



Fachhochschule Rosenheim

**Fachbereich Informatik
Diplomarbeit**

Diplomprüfung SS 2004

Marten Wobst

**Ein Framework für die Spielentwicklung auf dem
GameBoyAdvance**

Erstprüfer:
Zweitprüfer:

Prof. Dr. Franz Josef Schmitt
Prof. Dr. Theodor Tempelmeier

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, im Juli 2004

Marten Wobst

DATENBANKERFASSUNG

„FRAMEWORK / GAMEBOYADVANCE / ASSEMBLER /
ARM7TDMI / SPIELENENTWICKLUNG“

I N H A L T S V E R Z E I C H N I S

1	Einleitung	19
1.1	Kurzzusammenfassung.....	19
1.2	Übersicht.....	19
1.3	Zielsetzung	20
2	Der Hauptprozessor	21
2.1	Anforderungen.....	21
2.2	ARM7tdmi Überblick	21
2.3	Speicher und seine Anbindung.....	22
2.4	Registermodel.....	23
2.5	Die Codearten.....	24
2.6	Der Befehlsatz.....	26
2.6.1	Arithmetische Befehle	26
2.6.2	Logische Befehle	26
2.6.3	Datentransfer.....	27
2.6.4	Vergleichsfunktionen.....	27
2.6.5	Datentransfers (Speicherzugriffe).....	28
2.6.6	Sprungfunktionen	30
2.6.6.1	Unbedingte Sprünge.....	30
2.6.6.2	Bedingte Sprünge.....	31
2.6.7	Software-Interrupts.....	31
2.6.8	Multiplikation und Division	31
2.6.9	Pseudo-Befehle.....	32
2.7	Exceptions.....	33
2.8	Fazit	34
3	GBA Hardware	37
3.1	Technische Daten.....	37
3.1.1	Betriebsarten.....	37
3.1.2	Interne Speicher	37
3.1.3	Grafikdaten.....	37
3.1.4	Sound.....	38
3.1.5	Steuerung.....	38
3.1.6	Kommunikationsanschlüsse.....	38
3.1.7	Externer Speicher.....	38
3.1.8	Stromversorgung.....	38
3.2	Speichermap	39
3.3	Registerübersicht	39
3.4	Videokontroller.....	42
3.4.1	Grafikdarstellung.....	42
3.4.2	Grafikmodies	43
3.4.3	Sprites	44

3.5	Soundcontroller	45
3.6	Timer	45
3.7	DMA Transfer	45
3.8	Tastatureingabe	45
3.9	Interruptkontrolle	46
3.10	BIOS	48
3.11	Fazit	48
4	Entwicklerwerkzeuge	49
4.1	Software	49
4.1.1	Emulatoren	49
4.1.2	Entwicklungs-Tools	50
4.1.3	Editoren	53
4.1.4	Grafikprogramme	53
4.1.4.1	Mapkonverter	56
4.1.4.2	Binär zu Quelltext	56
4.1.4.3	Texte zu binär	57
4.1.5	Musik und Sound	58
4.2	Hardware	58
4.2.1.1	Flasher	58
4.2.2	Flashkarten	59
5	Standard Funktionen	61
5.1	Mathematische Funktionen	61
5.1.1	Division	61
5.1.2	Wurzel	61
5.1.3	Zusatz	62
5.1.4	Funktions- und Parameterbeschreibung	62
5.2	Allgemeine Funktionen	63
5.2.1	Speichertransfer-Funktionen	63
5.2.2	Quicksort	63
5.2.3	V-Blank	65
5.2.4	Debugausgabe	65
5.2.5	Funktions- und Parameterbeschreibung	65
5.2.6	Interrupt-Handler	68
5.2.6.1	Funktions- und Parameterbeschreibung	70
6	Musik und Soundprogrammierung	73
6.1	Allgemeine Einführung in die Problematik	73
6.2	Die GBA Sound-Hardware	73
6.3	Die GBA-Soundgeneratoren im Detail	74
6.3.1	Sound-Generator 1	74
6.3.2	Sound-Generator 2	75
6.3.3	Sound-Generator 3	76

6.3.4	Sound-Generator 4	76
6.4	Direkter Sound	77
6.5	Tracker und ihre Geschichte	79
6.6	Kanäle Mischen	80
6.7	Tracker im Detail.....	84
6.7.1	Noten-Aufbau	85
6.7.2	MOD-Daten-Struktur	86
6.7.3	Speed-Werte.....	86
6.7.4	Effekte	87
6.7.4.1	Normal Play oder Arpeggio	88
6.7.4.2	Beschleunigen.....	89
6.7.4.3	Abbremsen	89
6.7.4.4	Setze Sampleoffset.....	89
6.7.4.5	Volumeslide	89
6.7.4.6	Positionssprung.....	89
6.7.4.7	Setze Volume.....	89
6.7.4.8	Pattern Break.....	90
6.7.4.9	Setzte Geschwindigkeit.....	90
6.7.4.10	Feines Slide hoch.....	90
6.7.4.11	Feines Slide herunter.....	90
6.7.4.12	Setze Finetuning	90
6.7.4.13	Feines Volumeslide hoch	90
6.7.4.14	Feines Volumeslide herunter	90
6.8	Interner Ablauf	91
6.9	Konvertierung vom MOD-Dateien.....	92
6.10	Funktions- und Parameterbeschreibung.....	94
6.10.1	MOD-Player	94
6.10.2	Generatoren.....	95
6.10.3	Direkter Sound.....	98
6.11	Fazit.....	99
7	Animation von Bildern	101
7.1	Allgemeine Einführung in die Problematik.....	101
7.2	Anforderungen an den Algorithmus	101
7.3	Spezielle Problemanalyse der Anforderungen.....	101
7.4	Implementierung der Kodierungsalgorithmen.....	103
7.5	Funktions- und Parameterbeschreibung.....	108
7.6	Das Datenformat.....	109
7.7	Das Konvertertool	111
7.8	Anmerkung.....	113
7.9	Fazit	114
8	Kollisionserkennung.....	115
8.1	Allgemeine Einführung in die Problematik.....	115

8.2	Späherische Kollision.....	117
8.3	Polygon – Polygon Kollision.....	119
8.4	Punkt im Polygon.....	121
8.5	Standard Ablaufschema.....	125
8.6	Strukturbeschreibung.....	126
8.7	Funktions- und Parameterbeschreibung.....	126
8.7.1	Externe Funktionen.....	127
8.7.2	Interne Funktionen.....	127
9	Speichermanagement	131
9.1	Heap - Speichermanagement.....	131
9.1.1	Strukturaufbau.....	131
9.1.2	Funktions- und Parameterbeschreibung	132
9.2	Stack – Speichermanagement	133
9.2.1	Funktions- und Parameterbeschreibung	133
10	Grafikmanagement.....	135
10.1	Allgemeine Einführung in die Problematik.....	135
10.2	Allgemeine Grafikfunktionen.....	135
10.2.1	Linie	135
10.2.1.1	Funktions- und Parameterbeschreibung.....	136
10.2.2	Kreis.....	136
10.2.2.1	Funktions- und Parameterbeschreibung.....	136
10.2.3	Füllen	137
10.2.3.1	Funktions- und Parameterbeschreibung.....	137
10.2.4	Bézier-Kurve (3 Haltepunkte).....	138
10.2.4.1	Funktions- und Parameterbeschreibung.....	140
10.3	Spezielle Grafik-Funktionen.....	140
10.3.1	Spritmanager.....	140
10.3.1.1	Funktions- und Parameterbeschreibung.....	141
10.3.2	Scrollfunktionen.....	142
10.3.2.1	Funktions- und Parameterbeschreibung.....	145
10.3.3	Fontengine	146
10.3.3.1	Funktions- und Parameterbeschreibung.....	148
11	Testimplementierung eines Spiels	149
11.1	System-Start.....	149
11.2	Das Logo	150
11.3	Das Titelbild.....	150
11.4	Das Spiel.....	151
11.5	Fazit.....	156
12	Zusammenfassung und Ausblick.....	159

13	Anmerkungen	161
13.1	Danksagungen.....	161
14	Literaturverzeichnis	162

ABBILDUNGSVERZEICHNIS

Nummer	Seite
Abbildung 2-1 Speicher im Detail	23
Abbildung 2-2 Registersatz	23
Abbildung 2-3 CPSR	24
Abbildung 2-4 M-Bits	24
Abbildung 2-5 Conditions	28
Abbildung 2-6 ldm/stm-Erweiterungen	29
Abbildung 2-7 ldm/stm-Mnemonik	30
Abbildung 2-8 Pseudo-Befehle	32
Abbildung 2-9 "Byte zu Long" Pseudo-Befehle	33
Abbildung 2-10 IRQ-Modus mit Registersatz	34
Abbildung 2-11 IRQ-Vektoren	34
Abbildung 3-1 GameBoyAdvance©	38
Abbildung 3-2 GameBoyAdvance (von oben)	39
Abbildung 3-3 Registermap	42
Abbildung 3-4 Farbaufteilung	42
Abbildung 3-5 Tilemodus	43
Abbildung 3-6 Spritedimensionen	44
Abbildung 3-7 schnelle Adressbildung	46
Abbildung 3-8 BIOS IRQ-Behandlung	46
Abbildung 3-9 LCD	48
Abbildung 4-1 BatGBA	49
Abbildung 4-2 Cygwin-Shell	50
Abbildung 4-3 Standard-Makefile	51
Abbildung 4-4 Startup	52
Abbildung 4-5 Ultraedit mit geladener Quell-Datei	53
Abbildung 4-6 PPaint	54
Abbildung 4-7 Tilestudio mit geöffneter Map	55
Abbildung 4-8 Kaleid mit geladenen Bild	56
Abbildung 4-9 ProTracker 4.0b	58
Abbildung 4-10 Flascheinheit mit eingeschobener Flashkarte	59
Abbildung 4-11 64 MBit Flashkarte	59
Abbildung 5-1 QuickSort Beispiel Implementierung	65
Abbildung 5-2 Vektorobjekt	65
Abbildung 5-3 normaler IRQ	69
Abbildung 5-4 Multipler IRQ	69
Abbildung 5-5 IRQ-Handler Ablauf	70
Abbildung 6-1 Sweep Shifts	74

Abbildung 6-2 Frequenzänderung durch Schieben	75
Abbildung 6-3 Hüllkurven	75
Abbildung 6-4 7-Stufen-LSFR	76
Abbildung 6-5 Soundausgabe mit DMA	78
Abbildung 6-6 Soundausgabe mit der CPU	78
Abbildung 6-7 Sample mit Loop	81
Abbildung 6-8 Mischschleife (2 zu 1)	83
Abbildung 6-9 Mad-Tracker	84
Abbildung 6-10 Note mit Erweiterung	85
Abbildung 6-11 Noten-Kodierung	85
Abbildung 6-12 MOD-Struktur	86
Abbildung 6-13 Effekt-Implementierung	87
Abbildung 6-14 E-Effekt-Implementierung	88
Abbildung 6-15 MOD-Player (Ablauf)	91
Abbildung 6-16 Kanal-Kodierung	92
Abbildung 6-17 MOD-Konverter Testausgabe	93
Abbildung 6-18 MOD-Test	93
Abbildung 7-1 RLE-Kompression	104
Abbildung 7-2 RLE-Dekompression	105
Abbildung 7-3 RLE Dekompression mit Quelle-Ziel Test und DMA Transfer	106
Abbildung 7-4 Dekompression von LZSS Daten	107
Abbildung 7-5 Kompressions-Statistik	107
Abbildung 7-6 Datenformat	110
Abbildung 7-7 Beispielausgabe des Konverter-Tools	112
Abbildung 7-8 Datenformat	113
Abbildung 8-1 Pixelobjekte	115
Abbildung 8-2 Pixel-Kollision	115
Abbildung 8-3 Polygon-Kollision	116
Abbildung 8-4 Punkt in Polygon	117
Abbildung 8-5 Sphärische-Kollision	118
Abbildung 8-6 prüfe zwei Linien auf einen Schnittpunkt	120
Abbildung 8-7 Winkel-Addition	121
Abbildung 8-8 Quadranten-Test	122
Abbildung 8-9 Schnitt-Test	123
Abbildung 8-10 BlitzBasic Implementierung	124
Abbildung 8-11 Standard Ablauf einer Kollision-Überprüfung	125
Abbildung 8-12 Kollisions-Objekt-Struktur	126
Abbildung 9-1 Element-Aufbau	132
Abbildung 10-1 Funktionstest	135
Abbildung 10-2 Vier- oder Achtwegen-Test	137
Abbildung 10-3 zeilenweise Füllen	137
Abbildung 10-4 Bézier Testimplementierung	139
Abbildung 10-5 Mapaufbau	143
Abbildung 10-6 Overlay	144

<i>Abbildung 10-7 Mapbeispiel</i>	145
<i>Abbildung 10-8 Buchstaben</i>	146
<i>Abbildung 10-9 Beispielausgabe des Fontkonverters</i>	147
<i>Abbildung 10-10 Fonttest</i>	147
<i>Abbildung 11-1 Logo</i>	150
<i>Abbildung 11-2 Sprite-Aufteilung</i>	151
<i>Abbildung 11-3 Titelbild</i>	151
<i>Abbildung 11-4 Spielablauf</i>	152
<i>Abbildung 11-5 Grafik-Teile</i>	153
<i>Abbildung 11-6 Sprite-Animation</i>	153
<i>Abbildung 11-7 durch eine Kurve fahren</i>	154
<i>Abbildung 11-8 Weg-Marken</i>	155
<i>Abbildung 11-9 Höhen-Unterschiede</i>	155
<i>Abbildung 11-10 Kollisions- und Höhen-Änderung</i>	156

A B K Ü R Z U N G S V E R Z E I C H N I S

ALU	Arithmetic Logic Unit (Rechenwerk)
Amiga-IFF	Amiga - Interchange File Format (universelles Datei-Format)
ARM	(1.) Advanced Risc Machines (2.) 32-Bit Code-Modus
BG	Background
BIOS	Basic Input Output System
BMP	Bitmap (Bildformat)
CAD	Computer Aided Manufacturing
CPSR	Current Program Status Register
CPU	Central Processing Unit
DAC	Digital Analog Converter
DMA	Direct Memory Access
EEPROM	Electrically Erasable Programmable Read Only Memory
EW RAM	External Work Random Access Memory
FIFO	First-In First-Out
FIQ	Fast Interrupt Request
FPU	Floating Point Unit
GB	GameBoy
GBA	GameBoyAdvance
GBC	GameBoyColor
GPR	General Purpose Register
IO-RAM	Input/Output Random Access Memory (in Speicher gemappte Register)
IRQ	Interrupt Request
IWRAM	Internal Work Random Access Memory
LCD	Liquid Crystal Displays
LR	Link Register
LFSR	Lineare-Feedback Shift Register
LZ77	Lempel-Ziv 77
LZSS	Lempel-Ziv-Storer-Szymanski
M68000	Motorola 68000 CPU (intern 32 Bit)
MMU	Memory Managment Unit
MOD	Modul (Musikformat)
MP3	MPEG Layer 3
MPEG	Moving Picture Experts Group
OAM	Object Attribute Memory
OAM-RAM	OAM
OBJ	Objekt (Sprites)
PAL	Phase Alternate Line (TV/Video-Format)
PC	(1.) Programm Counter (2.) Personal Computer
PNG	Portable Netwok Grafic (Bildformat)
RAM	Random Access Memory

RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
RLE	Run-Length-Encoding
ROM	Read Only Memory
SID	Sound Interface Device (C64 Sound-Prozessor)
SIO	Seriell Input Output
SP	Stack Pointer
SRAM	Static Random Access Memory
TFT	Thin Film Transistor
UAP	Unterbrechungs Antwort Programm
VBA	Visual Boy Advance (GBA Emulator)
VLSI	Very Large Scale Integration
VRAM	Video Random Access Memory
XOR	exklusive or (exklusives Oder)

G L O S S A R / T E R M I N O L O G I E

Amiga	Amiga - Computer
BatGBA	GameBoyAdvance Emulator
DIVX	Video - Format
FLASH	wiederbeschreibbarer Speicher
GNU	GNU ist eine rekursive Abkürzung von "GNU's Not Unix"
HAM	Entwicklungskid
H-Blank	horizontaler - Blank (horizontaler Null-Punkt des Bildes)
Paletten-RAM	Farbdaten-Speicher
S3M	Scream-Module (Musik-Modul)
THUMB	16 Bit Code-Modus
V-Blank	vertikaler - Blank (vertikaler Null-Punkt des Bildes)
XM	Fastracker-Module (Musik-Modul)
Z80	8-Bit CPU von Zilog

1 Einleitung

Schon immer spielten Spiele eine große Rolle für den Menschen. Einige Spiele sind aus dem heutigen Leben kaum wegzudenken und werden grenzenübergreifend bestritten. Mit der Entwicklung der ersten Spielkonsolen, entstand ein völlig neues Medium für Spiele. Es war ohne großen Aufwand möglich ein Spiel zu spielen und es war in vielen Fällen kein menschlicher Gegner nötig. Durch die große Anzahl an Möglichkeiten, welche diese neuen Systeme boten, entstand eine Fülle an neuen Spielkonzepten. Das führte zu einer weiten Verbreitung der Spiele-Systeme. Selbst der Computermarkt und seine Akzeptanz im Privathaushalt wären ohne Computer-Systeme, wie Amiga und C64, in der jetzigen Form nicht denkbar. Aus der heutigen Zeit sind Computer- und Konsolen-Spiele nicht mehr wegzudenken, sie sind ein akzeptierter Bestandteil des Lebens geworden. Die Umsätze¹ der Spiele-Industrie beliefen sich im Jahr 2003 auf 23,2 Mrd. Dollar, das beweist das enorme Interesse an Spielen.

1.1 Kurzzusammenfassung

Diese Diplomarbeit befasst sich mit der Entwicklung und Beschreibung eines Frameworks für die Spielentwicklung auf dem GameBoyAdvance (GBA).

Der erste Teil befasst sich mit der CPU des GBA, hier wird besonders auf den Befehlsatz eingegangen, welcher ein wichtiger Bestandteil für das Verständnis der Programmierung des Frameworks darstellt. Im dritten Kapitel gehe ich auf die GBA-Hardware ein, beschreibe aus Gründen des Umfangs nur kurz den Registersatz. In Teil vier, wird auf die Entwicklungsumgebung und die Vielzahl an Programmen eingegangen, mit welcher eine Entwicklung auf dem GBA besritten werden kann. Kapitel 5 befasst sich kurz mit Standardfunktionen. In Kapitel 6, wird die Soundausgabe des GBA beschrieben und einige Möglichkeiten des Frameworks dargestellt um erstellte Musik abzuspielen. Kapitel 7, erläutert einige Arten zum Abspielen von grafischen Animationen und führt den Benutzer, in das zu Verfügung gestellte Framework ein. Kapitel 8, beschreibt die Probleme die hinter einer Kollisionserkennung stecken und stellt mehre Ansätze und das vom Framework bereitgestellte System dar. Kapitel 9, beschreibt das Speicher-Management. Kapitel 10, geht auf die Vielzahl an grafischen Methoden ein. Kapitel 11, zeigt eine Test-Implementierung und beweist die Funktionalität des Frameworks in der Realität.

1.2 Übersicht

Es gibt mehrere Frameworks für den GBA, die den Programmierer bei der Entwicklung von Software (vornehmlich Spiele) unterstützen. Das umfangreichste Framework für den GBA nennt sich HAM², es deckt von der Entwicklungsoberfläche bis zum Compiler alles ab. Ist also besonders für Einsteiger geeignet. Der Nachteil liegt, bedingt durch die Orientierung in Richtung Programmieranfänger, in der starken Kapselung der eigentlichen Hardware zur Software.

¹ URL: <http://derstandard.at/?id=1628030> (27.07.2004)

² URL: www.ngine.de/ham.html (27.07.2004)

Dadurch wird für fortgeschrittene Anwender der Funktionsumfang schlecht nutzbar, da er für viele Sonderfälle und Erweiterungen nicht verwendbar ist. Andere Frameworks, wie z.B. die „Mushroom GBALib“³, beschreiten einen anderen Weg. Sie bieten dem Benutzer nur Bibliotheken mit Funktionen an, die er dann in seinen Projekten verwenden kann. In vielen Fällen, werden die Quell-Texte mitgeliefert. Der Vorteil solcher Systeme ist, dass der Anwender die Quelltexte je nach Wunsch verändern kann. Die Kapselung zwischen Hardware und Software bei solchen Frameworks ist meist gering und erlaubt so eine größere Bewegungsfreiheit für den Benutzer. Leider bieten alle die von mir getesteten Systeme einen nur sehr geringen Funktionsumfang, behandeln besonders für die Spiele-Programmierung wichtige Funktionalitäten nicht, oder nur unzureichend. Sie sind meist durch unnötige Kapselungen unperformant und schlecht erweiterbar.

1.3 Zielsetzung

Das Ziel dieser Diplomarbeit ist es, ein Framework zu erstellen, das dem Benutzer Funktionen zur Verfügung stellt, die auch eine reale Bedeutung in der Spiel-Entwicklung haben. Alle Funktionen sollen inkl. Quelltext für den Benutzer verwendbar sein. Als Programmiersprache wird Assembler gewählt, da besonders auf der GBA-Hardware eine hohe Verarbeitungsgeschwindigkeit erreicht werden kann. Als Beweis der Funktionalität des Frameworks soll ein kurzes Testspiel implementiert werden.

³ URL: <http://mushroom.sourceforge.net> (27.07.2004)

2 Der Hauptprozessor

In den nächsten Kapiteln, möchte ich etwas näher auf den zentralen Prozessor des GBA eingehen. Dabei wird unter anderem die Integration in das Gesamtsystem und ihre Auswirkung auf die Leistungsfähigkeit bzw. die Art des Programmierens eingegangen. Schlussendlich gebe ich einen kurzen Überblick des Befehlsatzes wieder, um das spätere Arbeiten mit Assembler-Programmierungen zu vereinfachen.

2.1 Anforderungen

Ein wichtiger Punkt bei einem tragbaren System ist der Stromverbrauch und die Art der Integration eines Prozessors in das System. In den letzten Jahren ist die ARM-Architektur (IP von Advanced Risc Machines) zum Marktführer in diesem Bereich aufgestiegen. Es gibt dafür verschiedene Gründe. Zum einen handelt es sich um eine 32-Bit-RISC-Architektur, die in mehreren Ausbaustufen erhältlich ist (erweiterbar mit FPU/MMU) und so je nach Bedarf konfiguriert werden kann. Weiterhin ist ein Hauptargument für diese Modellfamilie, die Möglichkeit, die CPU als VLSI-Modell (Very Large Scale Integration) zu kaufen. Dadurch kann die CPU in ein bestehendes Design integriert werden und muss nicht extern auf einer Platine mit dem System verdrahtet werden. ARM erlaubt auch eine Lizenzierung seiner Technologie und so die Möglichkeit, die CPU zu verändern. Einige bekannte Beispiele, hierfür sind StrongARM und XScale von Intel, wie auch OMAP von Texas Instrument. Nintendo geht im GBA den ersten Weg und integriert eine fertige CPU in ihr System (AGBCPU).

2.2 ARM7tdmi Überblick

Der verwendete Prozesserkern ist vom Typ: ARM7tdmi, welcher mit 16777216 Hz getaktet wird.

Als Besonderheiten dieses Prozessortyps werden von ARM⁴ folgende Punkte angegeben:

- 32/16-bit RISC Architektur (ARM v4T, ARM und Thumb Modus)
- 32-bit ARM Instruktionsset für maximale Geschwindigkeit und Flexibilität
- 16-bit Thumb Instruktionsset zur Verbesserung der Codedichte
- Einheitliches Businterface, 32-bit Datenbus für Daten und Code
- Dreistufige Pipeline
- 32-bit ALU
- Geringe Größe und geringer Stromverbrauch
- Coprozessorinterface

Ergänzend ist noch zu erwähnen, dass die Pipelinestufen wie folgt aufgebaut sind.

- Fetch (hole Instruktionen aus dem Speicher)

⁴ ARM Inc. URL: <http://www.arm.com> (27.07.2004)

- Decode (dekodieren der im OpCode angegebenen Register)
- Execute (Register aus Registerbank lesen, Shift und ALU-Operationen ausführen, Register zurück schreiben)

Datenzugriffe können nur über load/store und swap Funktionen erfolgen, ihre möglichen Datentypen sind 8 (Byte), 16 (Halfword) und 32 Bit (Long). Bei Speicherzugriffen muss die Ausrichtung entsprechend dem Datentyp der Operation beachtet werden.

2.3 Speicher und seine Anbindung

Es ist vielleicht etwas ungewöhnlich bei der Beschreibung einer CPU und ihrer Anbindung an das System, die Speicheranbindung als dominanten Punkt anzuführen. Aber in vielen Systemen ist die Rechenleistung stark von der Speicheranbindung abhängig, zumal der verwendete Prozessor keinen Cache besitzt und über langsame 16-Bit Busse auf den Speicher zugreifen muss.

Ein Grundproblem ist, dass sich die eigentlichen Daten im externen ROM befinden. Das Problem ist, dass der externe ROM nur eine 16-Bit-Anbindung an die CPU ermöglicht. Dies verursacht einen Performanceverlust beim Lesen von Daten aus dem externen ROM.

Dieses Problem existiert aber analog im internen Speicher des GBA (EWRAM), denn auch dieser hat nur eine 16 Bit Anbindung an die CPU. Für den Programmierer ergibt sich hier als Lösungsmöglichkeit nur das Verwenden von Thumb-Code um Wartezyklen (Wait-States) zu verhindern. Da diese Codeart aber in vielen Fällen die Möglichkeiten des Prozessors nicht optimal nutzt, ergibt sich ein sehr unbefriedigender Programmierstil.

Aus diesem Grund besitzt der GBA noch einen 32 KByte großen internen Speicher (IWRAM (Internal Work Random Access Memory)), welcher ohne Wait-States und mit einem 32 Bit-Bus arbeitet. Die soeben genannten Speicher sind im Normalfall für Code und Daten zu verwenden, sind aber immer allgemein zu betrachten und nicht für spezielle Arten von Daten gedacht. Wie in vielen für Spiele entworfenen Geräten, existieren für mehrere unterschiedlich Aufgaben getrennte Speicherbereiche.

Das externe ROM kann maximal einen adressierbaren Bereich von 32 MByte ausfüllen. Der Arbeitsspeicher hat eine Größe von 256 KByte und das schnelle interne RAM hat eine Größe von 32 KByte. Neben dem ROM gibt es noch einen zweiten Read-Only-Speicher mit 16 KByte Größe. Bei diesem Speicher handelt es sich um das BIOS (Basic Input Output System). Darin befinden sich mehrere Funktionen, die dem Benutzer in vielen Punkten hilfreich sein können (siehe Kapitel 3.10). Im IO-RAM (Input Output Random Access Memory) befinden sich alle Hardware-Register die benutzt werden können (siehe Kapitel 3.3). Das VRAM (Video Random Access Memory) enthält Daten die auf dem Display dargestellt werden können und ist je nach verwendetem Grafikmodus, unterschiedlich aufgeteilt. Das Paletten-RAM enthält Farbdaten die über das Video-RAM indiziert werden. Spritedaten werden in dem OAM RAM (Object Attribute Memory Random Access Memory) abgelegt. Das Cartridge-RAM ist eine optionale Komponente und wird im Allgemeinen zur Sicherung von Spieledaten verwendet. Diese Daten müssen auch

nach dem Ausschalten des GBA erhalten bleiben. Um dies zu realisieren gibt es im Normalfall drei Arten der Sicherung. Die Erste ist die Verwendung von SRAM, gefolgt von der Sicherung auf FLASH-Speichern und EEPROMS.

Die folgende Abbildung stellt die Speichertypen mit ihrem Bussystem und zugehörigen Wartezyklen dar.

Type	Busbreite	Lesen	Schreiben	Zyklen
Cartridge ROM	16	8/16/32		5/5/8 (Default)
Externer-Work-RAM (WRAM)	16	8/16/32	8/16/32	3/3/6 (Default)
Interner-Work-RAM (IRAM)	32	8/16/32	8/16/32	"1/1/1"
BIOS ROM	32	8/16/32		"1/1/1"
I/O	32	8/16/32	8/16/32	"1/1/1"
OAM (Sprites)	32	8/16/32	16/32	"1/1/1" + 1 wenn Zugriff von GBA
Palette RAM	16	8/16/32	16/32	"1/1/1" + 1 wenn Zugriff von GBA
VRAM (Video-RAM)	16	8/16/32	16/32	"1/1/1" + 1 wenn Zugriff von GBA
FLASH (Cartridge)	16	8/16/32	16/32	5/5/8 (Default)
SRAM (für Spielstände)	8	8	8	5 (Default)

Abbildung 2-1 Speicher im Detail

Das Cartridge ROM, kann je nach verwendetem ROM auch mit schnelleren Timings benutzt werden. Ein häufig genutzter Fall ist der Zugriff mit drei Wartezyklen bei dem ersten Zugriff und einem Wartezyklus beim zweiten Zugriff. Der zweite Zugriff, mit seinem Wartezyklus ist nur bei sequentiellem Zugriff möglich. D.h. es müssen mindestens zwei Worte übertragen werden, in diesem Fall gilt, erster Zugriff mit drei Wartezyklen, alle folgenden Zyklen mit einem Wartezyklus. Bei zufälligem Zugriff auf den Speicherbereich entstehen immer drei Wartezyklen pro Zugriff.

2.4 Registermodel

Der ARM7 hat 16 generelle Register und ein spezielles Register. In privilegierten (siehe Kapitel 2.7), Betriebsarten werden wie in Abbildung 2-3 dargestellt, einige Register gesichert.

Modus	Register
Normal	r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13 (sp), r14 (lr), r15 (pc)
FIQ	r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq (sp_fiq), r14_fiq (lr_fiq)
Supervisor	r13_spv (sp_spv), r14_spv (lr_spv)
Abort	r13_abr (sp_abr), r14_abr (lr_abr)
IRQ	r13_irq (sp_irq), r14_irq (lr_irq)
Undefined	r13_und (sp_und), r14_und (lr_und)

Abbildung 2-2 Registersatz

Der ersten 13 Register sind GPR's (General Purpose Register), haben also keine gesonderte Aufgabe. Register R13 wird als Stackpointer verwendet. Register R14 ist der Linkregister, dieser Register findet eine Verwendung in der Speicherung der Rücksprungadresse bei einem

Modus (THUMB-Modus), in diesem Betriebszustand sind alle Befehle nur noch 16 Bit lang. Die Befehlssätze sind sich sehr ähnlich, der Grund dafür ist, dass die Thumb-Befehle in der CPU ohne unmittelbaren Zeitverbrauch in ARM-Code umgewandelt werden.

Betrachten wir die Vor- und Nachteile der beiden Codearten:

ARM Pro:

- Befehle mit meist 3 Operanten
- Shift/Rollfunktionen in vielen Befehlen integriert
- Bedingungen für die Ausführung der Befehle setzbar
- Schnelle Ausführung auf 32 Bit Bussystemen

ARM Contra:

- langsam bei 16 Bit Speicher
- mehr Speicherverbrauch, da Befehle 4 Byte lang

THUMB Pro:

- schnelle Ausführung auf 16 Bit Bussystemen (ca. 160% schneller als ARM)
- weniger Speicher, da nur 2 Byte pro Befehl

THUMB Contra:

- da die Befehle kleiner sind, haben sie eine niedrige Komplexität, deshalb fallen Funktionen wie integrierter Shift und Bedingungen pro Befehl etc. weg
- es sind bei den meisten Befehlen nur die Register R0-R7 benutzbar

Wie bereits im Kapitel Speicher und seine Anbindung beschrieben, gibt es im GBA mehrere unterschiedliche Speicher. Für den ROM und EWRAM, ist bedingt durch sein 16 Bit-Bussystem, im Regelfall Thumb-Code zu benutzen. Im schnellen IWRAM, welcher auf einen 32 Bit-Bussystem zurückgreifen kann, ist in fast allen Fällen ARM Code zu bevorzugen.

Um diesen Flaschenhals beim Datentransfer zu beseitigen, ist ein spezieller Modus in den GBA integriert worden. Den so genannten **Prefetch Buffer**. Betrachtet man seine Vor- und Nachteile, ergibt sich folgendes Resultat:

- Der Vorteil ist, dass die CPU den Code aus dem ROM liest, obwohl sie noch nicht benötigt werden (Prefetch). Falls keine Abbrüche im linearen Ablauf auftreten, kann die CPU die nächsten Befehle ohne Verzögerung laden. Somit steigt die Geschwindigkeit, dadurch ist der ROM durchschnittlich schneller als der WRAM.

- Der Nachteil ist, dass dieses Verfahren mehr Strom benötigt, es sollen bis zu 10% mehr Strom verbraucht werden (laut GBATEK⁵).

2.6 Der Befehlssatz

Im Folgenden wird in kurzer Form auf den ARM-Befehlssatz eingegangen, da die Thumbcodierung in vielen Bereichen ähnlich ist, wird keine weitere Erläuterung über diese angegeben. Um weitere Informationen zu erhalten, sind die Dokumentationen von ARM⁶ zu empfehlen. In den Beispielen angegebene Kommentare beschreiben die Funktion zusätzlich, haben aber keine Auswirkung auf die Funktionen.

2.6.1 Arithmetische Befehle

Der ARM7tdmi bietet die gebräuchlichsten arithmetischen Funktionen an: *add* (Addition), *adc* (Addition mit Carry), *sub* (Subtraktion), *sbc* (Subtraktion mit Carry), *rsb* (Subtraktion mit vertauschten Quellen), *rsc* (Subtraktion mit vertauschten Quellen mit Carry). Diese können optional noch mit einem Shift-Befehl und einer Bedingung kombiniert werden. Durch das Anhängen eines kleinen ‚s‘ an den Befehlsnamen, erzeugt der Befehl „Conditioncodes“.

Ein Beispiel für die geläufigen arithmetischen Funktionen:

<code>add</code>	<code>r3,r2,r1</code>	<code>;r3 = r2 + r1</code>
<code>adc</code>	<code>r3,r2,r1</code>	<code>;r3 = r2 + r1 + C</code>
<code>sub</code>	<code>r3,r2,r1</code>	<code>;r3 = r2 - r1</code>
<code>sbc</code>	<code>r3,r2,r1</code>	<code>;r3 = r2 - r1 + C - 1</code>
<code>rsb</code>	<code>r3,r2,r1</code>	<code>;r3 = r1 - r2</code>
<code>rsc</code>	<code>r3,r2,r1</code>	<code>;r3 = r1 - r2 + C - 1</code>

Ein Beispiel für eine Erweiterung mit, Bedingung, Shift (schieben) und Flags setzen:

<code>addeq</code>	<code>r3,r2,r1</code>	<code>lsl r4</code>	<code>;Z-Flag = 1 dann r3 = r2 + (r1 << r4)</code>
			<code>;setze Flags</code>

2.6.2 Logische Befehle

Die zur Verfügung stehenden logischen Befehle sind: *and*, *orr*, *eor* und *bic*

And ist ein logisches UND, *orr* ein logisches ODER, *eor* ein logisches exklusiv ODER. *Bic* ist ein AND NOT, welches als Filter gesehen, wie ein Bit Clear (deshalb bic) wirkt. Notwendigerweise arbeiten diese Funktionen bitweise. Auch hier ist eine Erweiterung mit Shifts (schieben), Bedingung und Flags speichern möglich.

⁵ URL: <http://www.work.de/nocash/gbatek.htm> (27.07.2004)

⁶ Arm Inc. URL: <http://www.arm.com> (27.07.2004)

Zum Beispiel:

```
mov    r1,#80
mov    r2,#88
bic    r3,r2,r1          ;$08 = $88 AND NOT $80
orrs   r1,r2,r1 lsl #8   ;$8088
```

2.6.3 Datentransfer

Die „move“ Befehle haben bei RISC Prozessoren nur noch eine geringe Bedeutung, da sie nur auf Register und Konstanten zugreifen können. Jeglicher Zugriff auf Speicherbereiche wird über spezielle Befehle abgewickelt (siehe Kapitel 2.6.5). Wenn konstante Werte übergeben werden, ist mit der Einschränkung zu arbeiten, dass diese nur maximal 8 Bit breit sind. Jedoch noch zusätzlich bitweise nach links verschoben werden können. Auch hier ist eine Erweiterung mit Shift, Bedingung und Flags setzen möglich.

Zum Beispiel:

```
mov    r2,r1              ;r2 = r1
mvn    r2,r1,ror #4       ;r2 = ~(r1 rolle rechts um 4)
mov    r4,#$68000
```

2.6.4 Vergleichsfunktionen

Die ARM7tdmi CPU unterscheidet sich hier kaum von anderen Architekturen. **cmp** (normaler Vergleich), **cmn** (negativer Vergleich), **tst** (logischer Vergleich mit und), **teq** (logischer Vergleich mit Exklusiv-Oder). Logischerweise besitzen die Vergleichsfunktionen keine optionales Flag zum Condition-Code setzen, sondern setzen diese immer.

```
cmp    r2,r1              ;r2 - r1
cmn    r2,r1              ;r2 + r1
tst    r2,r1              ;r2 and r1
teq    r2,r1              ;r2 xor r1
```

Die setzbaren Flags:

- Z (Zero)
- N (Negative)
- V (Overflow)
- C (Carry)

Die dadurch erzeugten möglichen Konditionen werden in Abbildung 2-5 Conditions beschrieben.

Name	Mnemonic	Opcode	Status Flags
Equal (gleich)	eq	0	z=1
Not Equal (ungleich)	ne	1	z=0
Carry Set	cs	2	c=1
Carry Clear	cc	3	c=0
Unsigned Higher or Same (>=)	hs	2	c=1
Unsigned Lower (<)	lo	3	c=0
Minus/Negativ	mi	4	n=1
Plus/Positive oder Null	pl	5	n=0
Overflow (Überlauf)	vs	6	v=1
No Overflow (kein Überlauf)	vc	7	v=0
Unsigned Higher (>)	hi	8	c=1, z=0
Unsigned Lower of Same (<=)	ls	9	c=0, z=1
Signed Greater Than or Equal (>=)	ge	10	n=v
Signed Less Than (<)	lt	11	n<>v
Signed Greater Than (>)	gt	12	z=0, n=v
Signed Less Than or Equal (<=)	le	13	z=1, n<>v
Always	al	14	
Never	ne	15	

Abbildung 2-5 Conditions

2.6.5 Datentransfers (Speicherzugriffe)

Da die ARM-CPU auf einer "load and store" Architektur basiert, beziehen sich alle Speicherzugriffe auf load- (lesen) und store- (schreiben) Befehle. Es gibt nur eine Ausnahme, den *swp* (swap) Befehl, er dient zur einfachen Realisierung von Softwaresemaphoren. Deswegen ist es nicht möglich, direkt oder indirekt auf Speicher zuzugreifen, ohne diese Befehle zu benutzen.

Der *ldr* Befehl, ist der Standard lade Befehl.

zum Beispiel:

<code>ldr r0,[r1]</code>	<code>;r0 = Speicher[r1]</code>
--------------------------	---------------------------------

Es wird der Inhalt der Speicherzelle, deren Adresse in R1 steht in R0 geladen.

Größenangaben für die zu ladenden Datentypen sind zusätzlich setzbar.

<code>ldrb r0,[r1]</code>	<code>;Byte</code>
<code>ldrh r0,[r1]</code>	<code>;Halbwort (16Bit)</code>
<code>ldr r0,[r1]</code>	<code>;Wort (32Bit)</code>

Speicherinhalt schreiben erfolgt analog:

<code>str r0,[r1]</code>	<code>; Speicher [r1] = r0</code>
<code>strh r0,[r1]</code>	<code>; Speicher [r1] = r0 (Halbwort)</code>

Zu bemerken ist in diesem Fall, dass das Ziel rechts steht und nicht wie beim ARM üblich links. Die Typengrößen sind wie beim *ldr*, also *lshb* (Byte), *lsh* (16 Bit Wort) und *lshl* (32 Bit Wort). Die Lade/Schreibbefehle weisen noch vielfältige Optionen auf.

Zum Beispiel ist ein konstantes Offset zusätzlich möglich.

```
ldr    r0,[r1,#4]        ;r0 = Speicher [r1 + 4]
```

Diese Offset-Befehle sind immer "pre-indexed" und können über ein Shift erweitert werden. Weiterhin gibt es noch ein "auto-indexing", welches Zieladressen inkrementiert.

```
ldr    r0,[r1,#4]!       ;r0 = Speicher [r1 + 4]
                               ;r1 = r1 + 4
```

Ebenso ist ein "post-indexed" vorhanden, mit welchen Zieladressen in- oder dekrementiert werden können.

```
ldr    r0,[r1],#4        ;r0 = Speicher [r1]
                               ;r1 = r1 + 4
```

Wie viele andere Architekturen, bietet der ARM7tdmi auch das Laden und Schreiben von mehreren Registern. Nicht zu verachten ist die gebotene Komplexität dieser Befehle, denn es gibt ein Post/Pre-increment und auch ein Post/Pre-decrement.

Zum Beispiel:

```
ldmia r1,{r2,r3,r4}      ;r2 = Speicher [r1]
                               ;r3 = Speicher [r1+4]
                               ;r4 = Speicher [r1+8]

ldmia r1!,{r2,r3,r4}     ;r2 = Speicher [r1]
                               ;r3 = Speicher [r1+4]
                               ;r4 = Speicher [r1+8]
                               ;r1 = r1 + 4 + 4 + 4
```

Das Kürzel „ia“ beim ldmia bedeutend „increment after“ (bezieht sich also auf r1).

Insgesamt gibt es aber vier Möglichkeiten:

```
ia = increment after
ib = increment before
da = decrement after
db = decrement before
```

Abbildung 2-6 ldm/stm-Erweiterungen

Bei Stackoperationen verwenden diese Funktionen eine andere Mnemonik.

(Stack)	(Normal)	(Function)
ldmed	ldmib	pre increment
ldmfd	ldmia	post increment
ldmea	ldmdb	pre decrement
ldmfa	ldmda	post decrement
stmea	stmib	pre increment
stmfa	stmia	post increment
stmed	stmdb	pre decrement
stmfd	stmda	post decrement

Abbildung 2-7 ldm/stm-Mnemonik

2.6.6 Sprungfunktionen

Wie bei fast allen Architekturen bietet der ARM7tdmi dem Benutzer bedingte und unbedingte Sprünge an. Um den Betriebsmodus von ARM in THUMB bzw. umgekehrt zu wechseln, existiert ein weiterer Befehl, der auch einen Sprung ausführt.

2.6.6.1 Unbedingte Sprünge

Der Standardsprungbefehl hat die Mnemonik „b“ (für Branch). Seine maximale Sprungweite beträgt im ARM-Modus 24 Bit. Bei größeren Sprüngen kann auch ein direktes Laden des Programm-Counters (PC) benutzt werden.

