


从零开始实现GameBoy模拟器 #7 PPU

 銀葉吉祥 
浙江大学 软件工程硕士

已关注

37 人赞同了该文章

目录

收起

- 图块
- 图块内存
- 背景图块
- 窗口图块
- 精灵图块
- 像素处理单元
- 0xFF40 - LCDC (LCD Contro
- 0xFF41 - LCDS (LCD Statu
- 0xFF44 - LY
- 0xFF45 - LYC (LY Compar
- 其余寄存器
- 实现PPU
- 查看显示内存的图块



欢迎来到从零开始实现GameBoy模拟器第七章。从本章开始，我们将进入像素处理单元（Pixel Processing Unit、PPU）的实现。PPU是GameBoy的SoC中一个相对独立的组件，其负责将存储在显示内存中的图块（tile）组装成GameBoy的画面，并将像素通过逐行流送（streaming）的方式显示在屏幕上。在本章中，我们将首先对GameBoy图像绘制和PPU的工作原理有一个基本的了解，然后通过代码实现PPU的状态机模型和更新循环，而在接下来的两章中，我们将开始实现PPU的绘制和像素流送功能，以便我们能够真正显示游戏画面。那么，让我们开始吧！

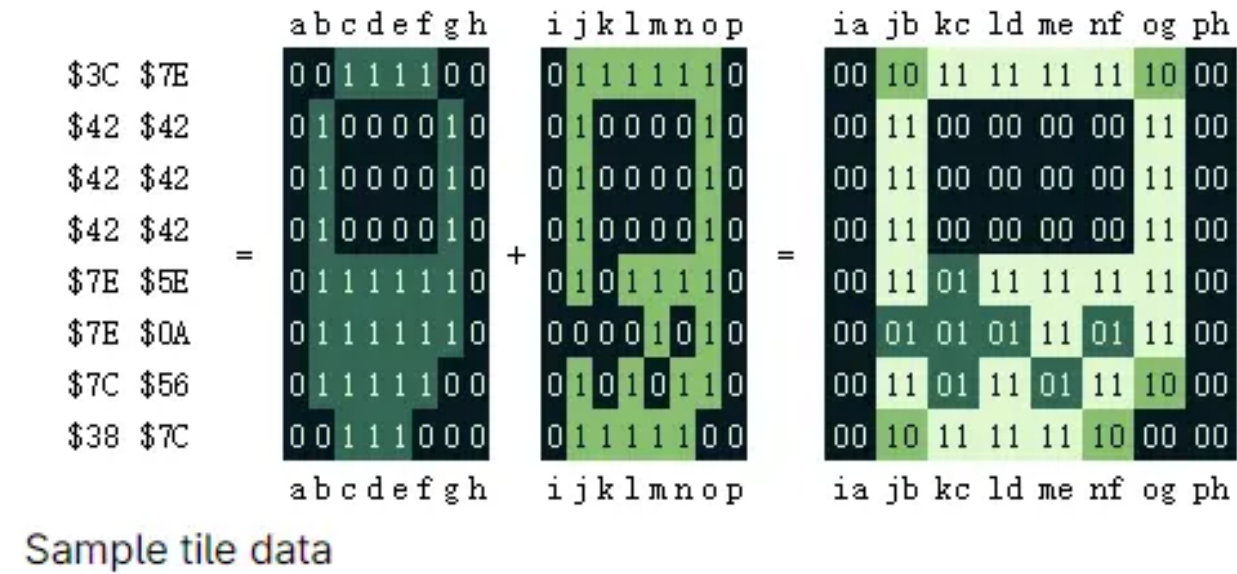
本章节代码已经上传至项目仓库，名称为LunaGB-07，已经下载项目代码的读者可以更新代码库来查阅最新代码。同时，欢迎大家加入GameBoy模拟器开发交流群：33854734
2024.2.12更新：修改PPU::bus_write函数，在LCDC.7从1变为0（禁用LCD&PPU）时需要重置LY为0，并将LCDS.0和LCDS.1设置为0（设置模式为HBLANK）。
2024.2.15更新：修改一处笔误，背景和窗口的分辨率为256x256，应当由32x32个图块构成，占用1024个字节的空间来存储图块索引，而非16x16个图块。

图块

GameBoy所使用的CPU性能相当有限，不足以支持游戏程序在每一帧中实时绘制所有的像素，因此和当年的大部分其它游戏设备一样，GameBoy要求程序先在显示内存（VRAM）中预先保存游戏需要使用的所有画面组件，然后通过专门的像素处理单元（PPU）将显存中的画面组件拼装成最终的游戏画面。在GameBoy中，画面组件以图块（tile）的形式保存在显示内存中，每一个图块是一个8x8像素的位图，占用16个字节，因此每一个像素具有2个bit的像素宽度，可以表示一共4个不同的色值。图块的像素色值（color index）用于在调色板（palette）中索引对应的颜色：在单色GameBoy中，每一个色值可以映射到四个不同等级的灰度中的一个；而在彩色GameBoy中，每一个色值可以映射到一个不同的RGB颜色。

单个图块内的16个字节逐行存储，每一行占用2个字节，因此图块最开头的2个字节存储图块第一行的数据，之后的两个字节存储图块第二行的数据，以此类推。图块的每一行具有8个像素，每一个像素都在两个字节中分别使用一个bit来存储像素信息，从高到低排列，因此第一个像素使用两个字节的第7位（从最低位为0开始算）来存储信息，第二个像素使用两个字节的第6位来存储信息，以此类推。在读取每个像素信息时，PPU会分别读取两个字节指定位的数据，然后将第二个字节的读到的数据向左位移一位，加上第一个字节读到的数据，来计算该像素最终的颜色值。

