universität koblenz
Informatik

Konzeption und Implementierung eines „Game Boy“-Emulators

Bachelorarbeit

Zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von

Jonathan Engel

Erstgutachter:	Prof. Dr. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Alexander Maximilian Nilles, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im März 2024

Abstract

This thesis deals with the conception and implementation of a prototype emulator software that can be used to play a broad range of Game Boy games on a conventional desktop computer. The development of such an application is a technically demanding task defined by various challenges such as the correct interpretation of machine instructions, graphics emulation, as well as playability and correctness. As there is no accessible official documentation of Game Boy hardware, the emulator was developed based on the knowledge amassed by Game Boy reverse engineers. Nevertheless, an emulator was developed that can already run a large selection of games. The correctness of the implemented components was verified using dedicated test programs.

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Konzeption und Implementierung einer prototypischen Emulator-Software, mit der eine umfangreiche Palette klassischer Game Boy-Spiele auf einem herkömmlichen Desktop-Computer gespielt werden kann. Die Entwicklung einer solchen Anwendung ist eine technisch anspruchsvolle Aufgabe, die durch verschiedene Herausforderungen, wie die korrekte Interpretation von Maschinenbefehlen, der Emulation von Grafik sowie der Spielbarkeit und Korrektheit geprägt ist. Da keine zugänglichen offiziellen Dokumente über die Game Boy-Hardware vorliegen, wurde der Emulator auf Basis des Wissens von Game Boy-Enthusiasten konzipiert. Dennoch konnte eine Anwendung entwickelt werden, welche bereits eine große Auswahl an Spielen ausführen kann. Die Korrektheit der implementierten Komponenten wurde mittels Testprogrammen verifiziert.

Inhaltsverzeichnis

1 Einführung

Dieses Kapitel befasst sich mit der Grundidee von Emulatoren sowie dem Zweck, einen prototypischen Emulator für die Game Boy-Spielkonsole zu entwickeln. Der Umfang des entwickelten Emulators wird ebenfalls festgelegt.

1.1 Kontext

Ein Emulator ist ein Computersystem, das die Funktionalität eines anderen Computersystems nachahmt, um so die Ausführung von Anwendungen oder Videospielen zu ermöglichen, die ursprünglich nur für ein bestimmtes System entwickelt wurden und daher nicht mit anderen Systemen kompatibel sind. Emulatoren können beispielsweise eingesetzt werden, um Computerspiele, welche nur für dedizierte Spielkonsolen entwickelt wurden, auf einem modernen Computer spielbar zu machen (1, 2).

Der Game Boy ist eine beliebte Handheld-Spielkonsole des japanischen Herstellers Nintendo, die erstmals im Jahr 1989 verkauft wurde und seither Millionen von Spielern auf der ganzen Welt begeistert hat. Zu Beginn erschienen wegweisende Titel wie etwa „Tetris“ und „Super Mario Land“, aber auch in späteren Jahren kamen weitere erfolgreiche Game Boy-Spiele auf den Markt, darunter die erste Generation an „Pokémon“-Spielen, die auch noch in den späten 1990er Jahren dem Game Boy zu größerer Bekanntheit und Aufmerksamkeit verhalfen (3).

Abbildung 1 Game Boy-Konsole der ersten Generation. Aus (4).

Das erste Game Boy-Modell, welches schlichtweg als Game Boy bezeichnet wird, besteht aus einem grünen LC-Display, auf dem 160×144 Pixel sowie vier unterschiedliche Graustufen dargestellt werden können. Spieler können mittels der dafür vorgesehenen acht Eingabeknöpfe das Spielgeschehen steuern. Zudem können unterschiedliche Spiele ausgeführt werden, da diese auf physischen Speichermedien, sogenannten Game Paks, veröffentlicht wurden, die in das dafür vorgesehene Fach der Konsole eingesteckt werden können. In den Jahren nach der Veröffentlichung des Game Boys erschienen weiterentwickelte Modelle der Konsole, darunter der Game Boy Color aus dem Jahr 1998, welcher bis zu 56 verschiedene Farben gleichzeitig darstellen kann (3, 5).

1.2 Zweck

Auch klassische Spielkonsolen, wie der Game Boy, erfreuen sich nach wie vor großer Beliebtheit. Jedoch ist solche Hardware heutzutage kaum noch zugänglich für die Massen, da die Produktion des Game Boys eingestellt wurde. Emulatoren werden genutzt, um Game Boy-Spiele, wie zum Beispiel „Tetris“ oder „Pokémon Gelbe Edition“, auf einem PC, Laptop oder Smartphone spielbar zu machen.

Zweck dieser Arbeit ist die Konzeption und Implementierung einer prototypischen Emulator-Software, mit der klassische Game Boy-Spiele ausgeführt werden können. Die Entwicklung eines solchen Emulators ist eine technisch anspruchsvolle Aufgabe, die durch verschiedene Herausforderungen, wie die korrekte Interpretation von Maschinenbefehlen, der Emulation von Grafik, Audio und Peripheriegeräten sowie der Spielbarkeit und Genauigkeit geprägt ist.

1.3 Umfang

Im Fokus dieser Arbeit steht die Emulation der essenziellen Komponenten des Game Boys, die für die Spielbarkeit unabdingbar sind. Dies beinhaltet vor allem die visuelle Repräsentation des Spielgeschehens durch die Emulation des LC-Displays sowie die Steuerbarkeit mithilfe eines Eingabegeräts, wie etwa einer Computertastatur, durch den Anwender. Um die Kompatibilität mit möglichst vielen Spielen zu gewährleisten und somit die Spielbarkeit zu steigern, muss auch Game Pak-Hardware emuliert werden. Es existieren diverse Zusammensetzungen dieser Speichermedien, wobei einige Konstellationen häufiger genutzt werden als andere. Somit konzentriert sich die Emulation von Game Pak-Hardware zunächst auf die häufigsten Hardware-Typen, während die Unterstützung für andere Arten optional angedacht ist.

Weitere Elemente des Game Boy-Systems, wie serielle Kommunikation mit Peripheriegeräten sowie die Emulation von Audio, wurden höchstens als optionale Erweiterungen eingestuft, und sind daher nicht Teil des Projektumfangs. Des Weiteren fokussiert sich diese Arbeit im Kern auf die Emulation des originalen Game Boy-Modells, sodass neuere Modelle, wie etwa der Game Boy Color, und deren Funktionen nicht berücksichtigt werden.

2 Theorie

In diesem Kapitel wird die Architektur des Game Boys eingeführt und die darin enthaltenen Komponenten vorgestellt. Weiterhin werden exemplarische Game Boy-Emulatoren präsentiert.

2.1 Architektur

Ein Game Boy-System ist aus unterschiedlichen Teilkomponenten zusammengesetzt. Den zentralen Bestandteil bildet die Sharp SM83-CPU, welches das Spielprogramm ausführt und mit den umliegenden Komponenten interagiert. Die Pixel Processing Unit, kurz PPU, verarbeitet Bilddaten zu Pixeln, welche über das LC-Display dargestellt werden. Der Sound wird wiederum von der Audio Processing Unit, kurz APU, erzeugt und über die Lautsprecher oder über den Kopfhöreranschluss ausgegeben. Nutzereingaben können von Spielern über die acht Eingabeknöpfe getätigt werden. Deren Druckzustände werden wiederum von der CPU ausgelesen. Das eigentliche Spielprogramm, welches von der Konsole ausgeführt werden soll, befindet sich als ROM auf einem physischen Speichermedium, das als Game Pak bezeichnet wird. Es gibt diverse Zusammensetzungen dieser Game Paks, sodass manche auch über speicherbares RAM verfügen, um beispielsweise den Spielfortschritt abzuspeichern. Serielle Kommunikation mit anderen Game Boy-Konsolen oder auch Peripheriegeräten wird über einen seriellen Port per Link Cable realisiert (6).

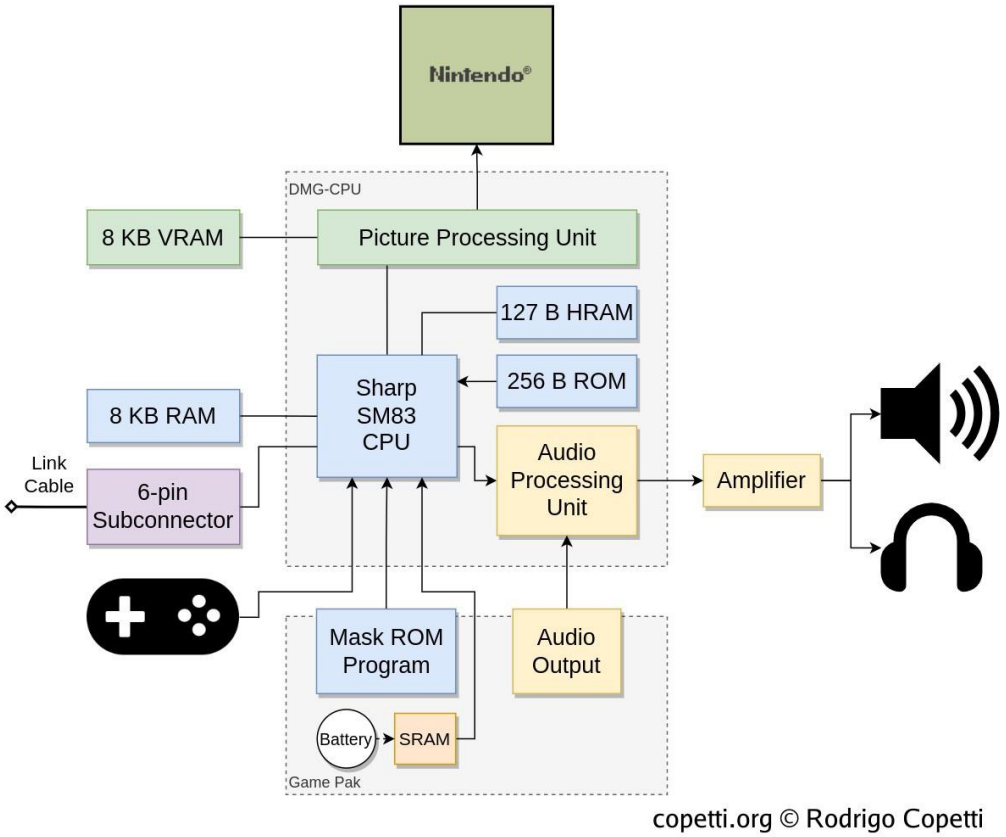

Abbildung 2 Veranschaulichung der grundlegenden Bausteine eines Game Boy-Systems. (6)

2.2 SM83-CPU

Die CPU des Game Boys ist ein Sharp SM83, welcher speziell für den Game Boy entworfen wurde. Die Taktfrequenz des Oszillators beträgt ungefähr 4,194 MHz. Ein einzelner Maschinenzyklus beansprucht vier Takte, sodass die Zyklusfrequenz in etwa 1,048 MHz entspricht (7).

2.2.1 Register

In diesem Abschnitt werden die Register der SM83-CPU vorgestellt, die als Operanden für Instruktionen verwendet werden. Die adressierbaren und internen Register werden in den entsprechenden Unterkapiteln vorgestellt, in welchen sie zum Einsatz kommen.

Der SM83 besitzt die sechs generischen 8-Bit-Register B, C, D, E, H und L. Es besteht die Möglichkeit diese paarweise zu 16-Bit-Registern zu kombinieren, sodass zusätzlich die drei Register BC, DE und HL verwendet werden können. Exemplarisch setzt sich das Register BC aus den Registern B und C zusammen, wobei ersteres die oberen acht Bits und zweiteres die unteren acht Bits zur Verfügung stellt (8).

Register F speichert Zustandsbits, die bei der Ausführung von Instruktionen ausgelesen, gesetzt oder genullt werden. Die unteren vier Bits von F sind stets nicht gesetzt und können auch nicht gesetzt werden (8).

Bit	Kürzel	Bezeichnung	Bedeutung
7	Z	Zero	Gibt an, ob das Ergebnis der letzten Operation gleich null ist.
6	N	Negative	Gibt an, ob die letzte Operation eine Subtraktion war.
5	H	Half-carry	Gibt an, ob es in der letzten Operation einen Übertrag (Carry) von Bit 3 nach Bit 4 gab.
4	C	Carry	Gibt an, ob das Ergebnis der letzten Operation die 8-Bit-Grenze überschritten hat.

Register A, oder auch Akkumulator genannt, ist ein besonderes 8-Bit-Register, da nur dieses als Ergebnisregister für arithmetische und logische Instruktionen agieren kann. A und F können zum 16-Bit-Register AF kombiniert werden (8).

Des Weiteren nutzt die CPU zwei besondere 16-Bit-Register. Das Register PC agiert als Programmzähler, der die Adresse der aktuell ausgeführten Instruktion angibt. SP, abkürzend für Stack Pointer, zeigt auf das oberste Element des Stack-Speichers (7, 8).

2.2.2 Instruktionen

Instruktionen sind atomare Befehle, mit denen grundlegende arithmetische Berechnungen und logische Operationen durchgeführt werden können. Aber auch der Kontrollfluss des Programms sowie Speicherzugriffe werden über solche Befehle gesteuert. Die CPU unterstützt genau 500 verschiedene Instruktionen (9).

Jede Instruktion wird eindeutig durch einen sogenannten Opcode repräsentiert. 244 Instruktionen werden mit 8-Bit-Opcodes kodiert. Die verbleibenden 256 Instruktionen werden durch zwei Bytes kodiert, wobei das erste Byte stets der Präfix `0xCB` ist und das zweite Byte die Funktion bestimmt. Einige Instruktionen nutzen zudem noch weitere Daten als Operanden, sodass mehrere Bytes für die Abarbeitung mancher Instruktionen gelesen werden müssen.

Die Abarbeitungsdauer variiert, da Instruktionen verschiedenartige Zwischenschritte benötigen. Ein einzelner Zwischenschritt beansprucht stets einen Maschinenzyklus (9, 10).

Eine vollständige tabellarische Übersicht aller Instruktionen sowie deren wesentliche Eigenschaften ist in (9) gegeben. Detaillierte Informationen über die Funktionsweise aller Instruktionen sind in (10) enthalten. Die folgende Abbildung enthält einen annotierten Auszug der Tabelle aus (9), aus dem exemplarisch die wesentlichen Informationen der Instruktion „AND A, B“ ersichtlich werden:

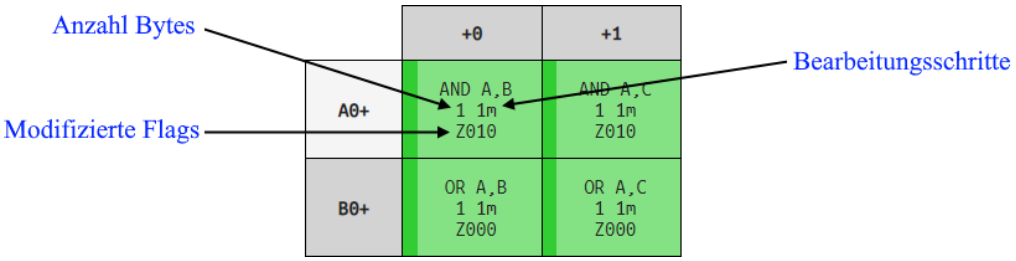

Abbildung 3 Beispielhafter Auszug der Opcode-Tabelle aus (9). Die deutschsprachigen Annotationen wurden zur Veranschaulichung ergänzt.

Die Instruktion „AND A, B“ wird durch den Opcode `0xA0` kodiert und benötigt lediglich ein Byte. Zur Berechnung wird nur ein einziger Bearbeitungsschritt benötigt. Zudem werden alle vier Zustands-Flags modifiziert. Die Flags Subtract und Carry werden genullt. Hingegen wird Half-carry gesetzt. Zero wird entweder genullt oder gesetzt, was von den Operanden zur Laufzeit abhängt (9, 10).

2.2.3 Interrupts

Bei einem Interrupt wird der eigentliche Programmablauf durch ein besonderes Ereignis vorübergehend unterbrochen, welches dann durch eine entsprechende Ablaufroutine verarbeitet wird. Die Game Boy-Hardware unterstützt fünf Arten von Interrupts (11):

Bit	Adresse	Bezeichnung	Ereignis
0	`$0040`	VBLANK	PPU betritt die V-Blank-Phase (vgl. Kapitel 2.5.6).
1	`$0048`	STAT	Konfigurierbares Interrupt (siehe Kapitel 2.5.8).
2	`$0050`	TIMER	Der Timer ist ausgelaufen (vgl. Kapitel 2.2.4).
3	`$0058`	SERIAL	Die serielle Übertragung eines vollständigen Bytes ist abgeschlossen.
4	`$0060`	JOYPAD	Der Zustand eines Eingabeknopfs wechselt von „nicht betätigt“ zu „betätigt“ (vgl. Kapitel 2.7).

Sobald eine Interrupt-Voraussetzung aus der obigen Tabelle erfüllt ist, wird ein Interrupt angefragt, indem das entsprechende Bit im Register IF („Interrupt Flags“, `$FF0F`) gesetzt wird (11).

Das Register IE („Interrupt Enable“, `$FFFF`) gibt für jede Interrupt-Quelle an, ob eine entsprechende Interrupt-Anfrage verarbeitet werden kann. Wenn beispielsweise Bit 0 von IE nicht gesetzt ist, werden keine VBLANK-Interrupts verarbeitet, auch wenn das entsprechende Bit in IF gesetzt ist (11).

Ob Interrupts überhaupt verarbeitet werden, hängt vom Zustand der internen IME-Flag („Interrupt Master Enable“) ab. Wenn IME gesetzt ist, können Interrupts verarbeitet werden, andernfalls wird die Verarbeitung ausgesetzt. IME kann lediglich durch die Instruktionen EI und RETI eingeschaltet beziehungsweise durch die Instruktion DI ausgeschaltet werden (11).

