixels来处理背景像素的解码和入队操作，该函数会在下面专门的章节中讲解。只有当像素被成功压入队列中，fetcher_push_pixels才会将状态重新设为tile，否则不做修改，从而让下一次fetcher更新时仍然执行fetcher_push_pixels，重复尝试提交像素。

在lcd_draw_pixel中，我们会从背景队列中取出一个像素，根据像素的色值和调色板颜色确定像素的最终颜色，然后根据最终颜色设置屏幕上对应像素位置的颜色。在这里我们使用了4个不同明度的绿色，从而模拟原版GameBoy的显示效果，读者也可以根据喜好使用其它不同的颜色。

在绘制每一行的像素时，我们使用draw_x变量来追踪当前绘制的像素的X值，该变量会在每一行绘制开始时设置为0，并随着lcd_draw_pixel的调用而逐渐增加。在每一次绘制像素前，我们都会判断draw_x的范围，确保我们不会绘制超出屏幕边缘的像素。

接着我们需要定义set_pixel函数以及用于存储最终颜色的位图对象。在PPU类中添加以下变量和函数：

```
u8 pixels[PPU_XRES * PPU_YRES * 4 * 2];
u8 current_back_buffer;
void set_pixel(i32 x, i32 y, u8 r, u8 g, u8 b, u8 a)
{
    luassert(x >= 0 || x < PPU_XRES);
    luassert(y >= 0 || y < PPU_YRES);
    u8* dst = pixels + current_back_buffer * PPU_XRES * PPU_YRES * 4;
    u8* pixel = (u8*)pixel_offset(dst, (usize)x, (usize)y, 4, 4 * PPU_XRES);
    pixel[0] = r;
    pixel[1] = g;
```

```
    pixel[2] = b;
    pixel[3] = a;
}
```

为了避免画面撕裂，在这里我们使用了双后台缓存交换的思路来绘制和显示画面。在定义位图内存的时候，我们申请了2倍于屏幕分辨率的内存，并划分成了两个位图。current_back_buffer存储了当前正在进行绘制的位图，而另一张位图则用于提供给RHI显示当前的游戏画面；当游戏绘制完毕一帧画面时，就通过current_back_buffer = (current_back_buffer + 1) % 2将两张位图交换，以展示新绘制完的画面，以此类推。

然后，在PPU::init中，我们需要将位图数据和current_back_buffer清零：

```
void PPU::init()
{
    /*...*/

    line_cycles = 0;
    memzero(pixels, sizeof(pixels));
    current_back_buffer = 0;
}
```

在PPU::tick_hblank中，我们需要在进入VBLANK模式的时候交换用于绘制和展示的位图：

```
void PPU::tick_hblank(Emulator* emu)
{
    if(line_cycles >= PPU_CYCLES_PER_LINE)
    {
        increase_ly(emu);
        if(ly >= PPU_YRES)
        {
            set_mode(PPUMode::vblank);
            emu->int_flags |= INT_VBLANK;
            if(vblank_int_enabled())
            {
                emu->int_flags |= INT_LCD_STAT;
            }
            current_back_buffer = (current_back_buffer + 1) % 2;
        }
        else
        {
            /*...*/
        }
        line_cycles = 0;
    }
}
```

实现背景绘制

接下来，我们首先实现背景像素的绘制。在PPU类中声明以下函数：

```
void fetcher_get_background_tile(Emulator* emu);
void fetcher_push_bgw_pixels();
```

fetcher_get_background_tile的实现如下所示：

```
void PPU::fetcher_get_background_tile(Emulator* emu)
{
    // The y position of the next pixel to fetch relative to 256x256 tile map c
    u8 map_y = ly + scroll_y;
    // The x position of the next pixel to fetch relative to 256x256 tile map c
    u8 map_x = fetch_x + scroll_x;
    // The address to read map index.
    // ((map_y / 8) * 32) : 32 bytes per row in tile maps.
    u16 addr = bg_map_area() + (map_x / 8) + ((map_y / 8) * 32);
    // Read tile index.
    u8 tile_index = emu->bus_read(addr);
    if(bgw_data_area() == 0x8800)
    {
        // If LCDC.4=0, then range 0x9000~0x97FF is mapped to [0, 127], and ra
        // We can achieve this by simply add 128 to the fetched data, which wi
        // than 127.
        tile_index += 128;
    }
    // Calculate data address offset from bgw data area beginning.
```

```
    // tile_index * 16 : every tile takes 16 bytes.
    // (map_y % 8) * 2 : every row takes 2 bytes.
    bgw_data_addr_offset = ((u16)tile_index * 16) + (u16)(map_y % 8) * 2;
    // Calculate tile X position.
    i32 tile_x = (i32)(fetch_x) + (i32)(scroll_x);
    tile_x = (tile_x / 8) * 8 - (i32)scroll_x;
    tile_x_begin = (i16)tile_x;
}
```