举个例子，假设图块某一行的两个字节的的数据分别为0x3C（00111100b）和0x7E（01111110b），则8个像素的最终值是00b 10b 11b 11b 11b 11b 10b 00b。下图形象地展示了一个图块中16个字节与最终形成的位图的映射关系：



来源：https://gbdev.io/pandocs/Tile_Data.html

图块内存

单色GameBoy总共具有8KB的显示内存（VRAM），可以通过地址0x8000至地址0x9FFF访问。其中，地址0x8000至0x97FF的内存可以用于存储图块数据，因此其最多可以同时存储384个不同的图块。

GameBoy按照用途的不同，将图块分为了两种类型：**背景/窗口图块**以及**精灵图块**。不同类型的图块只能在规定的内存区域中存储，其中精灵图块只能存储在0x8000~0x8FFF的地址区域内，因此最多只能同时存储最多256个精灵图块；背景/窗口图块可以按照游戏需求存储在0x8000~0x8FFF地址区域内或0x8800~0x97FF地址区域内，因此同样最多只能同时存储最多256个背景/窗口图块。同时，由于背景/窗口图块和精灵图块有重叠的地址区域，而每个地址只能保存一种类型的图块，因此实际可以存储的图块数量比理论限制要更少。

所有的图块均使用8位的**图块索引**来表示。对于精灵图块来说，该值的范围为0~255，因此第0个图块为0x8000地址所存储的图块，第255个图块为0x8FFE地址所存储的图块；对于背景/窗口图块来说，如果图块地址从0x8000开始，则其使用和精灵图块相同的范围和映射关系（0~255），但是如果图块地址从0x8800开始，则图块的索引范围为-128~127（有符号），因此第-128个图块为0x8800地址所存储的图块，第0个图块为0x9000地址所存储的图块，第127个图块为0x97FE地址所存储的图块。下表总结了不同类型的图块的地址与索引号的映射关系：

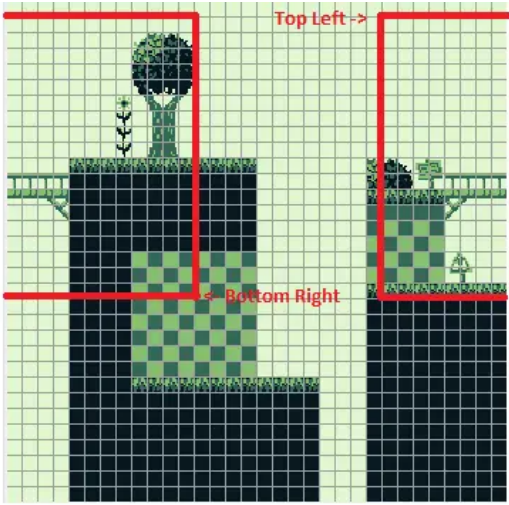
地址范围	精灵图块编号	背景/窗口图块编号 （从0x8000开始时）	背景/窗口图块编号 （从0x8800开始时）
0x8000~0x87FF	0~127	0~127	无法使用
0x8800~0x8FFF	128~255	128~255	-128~-1（有符号）或 128~255（无符号）
0x9000~0x97FF	无法使用	无法使用	0~127

背景图块

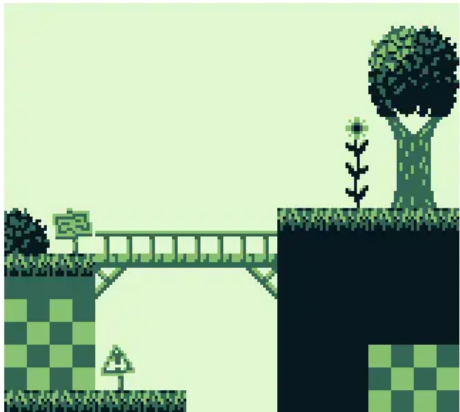
背景图块用于构成GameBoy的背景，例如游戏的地图或者展示的静态图片。GameBoy支持表示256x256像素（也就是32x32图块）的背景，背景中的每一个图块都使用上文所述的图块索引来表示，因此我们需要1024字节的存储空间来保存所有32x32个图块索引值。在GameBoy中，这1024字节的背景图块索引可以根据程序配置保存在0x9800或者0x9C00开始的1024个字节中，逐行存储（即前32个字节保存第一行32个图块的索引，接下来的32个字节保存第二行32个图块的索引，以此类推）。

虽然GameBoy支持表示256x256像素的背景，但是GameBoy的屏幕实际上只有160x144的像素分辨率，因此GameBoy只能显示背景的一部分区域。GameBoy允许程序通过SCX和SCY寄存器（在下一章中详细讲解）来控制背景需要显示的区域。在使用时，SCX和SCY分别控制显示区域左上角相对于背景左上角像素的相对位置（X轴向右、Y轴向下），以像素单位表示，而显示区域的长和宽则固定为160和144像素。在读取背景像素时，如果显示区域超出了背景像素的范围，则超出的部分会以256取模以后再从地图中读取像素显示，表现出来的效果类似于将背景像素在水平和垂直方向上以256x256像素为单位无限重复以后再显示，如下图所示：

Background Tilemap:



Output:



<https://gbdev.io/pandocs/Scrolling.html>

窗口图块

窗口图块与背景图块在功能上类似，也是一个使用32x32个图块索引表示的256x256像素的位图，并且这1024字节的窗口图块索引也可以根据程序配置保存在0x9800或者0x9C00开始的1024个字节中，逐行存储。在绘制时，由这256x256像素构成的“窗口”可以叠加显示在背景上，覆盖掉背景像素，从而在背景滚动的时候令某些像素的颜色保持不变。在实际的游戏中，窗口通常用于显示一些玩家的当前状态信息，例如血量、积分、倒计时等，但是这并不是强制规定——游戏可以使用别的方式实现状态信息显示，窗口也可以用于别的目的。

