GameBoyEmulator 项目手册

简介

本**Gameboy模拟器**由**CPU**、**Window**、**内存**、**文件操作系统**、**Timer**五部分组成。

CPU主要负责进行命令的执行、中断的判断
Window主要负责进行图像的呈现、输入的处理
内存主要负责游戏中的内存模拟
文件操作系统主要负责游戏ROM的读取。
Timer主要负责游戏中时间的同步

目前本**GameBoy模拟器**依然存在着一些bug，导致游戏虽然可以正常运行显示标题、选关、等功能，但会在选完关卡后卡住。虽然如此，本**GameBoy模拟器**已经基本完成了以下基本功能：

- 实现 Gameboy Z80 CPU 模拟
- 实现时钟模拟
- 实现内存模拟
- 支持基本图形操作
- 支持对游戏进行交互操作
- 可以载入ROM

和以下附加功能：

- Background Window Graphic
- Sprite

游戏玩法

载入游戏

在exe文件夹中，附带了测试用游戏*tank.gb*，还在*Games*文件夹里准备了几个游戏 **Gameboy模拟器**有两种载入游戏的方法。
第一种：直接打开*GameBoyEmulator.exe*,输入 `tank.gb`
第二种：用参数传入文件名，在命令行输入 `GameBoyEmulator.exe tank.gb`

游戏按键

在游戏中**上下左右**键对应键盘的**上下左右**键
（你在按键后可能需要等待片刻来等待程序处理）

- **SPACE – SELECT键**
- **ENTER – ENTER键**
- **Z – A键**
- **X – B键**

项目编写过程

难点

- 由于没有相应的框架，需要自己搭建大体的结构。
- 由于参考手册中没有给出CPU指令的具体实现方法，需要去网上学习汇编指令的实现方法
- 由于知识水平不够，HBlank、VBlank等概念难以理解，需要学习Interrupt等新概念
- 一直以为屏幕上的像素以位图形式存于RAM中，实际上是以TileMap形式存在，且共有两个TileMap需要切换，需要自己实现TileMap，所以花了很长时间理解、编写显示部分
- CPU 部分的DEBUG需要手动DEBUG，占用了大部分的时间，显示部分的DEBUG由于GB游戏相当于黑盒，没有能力看懂其实现部分，所以DEBUG也比较困难
- SDL库的学习
- 写timer时关于所需要输入和输出的内容参考资料不足，导致起初对于类中的函数无从下手。