首先我们找到屏幕上的（fetch_x, ly）点在地图上的坐标（map_x，map_y），该操作可以通过将坐标加上窗口左上角在地图空间中的位置（SCX、SCY）来求得。然后我们可以将（map_x，map_y）分别乘除8，就可以得到该点对应的图块的（tile_x、tile_y）坐标。由于地图的每一行具有32个图块，因此我们可以使用tile_x + tile_y * 32来求得当前图块在背景图块索引内存中的位置，并加上bg_map_area()来求得存储背景图块索引的地址。

在得到地址以后，我们调用bus_read从该地址中读取图块的索引值，并且根据LCDC.4的值判断索引的范围是-128~127还是0~256。当范围为-128~127时，由于索引值0对应的是0x9000的地址值，而非0x8800，因此我们需要将实际的索引值整体向上偏移128。在这种情况下，如果原先的索引值大于127，则在加上128以后会溢出，并自动丢弃最高位，从而正好映射到0x8800~0x9000的范围，与GameBoy所描述的映射规则保持一致。

最后，我们根据索引值和当前的map_y值，确定图块的位置和当前要读取的那一行数据在图块中的位置。图块的位置可以通过tile_index * 16求得，因为每一个图块具有16字节的尺寸，同时当前行在图块中的位置可以通过(map_y % 8) * 2求得（map_y % 8用于求得当前行是图块中的第几行，而每一行具有2字节的尺寸）。最后，我们将图块的位置和要读取的图块行在图块中的位置相加，就可以求得最终需要获取的图块行的数据位置。

在计算出图块行的数据位置后，我们在这一步中还需要计算当前图块的8个像素在屏幕空间的X坐标上的分布区间。由于每一个图块的像素均为8，因此我们实际上只需要求得图块的第一个像素的位置tile_x_begin，就可以确定所有像素的位置。tile_x_begin可以通过tile_x * 8 - SCX求得，为了避免正整数类型由于表示范围问题导致的计算错误，我们在代码中先将所有变量全部转换为i32类型，然后重新计算了一遍tile_x，以求得最终的tile_x_begin值。

fetcher_push_bgw_pixels的实现如下：

```
void PPU::fetcher_push_bgw_pixels()
{
    // Load tile data.
    u8 b1 = bgw_fetched_data[0];
    u8 b2 = bgw_fetched_data[1];
    // Process every pixel in this tile.
    for(u32 i = 0; i < 8; ++i)
    {
        // Skip pixels not in the screen.
        // Pushing pixels when tile_x_begin + i >= PPU_XRES is ok and
        // they will not be drawn by the LCD driver, and all undrawn pixels
        // will be discarded when HBLANK mode is entered.
        if(tile_x_begin + (i32)i < 0)
        {
            continue;
        }
        // Now we can stream pixel.
        BGWPixel pixel;
        if(bg_window_enable())
        {
            u8 b = 7 - i;
            u8 lo = (!!(b1 & (1 << b)));
            u8 hi = (!!(b2 & (1 << b))) << 1;
            pixel.color = hi | lo;
            pixel.palette = bgp;
        }
        else
        {
            pixel.color = 0;
            pixel.palette = 0;
        }
        bgw_queue.push_back(pixel);
        ++push_x;
    }
}
```

该函数先读取了步骤2和步骤3中加载的2个字节的数据，然后根据上一章中的图块数据解析规则，依次解析数据包含的8个像素色值，将它们压入背景/窗口队列中。对于每一个像素，我们需要判断其是否超出了屏幕左侧边缘，并当像素处于屏幕边缘外时丢弃像素，但是我们不需要判断像素是否超出屏幕右侧边缘，因为LCD驱动从左到右绘制像素，超出屏幕右侧边缘的像素并不会

被取出，并且会在下一次进入DRAWING模式时被清空。事实上，由于LCD驱动只有当队列中像素数量大于等于8时才会工作，我们确实需要为每一扫描行多提交8个像素的数据，以便最后一个像素的颜色能够被正确取出和绘制。

在通过了边缘判断以后，我们就开始真正提交像素至队列中。需要注意的是，在背景/窗口被禁用时，我们仍然需要提交背景/窗口像素，因为此时精灵像素有可能需要绘制，因此LCD驱动仍然需要对应的背景/窗口像素来保证背景/窗口像素和精灵像素是一一对应的关系。我们在这里提交一个空白像素（色值为0，同时调色板也为0），在绘制的时候，该像素会呈现出空白的颜色。

