从零开始实现GameBoy模拟器 #15 高通滤波器，总结

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模拟器最终章！在本章中，我们将为APU添加高通滤波器，以解决音频信号原点不为0的问题，然后回顾我们到目前为止所学习的全部内容，并对本专题进行总结，。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-15，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

高通滤波器

由于GameBoy的硬件限制，当GameBoy APU的某个通道处于关闭状态，但是其DAC并没有关闭时，其输出的电平并不是0，而是其保持其最后一次更新后输出的信号。例如，如果通道1在输出0xE信号时，通道1被禁用，但是DAC没有关闭（NR12.3~7任意位不为0），则通道会一直保持输出0x0E信号，即数模转换后的+0.867电平。由于信号在通道被关闭后稳定不变，因此其并不会产生任何声音（声音需要通过变化的信号来产生），但是该信号会导致最终输出的信号并不是以0为振动中点，而是会有一定的偏差，并导致最终的信号产生轻微的失真。

为了解决这个问题，GameBoy的混音器中带有一个高通滤波器（high-pass filter），该滤波器可以检测信号中频率低于某个特定阈值的部分，并将该部分从音频信号中移除。由于关闭的通道信号保持不变，也可以认为其频率为0，因此高通滤波器可以移除这部分导致音频信号整体偏差的恒定信号，将音频信号重新设置为以0为振动中点。

GameBoy APU的高通滤波器针对模拟信号进行操作，且并没有文档介绍其具体实现，因此我们需要自己选择算法实现高通滤波器。常见的高通数字滤波算法包括滑动平均、IIR、FIR等，这里我们使用最简单的滑动平均算法进行滤波，该算法的思路是计算当前音频样本之前一段时间内音频样本的平均值，将平均值认为是声音信号的恒定偏差，并从最终输出的音频样本中减去该平均值，从而消除恒定偏差。在掌握了滑动平均实现的高通滤波器后，读者也可以尝试使用IIR、FIR等算法实现高通滤波器，并比较不同滤波器产生的听感差异。

为了实现高通滤波器，我们首先需要定义用于存储当前音频样本之前一段时间内音频样本的缓存：

```
struct APU
{
    /*..*/
    void tick_ch4(Emulator* emu);

    //! Used for high-pass filtering.
    u8 history_samples_l[65536];
```

```
    u8 history_samples_r[65536];
    u16 history_sample_cursor;
    u32 sample_sum_l;
    u32 sample_sum_r;

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

可以看到，我们使用history_samples_l和history_samples_r为左右声道分别定义了音频历史缓存，长度均为65536，即大约1/16秒的音频数据。由于每个音频数据最多只会有60个不同的数值（每个通道具有0~15的数字输出，四个通道加起来为0~60），因此我们可以使用一个8位无符号整型来保存历史音频数据，以节省历史音频缓存所占用的空间。其次，我们使用history_sample_cursor来指向音频历史缓存中需要更新的下一个音频采样。在工作的时候，history_sample_cursor的值会在0~65535间循环，从而将在每次更新时覆盖历史缓存中最老的音频采样值。最后，我们使用sample_sum_l和sample_sum_r分别统计history_samples_l和history_samples_r中所有样本的值之和，从而可以在每一次输出音频采样时快速计算平均值。在APU初始化的时候，上述所有变量的值均会清零。

接着我们在APU::tick中添加滤波的实现代码：