Zum Beispiel:

```
ldr    pc,=label_far    ;lädt aus pool
.
.
b      label_near
```

Wie schon angemerkt, existiert noch ein weiterer Befehl, der neben einem Sprung noch den Maschinensatz (ARM/THUMB) setzen kann. Seine Mnemonik ist „bx“, welches für „Branch and Exchange“ steht. Der Unterschied zum normalen Branch ist, dass die an die Zieladresse, mit einer „1“ verodert werden muss, falls es einen Sprung auf THUMB-Segment geben soll und diese Adresse in einem Register steht.

Zum Beispiel:

```
ldr    r0,=draw_circle_thumb|1 ; Zieladresse oder 1
bx     r0
```

Der letzte unbedingte Sprungbefehl ist „bl“. Dieses Kürzel steht für „Branch with Link“. Er ist identisch mit dem normalen Branch, nur mit der Besonderheit, dass die Rücksprungadresse im Linkregister gespeichert wird.

Zum Beispiel:

```

    bl    loesche_r6          ; pc = loesche_r6, lr = nächster Befehl
    .
    .

loesche_r6
    mov   r6,#0
    bx    lr                  ; pc = lr

```

2.6.6.2 Bedingte Sprünge

Da im ARM-Modus alle Befehle mit einer Bedingung ausgestattet werden können, sind auch alle zuvor genannten unbedingten Sprünge mit einer Bedingung kombinierbar. Im THUMB-Modus, gibt es aber nur die unbedingten Sprünge „b“ und „bl“.

2.6.7 Software-Interrupts

Eine weitere Möglichkeit einen Sprung hervorzurufen, ist der Aufruf eines Softwareinterrupts. Der dafür benutzbare Befehl lautet „swi“ und akzeptiert eine 20 Bit breite Interrupt-Nummer. Da alle Interrupt-Vektoren im BIOS (siehe Kapitel 3.10) liegen und auch von diesem behandelt werden, ergibt sich nur eine sehr eingeschränkte Verwendung für den Benutzer.

2.6.8 Multiplikation und Division

Der ARM7tdmi stellt sechs Multiplikationsfunktionen zur Verfügung:

mul, mla, umull, umlal, smull, smlal

Zu beachten ist, dass ***mul, mla*** (Multiplikation mit zusätzlicher Addition) nur ein 32 Bit Ergebnis zurück liefert.

Zum Beispiel:

```

    mul   r3,r2,r1           ; r3 = (r2 * r1) & $ffffffff
    mla   r4,r3,r2,r1        ; r4 = (r3 * r2 + r1) & $ffffffff

```

Um ein 64 Bit großes Ergebnis zu sichern, müssen die Funktionen ***umull, umlal, smull, smlal*** benutzt werden. Das 64 Bit große Ergebnis wird in zwei Registern gespeichert. Das Angeführte „s“ oder „u“, steht für vorzeichenbehaftet und nicht vorzeichenbehaftet,

<code>umull r4,r3,r2,r1</code>	<code>;r4:r3 = r2 * r1</code>
<code>umlal r4,r3,r2,r1</code>	<code>;r4:r3 = r2 * r1 + r4:r3</code>
<code>smull r4,r3,r2,r1</code>	<code>;r4:r3 = r2 * r1</code>
<code>smlal r4,r3,r2,r1</code>	<code>;r4:r3 = r2 * r1 + r4:r3</code>

Leider bietet der ARM7tdmi keine Divisionsfunktion an. Aus diesem Grund müssen alle Divisionen über Softwareimplementierungen errechnet werden (siehe Kapitel 5.1). Da diese Verfahren sehr viel Zeit verbrauchen, sollten Divisionen vermieden werden.

2.6.9 Pseudo-Befehle

Pseudo-Befehle sind Befehle die nicht wirklich in Hardware existieren, sondern die in Befehle mit gleicher Funktion umgesetzt werden. Sie werden meist vom Assembler bereitgestellt um den Benutzer die Arbeit zu erleichtern. In Abbildung 2-8 werden die gebräuchlichsten Pseudo-Befehle dargestellt. Bei dem *mov*-Befehl im Thumb-Modus ist zu beachten, dass er nur in `add Rd,Rs,#0` umgesetzt wird, falls Rd oder Rs keine High-Register sind (kleiner R8).

Pseudo-Befehl	Arm	Thumb
<code>adr Rd,<Addr></code>	<code>add Rd,PC,#<Addr Offset></code>	<code>add Rd,PC,#<Addr Offset><<2</code>
<code>nop</code>	<code>mov R0,R0</code>	<code>mov R8,R8</code>
<code>mov Rd,Rs</code>	<code>mov Rd,Rs</code>	<code>add Rd,Rs,#0</code>

Abbildung 2-8 Pseudo-Befehle

Aus Gründen der nicht gestatteten Byte-Zugriffe (mindesten 16-Bit Zugriff) in den VRAM. Bietet mein Framework Pseudo-Operationen an (Macros), welches sequentiell zuschreibende Bytes sichert und gegebenenfalls als Long-Worte schreibt. Für die Verarbeitung die Sicherung der Daten wird ein temporärer Register verwendet. Der Befehl *strbli* initialisiert den temporären Register (Rt). Der Befehl *strbl* schreibt den Register Rq in die Zieladressen (Rd), je nach Aufruf wird nur in Rt gesichert oder direkt geschrieben. *Strblf* hat die gleiche Syntax wie *strbl*, ist nur als letzter Aufruf zu verwenden, um die restlichen Bytes im temporären Register zu schreiben (flush).

Pseudo-Befehl	Little - Endian	Big - Endian
<code>strbli Rt</code>	<code>mov Rt,\$80000000</code>	<code>mov Rt,\$1</code>
<code>strbl Rq,Rd,Rt</code>	<code>mov Rq,Rq,lsr #24</code> <code>orrs Rt,Rq,Rt,lsr #8</code> <code>strcs Rt,[Rd],#4</code> <code>movcs Rt,\$80000000</code>	<code>orrs Rt,Rq,Rt,lsr #8</code> <code>strcs Rt,[Rd],#4</code> <code>movcs Rt,\$1</code>
<code>strblf Rq,Rd,Rt</code>	<code>cmp Rt,\$80000000</code> <code>moveq Rt,#0</code> <code>and Rq,Rt,\$fff</code> <code>cmp Rq,\$80</code> <code>mov Rt,Rt,lsr #8</code> <code>streq Rt,[Rd],#4</code> <code>moveq Rt,#0</code> <code>and Rq,Rt,\$fff</code> <code>cmp Rq,\$80</code> <code>mov Rt,Rt,lsr #8</code> <code>streq Rt,[Rd],#4</code> <code>movne Rt,Rt,lsr #8</code> <code>strne Rt,[Rd],#4</code>	<code>cmp Rt,\$01</code> <code>moveq Rt,#0</code> <code>and Rq,Rt,\$fff</code> <code>cmp Rq,\$0100</code> <code>movne Rt,#0</code> <code>moveq Rt,Rt,lsr #24</code> <code>streq Rt,[Rd],#4</code> <code>and Rq,Rt,\$fff0000</code> <code>cmp Rq,\$010000</code> <code>moveq Rt,Rt,lsr #16</code> <code>streq Rt,[Rd],#4</code> <code>movne Rt,Rt,lsr #8</code> <code>strne Rt,[Rd],#4</code>

Abbildung 2-9 "Byte zu Long" Pseudo-Befehle

Zu Beachten ist, dass die Little-Endian Version das Quell-Register (Rq) bei den Operationen *strbl* und *strblf* verändert. Die Big-Endian Version verändert Rq nur bei *strblf*. Außerdem ist sie bei *strbl* um 2 Taktzyklen schneller, als die Little-Endian Variante. Leider ist der ARM7tdmi Prozessor im GBA auf Little-Endian konfiguriert.

Wenn möglich sollte der Einsatz dieser Operationen vermieden werden und ein anderer Weg, für einen vermeiden des Byte-Zugriffs, gesucht werden. Jedoch ist bei vielen Vorgefertigten C-Funktionen eine Änderung im Code sehr schwierig möglich, in diesen Fällen bieten diese Funktionen eine schnelle und einfache Lösung ohne den Code grundlegend abzuändern. Eine Beispielimplementierung (LZSS-Dekoder) ist unter dem Pfad `source/anim/lzss_dec_big_test/` zu finden.

2.7 Exceptions

Exceptions (Ausnahmen) werden durch Interrupts ausgelöst. Der ARM7tdmi Prozessor erlaubt folgende Ausnahmen (nach ihrer Priorität geordnet).

- Reset
- Data Abort
- FIQ (Fast (schneller) Interrupt)
- IRQ (normaler Interrupt)
- Prefetch Abort
- Software Interrupt (durch swi-Instruktion auslösbar)
- Undefined Instruction (unbekannter Befehl)

Je nach ausgelöstem Interrupt, verwendet die ARM7tdmi-CPU andere Register-Sätze (siehe Abbildung 2-10). Dadurch kann ein manuelles sichern dieser Register bei Verwendung vermieden werden.

Interrupt-Modus	gesicherte Register
FIQ Mode	R8,R9,R10,R11,R12,R13,R14
Supervisor Mode	R13,R14
Abort Mode	R13,R14
IRQ Mode	R13,R14
Undefined Mode	R13,R14

Abbildung 2-10 IRQ-Modus mit Registersatz

Die Vektoren zur Vergabe der entsprechenden Softwarefunktionen befinden sich, wie bei vielen Prozessoren üblich, auf den niedrigsten Adressen (Zeropage). Abbildung 2-5 verdeutlicht die Interrupts und ihre zugehörigen Vektoren-Adressen.

Adresse	IRQ
\$0	Reset
\$4	Undefined Instruction
\$8	Software Interrupt
\$c	Prefetch Abort
\$10	Data Abort
\$14	(Reserviert)
\$18	Normaler Interrupt (IRQ)
\$1c	Fast Interrupt (FIQ)

Abbildung 2-11 IRQ-Vektoren

Da beim GBA, diese Vektoren im BIOS liegen, kann kein direkter Zugriff vom Benutzer erfolgen. Alle auftretenden Interrupts, werden infolgedessen vom BIOS verarbeitet. Im Fall von Software-Interrupts, werden die entsprechenden vom BIOS definierten Funktionen aufgerufen (siehe Kapitel 3.10). Die Normalen Interrupts (IRQ) werden auf eine vom Benutzer festlegbare Adresse weitergeleitet. Da diese Interrupts von der Hardware ausgelöst werden wird in Kapitel 5.2.1 näher auf diese Problematik eingegangen.

2.8 Fazit

Es existieren viele Systeme mit einer schlecht konditionierten CPUs. Darunter ist nicht in allen Fällen eine zu geringe Rechenleistung zu verstehen, sondern in vielen Fällen auch ein unnötige hohe Rechenleistung und einen in fast allen Fällen damit verbundenen hohen Stromverbrauch. Der ARM7tdmi war eine sehr gute Wahl für den GBA. Die Rechenleistung ist ausreichend. Sie liegt bei ca. 0.9 Befehlen pro Taktzyklus, als Vergleich benötigt ein 68040 CPU mit 6-Stufiger-Piplen und Cache ca. 0.8 Befehle pro Taktzyklus. Der Stromverbrauch ist ebenfalls sehr gering. Die CPU selber bietet viele interessante Features und lässt sich auch leicht programmieren. Als Kritikpunkt müssen die langsame Speicheranbindung (16 Bit) an den externen Arbeitsspeicher (EWRAM) hervorgehoben werden. In diesem Fall wäre eine schnellere Anbindung von Vorteil gewesen. Weiterhin ist der Thumb-Code-Modus, bedingt durch die schnelle interne

Konvertierung in ARM-Code, in seinem Befehlsatz eingeschränkt. Änderungen diesbezüglich werden mit Thumb2 in neuen ARM-Cores beschränkt. Die Verwendung des Little-Endian-Formats ist ebenfalls unnötig, besonders da die komplette Hardware des GBA Big-Endian ist (Hardwareregister wurden um 16Bit gedreht). Allerdings ist die ARM-CPU extern auf beide Modi einstellbar, warum Nintendo sich für den Little-Endian Modus entschieden hat liegt in der zusätzlichen Verwendung der Z80 CPU im GB- bzw. GBC-Modus, da bei 8-Bit CPUs das Little-Endian-Format geeigneter ist (Der Vorteil war, dass das niedrigere Adressbyte zu einem Index-Register addiert werden konnte , während das obere Byte noch geholt wurde).

3 GBA Hardware

3.1 Technische Daten

3.1.1 Betriebsarten

Der GBA besitzt mehrere Betriebsarten. Neben den normalen GBA-Modus existieren noch Abwärtskompatibilitäten zum GameBoyColor© und dem Gameboy©. Da diese Geräte einen andere CPU und eine andere Hardware benutzen, wurden beiden Systeme in Hardware in das System integriert. Durch das eingeführte Cartridge wird entschieden, in welchen Betriebsmodus gestartet werden soll. Es besteht keine Möglichkeit der Kommunikation zwischen den unterschiedlichen Systemen.

Die folgenden Punkte, beschreiben den Betriebsmodus, seine CPU und deren Taktfrequenz.

- GameBoyAdvance© ARM7TDMI mit 16.777216 MHz
- GameBoyColor© Z80 CPU mit 4.2 MHz oder 8.4 MHz
- GameBoy© Z80 CPU mit 4.2MHz

3.1.2 Interne Speicher

- BIOS ROM 16 kBytes
- Work RAM 288 kBytes
- Video RAM 96 kBytes
- OAM 1 kByte
- Palette RAM 1 kByte

3.1.3 Grafikdaten

- Bildschirm 240x160 Pixels (2.9 Zoll TFT-Farb-Display)
- Hintergrundebenen 4 Hintergrundebenen
- Hintergrundarten Tile/Map basierendes Bitmap
- Hintergrundfarben 256 Farben, 16*16 Paletten oder 32768 Farben
- Objektfarben 256 Farben, 16*16 Paletten
- Effekte Rotation/Skalierung, Alpha-Blending, Fade-in/Out, Mosaic, Window
- Objektgrößen 12 Möglichkeiten (von 8*8 bis zu 64*64 Pixel)
- Objekte pro Bild maximal 128
- Objekte pro Linie maximal 128 mit 8*8 Pixel
- Prioritäten OBJ/OBJ: 0-127, OBJ/BG: 0-3, BG/BG: 0-3
- Effekte Rotation/Scaling, Alphablending, Fade-in/out, Mosaic, Window

3.1.4 Sound

- Analog 4 Kanal GameBoyColor© kompatibel
- Digital 2 DMA Sound Kanäle
- Ausgabe eingebauter Lautsprecher oder Kopfhörer

3.1.5 Steuerung

- Gamepad Vierwege-Richtungstasten (Abbildung 3-1: A), 6 Knöpfe (Abbildung 3-1: B, C, D, E)

3.1.6 Kommunikationsanschlüsse

- Serieller Port (256KBit oder 2Mbit pro Sekunde) (Abbildung 3-2: F),

3.1.7 Externer Speicher

- GBA© bis 32 MByte ROM (Abbildung 3-2: G)
inkl. SRAM,Flash oder EEPROM
- GameBoy© bis 32 KByte ROM inkl. 8 KByte SRAM

3.1.8 Stromversorgung

- Batterien/Akkumulatoren 2.4 bis 3 Volt



Abbildung 3-1 GameBoyAdvance©



Abbildung 3-2 GameBoyAdvance (von oben)

3.2 Speichermap

Der ARM7tdmi bietet einen 32 Bit breiten Adressraum. In diesen Adressraum liegen alle Arten von Speichern, sowie alle Hardwareregister. Je nach Typ des Speichers, ist nur ein Lesen oder Schreiben von Code/Daten bzw. Ausführen von Code möglich. In der Abbildung Abbildung 3-2 werden alle Speichertypen inkl. ihrer Größe, sortiert nach ihren Anfangsadressen dargestellt.

Adresse	Type	Größe
\$00000000	System ROM (BIOS)	16kByte
\$02000000	Work RAM	256kByte
\$03000000	Internal Work RAM	32kByte
\$04000000	IO RAM	1kByte
\$05000000	Palette	1kByte
\$06000000	Video RAM	64+32kByte
\$07000000	OAM	1kByte
\$08000000	ROM	Bis 32MByte
\$0A000000	ROM Image1	
\$0C000000	ROM Image2	
\$0e000000	Cart Ram	Bis 64kByte

Abbildung 3-2 Speichermap

3.3 Registerübersicht

Der GBA spiegelt alle für den Benutzer zugänglichen Register in den Speicher (DMA). Dieser besondere Speicherbereich nennt sich I/O RAM und beginnt ab Adresse \$40000000. In der folgenden Tabelle werden alle Register nach Adressen sortiert dargestellt. Um eine genauere

Beschreibung der Register zu erhalten, ist das Online-Guide GBATEK⁷ von Martin Korthoder oder die „CowBite Virtual Hardware Specifications“⁸ von Tom Happ zu empfehlen.

Adresse	Zugriff	Name	Beschreibung
0	R/W	DISPCNT	LCD - Kontrolle
2	R/W		Grün-Swap
4	R/W	DISPSTAT	LCD - Status Kontrolle
6	R	VCOUNT	vertikaler Zähler
8	R/W	BG0CNT	BG0 Kontrolle
A	R/W	BG1CNT	BG1 Kontrolle
C	R/W	BG2CNT	BG2 Kontrolle
E	R/W	BG3CNT	BG3 Kontrolle
10	W	BG0HOFS	BG0 X-Offset
12	W	BG0VOFS	BG0 Y-Offset
14	W	BG1HOFS	BG1 X-Offset
16	W	BG1VOFS	BG1 Y-Offset
18	W	BG2HOFS	BG2 X-Offset
1A	W	BG2VOFS	BG2 Y-Offset
1C	W	BG3HOFS	BG3 X-Offset
1E	W	BG3VOFS	BG3 Y-Offset
20	W	BG2PA	BG2 Rotations/Skalierungs Parameter A
22	W	BG2PB	BG2 Rotations/Skalierungs Parameter B
24	W	BG2PC	BG2 Rotations/Skalierungs Parameter C
26	W	BG2PD	BG2 Rotations/Skalierungs Parameter D
28-2A	W	BG2X	BG2 Referenzpunkt X-Koordinaten
2C-2E	W	BG2Y	BG2 Referenzpunkt Y-Koordinaten
30	W	BG3PA	BG3 Rotations/Skalierungs Parameter A
32	W	BG3PB	BG3 Rotations/Skalierungs Parameter B
34	W	BG3PC	BG3 Rotations/Skalierungs Parameter C
36	W	BG3PD	BG3 Rotations/Skalierungs Parameter D
38-3A	W	BG3X	BG3 Referenzpunkt X-Koordinaten
3C-3E	W	BG3Y	BG3 Referenzpunkt Y-Koordinaten
40	W	WIN0H	Fenster 0 horizontal
42	W	WIN1H	Fenster 1 vertikal
44	W	WIN0V	Fenster 0 horizontal
46	W	WIN1V	Fenster 1 vertikal
48	R/W	WININ	Kontrolle inside
4A	R/W	WINOUT	Kontrolle outside
4C	W	MOSAIC	Mosaik Effekt
50	R/W	BLDCNT	Farbeffekt Kontrolle
52	W	BLDAPLPHA	Alphablending
54	W	BLDY	Helligkeit/Koeffizienten
60	R/W	SOUND1CNT_L	Kanal 1 Sweep
62	R/W	SOUND1CNT_H	Kanal 1 Duty/Length/Envelope
64	R/W	SOUND1CNT_X	Kanal 1 Frequenz/Kontrolle
68	R/W	SOUND2CNT_L	Kanal 2 Duty/Length/Envelope
6C	R/W	SOUND2CNT_H	Kanal 2 Frequenz/Kontrolle

⁷ URL: www.work.de/nocash/gbatek.htm (27.07.2004)

⁸ URL: www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm (27.07.2004)

70	R/W	SOUND3CNT_L	Kanal 3 Stop/Waveram Auswahl
72	R/W	SOUND3CNT_H	Kanal 3 Länge/Volume
74	R/W	SOUND3CNT_X	Kanal 3 Frequenz/Kontrolle
78	R/W	SOUND4CNT_L	Kanal 4 Length/Envelope
7C	R/W	SOUND4CNT_H	Kanal 4 Frequenz/Kontrolle
80	R/W	SOUNDCNT_L	Kontrolle Stereo/Volume/Enable
82	R/W	SOUNDCNT_H	Kontrolle Mixing/DMA Kontrolle
84	R/W	SOUNDCNT_X	Kontrolle Sound an/aus
88	BIOS	SOUNDBIAS	Sound PWM Kontrolle
90-9E	R/W	WAVE_RAM	Kanal 3 Wavedaten
A0-A2	W	FIFO_A	Kanal A FIFO
A4-A6	W	FIFO_B	Kanal B FIFO
B0-B2	W	DMA0SAD	Quelladresse
B4-B6	W	DMA0DAD	Zieladresse
B8	W	DMA0CNT_L	Datenzähler
BA	R/W	DMA0CNT_H	Kontrolle
BC-BE	W	DMA1SAD	Quelladresse
C0-C2	W	DMA1DAD	Zieladresse
C4	W	DMA1CNT_L	Datenzähler
C6	R/W	DMA1CNT_H	Kontrolle
C8-CA	W	DMA2SAD	Quelladresse
CC-CE	W	DMA2DAD	Zieladresse
D0	W	DMA2CNT_L	Datenzähler
D2	R/W	DMA2CNT_H	Kontrolle
D4-D6	W	DMA3SAD	Quelladresse
D8-DA	W	DMA3DAD	Zieladresse
DC	W	DMA3CNT_L	Datenzähler
DE	R/W	DMA3CNT_H	Kontrolle
100	R/W	TM0CNT_L	Timer 0 Zähler/Reload
102	R/W	TM0CNT_H	Timer 0 Kontrolle
104	R/W	TM1CNT_L	Timer 1 Zähler/Reload
106	R/W	TM1CNT_H	Timer 1 Kontrolle
108	R/W	TM2CNT_L	Timer 2 Zähler/Reload
10A	R/W	TM2CNT_H	Timer 2 Kontrolle
10C	R/W	TM3CNT_L	Timer 3 Zähler/Reload
10E	R/W	TM3CNT_H	Timer 3 Kontrolle
120	R/W	SIODATA32_L/SIOMULTI0	SIO Data (bei 32) oder Data 0
122	R/W	SIODATA32_H/SIOMULTI1	SIO Data (bei 32) oder Data 1
124	R/W	SIOMULTI2	SIO Data 2
126	R/W	SIOMULTI3	SIO Data 3
128	R/W	SIOCNT	SIO Kontrolle
12A	R/W	SIOMLT_SEND/SIODATA8	SIO Data oder SIO Data 8Bit
130	R	KEYINPUT	Tastenstatus
132	R/W	KEYCNT	Tasteninterruptkontrolle
134	R/W	RCNT	SIO Mode
140	R/W	JOYCNT	SIO JOY-Bus Kontrolle
150-152	R/W	JOY_RECV	Receive Data
154-156	R/W	JOY_TRANS	Transmit Data
158	R/W	JOYSTAT	Receive Status
200	R/W	IE	Interrupt Enable
202	R/W	IF	Interrupt Request Flags/ IRQ Ack.
204	R/W	WAITCNT	Waitstat Kontrolle

208	R/W	IME	Interrupt Master Enable
300	R/W	HALTCNT	Power Down Kontrolle
800-802	R/W	UNKNOWN	Internal Memory Kontrolle

Abbildung 3-3 Registermap

3.4 Videokontroller

3.4.1 Grafikdarstellung

Der GBA kann Grafiken in unterschiedlichen Verfahren darstellen. Die üblichen auch aus dem PC bereich bekannten Formate sind die Bitmapdarstellung und die Echtfarbendarstellung.

Im Bitmapformat besitzt jedes Pixel einen Index auf eine Farbtabelle, somit kann ein Bild nur eine bestimmte Menge von Farben darstellen. Der GBA stellt diesen Modus zur Verfügung, mit den Eigenschaften, dass die Farbtabelle 256 Einträge erlaubt (mit je einem 15 Bit RGB Wert) und somit jedes Pixel eine Größe von einem Byte hat. Die Auflösung ist auf 240*160 Pixel beschränkt. Zusätzlich existieren zwei Puffer, wobei ein Puffer immer auf den Bildschirm dargestellt wird und der andere nicht sichtbar ist. Das Umschalten der Puffer ist mit setzen/löschen eines Flags möglich.

Im Echtfarbenformat enthält jedes Pixel alle benötigten Farbinformation und setzt sich aus den Grundfarben rot, grün und blau zusammen. Da der GBA auf eine 15 Bit Farbausgabe (siehe Abbildung 3-4) beschränkt ist, wird hier ein Pixel als 16 Bit Wert gespeichert.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
	B	B	B	B	B	G	G	G	G	G	R	R	R	R	R

Abbildung 3-4 Farbaufteilung

Für die Bildschirmauflösung kann zwischen 240*160 und 160*120 Pixel gewählt werden, wobei die zweite Auflösung ein Doppeltes Puffern erlaubt.

Diese Typen von Grafikdarstellungen sind für viele Spiele nicht geeignet, da sie viel Speicher verbrauchen (240*160 mit 32768 Farben = 76800 Byte) und auch ein hohes Maß an Rechenzeit bei der Bearbeitung benötigen.

Um diese Probleme zu beseitigen, gibt es den für „ältere“ Spielkonsolen üblichen Teilemodus. Im PC-Bereich ist er noch annähernd vergleichbar mit dem reinen Textmodus. Die Darstellung des Bildes im Speicher ist nicht mehr direkt Pixelweise, sondern Teilweise. D.h. der Bildschirm wird in 30*20 Teilen mit einer Größe von 8*8 Pixel aufgeteilt (bei 240*160 Bildpunkten Auflösung).

Jedes dieser Teile hat eine Referenz auf ein 8*8 Pixel großes Grafikeil im Grafikspeicher (siehe Abbildung 3-5).

Durch dieses Verfahren ergeben sich einige Vorteile, zum Beispiel ist Redundanz erzielbar indem man Grafiken, die gleich sind (z.B. Hintergrund), durch ein Teil mehrfach ausdrücken kann. Somit schrumpft der Speicherverbrauch eines Bildschirms auf nur noch 1/64 des dargestellten

Bereiches. Weiterhin ist eine Animation von Teilen einfach durch eine Änderung des Index möglich.

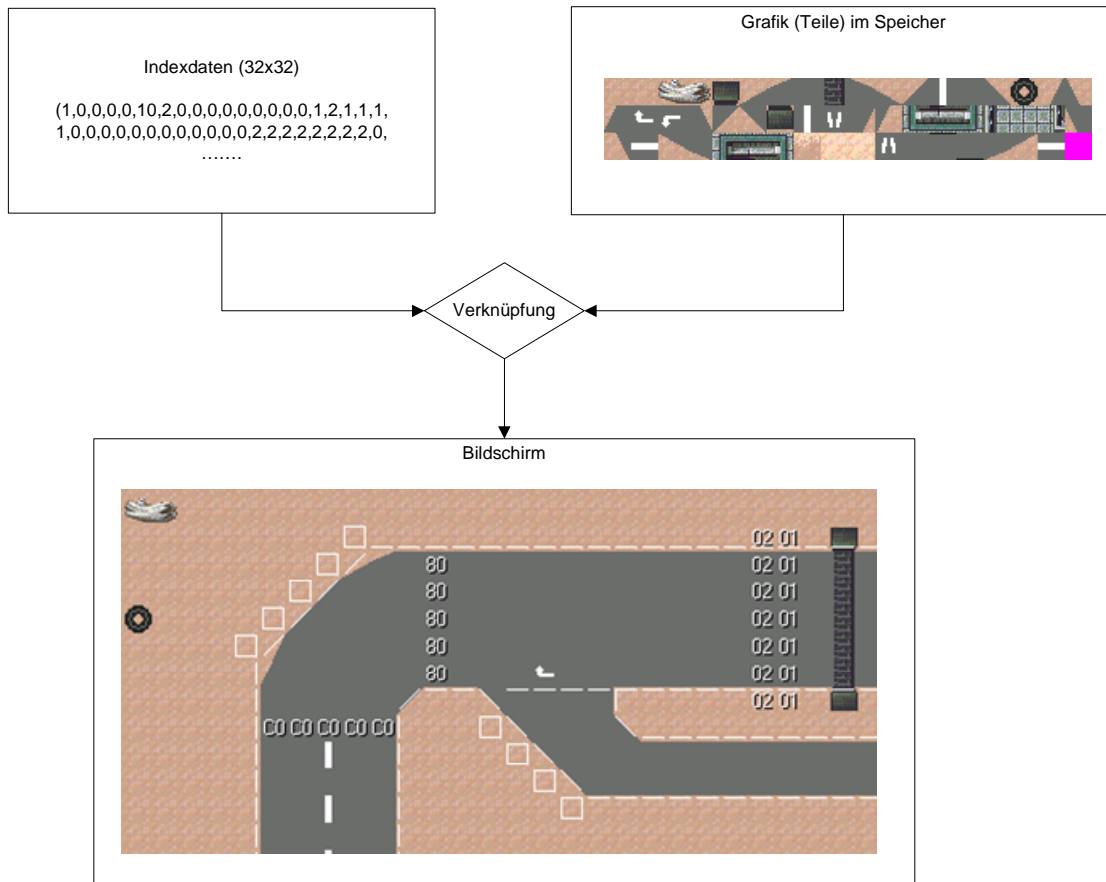


Abbildung 3-5 Tilemodus

Der Teileindex kann je nach Modus unterschiedlich Größen haben (32*32, 32*64, 64*32,...). Die Teilegrafiken können in der Anzahl entweder 256 oder 65536 betragen, wobei zu beachten ist, dass der Teileindex und die Teilegrafiken in einen Speicherbereich der Größe von 64 kByte Speicher Platz finden müssen.

Ein weiter Vorteil ist, dass mehrer solcher Bildschirme übereinander gelegt werden können, wobei der Palettenindex 0, immer eine Transparenz charakterisiert und dadurch alle unter diesem liegende Schichten dargestellt werden. Zusätzlich, kann eine solche Schicht auch rotiert und vergrößert werden.

3.4.2 Grafikmodies

Der GBA bietet dem Benutzer 6 verschiedene Grafikmodi an:

Modus 0:

Es können bis zu 4 Teile-Layers übereinander liegen. Rotation/Zoom ist nicht möglich.

Modus 1:

Es können bis zu 3 Teile-Layer dargestellt werden, wobei einer davon rotier und skalierbar ist.

Modus 2:

Es können bis zu 2 Teile-Layer dargestellt werden, wovon beide rotierbar und skalierbar sind.

Modus 3:

Einfacher 15 Bit RGB Modus (240*160 Pixel), da mehr als 64 KByte verbraucht werden 11 KByte vom Sprite-Speicher benutzt.

Modus 4:

8-Bit Bitmap Modus, doppelt gepuffert. Auflösung 240*160 Pixel.

Modus 5:

Einfacher 16 Bit RGB Modus (160*120 Pixel), doppelt gepuffert.

Die Teilemodi haben immer eine Auflösung von 240*160 Pixel, die Farbanzahl der Teile ist umschaltbar zwischen einer 256'er oder einer 16'er Farbpalette. Bei der 16'er Farbpalette ergibt sich bei den Teilegrafiken dadurch immer ein Farbwert von 4 Bit.

Ein Nachteil bei der Wahl von rotier- und skalierbaren Layern ist, dass der Teileindex nur 8 Bit groß ist und somit nur 256 Teile adressiert werden können. In den normalen Modus ist eine 16 Bit Adressierung möglich.

Da alle Teilemodi, je nach Konfiguration einen größeren Bildbereich besitzen als der auf dem Bildschirm darstellbare, es ist möglich, den Grafikbereich von 0 bis 511 Pixel horizontal/vertikal zu bewegen.

3.4.3 Sprites

In fast allen Spielkonsolen (die auf 2D-Grafik ausgelegt sind) existiert die Möglichkeit, Sprites zu benutzen. Sprites sind einfach Grafikobjekte die auf dem Bildschirm dargestellt werden. Ihr Vorteil ist es, dass sie andere Objekte (z.B. Hintergrund) nicht überschreiben. D.h. es ist nicht notwendig, Grafikbereiche zu sichern oder auszumaskieren. Ein populäres Beispiel hierfür ist der Mauszeiger.

Der GBA bietet dem Benutzer eine sehr große und flexible Art Sprites zu verwenden. Es ist möglich bis zu 128 Sprites darzustellen.

Die Größe eines Sprites kann aus folgenden Dimensionen gewählt werden.

8x8	16x8	8x16
16x16	32x8	8x32
32x32	32x16	16x32
64x64	64x32	32x64

Abbildung 3-6 Spritedimensionen

Zusätzlich können die Sprites in ihrer Ausgabepriorität verändert werden und auch transparent zu verschiedenen Hintergründen gezeichnet werden.

Eine Hardwarekollision zwischen Sprites - Sprites und zwischen Sprites - Hintergründen bietet der GBA nicht an. Eine Ersatzlösung bietet mein Framework an (siehe Kapitel 8).

3.5 Soundcontroller

Alle Eigenschaften des GBA Soundcontrollers sind ausführlich im Kapitel 6 (Musikprogrammierung) beschrieben.

3.6 Timer

Der GBA besitzt vier Timer mit einer Auflösung von je 16 Bit. Sie zählen abwärts und können je nach Konfiguration einen Interrupt auslösen oder eine DMA Operation bewirken. Außerdem kann ihre Frequenz gewählt werden. Die Standardfrequenz ist die Systemfrequenz (16777216 Hz). Eine Skalierung von 1/1, 1/64, 1/256 oder 1/1024 ist wählbar.

3.7 DMA Transfer

Der GBA besitzt vier voneinander getrennte DMA-Kanäle. Ihr Ablauf ist nach Prioritäten geordnet. Die CPU ist bei aktivem DMA-Transfer im Pause-Zustand, d.h. es ist nicht wie bei anderen Systemen (z.B. AMIGA) möglich, ein simultanes Arbeiten von DMA und CPU zu ermöglichen.

Bedingt durch ihre Prioritäten zueinander sind die DMA-Kanäle wie folgt zu verwenden:

- DMA 0 für zeitkritische Angelegenheiten (z.B. HBlank-Transfers)
- DMA 1 und DMA 2 zum kopieren von Sample-Daten in die Sound-FIFO's
- DMA 3 Standard DMA, für allgemeine Anwendungen

Zusätzlich ist es möglich, die Quelle und das Ziel je nach Transfer um 16 oder 32 Bit zu inkrementieren, zu dekrementieren oder beizubehalten. Um Zeitverzögerungen (durch IRQ Overhead) zu vermeiden, kann zwischen einer Benachrichtigung nach einem Transfer oder der Benachrichtigung durch einen IRQ, noch zusätzlich eine Autorepeat gewählt werden. D.h. nach dem Beenden des Transferauftrags, beginnt die gesamte Operation von neuem.

3.8 Tastatureingabe

Der GBA bietet insgesamt sechs Tasten (A, B, L, R, Start und Select) und ein Vierwege-Steuerkreuz. Alle Zustände können über einen Register (REG_KEY (\$4000130)) abgefragt werden. Weiterhin ist es möglich mit einer Und/Oder-Verknüpfung auf die Tastenzustände eine IRQ auszulösen.

3.9 Interruptkontrolle

Da alle Interruptvektoren im BIOS liegen, ist bei Softwareinterrupts keine Abarbeitung durch den Benutzer möglich. Alle normalen Interrupts (IRQ) werden nach ihrer Aktivierung, auf eine vom Benutzer bestimmbare Adresse verzweigt und an dieser Stelle kann der ausgelöste Interrupt bearbeitet werden.

Die globale Adresse für den vom Benutzer festlegbaren Interrupt-Handler ist `$03007ffc`. Alle Adressen im Bereich `$03007xxx` werden auf den Adress-Bereich `$03fffxxx` gespiegelt. Der Grund für diese Maßnahme, liegt in der schnellen Adressbildung durch den Programmierer. Z.B. lässt sich die Adresse `$03007ffc` durch Adress-Spiegelung mit zwei Instruktionen (ohne Lesezugriff) setzen. Abbildung 3-7, verdeutlicht drei Möglichkeiten zur Bildung der Adresse `$03007ffc`. Wobei im ersten Beispiel die Adress-Spiegelung genutzt wird.

```
mov    r0, #0x4000000
str     r1, [r0, #-4]

;      oder

ldr     r0, =0x03007ffc    ;lade aus Pool
str     r1, [r0]

;      oder

mov     r0, #0x3000000
orr     r0, r0, #0x7f00
orr     r0, r0, #0x00fc
str     r1, [r0]
```

Abbildung 3-7 schnelle Adressbildung

Diese Verfahren hat durchaus seine Berechtigung, besonders da sich alle CPU-Externen Interrupts einen Vektor teilen und deshalb eine schnelle Verarbeitung notwendig ist. Abbildung 3-7 zeigt den vom Interrupt-Vektor (IRQ) angesprungenen Code im BIOS. Neben der Adressbildung und dem Sprung zum vom Benutzer festgelegten Code, werden noch zusätzlich die Register R0 bis R3 und R12 gesichert. Der Linkregister und der Programmcounter sind durch den Interrupt selbst gesichert (siehe Kapitel 2.7 (Exceptions)). Unklar ist warum der Interrupt-Vektor (für IRQ) nicht gleich auf den Wert `$03007ffc` gesetzt wurde.

```
stmfd   sp!, r0-r3, r12, lr
mov     r0, #0x4000000
add     lr, pc, #0
ldr     pc, [r0, #-4]
ldmfd   sp!, r0-r3, r12, lr
subs    pc, lr, #4
```

Abbildung 3-8 BIOS IRQ-Behandlung

Je nach dem vom Benutzer erlaubten Interrupt und dessen Auftreten, wird die vom Benutzer gesetzte Adresse angesprungen. Der erste Schritt ist das Auslesen des erhaltenen Interrupts und das bestätigen dieses (`RE_IF = REG_IE`).

Folgende Interrupt-Quellen sind aktivierbar:

- LCD V-Blank
- LCD H-Blank
- LCD V-Counter Match
- Timer 0 Overflow
- Timer 1 Overflow
- Timer 2 Overflow
- Timer 3 Overflow
- Serial Communication
- DMA 0
- DMA 1
- DMA 2
- DMA 3
- Keypad
- Game Pak (ROM)

Die Interrupts für die Timer 0 bis 3 werden bei einem Überlauf aktiviert (nur wenn ein Interrupt ausgelöst werden soll). Bei DMA 0 bis 3 wird, wenn aktiviert, ein Interrupt ausgelöst. Für die Tasten ist bei Betätigung, ein Interrupt auslösbar. Der Game-Pak Interrupt ist eine Besonderheit, er wird ausgelöst, wenn die Übertragung zum externen ROM fehlschlägt (z.B. das Herausziehen des Spielmoduls). Aus diesem Grund sollte der vom Benutzer geschriebene Interrupt-Handler nicht im ROM liegen, da bei Ausfall dieses, keine weitere Verarbeitung gewährleistet ist.

Von großer Bedeutung und mit Abstand am häufigsten genutzte Interrupts sind die H-Blank, V-Blank und V-Counter Interrupts. Der V-Blank löst, wenn aktiviert, ab der LCD-Zeile 160, einen Interrupt aus. Dies geschieht ca. 60-mal pro Sekunde. (In vielen anderen Systemen bedeutet H-Blank, Rasterzeile Null.) Durch diese Funktionalität ist eine Synchronisierung der Software mit dem Bildaufbau möglich.

Der H-Blank arbeitet horizontal und wird nach dem Zeichnen des letzten sichtbaren horizontalen Pixels (240) ausgeführt. Der Y-Trigger kann eine frei definierbare horizontale Position abfragen und dann einen Interrupt auslösen. Abbildung 3-9 visualisiert dieses Problem.

Der vertikale nicht sichtbare Bereich (virtuell) beträgt 68 Pixel und wird in 4.99 ms ausgeführt (bis zu Rasterzeile 228), somit stehen 83776 Taktzyklen zur Verfügung. Der horizontale beträgt ebenfalls 68 Pixel und hat eine Dauer von 16.212 μ s (272 Taktzyklen).

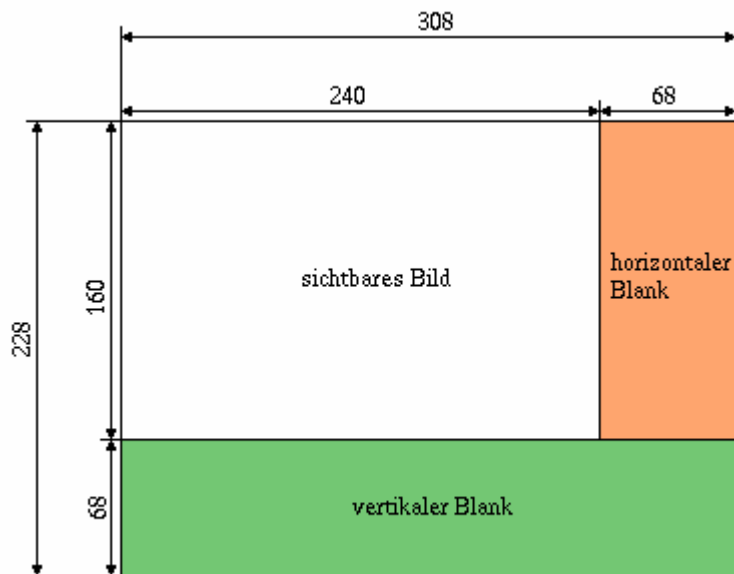


Abbildung 3-9 LCD

Diese kritischen Zeiten, besonders beim H-Blank, verdeutlichen die zuerst dargestellte Problematik mit der möglichst schnellen Auswertung eines Interrupt-Requests.

In Kapitel 5.2.6 wird ein von meinem Framework angebotener, Interrupt-Handler beschrieben.

3.10 BIOS

Das GBA BIOS bietet dem Benutzer mehrerer Funktionen an. Der Aufruf einer solchen Funktion wird über einen Software-Interrupt ausgelöst.

Folgende Grundfunktionen werden vom BIOS angeboten:

- arithmetische Funktionen
- Rotation und Skalierung Funktionen
- Dekomprimierungs-Funktionen
- Speicher Funktionen
- Halt Funktionen
- Reset Funktionen
- Multi Boot Funktionen
- Sound Funktionen

3.11 Fazit

Die GBA-Hardware ist hervorragend ausgestattet für die Spielentwicklung, nur der Sound-Chip und die DMA-Kanäle lassen in ihrer Ausführung zu Wünschen übrig.

4 Entwicklerwerkzeuge

Da Nintendo nur offiziellen Entwicklern Programme zu Entwicklung von GBA-Software zur Verfügung stellt, müssen alle nicht offiziellen Entwickler auf freie Programme zurückgreifen. In den folgenden Kapiteln werde ich die Grundlegenden Programme beschreiben.

4.1 Software

4.1.1 Emulatoren

Eine sehr wichtige Komponente des Softwaretests sind die Emulatoren. Ihre Hauptaufgabe ist es, die Hardware des Zielgerätes auf einem anderen System zu Simulieren. Für den GBA stehen mehrer Emulatoren zur Verfügung. Ich verwende für meine Softwaretests BatGBA⁹ und VisualBoyAdvance¹⁰ (VBA).