实现窗口绘制

在实现背景绘制以后，我们可以按照类似的逻辑实现窗口的绘制。在PPU类中添加以下函数的声明：

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

fetcher_get_window_tile的实现如下：

```
void PPU::fetcher_get_window_tile(Emulator* emu)
{
    u8 window_x = (fetch_x + 7 - wx);
    u8 window_y = window_line;
    u16 window_addr = window_map_area() + (window_x / 8) + ((window_y / 8) * 32
    u8 tile_index = emu->bus_read(window_addr);
    if(bgw_data_area() == 0x8800)
    {
        // If LCDC.4=0, then range 0x9000~0x97FF is mapped to [0, 127], and ran
        // We can achieve this by simply add 128 to the fetched data, which wil
        // than 127.
        tile_index += 128;
    }
    // Calculate data address offset from bgw data area beginning.
    // tile_index * 16 : every tile takes 16 bytes.
    // (window_tile_y % 8) * 2 : every row takes 2 bytes.
    bgw_data_addr_offset = ((u16)tile_index * 16) + (u16)(window_y % 8) * 2;
    // Calculate tile X position.
    i32 tile_x = (i32)(fetch_x) - ((i32)(wx) - 7);
    tile_x = (tile_x / 8) * 8 + (i32)(wx) - 7;
    tile_x_begin = (i16)tile_x;
}
```

可以看到窗口图块数据地址获取的逻辑和背景图块使用的fetcher_get_background_tile差不多，只不过我们需要将之前的（map_x，map_y）替换为（window_x和window_y），表示当前需要获取的像素（fetch_x，window_line）在窗口空间下的坐标。由于WX存储的是窗口的X坐标+7的值，因此我们需要在计算的过程中将该值减7，以得到窗口在屏幕空间下真正的X坐标。由于我们在刚开始绘制窗口时就会将fetch_x设置为窗口显示区域内第一个像素的X坐标，因此可以保证window_x是一个非负整数，即fetch_x一定大于WX-7。

由于背景和窗口并不是两个单独的管线，fetcher在阶段1中要么获取背景图块，要么获取窗口图块，因此阶段2、3、5都同时对背景和窗口生效，我们唯一要修改的是在阶段5提交像素之前判断像素是否被背景覆盖，并在像素首次被背景覆盖时将fetch_window设置为true，并丢弃后续的背景像素，如下所示：

```
void PPU::fetcher_push_bgw_pixels()
{
    // Load tile data.
    u8 b1 = bgw_fetched_data[0];
    u8 b2 = bgw_fetched_data[1];
    // Process every pixel in this tile.
    for(u32 i = 0; i < 8; ++i)
    {
        // Skip pixels not in the screen.
        // Pushing pixels when tile_x_begin + i >= PPU_XRES is ok and
        // they will not be drawn by the LCD driver, and all undrawn pixels
        // will be discarded when HBLANK mode is entered.
        if(tile_x_begin + (i32)i < 0)
        {
            continue;
        }
        // If this is a window pixel, we reset fetcher to fetch window and disc
        if(!fetch_window && is_pixel_window(push_x, ly))
        {
            fetch_window = true;
            fetch_x = push_x;
```

```
            break;
        }
        // Now we can stream pixel.
        BGWPixel pixel;
        if(bg_window_enable())
        {
            u8 b = 7 - i;
            u8 lo = (!!(b1 & (1 << b)));
            u8 hi = (!!(b2 & (1 << b))) << 1;
            pixel.color = hi | lo;
            pixel.palette = bgp;
        }
        else
        {
            pixel.color = 0;
            pixel.palette = 0;
        }
        bgw_queue.push_back(pixel);
        ++push_x;
    }
}
```

将绘制结果渲染到窗口

最后，我们需要将绘制到pixels变量中的像素信息绘制到窗口上，以便我们可以查看显示结果。为了实现这一点，我们需要使用LunaSDK的RHI模块先将像素信息拷贝到一个纹理对象（ITexture）中，然后发起一个绘制调用（draw call），将纹理绘制到窗口的back buffer上。我们先在App类中定义发起绘制调用所需要的RHI对象和函数：

```
//! Render resources.
Ref<RHI::ITexture> emulator_display_tex;
Ref<RHI::IBuffer> emulator_display_ub;
Ref<RHI::IBuffer> emulator_display_vb;
Ref<RHI::IBuffer> emulator_display_ib;
Ref<RHI::IDescriptorSetLayout> emulator_display_dlayout;
Ref<RHI::IDescriptorSet> emulator_display_desc_set;
Ref<RHI::IPipelineLayout> emulator_display_playout;
Ref<RHI::IPipelineState> emulator_display_pso;
RV init_render_resources();
```

