

8 bits de PODER

Auf Ihrem AMSTRAD CPC



"Beschränkungen sind kein Problem,
sondern eine Quelle der Inspiration".

V42_00

Jose Javier García Aranda

INDEX

1	WARUM HEUTE EINE MASCHINE VON 1984 PROGRAMMIEREN?.....	9
2	8BP-FUNKTIONEN UND SPEICHERNUTZUNG	11
2.1	WAS IST EINE RSX-BIBLIOTHEK?.....	12
2.2	FUNKTIONEN VON 8BP	13
2.3	AMSTRAD CPC-ARCHITEKTUR	14
2.3.1	<i>GOSUB /RETURN Stapel</i>	19
2.3.2	<i>Ein Experiment, um das ROM mit Winape zu sehen</i>	20
2.4	8BP-SPEICHERABBILDUNG UND MONTAGEOPTIONEN.....	21
3	BENÖTIGTE WERKZEUGE.....	25
4	ERSTE SCHRITTE MIT 8BP.....	27
4.1	WINAPE INSTALLIEREN	27
4.2	VERTRAUT WERDEN MIT WINAPE: "HALLO WELT"	27
4.3	DOWNLOAD DER 8BP-BIBLIOTHEK	27
4.4	AUSFÜHREN DER DEMOS	28
4.5	ERSTELLEN SIE IHR ERSTES PROGRAMM MIT 8BP	29
4.6	ERSTELLEN SIE IHRE .DSK MIT IHREM 8BP-SPIEL	32
5	SCHRITTE, DIE SIE UNTERNEHMEN MÜSSEN, UM EIN SPIEL ZU ENTWICKELN	33
5.1	VERZEICHNISSTRUKTUR IHRES PROJEKTS.....	33
5.2	IHR SPIEL IN NUR 3 DATEIEN.....	34
5.3	ERSTELLEN SIE EINE DISC ODER KASSETTE MIT IHREM SPIEL.....	36
5.3.1	<i>Herstellung einer Scheibe</i>	36
5.3.2	<i>Herstellung eines Bandes mit Winape</i>	36
5.3.3	<i>Einfaches Erstellen eines Bandes mit CPCDiskXP, 2cdt und tape2wav.....</i>	38
5.3.4	<i>Fehlersuche bei LOAD und MEMORY.....</i>	39
6	BIBLIOTHEKSMONTAGE, MUSIK UND GRAFIK	42
6.1	MAKE_ALL.ASM.....	43
6.2	AUFBAU DER BILDDATEI.....	45
6.3	STRUKTUR DER ANIMATIONSSEQUENZDATEI	45
6.4	AUFBAU DER ROUTINGDATEI	46
6.5	AUFBAU DER WELTKARTENDATEI.....	46
7	ZYKLUS DES SPIELS	47
7.1	SO MESSEN SIE DIE FPS IHRES SPIELZYKLUS	47
8	SPRITES.....	49
8.1	BEARBEITEN VON SPRITES MIT SPEDIT UND ZUSAMMENSETZEN VON SPRITES	49
8.2	EIN SPRITE DRUCKEN	54
8.3	SPRITE SPIEGELN	55
8.4	SPRITES MIT ÜBERSCHREIBUNG	57
8.4.1	<i>Sprite-Überlappungen durch Überschreiben verbessern.....</i>	61
8.4.2	<i>Überschreiben mit 4 Hintergrundfarben.....</i>	62
8.4.3	<i>Überschreiben in MODE 1.....</i>	63
8.4.4	<i>Wie man Sprites "hinter" den Hintergrund malt.....</i>	64
8.4.5	<i>Wie man mehr Farben mit Überschreiben verwendet</i>	65

8.5	SPRITES MIT HINTERGRUNDBILDERN.....	67
8.6	TABELLE DER SPRITE-ATTRIBUTE.....	69
8.7	ALLE SPRITES GEDRUCKT UND SORTIERT.....	74
8.8	KOLLISIONEN ZWISCHEN SPRITES	76
8.9	ANPASSUNG DER KOLLISIONSEMPFINDLICHKEIT VON SPRITES	78
8.10	WER KOLLIDIERT UND MIT WEM: COLSPALL	79
8.10.1	<i>Programmieren eines Mehrfachschusses mit COLSPALL</i>	80
8.10.2	<i>Wer kollidiert, wenn es mehrere Überschneidungen gibt.....</i>	81
8.10.3	<i>Erweiterte Verwendung des Statusbytes bei Kollisionen.....</i>	83
8.11	TABELLE DER ANIMATIONSSEQUENZEN	84
8.12	SPEZIELLE ANIMATIONSSEQUENZEN	85
8.12.1	<i>Todesfolgen</i>	86
8.12.2	<i>Sequenzen beenden.....</i>	87
8.12.3	<i>Verkettete Sequenzen.....</i>	87
8.12.4	<i>Animation Makro-Sequenzen</i>	87
9	IHR ERSTES EINFACHES SPIEL.....	91
9.1	ALSO, LASST UNS SPRINGEN! BOING, BOING!	91
10	BILDSCHIRMSÄTZE: LAYOUT ODER KACHELPLAN	95
10.1	DEFINITION UND VERWENDUNG DES LAYOUTS	95
10.2	BEISPIEL FÜR EIN SPIEL MIT LAYOUT	98
10.3	WIE MAN EIN TOR IM LAYOUT ÖFFNET	100
10.4	EIN PUZZLESPIEL: LAYOUT MIT HINTERGRUND	101
10.5	WIE SIE IN IHREN LAYOUTS SPEICHERPLATZ SPAREN KÖNNEN.....	103
11	FORTGESCHRITTENE PROGRAMMIERUNG UND "MASSENLOGIK".....	105
11.1	MESSUNG DER GE SCHWINDIGKEIT VON BEFEHLEN	105
11.2	EINE EINZIGE LOGIK FÜR ALLE IHRE BILDSCHIRME	112
11.3	MASSENLOGIK" TECHNIK.....	113
11.3.1	<i>Bewegt 32 Sprites mit massiver Logik</i>	114
11.3.2	<i>Alternierende und periodische Kaskadenausführung</i>	115
11.3.3	<i>Einfaches Beispiel für Massenlogik</i>	117
11.3.4	<i>Blockieren" der Bewegung der Geschwader</i>	118
11.3.5	<i>Massive Logiktechnik in "Pacman"-artigen Spielen</i>	119
11.3.6	<i>Reduzierung der Anzahl der Befehle im eingestellten Zyklus</i>	121
11.3.7	<i>Routen, die das Spiel durch Manipulation des Zustands beschleunigen.....</i>	125
11.3.8	<i>Routing von Sprites mit "massiver Logik".</i>	125
12	KOMPLEXE PFADE: BEFEHL ROUTEALL	129
12.1	PLATZIERT EIN SPRITE IN DER MITTE EINER ROUTE : ROUTESP	131
12.2	ERWEITERTE ROUTEN ERSTELLEN.....	133
12.2.1	<i>Erzwungene Zustandsänderungen von Routen</i>	133
12.2.2	<i>Erzwungene Reihenfolgeänderungen von Routen</i>	135
12.2.3	<i>Erzwungener Bildwechsel von Routen.....</i>	135
12.2.4	<i>Erzwungenes Rerouting von Routen</i>	139
12.2.5	<i>Erzwungene Pfadänderungen aus BASIC.....</i>	139
12.2.6	<i>Erzwungene Animation von Routen.....</i>	141
12.2.7	<i>Wie man "dynamische" (nicht vordefinierte) Routen erstellt</i>	141
12.2.8	<i>Programmierung von Routen mit Mustern</i>	142

12.2.9	<i>Streckentypologie</i>	143
13	HALBBYTE-GLEITBEWEGUNG	145
14	SCROLLING-SPIELE	147
14.1	STARS: SCHRIFTROLLE MIT STERNEN ODER GESPRENKELTER ERDE.....	147
14.2	BLÄTTERN MIT MOVERALL UND/ODER AUTOALL	150
14.3	TECHNIK DES "SPOTTING"	152
14.4	MAP2SP: SCHRIFTROLLE AUF DER GRUNDLAGE EINER WELTKARTE.....	155
14.4.1	<i>Karte der Welt (Kartentisch)</i>	157
14.4.2	<i>Verwendung der MAP2SP-Funktion</i>	158
14.4.3	<i>Beispiel für eine Phasendatei</i>	161
14.4.4	<i>Feindliche Kollision mit Karte</i>	163
14.4.5	<i>Hintergrundbilder in Ihrer Schriftrolle</i>	164
14.5	PARALLAXE-SCHRIFTROLLE.....	165
14.6	DYNAMISCHE KARTENAKTUALISIERUNG: UMAP	166
14.7	ANIMATION UND FARBVERSCHIEBUNG: BEFEHL RINK	168
14.7.1	<i>2D-Autorennen</i>	169
14.7.2	<i>Ziegelstein-Schriftrolle</i>	171
15	PLATTFORM-SPIELE	173
16	HORDEN VON FEINDEN IN ROLLENSPIELEN	175
17	NEU DEFINIERBARE MINI-ZEICHEN: PRINTAT	177
17.1	ERSTELLEN SIE IHR EIGENES MINI-ALPHABET	178
17.2	STANDARDALPHABET FÜR MODE 1	179
18	PSEUDO 3D	181
18.1	3D-PROJEKTION	183
18.1.1	<i>Mathematik der Pseudo-3D-Projektion</i>	184
18.1.2	<i>Kurven</i>	188
18.2	BILDER VERGRÖßERN	189
18.3	VERWENDUNG VON SEGMENTEN	191
19	MUSIK	193
19.1	MUSIKBEARBEITUNG MIT WYZ-TRACKER	193
19.2	ZUSAMMENSTELLUNG DER LIEDER	194
19.3	WAS TUN, WENN MAN KEINE MUSIK IN 1400 BYTES UNTERBRINGEN KANN?	195
20	C PROGRAMMIEREN MIT 8BP	197
20.1	ERSTER SCHRITT: PROGRAMMIERE DEIN BASIC-SPIEL	198
20.2	SCHRITT 2: ÜBERSETZEN SIE IHREN SPIELZYKLUS VON BASIC NACH C	199
20.2.1	<i>GOSUB und RETURN in C</i>	203
20.2.2	<i>Kommunikation zwischen BASIC und C mit BASIC-Variablen</i>	204
20.2.3	<i>BASIC- und C-Textstrings</i>	207
20.3	DRITTER SCHRITT: KOMPILIEREN MIT "COMPILA.BAT"	208
20.4	SCHRITT 4: ÜBERPRÜFUNG DER SPEICHERGRENZEN	210
20.5	FÜNFTER SCHRITT: ERMITTlung DER ADRESSE DER AUFZURUFENDEN FUNKTION VON BASIC211	
20.6	SCHRITT 6: FÜGEN SIE DIE NEUE BINÄRDATEI IN IHR .DSK-SPIEL EIN	212

20.7	8BP FUNKTIONSREFERENZ IN C.....	213
20.8	BASIC-FUNKTIONSREFERENZ IN C ("MINIBASIC")	215
21	REFERENZHANDBUCH FÜR DIE 8BP-BIBLIOTHEK.....	217
21.1	FUNKTIONEN DER BIBLIOTHEK	217
21.1.1	<i>3D</i>	217
21.1.2	<i>ANIMA</i>	218
21.1.3	<i>ANIMALL</i>	219
21.1.4	<i>AUTO</i>	220
21.1.5	<i>AUTOALL</i>	220
21.1.6	<i>COLAY</i>	220
21.1.7	<i>COLSP</i>	221
21.1.8	<i>COLSPALL</i>	223
21.1.9	<i>LAYOUT</i>	224
21.1.10	<i>LOCATESP</i>	226
21.1.11	<i>MAP2SP</i>	226
21.1.12	<i>MOVER</i>	228
21.1.13	<i>MOVERALL</i>	228
21.1.14	<i>MUSIK</i>	229
21.1.15	<i>PEEK</i>	229
21.1.16	<i>POKE</i>	230
21.1.17	<i>PRINTAT</i>	230
21.1.18	<i>PRINTSP</i>	231
21.1.19	<i>PRINTSPALL</i>	232
21.1.20	<i>RINK</i>	234
21.1.21	<i>ROUTEALL</i>	235
21.1.22	<i>ROUTESP</i>	237
21.1.23	<i>SETLIMITS</i>	238
21.1.24	<i>SETUPSP</i>	238
21.1.25	<i>STARS</i>	240
21.1.26	<i>UMAP</i>	242
22	WIE MAN EINE ANZEIGETAFFEL ERSTELLT	243
23	MÖGLICHE ZUKÜNTIGE VERBESSERUNGEN DER BIBLIOTHEK .	245
23.1	SPEICHER FÜR DAS AUFFINDEN NEUER FUNKTIONEN	245
23.2	PIXELAUFLÖSUNG BEIM DRUCKEN.....	245
23.3	LAYOUT DES MODUS 1	245
23.4	VERFILMUNGSKAPAZITÄT	245
23.5	HARDWARE-ROLLFUNKTIONEN.....	246
23.6	ÜBERTRAGEN DER 8BP-BIBLIOTHEK AUF ANDERE MIKROCOMPUTER	247
24	EINIGE SPIELE, DIE MIT 8BP GEMACHT WURDEN	249
24.1	MUTANT MONTOYA.....	249
24.2	ANUNNAKI, UNSERE AUßERIRDISCHE VERGANGENHEIT.....	250
24.3	NIBIRU	251
24.4	FRISCHES OBST UND GEMÜSE.....	252
24.5	"3D RACING ONE.....	252
24.6	WELTRAUM-PHANTOM	253
24.7	EWIGER FROGGER.....	254
24.8	ERIDU: DER RAUMHAFEN.....	255
24.9	GLÜCKLICHER MONTY.....	256

24.10	BLASTER-PILOT.....	256
24.11	NOMWARS.....	257
24.12	PACO, DER MANN	258
24.13	MINI-SPIELE.....	259
24.13.1	<i>Mini-Pong</i>	259
24.13.2	<i>Mini-Invaders</i>	260
25	ANHANG I: ORGANISATION DES VIDEOSPEICHERS.....	261
25.1	DAS MENSCHLICHE AUGE UND DIE AUFLÖSUNG DES CPC.....	261
25.2	VIDEO-SPEICHER.....	261
25.2.1	<i>Modus 2</i>	261
25.2.2	<i>Modus 1</i>	261
25.2.3	<i>Modus 0</i>	262
25.2.4	<i>Display-Speicher</i>	262
25.3	BERECHNUNG EINER BILDSCHIRMADRRESSE.....	264
25.4	BILDSCHIRMTÜCHER	264
25.5	WIE MAN EINEN LADEBILDSCHIRM FÜR SEIN SPIEL ERSTELLT	265
26	ANHANG II: DIE PALETTE.....	269
27	ANHANG III: INKEY-CODES.....	271
29	ANHANG IV: AMSTRAD CPC ASCII-TABELLE	273
30	ANHANG V: EINIGE GERÄUSCHEFFEKTE	275
31	ANHANG VI: INTERESSANTE FIRMWARE-ROUTINEN	277
32	ANHANG VII: TABELLE DER SPRITE-ATTRIBUTES	279
33	ANHANG VIII: 8BP SPEICHERPLAN.....	281
34	ANHANG IX: VERFÜGBARE BEFEHLE 8BP	283
35	ANHANG X: 8BP-MONTAGEOPTIONEN.....	285
36	ANHANG XI: RSX/AUFRUF-ZUORDNUNGEN	287
37	ANHANG XII: 8BP-FUNKTIONEN IN C.....	289
38	ANHANG XIII: MINIBASIC IN C.....	291

1 Warum sollte man heute eine Maschine von 1984 programmieren?

Denn Einschränkungen sind kein Problem, sondern eine Quelle der Inspiration.

Beschränkungen, sei es bei einer Maschine oder einem Menschen, oder allgemein bei jeder verfügbaren Ressource, regen unsere Phantasie an, sie zu überwinden. Die AMSTRAD, eine Maschine aus dem Jahr 1984, die auf dem Z80-Mikroprozessor basiert, hat einen kleinen Speicher (64 KB) und eine geringe Verarbeitungskapazität, aber nur im Vergleich zu den heutigen Computern. Diese Maschine ist sogar eine Million Mal schneller als die Maschine, die Alan Turing 1944 baute, um die Nachrichten der Enigma-Maschine zu entschlüsseln. Wie alle Computer der 1980er Jahre war der AMSTRAD CPC in weniger als einer Sekunde hochgefahren, und der BASIC-Interpreter war bereit, Benutzerbefehle zu empfangen, denn BASIC war die Sprache, mit der Programmierer ihre ersten Entwicklungen erlernten. Der AMSTRAD BASIC war im Vergleich zu seinen Konkurrenten besonders schnell, und ästhetisch war er ein sehr attraktiver Computer!



Abb. 1: Das legendäre AMSTRAD-Modell CPC464

Der Z80-Mikroprozessor kann nicht einmal multiplizieren (in BASIC kann man multiplizieren, aber das basiert auf einem internen Programm, das die Multiplikation mit Hilfe von Additionen oder Registerverschiebungen durchführt), er kann nur Additionen, Subtraktionen und logische Operationen durchführen. Trotzdem war er die beste 8-Bit-CPU und bestand nur aus 8500 Transistoren, im Gegensatz zu anderen Prozessoren wie dem M68000, dessen Name gerade von den 68000 Transistoren herrührt.

CPU	Anzahl der Transistoren	MIPS (Millionen Anweisungen pro Sekunde)	Computer und Konsolen, die es enthalten
6502	3.500	0,43 @1Mhz	COMMODORE 64, NES, ATARI 800...
Z80	8.500	0,58 @4Mhz	AMSTRAD, COLECOVISION, SPECTRUM, MSX...
Motorola 68000	68.000	2.188 @ 12,5 Mhz	AMIGA, SINCLAIR QL, ATARI ST...
Intel 386DX	275.000	2.1 @16Mhz	PC
Intel 486DX	1.180.000	11 @ 33 Mhz	PC
Pentium	3.100.000	188 @ 100Mhz	PC
ARM1176		4744 @ 1Ghz (1186 pro Kern)	Raspberry pi 2, Nintendo 3DS, Samsung galaxy, ...
Intel i7 (5. Generation)	2.600.000.000	238310 @ 3Ghz (fast! 500.000 mal schneller als ein Z80)	PC
AMD Ryzen 9 3950X (16 Kerne)	3.800.000.000	749070 @4.8Ghz (1,3 Millionen Mal schneller als Z80)	PC

Tabelle 1Vergleich von MIPS

Dies macht die Programmierung äußerst interessant und anregend, um zufriedenstellende Ergebnisse zu erzielen. Unsere gesamte Programmierung muss darauf ausgerichtet sein, die räumliche (Speicher) und zeitliche (Operationen) Komplexität der Berechnungen zu reduzieren, was uns zwingt, Tricks, Kniffe, Algorithmen usw. zu erfinden, und das Programmieren zu einem spannenden Abenteuer macht. Aus diesem Grund ist die Programmierung von Maschinen mit geringer Verarbeitungskapazität ein zeitloses Konzept, das weder Moden unterliegt noch durch die technische Entwicklung bedingt ist.

Der gesamte Code für dieses Buch, einschließlich der Bibliothek, mit der Sie Ihre eigenen Spiele erstellen oder Beiträge zur Bibliothek leisten können, ist im GitHub-Projekt "8BP" unter dieser URL zu finden. Laden Sie einfach die Zip-Datei herunter (in einem späteren Kapitel werde ich Ihnen eine Schritt-für-Schritt-Anleitung geben).

<https://github.com/jjaranda13/8BP>

Es gibt auch einen Blog mit vielen Informationen unter:

<http://8bitsdepoder.blogspot.com.es>

Und einen youtube-Kanal:

https://www.youtube.com/channel/UCThvesOT-jLU_s8a5ZOjBFA

2 8BP-Funktionen und Speichernutzung

Die 8BP-Bibliothek ist keine "Spiel-Engine". Sie ist etwas zwischen einer einfachen BASIC-Befehlserweiterung und einer Spiel-Engine.

Game-Engines wie Game-Maker, AGD (Arcade Game Designer), Unity und viele andere schränken die Vorstellungskraft des Programmierers bis zu einem gewissen Grad ein und zwingen ihn, bestimmte Strukturen zu verwenden, die Logik eines Gegners in einer begrenzten Skriptsprache zu programmieren, Spielbildschirme zu definieren und zu verknüpfen usw.

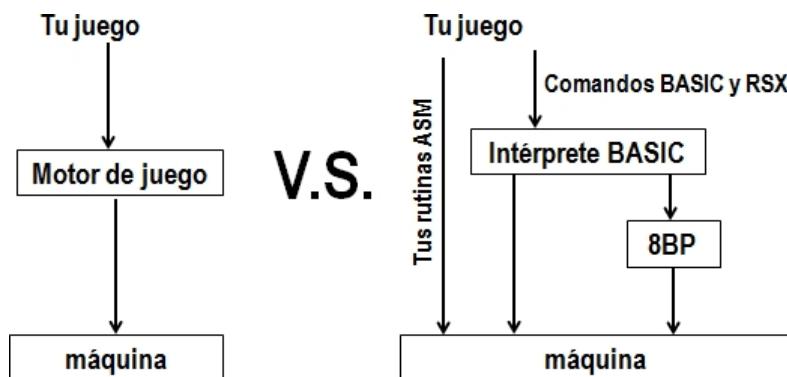


Abb. 2 Spielmotoren gegenüber 8BP

Die 8BP-Bibliothek ist anders. Sie ist eine Bibliothek, die in der Lage ist, schnell auszuführen, was BASIC nicht tun kann. Dinge wie das Drucken von Sprites bei voller Geschwindigkeit oder das Bewegen von Sternenbänken auf dem Bildschirm sind Dinge, die BASIC nicht kann, und 8BP kann sie.

Und er benötigt nur 8 KB!

BASIC ist eine interpretierte Sprache. Das bedeutet, dass der Computer jedes Mal, wenn er eine Programmzeile ausführt, zunächst überprüfen muss, ob es sich um einen gültigen Befehl handelt, indem er die Befehlszeichenfolge mit allen gültigen Befehlszeichenfolgen vergleicht. Anschließend muss er den Ausdruck, die Befehlsparameter und sogar die zulässigen Bereiche für die Werte dieser Parameter syntaktisch überprüfen. Außerdem werden die Parameter im Textformat (ASCII) gelesen und müssen in numerische Daten umgewandelt werden. Sobald all diese Arbeiten erledigt sind, wird mit der Ausführung fortgefahrene. Nun, all diese Arbeit, die in jeder Anweisung ausgeführt wird, unterscheidet ein kompiliertes Programm von einem interpretierten Programm, wie es in BASIC geschrieben wurde.

Durch die Ausstattung von BASIC mit den von 8BP bereitgestellten Befehlen ist es möglich, Spiele in professioneller Qualität zu erstellen, da die von Ihnen programmierte Spiellogik in BASIC ausgeführt werden kann, während CPU-intensive Operationen wie das Drucken auf dem Bildschirm oder die Erkennung von Kollisionen zwischen Sprites usw. von der Bibliothek im Maschinencode ausgeführt werden. Es ist jedoch nicht alles einfach und problemlos. Obwohl die 8BP-Bibliothek Ihnen sehr nützliche Funktionen für Videospiele bietet, müssen Sie sie mit Vorsicht verwenden, da jeder Befehl, den Sie aufrufen, die Parsing-Schicht von BASIC durchläuft, bevor er die Maschinencode-Unterwelt erreicht, in der sich die Funktion befindet, so dass die Leistung nie optimal sein wird. Sie müssen schlau sein und Anweisungen speichern, die Ausführungszeiten von Anweisungen und Teilen Ihres Programms messen und sich Strategien ausdenken, um Ausführungszeit zu sparen. Das ist ein Abenteuer voller Einfallsreichtum und Spaß.

Hier erfahren Sie, wie es geht, und ich werde Sie sogar in eine Technik einführen, die ich "massive logics" genannt habe und die es Ihnen ermöglicht, Ihre Spiele bis zu Grenzen zu beschleunigen, die Sie vielleicht für unmöglich gehalten haben.

Neben der Bibliothek stehen Ihnen ein einfacher, aber vollständiger Sprite- und Grafik-Editor sowie eine Reihe großartiger Werkzeuge zur Verfügung, mit denen Sie das Abenteuer der Programmierung eines Mikrocomputers im 21. Jahrhundert genießen können.

Die folgende "nette" Zeichnung zeigt schematisch die Möglichkeiten, die 8BP bot, und wurde auf einer "Retro"-Messe verwendet. Heutzutage hat es mehr Möglichkeiten.

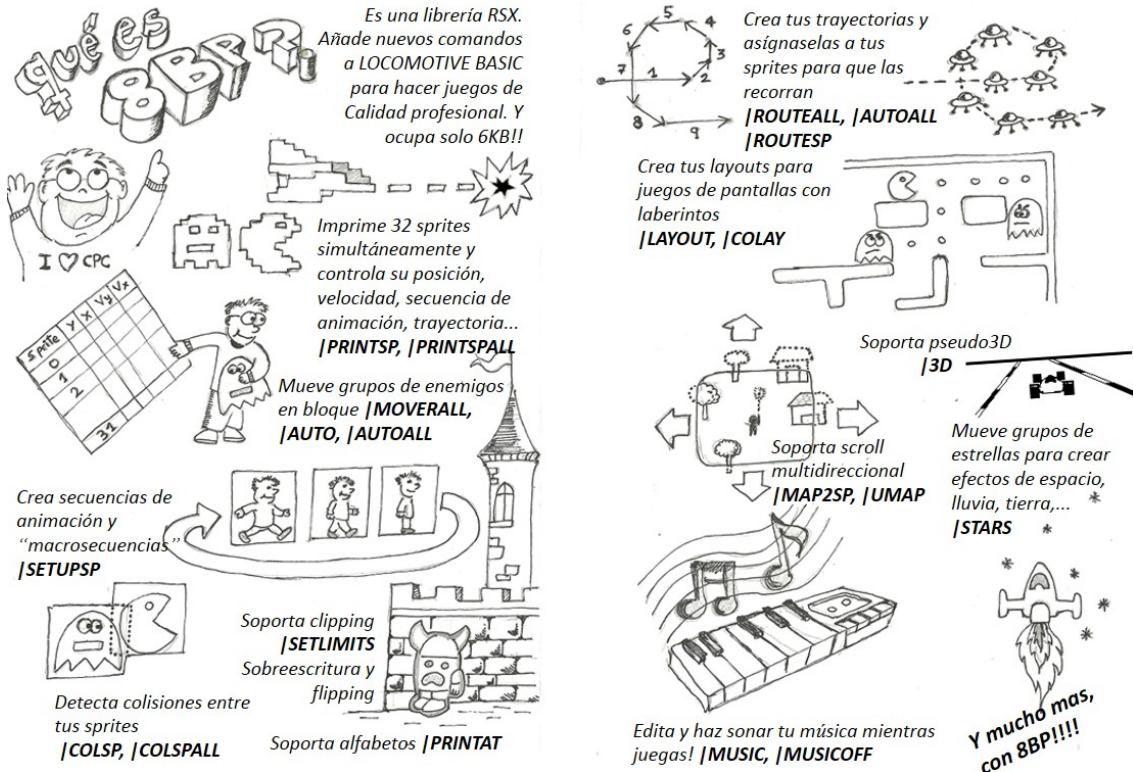


Abb. 3 Zusammenfassung von 8BP (hat derzeit mehr Befehle)

2.1 Was ist eine RSX-Bibliothek?

RSX ist das Akronym für Resident System eXtensions. Bibliotheken wie 8BP, die Befehle zur Erweiterung von BASIC bereitstellen, werden als RSX-Bibliotheken bezeichnet.

Auf dem CPC6128 sind einige der Befehle, die zur Steuerung des Laufwerks verwendet werden, vorinstallierte RSX-Befehle, wie z. B. |TAPE, |DISC, |A, |B, |CPM und andere. Gäbe es diese Funktionalität nicht, müsste jede 8BP-Routine mit einem CALL <Adresse> aufgerufen werden, so dass die Existenz von RSX die Programme verständlicher macht.

Nicht alles ist friedlich und harmonisch. Die Verwendung von RSX ist langsamer als die direkte Verwendung von CALL, und auch wenn wir 10 neue Befehle in einer Bibliothek deklarieren, kann es sein, dass der zehnte Befehl 1 ms länger braucht, um mit der Ausführung zu beginnen als der erste. Die 8BP-Bibliothek hat 27 Befehle und der letzte beginnt 2 ms später mit der Ausführung, weil er am Ende der Liste steht. Dies ist eines der Probleme, wenn man unter dem BASIC-Interpreter arbeitet.

8BP kompensiert dieses Problem, indem es die am häufigsten verwendeten Befehle an den Anfang der Liste stellt und die weniger häufig verwendeten Befehle am Ende der Liste belässt. Wie Sie bald feststellen werden, ist der am häufigsten verwendete Befehl in 8BP |PRINTSPALL, der alle Sprites auf dem Bildschirm ausgibt. Dieser Befehl steht daher ganz oben in der Liste.

2.2 Funktionen von 8BP

Nach dem Laden der Bibliothek mit dem Befehl: LOAD "8BP.BIN" und rufen Sie unter BASIC die Funktion _INSTALL_RSX (im Maschinencode definiert) mit dem BASIC-Befehl auf:

ANRUF &6b78

Folgende Befehle stehen Ihnen zur Verfügung, deren Anwendung Sie mit diesem Buch lernen werden

3D, <Kennzeichen>, #, Offsetdruck 3D, 0	Aktiviert den Pseudo-3D-Projektionsmodus.
ANIMA, #	Ändert den Rahmen eines Sprites entsprechend seiner Sequenz
ANIMALL	Ändert den Frame von Sprites mit aktiviertem Animationsflag (muss nicht aufgerufen werden, ein Flag in der PRINTSPALL-Anweisung reicht aus, um es aufzurufen).
AUTO, #	Automatische Bewegung eines Sprites entsprechend seiner Vy,Vx
AUTOALL, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>.	Bewegung aller Sprites mit automatischem Bewegungsflag auf
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	Erkennt Kollisionen mit dem Layout und gibt 1 zurück, wenn eine Kollision vorliegt. Akzeptiert eine variable Anzahl von Parametern (immer in der gleichen Reihenfolge) von 4 bis keine.
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, Ende COLSP, 33, @collided% COLSP, 34, dy, dx COLSP, #	Gibt das erste Sprite zurück, mit dem # kollidiert. Der Befehl kann mit den Codes 32, 33 und 34 konfiguriert werden.
COLSPALL,@who%, @withwho%. COLSPALL, Kollider COLSPALL	Gibt zurück, wer kollidiert ist (collider) und mit wem er kollidiert ist (collided).
LAYOUT, y, x, @String\$, @String\$, @String\$, @String\$, @String\$, @String\$.	Druckt 8x8 Bildstreifen und füllt das Kartenlayout
LOCATESP, #, y, x	Ändert die Koordinaten eines Sprites (ohne es zu drucken)
MAP2SP, y, x MAP2SP, Status	Erzeugt Sprites, um die Welt in scrollenden Spielen zu zeichnen. Sprites werden mit state = status erstellt
MOVER, #, dy, dx	relative Bewegung eines einzelnen Sprites
MOVERALL, dy,dx MOVERALL	Relative Bewegung aller Sprites mit aktivem Flag für relative Bewegung
MUSIC, C, Flagge, Lied, Geschwindigkeit MUSIC, Flagge, Lied, Geschwindigkeit MUSIK	Eine Melodie beginnt zu spielen. Kanal C kann für die Verwendung mit Effekten ausgeschaltet werden, falls gewünscht.
PEEK, dir, @Variable%	Liest einen 16-Bit-Wert (kann negativ sein)
POKE, dir, wert	einen 16-Bit-Wert eingeben (der auch negativ sein kann)
PRINTAT, flag, y, x, @string	Druckt eine Zeichenkette aus umdefinierbaren "Mini-Zeichen".
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, Bits	druckt ein einzelnes Sprite (# ist seine Nummer) unabhängig vom Statusbyte. Wenn 32 angegeben ist, werden die Hintergrundbits gesetzt
PRINTSPALL, ini, fin, anima, sync PRINTSPALL, Auftragsmodus PRINTSPALL	Druckt alle Sprites mit aktivem Druck-Flag. Beim Aufruf mit einem einzigen Parameter wird der Ordnungsmodus festgelegt.

RINK,tini,Farbe1,Farbe2,...,FarbeN RINK, Sprung	Rotiert einen Satz von Druckfarben nach einem definierbaren Muster, das aus einer beliebigen Anzahl von Druckfarben besteht
ROUTESP, #, Schritte	Lässt dich N Schritte der Sprite-Route auf einmal durchlaufen.
ROUTEALL	Ändert Sie die Sprite-Geschwindigkeit mit dem Pfad-Flag (Sie brauchen es nicht aufzurufen, nur das Flag in AUTOALL).
SETLIMITS, xmin, xmax, ymin, ymax	Definiert das Spielfenster, in dem das Clipping durchgeführt wird.
SETUPSP, #, param_number, value SETUPSP, #, 5, Vy, Vx	Ändert einen Parameter eines Sprites. Wenn Parameter 5 angegeben wird, kann Vx optional angegeben werden.
STARS, initstar, num, color, dy, dx	Schriftrolle aus einer Reihe von Sternen
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Aktualisiert Weltkartenelemente mit einer Teilmenge von Elementen aus einer größeren Karte

Tabelle 2 In der 8BP-Bibliothek verfügbare Befehle

Beachten Sie, dass am Anfang jeder Variablen ein vertikaler Balken erscheint, da es sich um BASIC-"Erweiterungen" handelt. Einige Variablen erscheinen mit dem Symbol "%", um anzugeben, dass es sich um ganze Zahlen handelt (nicht um Dezimalzahlen), aber wenn Sie DEFINT verwenden, um alle Variablen zu erzwingen, dass sie ganze Zahlen sind, brauchen Sie das "%" nicht.

Außerdem haben Sie einen experimentellen Befehl:

|RETROTIME, Datum

Mit diesem Befehl können Sie Ihren CPC in eine Zeitmaschine verwandeln, indem Sie einfach das gewünschte Zieldatum eingeben. Die einzige Einschränkung des Befehls ist, dass Sie ein Datum eingeben müssen, das gleich oder nach dem Geburtsdatum des AMSTRAD CPC, April 1984, liegt,

|RETROTIME, "01/04/1984".

Bitte verwenden Sie diese Funktion mit Vorsicht. Sie könnten ein Zeitparadoxon erzeugen und die Welt zerstören.

Auch wenn Sie im Moment vielleicht noch skeptisch sind, was Sie mit der 8BP-Bibliothek machen können, werden Sie bald feststellen, dass Sie mit dieser Bibliothek und den fortgeschrittenen Programmiertechniken, die Sie in diesem Buch lernen, professionelle Spiele in BASIC entwickeln können - etwas, das Sie vielleicht für unmöglich gehalten haben.

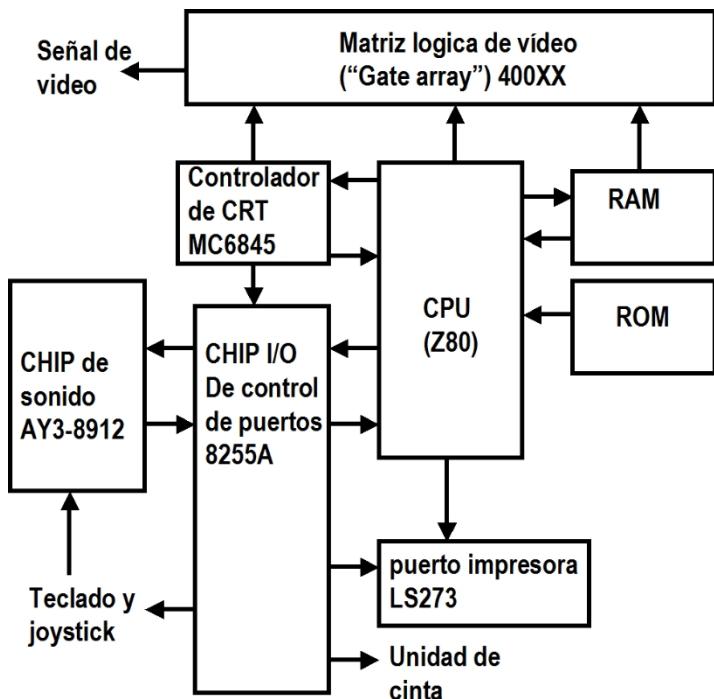
Wichtiger Hinweis für den Programmierer:

Die 8BP-Bibliothek ist auf hohe Geschwindigkeit optimiert. Deshalb prüft sie nicht, ob Sie die Parameter der einzelnen Befehle richtig gesetzt haben und ob sie den richtigen Wert haben. Wenn ein Parameter falsch gesetzt ist, ist es sehr wahrscheinlich, dass der Computer beim Ausführen des Befehls hängen bleibt. Die Überprüfung dieser Dinge erfordert Ausführungszeit, und Zeit ist eine Ressource, die nicht verschwendet werden darf, nicht einmal eine Millisekunde.

2.3 AMSTRAD CPC Architektur

Dieser Abschnitt ist nützlich, um später zu verstehen, wie die 8BP-Bibliothek Speicher verwendet.

Der AMSTRAD ist ein Computer, der auf dem Z80-Mikroprozessor basiert und mit 4MHz läuft. Wie im Architekturdiagramm zu sehen ist, greifen sowohl die CPU als auch das Video-Logik-Array (genannt "Gate-Array") auf den Arbeitsspeicher zu. Um sich abzuwechseln, werden die Speicherzugriffe der CPU verzögert, was zu einer effektiven Geschwindigkeit von 3,3Mhz führt. Das ist immer noch eine ganze Menge Strom.



Der Videospeicher, den wir auf dem Bildschirm sehen, ist ein Teil des 64KB großen RAMs, genauer gesagt die 16KB im oberen Bereich des Speichers. Der Speicher ist von 0 bis 65535 Bytes nummeriert. Nun, die 16KB zwischen Adresse 49152 und 65535 sind der Videospeicher. In hexadezimaler Form wird er als &C000 bis &FFFFF dargestellt.

Abb. 4 Architektur des AMSTRAD

Der Gate-Array-Chip greift 50 Mal pro Sekunde auf den Video-RAM zu, um ein Bild an den Bildschirm zu senden. Bei älteren Computern (z. B. dem Sinclair ZX81) wurde diese Aufgabe dem Prozessor anvertraut, was diesem noch mehr Leistung entzieht.

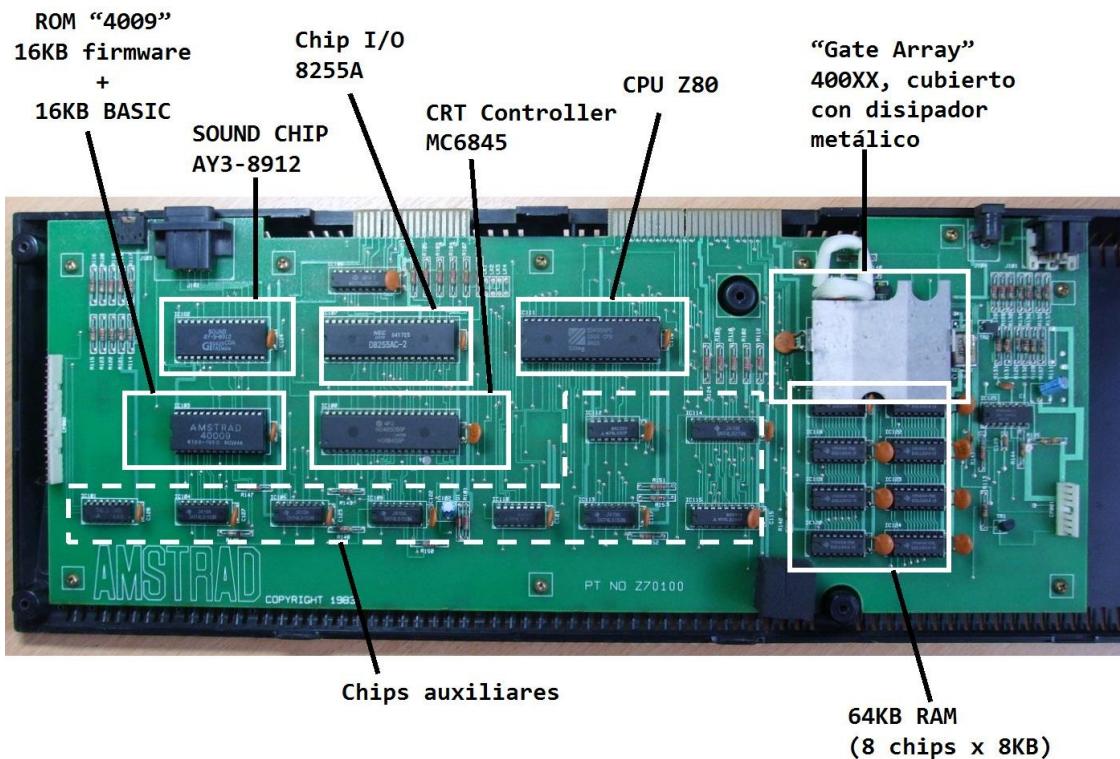


Abb. 5 Kennzeichnung der Bauteile auf der Platine

Das Gate Array ist ein Chip mit vielen Logikgattern, der speziell für AMSTRAD entwickelt wurde. Im ZX Spectrum gibt es einen ähnlichen Chip namens ULA (Uncommitted Logic Array - nicht zu verwechseln mit ALU). Sowohl der Amstrad als auch der ZX sind Chips, die ausschließlich für diese Computer entwickelt wurden. Auf dem ZX wird er nicht nur zur Erzeugung von

wurde das Videobild auch zum Lesen der Tastatur und der Kassetteneingabe verwendet, beim AMSTRAD werden diese Funktionen jedoch von anderen Chips wie dem 8255A übernommen.

Der Z80 hat einen 16-Bit-Adressbus, so dass er nicht mehr als 64 KB adressieren kann. Der Amstrad hat jedoch 64kB RAM und 32kB ROM. Um diese zu adressieren, ist der AMSTRAD in der Lage, zwischen den Bänken "umzuschalten", so dass er z.B. bei einem BASIC-Befehl zu der ROM-Bank wechselt, in der der BASIC-Interpreter gespeichert ist, was sich mit dem 16KB großen Bildschirm überschneidet. Dieser Mechanismus ist einfach und effektiv.

Zusätzlich zu dem ROM, das den 16 KB großen BASIC-Interpreter enthält, der sich im oberen Speicherbereich befindet, gibt es weitere 16 KB ROM im unteren Speicherbereich, wo sich die Firmware-Routinen befinden (was als Betriebssystem dieser Maschine angesehen werden könnte). Insgesamt (BASIC-Interpreter und Firmware) summieren sich diese auf 32 KB.

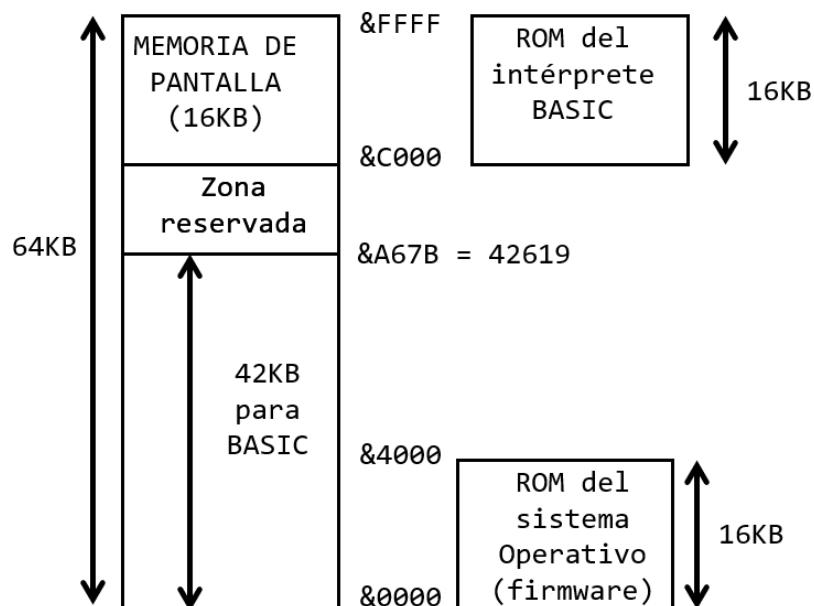


Abb. 6 AMSTRAD-Speicher

Wie in der Speicherkarte zu sehen ist, sind von den 64KB des RAM 16KB (von &C000 bis &FFFFF) Videospeicher. BASIC-Programme können den Bereich von &40 (Adresse 64) bis 42619 belegen, da sich darüber hinaus Systemvariablen befinden. Das bedeutet, dass etwa 42KB für BASIC zur Verfügung stehen, wie wir durch Ausdrucken der Systemvariablen HIMEM (Abkürzung für "High Memory") feststellen können.

```
print HIMEM
42619
Ready
```

Abb. 7 HIMEM-Systemvariable

Die Funktionsweise von BASIC berücksichtigt die Speicherung des Programms in aufsteigenden Adressen ab Position &40. Während der Ausführung müssen die deklarierten Variablen Platz belegen, um die Werte zu speichern, die sie annehmen, und da sie nicht denselben Bereich belegen können, in dem das Programm gespeichert ist, werden sie einfach oberhalb der letzten von der BASIC-Liste belegten Adresse gespeichert.

Auf dem AMSTRAD belegt jede Variable Informationen mit ihrem Namen und ihrem Wert. Eine numerische Variable vom Typ Integer belegt 2 Byte Speicherplatz für den Wert, aber ebenso viele Bytes für den Namen. Reelle Variablen (mit Dezimalzahlen) belegen 5 Bytes. Jedes Mal, wenn wir eine Variable verwenden, verbrauchen wir Speicher, beginnend bei der ersten freien Adresse oberhalb des BASIC-Listings. Wenn wir viele Variablen erstellen und unsere BASIC-Liste sehr lang ist, besteht die Gefahr, dass der Stapel der Variablen den gesamten freien Speicher auffrisst. In diesem Fall wäre das BASIC-Programm beschädigt und würde nicht mehr funktionieren. Sobald das Programm läuft, verbrauchen die Variablen Speicherplatz, und wenn sie den gesamten verfügbaren RAM-Speicher ausfüllen, gibt es eine

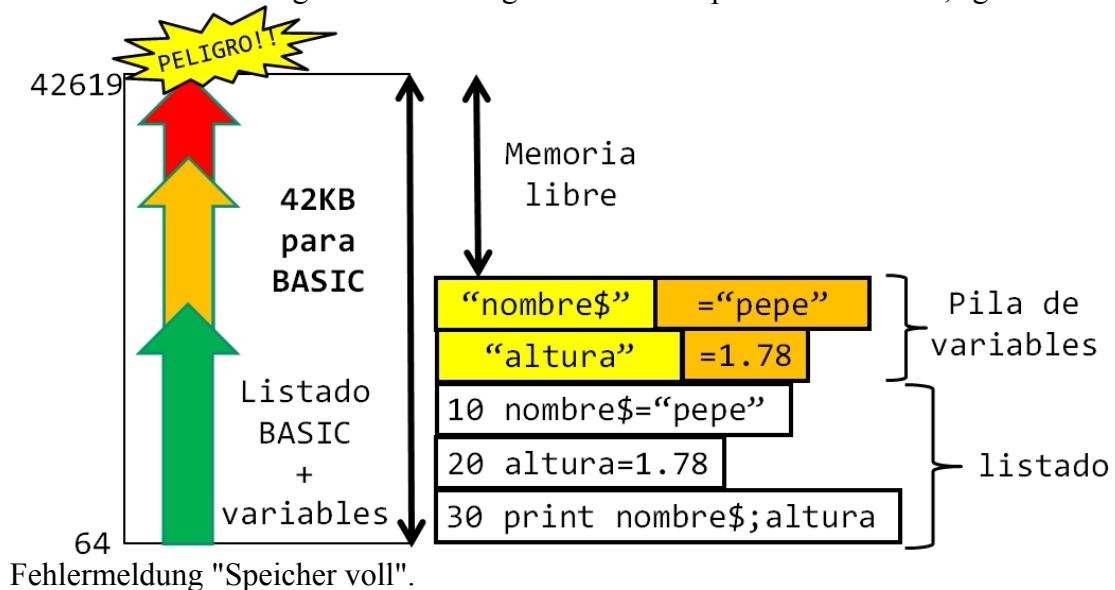


Abb. 8 Wachstum des Speicherverbrauchs von BASIC-Listen und Variablen

Mit dem Befehl **FRE(0)**, dessen Wert Sie ausdrucken oder in eine Variable laden können, können Sie jederzeit überprüfen, wie viel Speicher Sie zur Verfügung haben. Sie können dieses Experiment durchführen, um zu sehen, wie Sie es schaffen, den gesamten Speicher des CPC mit einem Array von Ganzzahlen zu verbrauchen:

```
print fre(0)
42209
Ready
list
10 CLEAR:DEFINT a-z
20 DIM a(21099)
30 PRINT FRE(0)
Ready
run
0
Ready
```

Dieses einfache Programm belegt den gesamten Speicherplatz zwischen der Summe der BASIC-Liste und dem Array.

Beachten Sie, dass FRE(0) anzeigt, dass kein einziges Byte mehr vorhanden ist.

Jedes Mal, wenn einer Literalvariablen wie name\$ ein Wert zugewiesen wird, wird die Variable verschoben, so dass immer weniger Platz zur Verfügung steht, obwohl der zuvor belegte Platz als "verfügbar" gekennzeichnet wird. Wenn einem Programm der Speicher ausgeht, werden alle Variablen komprimiert, wodurch der gesamte "verfügbare" Speicherplatz freigegeben wird. Dieser Effekt wird auch durch die Ausführung von **FRE("")** erreicht. Aber Vorsicht, Amstrad BASIC erlaubt es Ihnen nicht, einfach **FRE("")** auszuführen, weil es Ihnen einen "Syntaxfehler" gibt. Sie müssen mindestens eine Variable zuweisen, zum Beispiel **p=FRE("")**. Dieser Befehl führt das aus, was heute in einigen Sprachen als "Garbage Collection" bekannt ist.

Wenn Sie während der Ausführung eines Programms ein Problem mit vollem Speicher haben, können Sie es möglicherweise lösen, indem Sie in regelmäßigen Abständen eine Zeile wie diese ausführen:

```
100 c=c+1
110 wenn c und 63 =0 dann 120 sonst p=fre("") :FRE alle 64
Zyklen
das Programm wird fortgesetzt
```

Diese Lösung funktioniert nur, wenn das Problem auf eine Anweisung zurückzuführen ist, die etwas mehr verbraucht, als kurz vor der automatischen Ausführung der FRE noch frei ist. Wenn das der Fall ist, wird diese Lösung funktionieren. Es ist jedoch sehr "selten", das Problem auf diese Weise zu lösen, denn wenn der Amstrad keinen Speicher hat, führt er FRE automatisch aus und braucht es theoretisch nicht zu tun. Eine andere einfachere (und effektivere) Lösung besteht darin, REM-Zeilen zu löschen oder zu kürzen, um dem Amstrad etwas mehr freien Speicher zur Verfügung zu stellen. Wenn genügend Speicher vorhanden ist und der Speicher trotzdem voll ist, handelt es sich um ein Problem mit zu vielen GOSUB ohne RETURN.

Jede numerische Variable verbraucht den folgenden Speicherplatz:

- den Namen der Variablen: N Bytes
- den Wert, wobei das letzte Byte auf Null gesetzt wird, um das Ende des Literals anzuzeigen: 2 Bytes

Jedes Variablenliteral verbraucht den folgenden Speicherplatz:

- den Namen der Variablen: N Bytes
- Speicheradresse (oder "Zeiger"), an der sein Wert beginnt: 2 Bytes
- den Wert, wobei das letzte Byte auf Null gesetzt wird, um das Ende des Literales anzuzeigen: N Bytes

Wenn eine Literalvariable durch Zuweisung eines neuen Wertes verschoben wird, wird der neue Wert in den verfügbaren Speicherbereich geladen und der Zeiger neu zugewiesen, so dass er auf die neue Adresse zeigt, wobei der alte Wert nicht mehr von einer Variablen angezeigt wird. Der frühere Speicherplatz oder die "Lücke" ist verfügbar, aber eine Bereinigung ist erforderlich, um alle Variablen zu verdichten und alle verfügbaren Lücken freizugeben.

Sehen wir uns ein einfaches Beispiel an, das Ihnen den verfügbaren Speicherplatz beim Verschieben einer literalen (oder "String"-)Variable zeigt, wobei offensichtlich immer mehr Speicher verbraucht wird, obwohl der ungenutzte Speicher verfügbar bleibt (es handelt sich um "Löcher" im Variablenstapel). Wenn Sie das Gleiche mit einer numerischen Variable versuchen, wird der Speicherbedarf konstant sein, da numerische Variablen immer den gleichen Platz belegen und bei einer Wertänderung nicht verschoben werden, während literalen Variablen (oder "Strings") bei einer Wertänderung ihre Länge ändern.

```

10 Zahl=rnd*100
20 c$=str$(Zahl)
30 print fre(0)
40 goto 10

```

Wie Sie sehen können, steht bei jeder Iteration weniger Speicher zur Verfügung, aber wenn Sie das Programm unbegrenzt laufen lassen, führt AMTRAD eine Prozedur aus, um den ungenutzten Speicher aufzuräumen, so dass wieder Speicher zur Verfügung steht. Diese Prozedur ist die gleiche, die Sie mit FRE(" ") ausführen.

Nicht zu verwechseln mit dem CLEAR-Befehl, der Platz freimacht, indem er alle Variablen aus dem Variablenstapel entfernt.

Es ist ratsam, FRE(" ") in regelmäßigen Abständen in Ihrem Programm auszuführen, da der Vorgang des Verdichtens aller Variablen, wenn der gesamte Speicher verbraucht ist, mehr Arbeit macht (und daher langsamer ist), als wenn Sie regelmäßig ein FRE(" ") ausführen, das nur wenig Arbeit zu tun hat.

Wenn Ihr Programm nach einiger Zeit die Fehlermeldung "Speicher voll" ausgibt, versuchen Sie, in regelmäßigen Abständen ein FRE("") auszuführen, löschen Sie "REM"-Zeilen, um mehr freien Speicher zu erhalten, oder überprüfen Sie, ob es sich nicht um einen Überschuss handelt.

Verschachtelung von GOSUB, was der häufigste Fehler ist.

Ready
run
42150
42137
42124
42111
42098
42086
42073
42060
42047
42034
42021
42008
41995
41982
41969
41956
41943
41931
41918

<nach vielen Iterationen>

141
128
115
102
89
76
64
51
38
25
12
42137
42124
42111
42098
42085
42072
42060

Zum Schluß noch eine Kuriosität: der CPC 464 gibt Ihnen einen FRE(0) von 43.553 Bytes frei, während der 6128 weniger Speicher, nämlich 42.249 Bytes, zur Verfügung stellt. Dies steht in direktem Zusammenhang mit der Tatsache, daß man bei einem PRINT HIMEM auf dem CPC464 43903 erhält, während man auf dem CPC6128 42619 erhält.

2.3.1 GOSUB /RETURN Stapel

Jedes Mal, wenn Sie GOSUB im Amstrad BASIC ausführen, muss das System darauf hinweisen, wohin es zurückkehren muss, wenn Sie RETURN eingeben. Nun, der Amstrad CPC hat einen Stapel von 83 Positionen, um die Sprünge zu speichern. Es kommt häufig vor, dass man beim Programmieren einen Fehler macht und in einigen IF des Unterprogramms mit einem GOTO zurückkehrt (d.h. nicht zurückkehrt), so dass sich diese Sprünge ansammeln, wenn man nicht aufpasst, und man kann während der

Ausführung des Programms einen "Speicher voll"-Fehler erhalten.

<pre> 10 GOSUB 100 20 REM hier kommt nie an 100 i=i+1 110 DRUCKEN i 120 GOTO 10 </pre>	<pre> 73 74 75 76 77 78 79 80 81 82 83 Memory full in 100 Ready </pre>
---	--

Auch wenn Sie in diesem Beispiel den verbleibenden Speicherplatz ausdrucken, indem Sie die Zeile 110 durch :

110 print i: print fre(0)

Sie werden sehen, dass der verfügbare Speicher nicht abnimmt, und obwohl Sie fast den gesamten Speicher frei haben, meldet das Amstrad Speicher voll. Das liegt an der Verschachtelung von **GOSUBs**, nicht daran, dass kein freier Speicher vorhanden ist. Es sind nur 83 **GOSUB** ohne **RETURN** erlaubt.

Ich kann Ihnen nicht mit Sicherheit sagen, in welchem Speicherbereich der Amstrad BASIC-Interpreter die 83 Adressen für RETURN speichert, aber es ist wahrscheinlich im Systemspeicherbereich (6KB von 42619 bis 49152). Es handelt sich um einen 6KB großen Bereich direkt vor dem Bildschirmspeicher (49152 ist &C0000), in dem BASIC die wiederbeschreibbaren Zeichen und auch den "Stack" speichert. Dies ist ein Bereich, in dem die Speicheradresse gespeichert wird, an der sich ein Programm befindet, bevor es zu einer Routine springt, so dass es an die Stelle zurückkehren kann, an der es sich befand, wenn die Routine endet. Er wird in niedriger Ebene (Assemblersprache) verwendet. Der Stack wächst zu niedrigeren Adressen, wenn er Adressen speichert (d. h. er beginnt bei 49152 und nimmt immer kleinere Adressen auf), und bei der Rückkehr aus einer Routine wird er wieder kleiner. Man könnte meinen, dass, wenn eine Routine eine andere Routine aufruft und diese wiederum eine andere Routine aufruft und so weiter, der Stack so sehr wachsen würde, dass er den "System"-Bereich verlässt und in andere Bereiche eindringt, aber keine Sorge, 8BP hält den Stack unter Kontrolle und das Amstrad BASIC auch.

2.3.2 Ein Experiment, um das ROM mit Winape zu sehen

Sie können Winape benutzen, um zu sehen, was im ROM des BASIC-Interpreters enthalten ist, das sich mit den Bildschirmadressen überschneidet, und dasselbe gilt für das ROM des Betriebssystems.

Geben Sie im Emulationsfenster
STOCHERN &C000, &FF

Drücken Sie dann die Pausetaste des Emulators und Sie sehen einen Bildschirm mit Speicheradressen und Werten. Suchen Sie die Adresse &C000 und ihren Inhalt. Sie sollte &FF lauten. Unten sehen Sie etwas wie:

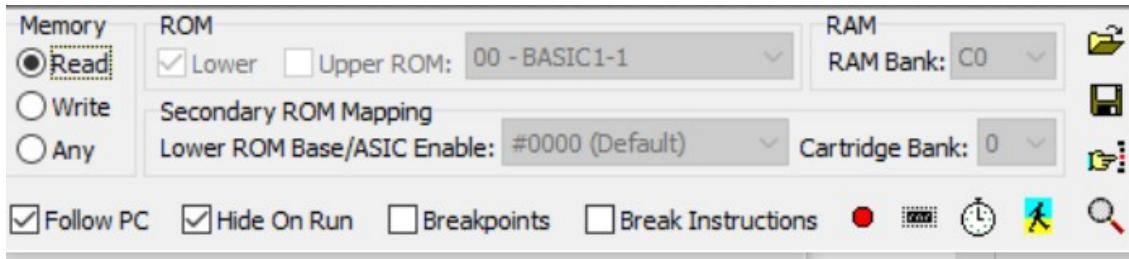


Abb. 9 Winape zeigt RAM

Nun, wenn Sie es jetzt so einrichten:

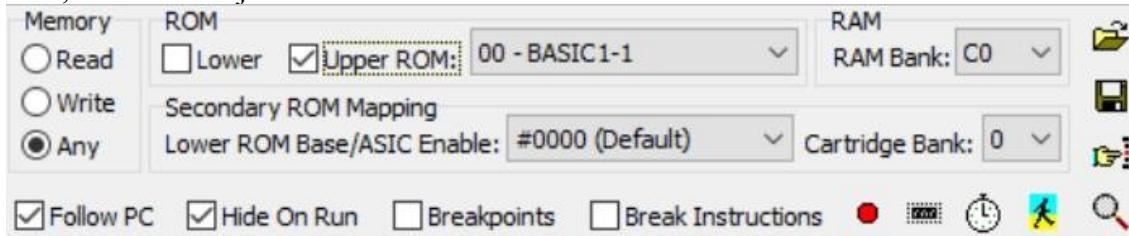


Abb. 10 Winape zeigt ROM

Sie werden den Inhalt der Adresse &C000 in der ROM-Bank sehen, die den BASIC-Interpreter enthält. Es ist sehr interessant, dies mit Winape tun zu können.

2.4 8BP Speicherabbildung und Montageoptionen

Da das Amstrad-BASIC zunächst die niedrigsten Adressen verbraucht, wird die 8BP-Bibliothek in den höchsten verfügbaren Speicherbereich geladen, damit sie "koexistieren" kann. Es ist wichtig zu verstehen, wie BASIC funktioniert, um die Bibliothek zu verwenden, da Sie einen BASIC-Befehl "**MEMORY**" verwenden müssen.

<pre> 10 a=5 20 k=@a 30 PRINT k Ready run 411 Ready </pre>	<p>Der Text eines in BASIC geschriebenen Programms wird ab der Adresse &40 gespeichert (dezimal ist das 64, eine sehr "niedrige" Adresse), und die Variablen, die Ihr Programm erstellt, werden direkt hinter dem von der Liste belegten Speicherplatz gespeichert. Das folgende Beispiel zeigt, wo die Variable "A" gespeichert ist, nämlich an der Adresse 441.</p>
--	---

Die BASIC-Liste und die Variablen können sehr groß werden, wenn Ihr Programm sehr groß ist, und könnten in den Speicherbereich eindringen, in dem die 8BP-Routinen gespeichert sind. Um dies zu vermeiden, müssen Sie einen **MEMORY-Befehl** ausführen, der den Speicherbereich schützt, in dem 8BP, Ihre Grafiken und Ihre Musik gespeichert sind. Der MEMORY-Befehl ist wie eine "**Barriere**", die BASIC und seine Variablen daran hindert, den Speicher über die von uns festgelegte Grenze hinaus zu belegen. Falls dies doch geschieht, wird die Fehlermeldung "**MEMORY FULL**" ausgegeben und die Arbeit eingestellt.

Funciones de 8BP,tus gráficos y tu música



Programa BASIC y variables

O bien, programa en C y sus variables

Die 8BP-Bibliothek lässt zwischen 24 und 25 KB für Ihr BASIC-Listing oder Ihr C-kompiliertes Programm frei (Sie können auch in C in 8BP programmieren). Ab der Version V42 von 8BP ist es möglich, verschiedene **Assembler-Optionen** zu wählen, abhängig von der Art des Spiels, das Sie entwickeln wollen. Die Idee ist einfach: Wenn Sie ein Spiel mit Scrolling machen wollen, brauchen Sie die Befehle, die sich mit Scrolling befassen, aber Sie brauchen nicht die Befehle, die sich mit dem Zeichnen von Labyrinthen (LAYOUT in 8BP) oder dem Erkennen von Kollisionen mit den Labyrinthwänden befassen. Das heißt, dass ab V42 einige 8BP-Funktionen **keinen Speicher mehr verbrauchen, wenn sie nicht benötigt werden**. So kann 8BP Ihnen mehr freien Speicher für Ihr BASIC-Listing oder Ihr C-Programm zur Verfügung stellen. Je nach Art des Spiels, das Sie programmieren wollen, müssen Sie eine **Assembly-Option** wählen. Diese Tabelle gibt einen Überblick über die verfügbaren Optionen. Sie werden bald den Befehl SAVE verstehen, der in jeder Beschreibung erscheint. Diese Tabelle ist auch in einem Anhang verfügbar

Option	Beschreibung der Option	Beispiel für ein typisches Spiel
0	Sie können jedes Spiel spielen Alle Befehle verfügbar Sie müssen MEMORY 23499 verwenden Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP0.bin",b,23500,19119	jeder
1	Labyrinth- oder Screen-Passing-Spiele Sie müssen MEMORY 24999 verwenden In diesem Modus nicht verfügbar: MAP2SP, UMAP, 3D Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP1.bin",b,25000,17619	
	Für Rollenspiele müssen Sie MEMORY 24799 verwenden. In diesem Modus nicht verfügbar: LAYOUT, COLAY, 3D Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP2.bin", b,24800,17819	

Für Spiele mit Pseudo-3D müssen Sie **MEMORY 23999** verwenden.
In diesem Modus nicht verfügbar:
|LAYOUT, |COLAY
Zum Speichern von Bibliothek + Grafiken + Musik:
SAVE "8BP3.bin", b,24000,18619



Wie Sie sehen können, ist die Option 1 diejenige, die Ihnen den meisten freien Speicherplatz für Ihr Programm lässt, da sie Ihnen 25 KB für Ihr Spiel überlässt. Bei Option 3 sind es 24 KB und bei Option Null 23,5 KB. Ich empfehle Ihnen nicht, die Option Null zu verwenden, es sei denn, Sie brauchen sie wirklich. Es ist besser, die richtige Option für Ihr Spiel zu wählen, dann haben Sie mehr Speicher zur Verfügung.

Diese 24-25 KB sind für Ihr Listing frei, da die Musik und die Grafiken Ihres Spiels in einem anderen "höheren" Bereich gespeichert werden. Zum Beispiel, mit Option 1 haben Sie:

- 25 KB frei für Ihr Listing (BASIC oder C) und Variablen
- 1,5 KB kostenlos für Ihre Musik
- 8,5 KB frei für Ihre Grafiken

TOTAL: 35 KB frei für Ihr Spiel

Die von Ihnen gewählte Assembly-Option wird in einer 8BP-Datei namens "make_all_mygame.asm" definiert. Diese Datei enthält eine Zeile, die **Sie bearbeiten müssen, um die von Ihnen bevorzugte Option zu wählen.**

```

; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos

; DESDE LA V42 EXISTEN "OPCIONES" DE ENSAMBLAJE
; -----
; ASSEMBLING_OPTION = 0 --> todos los comandos disponibles.

; ASSEMBLING_OPTION = 1 --> para juegos de laberintos. MEMORY 25000
;                               disponibles los comandos |LAYOUT, |COLAY
;
; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 24800
;                               disponibles los comandos |MAP2SP, |UMA
;
; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 24000
;                               disponible comando |3D
;
; ASSEMBLING_OPTION = 4 --> uso futuro

let ASSEMBLING_OPTION = 1
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"

;-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"

;-----GRAFICOS-----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos_mygame.asm"

```

Bevor Sie die Bibliothek laden, müssen Sie den **MEMORY-Befehl** mit dem Limit ausführen, das mit dem von Ihnen für Ihren Spieltyp gewählten Assembler-Modus verbunden ist. Ab Version V42 sind je nach gewählter Assembler-Option 24KB, 24,8 KB oder 25KB frei.

Unabhängig davon, welche Assembler-Option Sie wählen, sind die Speicheradressen, an denen sich alle Befehle befinden, genau dieselben (Anhang XI). Für

Zum Beispiel kann der Befehl LAYOUT aufgerufen werden, wenn Sie die Montageoption 1 oder 2 wählen, aber wenn Sie die Montageoption 2 verwenden, wird der Aufruf des Befehls LAYOUT absolut nichts bewirken, ebenso wie der Befehl MAP2SP, wenn Sie die Montageoption 1 verwenden.

```

AMSTRAD CPC464 SPEICHERKARTE von 8BP

; ; &FFFF +-----+
; | Anzeige + 8 versteckte Segmente von je 48 Byte
; &C000 +-----+
; | System (umdefinierbare Symbole, Stapelzeiger, usw.)
; 42619 +-----+
; | Bank mit 40 Sternen (von 42540 bis 42619 = 80 Bytes)
; 42540 +-----+
; | Zeichen-Layout-Map (25x20 =500 Bytes)
; | und Weltkarte (bis zu 82 Elemente passen in 500 Bytes)
; | Beide sind im selben Speicherbereich gespeichert.
; | weil man entweder das eine oder das andere benutzt.
; 42040 +-----+
; | Sprites (fast 8,5KB für
; | Zeichengrafiken 8440 Bytes, wenn es keine Sequenzen und
; | keine Routen gibt)
; | Auch Alphabetbilder werden hier gespeichert.
; | Routendefinitionen (jeweils mit variabler Länge)
; +-----+
; | Animationssequenzen mit 8 Bildern (je 16 Byte)
; | und Gruppen von Animationssequenzen (Makro-
; 33600 Sequenzen)
; | Lieder
; | (1500 Bytes für Musik, die mit WYZtracker 2.0.1.0
; 32100 +-----+ bearbeitet wurde)
; | 8BP-Routinen (8100 Bytes oder 7100 Bytes)
; | hier sind alle Routinen und die Sprite-Tabelle
; | enthält den Musik-Player "wyz" 2.0.1.0
; 25000 +-----+
; | IHRE BASIS- oder C-LISTE
; | 24KB, 24,8 KB oder bis zu 25KB frei für BASIC oder C, je
; | nachdem, welche Assembleroption Sie für 8BP
; | verwenden
; |
; 0 +-----+

```

Abb. 11 Speicher mit 8BP

Wenn Ihnen der Platz für Grafiken oder Musik ausgeht und Sie mehr brauchen, können Sie mit 8BP Bilder und Musik in anderen Bereichen unterbringen (unterhalb der Adresse 24000), und es wird gut funktionieren.

3 Erforderliche Werkzeuge

Winape: Emulator für Windows OS mit Editor zum Bearbeiten und Testen Ihrer BASIC-Programme. Und auch zum Zusammenstellen von Grafiken und Musik.

SPEDIT: ("Simple Sprite Editor") BASIC-Tool zur Bearbeitung Ihrer Grafiken. Die Ausgabe von spedit ist Assembler-Code, der an den Amstrad CPC-Drucker gesendet wird. Wenn Sie das Tool in Winape ausführen, wird der Drucker auf eine Textdatei umgeleitet, so dass Ihre Grafiken in einer txt-Datei gespeichert werden. Dieses Tool wurde als Ergänzung zur 8BP-Bibliothek entwickelt und die Grafiken aller Spiele, die ich erstellt habe, wurden mit SPEDIT erstellt.

Wyztracker: zum Komponieren von Musik, unter Windows. Das Programm, das die von Wyztracker komponierten Melodien abspielen kann, ist Wyzplayer, das in 8BP integriert ist. Nachdem Sie die Musik zusammengestellt haben, können Sie sie mit einem einfachen Befehl abspielen lassen |MUSIC

8BP-Bibliothek: Installieren Sie neue Befehle, die von BASIC aus für Ihr Programm zugänglich sind. Wie Sie sehen werden, wird dies das "Herz" sein, das die Maschinerie antreibt, die Sie bauen.

CPCDiskXP : ermöglicht es Ihnen, eine 3,5"-Diskette aufzunehmen, die Sie dann in Ihren CPC6128 einlegen können, wenn Sie ein Kabel zum Anschluss eines Diskettenlaufwerks haben. Wenn Sie ein CPC464-Audioband erstellen wollen, ist dieses Tool nicht notwendig.

2CDT: unverzichtbares Werkzeug zur Erstellung von .cdt-Dateien. Normalerweise extrahiere ich die Dateien aus einer .dsk-Datei mit CPCDiskXP auf die Windows-Festplatte und verwende dann 2cdt, um die cdt-Datei zu erstellen.

Tape2wav: Werkzeug zum Erstellen von .wav-Dateien aus .cdt-Dateien

OPTIONAL:

ConvImgCPC: Upload-Bild-Editor für Ihre Spiele. Konvertiert auch von BMP. Programmiert von Ludovic Deplanque ("DEMONIAK")

RGAS: (Retro Game Asset Studio) leistungsfähiger Sprite-Editor, der aus dem AMSprite-Tool hervorgegangen ist und von Lachlan Keown entwickelt wurde. Dieser Sprite-Editor ist 8BP-kompatibel und läuft unter Windows. Wenn Sie aus Spedit herausgewachsen sind, könnte dies die beste Option sein.

NICHT EMPFOHLEN:

Fabacom: Compiler, der im AMSTRAD CPC 6128 oder im Winape-Emulator ausgeführt werden kann, um Ihr BASIC-Programm zu komplizieren und schneller laufen zu lassen. Es ist kompatibel mit den 8BP-Bibliotheksaufrufen. Er wird jedoch aus mehreren Gründen nicht empfohlen:

- Ihr Programm wird viel mehr Platz benötigen, da fabacom zusätzliche 10 KB für seine Bibliotheken benötigt, und auch nach dem Kompilieren Ihres Programms nimmt es immer noch die gleiche Menge an Platz ein.

so dass ein 10KB BASIC-Programm in ein 20KB BASIC-Programm umgewandelt wird.

- Es gibt einige dokumentierte Inkompatibilitätsprobleme dieses Compilers mit einigen BASIC-Befehlen.
- Außerdem können Sie, wie Sie im Laufe dieses Buches sehen werden, auch ohne Kompilierung eine sehr hohe Geschwindigkeit erreichen.

CPC BASIC Compiler: ausführbarer Compiler für Windows. Er ist kompatibel mit den Befehlsaufrufen der 8BP-Bibliothek. Im Gegensatz zu fabacom benötigt das kompilierte Programm nur etwa 5KB extra, allerdings reserviert er 16KB für die Ausführung, so dass er kaum Platz für Ihr Programm lässt, da er insgesamt 20 KB "stiehlt". Und es ist nicht 100%ig Lok-BASIC-kompatibel.

Der Geschwindigkeitsgewinn mit fabacom und CPC Basic Compiler kann je nach Spiel bis zu 50% betragen. Das heißt, ein Spiel, das mit 20 FPS läuft, würde mit 30 FPS laufen. Das ist nicht schlecht, aber bedenken Sie, dass wir von interpretiertem BASIC zu Maschinencode übergegangen sind, und normalerweise heißt es, dass die Geschwindigkeit mindestens mit 100 multipliziert werden muss (wir würden über eine Steigerung von 10000% sprechen). Wir haben jedoch nur 50% gewonnen. Der Grund für diesen "geringen" Gewinn ist, dass die 8BP-Befehle bereits die ganze harte Arbeit machen und der Compiler nur den weniger schweren Teil, die Spiellogik, in Maschinencode übersetzt.

Schließlich sollten Sie wissen, dass, wenn Sie wollen, dass Ihre Programme viel schneller laufen, seit v40 von 8BP gibt es C-Sprachunterstützung, so dass Sie entweder Ihr ganzes Spiel direkt in C programmieren können oder in BASIC programmieren und nur den "Spielzyklus" in C übersetzen. Es gibt ein Kapitel in diesem Handbuch, das ausschließlich diesem Thema gewidmet ist.

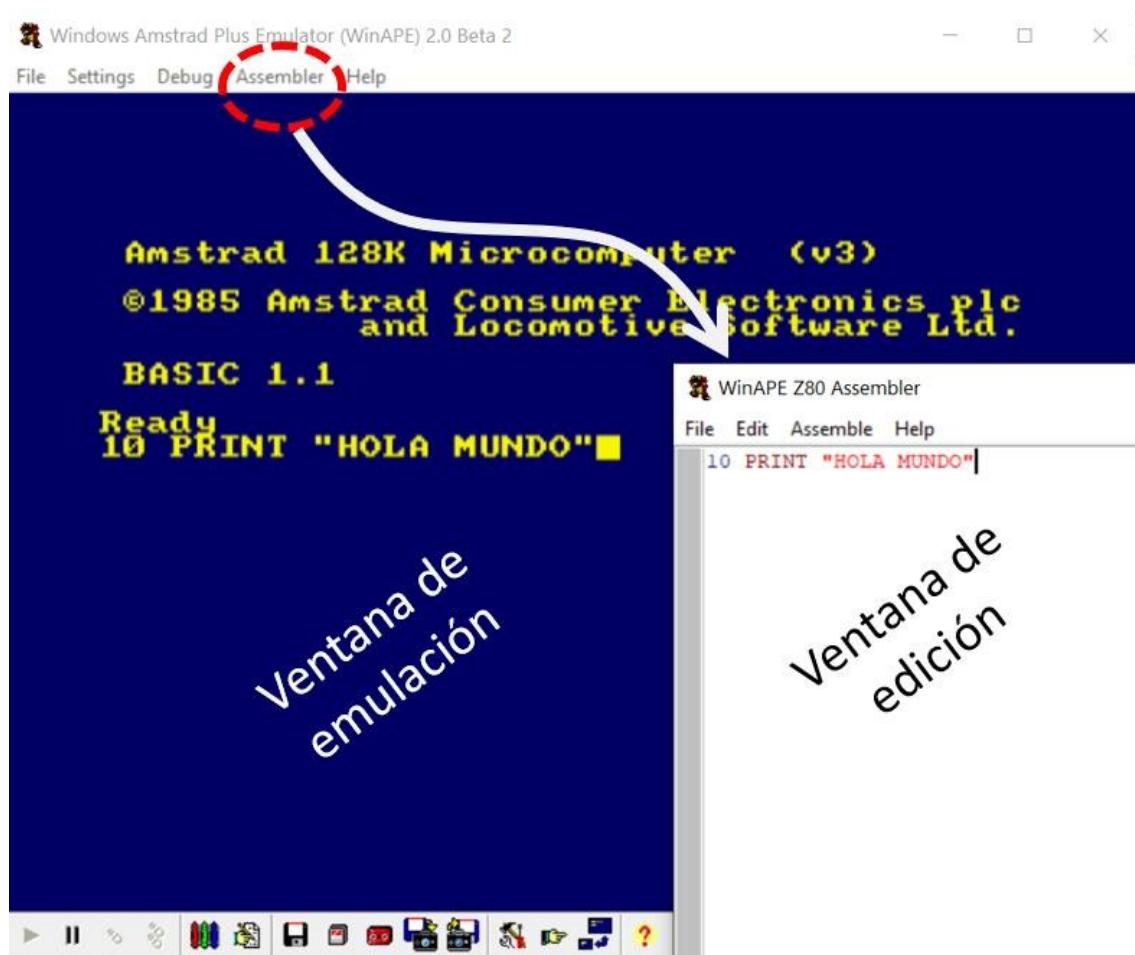
4 Erste Schritte mit 8BP

4.1 winape installieren

Als erstes müssen Sie die neueste Version von **Winape** installieren, einem Amstrad-Emulator, Editor und Assembler. Sie können ihn von www.winape.net herunterladen.

4.2 Vertraut werden mit winape: "Hallo Welt".

Sobald Winape installiert ist, machen Sie sich mit ihm vertraut, indem Sie einige Amstrad-Spiele ausprobieren und versuchen, die Konfiguration zu ändern. Versuchen Sie, das eingebaute Assembler-Menü zu öffnen und ein "Hallo Welt" im Assembler-Fenster zu editieren. Kopieren Sie dann den Text und fügen Sie ihn in das Emulationsfenster ein. Sie werden sehen, wie er Zeichen für Zeichen eingefügt wird



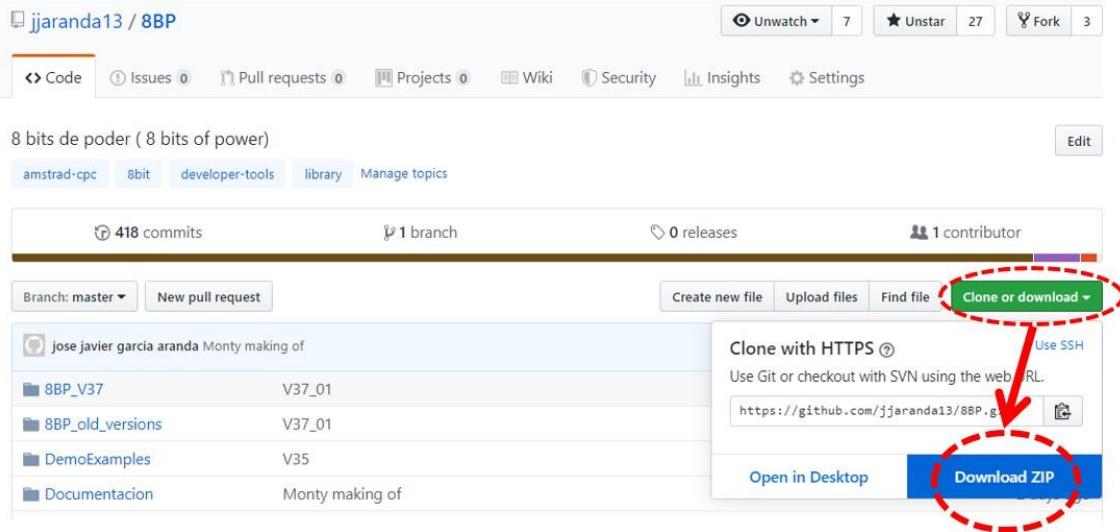
Wenn Sie ein längeres Programm erstellen, um es in das Emulationsfenster zu kopieren, ist es sehr interessant, die Option Einstellungen->Hohe Geschwindigkeit zu verwenden. Sie werden sehen, wie schnell es kopiert wird.

4.3 Download der 8BP-Bibliothek

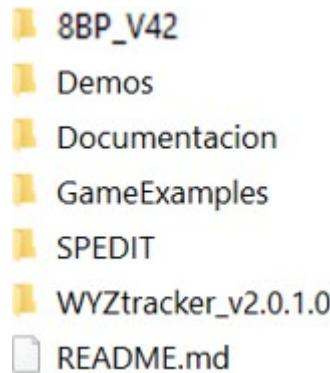
Wir kommen jetzt zum interessanten Teil. Rufen Sie die Website <https://github.com/jjaranda13/8BP> auf und laden Sie die Zip-Datei herunter

(<https://github.com/jjaranda13/8BP/archive/master.zip>). Um dies zu tun

können Sie einfach auf die grüne Schaltfläche "Klonen oder Herunterladen" klicken und dann die Zip-Datei auswählen.



Nachdem Sie es heruntergeladen haben, entpacken Sie es in ein Verzeichnis Ihrer Wahl. Sie erhalten dann das folgende Ergebnis:



4.4 Ausführen der Demos

Jetzt werden wir einen ersten Kontakt mit 8BP herstellen, indem wir uns einige Demos ansehen. Gehen Sie in das Verzeichnis Demos. Darin finden Sie eine Reihe von Unterordnern: **ASM, BASIC, C, DSK, MUSIC, ASM, BASIC, C, DSK, MUSIC**.

Gehen Sie zu DSK . Dort finden Sie eine .dsk-Datei mit den Demos. Gehen Sie in winape auf das Menü Datei.

>Laufwerk A-> Disk-Image einlegen und die Demodatei auswählen

Nach der Auswahl geben Sie im Fenster der Amstrad-Emulation **CAT** ein, um die Dateien anzuzeigen.

cat

Drive A: user 0

8BP0	.BIN	18K	DEMO15	.BAS	2K
8BP1	.BIN	18K	DEMO2	.BAS	2K
8BP2	.BIN	19K	DEMO3	.BAS	1K
CICLO	.BIN	4K	DEMO4	.BAS	2K
DEMO1	.BAS	2K	DEMO5	.BAS	2K
DEMO10	.BAS	2K	DEMO6	.BAS	1K
DEMO11	.BAS	1K	DEMO7	.BAS	2K
DEMO11	.BIN	1K	DEMO8	.BAS	1K
DEMO12	.BAS	3K	DEMO9	.BAS	1K
DEMO13	.BAS	2K	LOADER	.BAS	2K
DEMO14	.BAS	3K			

89K free

Ready

Jede .BAS-Datei ist eine Demo, in der Sie einige der Funktionen von 8BP sehen können (nicht alle Funktionen sind in den Demos zu sehen, aber es sind einige repräsentative). Führen Sie den **RUN-Befehl "LOADER.BAS"** aus. Sie erhalten dann das folgende Menü:

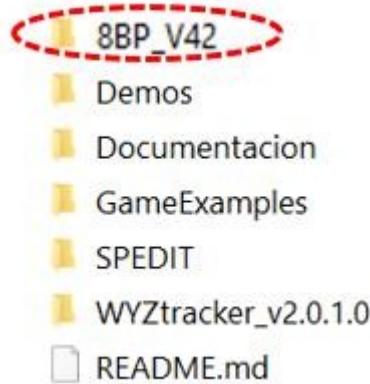


Sie können jetzt eine Demo auswählen und sie ausprobieren. Viel Spaß! Im nächsten und letzten Schritt werden wir mit der Erstellung eines Spiels beginnen.

4.5 Erstellen Sie Ihr erstes Programm mit 8BP

Wir haben eine .dsk-Datei getestet, die viele Demos mit Grafiken und Musik enthält. Das Verzeichnis **Demos/ASM** und das Verzeichnis **Demos/MUSIC** enthalten die Grafiken und die Musik der von Ihnen getesteten Demos. Wenn Sie jedoch ein eigenes Spiel oder eine eigene Demo erstellen möchten, ist es besser, mit "sauberer" Dateien ohne alle Grafiken zu beginnen, die für die von Ihnen getesteten Demos erforderlich sind.

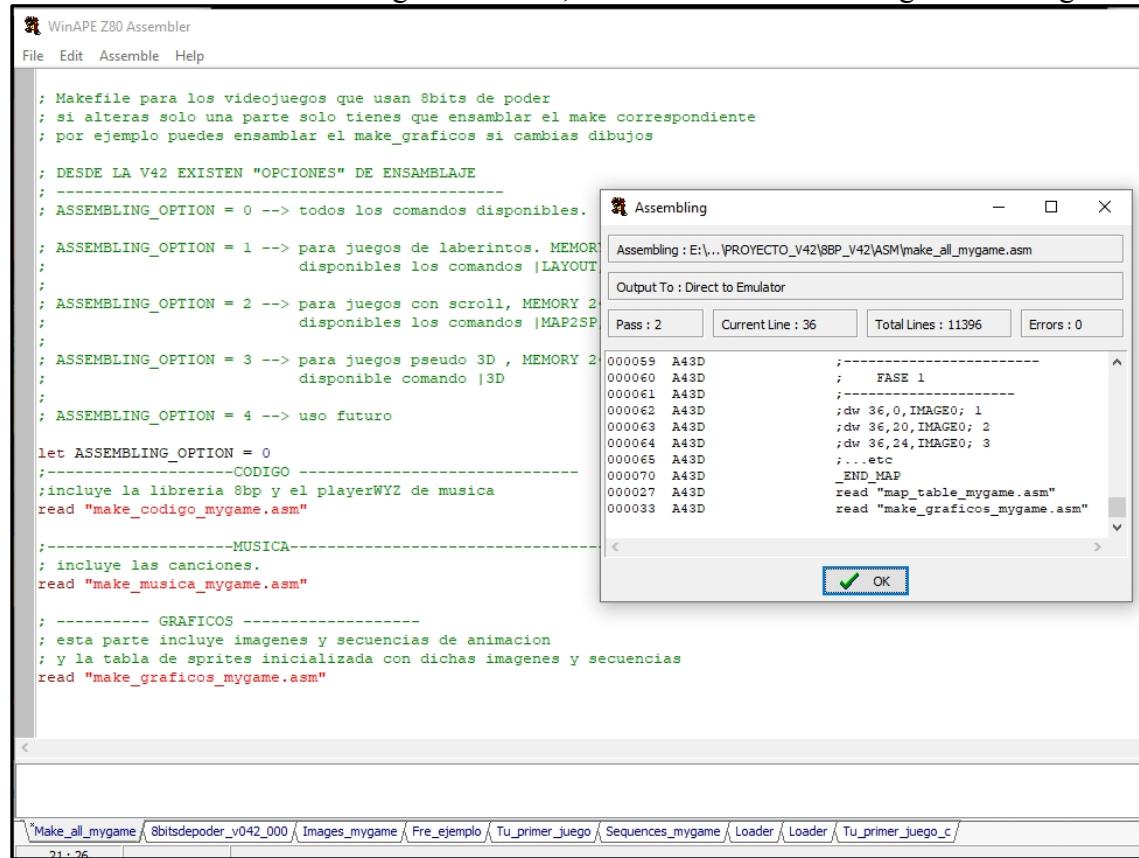
Wir gehen in das Stammverzeichnis. Dort finden Sie einen Ordner namens "**8BP_V42**". Ich empfehle Ihnen, eine Kopie dieses Ordners zu erstellen und ihn in "**my_game**" umzubenennen. Auf diese Weise bleibt der ursprüngliche "**8BP_V42**-Ordner erhalten, auch wenn Sie anfangen, Dinge zu ändern.



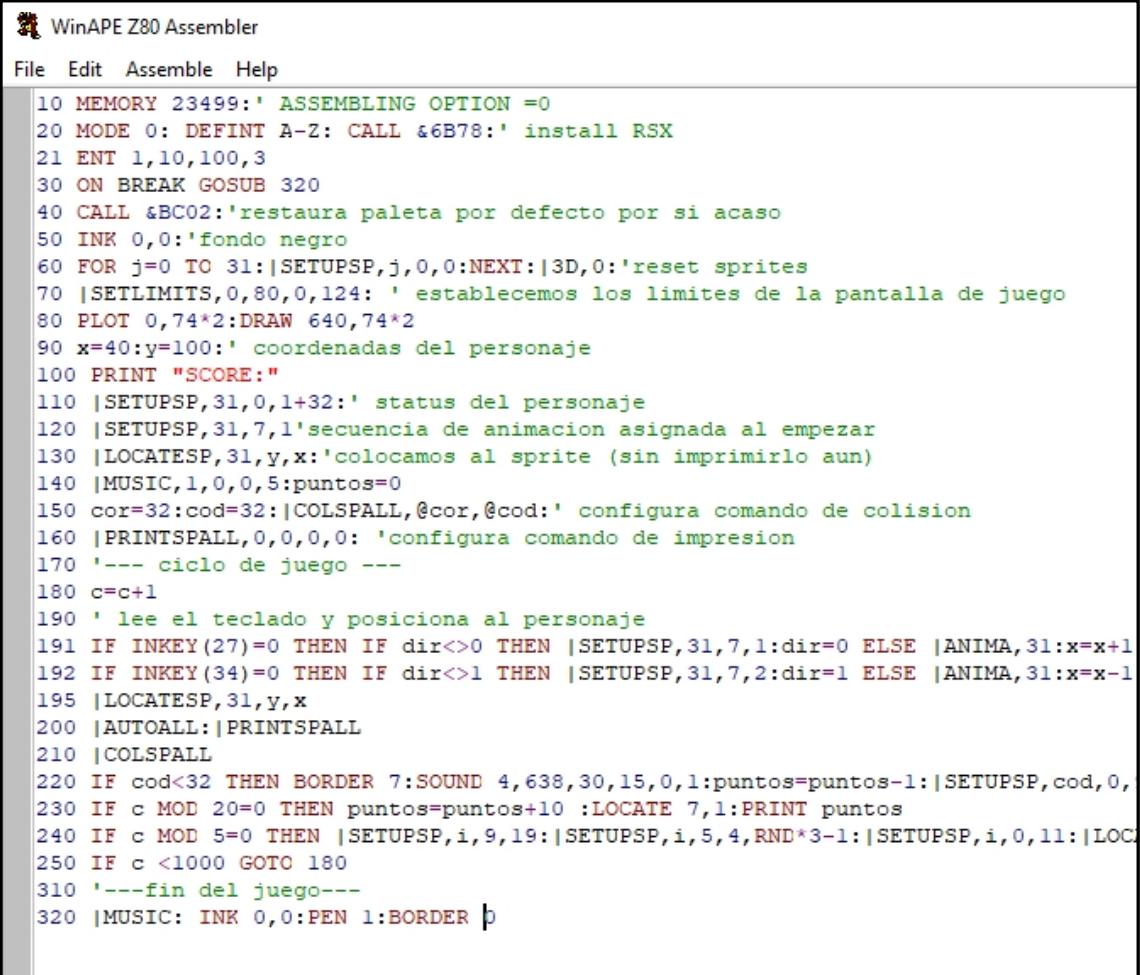
Im Ordner "**8BP_V42**" finden Sie die Ordner ASM, BASIC, DSK, MUSIC, TAPE und output_spedit.