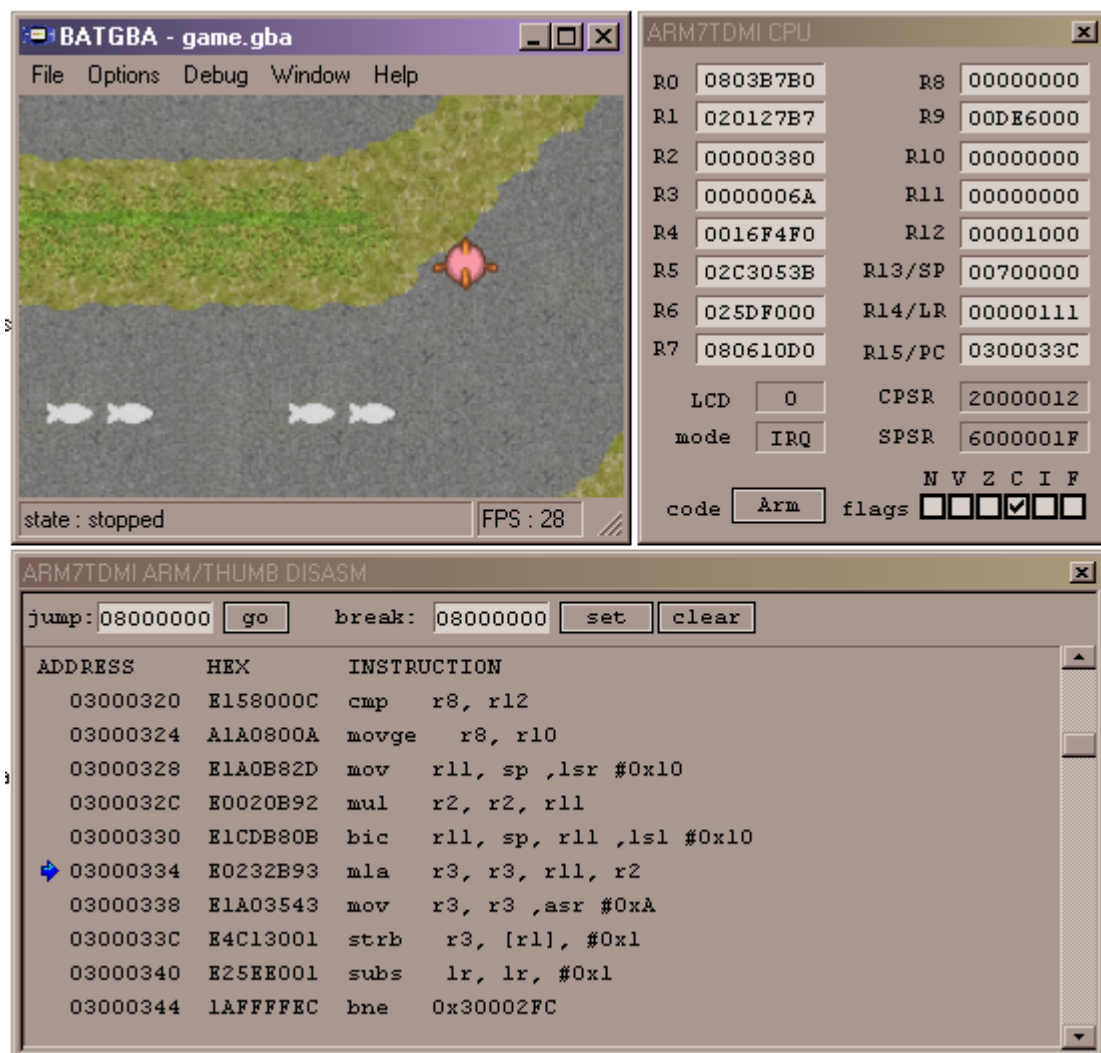


Abbildung 4-1 BatGBA

⁹ URL: <http://batgba.zophar.net/> (27.07.2004)

¹⁰ URL: <http://vba.ngemu.com/> (27.07.2004)

Beide Emulatoren bieten den Benutzer weitere Funktionen an. Unter anderem ist das Anzeigen des Codes, der Register und Speicherbereiche zu erwähnen. In Abbildung 4-1 ist BatGBA mit offenem Disassembler und Registersatz dargestellt.

Wenn die Genauigkeit der Emulation betrachtet wird, befinden sich beide Programme sehr nah an der wirklichen Hardware. Leider ist VBA nicht Zyklengenau bei der Abarbeitung von CPU-Befehlen. In fast allen Fällen arbeitet er schneller. Da aber fast alle Spiele über den Vertikalen-Blank synchronisiert werden, stellt dieses Fehlverhalten kein Problem dar. Die größte Schwachstelle beider Emulatoren ist die nicht vorhandene Emulation aller Hardware-Register pro Takt. Dadurch lassen sich zum Beispiel Horizontale Farbänderungen in einen Horizontalen-Interrupt nicht darstellen.

4.1.2 Entwicklungs-Tools

Als Entwicklungskid benutze ich den Cross-Compiler devkitARM¹¹ für Windows und Linux Betriebssysteme. Er enthält die wichtigsten GNU-Entwickler-Tools, darunter fallen z.b:

- GCC (C++ Compiler)
- GAS (Assembler)
- LD (Linker)
- OBJCOPY (Objekt-Kopie mit Erweiterung)

Da Windows eine sehr mangelhafte Shell (DOS) zur Verfügung stellt, habe ich den Shell-Ersatz Cywin¹² benutzt. Mit Cygwin erhält man eine komplette LINUX-Shell für Windows-Systeme.

```

/cygdrive/d/GameBoyAdvance/gbadev_as/ss2/source/tests/tunnel
/cygdrive/d/GameBoyAdvance/gbadev_as/ss2/source/tests/tunnel
$ make
../armcc/bin/arm-agb-elf-ld.exe tunnel_mode0.o      ../stdfkt/si
mple.o --script ../armcc/system/lnkscrip -L ../armcc/lib/gcc
-lib/arm-agb-elf/3.3.2/interwork/ -o tunnel_mode0.elf
../armcc/bin/arm-agb-elf-objcopy.exe -v -O binary tunnel_mode0.elf tunn
el_mode0.gba
copy from tunnel_mode0.elf(elf32-littlearm) to tunnel_mode0.gba(binary)
../armcc/bin/arm-agb-elf-readelf.exe -l tunnel_mode0.elf

Elf file type is EXEC (Executable file)
Entry point 0x80000000
There are 6 program headers, starting at offset 52

Program Headers:
Type           Offset    VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
LOAD           0x000000  0x02ff8000  0x02ff8000  0x080b4 0x081b4  RW  0x8000
LOAD           0x010000  0x08000000  0x08000000  0x34e38 0x34e38  R E  0x8000
LOAD           0x048000  0x03000000  0x08034e38  0x000b4 0x000b4  R E  0x8000
LOAD           0x050000  0x02000000  0x08034eec  0x00410 0x00410  R E  0x8000
LOAD           0x0581b4 0x030001b4 0x08034eec  0x00000 0x00000  RW  0x8000
LOAD           0x060000  0x02000000  0x080352fc  0x00020 0x00020  R E  0x8000

Section to Segment mapping:
Segment Sections...
00      .iwrpm .bss.1
01      .text
02      .iwrpm
03      .ewram .swram
04
05      .swram
$

```

Abbildung 4-2 Cygwin-Shell

¹¹ URL: <http://homepage.ntlworld.com/wintermute2002/> (27.07.2004)

¹² URL: www.cygwin.com (27.07.2004)

Ohne die Verwendung von Cygwin bzw. einer richtigen UNIX/Linux-Shell ist das Verarbeiten/Assemblieren meiner Sourcen nicht möglich.

Da der GNU-Assembler einen sehr an die Programmiersprache C angelegten Syntax hat und dies sehr ungewöhnlich bei Assemblern ist (im Vergleich zu M68k und 6502 Assembler), habe ich ein Skript¹³ entwickelt, welches einen assemblernäheren Syntax erlaubt.

Die wichtigsten Änderungen sind:

- Kommentare beginnen mit ; (Semikolon)
- hexadezimale Zahlen werden mit dem Sonderzeichen \$ (Dollar) symbolisiert
- binäre Zahlen werden mit dem Sonderzeichen % (Prozent) symbolisiert
- Labels benötigen keinen Doppelpunkt am Namensende
- weitgehende Syntaxkompatibilität zu den Assembler goldroad¹⁴ (Freeware Assembler)

Die Codeerzeugung erfolgt wie üblich durch die Steuerung von Makefiles. Abbildung 4-3 zeigt ein von mir erstelltes Standard-Makefile zur Assemblierung von Files.

```
#####
ASC      = exec ../../armcc/asmconv.sh
AS       = ../../armcc/bin/arm-elf-as.exe -EL -mthumb-interwork
LD       = ../../armcc/bin/arm-elf-ld.exe
OC       = ../../armcc/bin/arm-elf-objcopy.exe
RE       = ../../armcc/bin/arm-elf-readelf.exe
CC       = ../../armcc/bin/arm-elf-gcc.exe

INC      = -I . -I ../../\
          -I ../source/sound/\
          -I ../source/mem/\
          -I ../source/kollision/\
          -I ../source/

LDLFLG   = --script ../../armcc/system/lnkscript\
          -L ../../armcc/lib/gcc-lib/arm-elf/3.4.0/interwork/\

CFLAGS   = -mthumb-interwork -mlong-calls -fverbose-asm -Wall -mcpu=arm7tdmi -O3

EOBJS    =
OBJ      =      fill_rowbyrow.o\
              circle.o\
              bresenham.o\
              spline.o\
              search_rek.o\
              irq_handler.o

all:     $(OBJ)

%.o:     %.s
          $(ASC) $< >temp.s
          $(AS) temp.s $(INC) -o $@

%.o:     %.c
          $(CC) -S $< $(CFLAGS) -o $@

clean:
          rm -rf $(OBJ)
```

Abbildung 4-3 Standard-Makefile

¹³ Quelle: armcc/asmconv.sh

¹⁴ URL: www.goldroad.co.uk (27.07.2004)

Die Besonderheiten der Speicherbereiche sind im Link-Skript¹⁵ definiert, darunter fallen die Aufteilungen der drei wichtigen Hauptspeichertypen in Sektionen. Die Sektionsnamen sind wie folgt aufgeschlüsselt.

- „text“ – Sektion ist der ROM (ab Adresse \$8000000)
- „iwrām“ – Sektion ist der IWRAM (ab Adresse \$3000000)
- „ewram“ – Sektion ist der EWRAM (ab Adresse \$2000000)
- „data“ – Sektion ist der ROM (ab Adresse \$8000000)
- „bss“ – Sektion (Platzhalter) ist der IWRAM (ab Adresse \$3000000)
- „swram“ – Sektion wie EWRAM für Single-Pack Programme erstellt

Der Startup-Code für C-Programme befindet sich im Normalfall in der Datei crt0.o. In den von mir verwendeten System wird auf diesen Startup-Code verzichtet, da er unter anderem einen ineffektiven Interrupt-Kontroller enthält. Aus diesem Grund müssen startbare Programme die Sektionen selbständig kopieren und löschen. Abbildung 4-4 zeigt eine Standard-Implementierung für diesen Sachverhalt.

```
;**** setze Stacks

mov    r0, #$12
mov    r1, #$1f
msr    cpsr, r0
ldr    sp,=myirqstackend
msr    cpsr, r1
ldr    sp,=mystackend

;**** lösche bss

ldr    r1,=__bss_start
ldr    r2,=__bss_end
sub    r2,r2,r1
bl     memclear32

;**** kopiere iwrām-Code/Daten

ldr    r0,=__iwrām_lma
ldr    r1,=__iwrām_start
ldr    r2,=__iwrām_end
sub    r2,r2,r1
bl     dmacpy

;**** kopiere ewram-Code/Daten

ldr    r0,=__ewram_lma
ldr    r1,=__ewram_start
ldr    r2,=__ewram_end
sub    r2,r2,r1
bl     dmacpy

;**** kopiere data-Sektion

ldr    r0,=__data_lma
ldr    r1,=__data_start
ldr    r2,=__data_end
sub    r2,r2,r1
bl     dmacpy
```

Abbildung 4-4 Startup

¹⁵ Quelle: armcc/system/lnkscrip

4.1.3 Editoren

Je nach Wunsch des Benutzers, kann jeder beliebige Editor genutzt werden. Die Verwendung von einer kompletten Entwicklerumgebung wäre auch denkbar. Ich benutzte den Editor Ultraedit¹⁶ der durch seine Vielfältigkeit ein angenehmes Arbeiten ermöglicht.

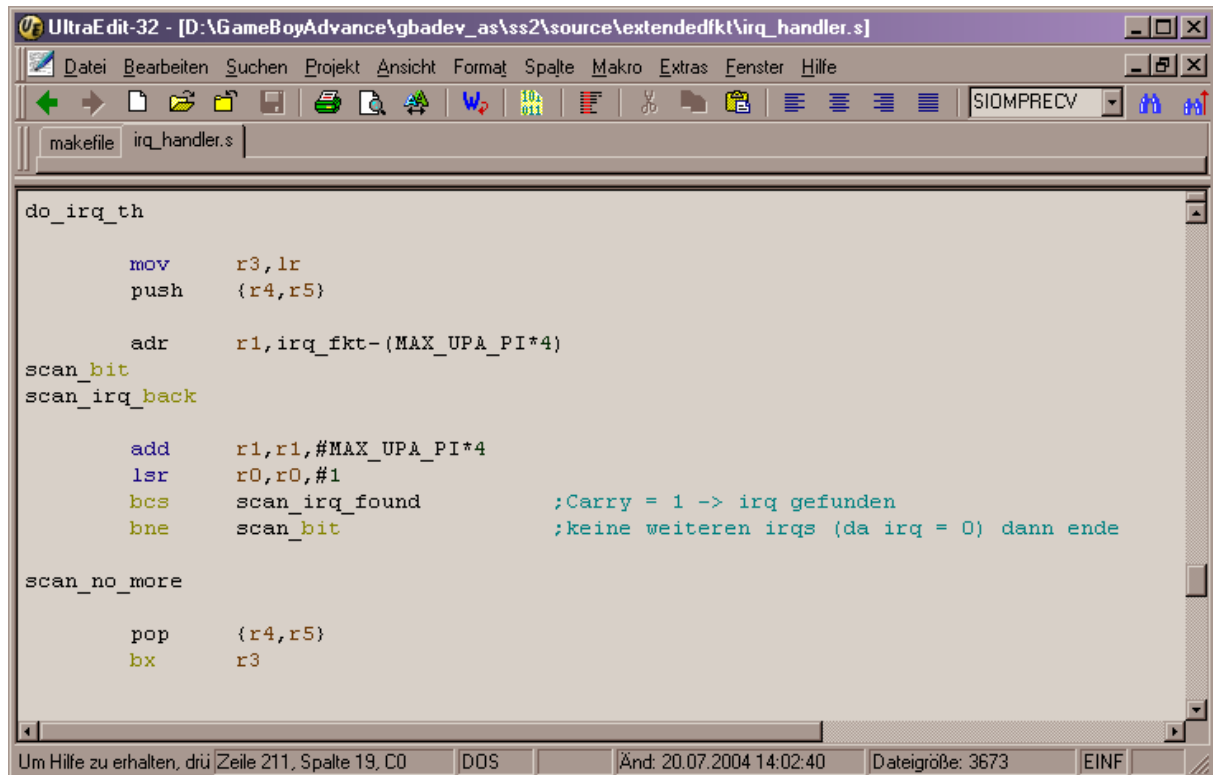


Abbildung 4-5 Ultraedit mit geladener Quell-Datei

4.1.4 Grafikprogramme

Je nach Anwendungsgebiet gibt es mehrer Arten von Programmen. Im Wesentlichen kann zwischen folgenden drei Arten unterschieden werden.

- Pixelzeichenprogramm
- Map (Mappen) - Editor
- Grafikkonverter

Das reine Pixelprogramm ist zum Zeichnen von Bitmapgrafiken verwendbar. Als besonders empfehlenswert ist PPaint¹⁷ (Amiga) und Ultimate Paint¹⁸ (PC) zu erwähnen. Der Name Pixelprogramm deutet schon auf die Art des Programms hin, denn es arbeitet im Gegensatz zu vielen Grafikprogrammen pixelgenau, hat festlegbare Palettengrenzen und spezielle Masken und

¹⁶ IDM Computer Solutions Inc. URL: <http://www.ultraedit.com> (27.07.2004)

¹⁷ Cloanto Inc. Download URL: <ftp://de.aminet.net/pub/aminet/biz/cloan/PPaint.lha> (27.07.2004)

¹⁸ J-T-L Development URL: <http://www.ultimatepaint.com/> (27.07.2004)

Pinzel-Operationen. In vielen Fällen steht noch ein Animationsystem zur Verfügung. In Abbildung 4-6 wird das Grafikprogramm PPaint dargestellt.

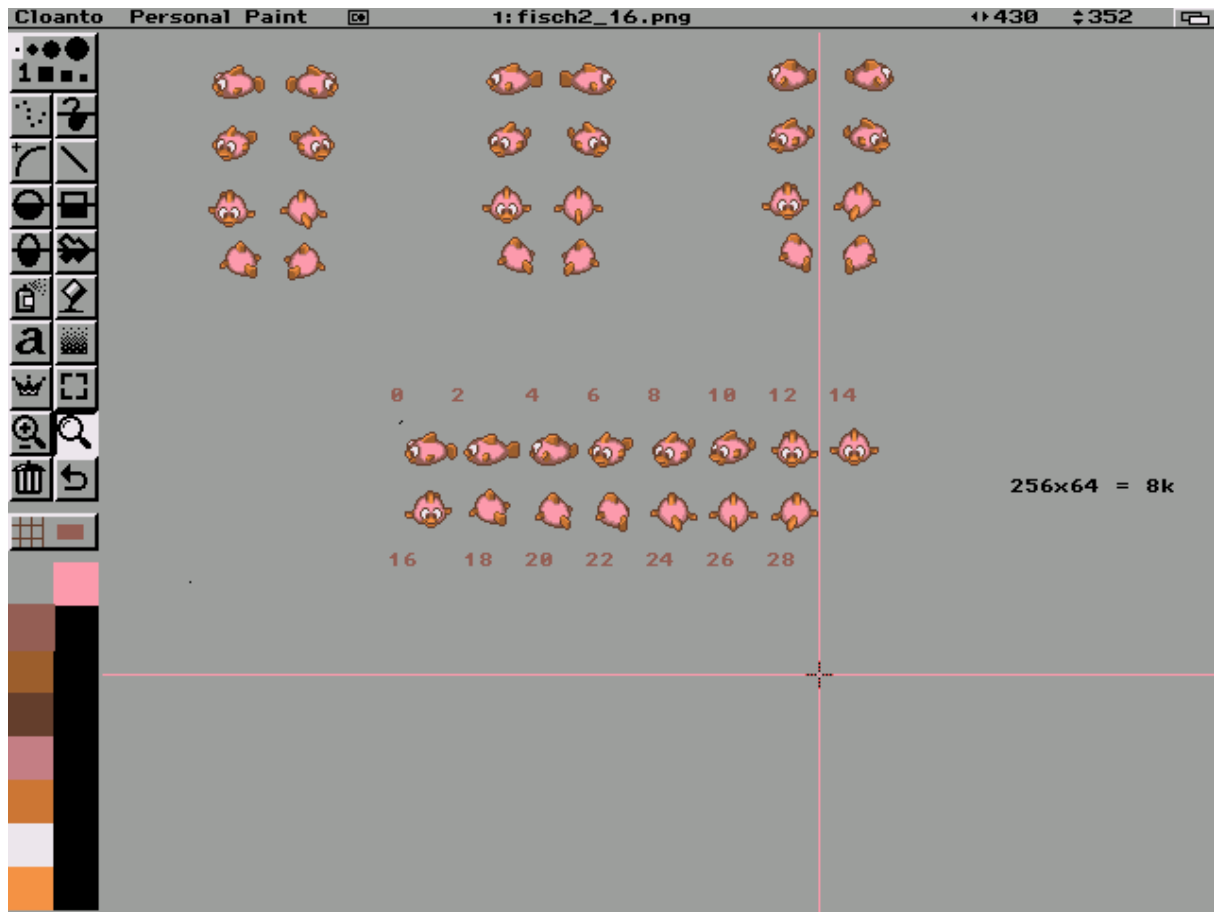


Abbildung 4-6 PPaint

Map-Editoren finden ihrer Anwendung im Design von Spiel-Leveln (Spiel-Welten), denn sie berücksichtigen das spezielle Teileformat des GBA, d.h. der Benutzer kann erstellte Teilegrafiken auf einer Mappe aufbringen und dadurch eine neue Grafik erzeugen. Es existieren eine Vielzahl an Map-Editoren, leider ist der Großteil auf bestimmte Einsatzszenarien festgelegt und deshalb zu unflexibel für die GBA Hardware. Ein benutzbarer Map-Editor ist Tilestudio¹⁹, siehe Abbildung 4-7.

¹⁹ Wiering Software (Open Source) URL: <http://tilestudio.sourceforge.net/> (27.07.2004)



Abbildung 4-7 Tilestudio mit geöffneter Map

Um die Grafikdaten in ein von der Software verwendbares Format zu transferieren, ist die Verwendung von Grafik-Konvertern unumgänglich. Zur allgemeinen Formatttransformation (z.B. das PNG-Format in das BMP-Format umwandeln) verwende ich XnView²⁰. Dieses Programm ist kostenlos erhältlich und unterstützt eine Vielzahl von Formaten. Zusätzlich kann auch ein Batchbetrieb erstellt werden, um mehrer Bildfolgen zu konvertieren (z.B. um Animationen zu konvertieren).

Um Grafikdaten in das GBA-Grafikformat übertragen zu können, verwende ich das Programm kaleid²¹. Seine eigentliche Funktion ist das Umwandeln von BMP-Bildern in rohe Daten. Es werden alle GBA Grafikmodis unterstützt und unter anderem ist eine automatische Erkennung der Palettendaten pro Teil möglich (bei 16*16 Paletten in einer 256 Palette).

²⁰ Freeware URL: <http://www.xnview.com> (27.07.2004)

²¹ URL: http://www.geocities.com/kaleid_gba/ (27.07.2004)

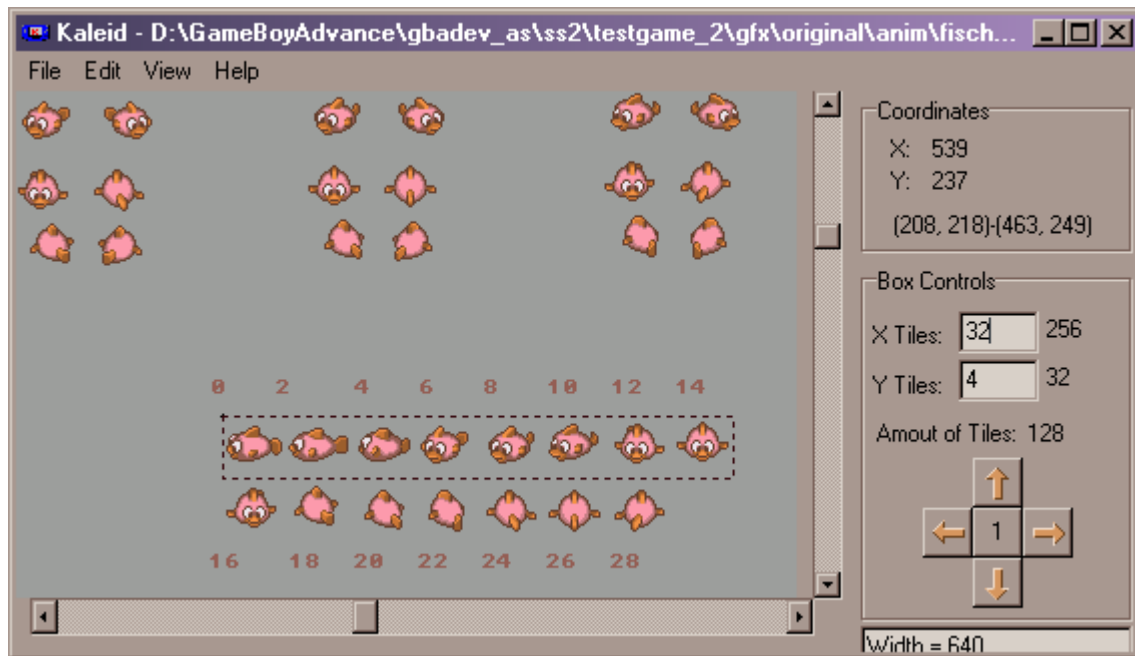


Abbildung 4-8 Kaleid mit geladenen Bild

Für die Konvertierung in für den Benutzer verwendbare Formate habe ich folgende Programme entwickelt.

4.1.4.1 Mapkonverter

Beim Arbeiten mit Map-Editoren ist es in vielen Fällen unerwünscht, ein Teileformat von 8*8 Pixel zu verwenden. In vielen Fällen ist ein Teileformat von 16*16 Pixel sinnvoller, da im Normalfall 8*8 Pixelobjekte zu klein zur Levelerstellung sind. Um eine Rücktransformation zu erreichen biete ich das Programm 16to8.exe (unter dem Pfad tools/) an. Die C-Quellen befinden sich im Verzeichnis tools_source/.

Name:	16to8.exe
Synopsis:	16to8.exe inputmap.map MAP-Breite GFX-Breite
Parameter:	MAP-Breite gibt die horizontale Breite der Map an (in 16 Pixel Teilen) GFX-Breite gibt die horizontale Breite eines Teils an (im Normalfall 16)
Beispiel:	./16to8.exe rallymap0.map 64 16 >rallymap0.s

4.1.4.2 Binär zu Quelltext

Eine Umwandlung von Binärdaten in ein von dem Assembler/Compiler lesbare Formate, wird von dem Programm bin2gold.exe (unter dem Pfad tools/) erreicht. Es bietet dem Benutzer zusätzlich folgende Funktionen an:

- 4 Byte „Little Endian“ zu „Big Endian“ Transformation
- Startoffset für Eingangsdatei
- Korrigieren der Horizontalen/Vertikalen-Flip Daten bei Tilestudio Projekten

Name:	bin2gold.exe
Synopsis:	bin2gold file [-flip] [-offset %%d] [-gbahv]
Parameter:	"flip" wandelt 32 Bit Daten von Big- zu Little Endian "offset" erwartet einen Dezimalwert, welcher den neuen Offset für die Daten angibt "gbahv" ändert H/V-Flipbits von Tilestudio auf GBA-Hardware-H/V-Bits
Beispiel:	./bin2gold.exe picture1.gfx >picture1.s

4.1.4.3 Texte zu binär

In einigen Fällen kann es dazu kommen, dass die Quelldaten nicht in einem Binärformat vorliegen (z.B. Tilestudio Mapcodes). Das Tool txt2bin.exe (unter dem Pfad tools/) wandelt Reihen von Dezimalzahlen in Binärdaten byteweise um.

Name:	txt2bin.exe
Synopsis:	txt2bin.exe inputfile outputfile
Beispiel:	./txt2bin.exe mapcodes.bb >mapcodes.raw

4.1.5 Musik und Sound

Bei der Analyse von MOD-Dateien verwende ich Mad-Tracker²². Da dieser aber keinen MOD-Export erlaubt, verwende ich zum MOD-Editieren den originalen ProTracker²³ auf dem Amiga.



Abbildung 4-9 ProTracker 4.0b

4.2 Hardware

Um die Programme auf der Hardware zu testen, ist es notwendig die Programme auf ein beschreibbares Medium zu übertragen, das kompatibel mit dem Hardwaretarget ist. Für den GBA existieren Flashkarten, mit welchen ein Ausführen der erstellten Software möglich ist.

4.2.1.1 Flasher

Das Beschreiben der Flashkarten geschieht je nach verwendetem System mit einer externen Einheit oder dem direkten Anschluss an den GBA. Das von mir verwendete System hat eine

²² URL: www.madtracker.org (27.07.2004)

²³ Download URL: ftp://de.aminet.net/pub/aminet/mus/edit/PT4_Beta2.lha (27.07.2004)

externe Flascheinheit (Abbildung 4-10 Punkt A). Der Anschluss zur PC-Seite erfolgt über einen USB-Anschluss (Abbildung 4-10 Punkt C) oder den Parallel-Port (Abbildung 4-10 Punkt D). Zum Beschreiben einer Flashkarte wird diese in das Flashgerät eingesetzt (Abbildung 4-10 Punkt B). Die Programmierung erfolgt über die zum Gerät mitgelieferte Software.

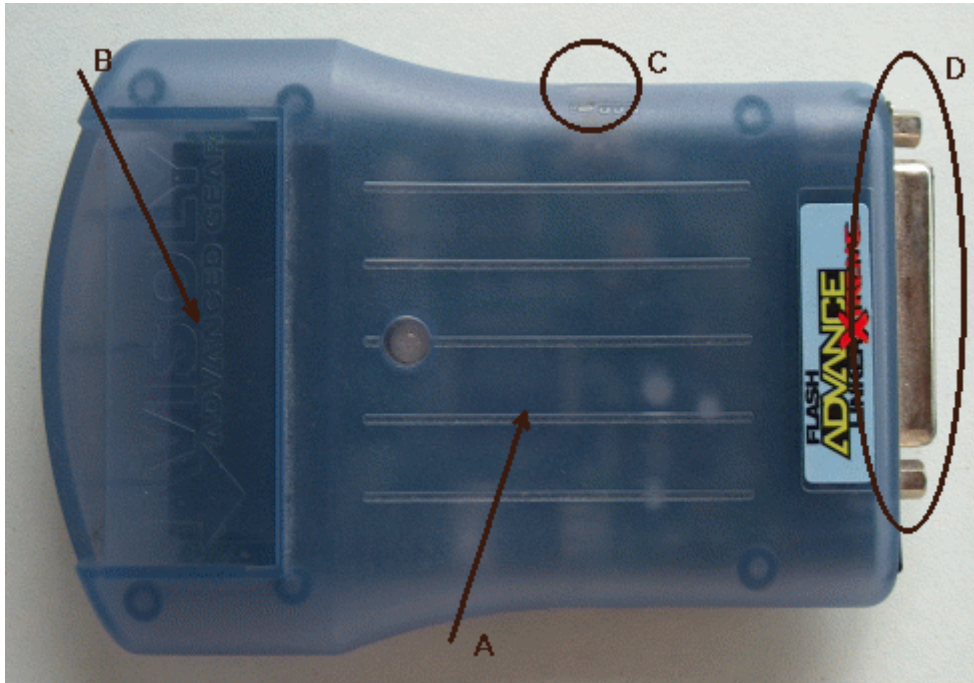


Abbildung 4-10 Flascheinheit mit eingeschobener Flashkarte

4.2.2 Flashkarten

Es gibt mehrere in der Speicherkapazität unterschiedliche Flashkarten. Die von mir verwendete Flashkarte (siehe Abbildung 4-11) hat folgende Merkmale:

- 64 MBit (8 MegaByte) FLASH
- 256 kByte SRAM



Abbildung 4-11 64 MBit Flashkarte

5 Standard Funktionen

5.1 Mathematische Funktionen

Die folgenden mathematischen Funktionen sind nur Ersatzimplementierungen für bereits vorhandene Funktionen. Da der Anwender bedingt durch das Einbinden der Standard-C Bibliotheken eine reichhaltige Auswahl hat und des weiteren das GBA-BIOS (siehe Kapitel 3.10) mehrere mathematische Funktionen anbietet, beschränke ich meine Implementierung auf oft auftretende Funktionen.

5.1.1 Division

Die Division stellt das Hauptproblem dar, besonders weil keine Hardwareimplementierung in der CPU existiert. Für konstante Divisionen mit 3,5 und 10 biete ich eigene Funktionen an. Der Grund ist, dass in vielen Fällen eine Division durch diese Werte auftritt bzw. durch die Kombination dieser Dividenden erreichen lässt und deshalb eine schnelle Division dafür nützlich ist. Die normale Division ist über einen SRT-Divisions Algorithmus implementiert. Der SRT-Divisions-Algorithmus ist ein schneller Divisionsalgorithmus und heißt nach Sweeney, Robertson und Tocher, die ihn unabhängig voneinander vorschlugen.

Der Algorithmus arbeitet wie folgt:

Um a durch b zu teilen, müssen beide Werte in die Register A und B geladen werden.

- Hat B keine führende Nullen bei insgesamt n Bit, verschiebe alle Register k Bit nach links. Wenn b nach dieser Verschiebung n+1 Bit hat, ist anschließend das höchstwertige Bit 0 und das Bit danach, 1.
- Für $i=0$ bis $n-1$
 - Sind die führenden drei Bit von P gleich, setze $q_i=0$, und schiebe (P,A) ein Bit nach links.
 - Sind die führenden drei Bit von P nicht gleich, und ist P negativ, setze $q_i=-1$, schiebe (P,A) ein Bit nach links, und addiere B.
 - Ansonsten setze $q_i=1$, schiebe (P,A) ein Bit nach links, und subtrahiere B.
- Ist der Schlussrest negativ, korrigiere durch Addition von B und den Quotienten durch Subtraktion von 1 von q. Schließlich muss der Rest k Bit nach rechts verschoben werden, wobei k die Anfangsverschiebung ist.

Die Schleife kann durch eine Linearisierung des Algorithmus vermieden werden und dadurch die Geschwindigkeit erhöht werden.

5.1.2 Wurzel

Um eine ganzzahlige Wurzel zu ziehen, gibt es mehrere Lösungsmöglichkeiten. Die momentan schnellste für den ARM optimierte Wurzelfunktion wurde von Wilco Dijkstra entwickelt.

5.1.3 Zusatz

Zusätzlich zu diesen wichtigen Funktionen, habe ich noch einen schnellen Zufallszahlengenerator implementiert.

Die Quelltexte für die betreffenden Funktionen, sind unter dem Pfad `source/stdmath/` zu finden.

5.1.4 Funktions- und Parameterbeschreibung

Die Funktionen *udiv3*, *udiv5*, *udiv10* sind schnelle lineare Funktionen, Register R2 wird als temporärer Register verwendet. Ein Divisionsrest wird nicht zurückgegeben.

Funktionsname:	udiv3, udiv5, udiv10
Beschreibung:	Division (nicht vorzeichenbehaftet) durch 3,5,10
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r1 = Wert
Rückgabewerte:	r1 = Wert / 3 oder 5 oder 10

Die Funktionen *fdiv*, *fdiv_m* benötigen bei ihrer Verwendung IWRAM. Als langsamere Alternative wäre die BIOS-Division (siehe Kapitel 3.10) zu nennen.

Funktionsname:	fdiv
Beschreibung:	Division (nicht vorzeichenbehaftet)
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Dividend r1 = Divisor
Rückgabewerte:	r0 = Quotient

Funktionsname:	fdivs_m
Beschreibung:	Division (vorzeichenbehaftet)
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Dividend r1 = Divisor
Rückgabewerte:	r0 = Quotient

Die Wurzelfunktion *isqrt* benötigt 104 Byte IWRAM und verwendet Register R1 und R2 als temporäre Register.

Funktionsname:	isqrt
Beschreibung:	Wurzel ziehen
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Wert
Rückgabewerte:	r0 = Wert ^{1/2}

Die Funktion *rand* benötigt eine globale Variable mit dem Namen seed. Diese Variable ist 64 Bit breit und muss auf einer durch 4 teilbaren Adresse liegen. In ihr sollte ein zufälliger Wert (z.B. Systemzeit) zum gespeichert werden um eine immer gleiche Zufallszahlen-Generierung zu vermeiden.

Funktionsname:	rand
Beschreibung:	erzeugt Zufallszahl
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = 32 Bit Wert
Besonderheiten:	der globale Seed-Wert sollte vor Benutzung gesetzt werden

Alle zum linken benötigten Objekte befinden sich unter dem Pfad source/stdmath/.

5.2 Allgemeine Funktionen

Da durch Einbinden von Standard C-Bibliotheken und die Verwendung der BIOS-Funktionen eine große Zahl an Standard Funktionen abgedeckt werden, habe ich ähnlich wie bei den mathematischen Funktionen nur eine kleine Auswahl an Funktionen, die insbesondere zu ihren Standard-C Pedanten optimiert sind.

5.2.1 Speichertransfer-Funktionen

Die wichtigsten Reimplementierungen beziehen sich auf Speicherinitialisier- bzw. Speicherkopierfunktionen. Diese Funktionen arbeiten nur mit 32-Bit Datentypen zusammen und benutzen für die Speichertransfers den DMA-Kanal-3 (siehe Kapitel 3.7). Zu beachten ist, dass bei DMA-Operationen die CPU angehalten wird und nur durch eine Unterbrechung (IRQ) bzw. das Ende der DMA-Operation wieder arbeiten kann.

5.2.2 Quicksort

Eine weitere Neuimplementierung betrifft den Quicksort Algorithmus (qsort). Der Grund für eine Überarbeitung ist der hohe Stackverbrauch des Standard-C Algorithmus. Dieses Problem wird durch das Sichern aller Register beim Aufruf der Funktion erzeugt. Leider ist dem Compiler nicht bewusst, dass es sich hierbei um eine rekursive Funktion handelt und er deshalb den

Stackverbrauch optimieren sollte. Besonders durch die geringe Speichergröße des GBA ist vom Standard-C-Quicksort abzuraten. Aus Gründen der Stackoptimierung, kann diese Implementierung nicht die Flexibilität des Standard-C-Quicksort bieten. Dieses Quicksort beschränkt sich auf eine absteigende Sortierung der Daten (kann durch die Testfunktion verändert werden (z.B. Werte negieren für aufsteigendes Sortieren)), wobei der Benutzer mittels einer Funktion die Daten an seinen Datentyp anpassen kann. Selbiges gilt für den Tausch von Datentypen. Um bei diesen Funktionen unnötige Stackoperationen zu vermeiden, werden zwei frei verwendbare Register (R7, R12) für den Benutzer bereitgestellt.

Die folgende Abbildung veranschaulicht eine Testimplementierung für den Quicksort-Algorithmus. Ziel der Anwendung ist es, eine Array von Polygonen nach ihrer Tiefe zu sortieren (das ist eine einfache Art zur Bestimmung einer korrekten Anzeige von aus Polygonen bestehenden Objekten). Die Funktion *getcont* gibt in diesem Beispiel den Vergleichswert zurück der vom Quicksort-Algorithmus an einer bestimmten Stelle verlangt wird (R0[R3]). Die Funktion *swapcont* hat die Funktion des Tauschens von zwei Objekten, die durch den Quicksort Algorithmus bestimmt werden (R0[R8] und R0[R9])

```

;*****
;*
;* getcont (bekomme Key an Stelle r3 von Polygonliste r0)
;*
;* r0 = adr, r3 = pos
;* ret: r3 = cont (r7,r12 for free use)
;*
;*****
getcont
    mov     r7,r3, lsl #3
    add     r7,r7,#6                ;Polygon Tiefe
    ldrsh   r3,[r0,r7]
    bx      lr

;*****
;*
;* swapcont      (tausche Polygon Stelle r3 von Polygonliste r0)
;*
;* r0 = adr, r8 = i (tmp), r9 = j (tmp) (r7,r12 for free use)
;*
;*****
swapcont
    add     r8,r0,r8, lsl #3
    add     r9,r0,r9, lsl #3

    ldr     r7,[r8]                ;Tausche 8 Byte
    ldr     r12,[r9]
    str     r7,[r9],4
    str     r12,[r8],4

    ldr     r12,[r9]
    swp     r7,r12,[r8]
    str     r7,[r9]

    bx      lr

.
.
.

```



```

;****   sortiere Polygone absteigend nach ihren Tiefe

ldr     r0,=polys           ;Polygone
ldr     r2,=polycnt         ;Polygonanzahl

mov     r1,#0               ;Start = 0
sub     r2,r2,#1            ;Ende  = Anzahl-1

ldr     r10,=getcont        ;get[x]
ldr     r11,=swapcont       ;swap[a,b]
bl      qsort_ex            ;starte Sortieren

```

Abbildung 5-1 QuickSort Beispiel Implementierung

Unter dem Pfad \source\extern\vector\ steht ein Beispielprogramm zur Verfügung, dass den von implementierten Quicksort verwendet. Abbildung 5-2 zeigt einen Screenshot des Programms.



Abbildung 5-2 Vektorobjekt

5.2.3 V-Blank

Ein kleine aber wichtige Funktion zur V-Blank (vertikaler „Rasterstrahl“ des Displays) - Überwachung ist mit *wait* implementiert worden. Mit ihr kann auf eine beliebige VBlank-Positionen gewartet werden. Diese Funktion ist besonders zum Softwaretest sehr hilfreich. Für häufige VBlank-Überprüfungen ist sie aber bedingt durch ihre Busy-Wait Implementierung aus Gründen des Energieverbrauchs durch BIOS-Funktionen zu ersetzen. Da diese, solange kein Interrupt auftritt (z.B. VBlank), die CPU im Schlafmodus halten.

5.2.4 Debugausgabe

Zur schnellen Debugausgabe auf dem Bildschirm habe ich ein kleines System implementiert, dass es erlaubt ASCII-Texte, Dezimal- und Hexadezimalzahlen auszugeben. In einigen Fällen ist die Verwendung der String Funktionen (z.B. *sprintf*) der C-Bibliotheken, bedingt durch die Vielfalt der Formatiermöglichkeit, dieser Implementierung vorzuziehen. Aber leider ist die Verarbeitungsgeschwindigkeit der C-String-Funktionen in vielen Fällen absolut unakzeptabel und deshalb für eine schnelle Debugausgabe nicht geeignet.

5.2.5 Funktions- und Parameterbeschreibung

Die Funktionen *mendeat32*, *memset32* und *dmaqpy* sind auf Verwendung der GBA-DMA basierende Lösch/Setz/Kopier-Funktionen. Diese Funktionen arbeiten nur mit 32 Bit

Datentypen zusammen und benutzen DMA-Kanal 3. Bei Aufrufen in Interrupt-Funktion sollte strikt darauf geachtet werden, dass keine DMA-Operation eines unterbrochenen Programms angehalten wurde.

Funktionsname:	memclear32
Beschreibung:	löscht Speicherbereich (Adresse muss auf 4 Byte gerade sein)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r1 = Ziel r2 = Anzahl Bytes (wird auf 4 aufgerundet)

Funktionsname:	memset32
Beschreibung:	Setzt den Speicherbereich (Adresse muss auf 4 Byte gerade sein)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = Füllbyte r1 = Ziel r2 = Anzahl Bytes (wird auf 4 aufgerundet)

Funktionsname:	dmacpy
Beschreibung:	kopiere Daten (Adresse muss auf 4 Byte gerade sein)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = Quelle r1 = Ziel r2 = Anzahl Bytes (wird auf 4 aufgerundet)

Die Funktion *wait* sollte beim Warten auf die V-Blank-Position, durch die vom Bios angebotenen Funktionen ausgetauscht werden.

Funktionsname:	wait
Beschreibung:	warte auf VBlank-Position
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r2 = Position

Die folgenden Funktionen sind wie beschrieben zur schnellen Debugausgabe konstruiert. Eine Testimplementierung ist im MOD-Player unter dem Pfad `source\sound\mod_test\` zu finden.