然后在App.cpp中实现init_render_resources函数，用于初始化这些对象：

```
#include <Luna/ShaderCompiler/ShaderCompiler.hpp>
#include <Luna/RHI/ShaderCompileHelper.hpp>
#include <Luna/RHI/Utility.hpp>
struct EmulatorDisplayUB
{
    Float4x4U projection_matrix;
};
struct EmulatorDisplayVertex
{
    Float2U pos;
    Float2U uv;
};
RV App::init_render_resources()
{
    lutry
    {
    luset(emulator_display_tex, rhi_device->new_texture(RHI::MemoryType::lo
        RHI::TextureDesc::tex2d(RHI::Format::rgba8_unorm, RHI::TextureUsage
            PPU_XRES, PPU_YRES, 1, 1)));
        u32 ub_align = rhi_device->check_feature(RHI::DeviceFeature::unifor
    luset(emulator_display_ub, rhi_device->new_buffer(RHI::MemoryType::uplo
        align_upper(sizeof(EmulatorDisplayUB), ub_align))));
    luset(emulator_display_vb, rhi_device->new_buffer(RHI::MemoryType::uplo
        sizeof(EmulatorDisplayVertex) * 4)));
    luset(emulator_display_ib, rhi_device->new_buffer(RHI::MemoryType::uplo
        sizeof(u16) * 6)));
    u16* index_data;
    luexp(emulator_display_ib->map(0, 0, (void**)&index_data));
    index_data[0] = 0;
    index_data[1] = 1;
    index_data[2] = 2;
    index_data[3] = 0;
    index_data[4] = 2;
```

```
        index_data[5] = 3;
        emulator_display_ib->unmap(0, sizeof(u16) * 6);
        luset(emulator_display_dlayout, rhi_device->new_descriptor_set_layout(R
            { RHI::DescriptorSetLayoutBinding::uniform_buffer_view(0, 1, RHI::S
              RHI::DescriptorSetLayoutBinding::read_texture_view(RHI::TextureVi
              RHI::DescriptorSetLayoutBinding::sampler(2, 1, RHI::ShaderVisibil
        luset(emulator_display_desc_set, rhi_device->new_descriptor_set(RHI::De
        RHI::IDescriptorSetLayout* dlayout = emulator_display_dlayout;
        luset(emulator_display_playout, rhi_device->new_pipeline_layout(RHI::Pi
            RHI::PipelineLayoutFlag::allow_input_assembler_input_layout)));
        const c8 vs[] = R"(
cbuffer ub_b0 : register(b0)
{
    float4x4 projection_matrix;
};
Texture2D texture0 : register(t1);
SamplerState sampler0 : register(s2);
struct VS_INPUT
{
    [[vk::location(0)]]
    float2 pos : POSITION;
    [[vk::location(1)]]
    float2 uv  : TEXCOORD0;
};
struct PS_INPUT
{
    [[vk::location(0)]]
    float4 pos : SV_POSITION;
    [[vk::location(1)]]
    float2 uv  : TEXCOORD0;
};
PS_INPUT main(VS_INPUT input)
{
    PS_INPUT output;
        output.pos = mul( projection_matrix, float4(input.pos.xy, 0.f, 1.f));
        output.uv  = input.uv;
        return output;
})";
            const c8 ps[] = R"(
struct PS_INPUT
{
    [[vk::location(0)]]
    float4 pos : SV_POSITION;
    [[vk::location(1)]]
    float2 uv  : TEXCOORD0;
};
cbuffer ub_b0 : register(b0)
{
    float4x4 projection_matrix;
};
Texture2D texture0 : register(t1);
SamplerState sampler0 : register(s2);
[[vk::location(0)]]
float4 main(PS_INPUT input) : SV_Target
{
    return texture0.Sample(sampler0, input.uv);
}
)";
        auto compiler_vs = ShaderCompiler::new_compiler();
        compiler_vs->set_target_format(RHI::get_current_platform_shader_target_
        compiler_vs->set_source({ vs , sizeof(vs) });
        compiler_vs->set_shader_type(ShaderCompiler::ShaderType::vertex);
        compiler_vs->set_shader_model(6, 0);
        luexp(compiler_vs->compile());
        auto vs_data = compiler_vs->get_output();

        auto compiler_ps = ShaderCompiler::new_compiler();
        compiler_ps->set_target_format(RHI::get_current_platform_shader_target_
        compiler_ps->set_source({ ps , sizeof(ps) });
        compiler_ps->set_shader_type(ShaderCompiler::ShaderType::pixel);
        compiler_ps->set_shader_model(6, 0);
        luexp(compiler_ps->compile());
        auto ps_data = compiler_ps->get_output();

        RHI::GraphicsPipelineStateDesc pso;
        pso.primitive_topology = RHI::PrimitiveTopology::triangle_list;
        pso.rasterizer_state = RHI::RasterizerDesc(RHI::FillMode::solid, RHI::C
        pso.depth_stencil_state = RHI::DepthStencilDesc(false, false, RHI::Comp
        RHI::InputBindingDesc input_bindings[] = {
```

```
        RHI::InputBindingDesc(0, sizeof(EmulatorDisplayVertex), RHI::InputR
    };
    RHI::InputAttributeDesc input_attributes[] = {
        RHI::InputAttributeDesc("POSITION", 0, 0, 0, 0, RHI::Format::rg32_f
        RHI::InputAttributeDesc("TEXCOORD", 0, 1, 0, 8, RHI::Format::rg32_f
    };
    pso.input_layout.bindings = { input_bindings, 1 };
    pso.input_layout.attributes = { input_attributes , 2 };
    pso.vs = vs_data;
    pso.ps = ps_data;
    pso.pipeline_layout = emulator_display_playout;
    pso.num_color_attachments = 1;
    pso.color_formats[0] = RHI::Format::bgra8_unorm;
    luset(emulator_display_pso, rhi_device->new_graphics_pipeline_state(psc
    luexp(emulator_display_desc_set->update_descriptors({
        RHI::WriteDescriptorSet::uniform_buffer_view(0, RHI::BufferViewDesc
        RHI::WriteDescriptorSet::read_texture_view(1, RHI::TextureViewDesc(
        RHI::WriteDescriptorSet::sampler(2, RHI::SamplerDesc(RHI::Filter::n
            }));
    }
    lucatchret;
    return ok;
}
```