Öffnen Sie im winape Assembler-Fenster die Datei "ASM/make_all_mygame.asm" und führen Sie sie aus (wählen Sie im winape z80 Assembler-Menü einfach "Assemble" oder drücken Sie Strg+F9). Dies wird damit beginnen, die Bibliothek und die Grafiken in den Amstrad-Speicher zu assemblieren (in den Speicher zu kopieren). In diesem Fall werden wir nur sehr wenige Grafiken assemblieren, nur die wichtigsten für ein kleines Spiel. Die Grafiken befinden sich in der Datei "**images_mygame.asm**".

Nachdem Sie alles zusammengebaut haben, erhalten Sie eine Meldung wie die folgende:



Drücken Sie "ok" und öffnen Sie im Assembler-Fenster eine weitere Datei. In diesem Fall öffnen wir eine BASIC-Datei, nämlich "**your_first_game.bas**", die sich im BASIC-Ordner befindet. Nach dem Öffnen sehen Sie die folgende Liste auf dem Bildschirm, die 32 Zeilen enthält:



```

WinAPE Z80 Assembler
File Edit Assemble Help
10 MEMORY 23499:' ASSEMBLING OPTION =0
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
21 ENT 1,10,100,3
30 ON BREAK GOSUB 320
40 CALL &BC02:'restaura paleta por defecto por si acaso
50 INK 0,0:'fondo negro
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'reset sprites
70 |SETLIMITS,0,80,0,124: ' establecemos los limites de la pantalla de juego
80 PLOT 0,74*2:DRAW 640,74*2
90 x=40:y=100:' coordenadas del personaje
100 PRINT "SCORE:"
110 |SETUPSP,31,0,1+32:' status del personaje
120 |SETUPSP,31,7,1'secuencia de animacion asignada al empezar
130 |LOCATESP,31,y,x:'colocamos al sprite (sin imprimirla aun)
140 |MUSIC,1,0,0,5:puntos=0
150 cor=32:cod=32:|COLSPALL,@cor,@cod:' configura comando de colision
160 |PRINTSPALL,0,0,0,0: 'configura comando de impresion
170 '--- ciclo de juego ---
180 c=c+1
190 ' lee el teclado y posiciona al personaje
191 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,31,7,1:dir=0 ELSE |ANIMA,31:x=x+1
192 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,31,7,2:dir=1 ELSE |ANIMA,31:x=x-1
195 |LOCATESP,31,y,x
200 |AUTOALL:|PRINTSPALL
210 |COLSPALL
220 IF cod<32 THEN BORDER 7:SOUND 4,638,30,15,0,1:puntos=puntos-1:|SETUPSP,cod,0,
230 IF c MOD 20=0 THEN puntos=puntos+10 :LOCATE 7,1:PRINT puntos
240 IF c MOD 5=0 THEN |SETUPSP,i,9,19:|SETUPSP,i,5,4,RND*3-1:|SETUPSP,i,0,11:|LOC
250 IF c <1000 GOTO 180
310 '---fin del juego---
320 |MUSIC: INK 0,0:PEN 1:BORDER |p

```

Markieren Sie alles und kopieren Sie es. Gehen Sie dann zum CPC-Emulationsfenster und fügen Sie es über das Menü **DATEI->Einfügen ein**.

Da die Liste etwas länger ist als "hello world", wähle im Winape-Menü **Einstellungen->Hohe Geschwindigkeit**, um sie zu kopieren, und gehe dann zurück auf "normale Geschwindigkeit". Da die Bibliothek und die Grafiken bereits zusammengebaut sind, kannst du RUN und das Spiel wird laufen. Du musst vom Himmel fallenden Bällen ausweichen, um nicht zu sterben, und dich dabei nach links und rechts bewegen.



Sie können versuchen, das Programm zu verändern und seine Auswirkungen zu sehen. Nach und nach lernst du 8BP kennen und kannst interessante Modifikationen vornehmen, z.B. die Häufigkeit, mit der die gegnerischen Kugeln herauskommen, oder ihre Geschwindigkeit ändern, oder den Soldaten durch ein Raumschiff und die Kugeln durch gegnerische Schiffe mit unterschiedlichen Flugbahnen ersetzen.

Wenn du testen willst, wie schnell es in C funktioniert, lade einfach die .dsk und führe "loader.bas" aus. Es wird eine Version des Spiels geladen, bei der du zwischen der BASIC-Version und der C-Version wählen kannst, damit du vergleichen kannst. Es gibt ein Kapitel in diesem Buch, das der C-Programmierung mit 8BP und einem Mini-BASIC gewidmet ist, so dass Sie in C genauso programmieren können wie in BASIC. Wenn Sie noch nicht viel Erfahrung in der C-Programmierung haben, empfehle ich Ihnen, sich langsam heranzutasten und in BASIC zu programmieren, die Ergebnisse werden schnell und professionell sein.

4.6 Erstellen Sie Ihr .dsk mit Ihrem 8BP-Set

Zum Schluss werden wir eine Diskette mit Ihrem Spiel erstellen. Dazu müssen Sie, nachdem Sie das Spiel gestartet haben, die folgenden Schritte ausführen

- Erstellen Sie eine neue Disc über winape: DATEI-> Laufwerk A-> neue Blanc Disc
- Formatieren Sie es: DATEI->Laufwerk A->Diskettenabbild formatieren
- Nachdem Sie Ihre dsk-Datei erstellt haben, führen Sie im Emulationsfenster die folgenden Befehle aus:

SAVE "8BPO.bin", b, 23499,19119

SAVE "juego.bas"

Der erste Befehl speichert die 8BP-Bibliothek, Grafiken und Musik auf der Festplatte. In diesem Fall verwenden wir die Assembler-Option 0 (siehe Optionen im vorigen Kapitel), aber Sie könnten jede Option für dieses Beispiel verwenden. Ich habe die Datei "**8BPO**" genannt, um zu verdeutlichen, dass die Assembly-Option 0 verwendet wurde, aber der Name dieser Binärdatei kann beliebig sein.

Wir sind fast fertig. Jetzt müssen Sie eine andere Festplatte aus dem Winape-Menü auswählen oder Winape beenden, damit die von Ihnen erstellte .dsk im Windows-Dateisystem Realität wird.

Beenden Sie winape und öffnen Sie es erneut.

Wählen Sie den von Ihnen erstellten Datenträger aus und führen Sie ihn aus:

```
SPEICHER 23499
LOAD "8BPO.BIN"
RUN "game.bas" RUN "game.bas" RUN "game.bas" RUN "game.bas" RUN
```

Halleluja!

5 Schritte bei der Erstellung eines Spiels

5.1 Verzeichnisstruktur Ihres Projekts

Bei der Programmierung Ihres Spiels empfiehlt es sich, die verschiedenen Dateien in 7 Ordner zu strukturieren, je nachdem, um welche Art von Datei es sich handelt. Es ist durchaus möglich, alles in ein und dasselbe Verzeichnis zu legen und ohne Ordner zu arbeiten, aber es ist "sauberer", es so zu machen, wie ich es im Folgenden darlegen

- ASM
- BASIC
- C
- dsk
- MUSIC
- output_spedit

werde.

Abb. 12 Verzeichnisstruktur

- **ASM:** hier werden Textdateien in Assembler (.asm) abgelegt, wie die 8BP-Bibliothek selbst, die mit dem SPEDIT-Sprite-Editor erzeugten Sprites und einige Hilfsdateien.
- **BASIC:** Hier werden Sie Ihr Spiel und Dienstprogramme wie SPEDIT und Loader ablegen.
- **C:** Dieser Ordner ist für "fortgeschrittene" Benutzer, die das gesamte Spiel oder zumindest den Spielzyklus in C programmieren wollen. Er ist nur notwendig, wenn Sie einen Teil Ihres Spiels in C programmieren wollen.
- **Dsk:** Hier legen Sie die .dsk-Datei ab, die auf einem Amstrad CPC laufen soll. Darin müssen Sie 5 Dateien ablegen, über die wir im folgenden Abschnitt sprechen werden
- **Musik:** Mit dem WYZtracker-Musiksequenzer können Sie Ihre Songs erstellen und sie im .wyz-Format in diesem Verzeichnis speichern. Sobald Sie sie "exportieren", wird eine .asm-Datei erzeugt, die Sie im ASM-Ordner speichern müssen, und eine Binärdatei, die Sie ebenfalls im ASM-Ordner speichern (Sie können sie auch in diesem Ordner belassen, solange Sie sie in der Datei make_musica.asm im ASM-Ordner richtig referenzieren).
- **Output_spedit:** in diesem Ordner können Sie die von SPEDIT erzeugte Textdatei speichern. SPEDIT sendet die Sprites im Assembler-Format an den Drucker und der Winape-Emulator kann die Ausgabe des Amstrad-Druckers in einer Datei sammeln. Hier werden wir sie ablegen
- **Tape:** Hier können Sie die .wav-Datei speichern, wenn Sie eine Kassette zum Laden in den Amstrad CPC464 erstellen wollen, oder die .cdt-Datei.

5.2 Ihr Spiel in nur 3 Dateien

Um Ihr Spiel zu erstellen, benötigen Sie eine Binärdatei und zwei BASIC-Dateien. Sie können mehrere unabhängige Binärdateien erstellen (eine mit der 8BP-Bibliothek, eine mit den Grafiken, eine mit der Musik...), aber da diese Speicherbereiche zusammenhängend sind, ist es besser, eine einzige Binärdatei zu erstellen.

Die Binärdatei enthält die Bibliothek, die Musik, die Grafiken, den Speicherbereich oder das Layout der Weltkarte und optional die Sternenbank. Die Idee ist, alle binären Komponenten zusammen in einer Datei zu speichern, etwa so (je nach Assembler-Option werden Sie den einen oder den anderen dieser Befehle verwenden). Der Name der Datei endet mit einer Zahl, die die Assembler-Option angibt, aber sie kann heißen, wie Sie wollen.

```
SAVE "yourgame0.bin",b,2350 19119
      0,
SAVE "yourgame1.bin",b,2500 17619
      0,
SAVE "yourgame2.bin",b,2480 17819
      0,
SAVE "yourgame3.bin",b,2400 18619
      0,
```

Die Länge ist die endgültige Adresse minus die Anfangsadresse. Die endgültige Adresse, einschließlich Musik und Grafiken, der Karte und der Sternenbank, ist 42619, so dass die Länge berechnet werden kann, indem die Anfangsadresse von 42620 abgezogen wird. In der Assembler-Option 1 haben wir zum Beispiel $42619 - 25000 = 17619$, also genau die Länge, die in diesem Befehl angegeben ist.

Die zweite Datei ist der Grundstock

```
SAVE "tujuego.bas"
```

Und die dritte Datei ist der Lader ("**loader.bas**"), der einfach sein wird:

```
10 SPEICHER 23499
15 LOAD "!pant.scr",&c000: 'nur wenn Ihr Spiel einen Ladebildschirm
hat
20 LOAD " yourgame0.bin"
50 RUN " !yourgame.bas"
```

Ich habe einen Startbildschirm an die Adresse des Startbildschirmspeichers (&C000) gesetzt. Am Ende dieses Handbuchs finden Sie eine der vielen Möglichkeiten, einen Bildschirm zu laden. Sie ist optional. Sie können ein Spiel mit oder ohne Ladebildschirm erstellen.

Diese 3-Dateien-Methode ist vor allem dann nützlich, wenn Sie fast den gesamten für die Grafik verfügbaren Speicher belegen. Bei Kassettenspielen kann das Laden von 18 KB eine Weile dauern, und wenn Sie die 8,5 KB für die Grafik nicht verwenden, ist es vielleicht besser, die verschiedenen Binärdateien getrennt zu laden, um Ladezeit zu sparen. Wenn Sie z.B. nur 2KB an Grafiken verwenden, würden Sie mit einer einzigen Binärdatei der Länge 18619 8,5KB an Grafiken laden, d.h. 6,5KB extra leer. Das kann bei der Bandladezeit fast zwei Minuten zusätzliche Zeit bedeuten. Auf der Festplatte (CPC 6128) spielt das keine Rolle, weil das Laden der Grafiken überhaupt keine Zeit in Anspruch nimmt. In diesem Fall ist es besser, ein oder zwei Binärdateien zu erstellen, die nur den von Ihnen benötigten Speicherplatz speichern.

Um diese 3 Dateien zu erstellen, müssen Sie die folgenden Schritte ausführen:

SCHRITT 1

Bearbeiten Sie Grafiken mit SPEDIT und kopieren Sie das Ergebnis (SPEDIT sendet es in eine .txt-Datei) nach
images_mygame.asm

SCHRITT 2

Musik bearbeiten mit WYZtracker

Ändern Sie music_mygame.asm, um die erstellten Musikstücke einzubinden

Die Melodien werden nacheinander zusammengesetzt, so dass jede Melodie an einer anderen Speicheradresse beginnt, abhängig von der Größe der Melodie.

SCHRITT 3

Bauen Sie die 8BP-Bibliothek neu zusammen, damit der Teil der Bibliothek, der die Melodien auswählt (der Player wyz), weiß, an welchen Speicheradressen sie zusammengebaut wurden (es gibt noch mehr Abhängigkeiten, aber das ist eine davon). Nach dem Zusammenbau müssen Sie alles mit einem dieser Befehle speichern, je nachdem, welche Zusammenbauoption Sie verwenden (der Name kann sein, was immer Sie wollen, ich habe eine Zahl am Ende hinzugefügt, um irgendwie die Zusammenbauoption anzugeben):

```
SAVE "yourgame0.bin",b,2350 19119
      0,
SAVE "yourgame1.bin",b,2500 17619
      0,
SAVE "yourgame2.bin",b,2480 17819
      0,
SAVE "yourgame3.bin",b,2400 18619
      0,
```

Dabei handelt es sich um eine für Ihr Spiel spezifische Version der Bibliothek. Zum Beispiel, der Befehl

|MUSIC,0,0,0,3,6 spielt die Melodie Nummer 3, die Sie selbst komponiert haben. Die Melodie Nummer 3 kann in einem anderen Spiel völlig anders sein.

SCHRITT 4

Programmieren Sie Ihr Spiel, das zuerst den Aufruf zur Installation der RSX-Befehle ausführen muss, d.h. CALL &6b78. Und etwas sehr Wichtiges: Vergessen Sie nicht, den **MEMORY-Befehl** am Anfang einzufügen, um zu vermeiden, dass das laufende BASIC Variablen oberhalb der Adresse speichert, an der 8BP beginnt.

MEMORY 23499 :rem verwende diesen SPEICHER für Option 0
MEMORY 24999 :rem verwende diesen SPEICHER für Option 1
MEMORY 24799 :rem verwende diesen SPEICHER für Option 2
MEMORY 23999 :rem verwende diesen SPEICHER für Option 3
MEMORY 23999 :rem verwende diesen SPEICHER für Option 3

Ihr Spiel kann mit dem winape-Editor programmiert werden, der vielseitiger ist als der AMSTRAD-Editor und sowohl für die Bearbeitung von Assembler (.asm) als auch BASIC (.bas) verwendet werden kann. Der winape-Editor reagiert auf Schlüsselwörter und ändert deren Farbe automatisch, was die Programmierung erleichtert. Nachdem Sie ein BASIC-Programm geschrieben haben, müssen Sie es in das CPC-Fenster von winape kopieren/einfügen. Um dies zu beschleunigen, können Sie die "High Speed"-Option von winape während des Einfügens aktivieren, so dass der Einfügevorgang sofort erfolgt.

SCHRITT 5

Laden Sie alles mit einer loader.bas , was Sie in BASIC tun müssen.

SCHRITT 6

Erstellen Sie eine Kassette oder Disc mit Ihrem Spiel

5.3 Erstellen Sie eine Disc oder Kassette mit Ihrem Spiel

5.3.1 Herstellung einer Disc

Um einen neuen Datenträger aus winape zu erstellen, gehen wir wie folgt vor

Datei->Laufwerk A-> neuer leerer Datenträger

Daraufhin wird ein Dateiverwaltungsfenster angezeigt, in dem Sie die neue .dsk-Datei benennen können.

Einmal erstellte Dateien können Sie mit dem Befehl SAVE speichern. Um eine Datei zu löschen, verwenden Sie den Befehl "|ERA" (kurz für ERASE), den es nur auf dem CPC 6128 als Teil des Betriebssystems "AMSDOS" gibt (auf dem CPC464 gab es diesen Befehl nicht, da er mit Kassettenband arbeitete).

|ERA, "game.*"

Und sie werden ausgelöscht

Um das Spiel zu laden, brauchen Sie einen Lader, der die notwendigen Dateien nacheinander lädt. Etwas wie (der MEMORY-Befehl hängt von der Assembly-Option ab):

```
10 MEMORY 23499: 'memory command depends on assembly option
15 LOAD "!pant.scr",&c000: 'nur wenn Ihr Spiel einen Ladebildschirm hat
20 LOAD "yourgame0.bin"
50 RUN " !yourgame.bas"
```

Um jede der Dateien zu speichern, müssen Sie den Befehl SAVE mit den erforderlichen Parametern verwenden, z. B:

```
"LOADER.BAS" SPEICHERN
SAVE "yourgame0.bin",b,23500,
19119 SAVE
"yourgame0.bin",b,23500, 19119
SAVE "yourgame.BAS"
```

Wenn Sie die .dsk auf eine 3,5"-Diskette schreiben und diese an ein externes Diskettenlaufwerk Ihres AMSTRAD CPC 6128 anschließen wollen, benötigen Sie das einfach zu bedienende Programm CPCDiskXP. Aus einer .dsk können Sie eine 3,5"-Diskette in doppelter Dichte brennen (vergessen Sie nicht, das Loch der Diskette abzudecken, um den PC zu "überlisten").

5.3.2 Erstellen eines Bandes mit Winape

Das Wichtigste bei der Erstellung eines Bandes ist, die Dateien in der Reihenfolge zu speichern, in der sie vom Computer geladen werden sollen. Ein Band ist nicht wie eine Diskette, auf die Sie jede gespeicherte Datei laden können, sondern die Dateien sind hintereinander angeordnet, so dass Sie in diesem Punkt besondere Sorgfalt walten lassen müssen.

Wenn Ihr Game Loader so aussieht:

```
10 SPEICHER 23499
15 LOAD "screen.scr",&c000 : rem wenn Ihr Spiel einen
Ladebildschirm hat
20 LOAD " !yourgame0.bin"
50 RUN " !yourgame.bas"
```

Der Ausruf "!" ist sehr wichtig, damit das Amstrad beim Ausführen jedes LOADs nicht die Meldung "press play then any key" erhält.

Zuerst müssen Sie den Lader speichern (sagen wir, er heißt "**loader.bas**"), dann die Datei "**tujuego.bin**" und schließlich "**tujuego.BAS**".

So erstellen Sie eine ".wav" oder aus winape

file->tape->press record

Daraufhin wird ein Menü zur Dateiverwaltung angezeigt, in dem Sie entscheiden können, wie Sie die Datei ".wav" nennen möchten.

Wenn Sie sich im CPC 6128-Modus befinden, dann müssen Sie als nächstes von BASIC aus

|TAPE

Und dann

SCHNELLES SCHREIBEN 1

Mit diesem Befehl haben wir den AMSTRAD angewiesen, mit 2000 Baud aufzuzeichnen. Dadurch wird die Ladezeit verkürzt. Wenn Sie diesen Befehl nicht ausführen, erfolgt die Aufzeichnung mit 1000 Baud, was sicherer, aber viel langsamer ist.

"LOADER.BAS" SPEICHERN

Sie erhalten eine Meldung, in der Sie aufgefordert werden, rec&play zu drücken, und dann "ENTER". Speichern Sie dann jede einzelne Datei:

```
SAVE "yourgame0.bin",b,23500,  
19119 SAVE  
"yourgame0.bin",b,23500, 19119  
SAVE "yourgame.BAS"
```

Schließlich müssen wir noch eine letzte Operation durchführen, damit winape die Datei schließt.

file->remove tape

Nachdem Sie das "Band entfernen" durchgeführt haben, wird die Datei ihre Größe annehmen (wenn Sie es nicht tun, können Sie sehen, dass die Datei auf der Festplatte Ihres PCs nicht wächst, weil sie nicht auf die Festplatte übertragen wurde).

Um das Spiel zu laden, wenn Sie einen CPC6128 benutzen

|TAPE
LAUF ""

So verwenden Sie die Disc wieder

|DISC

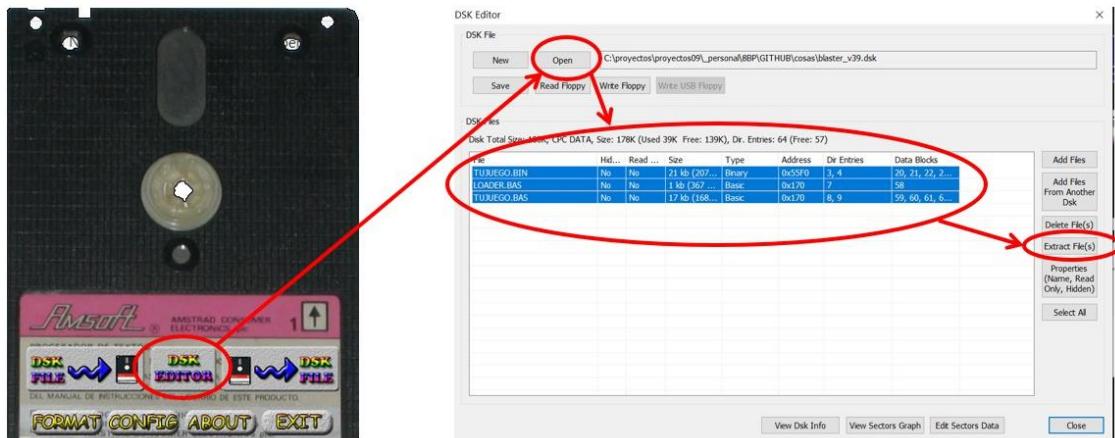
Wenn Sie einen Ladebildschirm speichern wollen, lesen Sie bitte Anhang I über die Organisation des Videospeichers, wo ich erkläre, wie man das macht.

5.3.3 Einfaches Erstellen eines Bandes mit CPCDiskXP, 2cdt und tape2wav

Winape ist ein großartiges Werkzeug für die Programmierung und Emulation. Allerdings hat es eine kleine Einschränkung: Es erlaubt nicht, ein Band im cdt-Format zu erstellen, nur im wav-Format.
Es gibt eine sehr schnelle und zuverlässige Methode, mit der Sie .cdt- und wav-Dateien für die benötigen Sie die Tools CPCDiskXP, 2cdt und tape2wav.

Gehen wir von der Annahme aus, dass Sie eine .dsk-Datei mit Ihren Dateien erstellt haben. Dann müssen Sie die folgenden Schritte ausführen:

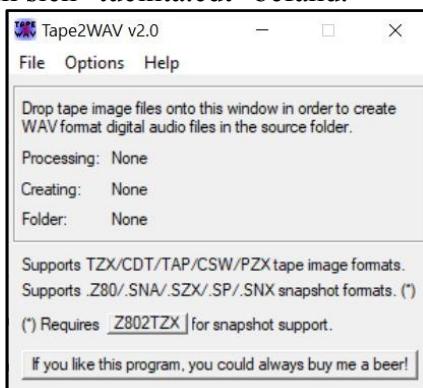
- 1) Zunächst können Sie mit dem CPCDiskXP-Tool die .dsk öffnen und die notwendigen Dateien für Ihr Spiel extrahieren (die "loader.bas", die "tujuego.bin" und "tujuego.bas"). Dazu klickst du einfach auf "Disk Edition", dann auf "Öffnen", öffnest deine .dsk, wählst die Dateien aus und klickst schließlich auf "Dateien extrahieren". Sobald Sie das getan haben, befinden sich die Dateien im Windows-Dateisystem.



- 2) Anschließend brennen Sie die Dateien mit dem Tool 2cdt nacheinander in eine .cdt-Datei. Die Befehle lauten:

```
2cdt.exe -n -s 1 -r "LOADER.bas" "loader.bas" tucinta.cdt
2cdt.exe -b 2000 -r "tujuego0.bin" "tujuego0.bas" tucinta.cdt
2cdt.exe -b 2000 -r "tujuego.bas" "tujuego.bas" tucinta.cdt
```

- 3) Jetzt haben Sie die Datei tucinta.cdt erstellt. Wenn Sie auch eine .wav-Datei haben wollen, um sie auf einen echten CPC 464 laden zu können, können Sie das tape2wav-Tool benutzen. Sie starten es einfach und ziehen die .cdt-Datei mit der Maus in das Tool. Das tape2wav erzeugt sofort eine Datei "tucinta.wav" in dem Verzeichnis, in dem sich "tucinta.cdt" befand.



5.3.4 Fehlersuche bei LOAD und MEMORY

Bevor eine BASIC-Datei geladen wird, vergewissert sich der Amstrad, dass genügend Speicherplatz vorhanden ist, um sie auszuführen. Das BASIC-Programm braucht vielleicht nur 1KB an Variablen, aber das weiß der Amstrad nicht, also ist er konservativer und verlangt, dass Sie zusätzlich 5KB an leerem Speicher haben. Diese 5 KB mögen übertrieben erscheinen, aber der Amstrad weiß nicht von vornherein, wie viele Variablen Sie in Ihrem Programm deklarieren werden, und es ist ihm lieber, viel freien Speicherplatz für Variablen zu haben, als zu wenig und das Programm scheitern zu lassen.

Das bedeutet, dass Sie, wenn Sie zuvor eine oder mehrere Binärdateien (mit Grafiken, 8BP-Bibliothek oder was auch immer) geladen haben und noch 20 KB frei sind, nicht in der Lage sein werden, ein 20 KB großes BASIC-Spiel zu laden, sondern höchstens ein 15 KB großes Spiel. Aber keine Sorge, es gibt mehrere Möglichkeiten, dieses Problem zu lösen. Ich werde die einfachste erklären

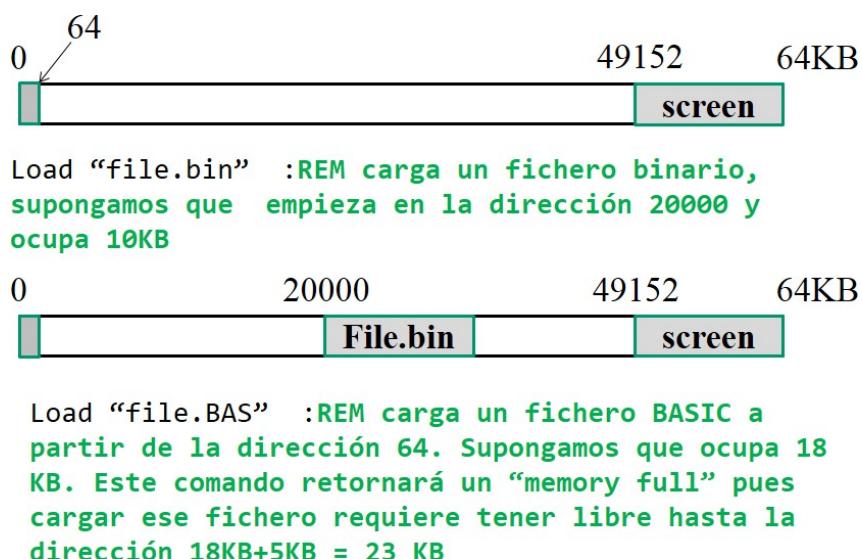


Abb. 13 Das LOAD-Problem

Die Lösung besteht darin, eine Datei "loader.bas" zu erstellen, die den SPEICHER verändert. Nehmen wir an, Sie haben Binärdaten ab Adresse 20.000 und Ihr Programm benötigt 18 KB, so dass weniger als 5 KB Platz bleiben. Alles, was Sie tun müssen, ist:

```

10 SPEICHER 19999
20 LOAD "!game.bin": rem lädt Daten von 20000
30 CLEAR: MEMORY 23000 : rem damit wir dir mehr freien RAM zur Verfügung stellen.
40 RUN "!game.bas": rem die erste Zeile von game.bas muss Speicher 19999 sein

```

Diese Methode ist sehr einfach und 100% zuverlässig, denn obwohl Sie Teile der Binärdaten während des BASIC-Ladens ungeschützt lassen, ist das erste, was Sie in BASIC tun, die Ausführung des SPEICHERS, so dass dieser wieder geschützt ist, bevor Sie irgendwelche Variablen erstellt haben.

Ein weiteres typisches Problem im Zusammenhang mit dem Fehler MEMORY FULL tritt auf, wenn wir ein Programm laden und es mitten in der Ausführung stoppen (durch zweimaliges Drücken von ESC). Es ist möglich, dass wir **MEMORY FULL** erhalten, wenn wir versuchen, mit einem CAT-Befehl auf die Festplatte zuzugreifen.

```
Break in 420
Ready
CAT
Memory full
Ready
```

Das liegt daran, dass unser BASIC-Programm eine Menge RAM-Speicher für Variablen verbrauchen kann. Dass wir es angehalten haben, bedeutet nicht, dass die Variablen aus dem Programm verschwunden sind, sie sind immer noch vorhanden und man kann sie sogar ausdrucken, um ihren Wert zu sehen. Was wir in diesem Fall tun werden, ist einfach den **CLEAR-Befehl** auszuführen, der den Speicher dieser Variablen freigibt, und dann unseren CAT-Befehl (oder den, den wir wollen).

```
Break in 420
Ready
CAT
Memory full
Ready
CLEAR
Ready
CAT

Drive A: user 0
LOADER :BAK 1K SP .BAS 18K
LOADER :BAS 1K SP :BIN 19K
SP :BAK 18K SP :SCR 17K

104K free
Ready
```

Wenn Sie ein BASIC-Programm so groß gemacht haben, dass Sie keine 5 KB zwischen dem BASIC-Programm und dem assemblierten Binärprogramm frei haben, dann erhalten Sie logischerweise auch den Fehler **SPEICHER VOLL**, wenn Sie Ihr Spiel auf der Festplatte speichern wollen.

Wenn Sie versuchen, die Binärdatei zu speichern, erhalten Sie die Fehlermeldung **MEMORY FULL**, aber das lässt sich ganz einfach beheben. Laden Sie einfach nicht das BASIC-Listing in Ihrem Emulator und führen Sie keine MEMORY-Befehle aus. Wenn Sie mit winape die Datei "make_all.asm" assemblieren, haben Sie alle Grafiken, Musik und die 8BP-Bibliothek im Speicher des Amstrad. Führen Sie dann Ihren SAVE-Befehl aus und es wird funktionieren. Der Befehl, um alles in einer einzigen Binärdatei zu speichern, hängt von der Assembler-Option ab, die Sie in der Datei "**make_all_mygame.asm**" angegeben haben, **und wird einer der folgenden sein:**

```
SAVE "yourgame0.bin",b,2350 19119
      0,
SAVE "yourgame1.bin",b,2500 17619
      0,
SAVE "yourgame2.bin",b,2480 17819
      0,
SAVE "yourgame3.bin",b,2400 18619
      0,
```

Wenn Sie jedoch binäre Dinge (zusätzliche Grafiken, Karten usw.) unterhalb der Anfangsadresse 8BP gespeichert haben, müssen Sie dies berücksichtigen. Wenn Ihr

Spiel zum Beispiel an der Adresse 20000 beginnt, lautet der Befehl (ich habe die Länge erhöht, so dass er von 20000 bis 42619 reicht)

```
SAVE "yourgame.bin",b,20000, 22619
```

Sie können nun Ihr BASIC-Programm kopieren und in den Emulator einfügen. Führen Sie nach dem Kopieren keinen **MEMORY-Befehl** aus und speichern Sie Ihre .BAS-Datei auf der Festplatte.

Da Sie zu diesem Zeitpunkt noch keinen **MEMORY-Befehl** ausgeführt haben, "denkt" das Amstrad, dass Sie mehr als 5KB oberhalb Ihres BASIC-Programms frei haben und wird uns nicht die Meldung **MEMORY FULL anzeigen**. Sobald Sie Ihren **MEMORY-Befehl** ausgeführt haben (wenn z.B. Ihre Binärdaten mit 20000 beginnen, wird es ein **MEMORY 19999** statt 23999), prüft das Amstrad, ob zwischen Ihrem BASIC-Programm und der **MEMORY-Adresse** noch 5KB frei sind, und wenn das nicht der Fall ist, gibt es beim Ausführen des **SAVE-Befehls** eine Fehlermeldung aus, auch wenn das Spiel funktioniert. Wenn Ihr Spiel während der Ausführung versucht, mehr Speicherplatz zu verbrauchen, als es bis zur **MEMORY-Adresse** frei hat, hält es an und gibt die Fehlermeldung **MEMORY FULL aus**.

6 Bibliothek, Musik und Grafikmontage

Dieses Kapitel beschreibt etwas genauer, was passiert, wenn du die "make_all"-Datei ausführst, und ermöglicht es dir, den ganzen Prozess besser zu verstehen. **Wenn du aber erst einmal lernen willst, wie man mit 8BP programmiert, kannst du es überspringen und später hierher zurückkommen**, wenn du besser verstehen willst, wo die Grafiken und die Musik hingehören und wie man die Bibliothek mit ihnen zusammenstellt.

Der Grund dafür ist, dass beispielsweise der Musikplayer in die Bibliothek eingebettet ist und wissen muss, wo jedes Lied beginnt (Speicheradresse). Daher ist es notwendig, die für Ihr Spiel spezifische Version der Bibliothek sowie die zusammengestellte Grafikdatei und die zusammengestellte Musikdatei neu zusammenzustellen und zu speichern.

Wie ich im Abschnitt "Schritte" erklärt habe, handelt es sich dabei um eine für Ihr Spiel spezifische Version der Bibliothek. Zum Beispiel spielt der Befehl **|MUSIC,0,0,3,6 die** Melodie Nummer 3, die Sie selbst komponiert haben. Die Melodie Nummer 3 kann in einem anderen Spiel ganz anders klingen. Das Gleiche gilt für die Daten in der Instrumentendatei. Es gibt bestimmte Abhängigkeiten zwischen dem Code des Musikplayers und den Adressen, an denen die Instrumentendaten und Melodien zusammengestellt werden.

Es ist sehr einfach, aber Sie müssen die Struktur der Bibliothek verstehen, d.h. die Struktur der .asm-Dateien, die Sie bearbeiten müssen, und ihre Abhängigkeiten.

Das folgende Diagramm zeigt alle .asm-Dateien eines Spiels, das 8BP verwendet, sowie die Abhängigkeiten zwischen ihnen. In der Abbildung **sind die Dateien, die Sie bearbeiten müssen, grau dargestellt**, z. B.:

- die Songs und Instrumentendatei, die Sie mit dem WYZtracker erzeugen
- die Datei **make_musica**, in der Sie angeben, welche ".mus"-Dateien zusammengefügt werden sollen
- die Bilddatei, die Sie mit SPEDIT erstellen
- die Sprite-Tabelle, in der Sie den Sprites Bilder zuweisen (obwohl dies nicht unbedingt notwendig ist, da Sie den Befehl **|SETUPSP** haben).
- die Sequenztabelle, in der Sie festlegen, welche Bilder zu einer Sequenz gehören,
- die Weltkarte: Hier können Sie bis zu 64 Elemente definieren, aus denen die Welt besteht.
- die Pfade-Datei: Hier legen Sie die Pfade der Sprites fest, die Sie verwenden möchten.
- die Datei alphabet.asm: wenn Sie ein anderes Alphabet als das 8BP-Standardalphabet erstellen wollen

Sie können alles zusammensetzen, indem Sie die Datei "**make_all.asm**" öffnen und im Winape-Menü auf "assemble" klicken. Dann können Sie den Befehl SAVE verwenden, um die Bilder, die Musik und die 8BP-Bibliothek in verschiedenen Binärdateien oder in einer einzigen Datei zu speichern, wie wir gesehen haben.

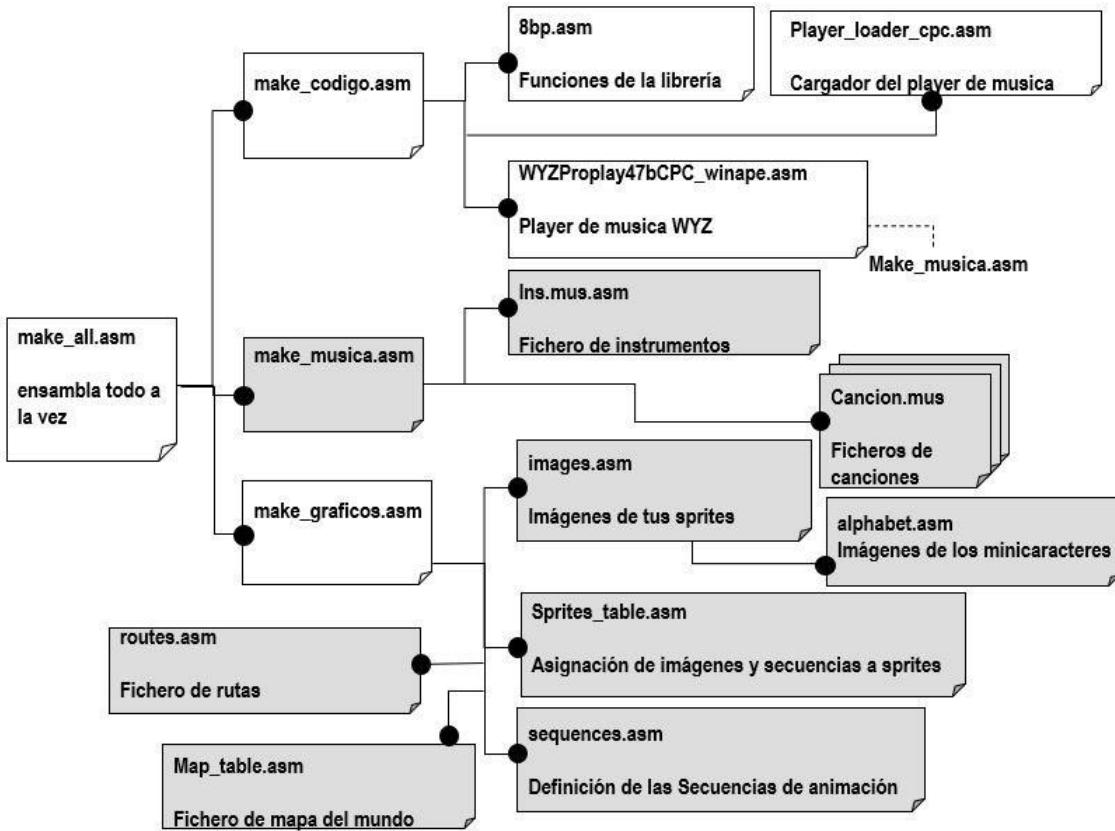


Abb. 14 Montagedateien

Wenn Sie nur die Grafiken ändern, können Sie diese separat zusammensetzen, indem Sie die Datei "make_graphics.asm" auswählen und auf assemble drücken.

Wenn Sie die Musik ändern, müssen Sie den Bibliothekscode neu zusammenstellen, da eine Abhängigkeit zwischen dem Code und den Liedern besteht, da der Code wissen muss, wo jedes Lied beginnt. Wenn du also Lieder änderst oder hinzufügst, musst du sie mit `make_all.asm` zusammenbauen. Ich habe diese Abhängigkeit mit einer gepunkteten Linie zwischen dem Player und der Datei `make_musica.asm` dargestellt.

Möglicherweise müssen Sie etwas anderes zusammenstellen, z. B. eine Rennstreckenkarte, die Ihre Bilder verwendet. In diesem Fall fügen Sie es der Datei "`Make_graphics.asm`" hinzu, so dass es nach der Datei "`images.asm`" assembliert wird. Die Reihenfolge des Zusammenbaus ist wichtig. Zuerst müssen Sie die Bilder zusammenstellen, ihnen Beschriftungen zuordnen und dann können Sie die Karten oder Strecken zusammenstellen, die diese Beschriftungen verwenden.

6.1 `Make_all.asm`

Dies ist die Datei, mit der alles zusammengefügt wird. Intern ruft sie drei Dateien auf, die den Code der Bibliothek und des Musikplayers, die Lieder und die Grafiken zusammenstellen.

Makefile für Videospiele mit 8bit-Power Wenn Sie einen Teil davon ändern, müssen Sie nur die entsprechende Marke zusammenstellen Sie können zum Beispiel <code>make_graphics</code> zusammenstellen, wenn Sie Zeichnungen ändern SEIT V42 GIBT ES "MONTAGEOPTIONEN".⁵⁴
--

```

; ZUSAMMENBAU_OPTI = 0 --> alle verfügbaren Befehle.
ON

; ZUSAMMENBAU_OPTI = 1 --> für Labyrinthspiele. SPEICHER 25000
ON
; verfügbar die Befehle |LAYOUT, |COLAY
;

; ZUSAMMENBAU_OPTI = --> für scrollende Spiele, SPEICHER 24800
ON
; verfügbar die Befehle |MAP2SP, |UMA
;

; ZUSAMMENBAU_OPTI = --> für Pseudo-3D-Spiele, SPEICHER 24000
ON
; verfügbarer Befehl |3D
;

; ZUSAMMENBAU_OPTI = --> zukünftige Verwendung
ON

let ASSEMBLING_OPTION = 0
-----CODO-----
;enthält die 8bp-Bibliothek und den Musik-
PlayerWYZ lesen Sie "make_codigo_mygame.asm".

-----MUSICA-----
read "make_musica_mygame.asm";
enthält die Lieder.

----- GRAFIKEN -----
Dieser Teil enthält Bilder und Animationssequenzen.
; und die Sprite-Tabelle wird mit diesen Bildern und Sequenzen aus
"make_graficos_mygame.asm" initialisiert.

```

Jede dieser drei Dateien ist für die Zusammenstellung verschiedener Dinge zuständig. So ruft beispielsweise die Grafikdatei andere Dateien auf, wie die Bilddatei, die Sequenzdatei, die Routendatei und die Weltkarte.

Verwenden Sie immer die Montageoption, die Ihnen den meisten freien Speicherplatz bietet. Wenn es sich bei Ihrem Spiel um ein Labyrinth handelt, verwenden Sie Option 1, wenn es sich um ein Scrolling-Spiel handelt, Option 2 und wenn es sich um ein Pseudo3D-Spiel handelt, Option 3. Im Allgemeinen empfehle ich Ihnen nicht, die Option 0 zu verwenden, da sie den geringsten freien Speicherplatz bietet und Sie wahrscheinlich in der Lage sein werden, eine der anderen Optionen zu verwenden.

6.2 Aufbau der Bilddatei

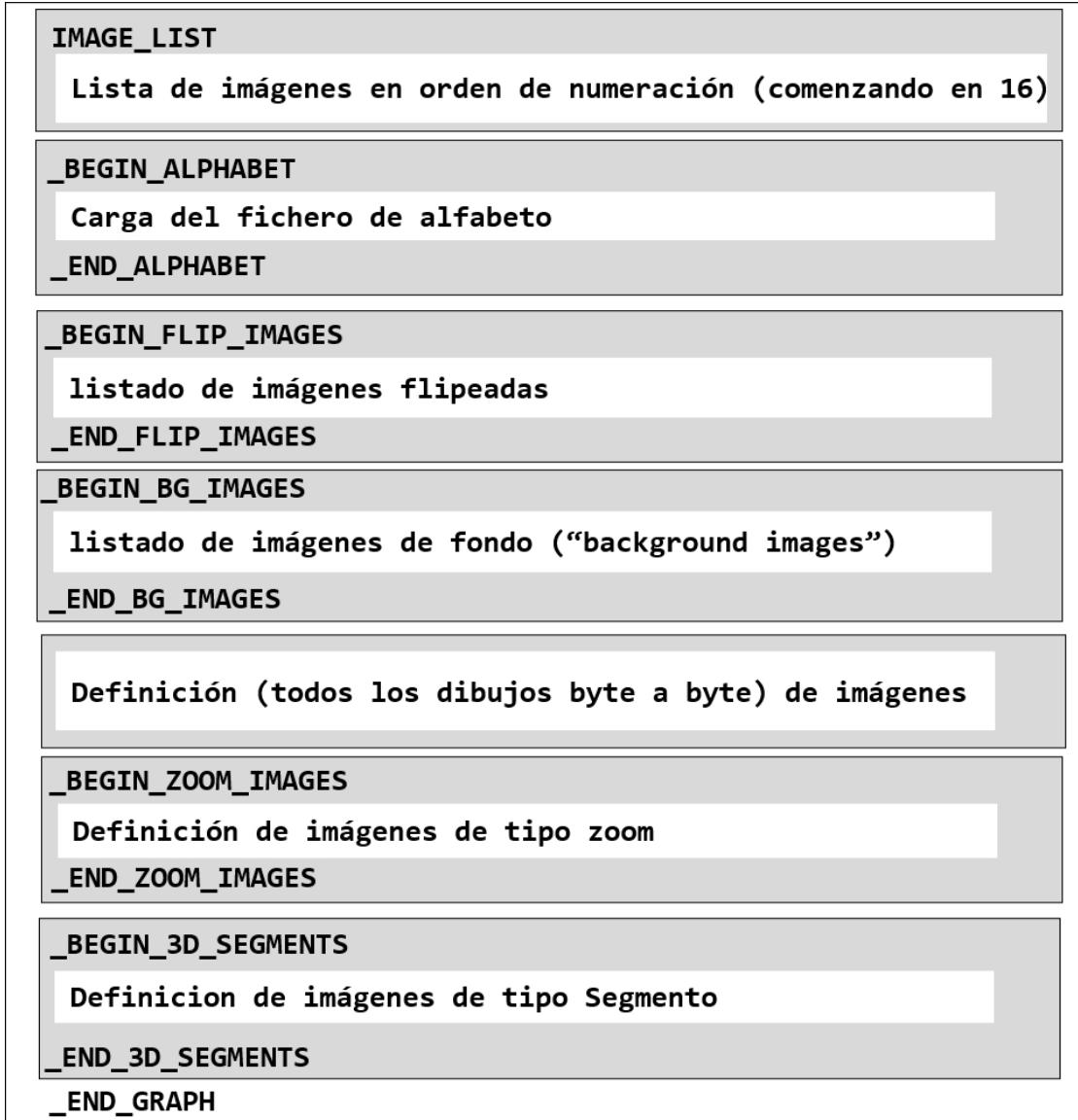


Abb. 15 Aufbau der Bilddatei

6.3 Struktur der Animationssequenzdatei

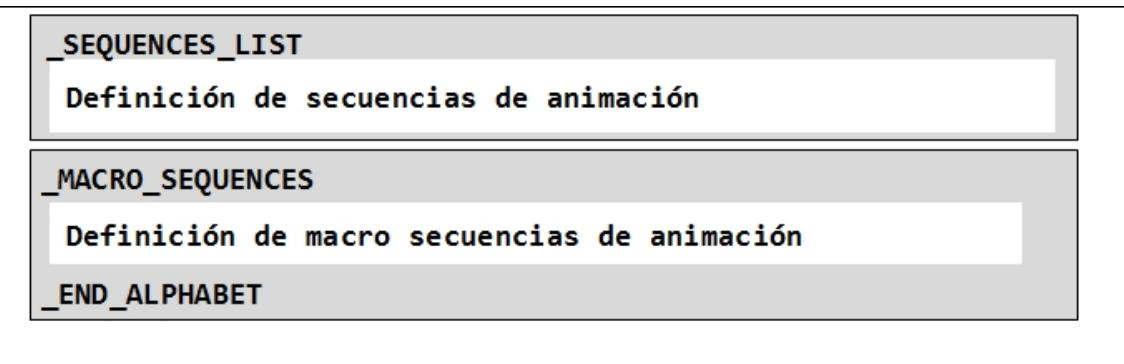


Abb. 16 Struktur der Animationssequenzdatei

6.4 Aufbau der Routingdatei

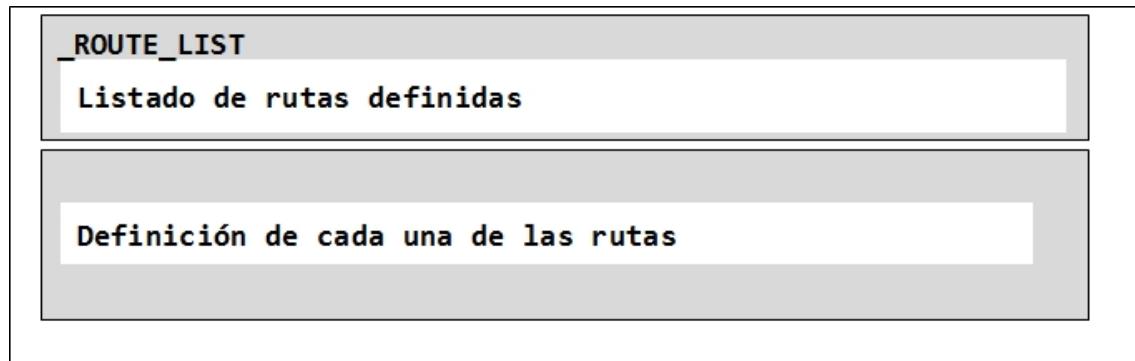


Abb. 17 Aufbau der Routingdatei

6.5 Aufbau der Weltkartendatei

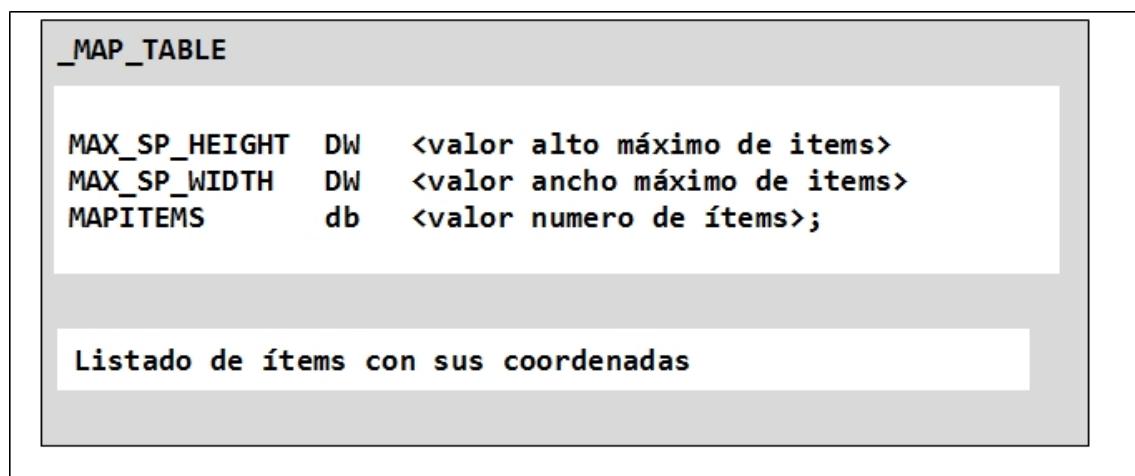


Abb. 18 Struktur der Weltkartendatei

7 Zyklus des Spiels

Ein Arcade-, Plattform- oder Abenteuervideospiel hat im Allgemeinen eine ähnliche Struktur, bei der bestimmte Vorgänge zyklisch wiederholt werden, was wir als "Spielzyklus" bezeichnen.

In jedem Spielzyklus werden die Sprite-Positionen aktualisiert und die Sprites auf dem Bildschirm ausgegeben, so dass die Anzahl der Spielzyklen, die pro Sekunde ausgeführt werden, den "Bildern pro Sekunde" (FPS) des Spiels entspricht. Der folgende Pseudocode skizziert die grundlegende Struktur eines Spiels

INICIO

```
Inicialización de variables globales: vidas, etc  
Espera a que el usuario pulse tecla de comienzo de juego  
GOSUB pantalla1  
GOSUB pantalla 2  
...  
GOSUB pantalla N  
GOTO INICIO
```

Código de PANTALLA N (siendo N cualquier pantalla)

```
Inicialización de coordenadas de enemigos y personaje  
Pintado de la pantalla (layout), si procede
```

BUCLE PRINCIPAL (ciclo del juego en esta pantalla)

```
Lógica de personaje : Lectura de teclado y actualización de  
coordenadas del personaje y si procede, actualización de su  
secuencia de animación
```

```
Ejecución de lógica de enemigo 1  
Ejecución de lógica de enemigo 2  
...  
Ejecución de lógica de enemigo n
```

```
Impresión de todos los sprites  
Condición de salida de esta pantalla (IF ... THEN RETURN)  
GOTO BUCLE PRINCIPAL
```

Abb. 19 Grundstruktur eines Spiels

Wenn die Feindlogik aufgrund zu vieler Feinde oder zu komplex ist, verbraucht dies mehr Zeit pro Spielzyklus und die Anzahl der Zyklen pro Sekunde wird daher reduziert. Versuchen Sie, nicht unter 10fps zu gehen, damit das Spiel ein akzeptables Niveau an Action beibehält.

7.1 So messen Sie die FPS Ihres Spielzyklus

Um herauszufinden, ob Ihr Spiel ein akzeptables Maß an Action hat, gibt es nichts Besseres, als es zu spielen, und wenn es Ihnen gefällt, dann ist es in Ordnung. Sie sollten jedoch genau messen, wie viele Bilder pro Sekunde Ihr Spiel erzeugen kann, denn dann können Sie Entscheidungen in der Logik Ihres Programms treffen und messen, wie sehr diese Programmierentscheidungen dem Spiel schaden oder nützen.

Zur Messung wird einfach der Zeitpunkt vor dem Start des ersten Spielzyklus, zu Beginn des "N-Bildschirm-Codes", notiert. Dann notieren wir die Zeit nach ein paar Spielzyklen und führen eine einfache Division durch. Schauen wir uns das Schritt für Schritt an:

A=TIME : rem diese Zeile speichert in der Variablen A die Zeit in 1/300 Sekundenbruchteilen.

Die Zahl, die in A gespeichert werden soll, kann sehr groß sein, sogar größer als das, was eine Integer-Variable wie "A" speichern kann. Damit die Zuweisung nicht zu einem Fehler führt, ist es zweckmäßig, den Zeitgeber des AMSTRAD zurückzusetzen, der bei jedem Start der Maschine gestartet wird. Um ihn zurückzusetzen, bevor Sie die Variable "A" zuweisen, führen Sie einfach aus:

Auf einem CPC 6128

POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0

Mit einem CPC 464

POKE &b187,0: POKE &b188,0: POKE &b189,0: POKE &b18a,0

Um zu unterscheiden, auf welchem Gerät Ihr Programm läuft, müssen Sie die Musik deaktivieren und eine Adresse mit PEEK abfragen.

**| MUSIC: Wenn peek(&39)=57 dann Model=464 sonst
model=6128 Wenn model=464 dann ...**

Damit haben Sie die Speicheradressen, an denen der AMSTRAD den Timer speichert, auf Null gesetzt. Dann lassen wir so viele Spielzyklen laufen, wie wir wollen, und wir steuern, in welchem Zyklus wir uns befinden, mit der Variablen "cycle", die wir in jedem Zyklus um eine Einheit erhöhen. Nach dem Verlassen dieser Phase oder dieses Bildschirms führen wir :

FPS= Zyklus * 300/ (TIME - A)

Und jetzt haben wir die FPS unseres Spiels. Ich werde alles in der Reihenfolge für Sie unten setzen:

Gehen wir davon aus, dass wir uns auf einem 6128

POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0 A=TIME

<hier gehen Sie zu die Programm . das läuft. Ihr Zyklus von Zyklus, einschließlich Zyklus=Zyklus+1 >

Wir erhalten hier nach dem Ausgangszustand der Anzeige

FPS= cycle * 300/ (TIME - A)

PRINT "FPS =";FPS

Zur Verdeutlichung: Die Zeit, die vom Beginn der Sendung bis zu ihrem Ende vergeht, ist TIME-A, ausgedrückt in 1/300 Sekundenbruchteilen. Um sie in Sekunden umzurechnen, dividieren Sie durch 300.

Sekunden= (TIME -A) /300

Wenn in diesen Sekunden n Zyklen ausgeführt wurden (z. B.), dann hat ein Zyklus gedauert: **Tc= 300*(TIME-A)/n**

Und die Anzahl der Zyklen, die in einer Sekunde ausgeführt werden können (die FPS), ist der Kehrwert, d. h. **FPS = 1/Tc = n*300/(TIME-A)**.

8 Sprites

8.1 Bearbeiten von Sprites mit SPEDIT und Zusammensetzen von Sprites

Spedit (Simple Sprite Editor) ist ein Werkzeug, mit dem Sie Ihre eigenen Charakter- und Feindbilder erstellen und in Ihren BASIC-Programmen verwenden können.

Spedit ist in BASIC geschrieben, und es ist sehr einfach, so dass Sie es modifizieren können, um Dinge zu tun, die nicht in Betracht gezogen werden und an denen Sie interessiert sind. Es läuft auf dem Amstrad CPC, obwohl es für die Verwendung mit dem Winape-Emulator konzipiert ist.

Als erstes muss winape so konfiguriert werden, dass die Druckausgabe in eine Datei ausgegeben wird. In diesem Beispiel habe ich die Druckausgabe in der Datei printer5.txt abgelegt

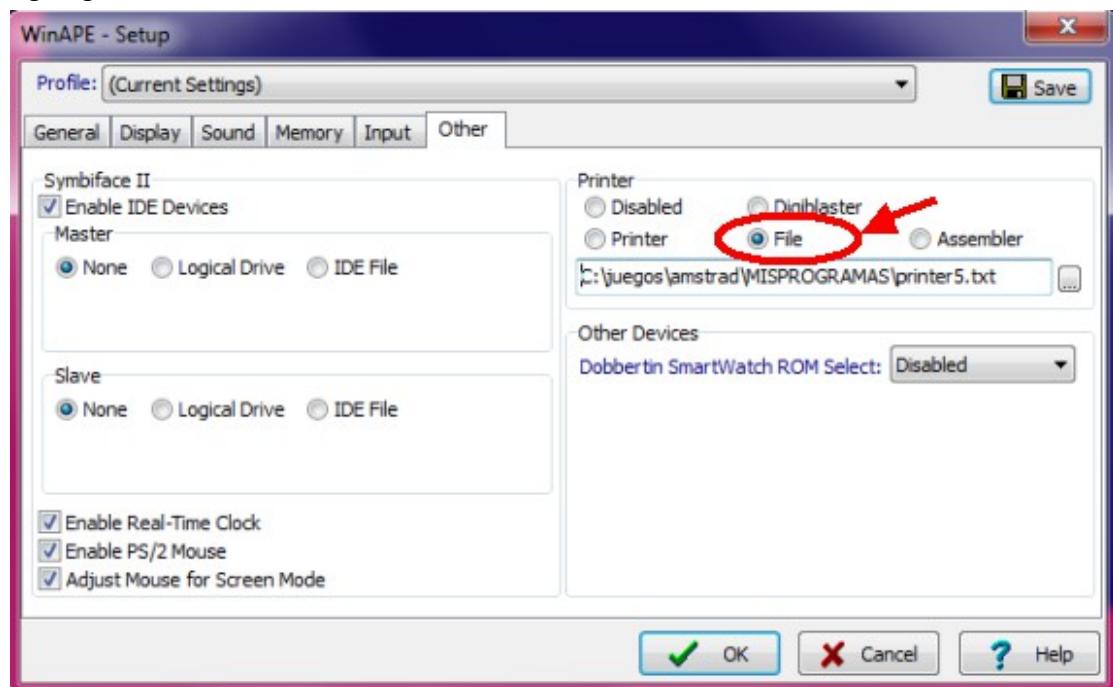


Abb. 20 Umleitung des CPC-Druckers auf eine Datei mit Winape

Wenn Sie SPEDIT starten, erhalten Sie das folgende Menü, in dem Sie wählen können, ob Sie ein Sprite bearbeiten oder ein Sprite aus einer Bilddatei (.scr) aufnehmen wollen.

Sprite Capture ist ab SPEDIT V14 verfügbar. Wenn Sie ein Sprite einfangen wollen, müssen Sie eine Bilddatei (.scr) auf der Festplatte haben, die nur eine Binärdatei mit 16384 Bytes ist. Dabei kann es sich um ein Bild aus einem Spiel handeln, das Sie aufgenommen haben und in dem es Bilder von Charakteren gibt, die Ihnen gefallen und die Sie nicht lange bearbeiten wollen.

Wenn Sie sich für die Bearbeitung eines Sprites entscheiden, fragt das Programm Sie nach der zu verwendenden Palette. Sie können eine Standardpalette oder eine eigene

Palette wählen, die Sie definieren möchten. Sie können auch eine Palette verwenden, die Sie zuvor gespeichert haben (sie ist immer in der Datei pal.dat gespeichert, drücken Sie einfach "i", während Sie ein Sprite bearbeiten). Wenn du dich entscheidest, deine eigene Palette zu definieren, musst du die BASIC-Zeilen, in denen die alternative (oder "benutzerdefinierte") Palette definiert ist, neu programmieren.

ist ein Unterprogramm, das von GOSUB aufgerufen wird, wenn Sie in der Antwort auf die Frage, welche Palette Sie verwenden möchten, "2" drücken.

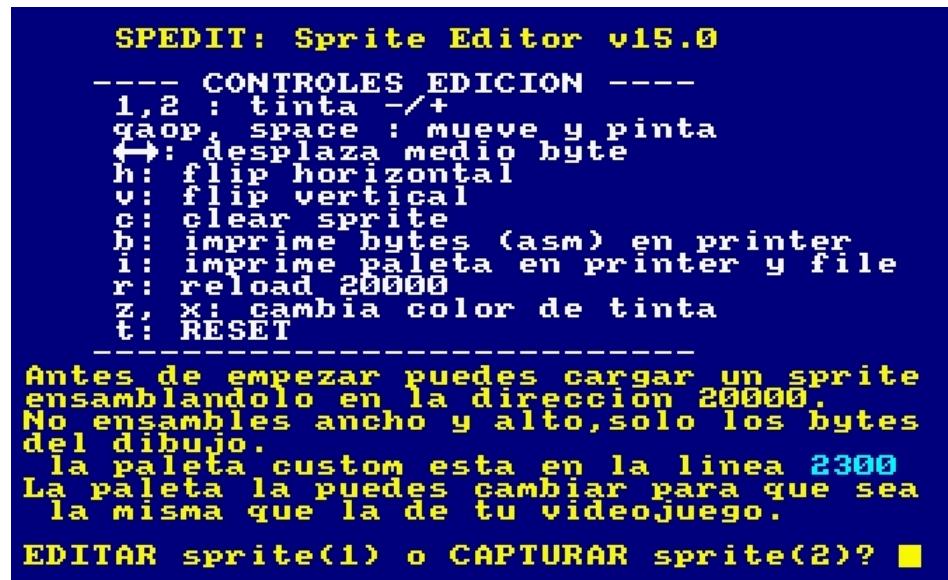


Abb. 21 SPEDIT-Startbildschirm

Mit dem Werkzeug können Sie zwischen Modus 1 und Modus 0 wählen. Im Bearbeitungsmodus können Sie Zeichnungen mit Hilfe des Bildschirms bearbeiten. Sie bearbeiten ein blinkendes Pixel und die Koordinaten, an denen Sie sich befinden, sowie der Wert des Bytes, in dem Sie sich befinden, werden unten angezeigt.

Wenn er nach der Breite und Höhe des Sprites fragt, denken Sie daran, dass **die maximale Höhe eines Sprites in 8BP 127 Zeilen beträgt, ebenso wie seine maximale Breite in Bytes**. Beachten Sie auch, dass die Breite in Pixeln gefragt wird, aber Sie sollten wissen, dass der Editor intern in Bytes arbeitet. Wenn Sie also ein Bild im Modus 0 erstellen wollen, muss die Breite eine gerade Zahl sein (ein Byte = 2 Pixel) und wenn Sie ein Bild im Modus 1 erstellen wollen, muss die Breite ein Vielfaches von 4 sein (ein Byte = 4 Pixel).



Abb. 22 SPEDIT Bearbeitungsbildschirm

Mit SPEDIT können Sie Ihr Bild "spiegeln", um dieselbe Figur mühelos nach links laufen zu lassen, indem Sie einfach H (horizontales Spiegeln) drücken, und dasselbe kann auch vertikal geschehen. Außerdem können Sie das Bild an einer imaginären Achse spiegeln, die sich in der Mitte der Figur befindet, sowohl vertikal als auch horizontal. Dies ist sehr nützlich für symmetrische oder fast symmetrische Zeichen, wo eine Zeichenhilfe immer nützlich ist.

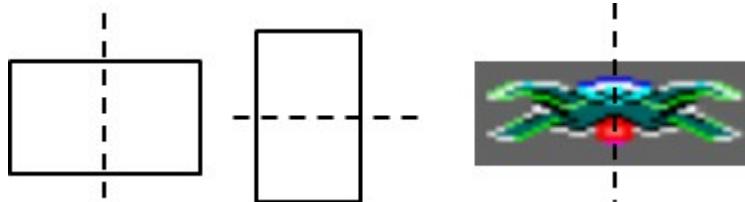


Abb. 23 symmetrische Sprites mit SPEDIT

Seit Version 11 von SPEDIT wird der AMSTRAD-Modus 1 unterstützt, so dass Sie Sprites im Modus 1 problemlos bearbeiten und auch den Spiegelungsmechanismus nutzen können.



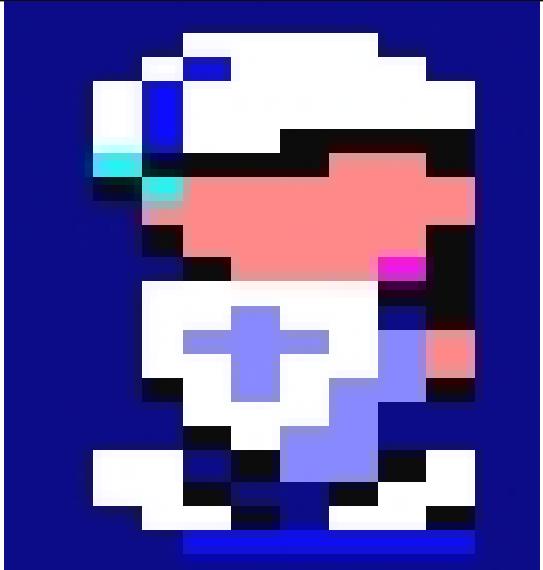
Abb. 24 Bearbeiten von Sprites in MODE 1 mit SPEDIT

Sobald Sie Ihren Dummy definiert haben, müssen Sie die Taste "b" drücken, um den Assemblercode zu extrahieren. Dadurch wird an den Drucker (in die Datei, die wir als Ausgabe definiert haben) ein Text wie der folgende gesendet, dem Sie einen Namen hinzufügen können, ich habe ihn "SOLDADO_R1" genannt.

```

;----- BEGIN IMAGE -----
SOLDADO_R1
db 6 ; ancho
db 24 ; alto
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;----- END IMAGE -----

```



Beachten Sie, dass ich immer ein Byte links von der Null stehen gelassen habe. Ich habe dies getan, damit der Soldat, wenn er sich nach rechts bewegt, "sich selbst löscht", sonst würde er eine Spur hinterlassen und den Bildschirm "verschmieren", wenn er sich vorwärts bewegt.

Abb. 25 Soldat im .asm-Format

Sobald Sie den ersten Rahmen Ihres Soldaten erstellt haben, können Sie die Arbeit verlassen und an einem anderen Tag fortfahren. Um mit dem gezeichneten Soldaten zu beginnen und ihn weiter zu retuschieren oder zu modifizieren, um einen weiteren Rahmen zu erstellen, können Sie den Soldaten an der Adresse **20000** zusammensetzen und die Breite und Höhe entfernen. Sobald Sie den Soldaten aus dem Winape zusammengesetzt haben, teilen Sie SPEDIT mit, dass Sie ein Sprite der gleichen Größe bearbeiten wollen und drücken Sie im Bearbeitungsbildschirm "r" (reload). Das Sprite wird von der Adresse **20000** geladen, wo Sie es "assembliert" haben.

Ein großer Teil des Reizes eines Spiels sind seine Sprites. Sparen Sie nicht daran, machen Sie es langsam und geschmackvoll und Ihr Spiel wird viel besser aussehen.

```

org 20000
----- BEGIN IMAGE -----
SOLDADO_R1
;db 6 ; ancho ojo! comentamos esta linea
;db 24 ; alto ojo! comentamos esta linea
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
----- END IMAGE -----

```

Damit wissen Sie, was es bedeutet, ein Sprite "zusammenzusetzen". Es bedeutet einfach, die Datenbytes, aus denen es besteht, an aufeinanderfolgende Speicheradressen zu setzen, in diesem Fall beginnend mit **20000**.

SPEDIT belegt nur sehr wenig Speicherplatz und diese Adresse ist weit vom Programm entfernt, so dass es kein Problem ist, das SPEDIT-Programm zu "beschädigen", wenn wir es assemblieren.

Abb. 26 Montage der Grafiken

Um zu wissen, an welcher Speicheradresse jedes Bild assembliert wurde, benutzen Sie das winape Menü: Assemble->symbols

Damit sehen Sie eine Beziehung zwischen den von Ihnen definierten Tags, wie z.B. "SOLDADO_R1" und der Speicheradresse (in hexadezimal), aus der sie zusammengesetzt wurde. Um die Adressen von hexadezimal in dezimal umzuwandeln, können Sie den Windows-Rechner im Modus "Programmierer" verwenden.

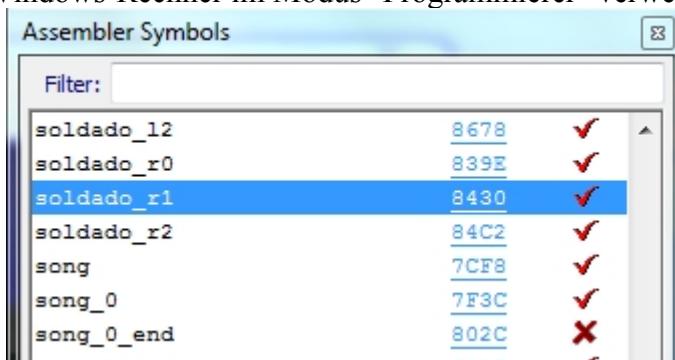


Abb. 27: Detail der Symbole in Winape

Wenn Sie die verschiedenen Animationsphasen Ihres Soldaten erstellt haben, können Sie diese zu einer Animations-"Sequenz" zusammenfassen. Animationssequenzen sind Listen von Bildern und werden nicht mit SPEDIT definiert. Mit SPEDIT bearbeiten Sie einfach die

"Bilder". In einem späteren Abschnitt werde ich erklären, wie man der 8BP-Bibliothek mitteilt, aus welchen Bildern eine Animationssequenz besteht.

Die Bilder, die Sie für Ihr Spiel erstellen, werden in einer einzigen Datei gespeichert, die z.B. "images_mygame.asm" heißt. Diese Datei beginnt mit einer Liste von Bildern, die Sie in den 8BP-Befehlen in BASIC mit einem Index referenzieren können, unabhängig von der Adresse, an der sie z. B. zusammengesetzt sind:

BILD_LISTE

**Das erste Bild wird immer dem Index 16 zugeordnet, das nächste Bild dem Index 16 und das folgende Bild dem Index 16.
17 und so weiter**

**SOLDIER_R0; 16
dw SOLDIER_R1; 17
dw SOLDIER_R2; 18**

dw

Sobald alle Bilder fertig sind, können Sie die Bibliothek zusammen mit der Musik und den Grafiken zusammenstellen.

SEHR WICHTIG: Achten Sie darauf, dass die Grafik nicht mehr als 8440 Bytes umfasst. Prüfen Sie dazu, wo der "**_END_GRAPH**"-Tag eingebaut wird, der kleiner als 42040 sein muss (da $42040 - 33600 = 8440$ Bytes). Wenn es an einer höheren Adresse assembliert ist, dann "zerkleinern" Sie Adressen, die der BASIC-Interpreter benötigt, und der Computer kann abstürzen. Wenn Sie mehr Speicher für die Grafik benötigen, sollten Sie die "zusätzliche" Grafik in einen freien Speicherbereich, z.B. 22000, einbauen und in Ihrem Programm einen Speicherplatz 21999 verwenden, wodurch der für BASIC verfügbare Speicher reduziert wird.

8.2 Ein Sprite drucken

Schauen wir uns die Grundlagen des Druckens eines Sprites an. Nehmen wir an, Sie haben einen Soldaten gezeichnet und ihn in die Datei images_mygame.asm eingefügt. Nehmen wir an, es ist Ihr erstes Bild und hat daher den Bezeichner 16.

In 8BP gibt es 32 Sprites (nummeriert von 0 bis 31). Sie können jedem Sprite mit dem Befehl |SETUPSP ein Bild zuweisen. Mit diesem Befehl können Sie viele Attribute eines Sprites ändern, nicht nur sein Bild. Das Attribut des Sprites, das Sie ändern wollen, ist der zweite Parameter des SETUPSP-Befehls.

|SETUPSP, <Seitennummer>, <Parameter> , <Wert>, <Seitennummer>, <Parameter> , <Wert>, <Wert>.

Um ein Bild zuzuweisen, müssen Sie den Parameter 9 verwenden. Ein sehr einfaches Programm, das ein Sprite druckt, wäre das folgende:

```
10 SPEICHER 23499
20 CALL &6B78: REM installiert die RSX-Befehle
30 DEFINT A-Z : REM ganzzahlige numerische Variablen (schneller)
40 |SETUPSP,31,9,16: REM weist dem Sprite 31 das Bild 16 zu
50 x=40:y=100: REM-Koordinaten, wo gedruckt werden soll
60 |LOCATESP,31,y,x: REM platziert Sprite 31
70 |PRINTSP,31: REM druckt Sprite 31
```

Dies wäre das Ergebnis



Abb. 28 Drucken eines Sprites auf dem Bildschirm

Sie werden den **SETUPSP-Befehl** häufig verwenden und ihn nach und nach in allen Einzelheiten kennenlernen. Die SETUPSP-Parameter sind nicht einfach irgendwelche Zahlen. Es gibt 7 mögliche Werte und zwar die folgenden (0,5,6,7,8,9,15):

- Parameter 0: ändert das Statusbyte des Sprites
- Parameter 5: Vy ändern. Vx kann auch gleichzeitig geändert werden, wenn man ihn am Ende als zusätzlichen Parameter hinzufügt (etwa so: SETUPSP, <id>,5, Vy,Vx)
- Parameter 6: Änderungen Vx
- Parameter 7: Änderungsreihenfolge (nimmt die Werte 0..31 an)
- Parameter 8: change frame_id (nimmt die Werte 0..7 an)
- Parameter 9: Bild ändern. Das angegebene Bild kann eines aus der anfänglichen Liste der Bilder in der Datei images_mygame.asm sein,
- Parameter 15: Pfad ändern (belegt 1Byte)

Im Laufe der Lektüre dieses Handbuchs werden Sie die Bedeutung der Attribute eines Sprites verstehen und wissen, wie Sie sie nach Ihren Bedürfnissen einsetzen können.

8.3 Sprite-Spiegelung

Es kommt häufig vor, dass Sie Figuren zeichnen müssen, die in verschiedene Richtungen gehen, und zwar mit jeweils unterschiedlichen Bildern. Das Bild des Sprites in der linken Richtung wird das Spiegelbild der rechten Richtung sein. Sie können zwei Bilder definieren und sie im Speicher ablegen, aber seit V33 gibt es eine Möglichkeit, den RAM-Verbrauch für diese Bilder zu vermeiden. Dies wird "gespiegelte" Bilder genannt.



Abb. 29: Beispiel für gespiegelte Bilder

Ein "gespiegeltes" Bild ist ein Spiegelbild eines anderen Bildes, das erstellt und in die Bilddatei aufgenommen wurde. Wenn Sie ein Bild auf diese Weise definieren, müssen Sie es nicht speichern. Dazu fügen Sie einfach eine Liste der gespiegelten Bilder in die Bilddatei ein (die ich normalerweise "images_mygame.asm" nenne). Zu diesem Zweck finden Sie am Anfang der Datei einen Abschnitt, der durch die Tags "BEGIN_FLIP_IMAGES" und "END_FLIP_IMAGES" begrenzt ist.

```
;;
_BEGIN_FLIP_IMAGES
Hier werden Bilder eingefügt, die als andere vorhandene Bilder definiert sind, aber horizontal gespiegelt werden.
JOE_LEFT dw JOE_RIGHT; joe_left ist die geflippte Version von joe_right

Ich definiere die Frames des Soldaten auf der linken Seite als gespiegelte
SOLDIER_L0 dw SOLDIER_R0;
SOLDIER_L1 dw SOLDIER_R1; SOLDIER_L2 dw
SOLDIER_R2; SOLDIER_L1_UP dw
SOLDIER_R1_UP SOLDIER_L1_DOWN dw
SOLDIER_R1_DOWN

_END_FLIP_IMAGES
;;
```

Umgedrehte Bilder können genauso wie normale Bilder verwendet werden. Im Folgenden erfahren Sie, wie Sie Animationssequenzen erstellen, die Sie sowohl mit gespiegelten als auch mit nicht gespiegelten Bildern erstellen können. Im Grunde ist ein "gespiegeltes" Bild so, als wäre es real, obwohl es sich um ein "virtuelles" Bild handelt, das nicht gespeichert wird und beim Ausdruck als Spiegelbild eines vorhandenen Bildes berechnet wird. Gespiegelte Bilder werden sowohl im Modus 0 als auch im Modus 1 unterstützt.

Der Nachteil der gespiegelten Bilder ist, dass sie teurer im Druck sind und 1,8 Mal so lange brauchen wie ein normaler Druck, was sich in einer geringeren Geschwindigkeit für Ihr Spiel niederschlagen könnte. Wenn es sich bei Ihrem Spiel um ein Arcade-Spiel (ein Shoot'em Up) handelt, bei dem es auf maximale Geschwindigkeit ankommt, empfehle ich, keine massiv flimmernden Bilder zu verwenden. In Abenteuerspielen, Screen Passing Games, Labyrinthen usw. sind sie jedoch eine ausgezeichnete Wahl. Versuchen Sie auf jeden Fall, sie in Ihrer Spielhalle zu verwenden, denn wenn es nicht viele Flips gleichzeitig gibt, kann die resultierende Geschwindigkeit sehr akzeptabel sein.

Ich habe horizontal gespiegelt und nicht vertikal, denn normalerweise ist eine Figur, die nach links geht, das Spiegelbild der gleichen Figur, die nach rechts geht, während beim Gehen nach oben der Rücken und beim Gehen nach unten die Brust und das Gesicht zu sehen sind. Daher ist das vertikale Spiegeln nicht so nützlich wie das horizontale Spiegeln, und im Interesse der Verringerung der Größe von 8BP habe ich es nicht in

seine Möglichkeiten aufgenommen.

WICHTIG: Das Spiegeln ist nicht auf Segmentbilder anwendbar, die im 8BP-Pseudo-3D-Modus verwendet werden können.

8.4 Sprites mit Überschreibung

Seit der Version v22 von 8BP ist es möglich, transparente Sprites zu editieren, d.h. Sprites, die über einen Hintergrund fliegen können und diesen beim Passieren zurücksetzen. Dazu müssen Sprites, die diese Möglichkeit haben, mit einer "1" im Overwrite-Flag des Statusbytes (Bit 6) konfiguriert werden. Im nächsten Abschnitt wird das Statusbyte im Detail erklärt. Schauen wir uns nun an, wie man ein Sprite mit dieser Fähigkeit mit SPEDIT bearbeiten kann.

Viele Spiele verwenden eine Technik namens "Double Buffering", um den Hintergrund wiederherstellen zu können, wenn sich ein Sprite über den Bildschirm bewegt. Dabei wird eine Kopie des Bildschirms (oder des Spielbereichs) in einem anderen Bereich des Speichers abgelegt, so dass wir, selbst wenn unsere Sprites den Hintergrund zerstören, immer in diesem Bereich nachsehen können, was darunter lag, und ihn wiederherstellen können. Das ist eigentlich das Grundprinzip, aber es ist ein bisschen komplexer. Es wird in den Doppelpuffer (auch "Backbuffer" genannt) gedruckt, und wenn alles gedruckt ist, wird es entweder auf den Bildschirm ausgegeben oder die Startadresse des Videospeichers wird von der ursprünglichen Bildschirmadresse auf die neue, die Doppelpufferadresse, umgestellt. Die Umschaltung erfolgt augenblicklich (je nach Maschinentyp). Zum Aufbau des nächsten Bildes wird die ursprüngliche Bildschirmadresse verwendet, auf die der Bildspeicher nicht mehr zeigt. Dort wird das neue Bild aufgebaut und bei jedem Bild abwechselnd zurückgeschaltet. Diese Techniken funktionieren zwar sehr gut, haben aber für unsere Zwecke einige Nachteile: Sie benötigen mehr CPU-Zeit und verbrauchen viel mehr Speicher (bis zu 16 KB zusätzlich), so dass nur sehr wenig Speicher für unser BASIC-Programm übrig bleibt. Wenn ein Spiel komplett in Assembler entwickelt wird, ist das nicht so schlimm, denn 10KB Assembler reichen sehr weit, aber 10KB BASIC sind zu wenig. Manche Videospiele verkleinern den Spielbereich, um nicht so viel Speicherplatz zu verbrauchen, aber das macht sie ein bisschen ärmer.

Die bei 8BP angewandte Lösung ist von dem Programmierer Paul Shirley (Autor von "Mission Genocide") inspiriert, unterscheidet sich jedoch geringfügig. Ich werde die Geschichte von 8BP direkt erzählen:



Abb. 30 Sprites mit Überschreibung in 8BP

Die Idee ist, dass **der Hintergrund niemals durch die Sprites zerstört wird, die über ihn hinweggehen**, so dass es nicht notwendig ist, ihn zu speichern. Diese scheinbare "Magie" hat ihre Logik: Sie besteht darin, die Hintergrundfarbe in der Farbe des Sprites zu "verstecken", das darüber gemalt wird.

Auf dem AMSTRAD wird ein Pixel des Modus 0 mit 4 Bits dargestellt, so dass bis zu 16 verschiedene Farben aus einer Palette von 27 Farben möglich sind. Wenn wir also ein Bit für die Hintergrundfarbe und 3 Bits für die Sprite-Farben verwenden, haben wir insgesamt 2 Hintergrundfarben.

7 Farben + 7 Farben + 1 Farbe für die Transparenz = 9 Farben insgesamt. Auf diese Weise kann die Hintergrundfarbe in der Farbe des Sprites "versteckt" werden, allerdings um den Preis, dass die Anzahl der Farben von 16 auf nur 9 reduziert wird. Bestimmte Zierelemente auf dem Spielbildschirm können jedoch mehr Farbe haben, da die Sprites nicht über sie hinweggehen (z. B. die Blätter an den Bäumen oder das Dach im Beispiel unten), so dass wir eine gewisse Menge an Farbe in unserem Spiel erhalten können.

Um diese Art von Sprites zu bearbeiten, müssen wir eine richtige Palette mit 9 Farben verwenden, wobei für jede Sprite-Farbe zwei binäre Codes verwendet werden (entsprechend der 0 und 1 des Hintergrundbits). Wenn Sie in SPEDIT die Palettenoption "2" wählen, haben Sie eine auf diese Weise definierte Palette, die Sie jedoch nach Belieben ändern können. Sie ist wie folgt aufgebaut:

```

2300 REM ----- PALETA sprites transparentes MODE 0-----
2301 INK 0,11: REM azul claro
2302 INK 1,15: REM naranja
2303 INK 2,0 : REM negro
2304 INK 3,0 :
2305 INK 4,26: REM blanco
2306 INK 5,26:
2307 INK 6,6: REM rojo
2308 INK 7,6:
2309 INK 8,18: REM verde
2310 INK 9,18:
2311 INK 10,24: REM amarillo
2312 INK 11,24 :
2313 INK 12,4: REM magenta
2314 INK 13,4 :
2315 INK 14,16 : REM naranja
2316 INK 15, 16:
2317 AMARILLO=10
2420 RETURN

```

Abb. 31 Beispiel einer Überschreibungspalette

Wie Sie sehen, werden nach Farbe 0 und 1 alle Farben zweimal wiederholt. Auf diese Weise können Sie Ihre eigene Palette zusammenstellen. Sie können sich selbst helfen, indem Sie den Anhang dieses Handbuchs über die Farbpalette konsultieren.

000 1	=		<i>Color de fondo</i>	Paleta ejemplo
110 0	=		<i>Color de sprite</i>	0000 =  0001 = 
<i>Cuando el sprite se imprime:</i>				
Fondo OR sprite = 1101 =			1100 =  1101 = 	
<i>Cuando el sprite se marcha:</i>				
Pixel OR 0001 = 0001 =				
<i>El fondo nunca fue destruido, estaba "escondido" en el sprite</i>				

Die Technik könnte sein Zusammenfassend lässt sich sagen, dass **der Hintergrund nie wirklich von den Sprites zerstört wird**, sondern in den Sprites selbst "versteckt" ist, die auf den Hintergrund gedruckt werden.

Abb. 32 Überschreibemechanismus beim 8BP

Mit dem SPEDIT-Editor können Sie die Palette nach Ihren Wünschen verändern, ohne sie manuell mit INK-Befehlen bearbeiten zu müssen, und Sie können sie exportieren, um sie in unsere BASIC-Programme zu kopieren. Der Export erfolgt, indem die INK-Befehle, aus denen die Palette besteht, an den Drucker gesendet werden (der Drucker wird von winape in eine Datei umgeleitet). Wir haben die Tasten z/x, um die Palette zu verändern und die Option "i", um sie in die Ausgabedatei zu exportieren. Dies ist ein Beispiel dafür, wie es exportiert wird (es ist eine Palette ohne Überschreiben):

```
TIN 0 , 1
TE
TIN 1 ,
TE
TIN ,
TE
TIN ,
TE
TIN , 26
TE
TIN 5 , 0
TE
TIN ,
TE
TIN , 8
TE
TIN 8 → 10 PALETA -----
TE
TIN ,
TE
TIN 10 , 14
TE
TIN , 16
TE
TIN , 18
TE
TIN , 22
TE
TIN , 0
TE
TIN , 11
TE
```

Wenn Sie die Taste "i" drücken, wird auch die Datei "pal.dat" auf der Festplatte gespeichert (die .dsk), so dass Sie sie später laden können, indem Sie die Option 3 wählen, um die Palettenauswahlfrage zu beantworten.

Die Sprites, die du für die Hintergrundzeichnungen verwendest, können nur die Farben 0 und 1 haben, aber die Sprites, die du für die Ornamente verwendest, durch die sich bewegenden Sprites nicht hindurchgehen, können alle 9 Farben haben.

Sie können die Farbigkeit der Szenerie auch durch Sprite-Elemente anstelle von Hintergründen erhöhen, wie z. B. den grünen Kessel im obigen Beispiel. Auf diese Weise können Sie sehr bunte Ergebnisse erzielen.

Tinte 0001 hat eine "besondere" Verwendung. Wenn Sie ein Sprite bearbeiten, das das Überschreibungsflag nicht verwendet, ist Tinte 1 einfach eine Farbe. Wenn Sie jedoch ein Sprite mit einem aktiven Überschreibungsflag in seinem Statusbyte bearbeiten,

werden diese Pixel beim Drucken ungefärbt gelassen und respektieren das, was darunter liegt. Dies ermöglicht es, dass Kollisionen zwischen Sprites nicht "rechteckig" sind, sondern die Form des Sprites erhalten bleibt.

Code	Bedeutung
0000	Hintergrundfarbe 1. Wenn ein Sprite sie verwendet und Sie das Überschreibungsflag aktivieren, bedeutet das "Transparenz", d.h. Drucken durch Zurücksetzen der Fonds
0001	Hintergrundfarbe 2. Wenn ein Sprite sie verwendet und Sie das Überschreibungsflag aktivieren, bedeutet sie nicht mehr eine Farbe, sondern "nicht drucken". Was auch immer in diesem Pixel ist, wird respektiert, z. B. ein Pixel, das von einem anderen Sprite gefärbt wurde, das zuvor mit dem Flag dass wir uns überschneiden.
0010	Sprite-Farbe 1
0011	Sprite-Farbe 2
0100	Sprite-Farbe 3
0101	Sprite-Farbe 4
1010	Sprite-Farbe 5
1011	Sprite-Farbe 6
1101	Sprite-Farbe 7
1110	Sprite-Farbe 8
1111	Sprite-Farbe 9

Insgesamt 9 Farben:

- 2 im Hintergrund
- 7 für Sprites (eigentlich 8, aber ein -000- bedeutet Transparenz)
- Die Elemente Zierelemente können alle 9 verwenden.

Als Nächstes zeige ich Ihnen ein Sprite, bei dem ich mit der Farbe 0001 das gemalt habe, was nicht ~~Sprite-Farbe 2~~ soll, d. h., bei dem der Hintergrund nicht einmal wiederhergestellt wird, weil es mit den restlichen Pixeln bei 0000 ausreicht, die Spur des Sprites zu löschen, während es sich bewegt.

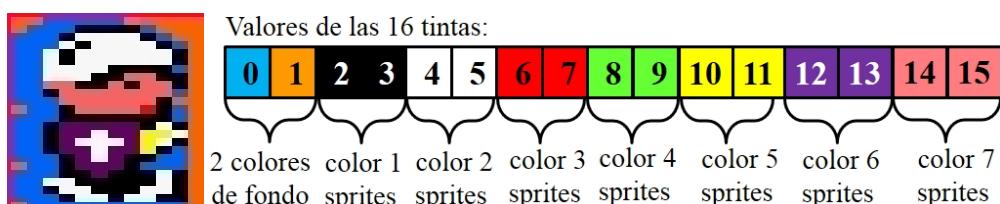


Abb. 33 Sprite und Palette zum Überschreiben ausgelegt

WICHTIG, WENN SIE IHRE SPRITES MIT OVERWRITE BEARBEITEN:

Verwenden Sie die Hintergrundbits nicht für Ihre Sprite-Tinten, es sei denn, Sie suchen nach speziellen Effekten, wie später in diesem Kapitel beschrieben. Wenn Sie sich nicht an diese Regel halten, können Sie "seltsame" Effekte feststellen. Mit anderen Worten:

- Wenn der Hintergrund aus 1 Bit besteht, dann färben Sie Ihre Sprites mit Tinten, die auf 0 enden (d.h. Tinten 2, 4, 6, 8, 10, 12, 14).
- Wenn der Hintergrund 2 Bits hat, dann färben Sie Ihre Sprites mit Tinten, die auf 00 enden (d.h. Tinten 0100, 1000, 1100, die jeweils Tinten 4, 8 und 12 sind).

Wie Sie sich vorstellen können, wird im Fall des Kessels, der sich nicht bewegt und sich daher nicht selbst löscht, sein gesamter Umriss mit der Farbe 0001 gemalt. Dies

ermöglicht perfekte Kollisionen, ohne rechteckige Formen, die deutlich machen, dass die Sprites eigentlich

sind Rechtecke. Das Endergebnis ist, wie unten gezeigt, eine Mehrfachkollision.



Wie Sie vielleicht schon vermutet haben, ist die Kollision nicht nur perfekt, sondern zeigt auch, dass die Sprites nach ihrer Y-Koordinate geordnet wurden, so dass das letzte gedruckte Sprite dasjenige ist, das sich an der untersten Position befindet. Dies wird durch einen einfachen Parameter beim Drucken der Sprites mit dem Befehl |PRINTSPALL erreicht, den wir später sehen werden.