Funktionsname:	Drawc
Beschreibung:	setze Zeichen auf Map (Mode 0)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = ASCII Zeichen r1 = Mapbase r2 = X r3 = Y

Funktionsname:	puts
Beschreibung:	setze String auf Map (Mode 0)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = String (0 terminiert) r1 = Mapbase r2 = X r3 = Y

Funktionsname:	puts_hex
Beschreibung:	schreibe Hexadezimalzahl auf Map (Mode 0)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = Wert r1 = Mapbase r2 = X r3 = Y

Funktionsname:	puts_dec
Beschreibung:	schreibe Dezimalzahl auf Map (Mode 0)
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = Wert r1 = Mapbase r2 = X r3 = Y

Funktionsname:	clrscr
Beschreibung:	lösche Map
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r1 = Mapbase

Alle zum linken benötigten Objekte befinden sich unter dem Pfad `source/stdfkt/`.

5.2.6 Interrupt-Handler

In den Kapitel 2.7 (Exceptions) und dem Kapitel 3.9 (Interrupt-Kontroller) wurde bereits auf die verschiedenen Interrupts und ihrer Ursachen eingegangen. Bei vielen Spiele-Systemen existieren mehre Interrupt-Vektoren für unterschiedliche Hardware-Interrupts. Dies ist meist dadurch begründet, dass in vielen älteren Systemen die Motorola 68000 CPU verwendet wurde. Diese CPU bietet von 256 Vektoren: 192 User-Vektoren und 7-Level Auto-Vektoren inkl. einer Vielzahl von Trap-Vektoren und den normalen CPU-Vektoren. Der ARM7tdmi ist diesbezüglich sehr schwach ausgestattet. Das hat zur Folge, dass alle externen Interrupts (IRQ) auf eine Adresse geleitet werden (`$300ffffc`). Es ist für den Benutzer erforderlich ab dieser Stelle zu erkennen, um welchen Interrupt es sich handelt und welche Aktion ausgeführt werden soll. Es ist in den seltensten Fällen sinnvoll den Code direkt zu implementieren, da es oft zu wechseln der Interrupt-Routinen kommt. Die einfachste Möglichkeit ist für jeden Interrupt einen eignen Vektor bereit zu stellen und diesen vom Interrupt-Handler aus aufzurufen. Dieses System ist in der von DevKitArm²⁴ bereitgestellten `crt0.o` Datei implementiert. Für viele Benutzer ist dieses System ausreichend. Leider ist mit diesem System nur ein Interrupt-Programm-Aufruf pro Interrupt möglich, deshalb muss der Benutzer sich von Hand um die Interrupt-Belegung kümmern. Weiterhin ist im Fall, eines sehr schnell abzuarbeitenden Interrupts, keine Sonderbehandlung möglich.

Aus diesen Gründen bietet mein Framework eine eigene Implementierung des Interrupt-Handlers an. Dieser Handler erlaubt eine einfaches Hinzufügen und Löschen von Interrupts. Es kann vom Benutzer festgelegt werden, wie viel Interrupts für die Interrupt-Quellen erlaubt sind. Weiterhin ist der Interrupt-Handler zu Multiplen- und Normalen-Interrupt-Aufrufen kompatibel. (Multiple-Interrupt bedeute, dass nach auftreten des Interrupts, das Interrupt-Programm im unprivilegierten System-Modus ausgeführt wird und dadurch neue Interrupts möglich sind.) Um den Benutzer selbst entscheiden lassen zu können, ob er Multiplen- und Normalen-Interrupt-Aufrufe benutzt, muss der Aufruf des Interrupt-Handler vom Benutzer selbst implementiert werden. Abbildung 5-3 zeigt eine Implementierung für einen nicht-Multiplen-Interrupt. Abbildung 5-4 einen Mutiple Version. Der Schreib-Zugriff auf die Adresse `$30007ff8` ist für einige BIOS-Funktionen notwendig (z.B. VBlankWait), kann aber bei Nichtgebrauch entfernt werden. Eine Testimplementierung beider Systeme wird unter dem Pfad `/source/extendedfkt/irq_test` bereitgestellt.

²⁴ URL: www.devkit.tk (27.07.2004)

```

;*****
;*
;* GBA-IRQ (nicht multiple)
;*
;* r0 = $40000000      (vom Bios belegt)
;* r0-r3,r12 saved
;*
irq
    add     r1,r0,#$200

    ldrh    r2,[r1,#2]      ;irq lesen
    strh    r2,[r1,#2]      ;irq ausschalten
    strh    r2,[r0,#-8]     ;save irq bits fürs bios $30007ff8
                                ;(mirror auf $3ffffff8 )

    mov     r0,r2
    ldr     r1,=do_irq_th+1  ;irq handler
    bx      r1

```

Abbildung 5-3 normaler IRQ

```

;*****
;*
;* GBA-IRQ (multiple)
;*
;* r0 = $40000000      (vom Bios belegt)
;* r0-r3,r12 saved
;*
irq_multi
    add     r1,r0,#$200
    ldrh    r2,[r1,#2]      ;irq lesen
    strh    r2,[r1,#2]      ;irq ausschalten
    strh    r2,[r0,#-8]     ;save irq bits fürs bios $30007ff8
                                ;(mirror auf $3ffffff8 )

    mrs     r0,spsr          ;save spsr
    stmfd   sp!,{r0}
    mrs     r0,cpsr
    bic     r0,r0,#$df        ;
    orr     r0,r0,#$1f        ;system mode (ab jetzt user stack)
    msr     cpsr,r0

    mov     r12,lr            ;sichere lr
    mov     r0,r2             ;ie
    ldr     r1,=do_irq_th+1   ;irq handler
    mov     lr,pc             ;back
    bx      r1                ;go
    mov     lr,r12

    mrs     r0,cpsr
    bic     r0,r0,#$df        ;
    orr     r0,r0,#$92        ;irq aus, irq mode
    msr     cpsr,r0
    ldmfd   sp!,{r0}
    msr     spsr, r0          ;set spsr
    bx      lr

```

Abbildung 5-4 Multipler IRQ

Das Ablaufschema der Funktionen wird Abbildung 5-5 visualisiert. Das Unterbrechungs-Antwort (UAP) -Array, welches die UAP-Adressen enthält, hat pro Interrupt eine maximale Anzahl von Adresswerten. Die Standard-Definition für diesen Wert (MAX_UPA_PI) ist 4. Er kann vom Benutzer, je nach Bedarf, verändert werden. Es gibt keinerlei Beschränkung für Aufrufe in privilegierten System-Modi.

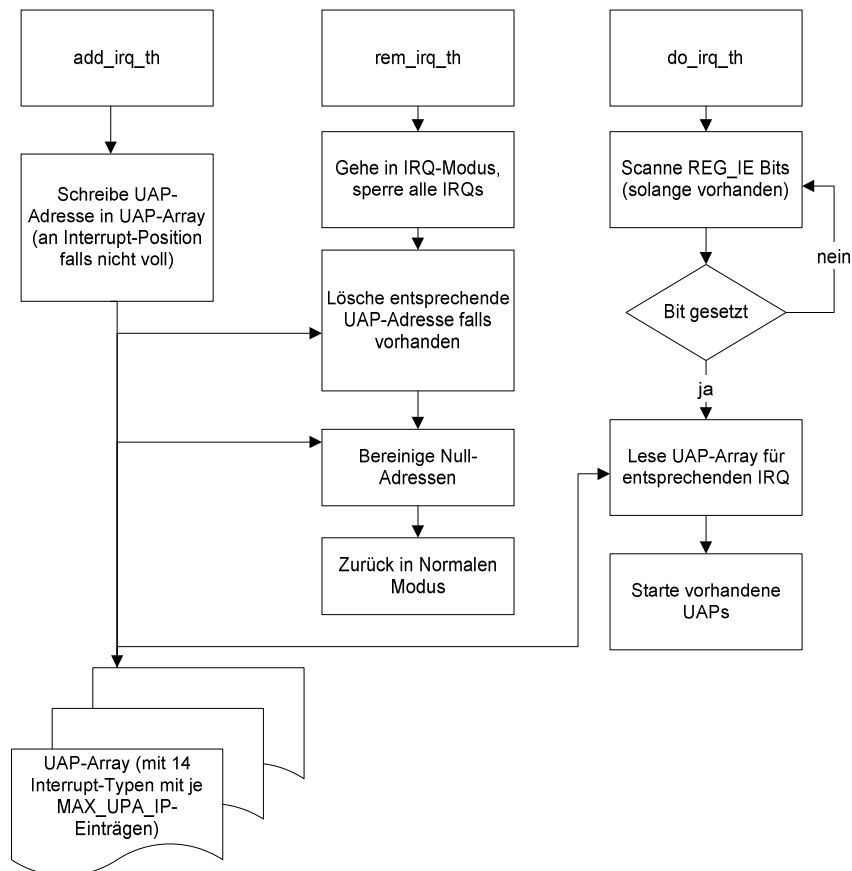


Abbildung 5-5 IRQ-Handler Ablauf

5.2.6.1 Funktions- und Parameterbeschreibung

Das Einhängen eines neuen Unterbrechungs-Antwort-Programms (UAP) erfolgt über die Funktion **add_irq_th**. Als Übergabeparameter wird in Register R0 die Adresse des UAP angegeben. Zu beachten ist, dass dieses die Register R0 bis R3 nicht sichern muss, da dies der Interrupt-Handler erledigt.

Als Interrupt-Quelle kann unter folgenden Interrupts gewählt werden (siehe Kapitel 3.9).

- LCD V-Blank
- LCD H-Blank
- LCD V-Counter Match
- Timer 0 Overflow
- Timer 1 Overflow
- Timer 2 Overflow
- Timer 3 Overflow
- Serial Communication

- DMA 0
- DMA 1
- DMA 2
- DMA 3
- Keypad
- Game Pak (ROM)

Die Schlüsselwörter des Interrupt-Typs sind wie folgt definiert

```
IRQ_HANDLER_VBLANK
IRQ_HANDLER_HBLANK
IRQ_HANDLER_VCOUNT
IRQ_HANDLER_TIMER_0
IRQ_HANDLER_TIMER_1
IRQ_HANDLER_TIMER_2
IRQ_HANDLER_TIMER_3
IRQ_HANDLER_SERIAL_COMMUNICATION
IRQ_HANDLER_DMA_0
IRQ_HANDLER_DMA_1
IRQ_HANDLER_DMA_2
IRQ_HANDLER_DMA_3
IRQ_HANDLER_KEY
IRQ_HANDLER_CASSETTE
```

Die Funktion ***add_irq_th*** gibt bei erfolgreichen Einhängen des Interrupts den Wert Null im Register R0 zurück.

Funktionsname:	add_irq_th
Beschreibung:	hänge neuen Interrupt ein
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Adresse des Unterbrechungs-Antwort-Programms r1 = Interrupt-Typ
Rückgabewerte:	r0 = wenn nicht Null dann nicht eingehangen

Das Entfernen eines Unterbrechungs-Antwort-Programms (UAP), erfolgt über die Funktion ***rem_irq_th***. Zu beachten ist, dass diese Funktion beim Aushängen des UAP, in den privilegierten System-Modus wechselt und weitere Interrupts kurzzeitig verbietet. Nur dadurch ist ein fehlerfreies Verändern der internen IAP-Arrays möglich, ohne das durch einen Interrupt falsche Abläufe entstehen können.

Funktionsname:	rem_irq_th
Beschreibung:	entfernen eines Unterbrechungs-Antwort-Programms
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Adresse des Unterbrechungs-Antwort-Programms r1 = Interrupt-Typ

Die Funktion ***do_irq_th*** verarbeitet die angegebenen Interrupt-Bits und startet die für die entsprechenden IAP. Abbildung 5-3 und Abbildung 5-4 verdeutlichen die üblichen Aufrufe dieser

Funktion. Um im Falle eines Cartridge-Interrupts (externe ROM-Unterbrechung) korrekt zu arbeiten, befindet sich diese Funktion im erweiterten WRAM.

Funktionsname:	do_irq_th
Beschreibung:	starte Unterbrechungs-Antwort-Programms (je nach Aktivierung)
Codetype:	THUMB
Speichertyp:	EWRAM
Übergabeparameter:	r0 = REG_IE

6 Musik und Soundprogrammierung

6.1 Allgemeine Einführung in die Problematik

Ein Spiel ohne Musik und Soundeffekte auszustatten ist für professionelle Entwicklungen nicht denkbar. Der Aufwand, der für die Musikuntermahlung aktueller Spiele betrieben wird ist teilweise sehr enorm, denn er spielt für die Bewertung eines Spieles eine große Rolle. Sie können sich sicherlich vorstellen, dass zu einem zeitgemäßen Autorennen auch ein realistisch klingender Motorsound gehört, gepaart mit einer mitreißenden Hintergrundmusik. Ohne diese Effekte würde solch ein Spiel sicherlich bei potenziellen Käufern schlecht ankommen und egal wie schön es auch aussieht, nur unter die Kategorie amateurhaft fallen.

Im Allgemeinen ergeben sich also zwei unterschiedliche Arten, zum einem die Musik und zum anderen der Sound. Der Unterschied ist offensichtlich, Musik ist eine Komposition von Tönen die in ihrer Reihenfolge festgelegt sind. Als Sound definiert man einen Ton, der zu einem bestimmten Ereignis abgespielt wird, dieses Ereignis kann natürlich eine Note sein, somit ist die Musik von den Tönen (Sounds) abhängig.

6.2 Die GBA Sound-Hardware

Der GBA ist für den Bereich Musik und Sound ausreichend ausgestattet. Traditionell besitzt er 4 Soundgeneratoren und zwei direkte FM-Kanäle.

Unter Soundgeneratoren versteht man Geräte, die eine definierte Schwingung mit einer festlegbaren Frequenz erzeugen, z.B. wäre hier eine Sägezahnschwingung zu nennen.

Bevor ich auf die technischen Genauigkeiten eingehe, möchte ich noch etwas näher auf die Soundgeneratoren im Allgemeinen eingehen. Wie sie sich vorstellen können, sind die erzeugten Töne dieser Generatoren akustisch sehr beschränkt. Aber dafür haben sie mehrer Vorteile, die in der Vergangenheit eine große Rolle spielten. Denn nicht zu vergessen ist, dass Generatoren weder Rechenzeit von der CPU benötigen, noch Speicher für ihre Arbeit benutzen. Die technische Implementierung eines Generators ist relativ trivial und die Kosten sind sehr niedrig. Bevor der Amiga eine neue Generation in der Computermusik einläutete, hatten fast alle Computer und Spielkonsolen Soundgeneratoren an Bord. Als Beispiel wären hier der C64 mit seinem Soundprozessor (SID) zu nennen der 3 Generatoren beherbergte. Auch hatte jede Spielkonsole in den 80'er und Anfang der 90'er Jahren Soundgeneratoren integriert. Trotz der starken Beschränkungen die Soundgeneratoren aufwiesen, wurden in der damaligen Zeit viele Musikstücke geschrieben, die in ihrer Qualität kaum auf ein solch beschränktes System hinwiesen.

6.3 Die GBA-Soundgeneratoren im Detail

Der eigentliche Grund für die Existenz dieser Generatoren ist auf die Abwärtskompatibilität zum original Gameboy zurückzuführen.

Ich gehe in den folgenden Kapiteln etwas ausführlicher auf die Funktion dieser Geräte ein, da sie immer noch eine Verwendung für synthetische Soundeffekte erfüllen können und stelle am Schlussendlich nur ein minimales Framework für diese Generatoren zur Verfügung.

6.3.1 Sound-Generator 1

Dieser Soundgenerator erzeugt Rechtecksignale mit einer variablen Frequenz deren Amplitude zu einer bestimmten Frequenz verändert werden kann. Eine festlegbare Amplitude des Rechtecks ist weiterhin definierbar. Außerdem bietet er die Möglichkeit einer Hüllenkurve, das bedeutet, dass die Amplitude inkrementiert oder dekrementiert werden kann.

Gehen wir etwas näher auf die Amplitudenänderung ein, im Folgenden „Sweep Shifts“ genannt, da dieser Begriff im Normalfall verwendet wird.

Formel gesehen ergibt sich folgendes:

$$T = T \pm \frac{T}{2^n}$$

Wobei gilt, T ist die Periode und n ist die Anzahl der Verschiebungen.

Es gibt 7 verschieden Möglichkeiten für die Definition eines solchen „Sweep Shifts“:

Modus	Ts
0	Sweep aus
1	Ts=1 / 128Khz (7.8 ms)
2	Ts=2 / 128Khz (15.6 ms)
3	Ts=3 / 128Khz (23.4 ms)
4	Ts=4 / 128Khz (31.3 ms)
5	Ts=5 / 128Khz (39.1 ms)
6	Ts=6 / 128Khz (46.9 ms)
7	Ts=7 / 128Khz (54.7 ms)

Abbildung 6-1 Sweep Shifts

Zu beachten ist, dass es noch die Möglichkeit gibt, zu wählen, ob eine Erhöhung oder eine Verringerung der Periode angewendet werden soll.

Betrachten wir jetzt ein Beispiel für dieses Problem. Es ist eine Startperiode von 7.8 ms festgelegt mit einer Zeit von 54.7 ms und einem „Sweep Shift“ von 1.

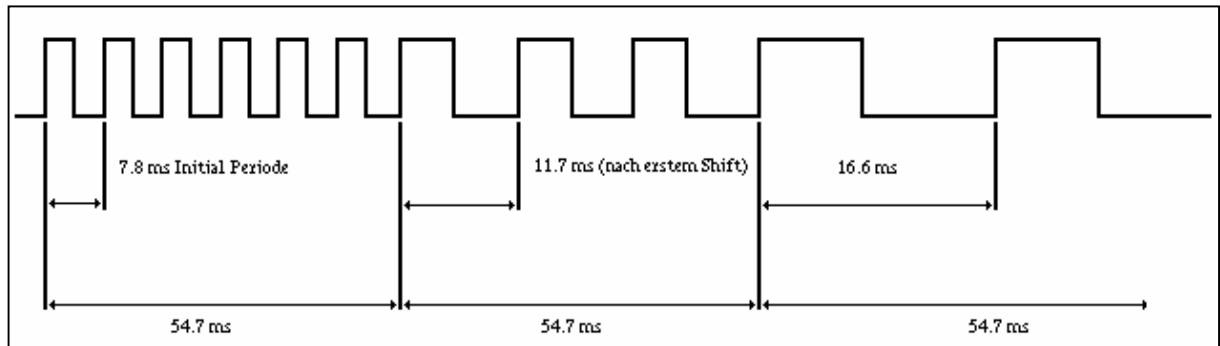


Abbildung 6-2 Frequenzänderung durch Schieben

Somit ergibt sich immer eine Addition der Periode mit der Periode/2 hoch den Shift, der nach abgelaufener Zeit von 54.7 ms erhöht wird. Akustisch bedeutet dies einen sehr hohen Ton der schnell in einen tiefen Ton abgeleitet.

Betrachten wir jetzt die Hüllkurven-Funktion (envelope function), der Sinn dieser Funktion ist es, einen Sound bezüglich seiner Lautstärke ein- oder auszublenden. Die Auflösung für einen solchen „fade-in“ oder „fade-out“ ist 4 Bit. Die Funktion der Wartezeit ist wie folgt definiert:

$$T = \text{steptime} \cdot \frac{1}{64}$$

Eine weitere Besonderheit ist es, die Breite des aktiven Rechtecksignals zu wählen. Wenn dieses „duty cycling“ aktiviert ist ergeben sich 4 Möglichkeiten für die Amplitude.

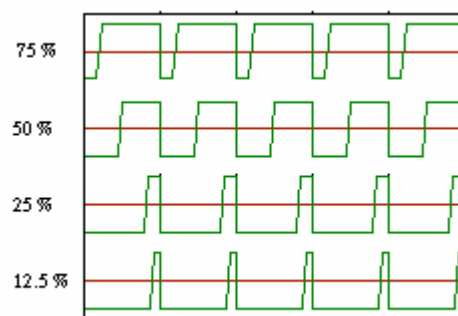


Abbildung 6-3 Hüllkurven

6.3.2 Sound-Generator 2

Dieser Generator ist identisch mit Generator 1, jedoch fehlt ihm das „Sweep Shifting“, deswegen behält ein Ton immer seine Startfrequenz. Dieser muss direkt neu besetzt werden um diese zu ändern.

6.3.3 Sound-Generator 3

Bei Sound-3 handelt es sich in Wahrheit um einen einfachen 4-Bit DAC und nicht um einen Soundgenerator. Jedoch sind die gegebenen Beschränkungen so stark, dass man auch nicht wirklich von einem vernünftigen Sound-DAC reden kann. Im eigentlichen Sinne handelt es sich um ein sehr großes Schieberegister, das $32 \cdot 4$ Bit in den DAC schiebt. Diese $32 \cdot 4$ Bit großen Speicherbereiche sind doppelt vorhanden und bieten dadurch die Möglichkeit, parallel oder hintereinander abgespielt zu werden. Natürlich kann die Abspielgeschwindigkeit geändert werden.

6.3.4 Sound-Generator 4

Dieser Soundgenerator ist ein sehr typischer Soundgenerator und in fast jedem System welches auf generierten Sound setzte, vorhanden. Es handelt sich um einen Generator der ein Rauschen erzeugt (Pseudo-Noise). Dieses Rauschen wird von einem Lineare-Feedback Shift Register (LFSR) erzeugt. In der folgenden Abbildung ist das Schaltbild für einen 7-Stufen LSFR dargestellt.

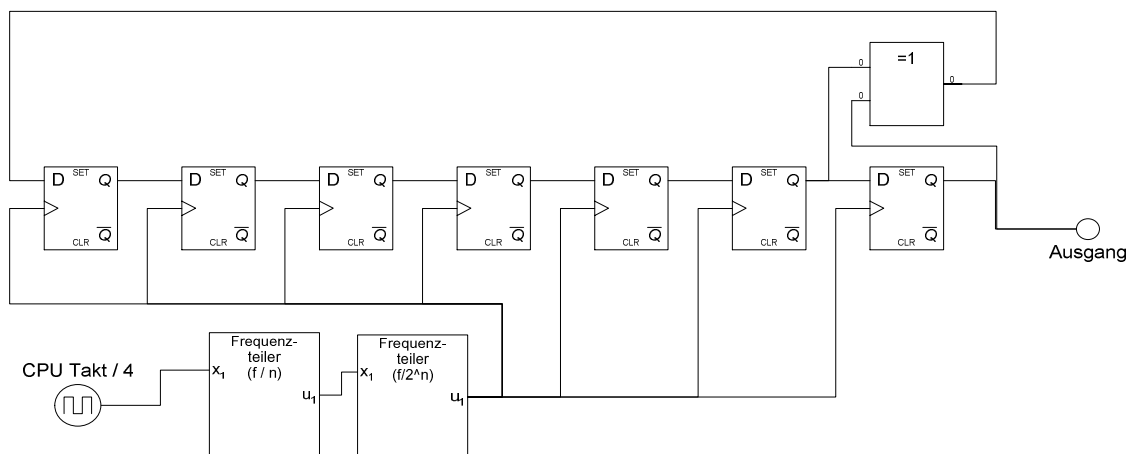


Abbildung 6-4 7-Stufen-LSFR

Im GBA gibt es noch die Möglichkeit zwischen den hier gezeigten 7-Stufen LSFR und einen 15-Stufen-LSFR umzuschalten. Die Frequenzen können über 2 Register geregelt werden (zwei Frequenzteiler) und wie schon in den anderen Generatoren ist auch hier eine Volume-Envelope vorhanden.

Ein interessanter Punkt ist das Erzeugen der Eingangsfrequenz für dieses LSFR. Bei allen anderen Generatoren war diese Frequenz in einem festgelegten Bereich wählbar. In diesem System wird aber der Soundsystemtakt (4.194304Mhz) als Eingangstakt verwendet. Um jetzt diese hohe Frequenz wieder in einen für den Menschen hörbaren Bereich zu bringen, werden zwei Frequenzteiler verwendet. Der Erste teilt die Frequenz im ersten Schritt durch 8 und gibt dann seinen Benutzer noch die Möglichkeit, durch einen wert von 1 bis 7 zu teilen oder die Frequenz zu verdoppeln. Wie sie sicherlich festgestellt haben liegt die jetzt erzeugte Frequenz im

kleinsten Fall bei ca. 75 kHz, also immer noch viel zu hoch. Ab jetzt kommt aber der zweite Frequenzteiler zum tragen, er bietet die Möglichkeit die ankommende Frequenz durch 2^n zu teilen, wobei zu beachten ist, dass n zwischen 0 und 15 liegen kann. Durch diesen zweiten Schritt können die Frequenzen in ein vernünftiges Spektrum gebracht werden und so auch eine Vielfalt an unterschiedlichen Geräuschen erzeugt werden.

Abschließend ergibt sich folgender mathematischer Zusammenhang zwischen Ein- und Ausgangsfrequenz:

$$f_{out} = \frac{(f_{in} / n)}{2^m} \quad [n, m \in \mathbb{R}; 1 \leq m \leq 14; 1 \leq n \leq 7 \mid n = 0.5]$$

Ich möchte an diesem Punkt auch schon das Kapitel Soundgeneratoren beenden, da diese nur eine sehr nebensächliche Rolle in der heutigen Computermusik bzw. Soundeffekterzeugung spielen. Anzumerken ist noch, dass es nur bedingt Sinn macht, über die weit reichenden technischen Funktionen dieser Generatoren zu reden. Denn spätestens bei der Arbeit mit Generatoren, werden sie feststellen, dass eine mathematische Beschreibung eines Tones und dessen gehörter Ton recht abstrakt voneinander abweichen. Deshalb gilt hier die Regel „probieren geht über studieren“.

6.4 Direkter Sound

Unter direktem Sound ist zu verstehen, dass ein Sample (definierter Wellenbereich) direkt ausgegeben wird. Der GBA besitzt zwei solcher direkten Soundkanäle. Ihre Auflösung beträgt 8 Bit, wobei zu beachten ist, dass es sich um vorzeichenbehaftete Zahlen handelt.

Die Frequenz bzw. die Geschwindigkeit in der ein solches Sample ausgegeben werden kann, wird über einen Timer gesteuert und sollte im Normalfall zwischen 1 und 65535 Hz liegen.

Leider sind diese Soundkanäle nicht DMA fähig, d.h. es ist nicht wie üblich möglich dem Soundprozessor zu sagen, wo die Sampledaten liegen und mit welcher Frequenz er diese abzuspielen hat.

Der GBA besitzt nur zwei je 4 Byte breite FIFO's für die Soundausgabe, aber es gibt keine Möglichkeit dem Soundprozessor (da nicht vorhanden) zu diktieren, wo er die Daten im Speicher findet bzw. in welcher Frequenz er diese abspielen muss. Es gibt prinzipiell zwei Standard Lösungen für dieses Problem. Die erste und auch gebräuchlichste Möglichkeit besteht darin, einen DMA Kanal zu benutzen. Der Vorteil der GBA-DMA-Hardware besteht darin, diese mit einem Timer zu koppeln. Somit kann durch in einer vom einen Timer festgelegten Zeitperiode, immer die DMA dazu gezwungen werden, Daten in den Sound-FIFO zu kopieren. Mit einem zweiten Timer ist es noch zusätzlich möglich, eine Zeitgrenze für das Sample festzulegen und wenn diese vorüber ist, den DMA-Vorgang zu stoppen.

In der folgenden Abbildung wird diese Technik noch einmal verdeutlicht. In diesem Fall gibt Timer 2 den DMA-Kanal den „Takt“ für den Datentransfer von den Sampledaten zum Sound-FIFO. Timer 1 schaltet den DMA-Kanal nach einer bestimmten Zeit wieder ab und definiert somit auch die Länge der Sampledaten.

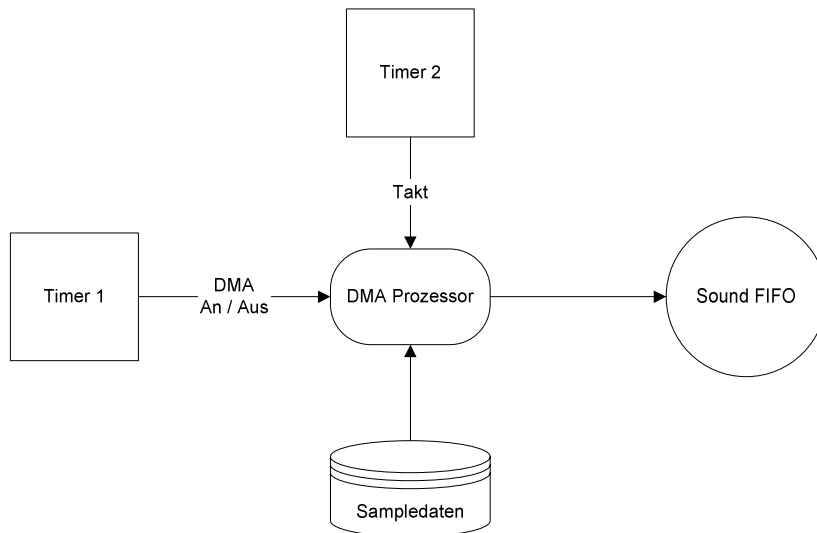


Abbildung 6-5 Soundausgabe mit DMA

Dieses auf den ersten Blick etwas exotisches System zur Soundwiedergabe, hat einen Nachteil der aber im Normalbetrieb nicht auftritt. Er tritt nur bei der Verwendung der Multiplayerfunktion auf, da hier unter Umständen ein IRQ auftritt und dieser den DMA-Strom unterbricht. Dadurch kann es zu Verzögerungen der Soundausgabe kommen. Um diesen Fall zu vermeiden ergibt sich folgende Möglichkeit, den Timer, der normalerweise die DMA steuert, muss nun einen Interrupt auslösen. In diesem Interrupt werden dann die Daten von der CPU in den FIFO kopiert. In der folgenden Abbildung wird dieses Verfahren noch einmal skizziert.

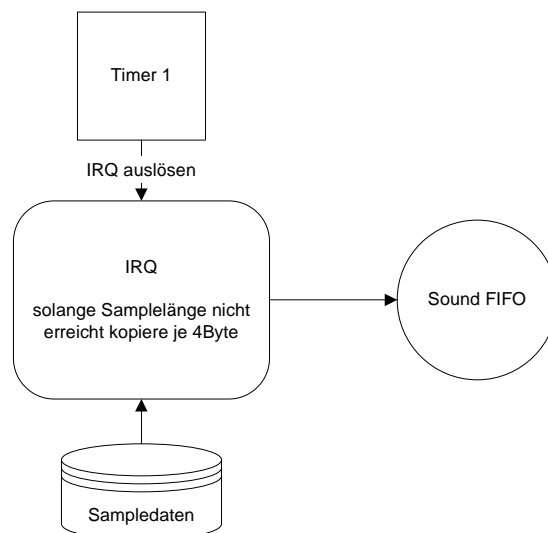


Abbildung 6-6 Soundausgabe mit der CPU

Diese Verfahren vermeidet zwar das Multiplayerproblem, aber der enorme Overhead der durch die Interrupts entsteht, ist im Normalfall nicht akzeptabel. Deshalb sollte das erste System in jedem Fall vorgezogen werden.

Es wurde sich in den letzten Beispielen immer nur mit einer Sound FIFO beschäftigt, der GBA besitzt aber zwei, um dadurch Stereo-Ton zu ermöglichen. Bedingt dadurch, gibt es auch die Möglichkeit beide zu bedienen. Das System ist fast identisch mit den bereits besprochenen. Der einzige Unterschied ist, dass jetzt zwei DMA-Kanäle benutzt werden, jeder separat für einen FIFO.

Wenn diese Probleme gelöst sind, ergibt sich die Möglichkeit digitalisierte Sprache, Musik und Instrumente abzuspielen. Praktisch gesehen können nun alle Arten von Tönen für ein Spiel genutzt werden. Möchten man jetzt Musik erzeugen, ergeben sich wieder eine Vielfalt von Möglichkeiten diese abzuspielen. Für viele PC-Benutzer stellt sich die Lösung recht schnell. Es ist die Benutzung von so genannten Soundstreams, d.h. es wird die digitalisierte Musik, die von einem Musiker davor erzeugt wurde, auf einem Gerät wiedergegeben. Das größte Problem dabei liegt in der Größe dieser Samples, ohne Kompression sind solche Samples für den begrenzten Speicherplatz nicht akzeptabel. Eine vernünftige Kompression, wie z.B. mp3 reduziert die Größe der Daten, benötigt aber auch enorme Rechenzeit zur Decodierung. Leider steht nicht genügend Rechenzeit zur Verfügung um eine geeignete Dekompression zu unterstützen. Die Lösung dieses Problem wurde schon vor der Entwicklung vom mp3 & Co. ermittelt.

6.5 Tracker und ihre Geschichte

Wie viele sehr innovative Ideen und Programme, entstand auch das Soundtracking, mit der Unterstützung von digitalisierten Instrumenten, auf dem Amiga. Der erste Schritt wurde im Jahre 1987 getan, als *Karsten Obarski* sein Programm namens "Soundtracker" für den Amiga veröffentlichte. Er legte damit sozusagen den Grundstein aller Tracker. Die Idee hinter einen Tracker ist es, verschiedene Samples nach einer Art Notentabelle abzuspielen. Allerdings wurden die Noten mit Erweiterungen bestückt, bzw. es gab z.B. Kommandos die die Geschwindigkeit des Liedes steuerten oder die Lautstärke eines Samples (Darüber wird in den nächsten Kapiteln ausführlich eingegangen).

Vielleicht werden sie sich fragen, warum gerade auf dem Amiga ein solches Programm erfunden wurde. Die Antwort ist einfach, der Amiga hat zur damaligen Zeit einen Soundprozessor namens Paula an Bord. Dieser Soundprozessor war DMA fähig und bot 4 von einander unabhängige Kanäle mit je einer 8-Bit Auflösung an, der Frequenzbereich ist frei festlegbar gewesen (unter PAL bis ca. 32 kHz). Zur damaligen Zeit war diese Leistung etwas besonderes, zumal der Amiga auch einen vernünftigen Preis hatte und so für viele Menschen erschwinglich war. Der PC bot zu dieser Zeit nichts vergleichbares, hier vergingen noch Jahre bis Worte wie Multimedia, Plug & Play, Multitasking usw. Einzug hielten. Zur damaligen Zeit entwickelte sich, bedingt durch dieses Programm auch eine große Szene an Musikern die mit solchen Programmen arbeiteten. Ich sage nicht umsonst solche Programme, denn es entstanden immer mehr solcher Tracker die auch

mehr als 4 Kanäle zuließen, jedoch immer den Uhrtyp die so genannte MOD-Datei akzeptierten. Als Amiga-Programme sind besonders Protracker²⁵, Octamed²⁶, Octalyzer, Digibooster²⁷ zu nennen, die fast alle den Benutzern kostenlos zur Verfügung gestellt wurden. Auf dem PC entstanden später Programme wie Screamtracker²⁸, Fasttracker²⁹ und aktuell noch Soundtracker³⁰ und Madtracker³¹. Durch die Vielzahl an Programmen und der immer größer werden Szene entstanden auch unzählige Musikstücke, deren Anzahl sich im 5-stelligen Bereich bewegen. Weiterhin hatte diese große Anzahl an Programmen zur Folge, dass viele unterschiedliche Arten der Speicherung der Musikdaten entstanden. Mit der Zeit kristallisierten sich aber 3 wichtige Formate heraus, zum einen das alte MOD-Format, gefolgt von dem XM (Fasttracker) und S3M (Screamtracker) Format.

Das MOD-Format hatte im Normalfall die Beschränkung auf 4 Kanäle, wurde aber trotzdem oft benutzt, weil auf den Amiga technisch bedingt nur 4 Kanäle zur Verfügung standen. Selbst mit steigender Prozessorleistung wurde noch viel an dem alten Format festgehalten.

XM und S3M unterstützten mehr als 4 Kanäle und ließen auch erweiterte Effekte zu, man kann diese Formate als Nachfolger vom MOD-Format betrachten. Allerdings werden sie es wohl nie komplett verdrängen können.

In den nächsten Kapiteln wird meine Implementierung für das Abspielen einer solchen MOD-Datei auf dem GBA beschrieben. Das Resultat ist ein Framework, das es den Benutzer ermöglicht mit einfachstem Mitteln, in seinem Spiel Musik abzuspielen.

6.6 Kanäle Mischen

Wie bereits beschrieben (siehe Kapitel 6.4), stellt die GBA-Hardware zwei direkte Soundkanäle zur Verfügung, das MOD-Format benötigt aber vier. Also müssen softwaretechnisch zwei Kanäle in einem gemischt werden. Nicht zu vergessen ist, dass beide Samples, die gemischt werden eine unterschiedliche Frequenz, Länge und Lautstärke haben können. Ein weiteres sehr wichtiges Feature ist der mit dem MOD-Format eingeführte Loop-Modus (Wiederholungen im Sample).

²⁵ URL (Download): ftp://de.aminet.net/pub/aminet/mus/edit/PT4_Beta2.lha (27.07.2004)

²⁶ URL: www.med.uk.com/amiga.htm (27.07.2004)

²⁷ URL: www.digiboosterpro.de (27.07.2004)

²⁸ URL (Download): www.united-trackers.org/resources/software/screamtracker.htm (27.07.2004)

²⁹ URL: www.gwinternet.com/music/ft2/ (27.07.2004)

³⁰ URL: www.soundtracker.org (27.07.2004)

³¹ URL: www.madtracker.org (27.07.2004)

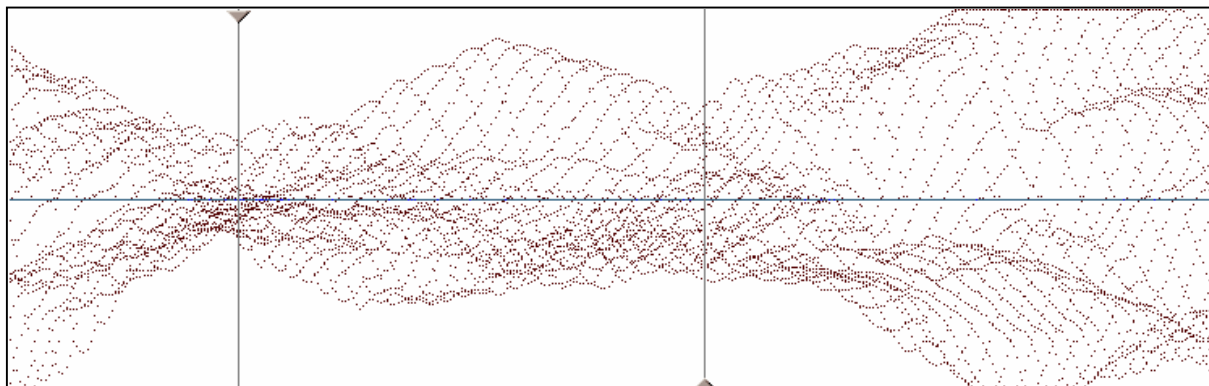


Abbildung 6-7 Sample mit Loop

Abbildung 6-7 stellt einen Sample dar. Bei der vorletzten Erhebung, können sie die zwei vertikalen Linien erkennen. Die Erste ist der Startpunkt für eine Loop (Schleife) und der zweite ist der Endpunkt für eine Loop. Die Bedeutung einer Loop ist trivial, die X-Achse dieses Bildes charakterisiert die Zeit. D.h. wenn ein Sample abgespielt wird, wird das Sample von links nach rechts durchlaufen. Kommen man zu einem bestimmten Zeitpunkt zu dem Endpunkt der Schleife, bedeutet dies, dass im nächsten Moment zum Loop Startpunkt zurückgekehrt wird.

Dadurch ist die Abspieldauer des Samples unendlich lang. Für jemanden der noch nie ein Sample mit einer Loop gehört hat, ist dieses Verfahren bestimmt etwas ungewöhnlich, aber es macht in vielen Fällen Sinn. Eine im XM und S3M System vorhandene Methode ist es noch möglich, ab den Endpunkt rückwärts zum Startpunkt zu laufen und von diesem dann wieder in positiver Richtung zum Endpunkt. Der Name für dieses Verfahren ist „Ping Pong Loop“. Da ich aber nur auf dem MOD-Standard zurückgreife unterstütze ich dieses Verfahren nicht.

Der Loop-Effekt hat den Vorteil, dass eine Längenprüfung für ein Sample nicht mehr relevant ist. Der MOD-Standard unterstützt zwar eine Samplelänge, aber in den von mir programmierten Mixer ist eine Längenabfrage irrelevant, da die Samples vor dem Abspielen etwas modifizieren wurden. Denn wenn ein Sample eine Schleife besitzt, ist es nicht notwendig eine Abbruchbedingung zu formulieren, da das Sample niemals endet. Hat ein Sample aber keine Schleife hat er auch ein Ende. Dieses Ende definiere ich aber genau ein Byte hinter dem Sample, wobei der Loop End- und Startpunkt identisch sind. Also verharrt die Abspielroutine nach dem Ende des Samples auf den letzten plus einem Byte. Dieses Byte wird mit Null gefüllt und somit ist kein Ton hörbar. Es wäre auch möglich gewesen eine Abbruchbedingung für den Länge eines Samples zu formulieren, davon wird aber abgesehen. Der Grund dafür ist, eine Schleifeüberprüfung muss implementiert werden, aber eine Längenprüfung ist nicht unbedingt notwendig, wenn die Samples vorgearbeitet werden. Denn jede Überprüfung kostet Zeit. Würden vier Samples mit 24 kHz gemischt werden, ergeben sich schon 96000 Durchläufe pro Sekunde und hier schlagen einigen wenige Befehle zu einigen Prozents an verbrauchter Rechenzeit an.

Ein weiteres Problem besteht darin, dass beide abgespielte Samples natürlich eine unterschiedliche Frequenz haben können. Es gibt mehrere Lösungsansätze für dieses Problem. Ich stelle für das Abspielverhalten eine Festfrequenz von 24 kHz ein und skaliere die Samples betreffend ihrer

Frequenz. Zum Beispiel würde bei einem Sample, das mit einer Frequenz von 6 kHz abgespielt werden soll, nur alle 4 Durchläufe einen neuen Wert geladen, da $24 \text{ kHz} / 6 \text{ kHz} = 4$ ist. Da Fließkommazahlen nicht vorhanden sind, verwende ich Fixed-Point Werte mit 32 Bit wovon 11 Bit den Nachkommabereich abdecken.

Weiterhin wurde besprochen, dass jeder Kanal eine eigene Lautstärke besitzen kann, MOD definiert hier die Werte 0-64. Sicherlich eine etwas merkwürdiger Bereich, das hat aber wiederum etwas mit der Amiga Hardware zu tun, da der Amigasoundchip (Paula) eine Lautstärke zwischen 0 und 63 zulässt und dann extra noch die 64 als volle Lautstärke interpretiert. Der Mixer-Algorithmus rechnet intern aber mit 9 Bit also 0 bis 511 Stufen.