与背景图块不同的是，窗口不可以滚动，其永远从左上角的像素开始显示。程序唯一能控制的是窗口的左上角在屏幕上的位置，该位置通过WX和WY寄存器存储。由于窗口的大小是256x256像素，大于屏幕的160x144像素，因此窗口会占满从指定的左上角位置开始的整个右下方区域。例如，将左上角位置设置为(0, 0)可以使得窗口覆盖整个背景，而将左上角位置设置为 (160, 144) 则可以隐藏窗口。

精灵图块

精灵图块用于在游戏画面上绘制精灵。精灵（sprite）表示一个可以自由放置在场景中任意位置的位图，可以用来表示玩家、怪物、金币等动态物体。一个精灵（sprite）可以由单个精灵图块表示，并具有8x8的像素；也可以由连续的两个精灵图块共同表示，并具有8x16的像素。当精灵在场景中被绘制时，其会覆盖背景和窗口像素，除非精灵图块中某些像素的色值为0。在精灵图块中，像素色值0表示该像素为透明像素，因此其不会覆盖背景和窗口像素。也正是因为这个原因，每个精灵图块只能使用三种不同的色值来表示具体的颜色，而非背景/窗口图块的四种。

精灵的绘制参数存储在一块特殊的内存区域中，这块区域被称为对象属性内存（Object Attribute Memory, OAM），并可以通过地址0xFE00~0xFE9F访问。OAM区域一共有160字节的存储空间，同时一个精灵对象需要使用4个字节来存储属性，因此GameBoy最多允许同时绘制40个精灵。同时，由于GameBoy的硬件限制，屏幕的每一扫描行（scan line）最多只能绘制10个精灵，且最多只能处理3个精灵的重叠绘制。

由于精灵的绘制规则较为复杂，我们将会在之后的章节中再详细介绍精灵的绘制规则。

像素处理单元

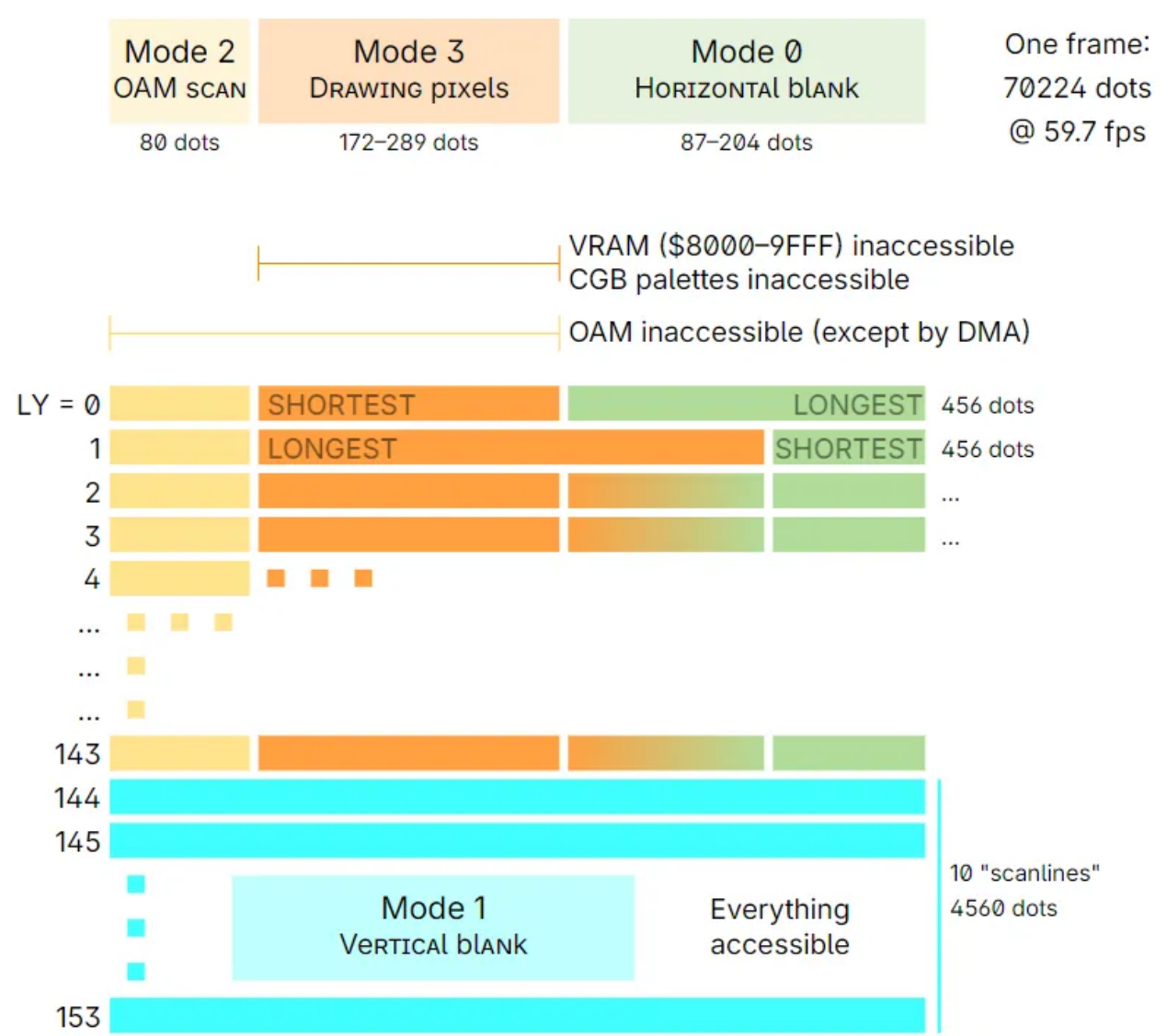
像素处理单元（Pixel Processing Unit, PPU）是一个集成在GameBoy SoC中的专用电路，负责读取以图块形式保存的位图数据，并将位图数据解析为具体的像素颜色，发送给LCD驱动电路来显示。像素处理单元与CPU并行工作，但是使用同一个系统时钟驱动，因此其与CPU的时序可以严格保证。

