Projektdokumentation

Gameboy Emulator

Matthias Merkl
BTIN2
Informatiktechnik
Technikerschule München

Inhaltsangabe

Tabellenverzeichnis

Abbildungsverzeichnis

Hinführung

Im Rahmen der Projektarbeit im Abschlussjahr des Informatiktechnikers an der Technikerschule München, habe ich mich dazu entschieden in Eigenregie einen Gameboy Emulator zu konzipieren und die Programmierung davon umzusetzen.

Die folgenden Abschnitte sind grundsätzlich in Sektionen zu den einzelnen Hardwarebestandteilen des Gameboys aufgebaut. Innerhalb jedes Abschnittes wird zunächst auf die grundsätzlichen technischen Gegebenheiten eingegangen und jeweils im Anschluss ein vereinfachter Einblick in die programmiertechnische Umsetzung verschafft. So soll gewährleistet sein, dass der Leser zunächst ein Verständnis für die jeweilige Komponente entwickelt und im Anschluss anhand der Implementierung in C++ das gesamte Projekt nachverfolgen kann.

Gameboy Emulator

Projektstruktur

Das Projekt kann über folgenden Link bezogen werden:

GameboyX Emulator

Oder direkt über folgendes Kommando:

```
git clone --recurse-submodules https://github.com/MatthewMer/gameboyx.git
```

Grundsätzlich ist das Projekt in drei Bereiche untergliedert. Die Klasse GuiMgr für das Management des User Interfaces und die Steuerung der Interaktion mit den anderen Komponenten, die Klasse HardwareMgr für die Ansteuerung der physisch vorhandenen Hardware und die Klasse VHardwareMgr für die Initialisierung und Steuerung der emulierten Hardware. Bei sämtlichen Klassen handelt es sich um Singletons, da jeweils lediglich eine Instanz gebraucht wird, wie in Abbildung 2 zu sehen. Diese Verwalten jeweils wiederum ihnen untergeordnete Instanzen der Klassen der physischen und virtuellen Hardware, oder interagieren über die übergeordneten Instanzen mit den Instanzen der anderen Kategorien, um trotz der vielen Zugriffe auf die Funktionalitäten eine Grundstruktur zu erhalten.

Das GUI besteht aus einer einfachen Menüleiste, um dem Benutzer die verschiedenen implementierten Funktionen zur Verfügung zu stellen und aus einer Tabelle mit den vom Benutzer hinzugefügten Spielen, wie in Abbildung 1 zu sehen. Diese werden automatisch in einen Ordner bei der Executable kopiert.

Abbildung 1: Menü

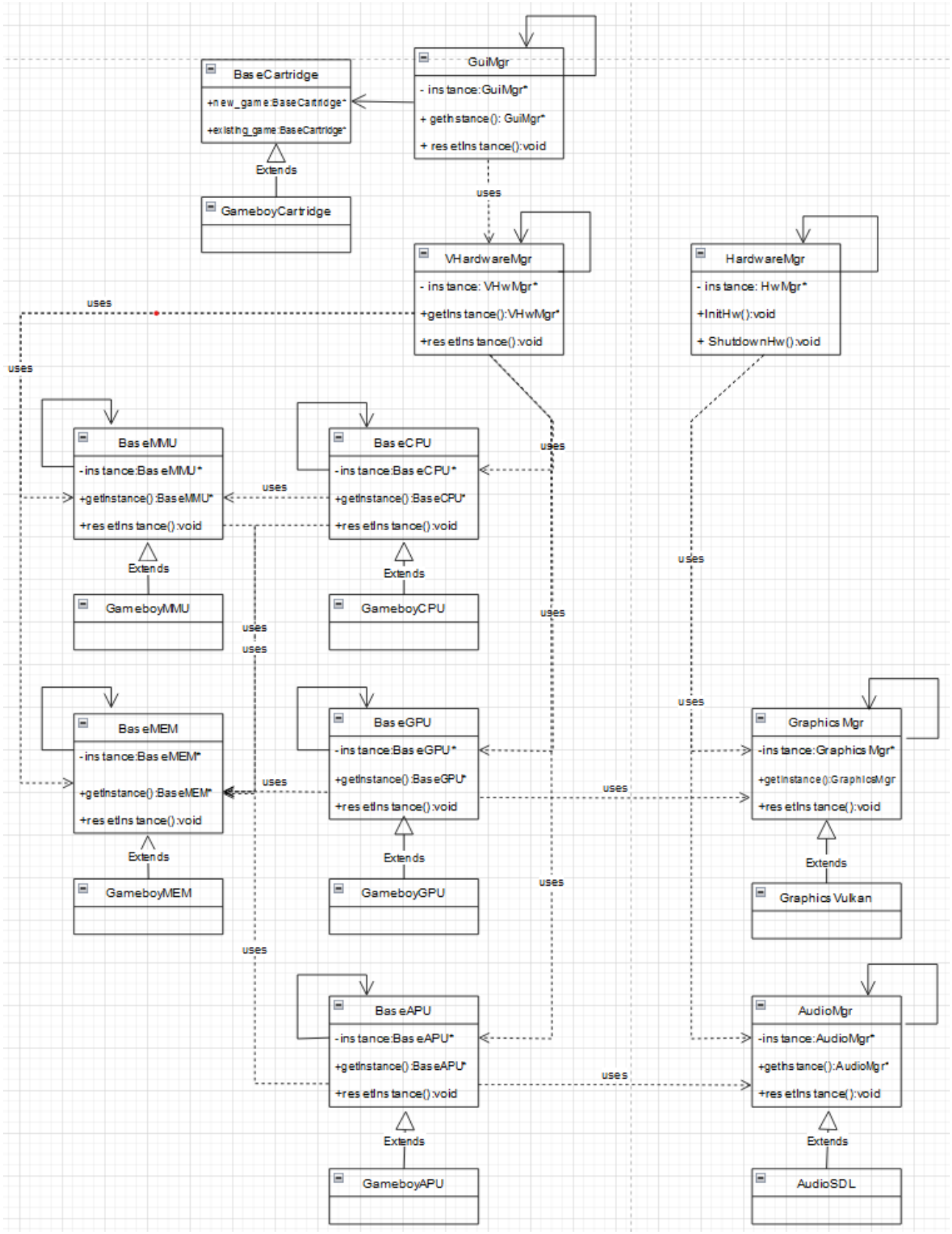

Abbildung 2: UML-Diagramm

CPU

Technische Details

Im folgenden wird zunächst auf die grundlegenden technischen Gegebenheiten des Gameboys eingegangen, der Grund dafür ist dem Leser einen generellen Überblick über die vorliegende Hardware zu vermitteln und um die Implementierung in den darauf folgenden Abschnitten zugänglicher zu machen.

Basis Takt	4.194304 MHz
Adress-Bus	16 Bit (parallel)
ALU	8 Bit (intern 4 Bit mit Halfcarry)
Register	16 Bit

Tabelle 1: Sharp SM83

Wie in Tabelle 1 zu sehen arbeitet die Gameboy CPU, ein Sharp SM83, welcher eine Art Mischung aus Intel 8080 und Z80 [Z80] ist, mit kleineren Unterschieden beim Instruction Set und technischen Details. Er arbeitet bei einer Basis Takt Frequenz von 4.194304 MHz [SM83 Specs], wobei es sich wiederum um eine 2er Potenz handelt:

2^22 Hz = 4194304 Hz = 4.194304 MHz

In vielen Ressourcen ist hierbei oft von einer Takt Frequenz von 1.048576 MHz die Rede. Dies hängt damit zusammen, dass viele Prozesse im Gameboy in Schritten von 4 Clock Cycles stattfinden, weswegen auf ein Viertel der Basis Takt Frequenz verwiesen wird. Daher soll in diesem Dokument der Begriff „Machine Cycle“ für 4 Clock Cycle bei 4.194304 MHz, also 1.048576 MHz, verwendet werden, um etwaiger Intransparenz vorzubeugen. „Clock Cycle“ beziehen sich nach wie vor auf den genannten Basistakt von 4.194304 MHz.

Des Weiteren verfügt der Gamboy über einen 16 Bit Adress-Bus, um einen Speicherbereich von 0x0000 bis 0xFFFF bzw. 65536 Bytes = 64 KiB adressieren zu können. Viele Spiele verfügen allein im Bezug auf ROM bis zu 128 KiB, dabei setzt der Gameboy auf verschiedene Techniken um diesen Speicherbereich drastisch zu erhöhen und die Limitierung eines 16 Bit Adress-Busses zu umgehen. Darauf soll im Abschnitt zur Speicherimplementierung genauer eingegangen werden.

Die Kernkomponente zur Ausführung von Prozessorinstruktionen stellt die ALU (Arithmetic logic unit) dar. Wie beim Z80 handelt es sich dabei intern um ein 4 Bit Register, bei Berechnungen werden zuerst die unteren 4 Bit (Nibble) verarbeitet und im Anschluss die oberen 4 Bit. Dies ist erst durch die Verwendung eines Halfcarry Flags möglich, um beispielsweise Überläufe bei Additionen der unteren 4 Bits zwischenzuspeichern.