Wurden nun die geladen Daten mit ihrer festgelegten Lautstärke multipliziert müssen sie wieder durch 512 geteilt werden. Da aber sowieso gemischt wird, kann davor noch addiert werden. Also multiplizieren man jeden Wert mit seiner Lautstärke, addieren diese und teilen sie durch 1024.

$$D_{out} = \frac{Quelle_1 \cdot Vol_1 + Quelle_2 \cdot Vol_2}{1024}$$

Ein Problem ergibt sich aber leider immer, es ist ein Lautstärkenverlust um die Hälfte bei der Mittelung der beiden Werte. Hierfür wurde noch zusätzlich, ein so genannter Volumeboost eingebaut, welcher die Werte verstärkt und am Ende auf seine Grenzen zurückstutzt. Dieses Boostschutz kann aber, auch wenn er nicht benötigt wird, auskommentiert werden, da auch er Rechenzeit verbraucht (BOOST_PREVENTION).

Das folgende Quellcodebeispiel zeigt die Standard Mischsschleife, welche zwei Samples in einen Puffer mischt.

```

mixloop

;r0 = source1          r8 = pos2
;r1 = dest             r9 = sladr
;r2 = frei            r10 = s2adr
;r3 = frei            r11 = frei
;r4 = pos1            r12 = length2
;r5 = freqstep1:freqstep2  r12 = length2
;r6 = length1         lr = count
;r7 = source2         sp = voll:vol2

;**** get data of Source1&2

mov     r3,r4,lsr #11      ;pos
mov     r11,r8,lsr #11     ;pos

ldrsb   r2,[r3,r0]        ;r2=[pos+adr]
ldrsb   r3,[r11,r7]       ;r3=[pos+adr]

;**** inc pos and check loops for Source 1

add     r4,r4,r5,lsr #16   ;pos + freqstep
cmp     r4,r6              ;epos?
movge   r4,r9              ;ja-> pos=spos

;**** inc pos and check loops for Source 2

mov     r11,r5,lsr #16
add     r8,r8,r11,lsr #16  ;pos + freqstep
cmp     r8,r12             ;epos?
movge   r8,r10             ;ja-> pos=spos

;**** mix + volume

mov     r11,sp,lsr #16     ;voll
mul     r2,r2,r11          ;source1*voll

bic     r11,sp,r11,lsr #16 ;vol2 = voll:vol2 AND NOT voll<<16
mla     r3,r3,r11,r2       ;source2*vol2+source1*voll

mov     r3,r3,asr #10      ;9 Vol * 2 Channels

;**** Boost-Schutz

#ifdef BOOST_PREVENTION

cmp     r3,#127
movge   r3,#127
cmp     r3,#-128
movle   r3,#-128

#endif

;**** schreibe

strb    r3,[r1],#1

subs    lr,lr,#1
bne     mixloop

```

Abbildung 6-8 Mischschleife (2 zu 1)

Das MOD-Format unterstützt in seiner Standardform genau 4 unabhängige Kanäle. In einigen Fällen werden aber nicht alle Kanäle verwendet. Deswegen sind noch zwei weitere Routinen implementiert worden, die diese Fälle behandeln. Für den Fall das nur ein Kanal gemischt wird

und für den Fall, dass gar kein Sound ausgegeben wird. Prinzipiell ergeben sich dadurch acht Möglichkeiten für die Soundausgabe auf 4 zu 2 Kanälen.

6.7 Tracker im Detail

Um eine MOD-Datei abspielen zu können muss man ihren Aufbau und ihren zeitlichen Ablauf kennen (alle wichtigen Informationen sind in der MOD-Strukturbeschreibung³² von Brett Paterson oder im „ProTracker support archive“³³ von Håvard Pedersen zu finden). Ein Musikstück besteht aus Patterns, diese Patterns sind nichts anderes als eine Ansammlung von Noten (für jeden Kanal). Im normalen MOD-Standard hat ein Pattern immer 64 Zeilen, somit also auch 64 Noten für jeden Kanal. Durch später beschriebene Befehle kann ein Pattern auch eher beendet werden und es ist nicht notwendig sein Musikstück immer in 64 Noten-Teile zu zerlegen. Weiterhin gibt es eine Pattern-Liste, diese Liste enthält alle Patterns die abgespielt werden sollen. D.h. beim Musikstart wird das erste Pattern aus der Patternliste abgespielt. Wenn dieses Pattern durchlaufen wurde, wird das nächste Pattern aus der Patternliste abgespielt. Im MOD-Format gibt es noch eine Begrenzung auf 64 Patterns und eine Patternliste mit der maximalen Größe von 128 Einträgen.

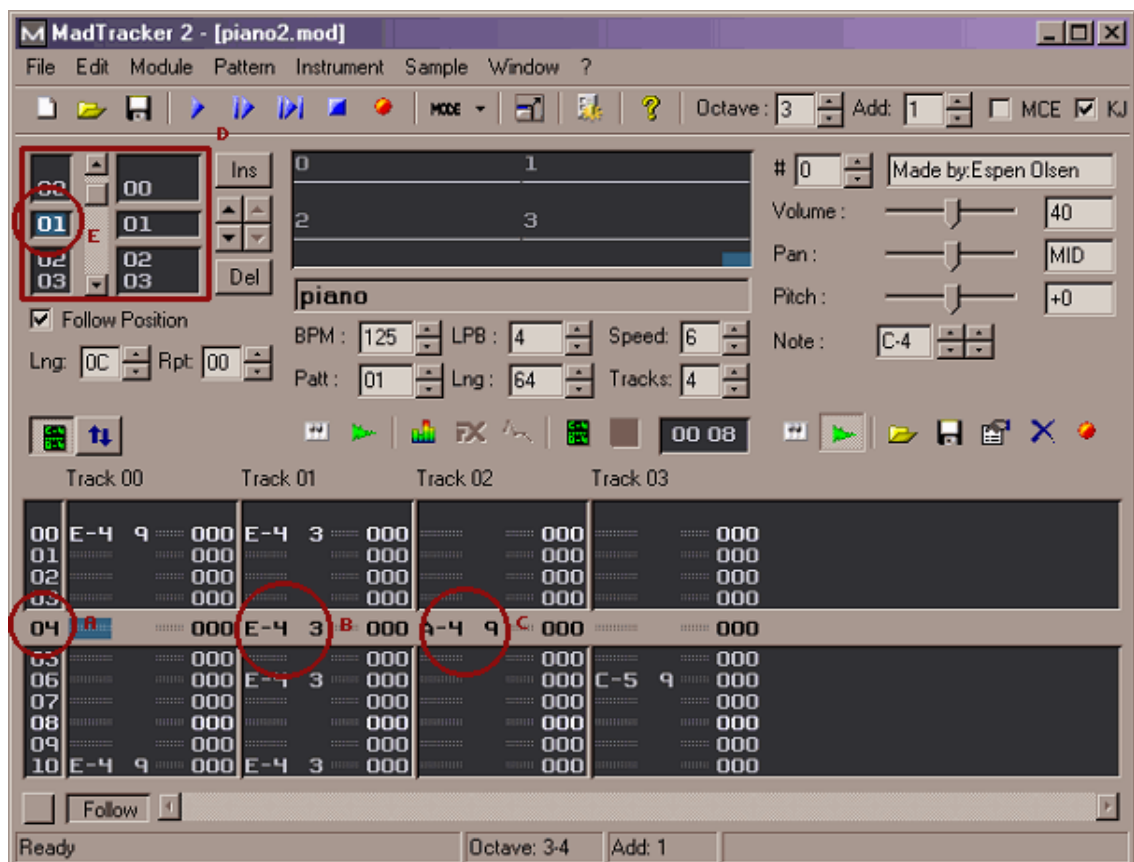


Abbildung 6-9 Mad-Tracker

³² URL: www.textfiles.com/music/fmoddoc.txt (27.07.2004)

³³ URL: ftp://de.aminet.net/pub/aminet/mus/edit/ptsupp.lha (27.07.2004)

In Abbildung 6-9 wird das Programm MadTracker2 dargestellt. Dieses Programm ist ein MOD/XM/S3M/MD – Tracker für den PC (unter Windows). In der Abbildung 6-9 können Sie im unteren Bereich das gerade laufende Pattern erkennen. Es befindet sich gerade an der Position 5 (Abbildung Punkt A) und aktiviert auf Kanal 1 und 2 zwei Noten (Abbildung Punkt B und C). In der linken oberen Ecke sieht man die Patternliste (Abbildung Punkt D), hier befinden es sich auf der zweiten Position (Abbildung Punkt E).

6.7.1 Noten-Aufbau

Wie sie sehen, gibt es pro Patternzeile vier Kanalspalten. In jeder dieser Spalten befindet sich eine Note mit einigen Erweiterungen. Betrachten wir uns eine solchen Note genauer (Abbildung 6-10). Nehmen wir als Beispiel die Note „E-3 5 A01“. Der erste Teil zeigt die eigentliche Note mit ihrem Feintuning an, also die Note E mit Tuning 3. Der nächste Teil ist eine Zahl im Bereich von 0-31, diese Zahl gibt das Sample (Instrument) an, welches zu der gegebenen Note abgespielt werden soll. Der dritte und letzte Wert ist eine 12 Bit großer Wert, er findet seine Anwendung in der Verwendung von verschiedenen Effekten. Zu beachten ist, dass z.B. auch ein Effekt ohne Note ausgeführt werden kann, zum Beispiel wenn die Geschwindigkeit erhöht werden soll (z.B. A01).

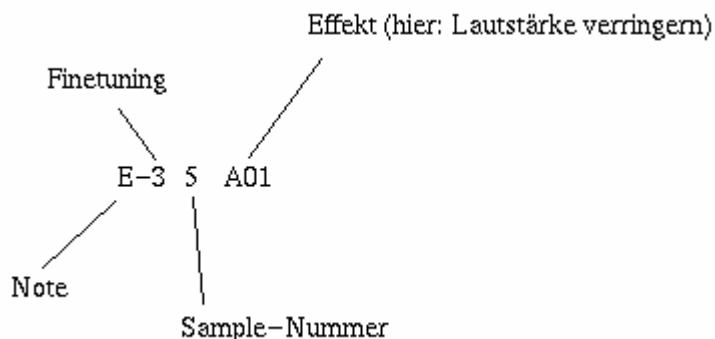


Abbildung 6-10 Note mit Erweiterung

Diese Werte werden in eine 4 Byte große Struktur gesichert. Die Aufschlüsselung dieser Werte geschieht wie folgt:

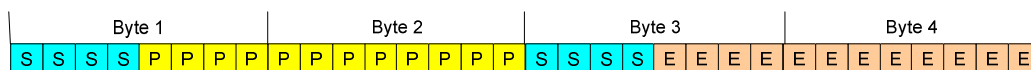


Abbildung 6-11 Noten-Kodierung

- S = Sample-Nummer
- P = Periode
- E = Effekt-Kommando

6.7.2 MOD-Daten-Struktur

Der Aufbau der MOD-Struktur ist in Abbildung 6-12 beschrieben. Da diese Struktur statisch ist, muss sie nicht im Programmablauf geändert werden und kann so im ROM liegen. Dadurch wird kein EWRAM- oder IWRAM-Speicher für die MOD-Datei benötigt.

Offset	Länge in Bytes	Erklärung
0	20	Songname (Null-Terminiert)
20	22	Samplename (Null-Terminiert)
42	2	Sample-Länge (Größe in 16 Bit Worten)
44	1	Finetuning (Reihenfolge 0 bis 7, -8 bis -1)
45	1	Sample-Volume (0-64)
46	2	Wiederholungspunkt für Loop (Größe in 16 Bit Worten)
48	2	Wiederholungslänge für Loop (Größe in 16 Bit Worten)
50	30	Daten für Sample 2
80	30	Daten für Sample 3
.	.	
.	.	
890	30	Daten für Sample 30
920	30	Daten für Sample 31
950	1	Songlänge (Bereich von 1-128)
951	1	muss 127 sein
		Songpositionen von 1-127 (jeweils ein Wert von 0 bis 63)
952	128	
1080	4	"M.K." für 31 Samples andernfalls für 15 Samples
1084	1024	Daten für Pattern 1
.	.	restliche Patterns die in Songpositionen definiert wurden
.	.	
.	.	
1084+1024*Patternmax.	größe Sampledaten	Sample Daten

Abbildung 6-12 MOD-Struktur

6.7.3 Speed-Werte

Bevor ich auf die Effekte eingehe, möchte ich noch einige grundlegende Bedingungen erörtern. Ein wichtiger Punkt ist die Abspielgeschwindigkeit. Die Synchronisation für eine interne Befehlsbearbeitung (Noten-Bearbeitung) liegt bei 50 Hz (bei MOD-Dateien Tickspeed genannt). Diese Frequenz ergibt sich aus dem Amiga-Trackern die sich im Standardfall ihre Synchronisation über den Vertikalen Blank (wenn das Bild neu gezeichnet wird) holen. Da der Amiga in den Standardauflösungen immer das Pal-Format benutzt (Europäische Geräte), ergibt sich somit auch eine Frequenz von 50 Hz.

Die Abarbeitungsgeschwindigkeit der einzelnen Patternzeilen wird durch den „Speed-Wert“ definiert. Dieser Wert ist im Normalfall 6. Die Geschwindigkeit ergibt sich durch die Division mit dem Tickspeed. Im Normalfall also 50/6 Hz, also ca. 8 Patternzeilen pro Sekunde. Vielleicht werden Sie sich fragen wozu es überhaupt einen zweiten Zähler gibt und nicht nur mit einem Wert gerechnet wird. Der Grund liegt bei den Effekten, denn es gibt Effekte die sind Tickspeed (50 Hz) abhängig. D.h. es können mehrere Operationen pro Patternzeile durchgeführt werden.

Das macht bei vielen Effekten Sinn, da z.B. eine Änderung der Notenperiode mit dem normaler Pattern-Geschwindigkeit einfach zu langsam ausgeführt würde.

6.7.4 Effekte

Ohne die Verwendung von Sound-Effekten ist die Implementierung eines MOD-Player relativ einfach. Die Implementierung der gesamten Effektpalette ist sehr zeitaufwendig, zumal einige Dokumentationen sich in einigen Punkten widersprechen und dadurch ein hoher Testaufwand nötig ist. Ich habe in meiner Implementierung nur einen Teil der möglichen Effekte programmiert, das hat mehrere Gründe. Zum einen der Zeitaufwand der bei der Programmierarbeit entsteht, zum anderen werden einige Effekte nur sehr selten von MOD-Dateien benutzt und somit ist ihre Implementierung für einen Grossteil der Lieder nicht notwendig. Der MOD-Standard definiert 14 grundlegende Effekte und 14 erweiterte Effekte. Folgende Tabelle stellt alle Effekte dar und gibt einen Überblick über die implementierten Effekte.

Effekttyp	implementiert	Beschreibung	Werte
0	ja	Normal Play oder Arpeggio	0xy (x- erste Halbnote/y-Sekunden)
1	ja	Beschleunigen	1xx
2	ja	Abbremsen	2xx
3		Ton Poramento	3xx
4		Vibrato	4xy
5		Ton Poramento + Vibrato	5xy
6		Vibrato + Volume Slide	6xy
7		Tremelo	7xy
8	unmöglich	keine Verwendung	
9	ja	Setze Sampleoffset	9xx
A	ja	Volume - Slide	Axy
B	ja	Position Jump	Bxx
C	ja	Setze Volume	Cxx
D	ja	Pattern Break	Dxx
E	ja	E-Kommandos	Exy
F	ja	Setze Speed	Fxx

Abbildung 6-13 Effekt-Implementierung

E-Kommandos	implementiert	Beschreibung	Werte
E0	unmöglich	Setze Filter	E0x
E1	ja	Feines Slide hoch	E1x
E2	ja	Feines Slide runter	E2x
E3		Glissando Kontrolle	E3x
E4		Setze Vibrato Wellenform	E4x
E5	ja	Setze Finetuning	E5x
E6		Springe zu Schleife	E6x
E7		Setze Tremelo Wellenform	E7x
E8	unmöglich	keine Verwendung	
E9		halte Note	E8x
EA	ja	Feines Volumeslide hoch	E9x
EB	ja	Feines Volumeslide runter	Eax
EC		Noten abschneiden	Ecx
ED		Note warten	Edx
EE		Pattern warten	EEX
EF		Invertier Schleife	Efx

Abbildung 6-14 E-Effekt-Implementierung

Ich möchte im Folgenden nur kurz auf die implementierten Effekte eingehen. Eine Ergänzung der fehlenden Effekte sollte unter nicht zu hohen Aufwand möglich sein, da alle Effekte Funktionen über eine Sprungtabelle aufgerufen werden. Somit ist eine Erweiterung unproblematisch durchführbar.

6.7.4.1 Normal Play oder Arpeggio

Dieser Effekt verändert die Periode in zwei Schritten. Als Übergabe erhält der Effekt zwei Werte (x und y). Der Ablauf relativ zur Tick-Position ist wie folgt:

1. mache nichts
2. addiere den x-Wert zur Periode
3. addiere den y-Wert zur Periode
4. setze Frequenz zurück

wiederhole solange bis kein anderer Effekt kommt

6.7.4.2 Beschleunigen

Dieser Effekt hat als Parameter einen Wert von 0 bis 255. Es wird bei jedem Tick eine Erhöhung der Periode um den gegebenen Wert ausgeführt.

z.B.:

```
Befehl: C-2 01 105
```

```
Tick 1: mache nichts
```

```
Tick 2: Periode plus 5
```

```
Tick 3: Periode plus 5
```

```
usw.
```

6.7.4.3 Abbremsen

Dieser Effekt ist identisch mit dem Vorherigen, nur wird hier subtrahiert, anstatt addiert.

6.7.4.4 Setze Sampleoffset

Dieser Effekt ermöglicht es, einem Sample an einer anderen Startposition abzuspielen. Als Parameter ist wieder ein Wert von 0 bis 255 erlaubt. Die Startposition ergibt sich aus dem Übergabewert multipliziert mit 256.

6.7.4.5 Volumeslide

Mittels dieses Effektes ist es möglich die Lautstärke eines Kanals zu ändern. Es gibt zwei getrennte Werte (x und y). Der X-Wert definiert die Erhöhung der Lautstärke der Y-Wert die Verringerung der Lautstärke. Zu beachten ist, dass diese Werte mit dem aktuellen Speed minus 1 multipliziert werden.

z.B.

```
b-4 01 a01 -> Lautstärke - 1 * (speed-1)
```

```
a01 -> Lautstärke - 1 * (speed-1)
```

```
a50 -> Lautstärke + 5 * (speed-1)
```

6.7.4.6 Positionssprung

Dieser Effekt ermöglicht es global in einem Pattern zu springen. Als Übergabeparameter wird ein Wert zwischen 0 bis 63 erwartet, der die neue Zeile im Pattern angibt.

6.7.4.7 Setze Volume

Setze die Lautstärke auf den übergeben Wert (0 bis 64).

6.7.4.8 Pattern Break

Bei diesem Effekt wird das vorhandene Pattern beendet und auf das nächste Pattern gesprungen. Es kann zusätzlich noch die Zeile des nächsten Patterns angegeben werden, an welcher fortgesetzt werden soll.

6.7.4.9 Setzte Geschwindigkeit

Durch diesen Effekt kann die Abspielgeschwindigkeit geändert werden. Der Übergabewert darf im Bereich von 0 bis 31 liegen und gibt den neuen Speed-Wert an. Alle größeren Werte werden ignoriert, da sie ein BPM (Beats per Minute) Wert angeben. Dieses System unterstütze ich aber nicht.

6.7.4.10 Feines Slide hoch

Dieser Effekt ist ähnlich dem normalen Slide-Effekt, jedoch wird hier nicht mit dem Speed-Wert multipliziert. Der Übergabeparameter wird auf die vorhandene Periode addiert.

6.7.4.11 Feines Slide herunter

Wie Feines Slide hoch, nur wird hier subtrahiert.

6.7.4.12 Setze Finetuning

Es wird der Finetuning-Wert eines Instrumentes geändert.

6.7.4.13 Feines Volumeslide hoch

Dieser Effekt erhöht die Lautstärke um den angegebenen Wert.

6.7.4.14 Feines Volumeslide herunter

Dieser Effekt verringert die Lautstärke um den angegebenen Wert.

6.8 Interner Ablauf

Die folgende Abbildung verdeutlicht den Ablauf des Soundsystems. Ein wesentlicher Punkt des Systems ist es, dass es über eine IRQ ausgelöst wird. Dieser IRQ tritt exakt 50-mal in der Sekunde auf und startet die eigentliche Verarbeitung. Im ersten Schritt werden die Kanäle gemischt, die Daten welche dafür nötig sind, wurden im vorherigen Lauf berechnet. Der nächste Schritt ist die Überprüfung, ob der „Speed“-Zähler übergelaufen ist. Wenn dies geschehen ist, muss eine neue Patternzeile bearbeitet werden. In diesem Fall, werden die Daten aus dem MOD geladen und dekodiert. Sollte ein Effektkommando aufgetreten sein, wird dieses je nach Typ sofort ausgeführt oder später bei dem Tick-Prozess (jede 50'tel Sekunde) verarbeitet. Wenn Tick-Effekte aufgetreten sind, werden diese anschließend bearbeitet. Alle jetzt neu berechneten Daten werden gespeichert und im nächsten Durchlauf ausgeführt. Durch dieses System ergibt sich immer eine Verzögerung von 1/50 Sekunde zwischen Note und Ausgabe, allerdings ist diese Verzögerung von dem Benutzer nicht wahrnehmbar.

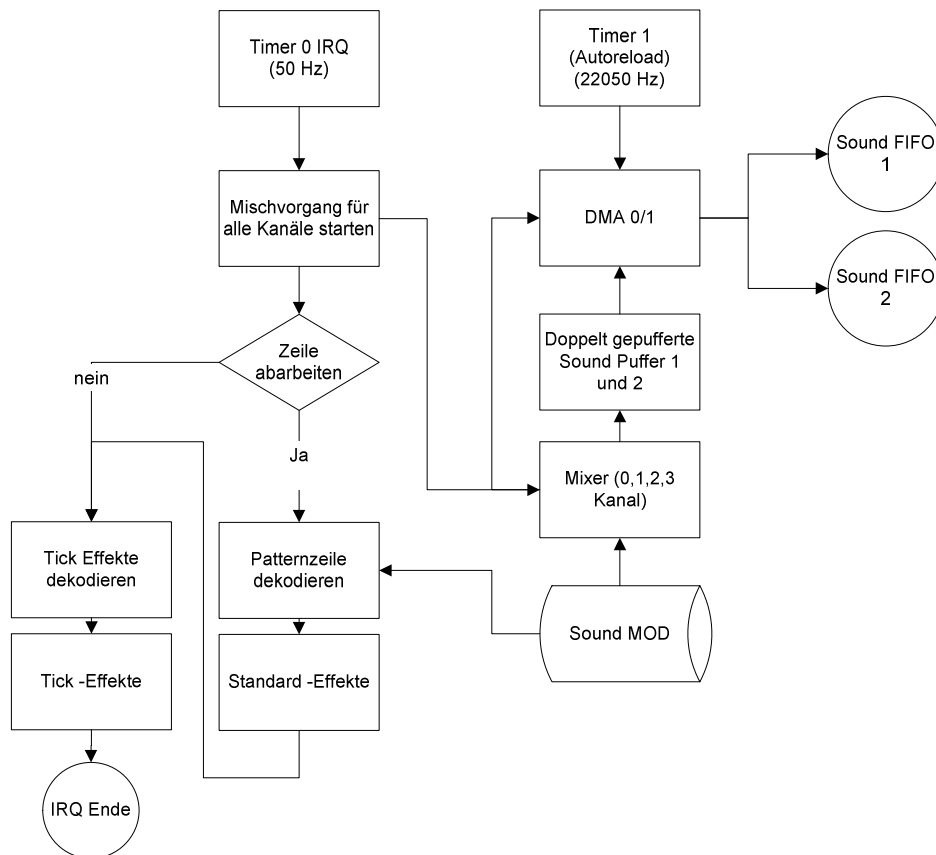


Abbildung 6-15 MOD-Player (Ablauf)

6.9 Konvertierung vom MOD-Dateien

Unter dem Verzeichnis `source/sound/mod_conv_gcc/mod.cpp` ist ein Konverter inkl. Quellen zu finden. Ich habe diesen Konverter programmiert, damit er das Format der MOD-Dateien abändert. Die Änderungen sind wie folgt:

- Big-Endian-Werte in Little-Endian-Werte umwandeln (da original MOD Big-Endian)
- Instrumente ohne Loop mit einer Schleife besetzen (Loop-Start/Ende = Sämplelänge+4 (mit Null besetzt))
- Neue effizientere Kodierung der Kanal-Daten (siehe Abbildung 6-16 (P = Periodenindex, S = Sample-Nummer, E = Effekt))

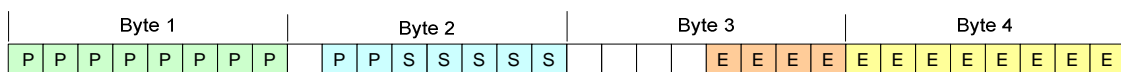


Abbildung 6-16 Kanal-Kodierung

Die gesamte Programmlogik ist in der Datei `mod.cpp` zu finden.

Dem Programmaufruf folgen zwei Parameter. Der Erste gibt den Dateinamen der originalen MOD-Datei an, der Zweite den der zu erzeugenden Datei.

Name:	<code>mod_conv.exe</code>
Synopsis:	<code>mod_conv.exe inputfile outputfile</code>
Beispiel:	<code>./mod_conv.exe mod1.mod mod1.mmod</code>

Zusätzlich werden noch weitere Informationen ausgegeben:

- Songname
- Alle vorhandenen Instrumente inkl. Ihrer Datenstrukturen
- Songlänge
- Songtrackercode
- Tracker-Id
- Song-Positionen
- Höchster Pattern
- Instrument-Daten-Positionen

Die Folgende Abbildung visualisiert eine Standardkonvertierung.

```
$ ./mod_conv.exe ../../mods/zubliminal.mod zb.mod
decode

Songname: 'zubliminal'
Sampleinfo (0): Name      ' helmer -nov 95....'
                  Length   5640
                  Finetune  0
                  Volume    64
                  LoopSt    $0
                  LoopEnd   $2

Sampleinfo (1): Name      'a wise man once said:'
                  Length  16352
                  Finetune  0
                  Volume    64
                  LoopSt    $286e
                  LoopEnd   $3fe0

.
.
.
Songlength:  '31'
Songtracker:  '127'
TrackerId   : 'M.K.'
Song positions: 0 2 2 1 3 2 7 4 5 5 6 7 a 8 8 e e 2 2 4 5 4 3 2 2 4 5 5 6 7
f
Highest Pattern: f

Instrument 00 at $443c (len=5640)
Instrument 01 at $5a44 (len=16352)
Instrument 02 at $9a24 (len=9764)
Instrument 03 at $c048 (len=10582)
.
.
.
writing mod
```

Abbildung 6-17 MOD-Konverter Testausgabe

Zum Testen der konvertierten MOD-Dateien ist in dem Verzeichnis source/sound/mod_test/ ein Beispielquelltext zu finden, welcher eine MOD-Datei abspielt und die wichtigsten Informationen auf dem Display ausgibt (siehe Abbildung 6-18).

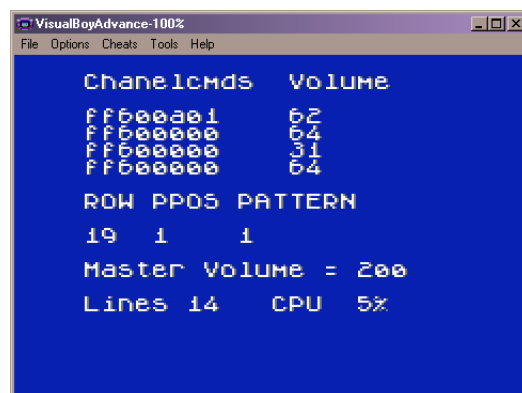


Abbildung 6-18 MOD-Test

6.10 Funktions- und Parameterbeschreibung

6.10.1 MOD-Player

Bevor die Initialisierung des MOD-Players mit der Funktion *init_mod* ausgeführt wird, muss die globale Funktion *irq_mod* mit dem Timer-1 Interrupt aufgerufen werden. Dazu kann der Interrupt-Handler (siehe Kapitel 5.2.6) des Frameworks verwendet werden.

Funktionsname:	init_mod
Beschreibung:	Initialisiert den Player
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	R0 = mod (Pointer auf die MOD-Datei)

Mit dem Aufruf der Funktion *start_mod* wird das Modul gestartet.

Funktionsname:	start_mod
Beschreibung:	Startet die Musik
Codetype:	ARM
Speichertyp:	ROM

Nach dem Anhalten des Moduls mit der Funktion *stop_mod* werden Timer 0/1 und die DMA 1/2 Kanäle freigegeben.

Funktionsname:	stop_mod
Beschreibung:	Stoppt die Musik
Codetype:	ARM
Speichertyp:	ROM

Die Änderung der Lautstärke ist mit der Funktion *changevolume_mod* möglich. Die erwarteten Werte liegen im Bereich von 0 bis \$200 bei nicht aktiviertem Boost-Schutz. Bei höheren Lautstärke-Werten sollte `BOOST_PREVENTION` im Sound-Mixer definiert werden um den Boost-Schutz zu aktivieren, andernfalls kann es zu fehlerhaften Ausgaben des Mixers kommen (Übersteuern).

Funktionsname:	changevolume_mod
Beschreibung:	ändert die Lautstärke der Musik
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	R0 = 0 bis \$200 (ohne Boostschutz)

Zu beachten ist, dass die DMA-Kanäle 0 und 1 verwendet werden, sowie die Timer 0 und 1. Da diese Timer und DMA-Kanäle die höchste Priorität haben, muss von einer Unterbrechung des eigenen Programms ausgegangen werden. Die Zeit bis zum Wiedereintritt kann je nach

Samplefrequenz variieren. Bei zeitkritischen Systemen sollte also bei der Verwendung des MOD-Players auf unerwünschte Seiteneffekte geachtet werden.

6.10.2 Generatoren

Zu jeden Generator existiert neben der allgemeinen Funktion zum Setzen der Werte, eine Funktion zum Auslesen der Daten. Bedingt dadurch, dass einige Register-Bits nicht lesbar sind (nur schreibbar), können nicht alle Daten durch die Lesefunktionen bestimmt werden.

Funktionsname:	set_sound_chanel_1
Beschreibung:	setze Daten für Sound Kanal 1
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Sound Länge im Bereich von (64-n)/256s r1 = Wave Pattern Duty r2 = Envelope Step-Time im Bereich von n/64s (0 = kein Envelope) r3 = Envelope Richtung (0 = absteigende, 1 = aufsteigend) r4 = Initial Volume of Envelope (1-15, 0 = kein Sound) r5 = Anzahl der Sweep Shift (n = 0-7) r6 = Sweep Frequency Direction (0 = absteigende, 1 = aufsteigend) r7 = Sweep Time in Teilen von je 7.8ms (0-7, min=7.8ms, max=54.7ms) r8 = Frequenz im Bereich von 131072/(2048-n)Hz (n=0-2047) r9 = Längen Flag (1= stoppt die Ausgabe wenn die länge in NR11 erreicht ist) r10 = Initial (1=Restart Sound)

Funktionsname:	get_sound_chanel_1
Beschreibung:	lese Daten von Sound Kanal 1
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r1 = Wave Pattern Duty r2 = Envelope Step-Time im Bereich von n/64s (0 = kein Envelope) r3 = Envelope Richtung (0 = absteigende, 1 = aufsteigend) r4 = Initial Volume of envelope (1-15, 0 = kein Sound) r5 = Nummer der „sweep shift“ (n = 0-7) r6 = Sweep Frequency Direction (0 = absteigende, 1 = aufsteigend) r7 = Sweep Time in Teilen von je 7.8ms (0-7, min=7.8ms, max=54.7ms) r9 = Längen Flag (1= stoppt die Ausgabe wenn die länge in NR11 erreicht ist)

Funktionsname:	set_sound_chanel_2
Beschreibung:	setze Daten für Sound Kanal 2
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Sound Länge im Bereich von (64-n)/256s r1 = Wave Pattern Duty r2 = Envelope Step-Time im Bereich von n/64s (0 = kein Envelope) r3 = Envelope Richtung (0 = absteigende, 1 = aufsteigend) r4 = Initial Volume of envelope (1-15, 0 = kein Sound) r8 = Frequenz im Bereich von 131072/(2048-n)Hz (n=0-2047) r9 = Längen Flag (1= stoppt die Ausgabe wenn die länge in NR11 erreicht ist) r10 = Initial (1=Restart Sound)

Funktionsname:	get_sound_chanel_2
Beschreibung:	lese Daten von Sound Kanal 2
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r1 = Wave Pattern Duty r2 = Envelope Step-Time im Bereich von n/64s (0 = kein Envelope) r3 = Envelope Richtung (0 = absteigende, 1 = aufsteigend) r4 = Initial Volume of envelope (1-15, 0 = kein Sound) r9 = Längen Flag (1= stoppt die Ausgabe wenn die länge in NR11 erreicht ist)

Funktionsname:	set_sound_chanel_3
Beschreibung:	setze Daten für Sound Kanal 3
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Sound Länge im Bereich von (64-n)/256s r1 = Wave Pattern Duty r2 = Envelope Step-Time im Bereich von n/64s (0 = kein Envelope) r3 = Envelope Richtung (0 = absteigende, 1 = aufsteigend) r4 = Divisions-Rate der Frequenz r5 = Zähler- Schritte/Breite (0=15 bits, 1=7 bits) r6 = Shift Clock Frequenz r8 = Längen Flag (1= stoppt die Ausgabe wenn die länge in NR11 erreicht ist) r9 = Initial (1=Restart Sound)

Funktionsname:	get_sound_chanel_3
Beschreibung:	lese Daten von Sound Kanal 3
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r1 = Wave Pattern Duty r2 = Envelope Step-Time im Bereich von n/64s (0 = kein Envelope) r3 = Envelope Richtung (0 = absteigende, 1 = aufsteigend) r4 = Divisions-Rate der Frequenz r5 = Zähler- Schritte/Breite (0=15 bits, 1=7 bits) r6 = Shift Clock Frequenz r8 = Längen Flag (1= stoppt die Ausgabe wenn die länge in NR11 erreicht ist)

Funktionsname:	set_sound_chanel_4
Beschreibung:	setze Daten für Sound Kanal 4
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 Bit 0 = Wave RAM Bank Nummer (0-1) r0 Bit 1 = Wave RAM Dimension (0 = Eine Bank/32, 1=Zwei Banken/64) r0 Bit 2 = Sound Channel 3 Aus (0=Stop, 1=Playback) r1 = Sound-Länge (256-n)/256s r2 = Sound Volume (0=Mute/Null, 1=100%, 2=50%, 3=25%) r3 = Force Volume (0=Use above, 1=Force 75% regardless of above) r4 = Frequenz; $131072/(2048-n)$ Hz (n=0-2047) r5 Bit 0 = Längen-Flag (1=Stoppt Ausgabe wenn Länge in NR31 erreicht) r5 Bit 1 = Initial (1=Restart Sound) r6 = WRAM0 Daten r7 = WRAM1 Daten r8 = WRAM2 Daten r9 = WRAM3 Daten

Funktionsname:	get_sound_chanel_4
Beschreibung:	lese Daten von Sound Kanal 4
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 Bit 0 = Wave RAM Bank Nummer (0-1) r0 Bit 1 = Wave RAM Dimension (0 = Eine Bank/32, 1=Zwei Banken/64) r0 Bit 2 = Sound Channel 3 Aus (0=Stop, 1=Playback) r2 = Sound Volume (0=Mute/Null, 1=100%, 2=50%, 3=25%) r3 = Force Volume (0=Use above, 1=Force 75% regardless of above) r5 Bit 0 = Längen-Flag (1=Stoppt Ausgabe wenn länge in NR31 erreicht) r6 = WRAM0 Daten r7 = WRAM1 Daten r8 = WRAM2 Daten r9 = WRAM3 Daten

6.10.3 Direkter Sound

Die Ausgabe von Samples über die direkten FIFO's A und B, sind bedingt durch das Benutzen von je zwei Timern und einem DMA Kanal etwas umständlich. Um den Anwender eine einfache Möglichkeit zur direkten Soundausgabe zu geben, wurden die folgenden Funktionen entwickelt. Zu beachten ist, dass bei der Verwendung des MOD-Players, keine direkte Soundausgabe benutzt werden darf, da der MOD-Player sie selbst verwendet.

Bevor mit der Funktion *play_sound*, das Abspielen eines Samples begonnen werden kann, muss der Benutzer den zu verwendenden Timer wählen. Der gewählte Timer führt, je nach Länge des Samples, Interrupts aus. Diese müssen vom Benutzer abgefangen und auf die Funktion *play_sound_irq* umgeleitet werden. Die Struktur *struct_sl*³⁴ enthält temporäre Daten die auch der Interrupt-Routine mitgegeben werden müssen (in R7)).

³⁴ Quelle: /sound/sound_direct.i

Ein Beispiel zur Verwendung dieser Funktionen ist unter `source/sound/direct_test/` zu finden.

Funktionsname:	play_sound
Beschreibung:	spiele direkten Sound ab
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r0 = Sample r1 = Sample-Länge r2 = $\$ffff-16777216/(\text{Abspiel-Frequenz})$ r3 = Kanal (0 = Kanal 1 (DMA1), 1 = Kanal 2 (DMA2)) r4 = Timer (0 = Timer 1, 1 = Timer 2) r10 = &struct_sd

Funktionsname:	play_sound_irq
Beschreibung:	spiele direkten Sound ab
Codetype:	ARM
Speichertyp:	ROM
Rückgabewerte:	r7 = &struct_sd

6.11 Fazit

Wie sie sicherlich bemerkt haben, ist das Abspielen von Musik mit Sicherheit nicht trivial, sondern ein enormer Aufwand. Deshalb wurde auch viel Zeit darin investiert diesen MOD-Player zu programmieren. Er nimmt den Spielentwickler viel Arbeitszeit ab und kann auch jederzeit modifiziert werden.

7 Animation von Bildern

7.1 Allgemeine Einführung in die Problematik

Der Begriff Animation kann in der Spielentwicklung in vielerlei Hinsicht gedeutet werden. In den folgenden Kapiteln, gehe ich aber auf die Animation von Bildern ein. Kurz gesagt, es ist prinzipiell nichts anderes, als das abspielen einer Bildfolge. Der erste Gedanke der den Einsatz eines solchen Animation darstellen könnte, ist das abspielen eines kleinen Films. Doch bei näherer Überlegung ist festzustellen, dass das Einsatzszenario Film eher selten zum Einsatz kommt. Denn in Spielen sind viele Objekte animiert, d.h. es ergeben sich unterschiedliche Anforderungen an die Art der Animation und deren Verarbeitung.

Zusammengefasst ergeben sich mehrere Probleme bei dem Abspielen von Bildern. Wie in fast allen Fällen stehen sich wieder Speicherplatz und Rechenleistung kontrovers gegenüber. Ohne die Verwendung von einer Kompression ergibt sich je nach Anzahl der Einzelbilder (in Zukunft Frames genannt) große Mengen an Daten. Im Falle einer Kompression ergibt sich aber wiederum ein hoher Verbrauch an Rechenleistung. Somit steht fest, dass ein statischer Ansatz zu der Lösung dieses Problems keinen Erfolg bringen wird. Auf den Fall, dass keine Kompression benutzt wird, gehe ich nicht näher ein, da dieses Problem so trivial ist, dass kein Framework dafür sinnvoll ist.

7.2 Anforderungen an den Algorithmus

Animationen unterscheiden sich auf den ersten Blick in ihrer Auflösung und ihrer Farbtiefe, gefolgt von der Anzahl ihrer Frames. Im Fall des GBA kommt noch der Faktor der Anordnung der Daten hinzu, damit auch direkt und ohne Umwandlung Tilebasierte Darstellungen benutzt werden können. Der nächste Punkt ist die Art der Kompression, der sich dadurch ergebender Speicherplatzverbrauch und die benötigte Rechenleistung beim Dekomprimieren der Daten.

7.3 Spezielle Problemanalyse der Anforderungen

Bevor wir uns genauer mit der Kompression der Daten befassen, untersuchen wir die allgemeinen Lösungen zur Verarbeitung von Animationen und blicken auch in schon vorhandene Lösung zu dieser Problematik. Im heutigen Multimediazeitalter gibt es viele Arten von Kompressionstechniken für Filme/Animationen, die bekanntesten sind MPEG 1, 2, 4 und Divx. Ihr Vorteil liegt in der sehr hohen Kompression, die sie dem Benutzer bieten, allerdings haben sie auch diverse Nachteile. Zum anderen arbeiten sie verlustbehaftet, d.h. die komprimierten Daten sind in den meisten Fällen nach einer Dekompression nicht mehr exakt mit den Originaldaten identisch. Je nach Aggressivität der Kompression ergeben sich merklich Mängel in der Bildqualität. Dies ist aber für Anwendung nicht akzeptabel. Ein weiterer Punkt ist die enorme Rechenleistung, die bei der Dekompression der Daten anfällt, für heutige PCs ist das jedoch kein Problem. Für viele Multimediageräte gibt es auch spezielle Hardwarelösungen um den Rechenaufwand abzufangen. Die GBA-Hardware bietet aber weder hoher Rechenleistung, noch eine spezielle Hardwarelösungen für dieses Verfahren. Betrachtet man die Vergangenheit, in

welcher die Rechenleistung diesem System ähnlicher war. Stößt man auf dem Amiga und finden auch gleich einige für diesen Problem benutzbarer Ansätze. Das Standardformat zum Abspielen von Animationen auf dem Amiga nennt sich kurz Anim-IFF-Format. Es wurde von der Sparta Inc. entwickelt, um einfache Videosequenzen auf dem Amiga abspielen zu können. Später wurde es ein viel genutztes Format für Grafiker, die Animationen erstellen wollten. Deshalb ist es auch in allen wichtigen Grafikprogrammen auf dem Amiga verwendbar. Auf der PC Seite entwickelte sich später das GIF-Format mit der Unterstützung von animierten Bildern. Jedoch hatte es den Nachteil, dass es nicht frei benutzbar war und auch ein Patent auf den verwendeten Algorithmus (lzw) bestand. Das Amiga Anim-IFF Format, war von Anfang an auf mehrere Arten der Kodierungen vorbereitet. Die ersten 5 Arten der Kodierung sind die am meist verbreitesten:

- XOR Mode,
- Long Delta Mode
- Short Delta Mode
- General Delta Mode
- Byte Vertical Compression

Die erste Methode nennt sich XOR-Mode. Die eigentliche Kompression erledigt ein einfacher Lauflängen-Kodieralgorithmus (RLE). Der eigentliche Trick ist aber, über alle Daten des letzten angezeigten Frames und des aktuellen Frames ein Exklusiv-Oder anzuwenden (deshalb auch der Name der Methode). Dadurch ergibt sich ein großer Vorteil für die später angewendete Lauflängenkodierung, denn wenn die Daten des letzten Frames identisch sind mit dem des aktuellen Frames, ergibt sich der Wert 0. Wenn also mehrere Pixel in beiden Frames an den gleichen Positionen identisch sind, werden diese zu Null kodiert und können dann über eine Lauflängenkodierung sehr gut komprimiert werden. Bei der Dekompression können dann aus dem letzten Frame und den aktuellen Daten das neue Frame erzeugt werden.