Abb. 34 Mehrfache Kollision, Farbeffekt 0001

Druckvorgänge mit diesem Mechanismus sind sehr schnell, ohne dass so genannte "Sprite-Masken" definiert werden müssen. Sprite-Masken sind Bitmaps in Sprite-Größe, die dazu dienen, Druckvorgänge zu beschleunigen. In diesem Fall sind sie nicht notwendig. Die folgende Abbildung zeigt eine typische Maske in Verbindung mit einem Sprite. Zuerst wird normalerweise die UND-Verknüpfung zwischen dem Hintergrund und der Maske durchgeführt und dann die ODER-Verknüpfung mit dem Sprite. In 8BP geht das schneller, weil das Sprite das für den Hintergrund bestimmte Bit nicht berührt, so dass die ODER-Verknüpfung zwischen dem Hintergrund und dem Sprite den Hintergrund respektiert, während das Sprite gemalt wird. Wenn du das nicht so gut verstehst, mach dir keine Sorgen, es ist nicht wichtig, es zu verstehen, da es in 8BP nicht notwendig ist.

sprite	mask	
0 2 2 0	1 0 0 1	Metodo convencional:
2 3 3 2	0 0 0 0	Se imprime
0 2 2 0	1 0 0 1	Fondo AND mask OR sprite
0 2 2 0	1 0 0 1	
0 0 0 0	1 1 1 1	Metodo 8BP:
		Imprimimos
		Fondo OR sprite

Abb. 35 Bei 8BP sind keine Masken erforderlich.

Das Drucken von Sprites mit aktivem Overwrite-Flag ist teurer als das Drucken ohne Overwrite. Obwohl es keine Maske erfordert und sehr schnell ist, dauert es etwa 1,6 Mal so lange wie das Drucken eines Sprites ohne Überschreiben. Benutzen Sie es daher nur, wenn es nötig ist, und verwenden Sie es nicht, wenn Ihr Spiel keine Hintergrundzeichnung hat, die die Sprites respektieren müssen. Die Kombination aus Überschreiben und Spiegeln ist sogar noch teurer (sie verbraucht 2,2 Mal so viel Zeit wie ein normaler Druck ohne Überschreiben und Spiegeln), also berücksichtigen Sie dies bei Ihren Spielen.

8.4.1 Sprite-Überlappungen durch Überschreiben verbessern

Ein sehr wertvolles Merkmal des Überschreibens ist, dass es die Überschneidungen zwischen Sprites "perfekt" macht, denn wenn du das Sprite (das du überschreiben willst) bearbeitest, benutzt du Tinte 1 um es herum, wenn du es drückst, werden alle

Pixel, die Tinte 1 haben, "überschrieben".

wird "transparent", d.h. wenn Sie zuvor ein anderes Sprite gemalt haben, werden dessen Pixel respektiert, was zu "perfekten" Überlappungen führt. Diese Funktion des Überschreibens von Sprites kann für Sie interessant sein, auch wenn Ihr Spiel kein Überschreiben erfordert, weil es einen schwarzen oder einfarbigen Hintergrund hat.

Nur die Pixel, die Sie mit null Tinte gezeichnet haben, werden gelöscht (indem der Hintergrund zurückgesetzt wird). Im Beispiel des Balls sind die Null-Pixel die hinteren Pixel, so dass Sie ihn löschen können, wenn Sie ihn nach rechts bewegen.

Anhand des folgenden Beispiels können Sie es gut nachvollziehen. Sie verlieren Farbe, weil es nur 9 Farben gibt, aber die Überlappungen zwischen den Sprites sind sehr gut. Seien Sie vorsichtig, denn Sie verlieren auch etwas an Druckgeschwindigkeit.

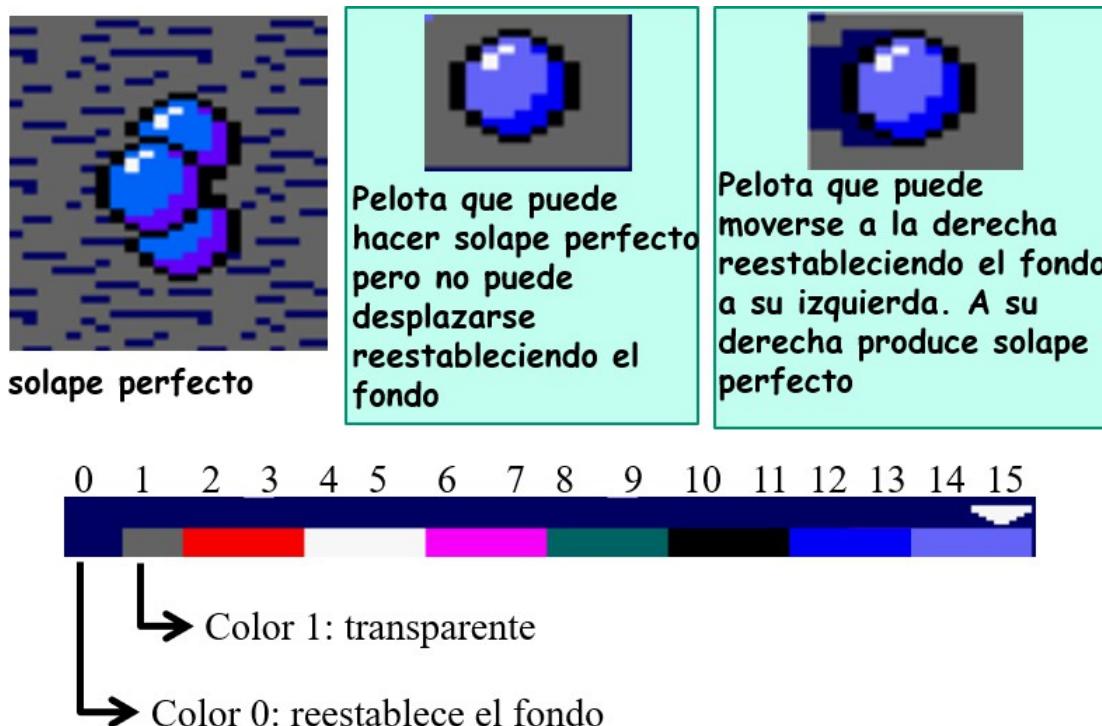


Abb. 36 Perfekte Überschneidungen

8.4.2 Überschreiben mit 4 Hintergrundfarben

Seit Version V33 ist es möglich, die Anzahl der für den Hintergrund verwendeten Bits zu wählen, indem man den Befehl |PRINTSP mit der Angabe von Sprite 32 aufruft, das nicht existiert. Sie können 1 oder 2 Bits für den Hintergrund wählen, was 2 bzw. 4 Hintergrundfarben ermöglicht.

|PRINTSP, 32, <Hintergrundbitnummer>.

Beispiele:

 PRINTSP, 32, 1 : ' mit 1 Bit Hintergrund haben wir 2 Farben für den Hintergrund
 PRINTSP, 32, 2 : ' mit 2 Bit Hintergrund haben wir 4 Farben für den Hintergrund

Sobald dieser Befehl aufgerufen wird, wird die 8BP-Bibliothek so konfiguriert, dass sie die Anzahl der Bits berücksichtigt, die als Hintergrundbits verwendet werden sollen. Wenn wir 2 Hintergrundbits einstellen, muss unsere Farbpalette mit diesem Umstand

übereinstimmen. Ein Beispiel ist unten abgebildet

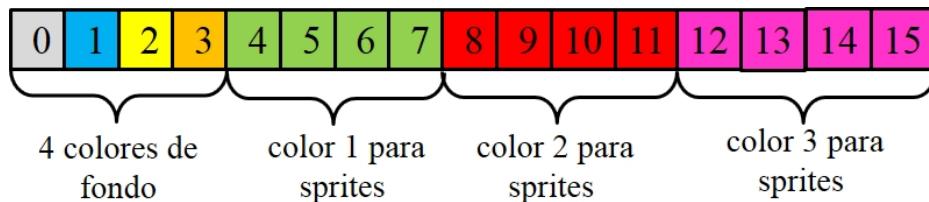


Abb. 37 Beispiel für eine Palette mit vier Hintergrundfarben (2-Bit-Hintergrund)

In diesem Beispiel haben Sie 4 Hintergrundfarben und drei Sprite-Farben. Genau wie bei der Verwendung von 1 Bit für den Hintergrund sollten Sie bei der Definition Ihrer Sprites daran denken, dass 0 Tinte Transparenz bedeutet und 1 bedeutet, dass der Hintergrund nicht zurückgesetzt wird, wodurch nicht-rechteckige Sprite-Formen möglich sind. Im folgenden Beispiel sind die 3 Farben, die für die Sprites gewählt wurden, schwarz, hellgrün und weiß.



Abb. 38 Beispiel für einen Palettensatz mit bis zu vier Hintergrundfarben (2 Bit)

Mit zwei Bits für den Hintergrund erhält man zwar schönere Dekorationen, aber der Nachteil ist, dass man nur 3 Farben für die Sprites übrig hat, während man bei Verwendung von 1 Bit für den Hintergrund bis zu 7 Farben für die Sprites hat.

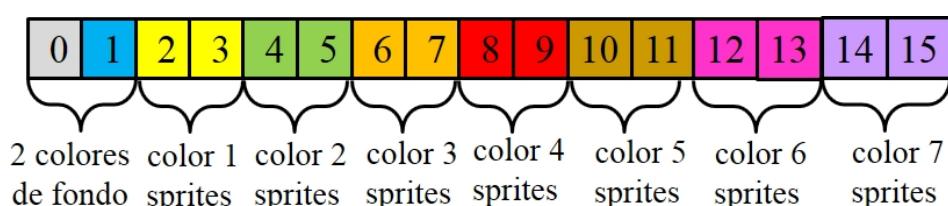


Abb. 39 Beispiel für eine Palette mit zwei Hintergrundfarben (1 Bit Hintergrund)

8.4.3 Überschreiben in MODE 1

Seit der Version V34 von 8BP ist es möglich, Sprites mit Overwrite in MODE 1 zu verwenden. Hier haben wir eine sehr starke Einschränkung, denn obwohl wir zwei Farben für den Hintergrund haben, haben wir nur eine Farbe für die Sprites.

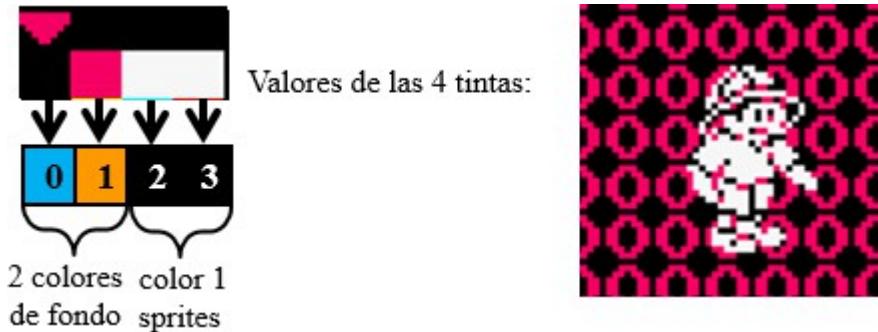
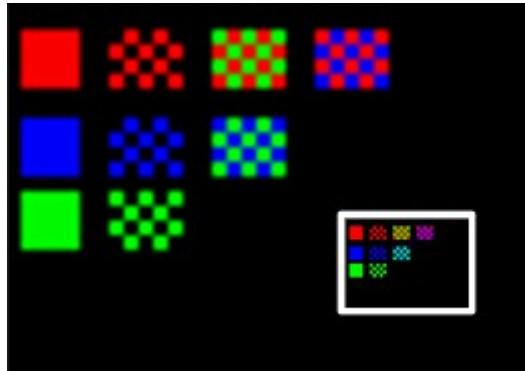


Abb. 40 Beispiel einer Palette für Modus 1

Man kann leicht den Fehler machen, zu denken, dass die Sprites eine beliebige Farbe haben und außerdem schwarz sind. Das ist aber nicht der Fall. Schwarz ist eine andere Farbe und als solche muss es bei diesem Mechanismus zwei Farben verbrauchen. Wir haben nur zwei Tinten für das Sprite, und wir haben sie für das Weiß (in diesem Beispiel) verwendet. Wie Sie sehen können, ist das Zeichen dort, wo es nicht weiß ist, transparent und nicht schwarz.

Trotz dieser strikten Beschränkung können Sie, wenn Sie sich Mühe geben, einige sehr attraktive Sprites in MODE 1 erstellen, und wenn Sie die Hintergrundfarben auf jedem Bildschirm ändern und Farbmischungen (Gitter) auf den Spielmarkern verwenden, können Sie ein sehr zufriedenstellendes Ergebnis erzielen.

Durch die Verwendung von Blending in MODE 1 können Sie 10 Farben mit nur 4 Farben simulieren. Die Gitterfarben werden gemischt und zum Beispiel sieht Grün + Rot gelb aus. Auf diesem Bild sehen Sie es vielleicht nicht so überzeugend, aber auf Ihrem Amstrad-Bildschirm werden Sie es gut sehen.



8.4.4 Wie man Sprites "hinter" den Hintergrund malt

Der gleiche Mechanismus für das Drucken von Sprites auf dem Hintergrund kann auch für das Malen hinter dem Hintergrund verwendet werden.

Wie wir bereits gesehen haben, verwenden wir zum Drucken vor dem Hintergrund Bits, die im Hintergrund nicht verwendet werden, so dass wir zwar die Anzahl der Farben reduzieren, aber das/die Hintergrundbit(s) nicht beschädigen. Wenn wir ein Bit für den Hintergrund verwenden, müssen wir zwei Farben verwenden, um dieselbe Sprite-Farbe darzustellen: eine mit dem Hintergrund-Bit auf Null gesetzt und eine mit dem Hintergrund-Bit auf 1 gesetzt.

Wenn wir jedoch diesen beiden Farben nicht dieselbe Farbe zuweisen, sondern die Farbe des Hintergrunds derjenigen Farbe zuweisen, bei der das Hintergrundbit auf 1 gesetzt ist, dann wird, wenn das Sprite den Hintergrund überlappt, die Hintergrundfarbe

zu sehen sein, so dass der Eindruck entsteht, dass das Sprite dahinter vorbeigeht. Dies funktioniert sowohl im MODE 0 als auch im MODE 1.

Im folgenden Beispiel wird ein Bit für den Hintergrund verwendet, der aus gelben Buchstaben auf schwarzem Grund besteht. Bei den Sprites handelt es sich um "Münzen", die, wie Sie sehen können, offensichtlich hinter den Hintergrund gemalt wurden.

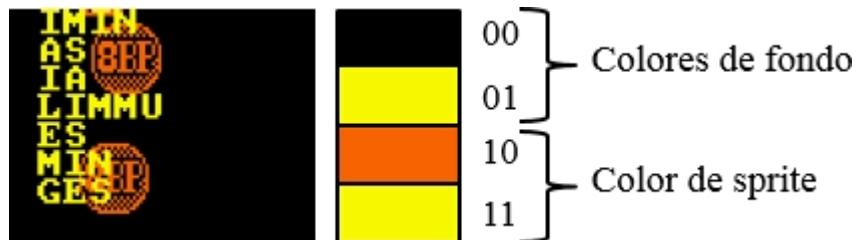


Abb. 41 Sprites "hinter" den Buchstaben gedruckt

8.4.5 Wie man mehr Farben beim Überschreiben verwendet

Wenn Sie Ihre ersten Versuche mit dem Überschreiben gemacht haben und mehr Farben in Ihrem Spiel brauchen, gibt es drei Möglichkeiten, dies zu erreichen, aber Sie müssen die 8BP-Methode verstehen:

- 1) Einige Sprites mit und einige ohne Überschreibung verwenden
- 2) Verwenden Sie Sprites, deren Farbe vom Hintergrund abhängt (bedingte Farbe).
- 3) Ein Sprite als Hintergrund verwenden

Schauen wir uns diese drei "Tricks" nacheinander an:

Verwenden Sie einige Sprites mit und einige ohne Überschreibung:

Dies ist der einfachste und am häufigsten verwendete der drei Tricks. Angenommen, nur eines Ihrer Sprites muss überschrieben werden und verbraucht 3 Farben. Das bedeutet, dass Sie diesem Sprite 6 Farben zuweisen müssen. Wenn die übrigen Sprites jedoch nicht überschrieben werden müssen, können Sie sie ohne Überschreibung drucken und mehr Farben für sie verwenden, d. h.:

- 2 Tuschen für den Hintergrund (2 Farben)
- 6 Farben für das Sprite mit Überschreibungen (3 Farben)
- 8 Farben für die anderen Sprites (8 Farben)

In diesem Fall können Sie insgesamt $2+3+8 = 13$ Farben verwenden!!!! Sie müssen lediglich das Überschreibungsflag bei dem Sprite aktivieren, das es benötigt, und es bei den anderen Sprites inaktiv lassen. Auf den Sprites, die das Überschreiben nicht verwenden, können Sie alle 13 Farben verwenden, auf dem Sprite mit Überschreiben würden Sie 3 und auf dem Hintergrund 2 verwenden.

Andere Beispiele sind möglich. Wenn z.B. Sprites mit Überschreibung 4 Farben benötigen, dann werden sie 8 Farben verwenden. Zusätzlich haben wir 2 Farben für den Hintergrund und die restlichen 6 Farben können 6 verschiedene Farben identifizieren, d.h. wir können insgesamt 12 Farben verwenden.

Verwenden Sie Sprites, deren Farbe vom Hintergrund abhängt (bedingte Farbe):

Es besteht darin, eine Palette zu verwenden, bei der wir nicht jede Farbe wiederholen,

sondern zwei verschiedene Farben verwenden. In diesem Fall, wenn der Hintergrund Null ist, haben wir ein Sprite mit einer Farbe und wenn der Hintergrund 1 ist, haben wir eine andere Farbe. Dies kann nützlich sein, um weiße Vögel zu zeichnen, die über einen blauen Himmel fliegen (Farbe 0), während rote Bären über einen braunen Boden laufen (Farbe 1). Mit anderen Worten, wir können mehr Farben verwenden, solange die Textur des Himmels oder des Bodens

Der Bär wird nicht allzu viele Farbwechsel haben, denn jedes Mal, wenn der Bär über ein blaues Hintergrundpixel fährt, sehen wir ein weißes Pixel, und wenn ein Vogel über ein braunes Hintergrundpixel fährt, sehen wir ein rotes Pixel. Schauen wir uns ein Beispiel an:

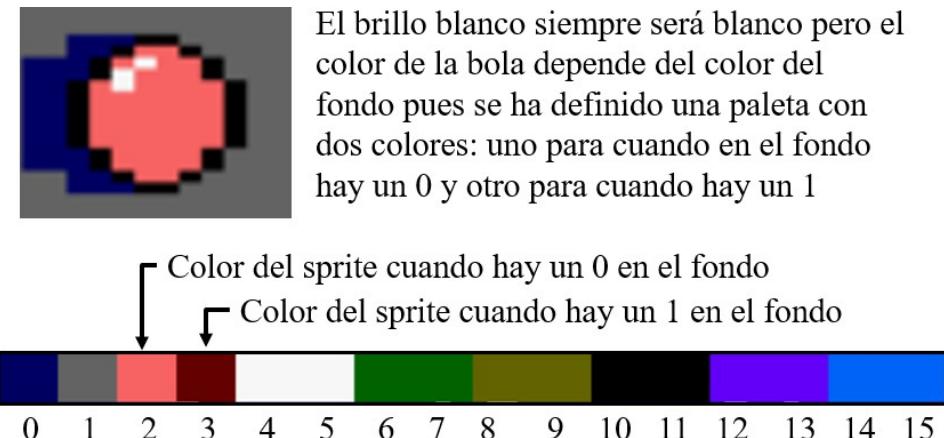


Abb. 42 Mit einer "bedingten" Farbe erstellte Sprites

Nachdem das Sprite mit bedingter Farbe erstellt wurde, können wir im folgenden Bild sehen, wie es wirkt, wenn es in zwei Bereichen des Bildschirms gedruckt wird, einer mit Hintergrund 0 und der andere mit Hintergrund 1.

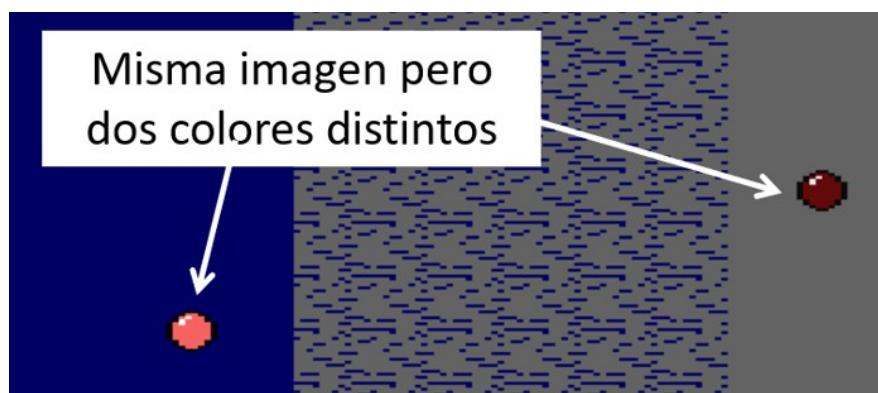


Abb. 43 "Bedingter" Farbeffekt

Verwenden Sie ein Sprite als Hintergrund:

Mit diesem Cheat können Sie bis zu 16 Hintergrundfarben mit Overwrite verwenden. Der Trick basiert auf der Tatsache, dass, wenn das Überschreiben auf einem Sprite aktiviert ist, alle mit "1" definierten Pixel den Wert des Pixels auf dem Bildschirm respektieren, unabhängig davon, ob es ein Pixel von einem echten Hintergrund oder von einem zuvor gedruckten Sprite ist. Wir drucken in jedem Zyklus sowohl das Sprite, das den Hintergrund bildet, als auch die Sprites, die überdruckt werden sollen. Oder zumindest in jedem Zyklus, in dem wir sie verschieben.

Das Sprite, das wir als Hintergrund verwenden, wird **ohne Aktivierung des Überschreibens** gedruckt, während es bei den Sprites, die wir darüber drucken, aktiviert ist. Der einzige Nachteil ist, dass wenn das Hintergrund-Sprite zu groß ist, es sehr lange dauert, bis es gedruckt wird und die Anzahl der fps des Spiels wird es nicht erlauben, ein Arcade-Spiel zu machen, obwohl es für Abenteuer-Spiele nützlich sein kann. Denken Sie auch daran, dass in 8BP die maximale Höhe eines Sprites 127

Zeilen beträgt. Die maximale Breite ist kein Problem, da sie ebenfalls 127 Bytes beträgt und der Bildschirm nur 80 Bytes breit ist.

Wir erstellen das Sprite, das wir oben drucken werden, umgeben von Einsen, ohne Nullen, da es den Hintergrund beim Bewegen nicht zurücksetzen wird. Jeder Zyklus druckt sowohl das Hintergrund-Sprite als auch unsere kleinen Sprites darüber. In diesem Beispiel haben wir eine Art Zeiger oder Pfeil gezeichnet, den man mit der Tastatur steuert, auf einem Hintergrund, der einen halben Bildschirm einnimmt, d.h. 8KB (80 Byte Breite x 100 Zeilen).

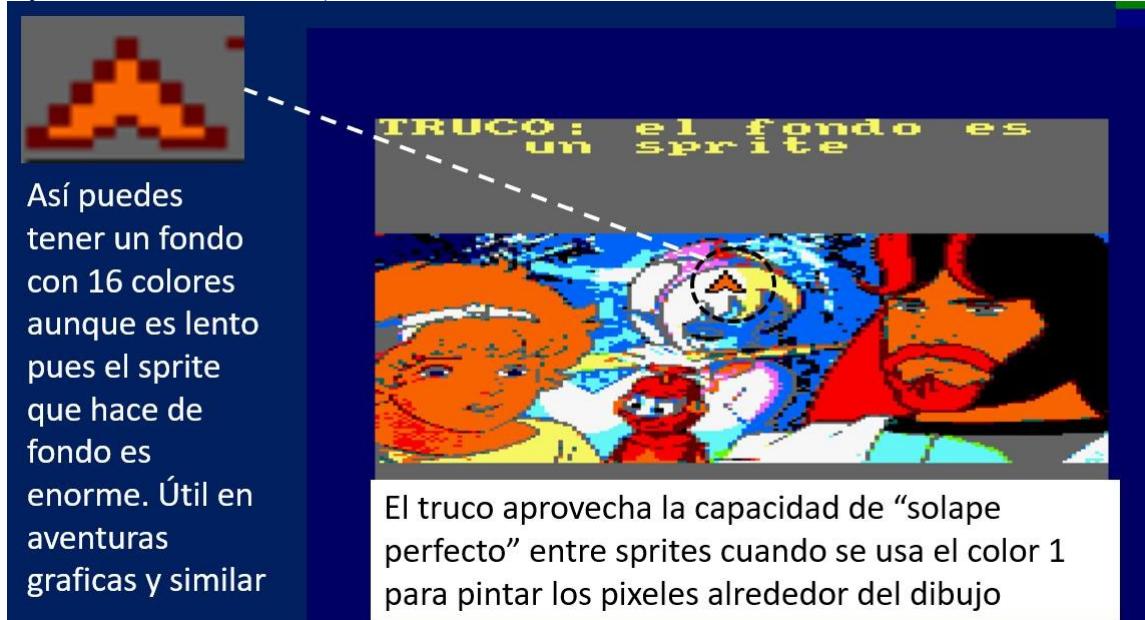


Abb. 44 Ein Sprite als Hintergrund

Da das Zeigersprite überschrieben wird, wird es die Auswirkungen des vorherigen Tricks (bedingte Farbe) erleiden, es sei denn, Sie definieren einige "doppelte" Farben, die diesen Effekt nicht erleiden. Das heißt, wenn Sie zum Beispiel eine Farbe wiederholen, um diesen Effekt auf einem Sprite nicht zu erleiden, dann haben Sie eine Palette von 15 verschiedenen Farben, nicht 16. Das heißt, Sie müssen irgendwie das Prinzip des ersten Tricks anwenden: ein Sprite ohne Überschreibung als Hintergrund und ein oder mehrere Sprites mit Überschreibung, die darüber gedruckt werden. Je mehr Farbe der Hintergrund hat, desto weniger Farbe haben die Überschreibesprites. Zum Beispiel (andere Kombinationen sind möglich) können Sie verwenden:

- **Hintergrund:** 2 Hintergrundfarben + 8 Farben + 3 wiederholte Farben = 13 Farben
- **Überdruckte Sprites:** 6 Druckfarben = 3 wiederholte Farben

Eine Möglichkeit, die Geschwindigkeit des "riesigen" Hintergrund-Sprites zu erhöhen, besteht darin, es in Streifen aufzuteilen. Zum Beispiel in 8 horizontale Sprites der Höhe 16. Horizontal gestreckte Sprites werden schneller gedruckt als vertikal gestreckte Sprites. In diesem Fall brauchen Sie nur den Streifen oder das Streifenpaar zu drucken, das Ihr Zeigersprite überschneidet.

8.5 Sprites mit Hintergrundbildern

Hintergrundbilder sind eine Funktion von 8BP V42. Die Idee ist, dass Bäume oder Häuser unter dem Flugzeug vorbeifliegen können, ohne dass das Flugzeug flackert. Selbst wenn das Flugzeug einen transparenten Druck hat und auf den Hintergrund gemalt ist, wird der Hintergrund, wenn er sich bewegt, das Sprite nicht berücksichtigen und ein Flackern verursachen. Mit dieser neuen Funktion können Sie Spiele mit

Bildlauf erstellen, bei denen Ihr Protagonist oder Ihr Schiff über Dinge hinwegfährt, ohne dass es zu einem **Flackern kommt**. Um dies besser zu verstehen, müssen Sie den Bildlaufmechanismus von 8BP verstehen, der im Folgenden erklärt wird.

Das 8BP-Demoset enthält eine Demo, die den Effekt der Verwendung von Hintergrundbildern in Ihrem Bildlauf demonstriert.



Hintergrundbilder von Häusern

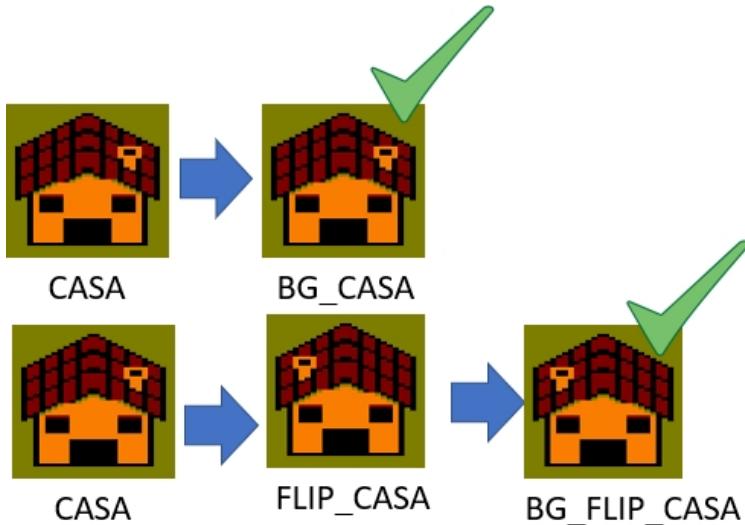
Ohne diese Fähigkeit konnten (vor der Version V42 von 8BP) Sprites ohne Flackern überschrieben werden, solange sich der Hintergrund nicht bewegte, aber die Bewegung des Hintergrunds (das Scrollen) verursachte unvermeidliches Flackern. Seit V42 ist es möglich, Hintergrundbilder zu erstellen und diese den Sprites zuzuordnen.

Was Sie tun müssen, ist, dem Sprite ein Hintergrundbild zuzuweisen, das in der Bilddatei innerhalb der Tags enthalten sein muss:

```
BEGIN_BG_IMAGES  
BG_HOME DW  
HOME  
BG_HOME_FLIP DW HOUSE_FLIP  
_END_BG_IMAGES
```

Das **CASA-Bild** ist ein Bild, das normalerweise mit einer Besonderheit erstellt wird: Es verwendet nur die Hintergrundfarben. Wenn also einem Sprite in einem Spiel das **BG_CASA-Bild** zugewiesen ist, wird ihm automatisch eine besondere Art von Transparenz zugewiesen, bei der nur die Bits, die die Hintergrundfarben darstellen, geändert werden und jedes Sprite über der Zeichnung respektiert wird, wodurch ein Flackern vermieden wird.

Das Verfahren zur Erstellung von Hintergrundbildern ist sehr einfach, aber auch sehr streng: Aus einem Bild (nur mit den Hintergrundfarben) können Sie ein Hintergrundbild erstellen. Wenn Sie ein gespiegeltes Bild verwenden möchten, müssen Sie das Bild zuerst spiegeln und dann das Hintergrundbild mit dem gespiegelten Bild erstellen.



Wenn Sie ein Hintergrundbild spiegeln wollen, müssen Sie es zuerst mit dem Abschnitt **FLIP_IMAGES** spiegeln und können es dann zum Erstellen eines Hintergrundbildes verwenden. Das muss die Reihenfolge sein und nicht andersherum. Das heißt, Sie können nicht ein Bild erstellen, ein Hintergrundbild damit erstellen und dann versuchen, ein Hintergrundbild zu spiegeln.

Die Hintergrundbilder werden, wenn sie einem Sprite zugewiesen werden, mit dieser speziellen Transparenz gedruckt, und es spielt keine Rolle, ob das Sprite das Transparenz-Flag in seinem Statusbyte zugewiesen hat oder nicht (wir werden jetzt sehen, was das ist).

WICHTIG: Hintergrundbilder sind teuer im Druck, und wenn man sie spiegelt, sind sie noch teurer. Wenn Ihr Spiel einen Bildlauf hat, versuchen Sie, sie für Elemente zu verwenden, über die Ihre Figur oder Ihre Feinde fliegen werden. Um den Bildlauf zu beschleunigen, sollten Sie zum Beispiel normale Bilder für Häuser oder Felsen verwenden, die an den Seiten des Bildlaufs erscheinen und nicht oft von Sprites überlagert werden.

8.6 Sprite-Attribut-Tabelle

Die Sprites werden in einer Tabelle gespeichert, die insgesamt 32 Sprites enthält. Jeder Eintrag in der Tabelle enthält alle Attribute des Sprites und belegt aus Performance-Gründen 16 Bytes, da 16 ein Vielfaches von 2 ist und somit der Zugriff auf jedes Sprite mit einer sehr günstigen Multiplikation möglich ist. Die Tabelle befindet sich an der Speicheradresse 27000, so dass man von BASIC aus mit PEEK und POKE darauf zugreifen kann, aber es gibt auch RSX-Befehle, um die Daten in dieser Tabelle zu manipulieren, wie |SETUPSP oder |LOCATESP.

Sprites haben eine Reihe von Parametern, von denen der erste das Byte mit den Statusflags ist. In diesem Byte steht jedes Bit für ein Flag und jedes Flag bedeutet eines, nämlich ob das Sprite bei der Ausführung bestimmter Funktionen berücksichtigt wird. Die folgende Tabelle fasst zusammen, was passiert, wenn sie aktiv sind (bei "1")

7 ROUTEALL lo ruta	6 Sobre- escritura	5 COLSPALL collider	4 MOVERALL lo mueve	3 AUTOALL lo mueve	2 ANIMALL lo anima	1 COLSP collided	0 PRINTSPALL lo imprime
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Abb. 45 Merker im Statusbyte

Um die Bedeutung dieser Flaggen zu verstehen, sollten wir uns einige Beispiele ansehen:

- **Bit 0:** Druckflagge: unser Charakter oder feindliche Schiffe haben sie aktiviert und in jedem Spielzyklus rufen wir |PRINTSPALL auf und sie werden alle zur gleichen Zeit gedruckt.
- **Bit 1:** Kollisionsflag: Eine Frucht oder eine Münze zum Beispiel kann kein Druckflag haben, aber ein Kollisionsflag. Dieses Flag bedeutet, dass ein "kollidierendes" Sprite mit dem Sprite kollidieren kann, das dieses Flag aktiviert hat.
- **Bit 2:** Kennzeichen für automatische Animation: es wird in |ANIMALL berücksichtigt. Im Fall der Figur empfehle ich, es zu deaktivieren, denn wenn ich stillstehe, muss ich den Rahmen nicht ändern.
- **Bit 3:** Flagge für automatische Bewegung. Bewegt sich nur, wenn AUTOALL aufgerufen wird, wobei seine Geschwindigkeit berücksichtigt wird. Nützlich bei Meteoren und Wächtern, die kommen und gehen.
- **Bit 4:** Flag für relative Bewegung. Alle Sprites mit diesem Flag bewegen sich gleichzeitig, wenn man "|MOVEALL, dy, dx" aufruft, was sehr nützlich bei der Formation von Schiffen und der Ankunft von Planeten ist. Es kann auch verwendet werden, um einen Bildlauf zu simulieren, wenn du deine Figur in der Mitte lässt und die Steuerung drückst, um Häuser oder umliegende Elemente zu scrollen. Es sieht dann so aus, als würde sich die Spielfigur durch ein Gebiet bewegen.
- **Bit 5:** Kollisions-Flag. Alle Sprites, bei denen dieses Flag aktiv ist, werden von der Funktion |COLSPALL bei der Erkennung einer möglichen Kollision mit dem Rest der Sprites berücksichtigt.
- **Bit 6:** Überschreibungsflagge: Wenn diese Flagge aktiv ist, kann sich das Sprite über einen Hintergrund bewegen und diesen beim Passieren zurücksetzen. Dies ist eine fortgeschrittene Option und erfordert die Verwendung einer speziell vorbereiteten Farbpalette, die in der Anzahl der Farben begrenzt ist. Das Überschreiben hat diesen "Preis".
- **Bit 7:** Pfadflag: Wenn dieses Flag gesetzt ist, ermöglicht der Befehl |ROUTEALL die zyklische Bewegung eines Sprites durch einen von Ihnen definierten Pfad, der durch eine Reihe von Segmenten definiert ist. Der Befehl |ROUTEALL weiß, in welchem Segment und an welcher Position sich jedes Sprite befindet, und wenn er einen Segmentwechsel erreicht, ändert er die Geschwindigkeit des Sprites entsprechend den Bedingungen des nächsten Segments.
|ROUTEALL bewegt das Sprite nicht, es ändert nur seine Geschwindigkeit. Um es zu bewegen, muss es in Verbindung mit |AUTOALL verwendet werden.

Beispiele für die Zuweisung von Statusbyte-Werten:

Typischer Feind: ein Sprite, das bei jedem Zyklus gedruckt wird, mit Kollisionserkennung mit anderen Sprites und Animation haben muss:

$$\text{Status} = 1(\text{Bit 0}) + 2 (\text{Bit1}) + 4 (\text{Bit 2}) = 7 = \&x0111$$

Ein Haus, das sich bewegt, wenn wir uns bewegen: ein Sprite, das bei jedem Zyklus gedruckt wird, aber ohne Kollisionserkennung und relative Bewegung.

$$\text{Status} = 1(\text{Bit 0}) + 0 (\text{Bit1}) + 0 (\text{Bit 2}) + 0 (\text{Bit 3}) + 16 (\text{Bit 4}) = 17 = \&x10001$$

Eine Bonusfrucht: Es handelt sich um ein Sprite, das nicht in jedem Zyklus gedruckt wird, sondern über eine Kollisionserkennung verfügt.

$$\text{Status} = 0 \text{ (Bit 0)} + 2 \text{ (Bit1)} = 2 = \&x10$$

Ein Schiff, das einer vordefinierten Trajektorie folgt. Es benötigt das Pfadflag, das Flag für automatische Bewegung, das Animationsflag, das Kollisionsflag und das Druckflag. Dieses Mal werde ich es direkt in Binärform eingeben. Wie Sie sehen können, habe ich Bit 7 auf 1 gesetzt, dann gibt es 3 Nullen, weil ich die Flaggen für Überschreiben, Kollision und relative Bewegung nicht gesetzt habe, und schließlich habe ich 4 aktive Flaggen gesetzt, die jeweils den Flaggen für automatische Bewegung, Animation, Kollision und Druck entsprechen.

status=10001111

Die Sprite-Attribut-Tabelle besteht aus 32 Einträgen zu je 16 Byte, beginnend bei Adresse 27000.

Der Grund für die 16 Bytes ist nichts anderes als die Leistungsfähigkeit, da die Berechnung der Adresse von Sprite N die Multiplikation mit 16 erfordert, was, da es sich um ein Vielfaches von 2 handelt, mit einem Shift durchgeführt werden kann. Dies ist nützlich für Operationen, die ein einzelnes Sprite betreffen. Bei Operationen, die die Sprite-Tabelle durchlaufen (wie |PRINTSPALL oder |COLSP), wird die Tabelle intern mit einem Index durchlaufen, zu dem 16 addiert wird, um von einem Sprite zum nächsten zu gelangen. Die Addition ist in diesem Fall am schnellsten.

Die Attribute, die jedes Sprite hat, sind:

Attribut	Byte	Länge (Bytes)	Bedeutung
Status	0	1	Byte mit den Statuskennzeichen für die Operationen PRINTSPALL, COLSPALL, ANIMALL, AUTOALL , MOVERALL, COLSPALL und ROUTEALL.
Y	1		Koordinaten Y [-32768..32768] Die Werte, die dem Inneren des Bildschirms entsprechen, sind [0..199].
X			X-Koordinate in Bytes[-32768..32768] die entsprechenden Werte innerhalb des Bildschirms sind [0..79].
Vy	5	1	Schritt zur automatischen Bewegung
Vx		1	Schritt zur automatischen Bewegung
Sequenz		1	Kennung der Animationssequenz [0..31]. Wenn es keine Sequenz hat, muss eine Null zugewiesen werden.
Standbild aus	8	1	Rahmennummer in der Reihenfolge [0..7].
Bild			Speicheradresse, an der sich das Bild befindet
Vorheriger Sprite	10		Interne Verwendung für den Sprite-Sortiermechanismus
Nächster Sprite			Interne Verwendung für den Sprite-Sortiermechanismus

Route	1	Pfadbezeichner, dem das Sprite folgen soll
-------	---	--

Die Speicheradresse, an der die Koordinaten der einzelnen Sprites gespeichert sind, kann wie folgt berechnet werden:

Y-Koordinaten-Adresse = $27000 + 16 \cdot N + 1$

X-Koordinaten-Adresse = $27000 + 16 \cdot N + 3$

Wenn man mit **POKE** auf diese Adressen zugreift, kann man ihren Wert ändern, obwohl man auch **|LOCATESP** verwenden kann.

Die 8BP-Bibliothek verwendet für die X-Koordinate keine "Pixel", sondern Bytes, so dass die X-Koordinate, die auf den Bildschirm fällt, im Bereich [0..79] liegt. Die Y-Koordinate wird in Zeilen dargestellt, so dass der darstellbare Bereich auf dem Bildschirm [0..200] ist. Wenn Sie ein Sprite außerhalb dieser Bereiche platzieren, sich aber ein Teil des Sprites auf dem Bildschirm befindet, wird die Bibliothek den Bereich beschneiden und den Teil malen, der zu sehen sein muss.

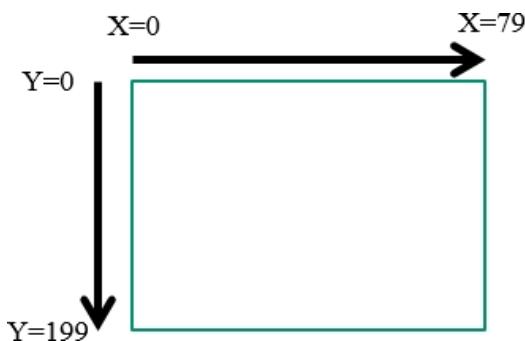


Abb. 46 Bildschirmkoordinaten

Die Adressen der Attribute der 32 Sprites können mit **PEEK** und **POKE** gehandhabt werden, obwohl die Animationssequenz und die Pfadzuweisung mehr Operationen beinhalten und wenn man sie ändern will, reicht ein **POKE** nicht aus, man muss **|SETUPSP** verwenden. Hier ist die Adressliste aller Attribute der 32 Sprites:

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	Imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Tabelle 3 Attributadressen der 32 Sprites

Der von jedem Sprite belegte Platz in der Tabelle beträgt 16 Byte. Wie Sie sehen können, sind die X- und Y-Koordinaten 2-Byte-Zahlen. Sprites akzeptieren negative Koordinaten, so dass man ein Sprite teilweise auf dem Bildschirm ausgeben kann, was den Eindruck erweckt, es käme Stück für Stück herein. Mit POKE kann man keine negativen Koordinaten setzen, aber mit |LOCATESP und auch mit |POKE, einer Version des POKE-Befehls von BASIC, die aber negative Zahlen (und 16-Bit-Zahlen) akzeptiert.

Es ist empfehlenswert, die Figur oder das Raumschiff an Position 31 zu platzieren (es gibt 32 Sprites, die von 0 bis 31 nummeriert sind). Wenn Ihr Raumschiff die Position 31 hat, wird es als letztes gedruckt, über den restlichen Sprites, falls diese sich überlappen.

8.7 Alle Sprites gedruckt und sortiert

In der 8BP-Bibliothek gibt es einen Befehl, der alle Sprites druckt, die gleichzeitig das Druck-Flag aktiviert haben. Dies ist der Befehl |PRINTSPALL

Dieser Befehl hat 4 Parameter, die Sie aber nur beim ersten Aufruf ausfüllen müssen, da er sich die Parameter bei den folgenden Aufrufen merkt und Sie sie nur dann erneut ausfüllen müssen, wenn Sie einen von ihnen ändern wollen. Dies ist nützlich, da die Parameterübergabe sehr zeitaufwendig ist (Sie können mehr als 1 ms sparen, wenn Sie die Parameterübergabe vermeiden).

Die Parameter sind:

|PRINTSPALL, <ordenini>, <ordenfin>, <animate>, <synchronism>.

- **Der Sync-Parameter** kann die Werte 0 oder 1 annehmen und gibt an, dass vor dem Drucken auf eine Unterbrechung des Bildschirms durchlaufs gewartet wird. Wenn Sie Geschwindigkeit wollen, empfehle ich das nicht. Wenn Sie mehr Gleichmäßigkeit wollen, vielleicht ja.
- **Der Animationsparameter** kann die Werte 0 oder 1 annehmen. Wenn er aktiv ist, wird vor dem Drucken jedes Sprites sein Animationsflag im Statusbyte überprüft und wenn es aktiv ist, wird es auf das nächste Bild umgeschaltet. Dies ist sehr nützlich bei Feinden, aber bei deinem Charakter vielleicht nicht, da du ihn vielleicht nur animieren willst, wenn du ihn bewegst und nicht in jedem Frame. WICHTIG: Die Animation wird vor dem Drucken durchgeführt, nicht nach dem Drucken. Das bedeutet, dass Sie, wenn Sie gerade eine Animationssequenz zugewiesen haben, das erste Bild dieser Sequenz nicht sehen werden.
- **Die Ordnungsparameter ("ordenini", "ordenfin")** geben die Anfangs- und End-Sprites an, die die Gruppe von Sprites definieren, die nach der "Y"-Koordinate geordnet sind und gedruckt werden sollen. Wenn wir zum Beispiel die Werte 0,0 zuweisen, werden sie nacheinander von Sprite 0 bis Sprite 31 gedruckt. Wenn wir 0,8 zuweisen, werden sie von 0 bis 8 geordnet (9 Sprites) und von 10 bis 31 nacheinander gedruckt. Wenn wir 0,31 setzen, werden alle Sprites in der Reihenfolge gedruckt. Bei der Einstellung 10,20 werden die Sprites nacheinander von 0 bis 9, dann geordnet von 10 bis 20 und schließlich nacheinander von 21 bis 31 gedruckt.

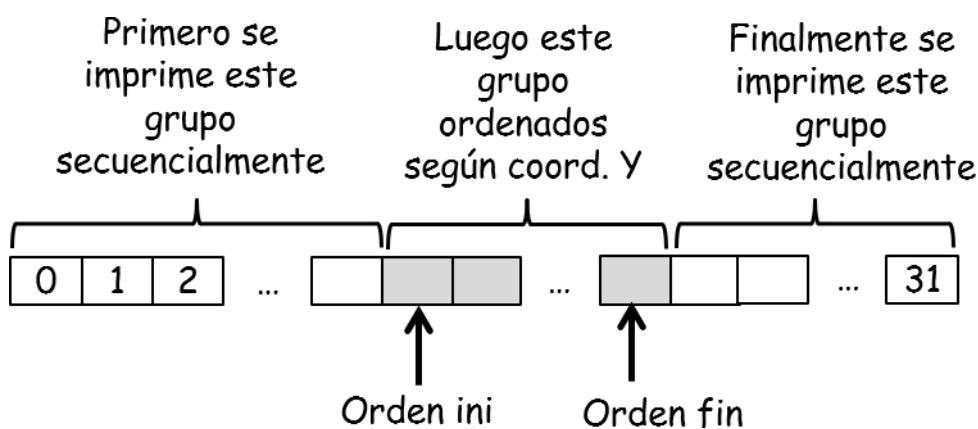


Abb. 47 Sequentielle und geordnete Sprite-Gruppen

Die Reihenfolge ist sehr nützlich für Spiele wie "Renegade" oder "Golden AXE", wo es notwendig ist, einen Tiefeneffekt zu erzeugen. Die Anordnung wird geschätzt, wenn es Überschneidungen zwischen Sprites gibt.



Abb. 48 Auswirkung der Sprite-Anordnung

- | PRINTSPALL, 0,0,1,0 : druckt nacheinander alle Sprites
- | PRINTSPALL, 0,31,1,0 : druckt alle Sprites in geordneter Weise
- | PRINTSPALL, 0,7,1,0: druckt 8 geordnet und den Rest sequentiell
- | PRINTSPALL, 16,24,1,0: 16 sequentiell, 9 sortiert und 7 sequentiell

Wenn der Parameter "ordenini" weggelassen wird, wird der zuletzt zugewiesene Wert berücksichtigt, oder Null, wenn noch nie ein Wert zugewiesen wurde. Wenn Sie einen der beiden Sortierparameter ändern, ist es außerdem ratsam, zuerst PRINTSPALL,0,0,0,0 auszuführen, damit die Sprites zunächst sequentiell neu sortiert werden, bevor sie mit einer neuen Konfiguration sortiert werden.

Das Drucken in der Reihenfolge ist rechenintensiver als das sequentielle Drucken. Wenn Sie nur 5 Sprites haben, die sortiert werden müssen, übergeben Sie z.B. 0,4 als Sortierparameter, nicht 0,31. Das Sortieren aller Sprites dauert etwa 2,5 ms, aber wenn Sie nur 5 Sprites sortieren, können Sie 2 ms einsparen. Vielleicht haben Sie eine Menge Sprites und es lohnt sich nicht, einige von ihnen zu sortieren, wie die Schüsse oder Sprites, von denen Sie wissen, dass sie sich nicht überschneiden.

Der zum Sortieren der Sprites verwendete Algorithmus ist eine Variante des sogenannten "Bubble"-Algorithmus. Obwohl man in der Literatur findet, dass der sogenannte "Bubble"-Algorithmus sehr ineffizient ist, wird dies von denjenigen gesagt, die über eine Liste von Zufallszahlen sprechen, die sortiert werden sollen. In diesem Fall sind die Sprites normalerweise fast geordnet und von einem Frame zum nächsten sind nur ein oder zwei Sprites ungeordnet, nicht mehr, da sich ihre Koordinaten "reibunglos" entwickeln. Der Algorithmus geht also durch die Liste der Sprites und wenn er ein paar ungeordnete Sprites findet, dreht er sie um und hört auf zu sortieren. Dieser Algorithmus ist extrem schnell, und auch wenn er nur ein paar Sprites auf einmal sortieren kann, ist er für diesen Anwendungsfall ideal. Nur in dem Fall, dass es zwei ungeordnete Sprites gibt und sie sich überlappen, wird es einen Frame geben, in dem wir sehen, dass eines der Sprites nicht in der richtigen Reihenfolge gedruckt wird, aber das wird im nächsten Frame korrigiert werden. Es ist nicht wahrnehmbar.

Manchmal möchten Sie vielleicht, dass die Sortierung bei jedem Bild vollständig ist. Das heißt, man möchte nicht bei jedem Aufruf von | PRINTSPALL ein paar Sprites sortiert haben, sondern sicher sein, dass sie alle sortiert sind. Die **8BP-Bibliothek** ermöglicht dies durch ihre vier Sortiermodi, die Sie durch den Aufruf des Befehls

|Das Folgende ist ein Ein-Parameter-PRINTSPALL (führen Sie ihn nur einmal aus, um den Sortiermodus festzulegen):

PRINTSPALL,0 : teilweise Sortierung nach Ymin PRINTSPALL,1 : vollständige Sortierung nach Ymin PRINTSPALL,2 : teilweise Sortierung nach Ymax PRINTSPALL,3 : vollständige Sortierung nach Ymax

Die Sortierung nach Ymax basiert auf der größten Y-Koordinate der Sprites, d.h. wo sich ihre Füße befinden und nicht ihre Köpfe. Wenn die Sprites gleich groß sind, kann eine Ymin-basierte Sortierung funktionieren, aber wenn die Sprites unterschiedlich hoch sind, möchten Sie vielleicht danach sortieren, wo sich die Füße der einzelnen Figuren befinden, und dafür müssen Sie den Sortiermodus 2 oder 3 verwenden.
Die Ymax-Sortiermodi sind langsamer, etwa 0,128 ms pro Sprite, und sollten daher nur bei Bedarf verwendet werden.

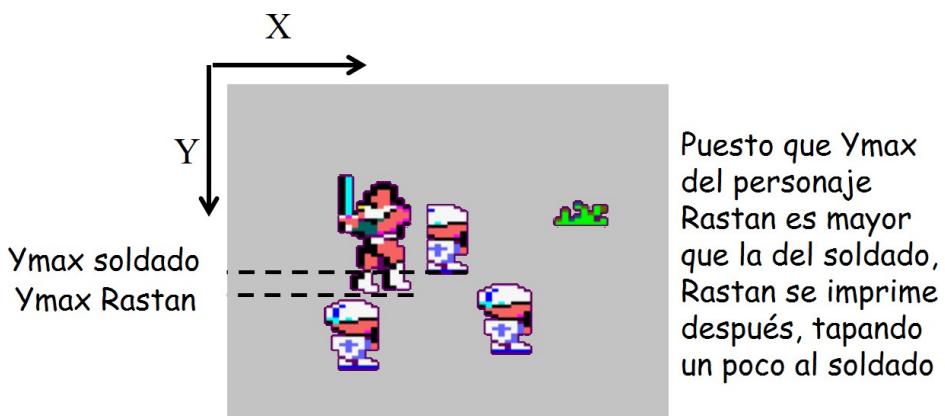


Abb. 49 Sortierung nach Ymax

Die vollständige Sortierung verbraucht nur wenig mehr als die teilweise Sortierung (etwa 0,3 ms). Das liegt daran, dass die Sprites kaum von einem Frame zum nächsten durcheinander gewürfelt werden, aber selbst diese 0,3 ms sind es wert, wenn möglich eingespart zu werden.

Erinnern Sie sich daran, dass der Befehl PRINTSPALL "Speicher" hat, so dass es ausreicht, ihn das erste Mal mit Parametern aufzurufen und von da an können wir PRINTSPALL ohne Parameter aufrufen, da der Befehl die Werte der Parameter, mit denen er aufgerufen wurde, "behält" und es nicht nötig ist, sie ihm zu übergeben, solange sie sich nicht ändern. Dies spart mehr als 1ms, da der Parser weniger arbeitet.

8.8 Kollisionen zwischen Sprites

Um zu überprüfen, ob dein Charakter oder dein Schuss mit anderen Sprites kollidiert ist, kannst du den Befehl

|COLSP, <Spruchnummer>, @Kollision%.

Dabei steht die Sprite-Nummer für das Sprite, das Sie testen möchten (Ihre Figur oder Ihr Schuss), und die Variable "collision" ist eine Integer-Variable, die zuvor definiert werden musste, z. B. durch Zuweisung eines Anfangswertes:

Kollision% = 0

|COLSP, 1, @Kollision%, 1, @Kollision%, 1, @Kollision%.

Die Variable "collision" wird mit dem ersten Sprite-Identifikator gefüllt, der mit dem Sprite kollidiert ist. Es können zwar mehrere Kollisionen auftreten, aber der Befehl liefert nur ein Ergebnis.

Intern geht die 8BP-Bibliothek die kollidierenden Sprites von 31 bis 0 durch (in umgekehrter Reihenfolge), und wenn sie das Kollisionsflag aktiv haben (Bit 1 des Statusbytes), dann wird geprüft, ob sie mit deinem Sprite kollidieren. Wenn es keine Kollision mit einem der Sprites gibt, wird die Variable collision% auf 32 gesetzt, und wenn doch, gibt sie die Nummer des Sprites zurück, das mit deinem Sprite kollidiert. Wenn zum Beispiel 4 und 12 kollidieren, gibt die Funktion eine 12 zurück, weil sie 12 vor 4 prüft.

Weder deine Figur noch dein Schuss dürfen das "collider"-Flag (Bit 1) und "collider" (Bit 5) gleichzeitig aktiv haben, sonst kollidieren sie immer... mit sich selbst! Das heißt, ein Sprite kann nicht gleichzeitig die Bits 1 und 5 aktiv haben.

Kollisionen zwischen Sprites sind eine aufwendige Aufgabe. Intern muss die Bibliothek den Schnittpunkt zwischen den Rechtecken berechnen, die jedes Sprite enthalten, um festzustellen, ob es Überschneidungen zwischen ihnen gibt. Um Berechnungen zu sparen, ist es am besten, Feinde in aufeinanderfolgenden Sprite-Positionen zu platzieren. Wenn zum Beispiel die Feinde, mit denen wir kollidieren können, die Sprites 15 bis 25 sind, können wir die Kollision so einstellen, dass nur diese Sprites überprüft werden. Dazu rufen wir die Kollision auf Sprite 32 auf, das nicht existiert. Dies teilt der 8BP-Bibliothek mit, dass es sich hierbei um Konfigurationsinformationen für den Befehl handelt, die den **Bereich der kollisionsfähigen Sprites** angeben, **der für jeden Collider gescannt werden soll:**

|COLSP, 32, <sprite initial>, <sprite final>.

Beispiel:

|COLSP, 32, 15, 25

Diese Optimierung ist nicht sehr bedeutsam, aber sie wird es, wenn COLSP mehrmals aufgerufen wird oder wenn der Befehl |COLSPALL verwendet wird, der COLSP intern mehrmals aufruft.

Eine weitere interessante Optimierung, die bei jedem Aufruf 1 Millisekunde einsparen kann, besteht darin, den Befehl anzuweisen, immer dieselbe BASIC-Variable zu verwenden, um das Ergebnis der Kollision zu hinterlassen. Zu diesem Zweck geben wir 33 als das Sprite an, das ebenfalls nicht existiert.

col%=0

|COLSP, 33, @col%, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%

Sobald diese beiden Zeilen ausgeführt wurden, belassen nachfolgende Aufrufe von COLSP das Ergebnis in der Variablen col, ohne dass es z. B. angegeben werden muss:

|COLSP, 23 : REM dieser Aufruf ist äquivalent zu |COLSP, 23, @col%.

WICHTIG: Die Kollisionsvariable im Befehl **|COLSP** ist nicht die, die im Befehl

|COLSPALL verwendet wird. Es handelt sich um unterschiedliche Variablen (es sei denn, Sie übergeben beiden Befehlen dieselbe Variable, um auf sie einzuwirken).

8.9 Anpassen der Sprite-Kollisionsempfindlichkeit

Es ist möglich, die Empfindlichkeit des COLSP-Befehls einzustellen und zu entscheiden, ob die Überlappung zwischen Sprites mehrere Pixel oder ein Pixel betragen muss, damit eine Kollision als erfolgt gilt.

Dies kann erreicht werden, indem man die Anzahl der Pixel (Pixel in Y-Richtung, Bytes in X-Richtung) der Überlappung in Y- und X-Richtung mit dem COLSP-Befehl festlegt und das Sprite 34 (das nicht existiert) angibt.

|COLSP, 34, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>

Die 8BP-Bibliothek verwendet keine "Pixel" in der X-Koordinate, sondern Bytes, so dass Sie berücksichtigen müssen, dass eine 1-Byte-Kollision tatsächlich 2 Pixel ist und dass dies die minimal mögliche Kollision ist, wenn Sie dx=0 setzen.

Bei der y-Koordinate arbeitet die Bibliothek mit Linien, so dass dy=0 eine einzelne Pixelkollision bedeutet.

Eine strenge Kollision, die für das Schießen nützlich ist, wäre eine, die keinen Spielraum toleriert und eine Kollision in Betracht zieht, sobald es eine minimale Überlappung zwischen Sprites gibt (1 Pixel in Y-Richtung oder ein Byte in X-Richtung).

|COLSP, 34, 0, 0, 0: rem Kollision, sobald es eine minimale Überlappung gibt

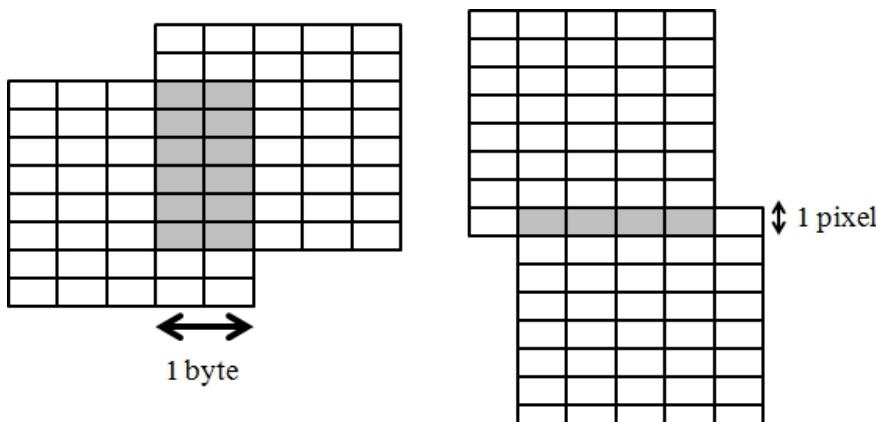


Abb. 50 Strenge Kollision mit COLSP, 34, 0, 0

Wenn wir jedoch ein Spiel im MODUS 0 machen, bei dem die Pixel breiter als hoch sind, ist es vielleicht angemessener, einen gewissen Spielraum auf der Y-Achse zu geben, aber keinen auf der X-Achse.

X. Zum Beispiel:

|COLSP, 34, 2, 0 : rem Kollision mit 3 Pixeln in Y und 1 Byte in X

Meine Empfehlung ist, dass, wenn es schmale oder kleine Schüsse, setzen Sie die Kollision zu (dy=1, dx=0), während, wenn es nur große Zeichen können Sie es mit mehr Spielraum (dy=2, dx=1) verlassen. Du solltest auch bedenken, dass, wenn deine Sprites einen Lösch-"Spielraum" um sich herum haben, um sich selbst zu löschen, dieser Spielraum nicht Teil der Kollisionsbetrachtung sein sollte, so dass es Sinn macht, dass

sowohl dy als auch dx ungleich Null sein sollten. In jedem Fall ist dies etwas, das Sie je nach Art des Spiels, das Sie machen, entscheiden werden.

8.10 Wer kollidiert und mit wem: COLSPALL

Mit der Funktion **|COLSP**, die wir bisher gesehen haben, ist die Kollisionserkennung eines Sprites mit allen anderen möglich. Wenn wir jedoch einen Mehrfachschuss haben, bei dem unser Schiff zum Beispiel bis zu 3 Schüsse gleichzeitig abfeuern kann, müssten wir die Kollision jedes einzelnen von ihnen und zusätzlich die unseres Schiffes erkennen, was zu 4 Aufrufen von **|COLSP** führt.

Denken Sie daran, dass jeder Aufruf die Parsing-Schicht durchläuft, so dass vier Aufrufe kostspielig sind. Hierfür gibt es einen sehr mächtigen Befehl: **|COLSPALL**.

Diese Funktion arbeitet in zwei Schritten: Zuerst müssen wir angeben, welche Variablen den Kollisionsrechner und das kollidierte Sprite speichern sollen. Die folgende Anweisung wird nur einmal ausgeführt und dient dazu, die Variablen zu definieren, auf denen wir die Ergebnisse erhalten werden, die vorher existieren müssen:

|COLSPALL, @collider%, @collider%, @collider%.

Und anschließend rufen wir in jedem Spielzyklus einfach die Funktion **|COLSPALL** ohne Parameter.

Die Funktion betrachtet diejenigen Sprites als "kollidierend", bei denen das Kollisionsflag im Statusbyte (Bit 5) auf "1" gesetzt ist, und als "kollidiert" diejenigen Sprites, bei denen das Kollisionsflag (Bit 1) im Statusbyte auf "1" gesetzt ist. Die kollidierenden Sprites sollten unser Schiff und unsere Schüsse sein, und die kollidierenden Sprites sollten all jene sein, mit denen wir kollidieren können: feindliche Schiffe und Schüsse, Berge, etc. Wie ich bereits sagte, dürfen bei einem Sprite nicht beide Bits gleichzeitig aktiv (=1) sein.

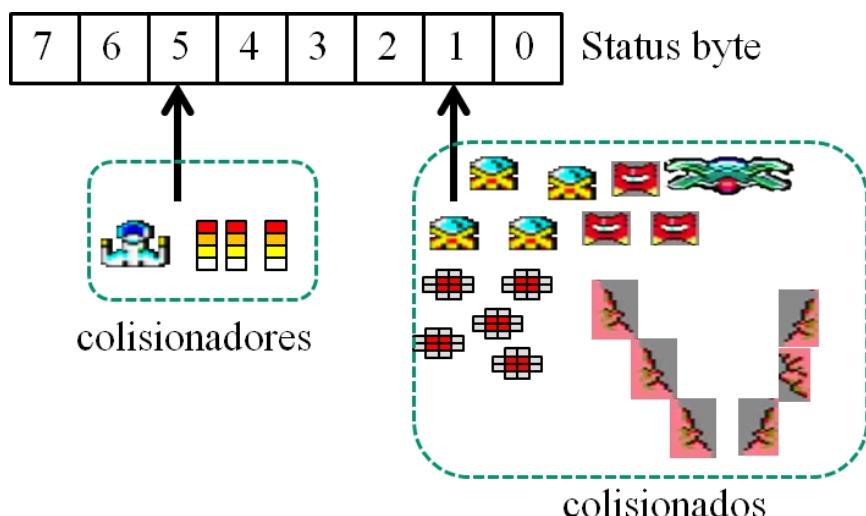


Abb. 51 Kollisionskörper gegen Kollisionskörper

Die Funktion **|COLSPALL** beginnt mit der Überprüfung von Sprite 31 (falls es sich um einen Collider handelt) und arbeitet sich bis zu Sprite 0 vor, wobei sie intern **|COLSP** für jedes Collider-Sprite aufruft. Für jeden Collider werden die Collider-Sprites ebenfalls in absteigender Reihenfolge durchlaufen (von 31 bis 0). Sobald es eine Kollision feststellt, unterbricht es seine Ausführung und gibt den Wert des Colliders und des Colliders zurück. Es ist daher wichtig, dass unser Schiff ein höheres Sprite hat als unsere Schüsse. Auf diese Weise können wir, wenn wir getroffen werden

werden wir sie entdecken, auch wenn wir im selben Moment einen Feind mit einem Schuss getroffen haben.

In jedem Spielzyklus kann nur eine Kollision erkannt werden, aber das ist genug. Es ist keine große Einschränkung, dass in jedem Frame nur ein Feind zu "explodieren" beginnen kann. Wenn man z.B. eine Granate wirft und eine Gruppe von 5 Soldaten betroffen ist, wird jeder Soldat in einem anderen Frame sterben, und nach 5 Frames werden alle explodieren. Wenn du |COLSPALL verwendest, explodieren nicht alle gleichzeitig, aber dein Spiel wird schneller, und das ist in einem Arcade-Spiel sehr wichtig.

Bei einem Aufruf von |COLSPALL mit einem einzigen Parameter,

|COLSPALL, <Anfangskollider>.

Kollisionskörper werden vom angegebenen Kollisionskörper -1 bis zum Sprite Null in absteigender Reihenfolge gescannt. Wenn Sie also mehr als eine Kollision pro Spielzyklus erkennen müssen, können Sie dies tun, indem Sie nacheinander **COLSPALL, <collider>** aufrufen, bis die Variable collider den Wert 32 annimmt.

Beispiel:

|COLSPALL, 7 : rem sucht nach Kollisionen aus Collider 6

8.10.1 So programmieren Sie einen Mehrfachschuss mit COLSPALL

Als Erstes müssen Sie entscheiden, wie viele aktive Schüsse es geben kann. Wenn Sie entscheiden, dass Ihr Schiff 3 Geschosse gleichzeitig abfeuern kann, dann sollten Sie

3 Sprite-Identifikatoren zum Auslösen reservieren. Als nächstes müssen Sie die Verzögerung anpassen

zwischen einem Schuss und dem nächsten, um zu verhindern, dass sich zwei Geschosse fast berühren, wenn Sie zu schnell schießen. Dies kann durch die Festlegung einer Mindestverzögerung zwischen den Schüssen erreicht werden.

Im folgenden Beispiel habe ich eine Verzögerung zwischen den Schüssen von 10 Spielzyklen eingestellt. Dazu wird durch Drücken der Leertaste (Taste 47) geprüft, ob seit dem letzten Schuss mindestens 10 Zyklen verstrichen sind. Wenn nicht, wird nicht geschossen.

```
130 ----- "Zyklus des Spiels".
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' Zeichenbewegungsroutine -----
    IF INKEY(47)=0 THEN IF delay<cycle-10 THEN delay=cycle:disp= 1+
    disp MOD 3 :|LOCATESP,10+disp,PEEK(27001)+8,PEEK(27003):
    |SETUPSP,10+disp,0,137: |SETUPSP,10+disp,15,3+dir
        WENN INKEY(27)=0 dir=0:|SETUPSP,0,6,1:'nach rechts
        DANN
    190 WENN INKEY(34)=0 dir=1:|SETUPSP,0,6,-1:'nach links gehen
        DANN
    193 Zyklus=Zyklus+1
    310 GOTO 150
```

Um das Sprite auszuwählen, das als Auslöser verwendet werden soll, wird die Anweisung

ausgeführt:

disp = 1+ disp Mod 3

Dadurch kann der Schuss abwechselnd die Werte 1,2,3,4 annehmen. Wenn die Sprites 20,21,22,23 sein sollen, kannst du **disp = 20 + disp Mod 3** verwenden.

denn wenn Sie 21 als Summe einsetzen, funktioniert es nicht (probieren Sie es selbst aus). Das ist die Sache mit der modularen Arithmetik.

Und nun kommen wir zu den Kollisionen und dem Vorteil der Verwendung von COLSPALL, die viel schneller ist als der mehrfache Aufruf von COLSP. Die einzigen wichtigen Empfehlungen sind:

- Unsere <Sprite-Nummer> soll höher sein als unsere Auslöser, so dass |COLSPALL vor der Aufnahme überprüfen.
- Dass wir |COLSP so konfiguriert haben, dass es nur die Liste der Sprites prüft, die Feinde sind und kollidieren müssen, indem wir |COLSP 32 verwenden, <Anfang>, <Ende>.

Bevor wir den Spielzyklus starten, definieren wir unsere Variablen:

collider=32:collided=32:|COLSPALL,@collider,@collided

In den Spielzyklus werden wir setzen:

**|COLSPALL:IF collider<32 THEN if collider=31 THEN GOSUB 300:goto 2000:
ELSE GOSUB 770**

Mit dieser Zeile wissen wir bereits, ob es eine Kollision gibt, denn dann wird die Variable "collider" <32 sein. Wenn sie gleich 31 ist, dann ist es unser Schiff (wir wurden getroffen) und wenn nicht, dann hat mit Sicherheit einer unserer Schüsse ein feindliches Schiff getroffen und wir gehen zu der Routine, die sich in Zeile 770 befindet, wo wir etwas wie dieses finden werden:

```
769' --- routinemäßiger Aufprallschuss ----  
770 |SETUPSP, collider, 9, img deleted:'Gelöschtes Bild der Datenbank  
zuordnen  
Schuss  
772 |PRINTSP, collider: 'wir haben die Aufnahme gelöscht  
775 |SETUPSP, collider, 0, 0: 'Auslösung deaktivieren  
777 if collided>=duros then return:'Feind unsterblich  
778 ' Sequenz 4 ist eine Animationssequenz von "Tod", einer  
Explosion  
780 |SETUPSP, kollidiert, 7, 4:|SETUPSP, kollidiert, 0, &x101: return
```

Kurz gesagt, mit einem einzigen Aufruf von COLSPALL wissen wir, wer kollidiert ist ("collider") und mit wem er kollidiert ist ("collided").

8.10.2 Wer kollidiert, wenn es mehrere Überschneidungen gibt

Es ist sehr wichtig zu beachten, dass 8BP die Collider von 31 bis 0 durchläuft und für jeden von ihnen die Colliderables von 31 bis 0. Wir müssen die Sprite-IDs mit unseren Sprites verknüpfen, je nachdem, wie wir sie kollidieren lassen wollen.

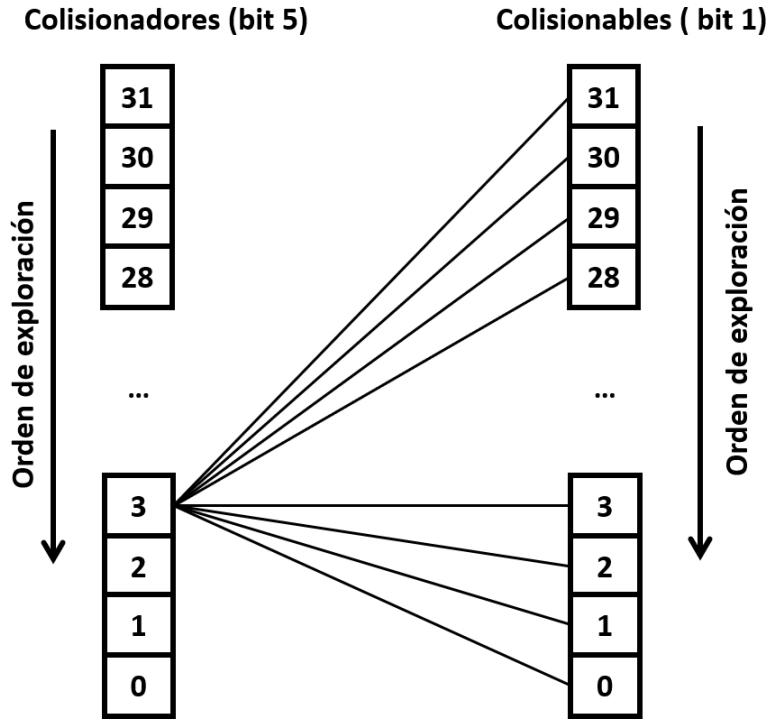


Abb. 52 Reihenfolge der Kollisionsprüfung

Wenn unser Sprite (Collider) mit zwei Sprites im selben Bereich kollidiert, können wir von vornherein wissen, mit welchem von ihnen wir kollidieren werden, was für bestimmte Spiele sehr nützlich ist.

Nehmen wir das Spiel "Frogger", in dem ein Frosch einen Fluss überqueren muss, indem er über Baumstämme springt. Wenn er mit einem Baumstamm kollidiert, wird er nicht sterben, aber wenn er mit einem Fluss kollidiert, wird er sterben.

Um es zu programmieren, können wir 4 Flüsse (4 unbewegliche, längliche Sprites) setzen und auf ihnen einige Sprites bewegen, die Stämme sind. Wir könnten die folgende Verteilung einrichten:

- Der Frosch ist Sprite 31 (ich nehme an, es ist ein Collider).
- Die Stämme sind die Sprites 4, 5, 6, 7 (Kollisionen).
- Die Flüsse sind die Sprites 0, 1, 2, 3 (kollisierbar).

Bei Flüssen kann das Druckkennzeichen deaktiviert werden, so dass sie mit dem Frosch kollidieren können (Kollisionskennzeichen), ohne gedruckt zu werden. Das heißt, wir würden ihnen den Status =2 geben



Abb. 53 Im Falle einer Überlappung sind wir an der Kollision des Stammes interessiert.

Da sich die Baumstämme und der Fluss überschneiden, kollidiert der Frosch, sobald er auf einen Baumstamm klettert, mit beiden, aber der Baumstamm wird zuerst überprüft, da er eine höhere ID hat. Der Kollisionsbefehl wird nur die Kollision mit dem Baumstamm erkennen. Wenn der Frosch dagegen über das Wasser springt, erkennt der Kollisionsbefehl den Fluss, und nachdem wir in **BASIC** die Variable "collided" ausgewertet haben und sehen, dass es sich um einen Fluss handelt, würden wir feststellen, dass unser Frosch sterben muss.

8.10.3 Erweiterte Verwendung des Statusbytes bei Kollisionen

Manchmal möchten Sie vielleicht, dass ein Feind Sie nicht tötet, indem er mit Ihrem Charakter kollidiert, weil er sich in einem besonderen Zustand befindet oder weil er in einem Spiel, das vorgibt, dass sich Feinde nähern, einfach weit weg ist.

Unter bestimmten Umständen kann es erforderlich sein, eine "Markierung" auf dem Sprite zu verwenden, um anzusehen, dass das Sprite trotz eines Zusammenstoßes nicht sterben oder Sie nicht töten soll.

Dazu kannst du die Flags, die du nicht im Sprite verwendest, in deiner Kollisionsroutine überprüfen.

Sehen wir uns ein Beispiel für einen Feind an, den wir unschädlich machen wollen, weil er weit entfernt ist (Simulation von 3D) oder eine geringe Energie hat, aber mit uns zusammenstößt.

Angenommen, dein Charakter kollidiert und der Feind ist ein Collider. Sie können erzwingen, dass die Kollision nur mit Sprites größer als 25 erkannt wird und wenn der Feind die Nummer 20 ist (zum Beispiel) wird es nicht in der Lage sein, mit sich selbst zu kollidieren.

|COLSP,32,25,31

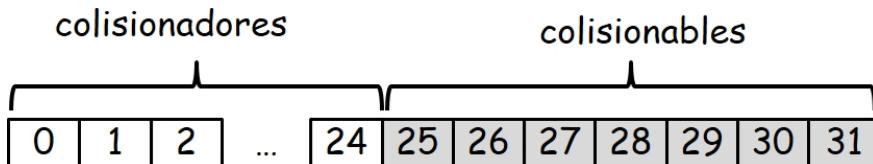


Abb. 54 Wirkung von **COLSP,32,25,31**

Als "harmlosen" Indikator verwenden wir das Kollisionsflag und setzen es auf 1. Wir setzen also das Flag des Feindes (Sprite 20) in seinem Statusbyte auf 1.

Nehmen wir nun an, der Befehl |COLSPALL erkennt eine Kollision und lässt das Ergebnis in collider und collided.

|COLSPALL

Wenn Koller<32 dann GOSUB 100
<Anweisungen>.

100 Fernkollisionsroutine

110 dir=27000 + collider*16 :rem byte adresse status des colliders
wenn PEEK (dir) und 2 THEN RETURN: rem harmlos, wenn bit
collided=1

130 <Ein Feind ist kollidiert, der nicht ungefährlich ist>.

Die Anweisung "**if PEEK (dir) and 2**" ist diejenige, die das kollidierte Bit prüft, da 2

im Binärformat 00000010 ist, nur die Position dieses Bits im Sprite-Status.
Wenn der Feind nicht mehr harmlos ist, setzen wir seine Kollisionsflagge einfach auf 0
und bei einer weiteren Kollision wird er uns töten.
Diese Technik ist auch mit jedem anderen nicht verwendeten Flag möglich.

8.11 Tabelle der Animationssequenzen

Animationen bestehen in der Regel aus einer geraden Anzahl von Einzelbildern, obwohl dies keine strikte Regel ist. Denken Sie zum Beispiel an eine einfache Charakteranimation mit nur zwei Bildern: Beine öffnen und schließen. Das sind zwei Frames. Denken Sie nun an eine erweiterte Animation mit einer dazwischen liegenden Bewegungsphase. Das bedeutet, dass Sie die Sequenz: geschlossen-zwischen-geöffnet-geöffnet-zwischen und wieder von vorne beginnen. Wie Sie sehen können, ist es eine gerade Zahl, also 4

8BP-Animationssequenzen sind Listen mit 8 Einzelbildern, mehr können sie nicht haben, obwohl man immer kürzere Sequenzen machen kann.

Die Frames einer Animationssequenz sind die Speicheradressen, an denen die Bilder, aus denen sie bestehen, zusammengesetzt werden, und sie können unterschiedlich groß sein, obwohl sie normalerweise gleich groß sind. Wenn Sie in der Mitte der Sequenz eine Null eingeben, bedeutet das, dass die Sequenz beendet ist.

Obwohl es vor V33 einen RSX-Befehl namens **|SETUPSQ gab**, um Sequenzen aus BASIC zu erstellen, habe ich ihn in V33 entfernt, da seine Verwendung komplexer ist als das Definieren von Sequenzen aus der Datei sequences.asm und ich den Befehl **|SETUPSQ** tatsächlich nie in einem meiner Spiele verwendet habe.

Schauen wir uns ein Beispiel für die Erstellung einer Sequenz in der Datei sequences.asm an.

```
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0,0,0,0
```

Bevor ich Ihnen erkläre, wie Sie einem Sprite eine Sequenz zuweisen, möchte ich Sie daran erinnern, wie Sie Sprites Bilder zuweisen können: Seit Version V26 von 8BP gibt es die Möglichkeit, eine Liste von Bildern (ihre Bezeichnungen) in eine Liste namens IMAGE_LIST in Ihrer images_your_game.asm-Datei aufzunehmen. Damit können Sie die Bilder von BASIC aus mit einem Index anstelle einer Speicheradresse referenzieren. Auf diese Weise müssen Sie nicht jedes Mal, wenn Sie assemblyn, Speicheradressen nachschlagen. Dies gilt für die Anweisung:

|SETUPSP, #, 9, <Adresse>.

Das Beispiel zeigt eine Animationssequenz mit 3 verschiedenen Frames, aber um sie vor dem erneuten Start flüssig zu machen, müssen Sie das "mittlere" Frame noch einmal durchlaufen (beachten Sie, dass das zweite und das vierte Frame das gleiche sind), so dass es am Ende 4 Frames sind:



und zurück zum
Anfang

Abb. 55 Ablauf einer Animation

Wenn Sie eine Sequenz mit mehr als 8 Bildern erstellen möchten, können Sie einfach zwei Sequenzen hintereinander schalten und, wenn das Zeichen das letzte Bild der ersten Sequenz erreicht hat, den Befehl |SETUPSP verwenden, um ihm die zweite Sequenz zuzuweisen.

Animationssequenzen werden ab Adresse 33600 zusammengestellt, und Sie können bis zu 31 Animationssequenzen definieren (ab Nummer 32 gelten sie nicht als Sequenzen, sondern als "Makrosequenzen", was ein anderes Konzept ist). Jede Sequenz wird durch eine Nummer im Bereich [1..31] identifiziert. Die Sequenz Null gibt es nicht, sie wird

verwendet, um anzuzeigen, dass ein Sprite keine Sequenz hat.

Um einem Sprite eine Sequenz zuzuweisen, verwenden Sie den Befehl SETUPSP mit Parameter 7:

 SETUPSP, <sprite_id>, 7, <Folgenummer>
--

Mit diesem Befehl wird die Animationssequenz dem Sprite im entsprechenden Feld der Sprite-Tabelle zugewiesen, und im Feld Frame-ID wird eine Null gesetzt. Außerdem wird das entsprechende Bild dem ersten Bild der Sequenz zugewiesen. Wenn Sie |ANIMALL vor dem Drucken oder |PRINTSPALL mit Animationsflag verwenden, springen Sie vor dem Drucken zu Bild 1, selbst wenn SETUPSP die Animation bei Bild 0 platziert. Normalerweise ist das kein Problem, aber im Falle einer "Todessequenz" (mehr dazu später), bei der das erste Bild zum Beispiel das Sprite löschen soll, möchten Sie vielleicht nicht direkt zu Bild 1 springen.

1. In diesem Fall kann ein einfacher Trick darin bestehen, Bild Null in der Definition der Todesfolge zu wiederholen. Auf diese Weise stellen Sie sicher, dass das Bild sichtbar ist. Eine andere Möglichkeit besteht darin, das Animationskennzeichen zu entfernen und die Sequenz nach dem Drucken mit |ANIMASP zu animieren.

Jede Sequenz speichert 8 Speicheradressen, die den 8 Rahmen entsprechen, d.h. 16 Bytes, die von jeder Sequenz verbraucht werden.

Ihre Animationssequenzdatei könnte etwa so aussehen:

```
=====
bis zu 31 Animationssequenzen
=====
muss eine feste Tabelle und keine variable Tabelle sein
Jede Sequenz enthält die Adressen der zyklischen Animationsbilder.
Jede Sequenz besteht aus 8 Bildspeicheradressen.
gerade Zahl, da die Animationen normalerweise eine gerade Zahl sind
eine Null bedeutet das Ende der Sequenz, obwohl immer 8 Wörter
ausgegeben werden.
;nacheinander
Wenn eine Null gefunden wird, wird ein neuer Anfang gemacht.
wenn es keine Null gibt, beginnt es nach Bild 8 von vorne.
Wenn einem Sprite die Sequenz Null zugewiesen wird, hat es keine
Sequenz.
Wir beginnen mit der Sequenz 1
----- zeichenanimationssequenzen montoya-----
SEQUENZEN_LISTE
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0,0,1
dw MONTOYA_UR0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0,2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0,3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0,4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0,5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0,6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0,0,0,7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0,8

----- Animationssequenzen der Soldaten ----- dw
SOLDIER_R0,SOLDIER_R2,SOLDIER_R1,SOLDIER_R2,0,0,0,0,9
dw SOLDIER_L0,SOLDIER_L2,SOLDIER_L1,SOLDIER_L2,0,0,0,0,10
```

8.12 Spezielle Animationssequenzen

In 8BP sind verschiedene Arten von Animationssequenzen verfügbar:

Art der Sequenz	Beschreibung
Normale Sequenz	Die Abfolge der Animationsbilder wird immer wieder wiederholt. Sie enden entweder in einer Bildrichtung oder in einer
	eine Null, wenn wir eine Folge von weniger als 8 Bildern machen wollen
Todesfolge	Das letzte Bild der Sequenz ist eine "1". Dies weist 8BP an, den Status des Sprites auf Null zu setzen. Normalerweise ist das Bild vor der "1" ein Löschbild.
Sequenz beenden	Nachdem die Sequenz durchlaufen wurde, hat das Sprite keine Animation mehr. Das letzte Bild der Sequenz ist eine "2". Damit wird 8BP angewiesen, das Animationsflag aus dem Sprite-Status zu entfernen.
Verkettete Sequenz	Nach dem Durchlaufen der Sequenz gibt der letzte Rahmen die Kennung der nächsten Sequenz an, die dem Sprite. Mit dieser Art von Sequenzen können Sie Animationssequenzen erstellen, die länger als 8 Bilder sind.
Makro-Folgen	Sie ermöglichen es, automatisch eine Sequenz entsprechend der Geschwindigkeit des Sprites zuzuordnen.

8.12.1 Todesfolgen

Mit der 8BP-Bibliothek können Sie "Todessequenzen" erstellen, d. h. Sequenzen, bei denen das Sprite am Ende der Sequenz in einen inaktiven Zustand übergeht. Dies wird durch eine einfache "1" als Wert der Speicheradresse des letzten Frames angezeigt. Diese Sequenzen sind sehr nützlich, um gegnerische Explosionen zu definieren, die mit |ANIMA oder

|ANIMALL. Nachdem du sie mit deinem Schuss getroffen hast, kannst du ihnen eine Todesanimationssequenz zuordnen und in den folgenden Spielzyklen durchlaufen sie die verschiedenen Animationsphasen der Explosion und wenn sie die letzte erreichen, gehen sie in den inaktiven Zustand über und drucken nicht mehr. Dieser inaktive Zustand wird automatisch hergestellt, d.h. du musst nur die Kollision deines Schusses mit den Gegnern überprüfen und wenn er mit einem von ihnen kollidiert, änderst du den Zustand mit SETUPSP, so dass er nicht mehr kollidieren kann und weist ihm die Animationssequenz des Todes zu, ebenfalls mit SETUPSP.

Wenn Sie eine Todessequenz verwenden, achten Sie darauf, dass das letzte Bild vor der "1" ein völlig leeres Bild ist, damit keine Spuren der Explosion zu sehen sind.

Beispiel für eine Todesfolge (beachten Sie, dass sie eine "1" enthält):

dw EXPLOSION_1,EXPLOSION_2,EXPLOSION_3,1,0,0,0,0,0,0

Ein interessanter Effekt ist das wiederholte Durchlaufen mehrerer Bilder, bevor es mit einem schwarzen Bild endet, das dazu dient, das Bild zu löschen.

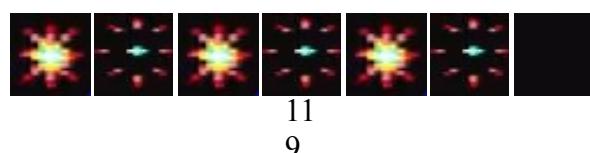


Abb. 56 Todesfolge

Die "1" erscheint nun an achter Stelle:

**dw EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2,
ExPLOSION_3,1**

Denken Sie daran, dass bei Verwendung des Befehls |PRINTSPALL mit aktivem Animationsflag zuerst animiert und dann gedruckt wird, so dass das erste Bild der Todessequenz nicht zu sehen ist. Ein einfacher Trick, um die Sequenz zu zeigen, besteht darin, sie zweimal zu wiederholen.

8.12.2 Sequenzen beenden

Endsequenzen gibt es seit Version V42. Sie ermöglichen es, einem Sprite eine Animationssequenz zuzuordnen, die nur einmal ausgeführt werden soll. Am Ende der Sequenz hat das Sprite keine Animationsflagge mehr. Die anderen Flaggen seines Zustands werden beibehalten.

Wir müssen einfach eine "2" in das letzte Bild der Sequenz setzen.

Hier sind zwei Beispiele. Sobald die Sequenz das Bild "2" erreicht, wird das Sprite nicht mehr animiert.

```
SEQUENCES_LIST
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,2,0,0,0,0
dw IMG1, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7,2 ;1
;2
```

8.12.3 Verkettete Sequenzen

Verkettete Sequenzen gibt es seit Version 8BP V42. Sie ermöglichen es, Sequenzen, die länger als 8 Bilder sind, durch Aneinanderreihen von Sequenzen zu erstellen.

Der Mechanismus ist einfach. Einfach das letzte Bild der Sequenz muss die Nummer der Sequenz sein, die Sie als nächstes zuweisen möchten.

Wie Sie sich vorstellen können, können Sie keine Sequenz "1" oder "2" zuweisen, da diese Zahlen "sterben" oder "enden" bedeuten. Das heißt, Sie können Sequenzen ab der Nummer 3 aneinanderreihen.

In diesem Beispiel habe ich die Sequenzen 4 und 5 so verkettet, dass nach der einen die andere zugewiesen wird und umgekehrt. Das ist so, als hätte man eine Sequenz mit 14 Bildern. Wir könnten mehr Sequenzen aneinanderreihen und die Animation viel länger machen. Jede hinzugefügte Sequenz besteht aus 7 weiteren Bildern (nicht 8), da wir das letzte Bild benötigen, um die nächste Sequenz anzuzeigen.

Sie können sie auch kürzer machen und mit Nullen bis zu 8 Bildern auffüllen, aber die Sequenznummer muss vor jeder Null erscheinen, denn wenn eine Null auftritt, läuft die Animation weiter.

```
SEQUENZEN_LISTE
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0,0,0,1
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U2,MONTOYA_U1,0,0,0,0,0,0,0,2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0,0,0,3
dw IMG11, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7, 5 ;4
dw IMG18, IMG9, IMG10, IMG11, IMG12, IMG13, IMG14, 4 ;5
```

8.12.4 Animation Makro-Sequenzen

Dies ist eine "erweiterte" Funktion, die ab Version V25 der 8BP-Bibliothek verfügbar ist. Eine "Makrosequenz" ist eine Sequenz, die aus mehreren Sequenzen besteht. Jede der

konstituierenden Animationssequenzen ist die Animation, die in einer bestimmten Richtung ausgeführt werden soll. Die Richtung wird durch die Geschwindigkeitsattribute des Sprites bestimmt, die sich in der Sprite-Tabelle befinden. Wenn wir also ein Sprite animieren mit |ANIMALL aufruft, wird es automatisch seine Animationssequenz ändern, ohne dass wir etwas tun müssen (Sie müssen |ANIMALL eigentlich nicht aufrufen, da |PRINTSPALL dies bereits intern tut, wenn Sie einen Parameter setzen).

Die Makrosequenzen sind mit 32 beginnend nummeriert. Es ist sehr wichtig, die Sequenzen innerhalb der Makrosequenz in der richtigen Reihenfolge zu platzieren, d.h. die erste Sequenz sollte für den Fall sein, dass das Zeichen stillsteht, die nächste für den Fall, dass das Zeichen nach links geht ($Vx < 0$, $Vy = 0$), die nächste für den Fall, dass es nach rechts geht ($Vx > 0$, $Vy = 0$), usw., und zwar in der folgenden Reihenfolge (seien Sie vorsichtig, denn es ist leicht, einen Fehler zu machen):



Abb. 57 Reihenfolge der Sequenzen in einer Makrosequenz

Ist die der stillstehenden Position zugewiesene Sequenz Null, wird sie einfach mit der zuletzt zugewiesenen Sequenz animiert.