Die Verarbeitung von Interrupts erfolgt im Zeitraum nach der Ausführung einer Instruktion und vor dem Laden der nächsten Instruktion. Wenn IME gesetzt ist und ein beliebiges IF-Bit sowie das entsprechende IE-Bit gesetzt sind, wird die Interrupt Service Routine (ISR) ausgeführt. Falls mehrere IF-Bits sowie die dazugehörigen IE-Bits gesetzt sind, wird das Interrupt-Signal mit der niedrigsten Adresse zuerst behandelt. VBLANK hat dabei die höchste Priorität, JOYPAD wiederum die niedrigste. Die ISR dauert fünf Maschinenzyklen und besteht laut (11) grob aus den folgenden Schritten:

1. In den ersten beiden Zyklen passiert nichts.
2. Der Programmzähler wird auf dem Stack-Speicher platziert. Das obere Byte von PC wird zuerst abgelegt, danach das untere Byte. Dies dauert zwei Zyklen.
3. Der Programmzähler wird auf die Adresse des Interrupts mit der höchsten Priorität gesetzt. Das entsprechende Bit in IF wird genullt und IME wird deaktiviert. Das Bit in IE bleibt unverändert.

2.2.4 Timer

Die CPU besitzt eine Timer-Komponente, mit der zeitkritische Aufgaben über ein Interrupt ausgeführt werden können. Den Kern dieser Komponente bildet der interne 16-Bit-Zähler SYSCLK, der nach jedem CPU-Takt um eins erhöht wird. Dieser Zähler kann zwar nicht direkt ausgelesen werden, jedoch werden die oberen acht Bits über das Register DIV („Divider“, `$FF04`) offengelegt. Wenn ein schreibender Zugriff auf DIV stattfindet, wird SYSCLK und somit auch DIV auf 0 gesetzt, unabhängig vom geschriebenen Wert (7).

Das Register TIMA („Timer Counter“, `$FF05`) speichert den eigentlichen Zähler, der für die Taktung von zeitkritischen Aufgaben genutzt wird. Der Wert von TIMA wird stets um eins erhöht. TAC („Timer Control“, `$FF07`) bestimmt, ob TIMA überhaupt erhöht werden soll. Zudem legt TAC die Frequenz fest, in der TIMA erhöht wird. Sobald TIMA überläuft, wird ein TIMER-Interrupt angefragt und der Wert des Registers TMA („Timer Modulo“, `$FF06`) wird in TIMA kopiert. TMA ermöglicht also eine feingranulare Anpassung der Frequenz, die in TAC festgelegt ist (7).

Je nach festgelegter Frequenz werden bestimmte Bits von SYSCLK ausgelesen, um zu überprüfen, ob TIMA erhöht werden soll. Nachdem diese Bits ihren Zustand zum vorherigen Takt gewechselt haben, wird TIMA erhöht. Eine Übersicht der verfügbaren Frequenzen sowie die Funktionsweise der Timer-Komponente sind in (7) präzisiert.

2.3 Speicher

Das Game Boy-System verfügt über einen 16-Bit-Bus, der die Adressierung von ROM, RAM und Registern ermöglicht. Die Datenübertragung erfolgt über einen 8-Bit-Bus. Der logische Adressraum umfasst 64 KiB und ist in mehrere Abschnitte unterteilt, die in der folgenden Tabelle zusammengefasst sind (12):

Start	Ende	Größe	Speicherinhalt
`$0000`	`$7FFF`	32 KiB	Bootstrap ROM & Game Pak ROM
`$8000`	`$9FFF`	8 KiB	Video RAM (VRAM)
`$A000`	`$BFFF`	8 KiB	External RAM (EXRAM)
`$C000`	`$DFFF`	8 KiB	Work RAM (WRAM)
`$E000`	`$FDFF`	~7,5 KiB	Echo RAM
`$FE00`	`$FE9F`	160 Byte	Object Attribute Memory (OAM)
`$FEA0`	`$FEFF`	96 Byte	Nicht nutzbar
`$FF00`	`$FF7F`	128 Byte	I/O-Register
`$FF80`	`$FFFE`	127 Byte	High RAM (HRAM)
`$FFFF`	`$FFFF`	1 Byte	Interrupt Enable-Register (IE)

2.3.1 Bootstrap ROM

Nachdem das Game Boy-System eingeschaltet wurde, führt der Prozessor zunächst ein Bootstrap-Programm aus, das auf der Konsole gespeichert ist. Dieses Programm überlagert ab Adresse `$0000` die Game Pak ROM. Im Anschluss an die Beendigung des Bootstrap-Programms ist die ROM des Game Paks adressierbar (12, 13). Der genaue Zweck des Bootstrap-Programms wird in Kapitel 2.8 erläutert.

2.3.2 Game Pak ROM

Über diesen Speicherbereich greift der Prozessor auf den Programmcode des Spiels zu. Obwohl dieser Bereich lediglich 32 KiB adressiert, gibt es Game Paks, auf denen Spiele mit größerem Datenvolumen gespeichert sind. Solche Spiele implementieren einen sogenannten Memory Bank Controller (MBC), um etwa den Zugriff auf größere Spieldaten zu realisieren. Üblicherweise erfolgt der Zugriff auf ROM-Speicher lesend. Es werden jedoch schreibende Zugriffe verwendet, um den MBC zu steuern (14).

2.3.3 External RAM

Einige Game Paks verfügen über eigene beschreibbare Speicherchips, um etwa den Spielfortschritt des Spielers zu sichern. Falls ein Game Pak einen externen Speicherchip unterstützt, wird dieser über External RAM (EXRAM) adressierbar. EXRAM ist optional und die Zusammensetzung sowie Funktionsweise ist individuell für jedes Game Pak (14).

2.3.4 Work RAM

Work RAM (WRAM) ist der Hauptarbeitsspeicher, der in der Konsole verbaut ist. Die CPU nutzt WRAM zur Speicherung von arbiträren Daten während der Laufzeit. Dieser Speicher ist nicht persistent, sodass der Dateninhalt beim Ausschalten des Game Boy-Systems verloren gehen kann (12).

2.3.5 Echo RAM

Echo RAM adressiert keinen weiteren Speicher. Stattdessen werden Zugriffe auf diesen Adressbereich in WRAM umgeleitet. Folglich haben alle Lese- und Schreibzugriffe auf den Bereich `[$E000, $FDFF]` denselben Effekt wie Zugriffe auf `[$C000, $DDFF]` (12).

2.3.6 High RAM

Dieser Speicherbereich wird ähnlich wie WRAM als einfacher Arbeitsspeicher genutzt. High RAM (HRAM) ist zwar nur 127 Byte groß und hat daher ein geringeres Datenvolumen im Vergleich zu WRAM, allerdings ermöglichen dedizierte CPU-Instruktionen einen schnelleren Zugriff auf den Adressraum `[$FF00, $FFFF]` (12).

2.3.7 Video RAM

Rohdaten, die für die Bildsynthese der PPU erforderlich sind, werden in Video RAM (VRAM) gespeichert. In Kapitel 2.5 wird die Nutzung dieses Speichers detailliert ausgeführt.

2.3.8 OAM

Im OAM (Object Attribute Memory) werden die Attribute von bewegbaren Bildobjekten, sogenannten Sprites, gespeichert. Dieser Speicher enthält die Attribute für bis zu 40 Sprites, wobei pro Sprite-Eintrag ein Block bestehend aus vier Bytes benötigt wird. Sowohl die PPU als auch die CPU greifen auf diesen Speicherbereich zu. Die genaue Zusammensetzung von Sprite-Objekten wird in Kapitel 2.5.5 genauer erklärt.

Zudem verfügt die CPU über einen DMA-Mechanismus, mit denen Daten aus einem Speicherbereich direkt ins OAM transferiert werden können, um so die CPU zu entlasten. Dieser Mechanismus wird in Kapitel 2.6 vorgestellt.

2.3.9 I/O-Register

Der Adressraum `[$FF00, $FF7F]` ist für I/O-Register vorgesehen, über welche die CPU mit den anderen Komponenten des Game Boy-Systems kommunizieren kann (12). Einzelne Register werden samt ihrer Bezeichnung, Adresse und Funktionsweise in den entsprechenden Unterkapiteln vorgestellt. Eine Auflistung aller verfügbaren adressierbaren Register ist im Anhang von (10) zu finden.

2.4 Game Pak

Als Game Pak wird eine ROM-Cartridge für die Game Boy-Konsole bezeichnet, auf welcher die Programmdaten in Form eines ROM-Abbilds gespeichert sind. Ferner beinhalten manche Cartridges sogenannte Memory Bank Controller (MBC), um weitere externe Hardware mit dem Game Boy-System zu verbinden. MBCs regeln aber auch, wie die Adressräume von ROM und EXRAM zugewiesen werden (14). Die folgende Tabelle stellt die allgemeine Datenstruktur eines ROM-Abbilds dar:

Offset	Größe	Bedeutung
`0x0000`	64 Byte	Routinen der RST-Instruktionen
`0x0040`	8 Byte	VBLANK-Interrupt-Routine
`0x0048`	8 Byte	STAT-Interrupt-Routine
`0x0050`	8 Byte	TIMER-Interrupt-Routine
`0x0058`	8 Byte	SERIAL-Interrupt-Routine
`0x0060`	8 Byte	JOYPAD-Interrupt-Routine
`0x0068`	152 Byte	Arbiträr
`0x0100`	4 Byte	Einstiegspunkt
`0x0104`	48 Byte	Nintendo-Logo
`0x0134`	28 Byte	Metadaten
`0x0150`	Rest	Arbiträr

Die Daten im Bereich von `0x0100` bis `0x014F` werden auch als Cartridge Header bezeichnet (15). Die Ablaufroutinen aller RST-Instruktionen sowie aller Interrupts befinden sich an festgelegten Punkten innerhalb des ROM-Abbilds (10, 11). Der Einstiegspunkt wird nach der Bootsequenz ausgeführt. Dieser Prozess ist in Kapitel 2.8 genauer beschrieben.

2.4.1 ROM-Metadaten

Die Metadaten der ROM-Datei speichern allgemeine Informationen über das Spiel sowie die Hardwareanforderungen, unter denen das Spiel ausgeführt werden soll. Diese Aspekte lassen sich über die Metadaten ermitteln (15):

- Titel des Spiels
- Versionsnummer
- Herstellercode
- Kompatibilität mit Super Game Boy-Systemen
- Kompatibilität mit Game Boy Color-Systemen
- Verbaute Cartridge-Hardware (z.B. MBC)
- Größe der ROM-Datei
- Größe des externen RAM
- Header-Prüfsumme
- ROM-Prüfsumme

2.4.2 Memory Bank Controller

Ein Memory Bank Controller (MBC) ermöglicht die Erweiterung der Adressierung von externem Speicher (ROM und EXRAM), sodass auch Spiele mit höherem Datenvolumen auf einem Game Boy-System ausgeführt werden können. Die externen Speicher werden in kleinere Abschnitte aufgeteilt, von denen wiederum nur eine Teilmenge gleichzeitig adressierbar ist. Ein solcher Abschnitt wird auch als Speicherbank bezeichnet. Die Speicherbänke, die über ROM und EXRAM verfügbar sind, werden über den MBC gesteuert und können bei Bedarf ausgetauscht werden (14). Die Implementierungen von MBCs sind vielfältig, um je nach Bedarf des Spiels unterschiedliche Speichervolumen oder etwa weitere externe Hardware zu unterstützen. Detaillierte Informationen über die verschiedenen MBC-Typen sind in (10) und (14) zu finden.

Im Folgenden wird der MBC1-Chip vorgestellt, der in zahlreichen frühen Game Boy-Spielen zum Einsatz kam (16). Der MBC1 unterstützt ROM-Größen bis zu 2 MiB (128 Speicherbänke à 16 KiB) sowie EXRAM-Größen bis zu 32 KiB (4 Speicherbänke à 8 KiB). Um die Speicherverwaltung zu realisieren, werden vier Register genutzt (10).

Speicherzugriffe auf EXRAM sind standardmäßig deaktiviert. Um diese zu aktivieren, muss die 4-Bit-Zahl `0b0101` in das RAMG-Register (`[$0000, $1FFF]`) geschrieben werden. Speicherzugriffe auf EXRAM können wieder deaktiviert werden, sobald eine andere Zahl in RAMG geschrieben wird (10).

Die genutzten Speicherbänke werden durch das 5-Bit-Register BANK1 (`[$2000, $3FFF]`) sowie das 2-Bit Register BANK2 (`[$4000, $5FFF]`) bestimmt. BANK1 und BANK2 werden jeweils mit den Werten `0b00001` und `0b00` initialisiert. Beim Versuch `0b00000` in BANK1 zu schreiben, wird stets der Wert `0b00001` geschrieben. Die Selektierung der Speicherbänke anhand von BANK1 und BANK2 wird wiederum durch das 1-Bit-Register MODE (`[$6000, $7FFF]`) bestimmt (10).

Die folgende Tabelle zeigt, wie die Bits der physischen ROM-Adresse bei einem Lesezugriff über eine gegebene ROM-Adresse `addr` zusammengesetzt werden (10):

Virtueller Adressraum	Bits der physischen Adresse		
	Banknummer		Offset in Bank
	20-19	18-14	13-0
`[$0000, $3FFF], MODE = 0b0`	`0b00`	`0b00000`	`addr<13:0>`
`[$0000, $3FFF], MODE = 0b1`	`BANK2`	`0b00000`	`addr<13:0>`
`[$4000, $7FFF]`	`BANK2`	`BANK1`	`addr<13:0>`

Auf eine ähnliche Weise wird die physische EXRAM-Adresse für beliebige Zugriffe über eine gegebene EXRAM-Adresse `addr` berechnet (10):

Virtueller Adressraum	Bits der physischen Adresse	
	Banknummer	Offset in Bank
	14-13	12-0
`[$A000, $BFFF], MODE = 0b0`	`0b00`	`addr<12:0>`
`[$A000, $BFFF], MODE = 0b1`	`BANK2`	`addr<12:0>`

2.5 Pixel Processing Unit

Die Pixel Processing Unit (PPU) verarbeitet die rohen Bilddaten in VRAM und OAM und stellt diese in Form von Pixeln auf dem LC-Display (kurz: LCD) dar. Zunächst lädt die CPU die rohen Bilddaten des Spiels in die entsprechenden Grafikspeicher hinein. Die PPU liest diese Daten aus, um Frames auf dem LCD zu rendern. Der Bildschirm hat eine Größe von 160×144 Pixeln und unterstützt vier Graustufen. Die Bildfrequenz beträgt ungefähr 59,7 Hz (6, 17).

Das Verhalten der PPU ist von der Konfiguration des LCDC-Registers („LCD Control“, `$FF40`) abhängig. Zudem können Informationen über den aktuellen Zustand der PPU über das Register STAT („LCD Status“, `$FF41`) ausgelesen werden (17).

2.5.1 Frames

Die drei Bildebenen Background, Window und Sprites bilden ein Frame. Die Sichtbarkeit einzelner Ebenen kann über LCDC festgelegt werden (6, 17).

- **Background** wird aus einem 256×256 Pixel großen Bild zusammengesetzt. Obwohl nur ein Ausschnitt von 160×144 Pixeln auf dem LCD sichtbar ist, kann der sichtbare Bereich durch die Register SCX („Scroll X“, `$FF43`) und SCY („Scroll Y“, `$FF42`) festgelegt und verschoben werden.
- **Window** misst 160×144 Pixel und überlagert die Background-Schicht. Der sichtbare Ausschnitt dieser Ebene wird nicht über die Register SCX und SCY beeinflusst. Window wird stattdessen zur Darstellung von Grafiken verwendet, die stets an einer festen Bildschirmposition angezeigt werden sollen, wie etwa eine GUI. Die Position dieser Ebene wird über die Register WX („Window X“, `$FF4B`) und WY („Window Y“, `$FF4A`) festgelegt.
- **Sprites** sind Grafikobjekte, die sich frei auf dem Bildschirm bewegen können. Anders als die Grafiken der Background- und Window-Ebenen, können Sprites transparente Pixel haben, sodass an diesen Stellen die hinteren Ebenen stets sichtbar bleiben. In Kapitel 2.5.5 werden diese Objekte genauer beschrieben.

Im Folgenden wird die Zusammensetzung eines Frames anhand der einzelnen Bildebenen visuell verdeutlicht (6).

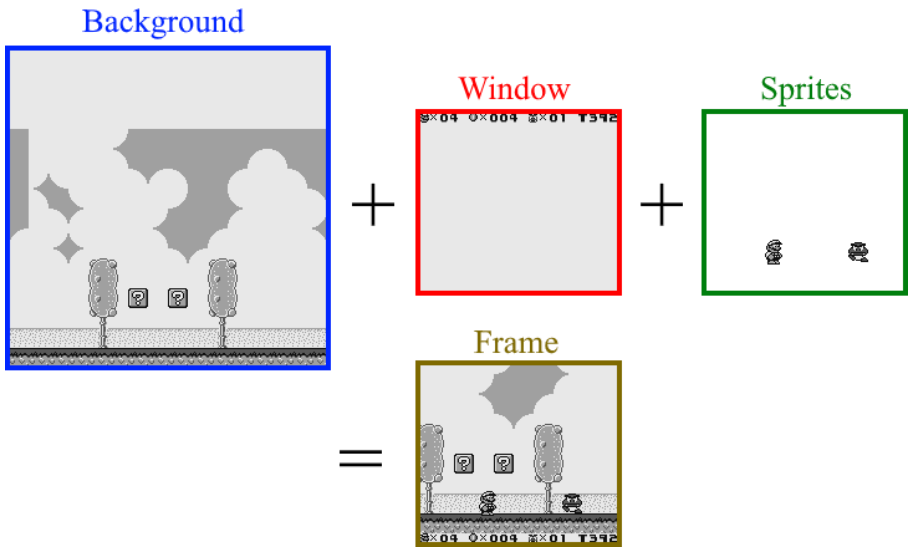

Abbildung 4 Bildebenen eines Frames. Beispielgrafiken aus (6) entnommen. Farbliche Hervorhebungen wurden zur Verdeutlichung ergänzt.