由于本专题的重点并不在实现LunaSDK上，因此我们对于这部分代码不做过多的说明。然后我们需要修改App::init函数，以调用init_render_resources函数：

```
RV App::init()
{
    lutry
    {
        /*...*/
        last_frame_ticks = get_ticks();
        luexp(init_render_resources());
    }
    lucatchret;
    return ok;
}
```

然后我们修改App::update函数，以加入模拟器屏幕绘制相关的功能：

```
RV App::update()
{
    lutry
    {
        /*...*/
        // Draw GUI.
        draw_gui();

        // Upload emulator screen pixels.
        if(emulator)
        {
            u8* src = emulator->ppu.pixels + ((emulator->ppu.current_back_buffe
            // Copy display data to texture for rendering.
            luexp(RHI::copy_resource_data(g_app->cmdbuf, {
                RHI::CopyResourceData::write_texture(emulator_display_tex, {0,
            }));
        }
        // Clear back buffer.
        lulet(back_buffer, swap_chain->get_current_back_buffer());
        Float4U clear_color = { 0.3f, 0.3f, 0.3f, 1.3f };
        RHI::RenderPassDesc render_pass;
        render_pass.color_attachments[0] = RHI::ColorAttachment(back_buffer, RH
        cmdbuf->begin_render_pass(render_pass);
        cmdbuf->end_render_pass();
        // Draw emulator screen.
        luexp(draw_emulator_screen(back_buffer));
        // Render GUI.
        /*...*/
    }
    lucatchret;
    return ok;
}
```

模拟器屏幕绘制主要分两步进行，首先，我们在模拟器启动以后调用RHI::copy_resource_data将

模拟器保存在ppu.pixels中的像素颜色信息上传到emulator_display_tex中，该对象是一个RHI::ITexture对象，用于将模拟器画面绘制到窗口上。我们在上一章实现图块预览调试窗口时已经使用copy_resource_data函数执行过类似的将数据从内存拷贝至显存的操作。接着，我们通过swap_chain->get_current_back_buffer()获取窗口当前的back buffer，在清屏以后，调用draw_emulator_screen函数绘制模拟器屏幕。

接下来，我们需要实现draw_emulator_screen函数。在App类中添加该函数的声明：

```
RV draw_emulator_screen(RHI::ITexture* back_buffer);
```