Register und Flags

Sämtliche Register der CPU, wie in Tabelle 2 zu sehen, sind 16 Bit lang, wobei sie in der Regel in zwei 8 Bit große Teilregister unterteilt sind. Sie können in Kombination (alle 16 Bits) oder die jeweils unteren und oberen 8 Bit separat angesteuert werden.

AF	Akkumulator Bits 15 bis 8 Flags Bits 7 bis 0
BC	B Register Bits 15 bis 8 C Register Bits 7 bis 0
DE	D Register Bits 15 bis 8 E Register Bits 7 bis 0
HL	H Register Bits 15 bis 8 L Register Bits 7 bis 0
SP	Stack Pointer 16 Bits
PC	Program Counter 16 Bits

Tabelle 2: Register [SM83 Register]

Das AF Register besteht zum einen aus dem Akkumulator (obere 8 Bit). In diesem wird jeweils 1 Byte zur Ausführung von Prozessorinstruktionen gespeichert. Falls bei einer Instruktion nicht anderweitig angegeben, bezieht sich die Instruktion stets auf den Inhalt des Akkumulators. Intern arbeitet die ALU, wie zuvor beschrieben, in 4 Bit mit dem jeweiligen Low und High Nibble.
Die unteren 8 Bit des AF Registers stellen dabei die Flags dar, welche bei der Ausführung der Instruktion entsprechend gesetzt werden. Wichtig ist dabei zu berücksichtigen, dass lediglich die oberen 4 Bit als Flag Bits verwendet werden, die unteren 4 Bits werden ignoriert und besitzen keine Funktion. Die Flags, die hierbei - abhängig von der Instruktion - gesetzt werden können sind Tabelle 3 zu entnehmen.

Zero Flag	Bit 7
Subtraction Flag	Bit 6
Half Carry Flag	Bit 5
Carry Flag	Bit 4

Tabelle 3: Flags

Das Subtraction Flag und das Half Carry Flag werden zwar von einer Vielzahl von Instruktionen gesetzt, werden aber lediglich von der BCD Instruktion (DAA: Opcode 0x26) [DAA] zur Verarbeitung von Dezimalzahlen in Binärcode (BCD-Code) ausgewertet. Die Berücksichtigung der internen Funktionsweise der 4 Bit ALU außen vorgenommen.
Das Zero Flag und das Carry Flag finden wiederum in vielen Instruktionen Anwendung und werden bei der Berechnung berücksichtigt.

Die Register BC, DE und HL haben generell keine spezielle Funktionsweise und werden zur Zwischenspeicherung von Werten verwendet. Einzig das HL Register wird bei zwei Instruktionen zur Speicherung von Adressen, welche auf jeweils 8 Bit im Speicher verweisen, verwendet um über Speicherbereiche iterieren zu können ohne auf eine zusätzliche Instruktionen zur Inkrementierung und Dekrementierung der Adresse zurückgreifen zu müssen (Opcodes: 0x22, 0x32, 0x2A, 0x3A).

Der 16 Bit Stackpointer verweist stets auf die nächsten freien 16 Bit im Stack, welcher sich grundsätzlich im HRAM befindet. Er kann jedoch nach belieben auch auf andere Speicherbereiche verweisen. Einige Spiele verlegen ihn kurz nach dem Programmstart beispielsweise in den externen RAM, da die HRAM Größe für ihre Zwecke nicht ausreichend ist, oder dort andere Daten gespeichert werden sollen. Der Stack beginnt grundsätzlich bei Adresse 0xFFFE und arbeitet in absteigender Reihenfolge, sodass die nächste im Stack Pointer Register gespeicherte Adresse, nachdem 16 Bit in den Stack kopiert wurden, 0xFFFC wäre.

Der Program Counter verweist stets auf das nächste zu fetchende Byte im binären Machinencode. Sobald es gelesen wurde, wird er automatisch inkrementiert. Es kann zu jeder Zeit auch eine andere beliebige Adresse in den Program Counter geschrieben werden, beispielsweise wenn Daten in den RAM kopiert wurden, die als Programmcode verstanden und ausgeführt werden sollen.

Darüber hinaus verfügt die CPU über zwei grundsätzliche Speedmodi. Dies betrifft jedoch nur die weiterentwickelte Variante im Gameboy Color. Technisch sind beide Derivate nahezu identisch, das Gameboy Color Pendant verfügt lediglich über ein paar Zusätze. Bei dieser Variante existiert, wie auch bei der regulären Gameboy CPU (DMG), der Singlespeedmode mit einer Takt Frequenz von 4.194304 MHz. Bei der weiterentwickelten Variante existiert wiederum der sogenannte Doublespeedmode, in dem die Takt Frequenz auf das doppelte angehoben wird, also 8.388608 MHz. Diese Verdoppelung hat Auswirkungen auf einige Hardware Komponenten jedoch nicht auf alle, daher wird an den entsprechenden Stellen wiederum eine Anmerkung dazu aufgeführt.

Instruction Set

	x0	x1	x2	x3	x4
0x	NOP 1 4 - - - -	LD BC,d16 3 12 - - - -	LD (BC),A 1 8 - - - -	INC BC 1 8 - - - -	INC B 1 4 Z 0 H -
1x	STOP 0 2 4 - - - -	LD DE,d16 3 12 - - - -	LD (DE),A 1 8 - - - -	INC DE 1 8 - - - -	INC D 1 4 Z 0 H -
2x	JR NZ,r8 2 12/8 - - - -	LD HL,d16 3 12 - - - -	LD (HL+),A 1 8 - - - -	INC HL 1 8 - - - -	INC H 1 4 Z 0 H -
3x	JR NC,r8 2 12/8 - - - -	LD SP,d16 3 12 - - - -	LD (HL-),A 1 8 - - - -	INC SP 1 8 - - - -	INC (HL) 1 12 Z 0 H -

Abbildung 3: Instruction Set Ausschnitt [Instructionset II]

Wie in Abbildung 3 zu sehen ist der Aufbau der Instruktionen relativ einfach. Grundsätzlich stellt das 1. Byte das an der Adresse im Program Counter gelesen wird, den sogenannten Opcode dar, in der Abbildung repräsentiert durch die Zeilen- und Spaltenbeschriftung. Jedes weitere Byte stellt wiederum Daten oder Adressen, abhängig vom jeweiligen Opcode, dar. Das Fetchen jedes einzelnen Bytes des Maschinencodes benötigt jeweils einen Machine Cycle, ebenso wie das Ausführen den Instruktion. [Cycle Accuracy II]

Als Beispiel die Binärsequenz (in Hexadezimal):

01 3A F8

Das erste Byte repräsentiert den Opcode 01 (in Abbildung 3 in Zeile 1, Spalte 2), anhand dessen müssten noch gemäß der Instruktion die 2 darauffolgenden Bytes für die Daten (d16) gelesen werden, hier die Sequenz 3A F8. Da der Gameboy in Little Endian Byteorder arbeitet, stellt 3A das entsprechende Lowbyte dar, während F8 das Highbyte repräsentiert. Dementsprechend würde sich die Instruktion in Assembler wie folgt lesen:

LD BC, F83A

Dies bedeutet „Lade die Daten 0xF83A in das 16 Bit Register BC“.

Insgesamt verfügt der Gameboy über eine Instruktionstabelle mit 256 Einträgen. Somit also 256 Instruktionen, da ein Opcode 8 Bit lang ist und somit 2^8 = 256 mögliche Kombinationen existieren. Nicht alle der 256 möglichen Opcodes wurden jedoch verwendet.

Darüber hinaus gibt es einen zusätzlichen Opcode (0xCB), der es der CPU ermöglicht von einer weiteren Instruktionstabelle Gebrauch zu machen. Diese bestehen hauptsächlich aus Bitshifts und Bittests, sowie Bitflip Operationen.

Timer

Die Gameboy CPU verfügt über ein internes Timersystem. Viele der Abläufe sind unmittelbar hieran gekoppelt. Die Kernkomponente davon stellt das sogenannte DIV Register dar (Adresse 0xFF04). Intern handelt es sich dabei eigentlich um ein 16 Bit Register, jedoch sind lediglich die oberen 8 Bit auf die entsprechende Adresse im Speicherbereich von 0x0000 bis 0xFFFF gemappt, weshalb auch lediglich diese 8 Bit beschrieben werden können. Jedoch führt das Beschreiben automatisch zum Reset der unteren 8 Bit. In Summe ist die Funktionsweise dieses Registers und Verhaltensweisen unter bestimmten Umständen etwas komplizierter. Aber da in dieser Dokumentation nur ein grundsätzlicher Überblick zum Verständnis der Hardware verschafft werden soll werden diese Vorgänge an dieser Stelle außen vor gelassen und sind in den Quellen einsehbar.