2.5.2 Tiles

Die Grafiken des Game Boys werden auf Basis von kleineren Bildern gerendert, sogenannten Tiles, die stets 8×8 Pixel groß sind. Ein Tile kodiert keine Farbwerte. Stattdessen wird jedem Pixel eine Farb-ID von 0 bis 3 zugewiesen. Die entsprechende Farbe einer Farb-ID wird über eine Farbpalette bestimmt (17).

Es können gleichzeitig bis zu 384 Tiles im Speicherbereich `[$8000, $97FF]` (VRAM) hinterlegt werden, wobei diese auf drei Blöcke mit jeweils 128 Tiles aufgeteilt sind. Für die unterschiedlichen Bildebenen können nur Tiles aus festgelegten Blöcken verwendet werden. Das folgende Schaubild veranschaulicht die Speicheraufteilung der 384 Tiles sowie die Bildebenen, die von diesen Tiles Gebrauch machen können (17):

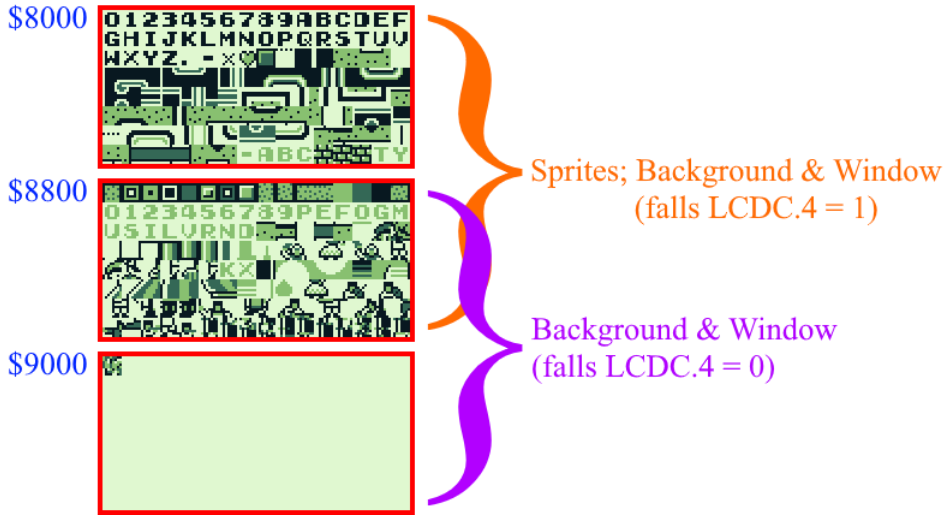

Abbildung 5 Nutzbare Tiles der jeweiligen Bildebenen.

Ein Pixel benötigt lediglich 2 Bits zum Speichern der Farb-ID, sodass ein ganzes Tile stets 16 Bytes groß ist. Jede Zeile eines Tiles wird durch paarweise-benachbarte Bytes repräsentiert. Das erste Byte besteht aus den unteren Bits (Bit 0) der Farb-IDs der entsprechenden Zeile, das zweite Byte besteht wiederum aus den oberen Bits (Bit 1) der Farb-IDs. In beiden Bytes steht Bit 7 für das ganz linke und Bit 0 für das ganz rechte Pixel (17). Dieses Prinzip wird in der folgenden Grafik exemplarisch demonstriert:

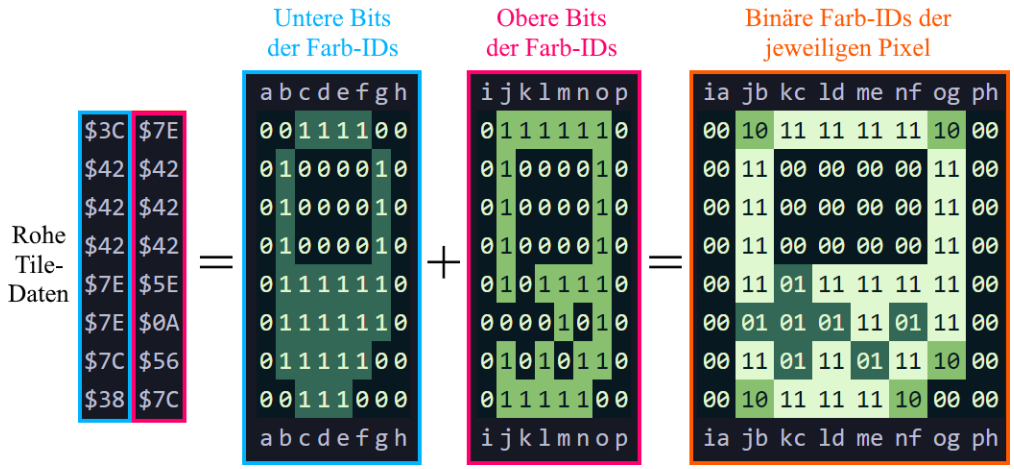

Abbildung 6 Exemplarische Interpretation von Tile-Daten. Entnommen aus (17). Die farblichen Annotationen und Hervorhebungen wurden zur Verdeutlichung ergänzt.

2.5.3 Tilemaps

Eine Tilemap definiert, wie einzelne Tiles zusammengesetzt werden müssen, um die Background- und Window-Ebenen darzustellen. Der Game Boy verfügt über zwei Tilemaps, die jeweils aus 32×32 Tiles bestehen. Sie speichern jedoch nicht rohe Tile-Daten, sondern enthalten lediglich Referenzen auf Tiles in Form von Byte-Indizes. Beide Tilemaps werden in VRAM gespeichert, jeweils in den Bereichen `[$9800, $9BFF]` und `[$9C00, $9FFF]`. LCDC bestimmt, welche Tilemap den Bildebenen zugeordnet wird (17).

2.5.4 Paletten

Um die Farbwerte einzelner Farb-IDs eines Tiles zu bestimmen, werden sogenannte Farbpaletten verwendet. Der Game Boy unterstützt drei Paletten, die über I/O-Register offengelegt werden. Das Register BGP („Background Palette“, `$FF47`) legt die Farbwerte für Tiles der Background- und Window-Ebenen fest. Die Farben von Sprite-Objekten werden über die Register OBP0 („Object Palette 0“, `$FF48`) und OBP1 („Object Palette 1“, `$FF49`) definiert (17).

Aus der folgenden Tabelle geht hervor, welche Bits der Register mit welchen Farb-IDs korrespondieren:

Bits	7-6	5-4	3-2	1-0
Farbwert für…	Farb-ID 3	Farb-ID 2	Farb-ID 1	Farb-ID 0

Die definierten Farbwerte für Farb-ID 0 der Paletten OBP0 und OBP1 werden ignoriert, da diese Farb-ID transparente Pixel von Sprite-Objekten repräsentieren.

Die folgende Tabelle listet alle verfügbaren Farbwerte des originalen Game Boys auf (17):

Farbwert	Farbe
0	Weiß
1	Hellgrau
2	Dunkelgrau
3	Schwarz

2.5.5 Sprites

Sprites sind Grafikobjekte, die sich unabhängig von den Background- und Window-Ebenen frei auf dem Bildschirm bewegen können. Anders als Tiles der anderen Bildebenen können Sprites auch transparent sein. Die PPU kann bis zu 40 solcher bewegbaren Objekte darstellen. Ein Sprite wird stets von einem Tile oder zwei benachbarten Tiles gebildet, sodass Sprites entweder 8×8 oder 8×16 Pixel groß sind. Die Größe von Sprites wird über das Register LCDC bestimmt. Alle 40 Sprite-Objekte werden in OAM definiert (6, 17).

Jeder Sprite-Eintrag besteht aus vier Bytes, die alle dasselbe Format teilen (17):

- **Byte 0 – Y-Position**: Dieser Wert bestimmt die vertikale Position. Um die tatsächliche Position auf dem LCD zu berechnen, muss zu diesem Wert 16 addiert werden. Die folgende Grafik veranschaulicht, wie sich dieses Byte auf die vertikale Position der unterschiedlichen Sprite-Größen auswirkt:

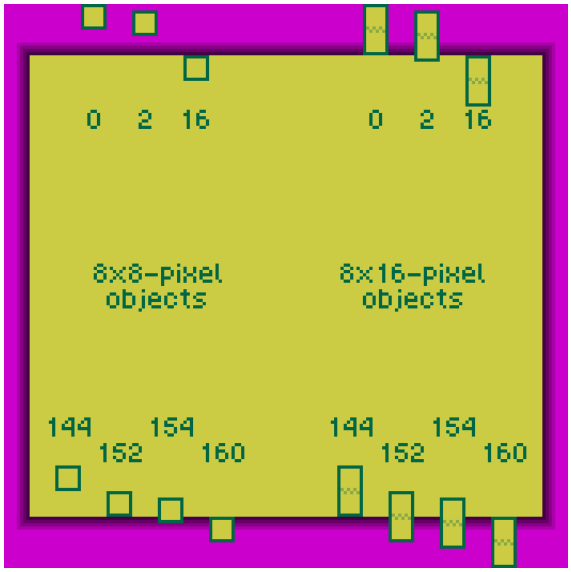

Abbildung 7 Reale vertikale Positionen von Sprite-Objekten. Entnommen aus (17).

- **Byte 1 – X-Position**: Das zweite Byte wird zur Berechnung der horizontalen Position genutzt. Für die reale Position auf dem LCD muss zu diesem Wert 8 addiert werden.
- **Byte 2 – Tile-ID**: Für 8×8-Sprites verweist dieses Byte auf das Tile, welches für die grafische Darstellung des Sprites genutzt werden soll. 8×16-Sprites werden dagegen paarweise aus zwei Tiles gebildet. Hierbei repräsentiert dieses Byte den Index des oberen Tiles. Für diese Objekte wird das unterste Bit der Tile-ID ignoriert, sodass das obere Tile niemals einen ungeraden Index haben kann.
- **Byte 3 – Attribute**: Dieses Byte speichert mehrere Informationen über das Sprite-Objekt ab. Es bestimmt, welche der beiden Paletten OBP0 und OBP1 genutzt wird, ob das Sprite-Objekt horizontal oder vertikal gespiegelt wird, und schließlich, mit welcher Priorität das Objekt gerendert wird. Falls das Priority-Bit gesetzt ist, werden Pixel der Background- und Window-Ebenen mit den Farb-IDs 1 bis 3 über das Sprite-Objekt gerendert. Andernfalls wird das komplette Sprite-Objekt über den hinteren Ebenen gerendert.

2.5.6 Rendering

Das Rendering eines vollständigen Frames ist kein atomarer Prozess. Stattdessen durchläuft die PPU vier Modi, um ein Frame Pixel für Pixel auf dem LCD darzustellen. Ein Frame wird stets zeilenweise von oben nach unten und spaltenweise von links nach rechts gerendert. Der Durchlauf einer ganzen Zeile wird auch als Scanline bezeichnet. Eine Scanline dauert 456 Maschinentakte. Ein Frame benötigt stets 154 Scanlines. Die Nummer der aktuellen Scanline kann über das Register LY („LCD Y Coordinate“, `$FF44`) ausgelesen werden (17).

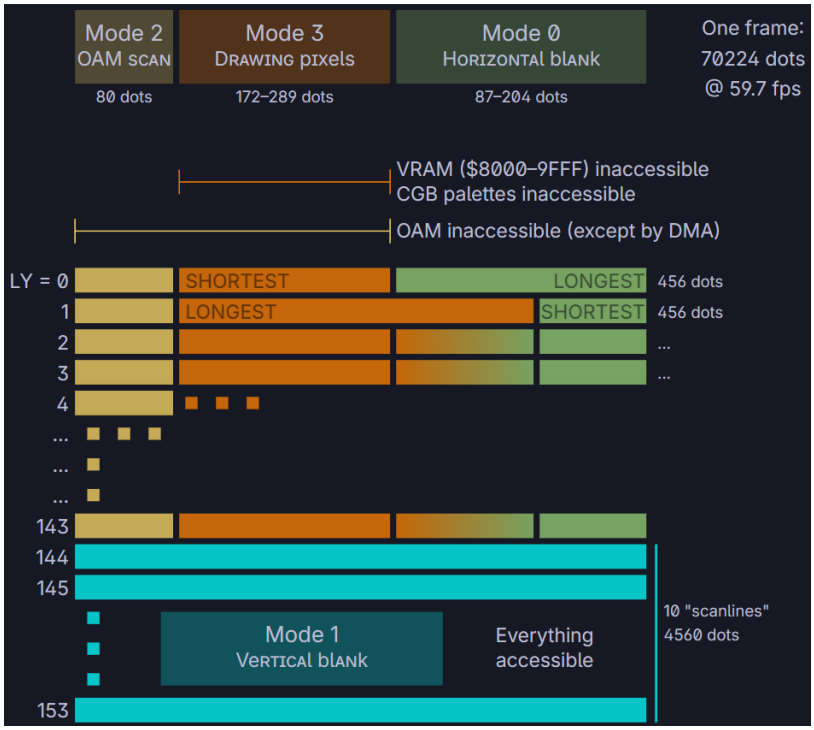

Abbildung 8 Übersicht über die vier PPU-Phasen sowie die entsprechenden Speicherzugriffs-einschränkungen für die CPU. Ein Dot entspricht einem Maschinentakt. Entnommen aus (17).

Im Folgenden wird die Funktionsweise der einzelnen Modi erläutert (17):

- **Mode 2 – OAM-Scan**: Jede der ersten 144 Scanlines startet in Mode 2, um das Rendering einer Scanline vorzubereiten. Die PPU wählt bis zu 10 Sprite-Objekte aus, die auf der aktuellen Zeile gezeichnet werden sollen. Alle weiteren Sprite-Objekte auf der Zeile werden ignoriert. Die CPU hat während dieser Phase keinen Zugriff auf das OAM. Anschließend betritt die PPU die Drawing-Phase.
- **Mode 3 – Drawing**: In dieser Phase werden 160 Pixel der Scanline gezeichnet, indem die Grafiken der Background-, Window- und Sprites-Ebenen kombiniert werden. In dieser Phase ist die CPU nicht in der Lage, OAM sowie VRAM zu nutzen, da die PPU diese Speicherbereiche für das Rendering ausliest. Danach betritt die PPU die H-Blank-Phase.
- **Mode 0 – Horizontal-Blank (H-Blank)**: Hierbei handelt es sich um einen Wartezustand, in dem die PPU nicht arbeitet. Folglich kann die CPU wieder auf OAM und VRAM zugreifen. Falls in dieser Phase die letzte visuelle Scanline (d.h. LY=143) gezeichnet wurde, betritt die PPU anschließend Mode 1. Ansonsten beginnt die folgende Scanline wieder in Mode 2.
- **Mode 1 – Vertical-Blank (V-Blank)**: Diese Phase kennzeichnet, dass ein vollständiges Frame auf dem LCD gerendert wurde. Sobald Mode 1 beginnt, wird ein VBLANK-Interrupt angefragt. Während der V-Blank-Phase verarbeitet die PPU 10 „leere“ Scanlines und stellt damit einen Wartezustand dar, um die Zeit bis zum Beginn des nächsten Frames zu überbrücken. In dieser Phase hat die CPU uneingeschränkten Zugriff auf OAM und VRAM.

Die genaue Dauer von Mode 3 ist variabel und hängt von diversen Faktoren, wie etwa der Anzahl von Sprite-Objekten auf der Scanline, ab. Je länger Mode 3 dauert, umso kürzer dauert Mode 0. In (17) werden die Faktoren, die die Dauer von Mode 0 und 3 beeinflussen, präzisiert.

2.5.7 LYC-Register

Der Wert aus LY wird jederzeit mit dem Register LYC („LY Compare“, `$FF45`) verglichen. Falls LY mit diesem Register übereinstimmt, wird Bit 2 des Registers STAT auf 1 gesetzt und gegebenenfalls ein STAT-Interrupt angefordert (siehe Kapitel 2.5.8). Andernfalls wird das entsprechende STAT-Bit auf 0 gesetzt und es erfolgt keine Interrupt-Anfrage (17).

2.5.8 STAT-Interrupts

Wie in Kapitel 2.2.3 eingeführt, ist das Auslösen des STAT-Interrupts beliebig konfigurierbar. Die folgenden Ereignisse können, sofern das entsprechende STAT-Bit gesetzt ist, ein STAT-Interrupt anfordern (17):

- **STAT-Bit 3 = 1**: Mode 0 (H-Blank) beginnt.
- **STAT-Bit 4 = 1**: Mode 1 (V-Blank) beginnt.
- **STAT-Bit 5 = 1**: Mode 2 (OAM-Scan) beginnt.
- **STAT-Bit 6 = 1**: LY = LYC.

Es können zwar mehrere Ereignisse gleichzeitig selektiert werden, jedoch kann es vorkommen, dass STAT-Interrupts blockiert werden, wenn bereits zuvor ein Interrupt durch ein STAT-Ereignis ausgelöst wird, das wiederum noch nicht verarbeitet wurde. Dieses Phänomen wird in (11) genauer beschrieben.

2.6 Direct Memory Access

Direct Memory Access (DMA) ist ein effizienter Mechanismus, um Daten in das OAM zu kopieren. Bei diesem Verfahren wird lediglich ein Maschinenzyklus benötigt, um ein einzelnes Byte zu kopieren. Ein solcher Datentransfer kann jedoch nicht abgebrochen werden. Die Größe der zu kopierenden Daten ist vordefiniert und kann daher nicht verändert werden. Bei dieser Art des DMA-Transfers werden stets alle 160 Bytes ins OAM überschrieben (10).