```
void APU::tick(Emulator* emu)
{
    if(!is_enabled()) return;
    // DIV-APU is ticked at 4194304 Hz.
    tick_div_apu(emu);
    // APU is ticked at 1048576 Hz, trus has 1048576 sample rate.
    if((emu->clock_cycles % 4) == 0)
    {
        /*..*/
        // Volume control.
        // Scale output volume to [-1, 1].
        sample_l /= 4.0f;
        sample_r /= 4.0f;
        sample_l *= ((f32)left_volume()) / 7.0f;
        sample_r *= ((f32)right_volume()) / 7.0f;
        // Write to histroy buffer.
        sample_sum_l -= history_samples_l[history_sample_cursor];
        sample_sum_r -= history_samples_r[history_sample_cursor];
        history_samples_l[history_sample_cursor] = (u16)((sample_l + 1.0f) / 2.
        history_samples_r[history_sample_cursor] = (u16)((sample_r + 1.0f) / 2.
        sample_sum_l += history_samples_l[history_sample_cursor];
        sample_sum_r += history_samples_r[history_sample_cursor];
        history_sample_cursor = (history_sample_cursor + 1) % 65536;
        // High-pass filter.
        f32 average_level_l = ((((f32)sample_sum_l) / 65536.0f) / 60.0f * 2.0f)
        f32 average_level_r = ((((f32)sample_sum_r) / 65536.0f) / 60.0f * 2.0f)
        sample_l -= average_level_l;
        sample_r -= average_level_r;
        // Prevent sample value over [-1, 1] limit.
        sample_l = clamp(sample_l, -1.0f, 1.0f);
        sample_r = clamp(sample_r, -1.0f, 1.0f);
        // Output samples.
        /*..*/
    }
}
```

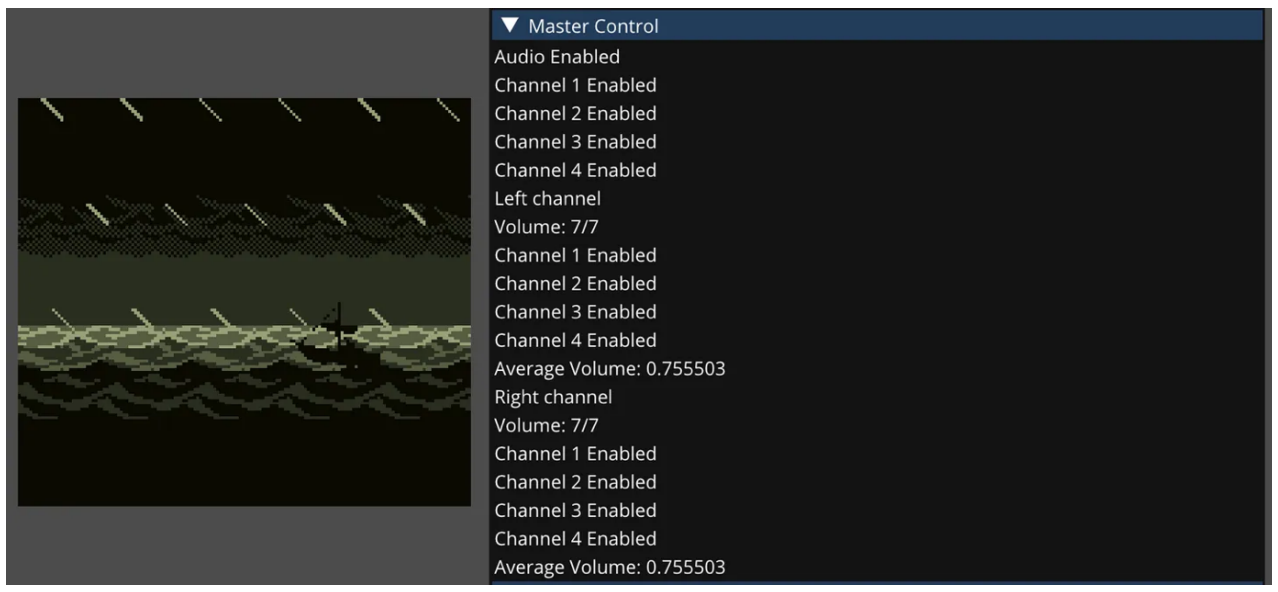
可以看到，滤波器介入的时机是在混音之后，输出音频信号到音频API之前。在执行滤波操作时，函数首先将当前帧的采样值转换成0~60的整数，并使用该整数更新历史音频缓存和样本和，同时将history_sample_cursor指针移动一位。接着，函数将样本和除以样本总数（65536），并重新转化成在-1至1之间的浮点数，得到历史缓存中所有样本的平均值；最后，函数使用该平均值调整输出的音频信号值，通过将输出的信号减去该平均值，从而去除输出信号中的恒定偏差。

为了更方便地观察左右声道的音频样本平均值，并预览通道3和通道4的总控参数，我们可以修改DebugWindow::apu_gui函数，添加对所有声道样本平均值以及通道3、通道4总控参数的预览：

```
void DebugWindow::apu_gui()
{
    if (g_app->emulator)
    {
        if (ImGui::CollapsingHeader("APU"))
        {
            if(ImGui::CollapsingHeader("Master Control"))
            {
```