Die Makro-Sequenzen müssen in der Datei sequences_yourgame.asm angegeben werden; ein Beispiel dafür finden Sie unten:

```
=====
Animationssequenzen
=====
SEQUENZEN_LISTE
dw NAVE,0,0,0,0,0,0,0,0,1
dw JOE1,JOE2,0,0,0,0,0,0,0,2 UP
JOE dw JOE7,JOE8,0,0,0,0,0,0,3 DW
JOE dw JOE3,JOE4,0,0,0,0,0,0,0,4 R
JOE dw JOE5,JOE6,0,0,0,0,0,0,0,5 L
JOE

_MAKRO_SEQUENZEN
-----MAKRO-SEQUENZEN -----
sind Gruppen von Sequenzen, eine für jede Richtung. die Bedeutung ist:
still, links, rechts, oben, oben-links, oben-rechts, unten, unten-links,
unten-rechts
Die Nummern sind von 32 an aufwärts nummeriert
db 0,5,4,2,5,4,3,5,4;Sequenz 32 enthält die Sequenzen des Soldaten Joe
=====
```

Mit dieser Sequenzdefinition können wir ein einfaches Spiel erstellen, mit dem wir "Joe" auf dem Bildschirm bewegen können, ohne seine Animationssequenz zu steuern. Wir weisen die Sequenz

32 und die Geschwindigkeit zu ändern, wird der Befehl **|ANIMA (der** aus dem Programm heraus aufgerufen wird)

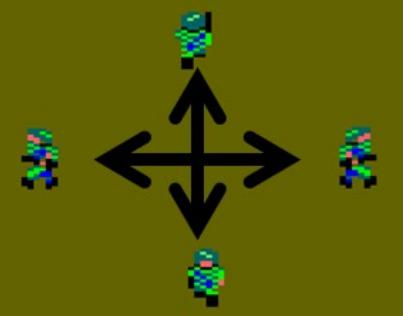
|PRINTSPALL) ist für die Änderung der Animationssequenz zuständig, wenn seine Geschwindigkeit eine Richtungsänderung bedeutet. Um das Sprite zu bewegen, müssen wir **|AUTOALL** aufrufen, da das Drücken der Steuerelemente nicht seine Koordinaten, sondern seine Geschwindigkeit ändert und **|AUTOALL die** Koordinaten des Sprites entsprechend seiner Geschwindigkeit aktualisiert.

10 SPEICHER 24999 20 MODUS 0:TINTE 0,12 30 BEI PAUSE GOSUB 280 40 ANRUF &6B78 50 DEFINT a-z 111 x=36:y=100 120 SETUPSP,31,0,0,&X1111 130 SETUPSP,31,7,2: SETUPSP,31,7,32	
---	--

```

140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,200
161 |PRINTSPALL,0,1,0
190 'Beginn des Spielzyklus
199 vy=0:vx=0
200 IF INKEY(27)=0 THEN vx=1: GOTO 220
210 IF INKEY(34)=0 THEN vx=-1
220 IF INKEY(69)=0 THEN vy=2: GOTO 240
230 IF INKEY(67)=0 THEN vy=-2
240 |SETUPSP,31,5,vy,vx
250 |AUTOALL:|PRINTSPALL
270 GOTO 199
280 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

```



Beachten Sie, dass ich die Sequenz für den Fall, dass sich das Zeichen nicht bewegt, nicht definiert habe. An dieser Stelle habe ich eine Null in die Makrosequenz gesetzt. Das bedeutet, dass Sie, wenn das Zeichen stillsteht, nicht wissen, welche Sequenz Sie zuweisen müssen, da es keine "letzte" Sequenz gibt. Deshalb weise ich die Sequenz 2 zu, bevor ich die 32 zuweise, damit ich sicher bin, dass das Zeichen bereits eine Sequenz hat, auch wenn es stillsteht.

130 SETUPSP, 31, 7, 7, 2: SETUPSP, 31, 7, 32
--

9 Ihr erstes einfaches Spiel

Sie haben nun das Wissen, um einen ersten Schritt bei der Erstellung von Videospielen zu versuchen. Betrachten wir dazu ein einfaches Beispiel eines Soldaten, den Sie steuern werden, indem Sie ihn nach links und rechts auf dem Bildschirm laufen lassen.

Nehmen wir an, wir haben einen Soldaten mit Hilfe von SPEDIT bearbeitet. Und wir haben seine Animationssequenzen erstellt, die in der Datei "**sequences_mygame.asm**" definiert und mit den Bezeichnern 9 und 10 für die linke bzw. rechte Bewegungsrichtung versehen wurden.

Die beiden Animationssequenzen wurden aus der Datei sequences.asm erstellt.

```
10 SPEICHER 24499
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restore default palette just in case'.
26 ink 0,0:'schwarzer Hintergrund
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
40 |SETLIMITS,12,80,0,186: ' die Grenzen des Spielbildschirms festlegen
50 x=40:y=100:' Zeichenkoordinaten 51|SETUPSP,0,0,1:'Zeichenstatus 52|SETUPSP,0,7,9:'Animationssequenz zum Start
zugewiesen 53|LOCATESP,0,y,x:'Platzieren des Sprites (ohne
es noch zu drucken)

60 'Spielzyklus
70 gosub 100
80 |PRINTSPALL,0,0
90 zu 60

99 ' Zeichenbewegungsroutine -----
100 IF INKEY(27)=0 THEN IF dir<>>0 THEN |SETUPSP,0,7,9:dir=0:return ELSE
|ANIMA,0:x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN IF dir<>>1 THEN |SETUPSP,0,7,10:dir=1:return ELSE
|ANIMA,0:x=x-1
120 |LOCATESP,0,y,x
130 RÜCKKEHR
```

Mit dieser Liste haben Sie bereits ein Minispiel, mit dem Sie einen Soldaten steuern und ihn horizontal laufen lassen können. Beachten Sie, dass, wenn Sie beim Laufen nach links den Mindestwert der mit |SETLIMITS eingestellten Grenze überschreiten, die Figur "abgeschnitten" wird und nur der Teil angezeigt wird, der sich innerhalb des erlaubten Spielbereichs befindet.

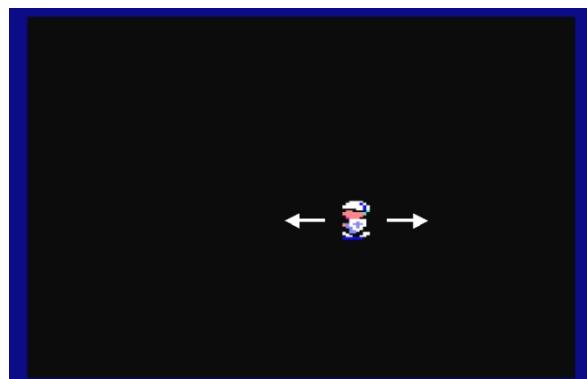


Abb. 58 Ein einfaches Spiel

9.1 Also, lasst uns springen! Boing, boing!

Im obigen Beispiel bewegt sich unsere Figur nur von links nach rechts. Wenn wir einen Sprung programmieren wollen, können wir das tun, indem wir die vertikale Flugbahn in einem

BASIC-Array. Später werden wir einen besseren Weg sehen (mit 8BP-Pfaden), aber für jetzt wird es als Beispiel dienen.

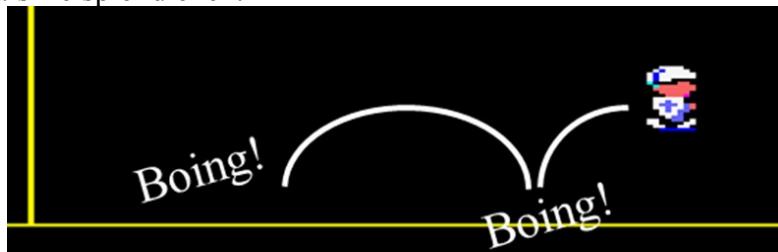


Abb. 59 unsere Figur kann springen

Die Sprungkurve wird für die Y-Koordinate definiert. Zuerst geht es 5 Zeilen auf einmal nach oben, dann 4, dann 3 usw., bis der Nullpunkt erreicht ist. An diesem Punkt kehrt sich die Bewegungsrichtung um und es geht zunächst eine Linie nach unten, dann 2, dann 3 usw. bis zur 5.

Damit die Puppe beim Auf- und Abstieg keine Spuren hinterlässt, brauchen wir eine Zeichnung der aufsteigenden Puppe mit 5 schwarzen Linien darunter, um sich beim Aufsteigen selbst zu löschen, und ein weiteres Bild mit 5 schwarzen Linien darüber, um abzusteigen. In diesem Fall sind es die Bilder 22 und 23, um nach rechts zu springen, und 24,25, um nach links zu springen.

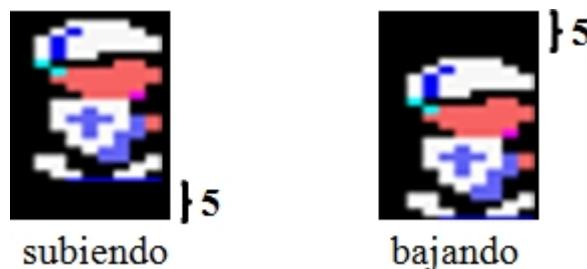


Abb. 60 Bild nach oben und unten

Am Scheitelpunkt des Sprungs muss das obere Bild in das untere Bild umgewandelt werden, aber zuerst müssen wir 5 Zeilen auf einmal nach oben gehen, denn wenn Sie beide Bilder vergleichen, werden Sie feststellen, dass es so ist, als ob Sie 5 Zeilen nach unten gehen, wenn Sie direkt von einem zum anderen gehen.

```

10 SPEICHER 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 BEI PAUSE GOSUB 2800
30 CALL &BC02:'restore default palette just in case'.
40 INK 0,0: 'schwarzer Hintergrund
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
80 |SETLIMITS,12,80,0,186: ' wir setzen die Bildschirmgrenzen
90 x=40:y=100:' Koordinaten des Zeichens
100 |SETUPSP,0,0,1:' Zeichenstatus
110 |SETUPSP,0,7,9:'Animationssequenz beim Start zugewiesen
120 |LOCATESP,0,y,x:'platziert das Sprite (ohne es zu drucken)
121 DIM jump(24):' jump data
122 for i=-5 to 5: k=k+1:skip(k)=i: k=k+1: skip(k)=i: next:
skip(11)=-5: skip(23)=5

125 PLOT 1,150:DRAW 640,150: plot 92,150:draw 92,400:'Boden und
Wand
126 |MUSIC,0,0,5: 'Musik beginnt zu spielen
130 ----- "Zyklus des Spiels".

```

```
150 |LOCATESP,0,y,x:|PRINTSPALL,0,0  
GOSUB 170  
160 GOTO 130:' Ende des Spielzyklus
```

```

170 ' Zeichenbewegungsroutine -----
171 IF jump =0 THEN IF INKEY(67)=0 THEN jump=1:|SETUPSP,0,9,DIR*2+22
180 IF INKEY(27)=0 THEN x=x+1:if jump=0 then IF dir<>0 THEN
|SETUPSP,0,7,9:dir=0:x=x-1:RETURN ELSE |ANIMA,0:GOTO 210
190 IF INKEY(34)=0 THEN x=x-1:if jump=0 then IF dir<>1 THEN
|SETUPSP,0,7,10:dir=1:x=x+1:RETURN ELSE |ANIMA,0
210 wenn Sprung=0 dann RETURN
260 Sprungtraining -----
270 IF jump=11 THEN |SETUPSP,0,9,DIR*2+23 ELSE IF jump=23 THEN
y=y+Sprung(jump):jump=0:|SETUPSP,0,7,DIR+9:return
280 y=y+Sprung(springen)
Sprung=Sprung+1
310 return
2800 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

```

Wie Sie in der Liste sehen können, ist die Variable "jump" gleich 1, wenn Sie die Taste "Q" drücken. In diesem Moment wird die Logik des Dummys kompliziert, da eine IF-Anweisung ausgeführt werden muss, um das Bild zu ändern, wenn es den Zenitpunkt erreicht, und es ist auch notwendig, die Y-Koordinate des Dummies und die Variable "jump" zu aktualisieren.

Später werden wir uns ansehen, wie man das mit einer fortgeschritteneren Technik macht, nämlich mit Sprite-"Routen". **Die programmierbaren Routen von 8BP bieten eine effizientere Methode, um diese Art von Dingen zu tun, so dass Sie Ihren Charakter viel schneller springen sehen werden.** Routen ermöglichen es dir, eine Flugbahn (einen Sprung, einen Kreis usw.) auszuführen, ohne dass du die Koordinaten jedes Mal überprüfen musst. Sie können auch den Zustand eines Sprites in der Mitte einer Route ändern, sein zugehöriges Bild, seine Sequenz oder sogar seine Route, indem Sie verschiedene Routen miteinander verknüpfen.

10 Bildschirmsätze: Layout oder "Kachelplan".

10.1 Definition und Verwendung des Layouts

Häufig sollen Ihre Spiele aus einer Reihe von Bildschirmen bestehen, auf denen die Spielfigur in einem Labyrinth Schätze sammeln oder Feinden ausweichen muss. In solchen Fällen ist es unverzichtbar, eine Matrix zu verwenden, in der Sie die einzelnen Blöcke des "Labyrinths" oder des so genannten "Layouts" des Bildschirms definieren. Manchmal wird dieses Konzept auch als "Kachelkarte" bezeichnet ("Kachel" ist das englische Wort für "tile").

In der **8BP-Bibliothek** gibt es dafür einen einfachen Mechanismus, der auch eine Kollisionsfunktion enthält, mit der man überprüfen kann, ob sich die Spielfigur in einen Bereich bewegt hat, der von einem "Stein" besetzt ist. Dieser Mechanismus wird "Layout" genannt. In 8BP wird ein Layout durch eine Matrix von 20x25 "Blöcken" von 8x8 Pixeln definiert, die besetzt oder nicht besetzt sein können. Das heißt, es gibt so viele Blöcke, wie es Zeichen auf dem Bildschirm im Modus 0 gibt.

Um ein Layout auf dem Bildschirm zu drucken, haben Sie den Befehl:

|LAYOUT, <y>, <x>, @string\$

Diese Routine druckt eine Reihe von Sprites, um das Layout oder "Labyrinth" eines jeden Bildschirms zu erstellen. Die Matrix oder "Layout-Map" wird in einem Speicherbereich gespeichert, der 8BP verarbeitet, so dass Sie beim Drucken von Blöcken **nicht nur auf den Bildschirm drucken, sondern auch den vom Layout belegten Speicherbereich (20x25 Byte) füllen**, wobei jedes Byte einen Block darstellt.

Die y,x-Koordinaten werden im Zeichenformat übergeben,

d.h. y nimmt die Werte [0,24] an.

x nimmt die Werte [0,19] an.

Die von der Funktion |LAYOUT gedruckten Blöcke sind aus Zeichenketten aufgebaut, und jedes Zeichen entspricht einem Sprite, das es geben muss. So entspricht der "Z"-Block dem Bild, das dem Sprite 31 zugewiesen ist, der "Y"-Block dem Bild, das dem Sprite 30 zugewiesen ist, und so weiter.

Die zu druckenden Sprites werden mit einer Zeichenkette definiert, deren Zeichen (32 mögliche) eines der Sprites darstellen, die dieser einfachen Regel folgen, wobei die einzige Ausnahme das Leerzeichen ist, das das Fehlen eines Sprites darstellt.

Zeichen	Sprite-ID	ASCII-Code
" "	KEINE	
";"	0	59
"<"	1	
"="		
">"		
"?"		63
"@"	5	
"A"		65
"B"		
"C"	8	67
"D"	9	
"E"	10	69
"F"		70
"G"		71
"H"		
"T"		73
"J"		
"K"		75
"L"		
"M"		
"N"		
"O"		79
"P"	21	80
"Q"		81
"R"	23	82
"S"		
"T"	25	84
"U"	26	85
"V"		86
"W"		87
"X"	29	88
"Y"	30	
"Z"	31	90

Tabelle 4 Zeichen- und Sprite-Zuordnung für den Befehl |LAYOUT

Der @string ist eine String-Variable. Sie können die Zeichenkette nicht direkt übergeben.
Das heißt, es wäre illegal, etwas zu tun wie:

|LAYOUT, 1, 0, "ZZZ YYY".

Das ist richtig:

String\$ = "ZZZ YYY".

|LAYOUT, 1, 0, @string\$

Achten Sie darauf, dass die Zeichenkette nicht leer ist, sonst kann Ihr Computer abstürzen!
Außerdem müssen Sie der String-Variable das Symbol "@" voranstellen, um

dass die Bibliothek zu der Speicheradresse gehen kann, an der die Zeichenkette gespeichert ist, und diese durchlaufen kann, um die entsprechenden Sprites nacheinander zu drucken.

Sie sollten beachten, dass Leerzeichen kein Sprite bedeuten, d.h. an den Positionen, die den Leerzeichen entsprechen, wird nichts gedruckt. Wenn sich an dieser Stelle vorher etwas befand, wird es nicht gelöscht. Wenn Sie etwas löschen wollen, müssen Sie ein 8x8-Lösche-Sprite definieren, bei dem alles aus Nullen besteht.

Obwohl Sie die Sprites verwenden, um das Layout zu drucken, können Sie direkt nach dem Drucken die Sprites mit |SETUPSP umdefinieren und ihnen Bilder von Soldaten, Monstern oder was auch immer Sie wollen zuweisen, d.h. das Layout "verlässt" sich auf den Sprite-Mechanismus, um zu drucken, aber es begrenzt nicht die Anzahl der Sprites, weil Sie alle 32 Sprites haben, um zu sein, was immer Sie wollen, direkt nach dem Drucken des Layouts.

Um Kollisionen mit dem Layout zu erkennen, gibt es die Funktion COLAY, die mit einer variablen Anzahl von Parametern verwendet werden kann.

|COLAY,<ASCII-Schwelle>, @collision , <Spruchnummer>
|COLAY, @collision , <Spruchnummer>
|COLAY, <Seitennummer>, <Seitennummer>.
|COLAY

Bei einem Sprite und abhängig von seinen Koordinaten und seiner Größe, findet diese Funktion heraus, ob es mit dem Layout kollidiert und warnt Sie durch die Kollisionsvariable, die vorher definiert werden muss.

Der Parameter **<ASCII threshold>** ist optional und wird verwendet, damit der Befehl ASCII-Codes unterhalb dieser Schwelle nicht als Kollisionen betrachtet. Standardmäßig beträgt er 32 (was dem Leerzeichen entspricht). Um dies zu verstehen, muss man die Korrespondenz zwischen ASCII-Werten und Sprites berücksichtigen, die in der vorherigen Tabelle gezeigt wurde. Wenn wir zum Beispiel den Schwellenwert auf 69 ("E"-Code, Sprite 10) setzen, dann sind die Sprites 9, 8, 7, 6, 5, 4, 3, 2, 1 und 0 nicht "kollisionsfähig", d.h. wenn unser Zeichen sie überfährt, wird die Kollision einfach nicht erkannt.

Es ist nur einmal erforderlich, COLAY mit dem Schwellenwertparameter aufzurufen, da die nachfolgenden Aufrufe diesen Schwellenwert bereits berücksichtigen.

Beispiel für die Verwendung:

col%=0

|COLAY, @col%, 20: REM dies ist ein Beispiel mit spriteID=20

Wenn Sie den COLAY-Befehl ohne Parameter aufrufen, berücksichtigt er die zuletzt verwendeten Parameter. Auf diese Weise können Sie sich die Parameterübergabe sparen und den Befehl um 0,5 ms beschleunigen.

Wenn es keine Kollision gibt, nimmt die Variable den Wert Null an. Eine Kollision tritt auf, wenn das Sprite mit einem beliebigen Layout-Element außer Leerzeichen (" ") kollidiert, dessen ASCII-Code 32 ist. Wenn der Schwellenwert verwendet wird, tritt

eine Kollision auf, wenn das Layout-Element einen ASCII-Wert hat, der größer ist als der von Ihnen festgelegte Schwellenwert.

Wir werden ein Beispiel für die Erstellung eines Layouts und das Verschieben eines Zeichens innerhalb des Layouts sehen, wobei seine Position korrigiert wird, wenn es kollidiert ist.

Zwei letzte Überlegungen zum Befehl |COLAY:

- Der Befehl |COLAY wird nicht von der Einstellung der Kollisionsempfindlichkeit des Sprites beeinflusst (konfigurierbar mit |COLSP,34, dy, dx). Die Einstellung der Kollisionsempfindlichkeit wirkt sich nur auf die Befehle |COLSP und |COLSPALL aus.
- Der Befehl |COLAY berücksichtigt nicht die tatsächliche Größe der Bilder, die als "Kacheln" oder "Blöcke" des Layouts verwendet werden. Das heißt, alle mit dem Befehl |LAYOUT angegebenen Blöcke werden als 8x8 Pixel des Modus 0 (4 Byte x 8 Zeilen) betrachtet, auch wenn Sie größere Bilder einfügen.

10.2 Beispiel für ein Spiel mit Layout

Wir werden das im vorigen Kapitel vorgestellte Gameplay ein wenig weiterentwickeln, was die Steuerung der Figur betrifft. Diesmal nehmen wir Montoya als Beispiel, der 8 Animationssequenzen hat, von denen sich jede in eine andere Richtung bewegt. Den Animationssequenzen wurde eine Nummer von 1 bis 8 zugewiesen.

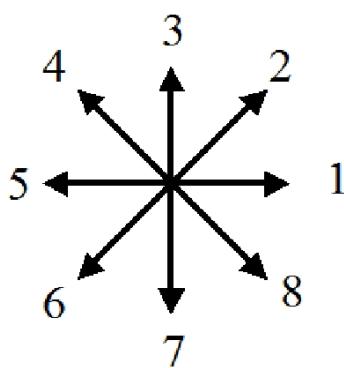


Abb. 61 Verwendung des Layouts in einem Spiel

In der Zeichensteuerungsroutine haben wir eine Kollision mit dem Layout vorgesehen. Je nachdem, in welche Richtung wir uns bewegen, ändern wir die "neuen" Koordinaten (yn , xn) und rufen die Kollisionsfunktion mit Layout |COLAY,0 auf, um zu prüfen, ob Sprite 0 (unsere Figur) kollidiert ist. Wenn es kollidiert ist, korrigieren wir die Koordinaten (eine oder beide), um es in einer kollisionsfreien Position zu belassen, bevor wir es erneut drucken.

```

10 SPEICHER 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 CALL &bc02:'restore default palette just in case'.
26 ink 0,0:'schwarzer Hintergrund
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
40 |SETLIMITS,0,80,0,200: ' die Grenzen des Spielbildschirms festlegen
50 dim c$(25):for i=0 to 24:c$(i)=" " :next
100 c$(1)= "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz".
110 c$(2)= "z          z"
           c$(3)= "z          z"
125 c$(4)= "z          z"
           c$(5)= "z  zzz  zzzz  zzz  z"

```

140 c\$(6)= "Z ZZZ ZZZZ ZZZ Z"
150 c\$(7)= "Z ZZZ ZZZZ ZZZ Z"

```

160 c$(8)= "Z" Z"
170 c$(9)="Z" Z"
190 c$(10)="Z" Z"
195 c$(11)="Z" ZZZZZ ZZZZZZZ Z"
              ZZZ
200 c$(12)="Z" ZZZZZ ZZZZZZZ Z"
              ZZZ
210 c$(13)="Z" Z"
220 c$(14)="Z" Z"
230 c$(15)="Z" Z"
240 c$(16)="ZZZZZZZZZZZZZZ ZZZZ"
              ZZZZZZZZZZZZZZ
              c$(17)="Z" Z"
260 c$(18)="Z" Z"
270 c$(19)="Z" Z"
271 c$(20)="Z" Z"
272 c$(21)="Z" Z"
273 c$(22)="Z" Z"
274 c$(23)="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
              ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
              gosub 550: ' druckt das Layout
310 xa=40:xn=xa:ya=150:yn=ya:' Koordinaten des Zeichens
311 |SETUPSP,0,0,&x111: ' Kollisionserkennung mit Sprites und Layout
312 |SETUPSP,0,7,1: ' Reihenfolge = 1
320 |LOCATESP,0,ya,xa: 'Platziere das Zeichen (ohne es zu drucken)
325 cl%=0:'Kollisionsvariable deklarieren, ausdrücklich ganze Zahl (%)'

330 '----- Spielzyklus -----
340 gosub 1500:'read keyboard and character movement routine 350|PRINTSPALL,0,0
360 goto 340

550 'Routinedruck Layout-----
560 FOR i=0 TO 23:|LAYOUT,i,0,@c$(i):NEXT
570 RÜCKKEHR

1500 ' Zeichenbewegungsroutine -----
1510 IF INKEY(27)<0 GOTO 1520
1511 IF INKEY(67)=0 THEN IF dir<>>2 THEN |SETUPSP,0,7,2:dir=2:GOTO 1533 ELSE
|ANIMA,0:xn=xa+1:yn=ya-2:GOTO 1533
1512 IF INKEY(69)=0 THEN IF dir<>>8 THEN |SETUPSP,0,7,8:dir=8:GOTO 1533 ELSE
|ANIMA,0:xn=xa+1:yn=ya+2:GOTO 1533
1513 IF dir<>>1 THEN |SETUPSP,0,7,1:dir=1:GOTO 1533 ELSE |ANIMA,0:xn=xa+1:GOTO
1533
1520 IF INKEY(34)<0 GOTO 1530
1521 IF INKEY(67)=0 THEN IF dir<>>4 THEN |SETUPSP,0,7,4:dir=4:GOTO 1533 ELSE
|ANIMA,0:xn=xa-1:yn=ya-2:GOTO 1533
1522 IF INKEY(69)=0 THEN IF dir<>>6 THEN |SETUPSP,0,7,6:dir=6:GOTO 1533 ELSE
|ANIMA,0:xn=xa-1:yn=ya+2:GOTO 1533
1523 IF dir<>>5 THEN |SETUPSP,0,7,5:dir=5:GOTO 1533 ELSE |ANIMA,0:xn=xa-1:GOTO
1533
1530 IF INKEY(67)=0 THEN IF dir<>>3 THEN |SETUPSP,0,7,3:dir=3:GOTO 1533 ELSE
|ANIMA,0:yn=ya-4:GOTO 1533
1531 IF INKEY(69)=0 THEN IF dir<>>7 THEN |SETUPSP,0,7,7:dir=7:GOTO 1533 ELSE
|ANIMA,0:yn=ya+4:GOTO 1533
1532 RÜCKKEHR
1533 |LOCATESP,0,yn,xn:ynn=yn:|COLAY,@cl%,0:IF cl%=0 THEN 1536
1534 yn=ya:|POKE, 27001,yn:|COLAY,@cl%,0:IF cl%=0 THEN 1536
1535 xn=xa: yn=ynn:|POKE, 27001,yn:|POKE, 27003,xn:|COLAY,@cl%,0:IF cl%=1 THEN
yn=ya:|POKE,27001,yn
1536 ya=yn:xa=xn
1537 RÜCKKEHR

```

10.3 So öffnen Sie ein Tor im Layout

Wenn du willst, dass deine Figur einen Schlüssel nimmt und ein Tor öffnet oder generell einen Teil der Anlage entfernt, um den Zugang zu ermöglichen, musst du zwei Schritte machen:

1) Definieren Sie ein 8x8-Wisch-Sprite, bei dem alles aus Nullen besteht. Mit |LAYOUT drucken es an den gewünschten Stellen

2) Wenn Sie dann erneut |LAYOUT verwenden, drucken Sie die Stellen aus, die Sie gelöscht haben. Das Layout wird dann mit dem Zeichen " " an diesen Stellen belassen und die Kollisionsfunktion mit dem Layout ergibt Null.

In dem Spiel "Mutant Montoya" wird diese Technik verwendet, um das Schlosstor zu öffnen und auch die Tore, die zur Prinzessin führen.



Abb. 62 Layoutänderung beim Abheben des Schlüssels

Das folgende Beispiel veranschaulicht das Konzept, indem es ein Tor an den Koordinaten (10, 12) mit einer Größe von 2 Blöcken öffnet, indem es einen Schlüssel aufnimmt, der mit Sprite 16 definiert ist.

Sobald man den Schlüssel aufhebt, öffnet sich das Tor und der Schlüssel wird deaktiviert, um die Kollision mit ihm nicht mehr auszuwerten, d.h. der Befehl |COLSP gibt ab dem Moment, in dem man den Schlüssel aufhebt, eine 32 zurück, wenn man wieder mit ihm kollidiert.

Wenn du die Figur nach dem Öffnen des Tores an die Stelle bewegst, an der sich das Tor befand, wird die Kollision mit dem Layout 0 ergeben.

```
'----- dieser Teil befindet sich innerhalb der
Logikschiife ---- 6410 |PRINTSPALL,1,0
6411 |COLSP,@cs%,0:IF cs%<32 THEN IF cs%>=15 then gosub 6500 (...
weitere Anweisungen . . . .)

'----- gate opening routine -----
6499 ' Prüfen Sie, ob Sie mit dem Schlüssel kollidieren, der Sprite 16
ist 6500 borra$="MM":spaces$="      ':' das Lösch-Sprite wurde als "M"
definiert (M ist das Sprite 18 in der "Sprache" des Befehls |LAYOUT)
6501 if cs%=16 then |LAYOUT,10,12,@delete$ : |LAYOUT,10,12,@spaces$:
|SETUPSP,16,0,0,0
6502 Rückgabe
```

10.4 Ein Puzzlespiel: LAYOUT mit einem Hintergrund

Als Nächstes sehen wir uns ein Beispiel an, bei dem Layout und Überschreibung verwendet werden und bei dem ein ASCII-Schwellenwert festgelegt wird, der es dem Befehl |COLAY ermöglicht, Kollisionen mit den Hintergrundelementen nicht zu berücksichtigen. Konkret handelt es sich bei dem Hintergrundelement um den Buchstaben "Y", der dem Sprite id= 30 entspricht, und der ASCII-Wert des "Y" ist 89.



Abb. 63 Layout mit einem Hintergrundmuster und Überschreiben

Wie im Beispiel zu sehen ist, muss COLAY nur einmal mit dem Schwellenwertparameter aufgerufen werden, da die nachfolgenden Aufrufe diesen Schwellenwert bereits berücksichtigen.

Ein weiterer interessanter Aspekt ist die Tastaturverwaltung in diesem Beispiel. Sie ist optimal, um möglichst wenige Operationen auszuführen, und vermittelt gleichzeitig ein sehr angenehmes Gefühl, wenn man einen Korridor hinuntergeht und mit zwei gleichzeitig gedrückten Tasten eine Verbindung zu einem anderen Korridor herstellt.

```

10 SPEICHER 23999
20 MODE 0: DEFINT A-Z: CALL &6B78: 'install RSX
21 bei Pause gosub 5000
25 call &bc02:'restore default palette just in case'.
26 gosub 2300:' Palette mit Überschreiben
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
40 |SETLIMITS,0,80,0,200: ' Grenzen des Spielbildschirms
45 |SETUPSP,30,9,&84d0:' Hintergrundgitter ("Y")
46 |SETUPSP,31,9,&84f2:' Ziegelstein ("Z")
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
110 c$(2)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
120 c$(3)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
125 c$(4)= "ZYZZZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
130 c$(5)= "ZYZZZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
140 c$(6)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
150 c$(7)= "ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
160 c$(8)= "ZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
170 c$(9)= "ZY      ZY      ZY"
190 c$(10)="ZY      ZY      ZY"
195 c$(11)="ZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
                  ZZZZZZZZZZZZZYYZ".
200 c$(12)="ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
210 c$(13)="ZYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYZ".
220 c$(14)="ZYZZZYZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
                  ZZZZZZZZZZZZZYYZ".

```

230 c\$(15)="ZYYYYYYYYYYYYYYYYYYZ
 ZYYYYYYYYYYYYYZ"
240 c\$(16)="ZYYYYYYYYYYYYYYYYYYZ
 ZYYYYYYYYYYYYYZ"
c\$(17)="ZYZZZZZZZZZZZZZZZZZZZZZ
 ZZZZZZZZZZZZZ".
260 c\$(18)="ZYYYYYYYYYYYYYYYYYYYYYY
 YYYYYYYYYZ".
270 c\$(19)="ZYYYYYYYYYYYYYYYYYYYYYY
 YYYYYYYYYZ".

```

271 c$(20)="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
272 c$(21)=""
273 c$(22)=""
274 c$(23)=""
    wir drucken das Layout
310 FOR i=0 TO 20:|LAYOUT,i,0,@c$(i):NEXT
311 locate 1,1:pen 9:print "DEMO OVERWRITE".
312 locate 3,23:pen 11:print "BASIC mit 8BP".
320 |SETUPSP,0,0,&x01000111:' Sprite- und Layout-Kollisionserkennung
330 |SETUPSP,0,7,1:dir=1: ' Reihenfolge = 1 (Kokosnuss rechts)
340 xa=20*2:xn=xa:ya=12*8:yn=ya:' Koordinaten des Zeichens
350 |LOCATESP,0,ya,xa: 'das Zeichen platzieren (ohne Druck)
360 |PRINTSPALL,0,1,0:' druckt Sprites
361 cl%=0:' Kollisionsvariable
362 |COLAY,89,cl%,0:'Schwellenwert chr$("Y") ist 89
    ' DAS SPIEL BEGINNT
401 |MUSIC,0,0,5
402 Tastaturauslesung und Kollisionen. Wenn wir in Richtung H (oder p) gehen, werden wir zuerst prüfen, ob die Richtungstaste V (q a) gedrückt ist und umgekehrt.
404 if dirn <3 then gosub 450: gosub 410 else gosub 410:gosub 450
405 |LOCATESP,0,yn,xn:|PRINTSPALL
406 ya=yn:xa=xn
407 zu 404

409 ' horizontale Richtung der Tastatur ---
410 wenn INKEY(27)<0 dann 430
420 xn=xa+1poke 27003,xn:|COLAY: IF cl%=0 then if dir<>>1 then
    |SETUPSP,0,7,1:DIR=1:xn=xa:return sonst dirn=1:return
421 xn=xa:poke 27003,xn:return:'es gibt eine Kollision
430 if INKEY(34)<0 then return
440 xn=xa-1poke 27003,xn:|COLAY: IF cl%=0 then if dir<>>2 then
    |SETUPSP,0,7,2:DIR=2:xn=xa:return sonst dirn=2:return
441 xn=xa:poke 27003,xn:'es gibt eine Kollision
442 return
449 vertikale Richtung der Tastatur
450 wenn INKEY(67)<0 dann 480
460 yn=ya-2poke 27001,yn:|COLAY: IF cl%=0 then if dir<>>3 then
    |SETUPSP,0,7,3:DIR=3:yn=ya:return sonst dirn=3:return
461 yn=ya:poke 27001,yn:'es gibt eine Kollision
    if INKEY(69)<0 then return
490 yn=ya+2poke 27001,yn:|COLAY: IF cl%=0 then if dir<>>4 then
    |SETUPSP,0,7,4:DIR=4:yn=ya:return sonst dirn=4:return
491 yn=ya:poke 27001,yn:'es gibt eine Kollision
492 Rückgabe

2300 REM ----- PALETA transparente Sprites MODE 0-----
2301 TINT 0,11: REM hellblau
    E
2302 TINT 1,15: REM orange
    E
2303 TINT 2,0 : TINTE 3,0: REM schwarz
    E
2305 TINT 4,26: TINTE 5,26: REM weiß
    E
2307 TINT 6,6: TINTE 7,6: REM rot
    E
2309 TINT 8,18: INK 9,18: REM grün
    E
2311 TINT 10,24: INK 11,24: REM gelb
    E

```

**2313 TINT 12,4: TINTE 13,4 :REM magenta
E**
**2315 TINT 14,16 : INK 15, 16:orange REM
E**
2317 GELB=10
**2420 RETURN
|MUSIK**
5010 Ende

10.5 Wie Sie in Ihren Layouts Speicherplatz sparen können

Wenn Ihr Spiel viele Bildschirme hat und Sie Platz sparen müssen, können Sie viele einfache Techniken anwenden, um Speicherplatz zu sparen. Ein Bildschirm verbraucht fast 0,5 KB, daher ist es wichtig, Methoden zu verwenden, um seine Größe zu reduzieren

Am einfachsten ist es, die Bildschirme so zu bearbeiten, als wären sie Sprites. Sie bearbeiten sie mit SPEDIT (zum Beispiel) und jedes Pixel stellt ein Element des Layouts dar. Je nach Farbe stehen sie für Felsen, Ziegelsteine, leeren Raum, Wasser, Erde, usw. Da im Modus 0 16 Farben zur Verfügung stehen, haben Sie 16 Arten von Steinen. Das von Ihnen erzeugte Sprite müssen Sie als Bild speichern und vor der Anzeige auf dem Bildschirm in ein Layout umwandeln, indem Sie es Pixel für Pixel einscannen und in den benötigten ASCII-Code umwandeln (Sie müssen es in BASIC oder wie auch immer programmieren). Wenn wir davon ausgehen, dass Sie 5 Zeilen für die Spielmarkierungen verwenden, wird ein Bildschirm 20x20 Pixel = 400 Pixel = 200 Bytes verbrauchen. In 1KB passen also 5 Bildschirme und in 10KB können Sie 50 Bildschirme unterbringen. Sie haben 4 Bits pro Layout-Element verwendet.

Die Bearbeitung von Bildschirmen als Sprites ist sehr "visuell", obwohl Sie sich entscheiden können, weniger Bits pro Layoutelement zu verwenden. Wenn Sie nur 2 Bits verwenden, haben Sie 4 Arten von Elementen und können auch Bildschirme bearbeiten, als wären es Bilder, indem Sie MODE 1 verwenden. In diesem Fall können Sie 100 Bildschirme in 10KB unterbringen. Wenn Sie eine andere Anzahl von Bits pro Baustein verwenden, wird es kompliziert, weil Sie die Bildschirme nicht wie Sprites zeichnen können, aber Sie können etwas programmieren, das ein von Ihnen gezeichnetes Bild in die benötigten Bits umwandelt.

Eine andere einfache und effektive Lösung besteht darin, jedes Layout mit großen Blöcken zu definieren, z.B. 8x16 Pixel. Und die Bildschirme mit Zeichen aufzubauen, die wir im Speicher ablegen können. In dem Videospiel "**Happy Monty**" wird der erste Bildschirm von "Mutant Monty" mit einer Matrix von 16x10 Zeichen = 160 Bytes erstellt, so dass 25 Bildschirme 4 KB belegen. Es stimmt, dass dieses Layout nur 16 Blöcke breit ist und nicht die 20, die es sein können, und auch nur 10 Blöcke vertikal definiert sind, statt der 25, die es sein können, aber auf diese Weise nimmt es sehr wenig Speicherplatz ein und wir können viele Bildschirme erstellen.

<pre>"IK m o JG" " IGGGGGGGGH IGGGGGGGHq" "z c xo " "C CCGK d" " F oC oC IDD DDDDDDDD " " F C F F " " Fv C JHz b xoF " " " F C IGGGGGGGGGGGH " " IGGH z a xw" " EEEEEoEEE "</pre>	
--	--

Abb. 64 ein mit 160 Zeichen definiertes Layout

11 Fortgeschrittene Programmierung und "Massenlogik".

11.1 Messung der Geschwindigkeit von Befehlen

Der BASIC-Interpreter ist sehr schwerfällig in der Ausführung, weil er nicht nur jeden Befehl ausführt, sondern auch die Zeilennummer analysiert, den eingegebenen Befehl parst, seine Existenz, die Anzahl und den Typ der Parameter überprüft, dass seine Werte in gültigen Bereichen liegen (z.B. ist PEN 40 illegal) und viele andere Dinge. Es ist die syntaktische und semantische Analyse jedes Befehls, die wirklich ins Gewicht fällt und nicht so sehr seine Ausführung. Der Fall der RSX-Befehle ist keine Ausnahme. Der BASIC-Interpreter prüft ihre Syntax, und das wiegt schwer, auch wenn es sich um in ASM geschriebene Routinen handelt, denn bevor er sie aufruft, hat der BASIC-Interpreter bereits viele Dinge getan.

Daher müssen Sie Befehlausführungen einsparen, indem Sie geschickt so programmieren, dass die Logik des Programms so wenige Befehle wie möglich durchläuft, auch wenn das manchmal bedeutet, mehr zu schreiben. Eine unverzichtbare Praxis ist die Verwendung von Befehlen, die eine Gruppe von Sprites bewegen oder beeinflussen, wie z.B. COLSPALL,

|AUTOALL oder |MOVERALL, wodurch die Verwendung von Schleifen mit Anweisungen, die ein einzelnes Sprite betreffen, vermieden wird.

Ein entscheidender Faktor beim Aufrufen eines Befehls ist die Anzahl der Parameter. Je mehr Parameter er hat, desto aufwendiger ist er für BASIC zu interpretieren, auch wenn es sich um eine ASM-Routine handelt, die durch CALL aufgerufen wird, denn der CALL-Befehl ist immer noch BASIC und vor dem Zugriff auf die Routine in ASM werden zwangsläufig Anzahl und Art der Parameter analysiert.

Um die Kosten für die Ausführung eines Befehls zu ermitteln, können Sie das folgende Programm verwenden. Sie können es auch verwenden, um die Leistung neuer Assembler-Funktionen zu bewerten, die Sie in die 8BP-Bibliothek einbauen, wenn Sie dies wünschen.

```
1 Aufruf &6b78
10 SPEICHER 23499
11 DEFINT a-z
12 c%=0: a=2
30 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT:'reset
31 Iterationen=1000
40 a!= ZEIT
50 FOR i=1 TO iterations
60 <hier geben Sie einen Befehl ein, z.B. PRINT "A">
70 NÄCHSTES
80 b!=TIME
90 PRINT (b!-a!): rem was es in cpc. Zeiteinheiten (1/300 Sekunden)
braucht
100 c!=((b!-a!)*1/300)/Iterationen: rem c! = wie lange jede Iteration in
Sekunden dauert
120 d!=(1/50)/c!
130 PRINT "Sie können ",d!, "Befehle pro Durchlauf (1/50 sec)" ausführen.
140 PRINT "Befehl dauert ";(c!*1000 -0.47); "Millisekunden"; "Befehl dauert
";(c!*1000 -0.47); "Millisekunden".
```

Hinweis: Für Assembler-Experten: Wenn Sie die Ausführungszeit einer Routine messen wollen, die intern Interrupts deaktiviert (DI-, EI-Befehle), ist die Zeit, die während der Deaktivierung vergeht, mit diesem BASIC-Programm nicht messbar. Die 8BP-Befehle

deaktivieren keine Interrupts und sind alle messbar.

Nachfolgend sehen wir das Leistungsergebnis einiger Befehle (gemessen mit dem obigen Programm). Es muss gesagt werden, dass ein direkter Aufruf der Speicheradresse (ein CALL &XXXX) schneller ist als der Aufruf des entsprechenden RSX-Befehls. In der folgenden Tabelle gilt: Je niedriger das Ergebnis (ausgedrückt in Millisekunden), desto schneller der Befehl. Die hier vorgestellte Tabelle sollten Sie immer im Hinterkopf behalten und Ihre Programmierentscheidungen auf dieser Grundlage treffen. Es handelt sich um eine Tabelle mit Messungen von BASIC-Befehlen und 8BP-Befehlen.

Befehl	ms	Kommentar
PRINT "A"	3.63	Sehr langsam. Denken Sie nicht einmal daran, es zu benutzen, außer gelegentlich, um die Anzahl der Leben zu ändern, aber drucken Sie keine Punkte in einem Spiel für jeden Feind, den Sie töten.
LOCATE 1,1: PRINT-Punkte	24.8 + 7	Das Platzieren des Textursors mit LOCATE und das Drucken der Mondwertvariablen "Punkte" ist sehr teuer. Wenn Sie Punkte aktualisieren, tun Sie dies nur von Zeit zu Zeit und nicht bei jedem Spielzyklus.
C\$=str\$(Punkte) PRINTAT,0, y, x, @c\$	10	Das Drucken der Punkte mit PRINTAT ist viel effizienter als mit LOCATE + PRINT (=32 ms), aber immer noch teuer. Verwenden Sie PRINTAT sparsam.
REM Guten Tag	0.20	Kommentare verbrauchen
' hallo	0.25	Sie sparen 2 Byte Speicherplatz, aber es ist langsamer!
GOTO 60	0.19	Sehr schnell! Sogar schneller als REM. Verwenden Sie diesen Befehl gnadenlos, verwenden Sie ihn!!!
A = 3	0.55	Ein einfacher Auftrag kostet. Alles kostet, jeder Auftrag muss durchdacht sein.
A = B	0.72	Die Zuweisung des Wertes einer Variablen an eine andere ist teurer als die Zuweisung eines Wertes. Und die Zuweisung des Wertes eines Arrays ist noch teurer, weil der Zugriff auf das Array Kosten verursacht. Und wenn das Array zweidimensional ist, kostet es sogar noch mehr.
A = miarray(x)	1.33	
A= miarray(x,y)	1.84	
 LOCATESP,i,10,20	2.8	Wenn Sie keine negativen Koordinaten verwenden, ist es besser, die Koordinaten mit dem BASIC-Befehl POKE zu setzen.
 LOCATESP,i,y,x	3.22	Wenn die Koordinaten variabel sind, dauert es länger.
CALL &XXXX,i,x,y	1.81	Das CALL-Äquivalent ist viel schneller.
 MOVER,31,1,1	3.23	Sie ist etwas langsam und sollte daher sparsam eingesetzt werden.
AUFRUF &XXXX,31,1,1	1.77	Der entsprechende Aufruf ist viel schneller
POKE &XXXX, Wert	0.71	Sehr schnell! Verwenden Sie es, um Sprite-Koordinaten zu aktualisieren (wenn positiv). POKE akzeptiert keine negativen Zahlen, aber Sie können die Formel 255+x+1 verwenden, wenn Sie eine negative Zahl eingeben wollen. Um zum Beispiel eine -4 einzugeben, würden Sie 255-4+1=252 eingeben. Eine weitere einfache Möglichkeit, positive und negative

		Zahlen einzugeben, ist die Verwendung von POKE-Adresse, x und 255.
POKE dir,data	0.85	Sehr schnell, wenn man bedenkt, dass er auch die Variable "dir" übersetzen muss.
 POKE,&xxxx,Wert	2.5	Es erlaubt negative Zahlen und ist besser als LOCATESP, wenn man nur eine Koordinate (X oder Y) aktualisiert.
X=PEEK(&xxxx)	0.93	Sehr schnell! Je nach Art des Spiels kann es eine Alternative zu COLSP sein, indem man sich die Farbe einer Bildschirmspeicheradresse ansieht. Im Anhang über den Videospeicher erkläre ich, wie man das macht.
X=INKEY(27)	1.12	Sehr schnell. Geeignet für Videospiele, obwohl Sie es intelligent verwenden müssen, wie in diesem Buch empfohlen.
IF x>50 THEN x=0	1.42	Jedes IF wiegt, müssen wir versuchen, sie zu retten, weil ein Spiel Logik wird viele haben
IF A=Wert THEN GOTO 100 Vs IF A=Wert THEN 100	1.24 Vs 1.18	Beide Sätze sind gleichwertig, aber der zweite braucht weniger Zeit.
IF inkey(27)=0 then x=5	1.75	Annehmbar. Es ist schneller als b=INKEY(27) und dann die IF...THEN
10 Wenn inkey(27) dann 30 20 x=5 30 <Anweisungen>.	1.0	Ein viel effizienterer Weg, das Gleiche zu tun
WENN x>0 dann Vs WENN x dann	1.3 Vs 0,8	In BASIC ist es möglich, 0,5ms zu sparen, wenn man berücksichtigt, dass jeder Wert ungleich Null TRUE bedeutet. Wenn wir einen bestimmten Wert kontrollieren wollen, werden wir das tun: 10 IF x-20 THEN 30 20 <was zu tun ist, wenn x=20> 30 ... Die Anwendung dieser Technik ist beim Lesen von Tastaturen sehr zu empfehlen.

A=A+1: Wenn A>4 dann A=0	2.6	Es ist viel besser, die zweite Option (MOD) zu verwenden. Andererseits sollte die Verwendung von MOD mit Vorsicht erfolgen. Wenn wir das tun: A=(A+1) MOD 3
Vs	Vs	
A=A MOD 3 +1	1.7	Es kostet uns 2 ms, weil die Klammern sehr teuer sind, aber wir bekommen das Gleiche. Wenn wir ab einer anderen Zahl als 1 zählen wollen, geben wir eine beliebige ungerade Zahl ein, und das genügt.
A=A MOD 3 + <impar>		Es gibt einen noch schnelleren Weg, dies zu tun, nämlich mit dem binären Operator AND, den wir uns jetzt ansehen werden.
A=1+A UND 7 (Anfangswert 0 Endwert 7)	1.6	Damit können Sie eine Variable zyklisch zwischen N Werten variieren , so dass Sie eine Sprite-ID für Ihren neuen Schuss oder für einen Feind, der den Bildschirm betritt, auswählen können.
A=20+A MOD 7 (Anfangswert 0 Endwert 26)	1.88	Es ist besser, AND als MOD zu verwenden, weil AND eine schnelle Binäroperation ist und MOD eine Division beinhaltet, die für unseren geliebten Z80-Mikroprozessor sehr teuer ist. Wenn wir jedoch ID-Sprites verwenden müssen, die nicht mit 1 beginnen, dann brauchen wir Klammern, weil der "+-Operator Vorrang vor "AND" hat und der Geschwindigkeitsvorteil von AND verloren geht. In diesem Fall
A=21 + (A und 7) (Ausgangswert 21 Endwert 28)	1.95	
		ist besser MOD. Wie auch immer, immer versuchen, es und wählen Sie, weil je nach der anfänglichen Zahl zu addieren Sie unterschiedliche Zeiten erhalten. Unglaublich aber wahr 20+A mod 7 → dauert 1.88 29+A mod 7 → dauert 1.94
Wenn A<0 dann	1.71	Prüfen, ob eine Zahl nicht negativ ist
A=15 A=A AND 15	1.24	Sie können die Prüfung vereinfachen, denn eine negative Zahl ist eigentlich eine Zahl, die eine "1" im höchstwertigen Bit hat, und wir entfernen die Negativität mit einem einfachen und
:	0.05	Es spart nicht viel, aber es ist schneller, ":" anstelle einer neuen Zeilennummer zu verwenden, und wenn Sie dies viele Male anwenden, sparen Sie am Ende erheblich. Zwei Anweisungen in zwei Zeilen benötigen 0,03 ms mehr, als wenn beide stehen in derselben Zeile, getrennt durch ":".
 PRINTSP,0,10,10	5.3	Einzelner 14 x 24 Sprite (7 Bytes x 24 Zeilen) Wenn Sie mehrere Sprites drucken wollen, ist es viel besser, alle Sprites auf einmal mit PRINTSPALL zu drucken.
CALL &xxxx,0,10,10	3.5	Äquivalent zu PRINTSP, also schneller, aber weniger lesbar

<p> PRINTSPALL</p> <p>(32 Sprites 8x16 des Modus 0, d.h. 4 Bytes x 16 Zeilen)</p>	55.4	<p>Das sind etwa 18 fps bei voller Sprite-Last. Was es braucht ist</p> $T = 3,25 + N \times 1,7$ <p>Das heißt, 1,7 ms pro Sprite und feste Kosten von 3 ms. Diese festen Kosten sind die Kosten für das BASIC-Parsing plus die Kosten für das Durchgehen der Sprite-Tabelle auf der Suche nach zu druckenden Sprites. Wenn Sie die Parameter weglassen (das ist möglich, und Sie würden die Werte des letzten Aufrufs übernehmen), sparen Sie 0,6 ms beim festen Anteil, das heißt:</p> $T = 2,6 + N \times 1,1$ <p>Wird der Druck überdrückt und/oder spiegelverkehrt gedruckt, ist er teurer. Die relativen Kosten der einzelnen Druckarten sind nachstehend aufgeführt:</p> <p>Normaler Druck: 100%. Druck mit Überschreiben: 164% Druck gespiegelt: 179% Druck mit Überschreiben: 164% Druck gespiegelt: 179% gespiegelt: 179% Gespiegelter Druck mit Überschreibung: 220%.</p>
<p> PRINTSPALL,N,0,0 (kein Sprite aktiv)</p> <p>N=0 N=10 N=31</p>	<p>2.6 4.3 5.9</p>	<p>Kosten für die Bestellung von Sprites: Wenn N=0 ist und keine Sprites zu drucken sind, muss die Funktion die Sprite-Tabelle der Reihe nach durchlaufen. Das Durchlaufen in der Reihenfolge ist jedoch teurer, wie die mit steigendem N verbrauchte Zeit zeigt. Die Zeitdifferenz (5,9 - 2,6 = 2,5 ms) entspricht den Kosten für das Sortieren aller Sprites.</p>
<p> COLAY,@x%,0</p>	<p>3.0 Vs</p>	<p>Verwenden Sie nur auf den Charakter, nicht auf Feinde oder das Spiel wird verlangsamt. Wenn die Figur ein Vielfaches von 8 ist, ist sie schneller. In diesem Beispiel war es 14x24 und logischerweise 14x24 war die Größe des Charakters.</p>
<p> COLAY</p>	2.4	<p>nicht ein Vielfaches von 8 ist. Je größer das Sprite, desto länger dauert es. Wenn Sie den Befehl ohne Parameter aufrufen, ist er viel schneller (Sie sparen 0,6 ms)!</p>
<p> COLAY gegen ANRUF &XXXX</p>	<p>2.4 ge ge n 2.0</p>	<p>Die Verwendung von CALL wie üblich ist schneller, aber weniger gut lesbar.</p>
<p>GOSUB / RETURN</p>	0.56	<p>Annehmbar schnell. Die Messung wurde mit einer Routine durchgeführt, die nur die Rückgabe vornimmt.</p>
<p> SETUPSP, id, param, Wert</p>	2.7	<p>Akzeptabel, obwohl POKE für bestimmte Parameter viel besser ist. Es gibt Parameter, die mit POKE gesetzt werden können, z. B. der Status, aber nicht andere (z. B. eine Route). Siehe das Referenzhandbuch</p>

FÜR / NÄCHSTES	0.6	Sie können damit mehrere Gegner durchlaufen und jeden nach der gleichen Regel bewegen lassen. Du solltest überlegen, ob du AUTOALL oder MOVEALL für deine Zwecke verwenden kannst, da ein Befehl alle gewünschten Personen bewegt, was viel besser ist als eine Schleife.
 COLSP,31, @c% COLSP,31	5.5 4.3	Unabhängig von der Anzahl der aktiven Sprites dauert es etwa gleich lang. Vermeiden Sie es, immer mit der Kollisionsvariablen aufzurufen, um den Vorgang auf 4,3 ms zu beschleunigen. Wenn du ein Schiff oder eine Figur und mehrere Schüsse hast, ist es viel effizienter, COLSPALL aufzurufen, anstatt COLSP mehrmals aufzurufen.
 ANIMALL (dieser Befehl ist nur mit einem Parameter von PRINTSPALL verfügbar, er ist nicht verfügbar von kann direkt aufgerufen werden)	3.5	Sie ist teuer, aber es gibt eine Möglichkeit, sie in Verbindung mit dem Aufruf von PRINTSPALL aufzurufen, und zwar durch einen Parameter, der bewirkt, dass diese Funktion vor dem Drucken der Sprites aufgerufen wird. Dies spart die BASIC-Schicht, d.h. die Zeit, die zum Senden des Befehls benötigt wird, die >1ms beträgt. Daher können wir sagen, dass dieser Befehl normalerweise etwas weniger als 2ms benötigt.
 AUTOALL	2.76	Er ist kostengünstig und kann alle 32 Sprites auf einmal bewegen.
 MOVEALL,1,1	3.4	Es ist nicht sehr teuer und kann alle 32 Sprites gleichzeitig bewegen.
TON	10	Der Sound-Befehl wird "blockiert", sobald der 5-Noten-Puffer voll ist. Das bedeutet, dass Ihre BASIC-Logik nicht mehr als 5 SOUND-Befehle verketten darf, sonst bleibt sie stehen, bis eine Note zu Ende ist... Wenn Sie sich entscheiden, ihn zu verwenden, müssen Sie sehr vorsichtig sein, da er sehr viel Energie verbraucht. Ausführungszeit (10 ms ist sehr lang)
WENN a>1 UND a>2 DANN a=2 Versus IF a>1 THEN IF a>2 THEN a=2	2.52 Vs 2.39	Eine einfache Möglichkeit, 0,13 ms zu sparen Denken Sie bei allem, was Sie programmieren, an diese Details, denn jede Einsparung ist wichtig.
A=RND*10	4.2	Die Funktion BASIC RND ist sehr teuer. Sie können sie verwenden, aber nicht in jedem Spielzyklus, sondern nur gelegentlich, zum Beispiel, wenn eine neue Feind oder ähnliches. Eine andere einfache Lösung ist die Speicherung von
		10 Zufallszahlen in einem Array und Verwendung dieser Zahlen, anstatt RND aufzurufen
Umrandung <x>	0.75	Ziemlich schnell. Nützlich in Kombination mit einer Art Sprite-Kollision zu verwenden, die den explosiven Effekt verstärkt.

IF a AND 7 then 30	1.19	Ich habe die Ausführungszeit angegeben, wenn die Bedingung erfüllt ist. Beide Fälle sind recht schnell.
IF A MOD 8 then 30	1.29	
ON x GOTO L1,L2,L3,L4 Vs 60 wenn x >2 dann 63 61 wenn x=1 dann 70 62 zu 70 63 wenn x=3 dann 70 64 bis 70	3.67 Vs 4.8	Ein ON GOTO-Befehl ist im Durchschnitt 1 ms schneller als sein Äquivalent mit "IF"-Befehlen, obwohl dies auch von der Wahrscheinlichkeit des Auftretens jedes der 4 Werte abhängt. Wenn die Wahrscheinlichkeit der 4 Werte gleich ist, können wir durch die Verwendung von ON GOTO 1 ms einsparen. Sie kann wie folgt weiter optimiert werden 100N X GOTO 30,40,50 20 <Anweisungen Fall x=4>: GOTO 60 30 <Anweisungen Fall x=1>: GOTO 60 40 <Anweisungen Fall x=2>: GOTO 60 50 <Anweisungen Fall x=3>: GOTO 60 60 Fortführung des Programms Mit dieser Strategie sind wir auf 3,54 ms heruntergekommen.

Tabelle 5 Liste der Ausführungszeiten für einige Befehle

Wichtige Empfehlungen:

- **Verwenden Sie DEFINT A-Z zu Beginn des Programms.** Die Leistung wird sich dadurch erheblich verbessern. Dies ist fast obligatorisch. Dieser Befehl löscht alle Variablen, die vorher existierten, und erzwingt, dass alle neuen Variablen Ganzzahlen sind, es sei denn, es wird durch Modifikatoren wie "\$" oder "!" anders angegeben (siehe das Amstrad BASIC-Programmierhandbuch). Beachten Sie, dass Sie bei der Verwendung von DEFINT eine Zahl größer als 32768 in hexadezimaler Form eingeben müssen.
- Wenn Sie es vermeiden können, ein IF zu durchlaufen, indem Sie ein GOTO einfügen, ist es immer vorzuziehen, ein
- Wenn es Ihnen an Geschwindigkeit mangelt und Sie etwas mehr Tempo brauchen, verwenden Sie **CALL**.
<Adresse> anstelle von RSX. In diesem Fall müssen Sie Parameter mit negativen Zahlen im hexadezimalen Format übergeben.
- Synchronisieren Sie den Befehl **|PRINTSPALL** nicht mit dem Screen Sweep, es sei denn, Ihr Spiel läuft sehr schnell. Die Synchronisierung kann Ihre FPS verringern. Solange Sie 12 FPS erreichen, ist Ihr Spiel im Allgemeinen "spielbar".

- **Beseitigen Sie Leerzeichen.** Jedes Leerzeichen in Ihrem BASIC-Listing verbraucht 0,01 ms bei der Ausführung.
- **Kürzen Sie die Namen Ihrer Variablen.** Je länger sie sind, desto mehr kostet der Zugriff auf sie.

Betrieb	Wetter
A=A+1	Ein Buchstabe, Dauer 1,18ms
HO=HO+1	Zwei Buchstaben, dauert 1,2ms (2% länger)
HELLO=HELLOA+1	5 Buchstaben, braucht 1,25 ms (6% länger)
HELLOFRIENDS=HELLOFRIENDS +1	10 Buchstaben 1,34 ms (13 % mehr)

- **Reduzieren Sie die Anzahl der Variablen.** Wenn es viele Variablen gibt, sind Lese- und Schreibzugriffe langsamer.
- Sobald Sie mit Parametern den Befehl **|STARS** oder den Befehl **|PRINTSPALL** oder **|COLAY** oder andere 8BP-Befehle verwendet haben, rufen Sie sie die folgenden Male nicht mit Parametern auf. Die 8BP-Bibliothek hat "Speicher" und wird die zuletzt verwendeten Parameter verwenden. Das spart Millisekunden beim Durchlaufen der Parsing-Schicht des BASIC-Interpreters.
- Denken Sie immer daran, dass ein Ausdruck, der nicht Null ist, TRUE ist. Dadurch können Sie bei jedem IF 0,5 ms einsparen, was beim Lesen der Tastatur und der Steuerung von Variablen genutzt werden kann.

Schlechte Option	Gute Wahl (spart 0,5ms)
IF x<>>0 THEN <Anweisungen>	IF x THEN <Anweisungen>
IF x=20 THEN...	10 Wenn x=20 Dann 30 20 <Anweisungen>. 30
IF INKEY(34)=0 THEN <Anweisungen>.	10 IF INKEY(34) THEN 30 20 <Anweisungen>. 30

- In Schiffsspielen, in denen du kein Überschreiben verwendest, solltest du sicherstellen, dass dein Schiff Sprite 31 ist, damit es über die Sprites, die den Hintergrund vorgeben, "drübergeht", da dein Schiff anschließend gedruckt wird.
- Prüfung alternativer Versionen desselben Vorgangs
A=A+1:IF A>4 then A=0 : REM dies verbraucht 2,6 ms
A=A MOD 3 +1 : REM dies verbraucht 1,84 ms
A=1 + A AND 3 : REM dies verbraucht 1,6 ms
- Vermeiden Sie die Verwendung negativer Koordinaten. So können Sie POKE verwenden, um die Position Ihrer Figur zu aktualisieren. Der POKE-Befehl (der BASIC-Befehl) ist sehr schnell, unterstützt aber nur positive Zahlen, ebenso wie PEEK. Falls du native Koordinaten verwendest, verwende **|POKE** und **|PEEK** (8BP-Befehle). Verwenden Sie **|LOCATESP nur**, wenn Sie beide Koordinaten gleichzeitig ändern wollen und diese positiv und/oder negativ sein können. Denken Sie auch daran, dass ein POKE eines negativen x-Wertes mit der POKE-Adresse 255+x+1 durchgeführt werden kann. Falls Sie negative Koordinaten

verwenden wollen, um zu zeigen, wie die Feinde langsam von links in den Bildschirm eindringen (Clipping), können Sie die Koordinaten vermeiden

Die Verwendung eines **|SETLIMITS** und damit den gleichen Effekt mit Koordinaten, die bei Null beginnen und einen etwas kleineren Spielbildschirm erzeugen.

- Wenn Sie etwas überprüfen müssen, überprüfen Sie es nicht in jedem Spielzyklus. Es kann ausreichen, dieses "Etwas" alle 2 oder 3 Zyklen zu überprüfen, ohne es in jedem Zyklus überprüfen zu müssen. Um entscheiden zu können, wann etwas ausgeführt werden soll, verwenden Sie die "modulare Arithmetik". In BASIC gibt es die MOD-Anweisung, die ein hervorragendes Werkzeug ist. Um zum Beispiel eine von 5 Operationen auszuführen, können Sie Folgendes tun: **IF cycle MOD 5 = 0 THEN ...** obwohl es besser ist, **UND-Operationen** als MOD-Operationen zu verwenden.
- Machen Sie Gebrauch von "**Todessequenzen**". Damit können Sie Anweisungen speichern, um zu prüfen, ob ein explodierendes Sprite sein letztes Animationsbild erreicht hat, um es zu deaktivieren.
- Überschreiben ist teuer: Wenn Sie Ihr Spiel ohne Überschreiben erstellen können, sparen Sie Millisekunden und gewinnen Farbe. Verwenden Sie es, wenn Sie es brauchen, aber nicht ohne Grund.
- Animationsmakros ersparen Ihnen BASIC-Zeilen, weil Sie die Bewegungsrichtung des Sprites nicht überprüfen müssen. Verwenden Sie sie, wann immer Sie können.

11.2 Unser Amstrad hat nur 64KB und wenn man den Videospeicher, den Speicher für die Sprites, die Musik und die 8BP-Bibliothek abzieht, bleiben 24KB BASIC übrig, die man verwenden kann.

sehr gut. Wenn jeder Bildschirm ein eigenes "Programm" in Ihrem Spiel hat, werden Sie kaum in der Lage sein
einen Satz von 10 Bildschirmen herstellen.

Es gibt zwei Dinge, die Sie tun sollten, um die Speichernutzung zu reduzieren

- Erstellen von Low-Byte-Bildschirmen
- Erstellen Sie eine einzige Logik, die für alle Bildschirme gilt, d.h. derselbe Spielzyklus soll auf allen Bildschirmen im Spiel ausgeführt werden.

Bei Screen-Passing-Spielen, die das Layout verwenden, kann jeder Bildschirm 20x25 Bytes, also 500 Bytes, belegen. Wenn Sie bestimmte "Tricks" anwenden, wie in Kapitel 8 erläutert, können Sie diesen Speicherplatz reduzieren. In dem Videospiel "**Happy Monty**" werden 25 Bildschirme mit jeweils nur 160 Byte gebaut und es gibt eine einzige Spielzykluslogik für alle Bildschirme.

Es ist sehr wichtig, dass Sie eine einzige Spielzykluslogik programmieren und diese auf alle Bildschirme anwenden. Wenn Sie für jeden Bildschirm eine Spielzykluslogik programmieren, wird der Quellcode Ihres Programms riesig und Sie können nur wenige Bildschirme programmieren, weil Ihnen bald der Speicherplatz ausgeht.

Sie können auch Speicherbeschränkungen überwinden, indem Sie Algorithmen verwenden, die Labyrinthe oder Bildschirme erzeugen, ohne sie zu speichern. Auf diese Weise können Sie viel mehr Bildschirme erstellen. Das erfordert natürlich Kreativität,

aber es ist möglich. Ein Algorithmus

benötigt immer weniger Speicherplatz als die von ihr erzeugten Daten, obwohl sie logischerweise mehr Zeit für die Ausführung benötigt als das einfache Lesen gespeicherter Daten.

Sie können die gegnerische Logik von einem Bildschirm zum anderen wiederverwenden und so Codezeilen sparen. Nutzen Sie dazu den GOSUB/RETURN-Mechanismus. Es ist auch sehr nützlich, Routen für Feinde zu verwenden. **Mit dem Pathing-Mechanismus bewegt sich der Feind, ohne BASIC-Logik auszuführen**, und arbeitet sehr schnell. Weisen Sie einem Feind einfach einen Pfad zu, und er wird ihn immer wieder durchlaufen, ohne dass teure IF-Anweisungen, Zuweisungen usw. erforderlich sind.

Sie können auch Spiele erstellen, die in Etappen geladen werden, so dass Sie nicht das ganze Spiel auf einmal im Speicher haben. Das ist ein bisschen ärgerlich für den Bandbenutzer (CPC464), aber nicht für den Plattenbenutzer (CPC6128).

Verwenden Sie **Sprite-Flipping**, um Speicher für Ihre Sprites zu sparen

11.3 Massenlogik" Technik

Sie werden oft viele Sprites bewegen müssen, vor allem in Weltraum-Arcade- oder "Commando"-Spielen (Capcoms Klassiker von 1985).

Du könntest die Koordinaten aller Sprites separat bearbeiten und mit POKE aktualisieren, aber das wäre sehr langsam und nicht praktikabel, wenn du eine flüssige Bewegung willst. Der beste (und einfachste) Weg ist die kombinierte Verwendung der Funktionen für automatische Bewegung und relative Bewegung, die |AUTOALL bzw. |MOVERALL heißen.

Der Schlüssel zum Erreichen von Geschwindigkeit in vielen Sprites ist die Technik, die ich als "massive Logik" bezeichnet habe. Bei dieser Technik geht es im Wesentlichen darum, weniger Logik pro Spielzyklus auszuführen (was als "Verringerung der Rechenkomplexität" bezeichnet wird), und es gibt mehrere Möglichkeiten, dies zu tun:

- Verwenden Sie **eine einzige Logik**, die auf viele Sprites gleichzeitig wirkt (unter Verwendung der automatischen und/oder relativen Bewegungsflags).
- Mehrere Aufgaben ausführen, aber nur eine oder einige von ihnen in jedem Spielzyklus, unter Verwendung **modularer Arithmetik (oder binärer Operationen) in Kaskade**.
- Einführung von **Beschränkungen im Spiel, die nicht wichtig sind** oder das Spielgeschehen nicht beeinflussen, um die Anzahl der Aufgaben, die in jedem Spielzyklus ausgeführt werden, zu reduzieren oder um Aufgaben zu vereinfachen, damit sie schneller ausgeführt werden.
- Verringern Sie in der Regel die Anzahl der Anweisungen, die Ihr Programm in jedem Spielzyklus durchläuft, indem Sie manchmal Algorithmen durch Vorberechnungen ersetzen oder mehr Anweisungen einfügen, so dass (paradoxerweise) weniger Anweisungen pro Zyklus ausgeführt werden.

Diese Ideen haben das gleiche Ziel: **weniger Logik in jedem Zyklus auszuführen**, so dass sich alle Sprites gleichzeitig bewegen können, aber weniger Entscheidungen in jedem Zyklus des Spiels getroffen werden. Dies wird als "**Reduzierung der**

Rechenkomplexität" bezeichnet, wobei ein Problem der Ordnung N (N Sprites) in ein Problem der Ordnung 1 (eine einzige Logik, die in jedem Frame ausgeführt wird) umgewandelt wird.

Der Schlüssel liegt darin, zu bestimmen, welche Logik oder Logiken in jedem Zyklus ausgeführt werden sollen. Im einfachsten Fall, wenn wir N Sprites haben, werden wir einfach eine der N Logiken ausführen. In komplexeren Fällen müssen wir jedoch genau überlegen, welche Logik ausgeführt werden soll.

11.3.1 Bewegt 32 Sprites mit massiver Logik

Schauen wir uns nun ein einfaches Beispiel an, bei dem 32 Sprites gleichzeitig und flüssig (mit 14fps) bewegt werden. Das ist durchaus möglich. In jedem Zyklus wird nur ein Geist Entscheidungen treffen, obwohl sich alle Geister in allen Zyklen bewegen werden. Wir können sie auch alle animieren (indem wir sie mit einer Animationssequenz verknüpfen und mit |PRINTSPALL,1,0) und es wird immer noch glatt sein, aber es wird immer noch so aussehen, als gäbe es mehr Bewegung, da der Flügelschlag einer Fliege (zum Beispiel) viel Gefühl von Bewegung erzeugt.

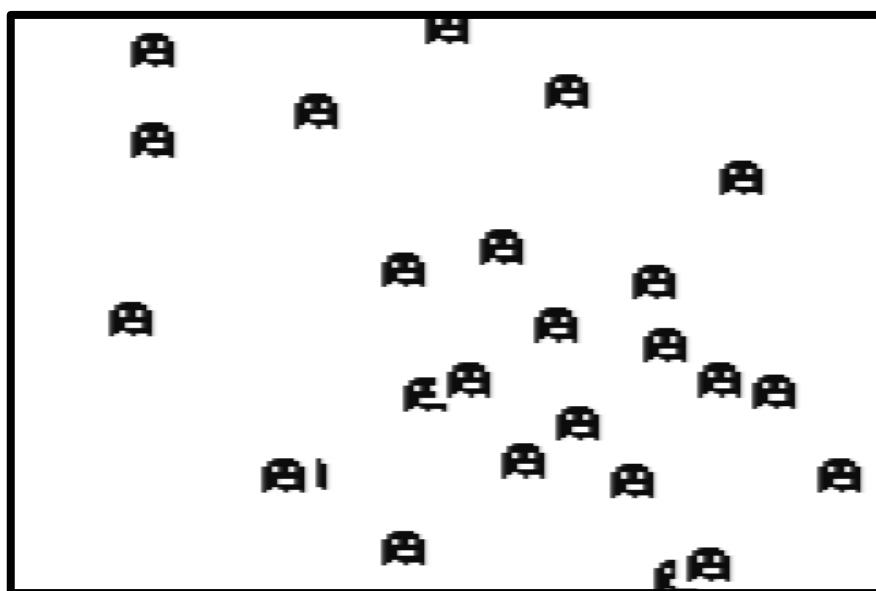


Abb. 65 mit massiver Logik kann man 32 Sprites gleichzeitig bewegen

Was wir getan haben, ist, die Rechenkomplexität zu reduzieren. Wir begannen mit einem Problem der "Ordnung N", wobei N die Anzahl der Sprites ist. Unter der Annahme, dass jede Sprite-Logik 3 BASIC-Befehle erfordert, müssten im Prinzip $N \times 3$ Befehle in jedem Zyklus ausgeführt werden. Mit der "Bulk-Logic"-Technik verwandeln wir das Problem der "Ordnung N" in ein Problem der "Ordnung 1". Ein Problem der "Ordnung 1" ist ein Problem, das unabhängig von der Größe des Problems eine konstante Anzahl von Operationen erfordert. In diesem Fall sind wir von $N \times 3 = 32 \times 3 = 96$ BASIC-Operationen auf nur 3 BASIC-Operationen gekommen. Diese Verringerung der Komplexität ist der Schlüssel zur hohen Leistung der Bulk-Logik-Technik.

```

1 MODUS 0
10 SPEICHER 24999: AUFRUF &6B78
20 DEFINT a-z
25 ' Feinde zurücksetzen
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35 ' num Feinde 12 x 16 (6 Bytes breit x 16 Zeilen)
36 num=32: x%=0:y%=0
40 FOR i=0 TO num-1:|SETUPSP,i,9,&8ee2: |SETUPSP,i,0,&X1111:
41 |LOCATESP,i,rnd*200,rnd*80
42 nächste

```

```

43 i=0
45 gosub 100
46 i=i+1: wenn i=num
dann i=0
50 |PRINTSPALL,0,0
60 |AUTOALL
70 goto 45
100 |peek,27001+i*16,@y%
110 |peek,27003+i*16,@x%
    if y%<=0 then |SETUPSP,i,5,2:|SETUPSP,i,6,0: return
130 if y%>=190 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0: return
140 if x%<=0 then |SETUPSP,i,5,0:|SETUPSP,i,6,1: return
150 if x%>=76 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1: return

160 Chance=rnd*3
170 if random=0 then |SETUPSP,i,5,2:
|SETUPSP,i,6,0:return
180 if random=1 then |SETUPSP,i,5,-
2:|SETUPSP,i,6,0:return
190 if random=2 then
|SETUPSP,i,5,0:|SETUPSP,i,6,1:return
200 if random=3 then |SETUPSP,i,5,0:|SETUPSP,i,6,1:
1:return

```

10.3.2 Alternierende und periodische Kaskadenausführung

Sie müssen nicht alle Aufgaben in jedem Spielzyklus ausführen. Wenn Sie z. B. prüfen möchten, ob der Schuss den Bildschirm verlassen hat, können Sie dies alle zwei oder drei Zyklen tun, anstatt jeden Zyklus zu prüfen.

Sie können die Feinde auch dazu bringen, alle paar Zyklen zu schießen und nicht jeden Zyklus (sonst würden sie oft auf Sie schießen!!).

Kurz gesagt, es gibt Dinge, die man nicht in jedem Zyklus tun muss, und man kann die Ausführung von Befehlen pro Zyklus einsparen und somit eine höhere Geschwindigkeit erreichen. Dies ist die Grundlage der "Bulk-Logic"-Technik.

Hierfür gibt es zwei grundlegende Techniken: die Verwendung von modularer Arithmetik und binären **UND-Operationen**. Binäre **UND-Operationen** sind schneller als **MOD-Operationen**.

Technik	Verbrauchte Zeit
A = A+1: wenn A=5 dann A=0: GOSUB <Routine>.	2,6 ms
IF cycle MOD 5 =0 THEN gosub routine	1,84 ms, vorausgesetzt, Sie haben bereits eine Variable namens cycle, die aktualisiert wird

Die MOD-Operation ist etwas kostspielig, weshalb manchmal eine binäre Operation besser ist.

Unter der Annahme, dass die Zyklusvariable jedes Mal aktualisiert wird, können wir eine binäre Operation durchführen, um zu sehen, wann eine Gruppe von Bits einen bestimmten Wert ergibt. Wenn wir uns zum Beispiel die 4 niederwertigsten Bits der Zyklusvariablen ansehen, gehen sie immer von 0000 nach 1111 und wieder zurück. Wenn wir diese Variable mit AND 15 verknüpfen, können wir das Gleiche tun wie mit

MOD 15. Die Zahl 15 in Binärform ist 1111 und eine AND-Verknüpfung zeigt den Wert dieser 4 Bits.

Technik	Verbrauchte Zeit
Wenn Zyklus AND 15=0 dann gosub Routine	1,6 ms (wird einmal in 16 Fällen ausgeführt)
Wenn Zyklus AND 1=0 dann gosub Routine	1,6ms (Läuft einmal alle 2 Male)

Wenn Sie mehrere periodische Aufgaben zu erledigen haben, können Sie so vorgehen:

```
c=Zyklus AND 15 :' rem 15 ist im Binärkode 1111
IF c=0 THEN GOSUB <routine1> (Routine1 wird einmal alle 16 Mal
ausgeführt) iF c=8 THEN GOSUB <routine2>... (Routine2 wird einmal alle 16
Mal ausgeführt, aber zeitlich weit von der Ausführung von Routine1
entfernt)
```

Auf diese Weise verteilen Sie die Zeit auf verschiedene Aufgaben, so dass Sie in jedem Zyklus nur eine Aufgabe erledigen, aber nach mehreren Zyklen alle Aufgaben erledigt haben.

Um die Zyklusvariable zu überprüfen und zu entscheiden, ob eine Aufgabe ausgeführt werden soll, gibt es einen besseren Weg, die binären Operationen auszuführen, und zwar den folgenden:

Technik	Verbrauchte Zeit
10 Wenn Zyklus und 7 dann 30 20 <Anweisungen die werden alle 8 Zyklen ausgeführt werden> 30 <Programmfortsetzung>	1,18 ms. Dies ist zweifelsohne die beste Strategie für die periodische Ausführung von Aufgaben.

Die Anwendung der gleichen Strategie auf MOD erhöht ebenfalls die Geschwindigkeit, wenn auch weniger als bei AND. Sie ist jedoch sehr gut, weil sie auch dann funktioniert, wenn die Periode kein Vielfaches von 2 ist (man kann MOD 7, MOD 5, MOD 10 usw. eingeben).

Technik	Verbrauchte Zeit
10 Wenn Zyklus MOD 8 dann 30 20 <Anweisungen die werden alle 8 Zyklen ausgeführt werden> 30 <Programmfortsetzung>	1,29 ms. Fast so gut wie AND und die beste Strategie, wenn wir eine Periode brauchen, die kein Vielfaches von 2 ist.

Wie Sie sehen, müssen wir für jede Aufgabe, die wir in die Logik einführen wollen, ein "IF" einführen. **Die Logik kann jedoch noch verbessert** und effizienter gestaltet werden, indem man Zeitintervalle für die Ausführung von Aufgaben verwendet, die ein Vielfaches davon sind. Wenn sie ein Vielfaches sind, **kann die "WENN" der Aufgabe 2 innerhalb der Aufgabe 1 und die "WENN" der Aufgabe 3 innerhalb der Aufgabe 2 in "Kaskade" ausgeführt werden**. Dadurch wird die Anzahl der in jedem Zyklus auszuführenden "IFs" erheblich reduziert, da es sich in vielen Fällen um einen einzigen "IF" handelt.

Schauen wir uns ein vollständiges Beispiel an, mit dem Sie 4 verschiedene Aufgaben multitaskingfähig machen können, wobei die Anzahl der gleichzeitig laufenden Aufgaben reduziert wird. Die folgende Sequenz stellt die Reihenfolge der Ausführung der Aufgaben und des Quellcodes dar.



```
10 WENN Zyklus UND 1 DANN 90
20 REM alle zwei Zyklen, die wir hier eingeben
25 WENN Zyklus UND 3 DANN 80
30 REM alle 4 Zyklen geben wir hier ein
35 WENN Zyklus UND 7 DANN 70
40 REM alle 8 Zyklen geben wir hier ein
50 <Aufgabe 4> : GOTO 100
70 <Aufgabe 3> : GOTO 100
```

80 <Aufgabe 2> : GOTO 100

90 <Aufgabe 1>.

100 REM --- Ende der Aufgaben ---

In diesem Beispiel haben wir die Intervalle 2, 4 und 8 gewählt.

UND 1 : das ergibt ein Intervall von 2, weil es alle 2 Zyklen Null

ist UND 3 : es ist alle 4 Zyklen Null

AND 7 : ist alle 8 Zyklen Null

Da wir Operationen in Multiplo-Intervallen gewählt haben, werden die IFs "in Kaskade" ausgeführt: Wir geben eine IF nur dann ein, wenn wir die vorherige eingegeben haben:

- Die Hälfte der Zyklen führt einen einzigen IF aus (Zeile 10).
- Die Hälfte der Zyklen führt zwei IF-Anweisungen aus (Zeilen 10 und 25), die andere Hälfte (d. h. 25 %) führt drei IF-Anweisungen aus (Zeilen 10, 25 und 35).

Im Durchschnitt werden pro Zyklus $1*50\%+2*25\%+3*25\% = 1,75$ IF-Anweisungen ausgeführt.

Dank dieser Strategie, **modulare Arithmetik mit binären Operationen in mehreren Intervallen zu verwenden, um sie zu kaskadieren**, können wir die Anzahl der "IF"-Operationen auf ein Minimum reduzieren und gleichzeitig die Rechenkomplexität von der Ordnung N (n Aufgaben) auf die Ordnung 1 (eine Aufgabe pro Zyklus) verringern. Dies beschleunigt Ihre Spiele enorm.

11.3.3 Einfaches Beispiel für Massenlogik

In dem Videospiel "Mutante Montoya" laufen die gegnerischen Sprites abwechselnd durch die verschiedenen Zyklen des Spiels. **Als ich dieses Spiel programmierte, hatte ich noch nicht den |ROUTEALL-Mechanismus programmiert, der es erlaubt, den Sprites feste Bahnen zuzuweisen, aber ich konnte es mit massiver Logik lösen.** Wenn man ein Spiel machen wollte, in dem die Feinde "Intelligenz" haben, würde ein fester Pfad nicht funktionieren, so dass man, selbst wenn man den ROUTEALL-Befehl hat, die Sprite-Logik wie unten beschrieben drehen müsste, daher ist dieses Beispiel interessant.

Angenommen, wir haben 3 feindliche Soldaten, die sich von rechts nach links und von links nach rechts bewegen. Um die Geschwindigkeit zu erhöhen, werden wir in jedem Spielzyklus nur die Logik eines Soldaten ausführen.

Um die x-Koordinate jedes Soldaten trotzdem in Bewegung zu halten, verwenden wir das automatische Bewegungsflag, anstatt es selbst zu aktualisieren.

10 SPEICHER 24999

20 MODE 0: DEFINT A-Z: CALL &6B78: 'install RSX

25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen

26 |SETLIMITS,0,80,0,0,200

30 "Parametrisierung von 3 Soldaten

40 dim x(3):x%=0

50 x(1)=10:xmin(1)=10:xmax(1)=60:

y(1)=60:direccion(1)=0:|SETUPSP,1,7,9 :|SETUPSP,1,0,&x1111:

|SETUPSP,1,5,0: |SETUPSP,1,6,1

60 x(2)=20:xmin(1)=15:xmax(2)=40:

y(2)=100:direccion(2)=1:|SETUPSP,2,7,10 :|SETUPSP,2,0,&x1111:

|SETUPSP,2,5,0: |SETUPSP,2,6,-1

```

70 x(3)=30:xmin(1)=5:xmax(3)=50:
y(3)=130:direccion(3)=0:|SETUPSP,3,7,9 :|SETUPSP,3,0,&x1111:
|SETUPSP,3,5,0: |SETUPSP,3,6,1
80 for i=1 to 3:|LOCATESP,i,y(i),x(i):next: 'wir platzieren die Sprites
81 i=0
89 ----- HAUPTSPIELSCHLEIFE (SPIELZYKLUS) -----
90 i=i+1:gosub 100
92 wenn i=3 dann i=0
93 |AUTOALL
94 |PRINTSPALL,1,0: ' animiert und druckt die 3 Soldaten
95 Gehe zu 90
96 ----- END-OF-GAME-CYCLE -----
99 ----- soldaten-routine -----
100 |PEEK,27003+i*16,@x%: x(i)=x%
101 IF address(i)=0 THEN IF x(i)>=xmax(i) THEN
address(i)=1:|SETUPSP,i,7,10: |SETUPSP,i,6,-1 ELSE return
110 IF x(i)<=xmin(i) THEN address(i)=0:|SETUPSP,i,7,9 :
|SETUPSP,i,6,1
120 Rückgabe

```

Jeder Soldat hat seine eigene Logik, aber wir führen in jedem Spielzyklus nur einen aus, wodurch der Spielzyklus viel leichter wird.

Die einzige Einschränkung besteht darin, dass die Koordinate die von uns für zwei Zyklen festgelegte Grenze überschreiten könnte, wenn wir die Logik jedes Soldaten nur einmal von drei Mal ausführen. Deshalb müssen wir bei der Festlegung des Limits vorsichtiger sein und sicherstellen, dass er nicht in eine Wand unseres Bildschirmlabyrinths eindringt und sie auslöscht. Ich werde versuchen, dieses Problem genauer zu erklären:

Angenommen, wir haben 8 Sprites und unser Sprite bewegt sich in allen Zyklen, aber wir führen seine Logik nur in einem von 8 Fällen aus. Stellen Sie sich ein Sprite vor, das sich an der Position x=20 befindet und wir wollen, dass es sich zur Position x=30 bewegt und sich umdreht. Nehmen wir an, dass das Sprite eine automatische Bewegung mit Vx=1 hat. In diesem Fall überprüfen wir seine Position bei x=20, x=28, x=36. Wenn wir 36 erreichen, stellen wir fest, dass wir zu weit gegangen sind!!! und wir ändern die Geschwindigkeit des Sprites auf Vx=-1.

Wie Sie sehen, ist die Kontrolle der Flugbahngrenzen nicht präzise, es sei denn, wir berücksichtigen diesen Umstand und setzen die Grenze auf einen Wert, den wir kontrollieren können, also Xfinal = Xinitial + n*8.

Diese Einschränkung ist winzig im Vergleich zu dem Vorteil, viele Sprites mit hoher Geschwindigkeit zu bewegen. Mit etwas Geschick können wir die Logik sogar weniger oft ausführen, so dass nur jeden zweiten Zyklus eine Art von Feindlogik ausgeführt wird.

11.3.4 Blockieren" der Bewegung von Staffeln

Wenn Sie ein Geschwader nur in eine Richtung bewegen wollen, können Sie eine der beiden folgenden Funktionen aus der 8BP-Bibliothek verwenden:

- Wenn Sie **|AUTOALL** verwenden, müssen Sie die Sprites in der gewünschten Richtung (in Vx, in Vy oder in beiden) automatisch beschleunigen und natürlich

Bit 4 des Statusbytes setzen. Der Befehl AUTOALL hat einen optionalen Parameter, um intern |ROUTEALL aufzurufen, bevor die Sprites bewegt werden.

- Wenn Sie **|MOVERALL** verwenden, müssen Sie Bit 5 des Statusbytes auf die Sprites setzen, die Sie bewegen wollen. Dieser Befehl erfordert als Parameter, wie viel relative Bewegung in Y und X Sie wollen.

Auf diese Weise bewegen Sie mit einer einzigen Anweisung viele Sprites gleichzeitig. Wenn sich jedes Sprite unabhängig und mit einer unabhängigen Logik bewegen muss, wie es in Spielen wie "Pacman" der Fall ist, müssen Sie schlauer sein, wie ich Ihnen im Folgenden erklären werde.

11.3.5 Massive Logiktechnik in "Pacman"-ähnlichen Spielen

Wenn man viele Feinde hat und diese an jeder Weggabelung in einem Labyrinth Entscheidungen treffen müssen, ist es keine gute Strategie, einfach abwechselnd die Logik der Feinde in jedem Zyklus auszuführen. Am besten ist es, die Logik auszuführen, wenn sie eine Entscheidung treffen muss. In Pacman-artigen Spielen geschieht dies, wenn ein Geist eine Gabelung erreicht, an der er aufgrund seiner Intelligenz eine neue Bewegungsrichtung einschlagen kann. Dies kann mit einem einfachen "Trick" erreicht werden. Er besteht einfach darin, die Feinde zu Beginn des Spiels an gut gewählten Positionen zu platzieren.

Angenommen, Sie haben 4 Feinde und die Abzweigungen des Labyrinths treten in Vielfachen von 4 auf. Wenn der erste Feind in einer Position ist, die ein Vielfaches von 4 ist, ist er an der Reihe, seine Logik auszuführen. Der zweite Feind darf seine Entscheidungslogik im nächsten Zyklus ausführen. Befindet er sich nicht in einer Labyrinth-Verzweigungsposition, kann er seinen Kurs nicht ändern.

Um seine Position mit einem Vielfachen von 4 zu "bestücken" und so entscheiden zu können, welchen Weg er an der Gabelung einschlagen soll, beginnen wir das Spiel einfach mit diesem zweiten Feind, der sich an einem Vielfachen von 4 minus eins befindet. Bei Koordinaten, die bei Null beginnen, sind die Vielfachen von 4:

Erster Feind: Position 0 oder 4 oder 8 oder 12 oder 16 oder 20 oder 24 oder XX (auf der x- oder y-Achse, das ist egal) Zweiter Feind: Position 1 oder 5 oder 9 ...
Dritter Feind: Position 2 oder 3 oder 10 ...

Und so weiter. Sie platzieren Ihre Feinde nach dieser Regel:

Position = Vielfaches von 4 - n, wobei n die Sprite-Nummer ist

Und jedes Mal, wenn ein Gegner an der Reihe ist, seine Logik auszuführen, kann er sich an einer Weggabelung wiederfinden. Wenn ein Geist zum Zeitpunkt "i" eine Entscheidung treffen muss, dann trifft er eine Entscheidung zum Zeitpunkt $t=i+4$, und die nächste Entscheidung fällt zum Zeitpunkt $t=i+4+4$, und so weiter.

Sehen wir uns ein grafisches Beispiel an, in dem ich mit einem Rechteck hervorgehoben habe, dass der Gegner eine Entscheidung treffen wird, weil er sich in einer Gabelung befindet. Wie Sie sehen können, wird in jedem Spielzyklus nur eine Fork-Logik ausgeführt.