Um einen DMA-Transfer durchzuführen, muss zunächst die Startadresse der Daten spezifiziert werden, die kopiert werden sollen. Dabei soll das untere Byte dieser Adresse `0x00` sein. Das obere Byte wird in das Register DMA (`$FF46`) geschrieben (10). Um etwa Daten beginnend ab Adresse `$7F00` per DMA-Transfer zu kopieren, muss das Byte `0x7F` in das DMA-Register geschrieben werden. Infolgedessen werden die Daten von `[$7F00, $7F9F]` nach `[$FE00, $FE9F]` kopiert.

Nachdem das DMA-Register beschrieben wurde, startet der DMA-Transfer im nächsten Maschinenzyklus. In dieser Phase kann die CPU nur auf HRAM zugreifen, da Zugriffe auf andere Speicher blockiert sind (18).

Speicherzugriffe durch die DMA-Komponente agieren aufgrund eines abgewandelten Adressierungsverhaltens anders als übliche Speicherzugriffe wie etwa durch die CPU. Die Daten in `[$FE00, $FFFF]` können infolgedessen nicht kopiert werden. Das genaue Verhalten ist in dieser Situation unklar und hängt von anderen Faktoren, wie etwa dem eingelegten Game Pak, ab (10).

2.7 Joypad

Die acht Eingabeknöpfe auf der Vorderseite des Game Boys werden auch als Joypad bezeichnet. Es gibt die vier Aktionstasten „A“, „B“, „Start“ und „Select“ sowie die vier Richtungstasten „Hoch“, „Runter“, „Rechts“ und „Links“. Jeder Knopf kann entweder gedrückt oder nicht gedrückt sein (19).

Die Zustände der einzelnen Knöpfe können über das Register JOYP („Joypad“, `$FF00`) ausgelesen werden. Dieses Register kann entweder nur die Zustände der Aktionstasten oder nur die Zustände der Richtungstasten enthalten. Die beschreibbaren Bits 4 und 5 legen fest, welche Kategorie von Knöpfen ausgelesen werden soll (19). Die Bedeutung der einzelnen Bits von JOYP sind in der folgenden Tabelle aufgeführt:

Bit	Bedeutung
0	Zustand von „Rechts“ oder „A“.
1	Zustand von „Links“ oder „B“.
2	Zustand von „Hoch“ oder „Select“.
3	Zustand von „Runter“ oder „Start“.
4	Auslesen der Zustände von Richtungstasten.
5	Auslesen der Zustände von Aktionstasten.
6	Nicht implementiert.
7	Nicht implementiert.

Für die Zustandsbits repräsentiert der Wert 0 eine gedrückte Taste, der Wert 1 dagegen eine nicht betätigte Taste. Die folgende Tabelle stellt dar, wie sich die Zustandsbits basierend auf den Werten von Bit 4 und 5 verhalten:

Bit 5	Bit 4	Bedeutung von Zustandsbits
0	0	Nicht definiert.
0	1	Enthalten Zustände der Aktionstasten.
1	0	Enthalten Zustände der Richtungstasten.
1	1	Alle vier Bits sind stets auf 1 gesetzt.

Sobald eine Taste den Zustand von „nicht gedrückt“ auf „gedrückt“ wechselt und die entsprechende Taste durch Bit 4 oder 5 selektiert ist, wird ein JOYPAD-Interrupt angefragt (19).

2.8 Bootsequenz

Beim Einschalten des Game Boy-Systems wird nicht das Spiel ausgeführt. Der Programmzähler wird stattdessen auf die Adresse `$0000` gesetzt, an der sich zunächst das Bootstrap-Programm befindet. Dadurch wird die Game Pak-ROM im Adressraum [`$0000`, `$00FF`] von der Boot-ROM überlagert. Dieses Programm unterscheidet sich je nach Game Boy-Modell, jedoch dienen alle Implementierungen dieses Programms der Initialisierung des Game Boy-Systems (10, 13). In diesem Unterkapitel wird ein Überblick über das Verhalten des Bootstrap-Programms des originalen Game Boys gegeben.

Zuerst wird das Nintendo-Logo aus dem Header der Cartridge-ROM ausgelesen und in VRAM geschrieben. Anschließend wird das Logo von oben nach unten scrollend auf dem LCD dargestellt und ein kurzer Jingle abgespielt (13). Die Game Pak-ROM wird erst im nächsten Schritt anhand der Inhalte des Headers validiert. Zunächst wird sichergestellt, dass die Bilddaten des Nintendo-Logos im Header den entsprechenden Daten in der Boot-ROM gleichen. Des Weiteren wird die Prüfsumme des Headers berechnet und mit der im Header hinterlegten Prüfsumme abgeglichen. Falls diese Überprüfungen fehlschlagen, wird das Spiel nicht gestartet. Andernfalls wird eine Zahl ungleich Null in das Register BOOT (`$FF50`) geschrieben, wodurch die Bootstrap-ROM entladen wird. Schließlich zeigt der Programmzähler auf die Adresse `$0100`, an der sich der Einstiegspunkt des Spielprogramms befindet (10, 13).

2.9 Stand der Technik

Die Entwicklung von Game Boy-Emulatoren ist seit einigen Jahrzehnten sehr aktiv, sodass bereits zahlreiche Game Boy-Emulatoren existieren, die sich großer Beliebtheit erfreuen und eine Vielzahl nützlicher Funktionen bieten, die über die Funktionalität der emulierten Konsole hinausgehen. Einige Emulationsprojekte dienen wiederum der Dokumentation der Game Boy-Hardware und verzichten zugunsten von Verständnis über die Hardware auf weitere Funktionalitäten, die das Spielerlebnis verbessern. In diesem Abschnitt werden einige dieser Projekte und Lösungen vorgestellt.

2.9.1 BGB

Einer der bekanntesten sowie ältesten Emulatoren ist BGB, der erstmals 2001 veröffentlicht wurde und seitdem stetig weiterentwickelt wird (20). Laut Entwickler ist BGB mit einer akkuraten Emulation aller Hardware-Komponenten des Game Boys sowie einer umfangreichen Kompatibilität zu Spielen ausgestattet, die darauf ausgeführt und gespielt werden können. Diese Anwendung unterstützt nicht nur das erste Game Boy-Modell, sondern auch neuere Modelle wie den Game Boy Color.

Zusätzlich bietet BGB eine Vielzahl an Funktionen, die das Spielerlebnis verbessern sollen, darunter die Unterstützung von Savestates, verschiedenen Eingabegeräten wie Gamepads und auch Online-Multiplayer. Für Entwickler von eigenen Game Boy-Spielen bietet dieser Emulator auch nützliche Entwicklertools, mit denen beispielsweise einzelne Speicher und Register des Emulators ausgelesen und manipuliert werden können. BGB ist jedoch nicht quelloffen, sodass die Implementierungsdetails ohne Reverse Engineering nicht nachvollziehbar sind.

2.9.2 Mooneye GB

Mooneye GB ist ein Forschungsprojekt, das sich mit der akkuraten und klar dokumentierten Emulation des Game Boys befasst (21). Laut des Entwicklers gibt es einige sehr genaue Emulatoren, wie die zuvor vorgestellte Anwendung BGB, die jedoch nicht klar dokumentiert sind und somit keine gute Referenz für Entwickler anderer Emulatoren darstellen. Mooneye GB soll genau diese Lücke füllen, indem möglichst klar und präzise erläutert wird, warum und wie bestimmtes Hardware-verhalten emuliert wird.

2.9.3 Virtual Console

Auch der Videospielgigant Nintendo erkannte das Potenzial von Emulation und entwickelte unter der Marke Virtual Console eine hauseigene Lösung für das Spielen klassischer Videospiele. Unter dieser Marke wurden Spiele verschiedener Nintendo-Konsolen neu veröffentlicht, die einzeln von Kunden gekauft und als digitale Anwendungen auf modernen Konsolen heruntergeladen werden konnten. Auf dem Nintendo 3DS konnten beispielsweise zahlreiche Game Boy-Spiele erworben und gespielt werden. Neben der vollständigen Emulation eines Spiels unterstützten einige dieser Anwendungen auch Savestates, sodass der Fortschritt von Spielen gespeichert werden kann, die diese Funktion normalerweise nicht unterstützen. Es wurde nur eine begrenzte Auswahl an Game Boy-Spielen unter dieser Marke veröffentlicht. Seit der Schließung des eShops im Frühjahr 2023 sind keine Virtual Console-Titel mehr auf dem Nintendo 3DS erwerbbar (22). Der Konzern verzichtete auf die Fortsetzung dieser Marke im Rahmen der Nintendo Switch, der aktuellen Videospielkonsole Nintendos.

2.9.4 Nintendo Switch Online

Nintendo Switch Online ist ein kostenpflichtiges Abonnement, das primär für Online-Multiplayer auf der Nintendo Switch-Konsole erforderlich ist. Abonnenten erhalten zudem Zugang zu einer wachsenden Bibliothek klassischer Nintendo-Spiele, die auf der Nintendo Switch-Konsole emuliert werden können (23). Auch diese hauseigene Lösung ermöglicht das Speichern des Spielfortschritts in Form von Savestates. Es ist jedoch erstmals offiziell möglich, das Spielgeschehen an beliebigen Stellen zurückzuspulen, um so etwa schwierige Passagen in Spielen einfacher bewältigen zu können. Seit einigen Jahren erweitert Nintendo die Auswahl an Spielen in unregelmäßigen Abständen. Einige beliebte Titel, wie „Pokémon Rote Edition“, fehlen bisher jedoch.

3 Praxis

In diesem Kapitel werden Konzeption und Implementierung des prototypischen Emulators beschrieben.

3.1 Konzeption

Die Entwicklung einer Anwendung zur Emulation von Game Boy-Konsolen setzt ein tiefes Verständnis der entsprechenden Hardware und beteiligten Systembestandteile voraus. Die erste Entwicklungsphase diente der Recherche des Aufbaus und der Funktionsweise eines Game Boy-Systems, die in Kapitel 2 vorgestellt wurden. Auf Basis dieses Wissens konnte eine Unterteilung der Game Boy-Architektur in Komponenten vorgenommen werden, um so einen vorläufigen Plan für die Implementierung des Emulators aufzubauen. Der initiale komponentenbasierte Entwurf ist im nachstehenden UML-Komponentendiagramm festgehalten:

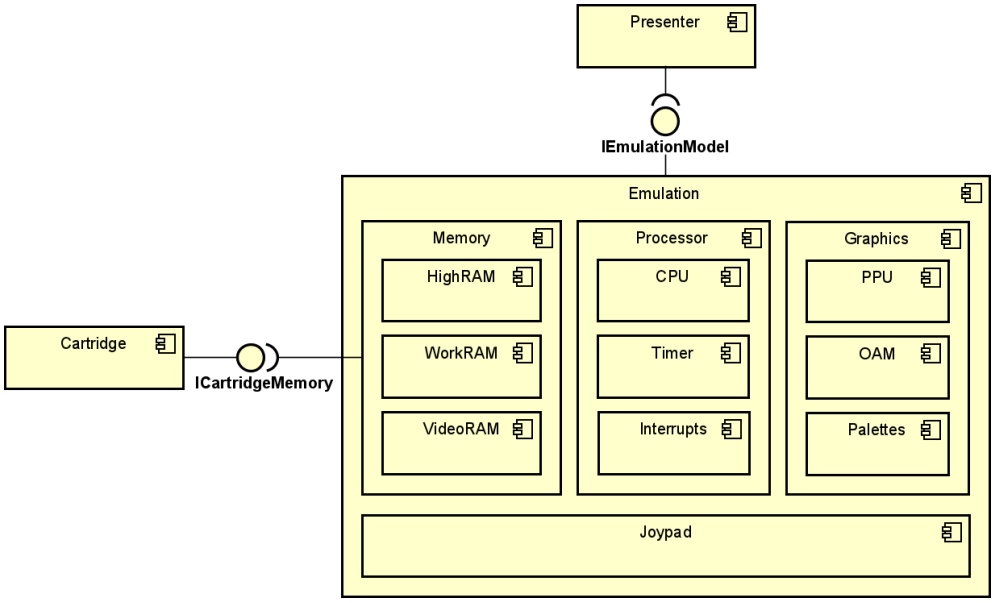

Abbildung 9 Früher Entwurf eines einfachen UML-Komponentendiagramms des Emulators.

Dieser Entwurf sah die folgende Hauptkomponenten vor:

- **Emulation** bildet den Kern des emulierten Game Boy-Systems und enthält alle Bestandteile, die auch in einem Game Boy verbaut sind. Um eine möglichst hohe Kohäsion anzustreben, wurden Bestandteile mit einer starken Bindung zueinander in größere Unterkomponenten (Memory, Processor, Graphics) zusammengefasst. Dies bedingt zudem lose Kopplung, da diese Unterkomponenten nur geringfügig miteinander verknüpft sind. Die PPU arbeitet beispielsweise unabhängig von der CPU, wobei diese über I/O-Register kommunizieren.
- **Cartridge** repräsentiert ein konkretes Spiel und implementiert die Funktionalität des MBCs, der zur Ausführung des Spiels benötigt wird. Konkret soll diese Komponente Speicherzugriffe auf externe Speicher (ROM und EXRAM) emulieren.
- **Presenter** implementiert das User Interface des Emulators. Es stellt gerenderte Frames der PPU auf dem Bildschirm des Anwenders dar. Zudem werden Nutzereingaben verarbeitet, um so die Zustände einzelner Knöpfe des Joypads zu emulieren.

Infolgedessen konnte ein Plan für die Entwicklung eines Game Boy-Emulators erarbeitet werden:

1. Zunächst müssen die Technologien bestimmt werden, mit denen die Anwendung entwickelt werden soll. Darunter fällt auch die Wahl der Programmiersprache. Für die Wahl der Programmiersprache sind zudem die Anforderungen zur Implementierung der Komponenten entscheidend.
2. Die CPU führt Programmcode aus, den sie aus dem Speicher ausliest. Somit muss zunächst das Speichermanagement des Emulators entwickelt werden, um dem emulierten Prozessor Zugriff auf alle integrierten Speicher (VRAM, HRAM, etc.) sowie externen Speicher (ROM und EXRAM) zu ermöglichen.
3. Da der Prozessor den eigentlichen Programmcode des Spiels prozessiert und somit das Herzstück des Systems bildet, muss die Funktionsweise aller 500 Instruktion sowie der beteiligten Mechanismen (Interrupts und Timer) realisiert werden. Die korrekte Interpretation der Instruktionen muss dabei gewährleistet werden, da etwaige Fehler durch eine falsche Implementierung im späteren Entwicklungsverlauf erschwert aufzuspüren wären.
4. Nachdem eine funktionierende Emulation des Prozessors realisiert wurde, müssen die Bilddaten, die über die CPU in die verschiedenen Grafikspeicher geschrieben wurden, gerendert werden. In diesem Schritt wird die Funktionsweise der PPU und damit das Rendering von Frames realisiert, damit Spieler das Spielgeschehen auch visuell wahrnehmen können.
5. Um den Spielablauf beeinflussen zu können, muss die Funktionsweise einzelner Game Boy-Knöpfe durch ein herkömmliches Eingabegerät wie etwa einer Tastatur ermöglicht werden.

3.2 Technologie

Die Programmiersprache Java wurde gewählt, da Java-Anwendungen sehr portabel sind und daher auch auf unterschiedlichen Betriebssystemen ausgeführt werden können. Java ermöglicht als objektorientierte Programmiersprache die Verwendung von Konzepten, wie etwa der Generalisierung, welche beispielsweise für die Realisierung abstrakter Komponenten wie den unterschiedlichen MBC-Arten geeignet ist.

Zudem beinhaltet Java standardmäßig das GUI-Toolkit Swing, mit dem einfache GUI-Anwendungen entwickelt werden können, sodass für die Implementierung der Benutzeroberfläche keine weiteren externen Bibliotheken benötigt werden. Im Kontext des Emulators ermöglichen GUI-Anwendungen in Swing die Darstellung gerenderter Frames sowie die Verarbeitung von Nutzereingaben über Eingabegeräte, wie Tastatur und Maus.

3.3 Virtueller Game Boy

Das Zentrum des Emulators bildet die `Emulation`-Klasse, welche die implementierten Teilkomponenten verwaltet und steuert. Das folgende vereinfachte UML-Klassendiagramm stellt die essenziellen Elemente dieser Klasse sowie die Beziehungen zu beteiligten Klassen dar:

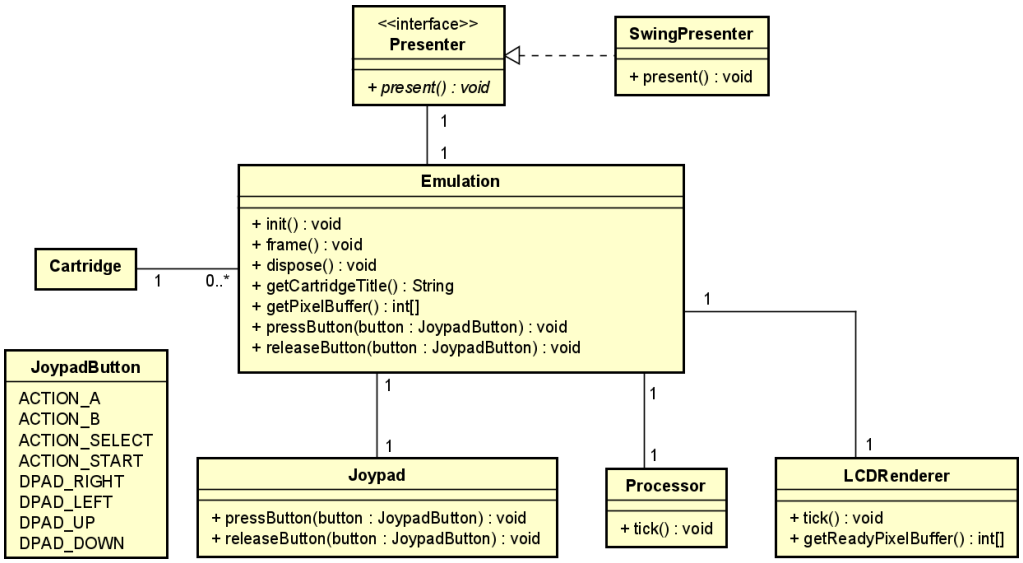

Abbildung 10 Vereinfachtes UML-Klassendiagramm des Emulatorkerns.