Die nächsten vier Methoden, arbeiten etwas anders. Grundlegend beschreiben sie in ihren Daten nur noch die Änderungen zum letzten Frame. Von der Kompressionsleistung unterscheiden sie sich aber nicht großartig von der XOR-Methode, jedoch sind sie von der Dekomprimierdauer etwas schneller.

Wie sie erkennen können, ist der technische Aufwand für diese Methoden gering und bietet dennoch gute Ergebnisse in der Kompression. Ein wichtiger Punkt ist natürlich das Datenmaterial. Diese Algorithmen profitieren stark von gezeichneten Animationen, da hier die Änderungen zwischen den Frames relativ gering sind. Bei Videodaten sind diese Algorithmen fast unbrauchbar, da hier die Frames untereinander fast immer unterschiedlich sind. Besonders durch Rauschen und Bewegungen des Bildes entstehen Probleme die diese Algorithmen nicht lösen können. Hierfür müssen andere Ansätze beschritten werden. Doch das ist für unseren Fall nicht von Bedeutung, da bei vielen 2D-Spielen, Animationen gezeichnet sind.

Für unseren Fall sind diese Arten der Kodierung geeignet, da sie eine schnelle Dekompression ermöglichen und auch die Daten auf ein vernünftiges Maß zu reduzieren. In einigen Fällen kommt es allerdings vor, dass etwas mehr Rechenleistung zur Verfügung steht und deshalb auch eine noch bessere Kompression erwünscht ist. Aus diesem Grund, bietet mein Framework die

Möglichkeit, nach der schon ausgeführten XOR-RLE-Kompression, eine weitere Kompressionstufe einzuführen. Ich habe mich hier für eine LZSS-Kodierung entschieden, weil sie sehr effektiv und noch in annehmbarer Geschwindigkeit ausführbar ist.

7.4 Implementierung der Kodierungsalgorithmen

Als erstes möchte ich auf die Implementierung des RLE – Algorithmus sowohl im Komprimierer und Dekomprimierer eingehen bzw. auf seine Erweiterungen mit XOR.

Es gibt mehrere Arten eine Lauflänge zu kodieren, in den meisten Fällen werden 8-Bit-Werte verwendet und ein gesonderter Schlüssel um bestimmte Lauflängen im Datenstrom zu markieren. Da der Einsatz der RLE-Dekompression meist das Ziel VRAM (Grafikspeicher) hat, habe ich eine Wortbreite von 16 Bit gewählt. Das beruht einfach auf der Tatsache, dass Schreibzugriffe in den VRAM nicht byteweise erfolgen dürfen. Sicherlich entsteht dadurch auch eine minimal geringere Kompression der Daten, da diese eigentlich byteweise organisiert sind. Aber der Gewinn an Geschwindigkeit durch 16 Bit Zugriffe ist höher als der geringe Verlust der Kompression im Vergleich zu einer 8 Bit Kodierung. Ein weiterer Vorteil ergibt sich bei 16 Bit Datentypen für das Schlüsselwort (Key), er wird von der Kompression über ein Histogramm ermittelt. Er charakterisiert das am wenigsten benutzte Zeichen im Eingabestrom und hat in den meisten Fällen eine Auftrittswahrscheinlichkeit von Null. In Abbildung 7-1 ist der Ablauf der Kodierung beschrieben. Es werden alle Wiederholungen größer drei, in eine Lauflänge der Art {Key,Anzahl-1,Wert} kodiert. Alle anderen Werte, mit Ausnahme des Schlüsselwertes (Key), werden normal kopiert. Der Schlüsselwert selber wird {Key,0,Key} kodiert, die sollte aber in extrem seltenen Fällen auftreten. In Abbildung 7-2 ist der Ablauf für die Dekompression dargestellt.

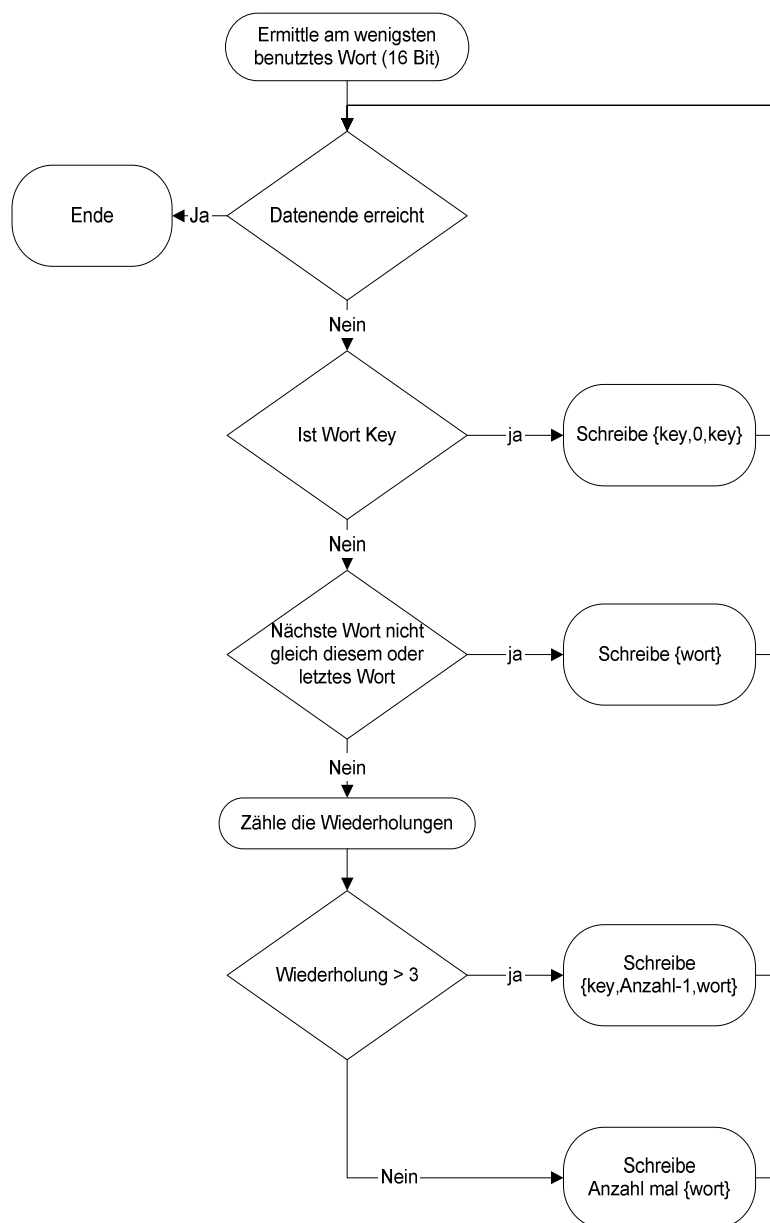


Abbildung 7-1 RLE-Kompression

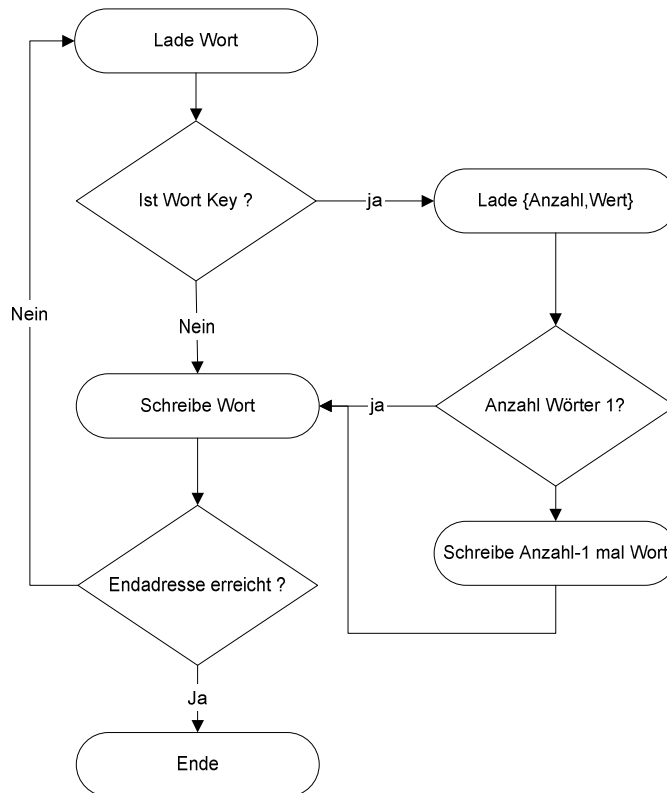


Abbildung 7-2 RLE-Dekompression

Dieser Algorithmus wird in der späteren Dekomprimerroutine auch nur einmal verwendet, nämlich zur Dekodierung des Keyframes (das erste Bild). Es stellt sich die Frage, warum es nur einmal benutzt wird, zumal er doch bei jedem Frame benutzt werden sollte. Natürlich tritt RLE – Algorithmus bei jeder Framedekodierung auf. Aber es wäre nicht sinnvoll gewesen ihn, von einem Puffer in den Anderen schreiben zu lassen und dann zusätzlich noch einen XOR über die Daten zu bilden. Hierfür ist es möglich, den RLE - Algorithmus zu erweitern um, Speicherplatz und Performance zu sparen. Abbildung 7-3 zeigt die Implementierung für die RLE – XOR – Dekompression der Frames. Sie unterscheidet sich zu vorher genannten Methode darin, dass sie zwei Quellen hat. Zum einen die zu dekomprimierenden Daten, zum anderen die Daten des letzten Frames. Der Vorteil liegt auf der Hand, er kann gleich bei der Dekomprimierung der Daten eine XOR-Verknüpfung mit dem alten Framedaten ausgeführt werden.

Abbildung 7-3 verdeutlicht diesen Ablauf. Die farbig markierten Stellen sind nicht unbedingt notwendig für eine korrekte Funktion des Algorithmus, erreichen aber eine höhere Performance. Sie bearbeiten den Sonderfall, dass bei einer Lauflänge der Wert Null auftritt, dieser Fall bedeutet nichts anderes, als das Quelle und Ziel identisch sind. Jetzt ergeben sich wiederum zwei Möglichkeiten zum optimieren. Sind Quelle (altes Frame) und Ziel nicht identisch, werden einfach die Daten kopiert. Ich habe hier ein Limit von 20 Worten gewählt, da ein Anstoßen der DMA auch etwas Zeit benötigt. Sollte aber Quelle (altes Frame) und Ziel identisch sein, werden die Daten einfach übersprungen, da sie schon existieren. In den meisten Fällen wird dieser Zweig

beschritten, da ein doppelter Puffer nur selten notwendig ist. Der Performancegewinn ist in diesem Fall enorm, da gleiche Daten übersprungen werden und nur Unterschiede zwischen den Frames errechnet werden und selbst diese können in einer Lauflänge zusammengefasst sein.

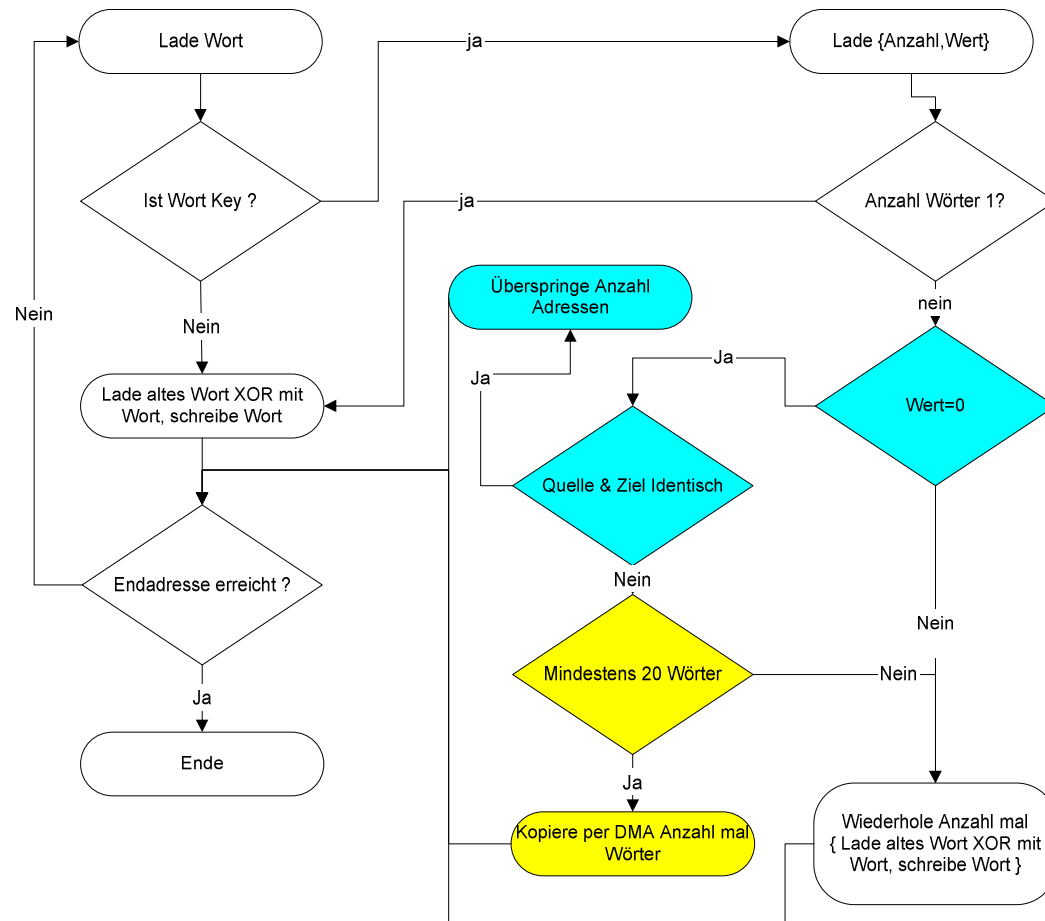


Abbildung 7-3 RLE Dekompression mit Quelle-Ziel Test und DMA Transfer

Für viele Fälle der Kompression von animierten Grafiken, sollten diese RLE - Algorithmen ausreichend sein. Jedoch nicht in allen Fällen, besonders dann, wenn etwas mehr Rechenzeit zur Verfügung steht, ist eine bessere Kompression sinnvoll.

Um diesen Punkt zu erfüllen, habe ich noch zusätzlich an die schon gezeigten Verfahren, einen Wahlweisen aktivierbare LZSS – Kodierung hinzugefügt. LZSS steht für Lempel-Ziv-Storer-Szymanski und kann als Nachfolger von LZ77 und Vorgänger von LZW (Lempel-Ziv-Welch) betrachtet werden. Der Grundgedanke von LZSS ist der Aufbau von einem Ringpuffer, der schon vorhandene Zeichenfolgen enthält. Bei der Kompression wird überprüft, ob schon vorhandene Daten im Ringpuffer enthalten sind und falls dies zutrifft, wird nur die Länge und die Position der im Ringpuffer liegenden Daten kodiert. So einfach dieses Verfahren ist, so effektiv ist es auch, da der Ringpuffer sich bei der Kompression schnell an ändernde Zeichenfolgen anpassen kann.

Beim LZSS - Algorithmus gibt es drei zentrale Konstanten: die Größe des Ringpuffers, die maximale Länge einer Zeichenfolge - mehr Zeichen werden beim Vergleichen einfach nicht herangezogen - und die Mindestlänge einer Zeichenfolge. Diese drei Konstanten werden wie folgt benannt: maximal Länge und minimal Länge. Der Wert minimal Länge-1 wird im Allgemeinen als Schwelle (Threshold) bezeichnet.

Weiterhin erzeugt die Kompression entweder die Originaldaten oder eine Längenangabe, gefolgt von einer Adressangabe. Damit der Dekompressor die erhaltenen Informationen auch verstehen kann, muss er zwischen einem normalen Zeichen und einer Längenangabe unterscheiden können. Dazu wird einfach die Zeichenmenge um Zeichen für die Längenangaben erweitert. Wie bei der RLE - Kodierung wäre auch ein Zeichen für „Längenangabe folgt“ verwendbar, das ist aber eigentlich auch nichts anderes, als eine Erweiterung der Zeichenmenge.

Im Folgenden möchte ich die Dekompression zur Verdeutlichung des Problems in Pseudocode beschreiben.

<p>Wenn normales Zeichen kommt dann (nicht reserviert)</p> <p> Lies Zeichen ein und gib das Zeichen aus</p> <p> Schreibe das Zeichen in Puffer an der stelle X</p> <p> Inkrementiere X</p> <p>Wenn das Zeichen reserviert war</p> <p> Lies Länge und Offset ein</p> <p> Wiederhole Länge mal</p> <p> Lade Zeichen aus Puffer an Stelle X – Offset</p> <p> Gib Zeichen aus</p> <p> Speichere Zeichen in Puffer an Stelle X</p> <p> Inkrementiere X</p>
--

Abbildung 7-4 Dekompression von LZSS Daten

Um den Algorithmus nicht neu implementieren zu müssen, baue ich auf einer schon vorhandenen Version von *Haruhiko Okumura*³⁵ auf, die ich nur minimal verändert und in ARM-Assembler optimiert habe.

Um jetzt den Unterschied zwischen RLE – XOR und RLE – XOR – LZSS zu verdeutlichen, habe ich mit meinen später vorgestellte Konverter eine Animation mit 861 Frames und einer Gesamtgröße von ca. 33 Megabyte komprimiert.

	unkomprimiert	RLE - XOR	RLE - XOR - LZSS
Größe in Byte	33062400	4566668	1590304
Kompressionsrate in %	0,00	86,19	95,19
Durchschnittliche Größe pro Frame in Byte	38400	5304	1847

Abbildung 7-5 Kompressions-Statistik

³⁵ URL: <http://asp.itdrp.com/ZaneStudio/zarticle/LZSS.htm> (27.07.2004)

Die Zahlen sprechen für RLE – XOR – LZSS, es darf aber nicht der Rechenzeitverlust vergessen werden, den der LZSS – Dekoder in sich birgt. Je nach Anwendungsfall muss der Benutzer selber entscheiden welches Verfahren er wählt.

7.5 Funktions- und Parameterbeschreibung

Dem Benutzer stehen zum Dekodieren der mit RLE und XOR-RLE komprimierten Daten, zwei Funktionen zur Verfügung.

Funktionsname:	rledec_th
Beschreibung:	Standard - RLE Dekoder
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r1 = Länge des Datenstroms r2 = Schlüsselwert (Key) r3 = Quelladresse r4 = Zieladresse

Funktionsname:	rlexordec_th
Beschreibung:	erweiterte RLE Dekoder der intern ein XOR über die Daten ausführt
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r1 = Länge des Datenstroms r2 = Schlüsselwert (Key) r3 = Quelladresse r4 = Zieladresse r7 = Quelladresse des letzten Frames

Für die LZSS Dekompression stelle ich zwei Dekoder zur Verfügung. Sie unterscheiden sich nur dadurch, dass einer in Thumbcode geschrieben wurde und im ROM liegt, der zweite ist in normalen ARM-Code geschrieben und befindet sich im IWRAM. Der zweite Dekoder ist schneller als der Erste, benötigt aber auch IWRAM. Der Benutzer kann selbst festlegen, für welchen er sich entscheidet.

Funktionsname:	lzss_dec
Beschreibung:	lzss Dekoder
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = die Adresse der Eingangsdaten stehen r1 = die Adresse der Ausgangsdaten r2 = die Länge addiert mit der Eingangsadresse r3 = Adresse zum Ringpuffer

Lzss_dec_th ist die Thumbversion und hat die gleichen Parameter.

Funktionsname:	lzss_dec_th
Beschreibung:	Lzss Dekoder
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = die Adresse der Eingangsdaten stehen r1 = die Adresse der Ausgangsdaten r2 = die Länge addiert mit der Eingangsadresse r3 = Adresse zum Ringpuffer

Zu beachten ist, dass der Ringpuffer eine Größe von 4116 Byte haben sollte und vor der Benutzung des Dekoders mit Nullen initialisiert werden muss. Dafür kann die von mir zur Verfügung gestellte Funktion memclear32 (Siehe Kapitel 5.2.1) benutzt werden.

7.6 Das Datenformat

Die komprimierten Daten müssen auf ein vernünftiges Format zugeschnitten werden, um den Benutzer ein komfortables Lesen und Verarbeiten der Daten zu ermöglichen. Abbildung 7-6 beschreibt die von meinem später beschriebenen Tool (Siehe Kapitel 7.7) erzeugte Struktur. Sie ist im Aufbau einfach und erlaubt nur eine eingeschränkte Flexibilität. Das stellt an sich keinen Nachteil dar, besonders weil diese Struktur nur für diesen Spezialfall erzeugt wird und ein schnelles Lesen und Verarbeiten wichtig ist.

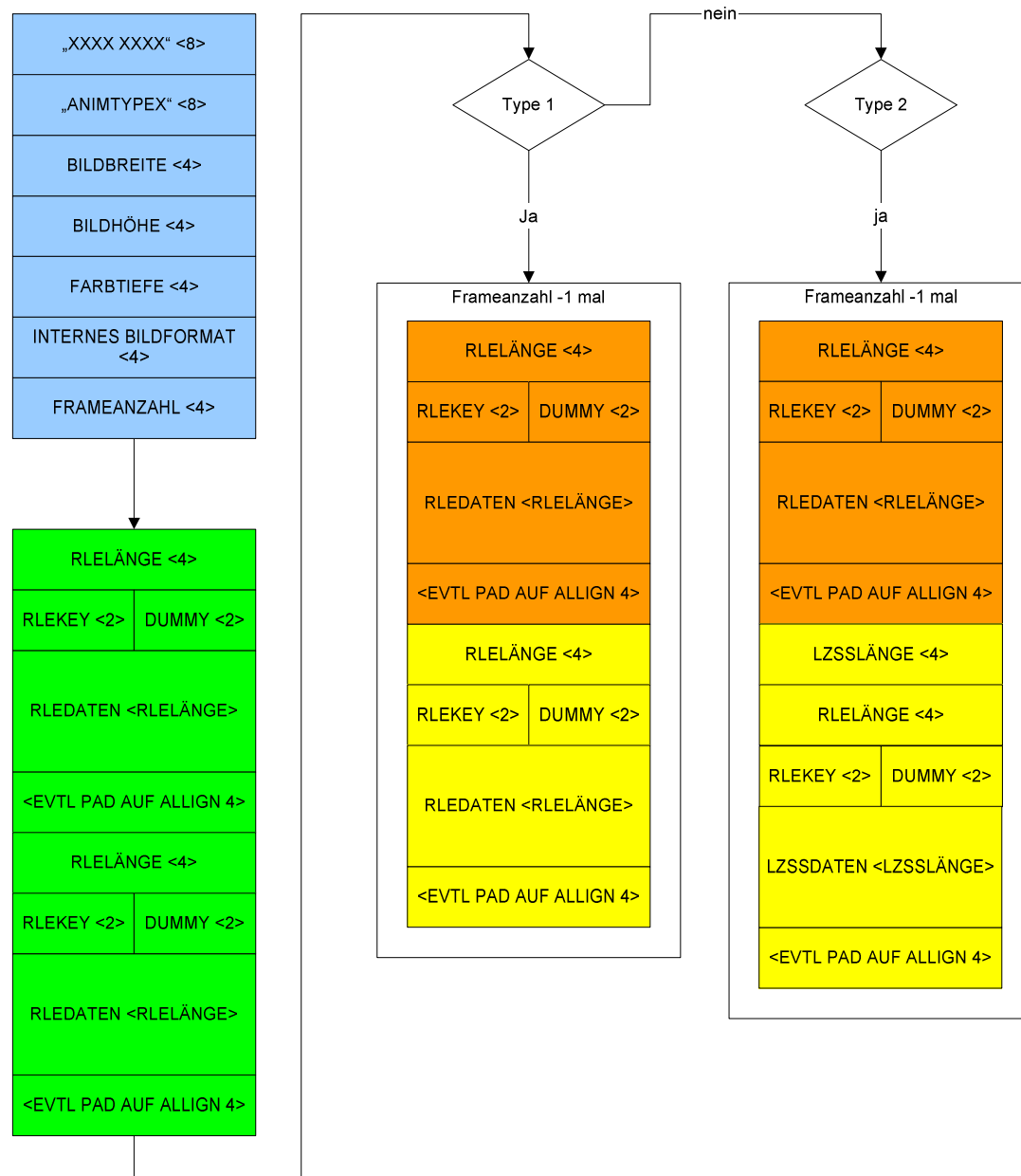


Abbildung 7-6 Datenformat

Das Format besteht im Wesentlichen aus drei großen Teilbereichen (siehe Abbildung 7-6), dem Header (Anfangsdaten, blau markiert), dem Firstframe (Quellbild, grün markiert) und den restlichen Frames (orang, gelb markiert). Der Header und das Firstframe sind vom Kompressions-Typ unabhängig immer vorhanden. Die restlichen Frames sind vom Kompressionstyp abhängig, in unterschiedlichen Strukturen definiert (Typ 1 oder Typ 2).

Der Header beschreibt die Grunddaten der Animation. Die ersten 8 Bytes sind frei wählbar und nur zu Identifikation gedacht, die nächsten 8 Bytes enthalten das Wort „Animtyp“ gefolgt von einem ASCII-Zeichen das den Wert „1“ oder „2“ annehmen kann. Die beiden Werte definieren die spätere Framekompression. Wobei „1“ für XOR – RLE steht und „2“ für XOR – RLE – LZSS. Dann kommen in je 4 Byte verpackt die Bildbreite, Bildhöhe und die Farbtiefe der Frames. In den nächsten 4 Byte wird das Grafikformat an sich beschrieben, hierfür gibt es momentan nur zwei Zustände. Mit dem Wert 0 wird ein normaler serieller Datenstrom definiert,

ist der Zustand aber 1, dann handelt es sich um Datenstrom der in einer 8*8 großen GBA-Teileform existiert. Danach folgt noch eine maximal 4 Byte große Zahl, welche die Frameanzahl der Animation beschreibt.

Nach dem Header kommt das Referenz-Bild (Firstframe), es besteht aus den Palettendaten. Also die Daten, welche die Farben des Bildes beschreiben, im Normalfall sollten es immer 256 Farben sein, gefolgt von den eigentlichen Bilddaten. Beide Blöcke sind RLE komprimiert. Wodurch sich auch für beide Blöcke folgender Datenablauf ergibt. Der erste Wert enthält die Länge der komprimierten Daten und ist 4 Byte lang, gefolgt wird er von dem 16 Bit breiten RLE - Schlüssel und einen 2 Byte breiten nicht benutzten Wert. Jetzt kommen die eigentlichen komprimierten Daten. Falls die Daten jetzt nicht auf eine durch vier teilbare Adresse ausgerichtet sind, werden noch leere Bytes ergänzt um eine „Allignement“ von vier zu erreichen. Diese Ergänzung ist später wichtig, damit die Daten auch in vier Byteschritten eingelesen werden können. Derselbe Datenablauf erfolgt noch einmal für die reinen Bilddaten.

An diesem Punkt befinden wir uns an einem Scheideweg. Die folgenden Daten sind vom anfänglich angegebenen Typen abhängig. Ist unser Animationstyp vom Type 1, dann folgen jetzt die Daten nach dem Prinzip des Firstframes, nur dass ihre Kompression nicht nur RLE ist sondern XOR – RLE. Befinden wir uns aber im zweiten Modus, entsteht ein geringer Unterschied in der Bildstruktur. Vor der RLE – Länge, befindet sich nun noch ein 4 Byte großer Wert, in dem die Länge der LZSS Daten gespeichert ist.

Praktisch gesehen, kann dieses Format noch um weitere Blöcke ergänzt werden, aber trotzdem eine Abwärtskompatibilität garantieren. Aber wie schon angemerkt, ist eine Erweiterung für den Normalfall unwahrscheinlich, da es sich um ein sehr spezifisches Problem handelt.

Als letzten Punkt ist noch darauf hinzuweisen, dass sich die Daten im Little - Endian - Format befinden. Davon sind also alle Informationen betroffen außer den LZSS – Daten, da hier in Bytes gearbeitet wird.

7.7 Das Konvertertool

Um den Benutzer die Möglichkeit zu geben, seine Daten in eine nach dem von mir spezifizierten Format umzuwandeln, habe ich ein Konvertertool (conv_anim2.exe³⁶) geschrieben. Es lädt mindestens zwei BMP-Bilder und fügt diese zu einer Animation zusammen. Weiterhin ermöglicht es das Kompressionsverfahren zu wählen und eine Bildtransformation in das GBA-Tile-Format durchzuführen.

³⁶ Programm: /source/anim/conv_anim/conv_anim2.exe

Name:	conv_anim2.exe
Synopsis:	conv_anim2.exe {1[t] 2[t]} [files.bmp] {outputfile}
Parameter:	"-1" RLE-XOR-Kompression "-2" RLE-XOR-LZSS-Kompression "t" Transformation ins Teileformat
Beispiel:	./conv_anim2.exe -2t frames2/bmps/anim0* output.anim

Der erste Parameter kann den Wert 1, 1t, 2 oder 2t besitzen. Wobei die Zahl 1 und 2 den Kompressionsmodus (XOR – RLE oder XOR – RLE – LZSS) angibt. Das optional nachgestellte „t“ gibt an, ob eine Teile-Transformation durchgeführt werden soll. Jetzt folgen die Eingangsbilder, welche vom Format BMP sein müssen. Die letzte Datei die angegeben wird, ist die zu erzeugende Ausgabedatei.

Eine erfolgreiche Eingabe könnte so aussehen (inkl. Ausgabe):

```
$ ./conv_anim2.exe 2 frames2/bmps/anim0* output.anim

ANIM-TYPE(1-2) Compressor
-----

.start compress 861 files to output.anim (type 2)
.reading files 861
.reading and decoding file 2
.using format 240x160x8
.reading and decoding file frames2/bmps/anim0859.bmp
.make first frame
.frame compress
.compress 859 with 860 (delta+rle+lzss)

Statistics
Frames      = 861
Inputsize   = 33062400 bytes
Outputfilesize = 1590304 bytes
Compressed to 4 %
Size per Frame ~ 1847 bytes (42.98^2 Pixel)
```

Abbildung 7-7 Beispielausgabe des Konverter-Tools

Das Programm selber habe ich in der Programmiersprache C – entwickelt. Es besteht aus folgenden Dateien³⁷ (Komponenten):

color.c	(Farbtransformation von 24 auf 15 Bit)
delta.c	(Bildet XOR-Werte zwischen Frames)
lzss.c	(Komprimiert Daten nach LZSS)
main.c	(Eingabeabwicklung, zentrale Steuerung)
readbmp.c	(Ließt BMP-Dateien)
rle.c	(RLE – Kompression)
tilefilter.c	(Teile-Transformation)

³⁷ Pfad: source/anim/conv_anim/

Ich möchte nicht näher auf die Implementierung dieses Tools eingehen, da es für den Benutzer eher unwesentlich sein sollte, bzw. der wesentliche Teil (Kompression) schon in den vorherigen Kapiteln angesprochen wurde.

7.8 Anmerkung

Da die Kompressionsroutinen auch in anderen Bereichen genutzt werden können, habe ich zusätzlich zwei getrennte Programme zur Kompression (LZSS, RLE) entwickelt. Die Dekompression erfolgt mit den bereits vorgestellten Funktionen.

Der Programmaufruf zur Kompression ist für beide Komprimierprogramme identisch. Erwartet werden eine Quell- und eine Zielfeile. Informationen zum Kompressionsgrad werden im Terminal ausgegeben.

Name:	rlpack.exe
Synopsis:	rlpack.exe {inputfile}{outputfile}
Beispiel:	rlpack.exe background.raw background.raw.rle

Name:	lzsspack.exe
Synopsis:	lzsspack.exe {inputfile}{outputfile}
Beispiel:	lzsspack.exe background.raw background.raw.rle

Die Datenstruktur der erzeugten Daten wird in Abbildung 7-8 beschrieben. Die 8 Byte große Identifikation kann vom Anwender frei gewählt werden.

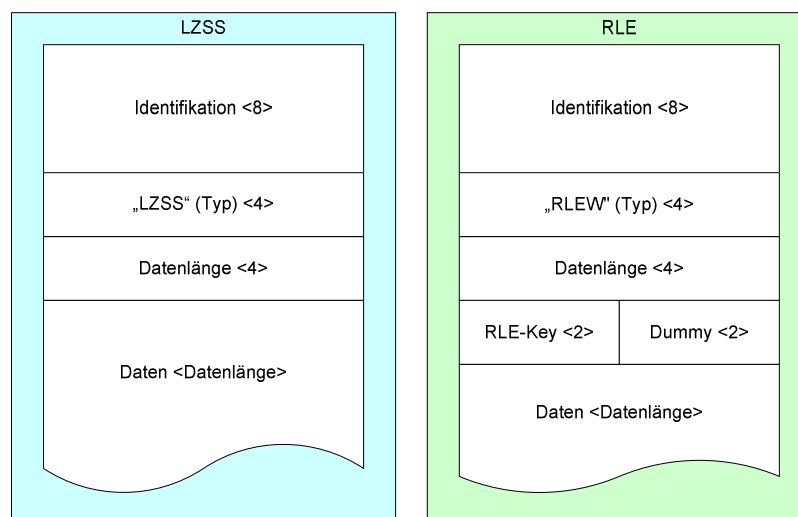


Abbildung 7-8 Datenformat

7.9 Fazit

Mit den vorgestellten Programmen sollte es dem Benutzer möglich sein, komfortabel Animationen in sein Spielprojekt zu integrieren, wobei er je nach Anforderung zwischen verschiedenen Kompressionsarten wählen kann. Das Animationsystem ist für zukünftige Erweiterungen offen und kann nach je nach Bedarf verändert werden. Ich habe absichtlich auf Funktionen verzichtet, welche die gesamte Animation kapseln. Der Benutzer muss am Ende noch selber entscheiden wohin er die Daten kopiert haben möchte oder ob er evtl. mehrere Puffer verwendet. Das hat zur Folge, dass das System zwar nicht geschlossen ist und somit nicht trivial zu benutzen, aber eine hohe Flexibilität der Komponentenverteilung bietet. Z.B. können die Dekompressionsalgorithmen auch für andere Daten verwendet werden.

Eine Beispielimplementierung zum Abspielen von Animationen des Typs 1 und 2 ist unter dem Verzeichnis `source/anim/anim_test` zu finden.

8 Kollisionserkennung

8.1 Allgemeine Einführung in die Problematik

Die Kollisionserkennung ist ein sehr wichtiges Thema in der Spielentwicklung. Bei der nähern Betrachtung der Thematik ist festzustellen, dass es fast keine Spiele ohne Kollision von Objekten mit anderen Objekten gibt, wobei ein Objekt z.B. auch ein Hintergrund darstellen kann. Das eigentliche Ziel der Kollisionserkennung ist, Objekte zu erkennen, die sich mit anderen Objekten überlappen.

Im ersten Schritt muss die Beschreibung der Dimensionen eines Objektes erfolgen. Im Bereich der zweidimensionalen Spiele sind grafische Objekte meist Sprites. D.h. sie sind einfache Grafiken mit einer festgelegten Größe mit der Erweiterung, dass sie in der Lage sind, bestimmte Pixel ihrer Grafik zum Hintergrund transparent zu erzeugen. Dadurch sind die Grenzen des Objektes schon definiert, denn alle Pixel, die nicht transparent sind, gehören zu dem Objekt.

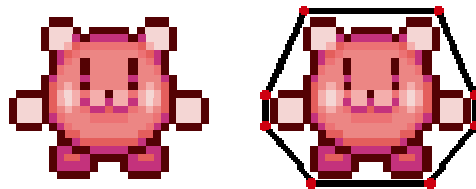


Abbildung 8-1 Pixelobjekte

Eine weitere Möglichkeit ist, das Objekt als Polygon zu betrachten. Um jetzt Pixelgrafik mit einem Polygon zu ersetzen, muss das Objekt so weit abstrahiert werden, bis man einen für sich ausreichenden Kollisionsbereich definiert hat. Im Regelfall umspannt das Polygon die eigentliche Pixelgrafik.

Betrachten wir jetzt die Kollision der beiden Modelle.

Im Fall der Pixelgrafik ergibt sich bei einer Kollision eine Überlappung der Grafik von Objekt 1 und Objekt 2. (Siehe Abbildung 8-2).



Abbildung 8-2 Pixel-Kollision

Diese Art der Kollisionserkennung ist pixelweise genau, hat aber auch den sehr großen Nachteil, dass ein Test im schlechtesten Fall über alle Pixel beider Objekte ausgeführt werden muss, solange sie sich nicht überlappen.

Einige Architekturen bieten diesen Test schon in Hardware an (z.B. der Amiga-Computer), leider steht diese Funktionalität auf dem GBA nicht zur Verfügung und müsste so softwaretechnisch nachgebildet werden. Bei wenigen und sehr kleinen Objekten ist der Aufwand noch in einem vertretbaren Maß zu halten, wobei einige Besonderheiten der Pixelgrafik zu beachten sind. Bei dem GBA können diese hardwareseitig noch rotiert und gezoomt werden. Somit wird das Überprüfen der Pixelkollision noch komplexer. Schlussendlich kam ich zu dem Resultat, dass bei diesem Verfahren die Nachteile den Vorteilen überwiegen. Aus diesem Grund, wird von einer Implementierung von Funktionen, die dieses Problem berücksichtigen, abgesehen.

Deshalb werden sich meine im Nachfolgenden gezeigten Funktionen immer auf einen Punkt und ein Polygonmodel beziehen. Im ersten Moment scheint es so, als wäre das Polygonmodel zum Pixeltestmodel sehr ungenau, da das Polygonmodel in den meisten Fällen ungenauer ist. Natürlich ist es mit etwas Aufwand auch möglich, das Pixelobjekt als Polygon zu beschreiben. Im schlechtesten Fall müsste der gesamte Rand des Objektes von Pixel zu Pixel durch einen Verbindungen ersetzt werden. Allerdings ist es im Normalfall nicht immer notwendig, eine so hohe Genauigkeit zu erreichen.

Widmen wir uns nun der eigentlichen Erkennung einer Kollision zwischen Polygonen.

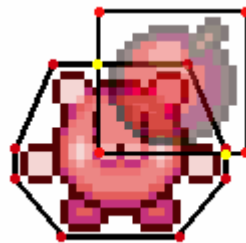


Abbildung 8-3 Polygon-Kollision

Wie in Abbildung 8-3 zu erkennen ist, handelt es sich in dieser Abbildung um zwei Objekte, die durch je ein Polygon definiert sind. Wenn sich diese beiden Polygone Überlappen, entsteht auch ein Schnittpunkt zwischen den Linien der Polygone. Wenn sich keine der Linien dieser Polygone schneiden, bedeutet das, dass sich entweder eins der beiden Objekte außerhalb des anderen befindet oder innerhalb des Objektes.

Im Fall, dass sich das Objekt innerhalb eines anderen befindet, ist auch eine Kollision erfolgt. Bei diesem Sonderfall muss ein weiterer Test erfolgen. Das klassische Linien-Schnittmodel wirkt hier nicht. Eine sehr einfache Lösung ist, zu überprüfen, ob ein Punkt eines Polygons innerhalb des anderen liegt. Wenn dies für beide Polygone angewandt zutrifft, befindet sich das eine Polygon nicht innerhalb des anderen.

Diese „Punkt im Polygon“ Verfahren kann auch für einige Fälle der Kollisionserkennung in Spielen genutzt werden, besonders wenn es sich um Objekte handelt, die ein Gebiet markieren sollen. Betrachtet man Abbildung 8-4, ist zu erkennen, dass ein Gebiet durch ein Polygon definiert wird. Das Punkt-Objekt (das Männchen) soll sich aber nur in dem Bereich, der durch das Polygon definiert ist, bewegen. Also ist ein Punkt für das Männchen zu definieren, wenn

dieser z.B. zwischen den Füßen definiert wird, wäre eine vernünftige Kollisionserkennung für dieses Problem möglich.

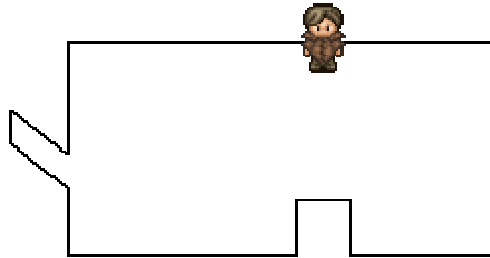


Abbildung 8-4 Punkt in Polygon

Der Vorteil an diesem Verfahren ist, dass es nicht notwendig ist, für bestimmte Objekte ein Polygon-Modell zu erzeugen. Sondern ein einfaches Punktmodell ausreichend ist und dadurch Rechenleistung gespart werden kann.

8.2 Sphärische Kollision

Bevor diese Methoden genauer beschrieben werden, möchte ich noch auf eine weitere Methode hinweisen, welche in einigen Fällen auch ausreichend für eine Kollisionserkennung ist, aber im Endeffekt auch immer der Vorläufer bei der normalen Kollisionserkennung ist. Es handelt sich dabei um eine sphärische Kollision.

Die Idee ist, dass ein Objekt einen Mittelpunkt und einen Radius besitzt. In dem dadurch erzeugten Kreis befinden sich im Normalfall alle Pixel des Objekts. Um jetzt auf die Kollision eines anderen Objektes mit diesen zu prüfen, muss der Abstand der beiden Mittelpunkte (g_x, g_y) kleiner oder gleich der beiden Radien (r_1, r_2) sein.

Für den Fall das eine Kollision eingetreten ist, gilt dies:

$$\sqrt{g_x^2 + g_y^2} \leq r_1 + r_2$$

$$g_x^2 + g_y^2 \leq (r_1 + r_2)^2$$

Die Wurzel kann man sich durch ein Quadrieren der Summe der Radien sparen. Der Hintergrund ist, dass eine einfache Multiplikation mit sich selbst wesentlich schneller ist, als eine Wurzel zu berechnen.

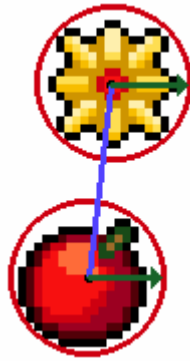


Abbildung 8-5 Sphärische-Kollision

In Abbildung 8-5 sind zwei Objekte zu erkennen (Apfel, Stern). Beide haben einen Mittelpunkt. Der grüne Pfeil in dem Objekte beschreibt den maximalen Radius. Dadurch ergibt sich ein Kreis, der das Objekt umgibt. Dieser Kreis nennt sich „die Sphäre“ des Objektes. Die blaue Linie charakterisiert den Abstand der beiden Mittelpunkte. Ich hatte zuerst erwähnt, dass diese Methode der Vorläufer der anderen Kollisionsarten ist. Denn bei einer Polygon-Polygon-Kollision müssen alle Verbindungen eines Polygons, mit denen der anderen überprüft werden, um eine Kollision zu erkennen. Weiterhin müssten noch je zwei Punkt-Polygontests erfolgen. Der dadurch entstehende Rechenaufwand ist sehr hoch, er ist sogar im Fall, dass keine Kollision auftritt, am höchsten. Allerdings ist im Normalfall eine Kollision eine Ausnahme und dadurch ist das Ergebnis nicht optimal.