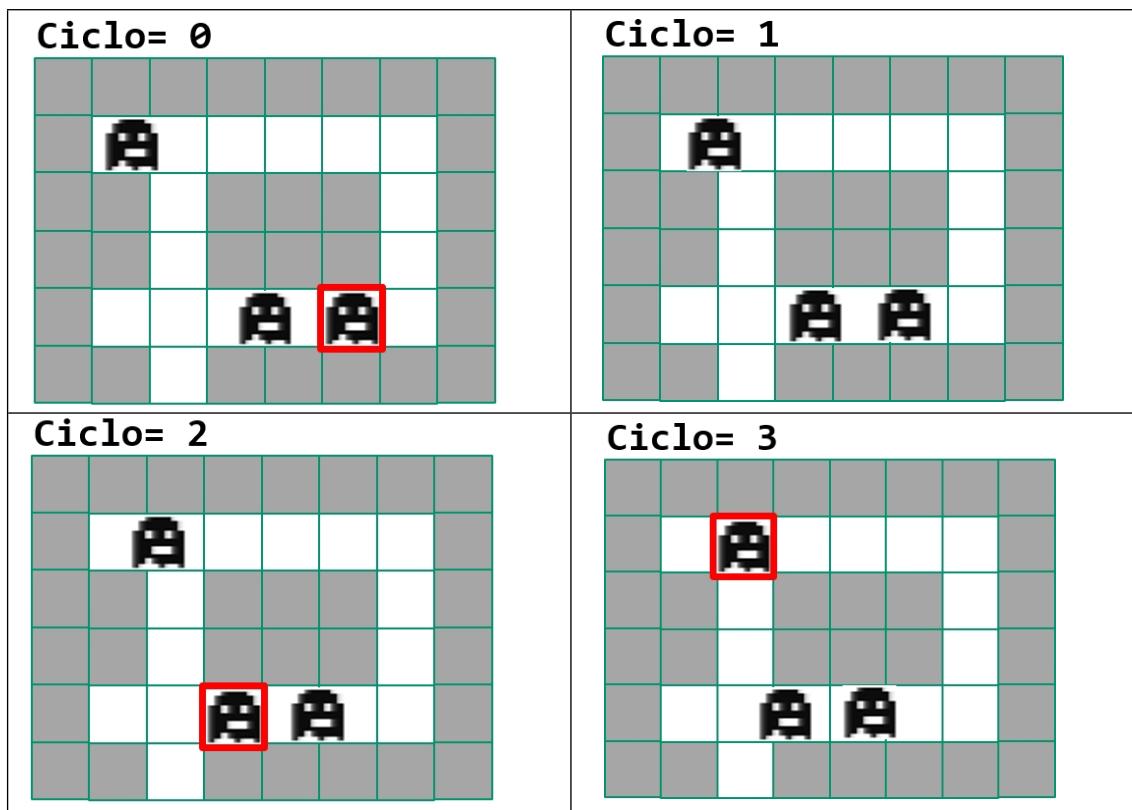


Abb. 66 Massive Logik in Spielen vom Typ PACMAN

Wenn Sie an der Reihe sind, Ihre Logik auszuführen, sollten Sie überprüfen, dass Sie sich nicht in der Mitte eines Korridors ohne Abzweigungen befinden. Sie sollten dies mit dem Befehl |COLAY überprüfen.

Das Videospiel "Paco, der Mann" verwendet diese Technik. Tatsächlich führt es die Logik von 4 Geistern aus, jeder in einem anderen Spielzyklus, und verteilt auch die restlichen Aufgaben auf die verschiedenen Zyklen. Es ist ein großartiges Beispiel für massive Logik, daher empfehle ich Ihnen, das "Making of" des Videospiels zu lesen, das Sie in der 8BP-Dokumentation finden.

	Zyklen		
Aufgabe	0	1	
Lesen der Tastatur	X		
Erkennung Kollisio n mit Geistern y mit Sprites unsichtbar		X	
Erkennung Kollisio n mit Layout y Umzug w enn Paco stürzt ab			X
Erkennung vo n			X

Das Videospiel **"Paco the man"** verteilt die Aufgaben auf 4 Spielzyklen, wodurch fast 20 FPS in BASIC erreicht werden können.

Punkte (Kokosnüsse)				
Phantom Logik 1	X			
Phantom Logik 2		X		
Phantom Logik 3			X	
Phantom Logik 4				X

11.3.6 Verringerung der Anzahl der Anweisungen im

Spielzyklus Die Verringerung der Anzahl der Anweisungen, die der Spielzyklus durchläuft, ist wichtig, um Ihr Programm zu beschleunigen. Manchmal bedeutet dies, dass das Programm mehr Zeilen hat, auch wenn durch weniger Zeilen in jedem Zyklus ausgeführt werden. In anderen Fällen können Sie Beschränkungen einführen "unentdeckbar", die Ihr Spiel beschleunigen, ohne dass der Spieler den Unterschied bemerkt. Werfen wir einen Blick auf einige Beispiele

11.3.6.1 Tastaturverwaltung mit weniger Anweisungen

Verwalten Sie die Tastatur (und das gilt im Allgemeinen für alles, was Sie tun), indem Sie so wenige Anweisungen wie möglich ausführen. Hier ist ein Beispiel (erst schlecht und dann gut gemacht), bei dem Sie höchstens 4 INKEYS-Operationen mit ihren entsprechenden IFs durchlaufen. Führen Sie es gedanklich aus und Sie werden sehen, was ich meine. Die zweite Variante ist viel schneller

Schlechtes Beispiel (schlimmster Fall = 8 "IF INKEY"-Ausführungen):

```
1000 rem ineffiziente Tastaturroutine
1010 IF INKEY(27)=0 und INKEY(67)=0 THEN <Anweisungen>:RETURN
1020 IF INKEY(27)=0 und INKEY(69)=0 THEN <Anweisungen>:RETURN
1030 IF INKEY(34)=0 und INKEY(67)=0 THEN <Anweisungen>:RETURN
1040 IF INKEY(34)=0 und INKEY(69)=0 THEN <Anweisungen>:RETURN
1050 IF INKEY(27)=0 THEN <Befehle>:RETURN 1060
IF INKEY(34)=0 THEN <Befehle>:RETURN 1070 IF
INKEY(67)=0 THEN <Befehle>:RETURN 1080 IF
INKEY(69)=0 THEN <Befehle>:RETURN 1080 IF
INKEY(69)=0 THEN <Befehle>:RETURN
```

Hier ist das Beispiel einer Tastenauslesung, aber gut gemacht (mit worst case = 4 "IF INKEY"-Ausführungen). Außerdem wurde berücksichtigt, dass Ausdrücke, die nicht Null sind, in einem IF TRUE sind. Die Anweisungen, die in Ihrem Spiel ausgeführt werden sollen, können unterschiedlich sein, aber das Schema zum Lesen der Tastatur sollte dasselbe sein, wenn es um die Handhabung von Diagonalen geht (z. B. gleichzeitig nach oben und nach rechts). Es mag länger erscheinen, aber es ist viel schneller als das vorherige Beispiel.

```
REM effiziente Tastaturroutine
'Sprung zu 1550, wenn "P" nicht
gedrückt wurde 1510 if inkey(27) THEN
1550: 'Taste P 1520 if inkey(67) THEN
1530: 'Taste Q
```

```
1525 <Hinweise für den Fall, dass "P" und "Q" gleichzeitig gedrückt
wurden
Zeit>:RETURN
1530 if inkey(69) THEN 1540:'A' Schlüssel
```

```
1535 <Hinweise für den Fall, dass "P" und "A" gleichzeitig gedrückt
wurden
Zeit>:RETURN
```

```
1540 <Anweisungen, falls Sie nur "P" gedrückt haben>:RETURN 1550 if
inkey(34) THEN 1590:'O' key
```

1560 if INKEY(67) THEN 1570:'Q' Taste

**1565 <Hinweise für den Fall, dass "O" und "Q" gleichzeitig gedrückt wurden
Zeit>:RETURN**

1570 if INKEY(69) THEN 1580: 'Schlüssel A

**1575 <Anweisungen, wenn Sie "O" und "A" gleichzeitig gedrückt haben>:
RETURN**

1580 <Anweisungen, wenn Sie nur "O" gedrückt haben>:RETURN

1590 IF INKEY(67) THEN 1600:'Q' Taste

1595 <Anweisungen, wenn "Q"> gedrückt wurde: RETURN

1600 IF INKEY(69) THEN return:'A' key

1610 <Befehle, wenn nur "A" gedrückt wurde>: RETURN

Eine weitere Möglichkeit, Ihr Spiel zu beschleunigen, besteht darin, "sekundäre" Tasten zu scannen, z. B. Tasten zum Aktivieren/Deaktivieren von Musik, Tasten zum Wechseln in ein Menü oder zum Anzeigen von etwas Besonderem usw. Diese Tasten können Sie in regelmäßigen Abständen und nicht bei jedem Zyklus abfragen. Sie müssen jedoch berücksichtigen, wie viel es kostet, sie zu scannen, was nicht viel ist (1 ms).

10 if inkey(47) then 30: ' das kostet 1,0 ms

20 <Hinweise, wenn Sie die Taste 47 drücken>.

30 rem Sie hierher gelangen, wenn Sie nicht darauf geklickt haben

Das Parsen einer Variablen mit AND für eine Aufgabe, die (zum Beispiel) alle 4 Zyklen stattfindet, kostet 1,18 ms, d.h. es kostet uns

1,18 ms x 4 Zyklen (Zyklusauswertung) + 1,0ms (Inkey) =5,72 ms

Wenn wir stattdessen den Inkey bei jedem Zyklus ausführen würden, hätten wir nur 4ms gebraucht, daher macht das Scannen bestimmter Tasten basierend auf dem Spielzyklus nur Sinn, wenn wir zumindest vermeiden wollen, in einigen Zyklen zwei Tasten zu scannen.

Gehen wir von dem folgenden Programm aus:

10 if cycle and 3 then 50: ' das kostet 1,18 ms

20 if inkey(47) ... ' das kostet 1 ms

30 if inkey(35) ...' das kostet 1 ms

50 <weitere Anweisungen>.

Derzeit wertet das Programm die Tasten 47 und 35 in jedem vierten Zyklus aus. Wenn es sie auswertet, braucht es $1,18 + 1 + 1 + 1 = 3,8$ ms, wenn es sie nicht auswertet, braucht es 1,18 ms. Die in 4 Zyklen verbrauchte Zeit ist also
Zeit $3 * 1,18 + 3,8 = 6,72$ ms

Hätten wir die Tasten hingegen in jedem Zyklus ausgewertet, hätten wir $4 * 2$ ms= 8 ms gebraucht. Es ergibt sich also eine Einsparung von $8 - 6,7 = 1,3$ ms für alle 4 Zyklen, also etwa 0,3 ms pro Spielzyklus.

Eine der Optionen, die Sie je nach Art des Spiels haben, besteht darin, einige Tasten in

geraden und andere in ungeraden Zyklen zu scannen. In einem Spiel, das QAOP-Tasten verwendet, können Sie zum Beispiel QA in geraden Zyklen und OP in ungeraden Zyklen scannen oder umgekehrt.

Auf diese Weise können Sie mehr Geschwindigkeit mit einer Einschränkung erreichen, die der Spieler wahrscheinlich nicht bemerkt. Diese Art von Einschränkung ist es manchmal wert, eingeführt zu werden, aber das hängt vom Spiel ab.

11.3.6.2 Vermeiden Sie den Weg über unnötige FIs

Wir werden uns zwei Möglichkeiten ansehen, dies zu tun:

```
10 WENN A=1 DANN < Anweisungen, wenn A=1> 10 WENN A=1 DANN <
Anweisungen, wenn A=1> 10 WENN A=1 DANN < Anweisungen, wenn A=1>
20 WENN A=2 DANN < Anweisungen, wenn A=2> 20 WENN A=2 DANN <
Anweisungen, wenn A=2> 20 WENN A=2 DANN < Anweisungen, wenn A=2>
30 IF A=3 THEN <Anweisungen, wenn A=3>.
40 <weitere Anweisungen>
```

Wenn Sie das Durchlaufen eines IF vermeiden können, indem Sie ein GOTO einfügen, ist dies immer vorzuziehen. Das GOTO ist ein großer Verbündeter der massiven Logiktechnik.

```
10 IF A=1 THEN <Anweisungen, wenn A=1> : GOTO 40
20 IF A=2 THEN < Anweisungen, wenn A=2> : GOTO 40
30 <Anweisungen, wenn A=3> : rem wenn A weder 1 noch 2 ist, dann
ist es 3
40 <mehr Anweisungen>
```

Eine andere Möglichkeit ist die Verwendung der Anweisung ON <Variable> GOTO oder der Anweisung ON <Variable> GOSUB.

Wie ich in der Tabelle von 11.1 dargestellt habe, können Sie durch die Verwendung von ON GOTO mehr als 1ms einsparen.

```
10 auf A GOTO 30,40,50
20 goto 60: rem wir hier, wenn A=4
30 rem wir hier ankommen, wenn A=1
35 bis 60
40 rem wir hier ankommen, wenn A=2
45 bis 60
50 rem kommen wir hier an, wenn A=3
60 rem hier geht die Logik weiter
```

11.3.6.3 Ersetzt Algorithmen durch Vorberechnungen

Stellen Sie sich einen hüpfenden Ball vor. Anstatt die Gleichungen der beschleunigten Bewegung zu verwenden, konstruieren Sie einen Pfad, der ein Sprite bei jedem Schritt mit einem größeren Y-Koordinateninkrement nach unten und dann mit einem immer kleineren Inkrement nach oben bewegt, wenn es auf den Boden trifft. Es müssen keine komplexen Gleichungen ausgeführt werden, und doch ist der visuelle Effekt derselbe. So habe ich die Sprünge im Spiel "Frisches Obst & Gemüse" gemacht. Es ist möglich, dies so zu programmieren, weil **das Universum im Spiel "deterministisch" ist. Das heißt, man kann zu jedem Zeitpunkt vorhersagen, was passieren wird**, egal wie komplex die Gleichungen sind, die den Sprung einer Figur oder die Bewegung eines Trupps bestimmen.

In Spielen, in denen die gegnerische Logik eine Funktionsberechnung erfordert (z. B. Kosinus), sollten Sie alles vorberechnen und in einem Array speichern, das Sie während der Logikausführung verwenden. Die Berechnung während der Spiellogik ist zu kostspielig.

Komplexe Logik ist langsame Logik. Wenn Sie etwas Kompliziertes machen wollen, eine komplexe Flugbahn, einen Mechanismus der künstlichen Intelligenz... tun Sie es nicht, sondern versuchen Sie, es mit einem einfacheren Verhaltensmodell zu "simulieren", aber denselben visuellen Effekt zu erzielen. Ein Beispiel: Ein intelligenter Geist, der dich verfolgt, trifft keine intelligenten Entscheidungen, sondern versucht, die gleiche Richtung wie deine Figur einzuschlagen, ohne jegliche Logik. Wenn Sie nicht auf diese Weise vereinfachen können, dann denken Sie daran, dass sogar künstliche Intelligenz "deterministisch" werden kann. Wenn ein Feind auf der Grundlage Ihrer Position und Geschwindigkeit entscheidet, wie er sich bewegen soll, könnten wir das Ergebnis dieses schweren Algorithmus in einem Array speichern und alle Berechnungen vermeiden.

```
10 Rem Vx, Vy, X, Y sind die Geschwindigkeit und Position meiner Figur.
20 rem nehmen wir an, dass ich Entscheidungen für 3 Geschwindigkeiten, 10 X-Koordinaten-Slots und 10 Y-Slots vorberechnet habe. Das ist weniger als 1KB
30 DIM pursue(3,3,10,100)
40 rem ' Laden der Werte in das Array, nachdem sie langsam berechnet wurden
50 newaddress=pursue(Vx,Vy,x,y): rem-Mechanismus in Aktion
```

Das Universum, das Sie programmieren, ist "deterministisch". Wie komplex das Verhalten von Gegnern und Bildschirmelementen auch sein mag, wenn ihr Verhalten nicht von Ihrer Interaktion abhängt, dann gibt es für jedes Ding zu jedem Zeitpunkt eine Position. Eine Position, die im Voraus berechnet werden könnte, um jeden komplexen Verhaltensalgorithmus zu vermeiden, und das Ergebnis wäre das gleiche.

11.3.6.4 Kommentare nicht ausführen

Entfernen Sie alle Kommentare in der Spiellogik, und wenn Sie welche hinterlassen, die REM (schneller) sind, verwenden Sie keine Anführungszeichen. Wenn Sie das Anführungszeichen verwenden, sparen Sie damit 2 Byte Speicherplatz und können den Rest des Programms auskommentieren (Initialisierungen usw.). Wenn Sie Teile der Logik auskommentieren wollen, können Sie wie folgt vorgehen:

```
Wenn x>23 gosub 500
...
499 rem wird nicht auf dieser Linie weitergegeben, und so kommentiere ich diese Routine.
500 if x > 50 THEN ...
...
550 RÜCKKEHR
```

Jeder Kommentar, den Sie ausführen, verbraucht 0,20 ms und es ist sehr einfach, seine Ausführung zu speichern, ohne Kommentare zu hinterlassen. Es gibt Zeiten, in denen Sie Kommentare in Zeilen einfügen können, solange es Sprünge gibt (GOTO und GOSUB/RETURN), ohne Angst zu haben, Zeit zu verschwenden, sehen wir uns einige Beispiele an:

```
10 goto 50 : rem dieser Kommentar ist nicht zeitaufwendig
20 gosub 200: rem dieser Kommentar ist nicht zeitaufwendig
200 zurück: dieser Kommentar ist nicht zeitaufwendig
```

11.3.6.5 In jedem Zyklus stirbt nur ein Sprite

Der Befehl 8BP **COLSPALL** ist mit "massiver Logik" konzipiert. Das bedeutet, dass er potenziell die Kollision aller Sprites erkennt, aber sobald er eine Kollision erkennt, gibt er an, welches Sprite der Kollidierende und welches das kollidierte Sprite ist. Zu diesem Zeitpunkt können Sie das kollidierende Sprite außer Kraft setzen (indem Sie seine Fähigkeit, kollidiert zu werden, in Bit 1 seines Statusbytes deaktivieren) und den Befehl erneut ausführen, bis es keine Kollisionen mehr gibt (er gibt eine 32 für den Kollider und den Kollidierten zurück). Wie auch immer,

Es ist am effizientesten, den Befehl erst im nächsten Spielzyklus erneut auszulösen. Das bedeutet, dass nur ein Feind pro Frame sterben kann, aber es ist eine unmerkliche Einschränkung, die Ihre Spiele sehr beschleunigen wird.

11.3.7 Routen, die das Spiel durch Manipulation des Zustands beschleunigen

Sie können Pfade erstellen, die abwechselnd den Kollisionsschalter oder das Kollisionsschalter-Flag in Ihren Sprites ein- und ausschalten. Je nach Art des Gegners kann dir das zusätzliche Geschwindigkeit beim Kollisionsbefehl geben. Pfade werden in einem späteren Kapitel erklärt, daher ist es eine gute Idee, sich vor dem Lesen dieses Kapitels mit ihnen vertraut zu machen.

Wenn Sie beispielsweise 8 Feinde auf dem Bildschirm haben, können Sie es so einrichten, dass in den geraden Zyklen 4 Feinde kollidieren können und in den ungeraden Zyklen die anderen 4:

ROUTE1;

----- db

1,0,1

**db 255,128+8+2+1,0 ; Änderung des Zustands auf
kollidierbar db 1,0,1
db 255,128+8+1,0 ; Zustandswechsel zu nicht
kollisionsfähig db 0**

Dieser Pfad ändert den Zustand deines Sprites, indem er es nach rechts verschiebt. Wenn einem Sprite dieser Pfad zugewiesen ist, bewegt es sich nach rechts und ist abwechselnd kollisionsfähig und nicht kollisionsfähig.

Sie können den Pfad 8 Sprites zuweisen und dann den folgenden Befehl auf 4 der Sprites ausführen:

|ROUTE8P, <sprite_id>, 1

Damit wird der Kollisionsbefehl **|COLSPALL** schneller, da er nur noch Kollisionen mit 4 kollidierenden Gegnern in jedem Frame erkennen muss. Dieser Trick beschleunigt Ihr Spiel ein wenig und zusammen mit anderen Tricks können Sie endlich die FPS erreichen, die Sie brauchen.

Mit dem gleichen Trick kann man das Druck-Flag (Bit 0 des Status) ein- und ausschalten. In einem Pfad, in dem der Gegner einige Frames verbringt, ohne sich zu bewegen, kann man das Druck-Flag durch den Pfad ausschalten und so den Befehl **PRINTSPALL** beschleunigen.

11.3.8 Routing von Sprites mit "massiver Logik".

Um Sprites durch Pfade zu bewegen, gibt es einen Befehl namens **|ROUTEALL**, der dies sehr effizient macht, aber als **Übung zum Verständnis der Philosophie der Bulk-Logik ist es sehr interessant, diesen schwierigen, aber emblematischen Fall von Bulk-Logik zu studieren**. Um das "Anunnaki"-Videospiel zu programmieren, verwendete ich die Technik, die ich im Folgenden beschreiben werde, da ich die Sprite-Routing-Fähigkeit in 8BP noch nicht programmiert hatte. Im Grunde habe ich massive Logik für das Routing der feindlichen Schiffe verwendet.

Die Schiffe passieren nacheinander eine Reihe von "Kontrollknoten", d. h. Orte im Raum, an denen sie ihre Richtung ändern müssen, definiert durch ihre Geschwindigkeiten in X und Y, d. h. (Vx,Vy).

Eine Möglichkeit zu kontrollieren, dass die 8 Schiffe an diesen Stellen die Richtung ändern, wäre, ihre X,Y-Koordinaten mit denen der einzelnen Kontrollknoten zu vergleichen, und wenn sie mit einem von ihnen übereinstimmen, dann wenden wir die neuen Geschwindigkeiten an, die mit den

Änderung in diesem Knoten. Da wir von 2 Koordinaten, 8 Schiffen und 4 Knoten sprechen, sehen wir uns das an:

2 x 8 x 4 = 64 Kontrollen auf jedem Rahmen

Dies ist nicht machbar, wenn wir von BASIC Geschwindigkeit erwarten, da es keine recheneffiziente Strategie ist. Da wir es mit einem "**deterministischen**" Szenario zu tun haben, können wir zu jedem Zeitpunkt sicher sein, wo sich jedes der Schiffe befinden wird, und daher können wir **uns**, anstatt im Raum zu prüfen, nur **auf die Zeitkoordinate konzentrieren** (die die Nummer des Spielrahmens oder die so genannte "Spielzyklus"-Nummer ist). Betrachten Sie die Zeit nicht in Sekunden, sondern in Frames.

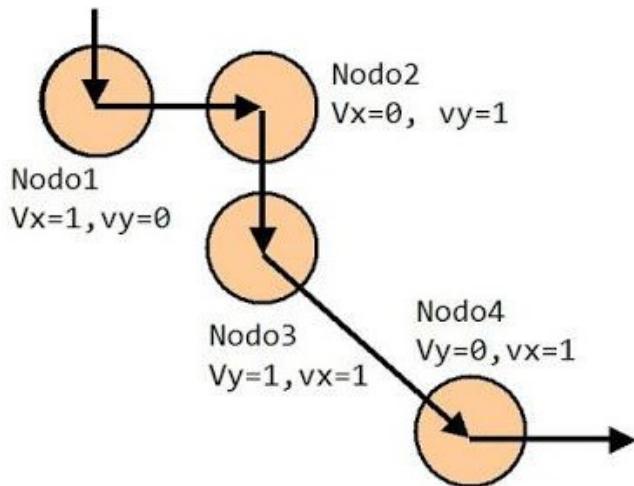


Abb. 67 Definierte Trajektorie mit "Kontrollknoten".

Da wir die Geschwindigkeit kennen, mit der sich die Schiffe bewegen, wissen wir, wann das erste Schiff den ersten Knoten passieren wird. Wir nennen diesen Zeitpunkt $t(1)$. Außerdem nehmen wir an, dass aufgrund des Abstands zwischen den Schiffen das zweite Schiff den Knoten zum Zeitpunkt $t(1)+10$ passieren wird. Das dritte zum Zeitpunkt $t(1)+20$ und das achte zum Zeitpunkt $t(1)+70$. Anstatt Rahmen als Zeiteinheiten zu verwenden, nehmen wir Zehnerrahmen: In diesem Fall lauten die Zeitpunkte $t(1), (1)+1, t(1)+2$ usw.

Mit diesem Wissen können wir die Zeit mit zwei Variablen steuern: eine wird die Zehner (i) und die andere die Einer (j) zählen. Um die Veränderung der 8 Schiffe im ersten Knoten zu steuern, können wir schreiben:

```
j=j+1: IF j=10 THEN j=0: i=i+1: IF i>=t(1) AND i<=t(1)+8 THEN
[aktualisiert die Geschwindigkeit des Schiffes i-t(1) mit den
Geschwindigkeitswerten von Knoten 1].
```

Wie wir sehen, können wir mit einer einzigen Zeile die Geschwindigkeiten der einzelnen Schiffe beim Durchlaufen des Knotens 1 ändern. Jedes Mal, wenn "j" Null wird, erhöhen wir die Variable "i" und aktualisieren eines der Schiffe. Während der ersten 80 Zeitpunkte (8 in Zehner-Frames) wird jedes der 8 Schiffe aktualisiert, wenn sie den Kontrollknoten passieren, d.h. zum Zeitpunkt $t(1)$ wird Sprite 0 aktualisiert, zum Zeitpunkt $(t1)+1$ wird Sprite 1 aktualisiert, zum Zeitpunkt $t(1)+2$ wird Sprite 2 aktualisiert usw.

Die Sprite-Nummer, die auf der Linie erscheint, ist $i-t(1)$, so dass, wenn $t(1) = 1$ ist, zu sein, die Frame 40, ($t1=4$) dann, wenn "i" ist 4 wird die Aktualisierung der Sprite 0, und wenn "i" ist 11 wird die Sprite 7 (8 Schiffe insgesamt) zu aktualisieren.

Nun wollen wir das Gleiche auf alle 4 Knoten anwenden. Wir könnten 4 Prüfungen anstelle von einer durchführen, aber das wäre ineffizient. Außerdem würde dies bei einer großen Anzahl von Knotenpunkten eine große Anzahl von Überprüfungen bedeuten. Wir können es mit nur einer tun, wenn wir berücksichtigen, dass das erste Schiff einen Knoten zu einem Zeitpunkt $t(n)$ passiert und das achte Schiff diesen Knoten zu $t(n)+7$ passiert.

Wenn das erste Schiff den ersten Knoten passiert, ist es sinnvoll, mit der Überprüfung von Knoten 2 zu beginnen, nicht aber von Knoten 3 oder 4.

Was den kleinsten Knoten betrifft, so können wir davon ausgehen, dass selbst bei 20 Knoten diese weit genug voneinander entfernt sind, so dass kein Schiff mehr als 3 Knoten auf einmal durchfährt (wir nehmen dies an und verwenden diese "3" als Parameter). Daher ist der kleinste zu prüfende Knoten der größte - 3. Wir werden den kleinsten Knoten "nmin" und den größten "nmax" nennen ($nmin = nmax - 3$). Wenn wir die volle Freiheit haben wollen, jede beliebige Flugbahn zu definieren, muss $nmin$ gleich $nmax$ minus der Anzahl der Schiffe in der Reihe sein.

```
10 j=j+1: IF j=10 THEN j=0: i=i+1: n=nmax  
20 IF n<nmin THEN 50: 'keine weiteren Schiffe zu aktualisieren  
30 IF i>=t(n) AND i<=t(n)+8 THEN [Aktualisierung von Schiff i-t(n) mit  
Knotengeschwindigkeiten n]:IF i-t(n)=0 THEN nmax=nmax+1: nmin=nmax-3  
40 n=n-1
```

50 ' plus Spielanleitung

Wie Sie sehen können, wird, wenn die Zeitdekade (Variable "i") um 1 erhöht wird, geprüft, ob sich ein Schiff in einem der Knoten von "nmax" bis "nmin" befindet, wobei in jedem Rahmen nur ein Schiff aktualisiert wird. Wenn das Schiff, das aktualisiert wird, Null ist, wird der maximale Knoten inkrementiert, da dieses Schiff auf dem Weg zum nächsten Knoten ist.

Für das nächste Bild wird die Anzahl der Knoten verringert (Anweisung $n = n-1$), so dass geprüft wird, ob sich im vorherigen Knoten ein Schiff befindet, und so weiter bis $nmin$. Es wird jedoch immer nur ein Schiff in jedem Bild geprüft.

Kurz gesagt, wir haben mit Hilfe der "Massenlogik" 64 Prüfungen in nur eine umgewandelt. Und wenn der Pfad 40 statt 4 Knoten hätte, hätten wir 640 Operationen in eine einzige umgewandelt!

Das Videospiel "**Annunaki**" verwendet diese Technik, um die Flugbahnen von zwei symmetrischen Reihen mit je 6 Schiffen zu steuern. Es ist kompliziert, aber wie Sie sehen können, können Sie mit BASIC die Kontrolle über 12 Schiffe übernehmen und sie auf skurrilen Bahnen bewegen, wobei Sie dank der massiven Logiktechnik mehr Intelligenz als Leistung einsetzen.

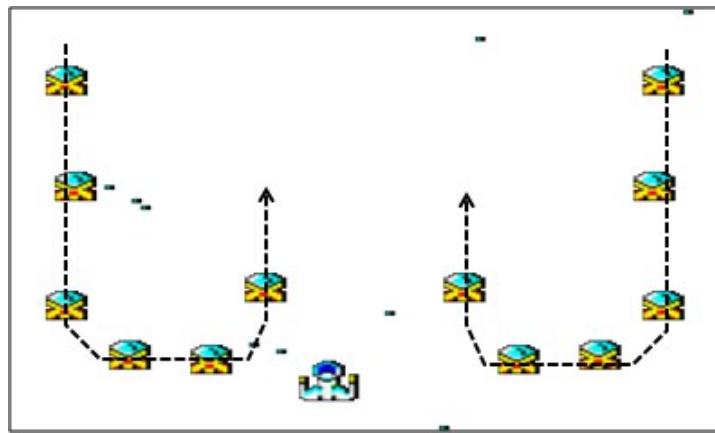


Abb. 68 Zwei Reihen mit massiven Logiken

12 Komplexe Trajektorien: Befehl ROUTEALL

Dies ist ein "erweiterter" Befehl, der seit Version V25 der 8BP-Bibliothek verfügbar ist. Er vereinfacht die Programmierung erheblich, da man einen Pfad definieren und ein Sprite diesen Schritt für Schritt mit dem Befehl ROUTEALL durchlaufen lassen kann.

Zuerst müssen Sie eine Route erstellen. Dazu müssen Sie sie in der Datei routes_yourgame.asm bearbeiten.

Jede Route hat eine unbestimmte Anzahl von Segmenten (die maximale Länge einer Route beträgt 255 Byte) und jedes Segment hat drei Parameter:

- Wie viele Schritte wir in diesem Segment machen werden (zwischen 1 und 250)
- Welche Geschwindigkeit Vy soll während des Segments beibehalten werden ($-127 \leq Vy \leq 127$)
- Welche Geschwindigkeit Vx während des Abschnitts eingehalten werden soll ($-127 \leq Vx \leq 127$)

Da eine Route maximal 255 Bytes lang sein kann und ein Segment 3 Bytes belegt, kann eine Route maximal 84 Segmente haben.

Am Ende der Segment-Spezifikation müssen wir eine Null setzen, um anzugeben, dass der Pfad beendet wurde und dass das Sprite den Pfad von Anfang an durchlaufen soll. Schauen wir uns ein Beispiel an:

LISTE DER ROUTEN

=====

Geben Sie hier die Namen aller Routen ein, die Sie erstellen ROUTE_LIST

dw ROUTE0
dw ROUTE1
dw ROUTE2
dw ROUTE3
dw ROUTE4
dw ROUTE4

DEFINITION DER EINZELNEN STRECKEN

=====

ROUTE0; ein Kreis

db 5,2,0; fünf Schritte mit Vy=2
db 5,2,-1; fünf Schritte mit Vy=2, Vx=-1
db 5,0,-1
db 5,-2,-1
db 5,-2,0
db 5,-2,1
db 5,0,1
db 5,2,1
db 0

ROUTE1; links-rechts

db 10,0,-1
db 10,0,1
db 0

ROUTE2; auf-ab

db 10,-2,0
db 10,2,0

```

db 0

ROUTE3; eine
Acht
;-----db 15,2,0-----
db 5,2,-1
db 5,0,-1
db 25,-2,-1
db 5,0,-1
db 5,2,-1
db 15,2,0
db 5,2,1
db 5,0,1
db 25,-2,1
db 5,0,1
db 5,2,1
db 0

```

ROUTE4; eine Schleife und geht nach links

```

;-----db 120,0,-1
db 10,-2,-1
db 20,-2,0
db 10,-2,1
db 5,0,1
db 10,2,1
db 20,2,0
db 10,2,-1
db 80,0,-1
db 0

```

Um nun die Routen von BASIC aus zu verwenden, weisen wir die Route einfach einem Sprite mit dem Befehl SETUPSP zu und geben an, dass wir den Parameter 15 ändern wollen, der die Route angibt. Außerdem müssen wir das Routenflag (Bit 7) im Statusbyte des Sprites aktivieren und es mit dem automatischen Bewegungsflag und den Animations- und Druckflags setzen.

```

10 SPEICHER 24999
11 BEI PAUSE GOSUB 280
20 MODUS 0:TINTE 0,0
21 LOCATE 1,20:PRINT "Befehl |ROUTEALL und Animation Makrosequenzen".
30 CALL &6B78:DEFINT a-z
31 |SETLIMITS,0,80,0,200
40 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT
41 x=10
50 FOR i=1 TO 8
51 x=x+20:IF x>=80 THEN x=10:y=y+24
60 |SETUPSP,i,0,143: rem Damit wird das Routenflag aktiviert
70 |SETUPSP,i,7,2:|SETUPSP,i,7,33: rem Makro-Animationssequenz
71 |SETUPSP,i,15,3: neu zugewiesene Routennummer 3
80 |LOCATESP,i,30,70
82 FOR t=1 TO 10:|ROUTEALL:|AUTOALL,0:|PRINTSPALL,1,0:NEXT
91 NÄCHSTES
100 |AUTOALL,1:|PRINTSPALL,1,0: rem hier ruft AUTOALL bereits ROUTEALL auf
120 GOTO 100

```

280 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

Wir haben alles im Griff. Diese fortschrittliche Technik wird Ihre Programmierung erheblich vereinfachen und zu spektakulären Ergebnissen führen.



Abb. 69 eine 8-förmige Route

Wie Sie gesehen haben, ändert der Befehl die Koordinaten der Sprites nicht, so dass sie mit **|AUTOALL** bewegt und mit **|PRINTSPALL** gedruckt (und animiert) werden müssen. Deshalb gibt es einen optionalen Parameter in **|AUTOALL**, so dass **|AUTOALL,1** intern **|ROUTEALL** aufruft, bevor das Sprite verschoben wird, was Ihnen einen BASIC-Aufruf erspart, der immer eine kostbare Millisekunde dauert.

12.1 Platziert ein Sprite in der Mitte einer Route : ROUTESP

Wenn Sie mehrere Sprites der gleichen Route zuweisen und Sie wollen, dass sie alle in eine einzige Datei gehen, wie im obigen Beispiel, dann müssen Sie sicherstellen, dass sich jedes Sprite an einem anderen Punkt der Route befindet. Es gibt zwei Möglichkeiten, dies zu tun

Die erste besteht darin, den Sprites nach und nach die Route zuzuweisen. Auf diese Weise wird das führende Sprite als erstes geroutet, und nach ein paar Spielzyklen wird das nächste geroutet, dann das nächste, und so weiter. In jedem Zyklus führen Sie **|AUTOALL,1** aus, und das leitet die Sprites, denen bereits eine Route zugewiesen wurde, die in der Anzahl der Schritte gegenüber den Sprites, denen noch keine Route zugewiesen wurde, voraus sind.

Die zweite Möglichkeit ist die Verwendung des Befehls
|ROUTESP.

|ROUTESP, <spriteid>, <steps>.

Dieser Befehl verschiebt ein Sprite um die gewünschte Anzahl von Schritten (**bis zu 255 Schritte**) entlang des ihm zugewiesenen Pfades. Im Beispiel habe ich die Zuweisung von Pfad 8 und die **|ROUTESP-Befehle**, die jedes der Sprites entlang des Pfades positionieren, rot hervorgehoben.

```
10 Speicher 24999
10 BEI PAUSE GOSUB 12
11 GOTO 20
12 |MUSIC:CALL &BC02:PAPER 0:OPEN 1:MODE 1:END
20 AUFRUF &BC02:DEFINT A-Z:MODE 0
```

50 AUFRUF &6B78

70 ' alle Schiffe an der gleichen Anfangskoordinate

75 ' und mit der gleichen Strecke, aber mit einer anderen Anfangsanzahl von Schritten.

```

76 locate 2,3: Druckt "Befehl |ROUTEESP "
80 für s=16 bis 21: |SETUPSP,s,9,33: |SETUPSP,s,0,137:
|SETUPSP,s,15,8: |LOCATESP,s,20,100:weiter
90 s=21: |ROUTEESP,s,40:'Schiffskopf
100 s=20: |ROUTEESP,s,30
110 s=19: |ROUTEESP,s,20
s=18: |ROUTEESP,s,10
130 s=17: |ROUTEESP,s,0
131 |PRINTSPALL,0,0,0,0
140 --- Spielzyklus ---
150 |AUTOALL,1:|PRINTSPALL
160 Gehe zu 150

```

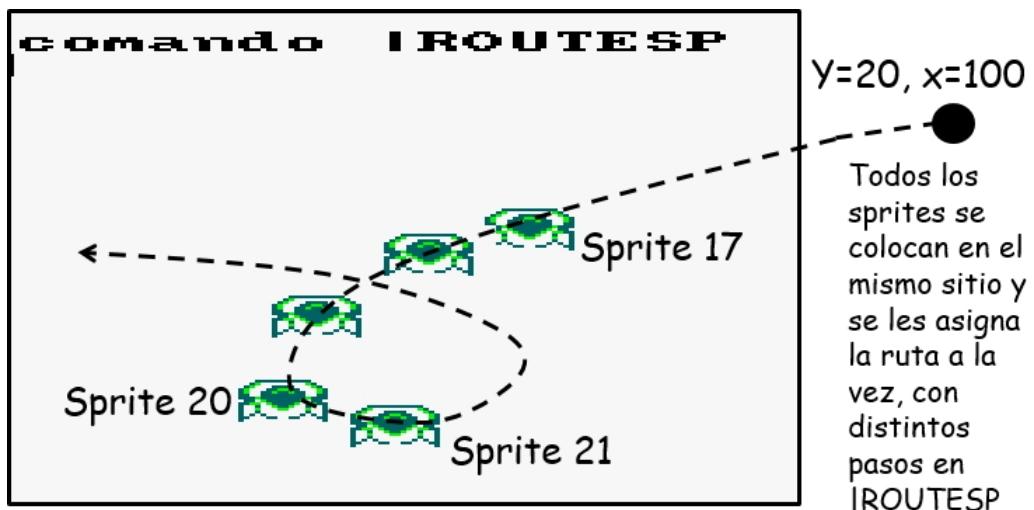


Abb. 70 Reihenhäuser durch den Einsatz von ROUTESP

Route 8 würde in der Datei routes_mygame.asm wie folgt definiert werden:

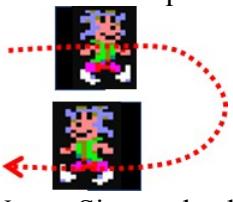
```

ROUTE8;
db 255, 128+64+32+32+8+1,0; Zustandsänderung
db 70,1,-1
db 10,3,-1
db 5,3,0
db 5,3,1
db 5,0,1
db 20,-3,1
db 5,0,1
db 5,3,1
db 5,3,0
db 10,3,-1
db 5,0,-1
db 20,-3,-1
db 40,-1,-1
Neuausrichtu
ng db 1,0,115
db 1,-30,0
db 120,0,0
db 120,0,0
db 0

```

12.2 Erweiterte Routen erstellen

Es gibt 4 Funktionen, die Sie in der Mitte jeder Route (**Achtung, in der Mitte, nicht am Ende**) verwenden können, indem Sie einen Escape-Code als Wert für die Anzahl der Schritte eines Abschnitts verwenden:

Fluchtcod e	Beschreibung	Beispiel
255	Änderung des Sprite-Zustands.	DB 255, 3, 0 Die Null am Ende ist ein Füllwort.
254	Änderung der Animationssequenz  Wenn Sie nach der Änderung der Reihenfolge auch das Bild ändern möchten, müssen Sie den Code 251 verwenden	DB 254, 10, 0 Die Folge 10 ist assoziiert. Die Null ist ein Füller Wenn die zugewiesene Sequenz diejenige ist, die das Sprite bereits hat, dann ist es harmlos (kein Zurücksetzen der Frame-ID). Falls Sie die Frame-ID zurücksetzen wollen, muss der dritte Parameter z.B. eine 1 sein: DB 254, 10, 1
253	Änderung des Images 	DB 253 DW new_img Es wird das Bild "new_img" zugeordnet, das eine Speicheradresse sein muss.
252	Änderung der Route	DB 252,2,0 Route 2 ist verbunden mit
251	Weiter zu zu nächsten Rahmen von der Animation. 	DB 251,0,0 Das Sprite ist animiert. Die zwei Nullen sind Füller

WICHTIG: Achten Sie sehr darauf, DB und DW dort zu schreiben, wo sie verwendet werden sollen, d.h. wenn Sie z.B. Bilder ändern, sollten Sie dem Bild DW und nicht DB voranstellen. Wenn Sie einen solchen Fehler machen, wird Ihre Route nicht funktionieren.

WICHTIG: Escape-Codes können in der Mitte einer Strecke verwendet werden, aber das letzte Segment kann kein Escape-Code sein, es muss eine Bewegung sein, auch wenn es stillsteht, etwa "DB 1,0,0".

12.2.1 Erzwungene Zustandsänderungen von Routen

Diese Fähigkeit ist sehr interessant, um Ihre Spiele zu beschleunigen und ist seit V27 verfügbar. Es bedeutet, dass wir einen Zustandswechsel in der Mitte einer Strecke erzwingen können. Um dies zu tun, geben wir an, dass wir einen Zustandswechsel wollen, indem wir die Anzahl der Schritte im Segment als Wert = 255 angeben.

Die Statusänderung ist ein weiteres Segment, und es ist wichtig, dass Sie die gleiche Anzahl von Parametern pro Segment, d. h. 3 Byte, beibehalten. Eine Statusänderung auf

status=13 könnte wie folgt geschrieben werden:

255,13,0

Der dritte Parameter (die Null) hat keine Bedeutung, er ist nur ein "Füller", damit das Segment 3 Bytes misst, aber er ist wichtig.

Der Wert 255 teilt dem Befehl **|ROUTEALL mit**, dass er dieses Mal den Zustand des Sprites ändern soll, indem er ihm den unten angegebenen Zustand zuweist. Die Zustandsänderung wird ausgeführt, ohne einen Schritt zu verbrauchen, so dass der nächste Schritt nach der Zustandsänderung immer ausgeführt wird. Wenn wir nicht wollen, dass sich das Sprite weiter bewegt, definieren wir einfach ein Ein-Schritt-Segment ohne Bewegung in X oder Y direkt nach der Zustandsänderung. Schauen wir uns ein Beispiel an:

```
ROUTE3; fire_dere
;-----
    db 40,0,2; vierzig Schritte nach rechts mit Vx=2 db
    255,0,0; Zustandsänderung auf Null
    db 1,0,0; noch Vy=0, Vx=0 db 0

ROUTE4; trigger_izq
;-----
    db 40,0,-2; vierzig Schritte nach links mit Vx=-2 db
    255,0,0; Zustandsänderung auf Null
    db 1,0,0; noch Vy=0, Vx=0 db 0
```

Wir werden diese beiden Wege benutzen, um mit unserer Figur zu schießen. Die erste, nachdem sie 40 Schritte durchlaufen hat, in denen sie 2 Bytes in X vorrückt, erfährt eine Zustandsänderung und das Sprite geht in den Zustand 0, d.h. deaktiviert. Das nächste Segment hat nur einen Schritt und keine Bewegung (vy=0, vx=0).

Mit diesem Mechanismus können wir Trigger auslösen und die Trigger selbst deaktivieren, wenn sie den Bildschirm verlassen. Das spart BASIC-Logik und beschleunigt unsere Spiele. Im folgenden Beispiel ist Bild 26 der Auslöser.

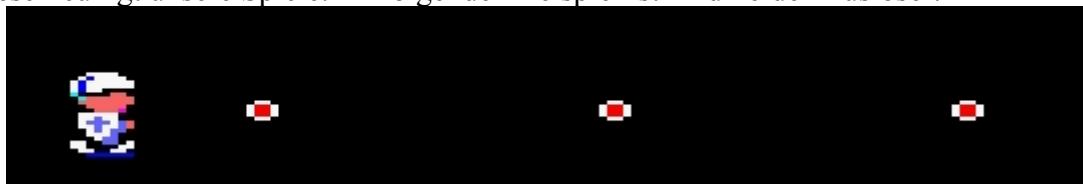


Abb. 71 Auslöser mit Änderung des Unterwegsstatus

```
10 SPEICHER 24999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 BEI PAUSE GOSUB 280
30 CALL &BC02:'restore default palette just in case'.
40 INK 0,0: 'schwarzer Hintergrund
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
80 |SETLIMITS,12,80,0,186: ' die Grenzen des Spielbildschirms
festlegen
90 x=40:y=100:' Koordinaten des Zeichens
100 |SETUPSP,0,0,1:' Zeichenstatus
110 |SETUPSP,0,7,1:dir=1:'Animationssequenz beim Start zugewiesen
120 |LOCATESP,0,y,x:'platziert das Sprite (ohne es zu drucken)
```

```

125 |MUSIC,0,0,6
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'Schüsse
130 Zyklus des Spiels...
150 |AUTOALL,1:|PRINTSPALL,0,0
170 ' Zeichenbewegungsroutine -----
    IF INKEY(27)=0 THEN IF dir=2 THEN dir=1:|SETUPSP,0,7,dir ELSE
    |ANIMA,0:x=x+1:GOTO 191
190 IF INKEY(34)=0 THEN IF dir=1 THEN dir=2:|SETUPSP,0,7,dir ELSE
    |ANIMA,0:x=x-1
191 IF wait<cycle-10 then if INKEY(47)=0 THEN wait=cycle:disp= 1+ disp
mod 4 :|LOCATESP,10+disp,y+8,x: |SETUPSP,10+disp,0,137:
    |SETUPSP,10+disp,15,2+dir
200 |LOCATESP,0,y,x
201 Zyklus=Zyklus+1
210 zu 150
280 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

```

Zustandsänderungen können an jedem Abschnitt der Route erzwungen werden, nicht unbedingt am Ende, obwohl es im Falle eines Auslösers sehr logisch ist, dies am Ende der Route zu tun.

12.2.2 Erzwungene Reihenfolgeänderungen von Routen

Wir können die Animationssequenz eines Sprites mit Hilfe eines speziellen Segments ändern. Wenn wir 254 als Wert für die Anzahl der Schritte eingeben, interpretiert der ROUTEALL-Befehl, dass eine Änderung der Animationssequenz für das Sprite durchgeführt werden soll. Beispiel:

254,10,0

Dieses Segment ändert die Animationssequenz des Sprites und setzt die Sequenznummer 10. Der dritte Parameter (die Null) bedeutet, dass die Sequenz nicht neu gestartet wird. Wenn sich das Sprite also in Frame 5 einer anderen Sequenz befindet, wird die neue Sequenz zugewiesen und der Disco-Frame wird beibehalten, obwohl er jetzt einem anderen Bild entspricht. Dies ist sehr nützlich, da wir einem Sprite innerhalb eines Pfades eine Sequenz zuweisen können, und wenn es diese bereits hatte, ist es harmlos. Damit die zugewiesene Sequenz neu gestartet wird (um zu Bild 0 zu gehen), muss man eine 1 in den dritten Parameter setzen:

254,10,1

Wenn Sie eine Sequenz zuweisen und diese neu starten wollen, ist es ratsam, den Animationscode zu verwenden, den 251, den wir später sehen werden. Auf diese Weise ändert sich das mit dem Sprite verbundene Bild in der Sprite-Tabelle und nicht nur die **frame_id**.

Wie bei der Zustandsänderung wird auch die Sequenzänderung ausgeführt, ohne einen Schritt zu verbrauchen, so dass der nächste Schritt der Sequenzänderung immer ausgeführt wird.

12.2.3 Erzwungene Bildänderungen von Routen

Wir haben gesehen, wie man Sprites mit ROUTEALL oder noch besser, mit AUTOALL,1

Oft wollen wir ein Sprite nicht durch eine Flugbahn leiten, sondern etwas Alltäglicheres: mit einer Figur springen. In dem Beispiel in Kapitel 9 haben wir gesehen, wie man das mit einem BASIC-Array macht, das die relativen Bewegungen der Y-Koordinate enthält. In diesem Fall werden wir das Gleiche mit einem Pfad machen, um eine schnellere Bewegung zu erreichen.

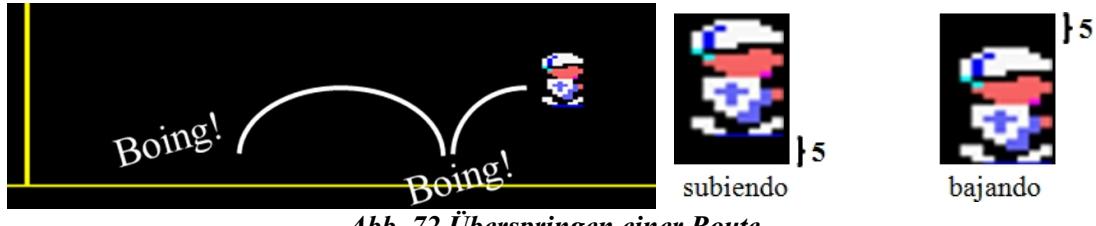


Abb. 72 Überspringen einer Route

Um nicht prüfen zu müssen, ob das Zeichen den Zenit des Sprungs erreicht hat, können wir ein spezielles Segment verwenden, das den Bildwechsel anzeigt. Wie jedes andere Segment verbraucht es 3 Bytes, aber in diesem Fall ist das erste der Bildwechselindikator (ein 253-Wert) und die nächsten beiden entsprechen der Speicheradresse des Bildes. **ACHTUNG**, Sie müssen "**dw**" vor dem Namen des Bildes, das Sie zuweisen wollen, verwenden, also müssen Sie dieses Bildänderungssegment in zwei Zeilen schreiben. Ein "**db**" für die 253 und ein "**dw**" für die Speicheradresse des Bildes.

```
db 253
dw SOLDIER_R1_UP
```

Im folgenden Beispiel haben wir eine springende Puppe. Auf dem Weg nach oben löscht sich die Puppe unten, auf dem Weg nach unten löscht sie sich oben. Damit es keine Unterbrechung in der Bewegung gibt, ist es nur beim Wechsel von einem Bild zum anderen notwendig, den Soldaten vertikal neu auszurichten, indem das untere Bild genau 5 Zeilen angehoben wird, damit es mit dem aufsteigenden Soldaten zusammenfällt.

Dies wäre die Datei sequences_mygame.asm

;=====
bis zu 31 Animationssequenzen
;=====
muss eine feste Tabelle und keine variable Tabelle sein
Jede Sequenz enthält die Adressen der zyklischen Animationsbilder.
Jede Sequenz besteht aus 8 Bildspeicheradressen.
gerade Zahl, da die Animationen normalerweise eine gerade Zahl sind
eine Null bedeutet das Ende der Sequenz, obwohl immer 8 Wörter verwendet werden.
Wenn eine Null gefunden wird, wird ein neuer Anfang gemacht.
wenn es keine Null gibt, beginnt es nach Bild 8 von vorne.

Die Nullfolge bedeutet, dass es keine Folge gibt.
Wir beginnen mit der Sequenz 1

;-----animation sequences -----
SEQUENZEN_LISTE
dw SOLDIER_R0,SOLDIER_R2,SOLDIER_R1,SOLDIER_R2,0,0,0,0,0 ; 1
dw SOLD_L0,SOLD_L2,SOLD_L1,SOLD_L2,0,0,0,0,0 ; 2 dw
SOLD_R1_UP,0,0,0,0,0,0,0,3
dw
SOLDIER_R1_DOWN,0,0,0,0,0,0,0,0;4
dw SOLDIER_L1_UP,0,0,0,0,0,0,0,0;5
dw
SOLDIER_L1_DOWN,0,0,0,0,0,0,0,0;6

MAKRO_SEQUENZEN

-----MAKRO-SEQUENZEN -----

sind Gruppen von Sequenzen, eine für jede Richtung.

; die Bedeutung ist:

still, links, rechts, oben, oben-links, oben-rechts, unten, unten-links,
unten-rechts

Die Nummern sind von 32 an aufwärts nummeriert

**db 0,2,1,3,5,3,4,6,4; 32 --> Soldatensequenzen , id=32. der nächste
wäre 33**

Wir werden zwei Routen verwenden, eine für den Sprung nach rechts und eine für den Sprung nach links. Dies wäre die Datei routes_mygame.asm.

LISTE DER ROUTEN

```
;=====


### hier die Namen aller von Ihnen erstellten Routen eintragen ROUTE_LIST


dw ROUTE0
dw ROUTE1
dw ROUTE2
dw ROUTE3
dw ROUTE4
dw ROUTE4
```

DEFINITION DER EINZELNEN STRECKEN

```
;=====
ROUTE0; jump_right
db 253
dw SOLDIER_R1_UP
db 1,-5,1
db 2,-4,1
db 2,-3,1
db 2,-2,1
db 2,-1,1
db 253
dw SOLDIER_R1_DOWN
db 1,-5,1; up so UP und down fit db 2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDIER_R1
db 1,5,1; gehe noch einen Schritt weiter nach unten, da R1 oben keine
schwarzen Zahlen hat
db 255,13,0; neuer Zustand, ohne Pfadflagge und mit Animationsflagge db
254,32,0; Makrofolge 32
db 1,0,0; quietooo.!!!!
db 0

ROUTE1; jump_left
db 253
dw SOLDADO_L1_UP
db 1,-5,-1
db 2,-4,-1
db 2,-3,-1
db 2,-2,-1
db 2,-1,-1
db 253
dw SOLDIER_L1_DOWN
db 1,-5,-1; anheben, um nach oben und unten zu
passen db 2,1,-1
```

```

db 2,2,-1
db 2,3,-1
db 2,4,-1
db 1,5,-1
db 253
dw SOLDIER_L1
db 1,5,-1; noch eine Stufe tiefer
db 255,13,0; neuer Zustand, ohne Pfadflagge und mit Animationsflagge db
254,32,0; Makrofolge 32
db 1,0,0; stillooo.!!!!
db 0

ROUTE2; Vogel
db 30,0,-1
db 10,0,0
db 20,2,-1
db 20,-2,-1
db 0

ROUTE3; fire_dere
;-----
db 40,0,2
db 255.0
db 1,0,0
db 0

ROUTE4; trigger_izq
;-----
db 40,0,-2
db 255.0
db 1,0,0
db 0

```

Und das wäre das Beispielprogramm. Vergleichen Sie seine Ausführung mit der in Kapitel 7, um den Geschwindigkeitsunterschied zu sehen. Sie werden einen großen

LÖSPEICHER 24999stellen

```

20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 BEI PAUSE GOSUB 2800
30 CALL &BC02:ink 0,0:'stellt die Standardpalette wieder her
50 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:'Sprites zurücksetzen
80 |SETLIMITS,12,80,0,186: ' die Grenzen des Spielbildschirms
festlegen
90 x=40:y=100:jump=0:cycle=40:' Zeichenkoordinaten
100 |SETUPSP,0,0,13:|SETUPSP,0,5,0,0:' Zeichenstatus
110 |SETUPSP,0,7,1:|SETUPSP,0,7,32:'Animationssequenz beim Start
zugewiesen

| LOCATESP,0,y,x:'wir platzieren das Sprite noch nicht drucken)
(ohne
123 Locate 1,1: print "press Q" : print "stop" skip": print "Beispiel
: print "for
mit Strecke"
124 print "SPACE zum Feuern drücken" print
"SPACE zum Feuern drücken" print "SPACE
zum Feuern drücken" print "SPACE zum
Feuern drücken" print
125 PLOT 1,150:ZEICHNEN 640,150: PLOT
92,150:ZEICHNEN
92.400: "Boden und
Wand

```

```
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'Schüsse
|MUSIC,0,0,5: 'Musik beginnt zu spielen
130 ----- "Zyklus des Spiels".
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' Zeichenbewegungsroutine -----
```

```

172 IF INKEY(47)=0 THEN if wait<cycle-10 then wait=cycle:disp= 1+ disp
mod
4:|LOCATESP,10+disp,peek(27001)+8,peek(27003):|SETUPSP,10+disp,0,137:|
SETUPSP,10+disp,15,3+dir
173 if peek(27000)>128 then 193 else |SETUPSP,0,6,0: ' if state is
>128 ist, dass ich springe (hat Route)
174 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'skip
180 IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1:'ir derecha
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1:'ir links
193 Zyklus=Zyklus+1
310 zu 150
2800 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

```

Um zu prüfen, ob der Dummy springt, frage ich einfach den Status per Peek (27000) ab. Auf diese Weise wissen wir, ob das Routing-Flag aktiv ist, und wenn ja, müssen wir nicht durch die Zeilen gehen, die ihn nach links und rechts bewegen.

12.2.4 Erzwungenes Rerouting von Routen

Wir können den Pfad eines Sprites mit Hilfe eines speziellen Segments ändern. Wenn wir 252 in den Wert für die Anzahl der Schritte eingeben, interpretiert der ROUTEALL-Befehl, dass eine Pfadänderung am Sprite vorgenommen werden soll. Beispiel:

252,2,0

Dieses Segment ändert den Pfad des Sprites und setzt die Pfadnummer 2. Der dritte Parameter (die Null) hat keine Bedeutung, er ist nur ein "Füller", damit das Segment 3 Bytes groß wird, aber er ist wichtig.

Wie bei der Zustandsänderung wird die Routenänderung ausgeführt, ohne einen Schritt zu verbrauchen, so dass der nächste Schritt nach der Routenänderung immer ausgeführt wird, nämlich der erste Schritt des ersten Abschnitts der neuen Route.

Da eine Route maximal 255 Bytes lang sein kann und ein Segment 3 Bytes lang ist, kann eine Route maximal 84 Segmente lang sein. Möglicherweise müssen Sie eine noch längere Route erstellen, und in diesem Fall können Sie dies tun, indem Sie das Ende einer Route mit einem Routenwechsel zu einer anderen Route verketten, und Sie können so viele Routen verketten, wie Sie möchten.

12.2.5 Erzwungene Pfadänderungen aus BASIC

Angenommen, Sie möchten, dass Ihre Figur nach rechts springt, und in der Mitte des Sprungs möchten Sie, dass die Figur nach links wechseln und den Sprung fortsetzen kann. Was wir vorschlagen, kann mit dieser Zeichnung dargestellt werden:

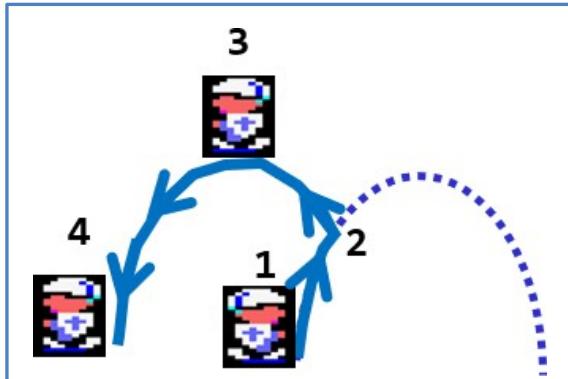


Abb. 73 Routenwechsel in der Mitte eines Sprungs

In diesem Beispiel springt unsere Figur, und in der Mitte der Sprungbahn nach rechts ändert sie bei Punkt 2 die Richtung und setzt die Sprungbahn nach links fort, ohne jedoch einen neuen Sprung zu beginnen. Sie setzt den Sprung einfach fort und erreicht die gleiche Höhe (Punkt 3), die sie beim Sprung nach rechts erreichen würde, um schließlich den Sprung nach links bei Punkt 4 zu beenden.

Um dies zu tun, müssen wir den Pfad zu unserer Figur aus BASIC an Punkt 2 ändern, aber wir können den Befehl **|SETUPSP** nicht verwenden, da dies den Pfad initialisieren würde, was zu einem viel größeren Sprung, beginnend an Punkt 2, führen würde. Was wir in diesem Fall tun können, ist einfach **POKE** an die Speicheradresse, die den Pfad des Sprites speichert, was die Adresse seines +15-Status ist, wie in der Sprite-Attribut-Tabelle gezeigt. Dieser **POKE** ändert den Pfad, lässt aber die Position intakt (Segmentnummer und Position, an der sich das Zeichen befindet). ACHTUNG: Die beiden Pfade müssen die gleichen Segmente und die gleiche Länge haben, denn wenn man zu einem kürzeren Pfad springt und sich auf einem Segment befindet, das es auf diesem Pfad nicht gibt, kann ein unvorhersehbarer Effekt auftreten.

Angenommen, wir haben zwei Sprungpfade (Pfad 0 springt nach rechts und Pfad 1 springt nach links), so veranschaulicht das folgende Beispiel das Konzept:

```

130 ----- "Zyklus des Spiels".
150 Zyklus=Zyklus+1:AUTOALL,1:| PRINTSPALL,0,1,0
170 ' Zeichenbewegungsroutine -----
173 WENN PEEK(27000)<128 DANN 178
174 wir befinden uns mitten in einem Sprung
175 IF INKEY(27)=0 THEN POKE 27015,0:GOTO 180:'Pfad nach rechts ändern
    IF INKEY(34)=0 THEN POKE 27015,1:GOTO 180:'Pfad nach links ändern
    GOTO 150
178 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'skip
    IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1:'ir derecha
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1:'ir links
310 GOTO 150

```

Die einzige Einschränkung bei der Verwendung von **POKE** für diesen Zweck ist, dass sich das Bild des Zeichens nicht ändert, bis es auf einen Bildänderungscode in der Mitte des Pfades trifft. Die Änderung des Bildes mit **|SETUPSP** ist möglich, aber gefährlich, da man nicht weiß, ob das Zeichen nach oben (mit den unteren Zeilen gelöscht) oder nach unten (mit den oberen Zeilen gelöscht) geht. Daher ist es besser, einfach den Pfad zuzuweisen und den Pfad selbst das Bild so schnell wie möglich ändern zu lassen. Sie können sogar in der Mitte der Strecke Bildänderungen vornehmen, auch wenn sie nicht notwendig sind, falls dieser Fall eintritt. Sie müssen allerdings darauf achten, dass das Sprungbild auf beiden Seiten Löschbytes hat, denn wenn es das nicht hat und die

Richtung ändert, wird es eine Spur hinterlassen.

12.2.6 Erzwungene Animation von Routen

Wir können das Animations-Flag eines Sprites inaktiv lassen und es nur zu bestimmten Zeitpunkten des Pfades mit Code 251 animieren:

251,0,0

Dies kann sehr nützlich sein, um in Spielen, die Pseudo-3D-Techniken verwenden, ein Gefühl für ein sich näherndes Sprite zu vermitteln. Zum Beispiel bei einem sich nähernden Meteoriten, den wir in der Größe verändern wollen, damit er größer erscheint. Der Meteorit bewegt sich ein paar Mal, bevor er die nächste Größe annimmt. Dieser Mechanismus ist dem Bildwechsel-Mechanismus sehr ähnlich, mit dem Unterschied, dass er es ermöglicht, den Bildwechsel ohne explizite Angabe des Bildes zu definieren, sondern einfach einen Frame-Wechsel in der dem Sprite zugewiesenen



Animationssequenz anzuzeigen.

Abb. 74 Erzwungene Animation aus der Route

Mit dieser Flagge können Sie die gleiche Route für einen sich nähernden Weltraumvogel oder einen Meteoriten verwenden. Wenn Sie das Bild nicht angeben, wird in jedem Fall das Bild verwendet, das der Sequenz entspricht, die jedes Sprite hat.

12.2.7 Wie man "dynamische" (nicht vordefinierte) Routen erstellt

Eine dynamische Route ist eine Route, deren Pfad zur Laufzeit in Ihrem BASIC-Programm festgelegt wird. Sie ist nützlich, wenn Ihr Programm dynamisch Labyrinthe oder Rennstrecken erzeugt, durch die ein Gegner laufen muss und die nicht im Voraus bekannt sind und daher nicht in der Datei "routes_mygame.asm" definiert werden können.

Um eine solche Route zu erstellen und sie einem Sprite zuzuweisen, müssen wir eine "leere" Route in der Datei "routes_mygame.asm" erstellen, in diesem Fall Route 2.

LISTE DER ROUTEN

;=====

Geben Sie hier die Namen aller Routen ein, die Sie erstellen ROUTE_LIST

dw ROUTE0
dw ROUTE1
dw ROUTE2
dw ROUTE3
dw ROUTE4
dw ROUTE4

DEFINITION DER EINZELNEN STRECKEN

;=====

ROUTE0; rechts links db

**10,0,1
db 10,0,-1
db 0**

ROUTE1; auf ab db

**10,-1,0
db 10,1,0
db 0**

**ROUTE2; dynamische
Leitweglenkung ds**

100

Wir haben Pfad 2 mit 100 freien Bytes erstellt, die von BASIC gefüllt werden können. Es kann sein, dass der Pfad, den wir erstellen, weniger benötigt, und in diesem Fall reicht es aus, weniger Bytes zu reservieren.

Sobald die Bibliothek und die Grafik zusammengebaut sind, müssen wir die Speicheradresse des Labels "ROUTE2" im winape-Symbolfenster suchen. Wenn wir sie haben, programmieren wir in BASIC die Route mit POKE von dieser und den folgenden Speicheradressen

POKE setzt ein Byte in eine Speicheradresse. Unsere Zahlen müssen im Bereich -127..128 liegen und POKE erlaubt keine negativen Zahlen. Um dies zu tun, müssen Sie den positiven Wert verwenden, mit dem der Amstrad intern negative Zahlen darstellt (d.h. das Zweierkomplement). Dazu brauchen Sie nur eine AND 255 Operation

10 A=-10

20 PRINT A: REM druckt eine 10

30 PRINT A AND 255: REM druckt 246, was gleichbedeutend ist mit -10

Kurz gesagt, wenn Sie eine -10 einfügen wollen, müssen Sie eine 246 einfügen und die gleiche Strategie für jede negative Zahl anwenden. Vergessen Sie nicht, dass der letzte POKE der Route die Einfügung einer Null sein muss, was das Ende der Route bedeutet.

12.2.8 Programmierung von Routen einschließlich Mustern

Nehmen wir an, Sie wollen einen Pfad für einen Feind erstellen, der den Bildschirm immer wieder von rechts nach links durchläuft, wobei wir von einer Ausgangsposition ausgehen, in der sich das Sprite ganz rechts auf dem Bildschirm befindet.

ROUTE0; einzelne Route

**DB 80,0,-1 ; 80 Schritte. In jedem Schritt wird 1 Byte verschoben
DB 1,0,80 ; bringe das Sprite wieder an seine
ursprüngliche Position DB 0**

Dieser Pfad bewegt ein Sprite mit einem 1-Byte-Schritt pro Frame. Wenn wir ihn verlangsamen wollten, indem wir jedes zweite Bild 1 Byte bewegen, könnten wir Folgendes tun:

ROUTE0; nicht so einfache

**Strecke DB 1,0,-1
DB 1,0,0
DB 0**

Auf diese Weise können wir ein Sprite langsamer bewegen, aber wir können das Sprite nicht an seine ursprüngliche Position zurückbringen. Da der Bildschirm 80 Byte breit ist, benötigen wir 160 Segmente, da das Sprite alle zwei Segmente um 1 Byte vorrückt. Der sich daraus ergebende Pfad wäre sehr lang und wir müssten 2 Pfade miteinander verbinden, da ein Pfad nur 255 Byte (84 Segmente) groß sein kann. Die optimale Lösung ist, einen kurzen Pfad zu definieren, ohne das Sprite neu zu positionieren, und es von BASIC aus neu zu positionieren, etwa so:

```
80 |SETUPSP,31, 0, 128+16+1: REM routingfähig, automatisch mov, druckbar
85 |LOCATESP,31,100,80: ' auf der rechten Seite des Bildschirms platziert.
90 rem Spielzyklus
100 Zyklus=Zyklus +1
110 |AUTOALL,1: |PRINTSPALL
120 if MOD cycle 160=0 THEN |LOCATESP,31,100,80: ' Neupositionierung
130 goto 100
```

Diese Art von Strategien sind immer dann nützlich, wenn wir nicht in jedem Bild dieselbe Bewegung wiederholen wollen, sondern ein sich wiederholendes Muster definieren wollen, bei dem sich das Sprite in bestimmten Bildern in eine Richtung und in anderen in eine andere Richtung oder sogar überhaupt nicht bewegt. Schauen wir uns ein anderes Beispiel an, einen schrägen Pfad, bei dem wir für jeweils 3 vertikale Bewegungen eine horizontale Bewegung ausführen.

```
ROUTE0; schräge Strecke
DB 2,1,0
DB 1,1,1,1
DB 0
```

12.2.9 Typologie der Routen

Mit all dem, was wir gesehen haben, können wir die Routen in die folgenden Typen einteilen:

- **Zyklische Endlospfade:** Sie positionieren das Sprite neu oder enden einfach an denselben Koordinaten, an denen sie begonnen haben.
- **Nicht-zyklische Endlospfade:** Sie bewegen sich unendlich weiter und positionieren das Sprite nicht neu, so dass sie sich unendlich weit vom Spielbereich entfernen können, es sei denn, wir positionieren das Sprite in BASIC neu.
- **Routen mit Ende:** Im letzten Schritt ändern Sie den Zustand des Sprites, indem Sie das Routing-Flag deaktivieren.
- **Verkettete Routen:** Von einer Route kann man zu einer anderen Route springen, und diese zweite Route kann zyklisch oder nicht zyklisch sein oder ein Ende haben oder sogar zu einer dritten Route führen.

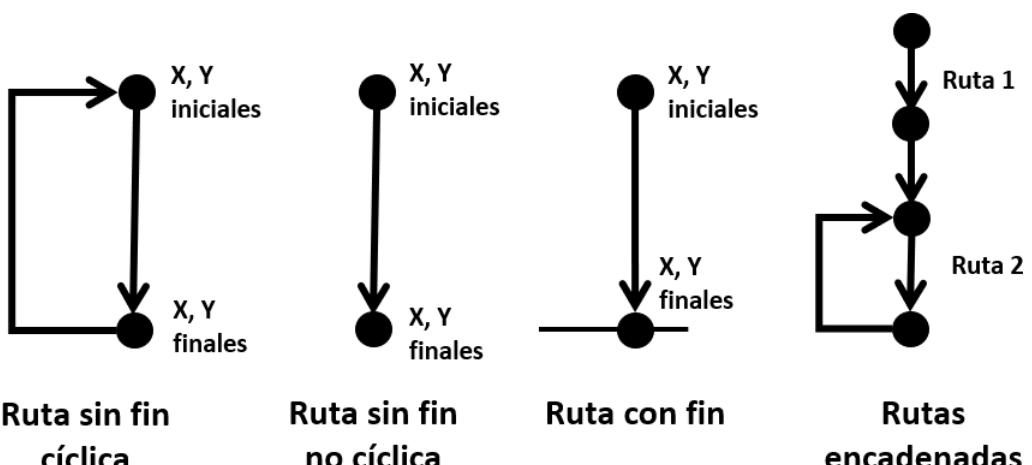
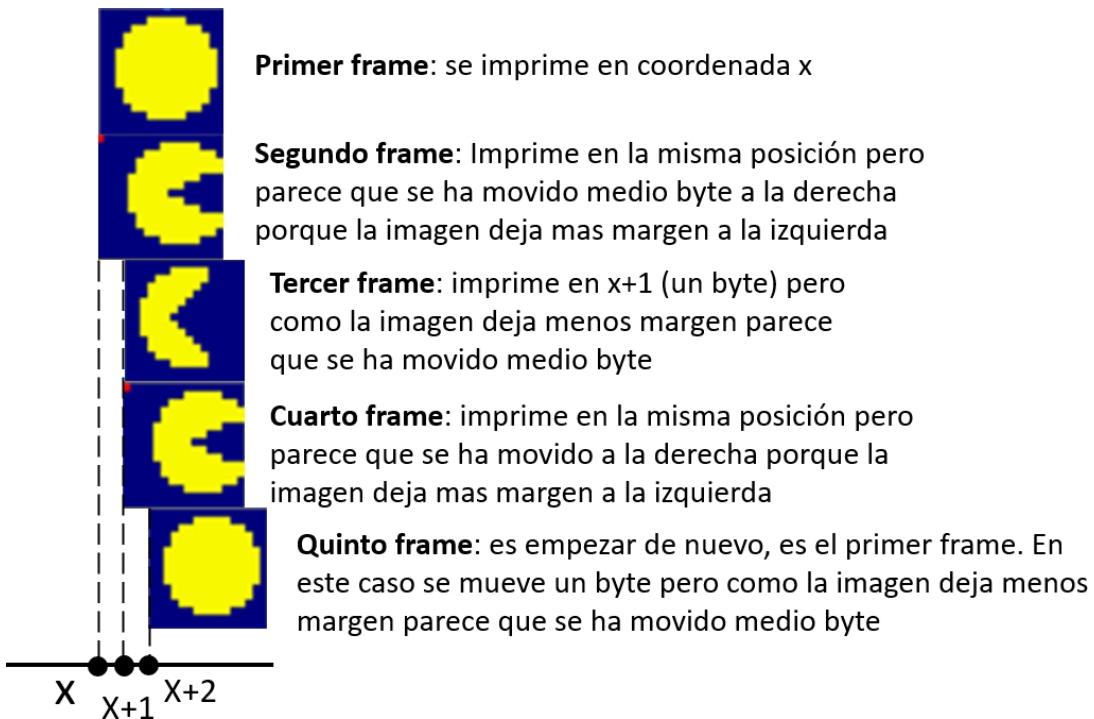


Abb. 75 Arten von Routen

13 Fließende Halbbyte-Bewegung

8BP bewegt die Sprites byteweise mit Befehlen wie MOVE, und sein Koordinatensystem ist byteweise, nicht pixelweise. Daher sprechen wir von 80 Positionen auf der horizontalen Achse und 200 auf der vertikalen Achse.

Ein Byte enthält 2 Pixel im Modus 0 oder 4 Pixel im Modus 1. Wir möchten vielleicht eine sanftere Bewegung (Pixel für Pixel im Modus 0 oder zwei Pixel im Modus 1). Es gibt einen einfachen Trick, den wir dafür verwenden können. Er besteht darin, ein um ein halbes Byte verschobenes Bild des Zeichens zu haben und es einfach dem Sprite zuzuweisen. Selbst wenn es an der gleichen Koordinate gedruckt wird, sieht es so aus, als hätte es sich bewegt.



In dieser Animationssequenz gibt es Zeiten, in denen sich der Pac-Man nicht bewegt, aber wenn wir das Bild verändern, ist es so, als ob er sich 2 Pixel (ein halbes Byte) bewegt. In den Momenten, in denen er sich um 1 Byte bewegt, erscheint das Bild um ein halbes Byte nach links verschoben, so dass das "Netto"-Ergebnis ebenfalls so aussieht, als hätte er sich um 2 Pixel (ein halbes Byte) bewegt. Um die Bewegung dieses Pac-Man zu definieren, können wir den Mechanismus der 8BP-Pfade verwenden. Dies wäre das Beispiel für den Pfad der Bewegung nach rechts:

```
ROUTE0; rechts db
253
dw COCO_R1
db 1,0,0
db 253
dw COCO_R2
db 1,0,1
db 253
dw COCO_R1
db 1,0,0
db 253
dw COCO_R0
db 1,0,1
db 0
```

Das Zeichnen von verschobenen Bildern kann ein wenig mühsam sein. Deshalb gibt es seit **SPEDIT** Version **V15** einen Mechanismus, um ein Bild um ein halbes Byte (2 Pixel im Modus 1 oder ein Pixel im Modus 0) nach rechts oder links zu verschieben. Wie Sie sehen können, erscheint im SPEDIT-Menü eine neue Option: mit den Pfeiltasten können Sie das gezeichnete Bild verschieben



```
SPEDIT: Sprite Editor v15.0
---- CONTROLES EDICION ----
1,2 : tinta -/+      gano, space : mueve u pinta
d: desplaza medio byte
n: flip horizontal
v: flip vertical
c: clear sprite
b: imprime bytes (asm) en printer
i: imprime paleta en printer y file
r: reload 20000
z, x: cambia color de tinta
t: RESET
-----
Antes de empezar puedes cargar un sprite
ensamblandolo en la direccion 20000.
No ensambles ancho y alto, solo los bytes
del dibujo.
La paleta custom esta en la linea 2300
La paleta la puedes cambiar para que sea
la misma que la de tu videojuego.
EDITAR sprite(1) o CAPTURAR sprite(2)? ■
```

Dies ermöglicht es uns, ein Bild leicht zu bewegen, um die beschriebene Technik der sanften Bewegung anzuwenden.

14 Scrollende Spiele

Die 8BP-Bibliothek ermöglicht das Blättern auf verschiedene Arten, die gleichzeitig kombiniert werden können, wobei die wichtigste Methode auf dem Befehl **|MAP2SP basiert**. Die verfügbaren Techniken sind im Folgenden zusammengefasst:

- **Mit Hilfe von Sprite-Block-Bewegungsbefehlen:** Eine einfache Möglichkeit, mit 8BP zu scrollen, besteht darin, einfach dekorative Sprites zu erstellen, deren Zustand wir mit den Befehlen **|MOVEALL** und/oder **|MOVEALL** so einstellen, dass sie sich bewegen.
|AUTOALL.
- **Verwendung von |MAP2SP:** Die Idee hinter dem multidirektionalen Scrollen, das durch **|MAP2SP** in 8BP ermöglicht wird, ist einfach: Alle Elemente, die auf dem Bildschirm dargestellt werden, sind Sprites, also sind die Elemente der Welt, die wir ausdrucken und auf dem Bildschirm bewegen werden, Sprites, deren zugehörige Bilder Berge, Häuser, Bäume oder was auch immer Sie brauchen, um Ihre "Welt" aufzubauen. Um einen Teil der Welt auszuwählen und ihn in eine Liste von Sprites zu verwandeln, wird die Funktion **|MAP2SP** verwendet. Mit der Funktion **|UMAP** können Sie die Welt mit einem Teil einer größeren Welt aktualisieren.
- **Mit dem Befehl |STARS** können Sie einen multidirektionalen Bildlauf mit einer Bank von 40 Pixeln durchführen, die Sie an beliebiger Stelle platzieren und in verschiedenen Ebenen und mit unterschiedlichen Geschwindigkeiten bewegen können.
- **Mit dem Befehl |RINK** können Sie ein Tintenmuster drehen und so ein Gefühl der Vorwärtsbewegung erzeugen, das Sie für bestimmte Arten von Schriftrollen verwenden können, z. B. für die Bewegung eines Steinbodens, von Wasser usw.

14.1 STARS: Schriftrolle mit Sternen oder gesprengelter Erde

In der 8BP-Bibliothek gibt es eine sehr einfach zu bedienende Funktion, mit der man einen Hintergrundeffekt aus sich bewegenden Sternen erzeugen kann, der den Eindruck eines Scrollings vermittelt. Dies ist die Funktion

|STARS. Mit dieser Funktion können Sie bis zu 40 Sterne gleichzeitig verschieben, ohne Ihre Sprites zu verändern, so dass es so ist, als würden sie "darunter" vorbeigehen.

|STARS,<Einstiegsstern>,<Anzahl Sterne>,<Farbe>,<dy>,<dx>,<dx>.

Sie verfügen über eine Bank von Sternen und können mehrere STARS-Befehle kombinieren, um mit Gruppen von Sternen mit unterschiedlichen Geschwindigkeiten zu arbeiten, was ein Gefühl von Ebenen mit unterschiedlichen Tiefen vermittelt.

Die Sternbank besteht aus 40 Bytepaaren, die (y,x)-Koordinaten darstellen. Sie belegen die Adressen 42540 bis 42619 (insgesamt 80 Bytes). Eine Möglichkeit, 40 zufällige Sterne zu erzeugen, wäre (beachten Sie, dass, wenn wir bereits DEFINT A-Z ausgeführt haben, die Zahl 42540 in hexadezimaler Form eingegeben werden muss, da sie größer als 32768 ist).

**FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200: POKE
dir+1,RND*80:NEXT**

Für eine detaillierte Beschreibung der Befehl, siehe das Kapitel "Referenzhandbuch". In diesem Kapitel finden Sie verschiedene Beispiele für die Simulation von Sternen, Erde, Sternen mit zwei Tiefenebenen, Regen oder sogar Schnee. Mit etwas Fantasie lassen sich mit dieser Funktion wahrscheinlich noch mehr Dinge simulieren. Wenn Sie z.B. die Sterne in Sequenzen von 2 oder drei Pixeln diagonal platzieren, anstatt sie zufällig zu verteilen, können Sie eine "segmentale" Bewegungsverschiebung erreichen, was ideal sein könnte.

um Regen zu simulieren.

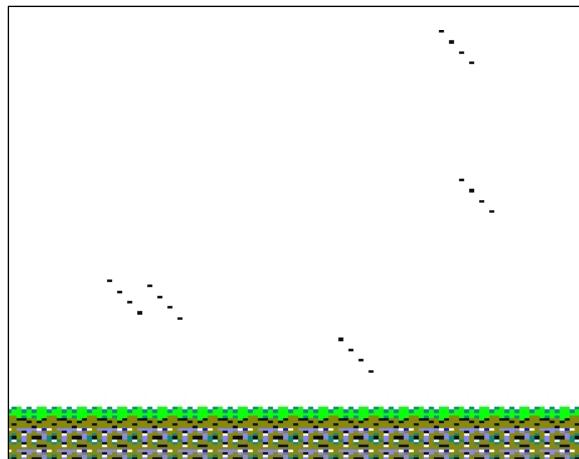


Abb. 76 Regeneffekt mit STARS

```

10 SPEICHER 24999
20 CALL &6B78:' install RSX
40 mode 0:CALL &BC02:'restore default palette just in case'.
50 Bank=42540
60 FOR dir=bank TO bank+40*2 STEP 8:
70 y=INT(RND*190):x=INT(RND*60)+4
80 POKE dir,y:POKE dir+1,x:
90 POKE dir+2,(y+4):POKE dir+3,x-1
100 POKE dir+4,(y+8):POKE dir+5,x-2
110 POKE dir+6,(y+12):POKE dir+7,x-3
120 NÄCHSTE

140 'REGENSZENARIO
141 '-----
150 |SETLIMITS,0,80,50,200: ' Spielbildschirmgrenzen
151 grass=&84d0:|SETUPSP,30,9,grass:'Buchstabe Y ist Sprite 31
152 rocks=&84f2:|SETUPSP,21,9,rocks: 'Buchstabe P ist Sprite 21
160 string$="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY".
170 |LAYOUT,22,0,@cadena$: 'das malt das Gras
180 string$="PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP"
190 |LAYOUT,23,0,@string$: 'malt eine Reihe von Steinen
200 |LAYOUT,24,0,@string$: 'malt eine weitere Reihe von Steinen
210 '----- Spielzyklus-----
211 defint a-z
220 LOCATE 1,10:PRINT "RAIN DEMO".
221 LOCATE 1,11:PRINT "press ENTER".
230 |STARS,0,40,4,4,2,-1
240 IF INKEY(18)=0 THEN 300
250 GOTO 230

```

Da das Beispiel einer doppelten Sternenebene im Referenzkapitel der Bibliothek zu finden ist, hier ein Beispiel eines Raumschiffs, das über einen gesprengelten Erdplaneten fliegt, mit einem vertikalen Bildlauf.

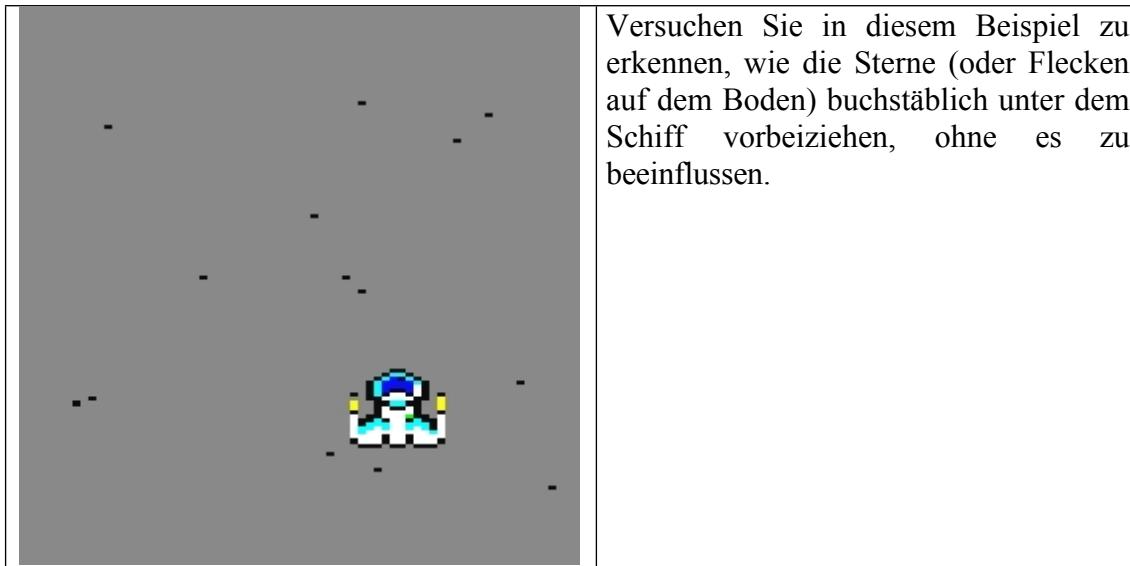


Abb. 77 Fleckiger Bodeneffekt mit STARS

Es gibt eine Möglichkeit, den STARS-Befehl auf optimierte Weise aufzurufen, und sie besteht darin, ihn einfach das erste Mal mit Parametern und die folgenden Male ohne Parameter aufzurufen. Der Befehl geht davon aus, dass die Werte der Parameter die gleichen sind wie beim letzten Aufruf mit Parametern, und das spart Zeit, die der BASIC-Interpreter für die Verarbeitung der Parameter aufwendet, bis zu 1,7 ms.

```

10 SPEICHER 24999
11 'Ich habe zufällige Sterne
12 FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200:POKE dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restore default palette just in case'.
26 Tinte 0.13:'grauer Hintergrund
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
40 |SETLIMITS,12,80,0,186: 'Grenzen des Spielbildschirms

41 ' wir werden ein Schiff in Sprite 31 erstellen
42 |SETUPSP,31,0,&1:' Status
43 Schiff = &a2f8: |SETUPSP,31,9,Schiff:' Bild dem Sprite 31 zuweisen
44 x=40:y=150: ' Schiffskoordinaten

49 '----- Spielzyklus-----
50 |STARS,0,20,5,1,0:' schwarze Sterne auf grauem Grund
55 gosub 100:' Bewegung des Schiffes
60 |PRINTSPALL,0,0
70 goto 50

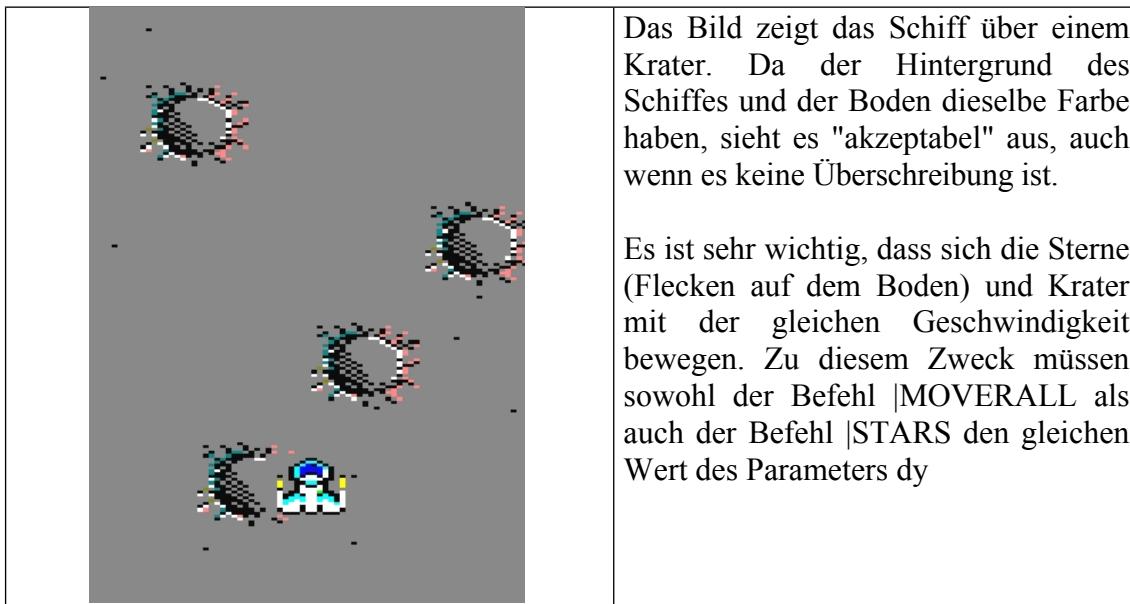
99 ' Routinebewegung Schiff -----
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RÜCKKEHR

```

14.2 Blättern mit MOVEALL und/oder AUTOALL

Indem wir nun die relative Bewegung, das Scrollen der Sterne und die Reihenfolge, in der die Sprites gedruckt werden, kombinieren, werden wir uns ein Beispiel ansehen, wie man ein Raumschiff simulieren kann, das über eine Mondlandschaft fliegt.

Zunächst einmal haben wir für unser Schiff das Sprite 31 gewählt, weil es dann als letztes gedruckt wird. Die Sprites werden in der Reihenfolge gedruckt, beginnend bei Null und endend bei 31. Wenn ein Krater ein Sprite ist, das niedriger als 31 ist, wird er vor dem Schiff gedruckt und das Schiff befindet sich "darüber", so dass der Eindruck entsteht, dass es darüber fliegt.



Das Bild zeigt das Schiff über einem Krater. Da der Hintergrund des Schiffes und der Boden dieselbe Farbe haben, sieht es "akzeptabel" aus, auch wenn es keine Überschreibung ist.

Es ist sehr wichtig, dass sich die Sterne (Flecken auf dem Boden) und Krater mit der gleichen Geschwindigkeit bewegen. Zu diesem Zweck müssen sowohl der Befehl |MOVEALL als auch der Befehl |STARS den gleichen Wert des Parameters dy

Abb. 78 Flug über den Mond

Dies ist der BASIC-Code:

```
10 SPEICHER 24999
11 'Ich habe zufällige Sterne
12 FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restore default palette just in case'.
26 Tinte 0.13:'grauer Hintergrund
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'Sprites zurücksetzen
40 |SETLIMITS,12,80,0,186: ' Grenzen des Spielbildschirms

41 ' wir werden ein Schiff in Sprite 31 erstellen
42 |SETUPSP,31,0,&1:' Status
43 Schiff = &a2f8: |SETUPSP,31,9,Schiff:' Bild dem Sprite 31 zuweisen
45 x=40:y=150: ' Schiffskoordinaten

46 ' jetzt die Krater
47 crater=&a39a: cy%=0
48 for i=0 to 3 : |SETUPSP,i,9,crater:
49 |SETUPSP,i,0,&x10001: ' Eindruck und relative Bewegung
50 x(i)=rnd*40+20:y(i)=i*40
60 |Ortep,i,y(i),x(i)
70 nächste
```

71 t=0

80 '----- Spielzyklus-----

81 |STARS,0,20,5,3,0:' schwarze Sterne Bewegung

82 gosub 100:' Bewegung des Schiffes

83 |MOVERALL,3,0: 'Kraterbewegung'.

84 t=t+1: if t> 10 then t=0:gosub 200:' Kraterkontrolle

90 |PRINTSPALL,0,0:' Schiffs- und Kraterausdruck

91 goto 81

99 ' Routinebewegung Schiff -----

100 IF INKEY(27)=0 THEN x=x+1:GOTO 120

110 IF INKEY(34)=0 THEN x=x-1

120 |LOCATESP,31,y,x

130 RÜCKKEHR

199 ' Krater-Wiedereintrittskontrolle

200 c=c+1: wenn c=6 dann c=0

220 |PEEK,27001+c*16,@cy% 220 |PEEK,27001+c*16,@cy%

230 if cy%>200 then |POKE,27001+c*16,-20

240 Rückgabe

Schauen wir uns ein letztes Beispiel an, bei dem die relative Bewegung genutzt wird, um das Gefühl des Scrollens zu vermitteln. Dabei werden Sprites mit Zeichnungen von Häusern, ein gesprenkelter Boden und eine Figur in der Mitte verwendet, die je nach Bewegungsrichtung alles um sich herum bewegt. Es ist ein sehr einfaches Beispiel, aber es gibt Ihnen eine Vorstellung vom Potenzial dieser Funktionen - hier ist es die ganze Stadt, die sich bewegt!