然后在App.cpp中添加实现：

```
RV App::draw_emulator_screen(RHI::ITexture* back_buffer)
{
    lutry
    {
        if(emulator.get())
        {
            auto window_sz = window->get_framebuffer_size();
            f32 window_width = window_sz.x;
            f32 window_height = window_sz.y;
            EmulatorDisplayVertex* vb_mapped;
            luexp(emulator_display_vb->map(0, 0, (void**)&vb_mapped));
            f32 display_width = PPU_XRES * 4;
            f32 display_height = PPU_YRES * 4;
            vb_mapped[0].pos = Float2U((window_width - display_width) / 2.0f,
            vb_mapped[0].uv = Float2U(0.0f, 0.0f);
            vb_mapped[1].pos = Float2U((window_width + display_width) / 2.0f,
            vb_mapped[1].uv = Float2U(1.0f, 0.0f);
            vb_mapped[2].pos = Float2U((window_width + display_width) / 2.0f,
            vb_mapped[2].uv = Float2U(1.0f, 1.0f);
            vb_mapped[3].pos = Float2U((window_width - display_width) / 2.0f,
            vb_mapped[3].uv = Float2U(0.0f, 1.0f);
            emulator_display_vb->unmap(0, sizeof(EmulatorDisplayVertex) * 4);
            EmulatorDisplayUB* ub_mapped;
            luexp(emulator_display_ub->map(0, 0, (void**)&ub_mapped));
            ub_mapped->projection_matrix = {
                { 2.0f / window_width, 0.0f,                                0.0
                { 0.0f,                2.0f / -window_height,  0.0f,
                { 0.0f,                0.0f,
                { -1.0f,               1.0f,                 0.5f,
            };
            emulator_display_ub->unmap(0, sizeof(EmulatorDisplayUB));
            cmdbuf->resource_barrier({
                RHI::BufferBarrier(emulator_display_ub, RHI::BufferStateFlag::a
                RHI::BufferBarrier(emulator_display_vb, RHI::BufferStateFlag::a
                RHI::BufferBarrier(emulator_display_ib, RHI::BufferStateFlag::a
                }, {
                RHI::TextureBarrier(emulator_display_tex, RHI::SubresourceIndex
                RHI::TextureBarrier(back_buffer, RHI::SubresourceIndex(0, 0), R
                });
            RHI::RenderPassDesc render_pass;
            render_pass.color_attachments[0] = RHI::ColorAttachment(back_buffe
            cmdbuf->begin_render_pass(render_pass);
            cmdbuf->set_graphics_pipeline_layout(emulator_display_playout);
            cmdbuf->set_graphics_pipeline_state(emulator_display_pso);
            cmdbuf->set_graphics_descriptor_set(0, emulator_display_desc_set);
            cmdbuf->set_vertex_buffers(0, { RHI::VertexBufferView(emulator_disp
            cmdbuf->set_index_buffer(RHI::IndexBufferView(emulator_display_ib,
            cmdbuf->set_viewport(RHI::Viewport(0.0f, 0.0f, window_sz.x, window_
            cmdbuf->set_scissor_rect(RectI(0, 0, window_sz.x, window_sz.y));
            cmdbuf->draw_indexed(6, 0, 0);
            cmdbuf->end_render_pass();
        }
    }
    lucatchret;
    return ok;
}
```