Zur Erzeugung einer `Emulation`-Instanz wird die ROM-Datei eines Spiels in Form eines `Cartridge`-Objekts und die Art der Hardware, die emuliert werden soll, benötigt. Im Rahmen dieser Arbeit wird lediglich der Hardware-Typ `GAME_BOY` unterstützt.

Die `init`-Methode dient der Initialisierung der Speicher und Register. Die Logik des emulierten Game Boy-Systems steuert die Methode `frame`, die stets 69.905 Maschinentakte emuliert. Dieser Wert entspricht dem abgerundeten Quotienten aus der maximalen Anzahl der Maschinentakte pro Sekunde (4.194.304) sowie der Bildfrequenz (60 Hz). Sobald die Emulation beendet wird, beispielsweise durch Schließen des Anwendungsfensters, wird zudem die Methode `dispose` aufgerufen, um Systemressourcen, wie offene Datei-Handles für EXRAM-Speicher, zu schließen.

Die visuelle Darstellung und die Delegation von Benutzereingaben werden über einen UI-Controller realisiert, der im Kontext dieser Arbeit als `Presenter` bezeichnet wird. Um den notwendigen Informationsaustausch zwischen `Emulation` und `Presenter` zu ermöglichen, stellt `Emulation` die folgenden Methoden zur Verfügung:

- `getCartridgeTitle()`: Gibt den Titel der emulierten Cartridge zurück.
- `getPixelBuffer()`: Gibt den Bildpuffer eines vollständig-gerenderten Frames zurück. Dieser Puffer enthält stets 160×144 ARGB-Farbpixel.
- `pressButton(button: JoypadButton)`: Bewirkt, dass der Zustand eines bestimmten emulierten Joypad-Knopfs auf „gedrückt“ gesetzt wird.
- `releaseButton(button: JoypadButton)`: Der Zustand eines emulierten Joypad-Knopfs wird auf „nicht gedrückt“ gesetzt.

Das `Presenter`-Objekt realisiert zudem auch die Hauptschleife der Anwendung, über die einzelne Frames prozessiert (`frame`-Methode von Emulation) und gerendert werden.

3.4 Speichermanagement

Der Datenaustausch zwischen der CPU und den anderen Komponenten des Game Boys erfolgt über adressierbare Speicher und stellt somit einen zentralen Bestandteil des Systems dar. Um Zugriffe auf die verschiedenen Speicherbereiche zu organisieren, greift die emulierte CPU nicht direkt auf diese zu. Stattdessen aggregiert die Klasse `MemoryBus` alle Komponenten, die über adressierbaren Speicher verfügen, sodass die CPU nur zwei Methoden für lesende und schreibende Speicherzugriffe benötigt, die von `MemoryBus` angeboten werden. Das folgende Klassendiagramm veranschaulicht die Zusammensetzung einiger Speicherkomponenten, die der Emulator implementiert:

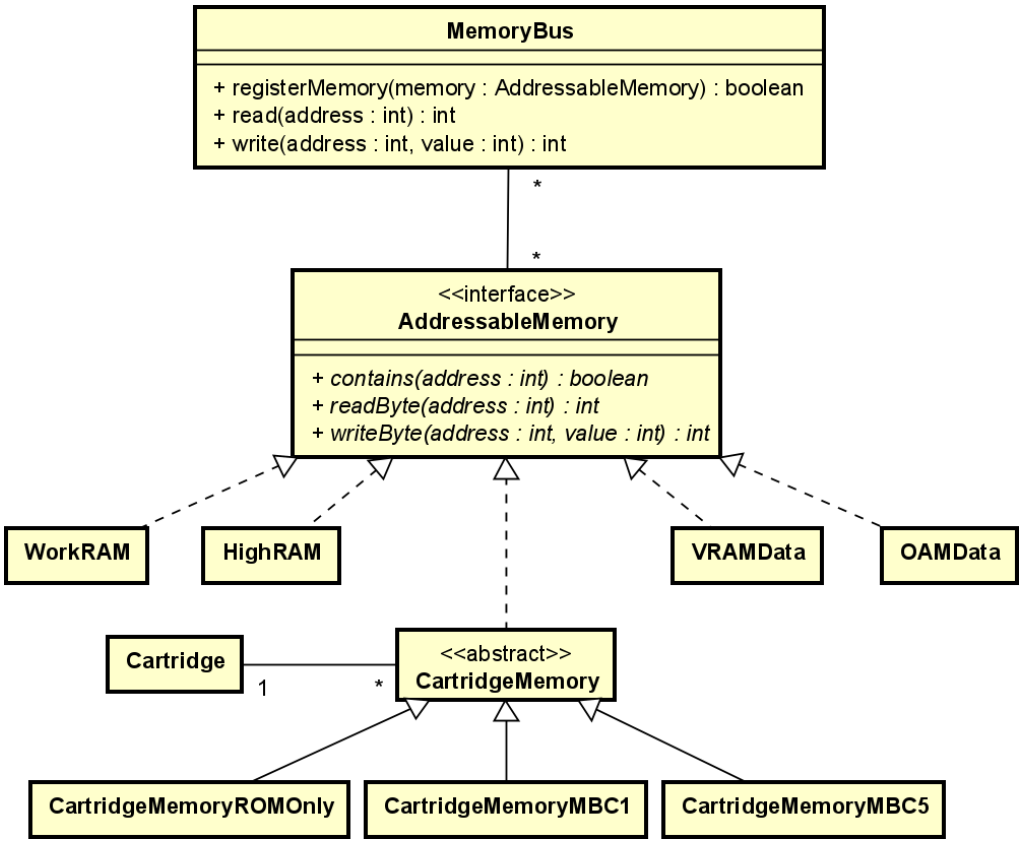

Abbildung 11 Vereinfachtes UML-Klassendiagramm der Speicherkomponenten.

Alle Komponenten, die über adressierbaren Speicher verfügen, implementieren das Interface `AddressableMemory`, welches die minimale Implementierung von lesenden (`readByte`) und schreibenden (`writeByte`) Speicherzugriffen vorschreibt. Die `contains`-Methode gibt an, ob eine gegebene Adresse die Speicherkomponente adressiert, und wird von `MemoryBus` genutzt, um den passenden Speicher einer Adresse zu lokalisieren, auf den über die `read`- und `write`-Methoden zugegriffen werden soll. Es werden nur 8-Bit-Speicherzugriffe unterstützt, sodass lediglich Werte zwischen 0 und 255 gelesen oder geschrieben werden können. Schreibende Zugriffe auf nicht-implementierte Adressen werden ignoriert; lesende Zugriffe geben stets den Wert `0xFF` zurück.

Der Emulator verwendet Spezialisierungen von `CartridgeMemory`, um MBCs zu realisieren. Die passende Implementation für das Spiel, das emuliert werden soll, wird anhand seiner Header-Daten bestimmt. Derzeit unterstützt der Emulator Spiele, die keinen MBC, MBC1, oder MBC5 benötigen. Weiterhin werden persistente Spielstände,

die ebenfalls über MBCs gesteuert werden, unterstützt, sodass der Spielfortschritt kompatibler Spiele auch abgespeichert wird.

Das Klassendiagramm auf der vorherigen Seite stellt lediglich eine Übersicht über größere Speicherkomponenten dar. Die nachstehende Tabelle enthält alle Speicheradressen, die bisher realisiert wurden. Hierdurch wird auch ersichtlich, dass einige Speicheradressen noch nicht implementiert sind, wie etwa die Register SB (`$FF01`) und SC (`$FF02`), die zur seriellen Datenübertragung genutzt werden.

Adressen	Beschreibung	Implementierung
`[$0000, $7FFF]`	ROM	`CartridgeMemory`
`[$8000, $9FFF]`	VRAM	`VRAMData`
`[$A000, $BFFF]`	EXRAM	`CartridgeMemory`
`[$C000, $DFFF]`	WRAM	`WorkRAM`
`[$E000, $FDFF]`	„Echo“ RAM	`WorkRAM`
`[$FE00, $FE9F]`	OAM	`OAMData`
`[$FF80, $FFFE]`	HRAM	`HighRAM`
`$FF00`	JOYP	`Joypad`
`$FF04`	DIV	`Timer`
`$FF05`	TIMA	`Timer`
`$FF06`	TMA	`Timer`
`$FF07`	TAC	`Timer`
`$FF0F`	IF	`InterruptMgr`
`$FF40`	LCDC	`LCDRenderer`
`$FF41`	STAT	`LCDRenderer`
`$FF42`	SCY	`LCDRenderer`
`$FF43`	SCX	`LCDRenderer`
`$FF44`	LY	`LCDRenderer`
`$FF45`	LYC	`LCDRenderer`
`$FF46`	DMA	`OAMData`
`$FF47`	BGP	`LCDRenderer`
`$FF48`	OBP0	`LCDRenderer`
`$FF49`	OBP1	`LCDRenderer`
`$FF4A`	WY	`LCDRenderer`
`$FF4B`	WX	`LCDRenderer`
`$FFFF`	IE	`InterruptMgr`

3.5 CPU-Komponente

Die Implementierung der CPU-Komponente stellte sich als sehr komplexe und zeitintensive Aufgabe heraus. Fortan wird die CPU-Komponente des Emulators auch vereinfacht als Prozessor bezeichnet. Im Kern ist die Aufgabe des Prozessors die Ausführung von Maschinenbefehlen eines Spiels sowie die Verarbeitung von ausgelösten Interrupts. Das folgende Klassendiagramm enthält die wesentlichen Elemente des Prozessors, die in diesem Abschnitt erläutert werden:

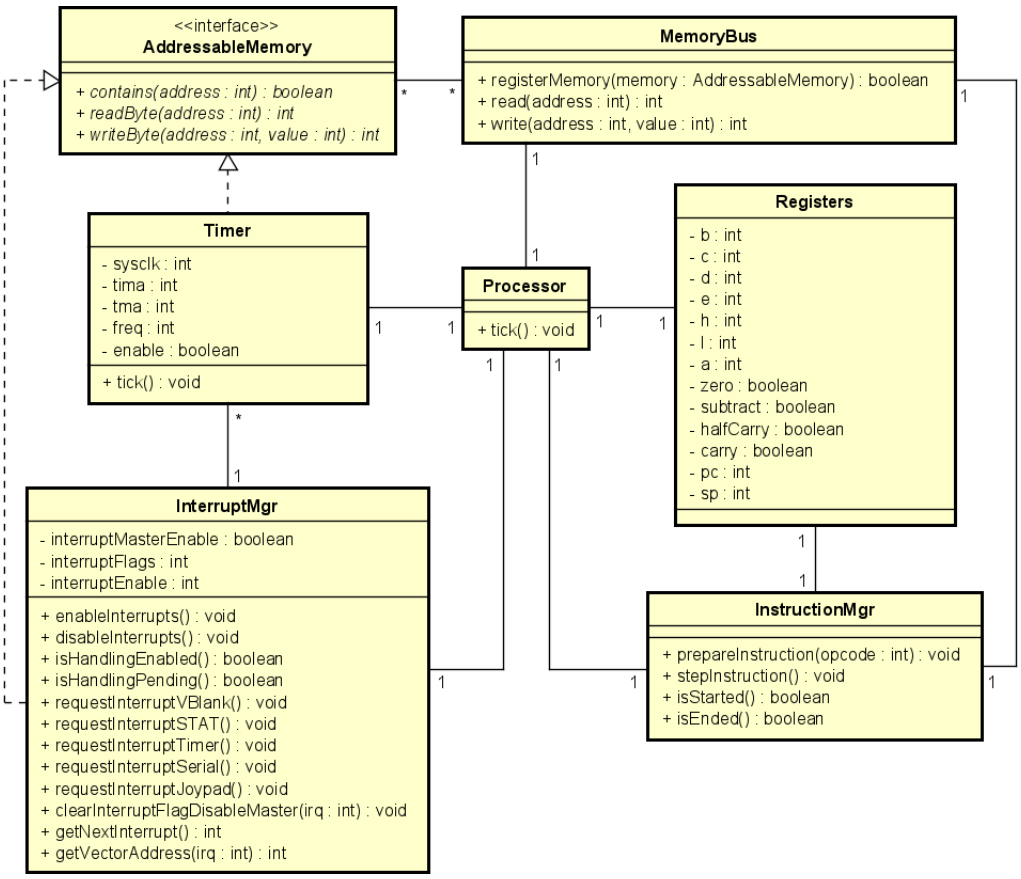

Abbildung 12 Vereinfachtes UML-Klassendiagramm der CPU-Komponente.

Um die Struktur des Prozessors aufzulockern, wurden die funktionalen Bestandteile in eigene Klassen ausgelagert. Dies ermöglicht zudem eine einfache Implementierung der I/O-Register der Teilkomponenten, indem deren Klassen `AddressableMemory` implementieren, um diese Register zu adressieren.

Alle Register, die als Operanden von Maschinenbefehlen genutzt werden, sind Teil der Klasse `Registers`. Weiterhin bietet diese Klasse Getter- und Setter-Methoden für alle 8-Bit- (B, C, D, E, H, L, A) sowie alle 16-Bit-Register (PC, SP, BC, DE, HL, AF) an. Alle Register werden dabei vereinfacht ohne Vorzeichen behandelt. Dieses Verhalten wird durch die Getter- und Setter erzwungen, indem überflüssige Bits entsprechend auf 0 gesetzt werden.

Ein signifikanter Teil der Entwicklung beanspruchte die Implementierung der Klasse `InstructionMgr`, welche die Funktionen aller 500 Maschinenbefehle realisiert. Um eine akkuratere Emulation der CPU zu ermöglichen, wird eine individuelle Instruktion nicht vollständig ausgeführt, da die Abarbeitungsdauer aufgrund verschiedenartiger Bearbeitungsschritte variiert. Um diese Eigenschaft genau abzubilden, werden Maschinenbefehle in ihre einzelnen Zwischenschritte zerlegt, wobei immer nur ein

Schritt pro Maschinenzyklus ausgeführt wird. Der folgende Codeausschnitt zeigt exemplarisch die implementierte Zerlegung der Instruktion „`LD (HL), u8`“ (Opcode `0x36`), wie sie etwa in (9) vorgestellt wird:

```
newInstruction(0x36, 2, 12, "ld (hl), $%02X", InstrArgType.D8,
        () -> {},
        () -> varLo = readAtPCInc(),
        () -> writeAtHL(varLo));
```

Abbildung 13 Zerlegung einer Instruktion in Bearbeitungsschritte.

Diese Instruktion dauert zwölf Maschinentakte, sodass drei Bearbeitungsschritte benötigt werden. Die Funktionsweisen vieler Instruktionen ähneln sich, sodass häufige Operationen als eigene Funktionen aufgefasst wurden, die wiederum von den entsprechenden Instruktionen genutzt werden. Der nachstehende Codeausschnitt implementiert beispielsweise die Funktionalität aller INC-Instruktionen:

```
public static int inc(Registers r, int v) {
    int rB = v + 1 & 0xFF;

    r.setZero(rB == 0);
    r.setSubtract(false);
    r.setHalfCarry((v & 0x0F) == 0x0F);

    return rB;
}
```

Abbildung 14 Implementierte Funktionalität der INC-Instruktionen.

`InstructionMgr` nutzt eine `Registers`-Instanz sowie eine `MemoryBus`-Instanz, um jeweils auf CPU-Register und adressierbare Speicher zuzugreifen. Über die Methode `prepareInstruction` kann die nächste auszuführende Instruktion anhand ihres Opcodes festgelegt werden, wodurch die entsprechenden Zerlegungsschritte geladen werden. Beim Aufruf der Methode `stepInstruction` wird immer nur ein Bearbeitungsschritt einer geladenen Instruktion ausgeführt. Um zu überprüfen, ob die Ausführung einer Instruktion begonnen hat oder abgeschlossen wurde, können jeweils die Methoden `isStarted` und `isEnded` genutzt werden.

Die Klasse `InterruptMgr` steuert die Verarbeitung von Interrupts, indem sie die Funktionen der Register IME, IE und IF erfüllt. Die Implementierung des Interface `AddressableMemory` ermöglicht die Adressierung von IE und IF. IME kann mit den Methoden `enableInterrupts` und `disableInterrupts` ein- beziehungsweise ausgeschaltet werden. Für alle fünf Interrupt-Arten existieren `requestInterrupt`-Methoden, um das entsprechende IF-Bit auf 1 zu setzen, um so ein bestimmtes Interrupt anzufragen. Über die Methode `getNextInterrupt` kann das Interrupt mit der höchsten Priorität erfragt werden, welches sowohl eingeschaltet als auch angefragt wurde. Die ISR ist jedoch Teil der Klasse `Processor` und wird auf der nächsten Seite genauer erläutert. Somit speichert der `InterruptMgr` lediglich die Zustände der Interrupt-Register und bietet entsprechende Methoden an, um diese geregelt zu steuern.

Der CPU-Timer wird über die gleichnamige Klasse `Timer` realisiert, und speichert die Werte der Register SYSCLK, TIMA, TMA und TAC. Das Register DIV wird implizit repräsentiert, da es stets aus den oberen acht Bits von SYSCLK besteht. Um die Register DIV, TIMA, TMA und TAC zu adressieren, implementiert diese Klasse das Interface `AdressableMemory`. Beim Aufrufen der Methode `tick` wird SYSCLK

inkrementiert und anschließend überprüft, ob der durch TIMA, TMA und TAC spezifizierte Timer ausgelaufen ist. Tritt dieser Fall ein, wird ein TIMER-Interrupt über den beteiligten `InterruptMgr` angefragt.