Abb. 79 Das ganze Dorf bewegt sich

10 SPEICHER 24999

20 MODE 0: Aufruf &6b78

30 DEFINT a-z

240 TINTE 0,12

241 Grenze 7

250 FOR i=0 TO 31

260 |SETUPSP,i,0,&X0

270 NÄCHSTE

280 FOR i=0 TO 3

```

|SETUPSP,i,0,&X10001
|SETUPSP,i,9,&A01c:rem Häuser
301 |LOCATESP,i,RND*150+50,rnd*60+10
310 NÄCHSTES
320 |SETUPSP,31,7,6: rem Zeichen
330 |LOCATESP,31,90,38
340 |SETUPSP,31,0,0,&X1111
xa=0:ya=0
410 IF INKEY(27)=0 THEN xa=-1:
420 IF INKEY(34)=0 THEN xa=+1:
430 IF INKEY(67)=0 THEN ya=+2
440 IF INKEY(69)=0 THEN ya=-2
450 |MOVERALL,ya,xa
460 |PRINTSPALL,1,0
470 |STARS,1,20,5,bereits,xa
GOTO 410

```

14.3 Technik des "Spotting"

Die Technik zum Malen von Bergen in Spielen mit horizontalem Scrolling und Seen in Spielen mit vertikalem Scrolling ist dieselbe. Wir werden nur den Anfang des Berges malen, indem wir ein Sprite verwenden, um seine linke Seite zu malen. Wir fügen so viele ein, wie wir wollen. In diesem Fall habe ich drei

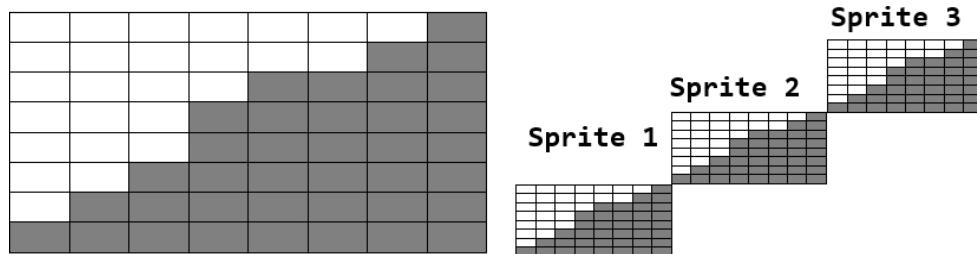


Abb. 80: Definieren des Abhangs eines Berges mit mehreren Sprites

Wir machen das Gleiche mit einem Spiegelbild, das wir mit 3 anderen Sprites verbinden, und wir platzieren sie auf der rechten Seite, um die rechte Seite des Berges zu bauen. Stellen Sie sicher, dass das Spiegelbild mindestens die letzten beiden Pixelspalten auf Null hat. So kann es sich selbst auslöschen, wenn es sich nach links bewegt. Beachten Sie, dass sich Sprites in 8BP byteweise bewegen (jeweils zwei Pixel auf einmal).

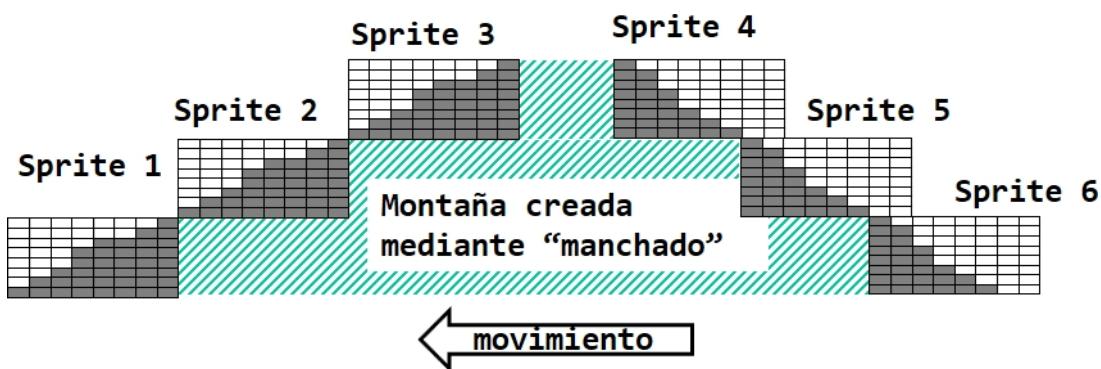


Abb. 81 Mit 6 Sprites und der Technik des "Spotting" erstellter Berg.

Wenn Sie alle Sprites automatisch oder relativ nach links verschieben, beginnen die

Sprites auf der linken Seite, den Hintergrund zu "verwischen" und somit

den Berg "füllen", während die Sprites auf der rechten Seite beginnen, ihn aufzuräumen. Wenn der Berg langsam auf dem Bildschirm erscheint, sieht er wie ein riesiges Berg-Sprite aus, obwohl es in Wirklichkeit nur 6 kleine Sprites sind.

Das Videospiel "Nibiru" verwendet die Technik des "Verschmierens", um die Berge und andere große Elemente zu zeichnen, in Kombination mit dem Befehl MAP2SP, den wir später sehen werden.

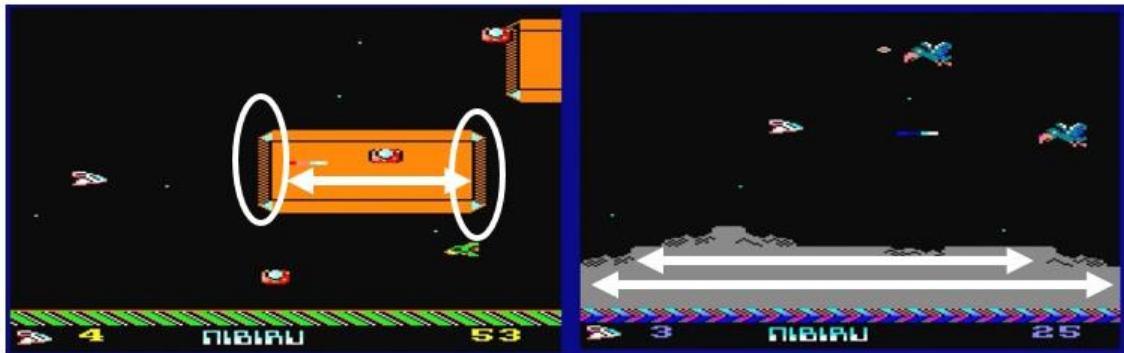


Abb. 82: Beispiele für die Färbetechnik

Ich habe die Verwischungstechnik auch in dem Videospiel "Eridu" verwendet, bei dem sich riesige Berge fließend über den Bildschirm bewegen.

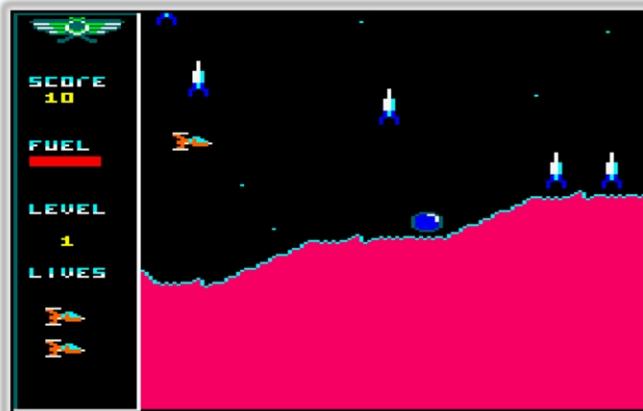


Abb. 83: Färbetechnik in "Eridu".

Wenn wir in einem vertikal scrollenden Spiel einen See auf ein braunes Gelände malen wollen, werden wir das Gleiche tun: einige Sprites werden das Gelände "beflecken" und andere, die weiter entfernt sind, werden es "säubern", so dass es wie ein riesiger See aussieht, in einem Stück.

Es gibt nur eine Vorsichtsmaßnahme, die Sie ergreifen müssen, und zwar, dass die Schiffe nicht über den See fliegen, sonst wird Ihr "Trick" aufgedeckt! Wenn Sie wollen, dass der See überflogen werden kann, müssen Sie die Sprite-Überschreibung verwenden, wie in Phase 2 des Videospiels "Nibiru", wo Ihr Schiff und die gegnerischen Schiffe große farbige Rechtecke überfliegen können, ohne sie zu zerstören.

14.4 MAP2SP: Blättern auf der Grundlage einer Weltkarte

Alle vorhergehenden Techniken sind für das Scrollen vollkommen gültig und sogar kompatibel mit dem, was wir jetzt sehen werden, nämlich die grundlegende Technik, die es dir erlaubt, eine "Weltkarte" zu entwerfen und deinen Charakter oder dein Schiff mit nur einer Zeile Code durch sie scrollen zu lassen.

Um den Befehl **|MAP2SP** zu verwenden, muss die "**Assembly-Option**" 2 gewählt werden, die uns 24,8 KB für das BASIC-Listing zur Verfügung stellt.

Die Idee ist einfach: Wir erstellen eine Liste von Elementen, aus denen die Weltkarte besteht (bis zu 82 Elementen, die wir "Kartenelemente" nennen). Jedes Element wird durch die Y,X-Koordinaten beschrieben, an denen es sich befindet, und durch die Speicheradresse, an der sich das Bild des betreffenden Elements befindet (ein Haus, ein Baum usw.). Das mit einem Kartenelement verbundene Bild kann eine beliebige Größe haben. Die Koordinaten eines jeden Elements sind eine positive ganze Zahl zwischen 0 und 32000.

Sobald die Karte erstellt ist, rufen wir die Funktion auf:

|MAP2SP, Yo, Xo

Diese Funktion analysiert die Liste der Elemente in der Welt und bestimmt, welche von ihnen angezeigt werden, wenn die Welt durch Platzierung der unteren Ecke des Bildschirms an den Koordinaten (I, Yo) betrachtet wird. Die Funktion wandelt die "Kartenelemente" in Sprites um, die die Positionen in der Sprite-Tabelle von Null an einnehmen. Dies kann viele oder wenige Sprites verbrauchen, je nach der Dichte der Kartenelemente, die Sie haben. Bei einem späteren Aufruf der gleichen Funktion werden Kartenelemente, die nicht mehr in der Szene vorhanden sind, keine Sprites in der Tabelle verbrauchen, und andere Kartenelemente übernehmen diese. Das bedeutet, dass **die MAP2SP-Funktion eine variable und unbestimmte Anzahl von Sprites verbraucht, abhängig von der Anzahl der Kartenelemente, die zu einem bestimmten Zeitpunkt auf dem Bildschirm sichtbar sind**. In dem folgenden Beispiel würden Sie 3 Sprites verwenden, wenn Sie MAP2SP an den angegebenen Koordinaten aufrufen.

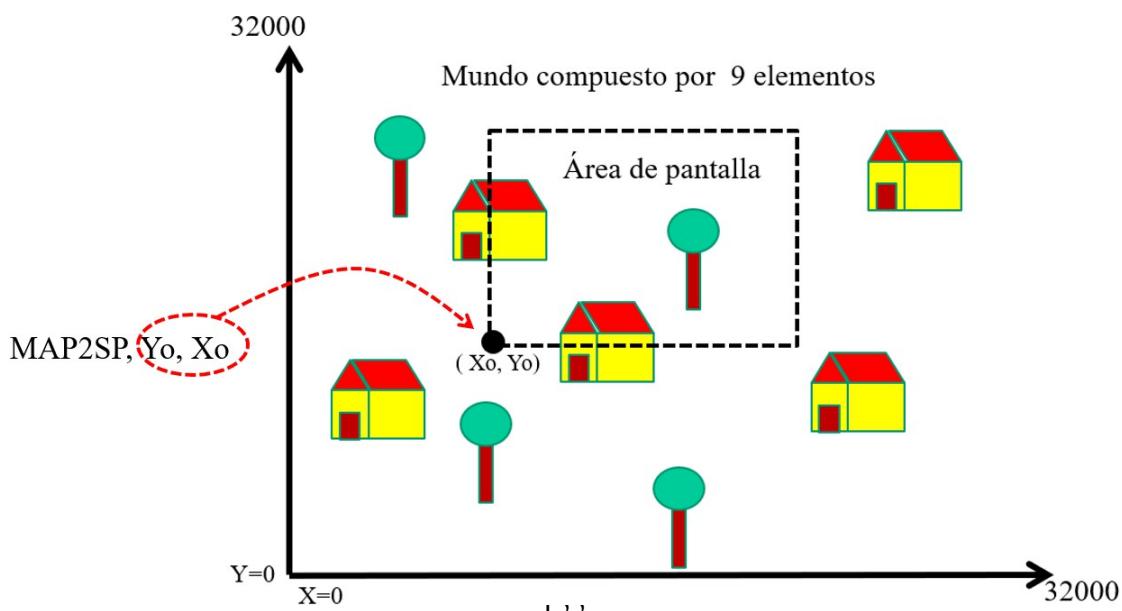


Abb. 84 Weltkarte und MAP2SP

Wenn Sie diesen Mechanismus verwenden, müssen Ihr Charakter und Ihre Feinde Sprites ab 31 verwenden, um mögliche Konflikte zwischen den Sprites, die vom Scroll-Mechanismus verwendet werden, und Ihren Charakteren zu vermeiden. Wenn MAP2SP zufällig auf mehr als 32 Elemente stößt, die in Sprites übersetzt werden müssen, werden die über 32 hinausgehenden ignoriert.

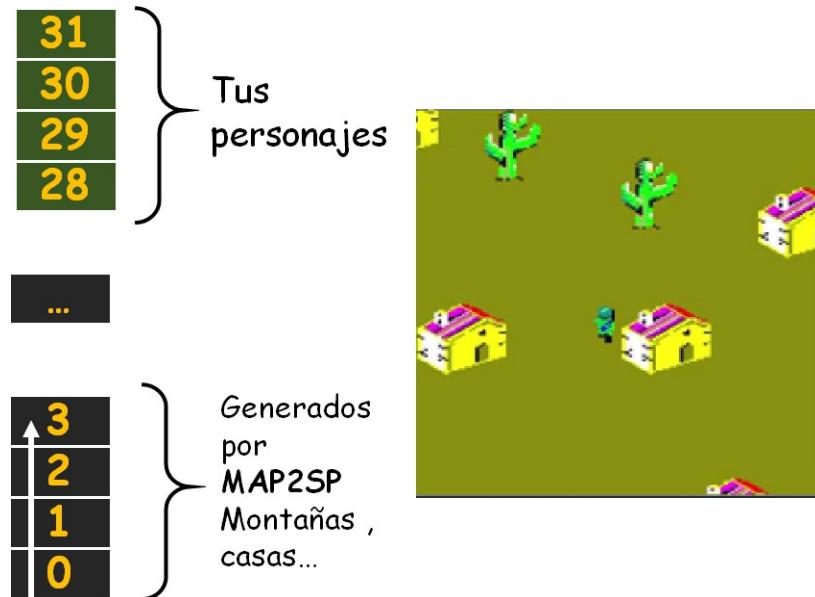


Abb. 85 von MAP2SP verbrauchte Sprites

Sie müssen MAP2SP in jedem Spielzyklus aufrufen oder zumindest jedes Mal, wenn Sie die Koordinaten des Blickpunkts ändern, von dem aus Sie die Welt betrachten wollen. Das "gleitende" Fenster, mit dem Sie die Bilder jagen, die in Sprites umgewandelt werden, misst immer das, was der Bildschirm misst (80 Bytes x 200 Zeilen), unabhängig davon, wie Sie den Befehl SETLIMITS einstellen. Das heißt, **selbst wenn SETLIMITS einen sehr kleinen Bildbereich festlegt, wird alles, was das 80 Byte x 200 Zeilen große Fenster "erjagt" hat, in Sprites umgewandelt.**

Die von MAP2SP erstellten Sprites werden standardmäßig mit Status 3 erstellt, d.h. mit aktivem Druckflag (PRINTSPALL druckt sie) und mit aktivem Kollisionsflag (COLSP kollidiert mit ihnen). Wenn Sie die Sprites mit einem anderen Zustand erstellen möchten, rufen Sie den Befehl MAP2SP einfach einmal mit einem einzigen Parameter auf, der den Zustand angibt, mit dem die Sprites erstellt werden sollen. Mit einem einzigen Aufruf dieses Typs wird der Befehl für die folgenden Aufrufe mit Koordinaten konfiguriert.

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>.

Beispiel:

| Damit wird der MAP2SP-Befehl so konfiguriert, dass er gedruckt wird , aber nicht sichtbar ist.

Nach der Klärung des Konzepts wollen wir uns nun im Detail ansehen, wie die Weltkarte spezifiziert wird und ein Beispiel für die Verwendung der Funktion MAP2SP geben.

14.4.1 Karte der Welt (Kartentisch)

Die Tabelle, in der wir alle Elemente der Karte registrieren, heißt MAP_TABLE und wird in einer .asm-Datei namens **map_table_tujuego.asm** angegeben

Diese Tabelle enthält die Elemente, die die Weltkartenbilder für Ihre Scroll-Spiele definieren. Die Tabelle wird an der gleichen Speicheradresse wie das LAYOUT, d.h. an der Adresse 42040, abgelegt. Das bedeutet, dass das Layout und die Weltkarte nicht gleichzeitig verwendet werden können, aber das ist kein Problem, da ein Rollenspiel das Layout nicht verwenden wird und umgekehrt. Außerdem besteht die Einschränkung nur darin, dass sie nicht gleichzeitig verwendet werden können, aber ein Spiel könnte eine Phase haben, in der es das Layout verwendet und eine andere, in der es auf der Grundlage der Weltkarte scrollt.

Die Weltkarten-Eingabetabelle beginnt mit 3 globalen Parametern (insgesamt 5 Bytes) und einer Liste von "Kartenelementen", die durch jeweils 3 Parameter (x, y, Bildrichtung) beschrieben werden.

Die Liste kann bis zu 82 Einträge enthalten, aber die Anzahl der Einträge kann durch einen der globalen Parameter begrenzt werden. Die Liste belegt höchstens die anfänglichen 5 Bytes + 82 Einträge x 6 Bytes = 5+492=497 Bytes. Würden wir ein weiteres Element einfügen, würden wir die für die Karte reservierten 500 Bytes überschreiten (die auch für das Layout reserviert sind).

Die Tabelle beginnt mit 3 Parametern:

- Die maximale Höhe eines beliebigen Kartenelements
- Maximale Breite eines Kartenelements (als negative Zahl anzugeben)
- Anzahl der Artikel (maximal 82)

Die ersten beiden Parameter sind wichtig, um zu prüfen, ob ein Sprite teilweise auf dem Bildschirm erscheint, da die Funktion MAP2SP die Breite und Höhe jedes Bildes nicht kennt oder herausfinden kann. Sie weiß nur, wo sich das Kartenelement befindet, und unter Annahme der maximalen Breite und Höhe prüft sie, ob dieses Element auf dem Bildschirm erscheinen kann. Wenn ja, wird ein Sprite aus dem Kartenelement erstellt. Wenn diese beiden Parameter auf Null gesetzt sind, muss sich die linke obere Ecke des Kartenelements innerhalb des Bildschirms befinden, damit das Element in ein Sprite umgewandelt werden kann.

Jedes Element ist ein Tupel aus 3 Parametern, denen die Abkürzung "DW" vorangestellt ist:

DW Y, X, <Bild>

Sehen wir uns ein Beispiel für die Datei map_table_tujuego.asm an

KARTE TABELLE

3 Parameter vor der Liste der "Kartenelemente".
dw 50; maximale Höhe eines Sprites für den Fall, dass es den oberen Rand durchbricht und man einen Teil davon malen muss.
dw -40; maximale Breite eines Sprites, wenn es sich um ein linksseitiges Sprite handelt (negative Zahl)
db 82; Anzahl der Kartenelemente. Es sollten höchstens 82 sein.

und von hier aus beginnen die Posten

dw 100,10,HOUSE; 1
dw 50,-10,CACTUS;2
dw 210,0,HOME;3

```
dw 200,20,CACTUS;4
dw 100,40,HOUSE;5
dw 160,60,HOUSE;6
dw 70,70,HOUSE;7
dw 175,40,CACTUS;8
dw 10,50,HOUSE;9
dw 250,50,HOUSE;10
dw 260,70,HOUSE;11
dw 260,70,HOUSE;11
dw 290,60,CACTUS;12
dw 180,90,HOUSE;13
dw 60,100,HOUSE;14
dw 60,100,HOUSE;14
```

...

Um deine Welt zu gestalten, empfehle ich dir, ein kariertes Notizbuch zu nehmen und darauf die Elemente zu zeichnen, die deine Welt haben soll. Jedes Quadrat des Notizbuchs kann eine feste Größe darstellen, z. B. 8 Pixel oder 25 Pixel. Wichtig ist, dass du dir Zeit nimmst, um die Welt zu zeichnen, die du dir wünschst, und die Art und Weise, wie du durch sie hindurchgehen möchtest. Es gibt zum Beispiel multidirektionale Spiele vom Typ "Gauner" und andere Spiele mit vertikalem Bildlauf wie Commando. Du hast die Wahl, aber in jedem Fall solltest du dir Zeit und Geduld nehmen, denn das Ergebnis wird es wert sein.

Jede Phase deines Spiels kann eine Karte sein. In 8BP können Sie die Karte jederzeit mit POKE-Funktionen ändern. Normalerweise benutze ich 1KB des von 8BP verwendeten Speicherplatzes, zum Beispiel von 23000 bis 24000, um alle Phasen (Maps) des Spiels zu speichern, und jedes Mal, wenn ich eine Phase betrete, lade ich die entsprechende Map durch PEEKing und POKEing an Adresse 42040. Das heißt, ich erstelle meine Kartendatei an Adresse 23000 und sie belegt 1 KByte, so dass 23 KB für mein BASIC-Programm übrig bleiben. Damit diese Karteninformationen nicht von Basic zerquetscht werden, muss ich zu Beginn des Spiels einen MEMORY 22999 anlegen.

14.4.2 Verwendung der MAP2SP-Funktion

Schauen wir uns nun ein Beispiel für die Verwendung dieser Funktion an. Im Grunde muss sie einmal in jedem Spielzyklus mit den neuen Koordinaten des Ursprungs, von dem aus die Welt beobachtet wird, aufgerufen werden.

Die Funktion erstellt eine variable Anzahl von Sprites ab Sprite 0, und zwar mit angepassten Bildschirmkoordinaten. Das heißt, selbst wenn ein Kartenelement eine Koordinate $x=100$ hat, wird dieses Sprite mit der Bildschirmkoordinate $x'=x-90=10$ erstellt, wenn der bewegliche Ursprung an der Position $x=90$ liegt. Bei der Y-Achsenkoordinate wird berücksichtigt, dass die Y-Achse auf dem Amstrad nach unten wächst, während die Weltkarte nach oben wächst. Also wird die Y-Koordinate mit der Gleichung $Y'= 200-(Y-Y_{orig})$ angepasst. Aber keine Sorge, diese Anpassung wird bereits von der MAP2SP-Funktion vorgenommen. Sie müssen nur noch den Ursprung der Weltkarte ändern, von dem aus sie angezeigt werden soll.

In diesem Minispiel wurde eine Welt aus Häusern und Kakteen geschaffen, und unsere Spielfigur bewegt sich zwischen den Elementen. In diesem Beispiel wird im Falle einer Kollision (erkannt mit

|COLSPALL), kann die Spielfigur nicht weiterfliegen. In einem Flugzeugspiel, in dem Kartenelemente "fliegbar" sind, könnten wir die Kollision mit **|COLSP, 32** so parametrieren, dass nur Kollisionen mit Feinden und Schüssen und nicht mit Hintergrundelementen erkannt werden,

<Spruchanfang>, <Spruchende_Endung>.



Abb. 86 Mini-Scrolling-Spiel, inspiriert von "Commando".

Wie Sie sehen können, ist es sehr klein, aber es hat alles: multidirektionales Scrollen, Tastaturlesen, wechselnde Animationssequenzen der Charaktere, Kollisionserkennung, Musik...

WICHTIG: Beachten Sie den MEMORY-Befehl. Wir haben die Montageoption 2 verwendet, die ideal für Scroll-Spiele ist und uns fast 25 KB frei lässt.

```

10 SPEICHER 24799
20 MODUS 0
30 BEI PAUSE GOSUB 280
40 ANRUF &6B78
50 DEFINT a-z
60 TINTE 0,12
70 FOR y=0 TO 400 STEP 2
80 PLOT 0,y,10:DRAW 78,y
90 PLOT 640-80,y,10:DRAW 640,y
100 NÄCHSTE
110 x=0:y=0
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1:' Anfangsrichtung nach oben
140 |locatesp,31,100,36
150 |MUSIK,0,1,5
160 |SETLIMITS,10,70,0,199: |PRINTSPALL,0,1,0
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0, 0: REM-Kollision, sobald es eine minimale Überlappung
gibt
190 'Beginn des Spielzyklus
200 IF INKEY(27)=0 THEN x=x+1:IF dir>>>3 THEN dir=3:|SETUPSP,31,7,3:
GOTO 220
210 IF INKEY(34)=0 THEN x=x-1:IF x<0 THEN x=0:ELSE IF dir>>4 THEN
dir=4:|SETUPSP,31,7,4
220 IF INKEY(67)=0 THEN y=y+2:IF x=xa AND dir <> 1 THEN
dir=1:|SETUPSP,31,7,1: GOTO 240
230 IF INKEY(69)=0 THEN y=y-2:IF y<0 THEN y=0:ELSE IF x=xa AND dir <>2
THEN dir=2:|SETUPSP,31,7,2:
240 IF xa=x AND ya=y THEN dir=0 ELSE |ANIMA,31
250 |MAP2SP,y,x:|COLSPALL: IF col<32 THEN x=xa:y=ya:|MAP2SP,y,x ELSE
xa=x:ya=y
260 |DRUCKSACK

```

270 GOTO 200

280 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

Sehen wir uns nun ein weiteres Beispiel für horizontales Scrollen an, bei dem ein interessanter Effekt einer "unendlichen" Weltkarte erzielt wurde, indem das Ende der Karte dem Anfang entspricht und ein abrupter Sprung erfolgt, wenn Xo einen bestimmten Wert erreicht. Tatsächlich hat diese Weltkarte nur 13 Elemente.

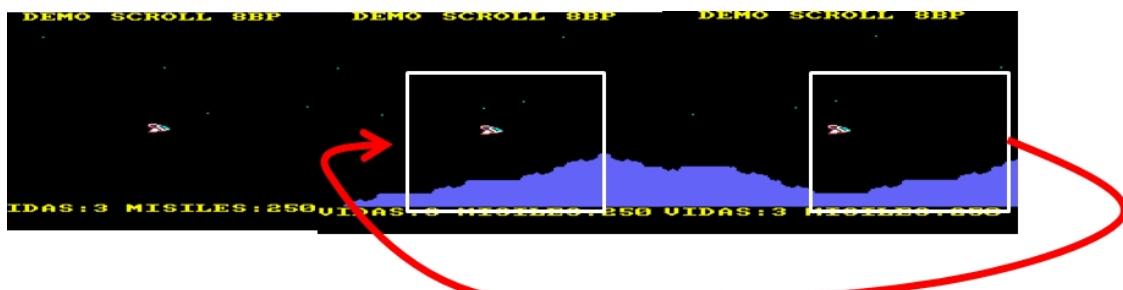


Abb. 87 Karte der "unendlichen" Welt

Dies ist die verwendete Karte

_MAP_TABLE

3 Parameter vor der Liste der "Kartenelemente".

dw 50; maximale Höhe eines Sprites für den Fall, dass es den oberen Rand durchbricht und ein Teil davon gemalt werden muss.

dw -18; maximale Breite eines beliebigen Kartenelements. muss als negative Zahl angegeben werden

db 13; Anzahl der Kartenelemente. Es sollten höchstens 82 sein.

und von hier aus beginnen die Posten

dw 36,80,MONTUP; 1

dw 48,100,MONTUP;2

dw 60,120,MONTUP;3

dw 72,130,MONTUP;4

dw 72,140,MONTDW;5

dw 60,160,MONTH;6

dw 60,180,MONTDW;7

dw 48,190,MONTDW;8

dw 48,190,MONTDW;8

hier wiederhole ich Elemente, um mit der Position 100 dw

48,210,MONTUP;9 zu passen

dw 60,230,MONTUP;10

dw 72,240,MONTUP;11

dw 72,250,MONTDW;12

dw 60,270,MONTH;13

dw 60,270,MONTH;13

;-----

Und das ist das BASIC-Programm, in dem ich die Zeile hervorgehoben habe, in der die Welt rückwärts läuft, ohne dass der Spieler etwas merkt.

10 SPEICHER 24799

11 FOR dir=42540 TO 42618 STEP 2: POKE dir,20+RND*110:POKE dir+1,RND*80:NEXT

20 MODUS 0

```

30 BEI PAUSE GOSUB 280
40 AUFRUF &6B78
50 DEFINT a-z
51 TINTE 0,0
52 |MUSIK,0,0,0,5
110 xo=0:yo=0
111 x=36:y=100
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1:' Anfangsrichtung nach oben
140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,176: |PRINTSPALL,0,1,0
161 LOCATE 1,23 :PEN 1: PRINT "LEBEN:3 RAKETEN:250" PRINT "LEBEN:3
RAKETEN:250" PRINT "LEBEN:3 RAKETEN:250" PRINT "LEBEN:3 RAKETEN:250
162 LOCATE 1,1:PRINT " DEMO SCROLL 8BP"
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0, 0: REM-Kollision, sobald es eine minimale Überlappung
gibt
190 'Beginn des Spielzyklus
200 IF INKEY(27)=0 THEN x=x+1: GOTO 220
210 IF INKEY(34)=0 THEN x=x-1:IF x<0 THEN x=0
220 IF INKEY(69)=0 THEN y=y+2: GOTO 240
230 IF INKEY(67)=0 THEN y=y-2:IF y<0 THEN y=0
240 IF xa=x AND ya=y THEN dir=0 ELSE |ANIMA,31
250 |MAP2SP,yo,xo:|COLSPALL:IF col<32 THEN END END
260 |DRUCKSACK
261 Zyklus=Zyklus +1: IF Zyklus=2 THEN      |STARS,0,5,2,0,-1:ciclo=0
262 xo=xo+1:IF xo=210 THEN xo=100
263 |LOCATESP,31,y,x
270 GOTO 200
280 |MUSIK:MODUS 1: TINTE 0,0:STIFT 1

```

14.4.3 Beispiel für eine Phasendatei

Wenn Sie mehrere Stufen in einem scrollenden Spiel haben wollen, können Sie sie, wie ich schon sagte, in einem Speicherbereich vorladen lassen. Zum Beispiel können Sie die Stages an Adresse 23000 zusammenstellen und haben dann 1000 Bytes, um mehrere Weltkarten zu speichern, da 8BP an Adresse 24000 beginnt. In diesem Fall muss dein Spiel mit einem MEMORY 22999 beginnen.

Das Videospiel "Nibiru" tut dies, obwohl es erstellt wurde, als 8BP an der Adresse 26000 begann (es wurde mit 8BP v26 erstellt), also speichert es die Karte ab 25000. Um eine Phase zu laden, liest es einfach den Bereich, in dem jede Phase zusammengesetzt wurde, und schreibt ihn über die Adresse, an der die Welttabelle stehen sollte, bevor es mit dem Spiel in dieser Phase beginnt. Diese drei BASIC-Zeilen zeigen, wie man Phase 1 des Spiels kopiert (&61a8 = 25000)

```

310 ' stochert in der Weltkarte herum
320 dirmap!=42040:FOR i!=&61A8 TO &620D
330 dato=PEEK(i!):POKE dirmap!,dato:dirmap!=dirmap!+1
340 NÄCHSTES

```

Es gibt einen schnelleren Weg, die Phasen mit dem Befehl |UMAP zu laden, den ich später erläutern werde, aber diese FOR-Schleife mit POKES ist vollkommen gültig.

Als Nächstes zeige ich Ihnen die Phasendatei des Spiels "**Nibiru**", die wir an der Adresse 25000 zusammensetzen (denken Sie daran, dass die Version von 8bp, mit der "**Nibiru**" erstellt wurde, bei 26000 begann, so dass von 25000 bis 26000 1000 Bytes frei waren. Mit der aktuellen Version von 8BP würden wir die Phasen zusammensetzen, ohne die Adresse 24800 zu erreichen).

```

org 25000
PHASE1
;=====
_START_PHASE1
3 Parameter vor der Liste der "Kartenelemente".
dw 50; maximale Höhe eines Sprites für den Fall, dass es den oberen Rand durchbricht und ein Teil davon gemalt werden muss.
dw -18; maximale Breite eines beliebigen Kartenelements. muss als negative Zahl angegeben werden
db 16; num items dw
36,82,MONTUP; 1 dw
48,104,MONTUP;2 dw
60,126,MONTUP;3 dw
72,138,MONTUP;4 dw
72,150,MONTDW;5 dw
60,172,MONTH;6 dw
60,194,MONTDW;7 dw
48,206,MONTDW;8 dw
60,172,MONTH;6 dw
60,194,MONTDW;7 dw
48,206,MONTDW;8


### hier wiederhole ich Elemente, um mit der Position 100 dw 48,228,MONTUP;9 zu passen


dw 60.250,MONTUP;10
dw 72.262,MONTUP;11
dw 72.274,MONTDW;12
dw 60.296,MONTH;13
dw 60.320,MONTDW;14
dw 48.350,MONTDW;15
dw 36.380,MONTDW;16
dw 36.380,MONTDW;16
_END_PHASE1
;=====
PHASE2
;=====
_START_PHASE2
dw 50; maximale Höhe eines Sprites für den Fall, dass es den oberen Rand durchbricht und ein Teil davon gemalt werden muss.
dw -6; maximale Breite eines beliebigen Kartenelements. muss als negative Zahl angegeben werden
db 15
dw 128,80,PLACA2_L_OV
dw 128,110,PLACA2_R_OV
dw 192,116,PLACA2_L_OV
dw 192,126,PLACA2_R_OV
dw 92,130,PLACA_L_OV dw
92,150,PLACA_R_OV dw
124,151,PLACA_L_OV dw
124,171,PLACA2_R_OV dw
128,200,PLACA2_L_OV dw
128,210,PLACA2_R_OV dw
92,220,PLACA2_L_OV dw
92,230,PLACA2_R_OV dw
164,240,PLACA2_L_OV dw
164,260,PLACA2_R_OV dw
156,254,PLACA2_R_OV dw
156,254,CUPULA2_OV

```

```

-END_PHASE2
=====
PHASE3
=====
START_PHASE3
dw 50; maximale Höhe eines Sprites für den Fall, dass es den oberen Rand
durchbricht und man einen Teil davon malen muss.
dw -80; maximale Breite eines beliebigen Kartenelements. muss als negative Zahl
angegeben werden.
db 4
dw 40,0,MAR; 1
dw 40,80,SEA; 2
dw 189,0,CLOUDS; 2
dw 189,80,CLOUDS; 2
-END_PHASE3

```

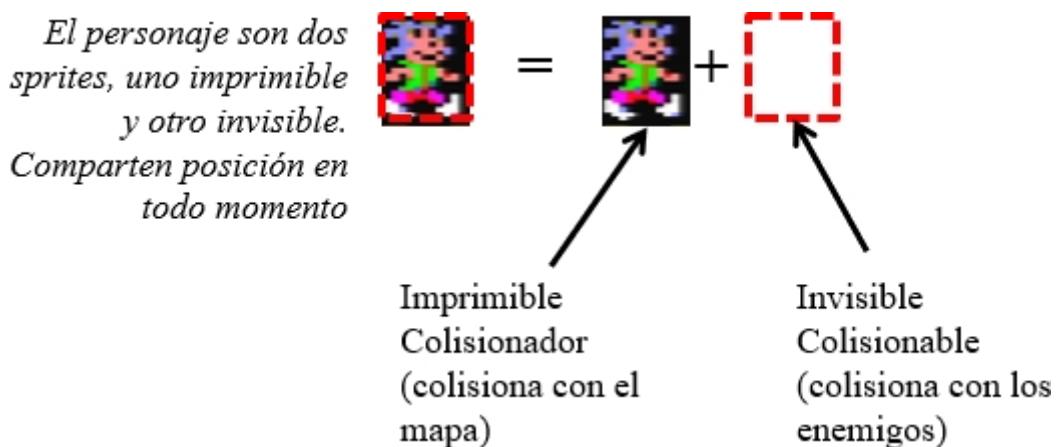
14.4.4 Feindliche Kollision mit Karte

Du möchtest vielleicht ein Spiel machen, in dem dein Charakter mit der Karte kollidiert (mit COLSPALL) und daher ein Kollider ist. Sagen wir, Sie haben dies:

- Dein Charakter: collider
- Kartenelemente: kollidierbar
- Ihre Aufnahme: Collider
- Feinde: kollidierbar

Wenn deine Feinde Kollider sind, können sie nicht mit der Karte kollidieren, und vielleicht möchtest du zum Beispiel ein Gaunerspiel machen. Die Lösung ist, die Feinde zu Kollidern zu machen. Aber in diesem Fall können sie dich natürlich nicht mehr töten. Und es gibt auch ein Problem damit, dass der Schuss ebenfalls ein Kollider ist.

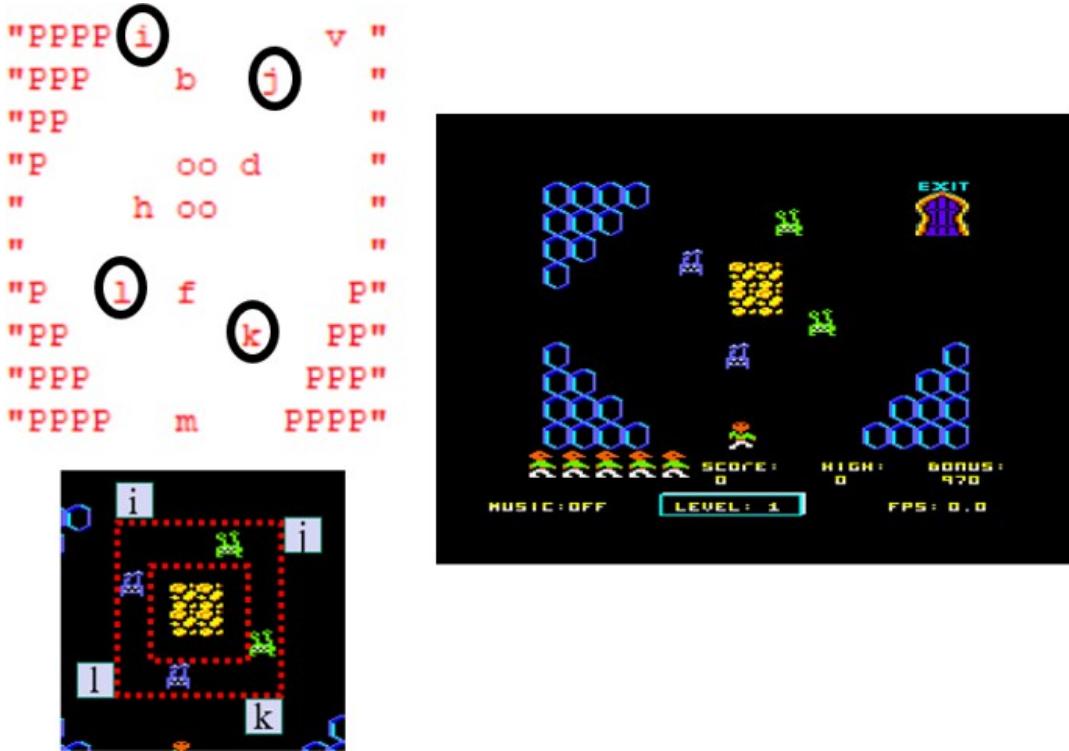
Die Lösung für dieses Problem basiert auf einem einfachen Trick, der auf nicht druckbaren "unsichtbaren" Sprites basiert, die keine CPU zum Drucken benötigen, aber dennoch vorhanden sind. **Ihr Charakter wird zwei Sprites verwenden:** ein druckbares Kollisionssprite und ein nicht druckbares Kollisionssprite, aber von gleicher Größe. Mit den Schüssen machen wir das Gleiche, damit sie mit Feinden und der Karte kollidieren können.



Ein ähnlicher Trick wird in dem Spiel "**Eternal Frogger**" verwendet. Durch die Verwendung unsichtbarer Sprites wird der Frosch zum Sterben gebracht, wenn er in den Fluss fällt (siehe den Abschnitt, der den Befehl COLSPALL erklärt). Dieser einfache

Trick lässt sich auch auf die Schüsse des Spiels

Zeichen. „Unsichtbare“ Sprites sind sehr nützlich. Im Spiel „Happy Monty“ werden sie verwendet, um Feinde dazu zu bringen, ihre Richtung zu ändern, d. h. wenn ein Feind mit einem unsichtbaren Sprite kollidiert, ändert er seine Richtung. Das bedeutete, dass es nicht notwendig war, viele an jeden Bildschirm angepasste Routen zu definieren, sondern lediglich in jedem Level eine Reihe von unsichtbaren Sprites zu platzieren, die es ermöglichten, die Flugbahnen der Feinde zu ändern (siehe Dokument „Abheben“ von **Happy Monty**).



14.4.5 Hintergrundbilder in Ihrer Schriftrolle

Hintergrundbilder sind für scrollende Spiele gedacht und sind eine Funktion, die 8BP ab Version V42 bietet. Es handelt sich dabei um eine Art transparenten Druck, bei dem die Sprites, die den Hintergrund (die Weltkarte) bilden, unter der Spielfigur und den Gegnern gedruckt werden können, ohne zu flackern.

Hintergrundbilder, die einem Sprite zugewiesen werden, werden mit dieser speziellen Transparenz gedruckt, und es spielt keine Rolle, ob das Sprite das Transparenz-Flag seinem Statusbyte zugewiesen hat oder nicht. Siehe Kapitel 8, um zu erfahren, wie man sie benutzt.

WICHTIG: Auf der Weltkarte können Sie normale Bilder mit „Hintergrundbildern“ kombinieren (Abschnitt 8.5). Hintergrundbilder sind immer transparent. Das Transparenz-Flag, das Sie in **|MAP2SP, <status>** verwenden, gilt nur für normale Bilder.

WICHTIG: Hintergrundbilder sind teuer im Druck, und wenn man sie spiegelt, sind sie noch teurer. Wenn Ihr Spiel über Bildlaufleisten verfügt, versuchen Sie, diese für

Elemente zu verwenden, über die Ihre Spielfigur oder Ihre Feinde fliegen werden. Um das Scrollen zu beschleunigen, verwenden Sie zum Beispiel

normale Bilder auf Häusern oder Felsen, die an den Seiten der Schriftrolle erscheinen und nicht oft von den Sprites überlagert werden

14.5 Parallaxe-Bildlauf

Schauen wir uns an, wie man einen Parallaxen-Scroll macht, d.h. mit mehreren "Ebenen" mit unterschiedlichen Geschwindigkeiten. Vielleicht fällt Ihnen noch ein anderer Weg ein, dies ist nur eine mögliche Methode, die im Spiel "Nibiru" verwendet wird.

Das erste, was Sie wissen sollten, ist, dass es viel schneller ist, ein sehr langes horizontales Sprite zu drucken als viele kleine Sprites. Das liegt an den Operationen, die nach dem Malen jeder Scanline eines Sprites durchgeführt werden müssen. Aus demselben Grund ist es viel schneller, ein sehr großes horizontales Sprite zu drucken als ein vertikales Sprite derselben Größe.

Wir werden zwei riesige Sprites auf der Weltkarte platzieren, um das Wasser darzustellen. Ich habe eines 160 x 8 gemacht, also habe ich zwei davon platziert, so dass, wenn es scrollt, das nächste erscheint. Wenn die MAP2SP-Funktion den ganzen Bildschirm durchläuft, kehrt sie zu x=0 zurück und das Ganze wiederholt sich auf unbestimmte Zeit. Auf der Weltkarte habe ich auch zwei Sprites für die Wolken eingefügt.

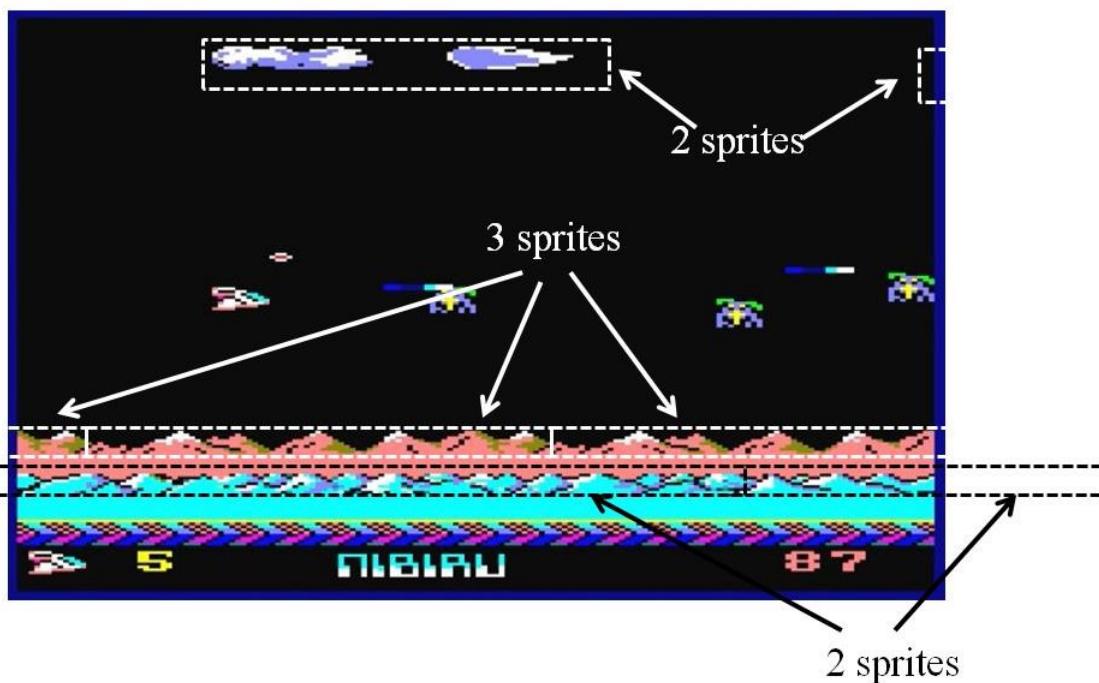


Abb. 88 Parallaxe-Schriftrolle

Für die Berge habe ich 3 normale Sprites verwendet, außerhalb der Weltkarte. Ich gab ihnen automatische Bewegung auf ihre Statusflagge und ließ sie sich nach links bewegen. Aber in ungeraden Zyklen deaktiviere ich sowohl die Druckflagge als auch die Flagge für die automatische Bewegung, so dass sie sich nur jeden zweiten Spielzyklus bewegen und drucken, wobei sie eine Geschwindigkeit erreichen, die halb so hoch ist wie die des Wassers. Die Sprites der Berge sind 16, 17 und 18, und mit diesen Stößen wirke ich auf ihr Statusbyte.

```
mc=cycle AND 1: IF mc THEN POKE 27256,0: POKE 27272,0: POKE 27288,0  
SONST STOCHERE IN 27256,11: STOCHERE IN 27272,11: STOCHERE IN 27288,11
```

Im Beispiel ist "mc" die Variable, die bestimmt, ob der Zyklus ungerade oder gerade ist.

14.6 Dynamische Kartenaktualisierung: |UMAP

Vielleicht brauchen wir in unserem Spiel eine Karte mit mehr als 82 Elementen. Oder wir wollen einfach, dass der Befehl |MAP2SP mit einer kleineren Karte, die wir regelmäßig aktualisieren, schneller läuft. Oder wir wollen beides!

Zu diesem Zweck gibt es seit Version 32 von 8BP den Befehl **|UMAP** (kurz für "UPDATE MAP"). Dieser Befehl aktualisiert die Karte mit Informationen, die sich in einem anderen Speicherbereich befinden, in dem wir eine größere Karte haben. Der Befehl bewirkt, dass die Karte komplett neu aufgebaut wird, wobei nur die Elemente aufgenommen werden, die bestimmten Bereichen von X- und Y-Koordinaten entsprechen (alle Parameter sind 16-Bit-Zahlen).

```
|UMAP, <map_ini>, <map_fin>, <y_ini>, <y_fin>, <x_ini>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>.
```

Bei UMAP handelt es sich nicht um ein einfaches Kopieren von Elementen. Es ist ein "selektives" Kopieren. Wenn wir zum Beispiel eine Karte an der Adresse 22500 haben, die 1500 Byte belegt, und wir wollen die Karte mit den Koordinaten unseres Zeichens aktualisieren, mit genügend Spielraum, um in der Y-Koordinate bis zu 100 Zeilen und in der X-Koordinate bis zu 20 Byte in alle Richtungen vorzurücken:

```
|UMAP, 22500,23999, y-100, y+100, x-20, x+20
```

Dieser Befehl prüft die Koordinaten der Elemente, die sich auf der Karte an der Adresse 22500 befinden, und wenn sie innerhalb der von uns festgelegten X- und Y-Ränder liegen, werden sie in den Speicherbereich kopiert, den 8BP für den Befehl |MAP2SP verwendet, d. h. er kopiert sie von der Adresse 42040. Es werden jedoch nur diejenigen kopiert, die die Bedingung erfüllen. Da weniger Elemente vorhanden sind, läuft der Befehl |MAP2SP schneller, da er lesen und prüfen muss, ob weniger Elemente auf dem Bildschirm vorhanden sind.

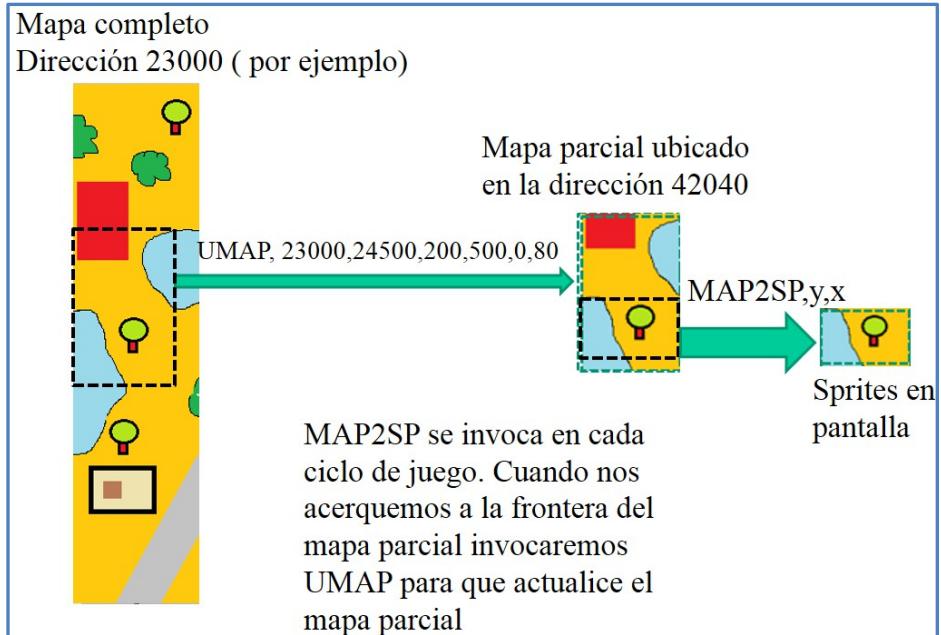


Abb. 89 UMAP

In regelmäßigen Abständen (nicht bei jedem Spielzyklus) können wir die Karte mit UMAP aktualisieren und so sehr große Welten mit vielen Elementen erstellen, während wir gleichzeitig mehr Geschwindigkeit in MAP2SP erhalten. Der UMAP-Befehl ist sehr schnell, aber es macht keinen Sinn, ihn bei jedem Spielzyklus aufzurufen, da MAP2SP mit einer viel größeren Karte arbeiten kann, als auf den Bildschirm passt, und wir können UMAP nur dann aufrufen, wenn wir Bereiche der ursprünglichen Karte benötigen, die in der Teilkarte nicht vorhanden sind. Ich habe es in dem Autorennspiel "**3D Racing one**" verwendet, da die Streckenkarte sehr groß war (mehr als 82 Elemente) und ich in regelmäßigen Abständen UMAP aufgerufen habe, wenn das Auto weiterfuhr.

In der vollständigen Kartenadresse (im Beispiel 22500) wird nur eine Liste von Elementen enthalten sein. **Die drei Ausgangsdaten** der Karte 42040 (d. h. die maximale Höhe eines Kartenelements, die maximale Breite eines Kartenelements und die Anzahl der Elemente) **sind nicht enthalten**. Die Anzahl der Elemente wird von **|UMAP** aktualisiert (je nachdem, wie viele Elemente die vorgegebenen Ränder einhalten). Die beiden anderen Parameter werden von Ihnen in der Datei "map_table_your_game.asm" festgelegt.

Der Befehl **|UMAP** fügt die Elemente in der Teilkarte in einer Reihenfolge ein, die die Teilkarte nach der Y-Koordinate des Bildschirms sortieren soll. Normalerweise bearbeiten Sie Ihre globale Karte in einer Datei namens "misupermapa.asm" oder so ähnlich. Und bauen Sie sie in die 22500 (zum Beispiel) ein. In diese Datei schreibst du nach und nach die Elemente deiner Karte, in aufsteigender Reihenfolge der Y-Koordinate. Nun, um die Sprites bereits nach Y-Koordinaten sortiert zu bekommen (in aufsteigender Reihenfolge der Bildschirmkoordinaten), liest der **|UMAP-Befehl** sie vom Ende zum Anfang. Auf diese Weise werden die Sprites, die später mit **|MAP2SP** erzeugt werden, nach der Y-Koordinate des Bildschirms sortiert. Denken Sie daran, dass der Bildschirm ein inverses Koordinatensystem in Bezug auf die Karte verwendet, d.h. die 150. Koordinate des Bildschirms ist die 50. Koordinate der Karte (im Fall von **|MAP2SP,0,0**). Wenn Sie das nicht so recht verstehen, machen Sie sich keine Sorgen. Es handelt sich hierbei um etwas aus dem Innenleben von **|UMAP** und **|MAP2SP**, um es

effizienter zu machen. In der folgenden Abbildung habe ich die Karte und den CPC-Bildschirm dargestellt. Wie Sie sehen können

die Karte kann sehr groß sein, aber auch ihre Koordinaten wachsen nach oben, während die Bildschirmkoordinaten nach unten wachsen.

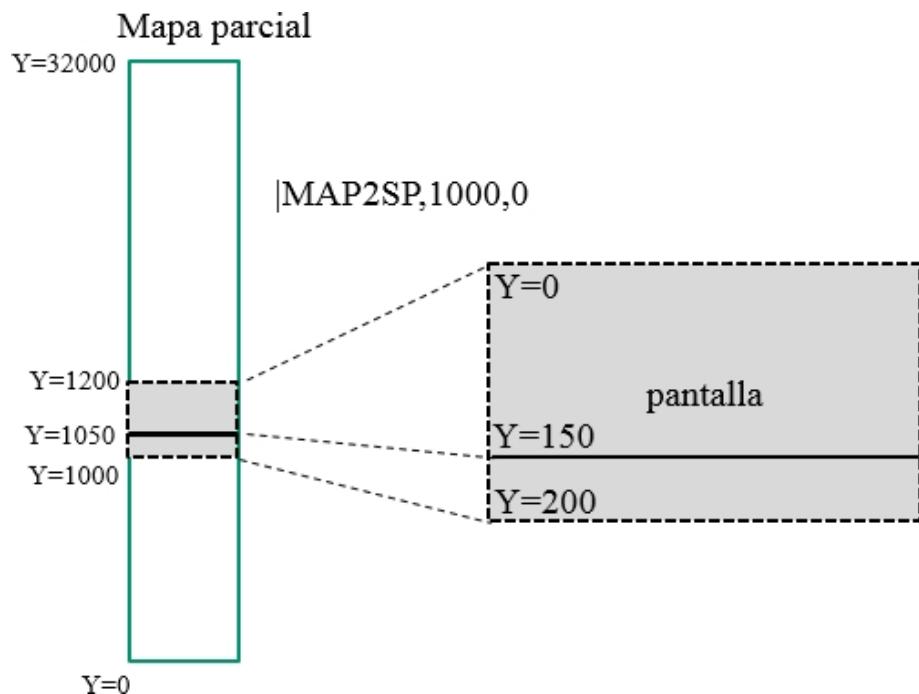


Abb. 90 MAP und Anzeige

14.7 Animation und Farbverschiebung: Befehl RINK

Es gibt Spiele, bei denen große Blöcke des Bildschirmspeichers verschoben werden müssen, um ein Gefühl der Bewegung zu vermitteln, z. B. die Seitenleisten in Rennspielen oder große Blöcke aus Ziegeln oder Erde. Mit dem MAP2SP-Befehl können Sie all das tun, aber die Geschwindigkeit ist nicht rasant schnell, weil es eine Menge CPU für das Bewegen von Sprites verbraucht. Die Farbanimation ist in diesen Fällen die perfekte Ergänzung.

Auf Computern wie dem AMSTRAD mit seiner leistungsstarken Palette von 16 gleichzeitigen Farben machen viele Spiele von der Tintenanimation Gebrauch. Ein deutliches Beispiel sind einige Autospiele, deren Straßenrandstreifen sich für diese Art der Animation eignen.

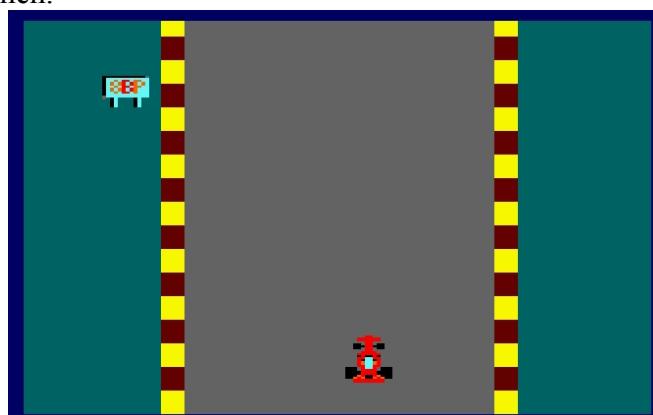


Abb. 91 Tinten-animierte Streifen

Bei der Animation durch Tinten wird eine Reihe von Tinten definiert, auf denen eine Reihe von Farben gedreht wird. Sehen wir uns ein Beispiel mit weißen/grauen Streifen und 8 Farben an:

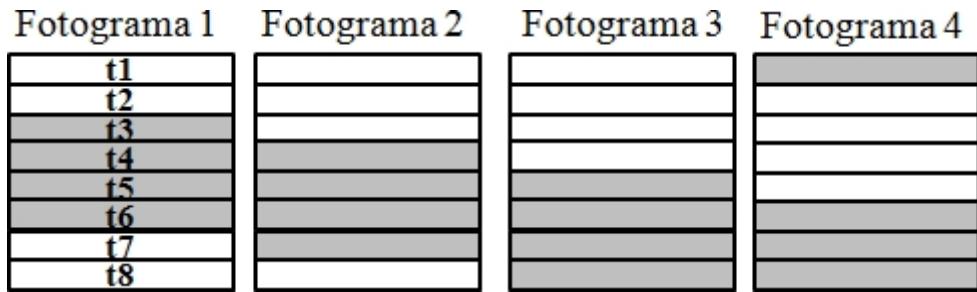


Abb. 92 Tintenanimation

Um ein Gefühl der Bewegung zu vermitteln, müssen Sie als erstes den Farben, die sich drehen sollen, Farben zuweisen. In diesem Fall die Farben Weiß (26) und Grau (26) und Grau (26).

(13) werden den Tinten t1..t8 zugewiesen. Nehmen wir an, dass die Tinte t1 8 ist, so wird die Tinte t8 15 sein. Die restlichen Tinten (0 bis 7) werden für die Sprites verwendet. Zu jedem Zeitpunkt müssen wir die Werte der 8 Tinten neu zuordnen, um das Gefühl der Rotation zu erzeugen. Dies wird mit dem Befehl **|RINK** (kurz für "Rotate INK") erreicht.

Mit RINK können Sie ein Farbmuster definieren, das über eine Reihe von Farben rotiert. Eine Tinte ist keine Farbe. Eine Tinte ist eine Kennung im Bereich [0..15], die eine Farbe im Bereich [0..26] identifiziert. Um das Muster der zu drehenden Farben zu definieren, verwenden wir den Befehl RINK wie folgt:

RINK, <initial_ink>, <Farbe1>,<Farbe2>, <Farbe3>, ... ,<FarbeN>, <FarbeN>, ... ,<FarbeN>, <FarbeN>, <FarbeN>.

Dies zeigt an, dass sie N Tinten, beginnend mit der ersten Tinte, mit dem angegebenen Farbmuster drehen werden. Beachten Sie, dass, wenn Sie viele Tinten für eine Animation verwenden, weniger Tinten für Ihre Sprites übrig bleiben.

Sobald das Muster festgelegt ist, können wir die Farben mehr oder weniger schnell mit

RINK, <Schritt>

Der Schrittwert ist die Anzahl der Verschiebungen, die die Druckfarben durchlaufen, und ein höherer Wert führt zu einem höheren Geschwindigkeitsverschiebungseffekt.

Empfehlung: Aufgrund der Verwendung von RINK-Unterbrechungen kommt RINK manchmal "ins Stocken", wenn es gleichzeitig mit dem Befehl |MUSIC auf Geschwindigkeit 6 verwendet wird. Wenn Sie beides gleichzeitig verwenden möchten, ohne dass es zu Störungen kommt, verwenden Sie eine andere Geschwindigkeit für Musik (Sie können Geschwindigkeit 5 oder 7 verwenden, beide funktionieren gut).

14.7.1 2D-Autorennen

Das Beispiel für das Auto-Spiel finden Sie in der folgenden Liste. Der Sound des Motors soll das Gefühl der Beschleunigung vermitteln. Für die Zeichen an den Seiten

wurde ein Relativbewegungs-Sprite verwendet, das sich mit der gleichen Geschwindigkeit wie der Schritt des Streifens bewegt. Das Tintenmuster wird in Zeile 100 definiert.

| RINK,1,3,3,3,3,3,3,24,24,24,24: rem gelbe und rote Streifen

```
1 SPEICHER 24999
2 AUFRUF &6B78
3 FOR i=0 TO 31:|SETUPSP,i,0,0:|NEXT:|AUTOALL,0:|PRINTSPALL,0,0,0
4 |SETUPSP,31,9,16: |SETUPSP,31,0,1: vy=0
10 MODUS 0
20 DEFINT a-z
31 |LOCATESP,31,160,40: x=40
32 |SETUPSP,30,9,17: |SETUPSP,30,0,17:|LOCATESP,30,-20,10
40 CALL &BC02:'Standardpalette
50 GOSUB 430
60 TINTE 0,13
70 TINTE 14,10
80 linestinta=3
90 rangotintas=8
91 ' Erstellung des Farbmusters
100 |RINK,1,3,3,3,3,3,24,24,24,24,24: rem gelbe und rote Streifen
101 |RINK,0
110 y=400
120 ' FARBE STRASSE -----
121 tini=1
130 FOR strips=1 TO 10
140 FOR t=tini TO rangotintas+tini-1
150 FOR j=1 TO linestinta
160 PLOT 0,y,14:DRAW 136,y
170 PLOT 140,y,t:DRAW 160,y
180 PLOT 480,y,t:DRAW 500,y
190 PLOT 504,y,14:DRAW 640,y
200 y=y-2
210 Nächstes j
220 NÄCHSTE
240 NEXT-Streifen
250 skipb=-16:xc=65: cosa=0
270 REM-Spielzyklus -----
293 IF saltob=-16 THEN 296
294 IF jumpb>0 THEN thing=-jump ELSE Ding=Ding-1
295 IF thing<0 THEN |RINK,thing:vy=3*thing:posv=posv-3*thing:|MOVERALL,-vy,0:IF
saltob <=0 THEN thing=2-saltob
296 |DRUCKSACK
351 cycle=cycle+1:IF posv>240 THEN posv=-30:|LOCATESP,30,posv,xc:IF xc=10
THEN xc=65 ELSE xc=10
361 IF INKEY(27)=0 DANN IF x<52 THEN x=x+1:POKE 27499,x:GOTO 370
362 IF INKEY(34)=0 DANN IF x>21 THEN x=x-1:POKE 27499,x
370 IF INKEY(67)=0 DANN IF jumpb<16 THEN jumpb=jumpb+1:jump=jumpb/4
380 IF INKEY(69)=0 DANN IF jumpb>-16 THEN jumpb=jumpb-1:jump=jumpb/4
390 TON 1 ,6000/(Überspringen+17),1,15
    GOTO 270
421 REM PALETA
430 TIN 0 ,
    TE
440 TIN 1 , 5
    TE
450 TIN   ,
    TE
460 TIN   ,
    TE
470 TIN   , 26
    TE
```

TIN 5 , 0
TE
490 TIN ,
TE
500 TIN , 8
TE
510 TIN 8 , 10
TE

```

520 TINT 9, 12
E
530 TINT 10,
E
TINT ,
E
550 TINT , 0
E
560 TINT , 23
E
570 TINT , 0
E
580 TINT ,
E
590
RÜCKKEHR

```

14.7.2 Ziegelstein-Schriftrolle

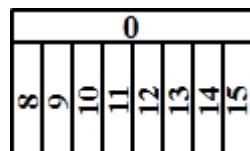
In diesem Beispiel kombinieren wir die Verwendung von Tintenanimation mit **|MAP2SP-basiertem** Scrollen. Mit der Tintenanimation werden wir ein Muster von Ziegeln bewegen, das wir mit einem sich wiederholenden Sprite gezeichnet haben, während das Schloss und der Baum mit **|MAP2SP** bewegt werden.

Das Verschieben der Steine wäre eine riesige Aufgabe, wenn es von der CPU erledigt würde, daher ermöglicht diese Technik das "Unmögliche". Es ist eine sehr leistungsfähige Technik, wenn man sie mit Einfallsreichtum einsetzt.



Abb. 93 Blättern mit Tintenanimation und MAP2SP gleichzeitig

Der verwendete Ziegelstein ist ein 8-farbiger Sprite mit dem unten abgebildeten Design:



Und das Farbmuster ist 0,6,3,3,3,3,3,3,3,3,3,3,3. Um es zu erstellen, führen Sie einfach den Befehl aus:

|RINK,8,0,6,3,3,3,3,3,3,3,3,3,3

Die Figur (Sprite 31) bleibt in der Mitte des Bildschirms, kann springen und sich in

beide Richtungen bewegen. Um die Bewegung von Tinten und MAP2SP zu synchronisieren, wird für den Befehl |RINK ein Schritt=2 eingestellt, da ein Byte zwei Pixel entspricht und MAP2SP sich auf Byte-Ebene bewegt.

Interessant ist, dass auch der Vogel (Sprite 30) von der Bewegung betroffen ist, da seine Geschwindigkeit zu der der Figur addiert oder von ihr subtrahiert werden muss.

Da ein 8-Farben-Muster verwendet und auch überschrieben wird, bleiben 2 Farben für den Hintergrund (Schloss, Äste) und 3 Farben für die Sprites (Figur und Vogel) übrig. Es ist einfach, das Programm so zu ändern, dass es ein 4-Farben-Rotationsmuster verwendet und somit 5 Farben für die Sprites hat, was vernünftiger ist.

Die vollständige Liste finden Sie unten.

```

10 SPEICHER 24999
11 BEI PAUSE GOSUB 5000
20 AUFRUF &6B78
30 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT:|AUTOALL,1:|PRINTSPALL,0,1,0
40 |SETUPSP,31,7,1:|SETUPSP,31,0,65:|LOCATESP,31,130,36:'Zeichen
50 |SETUPSP,30,7,7:|SETUPSP,30,0,157:|LOCATESP,30,50,80:
|SETUPSP,30,15,2: 'Vogel
60 MODE 0:DEFINT a-z
80 CALL &BC02:'Standard-Palette
90 GRENZEN 10
100 TINTE 6,15:TINTE 7,15
110 TINTE 4,26:TINTE 5,26
120 TINTE 2.0:TINTE 3.0
130 TINTE 1,14
140 ' Erstellung des Farbmusters
170 tini=8:|RINK,tini,0,6,3,3,3,3,3,3
200 |RINK,0
210 LOCATE 1,1:pen 4:PRINT "DEMO |RINK und |MAP2SP".
220 ' PAINT Wand -----
230 |SETUPSP,29,9,23:'Ziegelstein
231 y=152
232 FOR row=1 TO 6
240 FOR brick=xini bis 42 step 2:|PRINTSP,29,y,brick*2:next
241 xini=(xini-1) mod 2: y=y+8
242 weiter
390 dir=1:x=0:xp=80: cycle=40: stepy=2
400 |MUSIK,0,0,0,7
409 ' Spielzyklus -----
410 |AUTOALL:|PRINTSPALL
450 IF INKEY(27)=0 THEN |RINK, stepy:x=x+1:|MAP2SP,0,x:|MOVER,30,0,-
1:if jump=0 then IF dir=2 THEN dir=1:|SETUPSP,31,7,dir ELSE |ANIMA,31
460 IF INKEY(34)=0 THEN |RINK,-stepy::x=x-1:|MAP2SP,0,x:if jump=0 then IF
dir=1 THEN dir=2:|SETUPSP,31,7,dir ELSE |ANIMA,31
471 IF INKEY(67)+jump=0 THEN
jump=cycle:|SETUPSP,31,0,205:|SETUPSP,31,15,dir-1:|SETUPSP,31,7,31+dir
472 IF cycle-jump=20 THEN jump=0:|SETUPSP,31,7,dir:|MOVER,31,5,0:
490 |PEEK,27483,@xp:IF xp<-20 THEN |LOCATESP,30,50,80:'bird start again
501 Zyklus=Zyklus+1
502 IF xant=x THEN |MAP2SP,0,32000
503 xant=x:' IF still THEN I don't print the castle so it doesn't flash
510 GOTO 410
5000 |MUSIC:CALL &BC02:pen 1:MODE 2:END

```

15 Plattform-Spiele

Die Schwierigkeit bei der Programmierung eines Plattformspiels liegt vor allem darin, die Physik von Sprüngen und Plattformkollisionen korrekt zu handhaben. Etwas, das Sie in dem Videospiel "**Frisches Obst und Gemüse**" finden, das bei 8BP erhältlich ist.



Abb. 94 Frisches Obst und Gemüse

Sprünge:

Grundsätzlich habe ich für die Physik der Sprünge anstelle der Newtonschen Gleichung einen Pfad definiert, bei dem Vy bei -5 beginnt und von Bild zu Bild abnimmt. Wenn die Zenitposition erreicht ist, wird das Bild so verändert, dass es sich an der Spitze selbst auslöscht und die Geschwindigkeit Vy positiv wird und allmählich zunimmt.

Im Grunde genommen ist es wie die Anwendung der Newtonschen Gleichung, nur ohne die Berechnungen.

Bodenuntersuchung

Während die Figur läuft, prüfe ich in jedem Frame, ob es einen Boden gibt. Da die Plattformen zur Weltkarte gehören, verwenden sie niedrige Sprite-Identifikatoren (<10). Wenn ich also mit COLSPALL eine Zahl <10 erkenne, gibt es einen Boden. Wenn das Ergebnis der Erkennung eine 32 ist (Sprites gehen von 0 bis 31), dann gibt es nichts und der Charakter muss anfangen zu fallen. Feinde erkennen nichts, sie laufen einfach auf vordefinierten Routen, bei denen angegeben ist, wie viele Schritte sie in jede Richtung machen müssen und dann wieder von vorne beginnen. Von da an läuft das Sprite die Route Schritt für Schritt ab, indem es **|AUTOALL** aufruft (das intern bereits **|ROUTEALL** aufruft).

Kollisionen mit Plattformen:

Wenn sich die Figur auf dem Fallpfad befindet, bewege ich sie (ohne zu drucken) 5 Einheiten nach unten und stelle eine Sprite-Kollision fest. Dann bewege ich sie (ohne zu drucken) 5 Einheiten nach oben. Wenn die Kollision 32 ist, gibt es keine Kollision und ich lasse die Figur weiter fallen. Wenn die Kollision kleiner als 10 ist, ist die Figur mit einer Plattform kollidiert. In diesem Fall bewege ich die Figur so, dass sie perfekt mit dem Anfang der Plattform zusammenpasst, also: Position des Puppenkopfes ist "posy" Position der Füße ist posy+26, weil die Figur 21 misst und ich füge 5 hinzu, weil sie fällt (es gibt 5 der Selbstlöschung), so dass sie in Wirklichkeit 26 misst. Da die Plattformen in Vielfachen von 8 platziert wurden, behalte ich einfach den Rest der gesamten Division, die ich mit einem UND 7 erhalten kann, kurz gesagt, zwei sehr gut durchdachte Anweisungen, aber nur zwei:

```
dy = (posy%+26) AND 7  
|MOVER,31,5-dy,0
```

Dann weise ich die Geh-Animationssequenz zu, die nicht die Fallsequenz ist und deren Bilder 21 Bilder hoch sind.

16 Horden von Feinden in scrollenden Spielen

Scrolling-Spiele werden in der Regel als eine Abfolge von Horden von Gegnern verschiedener Typen und Flugbahnen gespielt, während Sie vertikal oder horizontal vorrücken.

Wenn Sie möchten, dass Ihre Karte 10 Horden zu verschiedenen Zeitpunkten des Kartenfortschritts (oder Spielzyklus) hat, könnten Sie 10 IF-Anweisungen verwenden, aber die Ausführung jedes Zyklus wäre sehr langsam (jede IF-Prüfung kostet eine Millisekunde).

Die beste Lösung ist die Beibehaltung von zwei Arrays, eines für die Position der Horde und eines für den Typ der Horde.

Index (Laufende Nummer der Bestellung)	Nexthorda (Zyklus, in dem es erscheinen soll)	Tipohorda (Art der Hordenfeinde)
0	100	1 - Flugzeug
1	200	2 - Flugkörper
		4 - UFOs
	320	3 - Drachen

```
10 dim tipohorda(10)
20 typeforda(0)=1: typeforda(1)=2: typeforda(2)=4:
t y p e f o r d a (3)=3:
30 dim nexthorda(0)
40 nexthorda(0)=100:nexthorda(1)=200: nexthorda(2)=250 ...
50 index=0

100 rem Spielzyklus
110 Zyklus=Zyklus +1
120 IF cycle = nexthorda(index) THEN GOSUB 500
...
500 Erstellung von horde
Fernbedienungsroute
nen
510 auf tipohorda(index) Gehe zu
600,700,800
520 rem RoutinErstellun horde Typ 4. AI end wir siehe
e g gülti werden
g tun
6000 rem RoutinErstellun horde Typ 1. AI end wir siehe
1010 RÜCKKEHR g gülti werden
g tun
rem RoutinErstellun horde Typ 2. AI end wir siehe
e g gülti werden
g tun
800 rem RoutinErstellun Horde Typ 3. AI end wir siehe
e g gülti werden
g tun
...
...
```

Anstatt den Zyklus zu verwenden, in dem er erscheinen soll, können Sie auch den Vergleich mit der Position auf der Karte machen, in der Sie sich befinden. Das hängt davon ab, wie Sie es machen wollen, aber mit dieser Idee der zwei Reihen haben Sie bereits den allgemeinen Ansatz, um Ihre Feindhorden zu organisieren.

17 Umdefinierbare Mini-Zeichen: PRINTAT

Der Amstrad-Zeichensatz ist schön und gut aufgebaut. Allerdings stehen im Modus 0 nur 20 Zeichen pro Zeile zur Verfügung, und diese erscheinen zu "breit", so dass sie manchmal nicht geeignet sind, um bestimmte Texte oder Markierungen in einem Spiel darzustellen. Außerdem ist der PRINT-Befehl sehr langsam, so dass es nicht empfehlenswert ist, die Markierungen häufig zu aktualisieren, da das Spiel während des Druckens "anhält", es sind zwar nur ein paar Millisekunden, aber es ist spürbar.

Aus diesem Grund wurde in der Version v31 von 8BP der Befehl PRINTAT hinzugefügt, der eine Zeichenkette mit einem neuen, kleineren Zeichensatz (ich nenne sie "Mini-Zeichen") drucken kann. Dieser neue Befehl erlaubt es, den Transparenzmechanismus der Sprites zu nutzen, so dass man Zeichen unter Berücksichtigung des Hintergrunds drucken kann. Er funktioniert wie folgt:

|PRINTAT, <Flag Transparenz>, y, x, @String

Beispiel:

```
cad$= "Hallo".
|PRINTAT, 0, 100, 10, @cad$
```



Abb. 95 PRINTAT

Der Befehl |PRINTAT druckt Zeichenketten und keine numerischen Variablen. Wenn Sie also eine Zahl ausdrucken möchten (z. B. die Punkte auf der Anzeigetafel in Ihrem Videospiel), müssen Sie dies tun:

```
points=points+1
cad$= str$(points)
|PRINTAT,0,100,10, @cad$
```

Der Befehl |PRINTAT wird nicht von den Grenzen für das Clipping beeinflusst, die mit |SETLIMITS. Dies ist sehr logisch, da Sie normalerweise PRINTAT verwenden werden, um Noten auf Ihren Markern zu drucken, die außerhalb des durch |SETLIMITS begrenzten Bereichs liegen.

Im Gegensatz zum BASIC-Befehl PRINT ist der Befehl |PRINTAT recht schnell und kann verwendet werden, um Ihre Spielmarkierungen häufig zu aktualisieren.

PRINTAT verwendet ein neu definiertes Alphabet, das eine reduzierte oder andere Version der "offiziellen" Amstrad-Zeichen enthalten kann. 8BP bietet standardmäßig ein kleines Alphabet bestehend aus Zahlen, Großbuchstaben, Leerzeichen und einigen Symbolen. Sie können keine Zeichen verwenden, die nicht in diesem Satz enthalten sind, wie z.B. Kleinbuchstaben. Die Zeichen dieses Alphabets sind alle gleich groß: 4 Pixel breit x 5 Pixel hoch, d. h. 2 Byte x 5 Zeilen.

Diese Zeichenfolge enthält alle Zeichen, die Sie mit dem 8BP-Serienalphabet verwenden können. Beachten Sie, dass es keine Kleinbuchstaben gibt und viele Symbole fehlen, obwohl Sie Ihr eigenes Alphabet erstellen können, das diese enthält. Das letzte Symbol ist das ". " "0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ:! ,."

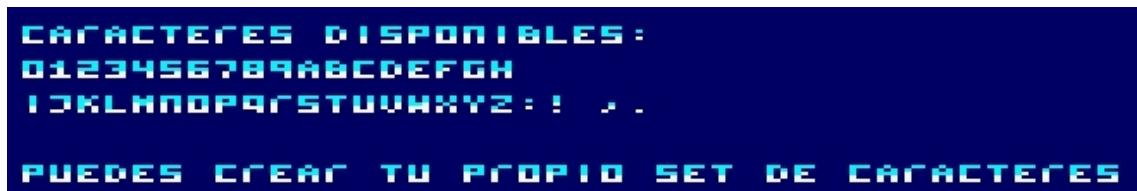


Abb. 96 Standard-Zeichensatz in 8BP zur Verwendung mit PRINTAT

WICHTIG: Wenn Sie versuchen, mit PRINTAT ein Zeichen zu drucken, das in dem erstellten Alphabet nicht vorhanden ist, wird das letzte Zeichen in der Zeichenliste gedruckt, was im Standardalphabet der Punkt "" ist.

Ich habe die Zeichen mit den Farben 2 und 4 erstellt, damit das Überdrucken möglich ist, da die Hintergrundfarben 0 und 1 sind und beim Überdrucken die Farbe 2 gleich 3 und die Farbe 4 gleich 5 sein muss (siehe das Kapitel, in dem ich das Überdrucken erkläre). Um Überdrucken zu verwenden, setzen Sie einfach das Transparenz-Flag im PRINTAT-Befehl auf "1".

17.1 Erstellen Sie Ihr eigenes Miniatur-Alphabet

Die mit dem Befehl PRINTAT zu verwendenden Zeichen werden in einer Datei im ASM-Verzeichnis definiert und aus der Datei "images.asm" importiert.

Schauen wir uns ein Fragment von "images.asm" an, so finden wir diese drei Zeilen:

```

wenn Sie nicht den Befehl |PRINTAT, sondern nur die Amstrad-Zeichen verwenden
wollen
Sie können dann die folgenden 3 Zeilen kommentieren
_ANFANG_ALPHABET
lesen "alphabet_default.asm"
_END_ALPHABET

```

Das Alphabet besteht aus einigen Daten und den Bildern der einzelnen Zeichen. All dies ist im 8BP-Bildspeicherbereich untergebracht. Das Standardalphabet ist knapp über 400 Byte groß.

Die Datei alphabet_default.asm enthält das Alphabet, das 8BP standardmäßig enthält. Sie können Ihre eigene Alphabetdatei erstellen. Diese Datei enthält 3 Variablen, die die Größe der Zeichen angeben, die Sie in jeder gewünschten Größe zeichnen können. Standardmäßig habe ich ein Alphabet mit einer Breite von 2 Byte und einer Höhe von 5 Zeilen erstellt, aber Sie können auch eine andere Größe verwenden oder sogar riesige Zeichen erstellen. Wenn Sie die Breite oder Höhe ändern, müssen Sie auch die Variable ALPHA_SIZE in Ihrer alphabet.asm-Datei entsprechend ändern.

```

ALPHA_width db 2; Breite Alphabet.alle Buchstaben sind gleich groß
ALPHA_height db 5; Höhe des Alphabets.alle Buchstaben sind
gleich ALPHA_span db 2*5

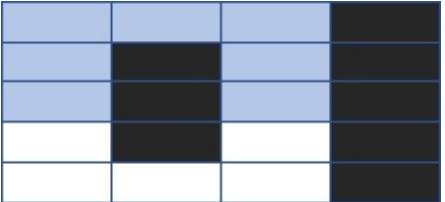
```

Als nächstes suchen wir die Zeichenfolge, die die gültigen Zeichen für den Befehl PRINTAT angibt. Diese Zeichen sind gültig, weil sie eine zugehörige Zeichnung haben. Nach dieser Zeichenkette werden die Zeichnungen all dieser Zeichen nacheinander gefunden, und zwar in der gleichen Reihenfolge, in der sie in der Textzeichenfolge ALPHA_LIST erscheinen

ALPHA_LISTE
"0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ:!. ; Zeichen erstellt
db 0 ;Null-Byte zeigt das Ende der Zeichenkette Alpha_list an

Die Bilder der einzelnen Zeichen in der ALPHA_LIST-Zeichenkette sind unten abgebildet. Ich habe sie mit dem SPEDIT-Tool erstellt. Das erste von ihnen muss das erste Zeichen der ALPHA_LIST-Zeichenkette sein, d. h. "0".

Die diesem Zeichen entsprechenden Bytes werden hier angezeigt:

Zeichen 0 db 12 , 8 db 8 , 8 db 8 , 8 db 32 , 32 db 48 , 32	
--	--

Dann werden wir einen nach dem anderen den Rest der Buchstaben, Zahlen und Symbole definieren. Mit etwas Geduld können Sie Ihre eigene Reihe von Mini-Zeichen erstellen, die Ihrem Videospiel mehr Persönlichkeit verleihen werden. Sie brauchen nur die zu erstellen, die Sie verwenden werden, und je weniger es sind, desto weniger Speicherplatz werden Sie im Bildbereich verwenden.

Denken Sie daran, dass, wenn Sie versuchen, ein Zeichen zu drucken, das Sie nicht erstellt haben, das letzte der in der ALPHA_LIST definierten Zeichen gedruckt wird.

17.2 Standardalphabet für MODE 1

Das Standardalphabet von 8BP wurde in MODE 0 erstellt und wenn Sie versuchen, es in MODE 1 zu verwenden, wird es nicht richtig funktionieren, da es sich um Zeichnungen handelt, die in Modus 0 erstellt wurden. 8BP wird standardmäßig mit einem Alphabet geliefert, das in MODE 1 funktioniert. Sie müssen nur eine Zeile in images.asm ändern. Dieses Alphabet ist inspiriert von einer Schriftart namens "5th agent".

_ANFANG_ALPHABET
lesen "alphabet_default_mode1.asm"
_END_ALPHABET



0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ : ! .

18 Pseudo-3D

In den 1980er Jahren waren Autospiele im Stil von "Pole Position" sehr beliebt. Sie vermittelten ein 3D-Gefühl, aber sie führten keine 3D-Berechnungen durch, nur für den Straßenplan, und manchmal nicht einmal das, da es sich um Annäherungen handelte, die ein gutes Gefühl für Geschwindigkeit vermittelten.

In den Arcade-Automaten wurden spezielle Chips verwendet, um die "Sprite-Skalierung" vorzunehmen, wodurch die Sprites gleichmäßig größer wurden, und die Berechnung der Straße mit Hilfe eines speziellen Chips (z. B. dem "Sega Road Chip"), der ausschließlich dazu diente, die Straße mit ihren Streifen zu zeichnen. Die Straßenkarte wurde dann zur Sprite-Map hinzugefügt, und das endgültige Bild wurde zusammengesetzt. Diese Chips wurden in Arcade-Automaten wie "Pole Position" und "Space Harrier" verwendet.



Abb. 97 Pole Position und Out Run (Arcade-Automaten)

Das gemeinsame Merkmal sowohl der Software- (auf 8-Bit-Computern) als auch der Hardware-Technik (auf Arcade-Automaten) ist, dass die Kurven eine Illusion sind: Die Straße wird verformt, während die Berge am Horizont verschoben werden, um den Eindruck einer Kurve zu erwecken, aber es gibt gar keine Kurve. Die Ergebnisse waren oft sehr überzeugend, aber die Kurven waren in Wirklichkeit eine Illusion.

Die auf 8-Bit-Computern verwendeten Techniken waren sehr vielfältig. Alles war akzeptabel, solange der Spieler den Eindruck hatte, er befände sich auf einer Rennstrecke. In vielen Fällen wurde die Tintenrotation verwendet, um ein Gefühl von Geschwindigkeit zu vermitteln. Viele Spiele litten unter einer niedrigen Bildrate von unter 5 fps. Zu den besten Spielen für Amstrad gehörten "**3D Grand Prix**" (unter Verwendung von Software-Sprite-Skalierung in Kombination mit Tintenanimation), "**Buggy Boy**", das auf einer sehr fortschrittlichen Programmiertechnik des Rastereffekts basierte, und einige andere wie "**Crazy Cars**" oder "**Chase HQ**".

Ab Version V32 von 8BP steht Ihnen die Pseudo-3D-Fähigkeit zur Verfügung, die in dem Demospiel "3D Racing one" verwendet wird. Es ist sehr einfach zu benutzen und mit ihm können Sie Ihre eigenen Rennspiele, Panzerspiele, Schiffsspiele, etc. erstellen.

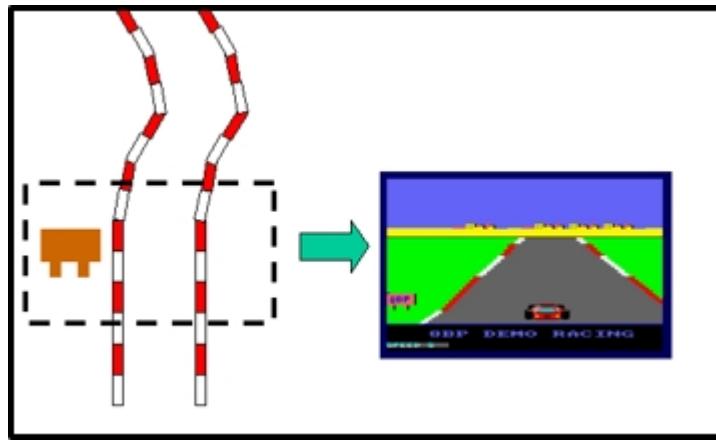
Um Pseudo-3D in 8BP zu verwenden, müssen Sie die "**Assembler-Option**" 3 verwenden, wodurch 24 KB für Ihr BASIC-Programm frei bleiben.

Die von 8BP bereitgestellten Möglichkeiten zur Bereitstellung von Pseudo-3D sind:

- 1) **3D-Projektion:** Diese besteht darin, eine 2D-Weltkarte in 3D zu projizieren, so dass wir, auch wenn wir sie in 2D durchlaufen, das Gefühl haben, sie in 3D zu sehen. Dies geschieht durch den Aufruf des Befehls |3D zur Konfiguration der

PRINTSPALL-Befehle.

und PRINTSP, so dass sie in 3D projizieren, bevor sie auf den Bildschirm gemalt werden. Es wird keine Möglichkeit geben, sich in der Ebene zu drehen, wir werden immer nach vorne "schauen", aber der kombinierte Effekt einer Kurve zusammen mit Häusern oder Bergen am Horizont, die sich bewegen, wird



simulieren, dass wir eine Kurve nehmen.

Abb. 98 3D-Projektion

- 2) **Zoom:** Hier geht es darum, verschiedene Versionen desselben Bildes eines Objekts zu verwenden, um einen Zoom-Effekt bei der Annäherung an das Objekt zu erzielen. Dies geschieht einfach in der Bilddatei, indem man 3 Versionen desselben Bildes definiert und sie in einer Struktur gruppier. Das Bild zeigt ein typisches Beispiel für das Heranzoomen des Posters. Die Befehle |PRINTSPALL und |PRINTSP wählen die am besten geeignete Version des Bildes je nach der Entfernung, in der es sich befindet, ohne dass mehr getan werden muss, als das Bild als "Zoom"-Bild zu definieren.



Abb. 99 Bilder zoomen

- 3) **Segmente:** Dies besteht aus der Möglichkeit, Sprites vom Typ "Segment" zu haben, die durch eine einzige horizontale Scanlinie definiert sind (so dass sie sehr wenig Platz einnehmen) und denen wir eine Länge und eine Neigung zuordnen können. Mit ihnen können wir Straßen, Flüsse usw. und Straßenränder bauen. Wir werden dies einfach in der Bilddatei tun, indem wir einen Bildtyp definieren, der zusätzlich zu Breite und Höhe auch eine Neigung hat. Wir werden diese Segmente bei der Erstellung der Weltkarte verwenden.



Abb. 100 Segmente mit unterschiedlichen Neigungen

Die im Spiel "3D Racing one" verwendeten Segmente sind rot oder weiß und obwohl sie lang sind, werden sie nur durch eine Scanline definiert. Zum Beispiel besteht das linke weiße Segment aus einigen grünen Bytes für das Gras, einigen weißen und einigen grauen Bytes für die Straße. Beim Drucken wird das so definierte Segment auf die gewünschte Länge vervielfältigt und perspektivisch gedruckt, so dass es, auch wenn es ein gerades Segment ist, krumm erscheint.

Schließlich ist zu erwähnen, dass es in vielen Fällen notwendig sein wird, die Karte dynamisch zu aktualisieren (Befehl |UMAP). Dank dieses Befehls kann die Karte sehr groß sein (was notwendig ist, wenn wir einen Schaltkreis mit sehr vielen Segmenten erstellen). Dieser Befehl wird im Kapitel Scroll beschrieben.

Im Folgenden werden wir uns diese 3 Funktionen genauer ansehen und erläutern, wie Sie sie in Ihren Programmen einsetzen können.

18.1 3D-Projektion

Zum Projizieren haben wir den Befehl |3D

Use

|3D, 1, <sprite_fin>, <offsety> : REM aktiviert die 3D-Projektion.
|3D, 0 REM deaktiviert die 3D-Projektion

Dieser Befehl aktiviert die 3D-Projektion im Befehl |PRINTSP und in |PRINTSPALL. Das bedeutet, dass vor der Ausgabe auf dem Bildschirm die "projizierten" Koordinaten berechnet und dann auf dem Bildschirm ausgegeben werden. Die Koordinaten der Sprites sind davon nicht betroffen, d.h. die Koordinaten bleiben dieselben, allerdings in der 2D-Welt. Auf dem Bildschirm haben wir nun eine 3D-Ansicht und die Koordinaten, an denen sie gedruckt werden, sind das Ergebnis der Ausführung der Projektionsfunktion auf ihre 2D-Koordinaten.