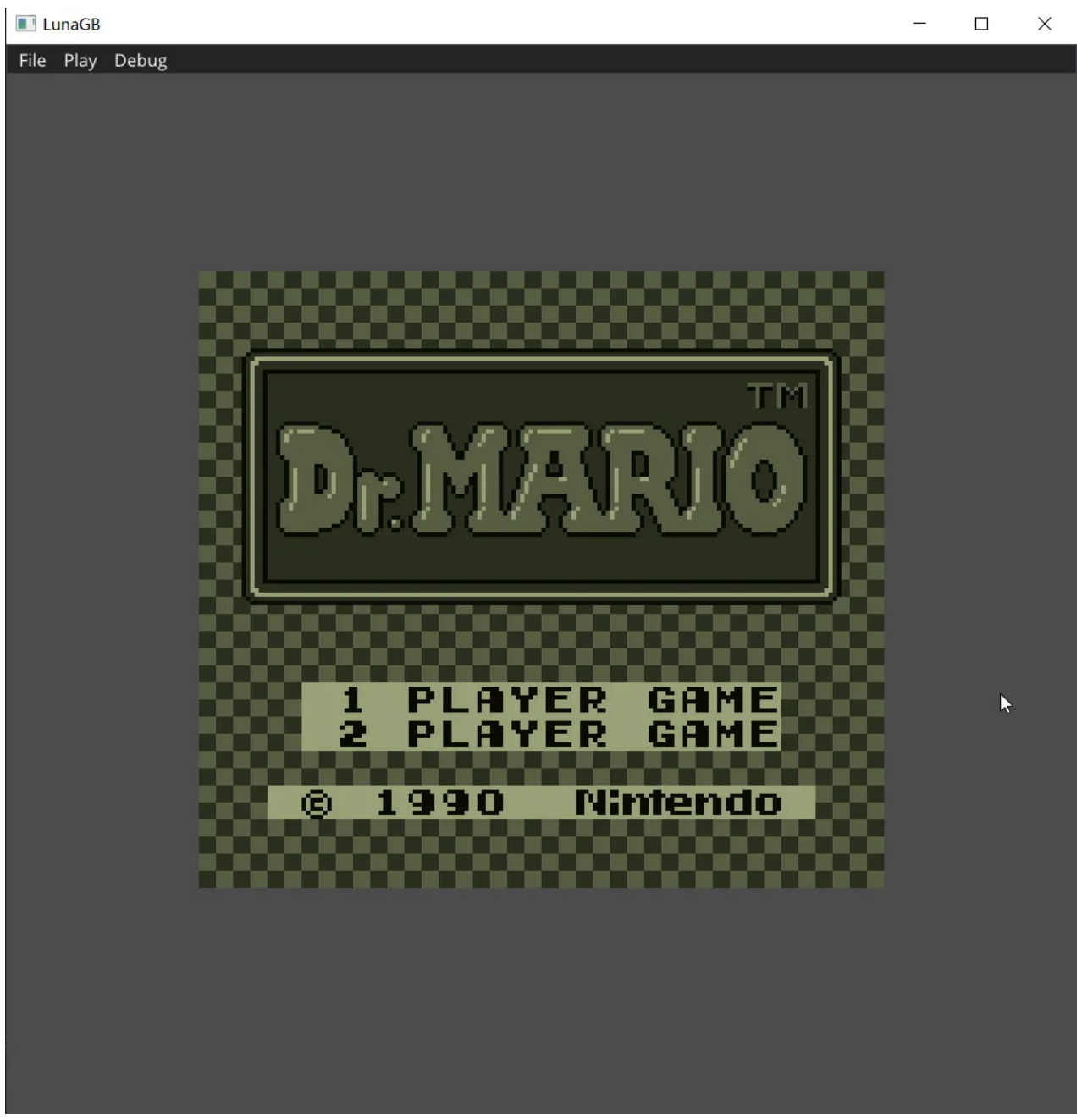
由于我们在初始化RHI资源的时候需要编译shader，因此我们需要在xmake.lua的add_deps中添加ShaderCompiler模块，这样我们的代码才能够正常编译和链接：

```
target("LunaGB")
    set_luna_sdk_program()
    add_headerfiles("**.hpp")
    add_files("**.cpp")
```