Das DIV Register (die vollen 16 Bit) wird mit der Takt Frequenz der CPU inkrementiert, folglich bei einer Frequenz von 4.194304 MHz. Da die CPU Zugriff auf die oberen 8 Bit dieses Registers hat, inkrementiert dieses von außen betrachtet bei einer Frequenz von:

4.194304 MHz / (2^8) = 16384 Hz

Darüber hinaus existiert noch das Timer Register TIMA (Adresse 0xFF05), das unmittelbar an DIV gekoppelt ist. Grundsätzlich inkrementiert dieser Timer bei einer High to Low Transition eines Bits von DIV, welches durch das Steuerungsregister TMA (Adresse 0xFF06) gewählt werden kann. Demnach inkrementiert dieses Register bei einem Bruchteil der Frequenz von DIV.

Beide Timer werden direkt vom Doublespeedmode beeinflusst und inkrementieren dementsprechend mit der doppelten Frequenz. [Sharp SM83 Timer]

Interrupts

VBLANK	Bit 0	Vertical Blanking
LCD (STAT)	Bit 1	Verschiedene Ereignisse in der PPU
Timer	Bit 2	Überlauf des Timers TIMA
Serial	Bit 3	Byte über serielle Verbindung empfangen
Joypad	Bit 4	Button gedrückt

Tabelle 4: Interrupt Flags [SM83 Interrupts]

In Summe existieren fünf Interrupt Quellen beziehungsweise Flags, wie in Tabelle 4 zu sehen.

Der VBLANK Interrupt wird getriggert, wenn die PPU beziehungsweise der LCD bei Scanline 144 ist. 0 bis 143 stellen die einzelnen horizontalen Pixelreihen des Displays dar in die die PPU das auszugebende Bild zeichnet. Ab 144 geht die PPU Vertical Blank Modus über und signalisiert der CPU über den Interrupt, dass sämtliche Pixelreihen abgearbeitet wurden.

Beim LCD (STAT) Interrupt handelt es sich um ein Flag, welches über verschiedene Interrupt Quellen getriggert werden kann. Dies hängt von der Konfiguration der jeweiligen Steuerungsregister der PPU ab. Prinzipiell kann gesteuert werden in welchen Modi (siehe Kapitel PPU) dieser Interrupt ausgelöst wird. Er wird vorzugsweise verwendet, um Daten in VRAM und OAM während einem Frame zu verändern und spezielle midframe Grafikeffekte zu erzeugen.

Der Timer Interrupt wird jedes mal getriggert, wenn der Timer TIMA überläuft. Dieser findet vorzugsweise Verwendung, um bestimmte zeitlich gesteuerte Events auszulösen oder die CPU aus dem Halt State zu holen. Auf diesen State soll in der Dokumentation nicht näher eingegangen werden. Kurz gesagt wird die CPU in einen Zustand versetzt, in dem sie keine Instruktionen mehr ausführt und die einzige Möglichkeit diesen zu verlassen, ist das triggern eines Interrupts.

Der Serial Interrupt wurde in dem Projekt nicht weiter berücksichtigt. Er hat die Funktion das Programm auf empfangene Datenbytes per serieller Verbindung reagieren zu lassen. Die serielle Verbindung bezieht sich auf ein sogenanntes Link-Kabel, durch welches zwei Systeme miteinander verbunden werden können, um beispielsweise miteinander spielen oder tauschen zu können.

Für das Verarbeiten von Button Inputs besteht die Möglichkeit vom Joypad Interrupt Gebrauch zu machen. Er ermöglicht es, das Programm auf Eingaben reagieren zu lassen und diese zu verarbeiten. Nicht jedes Spiel macht hiervon jedoch Gebrauch. Manche Programme setzen stattdessen darauf, die entsprechenden Register für Button Inputs selbständig in regelmäßigen Zeitabständen auszulesen.

Implementierung

Die Register wurden durch ein Struct umgesetzt in folgender Form:

```
struct registers {

	u8 F = 0;
	u8 A = 0;

	union {
		u16 BC = 0;
		struct {
			u8 C;
			u8 B;
		}BC_;
	};

	union {
		u16 DE = 0;
		struct {
			u8 E;
			u8 D;
		}DE_;
	};

	union {
		u16 HL = 0;
		struct {
			u8 L;
			u8 H;
		}HL_;
	};

	u16 SP = 0;
	u16 PC = 0;
};
```

union wird seitens des C++ Standards vom Compiler nicht garantiert – je nach Zugriff - als ein Speicherbereich angesteuert und als Undefined Behaviour definiert. Jedoch verfahren die meisten Compiler dennoch damit nach dem C Standard. Auf diese Weise müssen nicht einzelne Bytes maskiert und extrahiert werden.

Der prinzipielle Ablauf der CPU einzelne Bytes zu fetchen und auszuführen wurde mit einer einfachen while-Schleife umgesetzt:

```
void GameboyCPU::RunCycles() {
    currentTicks = 0;

    while ((currentTicks < (ticksPerFrame * machine_ctx->currentSpeed))) {
        if (stopped) {
            // check button press
            if (mem_instance->GetIO(IF_ADDR) & IRQ_JOYPAD) {
                stopped = false;
            } else {
                return;
            }

        } else if (halted) {
            TickTimers();

            // check pending and enabled interrupts
            if (machine_ctx->IE & mem_instance->GetIO(IF_ADDR)) {
                halted = false;
            }
        } else {
            CheckInterrupts();
            ExecuteInstruction();
        }
    }

    tickCounter += currentTicks;
}
```

Auf die CheckInterrupts() Methode soll an dieser Stelle nicht genauer eingegangen werden. Weitere Details sind den Quellen und dem Quellcode zu entnehmen.
Grundsätzlich führt das Programm solange Instruktionen aus bis die Anzahl an Clock Cycles erreicht wurde, welche in der Zeitspanne pro Frame (Bild) - bei 60 Bildern die Sekunde - in der echten Hardware durchlaufen würden.

Die ExecuteInstruction() Methode ist wie folgt aufgebaut:

```
void GameboyCPU::ExecuteInstruction() {
    curPC = Regs.PC;
    FetchOpCode();

    if (opcode == 0xCB) {
        FetchOpCode();
        instrPtr = &instrMapCB[opcode];
    }
    else {
        instrPtr = &instrMap[opcode];
    }

    functionPtr = get<INSTR_FUNC>(*instrPtr);
    (this->*functionPtr)();
}
```

Memory

Technische Details

Der Speicher im Gameboy ist grundsätzlich, wie zuvor bereits erwähnt, auf einen Speicherbereich von 0x0000 bis 0xFFFF begrenzt. Über die Adressen in diesem Bereich können sämtliche Speicher-Bausteine und IO Register angesteuert und ausgelesen werden.

ROM0	0x0000 – 0x3FFF	ROM Bank 0
ROMn	0x4000 – 0x7FFF	ROM Bänke 1 – n
VRAMn	0x8000 – 0x9FFF	VRAM Bänke 0 – 1
RAMn	0xA000 – 0xBFFF	RAM Bänke 0 – n
WRAM0	0xC000 – 0xCFFF	WRAM Bank 0
WRAMn	0xD000 – 0xDFFF	WRAM Bänke 1 – n
-	0xE000 – 0xFDFF	-
OAM	0xFE00 – 0xFE9F	Object Attribute Memory
-	0xFEA0 – 0xFEFF	-
IO Register	0xFF00 – 0xFF7F	IO Register (Peripherie, Timer, etc.)
HRAM	0xFF80 – 0xFFFE	High RAM
IE	0xFFFF	IE Register

Tabelle 5: Memory Layout [Gameboy Memory]

Wie Tabelle 4 zu entnehmen, ist der Adressbereich von 0x0000 bis 0xFFFF in verschiedene Teilbereiche unterteilt. Jeder dient zur Ansteuerung der verschiedenen Speicherbausteine oder für andere Funktionalitäten. Grundsätzlich arbeitet der Gameboy in der Little Endian Byteorder. Das heißt bei einem 16 Bit Datenwort liegt das untere Byte bei Adresse n und das obere Byte bei Adresse n+1 (Low Byte first, High Byte second).

Das Schlüsselwort „extern“ bezieht sich in den folgenden Paragraphen auf den Umstand, dass die entsprechende Hardware nicht im Gameboy selbst, sondern im Spielmodul (Cartridge) verbaut ist und über einen 16 Bit Adressbus angesteuert wird.
ROM0 stellt die erste externe ROM Bank dar, adressiert von Adresse 0x0000 bis 0x3FFF. ROMn stellt die einzelnen zusätzlichen ROM Bänke von 1 bis n dar, adressiert von 0x4000 bis 0x7FFF. Jede ROM Bank hat dementsprechend eine Größe von:

$$2^{14} \text{ Byte} = 16384 \text{ Byte} = 16 \text{ KiB}$$

Der Speicherbereich für die VRAM Bänke 0 bis n, also Adresse 0x8000 bis 0x9FFF, beinhaltet sämtliche Informationen für das zu zeichnende Bild, unter anderem Tilemaps, welche per Indizes angeben welche Tiles an welcher Stelle gezeichnet werden sollen. Somit wird auch erreicht, dass Tiles bei mehrfacher Verwendung lediglich einmal im Speicher liegen müssen.