Die betroffenen Sprites sind alle von Sprite 0 bis <sprite_fin>. Der Rest der Sprites wird nicht projiziert, sondern einfach in ihren 2D-Koordinaten auf dem Bildschirm ausgegeben.

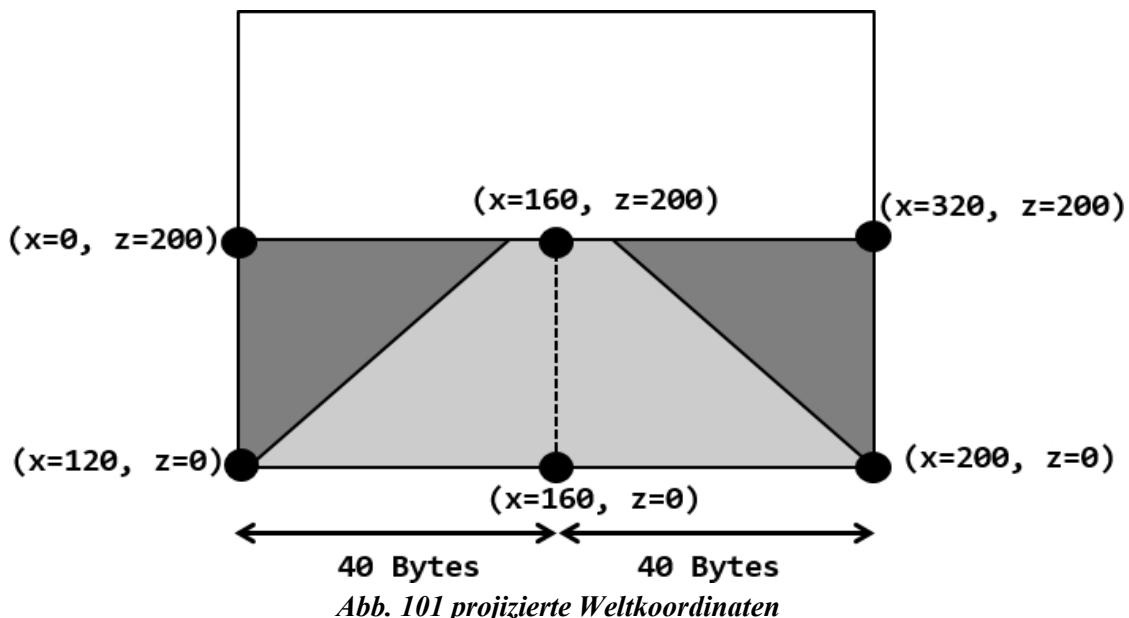
Dieser Befehl hat **keinen Einfluss auf Kollisionsmechanismen**, d.h. wenn wir COLSPALL verwenden und eine Kollision zwischen projizierten Sprites erkennen, wird die Kollision in der 2D-Ebene stattfinden. In manchen Fällen kann es vorkommen, dass

sich zwei projizierte Sprites teilweise in der

Das bedeutet jedoch nicht, dass sie kollidiert sind. Der eine kann dem anderen leicht voraus sein und sie können sich bei der Projektion überlappen, aber das ist keine echte Kollision. Kollisionen werden in der 2D-Ebene erkannt.

Der letzte Parameter `<offsety>` dient dazu, höher oder niedriger zu projizieren, so dass wir die Spielmarkierungen dort platzieren können, wo wir wollen. Beim Projizieren des Bildschirms, der 200 Pixel hoch ist, wird er 100 Pixel hoch, also können wir wählen, wie hoch wir die Projektion platzieren. Wenn ein Sprite nicht von der Projektion betroffen ist, weil es höher als `<Sprite_fin>` ist, dann ist es auch nicht von `<offsety>` betroffen. Dies ist zum Beispiel bei den Wolken und Häusern am Horizont im Spiel "3D Racing one" der Fall.

Um die Bildschirmkoordinaten zu verstehen, auf denen Ihre projizierten Sprites schließlich erscheinen, empfehle ich Ihnen, den folgenden sehr einfachen mathematischen Abschnitt zu lesen, um alles zu verstehen. Die folgende Abbildung zeigt die Koordinaten der Weltkarte, die auf bestimmte repräsentative Punkte des Bildschirms projiziert werden, wenn MAP2SP mit (`yo=0, xo=0`) aufgerufen wird. Die Z-Koordinate stellt die Entfernung dar, in der sich die Objekte befinden, auch "Tiefe" genannt.



Wenn wir statt (`xo=0, yo=0`) eine andere Koordinate für MAP2SP verwenden, werden die 2D-Weltkoordinaten, die den im Bild referenzierten Punkten entsprechen, um (x, z) verschoben, wie durch (xo, yo) angegeben.

18.1.1 Mathematik der Pseudo-3D-Projektion

Bei der Pseudo-3D-Projektion wird die Bodenebene, auf der sich unsere 2D-Weltkarte befindet, auf den Bildschirm projiziert.

Berechnung der Y-Koordinate

Unser Boden kann unendlich sein, aber wir projizieren nur 200 Pixel lang. Diese Länge wird "Tiefe" oder "Z-Koordinate" genannt. Diese 200 Pixel Tiefe werden bei der

Projektion auf 100 Zeilen Höhe komprimiert (projizierte Y-Koordinate). Die Pixel

Die am weitesten entfernten projizierten Objekte sind die Horizontlinie. Beachten Sie, dass wir keine sehr weit entfernten Objekte am Horizont sehen werden, sondern nur solche, die sich in einer Tiefe von höchstens 200 Metern befinden.

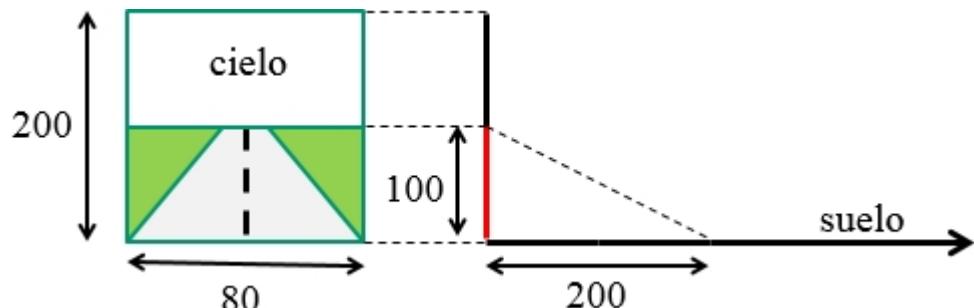


Abb. 102 Projektion des Bodens auf den Bildschirm

Je weiter Objekte entfernt sind, desto kleiner werden sie. Es handelt sich nicht um eine lineare Beziehung zwischen dem Bildschirm und dem Boden, d.h. um zu wissen, wie hoch ein Pixel gedruckt werden muss, das sich in einer bestimmten Entfernung befindet, ist es falsch zu denken, dass es ausreicht, durch zwei zu dividieren, da die abgedeckte Tiefe 200 beträgt und es auf 100 Zeilen hoch projiziert wird. Bei einer linearen Funktion (wie $y = f(z) = A \cdot z + B$) ist die Ableitung (A) konstant, aber bei der Projektion ist das, was konstant ist, die "zweite Ableitung" der Funktion $f(z)$. Wir werden uns dies nun im Detail ansehen.

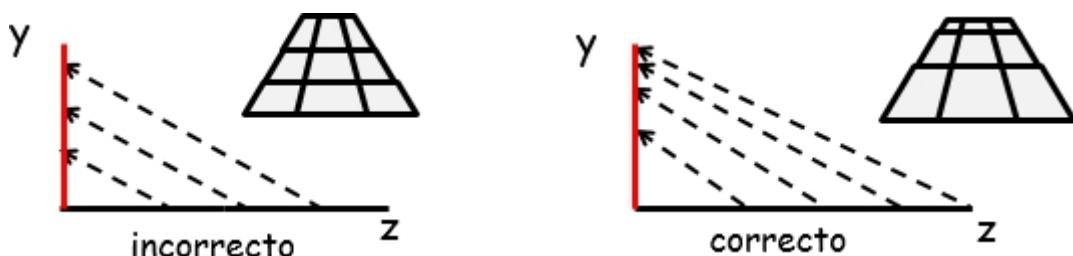


Abb. 103 Die richtige Projektion ist nicht linear

Angenommen, wir beginnen mit " $Z=0$ " und wollen wissen, wie viel " Y " wert ist. Das ist sehr einfach, denn an der Grundlinie ($z=0$) ist die y -Koordinate ebenfalls null. Wenn wir z mit einer kleinen Änderung " dz " inkrementieren, ist das " y " das vorherige " y " (Null) plus ein Inkrement dy , das anfangs gleich dem Inkrement dz ist, da das Inkrement klein war und wir uns in der Nähe des Horizonts befinden.

dy (ursprünglich) = dz

Wenn wir uns nun wieder von dz entfernen, muss das Inkrement dy kleiner werden, und wenn wir uns wieder von dz entfernen, wird das zu addierende dy immer kleiner. Mit anderen Worten:

**Jedes Mal, wenn wir uns von dz entfernen, fügen wir y ein Inkrement hinzu,
jedes Mal, wenn wir uns von dz entfernen, fügen wir y ein Inkrement hinzu,
jedes Mal, wenn wir uns von dz entfernen, fügen wir y ein Inkrement hinzu.**

Zeit ist weniger

199

$z=z+dz$

**$dy = dy + ddy$, wobei ddy negativ $y=y$
+ dy**

Der Anstieg von dy ist konstant. Wir haben diesen Zuwachs ddy genannt. Nun, dy ist die "Ableitung" von y , während " ddy " die so genannte "zweite Ableitung" ist. Hier ist die Ableitung nicht konstant, aber die zweite Ableitung ist es.

Der konstante Wert, den wir " ddy " zuweisen, führt zu mehr oder weniger übertriebenen Projektionen. In 8BP ist der ddy -Wert negativ, etwa -0,005, so dass der dy -Wert an der Grundlinie fast 1 beträgt, während an der Skyline durch die Akkumulation von 200 ddy der dy -Wert bei Null landet.

Trotz der Einfachheit dieser Berechnungen ist es für unseren geliebten Amstrad CPC kostspielig, sie in Echtzeit auszuführen. Daher macht 8BP eine Annäherung, um Berechnungen zu vermeiden und "z" in "y" auf eine viel einfachere Weise und mit einem sehr ähnlichen Ergebnis zu übersetzen.

Wenn $0 \leq z < 50$, dann ist $dy = dz$, also $y = z$

Wenn $50 \leq z < 110$, dann ist $dy = dz/2$, also $y = 50 + (z-50)/2$

Wenn $110 \leq z \leq 200$, dann ist $dy = dz/4$, also $y = 50 + (110-50)/2 + (z-110)/2$

Das heißt, wir haben den Bildschirm in drei Streifen unterteilt, und jeder Streifen wird als "lineare" Fläche behandelt (da " dy " konstant ist), aber mit einem unterschiedlichen Wert von " dy " im Verhältnis zu den anderen Streifen. Diese Annäherung führt zu visuell sehr ähnlichen Ergebnissen wie die mathematisch korrekten.

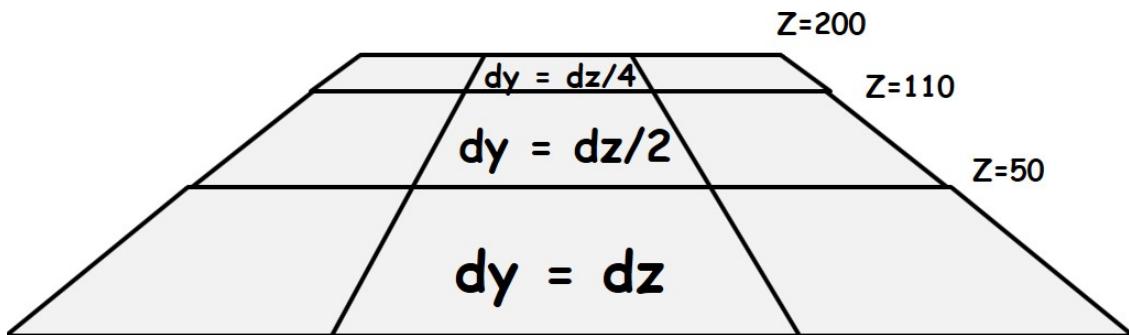


Abb. 104 Ansatz von 8BP

Die in 8BP verwendeten Gleichungen berücksichtigen auch, dass die Bildschirmkoordinaten eigentlich "invertiert" sind, d. h. die Null-Koordinate ist die obere Koordinate und die 200-Koordinate ist die untere Koordinate, aber dies ist nur eine sehr einfache letzte Anpassung.

Wenden wir uns nun der Berechnung der "X"-Koordinate zu:

Es gibt einen grundlegenden Unterschied zwischen der Horizontlinie und der Bodenlinie. Die Bodenlinie misst 80 Bytes, aber die Horizontlinie stellt mehr Breite dar, weil die Straße gerade ist und was auf dem Boden 80 misst, misst in der Horizontlinie einen Bruchteil, speziell in 8BP misst es 4-mal weniger. Die Straße verengt sich am Horizont, weil der Horizont viel mehr darstellt. In der von 8BP angewandten Projektion entspricht er 320 Bytes.

Und wenn der Horizont 320 und der Boden 80 misst, liegt das daran, dass die gesamte reale Fläche, die wir nach der Projektion auf dem Bildschirm sehen können, eine trapezförmige Fläche ist. Es ist NICHT so, dass der Boden ein Trapez ist, sondern dass die Fläche des Bodens, die wir auf dem Bildschirm sehen können, trapezförmig ist.

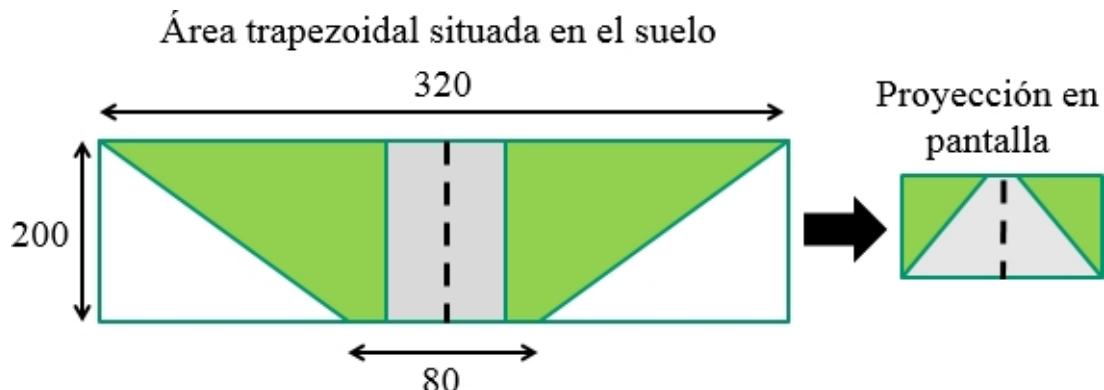


Abb. 105 Angezeigter trapezförmiger Bereich

Intuitiv und ohne irgendwelche Gleichungen zu zeigen, ist bereits klar, was wir tun wollen und wie die Projektion funktioniert. Schauen wir uns nun die Gleichungen an.

Die Mathematik der "X"-Koordinate ist im Wesentlichen dieselbe wie die der "Y"-Koordinate. Sobald wir die Y-Koordinate berechnet haben, können wir eine lineare Gleichung $x=f(y)$ aufstellen, denn aufgrund der Nichtlinearität von Y in Bezug auf Z ist es so, als würde man eine nichtlineare Beziehung zwischen X und Z herstellen. Vorhin haben wir gesagt, dass der Horizont 320 Bytes lang ist und der Boden 80 Bytes. Das bedeutet, dass der Boden 4-mal kleiner ist als der Horizont. In der Ferne müssen wir durch 4 dividieren (Dividieren ist teuer, daher ziehen wir es vor, mit dem Faktor 0,25 zu multiplizieren), während wir in der Nähe mit dem Faktor = 1 multiplizieren müssen.

Wir müssen die X-Koordinate des zu projizierenden Objekts in Bezug auf die Mitte des Bildschirms zentrieren. Anschließend multiplizieren wir mit einem Faktor, der von der projizierten "Y"-Koordinate abhängt. Auf diese Weise wird eine nichtlineare Beziehung zwischen "X" und "Z" hergestellt.

Wir müssen eine Gleichung aufstellen, die ein Ergebnis liefert, das z. B. am Horizont viermal kleiner ist:

$\text{Faktor} = ((100-y)+32)/2$ $x = (\text{x - Mitte}) * \text{Faktor} + \text{Zentrum}$ wenn z=200, dann y=100, und dann Faktor =16 wenn z=0 dann y=0, und dann Faktor =64 (4 mal mehr)

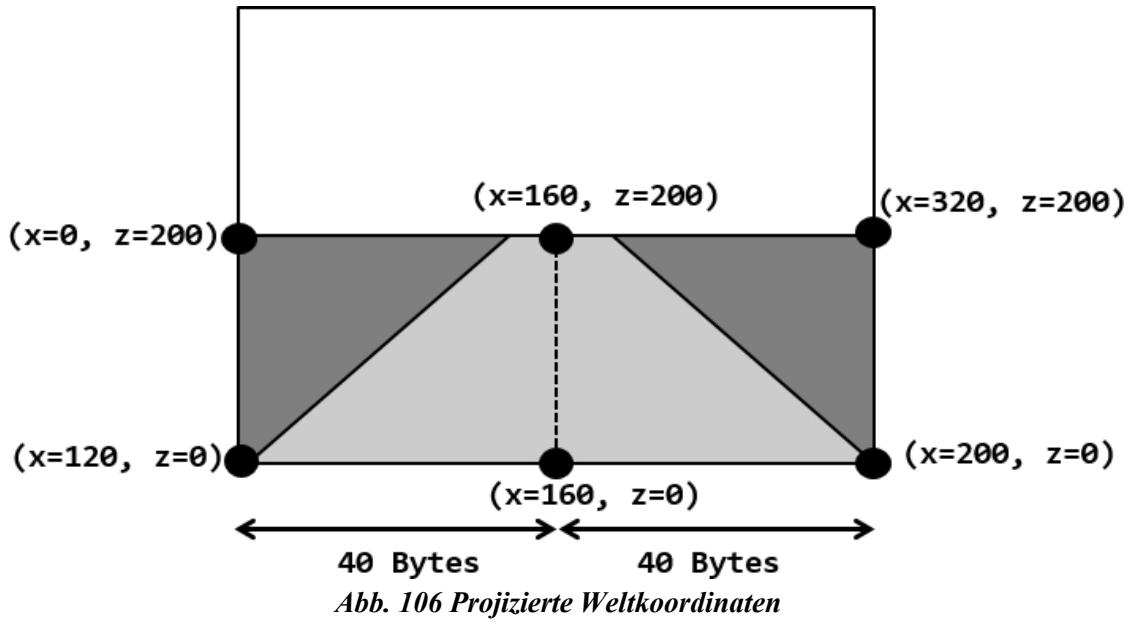
Die gewählte Gleichung ist interessant, weil die Division durch 2 etwas ist, das der Amstrad durch Drehen eines Bits im Binärformat durchführen kann. Das heißt, er kann es in ein paar Taktzyklen tun.

Wie aus der Gleichung hervorgeht, genügt es, die X-Koordinate zu zentrieren und sie dann mit dem erhaltenen Faktor zu multiplizieren. Die so erhaltene Zahl ist sehr groß, denn im Falle der Horizontlinie werden wir mit 16 und im Falle der Bodenlinie mit 64 multipliziert, d. h. wir haben die X-Koordinate mit einem Faktor multipliziert, dessen Wert zwischen 16 und 64 liegt. Zwischen dem Horizont und dem Boden haben wir alle 48 möglichen Werte für den Faktor, d.h. selbst wenn der Faktor eine ganze Zahl ist, wird er sich gleichmäßig entwickeln.

Dann dividieren wir durch 64, was einer 6-fachen Drehung im Binärformat entspricht, und das war's. Letzteres ist gleichbedeutend mit der Multiplikation des Horizonts mit 0,25, des Bodens mit 1 und des Dezimalwerts, der einer beliebigen Zwischenhöhe zwischen Boden und Horizont entspricht, aber wir haben es mit ganzen Zahlen gemacht, die der Amstrad schnell verarbeiten kann. Diese Technik wird "Festpunktarithmetik"

genannt.

Wenn wir all dies berücksichtigen und MAP2SP bitten, die Sprites der Weltkarte aus den Koordinaten ($yo=0$, $xo=0$) zu generieren, dann werden wir auf dem Bildschirm die folgenden Koordinaten der Welt sehen. Ich bin sicher, dass die Abbildung jetzt leicht zu verstehen ist.



Wie Sie ableiten können, wird der Punkt ($x=0, y=0$) der Weltkarte vom Bildschirm weg projiziert, er wird nicht angezeigt. Wenn wir anstelle von ($xo=0, yo=0$) eine andere Koordinate für MAP2SP verwenden, werden die 2D-Weltkoordinaten, die den im Bild referenzierten Punkten entsprechen, um (x, z) verschoben, wie durch (xo, yo) angegeben.

18.1.2 Kurven

Wie ich bereits zu Beginn dieses Kapitels erwähnt habe, erlaubt die von 8BP verwendete 3D-Projektion keine Drehungen der Ebene, so dass wir einen kleinen Trick anwenden müssen, um eine Kurve zu simulieren. Dazu müssen wir die Straße nach rechts oder links drehen, während wir Sprites wie Häuser oder Berge am Horizont verschieben. Diese Sprites werden nicht projiziert und um dies zu vermeiden, werden wir Bezeichner über `<Sprite_fin>` verwenden. Denken Sie daran, dass wir die 3D-Projektion aktivieren müssen:

|3D, 1, <Sprite_fin>, <offsety>, <offsety>, <offsety>, <offsety>, <offsety>.

Folglich wird ein scheinbarer Rundkurs mit Kurven auf unserer 2D-Karte einfach als eine Straße mit Steigungen nach rechts und links dargestellt.

Mit dieser Strategie können wir den Eindruck eines Rennfahrers erwecken, der auf einer Rennstrecke mit echten Kurven fährt, und mit Hilfe von Häusern oder Bergen am Horizont, die sich entgegengesetzt zur X-Koordinate bewegen, den Eindruck erwecken, dass sich sein Auto in den Kurven dreht. Wenn Sie ein guter Künstler sind und verschiedene Segmente nach und nach mehr oder weniger stark verdrehen, vermitteln Sie den Eindruck sehr realistischer Kurven.

Diese Strategie ist rechnerisch sehr effizient und ausreichend realistisch. Wenn wir die Bodenebene wirklich drehen wollten, müssten wir zum Drehen eine Matrixberechnung

mit vielen sehr teuren Operationen durchführen, und außerdem müssten die Sprite-Texturen auf dem Boden ebenfalls gedreht werden, so dass der Rechenaufwand enorm wäre.

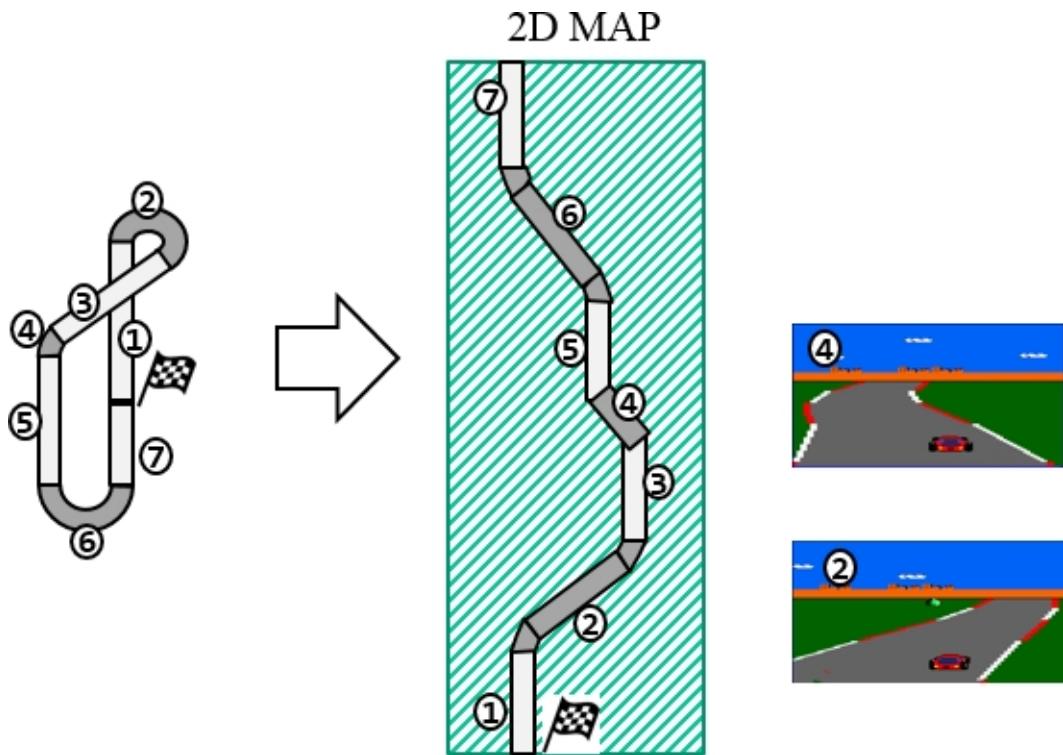


Abb. 107: Imaginärer Stromkreis und 2D-Weltkarte

Die heutigen Computer sind so leistungsfähig, dass die hier vorgestellten Strategien bedeutungslos sind, aber es sind Strategien, die der heutigen "rohen Gewalt" an Eleganz und Einfallsreichtum überlegen sind. Denken Sie daran, dass "Beschränkungen kein Problem sind, sondern eine Quelle der Inspiration".

Sie müssen den Befehl **|UMAP** verwenden, um große Schaltpläne zu durchlaufen, aber denken Sie daran, **|UMAP** nicht bei jedem Zyklus, sondern nur von Zeit zu Zeit aufzurufen.

18.2 Bilder vergrößern

Um ein "Zoom-Bild" zu definieren, erstellen wir einfach die verschiedenen Versionen des Bildes (3 Versionen). Dann suchen wir in der Bilddatei "images_mygame.asm" nach einem Tag namens "**_3D_ZOOM_IMAGES**".

Wir werden es finden:

```
_BEGIN_3D_ZOOM_IMAGES
;=====
Grenzen, die für alle gezoomten Bilder gelten
Der Horizont für diese Grenzwerte liegt bei 0 und steigt nach unten
bis 200
_ZOOM_LIMIT_A
db 120; zwischen 200 (Masse) und limitA wird auf Bild 3 gesetzt
_ZOOM_LIMIT_B
db 50
zwischen dieser Grenze und der Grenze A wird das Bild 2 gesetzt.
näher am Horizont als Grenzwert B ist Bild 1 eingestellt
;=====
```

CARTEL_ZOOM

db 1; Breite symbolisch

db 1; Höhe symbolisch

dw POSTER1, POSTER2, POSTER3

END_3D_ZOOM_IMAGES

Alle ZOOM-Bilder müssen nach dem Tag "**BEGIN_3D_ZOOM_IMAGES**" definiert werden. Dies sind Bilder, die eine symbolische Breite und Höhe haben, denn in der Realität wird ein Sprite, dem eines dieser Bilder zugewiesen ist, die Breite und Höhe der Version des Bildes verwenden, die automatisch entsprechend seiner Y-Koordinate ausgewählt wird. Das heißt, "CARTEL1" ist ein normales, zuvor definiertes Bild, dessen Breite, Höhe und Bytes die Zeichnung enthalten. Das Gleiche gilt für "CARTEL2" und "CARTEL3". Wenn wir das Bild "CARTEL_ZOOM" einem Sprite zuordnen (dies kann durch die Zuordnung eines Bezeichners zu diesem Bild am Anfang der Bilddatei geschehen), wird je nach Position des Sprites auf dem Bildschirm die eine oder andere Version des Bildes gedruckt.

Für die automatische Auswahl des Bildes werden Schwellenwerte für die Y-Koordinaten festgelegt. Sie können diese Schwellenwerte ändern, aber die Standardwerte funktionieren gut und sind es auch:

- zwischen Horizont =0 und 50, wird das erste Bild ausgewählt (im Beispiel "CARTEL1").
- zwischen 50 und 120 liegt, wird das zweite Bild ausgewählt (im Beispiel "CARTEL2").
- zwischen 120 und 200 liegt, wird das dritte Bild ausgewählt (im Beispiel "CARTEL3").

Die Auswahl dieser Grenzen zur Abgrenzung des Bildschirms ist konfigurierbar, und Sie können so viele einstellen, wie Sie möchten. Beachten Sie, dass die Auswahl auf der Grundlage der Y-Koordinate des nicht projizierten Sprites getroffen wird. Einmal projiziert, variiert seine Y-Koordinate stark, aber es ist nicht das projizierte "Y", das zur Abgrenzung der 3 Streifen verwendet wird, sondern das "Y" des Sprites in 2D. Wenn sich das Sprite von einem Streifen zum anderen bewegt, ändert sich sein Bild automatisch. Wenn Sie es vorziehen, ein Bild zuerst erscheinen zu lassen, können Sie die Schwellenwerte ändern.

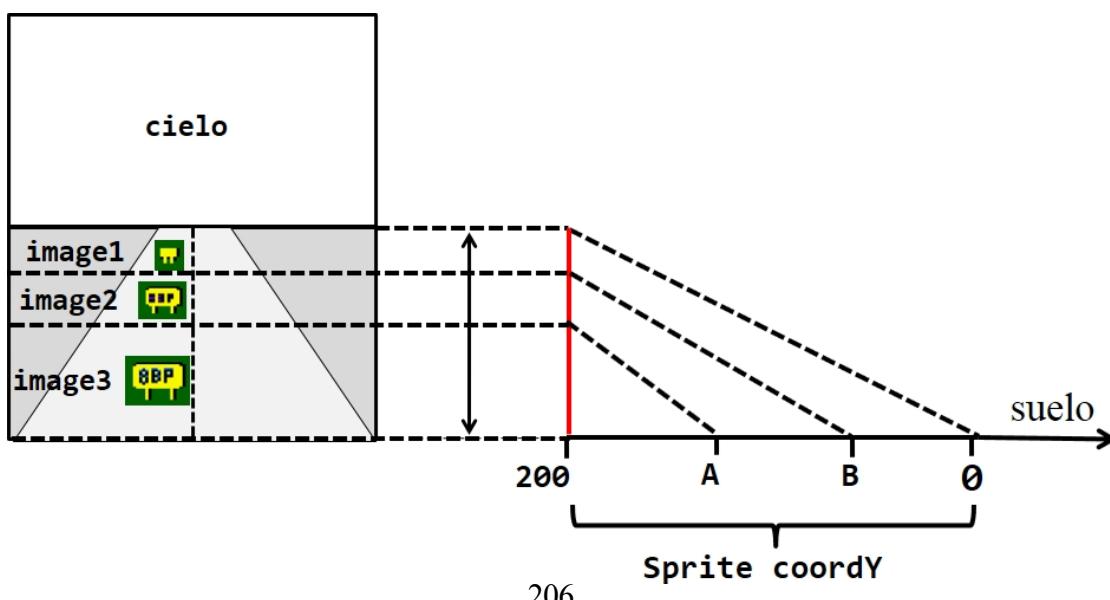


Abb. 108 Verwendungsbereiche der 3 Versionen des ZOOM-Bildes

Zum Beispiel:

```
REM nehmen wir an, dass CARTEL_ZOOM in der Bilddatei id=16 hat  
|SETUPSP,20,9,16 : REM assoziiert CARTEL_ZOOM mit Sprite 20  
|LOCATESP, 20,100, 160: REM Sprite bei auf Zentrum des  
"Trapezes  
(coordy=100)  
|3D,1,31,0: REM alle Sprites werden projiziert.  
|PRINTSP,20: REM Version 2 des POSTERs wird gedruckt.
```

18.3 Verwendung von Segmenten

Da Sie nun wissen, wie man eine 2D-Karte erstellt, die eine Rennstrecke mit Kurven "simuliert", finden Sie hier eine leistungsfähige Methode, um die Segmente mit verschiedenen Neigungsgredienten zu erstellen, die Sie für Rennstrecken oder Straßen benötigen.

Ein Segment ist ein einzelnes zeilenhohes Bild, das durch Wiederholung jede beliebige Länge erreichen kann. Zusätzlich zur Länge hat es einen weiteren Parameter, nämlich die Gesamtneigung des Segments in Bytes. Diese Neigung kann negativ (Neigung nach rechts) oder positiv (Neigung nach links) sein.

Um diese Art von Bildern zu definieren, müssen Sie sie in der Bilddatei Ihres Spiels "images_mygame.asm" nach dem Tag "_BEGIN_3D_SEGMENTS" erstellen.

```
;-----  
_BEGIN_3D_SEGMENTS  
;-----  
Es könnte eine andere Definition von Segmenten geben.  
Die Breite ist die Breite der Scanline.  
Der Höchstwert ist der 2D-Höchstwert des Segments.  
; dann kommt dx, das positiv (nach links gekippt) oder negativ (nach  
rechts gekippt) sein kann.  
db 0; dies ist für das erste Bild des Segmenttyps, das > sein soll  
_3D_SEGMENTE  
  
;----- SEGMENT_EDGE_LWI10 -----  
SEGMENT_KANTEN_LWI10  
db 22; Breite  
db 50; hoch  
db 10; dx  
db 192,192,192,192,192,192,192,192,192,192,192,192,192,192,192,240, 240  
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  
;
```

Im Beispiel haben wir ein Segment mit Gras auf der linken Seite (grüne Bytes), dann zwei weiße Bytes und dann graue Bytes (Straße) auf der rechten Seite definiert. Ob es grün oder grau ist, hängt von den Farben ab, die wir den Tinten zugeordnet haben.

Es handelt sich um ein Segment mit einer Schrägen von 10 Bytes nach links. Hätten wir eine Null in die Schrägen (dx) gesetzt, wäre es ein gerades Segment, nicht schief. Es ist 50 Zeilen hoch, aber wenn es projiziert wird, ist es weniger, es sei denn, es ist sehr eng.

Die Punkte des Segments, die projiziert werden, sind nur zwei. Nach der Projektion werden die Scanlines nacheinander mit einer bestimmten horizontalen Verschiebung gezeichnet, so dass die letzte Linie am Endpunkt beginnt. Beachten Sie, dass das Segment zwar perspektivisch gezeichnet wird, seine Breite aber konstant ist. Sie malen nicht den oberen Teil schmäler (weiter weg) und den unteren Teil breiter (näher), sondern Sie malen jede Scanlinie genau so, wie sie in der Bilddatei definiert wurde.

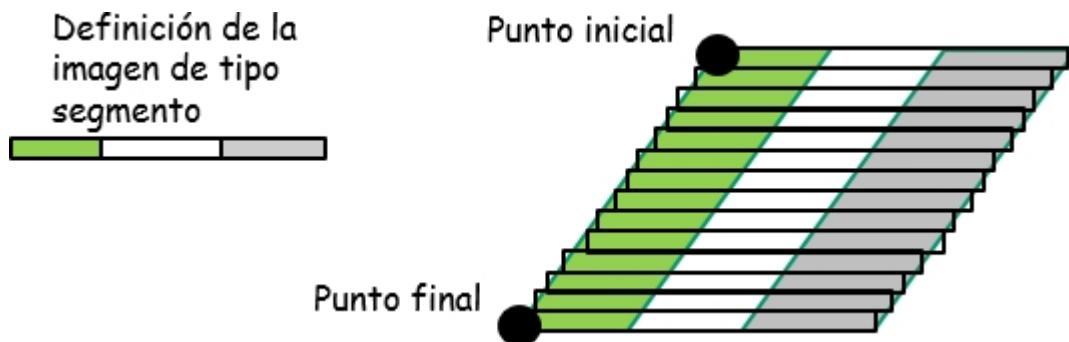


Abb. 109 2 Punkte werden auf ein Segment projiziert

Wie Sie sehen können, habe ich viele Gras- und Straßenabdrücke verwendet. Das ist so, dass sich das Segment "selbst löscht", wenn es sich vorwärts bewegt, obwohl, wenn das Auto zu schnell fährt, Spuren bleiben können. Sobald Sie Ihr Spiel zum Laufen gebracht haben, werden Sie überprüfen, ob die Geschwindigkeit zu hoch ist und Spuren zurückbleiben.



Abb. 110 Zusammenhang zwischen dem Neigungsgrad eines Segments und der Höchstgeschwindigkeit

Wie in der Abbildung zu sehen ist, kann die maximale Vorschubgeschwindigkeit für ein selbstlöschendes Segment höher sein, wenn der Grad der Schräglage moderat ist. Wenn es sehr schief ist, kann die Geschwindigkeit nicht so hoch sein, da sonst Spuren zurückbleiben. Sobald das Segment projiziert ist, wird es aufgrund des perspektivischen Effekts kürzer, und je nachdem, wo es sich befindet, kann es noch mehr oder weniger schief sein. Es ist ratsam, sie breit zu machen, damit sie sich sicherer auslöschen.

In dem Spiel "3D Racing one" gibt es eine Stufe namens "superfast", in der ich mit weniger gebogenen Segmenten die Geschwindigkeit des Autos erhöht habe, ohne dass die Segmente Spuren hinterlassen. Ein einfacher und sehr effektiver "Trick". Wenn das Spiel langsam ist (z. B. ein Panzerspiel, das langsam sein soll), dann können die Segmente sehr krumm sein, da sie aufgrund des Geschwindigkeitseffekts keine Spuren hinterlassen werden.

Kurz gesagt, um die Geschwindigkeit zu erhöhen, haben Sie zwei Möglichkeiten:

- Weniger verdrehte Segmente verwenden
- Verwenden Sie breitere Segmente (mit mehr Spielraum zum Selbstlöschen).

19 Musik

Die Werkzeuge, über die ich in diesem Abschnitt spreche, sind nicht von mir programmiert, aber sie sind in 8BP integriert und sie sind wirklich gut.

19.1 Musikbearbeitung mit WyZ-Tracker

Dieses Tool ist ein Musiksequenzer für den AY3-8912 Soundchip. Die erzeugte Musik kann exportiert werden und resultiert in zwei Dateien

- Eine Instrumentendatei ".mus.asm".
- Eine Datei mit Musiknoten ".mus".

Die einzige Einschränkung ist, dass alle Songs, die Sie in Ihr Spiel integrieren, dieselbe Instrumentendatei haben müssen.

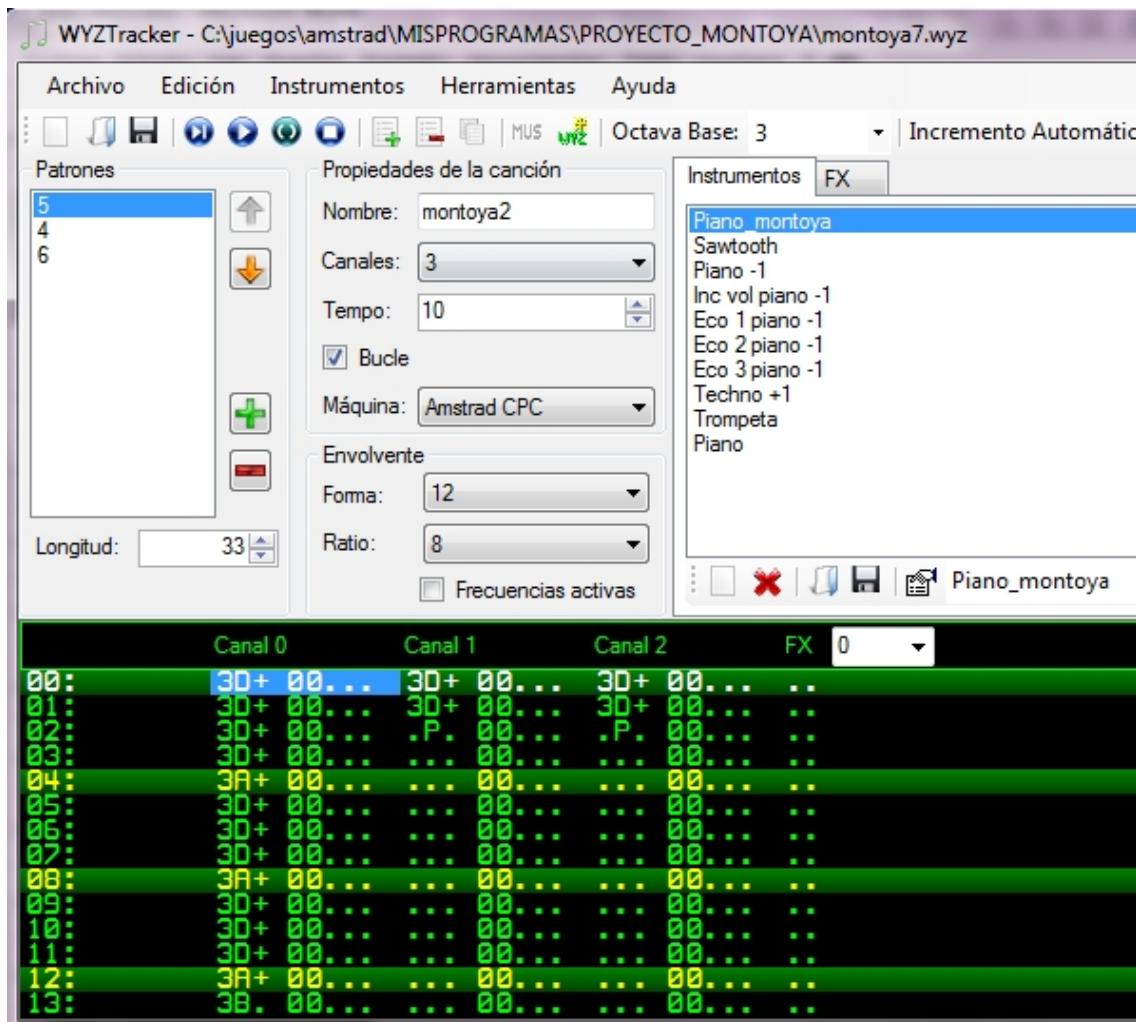


Abb. 111 WYZ-Tracker

Seit Version V41 können Sie den "FX"-Kanal verwenden, um Klangeffekte einzubinden. Bis zu dieser Version wurde dieser Pseudokanal im 8BP-Player ignoriert. Seien Sie vorsichtig, wenn Sie die Musik bearbeiten, denn obwohl der FX-Kanal immer mit einem Kanal verbunden ist, klingt er im WYZ-Tracker so, als wäre er unabhängig und das Ergebnis kann sich von dem unterscheiden, was Sie im WYZ-Tracker hören.

Wenn der Effekt dem Kanal 0 zugeordnet ist und Sie einen Effekt zu einem Zeitpunkt einsetzen, zu dem Kanal 0 nicht abgespielt wird, ist im Amstrad nichts zu hören, auch wenn Ihr Effekt im WYZtracker zu hören ist.

Ergänzt wird dieser Musiksequenzer durch den WYZplayer, der in die 8BP-Bibliothek integriert ist.

Seit Version V26 von 8BP kann Musik mit WYZtracker Version 2.0.1.0 komponiert werden und es funktioniert wirklich gut. Bis Version V25 von 8BP war die Kompatibilität mit WYZtracker 0.5.07 und es gab einige kleine Probleme, aber all das ist mit WYZtracker 2.0.1.0 verschwunden.

WICHTIG: Verwenden Sie nicht die Oktave 7, auch wenn der Tracker dies erlaubt. Diese Oktave kann nicht abgespielt werden und es kann zu Synchronisationsfehlern bei der Musikwiedergabe auf dem Amstrad kommen.

Eine wichtige Empfehlung beim Erstellen Ihrer Songs mit WyZtracker ist es, die Instrumente zu löschen, die Sie nicht verwenden werden. Auf diese Weise nimmt die Instrumentendatei (die auf ".mus.asm" endet) weniger Platz ein, und da 8BP nur **1.500 Bytes** für die Musik reserviert, ist jedes Byte wichtig.

WICHTIG: Wenn Sie die Musik mit einer Version von WYZtracker bearbeiten, die später als die 2.0.1.0 nicht richtig funktionieren und es kann zu seltsamen Effekten kommen, wie z. B. dass ein Kanal nicht abgespielt wird oder die Noten nicht synchronisiert sind. Wenn dies bei Ihnen der Fall ist, besteht die Lösung darin, eine Datei von Grund auf mit WYZtracker 2.0.1.0 zu erstellen und die Musik, die nicht funktioniert, manuell Note für Note zu kopieren.

19.2 Zusammenstellung der Lieder

Wenn du deinen Song komponiert hast und die beiden Dateien hast, bearbeitest du einfach die Datei make_music.asm und fügst deine Musikdateien wie folgt ein:

```
nach dem Zusammenbau speichern mit save "musica.bin",b,32200,1400

org 32200
;-----MUSICA-----
Die Einschränkung ist, dass Sie nur eine einzige Datei von
Instrumenten für alle Lieder
Die Einschränkung wird dadurch gelöst, dass einfach alle
Instrumente in einer einzigen Datei.

Instrumentendatei. HINWEIS ES MUSS NUR EINE SEIN
Lesen Sie      "instruments.mus.asm"

Musikdateien SONG_0:
INCBIN      "micancion.mus" ;
SONG_0_END:

SONG_1:
```

```

INCBIN      "another_song.mus"
; SONG_1_END:

SONG_2:
INCBIN      "third_song.mus"      ;
; SONG_2_END:

LIED_3:
LIED_4:
LIED_5:
LIED_6:
LIED_7:

```

Zum Schluss stellen Sie die 8BP-Bibliothek wieder zusammen, so dass der Musikplayer (der in die Bibliothek integriert ist) die Instrumentenparameter und den Ort, an dem die Songs zusammengestellt wurden, kennt.

Dazu müssen Sie lediglich die Datei make_all.asm zusammenstellen, die wie folgt aussieht

```

Makefile für Videospiele mit 8bit-Power
wenn Sie nur ein Teil ändern, müssen Sie nur das entsprechende
Fabrikat montieren
Sie können zum Beispiel make_graphics zusammenstellen, wenn Sie
Zeichnungen ändern
;-----CODIGO -----
;enthält die 8bp-Bibliothek und den
MusikplayerWYZ lesen "make_codigo.asm".

;-----MUSICA-----
; enthält die Lieder. lies
"make_musica.asm"

----- GRAFIKEN -----
Dieser Teil enthält Bilder und Animationssequenzen.
und die Sprite-Tabelle mit diesen Bildern und Sequenzen aus
"make_graphics.asm" initialisiert.

```

Und damit haben Sie alles zusammen. Jetzt müssen Sie Ihre 8BP-Bibliothek wie folgt erstellen:

SAVE "8BP.LIB", b, 24000, 8200

Und die Musik:

SAVE "music.bin", b, 32200, 1400

19.3 Was tun, wenn Sie keine Musik in 1400 Bytes unterbringen können?

Es ist möglich, dass 1400 Bytes nicht für Ihre Lieder ausreichen. Falls ein Lied nicht passt (und Sie wissen das, indem Sie überprüfen, wo das "_END_MUSIC"-Tag assembliert wird), können Sie eine andere Assembler-Adresse für dieses und die folgenden Lieder angeben. Im Fall des "Nibiru"-Videospels machst du das mit dem dritten Lied, indem du es an einer niedrigeren Adresse als die 8BP-Bibliothek

assemblierst (z.B. 23000 würde funktionieren). Falls Sie dies tun, muss Ihr BASIC-Spiel mit einer

MEMORY-Befehl, der eine Adresse unterhalb dieser neuen Grenze angibt, z.B. MEMORY 22999 würde funktionieren.

Ab 8BP Version 34 wird die Bibliothek ab Adresse 24000 zusammengebaut. Wenn Sie also zusätzlichen Platz für Musik verwenden wollen, müssen Sie Adressen unterhalb von 24000 belegen. Zum Beispiel, von 23500 bis 23999.

Dies ist die

```
nach dem Zusammenbau speichern mit save "musica.bin",b,32200,1400
org 32200
;-----MUSICA-----
ist darauf beschränkt, nur eine Datei von ; Instrumenten für
Die Einschränkung wird dadurch gelöst, dass einfach alle Lieder
eingefügt werden.
Instrumente in einer einzigen Datei.

Instrumentendatei. HINWEIS ES MUSS NUR EINE SEIN
Lesen Sie      "../MUSIC/nibiru5.mus.asm" ;

Musikdateien SONG_0:
INCBIN      "../MUSIC/attack5.mus" ;
SONG_0_END:

SONG_1:
INCBIN      "../MUSIC/nibiru5.mus" ;
SONG_1_END:

org 23500 ; Ich benutze diese Zeile, weil ich das dritte Lied nicht
unterbringen kann!!!
SONG_2:
INCBIN      "../MUSIC/gorgo3.mus" ;

SONG_3:
LIED_4:
LIED_5:
LIED_6:
LIED_7:
-END_MUSIC
```

Die gleiche Art von Lösung ist anwendbar, wenn Sie nicht alle Ihre Grafiken in den von 8BP reservierten Bereich einpassen können, obwohl Sie, da Sie 8540 Bytes für Grafiken haben, weniger wahrscheinlich dieses Problem haben.

20 C-Programmierung mit 8BP

Zu Beginn dieses Handbuchs habe ich Ihnen empfohlen, BASIC-Compiler wie Fabacom oder CPC BASIC 3 nicht zu verwenden, weil sie den Speicherplatz einschränken (Sie verlieren etwa 20 KB). Wenn Sie die Geschwindigkeit Ihrer Spiele erhöhen wollen, steht Ihnen ab 8BP v40 ein Wrapper der 8BP-Bibliothek zur Verfügung, der von C aus verwendet werden kann, sowie ein kleiner Satz von BASIC-Befehlen, die von C aus aufgerufen werden können (die ich "Minibasic" nenne), so dass Sie Ihr BASIC-Programm fast sofort übersetzen können und das gewünschte schnelle Ergebnis erhalten.

Mit dieser neuen Funktion haben Sie 3 Möglichkeiten:

- 1) **Erstellen Sie Ihr Programm zu 100 % in BASIC** (d. h. verwenden Sie die Funktionalität nicht). Diese Option vereinfacht die Programmierung erheblich, aber Sie sind weniger schnell.
- 2) **Erstellen Sie Ihr Programm zu 100 % in C**. Diese Option ist komplex, da das Programmieren, Kompilieren, Suchen und Korrigieren von Fehlern viel langsamer ist als das Programmieren in BASIC.
- 3) **Erstellen Sie Ihr Programm in BASIC und übersetzen Sie am Ende nur den Spielzyklus in C**. Diese Option ist genauso einfach wie die erste, mit Ausnahme der letzten Phase der C-Übersetzung.

Ich werde die dritte Möglichkeit erklären, die sehr interessant ist, weil sie es Ihnen erlaubt, in BASIC zu programmieren, und zwar mit den Vorteilen, die es bietet (einfache Programmierung, Fehlererkennung und -korrektur, usw.). Wenn Sie Ihr Spiel komplett in C programmieren wollen, dient Ihnen diese Erklärung genau so, also haben Sie keine Angst und lesen Sie weiter. Ein kommerzielles Beispiel für Option 3 ist das berühmte und mythische Spiel "galactic plague", eine Indescomp-Produktion, die 1984 von dem hervorragenden Programmierer Paco Suárez entwickelt wurde. Dem großen Paco Suárez verdanken wir viele Stunden der Unterhaltung.

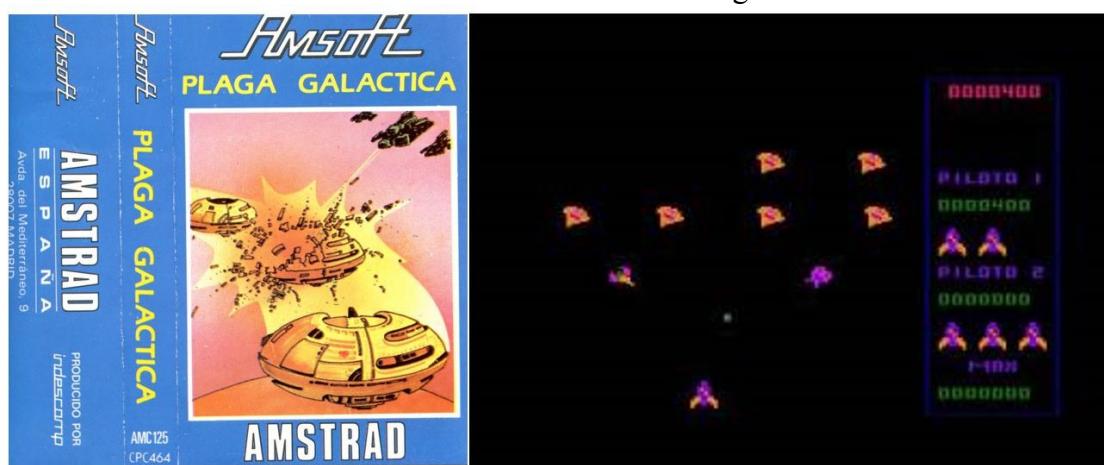


Abb. 112 Die "galaktische Seuche".

Um in C programmieren zu können, brauchen wir einige Werkzeuge, aber keine Sorge, Sie müssen nicht lernen, wie man sie benutzt, denn 8BP macht das für Sie, dank einer .bat, die sich um alles kümmert, wie Sie gleich sehen werden. Die Werkzeuge sind:

- **compila.bat**: ist die .bat-Datei, die die verschiedenen Werkzeuge in Cascade aufruft und aus einer .c-Datei eine .dsk-Datei mit einer Binärdatei macht. Sie finden dieses Tool im Unterverzeichnis "C" von 8BP.

- **SDCC** (Small Device C compiler): Sie müssen ihn herunterladen und installieren. Sie finden ihn unter <http://sdcc.sourceforge.net/>. Dies ist wahrscheinlich der beste C-Compiler für Z80 (obwohl SDCC auch andere Mikroprozessoren unterstützt). Es gibt einen weiteren

Ich wählte den Z88dk, aber ich zog es vor, den SDCC zu wählen, weil einige Tests zeigen, dass das resultierende kompilierte Programm mit SDCC schneller ist als das z88dk-Äquivalent.

- **Hex2bin.exe:** Wir werden es (von der .bat) benutzen, um die Datei, die wir mit SDCC in hexadezimaler Form erzeugt haben, in binäre Form umzuwandeln. Sie müssen es nicht herunterladen, es ist klein und Sie finden es im Unterverzeichnis "C" von 8BP.
- **ManageDsk.exe:** Wir werden diese Datei (aus der .bat) verwenden, um unsere kompilierte Binärdatei in eine .dsk-Datei zu übertragen. Sie brauchen sie nicht herunterzuladen. Sie ist klein und Sie finden sie im Unterverzeichnis "C" von 8BP.

Lassen Sie uns die notwendigen Schritte anhand eines Beispiels durchgehen, und dann werde ich detailliert erklären, wie man jede der 8BP-Funktionen von C aus aufruft, sowie die neuen "Minibasic"-Befehle, die 8BPV40 Ihnen zur Verfügung stellt, um in C zu programmieren, als ob Sie in BASIC wären.

20.1 Erster Schritt: Programmiere dein BASIC-Spiel

Wir werden das Beispielprogramm verwenden, das mit der 8BP-Bibliothek geliefert wird. Es ist ein sehr einfaches Spiel, in dem du einen Soldaten spielst, der Bällen ausweichen muss, die in verschiedene Richtungen vom Himmel fallen. Jedes Mal, wenn ein Ball Sie trifft, verlieren Sie Punkte, die mit der Zeit wachsen.



Abb. 113 Der Beispielsatz

Die Liste des Spiels ist wie folgt, wobei ich den Teil, der dem "Spielzyklus" entspricht, rot hervorgehoben habe.

```

10 SPEICHER 19999
11 LOAD "loop.bin",20000: REM Kompilierte Spielschleife laden
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
30 BEI PAUSE GOSUB 320
40 CALL &BC02:'restore default palette just in case'.
50 INK 0,0: 'schwarzer Hintergrund
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'Sprites zurücksetzen
70 |SETLIMITS,0,80,0,124: ' die Grenzen des Spielbildschirms festlegen
80 PLOT 0,74*2:ZEICHNEN 640,74*2
90 x=40:y=100:' Koordinaten des Zeichens
100 PRINT "ERGEBNIS:      FPS:"
```



```

110 |SETUPSP,31,0,1+32:' Zeichenstatus
    |SETUPSP,31,7,1'Animationssequenz beim Starten zugewiesen
```

```

130 |LOCATESP,31,y,x:'platziert das Sprite (ohne es schon zu
drucken)
140 |MUSIC,0,0,0,0,5: Punkte=0
150 cor=32:cod=32:|COLSPALL,@cor,@cod:' Kollisionsbefehl setzen
    LOCATE 1,20:INPUT "basic(1) oder C(2)", a: IF a=1 THEN 160 ELSE CALL &56b0
    GOTO 320
160 |PRINTSPALL,0,0,0,0,0: 'Druckbefehl konfigurieren
161 POKE &B8B4,0: POKE &B8B5,0: POKE &B8B6,0: POKE &B8B7,0:'Reset Timer
cpc6128. Dies ist notwendig, da TIME eine sehr große Zahl zurückgeben kann und
geb Überlauf mit DEDFINT
en. t1=TIME
162
170 '--- Spielzyklus. Dies ist der Teil, der in C übersetzt wurde ---  

c=c+1
190 ' liest die Tastatur und positioniert das Zeichen
191 IF INKEY(27)=0 THEN IF dir<>>0 THEN |SETUPSP,31,7,1:dir=0 ELSE
|ANIMA,31:x=x+1:GOTO 195
192 IF INKEY(34)=0 THEN IF dir<>>1 THEN |SETUPSP,31,7,2:dir=1 ELSE
|ANIMA,31:x=x-1
195 |LOCATESP,31,y,x
200 |AUTOALL:|PRINTSPALL
210 |COLSPALL
220 IF cod<32 THEN BORDER 7:dots=dots-1:LOCATE 7,1:PRINT          Punkte:GOTO
221 REM, um die FPS zu berechnen, nehmen wir an, dass TIME mich in Einheiten von
1/300 Sekunden angibt und ich alle 20 Zyklen messen werde. Daher fps= 20 Zyklen x 300
/ dt, wobei dt= t2-t1
230 IF c MOD 20=0 THEN dots=dots+10 :LOCATE 7,1:PRINT dots:
t2=t1:t1=TIME:fps=6000/(t1-t2):LOCATE 17,1:PRINT fps
240 IF c MOD 5=0 THEN |SETUPSP,i,9,9,19:|SETUPSP,i,5,4,RND*3-
1:|SETUPSP,i,0,11:|LOCATESP,i,10,RND*80: i=i+1:IF i=30 THEN i=0
    IF c<500 THEN GOTO 180
251 '--- Endspielzyklus ---
252 |POKE,42038,Punkte
310 Ende des Spiels
320 |MUSIC:INK 0,0: PEN 1:BORDER 0:|PEEK,42038,@dots
330 LOCATE 3,10:PRINT "FINAL SCORE:";dots

```

20.2 Schritt 2: Übersetzen Sie Ihren BASIC-Spielzyklus in C

Um den Spielzyklus in C zu übersetzen, müssen wir ein C-Programm schreiben, das wir cycle.c nennen werden.

Gehen Sie in den Unterordner "C" von 8BP. Dort finden Sie alles, was Sie brauchen, und das gleiche Beispiel mit der Datei ciclo.c.

Das erste, was Sie wissen müssen, wenn Sie den Spielzyklus in C programmieren, ist, dass Sie zwei kleine Bibliotheken einbinden müssen: den 8BP-Wrapper (8BP.h) und die minibasic (minibasic.h). Die 8BP.h befindet sich im Unterverzeichnis "8BP_wrapper" und die minibasic.h befindet sich im Unterverzeichnis "mini_basic".

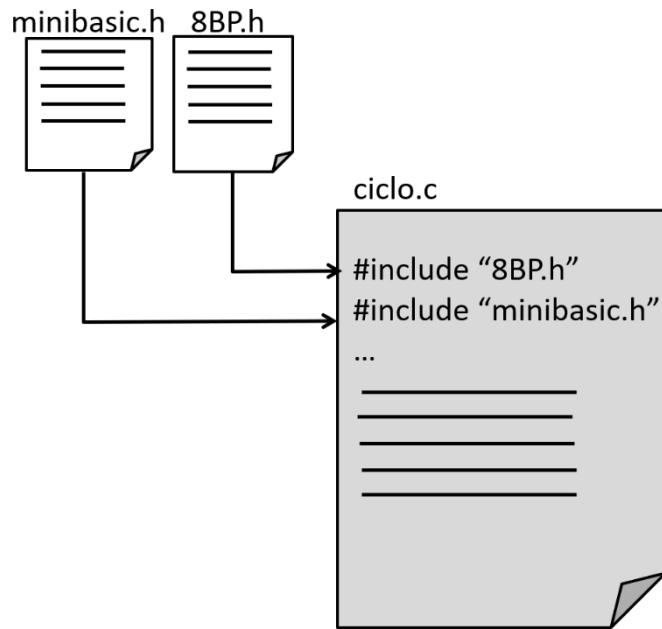


Abb. 114 zu kompilierende Dateien

Dies ist das C-Listing, das, wie Sie sehen können, eine direkte Entsprechung zu dem Teil des BASIC-Listings hat, **der dem Spielzyklus entspricht**, praktisch eine wörtliche Übersetzung. Sie werden einige Labels wie "label_195" sehen, die als Zeilennummern dienen, um mit GOTO springen zu können. Es handelt sich praktisch um die BASIC-Liste, die Anweisung für Anweisung übersetzt wird, ohne dass man über die Programmierung nachdenken muss. In diesem einfachen Fall gibt es nur eine Funktion (die Hauptfunktion) und sie kehrt zurück, wenn die Zeit abgelaufen ist.

Um diesen Schritt zu machen, haben Sie "minibasic", um Ihnen bei der Übersetzung von BASIC nach C zu helfen, aber wenn Sie ein Experte in C sind und die AMSTRAD-Firmware kennen oder eine andere Hilfsbibliothek haben, können Sie den Spielzyklus direkt in C machen, ohne ihn vorher in BASIC programmiert und validiert zu haben. Der Weg, den ich Ihnen vorschlage, ist einfach und leistungsfähig, Ihr Programm wird laufen wie ein Windhund, aber hier sind Sie frei, es zu tun, wie Sie wollen.

WICHTIG: Denken Sie bei der Programmierung in C daran, dass die Notation für Dezimal-, Hexadezimal- und Binärzahlen lautet:

mivariable = 165 ; //dezimale Schreibweise
mivariable = 0xA5 ; //hexadezimale
Schreibweise mivariable = 0b10100101; //binäre
Schreibweise

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdio.h>

#include "8BP.h"
#include "minibasic.h"
#include "8BP.h"
#include "minibasic.h"

//alle Variablen als global deklarieren, um die Möglichkeit zu haben
//Zugriff auf sie von jeder Funktion aus, wie in BASIC
// obwohl sie hier nicht initialisiert werden
//-----
int c;
```

```

char dir;
int x;
int y;
int cod;
int cor;
int i;
int Punkte;
int t1;
int t2;
int fps;
*****
MAIN
*****
int main()
{
    //Initialisieren Sie die
    Variablen c=0;
    dir=0;
    x=40;
    y=100;
    cod=32;
    cor=32;
    i=0;
    Punkte=0;
    fps=0;
    t1=_Grundzeit();

    //Befehle konfigurieren
    _8BP_printspall_4(0,0,0,0);
    _8BP_colspall_2(&cor,&cod);

    //Zyklus des Spiels
    //-----
    label_CICLE:
    c=c+1;

    if (_basic_inkey(27)==0) {
        if (dir !=0) {
            _8BP_setupsp_3(31,7,1);
            dir=0;
        }
        sonst {
            _8BP_anima_1(31);
            x=x+1;
            goto label_195;
        }
    }
    if (_basic_inkey(34)==0) {
        if (dir !=1) {

```

```

    _8BP_setupsp_3(31,7,2);
    dir=1;
}
sonst {
    _8BP_anima_1(31);
    x=x-1;
}
}

label_195:
//-----
_8BP_locatesp_3(31,y,x);
_8BP_autoall();
_8BP_printspall();
_8BP_colspall();

wenn (cod<32) {
    _basic_border(7);
    _basic_sound(1,100,14,0,0,1,0);
    puntos=puntos-1;
    _8BP_setupsp_3(cod,0,9);
    _basic_locate(7,1);
    _basic_print(_basic_str(dots)); goto
    _label_250;

}
else _basic_border(0);
if (c %20 ==0) {
    Punkte=Punkte+10;
    _basic_locate(7,1);
    _basic_print(_basic_str(points));
    t2=t1;t1= _basic_time();
    fps=6000/(t1-t2);
    _basic_locate(17,1);_basic_print(_basic_str(fps));
}

wenn (c %5 ==0){
    _8BP_setupsp_3(i,9,19);
    _8BP_setupsp_4(i,5,4,_basic_rnd(3)-1);

    _8BP_setupsp_3(i,0,11);_8BP_locatesp_3(i,10,_basic_rnd(80));
    i=i+1;if (i==30) i=0;
}

_label_250:
if (c<500 goto label_CICLE;

    _8BP_poke_2(42038,points);
    return 0;
}

```

}

Wie Sie sehen können, werden die 8BP-Funktionen wie folgt aufgerufen:

8BP<Funktion>_<N>(Parameter)

N ist die Anzahl der Parameter der Funktion, da es von jeder Funktion Versionen mit einer unterschiedlichen Anzahl von Parametern gibt (wie in den RSX-Versionen der Befehle).

Und grundlegende Funktionen, die in C nicht verfügbar sind, die wir aber in Minibasic erstellt haben, werden wie folgt aufgerufen:

basic<function>(Parameter)

Wie Sie sehen können, ist es sehr einfach, ein BASIC-Listing Ihres Spielzyklus in ein cycle.c-Listing zu übersetzen, wenn Sie die Übersetzung der einzelnen Befehle kennen.

Es ist notwendig, das LOCOMOTIVE BASIC mit dem C-Programm zu kommunizieren. Zum Beispiel gibt es Daten, die Sie an das C-Programm weitergeben möchten, wie die Anzahl der verbleibenden Leben oder die gesammelten Punkte. Hierfür können Sie die Funktionen verwenden:

- Von LOCOMOTIVE BASIC haben Sie **PEEK**, **POKE**, **|PEEK** und **|POKE**.
- Von C haben Sie **_basic_peek()**, **_basic_poke()**, **_8BP.Peek_2()**, **_8BP.Poke_2()**

Mit diesen Funktionen können Sie eine Speicheradresse reservieren, um die Leben, Punkte, Phasen usw. zu speichern und so den Spielzyklus jedes Mal aufzurufen, wenn Sie getötet werden, mit dem gesamten Spielkontext in ein paar Variablen, die sowohl von BASIC als auch von C gelesen und geändert werden können. Im Beispiel sehen Sie, wie dies mit der Variablen "Punkte" gemacht wird.

WICHTIG: Auch wenn Ihr gesamtes Programm in C ist, müssen Sie 8BP mit einem **CALL &6b78** initialisieren, da dieser Aufruf nicht nur die RSX-Befehle installiert, sondern auch die Tabellen der internen Bibliothek initialisiert.

20.2.1 GOSUB und RETURN in C

GOSUBs sollten in c-Funktionen übersetzt werden, weil man von überall im Programm zu einer GOSUB-Routine gelangen kann und man in der Lage sein muss, zurückzukehren. In BASIC kehren Sie mit RETURN zu der Stelle zurück, von der aus Sie GOSUB aufgerufen haben. In C ist es naheliegend, die Routine in eine Funktion zu übersetzen, damit man zu der Stelle im Programm zurückkehren kann, von der aus man sie aufgerufen hat

Schauen wir uns ein Beispiel an:

BASIC	C
-------	---

<pre> 10 a=5 20 GOSUB 100 30 PRINT "PEPE" 40 Ende 100-REM-ROUTINE 110 PRINT STR\$(a) 120 RÜCKKEHR </pre>	<pre> #include <stdlib.h> #include <string.h> #include <stdio.h> #include <stdio.h> #include "8BP.h" #include "minibasic.h" #include "8BP.h" #include "minibasic.h" </pre>
	<pre> void mifucion(int id); int a; int main() { a=5; mifucion(a); _basic_print(_"PEPE"); O zurückgeben; } void mifucion(int a) { _basic_print(_basic_str(a)); _basic_print("\r"); } </pre>

20.2.2 Kommunikation zwischen BASIC und C mit BASIC-Variablen

Wenn Sie gerade erst anfangen, C mit 8BP zu benutzen, können Sie mit dem nächsten Schritt fortfahren. Dies ist ein "fortgeschritten"er Abschnitt, in dem Sie lernen, wie Sie zwischen BASIC und C mit BASIC-Variablen anstelle von PEEK/POKE kommunizieren können, aber er ist nicht unbedingt erforderlich.

Um BASIC mit C zu kommunizieren, können Sie anstelle einer Speicheradresse und PEEK/POKE auch eine Variable verwenden, die in BASIC existiert. Das ist etwas komplizierter, aber es ist machbar. Sehen wir uns an, wie man es mit einer einfachen Variable macht, und dann sehen wir, wie man es mit Array-Variablen macht.

Das erste, was Sie wissen müssen, ist, wie Sie herausfinden können, wo BASIC eine Variable speichert. Hierfür gibt es den "@"-Operator. Schauen wir uns ein einfaches Beispiel an:

10 DEFINT A-Z: 'wichtig, dass der Datentyp int ist 20 a=5 30 print @a:'gibt die Speicheradresse aus, an der a gespeichert ist 40 poke @a,7: 'das ist dasselbe wie a=7 50 PRINT a : ' dies druckt a 7

Es gibt einen kleinen "Fehler" in dem Programm. Es ist die Anweisung POKE @a,7, weil sie nur ein Byte speichert und die Variable "a" 2 Bytes hat, weil sie eine ganze Zahl ist. In diesem Fall funktioniert es, weil das höchstwertige Byte von "a" Null ist, aber wenn "a" einen Wert größer als 255 hätte, dann wäre das höchstwertige Byte nicht Null und der POKE würde nur das niedrigstwertige Byte ändern. Ein 8BP POKE würde immer funktionieren, weil es 16 Bit hat:

| **POKE,@a,7: 'setzt eine 7 in die Variable "a".**

```
a=1000
Ready
print a
1000
Ready
print @a
432
Ready
poke @a,7
Ready
print a
775
Ready
```

Mit diesem Wissen müssen wir C nur die Adresse übergeben, an der unsere BASIC-Variable gespeichert ist. Dafür haben wir die Anweisung |POKE von 8BP, die mit 16 Bits arbeitet (denken Sie daran, dass eine Speicheradresse 16 Bits belegt). Wir werden die Adresse der Variablen "a" an der Adresse 40000 übergeben, obwohl wir auch jede andere freie Adresse verwenden könnten.

|POKE, 40000, @a: **wir lassen @a an der Adresse 40000**

Eine alternative Methode in BASIC ist die Verwendung von zwei POKEs:

dir=@a:

**POKE 40001, INT (dir/256) POKE
40000, INT (dir MOD 256)**

Was wir an die Adresse 40000 geschrieben haben, ist die Speicheradresse, an der die Variable a gespeichert ist.

WICHTIG: BASIC kann eine Variable verschieben, wenn neue Variablen erstellt werden. Das heißt, wenn Sie einer C-Routine eine Speicheradresse über ein |POKE übergeben und anschließend neue BASIC-Variablen anlegen, kann sich die Speicheradresse der übergebenen Variablen geändert haben. **Sie wird sich nur dann garantiert nicht ändern, wenn keine neuen Variablen angelegt werden.** Das folgende Beispiel verdeutlicht dies sehr gut, es gibt 2 Speicherplatzänderungen

<pre>10 DIM a(100): i=0 20 PRINT @a(0) 30 b=2:'neue Variable verlagert a() 40 DRUCKEN @a(0) 50 c=2:'neue Variable verlagert a() 60 PRINT @a(0) 70 goto 20</pre>	
---	--

Die Lösung für dieses Problem besteht darin, alle Variablen am Anfang des Programms zu deklarieren.

<pre>10 dim a(100) 20 b=2 30 c=3 40 print @a(0) 70 goto 40</pre>	
--	--

Von C aus können wir nun auf zwei Arten auf die Variable "a" zugreifen, wobei die zweite Möglichkeit (über eine "gemappte" C-Variable) die interessanteste ist.

<pre>// globale Variablen int dir_a; int data; int main() { //Lassen Sie uns in dir_a die Adresse der Variablen a speichern _8BP_peek_2(40000, &dir_a); _8BP_peek_2(dir_a, &data); //Dies setzt den Wert der BASIC-Variablen // "a" in die C-Variable "data". Dies ist eine Möglichkeit, den Wert von "a" // zu lesen. _8BP_poke_2(dir_a,7); //Dies setzt eine 7 in die BASIC-Variable "a". 0 zurückgeben; }</pre>

Betrachten wir dasselbe Beispiel auf andere Weise, nämlich mit einer C-Variablen, die auf die Basisvariable "abgebildet" wird. Hierfür müssen wir die Notation mit "*" verwenden.

```
// globale Variablen
int dir;
int *a;

int main()
{
    //Lassen Sie uns in dir_a die Adresse der Variablen a speichern
    _8BP_peek_2(40000, &dir);
    a=dir; //um eine Kompilierwarnung zu vermeiden, verwenden Sie
    a=(int*)dir
    *a=5; //Dies setzt eine 5 in die BASIC-Variable "a".
    O zurückgeben;
}
```

Wir haben gesehen, wie es mit einfachen Variablen funktioniert. Jetzt werden wir sehen, wie BASIC-Arrays funktionieren, um von C aus auf sie zuzugreifen. Als erstes werden wir verstehen, wie BASIC Array-Daten im Speicher speichert.

```
1 DEFINT a-z
2 MODUS 2
10 a=5
20 PRINT "das dir von a ist
@a=";@a 25 PRINT "-----"
30 DIM b(5)
40 FOR i=0 TO 5
50 PRINT "das dir von b(";i;") ist ";@b(i)
60 NÄCHSTE
70 PRINT "-----"
80 DIM c(3,4)
90 FOR j=0 TO 4:FOR i=0 TO 3
100 PRINT "das dir von
c(";i;",";j;") ist ";@c(i,j)
110 NEXT:NEXT

120 |POKE,40000,@c(0,0)

Mit Zeile 120 haben wir an Adresse 40000
die Speicheradresse gespeichert, an der die
ersten Daten des zweidimensionalen
Arrays gespeichert sind.

Wir könnten das eindimensionale Array
speichern mit
|POKE,40000, @b(0)
```

The screenshot shows a memory dump from a BASIC interpreter. The memory dump is organized into four columns: address (la), memory location (dir), data type (de), and value (es). The first column shows addresses starting from 681 down to 765. The second column shows the memory location (dir) for each value. The third column indicates the data type (de), which is consistently 'b' for byte. The fourth column shows the values themselves. The values for the first row (addresses 681-684) correspond to the variable 'a' in the C code, and the values for the subsequent rows (addresses 698-708) correspond to the elements of the array 'b'. The last row (address 765) is labeled 'Ready'.

la	dir	de	es
681		b(0)	681
682		b(1)	698
683		b(2)	700
684		b(3)	702
685		b(4)	704
686		b(5)	706
687		c(0 , 0)	708
688		c(1 , 0)	727
689		c(2 , 0)	729
690		c(3 , 0)	731
691		c(0 , 1)	733
692		c(1 , 1)	735
693		c(2 , 1)	737
694		c(3 , 1)	739
695		c(0 , 2)	741
696		c(1 , 2)	743
697		c(2 , 2)	745
698		c(3 , 2)	747
699		c(0 , 3)	749
700		c(1 , 3)	751
701		c(2 , 3)	753
702		c(3 , 3)	755
703		c(0 , 4)	757
704		c(1 , 4)	759
705		c(2 , 4)	761
706		c(3 , 4)	763
707		Ready	765

Das obige Programm lehrt uns, wie BASIC Variablen speichert.

- Die Daten eines eindimensionalen Arrays werden alle in einer Reihe gespeichert. Jedes Datenfeld belegt 2 Bytes, da wir mit ganzen Zahlen arbeiten.

In BASIC geben Sie zum Beispiel ein:

```
DIM b(20)  
|POKE,40000,@b
```

CALL <Routine C>:'Adresse, an der main() zusammengestellt wurde

```
PRINT b(8)
```

und von C schreiben Sie:

```
int dir;
int *b; //Variable, mit der wir auf das Array zugreifen werden
BASIC int main(){
    _8BP_peek_2(40000, &dir);
    b= dir; //b ist ein Zeiger und *b[] sind Variablen
    *b[8]=5; //Zufügen einer 5 zur BASIC-Variablen
    b(8) return 0;
}
```

- Die Daten der zweidimensionalen Arrays werden fortlaufend gespeichert, wobei die Variationen der ersten Dimension aufeinanderfolgende Daten erzeugen. Im Beispiel mit **DIM(3,4)** folgen die Daten (2,3) auf die (1,3), aber die Daten (2,3) sind $4 \times 2 = 8$ Bytes weiter als die Daten (2,2), weil die erste Dimension des Arrays 4 ist. Ein **DIM (3,4) ist ein Array der Dimensionen (4, 5), weil die Null mitzählt**. Sie können dies überprüfen, indem Sie `@c(3,2)` und `@c(2,2)` ausdrucken, Sie werden sehen, dass es einen Unterschied von 8 gibt. Um von C aus auf die Daten zuzugreifen, kann ein einfacher Zeiger verwendet werden, wobei die erste Dimension beim Zugriff berücksichtigt wird. Im folgenden Beispiel habe ich ein Array `c(12,16)` erstellt und für den Zugriff von C auf die Position (2,7) verwende ich $2 + 7 \cdot 13$, da die erste Dimension $12 + 1 = 13$ ist

In BASIC schreiben Sie:

```
DIM c(12,16)
|POKE, 40000, @c
CALL <Routine C>:' Adresse Assembly-Routine
PRINT c(2,7)
```

und von C schreiben Sie:

```
int dir;
int *b; //Variable, mit der wir auf das Array zugreifen werden
BASIC int main(){
    _8BP_peek_2(40000, &dir); //Lesen von dir 40000 und
    Schreiben in dir c= dir; //c ist ein Zeiger und *c[] sind Variablen
    c[2+7*13]=5; //Hinzufügen einer 5 zur BASIC-Variablen c(2,7)
    return 0;
}
```

Wenn Sie ein C-Programmierer sind, wissen Sie, dass es in C Doppelzeiger gibt, die mit einem doppelten Sternchen ausgedrückt werden (z. B. `**c`). Auf den ersten Blick scheinen sie geeignet zu sein, weil man mit `c[x][y]` auf Daten verweisen kann. Sie dienen jedoch nur dazu, reservierten Speicher dynamisch mit "malloc"- und "calloc"-Befehlen zu speichern, und erfordern Speicher sowohl für die Daten als auch für die Zeiger in jeder Zeile. Das bedeutet, dass Sie sie zwar verwenden könnten, aber Sie müssten Speicher für die Zeiger reservieren. Ich empfehle sie nicht.

20.2.3 BASIC- und C-Textstrings

Sie sollten wissen, dass ein String in C ein einfacher Zeiger auf ein Zeichen ist, während eine String-Variable in BASIC (z. B. `mivar$="hallo"`) aus einem 3-Byte-Deskriptor besteht, der die Speicheradresse, an der sich der String befindet, und die Länge des Strings enthält. Das heißt, **eine C-Zeichenkette und eine BASIC-Zeichenkette sind unterschiedliche Dinge**.

Wenn Sie also eine Variable ausdrucken wollen, die eine Textzeichenfolge enthält, können Sie diese nicht von BASIC an C übergeben, da es sich nicht um dasselbe handelt. Aber Sie können Textstrings ohne Probleme ausgeben, wenn Sie sie nicht von BASIC aus übergeben, sondern z. B. in C definieren:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdio.h>
#include "8BP.h"
#include "minibasic.h"
#include "8BP.h"
#include "minibasic.h"

//----globalvariablen
char* cad; //könnnte hier initialisiert werden

//----Funktionen-----
int main(){
cad="Sie sind bei Ihrer Mission gescheitert"; //Wir initialisieren es mit
einer Phrase
_basic_print(cad); //BASIC-Stil drucken
_8BP_printat4(0,0,60,cad); //printat im 8BP-Stil printat return
0;
}
```

20.3 Dritter Schritt: Kompilieren mit "compila.bat".

Wir sind dabei, den großen Schritt zu machen: Kompilieren, um eine AMSTRAD-Binärdatei zu erzeugen. Für diesen Schritt gibt es eine Hilfs-.bat, genannt "compila.bat". **Bevor Sie diesen Schritt ausführen, müssen Sie den SDCC-Compiler auf Ihrem Computer installiert haben, andernfalls wird er fehlgeschlagen.** Sie können ihn von <http://sdcc.sourceforge.net/> herunterladen.

Hinweis: Unter Windows 10 ist die Ausführung von SDCC bei mir fehlgeschlagen, weil es unter "Programme" installiert ist und Whitespace nicht dazu zu passen scheint. Wenn das bei Ihnen der Fall ist, installieren Sie es einfach in ein Verzeichnis, dessen Name keine Leerzeichen enthält.

Sobald SDCC installiert ist, öffnen Sie ein Befehlsfenster (das Skript ses.bat übernimmt diese Aufgabe) und geben compila.bat ein. Der Bildschirm ändert seine Farbe: grün, wenn alles gut gelaufen ist, und rot, wenn es Probleme gibt. Das Skript "compila.bat" führt die folgenden Schritte aus:

1. Löscht die Ausgabedateien des Skripts selbst (u.a. ciclo.dsk und ciclo.bin).
2. Ruft SDCC auf, um Ihr Programm zu kompilieren
3. SDCC-Ausgabe mit dem Tool hex2bin ins Binärformat übersetzen
4. Fügen Sie die erzeugte Binärdatei mit dem Tool manageDsk in eine Diskette ein, die Sie cycle.dsk nennen.

Nach der Ausführung von "compila.bat" erhalten Sie in den folgenden Fällen einen roten Bildschirm:

- Syntaxfehler in Ihrem C-Programm

- Die Datei ciclo.dsk, die compila.bat erzeugen wird, wird von winape geöffnet. In diesem Fall müssen Sie eine andere Festplatte mit winape verbinden, damit compila.bat sie neu erstellen kann.
- Kompilierungsfehler, z. B. eine Bibliothek, die Sie verwenden wollten, aber nicht eingebunden haben, usw.

Im Falle von C-Syntaxfehlern oder anderen C-Fehlern oder wenn die Datei ciclo.dsk, die compila.bat erzeugt, von winape geöffnet wird, erhalten Sie einen roten Fehlerbildschirm wie diesen:

```
8888  BBBB  PPPPP  
88 88  BB BB  PP PP  
88 88  BB BB  PP PP  
8888  BBBB  PPPPP  
88 88  BB BB  PP  
88 88  BB BB  PP  
8888  BBBB  PPPP  
8 bits de poder . Un tributo al AMSTRAD CPC  
Jose Javier Garcia Aranda 2016-2020  
  
*****  
*      compilacion con SDCC      *  
*****  
borramos los ficheros de compilacion anterior  
*****  
  
main.c : compilamos y linkamos, generando un main.ihx  
*****  
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --  
BP_wrapper -Imini_BASIC ciclo.c  
sdcc: Calling preprocessor...  
sdcc: sdcpp -nostdinc -Wall -std=c11 -I"8BP_wrapper" -I"mini_BASIC"  
SDCC_CHAR_UNSIGNED -D__SDCC_INT_LONG_REENT -D__SDCC_FLOAT_REENT -D__  
SDCC_VERSION_MINOR=0 -D__SDCC_VERSION_PATCH=0 -D__SDCC_REVISION=1  
=1 -D__STDC_NO_THREADS_=1 -D__STDC_NO_ATOMICS_=1 -D__STDC_NO_VLA_  
DC_UTF_16_=1 -D__STDC_UTF_32_=1 -isystem "C:\proyectos\proyectos00"  
" -isystem "C:\proyectos\proyectos09\_personal\8BP\SDCC\bin..\incl  
sdcc: Generating code...  
ciclo.c:36: syntax error: token -> 'puntos' ; column 8  
ciclo.c:37: error 1: Syntax error, declaration ignored at 'fps'  
ciclo.c:38: error 1: Syntax error, declaration ignored at 't1'  
ciclo.c:41: syntax error: token -> '0' ; column 21  
. .  
"-----+  
" | HAY ERRORES DE COMPILACION! | "  
"-----+"  
C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>
```

Abb. 115 Compila.bat meldet, dass Sie C-Fehler haben

Sollte alles gut gehen, wäre dies der Weg nach draußen.

```

transformamos el .ihx en un .bin
*****
hex2bin -output\ciclo.ihx
hex2bin v1.0.1, Copyright (C) 1999 Jacques Pelletier
Lowest address = 00004E20
Highest address = 00005A41

metemos el .bin en un disco de amstrad cpc
*****
managedsk -C -S"output\ciclo.dsk"
managedsk -L"output\ciclo.dsk" -I"output\ciclo.bin"/CICLO.BIN/BIN/20

*****
**          FIN DEL PROCESO
**  ASEGUrate DE QUE NO EXCEDES LA DIRECCION 24000
** es la (highest address) de la transformacion ihx en bin
**
** se ha generado ciclo.dsk y dentro esta ciclo.bin
**
** Pasos para cargarlo en el amstrad
** 1) carga o ensambla 8BP, con tus graficos, musica etc
** 2) carga tu juego BASIC
** 3) ejecuta LOAD "ciclo.bin", 20000
** para invocar a tu programa o rutina simplemente:
** call <direccion de main en fichero ciclo.map>
**
** Para mover ciclo.bin de ciclo.dsk a otro disco debes
** conocer su longitud:
** longitud=Highest address - Lowest address
** lo cargas desde ciclo.dsk
** LOAD "ciclo.bin", 20000
** Y salvas en el disco donde esta tu juego
** SAVE "ciclo.bin",b,20000,longitud
*****
C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>
```

Abb. 116 Compila.bat erzeugt einen grünen Bildschirm: Alles ist gut gegangen.

Wenn Sie den grünen Bildschirm erhalten, bedeutet dies, dass alle Schritte von compila.bat gut verlaufen sind (borSDCC, Sie werden wertvolle Informationen auf dem Bildschirm haben und im Unterverzeichnis output finden Sie Dateien, die Sie benötigen:

- Zyklus.dsk
- Zyklus.karte

20.4 Schritt vier: Überprüfung der Speichergrenzen

Das Skript compila.bat zeigt Ihnen auf seinem grünen Bildschirm zwei sehr wertvolle Informationen an: "**niedrigste Adresse**" und "**höchste Adresse**". Dies sind die Speicheradressen, an denen die erzeugte Binärdatei beginnt und endet. Das Skript "compila.bat" verwendet die Adresse 20000 als Kompilieradresse im SDCC-Aufruf und wenn die resultierende Binärdatei sehr groß ist, könnte sie die Adresse 24000 überschreiten und damit die 8BP-Bibliothek beschädigen. Sie müssen sicherstellen, dass die "**höchste Adresse**" kleiner als 24000 ist. Wenn sie nicht niedriger ist, **müssen Sie den SDCC-Aufruf im Skript "compila.bat" ändern**.

zu kompilieren, indem Sie eine Adresse kleiner als 20000 zuweisen. Auf diese Weise werden sich Ihre neue Binärdatei und die 8BP-Bibliothek nicht überschneiden. Sie müssen zwei Zeilen des Skripts compila.bat ändern. Die erste ist diejenige, die SDCC aufruft. Es ist eine sehr lange Zeile. Sie müssen den Parameter 20000 in die neue Adresse ändern

```
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --fomit-frame-pointer --opt-code-size -I8BP_wrapper -lmini_BASIC -o output/cycle.c
```

Die zweite Zeile, die Sie ändern müssen, ist diejenige, die managedsk aufruft, damit sie mit der neuen Speicheradresse übereinstimmt. Dies ist die Zeile und wie Sie sehen können, erscheint auch die Adresse 20000.

```
managedsk -L "Ausgabe "cycle.dsk" -I "Ausgabe  
"cycle.bin"/CYCLE.BIN/BIN/2000000 -I "ausgabe\cycle.bin"/CICLO.BIN/BIN/20000 -  
S "ausgabe\ciclo.dsk" -I "ausgabe\ciclo.bin"/CICLO.BIN/BIN/20000
```

Wenn Ihre Binärdatei bei 20000 beginnt, muss Ihr BASIC-Programm natürlich einen Speicherplatz von 19999 einbeziehen. Dadurch werden 4 KB von Ihrem freien Speicherplatz abgezogen, aber Sie speichern auch die BASIC-Zeilen, die dem Spielzyklus entsprechen, so dass Sie nichts verlieren.

Wenn die **höchste Adresse** kleiner als 24000 ist, **sollten Sie sie so weit wie möglich anpassen, d. h. so nah wie möglich an 24000, um keinen Speicher zu verschwenden.** Dies kann z.B. bedeuten, dass Sie beim Aufruf von SDCC die Adresse 21000 verwenden. Tun Sie dies, um so viel Speicher wie möglich für BASIC zu haben. Sie sollten in Ihrem BASIC-Programm einen MEMORY-Wert angeben, der mit dieser Adresse übereinstimmt. Wenn das Binärprogramm z.B. bei 21000 beginnt, müssen Sie ein MEMORY von 20999 einstellen.

Letztendlich müssen Sie möglicherweise zwei Zeilen im Skript "compila.bat" ändern, um die Startadresse für die Kompilierung anzupassen, die ich ursprünglich auf 20000 gesetzt habe.

20.5 Schritt 5: Suchen Sie die Adresse der Funktion, die von BASIC aus aufgerufen werden soll.

Nach dem Kompilieren mit "compila.bat" haben Sie im Unterverzeichnis "output" eine Datei namens ciclo.map erhalten. In dieser Datei müssen Sie die Speicheradresse der Funktion oder der Funktionen finden, die Sie von BASIC aus aufrufen wollen. In diesem Beispiel werden wir nur die Funktion main() aufrufen, die sich an der Adresse &56b0 befindet, wie Sie in diesem Ausschnitt der Datei ciclo.map sehen können

0000569E	__basic_paper
0000566B	__basic_plot
00005682	__basic_move
00005699	__basic_draw
000056B0	main
00005920	_abs
0000592C	_strlen
0000593B	__modchar
00005948	__modint
00005954	__moduchar

Abb. 117 die Adresse jeder Funktion steht in ciclo.map

Das bedeutet, dass wir zum Aufrufen der main()-Funktion in BASIC einfach Folgendes tun:

AUFRUF &56B0

Seien Sie vorsichtig, denn wenn Sie in der Kompilierungsphase Änderungen vornehmen (Änderung des .c-Zyklus oder Änderung der Speicheradressen des compila.bat-Skripts), kann sich die Adresse jeder Funktion beim erneuten Kompilieren ändern.

20.6 Schritt 6: Binden Sie die neue Binärdatei in Ihr .dsk-Spiel ein

Sie haben bereits eine Datei cycle.dsk erstellt, die die Datei cycle.bin enthält, die Sie laden müssen, um die Hauptfunktion (und/oder andere Funktionen, die Sie wünschen) aufzurufen. Um sowohl diese Datei als auch Ihr Spiel auf derselben Festplatte zu haben, müssen Sie cycle.dsk aus winape auswählen und einfach die Datei cycle.bin laden.