Die gesamte Funktionalität des Prozessors wird über die Klasse `Processor` gesteuert, welche die zuvor vorgestellten CPU-Komponenten erzeugt und nutzt. Die `Processor`-Methode `tick` emuliert genau einen Maschinentakt. Hierbei wird stets die `tick`-Methode von `Timer` aufgerufen, um SYSCLK zu erhöhen. Nach jedem vierten Takt wird jedoch ein vollständiger Maschinenzyklus emuliert, indem beispielsweise ein Bearbeitungsschritt einer Instruktion ausgeführt wird. Diese Methode arbeitet zustandsbasiert, um die unterschiedlichen Verhaltensmuster der CPU abzudecken. Das nachstehende vereinfachte UML-Zustandsdiagramm stellt die Zustände des Prozessors sowie deren Transitionen dar:

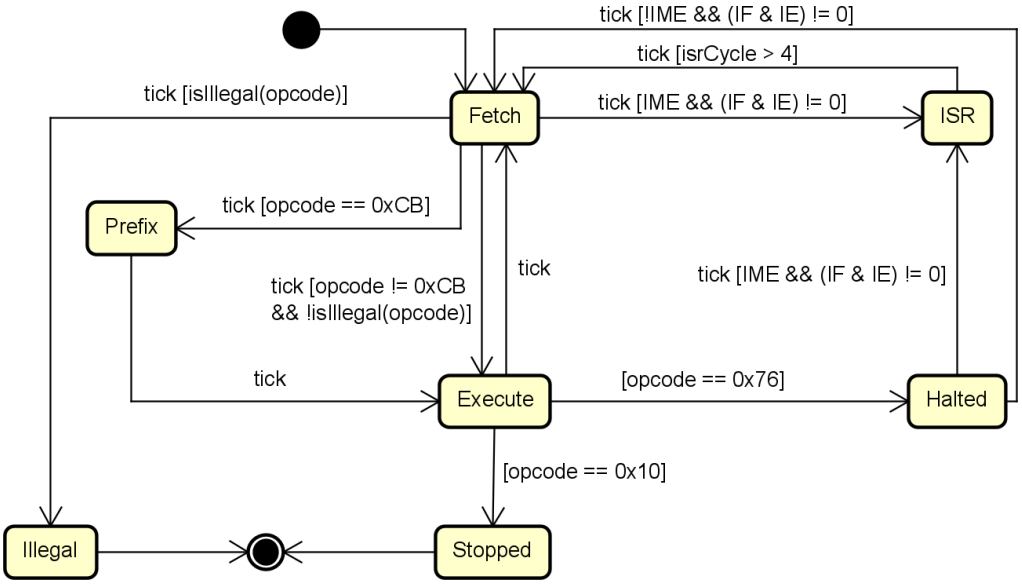

Abbildung 15 Vereinfachtes UML-Zustandsdiagramm des Prozessors. Transitionen mit dem Trigger „tick" finden jedoch nur nach jedem vierten Aufruf dieser Methode statt.

- **Fetch**: Falls ein Interrupt verarbeitet werden soll, also genau dann, wenn IME eingeschaltet ist und es ein Bit gibt, dass sowohl in IF als auch in IE gesetzt ist, wird direkt in den Zustand `ISR` gewechselt. Ansonsten liest der Prozessor das nächste Opcode-Byte aus dem Speicher heraus. Falls das Byte `0xCB` gelesen wurde, geht der Prozessor in den Zustand `Prefix` über. Falls es sich um einen undefinierten Opcode handelt, folgt wiederum der Zustand `Illegal`. Ansonsten wird anhand des Opcodes die nächste auszuführende Instruktion in den `InstructionMgr` geladen und der Zustand `Execute` ausgewählt.
- **Prefix**: Der Prozessor liest ein weiteres Opcode-Byte aus dem Speicher und konstruiert damit einen 16-Bit-Opcode. Anschließend wird die nächste auszuführende Instruktion geladen und es folgt ein Übergang in `Execute`.
- **Execute**: `InstructionMgr` führt die aktuelle Instruktion Schritt für Schritt aus. Nachdem alle Schritte einer Instruktion durchlaufen wurden, wird in den nächsten Zustand übergangen, wobei dieser von der ausgeführten Instruktion abhängt. Die STOP-Instruktion (Opcode `0x10`) führt zu einer Transition in `Stopped`. HALT (Opcode `0x76`) führt den Prozessor in den Zustand `Halted` über. In allen anderen Fällen beginnt der Prozessor wieder in `Fetch`.
- **Illegal**: Der Prozessor stoppt, da eine unbekannter Opcode gelesen wurde.
- **Stopped**: Der Prozessor stoppt vollkommen.

- **Halted**: Das Auslesen weiterer Instruktionen wird zunächst ausgesetzt. Falls ein weiteres Interrupt verarbeitet werden kann, geht der Prozessor in den Zustand `ISR` über. Falls `IME` deaktiviert ist, obwohl ein Interrupt eingeschaltet und angefragt wurde, geht der Prozessor wiederum in den Zustand `Fetch` über. Ansonsten verbleibt der Prozessor im Haltezustand.
- **ISR**: In diesem Zustand verarbeitet der Prozessor das Interrupt mit der höchsten Priorität, indem der aktuelle Programmzähler auf dem Stack-Speicher abgelegt wird. Danach wird die Adresse der Ablaufroutine des Interrupts im Programmzähler gespeichert. Infolgedessen wird das entsprechende IF-Bit genullt sowie IME deaktiviert und der Prozessor wechselt in den Zustand `Fetch`, sodass die Ablaufroutine des Interrupts gelesen und ausgeführt wird.

3.6 PPU-Komponente

Die Klasse `LCDRenderer` emuliert den Rendering-Prozess der PPU des Game Boys. Im folgenden Klassendiagramm sind die wesentlichen Elemente der PPU-Komponente des Emulators aufgelistet, welche in diesem Kapitel genauer vorgestellt werden:

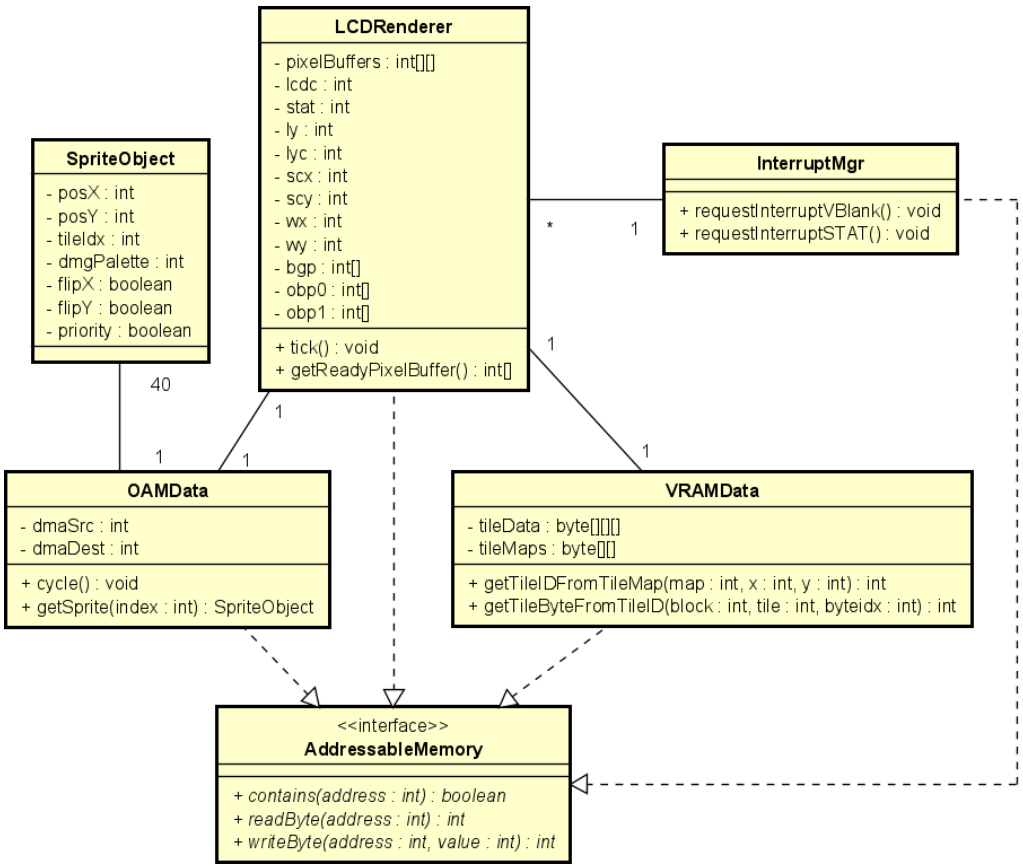

Abbildung 16 Vereinfachtes UML-Klassendiagramm der PPU-Komponente.

`VRAMData` implementiert den VRAM-Speicher des Game Boys und speichert die Daten für 384 Tiles sowie zwei 32×32 Tilemaps. Damit der Prozessor diesen Speicher beschreiben und auslesen kann, implementiert diese Klasse `AdressableMemory`. Weiterhin existieren zwei Hilfsmethoden zum Auslesen von Bilddaten aus VRAM, die von `LCDRenderer` während des Rendering-Prozesses genutzt werden:

- `getTileIDFromTileMap(map: int, x: int, y: int)`: Gibt den Index des Tiles zurück, das sich an den gegebenen Koordinaten (`x` und `y`) innerhalb der Tilemap mit der Nummer `map` befindet.
- `getTileByteFromTileID(block: int, tile: int, byteidx: int)`: Liest das Zeilenbyte mit dem Index `byteidx` eines gegebenen Tiles mit der Nummer `tile` innerhalb des Blocks `block`.

`OAMData` speichert die Daten für 40 Sprite-Objekte und ermöglicht dem Prozessor Speicherzugriffe durch die Realisierung von `AddressableMemory`. Die Methode `getSprite` wird wiederum von `LCDRenderer` genutzt, um die Attribute eines bestimmten Sprite-Objekts direkt auszulesen. Der DMA-Mechanismus zur effizienten Übertragung von Daten nach OAM wird ebenfalls von `OAMData` verwirklicht. Sobald das DMA-Register beschrieben wird, startet der DMA-Transfer im nächsten Maschinenzyklus. Pro Zyklus wird genau ein Byte kopiert, wofür die Methode `cycle` genutzt wird.

Die Klasse `LCDRenderer` bildet den Kern der emulierten PPU-Komponente und implementiert den Rendering-Prozess sowie die Funktionsweise der beteiligten I/O-Register (LCDC, STAT, LY, LYC, SCX, SCY, WX, WY, BGP, OBP0 und OBP1). Einzelne Pixel werden zunächst in Bildpuffer gespeichert. Um störende Artefakte wie Flickering zu vermeiden, wird Doppelpufferung eingesetzt, sodass ein Bildpuffer entweder nur gelesen oder beschrieben wird, aber nie gleichzeitig. Sobald ein vollständiges Frame gezeichnet wurde, werden die Puffer ausgetauscht. Die Methode `getReadyPixelBuffer` gibt dabei stets den Puffer zurück, in dem ein vollständiges Frame gezeichnet wurde. Der implementierte UI-Controller liest die Daten dieses Puffers aus, um das enthaltene Frame darzustellen. Die Methode `tick` emuliert grob die Funktionsweise der Zustände des Rendering-Prozesses, um ein Frame zeilenweise von oben nach unten sowie von links nach rechts zu zeichnen. Zu Beginn befindet sich `LCDRenderer` in Mode 2 (OAM-Scan) der ersten Scanline (LY=0).

- **Mode 2, OAM-Scan, 80 Takte.** Entgegen der Funktionsweise des Game Boys arbeitet die emulierte PPU in diesem Zustand nicht kleinschrittig, sondern wählt zu Beginn dieser Phase direkt bis zu zehn Sprite-Objekte aus, die sich auf der aktuellen Scanline befinden. Die Daten dieser Objekte werden anschließend für die nächste Phase zwischengespeichert. In den restlichen 79 Takten passiert dagegen nichts. Jedoch wird diese Wartezeit benötigt, um die Taktung der PPU einzuhalten. Danach geht `LCDRenderer` in Mode 3 über.
- **Mode 3, Drawing Pixels, 172 Takte.** Die Dauer von Mode 3 eines Game Boys ist variabel und wird von verschiedenen Faktoren beeinflusst. Die emulierte PPU arbeitet jedoch nicht entsprechend kleinschrittig und präzise, sondern beansprucht in dieser Phase immer 172 Takte, was der minimalen Dauer entspricht. Im ersten Takt werden alle 160 Pixel der Zeile in den beschreibbaren Framepuffer gerendert. Die verbleibenden 171 Takte verzögern den Übergang in Mode 0.
- **Mode 0, H-Blank, 204 Takte**: Da die Emulation von Mode 3 die minimale Menge an Takten beansprucht, gilt für H-Blank wiederum die maximale Dauer von 204 Takten. Innerhalb dieser Phase passiert nichts. Abschließend wird LY inkrementiert und es findet ein Zustandswechsel statt. Falls zuvor die letzte sichtbare Zeile gezeichnet wurde (LY=143), werden die Bildpuffer getauscht und es erfolgt ein Übergang in Mode 1 (V-Blank). Ansonsten geht `LCDRenderer` wieder in Mode 2 über.

- **Mode 1, V-Blank, 456 Takte**: Zum Schluss dieser Phase ändert sich erneut der Zustand. Falls die letzte Scanline eines Frames prozessiert wurde (LY=153), wird LY auf 0 gesetzt und `LCDRenderer` startet wieder in Mode 2. Ansonsten wird LY erhöht und die emulierte PPU verbleibt in Mode 1.

Die aktuelle Implementierung der vier Modi berücksichtigt jedoch nicht die Blockierung von Speicherzugriffen auf VRAM und OAM, sodass der Prozessor uneingeschränkten Zugriff auf diese Speicher hat.

Zudem unterstützt `LCDRenderer` das Auslösen von VBLANK-Interrupts sowie alle möglichen Konfigurationen des STAT-Interrupts. Sobald die emulierte PPU beispielsweise in Mode 1 wechselt, wird ein VBLANK-Interrupt angefragt sowie optional ein STAT-Interrupt, wenn das entsprechende STAT-Bit gesetzt ist.

Der folgende Programmcode des Emulators zeigt die allgemeine Funktionsweise des Renderings einer einzelnen Scanline und implementiert somit die Logik von Mode 3:

```
private void renderLinePixels() {
    int lcdY = ly;
    int windowStartX = wx - 7;
    int windowStartY = wy;

    for (int lcdX = 0; lcdX < LCD_WIDTH; lcdX++) {
        if (lcdcEnableBackground) {
            // Calculate pixel's coordinates on tilemap
            int mapX = (lcdX + scx) % 256;
            int mapY = (lcdY + scy) % 256;

            drawPixelsLineTiles(backgroundTileMapNum, lcdX, lcdY, mapX, mapY);
        }

        if (lcdcEnableWindow && lcdX >= windowStartX && lcdY >= windowStartY) {
            // Calculate pixel's coordinates on tilemap
            int mapX = lcdX - windowStartX;
            int mapY = windowLine;

            drawPixelsLineTiles(windowTileMapNum, lcdX, lcdY, mapX, mapY);
        }
    }

    if (lcdcEnableSprites) {
        drawPixelsLineSprites();
    }
}
```

Abbildung 17 Hauptfunktion zum Rendern der aktuellen Scanline.

Die Methode `renderLinePixels` zeichnet zunächst alle Pixel der Background- und Window-Ebenen, die auf der aktuellen Scanline liegen. Hierfür wird die Hilfsfunktion `drawPixelsLineTiles` verwendet, mit der ein Pixel einer Tilemap ausgelesen und in den beschreibbaren Bildpuffer gerendert werden kann. Hierfür wird die Nummer der Tilemap, welche die Tiles der entsprechenden Bildebene anordnet, die Position des Pixels innerhalb der Tilemap (`mapX` und `mapY`) sowie die Bildschirmposition (`lcdX` und `lcdY`) benötigt. Zum Rendern der Sprite-Objekte wird wiederum die Hilfsfunktion `drawPixelsLineSprites` verwendet, welche die zehn Sprites rendert, die zuvor in Mode 2 ausgelesen wurden.

3.7 Joypad

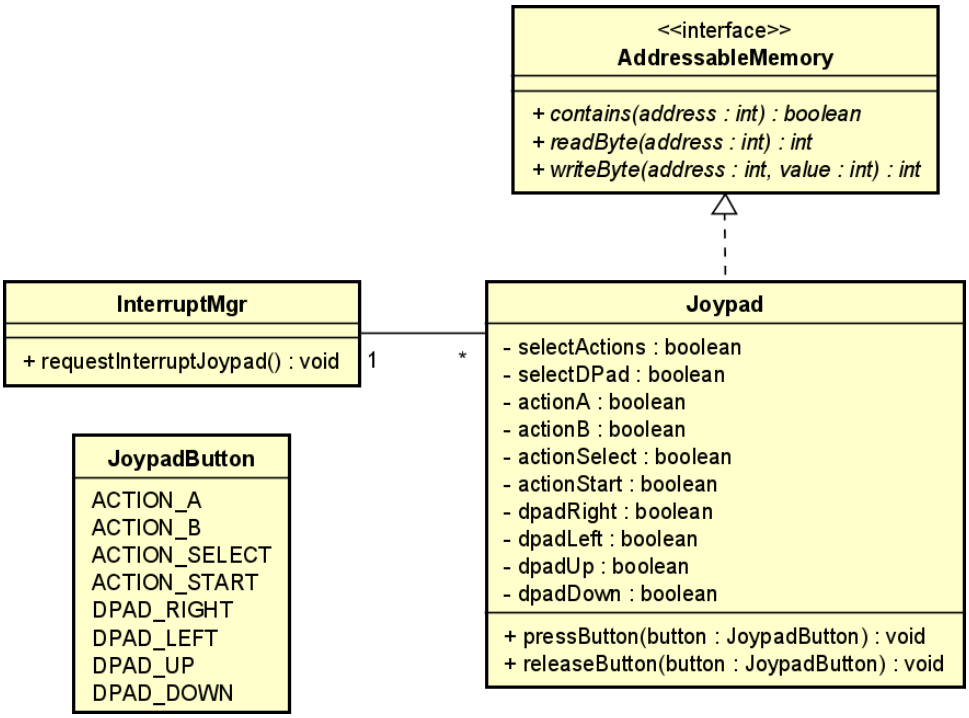

Abbildung 18 Vereinfachtes UML-Klassendiagramm der Joypad-Komponente.