Bei WRAM0 und WRAMn handelt es sich um den internen Arbeitsspeicher des Gameboys. Der reguläre Gameboy (DMG) verfügt über zwei Bänke, jeweils eine für WRAM0 und WRAMn. Der Gameboy Color (CGB) verfügt über insgesamt 8, eine für WRAM0 und 7 weitere gemappt auf WRAMn, abhängig von der ausgewählten Bank in einem Steuerungsregister.

HRAM stellt wie bereits erwähnt den regulären Speicherbereich für den Stack dar.

OAM beinhaltet die Informationen für die Objekte (Sprites), wie Spielfiguren, Gegenstände, etc., aber darauf soll hier nicht weiter eingegangen werden.

Bei IE handelt es sich um das Interrupt Enable Register. In diesem wird festgelegt welche Interrupts gegenwärtig aktiviert sind, beziehungsweise welche aktuell verarbeitet werden dürfen, falls sie über das entsprechende Flag angefragt wurden.

MMU

Um die Limitierung lediglich 2 16 KiB große ROM Bänke ansteuern zu können zu umgehen, kann zusätzlich ein sogenannter Mapper im Spielmodul verbaut sein, der es ermöglicht unterschiedliche Bänke auf diesen Speicherbereich zu mappen und anzusteuern.
Auf diese soll hier jedoch noch nicht im Detail eingegangen werden.
Bei den ROM Bänken handelt es sich um die Readonly Speicherbausteine, welche den Programmcode der Spiele in Maschinencode enthalten. In diesem Speicherbereich ließt die CPU nach Programmstart zunächst die einzelnen Instruktionen und führt diese aus (genauer gesagt an Adresse 0x0100). Wie zuvor bereits erwähnt können Instruktionen jedoch auch von anderen Speicherbereichen gelesen und ausgeführt werden, vorausgesetzt sie wurden vom Programmcode zuvor in diesen Bereich kopiert.

Die verschiedenen Mapper ermöglichen es den ROM und RAM durch Mapping um ein vielfaches zu vergrößern. In der Regel erfolgt die Steuerung dieser durch Schreibvorgänge in den Speicherbereich für den ROM (Readonly Memory). Um in verschiedene Register der Mapper schreiben zu können, ist dieser Bereich für Schreibzugriffe nochmals in Teilbereiche unterteilt, wobei jeder Teilbereich einem anderen Steuerregister entspricht. Da eine größere Anzahl an Mappern existiert und sie sich technisch unterscheiden, soll an dieser Stelle nicht näher darauf eingegangen werden.

Implementierung

Zur Umsetzung des Speichers selbst wurde eine Reihe von Funktionen implementiert, um mit den verschiedenen Speicherbereichen interagieren zu können. Diese sollen hier nur stichprobenartig zur Veranschaulichung aufgeführt werden:

```
u8 GameboyMEM::ReadROM_0(const u16& _addr) {
    return ROM_0[_addr];
}

u8 GameboyMEM::ReadVRAM_N(const u16& _addr) {
    if (graphics_ctx.mode == PPU_MODE_3) {
        return 0xFF;
    } else {
        return graphics_ctx.VRAM_N[IO[CGB_VRAM_SELECT_ADDR - IO_OFFSET]][_addr -
VRAM_N_OFFSET];
    }
}
```

Übergeben wird die entsprechende Adresse, auf welche zugegriffen werden soll. Je nach spezifischeren technischen Zusammenhängen und Gegebenheiten im Gameboy wird so der Zugriff auf die Adresse gewährleistet, oder anderweitig damit verfahren. Bei ungültigen oder anderweitigen Zugriffen wird unter bestimmten Umständen dezimal 256 beziehungsweise in hexadezimal 0xFF zurückgegeben.

Als Beispiel für einen Mapper soll hier lediglich für den MBC1 Mapper der Schreibzugriff auf den Adressbereich von ROM0 und ROMn, also Adresse 0x0000 bis 0x7FFF, aufgezeigt werden. Über diese Zugriffe kann gesteuert werden, welche ROM und RAM Bänke auf den jeweiligen Adressbereich gemappt sind.

```
if (_addr < ROM_N_OFFSET) {
      // RAM/TIMER enable
      if (_addr < MBC1_ROM_BANK_NUMBER_SEL_0_4) {
            ramEnable = (_data & MBC1_RAM_ENABLE_MASK) == MBC1_RAM_ENABLE;

            if (!ramEnable && machine_ctx->ram_present && machine_ctx-
>battery_buffered) {
                  WriteSave();
            }

      }
      // ROM Bank number
      else {
            int romBankNumber = _data & MBC1_ROM_BANK_MASK_0_4;
            if (romBankNumber == 0) romBankNumber = 1;
            if (advancedBankingMode) {
                  romBankNumber |= (advancedBankingValue << 5) &
MBC1_ROM_BANK_MASK_5_6;
            }
            machine_ctx->rom_bank_selected = romBankNumber - 1;
      }
}
```

PPU

Technische Details

Die PPU (Pixel processing unit) kann mit der GPU (Graphics Processing Unit) in anderen (modernen) Systemen verglichen werden, wenn auch nur sehr entfernt. Sie arbeitet grundsätzlich mit 8x8 oder 8x16 Pixel Sprites, aus denen sich das auszugebende Bild zusammensetzt. [Graphics Format]

Grundsätzlich lässt sich ein Bild in drei verschiedene Layer zerlegen, der Background, das Window und die Objekte.
Das Background Layer wird genutzt um die Spielwelt an sich darzustellen, dafür scrollt der Gameboy mit einer Viewportgröße, die der Pixelzahl des LCDs in Länge und Breite entspricht, durch die entsprechende Tilemap im VRAM. Jedoch handelt es sich bei diesen Längenangaben um ein achtel der Pixelzahl, da die Tilemap wie zuvor erwähnt lediglich die Indizes der Tiles enthält, wobei jedes Tile eine Größe von 8x8 Pixeln besitzt.
Das Window Layer funktioniert im Prinzip ähnlich wie das Background Layer, jedoch wird hier nicht gescrollt. Es beginnt stets im Ursprung der entsprechenden Tilemap im VRAM. Es wird direkt über das Background Layer gezeichnet und findet hauptsächlich Verwendung bei der Darstellung von Statusleisten, aber auch bei der Erzeugung spezieller Grafikeffekte. Darauf soll hier jedoch nicht näher eingegangen werden.
Die Objekte, welche ihre Informationen nicht in einer Tilemap, sondern im sogenannten OAM (Object Attribute Memory) liegen haben, finden Verwendung bei der Darstellung von interagierbaren Objekten und Spielfiguren (NPCs). Sie können, abhängig von den gesetzten Bits in den OAM-Einträgen, über oder hinter die beiden anderen Layer gezeichnet werden. Dabei wird jedoch bei den Farbpaletten für das Background und Window Layer eine Farbe als Transparenz interpretiert, wodurch sie das Objekt nicht vollständig verdecken.

Der reguläre Gameboy (DMG) besitzt grundsätzlich eine Farbpalette (Graustufen), wobei die einzelnen Farben je nach Verwendung unterschiedlich gemappt werden können, was direkten Einfluss auf die Reihenfolge bei der Verwendung hat. So lassen sich bei schnellem Wechsel ebenfalls verschiedene Grafikeffekte erzeugen.

Wie in Abbildung 4 zu erkennen produziert die PPU Bilder bei einer Frequenz von näherungsweise 60 Bildern die Sekunde. Die Taktung ist direkt an die Clock Cycle beziehungsweise die Basis Takt Frequenz gekoppelt. Pro Bild durchläuft sie 70224 Clock Cycle:

$$4.194304\ \text{MHz} / 70224 = 4194304\ \text{Hz} / 70224 = 59.7275\ldots\ \text{Bilder/Sekunde}$$

Strenggenommen werden während der Scanline 153 nicht ganz 456 Clock Cycle durchlaufen, wodurch es zu einer Abweichung kommt. Für diese Anwendung reicht die Genauigkeit jedoch aus. Genauere Investigation wäre erforderlich, um das emulierte System näherungsweise 100% akkurat laufen zu lassen. Einige Test ROMs, wie Blargg’s Test ROMs, prüfen auf diese Ungenauigkeiten und führen zu einem Fehler, obwohl der Emulator tendenziell sämtliche Spiele laufen lassen kann.

Der Gameboy Color verfügt über einige Zusatzfunktionen und Abweichungen im Bezug auf Farbdarstellung und Prioritäten der Layer. Hierauf soll an dieser Stelle jedoch nicht näher eingegangen werden.