像素处理单元在LCD模块启动以后就开始工作。在启动的情况下，像素处理单元会逐扫描行（scan line）更新游戏画面，每一帧画面由154行扫描行构成，其中前144行对应屏幕Y坐标从0至143的像素行，后10行则为垂直空白行（vertical blank），目的是允许游戏程序在这期间更新画面信息（例如图块数据、背景和窗口的索引和滚动、精灵位置等）而不至于引起画面撕裂。每一行的绘制（包括空白行）都固定消耗456个时钟周期，因此整个画面的绘制一共消耗70224个时钟周期。在4194304Hz的时钟频率下，这相当于像素处理单元可以以大约59.7FPS的速率更新游戏画面。

除了垂直空白行以外，像素处理单元对每一行的绘制又可以分为三个阶段：OAM扫描（OAM scanning）阶段、绘制阶段和水平空白（horizontal blank）阶段，这三个阶段依次进行。OAM扫

描阶段固定消耗80个时钟周期，在此期间，像素处理单元会遍历上文中提到的OAM区域，以选出所有与本扫描行相交的精灵对象，这些对象会在之后的绘制阶段进行绘制；绘制阶段的耗时从172至289个时钟周期不等，取决于需要绘制的精灵数量、是否开启窗口等一系列因素。在绘制阶段中，像素处理单元会按屏幕坐标从左到右解析背景、窗口和精灵的图块数据，生成具体的像素信息，并将像素流送至LCD驱动电路中显示。像素处理单元绘制完毕整行的像素以后，其会进入水平空白阶段。水平空白阶段的耗时从87至204个时钟周期不等，并且该阶段与绘制阶段的耗时加起来一定为259个时钟周期。水平空白阶段的主要功能是让像素处理单元和程序准备好进行下一行画面的绘制，同时确保每一行绘制的时间开销相等。在水平空白阶段结束以后，像素处理单元会跳转到下一扫描行，并且重新回到OAM扫描阶段或者垂直空白阶段。

下图展示了像素处理单元在绘制一帧画面时的状态分解图：



来源：<https://gbdev.io/pandocs/Rendering.html>

在实现PPU的帧绘制时，我们本质上就是实现了一个状态机，在上图所示的四个状态中切换，并在每个状态中执行对应的工作。

0xFF40 - LCDC (LCD Control)

在了解了PPU的工作原理后，让我们来了解一些与PPU工作有关的I/O寄存器吧。

LCDC (LCD Control) 寄存器用于控制PPU以及LCD屏幕的工作状态，其各个位的功能如下：

7	6	5	4	3	2	1	0
启用 LCD&PP U	窗口索引地址	绘制窗口	背景/窗口图块地址	背景索引地址	精灵尺寸	绘制精灵	绘制背景/窗口

- 绘制背景/窗口：如果该位为1，则在绘制阶段绘制背景、窗口像素，否则不显示背景/窗口。
- 绘制精灵：如果该位为1，则在绘制阶段绘制精灵像素，否则不显示精灵。
- 精灵尺寸：如果该位为1，则精灵尺寸为8x16，否则为8x8。
- 背景索引地址：如果该位为1，则背景图块索引从0x9C00~0x9FFF读取，否则从0x9800~0x9BFF读取。
- 背景/窗口图块地址：如果该位为1，则背景/窗口图块从0x8000~0x8FFF读取，且索引范围为0~255，否则背景/图块从0x8800~0x97FF读取，且索引范围为-128~127。
- 绘制窗口：如果该位为1，则显示窗口，否则不显示。
- 窗口索引地址：如果该位为1，则窗口图块索引从0x9C00~0x9FFF读取，否则从0x9800~0x9BFF读取。
- 启动LCD&PPU：PPU和LCD是否运行。将该位设置为0会停止PPU运行并关闭屏幕显示。

0xFF41 - LCDS (LCD Status)

LCDS (LCD Status) 寄存器保存PPU当前的工作状态，供程序读写，其各个位的功能如下：

7	6	5	4	3	2	1	0
-	LYC中断	OAM中断	VBLANK中断	HBLANK中断	LYC标志位	PPU模式	PPU模式

- PPU模式：位0和位1为只读，共同表示了当前的PPU工作模式，与上文所述的四个工作模式对应。值与模式的对应关系为：0 - HBLANK，1 - VBLANK，2 - OAM_SCAN，3 - DRAWING。该值由PPU实时更新。
- LYC标志位：只读，当下文所述的LYC寄存器的值与LY相等时设置为1，否则设置为0。该值由PPU实时更新。
- HBLANK中断：如果设置为1，则当PPU进入HBLANK模式时会触发LCD_STAT中断。
- VBLANK中断：如果设置为1，则当PPU进入VBLANK模式时会触发LCD_STAT中断。
- OAM中断：如果设置为1，则当PPU进入OAM_SCAN模式时会触发LCD_STAT中断。
- LYC中断：如果设置为1，则当LYC标志位从0变成1时会触发LCD_STAT中断。

0xFF44 - LY

LY寄存器保存了当前PPU正在绘制的扫描行的Y值，最上方的扫描行为0，从上往下增加，范围为0~153，其中144~153为VBLANK区域。

该值由PPU实时更新，对CPU只读。

0xFF45 - LYC (LY Compare)