Die `Joypad`-Komponente emuliert die Eingabeknöpfe des Game Boys. Das obige UML-Klassendiagramm veranschaulicht die Zusammensetzung von `Joypad` sowie die Beziehungen zu anderen Komponenten des Emulators, die an der Emulation der Game Boy-Steuerung beteiligt sind.

Für jeden der acht Eingabeknöpfe gibt jeweils ein Boolean-Wert an, ob der Knopf gedrückt ist (`true`) oder nicht (`false`). Die Enumeration `JoypadButton` sieht für jeden der acht Eingabeknöpfe genau einen Identifikator vor. Um die Manipulation von Eingabeknöpfen zu vereinfachen, bietet die Klasse `Joypad` die Methoden `pressButton` und `releaseButton`, mit denen der Zustand eines bestimmten Eingabeknopfs auf „gedrückt“ beziehungsweise auf „nicht gedrückt“ gesetzt werden kann. Der implementierte UI-Controller nutzt diese Methoden, um etwa Nutzereingaben über die Tastatur zu verarbeiten. Wird beispielsweise die Space-Taste gedrückt, ruft der UI-Controller die Methode `pressButton` mit dem Identifikator `ACTION_A` auf. Somit agiert die Space-Taste als A-Knopf des Game Boys.

Um die Funktionalität des JOYP-Registers zu realisieren, implementiert `Joypad` das Interface `AddressableMemory`, sodass der Prozessor die Zustände der Eingabeknöpfe auslesen kann. Zudem bestimmen Schreibzugriffe, welche Kategorie der Eingabeknöpfe ausgelesen werden. Die ausgewählte Kategorie wird über die Attribute `selectActions` und `selectDPad` repräsentiert. Falls der Zustand eines Eingabeknopfs auf „gedrückt“ wechselt, wird über den verknüpften `InterruptMgr` ein JOYPAD-Interrupt angefragt, sofern die entsprechende Kategorie selektiert wurde. Beim Drücken des A-Knopfs würde ein solches Interrupt nur angefragt werden, wenn auch `selectActions` wahr ist.

3.8 Bootsequenz

Das Bootstrap-Programm dient lediglich der Initialisierung der Hardware und der Validierung des Game Pak-Headers. Die visuelle Darstellung des Nintendo-Logos ist für die Ausführung des Spiels wiederum nicht notwendig. Das Verhalten des Bootstrap-Programms wird emuliert, indem zunächst die Inhalte des Game Pak-Headers validiert werden. Falls das ROM-Abbild des Spiels ungültig ist, wird die Emulation nicht gestartet. Andernfalls werden die entsprechenden Speicher und Register direkt mit den Werten initialisiert, die nach einem erfolgreichen Bootprozess erwartungsgemäß vorliegen würden. Die initialen Werte der Hardware-Register sind in (13) detailliert dokumentiert. Anschließend wird direkt die ROM-Datei des Spiels emuliert.

Eine akkurate Umsetzung der Bootsequenz würde ein ROM-Abbild des originalen Bootstrap-Programms erfordern. Da die ROM-Datei des Bootstrap-Programms urheberrechtlich geschützt ist, würde eine Notwendigkeit einer weiteren lizenzierten ROM-Datei die Benutzbarkeit einschränken.

3.9 Nutzung

Um ein Game Boy-Spiel auf dem Emulator spielen zu können, muss ein digitales Abbild des Spiels in Form einer ROM-Datei vorliegen. Anschließend muss die Java-Anwendung über ein CLI mit dem folgenden Befehl gestartet werden:

```
java -jar BachelorBoy.jar GAME_BOY <ROM_PFAD>
```

Hierbei muss `<ROM_PFAD>` durch den Pfad der ROM-Datei ersetzt werden. Infolgedessen liest der Emulator diese Datei ein, prüft ihren Header auf Kompatibilität und Korrektheit und startet anschließend die Ausführung des Spiels. Der Wert `GAME_BOY` im obigen Befehl spezifiziert, dass die Funktionsweise des ersten Game Boy-Modells emuliert werden soll. Andere Werte werden bisher nicht unterstützt.

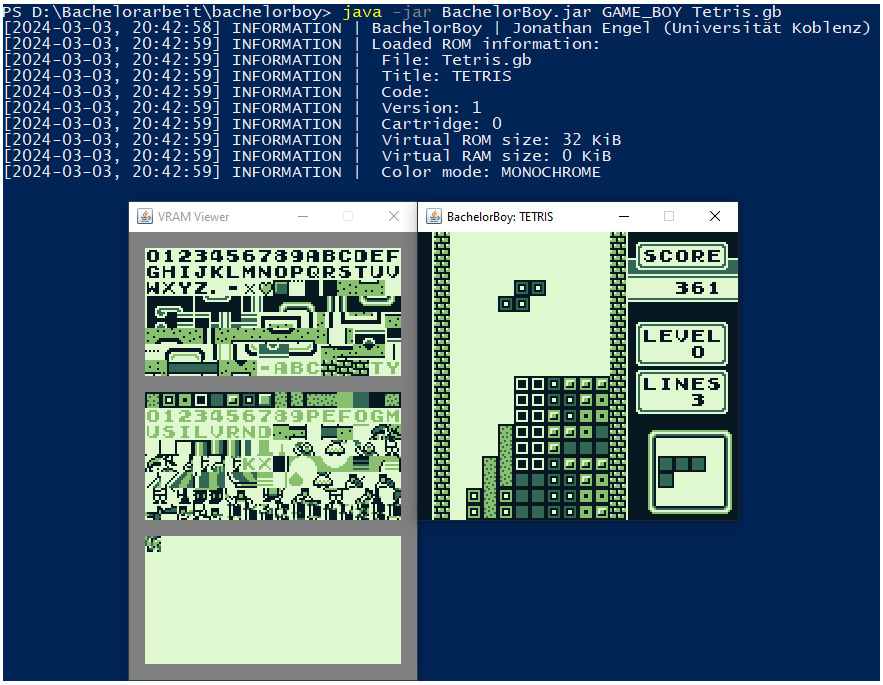

Abbildung 19 Screenshot der laufenden Anwendung samt emuliertem Spiel (Tetris, © Nintendo), VRAM-Debugger und Logging-Ausgabe.

Um das Spielgeschehen zu steuern, können die folgenden Tasten genutzt werden:

Game Boy-Knopf	Taste auf Tastatur
A	Space
B	B
Start	Enter
Select	Backspace
Steuerkreuz aufwärts	Pfeiltaste aufwärts
Steuerkreuz abwärts	Pfeiltaste abwärts
Steuerkreuz links	Pfeiltaste links
Steuerkreuz rechts	Pfeiltaste rechts

Weiterhin kann die Tabulator-Taste verwendet werden, um die Beschränkung der Emulationsgeschwindigkeit zu deaktivieren, sodass Spieler die Möglichkeit haben, lange Passagen des Spiels schneller ablaufen zu lassen.

4 Bewertung

In diesem Kapitel werden die Qualität des entwickelten Emulators sowie die Ergebnisse dieser Arbeit diskutiert.

4.1 Korrektheit

Der Mangel an offiziellen Informationen über die Hardware des Game Boys erschwert die Konzeption und Implementierung eines Emulators. Es gibt jedoch seit vielen Jahrzehnten eine Gruppe von Interessierten, die sich der Erforschung der Funktionsweise des Game Boys widmen. Eines der bekanntesten Dokumentationsprojekte ist „Pan Docs“, das im Jahr 1995 gestartet und seither von verschiedenen Autoren und Entwicklern mit Informationen angereichert wurde (24). Dieses Dokument enthält technische Informationen über die Funktionsweise aller Komponenten des Game Boy-Systems, wobei dieses Dokument primär für Entwickler von eigenen Game Boy-Spielen (Homebrew) gedacht ist. Das Dokument „Game Boy: Complete Technical Reference“ beinhaltet technische Details der Hardware (10). Dies umfasst beispielsweise detaillierte Beschreibungen aller Instruktionen, die zusätzlich mit Pseudocode veranschaulicht werden. Nach derzeitigem Stand fehlen Erklärungen und Details über den Aufbau wichtiger Komponenten, wie PPU und Joypad. Dieses Dokument enthält bisher lediglich Informationen zu den Registern dieser Bestandteile sowie zur APU und seriellen Schnittstelle. Folglich mussten unterschiedliche Dokumente zur Entwicklung des Emulators herangezogen werden, da sie sich inhaltlich ergänzen und bestätigen. Die Korrektheit des Emulators ist daher stark von gemeinschaftlichen Dokumentationsprojekten abhängig, da offizielle Informationen zur Hardware des Game Boys nicht öffentlich zugänglich sind und daher nicht berücksichtigt werden konnten.

Um die Funktionsweise der realen Game Boy-Hardware besser zu verstehen, entwickelten einige Autoren Test-ROMs, die die Funktionen dedizierter physischer Komponenten überprüfen und verifizieren sollten. Diese ROMs eignen sich daher auch zum Testen der Korrektheit von Emulatoren, da eine korrekte Emulation bestenfalls die gleichen Testergebnisse liefern sollte wie das entsprechende physische Gerät. Zur Verifizierung der emulierten Komponenten wurden daher einige solcher ROMs genutzt.

4.1.1 Korrektheit der CPU

Bereits während der Entwicklung der emulierten CPU hatte die Verifizierung ihrer Korrektheit eine hohe Priorität, da Fehler bei der Implementierung einzelner Instruktionen zu fehlerhaften Daten führen können, die auch auf andere beteiligte Komponenten des Emulators negativ wirken können. Würde beispielsweise aufgrund einer falschen Instruktion VRAM falsch beschrieben, produziert die emulierte PPU ein entsprechend fehlerhaftes Bild. Die Lokalisierung der Fehlerursache würde dadurch erschwert, da mehrere Komponenten beteiligt sind. Um die Funktionsweise des Prozessors auf Korrektheit zu prüfen, existieren Test-ROMs des Autors Blargg (25). Die ROMs `cpu_instrs`, `instr_timing`, `mem_timing` und `mem_timing-2` können zur Überprüfung von Instruktionen genutzt werden. Die Timer-Komponente sowie die Verarbeitung von Interrupts können durch die Tests `halt_bug` und `interrupt_timing` überprüft werden. Der prototypische Emulator besteht mit Ausnahme von `interrupt_timing` alle genannten Tests erfolgreich, sodass nach Blarggs Erwartungswerten die Emulation der CPU weitgehend korrekt ist. Lediglich das

zeitliche Verhalten von Interrupts ist aufgrund des nicht erfolgreichen Tests nicht korrekt implementiert.

4.1.2 Korrektheit der MBCs

Eine weitere bekannte Sammlung an Test-ROMs ist die „Mooneye Test Suite“, mit der die korrekte Emulation von CPU, PPU, MBCs und weiteren Komponenten überprüft werden kann (26). Da korrekte Speicherzugriffe auf Cartridge-Speicher ebenfalls für eine reibungslose Emulation notwendig sind, wurden die implementierten MBCs auf Korrektheit geprüft. Für alle MBCs, die der Emulator zum aktuellen Zeitpunkt unterstützt, bietet diese Sammlung ROMs an. Hierbei werden auch unterschiedliche ROM- und EXRAM-Volumen berücksichtigt. Die Emulator-Anwendung hat alle Tests erfolgreich absolviert, sodass die Implementierung von MBC1 und MBC5 den Testanforderungen von Javanainen entspricht.

4.1.3 Korrektheit der PPU

Zur Überprüfung des Renderings wurde die ROM `dmg-acid2` des Entwicklers Matt Currie verwendet. Durch das Ausführen dieser ROM soll ein Gesicht gerendert werden, wobei dieser Prozess von allen Grafikspeichern und PPU-Registern Gebrauch macht. Dadurch sollen alle Funktionen der PPU effektiv genutzt und getestet werden. Falls die emulierte PPU fehlerhaft arbeitet, wird das Gesicht falsch dargestellt, indem beispielsweise die Nase fehlt oder bestimmte Grafiken gespiegelt sind. Die möglichen Fehlerszenarien samt potenzieller Ursache wurden durch den Entwickler dokumentiert, sodass stets nachvollziehbar war, welche Elemente der PPU korrigiert werden mussten. Eine ausführliche Dokumentation dessen ist in (27) gegeben. Der prototypische Emulator rendert das erwartete Bild fehlerfrei, sodass die Bildsynthese auf Basis dieser Test-ROM korrekt ist:

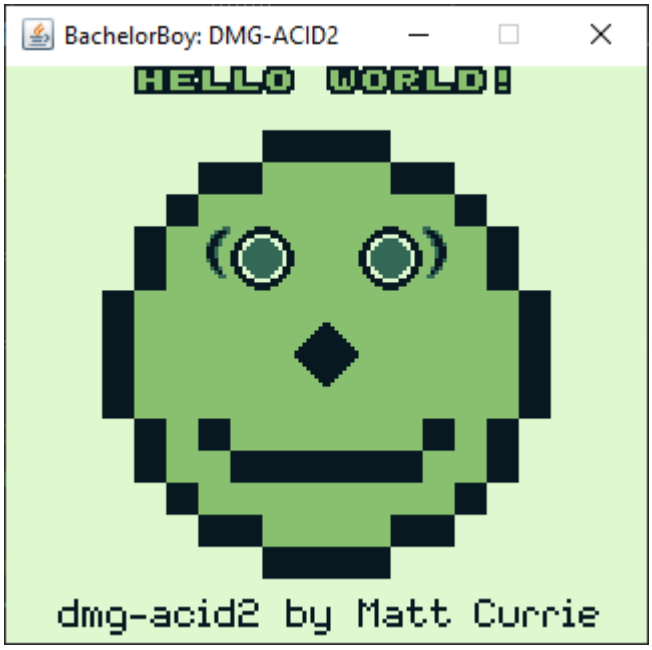

Abbildung 20 Korrektes Rendering von dmg-acid2.

Bei der Implementierung der PPU wurden jedoch einige Vereinfachungen der technischen Funktionsweise vorgenommen, da das taktgenaue Verhalten einzelner Modi sehr komplex ist. Beispielsweise hängt die Dauer von Mode 3 und Mode 0 von vielen Faktoren ab, wie etwa der Sprite-Anzahl auf einer Scanline. Die emulierte PPU rendert eine vollständige Scanline stets zu Beginn von Mode 3. Diese Implementierung verfälscht jedoch das Timing der PPU-Komponente, da einige STAT-Interrupts zu unerwarteten Zeitpunkten stattfinden können, was eventuell bei einigen Spielen, die eine strikte Taktung voraussetzen, zu Problemen führen kann.

4.2 Kompatibilität

Primär hängt die Kompatibilität eines Game Boy-Emulators zu einem Spiel von den implementierten MBC-Arten ab. Die Emulator-Anwendung, die im Rahmen dieser Arbeit entwickelt wurde, ist fähig, die Hardware von Cartridges ohne MBC, mit MBC1 sowie mit MBC5 zu emulieren. Daraus erschließt sich, dass der Emulator höchstens alle Spiele emulieren kann, die die genannten Ausprägungen erfordern. Umgekehrt gilt, dass Spiele, die keine der drei genannten Cartridge-Arten nutzen, nicht spielbar sind. Beispielsweise kann „Pokémon Goldene Edition“ nicht gespielt werden, da dieses Spiel MBC3 erfordert.

Die Kompatibilität kann jedoch nicht nur anhand der implementierten Cartridge-Hardware bewertet werden. Einige Spiele beinhalten Features, die zum derzeitigen Entwicklungszeitpunkt noch nicht nutzbar sind. Alle Elemente eines Spiels, die serielle Kommunikation per Link Cable erfordern, werden nicht unterstützt. Beispielsweise sind Multiplayer-Funktionen der Spiele „Tetris“ und „Pokémon Rote Edition“ daher nicht verfügbar. Da die entwickelte Anwendung lediglich die Hardware des ersten Game Boy-Modells emuliert, sind Elemente eines Spiels, die ausschließlich auf neueren Modellen nutzbar sind, ebenfalls nicht verfügbar. Für die Ausführbarkeit von „The Legend of Zelda: Oracle of Ages“ ist zwar die Emulation des MBC5 erforderlich, jedoch ist dieses Spiel nicht mit den Game Boy-Modellen kompatibel, die vor dem Game Boy Color erschienen sind.

Um die Kompatibilität des Emulators genauer zu bewerten, sollten Spiele individuell betrachtet und getestet werden, da eine fehlerhafte beziehungsweise nicht akkurate Emulation einen Einfluss auf die Funktionsweise eines Spiels haben kann. Im Rahmen dieser Arbeit war dies jedoch aufgrund der begrenzten Verfügbarkeit von Spielen nicht möglich. Daher wurden lediglich die folgenden Spiele für mindestens eine halbe Stunde getestet und problemlos gespielt:

- Tetris
- Super Mario Land
- Super Mario Land 2: 6 Golden Coins
- Pokémon Rote Edition
- Pokémon Blaue Edition
- Pokémon Gelbe Edition
- Kirby’s Dream Land
- The Legend of Zelda: Link’s Awakening DX

Des Weiteren wurde das Spiel „Kirby’s Dream Land 2“ getestet, jedoch kann der Emulator aufgrund eines bisher unbekannten Problems dieses Spiel nicht ausführen, sodass lediglich ein komplett weißer Bildschirm dargestellt wird.