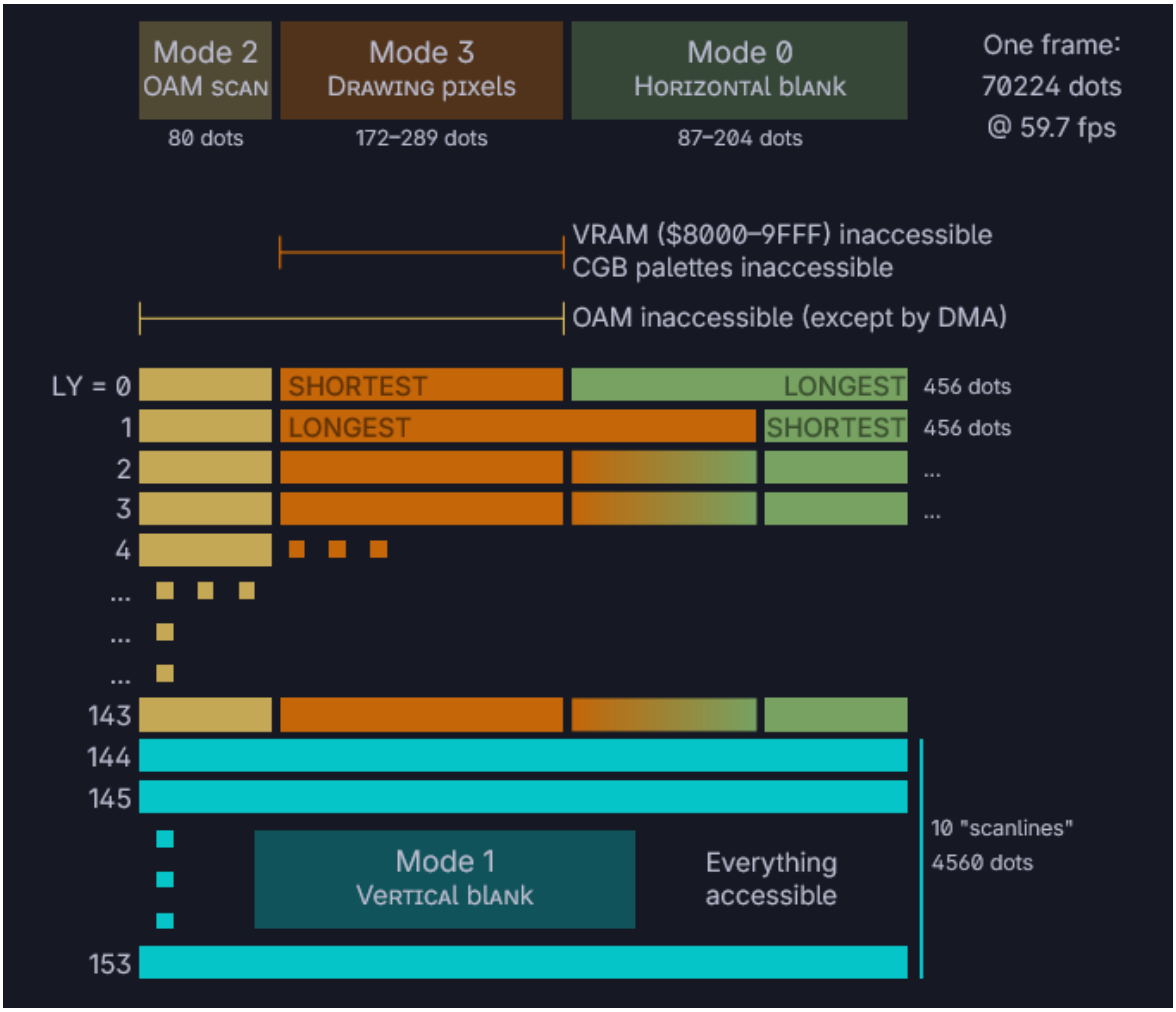

Abbildung 4: PPU Modi [LCD Scanlines]

Ebenfalls in Abbildung 4 ersichtlich, durchläuft die PPU pro Bild in Summe vier Modi, drei davon mehrmals. Die Scanlines (LY= 0 bis 143) entsprechen den Zeitintervallen der zu zeichnenden Pixelreihen. Hierbei wird zunächst ermittelt, welche Objekte in der gegenwärtigen Reihe darzustellen sind. Im Anschluss werden entsprechend der Attribute der Objekte und der darzustellenden Tiles des Background und Window Layers die Pixel der horizontalen Reihe gezeichnet.

Bei Beginn der Scanline 144 wird der zuvor erwähnte VBLANK Interrupt getriggert. In dieser Zeit kann die CPU Daten in VRAM, OAM und den Farbpaletten ändern. Prinzipiell ist dies in bestimmten Zeitintervallen (Modi) ebenfalls beschränkt möglich, wobei der sogenannte STAT Interrupt eine Rolle spielt. An dieser Stelle soll hierauf aber nicht näher eingegangen werden, genauere Informationen dazu sind in den Quellen und dem Quellcode einsehbar.

Grafik-Backend

Für das Grafik-Backend wurde auf die Vulkan Grafik API zurückgegriffen, eine low level, high performance API, welche eine präzise und umfangreiche Ansteuerung der physischen Grafikhardware ermöglicht. [Vulkan Spec]

Da dieses Thema den Rahmen der Projektdokumentation sprengen würde, wird hier nur auf die für die Emulation relevanten Aspekte eingegangen.

Im Prinzip wird für die Daten des Bildes, in das die PPU die gegenwärtig auszugebenden Pixelinformationen schreibt, ein eindimensionales Array verwendet. Das Format für Farbinformationen entspricht RGBA mit je 8 Bit pro Kanal. Der Alpha Kanal (Transparenz) bleibt stets bei einem Wert von 0xFF. Folglich hat das Array eine Größe von:

(4 * 8 Bit) * Höhe * Breite = 4 Byte * 144 Pixel * 160 Pixel = 92160 Byte

Dieses Array wird verwendet, um die Informationen der Textur, welche im VRAM der Grafikkarte liegen, für jeden Frame zu aktualisieren. In Vulkan geschieht dies über die Verwendung von sogenannten CommandBuffern. Diese Buffer werden von der CPU mit Kommandos für die Grafikkarte befüllt und an sie geschickt. Entsprechend der zugehörigen Synchronisierung führt die Grafikkarte diese aus und kopiert die Daten aus dem StagingBuffer in den VRAM. Der StagingBuffer ist erforderlich, da die Grafikkarte keinen direkten Zugriff auf den regulären RAM der CPU hat, ebenso wie die CPU keinen direkten Zugriff auf den VRAM der Grafikkarte hat, Resizable Bar außen vorgenommen. Resizable Bar ermöglicht den direkten Zugriff der CPU auf den VRAM der Grafikkarte, jedoch muss dies sowohl seitens der Hardware unterstützt sein, als auch vom Programmierer dementsprechend implementiert werden. Die CPU kopiert folglich die Textur-Daten in den StagingBuffer und startet auf der Grafikkarte den Transfer dieser Daten in den VRAM. Der StagingBuffer liegt in einem vorher speziell dafür reservierten Teil des RAMs, auf den sowohl die Grafikkarte als auch die CPU Zugriff haben.

Diese Textur im VRAM wird im Anschluss von der entsprechenden Pipeline, die aus einem Vertex- und Fragmentshader besteht, verwendet, um das Bild in den auszugebenden Frame zu zeichnen. Der Vertexshader verarbeitet kurz gesagt die Vertexinformationen, aus welchen sich Polygone zusammensetzen. In diesem Fall zunächst ein Quad, bestehend aus zwei Dreiecken, welche ein Rechteck bilden. Zusätzlich wird eine Skalierungsmatrix übergeben, welche es ermöglicht das Quad an die Skalierung des Fensters anzupassen, um die Seitenverhältnisse unabhängig von den Seitenverhältnissen des Fensters beizubehalten. Der Fragmentshader verwendet dieses Quad, um die kopierte Textur entsprechend der UV Koordinaten darauf zu mappen und zu zeichnen. Als Modus für das Samplen der Textur wird schlicht nach dem nächstgelegenen Pixel in der Textur entsprechend der Position auf dem Quad gesucht, um die Pixel-Grafik des Gameboys zu erhalten und einen Effekt entsprechend Interpolation zu vermeiden. [Vulkan Tutorial]

Implementierung

Auf die Implementierung der PPU soll an dieser Stelle nur kurz eingegangen werden.

```
void GameboyGPU::ProcessGPU(const int& _ticks) {
	if (graphicsCtx->ppu_enable) {
		tickCounter += _ticks;

		u8& ly = memInstance->GetIO(LY_ADDR);
		const u8& lyc = memInstance->GetIO(LYC_ADDR);
		u8& stat = memInstance->GetIO(STAT_ADDR);
		bool ly_lyc = false;

		if (ly == lyc) {
			stat |= PPU_STAT_LYC_FLAG;
			ly_lyc = graphicsCtx->lyc_ly_int_sel;
		} else {
			stat &= ~PPU_STAT_LYC_FLAG;
		}

		switch (graphicsCtx->mode) {
		case PPU_MODE_2:
			//...
			break;
		case PPU_MODE_3:
			//...
			break;
		case PPU_MODE_0:
			//...
			break;
		case PPU_MODE_1:
			//...
			break;
		}

		if (statSignal && !statSignalPrev) {
			memInstance->RequestInterrupts(IRQ_LCD_STAT);
		}
		statSignalPrev = statSignal;

		graphicsCtx->vblank_if_write = false;
	} else {
		statSignal = false;
		statSignalPrev = false;
	}
}
```