LYC寄存器可以保存一个用于和LY的值进行比较的值，当LYC==LY时，PPU会将LCDS.2设置为1，同时在LCDS.6为1时触发LCD_STAT中断；当LYC!=LY时，PPU会将LCDS.2设置为0。

其余寄存器

除了上述寄存器以外，PPU还具有其它寄存器，包括：

- 0xFF42 - SCY
- 0xFF43 - SCX
- 0xFF46 - DMA
- 0xFF47 - BGP
- 0xFF48 - OBP0
- 0xFF49 - OBP1
- 0xFF4A - WX
- 0xFF4B - WY

我们会在之后两章介绍到与这些寄存器相关的内容时再对这些寄存器进行展开说明。

实现PPU

在了解了PPU相关的理论知识以后，就让我们开始实现PPU吧！首先我们需要新建两个文件：PPU.hpp和PPU.cpp，用于保存PPU相关的代码。PPU.hpp的内容如下：

```
#pragma once
#include <Luna/Runtime/MemoryUtils.hpp>

using namespace Luna;

enum class PPUMode : u8
{
    hblank = 0,
    vblank = 1,
    oam_scan = 2,
    drawing = 3
};
constexpr u32 PPU_LINES_PER_FRAME = 154;
constexpr u32 PPU_CYCLES_PER_LINE = 456;
constexpr u32 PPU_YRES = 144;
constexpr u32 PPU_XRES = 160;
struct Emulator;
struct PPU
{
    ///! 0xFF40 - LCD control.
    u8 lcdc;
    ///! 0xFF41 - LCD status.
    u8 lcds;
    ///! 0xFF42 - SCY.
    u8 scroll_y;
    ///! 0xFF43 - SCX.
    u8 scroll_x;
    ///! 0xFF44 - LY LCD Y coordinate [Read Only].
    u8 ly;
    ///! 0xFF45 - LYC LCD Y Compare.
    u8 lyc;
    ///! 0xFF46 - DMA value.
```

```
u8 dma;
///! 0xFF47 - BGP (BG palette data).
u8 bgp;
///! 0xFF48 - OBP0 (OBJ0 palette data).
u8 obp0;
///! 0xFF49 - OBP1 (OBJ1 palette data).
u8 obp1;
///! 0xFF4A - WY (Window Y position plus 7).
u8 wy;
///! 0xFF4B - WX (Window X position plus 7).
u8 wx;

bool enabled() const { return bit_test(&lcdc, 7); }

PPUMode get_mode() const { return (PPUMode)(lcds & 0x03); }
void set_mode(PPUMode mode)
{
    lcds &= 0xFC; // clears previous mode.
    lcds |= (u8)(mode);
}
void set_lyc_flag() { bit_set(&lcds, 2); }
void reset_lyc_flag() { bit_reset(&lcds, 2); }
bool hblank_int_enabled() const { return !(lcds & (1 << 3)); }
bool vblank_int_enabled() const { return !(lcds & (1 << 4)); }
bool oam_int_enabled() const { return !(lcds & (1 << 5)); }
bool lyc_int_enabled() const { return !(lcds & (1 << 6)); }

void init();
void tick(Emulator* emu);
u8 bus_read(u16 addr);
void bus_write(u16 addr, u8 data);
};
```

首先我们定义了PPUMode枚举，用于表示当前PPU的工作模式，每个枚举的映射值正好对应到LCDS寄存器中不同模式的数值。然后我们定义了一些与PPU相关的常数，例如屏幕分辨率，每一行的时钟周期数，一帧画面的扫描行数等。然后我们定义了PPU类来保存我们的所有PPU相关的数据和结构，并为每一个寄存器分配了对应的存储空间。寄存器按照地址顺序排列，且中间没有空白，因此我们可以直接使用&lcdc的地址加上地址偏移来读写所有寄存器的数据。同时，我们还定义了一些辅助操作LCDC和LDCS寄存器中不同位的函数，以便在之后的代码编写中更加清晰。PPU的对外接口与定时器、总线一致：使用init函数初始化、使用tick函数更新状态，并使用bus_read和bus_write函数读写寄存器。

接着，我们在PPU.cpp中实现上述函数：

```
#include "PPU.hpp"
#include "Emulator.hpp"

void PPU::init()
{
    lcdc = 0x91;
    lcds = 0;
    scroll_y = 0;
    scroll_x = 0;
    ly = 0;
    lyc = 0;
    dma = 0;
    bgp = 0xFC;
    obp0 = 0xFF;
    obp1 = 0xFF;
    wy = 0;
    wx = 0;
    set_mode(PPUMode::oam_scan);

    line_cycles = 0;
}
void PPU::tick(Emulator* emu)
{
    /*TODO*/
}
u8 PPU::bus_read(u16 addr)
{
    luassert(addr >= 0xFF40 && addr <= 0xFF4B);
    return ((u8*)(&lcdc))[addr - 0xFF40];
}
void PPU::bus_write(u16 addr, u8 data)
{
    luassert(addr >= 0xFF40 && addr <= 0xFF4B);
    if(addr == 0xFF40 && enabled() && !bit_test(&data, 7))
```

```

{
    // Reset mode to HBLANK.
    lcds &= 0x7C;
    // Reset LY.
    ly = 0;
    line_cycles = 0;
}
if(addr == 0xFF41) // the lower 3 bits are read only.
{
    lcds = (lcds & 0x07) | (data & 0xF8);
    return;
}
if(addr == 0xFF44) return; // read only.
((u8*)(&lcdc))[addr - 0xFF40] = data;
}
```