4.3 Technologie

Der prototypische Emulator wurde in der Programmiersprache Java entwickelt, wodurch auch objektorientierte Konzepte zur Realisierung einiger Game Boy-Komponenten maßgeblich beigetragen haben. Dennoch birgt die Wahl der passenden Technologie weitere Herausforderungen.

Aufgrund der Vielzahl adressierbarer Speicher und I/O-Register eines Game Boy-Systems war die Reduktion der Komplexität der Speicherarchitektur eine wichtige Aufgabe, um im weiteren Entwicklungsverlauf eine reibungslose und strukturierte Implementierung der Speicherkomponenten zu ermöglichen. Diese Herausforderung konnte mit Hilfe des Generalisierungskonzepts bewältigt werden. Die aktuelle Implementierung sieht vor, die verschiedenen Speicher sowie I/O-Register auf dedizierte Klassen auszulagern, in denen sie benötigt werden. Ein alternativer, aber naiver Ansatz wäre die Realisierung eines riesigen Speicherpuffers, der 64 KiB des adressierbaren Game Boy-Speichers speichert und verwaltet. Dies würde jedoch die Wartbarkeit stark beeinträchtigen. Die Generalisierung ermöglichte zudem die Realisierung von emuliertem Datenaustausch zwischen Cartridge und Game Boy, der aufgrund unterschiedlicher MBC-Konstellationen variiert. Für jede MBC-Art ist eine Spezialisierung vorgesehen, um die jeweilige Funktionsweise zu realisieren.

Die Deklaration einiger Daten war jedoch umständlich, da Java keine vorzeichenlosen primitiven Datentypen unterstützt. Um ein vorzeichenloses Byte darzustellen, musste ein größerer Datentyp verwendet werden, der die Werte 0 bis 255 speichert. Um die vorgegebenen Bitgrenzen dennoch einzuhalten, wurden an vielen Stellen des Programmcodes Bitmasken verwendet, um den Wertebereich einzugrenzen. Die Getter- und Setter-Methoden der Klasse `Registers` gewährleisten beispielsweise die Einhaltung der für die entsprechenden Elemente vorgesehenen Bitgrenzen. Aufgrund der häufigen Verwendung solcher Bitmasken besteht auch ein erhöhtes Fehlerrisiko, dass beispielsweise Bitmasken fehlen oder sogar falsch sein können. In diesem Zusammenhang wäre eine Sprache empfehlenswert, die vorzeichenlose Datentypen unterstützt und die damit verbundenen Wertgrenzen forciert.

4.4 Entwicklungsziele

Die Ziele, die in Kapitel 1.3 festgelegt wurden, sind erreicht worden. Die prototypische Emulator-Anwendung ist in der Lage, viele Spiele gemäß den Einschränkungen aus Kapitel 4.2 zu laden und auszuführen. Das gerenderte Spielgeschehen kann über ein Fenster betrachtet werden. Zur Interaktion mit dem Spiel kann der Anwender eine Tastatur verwenden.

Es wurden keine der Erweiterungen implementiert, die ursprünglich als optional eingestuft wurden, da die Implementierung und das Testen der emulierten Komponenten sehr viel Zeit in Anspruch genommen haben. Die ursprünglich zur Implementierung von optionalen Erweiterungen vorgesehene Zeit wurde entsprechend zum Gestalten einer stabileren Anwendung genutzt, um die Spielbarkeit der kompatiblen Spiele zu erhöhen.

4.5 Ausblick

Die prototypische Emulator-Anwendung ist bereits mit einigen bekannten und beliebten Titeln kompatibel. Sie ist jedoch keineswegs perfekt, sodass immer noch ein großes Verbesserungspotenzial besteht.

4.5.1 Verifizierung der Korrektheit

Während dieses Projekts wurden die Test-ROMs nur individuell und in unregelmäßigen Abständen getestet. Diese Arbeit kann jedoch effizienter getätigt werden, indem etwa eine automatisierte Test-Pipeline kreiert wird, die alle Tests nach und nach automatisch ausführt und die produzierten Ergebniswerte der Tests mit den Erwartungswerten abgleicht. Die Mooneye-ROMs nutzen beispielsweise eine besondere Instruktion, um die Termination des Tests zu signalisieren, sodass beim Erreichen dieser Instruktion auch die Emulation der Test-ROM ausgeschaltet wird und die Ergebnisse ausgelesen werden können (26). Zudem sollten auch weitere Test-ROMs eingesetzt werden. Während der Entwicklung wurde für die Verifizierung der CPU nur die Test-ROMs von Blargg eingesetzt. Jedoch bietet Mooneye neben den verwendeten Tests für MBCs auch ausführliche Tests für CPU und PPU an, welche für die Verifizierung dieses Projekts noch nicht eingesetzt wurden (26).

Eine korrekte Emulation der Komponenten ist essenziell für die Gewährleistung der Spielbarkeit. Obwohl bereits einige Tests hierzu angewendet wurden, ist das jetzige Verfahren keineswegs empfehlenswert, sondern eher umständlich. Daher kann die Verifizierung der Korrektheit durch Automatisierung und Anwendung weiterer Tests effizienter und effektiver gestaltet werden.

4.5.2 Fehlende Komponenten

Da die Emulation der Audioausgabe bisher komplett fehlt, ist eine immersive Spielerfahrung, die durch Ton und Musik ermöglicht werden soll, derzeit nicht vorhanden. Einige Spiele, wie etwa Tetris, nutzen kultige 8-Bit-Musik, die bei der Anwendung des Emulators nicht abgespielt wird.

Einige Spiele nutzen auch Peripherie, die durch serielle Kommunikation per Link Cable realisiert wird. Neben Multiplayerfunktionen ermöglicht dies auch die Kommunikation mit besonderen Geräten, die mit einem realen Game Boy-System genutzt werden können. Beispielsweise ist „Pokémon Gelbe Edition“ mit dem Game Boy Printer kompatibel, mit dem Bilder des Spiels auf Papier gedruckt werden können.

Die Darstellung des Nintendo-Logos samt kurzem Jingle, die auf einem echten Game Boy-System beim Starten des Spiels geschieht, kann als ikonisches Merkmal dieser Konsole eingestuft werden, die für eine authentischere Nutzererfahrung sorgen kann. Dies setzt jedoch die zusätzliche Emulation einer Bootstrap-ROM voraus. Um die Benutzbarkeit jedoch nicht aufgrund von weiteren benötigten Dateien einzuschränken, könnte dieses Feature als Option implementiert werden, sodass Anwender mit einer eigenen Kopie des Bootstrap-Programms davon Gebrauch machen können.

Der Funktionsumfang des Emulators ist verglichen mit der realen physischen Hardware keineswegs vollständig. Um die Spielerfahrung weiter zu steigern, sollten auch die weiteren funktionalen Bestandteile des Game Boys implementiert werden.

4.5.3 Neuere Game Boy-Modelle

Die Emulation beschränkt sich bisher auf die Hardware des ersten Game Boy-Modells. Einige Spiele sind ausschließlich mit neueren Modellen kompatibel, sodass auch die Unterstützung dieser Modelle für die Kompatibilität relevant ist.

Andere Modelle, wie etwa der Game Boy Color, bieten zudem auch einen breiteren Funktionsumfang, der bisher nicht unterstützt wird. Beispielsweise ermöglicht der

Game Boy Color die Darstellung von bis zu 56 Farben gleichzeitig, wobei der originale Game Boy, und somit auch der Emulator, auf lediglich vier Graustufen beschränkt ist.

4.5.4 Kompatibilität durch MBCs

Um die Kompatibilität mit weiteren Spielen zu ermöglichen, sollten auch die verbleibenden MBC-Arten emuliert werden, sodass diese Spiele grundsätzlich gestartet werden können. Namentlich fehlen bisher MBC1M, MBC2, MBC3, MBC6, MBC7, MMM01, M161, HuC1 sowie HuC-3. Es gibt aber auch inoffizielle MBC-Arten, die etwa von unlizenzierten Spielen genutzt wurden.

4.5.5 Entwicklertools

Dedizierte Entwicklertools können für das Debugging von Emulation sowie eigenskreierten Homebrew-Spielen hilfreich sein. Eine Möglichkeit wäre eine Schnittstelle, über die die Zustände verschiedener Emulator-Komponenten präzise überwacht und gegebenenfalls manipuliert werden können. So können beispielsweise die aktuellen Werte einzelner Register und Speicher analysiert werden, um so potenzielle Probleme zu identifizieren, die durch eine fehlerhafte Programmlogik des Spiels verursacht werden. Zudem könnte auch eine Funktion zur schrittweisen Ausführung von Instruktionen positiv zur Nachvollziehbarkeit der Programmlogik eines Spiels beitragen.

4.5.6 Benutzerkomfort

Es besteht großes Potenzial, den Benutzerkomfort des Emulators zu verbessern, um Spielern eine angenehmere Anwendung zu ermöglichen.

Einige Spiele beinhalten keine Funktionen zum Speichern des Spielfortschritts, sodass Benutzer diese Spiele immer wieder von neu beginnen müssen, sobald das Spiel zu einem anderen Zeitpunkt wieder gestartet wird. Um diesem Problem entgegenzusteuern, können Savestates genutzt werden. Dabei wird der komplette Zustand des Emulators in einer Datei gespeichert, der dann zu einem anderen Zeitpunkt wieder geladen werden kann, sodass kein Spielfortschritt verloren geht. Dieses Feature ermöglicht es Nutzern aber auch vor einem schwierigen Abschnitt in einem Spiel zu speichern, sodass dieser problemlos bestritten werden kann.

Bisher kann das Spielgeschehen nur über eine festgelegte Auswahl an Tastaturtasten beeinflusst werden. Um Nutzern mit unterschiedlichen Steuerungsstilen hierbei mehr Freiheit einzuräumen, sollte die Tastenbelegung konfigurierbar sein. Weiterhin könnte auch die Kompatibilität mit anderen Eingabegeräten, wie etwa einem Gamepad, realisiert werden, die für Spieler eventuell intuitiver und angenehmer zu nutzen wären.

Die Emulator-Anwendung muss derzeit über ein CLI gestartet werden, was eventuell für einige Nutzer umständlich sein könnte. Die Benutzeroberfläche sollte übersichtlich gestaltet werden, indem beispielsweise eine Liste von Game Boy-ROMs, die sich in einem dedizierten Ordner befinden, im UI dargestellt wird, sodass ein Spiel per Doppelklick oder vergleichbaren Methoden gestartet werden kann.

5 Fazit

Im Rahmen dieser Arbeit konnte eine Emulator-Anwendung entwickelt werden, über die eine umfangreiche Palette an Game Boy-Titeln spielbar ist. Mittels dedizierten Testprogrammen konnte dabei gewährleistet werden, dass die Implementierung auf Basis des Wissens, das von Game Boy-Enthusiasten aufgebaut wurde, in großem Maße korrekt funktioniert. Aufgrund der vereinfachten Implementierung der PPU besteht jedoch weiterhin Potenzial zu Verbesserungen. Darüber hinaus gibt es zahlreiche Möglichkeiten, das Spielerlebnis und den Benutzerkomfort zu verbessern, etwa durch die Emulation weiterer Game Boy-Komponenten oder auch die Ergänzung nützlicher Funktionen, die über die eigentliche Funktionsweise des Game Boys hinausgehen.

Auch wenn es bereits seit einigen Jahren offizielle Angebote zum Spielen klassischer Game Boy-Spiele seitens Nintendo gibt, ist die Auswahl der Spiele unter den Marken Virtual Console sowie Nintendo Switch Online stark eingeschränkt, sodass sich Spieler nur mit einem festgelegten Spielekatalog begnügen können. Dagegen ermöglichen inoffizielle Emulatoren, eigene Kopien von Spielen jederzeit nach Belieben des Spielers auszuführen.

Game Boy-Emulatoren, mit denen die meisten Game Boy-Spiele problemlos gespielt werden können und die ein angenehmes Spielerlebnis ermöglichen, existieren bereits seit vielen Jahren. Von daher kann die berechtigte Frage gestellt werden, inwiefern die Entwicklung eines weiteren Emulators sinnvoll ist. Dieses Projekt ermöglichte es, Konzepte der technischen Informatik sowie der Softwareentwicklung zu vereinen und ein tieferes Verständnis in den jeweiligen Gebieten aufzubauen. Einige technische Konzepte, wie Register und Speicheradressen, konnten mittels höherer Programmierelemente anschaulich modelliert und realisiert werden. Da der Quellcode des Emulators sehr ausführlich dokumentiert und strukturiert ist, könnte dieser Emulator auch zu bildungstechnischen Zwecken genutzt werden, um etwa die Funktionsweise einer Hardware über höhere Technologien wie Java zu veranschaulichen. Es besteht aber auch die Hoffnung, dass diese Arbeit ein größeres Interesse und Verständnis für Emulation im Allgemeinen weckt, um so das kulturelle Gut Videospiele erhalten zu können.

Quellen

1. Retro-Sandi. Emulatoren für Retro-Videospiele: Was sind sie und wie funktionieren sie?; 2023 [Stand: 26.02.2024]. Verfügbar unter: https://retroreiz.de/emulatoren-fuer-retro-videospiele-was-sind-sie-und-wie-funktionieren-sie/.
2. IT-Service24. Emulation Definition & Begriffserklärung [Stand: 26.02.2024]. Verfügbar unter: https://www.it-service24.com/lexikon/e/emulation/.
3. Nintendo. Game Boy [Stand: 26.02.2024]. Verfügbar unter: https://www.nintendo.de/Hardware/Unternehmensgeschichte/Game-Boy/Game-Boy-627031.html.
4. Wikipedia. Game Boy; 2024 [Stand: 26.02.2024]. Verfügbar unter: https://de.wikipedia.org/wiki/Game_Boy.
5. Nintendo. Game Boy Color [Stand: 26.02.2024]. Verfügbar unter: https://www.nintendo.de/Hardware/Unternehmensgeschichte/Game-Boy-Color/Game-Boy-Color-627137.html.
6. Rodrigo Copetti. Game Boy / Color Architecture - A Practical Analysis 2019 [Stand: 28.01.2024]. Verfügbar unter: https://www.copetti.org/writings/consoles/game-boy/.
7. Antonio Nino Diaz. The Cycle-Accurate Game Boy Docs; 2014 [Stand: 30.01.2024]. Verfügbar unter: https://github.com/AntonioND/giibiiadvance/blob/master/docs/TCAGBD.pdf.
8. Antonio Nino Diaz. Pan Docs: CPU Registers and Flags; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/CPU_Registers_and_Flags.html.
9. izik1. gbops, an accurate opcode table for the Game Boy; 2018 [Stand: 17.01.2024]. Verfügbar unter: https://izik1.github.io/gbops/.
10. Joonas Javanainen. Game Boy: Complete Technical Reference; 2017 [Stand: 30.01.2024]. Verfügbar unter: https://gekkio.fi/files/gb-docs/gbctr.pdf.
11. Antonio Nino Diaz. Pan Docs: Interrupts; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/Interrupts.html.
12. Antonio Nino Diaz. Pan Docs: Memory Map; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/Memory_Map.html.
13. Antonio Nino Diaz. Pan Docs: Power-Up Sequence; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/Power_Up_Sequence.html.
14. Antonio Nino Diaz. Pan Docs: MBCs; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/MBCs.html.
15. Antonio Nino Diaz. Pan Docs: The Cartridge Header; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/The_Cartridge_Header.html.
16. Joonas Javanainen. Game Boy Cartridges [Stand: 07.02.2024]. Verfügbar unter: https://gbhwdb.gekkio.fi/cartridges/.
17. Antonio Nino Diaz. Pan Docs: Graphics; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/Graphics.html.

18. Antonio Nino Diaz. Pan Docs: OAM DMA Transfer; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/OAM_DMA_Transfer.html.
19. Antonio Nino Diaz. Pan Docs: Joypad Input; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/Joypad_Input.html.
20. BGB GameBoy Emulator [Stand: 21.03.2024]. Verfügbar unter: https://bgb.bircd.org/.
21. Joonas Javanainen. Mooneye GB [Stand: 21.03.2024]. Verfügbar unter: https://github.com/Gekkio/mooneye-gb.
22. Nintendo. Hinweis zur Beendigung der Kaufoption im Nintendo eShop für Wii U und Nintendo 3DS - Update März 2024 [Stand: 23.03.2024]. Verfügbar unter: https://www.nintendo.de/Support/Kauf/Download-Spielen/Nintendo-eShop/Hinweis-zur-Beendigung-der-Kaufoption-im-Nintendo-eShop-fur-Wii-U-und-Nintendo-3DS-Update-Marz-2024-2174073.html.
23. Nintendo. Game Boy - Nintendo Switch Online [Stand: 23.03.2024]. Verfügbar unter: https://www.nintendo.de/Spiele/Nintendo-Switch-Download-Software/Game-Boy-Nintendo-Switch-Online-2339836.html.
24. Antonio Nino Diaz. Pan Docs; 2024 [Stand: 30.01.2024]. Verfügbar unter: https://gbdev.io/pandocs/single.html.
25. Blargg. Blargg's Gameboy hardware test ROMs; 2013. Verfügbar unter: https://gbdev.gg8.se/files/roms/blargg-gb-tests/.
26. Joonas Javanainen. Mooneye Test Suite; 2021 [Stand: 20.03.2024]. Verfügbar unter: https://github.com/Gekkio/mooneye-test-suite.
27. Matt Currie. dmg-acid2; 2020 [Stand: 20.03.2024]. Verfügbar unter: https://github.com/mattcurrie/dmg-acid2.