Prinzipiell wird die PPU für jeden Clock Cycle der CPU (DIV Timer) laufen gelassen. Entsprechend des gegenwärtigen Modus werden weitere Verarbeitungsschritte eingeleitet. Zudem werden die zugehörigen Register wie LY und LYC berücksichtigt und gegebenenfalls ein STAT Interrupt angefragt. Wenn die PPU die Scanline 144 (LY = 144) erreicht wird die Textur im VRAM der Grafikkarte aktualisiert.

```
void GraphicsVulkan::UpdateTex2d() {

	int& update_index = tex2dData.update_index;

	switch (update_index) {
	case 0:
		if (tex2dData.submit_cmdbuffer_0.load()) { return; }
		break;
	case 1:
		if (tex2dData.submit_cmdbuffer_1.load()) { return; }
		break;
	case 2:
		if (tex2dData.submit_cmdbuffer_2.load()) { return; }
		break;
	}

	memcpy(tex2dData.mapped_image_data[update_index], virtGraphicsInfo.image_data-
>data(), virtGraphicsInfo.image_data->size());

	vkBeginCommandBuffer(tex2dData.command_buffer[update_index], &beginInfo);

	VkBufferImageCopy region = {};
	region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
	region.imageSubresource.layerCount = 1;
	region.imageExtent = { virtGraphicsInfo.lcd_width, virtGraphicsInfo.lcd_height,
1 };
	vkCmdCopyBufferToImage(tex2dData.command_buffer[update_index],
tex2dData.staging_buffer[update_index].buffer, tex2dData.image.image,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &region);

	vkEndCommandBuffer(tex2dData.command_buffer[update_index]);

	switch (update_index) {
	case 0:
		tex2dData.submit_cmdbuffer_0.store(true);
		break;
	case 1:
		tex2dData.submit_cmdbuffer_1.store(true);
		break;
	case 2:
		tex2dData.submit_cmdbuffer_2.store(true);
		break;
	}
	++update_index %= virtGraphicsInfo.buffering;
}
```

Dieser Ausschnitt des Codes stellt vereinfacht die Aktualisierung der Textur im VRAM der Grafikkarte dar. Jegliche Synchronisierung wurde zur Vereinfachung entfernt. Ausgenommen davon sind die Booleans zur Signalisierung, dass der CommandBuffer an die Grafikkarte geschickt wurde. Sie kann aber im Quellcode ohne weiteres eingesehen werden. In diesem Beispiel werden bis zu drei CommandBuffer verwendet um unnötige Wartezeiten zu vermeiden, da die CPU nicht vorhersagen kann, wann die Grafikkarte die Befehle des CommandBuffers ausführt. Zunächst prüft die CPU, ob die Grafikkarte den entsprechenden CommandBuffer bereits an die Grafikkarte geschickt hat. Dann wartet sie per Synchronisation auf den Abschluss der Ausführung dieses CommandBuffers und kopiert die Bildinformationen in den zugehörigen StagingBuffer.

Im Anschluss setzt sie die Kommandos an die Grafikkarte ab um den neuen Inhalt transferieren zu lassen.

```
void GraphicsVulkan::UpdateTex2dSubmit() {
	if (tex2dData.submit_cmdbuffer_0.load()) {
		VkSubmitInfo submitInfo = { VK_STRUCTURE_TYPE_SUBMIT_INFO };
		submitInfo.commandBufferCount = 1;
		submitInfo.pCommandBuffers = &tex2dData.command_buffer[0];

		std::unique_lock<mutex> lock_queue(mutQueue);
		if (vkQueueSubmit(queue, 1, &submitInfo, tex2dData.update_fence[0]) !=
VK_SUCCESS) {
			LOG_ERROR("[vulkan] queue submit texture2d update");
		}
		lock_queue.unlock();
		tex2dData.submit_cmdbuffer_0.store(false);
	}
	if (tex2dData.submit_cmdbuffer_1.load()) {
		VkSubmitInfo submitInfo = { VK_STRUCTURE_TYPE_SUBMIT_INFO };
		submitInfo.commandBufferCount = 1;
		submitInfo.pCommandBuffers = &tex2dData.command_buffer[1];

		std::unique_lock<mutex> lock_queue(mutQueue);
		if (vkQueueSubmit(queue, 1, &submitInfo, tex2dData.update_fence[1]) !=
VK_SUCCESS) {
			LOG_ERROR("[vulkan] queue submit texture2d update");
		}
		lock_queue.unlock();
		tex2dData.submit_cmdbuffer_1.store(false);
	}
	if (tex2dData.submit_cmdbuffer_2.load()) {
		VkSubmitInfo submitInfo = { VK_STRUCTURE_TYPE_SUBMIT_INFO };
		submitInfo.commandBufferCount = 1;
		submitInfo.pCommandBuffers = &tex2dData.command_buffer[2];

		std::unique_lock<mutex> lock_queue(mutQueue);
		if (vkQueueSubmit(queue, 1, &submitInfo, tex2dData.update_fence[2]) !=
VK_SUCCESS) {
			LOG_ERROR("[vulkan] queue submit texture2d update");
		}
		lock_queue.unlock();
		tex2dData.submit_cmdbuffer_2.store(false);
	}
}
```

In diesem Ausschnitt ist der Code dargestellt, der auf einem separaten Thread läuft. Seine einzige Aufgabe ist es auf das Signal des vorherigen Codeausschnitts zu warten, um den entsprechenden CommandBuffer zu transferieren. Der übergebene Fence ist eine Möglichkeit den Abschluss der Ausführung der Kommandos seitens der Grafikkarte zu signalisieren. Er ermöglicht dem vorherigen Ausschnitt festzustellen, ob der entsprechende StagingBuffer wieder zur Verfügung steht. Generell gilt es als gute Vorgehensweise sämtliche CommandBuffer zu befüllen und zu sammeln, beispielsweise in einem std::vector. Abschließend können diese an einem Stück an die Grafikkarte geschickt werden. In diesem Fall wurde jedoch darauf verzichtet, da hier gegenwärtig ein CommandBuffer pro Frame ausreichend ist.

APU

Technische Details

Die APU (Audio Processing Unit) verfügt prinzipiell über vier Audio Kanäle. [Audio]

Die Kanäle 1 und 2 sind grundsätzlich dafür ausgelegt PWM Signale zu erzeugen. Dabei gibt es jedoch nur vier verschiedene Wellenformen, welche anhand der Register Inhalte in Form von verschiedenen Duty Cyclen erzeugt werden können. Die Duty Cycle betragen dabei 12.5%, 25%, 50% oder 75%, wobei 25% und 75% quasi identisch sind, also quasi die jeweils andere Wellenform gespiegelt an der t-Achse ergeben.

Kanal 3 ist der einzige Kanal der etwas komplexere Wellenformen erzeugen kann. Dafür greift er auf einen Speicherbereich zurück, der vordefinierte Samples enthält. Diese wurden von der CPU wiederum zuvor dorthin kopiert. Dieser Speicherbereich beschränkt sich auf insgesamt 16 Byte, wobei jedes Byte zwei Samples zu je 4 Bit enthält. Das untere Nibble wird hierbei zuerst ausgelesen, im Anschluss das Obere. Daraufhin wieder das untere Nibble des nächsten Bytes und so weiter.

Kanal 4 setzt wiederum auf den Einsatz eines LFSR (Linear-Feedback Shift Register) mit 16 Bit. Hierbei wird initial das gesamte Register auf 0 gesetzt. Bei einer vorher festgelegten Taktrate wird aus den beiden unteren Bits mittels invertierter XOR Verknüpfung das nächste Bit ermittelt und je nach der Einstellung der Konfigurationsregister an Bit 15 oder Bit 15 und 7 kopiert. Im Anschluss wird das gesamte Register logisch nach rechts verschoben, also eine binäre 0 an Stelle von Bit 15 nachgeschoben. Der resultierende Ton, beziehungsweise das Rauschen, wird als Noise bezeichnet. Es handelt sich um einen Pseudo-Zufallszahlen Generator, das Resultat ist also entsprechend der zuvor erklärten Funktionsweise stets deterministisch. Das Generator-Polynom entspricht der Form:

$$x^{15} + x^{7} + x + 1$$

Oder in der anderen genannten Konfiguration:

$$x^{15} + x + 1$$

[GBA LFSR]

In der Implementierung bei diesem Emulator wird das gesamte Register invertiert, also bei der Initialisierung auf 0xFFFF gesetzt. Die Invertierung des Ergebnisses der XOR Verknüpfung wurde weggelassen. Durch dieses Vorgehen wird logischerweise ein invertiertes Resultat erzeugt. Das resultierende Audiosignal ist somit lediglich an der t-Achse gespiegelt, der Klang wird hierdurch aber nicht verändert. Laut Dokumentationen wird das Signal ohnehin bei der Analog-Umsetzung invertiert. Somit wird dieser Effekt sowieso ohne weitere Maßnahmen erreicht und entspricht dem erwarteten Signal in der echten Hardware.