踩过的坑

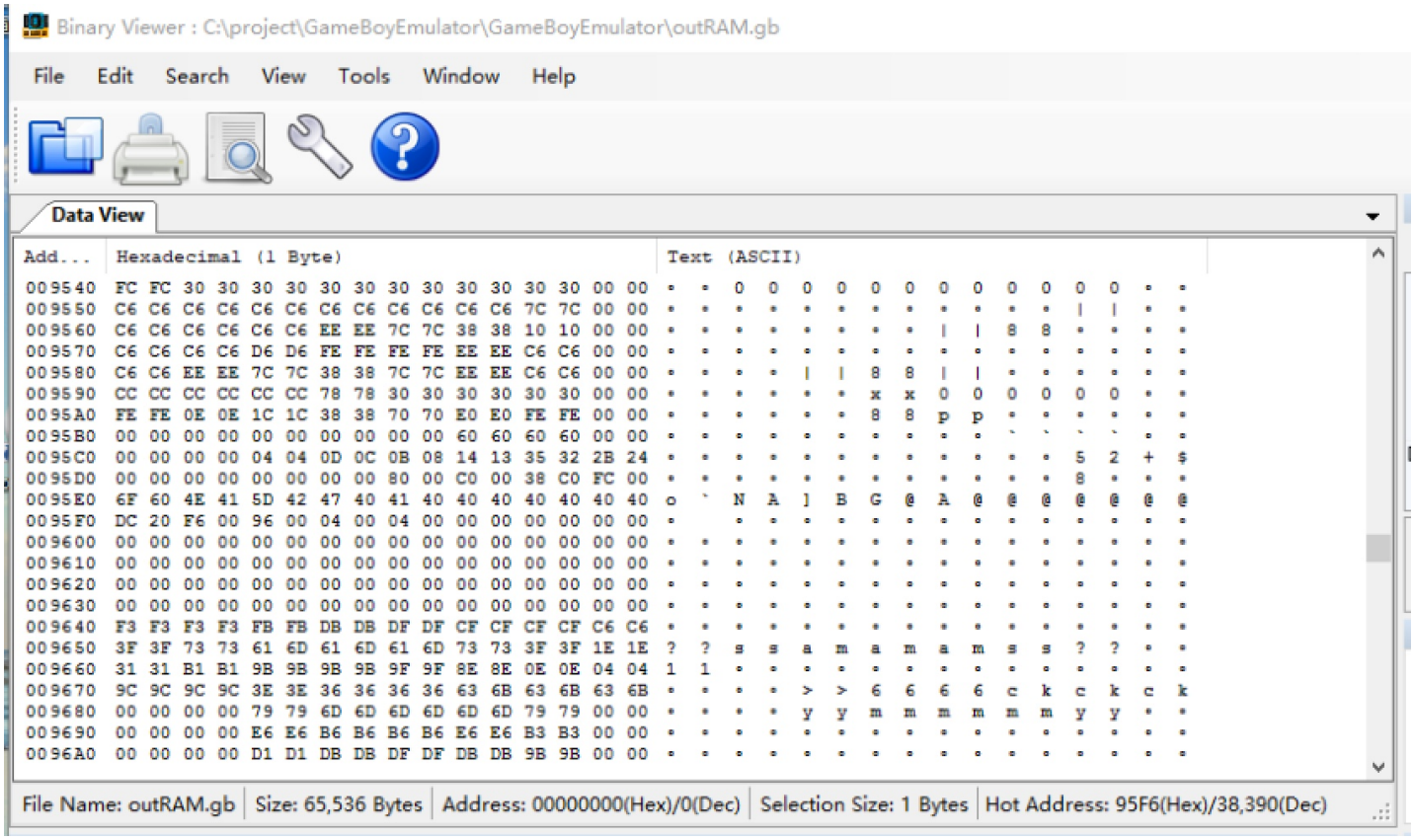
- Gameboy参考手册中关键地方有错字，而且有些地方解释不清
- SDL库中Surface与Texture 和 Renderer有冲突，最终去除了Texture和Renderer，只保留了Surface来显示像素
- CPU中的 F Register 与 CPU中的 Flag 有关联，一开始没有注意到导致了奇奇怪怪的BUG
- CPU指令中使用了lambda表达式，对捕获变量列表处理不当导致出现读取BUG
- CPU指令中关于位运算的部分有符号错误，难以察觉，花了好长时间才最后找出来
- 对Key的输入部分没有理解，导致曾经虽然加了key部分，也能检测输入，但就是没有反应
- 一开始没有注意到有一些运算是带符号运算，全都使用了unsigned变量，导致出现BUG
- 在读取CPU参数命令时应该使用LSB（低位优先），我没有注意到导致程序没有正常运行
- stack的出入顺序
- 没有注意到一次只能改变一位数据导致写入数据时浪费了很多时间。
- 某些函数只接受引用传值，一开始没有注意到从而报错。