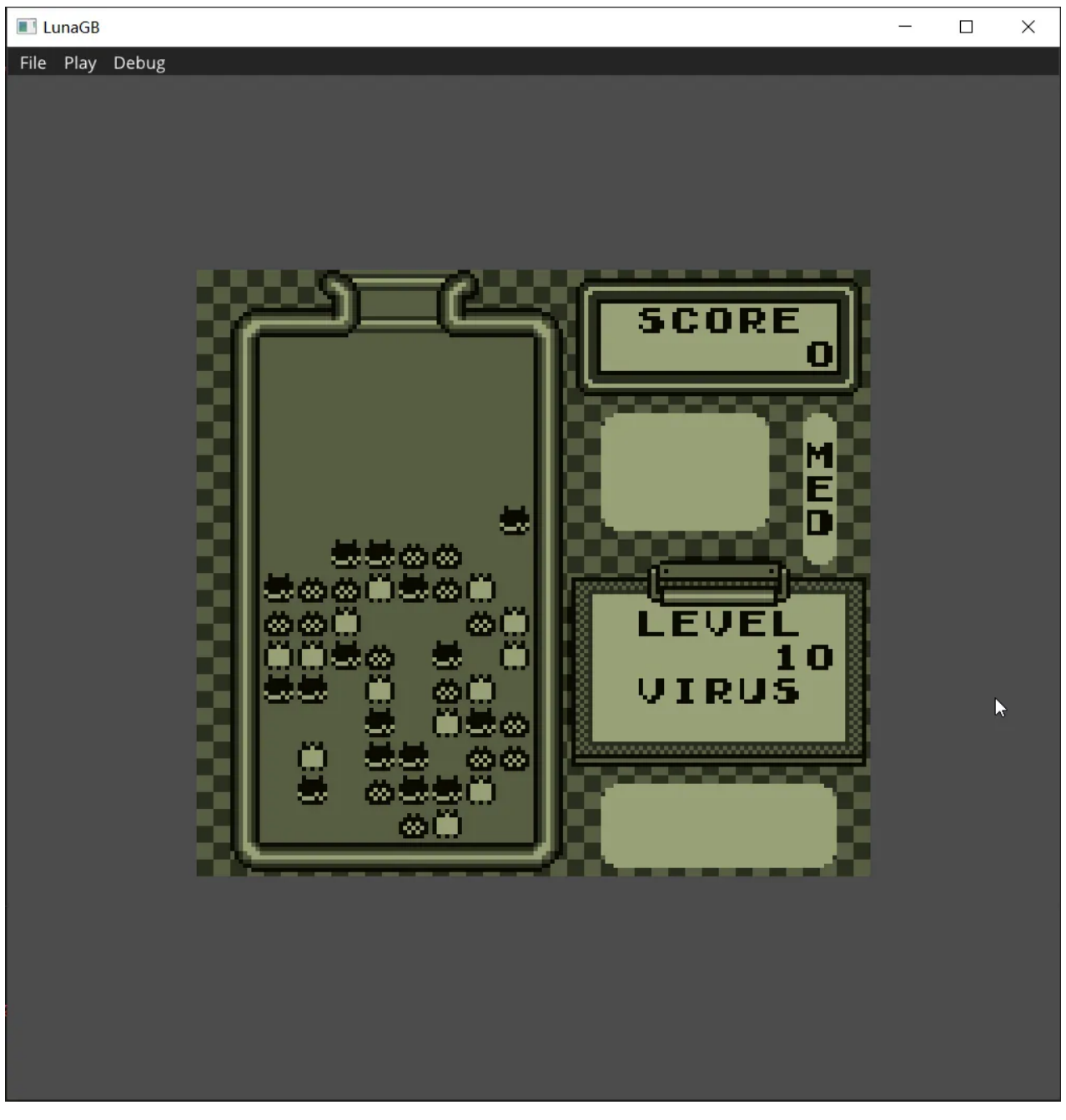
```
    add_deps("Runtime", "Window", "RHI", "ShaderCompiler", "ImGui", "HID", "AHI
target_end()
```

如果您是在本章发布以后才更新的代码，那么上述的ShaderCompiler模块应当已经添加在add_deps中了。

此时编译并运行模拟器，打开一个单ROM类型的打开文件，例如《马力欧医生》，可以看到模拟器已经能够正常渲染并输出背景/窗口画面了：



在《马力欧医生》标题界面等待一段时间，可以看到游戏会自动进入了演示游玩模式，但是画面中缺少了许多元素，例如马力欧的形象，和右下角代表剩余病毒的小人，因为这些元素并不是背景，而是作为精灵图块进行绘制的。



以上就是本章节的全部内容了。本章节涉及实现的内容较多，希望读者能够耐心琢磨每一个绘制

步骤的原理和代码，并最终在使用多个单卡带游戏（例如《网球》、《俄罗斯方块》、《马力欧医生》等）测试绘制画面的正确性。在下一章中，我们将实现精灵图块的绘制，并对PPU的实现进行收尾。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #9 绘制精灵
19 赞同 · 2 评论 文章

编辑于 2024-02-27 23:37 · IP 属地上海

Game Boy（GB） 游戏机模拟器

362

欢迎参与讨论

还没有评论，发表第一个评论吧

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

yuzu模拟器上的shader逆向，流程+工具开发（笔记）
最近买了塞尔达《王国之泪》，在switch上玩过后，想学习一下它的渲染技术，于是从网上下载了柚子模拟器的游戏版本，模拟器是可以使用renderdoc截帧的，但是反出来的shader代码实在是太难懂...
一片枫叶 发表于DSL元语...

MAC运行MAME 街机模拟器
主流的非主流

游戏主机模拟器的一般设计模式
satur... 发表于计算机技术

黑莓BBOS10教材之游戏篇：活用retroarch+ppsspp尽玩铁
黑莓10OS的游戏主要分有两类：一类是原生游戏，也就是BBOS10平台游戏，一类是非BBOS10平台游戏，包括：GBA游戏、街机游戏、PS1游戏、PSP游戏、安卓游戏。（怎没有IOS游戏，这不可能有
杞鋂

▲ 赞同 20 ▼ 添加评论 分享 喜欢 收藏 申请转载 ···