In diesen Moment kann aber eine sphärische Vorüberprüfung sehr viel Rechenzeit ersparen. Der Gedanke ist, für jedes Polygonobjekt den Mittelpunkt und den Radius zu berechnen. Wenn dies geschehen ist, kann ein Sphäretest stattfinden und nur im Fall, dass sich die Sphären überschneiden, muss ein Polygon-Polygon-Test erfolgen. Der dadurch gewonnen Zeitgewinn ist enorm und nur im schlechtesten Fall, wenn alle Objekte miteinander kollidieren, etwas höher, als ein Test ohne Sphäreprüfung.

8.3 Polygon – Polygon Kollision

In den nächsten Kapiteln möchte ich etwas näher auf den mathematischen Hintergrund der Polygon-Polygon-Kollisionserkennung und dem „Punkt in Polygone“ Problem eingehen.

Wie bereits bekannt ist, beruht die Polygon-Polygon Kollisionserkennung auf dem Prinzip der Überprüfung der Punktverbindungen auf Schnittpunkte des einen Polygons mit dem Anderen.

Deswegen ist das primäre Problem zu erkennen, ob sich zwei Linien schneiden oder nicht. Wo genau sie sich schneiden, ist für unser Problem erstmal unwesentlich.

Definieren man zwei Liniensegmente \overline{AB} und \overline{ST} wobei gilt:

$$A = (x_1, y_1) \quad B = (x_2, y_2) \quad S = (x_3, y_3) \quad T = (x_4, y_4)$$

$$P_1 = A + t_1(B - A) \quad 0 \leq t_1 \leq 1$$

$$P_2 = S + t_2(T - S) \quad 0 \leq t_2 \leq 1$$

Durch Gleichsetzen von P1 und P2 erhalten man die beiden Unbekannten t_1 und t_2

$$x_1 + t_1(x_2 - x_1) = x_3 + t_2(x_4 - x_3)$$

$$y_1 + t_1(y_2 - y_1) = y_3 + t_2(y_4 - y_3)$$

Lösen man im folgenden Schritt nach t_1 und t_2 auf, ergibt sich folgendes:

$$d = (y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)$$

$$t_1 = [(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)] \div d$$

$$t_2 = [(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)] \div d$$

Betrachten man nun die gegebenen Zusammenhänge. Wenn $d=0$ ist, sind die beiden Linien parallel. Wenn t_1 und t_2 zudem auch Null sind (ohne durchgeführte Division), sind die Linien identisch.

Falls d nicht Null ist und der Fall eintritt, dass $0 \leq t_1 \leq 1$ oder $0 \leq t_2 \leq 1$, dann Schneiden sich die Linien \overline{AB} und \overline{ST} . Durch Einsetzen des Wertes von t_1 oder t_2 in die Geradengleichung, ist auch der genaue Schnittpunkt ermittelbar (in vielen Fällen nicht notwendig).

$$x_s = x_1 + t_1(x_2 - x_1)$$

$$y_s = y_1 + t_1(y_2 - y_1)$$

Wenn also diese Schritte für alle Punktverbindungen der beiden Polygone vollzogen wurden, kann eindeutig festgestellt werden, ob sich Linien der beiden Polygone schneiden bzw. gleich sind. Abbildung 8-6 zeigt eine Testimplementierung (mit ganzzahligen Variablen) in BlitBasic.

```

Function line_x_line (x1,y1,x2,y2,x3,y3,x4,y4)

power = 12 ;Frac. in Bits
powern = 4096 ;2^
nix = 0 ;Return-Wert (0=kein Schnitt)

If(x1=x3 And y1=y3 And x2=x4 And y2=y4) Then Return nix ;identisch

y13 = y1-y3
y21 = y2-y1
y43 = y4-y3

x13 = x1-x3
x21 = x2-x1
x43 = x4-x3

den = (y43*x21) - (x43*y21)
t1nd = (x43*y13) - (y43*x13)
t2nd = (x21*y13) - (y21*x13)

If den=0 Then ;parallel
    nix=1
Else
    t1 = (t1nd Shl power)/den
    If (t1<=0) Or (t1>=powern) Then nix=1

    t2 = (t2nd Shl power)/den
    If (t2<=0) Or (t2>=powern) Then nix=1
End If

If nix=0 Then
    xs=x1+(t1*(x2-x1))Sar power ;Schnittpunkt
    ys=y1+(t1*(y2-y1))Sar power
End If

Return nix
End Function

```

Abbildung 8-6 prüfe zwei Linien auf einen Schnittpunkt

8.4 Punkt im Polygon

Da es wie bereits erläutert zu dem Sonderfall kommen kann, dass ein Polygon in einem anderen liegt, muss ein Punkt im Polygon-Test erfolgen. Der Test-Aufwand ist gering, da nur je ein Punkt von jedem Polygon gegen das andere getestet werden muss. Falls ein Test positive ausfällt, befindet sich ein Objekt in den anderen.

Es existieren mehrere Möglichkeiten, um einen Punkt im Polygon-Test durchzuführen. Betrachten man Abbildung 8-7.

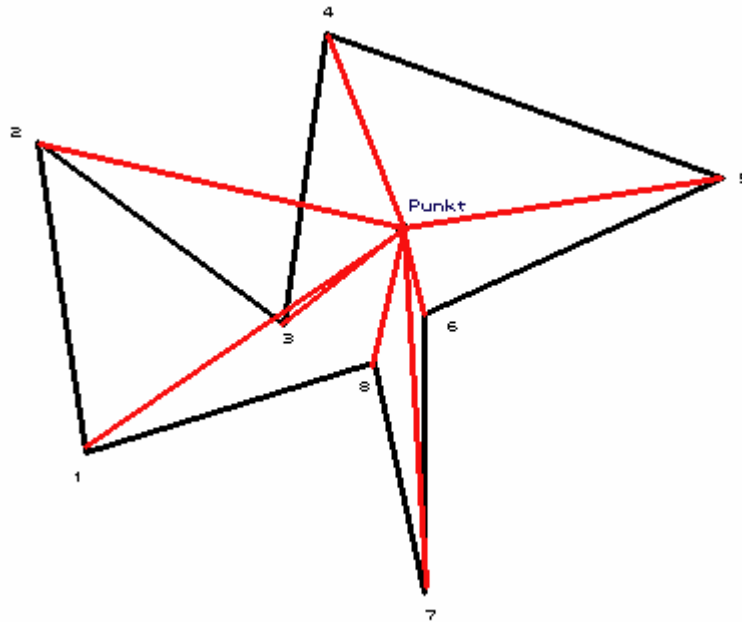


Abbildung 8-7 Winkel-Addition

Es handelt sich dabei um ein konkaves Polygon. Platziert man nun einen Punkt in das Objekt und verbindet diesen Punkt mit jedem Punkt des Polygons, so ergibt sich für jeden Eckpunkt des Polygons zu seiner Seite ein Winkel. Da sich dieser Punkt innerhalb des Polygons befindet (da konkav) ist dieser Winkel im Bereich von 0 bis 90 Grad. Sind jetzt alle Eckwinkel innerhalb dieses Bereiches, liegt der Punkt innerhalb des Polygons.

Diese Methode ist einfach, hat aber einige Nachteile, zum einen gilt diese Abhängigkeit nur für konkave Polygone. Leider sind aber viele Objekte konvex. Weiterhin müssen n -Winkel (bei n -Ecken) berechnet werden. Dieser arithmetische Aufwand ist für unsere Möglichkeiten nicht akzeptabel.

Untersuchen wir eine andere Methode, welche die Möglichkeit gibt, konvexe Objekte zu benutzen. Die Idee ist, dass die Summe der Winkel, die von einem Punkt in einem konvexen Objekt mit den Eckpunkten verbunden sind, gleich 360° sind.

Betrachten wir uns Abbildung 8-8. Ein Punkt liegt innerhalb eines Objektes, die roten Linien symbolisieren die Verbindungen mit den Eckpunkten. Summiert man die Winkel zwischen diesen

und den Vertex auf, erhält man einen Gesamtwinkel von 360° . Liegt aber der Punkt außerhalb, ergibt sich einen Winkel ungleich 360° .

Dieses Verfahren ist leider immer noch sehr unbefriedigend für das Kollisions-System, denn wiederum müssen Winkel berechnet werden, die einen enormen rechnerischen Aufwand bedeuten würden. Für Rechner mit schneller Fließkommaeinheit, wären solche Arten von Berechnungen akzeptabel.

Verlassen wir die Welt der Winkel und widmen uns etwas abstrakteren Lösungen.

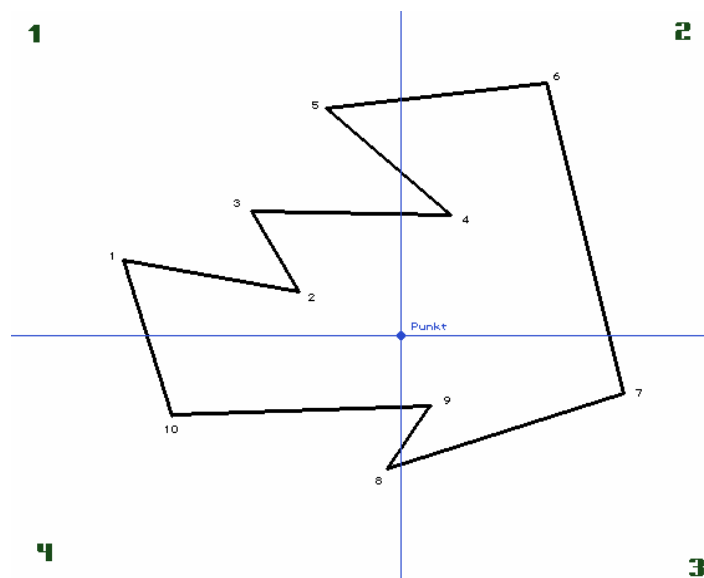


Abbildung 8-8 Quadranten-Test

Betrachten wir Abbildung 8-8, Ein Punkt wird horizontal und vertikal durchlaufen und zerschneidet dadurch das Polygon in 4 Quadranten.

Das Polygon hat also im Fall, das es konkav ist vier Schnittpunkte, sobald der Punkt in ihm liegt. Wenden wir diese Theorie auf konvexe Objekte an, hier muss bei dem Übergang von einen niedrigen in einen höheren Quadranten auf einen Zähler (mit 0 initialisiert) 1 addiert werden, falls jedoch wieder ein Rücklauf (hoher zu niedriger Quadrant) geschieht, wird eins vom Zähler subtrahiert. Wenn alle Punktverbindungen durchlaufen wurden und der Zähler auf 4 steht, liegt der Punkt im Polygon.

Für das Objekt in Abbildung 8-8, würde so vorgegangen werden. Von Punkt 1 nach Punkt 2 und Punkt 2 nach Punkt 3 wird keine Quadrantengrenze durchlaufen. Von Punkt 3 zu 4 erhöht sich der Quadrant also somit auch der Zähler um 1. Von 4 zu 5 wird wieder der Quadrant 1 erreicht, also muss subtrahiert werden. Bei Punkt 5 zu 6 muss aber wieder addiert werden. Selbiges gilt für 6 zu 7 und 7 zu 8. Weiterhin folgt eine Subtraktion von 8 zu 9, eine Addition von 9 zu 10 und einer weiteren Erhöhung bei 9 zu 10. Jetzt hat der Zähler den Wert 4, dadurch ist eindeutig festgestellt worden, dass der Punkt im Polygon liegt.

Dieses Verfahren ist sehr schnell und für die meisten Zwecke ausreichend. Allerdings gibt es noch ein Verfahren, dass schneller ist.

Die Idee bei diesem Algorithmus ist es, sich einen zweiten Punkt vorzustellen, der definitiv nicht im Polygon liegt. Verbindet man diese beiden Punkte ergibt sich eine Linie. Diese Linie wiederum schneidet mindesten einmal das Polygon. Werden bei diese Schnitte gezählt ergibt sich folgendes: Bei einer geraden Anzahl, liegt der Punkt außerhalb des Polygons. Bei einer ungeraden Anzahl, liegt er innerhalb des Polygons. In Abbildung 8-9, ist dieses Szenario dargestellt.

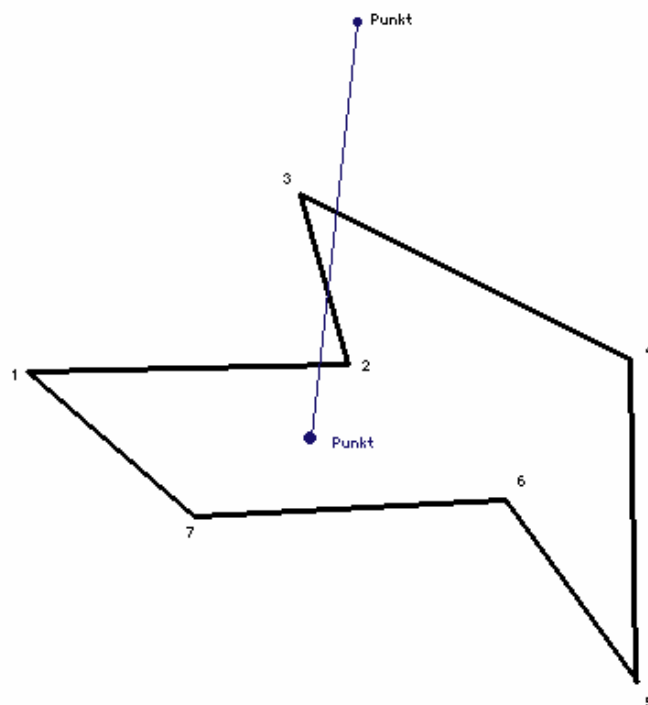


Abbildung 8-9 Schnitt-Test

Folglich muss das Programm über alle Verbindungen laufen und überprüfen ob ein Schnitt vorliegt. Im nachfolgenden Basic (BlitzBasic³⁸) Quellcode wurde der Algorithmus für diese Verfahren implementiert. Ich verwende oft vor der eigentlichen Assemblerprogrammierung ein Basic-Derivat, zum schnellen Austesten von Funktionen, besonders wenn sie graphisch dargestellt werden, bietet sich Basic besonders an. Vielleicht auch nicht ganz unwesentlich ist die Verwandtschaft von Basic und Assembler.

```
;npoints = Anzahl der Punkte des Objekts
;px und py sind die Koordinaten des Polygons

xold=px(npoints-1);
yold=py(npoints-1);

inside=0

For i=0 To npoints-1

    xnew=px(i)
    ynew=py(i)

    If (xnew > xold) Then
        x1=xold
        x2=xnew
        y1=yold
        y2=ynew
    Else
        x1=xnew
        x2=xold
        y1=ynew
        y2=yold

    EndIf

    If (((xnew<xt) = (xt<=xold))
        And ((yt-y1)*(x2-x1)) < ((y2-y1)*(xt-x1)))) Then inside=inside Xor 1

xold=xnew;
yold=ynew;

Next
```

Abbildung 8-10 BlitzBasic Implementierung

³⁸ URL: <http://www.blitzbasic.de> (27.07.2004)

8.5 Standard Ablaufschema

Die folgende Abbildung erläutert den Ablauf einer Kollision im Standardfall. Alle Objekte im gelb markierten Bereich werden von mir zur Verfügung gestellt.

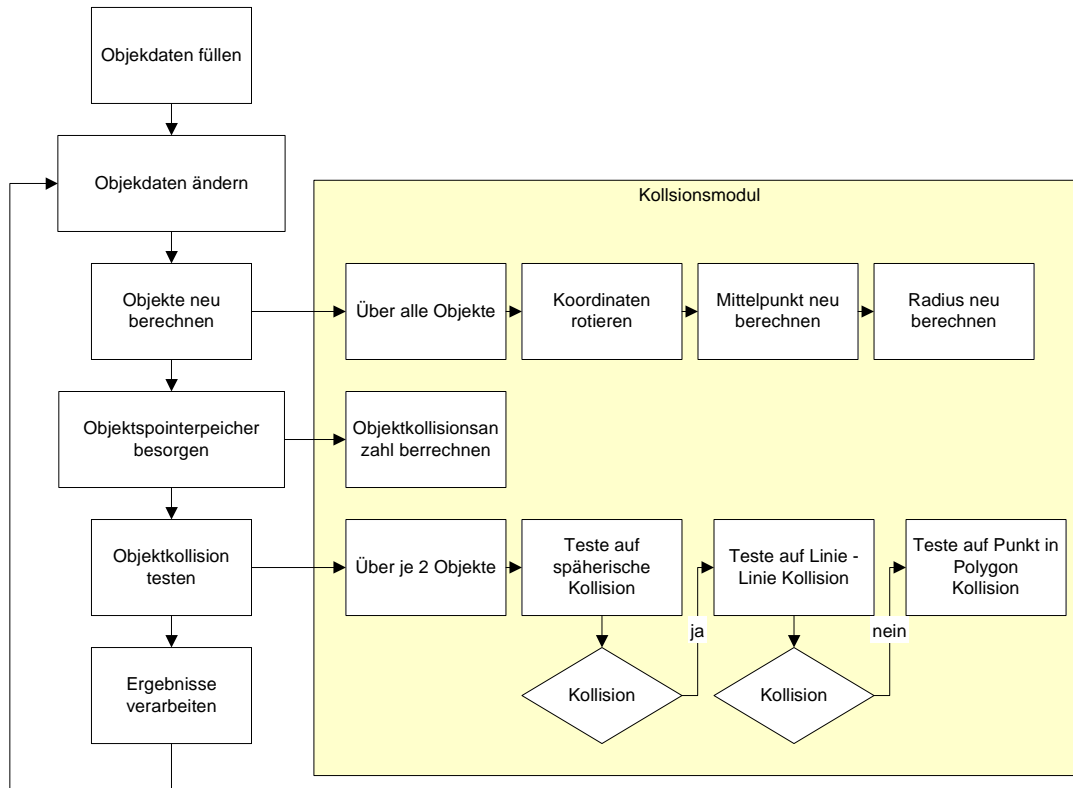


Abbildung 8-11 Standard Ablauf einer Kollision-Überprüfung

Der erste wichtige Punkt für den Benutzer ist, seine Objekte durch Koordinaten zu beschreiben. Je nach Lage der Objekte, muss unter Umständen, eine neue Berechnung der internen Koordinaten erfolgen. Das Kollisionsmodul kann für jedes Objekt eine Neuberechnung durchführen. Diese führt eine, je nach Winkel und Skalierung, Punkttransformation durch und berechnet den Radius und den Mittelpunkt des Objektes neu.

Im nächsten Schritt kann vom Benutzer bestimmt werden, welche Objekte auf eine Kollision überprüft werden. Je nach Anzahl der Objekte, kann die Rückgabeanzahl der Werte bestimmt werden.

Die maximale Anzahl ergibt sich aus: $Anzahl = \binom{n}{2} = \frac{n \cdot (n-1)}{2}$

Wenn der Objektausgabepuffer erzeugt wurde, kann die Kollision angestoßen werden. Intern werden jetzt alle Objekte gegeneinander auf eine sphärische Kollision überprüft. Falls diese eintritt, werden alle Linien der Objekte auf eine Kollision überprüft. Wenn das nicht der Fall ist, liegt ein Objekt entweder innerhalb des anderen Objekts oder es ist außerhalb dessen. Dieser

Sonderfall wird mit dem Punkt-Polygon-Test überprüft. Alle aufgetretenen Kollisionen werden im Objektausgabepuffer gesichert. Die Codierung erfolgt in einer 32 Bit Konstanten, wobei die ersten 16 Bit das erste Objekt angeben und die folgenden 16 Bit das zweite Objekt, welches an der Kollision beteiligt ist.

Mit diesen Daten kann der Benutzer weitere Verfahren ausführen, welche aber nicht Bestandteil der Kollisionserkennung sind.

8.6 Strukturbeschreibung

Jedes Objekt, das vom Kollisionssystem benutzt werden soll, muss nach folgender Struktur aufgebaut sein. Jeder Struktureintrag hat eine Breite von 4 Byte.

```
OBJ_STRUCT_X  
OBJ_STRUCT_Y  
OBJ_STRUCT_ROT  
OBJ_STRUCT_ZOOM  
OBJ_STRUCT_ECKEN  
OBJ_STRUCT_PO  
OBJ_STRUCT_PT  
OBJ_STRUCT_MX  
OBJ_STRUCT_MY  
OBJ_STRUCT_RAD
```

Abbildung 8-12 Kollisions-Objekt-Struktur

- OBJ_STRUCT_X und OBJ_STRUCT_Y, geben die Position des Objektes im zweidimensionalen Raum an.
- OBJ_STRUCT_ROT gibt den Winkel des Objektes an (im Bereich von 0 bis 1023).
- OBJ_STRUCT_ZOOM ist der Skalierungsfaktor (Standardwert \$100).
- OBJ_STRUCT_ECKEN ist die Anzahl der Ecken.
- OBJ_STRUCT_PO ist ein Pointer auf die erstellten Koordinaten, wobei gilt, dass jeder Punkt in je 16 Bit X- und 16 Bit Y-Wert aufgeteilt ist.
- OBJ_STRUCT_PT muss einen Speicherbereich mit der Größe alle Punkte bereitstellen. Bei jeder Rotation und/oder Skalierung werden hier die Neuberechneten Punkte hinein geschrieben.
- Die restlichen Daten sind privat und nur für die internen Funktionen wichtig.

8.7 Funktions- und Parameterbeschreibung

Im Folgenden unterscheide ich zwischen externen und internen Funktionen. Die externen Funktionen sind für den Benutzer interessant der nur den von mir beschriebenen Standardablauf einer Kollision beschreiten will. Die internen Funktionen sind für Änderungen oder anderweitige Nutzung interessant.

8.7.1 Externe Funktionen

Mit der Funktion *n_over_2* kann die Anzahl der maximal Kollision der Objekte berechnet werden.

Funktionsname:	n_over_2
Beschreibung:	berechnet Objektausgabepuffergröße
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Objekte
Rückgabewerte:	r1 = n über 2

Bei Änderung der Koordinaten oder der Größe oder des Winkels eines Objektes müssen die internen Daten mit der Funktion *recalc_obj* neu berechnet werden.

Funktionsname:	recalc_obj
Beschreibung:	Koordinaten neu berechnen
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r6 = &Objektstruktur

Mit der Funktion *check_obj_coll* können alle angegebenen Objekte (Strukturen hintereinander) auf eine Kollision geprüft werden.

Funktionsname:	check_obj_coll
Beschreibung:	Teste Objekte auf Kollision
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = &Objektstrukturen r1 = Anzahl Objekte r9 = &Objektausgabepuffer

8.7.2 Interne Funktionen

Alle internen Funktionen werden vom Kollisions-System verwendet. Da diese Funktionen aber in vielen Fällen für die Spiel-Entwicklung verwendbar sind, werden sie dem Benutzer zur Verfügung gestellt.

Funktionsname:	get_poly_middle_th
Beschreibung:	berechnet Mittelpunkt eines Polygons
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Anzahl Punkte von den Polygon r1 = Punktearray
Rückgabewerte:	r5 = X-Koordinate des Mittelpunktes r6 = Y-Koordinate des Mittelpunktes

Funktionsname:	get_poly_radius_th
Beschreibung:	berechnet den Radius eines Polygons
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Anzahl Punkte von den Polygon r1 = Punktearray r2 = X-Koordinate des Mittelpunktes r3 = Y-Koordinate des Mittelpunktes
Rückgabewerte:	r7 = Radius ²

Funktionsname:	pobj_in_obj_th
Beschreibung:	teste ob erster Punkt in Polygon
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r4 = Objekt 1 r5 = Objekt 2
Rückgabewerte:	r7 = 0 -> nein/ 1 -> ja

Die Funktion *obj_line_x_line* prüft zwei Linien auf ein Schneiden, falls dies erfolgt ist, wird das Register R1 mit 1 besetzt. Da es in vielen Fällen sinnvoll ist den genauen Schnittpunkt zu erfahren, kann diese Berechnung zusätzlich aktiviert werden (r11 ungleich Null besetzen). Das Kollisionsmodell benutzt diese Methode nicht, da ein Schnittpunkt nicht notwendig für die Kollisionserkennung ist.

Funktionsname:	obj_line_x_line
Beschreibung:	teste ob die Linien sich kreuzen
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0=x1 r1=y1 r2=x2 r3=y2 r4=x3 r5=y3 r6=x4 r7=y4 r10 = Adresse von fdiv r11 = Schnittpunkt berechnen (wenn <> 0) r2 = Schnittpunkt X (bei r11 true) r3 = Schnittpunkt Y (bei r11 true)
Rückgabewerte:	r7 = 0 (nicht geschnitten) / 1 (schneiden sich)

Wie ***obj_line_x_line*** nur werden die Daten in Objekt-Strukturen übergeben.

Funktionsname:	obj_line_test
Beschreibung:	teste 2 Objekte auf Linienkollision
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r4 = obj1 r5 = obj2
Rückgabewerte:	r7 = 0 -> nein/ 1 -> ja

Die Funktion ***point_in_poly*** überprüft ob ein Punkt in einem Polygon liegt. Das Polygon wird über Punkte definiert. Die Verbindungsfolge ist linear von Punkt zu Punkt, der letzte Punkt wird automatisch mit dem ersten Punkt verbunden. Die Koordinaten-Daten müssen als 16-Bit vorzeichenbehaftete Worte vorliegen. Der Testpunkt wird im Register R2 festgelegt. Hierbei ist zu beachten, dass die zwei Werte zu je 16-Bit (Vorzeichenbehaftet) in das Register vorliegen müssen.

Funktionsname:	point_in_poly
Beschreibung:	teste ob ein Punkt in Polygon liegt
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Anzahl Punkte von Polygon r1 = Punktearray r2 = 16 Bit (oberen) X, 16 Bit (unteren) Y
Rückgabewerte:	r7 = 0 -> nein/ 1 -> ja

9 Speichermanagement

Eine wichtige Eigenschaft von Betriebssystemen ist es, dem Benutzer Funktionen zur Verfügung zu stellen, mit denen er Speicher für seine Programme erhalten/freigeben kann. Im Grundlegenden gibt es zwei Systeme. Das Stacksystem legt Daten auf dem Stapelspeicher ab und es hat den Vorteil, dass schnell Speicherplatz verschafft werden kann. Je nach Betriebssystem kann dieser Speicherbereich automatisch freigeben werden. Der entscheidende Nachteil ist, dass der Speicher immer nur in einer aufsteigenden Funktionshierarchie verwendbar ist.

Das Heapsystem legt Daten in einen vom Betriebssystem festgelegten Bereich ab und ist so nicht vom Programmablauf abhängig. Allerdings ist das Anlegen/Löschen von Speicherbereichen langsamer als beim Stacksystem.

9.1 Heap - Speichermanagement

Das Speichermanagement soll es ermöglichen, einen bestimmten Speicherbereich mit einer vom Benutzer festgelegten Größe, als dynamischen Speicher behandeln zu dürfen, ohne sich um die Verwaltung der Speicherblöcke kümmern zu müssen. Dem Benutzer werden dazu gewöhnliche Funktionen, wie z.B. Anlegen und Löschen von Speicherbereichen zur Verfügung gestellt.

Die Verwaltung der Speicherblöcke basiert auf der Verwendung einer einfach verketteten Liste. Infolgedessen ergibt sich eine relativ einfache Implementierung. Es wird zugleich eine schnelle Fehleranalyse ermöglicht. Ein Performanceverlust ist nur beim Anlegen/Freigeben von Speicherplatz zu erwarten, besonders wenn sehr viele Elemente vorhanden sind. Dieses Problem ist aber bedingt durch die geringe Allokierungsanzahl zu vernachlässigen.

Eine Besonderheit des von mir implementierten Speichermanagement ist es, zu wählen, in welchen Bereich das Speichermanagement aufgebaut werden soll. Dadurch ergibt sich die Möglichkeit, mehrere Speicherbereiche mit einem Management abzudecken.

9.1.1 Strukturaufbau

Die Grundelemente eines Eintrages sind immer identisch. Dabei handelt es sich um einen Verweis auf das nächste Element und die Größe dieses Elementes. Alle folgenden Daten befinden sich nach diesen Einträgen.

Der erste Eintrag wird bei einer Initialisierung erstellt. Sein Grundaufbau ist mit allen anderen Speicherelementen identisch. Er enthält zusätzlich aber noch folgende Werte:

- die Anzahl der angelgten Elemente
- den freien Speicher
- den gesamten vorhandenen Speicher

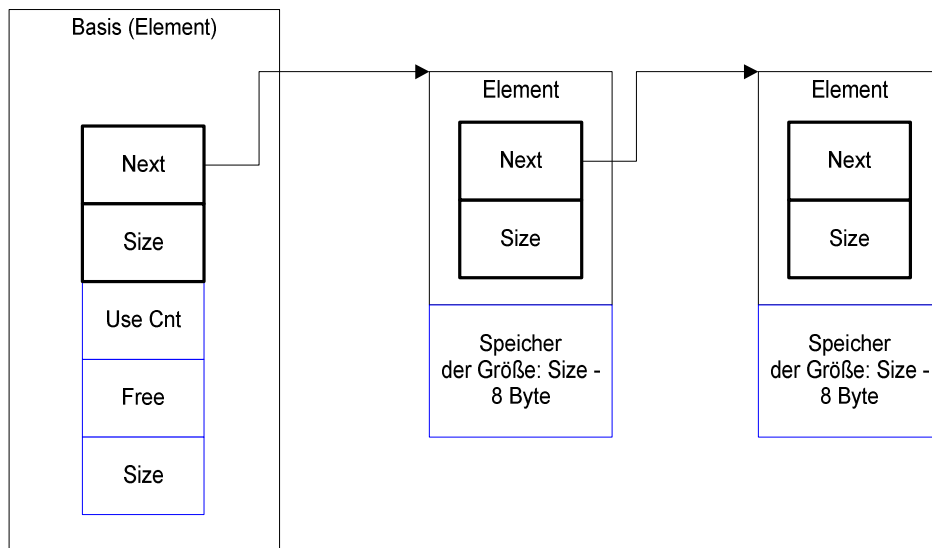


Abbildung 9-1 Element-Aufbau

9.1.2 Funktions- und Parameterbeschreibung

Die Funktion *mem_init* legt den Speicherbereich fest, auf dem mit den Funktionen *mem_malloc* und *mem_free* Speicher vergeben bzw. freigegeben werden kann.

Funktionsname:	mem_init
Beschreibung:	Speichermanagement anlegen
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Startadresse r2 = maximale Größe

Funktionsname:	mem_malloc
Beschreibung:	Speicher bekommen
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Startadresse (Basisadresse) r6 = maximale Größe
Rückgabewerte:	r2 = Adresse des allokierten Speichers

Funktionsname:	mem_free
Beschreibung:	Speicherbereich freigegeben
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Startadresse (Basisadresse) r6 = Adresse des allokierten Speichers

Die Funktion *mem_trace* gibt den aktuellen Status, des angegebenen Speicher-Managements aus. Ist insbesondere für die Fehlersuche bzw. Speicher-Leck-Suche zu verwenden.

Funktionsname:	mem_trace
Beschreibung:	Informationen über Speichermanager
Codetype:	ARM
Speichertyp:	IWRAM
Übergabeparameter:	r0 = Startadresse (Basisadresse)
Rückgabewerte:	r1 = Anzahl der Allokierungen
	r2 = freier Speicher
	r3 = Anzahl der Speicherlücken
	r4 = Größe der Speicherlücken
	r5 = absolut genutzter Speicher

9.2 Stack – Speichermanagement

Dieses Speichermanagement ist in seinem Aufbau sehr einfach. Es wird je nach angeforderter Speichergröße der Stackpointer verschoben. Als letzter Eintrag auf dem Stack kommt die Stackadresse vor der Allokierung. Somit ist ein Freigeben des Speichers durch ein einfaches Laden möglich.

9.2.1 Funktions- und Parameterbeschreibung

Für die Stackfunktionen existieren unter dem Pfad source/mem/mem_macros.i Makros, die übersichtlicheres Arbeiten ermöglichen. Ergänzt wurden die Makro-Funktionen durch *mem_alloc_mc*, welches nach dem Allokieren des Speichers, diesen mit Nullen füllt (BIOS-CpuSet).

Funktionsname:	stack_alloc
Beschreibung:	Speicher auf dem Stack anlegen
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Größe
Rückgabewerte:	r1 = Speicheradresse

Funktionsname:	stack_alloc_th
Beschreibung:	Speicher auf dem Stack anlegen
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Größe
Rückgabewerte:	r1 = Speicheradresse

Funktionsname:	stack_free
Beschreibung:	Speicher auf dem Stack freigeben
Codetype:	ARM
Speichertyp:	ROM

Funktionsname:	stack_free_th
Beschreibung:	Speicher auf dem Stack freigeben
Codetype:	THUMB
Speichertyp:	ROM

10 Grafikmanagement

10.1 Allgemeine Einführung in die Problematik

10.2 Allgemeine Grafikfunktionen

In vielen älteren Programmiersprachen (z.B. BASIC), hatte der Benutzer die Möglichkeit einfache Grafikfunktionen aufzurufen. Die Grundfunktionen sind das Ziehen von Linien, das Zeichnen von Kreisen und das Füllen von Flächen. Aus diesen primitiven Funktionen, kann eine Vielzahl komplexere Funktionen erstellt werden. Bedingt durch die Veränderung der Programmiersprachen und der Oberflächenprogrammierung, werden solche primitiven Funktionen heute nur selten verwendet. Meist ist der mitgetragene Overhead, bedingt durch äußere Umstände, enorm. Aus diesem Grund, habe ich die von mir erstellten Funktionen sehr flexibel belassen. Besonders da auch der GBA unterschiedliche Grafiksysteme verwendet. Aus diesem Grund zeichnet keine dieser Funktionen auf einen direkten Puffer, jede Funktion springt eine vom Benutzer angegebene Funktion an (Z.B. Punkt zeichnen). Dieser Weg benötigt etwas mehr Zeit, deswegen muss der Entwickler selber abschätzen, in wie weit er den Performance-Verlust billigen kann. Falls der Overhead zu groß ist, besteht immer noch die Möglichkeit, meine Funktionen zu verändern. Durch die Trennung von Algorithmus und Grafikzeichenfunktion, können diese Funktionen auch für andere Problemfälle von Nutzen sein. Z.B. kann das Füllen von Flächen für die Suche in einem zwei dimensional Puffer genutzt werden.

Eine Testimplementierung ist unter dem Pfad `\source\extendedfkt\extended_test\` zu finden. Abbildung 10-1 zeigt einen Funktionstest für das Zeichnen von Linien und Kreisen inkl. Füllfunktion.



Abbildung 10-1 Funktionstest

10.2.1 Linie

Um Linien zu zeichnen, ist die Verwendung des „Bresenham Algorithmus“ zu empfehlen. Dieser Algorithmus stellt eine effiziente Methode zur Rasterkonvertierung von Linien dar, da er nur ganzzahlige Addition und Subtraktion sowie die Multiplikation mit 2 beinhaltet. Der Grundgedanke des Algorithmus ist, dass nächste Pixel zu wählen, welches den geringsten

Abstand zu der wirklichen Linie hat (Approximation). Ich gehe im Folgenden nicht näher auf diesen Algorithmus ein, da er ein Grundbestandteil der Ausbildung eines Informatikers ist. Ein nicht unbedeutender Punkt ist die Koordinatentransformation, da der Grundalgorithmus nur in einem Quadranten zeichnen kann. Weiterhin habe ich die Bewegungsrichtung korrigiert, da der „Bresenham Algorithmus“ im Normalfall in die positive X-Richtung zeichnet. Für Grafikfunktionen ist dieses Verhalten irrelevant, wenn der Algorithmus aber z.B. einen Kollisionstrahl (ein Testpunkt bewegt sich durch eine Kollisionsarray) bilden soll, ist eine korrekte Bewegungsrichtung notwendig.

10.2.1.1 Funktions- und Parameterbeschreibung

Im Register R2 wird ein Funktionszeiger erwartet, der als Übergabeparameter die X und Y-Koordinaten enthält.

Funktionsname:	goline
Beschreibung:	bewege auf Bresenham-Linie
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = x1 r1 = y1 r3 = x2 r4 = y2 r2 = Funktionszeiger (r0=x,r1=y) (r12 = Rückgabewert, r0,r1,r6,r12 für frei Benutzung)

10.2.2 Kreis

Die Kreisimplementierung beruht ebenfalls auf dem „Bresenham Algorithmus“. Dieser Algorithmus wurde nur aus Vollständigkeitsgründen implementiert, spielt aber in der Spielentwicklung keine bedeutende Rolle.

10.2.2.1 Funktions- und Parameterbeschreibung

Im Register R2 wird ein Funktionszeiger erwartet, der als Übergabeparameter die X und Y-Koordinaten enthält. Zu beachten ist, dass der Kreis aus $8 * 1/8$ Kreis-Segmenten besteht, also die angegebene Funktion sich nicht auf einer direkten Kreisbahn bewegt.

Funktionsname:	draw_circle
Beschreibung:	bewege auf Kreisbahn (8 Teilwege)
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = x r1 = y r2 = Funktionszeiger (r0=x,r1=y) r3 = Radius

10.2.3 Füllen

Es gibt viele Möglichkeiten Flächen zu füllen. Eine der einfachsten Möglichkeiten ist ein rekursives pixelweißes Füllen. Dieser Algorithmus kann in einen 4-oder 8-Wege Modus arbeiten. Der Unterschied besteht darin, dass der 4-Wege Algorithmus nur nach Oben, Unten, Rechts und Links prüft und je nach Ergebnis eine neue Rekursion an dieser Stelle startet. Der 8-Wege Modus überprüft zusätzlich alle vier äußeren Pixel (siehe Abbildung 10-2).



Abbildung 10-2 Vier- oder Achtwege-Test

Für kleine Füllbereiche ist dieses Verfahren ausreichend, bei größeren Füllbereichen zeigt sich der Nachteil dieses Algorithmus. Denn pro neu erkannten und zu füllenden Pixel muss eine Rekursion ausgelöst werden, mit ihr fallen mehr Daten auf dem Stack an. Dadurch entsteht mit steigender Rekursionszahl ein enormer Stackverbrauch. Da der GBA aber ohnehin über wenig Speicher verfügt, ist die Verwendung eines anderen Algorithmus nahe liegender.

Eine sehr effiziente Möglichkeit ist das Verwenden eines zeilenbasierten Füllens. Im Gegensatz zum pixelbasierten Verfahren, werden pro Rekursion nur die neu zu überprüfenden bzw. zu füllenden Zeilen übergeben (siehe Abbildung 10-3). Die Funktionsweise ist wie folgt.

- Laufe nach rechts, solange Pixel nicht gesetzt und setze es (merke Grenze)
- Laufe nach links, solange Pixel nicht gesetzt und setze es (merke Grenze)
- Laufe von rechter zu linker Grenze
 - Teste, ob Pixel darüber gesetzt, wenn nicht, fülle diese Zeile
 - Teste, ob Pixel darunter gesetzt, wenn nicht, fülle diese Zeile

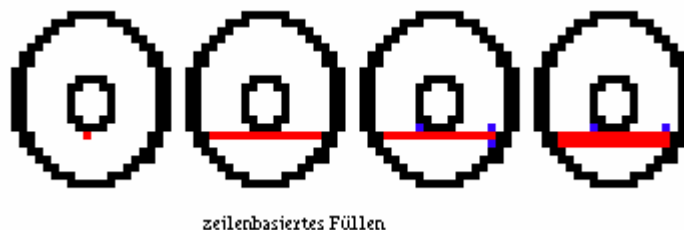


Abbildung 10-3 zeilenweise Füllen

10.2.3.1 Funktions- und Parameterbeschreibung

Zu beachten ist, dass die Pixeltest bzw. Pixelsetzfunktionen angegeben werden müssen. Sie erhalten als Parameter die X und Y Koordinate im aktuellen Puffer und müssen, je nach Art, einen Wert setzen oder überprüfen. Bei dem Pixeltest wird als Rückgabewert ein Wert ungleich Null erwartet, sobald es sich um ein bereits gesetztes Pixel handelt.

Funktionsname:	fillRowByRow_th
Beschreibung:	fülle zeilenbasiert
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = x r1 = y r2 = Funktionszeiger auf Setz-Funktion r0=f(r0=x,r1=y) r3 = Funktionszeiger auf Test-Funktion r0=(r0=x,r1=y) r0 = Rückgabewert der Testfunktion (0= nicht gesetzt) (r0,r1,r10,r11 für freie Benutzung)

10.2.4 Bézier-Kurve (3 Haltepunkte)

In einigen Fällen kommt es vor, dass eine lineare Bewegung eines Objektes nicht erwünscht ist. Aus diesem Grund gebe ich den Benutzer die Möglichkeit, eine Kurve durch drei Haltepunkte zu konstruieren und diese je nach Bedarf zu durchlaufen. Es gibt mehrere Arten Kurven zu erzeugen. Ich benutze in meinem System Bézier-Kurven. Die Methode zur Kurvenapproximation ist nach „Pierre Bézier“ benannt, der als Pionier des CAD-Bereiches zwischen 1960 und 1970 für den französischen Autohersteller Renault eine Modellierungshilfe für Autokarosserien entwickelte. Die Grundlage zur Gewichtung der Stützpunkte ist die Verwendung von Beinsteinpolynomen.

Die Herleitung der Beinsteinpolygone kann über folgende Formel erfolgen:

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Für diesen Fall ergibt sich dadurch $n=2$ und $0 \leq i \leq 2$, da nur drei Punkte benutzt werden.

$$B(2,2) = b_1(t) = t^2$$

$$B(1,2) = b_2(t) = 2 \cdot (1-t)t = 2 \cdot (t - b_1(t))$$

$$B(0,2) = b_3(t) = (1-t)^2 = 1 - 2t + b_1(t)$$

Durch Umstellen der angegebenen Formeln, ist nur eine Multiplikation notwendig. Die konstanten Multiplikation können durch ein Schieben nach links bzw. eine Addition des Wertes mit sich selbst erreicht werden.

Um das Ergebnis für die Bézier-Kurven im definierten Bereich von 0 bis 1 zu erhalten, muss die Summe über die einzelnen Werte der Punkte, welche mit dem Bernsteinpolynom vom Grad 0 bis 2 zu multiplizieren sind, gebildet werden.