Damit lassen sich in Summe mit den Kanälen 1 und 2 grundsätzlich die vom Gameboy gewohnten Töne erzeugen. Kanal 3 ermöglicht es auch komplexere Audiosignale zu erzeugen und Kanal 4 bringt wiederum die Funktionalität mit die Snare oder Hi-Hat eines Schlagzeugs nachzuahmen, oder das Rauschen des Meeres zu simulieren.

NRx0	Period Sweep
NRx1	Length Timer & Duty Cycle
NRx2	Volume & Envelope
NRx3	Period Low
NRx4	Period High & Control

Tabelle 6: Audio Register [Gameboy Audio]

Wie in Tabelle 6 zu sehen verfügt der Gameboy pro Kanal an sich über fünf Register um die verschiedenen Aspekte steuern zu können.

Die Register NRx0 sind grundsätzlich für den Period Sweep zuständig, was prinzipiell die Veränderung der Frequenz des Signals und damit der Tonhöhe über die Zeit bedeutet. Bei Kanal 3 ist es lediglich für die Steuerung des zugehörigen DAC (Digital Analog Converter) zuständig. Bei Kanal 4 entfällt es komplett.

NRx1 Register steuern zum einen den Length Timer, also den Timer pro Kanal der bei Erreichung eines Grenzwertes diesen deaktiviert, zum anderen wird hier im Fall von Kanal 1 und 2 auch der zuvor erwähnte Duty Cycle des Signals gesteuert. Für Kanal 3 und 4 sind hier lediglich die Informationen über den Length Timer gespeichert.

Die NRx2 Register sind grundsätzlich für die Lautstärke (Volume) und Envelope zuständig. Bei Envelope handelt es sich um die Veränderung der Lautstärke über die Zeit. Bei Kanal 3 wird hier lediglich die Lautstärke an sich gesteuert, die Envelope Funktion entfällt hier komplett.

Die NRx3 Register enthalten die unteren 8 Bit der verwendeten Periode des Signals (Frequenz = 1 / Periode). Für Kanal 4 hat dieses Register eine komplett andere Funktion, es steuert die Abläufe des LFSR, also die Frequenz bei der es taktet und auch welches der beiden zuvor beschriebenen Generator-Polynome verwendet wird.

NRx4 steuert die Funktionen des Kanals und (de)aktiviert diese. Im Falls von Kanal 1 und 2 sind hier auch die oberen 3 Bit der Periode gespeichert. Im Falle von Kanal 4 entfallen die Bits der Periode logischerweise. Hier wird ebenfalls gesteuert ob der Length Timer aktiv ist oder nicht.

Audio-Backend

Das Audio-Backend greift ebenfalls auf die SDL zurück [SDL Audio]. Hierbei wird prinzipiell ein separater Thread gestartet, der sich ausschließlich um die Generierung der Samples anhand der gegenwärtigen Zustände der APU kümmert. Hierfür wird zunächst eine Callback Funktion für SDL bereitgestellt, welche Samples aus einem Ringbuffer in den Buffer der Audio API kopiert. Der separate Thread reagiert unmittelbar auf das Voranschreiten dieses Callbacks im Ringbuffer und generiert entsprechend der Menge der entnommenen Samples aus diesem neue Samples anhand der APU Informationen. Dafür greift er ebenfalls auf eine Callback Methode zurück. Das Ganze geschieht folglich in 2 Schichten. [Audio Ringbuffer]

Implementierung

```
void GameboyAPU::ProcessAPU(const int& _ticks) {
	if (soundCtx->apuEnable) {
		for (int i = 0; i < _ticks; i++) {
			envelopeSweepCounter++;
			soundLengthCounter++;
			ch1SamplingRateCounter++;

			if (envelopeSweepCounter == ENVELOPE_SWEEP_TICK_RATE) {
				envelopeSweepCounter = 0;

				// ...
				if (soundCtx->ch4Enable.load()) {
					ch4EnvelopeSweep();
				}
			}

			if (soundLengthCounter == SOUND_LENGTH_TICK_RATE) {
				soundLengthCounter = 0;

				// ...
				if (soundCtx->ch4Enable.load()) {
					ch4TickLengthTimer();
				}
			}

			if (ch1SamplingRateCounter == CH1_FREQU_SWEEP_RATE) {
				ch1SamplingRateCounter = 0;

				if (soundCtx->ch1Enable.load()) {
					// frequency sweep
					ch1PeriodSweep();
				}
			}
		}

		TickLFSR(_ticks);
	}
}
```

Dieser Codeabschnitt stellt das Verarbeiten der APU Funktionen über die Zeit dar, also die Ausführung des Period Sweep oder das Voranschreiten der Length Timer. Er ist direkt an das Timersystem der CPU gekoppelt und läuft bei einer festen Taktrate von 512Hz. Hierbei werden auch in der Funktion TickLFSR() die Anzahl der Samples, welche in dieser angefallen sind und aus dem LFSR generiert werden, berücksichtigt.

```
void GameboyAPU::SampleAPU(std::vector<std::vector<complex>>& _data, const int&
_samples, const int& _sampling_rate) {
	bool right = soundCtx->outRightEnabled.load();
	bool left = soundCtx->outLeftEnabled.load();
	bool vol_right = soundCtx->masterVolumeRight.load();
	bool vol_left = soundCtx->masterVolumeLeft.load();

	// ...
	// channel 4
	float ch4_virt_sample_step = soundCtx->ch4SamplingRate.load() / _sampling_rate;

	bool ch4_right = soundCtx->ch4Right.load();
	bool ch4_left = soundCtx->ch4Left.load();

	float ch4_vol = soundCtx->ch4Volume.load();

	unique_lock<mutex> lock_wave_ram(soundCtx->mutWaveRam, std::defer_lock);
	unique_lock<mutex> lock_lfsr_buffer(mutLFSR, std::defer_lock);

	for (int i = 0; i < _samples; i++) {
		for (int n = 0; n < 4; n++) {
			_data[n].emplace_back();
		}

		float sample_0 = .0f;
		// ...
		{
			lock_lfsr_buffer.lock();
			ch4VirtSamples += ch4_virt_sample_step;
			while (ch4VirtSamples > 1.f) {
				ch4VirtSamples -= 1.f;

				int read_cursor = ch4ReadCursor.load();

				// (0 - 1) % 10 = -1 % 10 and is defined as -1 * 10 + 9,
where the added value is the remainder (remainder is by definition always positiv)
				if (read_cursor != (ch4WriteCursor.load() - 1) %
CH_4_LFSR_BUFFER_SIZE) {
					ch4LFSRSamples[read_cursor] = .0f;
					++read_cursor %= CH_4_LFSR_BUFFER_SIZE;
					ch4ReadCursor.store(read_cursor);
				}
			}

			if (ch4_right) {
				sample_1 += ch4LFSRSamples[ch4ReadCursor] * ch4_vol;
			}
			if (ch4_left) {
				sample_2 += ch4LFSRSamples[ch4ReadCursor] * ch4_vol;
			}
			lock_lfsr_buffer.unlock();
		}

		_data[0][i].real = sample_0 * vol_right * .05f;
		_data[1][i].real = sample_1 * vol_right * .05f;
		_data[2][i].real = sample_2 * vol_left * .05f;
		_data[3][i].real = sample_3 * vol_left * .05f;
	}
}
```

Der Codeabschnitt auf der vorherigen Seite stellt die Callback Methode für den Audio Thread dar. Dieser ruft diese auf und lässt anhand der Zustände der APU neue Samples generieren. Hier wurde exemplarisch nur der Abschnitt für Kanal 4 aufgezeigt. Die Schrittweite für einen Sample der APU wird über einen Quotienten ermittelt, bestehend aus der virtuellen Sampling Rate der APU und der Sampling Rate des physischen Audiogerätes. Somit ist sichergestellt, dass die Frequenz des Signals, und damit die Tonhöhe, erhalten bleibt. Bei einer physischen Sampling Rate von 44100Hz und einer virtuellen Sampling Rate von 22050Hz würde sich daraus folgender Quotient ergeben:

$$22050Hz / 44100Hz = 0.5$$

Somit würde die Schrittweite 0.5 betragen, also mit anderen Worten, jedes Sample der APU wird für zwei Samples des Ringbuffers verwendet. Die Frequenz des Signals bleibt unverändert. Der Zähler der Schritte addiert nun diese Schrittweite mit jedem Durchlauf auf und erhöht den Index des Samples um 1 sobald der Zähler größer 1 ist. Vom Zähler wird wiederum im Anschluss 1 abgezogen um Ungenauigkeiten vorzubeugen und die Frequenz des erzeugten Signals im Mittel konstant zu halten.