需要注意的是，init函数并不只是简单将所有寄存器清零，部分寄存器具有非零的初始值，例如LCDC的初始值是0x91，表示启用PPU、启动背景/窗口绘制、禁用精灵绘制，同时设置了背景和窗口的默认地址区间。bus_read和bus_write使用上述规则，将lcdc变量的地址直接作为u8类型数组的第一个元素，因此我们不需要使用判断语句将各个寄存器的地址一一映射，在执行时效率也更高。需要注意的是，由于LCDS的第三位以及LY寄存器为只读，因此在bus_read中，我们需要做特殊处理，从而保留这些只读寄存器原有的值。同时，将LCDC.7从1设置为0（即关闭屏幕和停止PPU）会将LCDS的最低2位设置为00（HBLANK模式），同时将LY寄存器重置为0，因此我们需要在写入地址的时候做特殊处理。

此处的核心是tick函数。在tick函数中，我们需要逐行扫描屏幕的所有像素（包括垂直空白行），然后根据当前的模式来确定需要做的操作。我们首先在PPU类中定义一个变量来保存当前行所消耗的时钟周期：

```

struct PPU
{
    /*...*/
    /// 0xFF4B - WX (Window X position plus 7).
    u8 wx;

    // PPU internal state.

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

    bool enabled() const { return bit_test(&lcdc, 7); }
    /*...*/
};
```

并在初始化时将时钟周期设置为0：

```

void PPU::init()
{
    /*...*/
    set_mode(PPUMode::oam_scan);

    line_cycles = 0;
}
```

这样一来，当我们第一次调用tick时，我们会从第一行（因为LY == 0）开始更新，且最初的模式是OAM_SCAN，这与上面描述的PPU行为一致。在每一次调用tick时，如果PPU没有被暂停，则我们会将line_cycles的值加1，然后根据当前的模式不同调用不同的tick函数，因此tick函数的实现如下：

```

void PPU::tick(Emulator* emu)
{
    if(!enabled()) return;
    ++line_cycles;
    switch(get_mode())
    {
        case PPUMode::oam_scan:
            tick_oam_scan(emu); break;
        case PPUMode::drawing:
            tick_drawing(emu); break;
        case PPUMode::hblank:
            tick_hblank(emu); break;
        case PPUMode::vblank:
            tick_vblank(emu); break;
        default:
            lupanic(); break;
    }
```



```
    }  
}
```

此处的lupanic宏用于通知程序出错并崩溃程序，其通常用于标记一个永远不会被执行的分支。在正常情况下，由于PPUMode只有四个枚举值，因此default分支永远不会被触发。这是一种防御性编程的习惯，其用于确保在之后程序被部分修改时，程序的这个部分能够按照原先的预期正常运行。lupanic宏只在debug编译

模式下生效，在release模式下会被移除，以避免产生不必要的程序开销。

接着让我们先在PPU类中添加四种模式下的tick函数声明：

```
struct PPU  
{  
    /*...*/  
    void bus_write(u16 addr, u8 data);  
  
    void tick_oam_scan(Emulator* emu);  
    void tick_drawing(Emulator* emu);  
    void tick_hblank(Emulator* emu);  
    void tick_vblank(Emulator* emu);  
};
```

然后在PPU.cpp中添加实现。

在tick_oam_scan阶段中，我们需要读取GameBoy的OAM区域，并选出与本扫描行相交的精灵对象，这些对象需要在之后的tick_drawing阶段绘制。我们将在之后绘制精灵的章节中实现tick_oam_scan的完整功能，在现阶段我们只需要检查当前扫描行的时钟周期数是否超过80，并在超过时将模式切换到DRAWING模式就行：

```
void PPU::tick_oam_scan(Emulator* emu)  
{  
    // TODO  
    if(line_cycles >= 80)  
    {  
        set_mode(PPUMode::drawing);  
    }  
}
```

在tick_drawing阶段中，我们需要进行实际的绘制操作，包括背景、窗口和精灵的绘制。同样，我们会在之后的实际绘制章节中完成tick_drawing的完整功能，在现阶段我们只需要完成状态切换代码就行：

```
void PPU::tick_drawing(Emulator* emu)  
{  
    // TODO  
    if(line_cycles >= 369)  
    {  
        set_mode(PPUMode::hblank);  
        if(hblank_int_enabled())  
        {  
            emu->int_flags |= INT_LCD_STAT;  
        }  
    }  
}
```

在实际的绘制中，drawing的耗时在172至289个时钟周期不等，但是由于我们现在还没有进行实际的绘制操作，因此我们固定等待289个时钟周期，加上OAM_SCAN的80个时钟周期，在line_cycles为369的时候切换到HBLANK模式。在切换至HBLANK模式时，如果LCDS.3的值为1，则我们需要触发一个LCD_STAT中断。

在tick_hblank中，PPU不需要执行任何操作，而是等待CPU完成数据的更新。当当前行的耗时达到456个时钟周期时，HBLANK阶段结束，其会将LY的值增加1，然后根据新LY的值确定是重新回到OAM_SCAN状态开始绘制下一行，还是进入HBLANK状态，并根据新的状态触发相应的中断。最后，当模式切换时，HBLANK的代码会将line_cycles重置为0，以便开始新一行的计时：

```
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;
        }
    }
    else
    {
        set_mode(PPUMode::oam_scan);
        if(oam_int_enabled())
        {
            emu->int_flags |= INT_LCD_STAT;
        }
    }
    line_cycles = 0;
}
}
```

由于更新LY的值需要同步更新多个寄存器， 因此我们单独定义一个函数increase_ly来进行更新：

```
struct PPU
{
    /*...*/
    bool lyc_int_enabled() const { return !(lcds & (1 << 6)); }

    void increase_ly(Emulator* emu);

    void init();
    /*...*/
};
```

increase_ly的实现如下：

```
void PPU::increase_ly(Emulator* emu)
{
    ++ly;
    if(ly == lyc)
    {
        set_lyc_flag();
        if(lyc_int_enabled())
        {
            emu->int_flags |= INT_LCD_STAT;
        }
    }
    else
    {
        reset_lyc_flag();
    }
}
```