Somit ergibt sich für jede Dimension eine Formel:

$$fx(t) = ax \cdot b1(t) + bx \cdot b2(t) + cx \cdot b3(t)$$

$$fy(t) = ay \cdot b1(t) + by \cdot b2(t) + cy \cdot b3(t)$$

Da ein Wert zwischen 0 und 1 für die Programmierung ohne Fließkommaeinheit unperformant ist, wird ein Wert zwischen 0 und 4095 verwendet. Je nach Bedarf ist eine andere Skalierung durch Ersetzen der Skalierungskonstante möglich (FRACT und FRACTN). Ein Nachteil von Bézier-Kurven ist, dass alle Stützpunkte, außer der Erste und Letzte, nicht durchlaufen werden. Die Kurve nähert sich nur an. Bei einer Bézier-Kurve die aber nur aus drei Punkten besteht, ist es einfach möglich, den mittleren Punkt so zu berechnen, dass er durchlaufen wird. Die folgenden Gleichungen errechnen aus einem gebenden Mittelpunkt, den für die Bézier-Kurven zu verwendenden Punkt.

$$bx_{neu}(t) = \frac{(4 \cdot bx - ax - cx)}{2} \quad by_{neu}(t) = \frac{(4 \cdot by - ay - cy)}{2}$$

Eine Testimplementierung ist unter dem Pfad `\source\extendedfkt\extended_test\` zu finden. Abbildung 10-4 zeigt einen Funktionstest für eine Bézier-Kurve.

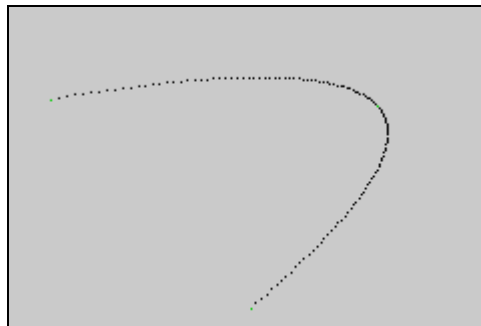


Abbildung 10-4 Bézier Testimplementierung

10.2.4.1 Funktions- und Parameterbeschreibung

Bevor die Funktion *spline3p_calc* durchlaufen wird, muss die Funktion *spline3p_init* für den Fall, dass der Mittelpunkt des Splines angepasst werden soll, aufgerufen werden.

Funktionsname:	spline3p_init
Beschreibung:	bereche mittleren Stützpunkt
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r7 = ax r8 = ay r9 = bx r10 = by r11 = cx r12 = cy
Rückgabewerte:	r9 = bx r10 = by

Funktionsname:	spline3p_calc
Beschreibung:	bereche x,y Koordinate aus B-Spline
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r2 = Counter (0 bis 4095) r7 = ax r8 = ay r9 = bx r10 = by r11 = cx r12 = cy
Rückgabewerte:	r0 = x r1 = y

10.3 Spezielle Grafik-Funktionen

In vielen Fällen sind die primitiven Grafikfunktionen nicht für die Spielentwicklung ausreichend. Vielmehr ergeben sich allgemeine Probleme mit dem Umgang von speziellen durch die Hardware erzeugten Problemen. Die folgenden Kapitel beschreiben die Probleme und Lösungen solcher Fälle.

10.3.1 Spritemanager

Der von mir implementierte Spritemanager erleichtert den Umgang mit Hardware-Sprites. Seine Hauptaufgabe ist, die Kapselung der Hardware-Adressen des Sprites. Der Sinn dieser Kapselung ist, dass im Normalgebrauch jedes Sprite eine feste Nummer inkl. ihres Adressebereichs hat. Für den Anwender ist dieses Verfahren aber in vielen Fällen ungünstig, da bei vielfachen Erstellen/Löschen von Sprites, es schwer fällt, eine Übersicht der freien und belegten Plätze zu behalten. Um diese Problematik zu beseitigen, habe ich ein System implementiert, dass es dem

Anwender ermöglicht, Sprites anzulegen und freizugeben. Ich habe es mit Absicht vermieden, den Grossteil der Spriteparameter zu kapseln, damit der Benutzer den kompletten Spriteparametersatz benutzen kann.

10.3.1.1 Funktions- und Parameterbeschreibung

Zum anlegen eines Sprites muss die Funktion *spr_new_sprite* mit den erforderlichen Parametern aufgerufen werden. Diese Funktion gibt einen 32 Bit Wert zurück. Alle Funktionen (außer *spr_calc_rotation*), welche mit dem Sprite-Management arbeiten benötigen diesen Wert. Eine Freigabe erfolgt über *rem_sprite*. Um einen schnellen Austausch der Sprite-Daten zu ermöglichen, können die Funktionen *spr_get_all* und *spr_set_all* benutzt werden. Damit ist es möglich die Sprite-Daten, in den nach der Hardware definierten drei Mal 16 Bit Worten, zu lesen oder zu beschreiben. Bei der Verwendung von Sprites, welche Rotation und Skalierung unterstützen, sollte zur Berechnung der korrekten Daten die Funktion *spr_calc_rotation* benutzt werden.

Funktionsname:	spr_new_sprite
Beschreibung:	anlegen eines neuen Sprites
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = X-Koordinate r1 = Y-Koordinate r3 = Tilenummer r4 = Priorität r5 = Modus r6 = Rotationsnummer
Rückgabewerte:	r7 = Spritedata

Funktionsname:	rem_sprite
Beschreibung:	löschen eines Sprites
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Spritedata

Funktionsname:	spr_change_xy
Beschreibung:	Koordinaten eines Sprites ändern
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Spritedata r1 = X-Koordinate

Funktionsname:	spr_change_prio
Beschreibung:	Priorität eines Sprites ändern
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Spritedata r1 = Priorität

Funktionsname:	spr_get_xy
Beschreibung:	Koordinaten eines Sprites erhalten
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Spritedata
Rückgabewerte:	r1 = X-Koordinate r2 = Y-Koordinate

Funktionsname:	spr_get_all
Beschreibung:	bekomme Hardwaredaten eines Sprites
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Spritedata
Rückgabewerte:	r1 = Wort 1 r2 = Wort 2 r3 = Wort 3

Funktionsname:	spr_set_all
Beschreibung:	setze Hardwaredaten eines Sprites
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Spritedata r1 = Wort 1 r2 = Wort 2 r3 = Wort 3

Funktionsname:	spr_calc_rotation
Beschreibung:	setze Rotations/Zoomwinkel
Codetype:	ARM
Speichertyp:	ROM
Übergabeparameter:	r0 = Zoomvalue (8 Bit Nachkomma) r1 = Winkel (16 Bit) r4 = Rotationsnummer

10.3.2 Scrollfunktionen

Eine für viele Spiele wichtige Eigenschaft, ist das Verwenden von Spielfeldern die größer als der sichtbare Bildschirmbereich sind. Damit andere Bereiche des Spielfeldes sichtbar werden, muss das Spielfeld bewegt werden. Der kleinste Bewegungsschritt ist ein Pixel. In vielen Spielen ist es wichtig, eine Bewegung in mehrere Richtungen zu erlauben.

Der GBA erlaubt es hardwaretechnisch, je nach verwendetem Modus (Textmodus), ein maximal 512*512 Pixel großes Spielfeld zu verwenden. In vielen Fällen ist diese Spielfeldgröße nicht akzeptabel. Aus diesem Grund habe ich ein System entwickelt, dass dem Anwender die

Möglichkeit bietet, beliebig große Spielfelder zu verwenden, welche in alle 4 Richtungen bewegt werden können.

Als Ausgangspunkt für das Scrollsystem verwende ich eine Hardwaremap mit der Größe von 256*256 Pixel. Der sichtbare Bereich beträgt bei meinem Scrollsystem immer 240*160 Pixel. Um die interne Funktionsweise des Scrollsystems zu verstehen, muss das eigentliche Hardwarescrolling in seiner Flexibilität betrachtet werden. Abbildung 11-5 stellt einen normalen Fall für eine Bildausgabe dar. Die eingegraute Hardwaremap ist 256*256 Pixel groß, das sichtbare Bild hat die Ausmaße von 240*160 Pixel. Der sichtbare Bereich kann durch die Hardware, Pixelweise verschoben werden.

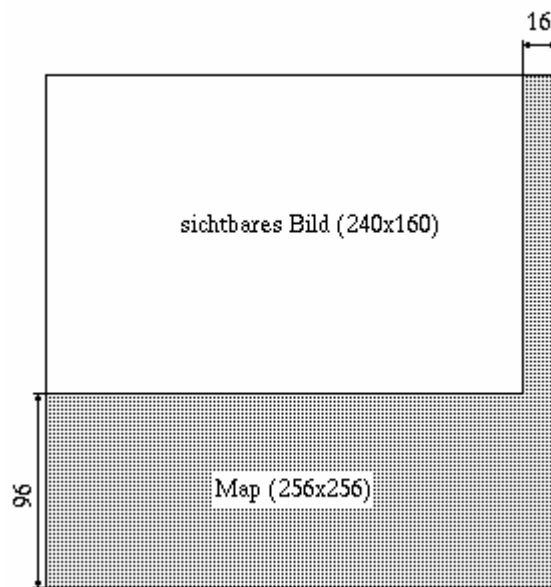


Abbildung 10-5 Mapaufbau

Ein sehr wichtiger Bestandteil der Hardware ist das Overlay-Verhalten des sichtbaren Bildes. In Abbildung 10-6 wird dieses Verhalten verdeutlicht. Das Overlay-Verhalten ermöglicht es, das sichtbare Bild in der gesamten Hardwaremap zu verschieben, ohne jedoch das Bild an den Hardwaremap-Grenzen abzuschneiden. In dem Fall einer Überlagerung der Grenzen, wird je nach Position, die linke oder obere Hardware-Zeile/Spalte weiterverwendet. Diese auf den ersten Blick unsinnig wirkende System, ist ein Grundbestandteil für die Verwirklichung von schnellen Scrollsystemen.

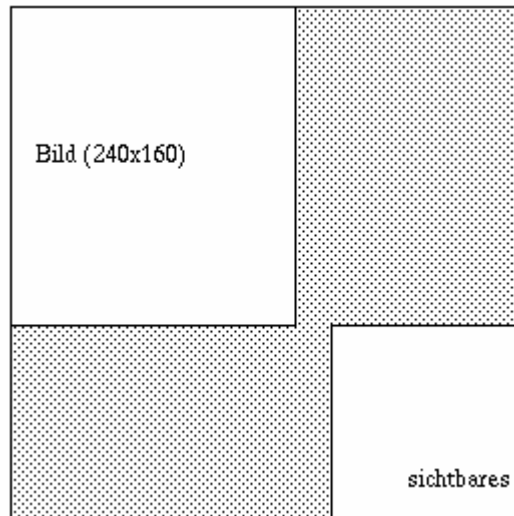


Abbildung 10-6 Overlay

Es gibt mehrere Lösungsansätze, ein beliebig großes Spielfeld zu bewegen. Einer der einfachsten Möglichkeiten wäre es, je nach Richtung der Bewegung, den gewünschten sichtbaren Ausschnitt der Quelldaten in die Hardwaremap zu kopieren. Ein Overlay wäre in diesem Fall nicht notwendig, da alle 8 Pixel ein Kopiervorgang ausgeführt werden kann, wodurch das Bild auch nur maximal 8 Pixel in jeder Richtung verschoben werden muss. Dieses Verfahren ist einfach implementierbar, hat aber den Nachteil, dass das Kopieren der Mapdaten erfolgen muss. Sicherlich ist der Datenaufwand von 240×160 Pixel an Map-Daten nicht viel (1200 Byte). Aber ein langsamer 16 Bit Zugriff und die Programmlogik verbrauchen dennoch unnötig Zeit. Eine elegantere aber komplexere Lösung wird in fast allen Spielen verwendet, die Levelbewegungen benutzen.

In diesem Fall ist eine Verwendung des Overlays unumgänglich. Das Verfahren kopiert je nach Bewegungsrichtung eine Spalte und/oder eine Zeile der Quelldaten in die Hardwaremap. Betrachten wir Abbildung 10-7. Bei einer Bewegung nach rechts wird der blau markierte Bereich durch Quelldaten gefüllt. Da dieser Bereich 8 Pixel breit ist, muss nur alle 8 Pixel ein Kopiervorgang gestartet werden. Falls eine Verschiebung nach links erfolgen würde, muss der rot markierte Bereich gefüllt werden. Dank des Overlays, ist keine komplizierte Behandlung notwendig. Analog arbeitet das Verfahren für vertikale Bewegungen. Somit wandert je nach Bewegungsrichtung der sichtbare Teil durch die Hardwaremap. Weiterhin werden neue Daten je nach Bewegung über, unter, rechts oder links um das sichtbare Bild kopiert. Der Vorteil ist klar erkennbar, es wird nur ein Bruchteil der Kopiarbeit (maximal 64 Byte pro Zeile oder Spalte) im Gegensatz zum vorher genannten Verfahren verwendet.

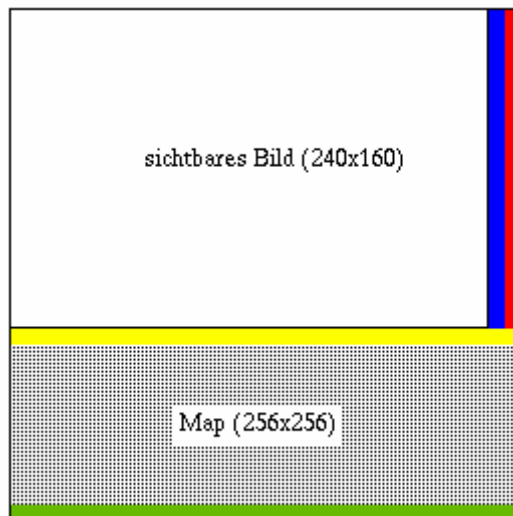


Abbildung 10-7 Mapbeispiel

10.3.2.1 Funktions- und Parameterbeschreibung

Zu beachten ist, dass immer erst (und nur einmal) die Hardwaremap mit Daten gefüllt werden muss, bevor Bewegungsvorgänge ausgeführt werden. Dafür sollte die Funktion *copymap* verwendet werden. Um eine fehlerfreies Arbeiten der *scroll_bg* Funktion zu garantieren, muss die verwendete Hardwaremap auf 32*32 Teile eingestellt sein und „Area overflow“ aktiviert werden.

Funktionsname:	scroll_bg
Beschreibung:	scrolle Background
Codetype:	ARM/THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = x r1 = y r2 = Hardware-Background Nummer (0 bis 3) r3 = Quellmap r4 = Zielmap (VRAM) r5 = Quellmap Höhe r6 = Quellmap Breite

Funktionsname:	copymap
Beschreibung:	scrolle Background
Codetype:	ARM/THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Quellmap r1 = Zielmap (VRAM) r2 = Quellmap-Modulo zu 32

10.3.3 Fontengine

Im Gegensatz zu konventioneller Programmentwicklung, ist eine Textausgabe bei der Spielentwicklung zweitrangig, da in vielen Fällen die Interaktion mit dem Spieler auf anderen Gebieten gesucht wird. Nichtsdestotrotz kann nicht auf eine Textausgabe verzichtet werden. Bedingt durch die GBA-Grafikhardware, gibt es mehrere Möglichkeiten, Texte auf dem Bildschirm auszugeben. Die einfachste Methode ist, die Verwendung einer Map als Textpuffer und die Teile-Daten als Buchstabenreferenz (siehe Kapitel 5.2.4). Eine andere Möglichkeit ist es, die Daten direkt in den Grafikspeicher zu kopieren. Die erste Methode benötigt kein Framework, da eine Textausgabe sehr einfach möglich ist. Die zweite Methode ist technisch bedingt, da eine Teile-Modus viel wahrscheinlicher ist, kaum von Bedeutung.

Der Großteil der Spiele die auf dem GBA existieren, nutzen den Teile-Modus, da dieser Modus sehr performant und Speichersparend ist. Aus diesem Grund arbeitet dieses Framework nur mit diesem Modus zusammen.

Folgende Punkte müssen vom Benutzer beachtet werden.

- Maximal 4 Bit Farbtiefe (Paletten durch Mapdaten änderbar)
- Automatische Maskierung zum Hintergrund (d.h. Hintergrund bleibt erhalten)
- Frei wählbarer Buchstabenabstand
- Mehr als 8 Bit Daten möglich (mehr als 256 Zeichen)
- Generierung der Buchstabenabstände durch beigefügtes Programm
- Buchstabenbreite bis 8 Pixel (einfach erweiterbar), Buchstabenhöhe beliebig

Das Generieren bzw. Zeichnen der Schriftarten (Fonts) ist in vielen Fällen eine aufwendige Arbeit. Gewöhnlich zeichnet der Grafiker die Buchstaben nach der Reihenfolge des ASCII-Codes. Allerdings mit der Ausnahme, dass er nicht von Stelle Null beginnt, sondern von Position 32 (Leerzeichen). Das hat den einfachen Grund, dass alle davor liegenden Zeichen nichtdruckbare Steuerzeichen sind. Die folgende Abbildung zeigt einen Teil einer solch erstellten Grafik.



Abbildung 10-8 Buchstaben

Gut zu erkennen ist die unter den Buchstaben liegende Linie mit den sich alle 8 Pixel wiederholenden Haken. Diese Konstruktion ist nur eine Orientierungshilfe für den Grafiker, damit er die Zeichen in den festgelegten Grenzen hält. Zur Bestimmung der Buchstabenbreiten wird ein von mir geschriebenes externes Programm verwendet. Dieses Programm erstellt eine Datei mit den Werten des Buchstabenstarts (von seiner angegebenen Mindest-Position aus) und der Buchstabenbreite in Pixeln. Diese Daten sind für die spätere Berechnung der Buchstabenabstände wichtig.

Das Quellformat für das Programm muss die später verwendete Grafikdatei sein, die Struktur muss bereits im Teile-Format vorliegen. Als Übergabeparameter müssen die Werte der

Schriftbreite übergeben werden (komplette Grafikbreite in Pixeln), die feste Zeichenbreite, die feste Zeichenhöhe und die Farbtiefe.

Name:	fontbcalc.exe
Synopsis:	fontbcalc.exe gfxwidth charwidth charheight colordepth
Parameter:	"gfxwidth" Font breite "charwidth" Buchstabenbreite "charheight" Buchstabenhöhe "colordepth" Font Farbtiefe in Bit
Beispiel:	./fontbcalc.exe font16.gfx 832 8 8 4

Ein Aufruf für ein 824 Pixel breites Font, mit maximal 8*8 Pixel breiten Buchstaben, die eine maximale Farbtiefe von 16 Farben haben, sieht wie folgt aus:

```
./fontbcalc.exe font16_2.gfx 824 8 8 4
```

Das Programm gibt wie schon beschrieben eine Liste mit dem Zeichenstart bzw. Zeichenbreiten aus. Damit dem Benutzer Änderungen dieser Daten einfach möglich sind, wird zu jedem Wert ein Kommentar mit dem gescannten Zeichen und seiner Position angegeben. Folgende Abbildung zeigt das abgemessene Zeichen W.

```
@:
@: .      .
@: .      .
@: .      .
@: .      .
@: .      .
@: . . . .
@: . . . .
@: .      .
@: .      .
@:
@:
@:
@: char 55
    .byte    0x08
```

Abbildung 10-9 Beispielausgabe des Fontkonverters

Sobald diese Daten erzeugt wurden, können diese zum Zeichnen benutzt werden. Eine Testimplementierung ist unter dem Pfad `\source\font\font_test\` zu finden. Sie gibt den folgenden Bildschirminhalt aus.

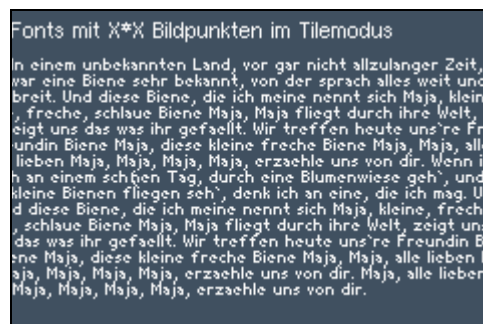


Abbildung 10-10 Fonttest

10.3.3.1 Funktions- und Parameterbeschreibung

Durch Veränderung des Tile-Offsets ist es möglich in den Sprite-Speicher, je nach gewähltem Grafikmodus, korrekt zu zeichnen. Eine Änderung der Farbindizierung durch die Map-Daten muss der Benutzer selbst vornehmen.

Funktionsname:	drawc8X4_th
Beschreibung:	zeichnet Buchstaben im Tilemodus
Codetype:	THUMB
Speichertyp:	ROM
Übergabeparameter:	r0 = Font r1 = Buchstabe (auch mehr als 8 Bit) r2 = Font Tabelle (durch externes Programm erzeugt) r9 = Tileoffset (bei Höhe größer 8 (fonttilelinesize:screentilelinesize) r10 = Font Höhe r11 = Zielpuffer r12 = X Koordinate des Buchstabens
Rückgabewerte:	r12 = neue X Koordinate nach dem Buchstaben

11 Testimplementierung eines Spiels

Um zu beweisen, dass das von mir entwickelte Framework einen funktionalen Sinn bei der Spielentwicklung hat, habe ich ein Spiel damit implementiert. Das Spiel ist natürlich in seinem Umfang aus Zeitgründen begrenzt, zeigt aber deutlich den Sinn der verwendeten Framework-Stufen.

Das Spiel simuliert ein Fahrzeugrennen, stellt damit einen klassischen Fall für ein Spiel dar. Enthält deswegen auch ein Großteil der vom Framework zur Verfügung gestellten Funktionen.

In den nächsten Kapiteln werde ich die einzelnen Komponenten des Spiels und ihre Verknüpfung mit dem Framework beschreiben.

11.1 System-Start

Bevor die einzelnen Spiel-Komponenten ausgeführt werden, muss das eigentliche Hintergrundsystem initialisiert werden.

Der erste Schritt ist, das Setzen des Interrupt-Stacks gefolgt vom Kopieren der entsprechenden Code/Datenteile in ihre Speicherbereiche (IWRAM, EWRAM) bzw. das Löschen der BSS-Sektionen. Diese wird mittels der von mir bereitgestellten Standard-Funktionen ausgeführt (siehe Kapitel 5.2.1).

Im nächsten Schritt erfolgt das Einhängen des Interrupt-Handlers, der in diesem Fall nicht multiple ist (siehe Kapitel 5.2.6).

Das Initialisieren des Heap-System (siehe Kapitel 9.1) ist ein Grundbestandteil des Hintergrundsystems, nur durch dynamischen Speicher lässt sich ein schneller und besonders ressourcenschonender Umgang mit dem knappen Speicher des GBA ermöglichen. Das Heap-System verwendet 128 KByte Speicher.

Das Ausführen der Logo- und Titel-Funktion erfolgt ohne besondere Vorkehrungen, da diese Funktionen keine Parameter benötigen.

Nach dem Ausführen der Vorschau-Funktionen, werden alle grafischen/technischen Daten, welche vom Rennen benötigt werden initialisiert (siehe Kapitel 11.4). Der davon benötigte Speicher wird durch das Stack-Speicher-Management (siehe Kapitel 9.2) bereitgestellt.

Nach dem Beenden des eigentlichen Rennens wird der benutzte Speicher freigegeben. Das System hält an dieser Stelle an, könnte aber z.B. ein neues Rennen auf einer anderen Strecke starten.

11.2 Das Logo

Bevor ein Spiel startet wird in der Regel das Hersteller-Logo präsentiert. Um diese Vorstellung etwas lebendiger zu gestalten wird eine Sound-Sample abgespielt und ein Teil des Logos durch verschieden Transparentstufen in den sichtbaren Bereich geführt.

Nach dem Verdunkeln des Bildschirms werden alle nötigen Grafikdaten kopiert. Danach wird für Timer 1 ein Interrupt angemeldet (siehe Kapitel 5.2.6). Der Grund für dieses Einhängen ist, das direkte Abspielen des Sound-Samples mittels der vorgestellten Funktionen im Kapitel 6.10.3. Da das Abspielen asynchron zum Programmverlauf ausgeführt wird, wird in dieser Zeit das Logo „Kinderkram“ sichtbar. Abbildung 11-1, zeigt das Resultat der Verarbeitung.



Abbildung 11-1 Logo

Nach dem Warten von zwei Sekunden, wird die Timer-1 Interrupt-Funktion ausgehangen (siehe Kapitel 5.2.6) und alle verwendeten Speicherbereiche freigegeben. Das Bild wird anschließend verdunkelt.

11.3 Das Titelbild

Bevor das eigentliche Rennen beginnt wird noch das Titelbild angezeigt, um auch dies etwas lebhafter zu gestalten, werden zwei Objekte (zwei Fische) nacheinander von Unten in das Bild bewegt. Danach springt der Spielname in das Bild und zwei Transparente Objekte („Grandprix“) vereinigen sich über den Titelnamen. Bewegte Präsentationen von Titeln sind in Spielen üblich, ein einfaches Titelbild ist nur sehr selten anzutreffen.

Der Programmablauf wird in folgenden Schritten ausgeführt:

Nach dem Kopieren der einzelnen Grafiken in ihre Speicherbereiche und das Füllen der Map-Daten, werden die zwei Objekte, welche in den Bildschirm bewegt werden sollen, mittels des Spritemanagers (siehe Kapitel 10.3.1) angelegt. Da Sprites nur eine maximale Größe von 64x64 Pixel haben können, werden die beiden Objekte unterteilt. Abbildung 11-2 zeigt die Aufteilung der Objekte. Mit den vom Spritemanager bereitgestellten Funktionen werden diese Objekte in den Bildschirmbewegt.

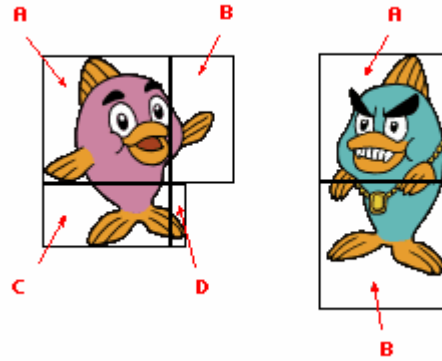


Abbildung 11-2 Sprite-Aufteilung

Das Herunterspringen des Spielnamens und das Verschieben des „Grandprix“-Schriftzuges erfolgt mit einfachem Background-Scrolling. Abbildung 11-3 die daraus resultierende Anzeige.



Abbildung 11-3 Titelbild

Sobald der Spieler die A-Taste gedrückt hat, werden die Sprite-Objekte freigegeben und das Bild verdunkelt.

11.4 Das Spiel

Das Rennen selbst besteht aus zwei Komponenten, der Initialisierung und der Rennsteuerung. Abbildung 11-4 Spielablauf stellt einen Überblick über beide Komponenten dar. Alle farblich markierten Ereignisse benutzen Funktionen des Frameworks.

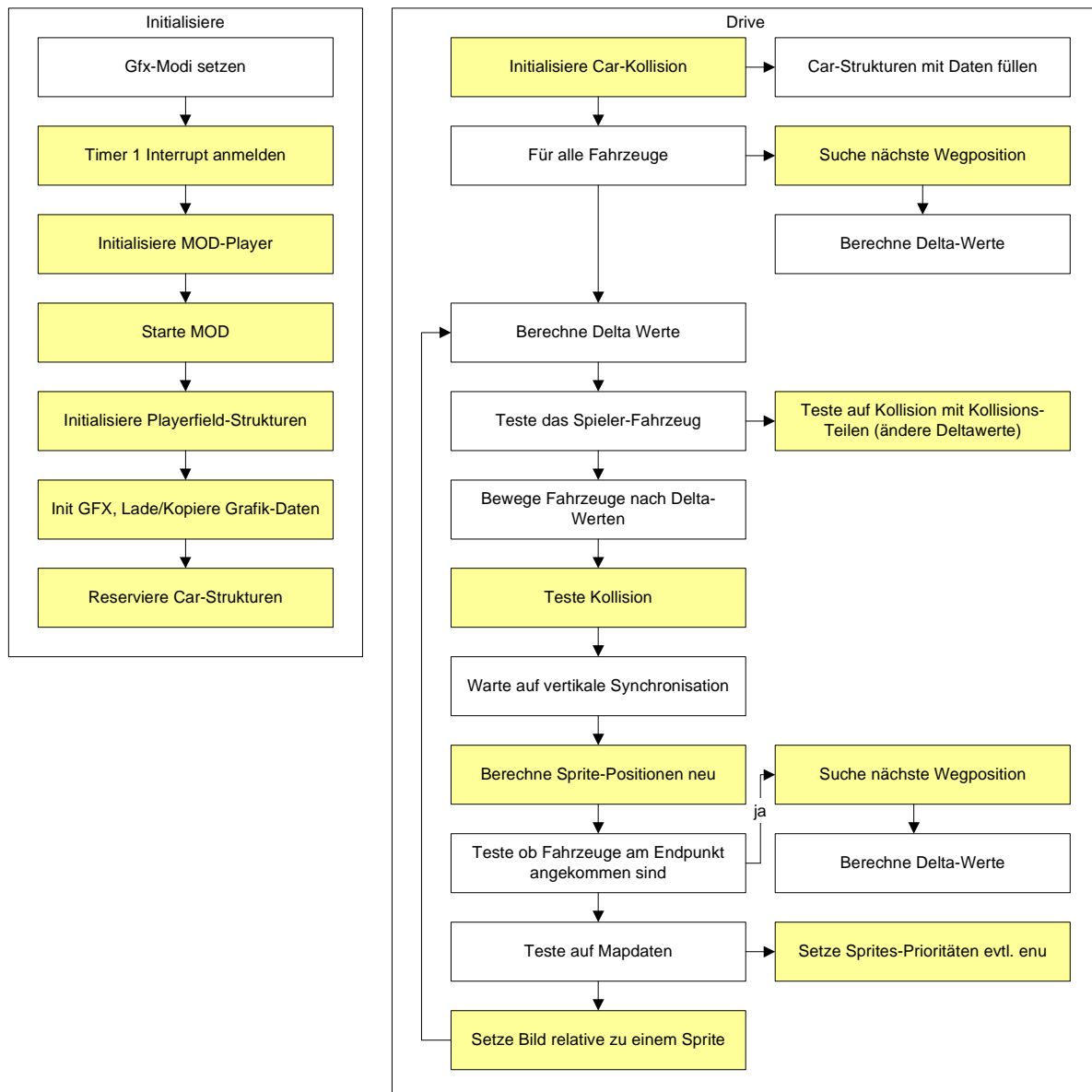


Abbildung 11-4 Spielablauf

Das Initialisierungs-Modul meldet durch den Interrupt-Handler einen neuen Timer-1 Interrupt an, dieser ist für den Gebrauch des MOD-Players notwendig. Der MOD-Player wird im ersten Schritt initialisiert, es wird ihm dabei die abzuspielende MOD-Datei mitgegeben. Im nächsten wird die MOD-Datei gestartet, ab diesem Zeitpunkt wird das Lied abgespielt. Das Reservieren/Initialisieren der benötigten Strukturen erfolgt je nach Anwendung über den Stack- oder Heap-Manager. Der Spiel-Hintergrund besteht aus 3 unabhängigen Layern die übereinander gelegt sind. Für die Anzeige und eine schnelle Bewegung dieser Layer, werden die Scrolling-Funktionen des Frameworks verwendet (siehe Kapitel 10.3.2). Abbildung 11-5 stellt alle verwendeten Teile des Spiellevels dar (komprimiert, siehe Kapitel 7.8), an diesem Beispiel kann man gut den Sinn der Ressourcenschonenden Teile-Grafik-Architektur erkennen.



Abbildung 11-5 Grafik-Teile

Die benötigten Sprites (4 Stück) werden mit Hilfe des Sprite-Managers (siehe Kapitel 10.3.1) reserviert und im späteren Spielablauf animiert. Abbildung 11-6 Sprite-Animation zeigt die unterschiedlichen Animations-Sequenzen.



Abbildung 11-6 Sprite-Animation

Nachdem alle Ressourcen initialisiert wurden, beginnt das eigentliche Rennen. Es besteht aus 3 Computer-Gegnern und einen vom Spieler steuerbaren Fahrzeug. Jedes Fahrzeug hat seine eigene Struktur die eine Vielzahl an Daten³⁹ enthält. Einer der wichtigsten Punkte im System ist die automatische Steuerung der Fahrzeuge⁴⁰. Es gibt mehrer Möglichkeiten dieses Problem zu lösen:

- eine simple Vordefinierte Steuerung wäre in jedem Fall ungeeignet, da die Bewegungen immer identisch sind.
- eine Orientierung der Fahrzeuge an einer vordefinierten Ideal-Linie wäre denkbar, könnte aber schlecht auf äußere Umstände (Fahrzeuge im Weg) reagieren
- eine Suche von Wegmarken die das Fahrzeug anfährt

Die letzte Methode ist eine etwas ungewöhnliche Methode, hat aber den Vorteil, dass sie sehr flexibel ist und die Fahrzeuge scheinbar intelligent reagieren (siehe Abbildung 11-7).

³⁹ Quelle: testgame/car.i

⁴⁰ Quelle: testgame/car_ki.i

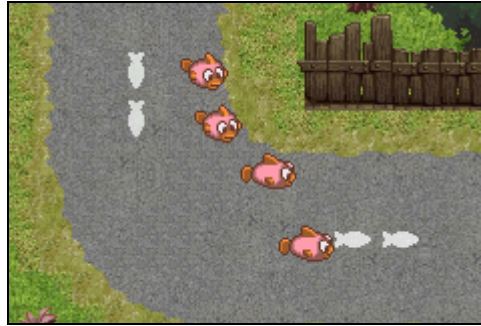


Abbildung 11-7 durch eine Kurve fahren

Die Wegmarken werden in einer zusätzlichen 2-Dimensionalen Array definiert, dass mit den Hintergründen verschoben wird. Die Erstellung erfolgt über den Teile-Editor (siehe Kapitel 4.1.15). Abbildung 11-8 Weg-Marken beschreibt die Suche der Wegmarken und daraus resultierenden Fahrweg. Als Beispiel befindet sich ein Fahrzeug auf Punkt A. Die Wegmarken sind immer in der Reihenfolge \$40, \$80 und \$c0 angeordnet. Um eine positive Fahrtrichtung (Wegmarken aufsteigend) zu erreichen, muss das Fahrzeug am Punkt A den Wegmarken-Wert \$c0 erhalten. Der nächste Wegmarken-Wert wäre \$40 (Überlauf von \$c0 ist \$40), die Suche erfolgt rekursiv im vorhandenen Gebiet. Wobei zu beachten ist, dass die Grenzen durch Kollisionsteile (die Striche in Abbildung 11-7) und die Wegmarken festgelegt sind. Der verwendete Such-Algorithmus ist der Row-By-Row-Algorithmus der zum füllen von Flächen verwendet wird (siehe Kapitel 10.2.3). Durch setzen, von speziellen Such- und Testfunktionen, arbeitet er mit der Wegmarken-Map zusammen. Nachdem dieser Algorithmus das Ziel \$40 gefunden hat (Punkt B), nimmt ein zweiter Algorithmus eine Position ein, die den Wert \$04 hat. Hierbei wird zufällig einer der \$04 Werte verwendet, dadurch wirken die Fahrzeugbewegungen unterschiedlich. Nachdem dies geschehen ist, wird das Fahrzeug vom Punkt A zum Punkt B bewegt. Wenn es den Punkt B erreicht hat, sucht es die nächste Wegeposition (\$80), sucht wiederum einen zufälligen \$04 Wert in der näheren Umgebung (+/- eine Position in X- und Y Richtung). Und fährt dann von Punkt B zu Punkt D. Dieses Verfahren wiederholt sich ständig und erlaubt dadurch ein selbstständiges Bewegen der Fahrzeuge.

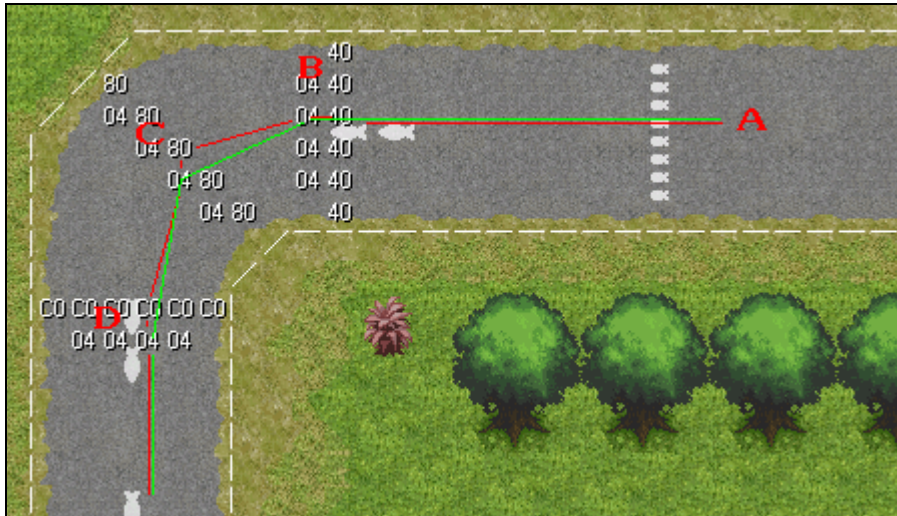


Abbildung 11-8 Weg-Marken

Um Durchfahrten durch Tunnel oder Kreuzungen von Straßen zuzulassen, ist dieses System⁴¹ um besondere Steuerwörter erweitert worden. Abbildung 11-8 zeigt einen klassischen Fall einer Durchfahrt mit ihren zugehörigen Steuerwörtern. Diese Vorfälle betreffen natürlich auch die Darstellung der Sprites im Verhältnis zu den Hintergrund-Layern.

Bei der Bewegung der Fahrzeuge kann es zu Kollision zwischen diesen kommen. Um diesen Fall⁴² zu erkennen wird das Kollisionsmodell aus diesem Framework verwendet (siehe Kapitel 8). Die Abstraktion jedes Fahrzeugs erfolgt durch ein Polygon mit 4 Verbindungen⁴³. Je nach Fahrzeugausrichtung werden diese Koordinaten gedreht um eine korrekte Kollisions-Erkennung zu garantieren. Je nach aufgetretener Kollision wird entsprechend reagiert (das hintere Fahrzeug bremst). Wichtig ist auch hier eine höhenabhängige Kollisionserkennung, damit Fahrzeuge über einen Tunnel nicht mit Fahrzeugen in diesem kollidieren (siehe Abbildung 11-9).



Abbildung 11-9 Höhen-Unterschiede

⁴¹ Quelle: /testgame/car_ki.s

⁴² Quelle: /testgame/car_koll.s

⁴³ Quelle: /testgmae/drive.s (Zeile 1156)

Der Ablauf des vom Menschen gesteuerten Fahrzeuges, verläuft in der Verarbeitung gleiche der Computergesteuerten Fahrzeuge, mit dem Unterschied, dass keine Wegmarken gesucht werden, sonder der Benutzer selbst die Bewegung angibt (über das Steuerkreuz). Weiterhin wird auf eine Banden-Kollision⁴⁴ geprüft (siehe Striche um Fahrbahnrand in Abbildung 11-10). Um mit dem Bildschirm relative zum Spieler mit zu bewegen, wird dieser wenn, der Spieler sich in bestimmten Bereichen befindet bewegt. Die Funktionen zur Bewegen des Bildes werden aus Framework verwendet (siehe Kapitel 10.3.2).

Dieses Renn-System läuft solange in einer Schleife bis eine Abbruchbedingung auftritt (z.B. ein Fahrzeug hat 4 Runden durchfahren).



Abbildung 11-10 Kollisions- und Höhen-Änderung

11.5 Fazit

Die Implementierung des kleinen Spiels zeigt den Sinn der vom Framework bereitgestellten Funktionen. Ein Aufbau eines komplett anderen Spieles wäre ohne Probleme mit diesem Framework möglich, zumal die Testimplementierung des Spieles nur einen Teil des Frameworks verwendet. Wie ich am Anfang der Diplomarbeit angedeutet habe, ist das Framework sehr flexibel, kann also für verschiedene Arten von Problemen angepasste werden (z.B. Flächenfüllen als schnelle Suchfunktion nutzen). Allerdings zeigt sich, dass der Anwender ein gewisses Grundkenntnis an Softwareentwicklung und der Hardware-Systemeigenschaften braucht um mit

⁴⁴ Quelle: /testgame/player_koll.s

diesen Funktionen umzugehen. Das war auch das Ziel dieses Frameworks, den Benutzer ein flexibles aber nicht triviales System anzubieten mit dem er schnell zu einem von ihm und nicht vom Framework geformten Ziel kommt. Durch das Bereitstellen der Quelltexte, ist der Anwender zudem in der Lage, Funktionen zu Ändern oder nach seinen Wünschen zu erweitern.

12 Zusammenfassung und Ausblick

Diese Diplomarbeit verdeutlicht den Unterschied zwischen einem Framework für die normale Software-Entwicklung und die der Spielentwicklung. Sie zeigt die meist unterschätzte Komplexität die hinter einem einfach aussehendes Spiel steckt und die Erfahrung die ein Programmierer sich aneignen muss um ohne jegliches Framework auszukommen. Mit dem von mir in dieser Diplomarbeit erstellten Framework sollte es dem erfahrenen Programmierer möglich sein, Spiele in kürzerer Zeit zu erstellen. Weniger erfahrene Anwender können mit dieser Arbeit die Prinzipien von heutzutage unbachteten aber enorm wichtigen Techniken kennen lernen.

Als Ausblick auf zukünftige Erweiterungen zu diesem Framework wäre unter anderem zu nennen:

- Frameworkerweiterungen für „Mode-7“-Effekte
- Erweitern des MOD-Players auf mehr als 4-Kanal mit Users-Sample Unterstützung
- Ergänzung der nicht implementierten MOD-Effekte
- Integration und Erweiterung von 3D-Grafik-Funktionen
- Erweiterung der Teilekollision im Testspiel
- Iff-Anim V5/7 Format laden (Animationsformat –Konverter)

13 Anmerkungen

Die Informationen in dieser Diplomarbeit werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Der Autor kann für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Alle Rechte vorbehalten, auch die fotomechanische Wiedergabe und Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten sind ohne Rücksprache mit dem Autor nicht zulässig.

- Amiga ist ein eingetragenes Warenzeichen der Amiga INC. USA
- GameBoy, GameBoyColor, GameBoyAdvance sind eingetragene Warenzeichen von Nintendo
- ARM und ARM7tdmi sind eingetragene Warenzeichen von ARM Corporate Communications

13.1 Danksagungen

Ich möchte Johann C. Vasold für die Bereitstellung der Grafiken in der Testimplementierung des Spieles danken und einiger verwendeten Grafiken in der Diplomarbeit.

14 Literaturverzeichnis

- [1] **Martin Korth:** Gameboy Advance Technical Info,
URL: <http://www.work.de/nocash/gbatek.htm> (27.07.2004)
- [2] **Tom Happ:** CowBite Virtual Hardware Specifications,
URL: <http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm> (27.07.2004)
- [3] **Brett Paterson:** A MOD Player Tutorial,
URL: <http://www.textfiles.com/music/fmoddoc.txt> (27.07.2004)
- [4] **Håvard Pedersen:** ProTracker support archive
URL: <ftp://de.aminet.net/pub/aminet/mus/edit/ptsupp.lha> (27.07.2004)
- [5] **Haruhiko Okumura:** LZSS Kompression/Dekompression
URL: <http://asp.itdrp.com/ZaneStudio/zarticle/LZSS.htm> (27.07.2004)
- [6] **Belogic:** The Audio Advance
URL: <http://belogic.com/gba/> (27.07.2004)