Für das Zählen der Schritte wurde auf eine while-Schleife zurückgegriffen. Das hat den Hintergrund, dass gerade bei Kanal 4 beispielsweise bei einer physischen Sampling Rate von 44100Hz und einer virtuellen Sampling Rate von 88200Hz der folgende Quotient zustande kommen würde:

$$88200Hz / 44100Hz = 2$$

Somit würde nach vorheriger Logik nur jedes zweite Sample der APU berücksichtigt werden. Folglich muss der Index nach jedem Sample zwei mal inkrementiert werden um die Frequenz des Signals korrekt umzusetzen.

Aufgrund der Funktionsweise von Kanal 4 wurde für die Zwischenspeicherung der Samples aus dem LFSR auf einen weiteren Ringbuffer zurückgegriffen, da die Samples beim Laufen der CPU erzeugt werden.

```
void audio_thread(audio_information* _audio_info, virtual_audio_information*
_virt_audio_info, audio_samples* _samples) {
	// ...

	while (_virt_audio_info->audio_running.load()) {


		SDL_LockAudioDevice(*device);
		int reg_1_size, reg_2_size;
		if (_samples->read_cursor < _samples->write_cursor) {
			reg_1_size = (int)_samples->buffer.size() - _samples->write_cursor;
			reg_2_size = _samples->read_cursor;
		} else if (_samples->read_cursor > _samples->write_cursor) {
			reg_1_size = _samples->read_cursor - _samples->write_cursor;
			reg_2_size = 0;
		} else {
			reg_1_size = 0;
			reg_2_size = 0;
		}

		if (reg_1_size || reg_2_size) {
			// get samples from APU
			int reg_1_samples = reg_1_size / channels;
			int reg_2_samples = reg_2_size / channels;
			for (auto& n : virt_samples) {
				n.clear();
			}

			sound_instance->SampleAPU(virt_samples, reg_1_samples +
reg_2_samples, _audio_info->sampling_rate);

			// TODO: use FFT and other algorithms for different effects

			// transfer samples into ringbuffer // for now just feed the real
part back into the output
			float volume = _audio_info->master_volume.load();

			float* buffer = _samples->buffer.data() + _samples->write_cursor;
			for (int i = 0; i < reg_1_samples; i++) {
				for (int j = 0; j < channels; j++) {
					buffer[j] = .0f;
				}
				for (int j = 0; j < virt_channels; j++) {
					(*speaker_fn)(buffer, virt_samples[j][i].real *
volume, virt_angles[j]);
				}

				buffer += channels;
			}

			// same for region 2 here
		}
		SDL_UnlockAudioDevice(*device);

		_samples->write_cursor = (_samples->write_cursor + reg_1_size +
reg_2_size) % (int)_samples->buffer.size();
	}
}
```

In diesem Codeabschnitt ist die Funktion dargestellt welche im Audio-Thread läuft. Hier wird geprüft ob dem Ringbuffer vom übergebenen Callback der SDL Audio API weitere Samples entnommen wurden und über den Callback aus dem vorherigen Abschnitt neue Samples erzeugt und entsprechend der virtuellen und physischen Sampling Rate der Ringbuffer weiter befüllt. speaker_fn ist dabei ein Funktionspointer für die passende Funktion entsprechend der Anzahl der physisch vorhandenen Audiokanäle. Diese setzt die Samples der virtuellen Kanäle auf die physischen Kanäle um. Dabei berücksichtigt wurden Mono (1 Ausgang), Stereo (2 Ausgänge), 5.1 Surround (6 Ausgänge) und 7.1 Surround (8 Ausgänge). Am Beispiel von 7.1 Surround entspricht dies den Kanälen Front-Left, Front-Right, Centre-Left, Centre-Right, Rear-Left, Rear-Right, Centre und Low Frequency (Bass). Die Verteilung geschieht über die Positionen der virtuellen und physischen Kanäle relativ zur Position des Hörers, also den von ihnen eingeschlossenen Winkel. Auf diese Funktionen soll hier aber nicht näher eingegangen werden. Die erzeugten Samples werden im Realteil von komplexen Zahlen gespeichert um per FFT weitere Audioeffekte erzeugen zu können. Gegenwärtig werden aber lediglich die Realteile unverändert in den Puffer der Ausgabekanäle kopiert.

Quellen

Im Folgenden sind weitere verwendete Quellen aufgelistet, welche aber nicht direkt in der Dokumentation referenziert wurden. Sie stellen entweder zusätzliche Informationsquellen dar oder beziehen sich auf kleinere Details der Funktionsweise der Hardware.

[Pandocs II], [Hacktix], [Instructionset I], [Cycle Accuracy], [Timer Detail], [Half Carry], [Carry Flags], [CP Instruction], [JP HL Instruction], [Scanlines], [HALT State], [CB Instruction], [Vulkan Cheat Sheet], [Vulkan Examples], [Engine Entwicklung], [Matrix Basics], [Coordinate Systems], [Camera], [Texture Mapping], [Vulkan Basics], [Projection]

Quellenverzeichnis

Z80: https://wikiti.brandonw.net/index.php?title=Z80_Instruction_Set
SM83 Specs: https://gbdev.io/pandocs/Specifications.html
SM83 Register: https://gbdev.io/pandocs/CPU_Registers_and_Flags.html
DAA: https://forums.nesdev.org/viewtopic.php?t=15944
Instructionset II: https://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html
Cycle Accuracy II: https://gekkio.fi/files/gb-docs/gbctr.pdf
Sharp SM83 Timer: https://gbdev.io/pandocs/Timer_and_Divider_Registers.html
SM83 Interrupts: https://gbdev.io/pandocs/Interrupt_Sources.html
Gameboy Memory: https://gbdev.io/pandocs/Memory_Map.html
Graphics Format: https://www.huderlem.com/demos/gameboy2bpp.html
LCD Scanlines: https://gbdev.io/pandocs/Rendering.html
Vulkan Spec: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#devsandqueues-physical-device-https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html
Vulkan Tutorial: https://www.youtube.com/playlist?list=PLStQc0GqppuXgs6do23v_HKRrR32gJMm3
Audio: https://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware
GBA LFSR: http://belogic.com/gba/channel4.shtml
Gameboy Audio: https://gbdev.io/pandocs/Audio_Registers.html
SDL Audio: https://wiki.libsdl.org/SDL2/SDL_AudioSpec
Audio Ringbuffer: https://github.com/etscrivner/sdl_audio_circular_buffer/blob/master/src/circular_buffer.cpp
Pandocs II: https://problemkaputt.de/pandocs.htm
Hacktix: https://github.com/Hacktix/GBEDG/tree/master
Instructionset I: http://www.devrs.com/gb/files/GBCPU_Instr.html
Cycle Accuracy: https://raw.githubusercontent.com/geaz/emu-gameboy/master/docs/The%20Cycle-Accurate%20Game%20Boy%20Docs.pdf
Timer Detail: https://www.reddit.com/r/EmuDev/comments/z6trul/question_about_timers_gameboy_emulation/
Half Carry: https://www.reddit.com/r/EmuDev/comments/knm196/gameboy_half_carry_flag_during_subtract_operation/
Carry Flags: https://gist.github.com/meganesu/9e228b6b587decc783aa9be34ae27841

CP Instruction: https://stackoverflow.com/questions/31409444/what-is-the-behavior-of-the-carry-flag-for-cp-on-a-game-boy
JP HL Instruction: https://www.reddit.com/r/EmuDev/comments/ivdf29/question_about_the_gb_jp_hl_instruction/
Scanlines: https://forums.nesdev.org/viewtopic.php?t=16434
HALT State: https://www.reddit.com/r/EmuDev/comments/5bfb2t/a_subtlety_about_the_gameboy_z80_halt_instruction/
CB Instruction: https://www.reddit.com/r/EmuDev/comments/gj69h3/how_many_cycles_are_required_to_execute_a/
Vulkan Cheat Sheet: https://www.khronos.org/files/vulkan11-reference-guide.pdf
Vulkan Examples: https://github.com/jherico/VulkanExamples
Engine Entwicklung: https://www.youtube.com/playlist?list=PL8327DO66nu9qYVKLDmdLW_84-yE4auCR
Matrix Basics: https://learnopengl.com/Getting-started/Transformations
Coordinate Systems: https://learnopengl.com/Getting-started/Coordinate-Systems
Camera: https://learnopengl.com/Getting-started/Camera
Texture Mapping: https://taidaesal.github.io/vulkano_tutorial/section_13.html
Vulkan Basics: https://www.youtube.com/playlist?list=PLmIqTlJ6KsE1Jx5HV4sd2jOe3V1KMHHgn
Projection: http://www.songho.ca/opengl/gl_projectionmatrix.html