```
                ImGui::Text("Audio %s", g_app->emulator->apu.is_enabled() ? "En
                ImGui::Text("Channel 1 %s", g_app->emulator->apu.ch1_enabled()
                ImGui::Text("Channel 2 %s", g_app->emulator->apu.ch2_enabled()
                ImGui::Text("Channel 3 %s", g_app->emulator->apu.ch3_enabled()
                ImGui::Text("Channel 4 %s", g_app->emulator->apu.ch4_enabled()
                ImGui::Text("Left channel");
                ImGui::Text("Volume: %u/7", (u32)g_app->emulator->apu.left_volu
                ImGui::Text("Channel 1 %s", g_app->emulator->apu.ch1_l_enabled(
                ImGui::Text("Channel 2 %s", g_app->emulator->apu.ch2_l_enabled(
                ImGui::Text("Channel 3 %s", g_app->emulator->apu.ch3_l_enabled(
                ImGui::Text("Channel 4 %s", g_app->emulator->apu.ch4_l_enabled(
                ImGui::Text("Average Volume: %f", ((f32)g_app->emulator->apu.
                ImGui::Text("Right channel");
                ImGui::Text("Volume: %u/7", (u32)g_app->emulator->apu.right_vo
                ImGui::Text("Channel 1 %s", g_app->emulator->apu.ch1_r_enabled(
                ImGui::Text("Channel 2 %s", g_app->emulator->apu.ch2_r_enabled(
                ImGui::Text("Channel 3 %s", g_app->emulator->apu.ch3_r_enabled(
                ImGui::Text("Channel 4 %s", g_app->emulator->apu.ch4_r_enabled(
                ImGui::Text("Average Volume: %f", ((f32)g_app->emulator->apu.
                }
                /*..*/
            }
        }
    }
```

此时编译并运行模拟器，就可以看到高通滤波器已经正常开始工作了：



至此我们就完成了PPU部分的所有功能的实现，并完成了本专题中GameBoy模拟器的所有开发内容，恭喜！

专题总结

在本专题中，我们一步一步地学习了GameBoy的各个组件，并通过编写代码模拟了这些组件的功能，这些组件包括：

1. 时钟和总线
2. CPU寄存器和指令执行
3. CPU中断
4. 计时器
5. 串口通信
6. 卡带读写，包括对MBC、持久存储、实时时钟功能的支持
7. 像素处理单元（PPU）
8. 音频处理单元（APU）
9. 按键输入

最终，我们将组件通过总线和时钟串起来，完成了自己的GameBoy模拟器。相信读者在这趟旅途中一定有了不少的收获，并摩拳擦掌准备开始实现自己的模拟器了！好，那么接下来是本专题的课后作业：

实现GBC模拟器

GameBoy Color（简称GBC，内部代号CGB）是GameBoy的后续机型，其最大的升级就是将GameBoy的单色液晶屏升级为了背光彩色屏。GBC的硬件架构与单色GameBoy高度相似，其同样采用了基于Z80的LR35902处理器，所有寄存器、组件的设计也与单色GameBoy基本兼容，因此我们可以很容易地扩展我们的模拟器来支持GBC游戏的模拟运行。具体来说，GBC相比单色GameBoy的主要升级包括：

1. GBC提供了两种不同的时钟频率：与单色GameBoy一致的4.194MHz以及正好翻倍的8.388MHz（称之为double speed mode）。前者主要用于以兼容模式运行单色GameBoy游戏卡带，而后者则用于运行GBC独占的新游戏。相比单色GameBoy的时钟频率，翻倍的时钟频

率将游戏机的处理性能提升了一倍，从而支持更为复杂的游戏逻辑和彩色画面的绘制需求。

2. GBC将板载WRAM和VRAM的容量分别从单色GameBoy的8KB提升到了32KB和16KB，并使用类似MBC的思路，通过寄存器设置可以将不同的内存分块动态映射到总线地址上，从而保持地址映射规则与单色GameBoy的兼容性。
3. PPU支持的调色板数量从0+2增加到了8+8，即背景和精灵的每一个图块都有8个独立的调色板可供选择。同时，每一个调色板可以保存四种不同的彩色，并通过与单色GameBoy类似的方式指定每一个色值对应的颜色。
4. GBC添加了对MBC5~MBC7卡带的支持。理论上来说，由于MBC卡带的支持只与卡带本身有关，因此单色GameBoy也同样支持MBC5~MBC7卡带，但是大部分的MBC5~MBC7卡带均为GBC卡带，因此我们也可以将其看成是GBC新增功能的一部分。

除此之外，GBC在诸多特性的实现细节上均与单色GameBoy有所差异，读者可以参考下面的文档来详细了解这些差异：

https://gbdev.io/pandocs/CGB_Registers.html
gbdev.io/pandocs/CGB_Registers.html

以上就是《从零开始实现GameBoy模拟器》专题的全部内容了，感谢诸位读者这两个月以来的陪伴，我们下个专题再会！

《完》

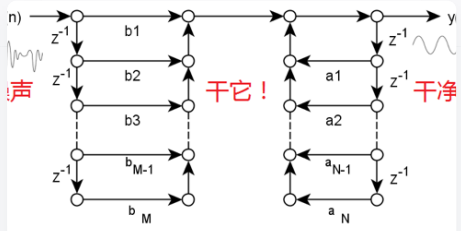
发布于 2024-03-19 21:03 · IP 属地上海

游戏机模拟器 Game Boy（GB）

飞控IMU数据进阶处理（FFT，滤波器）

前面的文章曾简单讲过IMU数据（陀螺仪、加速度数据）的校准以及一阶低通滤波。本文在此基础上更进一步讲一下数据的指标分析与滤波器的选择问题。 IMU数据的重要性IMU数据在飞控中处于最底

acros... 发表于飞控那些事...



手把手教系列之IIR数字滤波器设计实现

逸珺 发表于嵌入式客栈

（建议收藏）一文读懂RC滤波设计全过程

今天跟大家分享一篇关于RC滤波器设计的文章，在嵌入式系统中可以说,"无滤波器，不嵌入式"，各种传感器信号多多少少会携带一些噪声信号，那么通过滤波器就能够更好的降低和去除噪

张飞实战电子

（建议收藏）一文读懂RC滤波设计全过程

今天跟大家分享一篇关于RC滤波器设计的文章，在嵌入式系统中可以说,"无滤波器，不嵌入式"，各种传感器信号多多少少会携带一些噪声信号，那么通过滤波器就能够更好的降低和去除噪

X学无止境

▲ 赞同 28 ▼ 3 条评论 分享 喜欢 收藏 申请转载 ···