DEBUG

在编写**GameBoy模拟器**时，使用了#ifdef来决定在DUBUG模式下输出。
使用了三种debug途径：

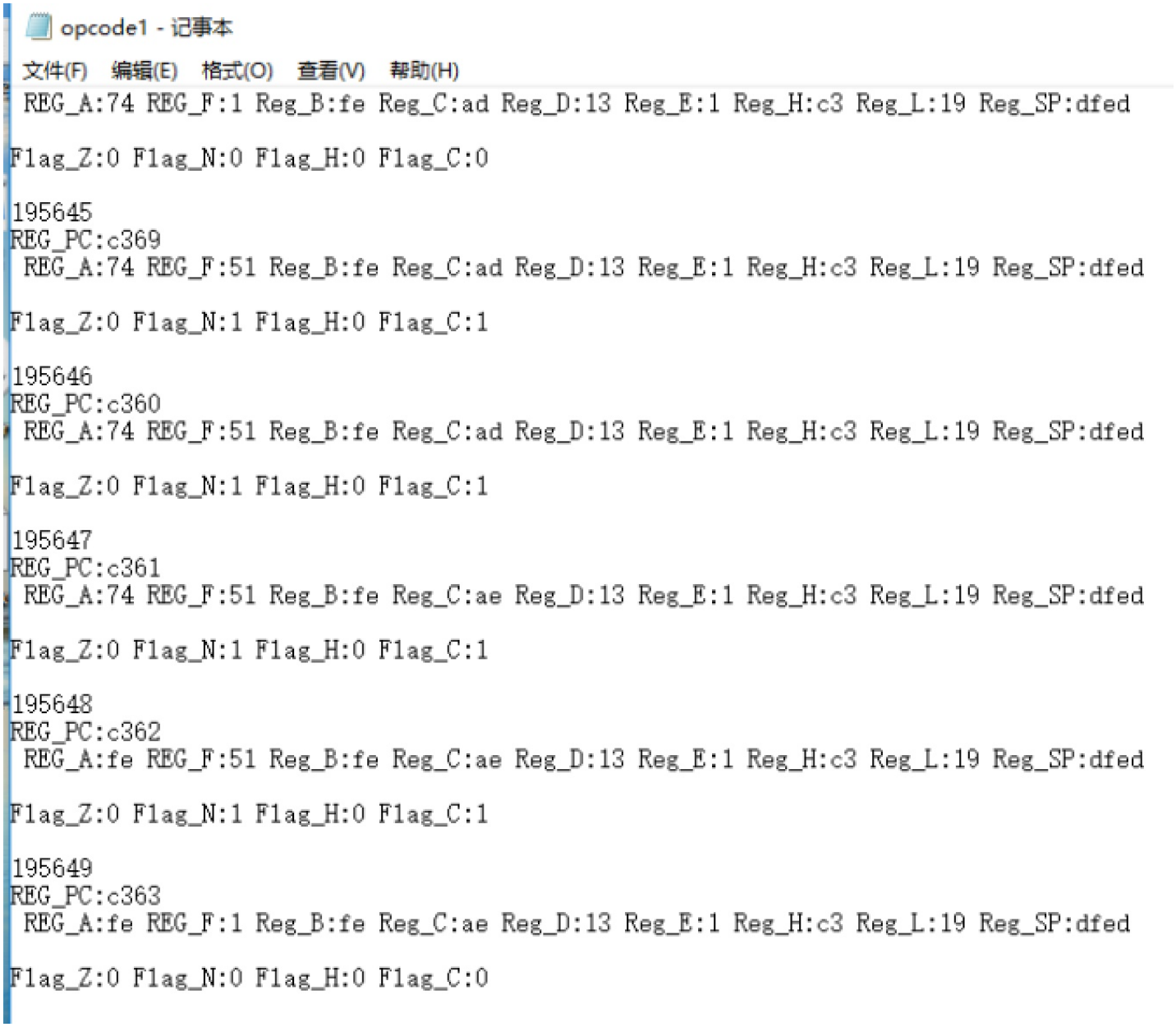
1.输出RAM

在程序运行了一定量代码后输出RAM，与no$gmb的RAM对比



2.输出CPU Log

每执行一条指令输出一条Log，包含当前执行命令、CPU的Registers值
便于对应no$gmb进行单步调试



3.输出Real-Time-Debug

在程序运行时 执行到某命令时ASSERT，同时查看各数据的值

项目结构

本**GameBoy模拟器**包含了如下几个模块：

- `main`
  包含了**命令行的处理**、**程序的打开**和**CPU主循环**
- `GB_Type`
  包含了 `GB_Byte` 、 `GB_DoubleByte` 、 `GB_SByte` 三个基本数据类型的声明
- `GB_Bits`

包含了 `SetBit` 和 `GetBit` 函数,用于设定和读取特定位的值

- `GB_Const`
  包含了**Register**的地址和各种常用的常量

- `GB_CPU`
  包含了主循环处理函数 `CPU_Step` 和命令码的模拟实现。
  命令全部用**lambda表达式**呈现，易读性强
  `GB_Opcode` 和 `GB_CBOpcode` 地址数组中存放了所有命令函数的地址。
  其中包括了CPU的自有Flag和操作命令 `SetFlag` 和 `GetFlag`