在将LY的值加1后，我们需要判断其是否等于LYC的值，并根据判断结果设置LCDS.2的值。同时，当LY==LYC时，如果LCDS.6的值为1，我们还需要触发一个LCD_STAT中断。

最后便是tick_vblank的实现。该函数与tick_hblank一样不执行任何操作，而是等待CPU完成数据的更新，并在计时结束时跳转到下一帧，重新进入第一行的OAM_SCAN模式：

```
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;
            if(oam_int_enabled())
            {
                emu->int_flags |= INT_LCD_STAT;
            }
        }
        line_cycles = 0;
    }
}
```

最后，我们同样需要修改Emulator类的代码，以接入我们的PPU组件。Emulator.hpp代码修改如

下:

```
/*...*/
#include "Serial.hpp"
#include "PPU.hpp"
using namespace Luna;

constexpr u8 INT_VBLANK = 1;
constexpr u8 INT_LCD_STAT = 2;
constexpr u8 INT_TIMER = 4;
constexpr u8 INT_SERIAL = 8;
constexpr u8 INT_JOYPAD = 16;

struct Emulator
{
    /*...*/
    Serial serial;
    PPU ppu;

    RV init(const void* cartridge_data, usize cartridge_data_size);
    /*...*/
};
```

Emulator.cpp代码修改如下:

```
RV Emulator::init(const void* cartridge_data, usize cartridge_data_size)
{
    /*...*/
    serial.init();
    ppu.init();
    return ok;
}

void Emulator::tick(u32 mcycles)
{
    u32 tick_cycles = mcycles * 4;
    for(u32 i = 0; i < tick_cycles; ++i)
    {
        ++clock_cycles;
        timer.tick(this);
        if((clock_cycles % 512) == 0)
        {
            // Serial is ticked at 8192Hz.
            serial.tick(this);
        }
        ppu.tick(this);
    }
}

u8 Emulator::bus_read(u16 addr)
{
    /*...*/
    if(addr == 0xFF0F)
    {
        // IF
        return int_flags | 0xE0;
    }
    if(addr >= 0xFF40 && addr <= 0xFF4B)
    {
        return ppu.bus_read(addr);
    }
    if(addr >= 0xFF80 && addr <= 0xFFFE)
    {
        // High RAM.
        return hram[addr - 0xFF80];
    }
    /*...*/
}

void Emulator::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr == 0xFF0F)
    {
        // IF
        int_flags = data & 0x1F;
        return;
    }
    if(addr >= 0xFF40 && addr <= 0xFF4B)
    {
        ppu.bus_write(addr, data);
    }
}
```

```
        return;
    }
    if(addr >= 0xFF80 && addr <= 0xFFFE)
    {
        // High RAM.
        hram[addr - 0xFF80] = data;
        return;
    }
    /*...*/
}
```

自此，我们就完成了PPU的初步接入工作。

查看显示内存的图块

虽然我们现在还没有实现完整的PPU功能，但是由于我们已经实现了PPU的初始化和tick逻辑，在程序来看，PPU实际上已经在正常工作了，因此大部分的游戏都可以正常将图块加载到显示内存中。在这一小节中，我们将实现一个简易的图块查看器，查看显示内存中保存的图块数据，以确定PPU是否正确实现。首先我们需要在DebugWindow中添加一个tiles选项卡，用于在调试面板中展示图块数据：

```
#pragma once
#include <Luna/Runtime/Vector.hpp>
#include <Luna/Runtime/String.hpp>
#include <Luna/Runtime/Ref.hpp>
#include <Luna/RHI/Texture.hpp>
using namespace Luna;

struct DebugWindow
{
    bool show = false;

    // CPU log.
    String cpu_log;
    bool cpu_logging = false;

    // Serial inspector.
    Vector<u8> serial_data;

    // Tiles inspector
    Ref<RHI::ITexture> tile_texture;

    void gui();
    void cpu_gui();
    void serial_gui();
    void tiles_gui();
};
```

RHI::ITexture接口表示LunaSDK的图形API中的一个纹理对象，其类似D3D11的ID3D11Texture2D对象，或者Vulkan中的VkImage对象。在显示图块时，我们会将解码后的图像数据保存至tile_texture纹理对象中，然后使用ImGui::Image方法来展示这个纹理，从而将图块显示在屏幕上。

tiles_gui的实现如下：

```
#include <Luna/Runtime/Log.hpp>
#include <Luna/RHI/Utility.hpp>

void DebugWindow::tiles_gui()
{
    if(g_app->emulator)
    {
        if(ImGui::CollapsingHeader("Tiles"))
        {
            u32 width = 16 * 8;
            u32 height = 24 * 8;
            // Create texture if not present.
            if(!tile_texture)
            {
                auto tex = g_app->rhi_device->new_texture(RHI::MemoryType::local,
                    RHI::TextureDesc::tex2d(RHI::Format::rgba8_unorm, RHI::Extent3d{
                        width, height, 1, 1}));
                if(failed(tex))
                {
                    log_error("LunaGB", "Failed to create texture for tile inspector");
                }
            }
        }
    }
}
```



```

        return;
    }
    tile_texture = tex.get();
}
// Update texture data.
usize num_pixel_bytes = width * height * 4;
usize row_pitch = width * 4;
Blob pixel_bytes(num_pixel_bytes);
u8* pixels = pixel_bytes.data();
for(u32 y = 0; y < height / 8; ++y)
{
    for(u32 x = 0; x < width / 8; ++x)
    {
        u32 tile_index = y * width / 8 + x;
        usize tile_color_begin = y * row_pitch * 8 + x * 8 * 4;
        for(u32 line = 0; line < 8; ++line)
        {
            decode_tile_line(g_app->emulator->vram + tile_index *
        }
    }
}
}
auto r = RHI::copy_resource_data(g_app->cmdbuf, {
    RHI::CopyResourceData::write_texture(tile_texture, {0, 0}, 0,
});
if(failed(r))
{
    log_error("LunaGB", "Failed to upload texture data for tile in
    ImGui::End();
    return;
}
// Draw.
ImGui::Image(tile_texture, {(f32)(width * 4), (f32)(height * 4)});
}
}
}