- `GB_FileOperator`
  包含了GB文件的读取和写入内存的功能。

- `GB_Memory`
  包含了模拟内存的读取、写入功能

- `GB_Timer`
  包含了模拟Timer的功能

- `GB_Window`
  包含了**BackGroundGraphic**、**Sprite**的显示和按键的检测。

项目逻辑

在程序运行开始时，由 `FileOperator` 打开**GameBoy ROM**文件，并导入内存 `memory_` 中，对**CPU** 和**Window**进行初始化后，运行**主循环**

```
while (1) {
        if (MainWindow.fresh())//get KEYS
        {
            int timing = CPU.CPU_Step();//get TIMING
            MainWindow.AddTime(timing);//fresh WINDOW
            GB_Timer.increase(timing);//fresh TIMER
        }
        else
            break;//quit
    }
```

在主循环里，每一个周期依次进行获取按键、执行CPU命令、同步显示、同步时间

获取到的按键会写入 `GB_Memory` 中的Keys变量中以便读取

同步显示可将显示系统在 OAM、VRAM、HBlank、VBlank 四个模式中切换并渲染画面

函数介绍

GB_Bits

```
void SetBit(GB_Byte &Byte, int pos, bool value)
```

设定 `Byte` 中第 `pos` 位为 `value`

```
bool GetBit(GB_Byte &Byte, int pos)
```

返回 `Byte` 中第 `pos` 位的值

GB_CPU

```
 void init()
```

对CPU按照手册进行初始化

```
int CPU_Step()
```

判断中断并执行下一个命令,返回命令所花费的时间数

```
void Opcode_load()
```

将命令载入 `GB_Opcode` 和 `GB_CBOpcode` 函数地址数组

```
void SetFlag(int flag, bool value)
```

设置CPU中 Z、N、H、C Flag的值

```
bool GetFlag(int flag)
```

获取 Z、N、H、C Flag的值

GB_FileOperator

```
 int OpenFile(string sfile)
```

打开 `sfile` 目录下的GB文件并以二进制方式读入内存

```
vector<GB_Byte> GetData()
```

返回一个vector，里面包含了GB文件的ROM

GB_Memory

```
void LoadMemory(const vector<GB_Byte> &inRom)
```

读取ROM内存

```
GB_Byte ReadByte(GB_DoubleByte Address)
```

返回内存 `Address` 地址中的Byte

```
bool WriteByte(GB_DoubleByte Address, GB_Byte Value)
```

将 `Value` 写入内存 `Address` 地址

GB_Window

```
 SpriteInfo(int id)
```

SpriteInfo类的初始化函数，获取id为 `id` 的精灵的信息。

```
void create (int WINDOW_WIDTH, int WINDOW_HEIGHT, int x, int y, string WINDOW_TITLE)
```

创建一个宽度为 `WINDOW_WIDTH` ,高度为 `WINDOW_HEIGHT` ,标题名为 `WINDOW_TITLE` 的窗口，显示在屏幕的 `(x,y)` 处

```
void AddTime(int clocks)
```

将Window的内置时钟时间增加 `clocks`

```
void setPixel(int x, int y, int color)
```

将buffer中 `(x,y)` 处的像素颜色设置为 `(R,G,B):(color,color,color)` ,

```
int fresh()
```

Window的周期时间，获得键盘输入

```
 void draw()
```

将buffer中的内容显示

```
void drawLine(int ly)
```

获取第 `ly` 行所需要显示的背景、精灵内容，写入buffer中

```
vector<SpriteInfo> getSprites(int ly)
```

获取第ly行中所需要渲染的精灵

```
void SetMode(int mode)
```

设置当前模式为 `mode`
其中
0：**HBlank mode** –将一行内容读入Buffer，待渲染行+1
1：**VBlank mode** –渲染整个屏幕，经过10个周期后，待渲染行=0
2：**OAM mode** –读取精灵内容
3：**VRAM mode** –读取显示内容

```
void updateLyc()
```

更新LYC Registor中的值