```

该函数主体分为两部分。首先，当我们第一次运行该函数时，由于`tile_texture`为空，所以我们会创建一个新的纹理对象，并保存在`tile_texture`中。根据上文可知，显示内存中用于存储图块的区域最多可以同时存储384个不同的图块，因此我们使用一个16x24的网格来预览这些图块，同时每一个图块具有8x8的像素尺寸，因此最终用于预览的纹理尺寸为128x192像素。在创建纹理时，我们指定了纹理的用处为`copy_dest`和`read_texture`，前者允许我们在之后将图像数据直接拷贝至纹理内存中，而后者允许我们在绘制GUI时采样该纹理。

在创建纹理以后，我们需要解码显示内存中的数据为使用RGBA8表示的像素信息，并保存在纹理数据中。由于我们创建的纹理内存分配在显存中，程序不可直接访问，因此我们需要先创建一个名为pixel_bytes的Blob对象，将数据先写入该对象中，然后再调用RH::copy_resource_data将数据从内存中拷贝至显存中。Blob是Binary Large Object的缩写，其用于分配一定的内存空间存储任意的二进制数据，并在对象销毁时自动清理分配的内存。在这里，我们分配了一个与纹理尺寸一致的Blob对象，然后将像素信息写入Blob对象的内存。

在解码图块数据时，我们采用了逐图块、逐行解码的思路。我们首先使用两个循环来遍历图块，使用x和y变量表示当前图块在纹理中的坐标，并推导出图块的索引tile_index。然后我们使用另一个循环来解码该图块的8行数据。根据上文的描述，我们知道图块的每一行数据使用连续的两个字节来表示，并且具有8个像素，在解码成RGBA8数据以后具有32字节的宽度。在这里我们使用一个单独的函数decode_tile_line来负责解码图块的单行数据：

```
inline void decode_tile_line(const u8 data[2], u8 dst_color[32])
{
    for(i32 b = 7; b >= 0; --b)
    {
        u8 lo = (!(data[0] & (1 << b)));
        u8 hi = (!(data[1] & (1 << b))) << 1;
        u8 color = hi | lo;
        // convert color.
        switch(color)
        {
            case 0: color = 0xFF; break;
            case 1: color = 0xAA; break;
            case 2: color = 0x55; break;
            case 3: color = 0x00; break;
            default: lupanic(); break;
        }
        dst_color[(7 - b) * 4] = color;
        dst_color[(7 - b) * 4 + 1] = color;
        dst_color[(7 - b) * 4 + 2] = color;
        dst_color[(7 - b) * 4 + 3] = 0xFF;
    }
}
```

$$\left. \begin{array}{l} \{ \\ \} \end{array} \right\}$$

该代码的逻辑与上文中图块解码的逻辑一致，对于每一个像素，其提取两个字节中的指定位，并组合成一个索引，然后根据索引的值来确定最终输出的颜色。聪明的读者可能已经看出来，由于显示内存中保存的只是颜色的色值，或者说是调色板的索引值，并非最终的颜色，因此我们在调试面板中显示的颜色与该图块最终显示出来的颜色并非是一致的，并且其也没有考虑精灵图块可能出现翻转的情况。但是作为调试功能来说，我们需要的是尽可能简单直观的展示数据，因此我们在这里使用了最简单的方式展示数据，不引入更多的复杂度。

tiles_gui函数最后的操作是调用ImGui::Image方法将上传好数据的纹理展示在GUI面板上，并且长宽都放大了4倍，以便我们能够更轻松看到图块的数据。最后我们将tiles_gui方法接入到gui方法中：

```
void DebugWindow::gui()
{
    if(ImGui::Begin("Debug Window", &show))
    {
        cpu_gui();
        serial_gui();
        tiles_gui();
    }
    ImGui::End();
}
```

编译并运行程序，找一个比较简单，只有单ROM的游戏文件打开，此时应该可以在调试面板中看到加载进显示内存的图块数据了。例如，如果打开《马里奥医生》的游戏，可以看到类似下面的图块数据：



以上就是本章节的全部内容了。在下一章中，我们将开始真正实现PPU的绘制逻辑。下一章我们

将首先实现PPU的背景和窗口绘制，然后在之后的一章中实现PPU的精灵绘制。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #8 绘制背景和窗口

20 赞同 · 0 评论 文章

编辑于 2024-02-15 15:02 · IP 属地浙江

Game Boy（GB）

游戏机模拟器



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



寻找25年前的印记，细数手机平台上的模拟器

朱小杰

发表于触乐



街机模拟器Mame 记载着你我的青葱岁月，让我们开始无限

囧王者

常用安卓模拟器介绍？PC模拟器哪个好

在电脑上玩手游，用哪个PC安卓模拟器好用？现在的手机模拟器有多款，涉及很多个厂商，很多用户使用的时候总归出现各种问题，最普遍的就是卡顿无影响等问题，因为用户不可能将所有的模拟器都...

走着走着就累了



网页里的电脑博物馆：模拟器≠盗版！模拟器玩家的正版指南

蓬岸 Dr.Quest

▲ 已赞同 37



● 添加评论

🚩 分享

❤ 喜欢

★ 收藏

📄 申请转载