ZYKLUS.BIN" LADEN,20000

Wählen Sie dann im Winape-Menü Ihren Datenträger aus (auf dem sich Ihr Spiel befindet) und speichern Sie diese Binärdatei

SAVE "CICLO.BIN", b, 20000, <Länge>.

wobei Länge = höchste Adresse - niedrigste Adresse +1

In Ihrer loader.bas-Datei müssen Sie nun diese neue zusätzliche Binärdatei laden und sie in Ihrem BASIC-Listing mit CALL <Adresse> aufrufen.

```
10 SPEICHER 24999
15 LOAD "!paint.scr",&c000: 'nur wenn Ihr Spiel einen Ladebildschirm hat
20 LOAD "yourgame.bin"
25 LOAD "cycle.bin", 20000
50 RUN " !yourgame.bas"
```

Das war's schon. Sie können jetzt mit 8BP in C programmieren!

20.7 8BP Funktionsreferenz in C

RSX	C-Prototyp
3D, 0 3D, <Kennzeichen>, #, Offsetdruck	void _8BP_3D_1(int flag); void _8BP_3D_3(int flag, int sp_fin, int offsety);
ANIMA, #	void _8BP_anima_1(int sp);
ANIMALL	void _8BP_animall();
AUTO, #	void _8BP_auto_1(int sp);
AUTOALL, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>.	void _8BP_autoall(); void _8BP_autoall_1(int flag);
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	void _8BP_colay_3(int threshold, int* collision, int sp); void _8BP_colay_2(int* collision, int sp); void _8BP_colay_1(int sp); void _8BP_colay();
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, Ende COLSP, 33, @collided% COLSP, # COLSP, 34, dy, dx	/* Operation 32, ini,fin oder Operation 34,dy,dx*/ void _8BP_colsp_3(int operation, int a, int b); /*Vorgang 33 oder sp*/ void _8BP_colsp_2(int sp, int* collision); void _8BP_colsp_1(int sp);
COLSPALL,@who%,@who%,@mitw hom% COLSPALL, Kollider COLSPALL	void _8BP_colspall_2(int* collider, int* collided); void _8BP_colspall_1(int collider_ini); void _8BP_colspall();
LAYOUT, y, x, @String\$, @String\$, @String\$, @String\$, @String\$, @String\$.	void _8BP_layout_3(int y, int x, char* cad);
LOCATESP, #, y, x	void _8BP_locatesp_3(char sp, int y, int x);
MAP2SP, y, x MAP2SP, Status	void _8BP_map2sp_2(int y, int x); void _8BP_map2sp_1(unsigned char status);
MOVER, #, dy, dx	void _8BP_mover_3(int sp, int dy,int dx); void _8BP_mover_1(int sp);
MOVERALL, dy,dx	void _8BP_moverall_2(int dy, int dx); void _8BP_moverall();
MUSIC, C, Flagge, Lied, Geschwindigkeit MUSIC, Flagge, Lied, Geschwindigkeit MUSIK	void _8BP_music_4(int flag_c, int flag_repetition,int song, int speed); void _8BP_music();
PEEK, dir, @Variable%	void _8BP.Peek_2(int address, int* data);
POKE, dir, wert	void _8BP.poke_2(int address, int data);
PRINTAT, flag, y, x, @string	void _8BP.printat_4(int flag,int y,int x,char* cad);
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, Bits	void _8BP.printsp_1(int sp) ; void _8BP.printsp_2(int sp, int bits_background) ; void _8BP.printsp_3(int sp,int y,int x) ;
PRINTSPALL, ini, fin, anima, sync PRINTSPALL, Auftragsmodus PRINTSPALL	void _8BP.printspall_4(int ini, int fin, int flag_anima, int flag_sync); void _8BP.printspall_1(int order_type); void _8BP.printspall();
RINK,tini,Farbe1,Farbe2,...,FarbeN RINK, Sprung	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step);
ROUTESP, #, Schritte	void _8BP_routesp_2(int sp, int steps); void _8BP_routesp_1(int sp);
ROUTEALL	void _8BP_routeall();
SETLIMITS, xmin, xmax, ymin, ymax	Void _8BP_setlimits_4 (int xmin, int xmax, int ymin, int ymax)

SETUPSP, #, param_number, value	void _8BP_setupsp_3(int sp, int param, int value);
SETUPSP, #, 5, Vy, Vx	void _8BP_setupsp_4(int sp, int param, int value1,int value2);
STARS, initstar, num, color, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int color, int dy, int dx); void _8BP_stars();
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	void _8BP_umap_6(int map_ini, int map_fin, int y_ini, int y_fin, int x_ini, int x_fin);

Hier sind einige Anwendungsbeispiele, die das in der BASIC-C-Übersetzung verwendete Beispiel ergänzen.

RSX	3D, <Kennzeichen>, #, Versatz 10 3D,1,10,200
C	void _8BP_3D_3(int flag, int sp_fin, int offsety); _8BP_3D_3(1, 10, 200);

RSX	COLSPALL,@who%,@who%,@mitwhom% 10 collider%=0: kollidiert%=0 20 COLSPALL, @collider%, @colliderd%, @colliderd%
C	void _8BP_colspall_2(int* collider, int* collided); Int cor=0; Inr cod=0; _8BP_colspall_2 (&cor, &cod);

RSX	RINK,tini,Farbe1,Farbe2,...,FarbeN RINK, Sprung 10 TRINKEN,1,2,2,2,3,3 20 RINK,1
C	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step); Int inks[5]={1,2,2,3,3}; _8BP_rink_N(5,inks); _8BP_rink_1(1);

RSX	LAYOUT, y, x, @String\$, @String\$, @String\$, @String\$, @String\$, @String\$. 10 cad\$="Y Y" 20 LAYOUT,10,1,@cad\$
C	void _8BP_layout_3(int y, int x, char* cad); _8BP_layout_3(10,1," YYYYYYY YYYYYY YYYYY YYYYY"; Oder: char* cad=" AA YYYYYY YYYY YYYYYY YYYYY"; _8BP_layout_3(10,1,cad)

RSX	PRINTAT, flag, y, x, @string 10 cad\$=str\$(125) 20 PRINTAT,0,100,40,@cad\$ 20 PRINTAT,0,100,40,@cad\$ 20
C	void _8BP_printat_4(int flg,int y,int x,char* cad) char* cad=_basic_str(125); _8BP_printat_4(0,100,40, cad);

20.8 BASIC-Funktionsreferenz in C ("minibasic")

Dies ist ein kleiner Satz von BASIC-ähnlichen Befehlen, die 8BP Ihnen durch die minibasic.h-Bibliothek zur Verfügung stellt, um Ihr BASIC-Listing einfach nach C zu übersetzen. Wenn Sie nur den Spielzyklus übersetzen wollen, wird dieser Satz von Befehlen ausreichen.

BASIC	C-Prototyp
BORDER	<code>void _basic_border(char Farbe); //Beispiel _basic_border(7)</code>
ANRUFEN	<code>void _basic_call(unsigned int address); // Beispiel _basic_call(0xbd19)</code>
ZEICHNEN	<code>void _basic_draw(int x, int y);</code>
TINTE	<code>void _basic_ink(char ink1,char ink2);</code>
INKEY	<code>char _basic_inkey(char key); //dauert etwa 0,3 ms. langsam aber einfach</code>
LOCATE	<code>void _basic_locate(unsigned int x, unsigned int y); // Beispiel: _basic_locate(2,25);_basic_print("TEST");</code>
MOVE	<code>void _basic_move(int x, int y);</code>
PAPIER	<code>void _basic_paper(char ink);</code>
PEEK	<code>char _basic.Peek(unsigned int address);</code>
GRAFIKEN	<code>void _basic_pen_graph(char ink);</code>
PEN	<code>void _basic_pen_txt(char ink);</code>
POKE	<code>void _basic_poke(unsigned int address, unsigned char Daten);</code>
PLOT	<code>void _basic_plot(int x, int y);</code>
DRUCKEN	<code>void _basic_print(char *cad); //Beispiel: _basic_print("Hallo")</code>
RND	<code>unsigned int _basic_rnd(int max); //Beispiel: num=_basic_rnd(50)</code>
TON	<code>void _basic_sound(unsigned char nChannelStatus, int nTonePeriod, int nDuration, unsigned char nVolume, char nVolumeEnvelope, char nToneEnvelope, unsigned char nNoisePeriod);</code>
STR\$	<code>char* _basic_str(int num); //ähnlich wie STR\$ //Beispiel: _basic_print(_basic_str(num))</code>
ZEIT	<code>unsigned int _basic_time(); //Rückgabe eines unsigned int,(0..65535). Als Ganzzahl, wenn // bei Erreichen von 32768 gehen Sie auf -32768</code>

21 8BP Library Referenzhandbuch

21.1 Funktionen der Bibliothek

21.1.1 |3D

Dieser Befehl aktiviert die 3D-Projektion in den Befehlen PRINTSP und PRINTSPALL. Zum Projizieren haben wir den Befehl |3D

Verwenden Sie

So aktivieren Sie die 3D-Projektion:

|3D, 1, <Sprite_fin>, <offsety>, <offsety>, <offsety>, <offsety>, <offsety>.

Zum Deaktivieren:

|3D, 0

Die betroffenen Sprites sind von Sprite 0 bis <Sprite_fin>. Dieser Befehl aktiviert die 3D-Projektion im Befehl **|PRINTSP** und in **|PRINTSPALL**. Das bedeutet, dass vor der Ausgabe auf dem Bildschirm die "projizierten" Koordinaten berechnet und dann auf dem Bildschirm ausgegeben werden. Die Koordinaten der Sprites sind davon nicht betroffen, d.h. die 2D-Koordinaten in der Sprite-Tabelle bleiben gleich.

Dieser Befehl hat **keinen Einfluss auf Kollisionsmechanismen**, d.h. wenn wir COLSPALL verwenden und eine Kollision zwischen projizierten Sprites erkennen, findet die Kollision in der 2D-Ebene statt.

Der letzte Parameter <offsety> dient dazu, höher oder niedriger zu projizieren, so dass wir die Spielmarkierungen dort platzieren können, wo wir wollen. Beim Projizieren des Bildschirms, der 200 Pixel hoch ist, wird er 100 Pixel hoch, also können wir wählen, wie hoch wir die Projektion platzieren. Wenn ein Sprite nicht von der Projektion betroffen ist, weil es höher als <Sprite_fin> ist, dann ist es auch nicht von <offsety> betroffen.

Die folgende Abbildung zeigt, welche Weltkartenkoordinaten an bestimmten repräsentativen Punkten auf den Bildschirm projiziert werden, wenn **|MAP2SP** mit (yo=0, xo=0) aufgerufen wird.

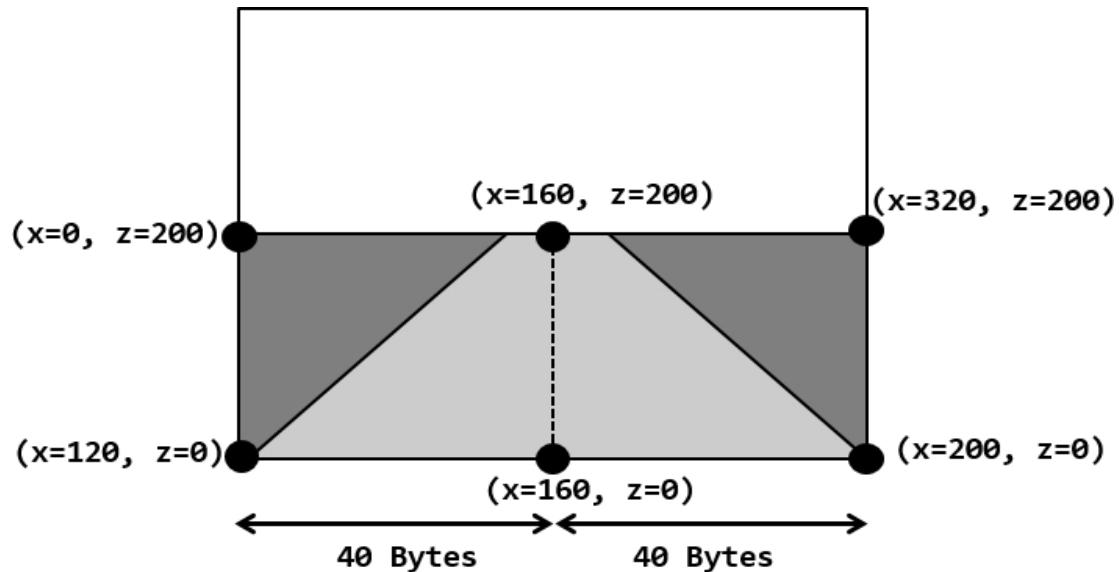


Abb. 118 Projizierte Weltkoordinaten

Wenn wir anstelle von $(xo=0, yo=0)$ eine andere Koordinate für MAP2SP verwenden, werden die 2D-Weltkoordinaten, die den im Bild referenzierten Punkten entsprechen, um (x, z) verschoben, wie durch xo und yo angegeben.

21.1.2 |ANIMA

Dieser Befehl ändert das Animationsbild eines Sprites unter Berücksichtigung der ihm zugewiesenen Animationssequenz.

Verwendung:

|ANIMA, <Spritnummer>, <Spritnummer>, <Spritnummer>, <Spritnummer>.

Beispiel:

|ANIMA,3

Der Befehl fragt die Animationssequenz des Sprites ab, und wenn diese nicht Null ist, geht er zur Animationssequenztabelle (die erste gültige Sequenz ist 1 und die letzte ist 31). Er wählt das Bild aus, dessen Position neben dem aktuellen Frame liegt, und aktualisiert das Frame-Feld der Sprite-Attribut-Tabelle.

Wenn das nächste Bild in der Sequenz Null ist, wird es umgeschaltet, d. h. das erste Bild der Sequenz wird ausgewählt.

Zusätzlich zur Änderung des Bildfeldes wird auch das Bildfeld geändert und die Speicheradresse, an der das neue Bild gespeichert wird, zugewiesen.

|ANIMA drückt das Sprite nicht, sondern lässt es druckbereit, so dass das nächste Bild seiner Sequenz gedruckt wird.

|ANIMA prüft nicht, ob das Animationsflag im Statusbyte des Sprites aktiv ist. In der Tat wird unsere Figur normalerweise nur animiert werden wollen, wenn sie sich bewegt und nicht immer, wenn sie gedruckt wird.

Handelt es sich bei der Animationssequenz um eine "Todessequenz" (mit einer "1" im letzten Bild), dann wird das Sprite bei Erreichen des Bildes, dessen Bildspeicheradresse 1 ist, inaktiv.

Mit der 8BP-Bibliothek können Sie "Todessequenzen" erstellen, d. h. Sequenzen, bei denen das Sprite nach Abschluss in einen inaktiven Zustand übergeht. Dies wird durch eine einfache "1" als Wert der Speicheradresse des letzten Frames angezeigt. Diese Sequenzen sind sehr nützlich, um Explosionen von Feinden zu definieren, die mit |ANIMA oder

|ANIMALL. Nachdem du sie mit deinem Schuss getroffen hast, kannst du ihnen eine Todesanimationssequenz zuordnen und in den folgenden Spielzyklen durchlaufen sie die verschiedenen Animationsphasen der Explosion, und wenn sie die letzte erreicht haben, gehen sie in den inaktiven Zustand über und werden nicht mehr gedruckt. Dieser inaktive Zustand wird automatisch hergestellt. Du musst also nur die Kollision deines Schusses mit den Feinden überprüfen und wenn er mit einem von ihnen kollidiert, änderst du den Zustand mit |SETUPSP, so dass er nicht mehr kollidieren kann, und du weist ihm die Animationssequenz des Todes zu, ebenfalls mit |SETUPSP.

Wenn Sie eine Todessequenz verwenden, achten Sie darauf, dass das letzte Bild vor der "1" ein völlig leeres Bild ist, damit keine Spuren der Explosion zu sehen sind.

Beispiel für eine Todesfolge

```
dw EXPLOSION_1,EXPLOSION_2,EXPLOSION_3,1,0,0,0,0,0,0
```

21.1.3 |ANIMALL

Dieser Befehl animiert alle Sprites, die das Animationsflag im Statusbyte gesetzt haben.
Dieser Befehl hat keine Parameter

Verwenden Sie

|ANIMALL

WICHTIG: Seit Version v37 der Bibliothek ist dieser Befehl nur noch über CALL (siehe Entsprechungstabelle im Anhang) und nicht mehr über den Befehl RSX zugänglich. Durch das Entfernen dieses Befehls aus der Liste konnten einige Bytes Speicherplatz eingespart werden, und er kann weiterhin entweder über einen Parameter in PRINTSPALL oder über einen CALL-Aufruf verwendet werden.

Es ist empfehlenswert, wenn Sie viele Sprites animieren wollen, da es viel schneller ist, als den Befehl |ANIMA mehrmals aufzurufen.

Da Sie normalerweise |ANIMALL in jedem Spielzyklus aufrufen wollen, bevor Sie die Sprites drucken, gibt es einen effizienteren Weg, ihn aufzurufen, nämlich den entsprechenden Parameter des Befehls |PRINTSPALL auf "1" zu setzen, d.h.

|PRINTSPALL,1,0

Diese Funktion ruft intern |ANIMALL auf, bevor sie die Sprites ausdrückt, um die 1,17 ms im Vergleich zu der Zeit, die für den separaten Aufruf von |ANIMALL und |PRINTSPALL

21.1.4 |AUTO

Dieser Befehl bewegt ein Sprite (ändert seine Koordinaten) entsprechend seiner Geschwindigkeitsattribute Vy, Vx. Diese Attribute sind die, die das Sprite in der Sprite-Tabelle hat.

Verwendung:

|AUTO, <Spritnummer>, <Spritnummer>, <Spritnummer>, <Spritnummer>.

Beispiel:

|AUTO, 5

Mit diesem Befehl werden die Koordinaten in der Sprite-Tabelle aktualisiert, indem die Geschwindigkeit zur aktuellen Koordinate hinzugefügt wird.

Die neuen Koordinaten sind

neu X = aktuelle X-Koordinate + Vx

neu Y = aktuelle Y-Koordinate + Vy

Es ist nicht erforderlich, dass das Sprite im Statusfeld das Kennzeichen für automatische Bewegung aktiviert hat.

21.1.5 |AUTOALL

Dieser Befehl bewegt alle Sprites, die das Flag für automatische Bewegung aktiviert haben, entsprechend ihrer Geschwindigkeitsattribute Vy, Vx.

Verwendung:

|AUTOALL, <Routing-Flag>, <Routing-Flag>.

Beispiel

|AUTOALL,1 ruft **|ROUTEALL** vor dem Bewegen von Sprites auf

|AUTOALL,0 ruft nicht **|ROUTEALL** auf

|AUTOALL der zuletzt verwendete Wert wird als Parameter verwendet (hat Speicher)

Das Routing-Flag ist optional. Da der Befehl **|ROUTEALL** die Koordinaten der Sprites nicht verändert, müssen sie mit **|AUTOALL** bewegt und mit **|PRINTSPALL** gedruckt (und animiert) werden. Aus diesem Grund gibt es einen optionalen Parameter in **|AUTOALL**, so dass **|AUTOALL,1** intern **|ROUTEALL** aufruft, bevor das Sprite bewegt wird, was einen BASIC-Aufruf erspart, der immer eine kostbare Millisekunde dauert.

21.1.6 |COLAY

Erkennt die Kollision eines Sprites mit der Screen Map (dem Layout). Es berücksichtigt die Größe des Sprites, um zu sehen, ob es kollidiert, und geht davon aus, dass die Layout-Elemente alle 8x8 Pixel des Modus 0 sind (d.h. 4 Bytes x 8 Zeilen). Er kann mit 3, 2, 1 oder ohne Parameter aufgerufen werden. Bei einem Aufruf ohne Parameter werden die Werte des letzten Aufrufs mit Parametern verwendet, was wesentlich schneller ist.

Verwendung:

```
|COLAY, <ASCII-Schwelle>, @Kollision%, <Spruchnummer>,  
<Spruchnummer>.  
|COLAY, @Kollision%, <Spruchnummer>  
|COLAY, <num_sprite>, <num_sprite>.  
|COLAY
```

Der optionale Parameter <ASCII-Schwelle> muss nur beim ersten Aufruf verwendet werden, um die Kollisionsschwelle im Befehl **|COLAY** festzulegen. Dieser Schwellenwert stellt den größten ASCII-Code des Layoutelements dar, der als "keine Kollision" betrachtet wird. Der Standardwert ist 32 (das Leerzeichen). Um den gewünschten Schwellenwert einzustellen, lesen Sie die ASCII-Tabelle des Befehls **|LAYOUT**.

Beispiel:

```
|COLAY, 65, @col%,31 : rem sprite ist 31, Schwelldwert ist 65
```

Die Variable, die Sie für die Kollision verwenden, kann heißen, wie Sie wollen. Ich habe "col" angegeben.

Diese Routine ändert die Kollisionsvariable (die eine ganze Zahl sein muss, daher das "%"), indem sie auf 1 gesetzt wird, wenn es eine Kollision des angegebenen Sprites mit dem Layout gibt. Wenn es keine Kollision gibt, ist das Ergebnis 0.

```
10 xvorher=x  
20 x=x+1  
30 |LOCATESP,0,y,x: ' Sprite an neuer Stelle positionieren  
40 |COLAY,@collision%,0: 'Kollisionsprüfung
```

Jetzt überprüfen wir die Kollision, und wenn es eine Kollision gibt, lassen wir sie an ihrer vorherigen Position.

```
50 if collision%=1 then x=xprevious: LOCATESP,0,y,x
```

Du kannst auch den Befehl **|MOVER** verwenden, um das Sprite zu positionieren und die Prüfung durchzuführen.

```
10 |COLAY,65,@col,31: 'Konfiguration. Wir machen das nur einmal  
20 |MOVER,31,1,1: ' wir verschieben es nach rechts und nach unten  
30 |COLAY:' rufen wir es ohne Parameter auf (schneller)  
30 if col THEN MOVER,31,-1,-1 : rem ist kollidiert, also lasse ich es, wo es war.
```

21.1.7 |COLSP

Dieser Befehl ermöglicht es, die Kollision eines Sprites mit anderen Sprites zu erkennen, die das Kollisionsflag aktiviert haben.

Verwendung:

Zum Konfigurieren:

| **COLSP, 32, <sprite initial>, <sprite final>.**
| **COLSP, 33, @collision%**
| **COLSP, 34, dy, dx**

Für die Kollisionserkennung:

|COLSP,<Seitennummer>, @colsp%.

Beispiel:

col%=0

|COLSP,0,@col%

Die Funktion gibt in der Variablen, die wir als Parameter übergeben, die Nummer des Sprites zurück, mit dem sie kollidiert, oder wenn es keine Kollision gibt, gibt sie eine 32 zurück, weil das Sprite 32 nicht existiert (es gibt nur 0 bis 31).

WICHTIG: Die Kollisionsvariable im COLSP-Befehl ist nicht diejenige, die im COLSPALL-Befehl verwendet wird. Es handelt sich um unterschiedliche Variablen (es sei denn, Sie übergeben beiden Befehlen dieselbe Variable, um auf sie einzuwirken).

Wie beim Sprite-Druck mit PRINTSPALL prüft die COLSP-Funktion Sprites, die bei 31 beginnen und bei Null enden. Wenn sie ein aktives Sprite-Kollisionsflag haben (Bit 2 des Statusbytes), wird die Kollision geprüft. Wenn zwei Sprites gleichzeitig mit unserem Sprite kollidieren, wird die größere Sprite-Nummer zurückgegeben, da sie zuerst geprüft wird.

Aufforderungen zur Konfiguration des Befehls:

Es gibt eine Möglichkeit, COLSP so zu konfigurieren, dass es weniger Arbeit macht, indem es weniger Sprites auf Kollisionen überprüft, um Ausführungszeit zu sparen. Die Konfiguration wird durch die Verwendung von Sprite 32 (das nicht existiert) angezeigt.

|COLSP, 32, <Anfangsprüfung>, <Endprüfung>.

Wenn zum Beispiel die Feinde unseres Charakters die Sprites 25 bis 30 sind und wir sie als Kollider (Nicht-Kollider) konfigurieren, können wir (nur einmal) den Befehl wie folgt aufrufen:

|COLSP, 32, 25, 30

Das bedeutet, dass jeder nachfolgende Aufruf des |COLSP-Befehls nur die Kollision der Sprites 25 bis 30 überprüfen sollte (solange sie das Flag "collided" aktiv haben).

Wenn wir beispielsweise nur 6 Feinde überprüfen müssen, können wir bis zu 2,5 ms pro Ausführung einsparen, wenn wir den Befehl so vorkonfigurieren, dass er erst ab 25 Feinde überprüft. Dies ist besonders wichtig in Spielen, in denen die Spielfigur schießen kann, da in jedem Spielzyklus zumindest die Spielfigur und die Schüsse auf Kollision geprüft werden müssen.

Eine weitere interessante Optimierung, die bei jedem Aufruf 1,1 Millisekunden einsparen kann, besteht darin, den Befehl anzuweisen, immer dieselbe BASIC-Variable zu verwenden, um das Ergebnis der Kollision zu hinterlassen. Zu diesem Zweck geben wir 33 als das Sprite an, das ebenfalls nicht existiert.

col%=0

|COLSP, 33, @col%, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%

Sobald diese beiden Zeilen ausgeführt wurden, belassen nachfolgende Aufrufe von COLSP das Ergebnis in der Variablen col, ohne dass es z. B. angegeben werden muss:

|COLSP, 23

Schließlich ist es möglich, die Empfindlichkeit des COLSP-Befehls einzustellen und zu entscheiden, ob die Überlappung zwischen den Sprites mehrere Pixel oder nur ein Pixel betragen muss, um eine Kollision als gegeben zu betrachten.

Dazu muss die erforderliche Anzahl von überlappenden Pixeln in Y- und X-Richtung mit dem COLSP-Befehl eingestellt und das Sprite 34 (das nicht existiert) angegeben werden.

|COLSP, 34, dy, dx

Die Standardwerte für dy und dx sind 2 bzw. 1. Beachten Sie, dass sie in y-Richtung als Pixel, in x-Richtung jedoch als Bytes gelten (ein Byte entspricht zwei Pixeln im Modus 0).

Für eine Erkennung mit einer minimalen Überlappung (ein Pixel vertikal und/oder ein Byte horizontal) müssen Sie dies tun:

|COLSP, 34, 0, 0

21.1.8 |COLSPALL

Verwendung:

Zum Konfigurieren:

**|COLSPALL,@collider%, @collider%, @collider%, @collider%,
@collider%, @collider%.**

So prüfen Sie auf Kollisionen

**|COLSPALL
|COLSPALL, <Anfangskollider>.**

Diese Funktion prüft, mit wem es kollidiert ist (aus der Gruppe der Sprites, bei denen das Kollisionsflag des Statusbytes auf "1" gesetzt ist) und mit wem es kollidiert ist (aus der Gruppe der Sprites, bei denen das Kollisionsflag des Statusbytes auf "1" gesetzt ist).

Dies ist eine sehr empfehlenswerte Funktion, wenn du Kollisionen deines Charakters und mehrere Schüsse bewältigen musst, da sie Aufrufe von |COLSP spart und somit dein Spiel beschleunigt.

Wichtig: Die Collider (Statusbit 5) werden von 31 bis 0 geprüft. Für jeden Collider werden auch die Colliderables (Statusbit 1) von 31 bis 0 geprüft.

Wenn COLSPALL mit einem einzigen Parameter aufgerufen wird,

|COLSPALL, <Anfangskollider>.

Kollider werden vom angegebenen Kollider -1 bis zum Sprite Null in absteigender Reihenfolge gescannt. Auf diese Weise können Sie, wenn Sie mehr als eine Kollision pro Zyklus von

Spiel können Sie dies tun, indem Sie nacheinander **COLSPALL**, <collider> aufrufen, bis die Variable collider den Wert 32

Beispiel:

|COLSPALL, 7 : rem sucht nach Kollisionen aus Collider 6

21.1.9 |LAYOUT

Verwendung:

|LAYOUT, <y>, <x>, <@String\$>, <@String\$>, <@String\$>, <@String\$>, <@String\$>.

Beispiel:

```
string$ = "XYZZZZ      ZZ"  
|LAYOUT, 0,1, @string$
```

Beachten Sie, dass die Verwendung von **|LAYOUT, 0,1, "XYZZZZZZZZ ZZ"** auf einem CPC464 falsch wäre, obwohl sie auf einem CPC6128 funktioniert. Außerdem können Sie auf dem CPC6128 das "@" weglassen, während es auf dem CPC464 obligatorisch ist.

Diese Routine druckt eine Reihe von Sprites, um das Layout oder "Labyrinth" für jeden Bildschirm zu erstellen. Zusätzlich zum Zeichnen des Labyrinths oder jeder Bildschirmgrafik, die mit kleinen 8x8-Sprites aufgebaut ist, können Sie auch Kollisionen eines Sprites mit dem Layout erkennen, indem Sie den Befehl **|COLAY verwenden**.

Die zu druckenden Sprites werden mit einer Zeichenkette definiert, deren Zeichen (32 mögliche) eines der Sprites darstellen, die dieser einfachen Regel folgen, wobei die einzige Ausnahme das Leerzeichen ist, das das Fehlen eines Sprites darstellt.

Zeichen	Sprite-ID	ASCII-Code
" "	KEINE	
".,"	0	59
"<"	1	
"="		
">"		
"?"		63
"@"	5	
"A"		65
"B"		
"C"	8	67
"D"		
"E"	10	69
"F"		70
"G"		71
"H"		
"T"		
"J"		

"K"		75
"L"		
"M"		
"N"		78
"O"		79
"P"	21	80
"Q"		81
"R"	23	82
"S"		
"T"	25	84
"U"	26	85
"V"		86
"W"		87
"X"	29	88
"Y"	30	
"Z"	31	90

Tabelle 6 Zeichen- und Sprite-Zuordnung für den Befehl |LAYOUT

WICHTIG: Nach dem Drucken des Layouts können Sie die Sprites in Zeichen ändern, so dass Sie immer noch die 32 Sprites haben.

Die y,x-Koordinaten werden im Zeichenformat übergeben. Die Bibliothek verwaltet intern eine 20x25-Zeichen-Map, so dass die Koordinaten die folgenden Werte annehmen:

y nimmt die Werte
[0,24] an, x nimmt
die Werte [0,19] an.

Die zu druckenden Sprites müssen 8x8 Pixel groß sein. Sie sind "Bricks", auch "Tiles" genannt, und werden oft auf die gleiche Weise verwendet.

Wenn Sie andere Sprite-Größen verwenden, wird diese Funktion nicht gut funktionieren. Die Sprites werden zwar gedruckt, aber wenn ein Sprite zu groß ist, müssen Sie Leerzeichen einfügen, um Platz zu schaffen.

Die Bibliothek unterhält eine interne Layout-Map und diese Funktion aktualisiert die internen Layout-Map-Daten, so dass es möglich ist, Kollisionen zu erkennen. Diese Karte ist ein Array von 20x25 Zeichen, wobei jedes Zeichen einem Sprite entspricht.

Der @string ist eine String-Variablen. Sie können den String nicht direkt übergeben, obwohl im CPC6128 die Parameterübergabe dies erlaubt, aber es wäre inkompatibel mit CPC464.

Vorsichtsmaßnahmen:

Die Funktion prüft die übergebene Zeichenkette nicht. Enthält sie Kleinbuchstaben oder andere Zeichen, die sich von den erlaubten unterscheiden, kann dies zu unerwünschten Effekten führen, z. B. zu einem Neustart oder Absturz des Computers. Es darf auch keine leere Zeichenfolge sein!

Die mit SETLIMITS festgelegten Grenzen sollten es Ihnen ermöglichen, dort zu drucken, wo Sie wollen. Wenn Sie später einen kleineren Bereich beschneiden möchten, können Sie SETLIMITS erneut aufrufen, wenn das gesamte Layout gedruckt wird.

Beispiel:

<pre> 2070 SETLIMITS,0,80,0,200 2090 c\$(1)= " PPPPPPPPPPPPPPPPPPPPPPPPPPPP PPPPPPPPPPPPPPPPPPPPP P" 2100 c\$(2)= "PU P" 2110 c\$(3)= "P P" 2120 c\$(4)= "P P" 2130 c\$(5)= " P TPPPPPPPPPU TPPPPPPPPPPPPPPPPPPPPP" 2140 c\$(6)= "P TP" 2150 c\$(7)= "P P" 2160 c\$(8)= "P P" 2170 c\$(9)= " P YYYYYYYYYYYYYYYY P" 2190 c\$(10)="P TPPPPPPPPPPPPPU P" 2195 c\$(11)="P P" 2200 c\$(12)="P P" 2210 c\$(13)="P P" 2220 c\$(14)="YYYYYYYYYYYYYYYYYP YYYYYYYYYYYYYY YY" 2230 c\$(15)="RRRRRRRRRRRRRRR R RRRRRRRRRR " 2240 c\$(16)="PPPPPPPPPPPPPPPP PPPPPPPPPP PPPPP" 2250 c\$(17)="PU TP PU TP" 2260 c\$(18)="P T U P" 2270 c\$(19)="P P" 2271 c\$(20)="P P" 2272 c\$(21)="P W P" 2273 c\$(22)="PP W PP" 2274 c\$(23)="PPPPPPPPPPPPPPPPPPPPPPPPPPPPPP". PPPPPPPPPPPPPPPPPP". 2280 for i=0 to 23 2281 LAYOUT,i,0,@c\$(i) 2282 nächste </pre>	<p>In diesem Beispiel werden mehrere Bausteine verwendet, die zuvor mit dem Werkzeug "SPEDIT" erstellt wurden. Sprite 20 --> O Busch</p> <p>Sprite 21 --> P-Felsen Sprite 22 --> Q-Wolke Sprite 23 --> R Wasser Sprite 24 --> S-Fenster Sprite 25 --> T-Bogen der rechten Brücke Sprite 26 --> U-Bogen der linken Brücke Sprite 27 --> V-Flagge Sprite 28 --> W Pflanze Sprite 29 --> X-Turm-Spike Sprite 30 --> Und Gras Sprite 31 --> Z-Stein</p>
---	---

21.1.10 |LOCATESP

Dieser Befehl ändert die Koordinaten eines Sprites in der Sprite-Attribut-Tabelle.

Verwenden Sie

|LOCATESP, <Seitennummer>, <y>, <x>

Beispiel

|LOCATESP,0,10,20

Eine Alternative zu diesem Befehl, wenn nur eine Koordinate geändert werden soll, ist die Verwendung des BASIC POKE-Befehls, bei dem der gewünschte Wert in die von der X- oder Y-Koordinate belegte Speicheradresse eingefügt wird. Soll eine negative Koordinate eingegeben werden, ist der |POKE-Befehl erforderlich, da dies mit dem

BASIC-POKE-Befehl unzulässig wäre.

Der Befehl **|LOCATE** druckt das Sprite nicht, sondern positioniert es nur für den Zeitpunkt des Drucks.

21.1.11 |MAP2SP

Diese Funktion durchläuft die Weltkarte, die in der Datei map_table.asm beschrieben ist, und wandelt die Kartenelemente, die teilweise oder vollständig auf dem Bildschirm erscheinen, in Sprites um.

Verwenden Sie

|MAP2SP, <y>, <x>

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>, <status>.

Beispiel

|MAP2SP, 1500, 2500

Die von MAP2SP erstellten Sprites werden standardmäßig mit dem Status 3 erstellt, d.h. mit aktivem Druckflag (**|PRINTSPALL** druckt sie) und mit aktivem Kollisionsflag (**|COLSP** kollidiert mit ihnen). Wenn Sie die Sprites mit einem anderen Zustand erstellen möchten, rufen Sie einfach den Befehl **|MAP2SP** einmal mit einem einzigen Parameter auf, der den Zustand angibt, mit dem die Sprites erstellt werden sollen.

Wenn **|MAP2SP zufällig** auf mehr als 32 in Sprites zu übersetzen Elemente stößt, werden die über 32 hinausgehenden ignoriert.

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>.

| Damit wird der MAP2SP-Befehl so konfiguriert, dass er zwar gedruckt wird, aber nicht kollidiert.

WICHTIG: Auf der Weltkarte können Sie normale Bilder mit "Hintergrundbildern" kombinieren (Abschnitt 8.5). Hintergrundbilder sind immer transparent. Das Transparenz-Flag, das Sie in **|MAP2SP, <status>** verwenden, gilt nur für normale Bilder.

Die Parameter **<y>,<x>** der Funktion sind der bewegliche Ursprung, von dem aus die Welt auf dem Bildschirm dargestellt wird. Es gibt drei weitere Parameter in der **MAP_TABLE**, der Tabelle, aus der die Welt definiert wird. Diese Parameter sind die maximale Höhe, die maximale Breite (in negativ) und die Anzahl der Elemente in der Welt (maximal 82).

Jedes Element ist ein Tupel aus 3 Parametern, denen die Abkürzung "DW" vorangestellt ist:

DW Y, X, <Bild>

KARTE TABELLE

;

3 Parameter vor der Liste der "Kartenelemente".

dw 50 ; maximale Höhe eines Sprites für den Fall, dass es den oberen Rand durchbricht und ein Teil davon gemalt werden muss.

dw -40 ; maximale Breite eines Sprites, wenn es sich um ein linksseitiges Sprite handelt (negative Zahl)

db 64 ; Anzahl der zu berücksichtigenden Kartenelemente. Es sollten höchstens 82 sein

und von hier aus beginnen die Posten

dw 100,10,HOUSE; 1

dw 50,-10,CACTUS;2

dw 210,0,HOUSE;3

dw 200,20,CACTUS;4

dw 100,40,HOUSE;5

dw 160,60,HOUSE;6

dw 70,70,HOUSE;7

dw 175,40,CACTUS;8

dw 10,50,HOUSE;9

dw

250,50,HOUSE;10

dw

260,70,HOUSE;11

dw

260,70,HOUSE;11

dw 290,60,CACTUS;12

dw 180,90,HOUSE;13

dw 60,100,HOUSE;14

dw 60,100,HOUSE;14

...

21.1.12 |MOVER

Dieser Befehl verschiebt ein Sprite relativ, d.h. durch Addition relativer Größen zu seinen Koordinaten.

Verwendung:

|MOVER,<Seitennummer>, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>.

Beispiel:

|MOVER,0,1,-1

Das Beispiel bewegt das Sprite 0 gleichzeitig nach unten und nach rechts. Für das Sprite muss das Flag für relative Bewegung nicht aktiviert sein.

Es gibt eine Möglichkeit, **|MOVER** zu verwenden, ohne "dy" oder "dx" anzugeben. Dazu geben wir das Sprite 32 an, das nicht existiert, und geben als Parameter die Speicheradressen der Variablen an, die wir zum Speichern von "dy" und "dx" verwenden wollen.

Die Speicheradresse einer Variablen erhält man, indem man ihr einfach das Symbol "@" voranstellt.

Beispiel:

dy%= 5

dx%= 2

|MOVER,32, @dy, @dx

Von diesem Moment an werden wir in der Lage sein, sie zu nutzen:

|MOVER, <id>

Und damit bewegt sich das Sprite "id" wie durch die Variablen dy, dx angegeben. Dieser Mechanismus funktioniert auch mit **|MOVERALL**

21.1.13 |MOVERALL

Mit diesem Befehl werden alle Sprites relativ bewegt, bei denen das Flag für relative Bewegung aktiviert ist.

Verwenden Sie

|MOVERALL, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>.

Beispiel

|MOVERALL,2,1

Das Beispiel verschiebt alle Sprites mit Relativbewegung-Flag nach unten (2 Zeilen) und 1 Byte nach rechts.

Wenn keine Parameter angegeben werden, werden die im MOVER-Aufruf mit Sprite 32 angegebenen Variablen verwendet, d. h.

|MOVER,32, @dy, @dx

|MOVERALL

Äquivalent zu **|MOVERALL, dy, dx**

Diese "fortgeschrittene" Verwendung des Befehls vermeidet die Übergabe von

Parametern bei jedem Aufruf und ist daher schneller, was in unseren BASIC-Programmen unerlässlich ist.

21.1.14 |MUSIK

Mit diesem Befehl kann eine Melodie abgespielt werden

Verwendung:

|MUSIC,<Kennzeichen_Kanal_C>,<Kennzeichen_Wiederholung>,

<Melodie_Anzahl>, <Geschwindigkeit>.

|MUSIC, <Flag_Wiederholung>, <Melodiennummer>, <Geschwindigkeit>.

|MUSIC :' ohne Parameter endet die Musikwiedergabe

Das C-Kanal-Flag mit dem Wert 1 erlaubt es, den dritten Soundkanal frei zu lassen, so dass er für Soundeffekte (Trigger etc.) mit dem Befehl verwendet werden kann

SOUND 4,<Note>,<Dauer>, ...

Beachten Sie, dass Sie Kanal 4 verwenden müssen, da in BASIC die Kanäle als A=1, B=2, C=4 eingegeben werden.

Wenn das Kanalflag weggelassen oder auf Null gesetzt wird, verwendet die Musik alle 3 Kanäle und Sie können den SOUND-Befehl nicht gleichzeitig verwenden (Sie können es, aber mit seltsamen Effekten).

Das Wiederholungskennzeichen muss Null sein, damit die Musik in einer Schleife abgespielt wird. Wenn Sie möchten, dass die Musik nur einmal abgespielt wird, sollten Sie den Wert 1 verwenden.

Die Nummer der Melodie muss zwischen 0 und 7 liegen.

Die "normale" Geschwindigkeit ist 6. Bei einer höheren Zahl wird langsamer, bei einer niedrigeren Zahl schneller gespielt.

Der Befehl |MUSIC, der ohne Parameter aufgerufen wird, deaktiviert die Musikunterbrechung und stoppt die Wiedergabe einer Melodie.

Beispiele:

|MUSIK,0,0,0,0,6

|MUSIK,1,0,0,0,6

|MUSIC,0,0,6

|MUSIK

Intern bewirkt der Befehl, dass ein Interrupt installiert wird, der 300 Mal pro Sekunde ausgelöst wird. Wenn wir die Geschwindigkeit 6 einstellen, wird bei jedem sechsten Auslösen die Musikwiedergabefunktion ausgeführt.

Da es sich um einen Interrupt-Befehl handelt, muss ein Programm laufen, damit die Musik gespielt werden kann, denn während der BASIC-Interpreter auf Befehle wartet, sind solche Interrupts nicht aktiviert. Wenn Sie einfach den Befehl |MUSIC ausführen, werden Sie nichts hören, aber wenn Sie ihn innerhalb eines Programms wie dem unten gezeigten ausführen, wird die Musik gespielt.

10 |MUSIK,0,0,0,5

20 goto 20: ' Endlosschleife. Wenn sie läuft, spielt die Musik

21.1.15 |PEEK

Dieser Befehl liest einen 16-Bit-Datenwert aus einer bestimmten Speicheradresse. Er ist

für die Abfrage der Koordinaten von Sprites gedacht, die sich mit automatischer oder relativer Bewegung bewegen.

Verwenden Sie
|PEEK,<Adresse>, @Daten%.

Beispiel

Daten%=0

|PEEK, 27001, @dato%, @dato%, @dato%, @dato%, @dato%, @pEEK

Wenn die Koordinaten nur positiv und kleiner als 255 sind, können Sie den BASIC-Befehl PEEK verwenden, da dieser etwas schneller ist.

21.1.16 |POKE

Dieser Befehl fügt einen 16-Bit-Wert (positiv oder negativ) in eine Speicheradresse ein. Er ist für die Änderung von Sprite-Koordinaten gedacht, da der POKE-Befehl keine negativen Koordinaten oder Koordinaten größer als 255 verarbeiten kann, da POKE mit Bytes arbeitet, während |POKE ein 16-Bit-Befehl ist.

Verwendung:

|POKE, <Adresse>, <Wert>.

Beispiel:

|POKE, 27003, 23

In diesem Beispiel wird der Wert 23 an die x-Koordinate von Sprite 0 gesetzt. Es ist eine sehr schnelle Funktion, aber wenn Sie nur positive Koordinaten verarbeiten wollen, ist es besser, POKE zu verwenden, da es noch schneller ist.

21.1.17 |PRINTAT

|PRINTAT kann eine Zeichenkette unter Verwendung eines neuen, kleineren Zeichensatzes (ich nenne sie "Mini-Zeichen") drucken. Dieser neue Befehl ermöglicht es Ihnen, den Transparenzmechanismus der Sprites zu nutzen, so dass Sie Zeichen drucken können, ohne den Hintergrund zu verdecken. Er funktioniert wie folgt:

Verwendung:

|PRINTAT,<Flag Transparenz>, y,x,@String

Beispiel:

cad\$= "Hallo".

|PRINTAT,0,100,10, @cad\$

Der Befehl |PRINTAT druckt Zeichenketten und keine numerischen Variablen. Wenn Sie also eine Zahl ausdrucken möchten (z. B. die Punkte auf der Anzeigetafel in Ihrem Videospiel), müssen Sie dies tun:

```
points=points+1
cad$= str$(points)
|PRINTAT,0,100,10, @cad$
```

Der Befehl |PRINTAT wird nicht von den Grenzen für das Clipping beeinflusst, die mit |SETLIMITS. Dies ist sehr logisch, da Sie normalerweise PRINTAT verwenden werden, um Noten auf Ihren Markern zu drucken, die außerhalb des durch |SETLIMITS begrenzten Bereichs liegen.

Im Gegensatz zum BASIC-Befehl PRINT ist der Befehl |PRINTAT recht schnell und kann verwendet werden, um Ihre Spielmarkierungen häufig zu aktualisieren. PRINTAT verwendet ein neu definiertes Alphabet, das eine reduzierte oder andere Version der "offiziellen" Amstrad-Zeichen enthalten kann. 8BP bietet standardmäßig ein kleines Alphabet, das aus Zahlen, Großbuchstaben und einigen Symbolen besteht. Es sieht folgendermaßen aus:

"0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ!: ,."

Sie können keine Zeichen verwenden, die nicht in diesem Satz enthalten sind, wie z. B. Kleinbuchstaben. Wenn Sie dies versuchen, wird das letzte in der Zeichenfolge definierte Zeichen (in diesem Fall das Leerzeichen) gedruckt.

Die Zeichen dieses Alphabets sind alle gleich groß: 4 Pixel breit x 5 Pixel hoch, d. h. 2 Byte x 5 Zeilen.

Das Standardalphabet enthält keine Kleinbuchstaben und viele Symbole fehlen, aber Sie können Ihr eigenes Alphabet erstellen, das diese enthält.

21.1.18 |PRINTSP

Verwendung:

|PRINTSP, <sprite id >, <y >,<x>, <x>.
|PRINTSP, <sprite id >
|PRINTSP, 32, <Anzahl der Hintergrundpixel>.

Beispiel:

Drücke Sprite 23 an den Koordinaten y=100, x=40 (Aktualisierung der Koordinaten).

|PRINTP, 23,100,40

Beispiel:

Drückt Sprite 23 an den bereits in der Sprite-Tabelle zugewiesenen Koordinaten:

|PRINTSP, 23

Wenn Sprite 32 (das nicht existiert) angegeben wird, dann wird der folgende Parameter verwendet, um die Anzahl der Hintergrundbits im transparenten Druck anzugeben. Wenn 1 Bit verwendet wird, können 2 Hintergrundfarben verwendet werden. Wenn 2 angegeben wird, können 4 Hintergrundfarben verwendet werden.

Wenn eine sprite_id <32 angegeben ist, drückt der Befehl ein Sprite auf den Bildschirm, und wenn Koordinaten angegeben sind, werden diese auch aktualisiert.

Die berücksichtigten Koordinaten sind:

- Anzahl der Zeilen in vertikaler Richtung [-32768..32768]. Die entsprechenden Zeilen innerhalb des Bildschirms sind [0..199].
- Anzahl der Bytes in der Horizontalen [-32768..32768]. Diejenigen, die dem Inneren des Bildschirms entsprechen, sind [0..79].

Normalerweise werden Sie in der Logik eines Videospiels **|PRINTSPALL** verwenden, da es schneller ist, sie alle auf einmal zu drucken. Zu anderen Zeitpunkten im Spiel kann es jedoch sinnvoll sein, Sprites separat zu drucken. Dieses Beispiel zeigt das Herablassen eines "Vorhangs", wobei ein einzelnes Sprite verwendet wird, das sich horizontal wiederholt und beim Herablassen den Bildschirm rot "färbt", wodurch der Eindruck eines herabfallenden Vorhangs entsteht.

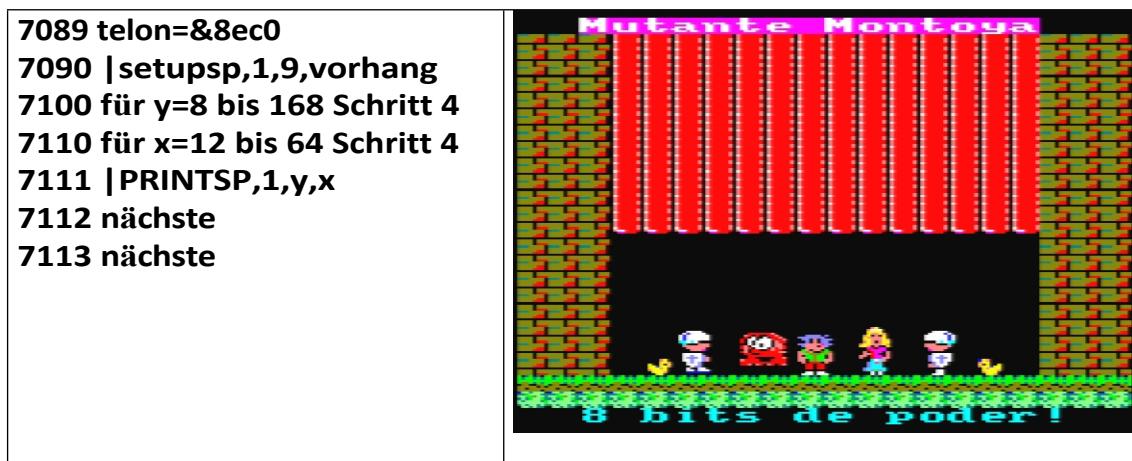


Abb. 119 Ein Beispiel für die Verwendung von PRINTSP

21.1.19 |PRINTSPALL

Diese Routine druckt alle Sprites mit gesetztem Statusbit0 auf einmal.

Verwendung:

|PRINTSPALL, <ordenini>, <ordenfin>, <flag anima>, <flag sync>.
|PRINTSPALL, <Auftragsart>,

Beispiel:

Mit den folgenden Werten gibt der Befehl alle Sprites aus, indem er sie zuerst und unsynchronisiert mit Sweep und ohne Sortierung animiert:

|PRINTSPALL, 0, 0, 0, 1, 0
|PRINTSPALL, 0, 1, 0: rem wenn commandini weggelassen wird, wird der zuletzt zugewiesene Wert genommen, oder Null, wenn er nie zugewiesen wurde

Animation Flag, kann auf 1 oder 0 gesetzt werden. Wenn 1 gesetzt ist, wird das Bild vor dem Drucken der Sprites in seiner Animationssequenz geändert, solange die Sprites das Statusbit 3 gesetzt haben. **WICHTIG:** Die Animation wird vor dem Drucken durchgeführt, nicht nach dem Drucken. Das heißt, wenn Sie gerade eine Animationssequenz zugewiesen haben, werden Sie das erste Bild dieser Sequenz nicht sehen.

Das <Flag sync> ist ein Synchronisationsflag mit der Bildschirmabtastung. Es kann 1 oder 0 sein. Die Synchronisation macht nur Sinn, wenn Sie das Programm mit einem Compiler wie "Fabacom" kompilieren. Die BASIC-Logik läuft langsam und die Synchronisierung mit dem Sweep führt zu kleinen zusätzlichen Wartezeiten in jedem Zyklus des Spiels, so dass es nicht sinnvoll ist.

Als Faustregel kann man sagen, dass sie nur dann geeignet ist, wenn Ihr Spiel in der

Lage ist, 50 fps pro Sekunde zu generieren, oder anders ausgedrückt, einen vollen Spielzyklus alle 20 Millisekunden. Wenn Sie Ihr Spiel mit einem Compiler wie "fabacom" kompilieren, ist es empfehlenswert, es mit dem Screen-Sweep zu synchronisieren, weil Sie dann mit ziemlicher Sicherheit die 50fps erreichen und einen vollen Spielzyklus alle 20 Millisekunden erhalten können.

Wenn Sie diese Werte überschreiten, produziert Ihr Spiel mehr Bilder, als der Bildschirm anzeigen kann, und dann werden einige nicht angezeigt und die Bewegung ist nicht flüssig.

Je mehr Sprites Sie auf dem Bildschirm haben, desto länger dauert der Befehl, obwohl er sehr schnell ist. Es gibt viele Sprites, die auf dem Bildschirm erscheinen können, aber nicht gedruckt werden müssen (sie können das 0-Statusbit ausgeschaltet haben), wie Früchte, Münzen, Bonuselemente im Allgemeinen und/oder Charaktere, die sich nicht bewegen und keine Animation haben. Auch wenn sie nicht gedruckt werden, kann bei ihnen das Kollisionsbit gesetzt sein und somit die Routine beeinflussen.

|COLSP und |COLSPALL

Die Ordnungsparameter ("ordenini", "ordenfin") geben die Anfangs- und End-Sprites an, die die Gruppe von Sprites definieren, die nach der "Y"-Koordinate geordnet sind und die wir drucken werden. Wenn wir zum Beispiel die Werte 0,0 zuweisen, werden sie nacheinander von Sprite 0 bis Sprite 31 gedruckt. Wenn wir 0,8 zuweisen, werden sie von 0 bis 8 geordnet (9 Sprites) und von 10 bis 31 nacheinander gedruckt. Wenn wir 0,31 setzen, werden alle Sprites in der Reihenfolge gedruckt. Bei der Einstellung 10,20 werden die Sprites nacheinander von 0 bis 9, dann geordnet von 10 bis 20 und schließlich nacheinander von 21 bis 31 gedruckt.

Die Reihenfolge ist sehr nützlich für die Erstellung von Spielen des Typs "Renegade" oder "Golden AXE", bei denen es notwendig ist, einen Tiefeneffekt zu erzielen.

Wenn der Parameter "ordenini" weggelassen wird, wird der zuletzt zugewiesene Wert berücksichtigt, oder Null, wenn noch nie ein Wert zugewiesen wurde. Wenn Sie einen der beiden Sortierparameter ändern wollen, ist es außerdem ratsam, zuerst PRINTSPALL,0,0,0,0 auszuführen, damit die Sprites zunächst sequentiell neu sortiert werden, bevor sie mit einer neuen Konfiguration sortiert werden.

Geordnetes Drucken ist rechenintensiver als sequentielles Drucken. Wenn Sie nur 5 Sprites haben, die sortiert werden müssen, übergeben Sie eine 4 als Sortierparameter, nicht eine 31. Das Sortieren aller Sprites dauert etwa 2,5ms, aber wenn Sie nur 5 sortieren, können Sie 2ms sparen. Vielleicht hast du viele Sprites und es lohnt sich nicht, einige davon zu sortieren, wie die Schüsse oder Sprites, von denen du weißt, dass sie sich nicht überlappen werden.

Die Sortierung von |PRINTSPALL ist partiell, d.h. bei jedem Aufruf wird nur ein Paar von ungeordneten Sprites sortiert. Manchmal möchte man, dass die Sortierung für jedes Bild vollständig ist. Das heißt, dass nicht bei jedem Aufruf von |PRINTSPALL ein Sprite-Paar sortiert werden soll, sondern dass man sicher sein will, dass alle Sprites sortiert sind. Die 8BP-Bibliothek ermöglicht Ihnen dies durch ihre vier Sortiermodi, die Sie durch den Aufruf des Befehls |PRINTSPALL mit einem einzigen Parameter einstellen können (führen Sie ihn nur einmal aus, um den Sortiermodus einzustellen):

PRINTSPALL,0 : teilweise Sortierung nach Ymin PRINTSPALL,1 : vollständige Sortierung nach Ymin PRINTSPALL,2 : teilweise Sortierung nach Ymax PRINTSPALL,3 : vollständige Sortierung nach Ymax

Die Sortierung nach Ymax basiert auf der größten Y-Koordinate der Sprites, d.h. wo sich ihre Füße befinden und nicht ihre Köpfe. Wenn die Sprites gleich groß sind, kann eine Ymin-basierte Sortierung funktionieren, aber wenn die Sprites unterschiedlich hoch

sind, möchten Sie vielleicht danach sortieren, wo sich die Füße der einzelnen Figuren befinden, und dafür müssen Sie den Sortiermodus 2 oder 3 verwenden.

Die Ymax-Sortiermodi sind langsamer, etwa 0,128 ms pro Sprite, und sollten daher nur bei wirklichem Bedarf verwendet werden.

Die vollständige Sortierung verbraucht nur wenig mehr als die teilweise Sortierung (etwa 0,3 ms). Das liegt daran, dass die Sprites kaum von einem Frame zum nächsten durcheinander gewürfelt werden, aber selbst diese 0,3 ms sind es wert, wenn möglich eingespart zu werden.

Es gibt ein sehr interessantes Verhalten dieser Funktion, um 1ms bei ihrer Ausführung zu sparen. Es besteht darin, sie einmal mit Parametern und die folgenden Male ohne Parameter aufzurufen. In diesem Fall wird davon ausgegangen, dass, auch wenn keine Parameter übergeben werden, ihre Werte gleich den zuletzt übergebenen sind. Auf diese Weise arbeitet der Parser weniger und verkürzt die Ausführungszeit.

21.1.20 |RINK

Mit diesem Befehl können Sie eine Animation mit Tinten durchführen. Verwendung:

|RINK, <initial_ink>, <Farbe1>, <Farbe2>, . . . , <FarbeN>, <FarbeN>, <FarbeN>, <FarbeN>, <FarbeN>, <FarbeN>, <FarbeN>, <FarbeN>, <FarbeN>.

|RINK, <Schritt>

RINK dreht eine Reihe von Farben, beginnend mit der Ausgangsfarbe N Farben (beliebige Anzahl von Farben), entsprechend der Größe des definierten Farbmusters. Die Geschwindigkeit der Rotation kann über den Parameter step gesteuert werden, der die Anzahl der Farbsprünge angibt, die jede Farbe bei einem Aufruf macht.

Empfehlung: Aufgrund der Verwendung von Unterbrechungen verursacht **|RINK** in einigen Fällen "Stottern", wenn es gleichzeitig mit dem Befehl **|MUSIC** in Geschwindigkeit 6 verwendet wird. Falls Sie beides gleichzeitig verwenden möchten, ohne dass es zu Störungen kommt, verwenden Sie eine andere Geschwindigkeit für Musik (Sie können Geschwindigkeit 5 oder 7 verwenden, beide funktionieren gut).

Beispiele:

Dieser Befehl definiert ein Muster aus 4 Rottönen (Farbe =3) und 4 Gelbtönen (Farbe =24), das von Farbe 8 bis Farbe 15 gedreht werden soll.

|TRINK, 8, 3, 3, 3 , 3, 3, 24, 24, 24, 24, 24, 24

Mit diesem Befehl wird ein Muster aus 2 Weiß- (Farbe =26) und 2 Grautönen (Farbe=13) definiert, das von Farbe 3 nach Farbe 6 gedreht wird.

|TRINKEN, 3, 26, 26, 13, 13, 13

Drehen Sie eine Farbe des Musters alle Druckfarben

|RINK, 1

Bei einem 8-Farben-Muster lässt der folgende Befehl alles so, wie es war

|RINK, 8

Dieser Befehl bewirkt nichts anderes, als dass die Farben die Farbe des Musters annehmen.

|RINK, 0

21.1.21 |ROUTEALL

Mit diesem Befehl können Sie alle Sprites, die in ihrem Statusbyte das Route-Flag aktiv haben, über die ihnen zugewiesene Route (Parameter 15 von **|SETUPSP**) routen.

Verwendung:

|ROUTEALL

Er hat keine Parameter und ist daher sehr einfach aufzurufen. Was dieser Befehl intern tut, ist, einen Schrittzähler für das Segment zu führen, das jedes Sprite durchläuft, so dass er, wenn das Segment zu Ende geht, die Geschwindigkeit des Sprites ändert.

Der Befehl ändert die Koordinaten der Sprites nicht, so dass sie mit **|AUTOALL** bewegt und mit **|PRINTSPALL** gedruckt (und animiert) werden müssen. Aus diesem Grund gibt es einen optionalen Parameter in **|AUTOALL**, so dass **|AUTOALL,1** intern **|AUTOALL,1** aufruft.

|ROUTEALL, bevor Sie das Sprite bewegen, was Ihnen einen BASIC-Aufruf erspart, der immer eine kostbare Millisekunde in Anspruch nehmen wird.

Die Routen werden in der Routendatei routes_yourgame.asm definiert

DEFINITION DER EINZELNEN STRECKEN

=====

ROUTE0; ein Kreis

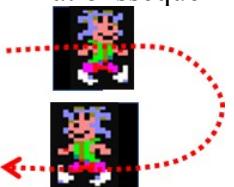
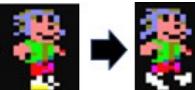
```
db 5,2,0
db 5,2,-1
db 5,0,-1
db 5,-2,-1
db 5,-2,0
db 5,-2,1
db 5,0,1
db 5,2,1
db 0
```

Das letzte Segment ist Null, was bedeutet, dass der Pfad beendet ist und das Sprite von vorne beginnen muss. Stellen Sie sicher, dass die Anzahl der Schritte in jedem Segment 250 nicht überschreitet und dass sowohl Vy als auch Vx zwischen -127 und 127 liegen.

Um einem Sprite einen Pfad zuzuweisen, müssen Sie den Befehl SETUPSP verwenden, indem Sie den Parameter 15 angeben. Das folgende Beispiel ordnet den Pfad 3 dem Sprite 31 zu

|SETUPSP, 31, 15, 3

Es gibt 4 Funktionalitäten, die Sie in der Mitte (**nicht am Ende**) einer Route nutzen können:

Escape-Code (Feld "Anzahl der Schritte")	Beschreibung	Beispiel
255	Änderung des Sprite-Zustands.	DB 255, 3, 0 Der Zustand geht auf den Wert 3. Die Null am Ende ist ein Füllzeichen.
254	<p>Änderung der Animationssequenz</p>  <p>Wenn Sie nach der Änderung der Reihenfolge auch das Bild ändern möchten, müssen Sie den Code 251 verwenden</p>	DB 254, 10, 0 Die Sequenz 10 ist assoziiert. Die Null ist eine Auffüllung. Wenn die zugewiesene Sequenz diejenige ist, die das Sprite bereits hat, dann ist es harmlos (die Frame-ID wird nicht zurückgesetzt). Wenn Sie die Rahmenkennung zurücksetzen wollen, muss der dritte Parameter eine 1 sein, z. B. DB 254, 10, 1.
253	<p>Änderung des Images</p> 	DB 253 DW new_img Es wird das Bild "new_img" zugeordnet, das eine Speicheradresse sein muss.
252	Änderung der Route	DB 252, 2, 0 Route 2 ist verbunden mit
251	<p>Weiter zu zu nächsten von der</p> <p>Rahmen Animation.</p> 	DB 251, 0, 0 Das Sprite ist animiert. Die zwei Nullen sind Füller

WICHTIG: Achten Sie sehr darauf, DB und DW dort zu schreiben, wo sie verwendet werden sollen, d.h. wenn Sie z.B. Bilder ändern, sollten Sie dem Bild DW und nicht DB voranstellen. Wenn Sie einen solchen Fehler machen, wird Ihre Route nicht funktionieren.

WICHTIG: Eine Route kann maximal 255 Bytes lang sein, und ein Segment ist 3 Bytes lang, also kann eine Route maximal 84 Segmente lang sein. Es kann sein, dass Sie eine noch längere Route erstellen müssen. In diesem Fall können Sie das Ende einer Route mit einem Routenwechsel zu einer anderen Route (Code 252) verketten, und Sie

können beliebig viele Routen verketten.

WICHTIG: Escape-Codes können in der Mitte einer Strecke verwendet werden, aber das letzte Segment kann kein Escape-Code sein, es muss eine Bewegung sein, auch wenn es stillsteht, etwa "DB 1,0,0".

In diesen Fällen interpretiert **|ROUTEALL**, dass eine Änderung des Zustands, der Sequenz, des Bildes, des Sprite-Pfads oder der Animation erzwungen werden muss und führt auch den nächsten Schritt aus. Änderungen können an jedem Segment des Pfades erzwungen werden, nicht unbedingt am Ende, und Sie können so viele Änderungen erzwingen, wie Sie wollen.

```
ROUTE0; ein Schuss übrig
;-----
db 100,0,-1; einhundert Schritte links bei
Geschwindigkeit Vx=-1 db 255,0,0; Deaktivierung des
Sprites mit Status=0
db 1,0,0 ; in diesem Schritt nichts
bewegen db 0
ROUTE1; ein Sprung nach
rechts db 253
dw SOLDIER_R1_UP
db 1,-5,1
db 2,-4,1
db 2,-3,1
db 2,-2,1
db 2,-1,1
db 253
dw SOLDIER_R1_DOWN
db 1,-5,1; up so UP und down fit db 2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDIER_R1
db 1,5,1; noch einen runter
db 255,13,0; neuer Zustand, ohne Pfadflagge und mit
Animationsflagge db 254,32,0; Makrofolge 32
db 1,0,0; quietooo.!!!!
db 0
```

21.1.22 |ROUTEESP

Mit diesem Befehl können Sie ein einzelnes Sprite, in dessen Statusbyte das Route-Flag aktiv ist, über die ihm zugewiesene Route (SETUPSP-Parameter 15) leiten.

Verwendung:

|ROUTEESP, <spriteid>, <steps>.

|ROUTEESP, <spriteid> : rem in diesem Fall gilt "steps" als=1 Dieser Befehl bewegt ein Sprite eine beliebige Anzahl von Schritten (**bis zu 255**) entlang des ihm zugewiesenen Pfades. Der Befehl bewegt das Sprite durch alle angegebenen Schritte und lässt es am Ende an der gleichen Position stehen, die es gehabt hätte, wenn wir **|AUTOALL,1** so oft wie die Anzahl der Schritte ausgeführt hätten.

WICHTIG: Die Schritte können keinen größeren Wert als 255 annehmen.

21.1.23 |SETLIMITS

Dieser Befehl legt die Grenzen des Bereichs fest, in dem Sprites oder Sterne gedruckt werden können.

Verwendung:

|SETLIMITS, <xmin>, <xmax>, <ymin>, <ymax>, <ymax>, <ymax>, <ymax>, <ymax>, <ymax>, <ymax>, <ymax>, <ymax>.

Beispiel, bei dem der gesamte Bildschirm als zulässiger Bereich festgelegt wird
|SETLIMITS,0,80,0,200

Außerhalb dieser Grenzen werden die Sprites beschnitten, d. h. wenn ein Sprite teilweise teilweise außerhalb des Bereich erlaubt ist, die Funktionen |PRINTSP y

|Der PRINTSPALL druckt nur den Teil, der innerhalb des erlaubten Bereichs liegt.

21.1.24 |SETUPSP

Dieser Befehl lädt Daten aus einem Sprite in die SPRITES_TABLE Verwendung:

|SETUPSP, <id_sprite>, <param_number>, <value>.

Beispiel:

|SETUPSP, 3, 7, 2

Ermöglicht z. B. die Zuweisung einer neuen Animationssequenz, wenn das Sprite die Richtung ändert, oder einfach die Änderung des Registers der Statusflags.

Mit dieser Funktion können wir jeden Parameter eines Sprites ändern, außer X, Y (was mit LOCATE_SPRITE gemacht wird).

Es kann immer nur ein Parameter auf einmal geändert werden. Der zu ändernde Parameter wird mit param_number angegeben. Die param_number ist eigentlich die relative Position des Parameters in der SPRITES_TABLE

Param-Nummer	Aktion	Mögliche Verwendung von POKE oder POKE als Alternative
0	ändert den Status (belegt 1 Byte)	YES
5	ändert Vy (belegt 1Byte, Wert in vertikalen Linien). Sie können auch ändern Vx zur gleichen Zeit, wenn wir es am Ende als Parameter hinzufügen	YES
	ändert Vx (belegt 1Byte, Wert in horizontalen Bytes)	YES
	Änderungssequenz (belegt 1Byte, nimmt die Werte 0..31 an)	NEIN, denn SETUPSP setzt auch die frame_id zurück und gibt Ihnen auch weist die Adresse des ersten Bildes zu.

8	change frame_id (belegt 1Byte, nimmt die Werte 0..7 an)	YES
	change dir image (belegt 2Byte). Das angegebene Bild kann eines aus der anfänglichen Liste der Bilder in der Datei images_mygame.asm sein,	Anders verhält es sich, wenn ein Bild <255 verwendet wird. Wenn eine Speicheradresse verwendet wird, kann alternativ POKE verwendet werden.
	den Pfad ändern (benötigt 1 Byte)	NEIN, denn SETUPSP macht mehr Dinge in internen Tabellen, damit die Route funktioniert.

Beispiel:

In diesem Beispiel haben wir Sprite 31 das Bild eines Schiffes gegeben, das an der Adresse &a2f8 zusammengesetzt ist.

Schiff = &a2f8

|SETUPSP, 31, 9, Kirchenschiff

Es gibt einen einfacheren Weg, das Bild für das Sprite festzulegen, indem man die IMAGE_LIST in der Datei images_yourgame.asm verwendet. Wenn wir das NAVE in der IMAGE_LIST haben, können wir einen Bezeichner zwischen 16 und 255 zuordnen

|SETUPSP,31, 9, 16 : rem die 16 ist die Kennung des SHIP in der IMAGE_LIST

BILD_LISTE

Wir werden hier eine Liste der Bilder einfügen, die wir verwenden wollen, ohne die Speicheradresse von basic anzugeben.

Der Befehl |SETUPSP,<id>,9,<Adresse> wird zu |SETUPSP,<id>,9,<Nummer>; somit wird der Befehl |SETUPSP,<id>,9,<Adresse> zu |SETUPSP,<id>,9,<Nummer>.

Der Vorteil des Verzichts auf die Verwendung von Speicheradressen in BASIC besteht darin, dass wir die Grafiken vergrößern oder sie in

Die Nummer, die wir vergeben, wird sich nicht ändern.

Sie müssen NICHT alle eine Nummer haben, nur die, die wir mit |setupsp, id, 9,<num> verwenden werden.

Die Nummerierung beginnt bei 16-----

Wir können bis zu 255 auf diese Weise spezifizierte Bilder verwenden.

Die Liste muss nicht 255 Elemente lang sein, sie ist von variabler Länge, sie kann sogar leer sein.

:

DW NAVE ; 16

DW ANDERES_SCHIFF ; 17

----- BEGIN IMAGE ----- NAVE

```
db 7 ; Breite
db 0 , 0 , 0 , 0 , 0 , 0 , 0
db 12 , Höhe , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0 , 0
db 0 , , 48 , 0 , 0 , 0 , 0
db 0 , , 240 , 48 , 0 , 0 , 0
db 0 , , 207 , 112 , 12 , 0 , 0
db 0 , 84 , 240 , 48 , 164 , 8 , 0
db 0 , 0 , 48 , 176 , 112 , 12 , 0
db 0 , 69 , 48 , 112 , 48 , 101 , 0
db 0 , 16 , 48 , 207 , 207 , 0 , 0
db 0 , , , 80 , 0 , 0 , 0
```

----- END IMAGE -----

db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0

Im Fall von param_number=5 können wir Vx als Parameter am Ende einfügen:
|SETUPSP, 31, 5, Vy, Vx

Auf diese Weise aktualisieren wir die beiden Geschwindigkeiten mit einem einzigen Befehl, der Folgendes kostet

3,73 ms im Gegensatz zu den 6,8 ms, die es dauern würde, zwei Befehle getrennt aufzurufen.

Bei Verwendung von param_number=7 wird nicht nur die Animationssequenz geändert, sondern der Befehl aktualisiert auch automatisch das Bild (frame id), indem es in das erste Bild (die Null) gesetzt wird, und die Bildadresse wird aktualisiert, so dass Sie param_number=9 nicht aufrufen müssen, um das Sprite-Bild in das erste Bild der neu zugewiesenen Sequenz zu ändern. Wenn Sie |ANIMALL vor dem Drucken verwenden, oder

|Auch wenn SETUPSP die Animation bei Bild Null ansetzt, springen Sie vor dem Drucken zu Bild 1. Normalerweise ist das kein Problem, aber im Falle einer Todessequenz, bei der das erste Bild z. B. das Löschen des Sprites ist, möchten Sie vielleicht nicht direkt zu Bild 1 springen. In diesem Fall kann ein einfacher Trick darin bestehen, Bild Null in der Definition der Todessequenz zu wiederholen. Auf diese Weise können Sie sicherstellen, dass das Bild sichtbar ist. Eine andere Möglichkeit ist, das Animationsflag zu entfernen und es nach dem Drucken mit ANIMASP zu animieren.

Im Fall von param_number=15 wird dem Sprite nicht nur der Pfad in der Attributabelle zugewiesen, sondern der Befehl führt auch eine interne Datenrücksetzung durch, so dass das Sprite diesen Pfad ab dem ersten Segment des betreffenden Pfades durchläuft.

21.1.25 |STARS

Bewegt eine Bank von bis zu 40 Sternen auf dem Bildschirm (innerhalb der Grenzen, die durch |SETLIMITS), ohne andere Sprites zu übermalen, die bereits gedruckt wurden.

|STARS,<Einstiegsstern>,<Anzahl Sterne>,<Farbe>,<dy>,<dx>,<dx>.

Beispiel:

|STERNE, 0, 15, 3, 1, 0

Im Beispiel werden 15 Sterne der Farbe 3 (rot) um ein Pixel vertikal verschoben (da dy=1 und dx=0). Bei wiederholtem Aufruf entsteht der Eindruck eines scrollenden Sternenhintergrunds. Wenn ein Stern die Bildschirmgrenze oder die mit |SETLIMITS festgelegte Grenze verlässt, taucht er auf der gegenüberliegenden Seite wieder auf, so dass ein Gefühl der Kontinuität im Sternenfluss entsteht.

Die Sternbank befindet sich an der Adresse 42540 (=A62C) und hat eine Kapazität von 40 Sternen bis zur Adresse 42619. Jeder Stern verbraucht 2 Bytes, eines für die Y-Koordinate und eines für die X-Koordinate.

Gruppen von Sternen können separat verschoben werden, beginnend bei einem Stern Ihrer Wahl. Die Anfangskoordinaten der Sterne müssen vom Programmierer initialisiert werden.

Beispiel für die Initialisierung und Verwendung in einer Schriftrolle mit vier Sternebenen, um ein Gefühl von Tiefe zu vermitteln. Jede Ebene bewegt sich mit einer anderen Geschwindigkeit

1 SPEICHER 24999

10 CALL &6b78: rem installiert die RSX-Befehle

```

20 bank=42540
30 FOR star=0 TO 39:' Schleife zur Erstellung von 40 Sternen
40 POKE Bank+Stern*2,RND*200
50 POKE Bank+Stern*2+1,RND*80
60 NÄCHSTE
70 MODUS 0
80 REM malen und bewegen wir 4 Sternenflugzeuge mit je 10 Sternen.
90 |STARS,0,10,3,0,-1: ' 3 ist rot. Die weiter entfernten bewegen sich
mehr
langsam
91 |STARS,10,10,2,0,-2: ' 2 ist blau
92 |STARS,20,10,1,0,-3: ' 1 ist gelb
93 |STARS,30,10,4,0,-4: ' die 4 ist weiß. Die nächstgelegenen gehen
mehr
schnell
95 goto 90

```

Die Verwendungsmöglichkeiten dieses Befehls können sehr vielfältig sein.

- Die gleichzeitige Verwendung von mehreren Sternenbänken mit unterschiedlicher Geschwindigkeit und Farbe kann einen Eindruck von Tiefe vermitteln.
- Wenn die Sterne diagonal ausgerichtet sind, können Sie einen "Regeneffekt" erzielen.
- Wenn die Farbe schwarz und der Hintergrund braun oder orange ist, können Sie den Eindruck erwecken, dass Sie über sandiges Gebiet vorrücken.
- Wenn die Bewegung eine rollende Bewegung ist und die Farbe der Sterne weiß ist, können Sie den Eindruck von Schnee erwecken. Die rollende Bewegung kann mit einem Zickzack in X erreicht werden, während die Geschwindigkeit in Y beibehalten wird, oder sogar mit trigonometrischen Funktionen wie Cosinus. Wenn Sie Cosinus in der Spiellogik verwenden, wird das natürlich sehr langsam sein, aber Sie können den vorberechneten Cosinus-Wert in einem Array speichern. Beispiel für einen Schneeffekt:

```

1 SPEICHER 23999: MODUS 0
30 ' Initialisierung der 40-Sterne-Bank
40 FOR dir=42540 TO 42619 STEP 2
45 POKE dir,RND*200:POKE dir+1,RND*80
48 NÄCHSTES
50 |STARS,0,20,4,2,dx1
60 |STARS,20,20,4,1,dx2
61 dx1=1*COS(i):dx2=SIN(i)
69 i=i+1: IF i=359 THEN i=0
70 GOTO 50

```

Es gibt eine Möglichkeit, die Ausführung zu beschleunigen, und zwar die Vermeidung der Übergabe von Parametern. In diesem Buch haben wir gesehen, wie teuer die Übergabe von Parametern ist, selbst wenn der aufgerufene Befehl nichts tut. Nun, wir haben es mit einem Befehl zu tun, der 5 Parameter benötigt, also ist er besonders teuer. Wenn wir die Zeit, die BASIC zur Interpretation der Parameter benötigt, reduzieren wollen, können wir den Befehl einfach einmal mit Parametern und die folgenden Male ohne Parameterübergabe aufrufen.

|STERNE,0,10,1,1,5,0

|STARS : dieser parameterlose Aufruf nimmt die gleichen Werte an wie der letzte

Aufruf.

Diese Möglichkeit ist vor allem in Spielen nützlich, in denen wir den Befehl zum Bewegen von Sternen bei jedem Spielzyklus aufrufen wollen, da wir dadurch etwa 1,7 ms sparen.

WICHTIG: Der Befehl STARS wird von den **|SETLIMITS-Grenzwerten** beeinflusst, aber nur, wenn Vx oder Vy ungleich Null sind. Wenn beide Null sind, wird **|SETLIMITS** nicht beeinflusst und die Sterne können auf dem ganzen Bildschirm gemalt werden.

21.1.26 |UMAP

Dieser Befehl aktualisiert die Karte mit Informationen, die sich in einem anderen Speicherbereich befinden, in dem wir eine größere Karte haben. Der Befehl bewirkt, dass die gesamte Karte rekonstruiert wird, wobei nur die Elemente berücksichtigt werden, die bestimmten X- und Y-Koordinatenbereichen entsprechen (alle Parameter sind 16-Bit-Zahlen).

Verwendung:

|UMAP, <map_ini>, <map_fin>, <y_ini>, <y_fin>, <x_ini>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>, <x_fin>.

Wenn wir zum Beispiel eine Karte an der Adresse 22.000 haben, die 1500 Bytes belegt, und wir wollen die Karte mit den Koordinaten unseres Zeichens aktualisieren, mit genügend Spielraum, um in der Y-Koordinate bis zu 100 Zeilen und in der X-Koordinate bis zu 20 Bytes in alle Richtungen vorzurücken:

|UMAP, 22000, 23500, y-100, y+100, x-20, x+20

Dieser Befehl prüft die Koordinaten der Elemente, die sich auf der Karte an der Adresse 23000 befinden, und wenn sie innerhalb der von uns festgelegten X- und Y-Ränder liegen, werden sie in den Speicherbereich kopiert, den 8BP für den Befehl |MAP2SP verwendet, d. h. er kopiert sie von der Adresse 42040. Es werden jedoch nur diejenigen kopiert, die die Bedingung erfüllen. Da es weniger Elemente gibt, wird der Befehl |MAP2SP schneller ausgeführt, da er lesen und prüfen muss, ob weniger Elemente auf dem Bildschirm vorhanden sind.

WICHTIG: Die zu kopierende Karte darf NICHT die 3 Parameter jeder Karte enthalten:

<Max_high> (das ist ein DW, d. h. 2 Byte)

<max_width> (das ist ein DW, d. h. 2 Byte)

<Anzahl_Einträge> (das ist ein DB, d.h. 1 Byte)

Das heißt, es gibt 5 Bytes, die nicht in der zu kopierenden Karte enthalten sein sollten.

22 Wie man eine Anzeigetafel erstellt

In vielen Spielen ist es interessant, einen Scoreboard-Mechanismus (auch "Hall of fame" genannt) zu haben, der die besten Ergebnisse aus verschiedenen Spielen in geordneter Weise speichert. Dies kann in BASIC mit Hilfe eines Arrays programmiert werden, das die Punktzahl jedes Spiels speichert, aber jedes Mal, wenn wir das Spiel anhalten und RUN ausführen, führt der BASIC-Interpreter intern ein CLEAR aus und alle Werte dieser Tabelle werden gelöscht. Eine Möglichkeit, dies zu vermeiden, besteht darin, den Benutzer daran zu hindern, das Programm zu unterbrechen, indem er zweimal die ESC-Taste drückt und die Firmware-Routine CALL &bb48 verwendet. Eine andere Möglichkeit besteht darin, die Punkte in einer Tabelle im Speicher abzulegen und sie von BASIC aus zu lesen und zu ändern. Dies kann auf viele Arten programmiert werden, hier ein Beispiel. Die folgenden Schritte sind zu beachten:

Fügen Sie in make_all.asm ein Lesen der Datei "score_table.asm" ein.

```
;-----CODIGO-----
;enthält die 8bp-Bibliothek und den Musik-
PlayerWYZ lesen Sie "make_codigo_mygame.asm".
;-----MUSICA-----
read "make_musica_mygame.asm";
enthält die Lieder.
-----GRAFIKEN-----
Dieser Teil enthält Bilder und Animationssequenzen. Lies
"make_graficos_mygame.asm".
"score_table.asm" lesen
```

Als nächstes müssen wir eine solche score_table.asm-Datei mit Beispieldaten erstellen. Ich habe eine mit sumerischen Göttinnennamen und Punktzahlen von 10 bis 1 erstellt. Jeder Name benötigt 8 Zeichen. Etwas sehr Wichtiges ist der **org _end_graph**. Mit diesem Befehl geben wir an, dass die Tabelle nach der Grafik in den folgenden Speicheradressen zusammengesetzt werden soll.

```
org _end_graph
_SCORE_TABLE
db "ISHTAR"
dw 10 db
"ANTU"
Wohnung 9
db "INANNA"
Wohnung 8
db "NIMUG"
Wohnung 7
db "NIMBARA"
Wohnung 6
db "ASTA"
Wohnung 5
db "DAMKINA"
Wohnung 4
db "NEBAT"
Wohnung 3
db "NISABA"
Wohnung 2
db "NINSUB"
Wohnung 1
END_SCORE_TABLE
```

Jetzt kommt der BASIC-Teil: Wir werden mit winape assemblieren und dann schauen wir (mit winape, Option assemble->symbols), in welcher Speicheradresse das end_graph-Tag assembliert wurde. In meinem Fall war es &9685. In unserem BASIC-Programm werden wir die wir berücksichtigen werden (ich speichere sie in der Variablen "dir")

```

190 ' --- Ruhmeshalle
200 DIM pts(11): DIM name$(11):'scores
210 GOSUB 2040: 'Punktetabelle lesen
220 INK 3,7: PEN 3: LOCATE 15,12: PRINT " Ruhmeshalle ": LOCATE 15,13: PRINT
" -----
230 p=1:FOR i=0 TO 9:LOCATE 1,i+14: PEN p :PRINT , name$(i), pts(i) :
p=1+(p MOD 3):NEXT
240 b$=INKEY$:IF b$="" THEN 240 ELSE 250

```

Schauen wir uns die Routine an, die die Punktetabelle in Zeile 2040 ausliest.

```

2040 '--- PUNKTETABELLE LESEN
2050 dir=&9685: FOR i=0 TO 9: name$(i)=""
2070 FOR j=dir TO dir +7: 'Iee Zeichen ein Zeichen      die 8 Buchstaben
2080 Buchstabe=PEEK (j):
name$(i)=name$(i)+CHR$(Buchstabe)
2090 NEXT j: dir=dir+8: 'nach den 8 Buchstaben      Punkte (eine ganze
kommen die Zahl)
2100 pts(i)=0: | PEEK,dir,@pts(i):dir=dir+2: 'eine Ganzzahl ist 2 Bytes
2110 NEXT i
2120 RETURN

```

Schließlich wird jedes Mal, wenn ein Spiel zu Ende ist, geprüft, ob der Punktestand (im Beispiel die Variable "score") höher ist als irgendein Eintrag in der Punktetabelle (Array "pts"), und wenn ja, wird die Tabelle geändert. Indem wir die Tabelle in den Speicheradressen ändern, gehen die Werte nicht verloren, selbst wenn wir RUN.

```

1800 '--- SPIEL BEENDEN & HIGHSCORE ÜBERPRÜFEN ---
1810 TINTE 0,0: RAND 5: TINTE 2,15: TINTE 1,20: | MUSIK
1820 j=10:FOR i=9 TO 0 STEP -1: IF score>pts(i) THEN j=i:NEXT
1830 IF j=10 THEN RUN: 'Spiel beenden & starten
1831 verschiebt alle niedrigeren Punktzahlen um eine Position.
1840 FOR i=8 TO j STEP -1: pts(i+1)=pts(i): name$(i+1)=name$(i): NEXT
1850 b$=INKEY$:IF b$<>"" THEN 1850: 'Puffertastatur reinigen
1860 MODE 1: BORDER 5: INK 3,8: LOCATE 6,8: PEN 3: PRINT "CONGRATULATIONS!
NEUER HIGHSCORE".
1880 ORTUNG          2: DRUCKEN SIE "GEBEN SIE IHREN NAMEN EIN".
14,10:STIFT
1900 ORTUNG          1: INPUT name$(j): name$(j)=MID$(name$(j),1,8)
15,12:STIFT
1910 pts(j)=Punktza
hl
'--- PARTITUR      TABELLE ZUM SPEICHER --
SCHREIBEN
1930 dir=&9685: FOR i=0 TO 9: k=1
1950 FOR j=dir TO dir +7: 'wir schreiben Zeichen für Zeichen, alle 8
Buchstaben
1960 dato$=MID$(name$(i),k,1): IF dato$="" THEN dato$=" "
1970 dato=ASC(dato$)
POKE j,Daten:k=k+1:NEXT j
1990 dir=dir+8: 'wir schreiben die Interpunktions nach dem Namen (8
Buchstaben)

```

2000 | POKE,dir,pts(i)
2010 dir=dir+2:**'der Spielstand ist eine ganze Zahl = 2 Bytes**
2020 NEXT i
2030 LAUFEN

23 Mögliche zukünftige Erweiterungen der Bibliothek

Die 8BP-Bibliothek kann durch Hinzufügen neuer Funktionen verbessert werden, die dem Programmierer neue Möglichkeiten eröffnen könnten. Hier sind einige Vorschläge für eine solche Erweiterung

23.1 Speicher für das Auffinden neuer Funktionen

Durch den Mechanismus der "**Assembler-Optionen**" überlässt Ihnen die Bibliothek derzeit eine von der jeweiligen Option abhängige Menge an freiem Speicher für Ihr BASIC-Listing.

- Option 0: 23,5 KB frei (sollte nur zu Testzwecken verwendet werden)
- Option 1: 25 KB frei (Labyrinth-Spiele)
- Option 2: 24,8 KB frei (Spiele mit Scrollfunktion)
- Option 3: 24 KB frei (Spiele mit Pseudo-3D)

Die Bibliothek könnte noch wachsen, durch eine Option 4, die die Möglichkeiten der Option 1 erweitert, z. B. durch "Verfilmungs"-Fähigkeiten. Diese Option 4 könnte solche Fähigkeiten unter Verwendung von 1 KB bereitstellen und dem Benutzer 24 KB frei lassen.

23.2 Druck in Pixelauflösung

Derzeit verwendet 8BP Byte-Auflösung und Byte-Koordinaten, die 2 Pixel des Modus 0 sind. Eine Möglichkeit, diese Einschränkung zu umgehen, besteht darin, zwei Bilder für dasselbe Sprite zu definieren, die um ein einziges Pixel versetzt sind. Wenn Sie das Sprite auf dem Bildschirm bewegen, können Sie abwechselnd einfach das Bild in das versetzte Bild ändern und das Sprite um ein Byte verschieben. Auf diese Weise erhalten Sie eine pixelgenaue Bewegung. Diese Technik wird in Kapitel 13 ausführlich beschrieben.

23.3 Layout des Modus 1

Das aktuelle Layout arbeitet mit einem Zeichenpuffer von $20 \times 25 = 500$ Bytes.

Es kann ohne Probleme in Modus-1-Spielen verwendet werden, aber es wird Dinge geben, die wir nicht tun können, wie z.B. einen Teil zu definieren, der 3 Zeichen der Modus-1-Breite einnimmt, da Modus-0-Zeichen die doppelte Breite von Modus-1-Zeichen einnehmen. Das ist kein Problem, aber es ist eine Einschränkung.

Ein Modus-1-Layout würde 1 KB belegen, also $40 \times 25 = 1000$. Da das Modus-0-Layout und das Modus-1-Layout nicht gleichzeitig verwendet würden, könnten sie sich im Speicher überlappen. Da das Modus-0-Layout zwischen 42000 und 42500 liegt, würden wir das Modus-1-Layout einfach zwischen 41500 und 42500 platzieren und 500 Byte aus dem 8-KB-Sprite-Speicher "stehlen", der sich zwischen 34000 und 42000 befindet.

Die Änderungen zur Unterstützung dieser Verbesserung sind minimal und betreffen nur zwei Funktionen

Das Layout0/Layout1 und **|LAYOUT** und **|COLAY** sollten sich des Bildschirrmodus bewusst sein, mittels einer Variablen, die als Flag fungiert (Layout0/Layout1). Diese Änderung wäre gut, aber auch ohne sie können wir Layouts in Modus-1-Spielen ohne Probleme verwenden.

23.4 Verfilmungskapazität

Es könnte interessant sein, einen "Filmation"-Modus für die Erstellung von Spielen des Typs "Knight Lore" zu entwickeln. Durch die Nutzung bestehender Funktionen in der Bibliothek könnte diese interessante Fähigkeit mit ein wenig zusätzlichem Code implementiert werden. Es ist möglich, dass diese Fähigkeit bald hinzugefügt werden wird.

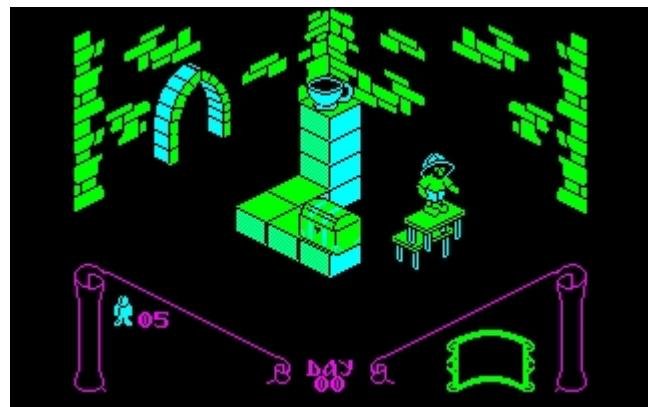


Abb. 120 die mythische "Ritterkunde".

23.5 Hardware-Rollfunktionen

Es gibt nur wenige Amstrad CPC-Spiele mit hochwertigem, flüssigem Scrolling, die mit den Fähigkeiten des M6845 Video-Controller-Chips programmiert wurden. Derzeit verfügt die Bibliothek über einen CPU-basierten Scroll-Mechanismus (nicht hardwarebasiert, aber effizient und vielseitig für Spiele mit Scrolling in jede Bewegungsrichtung).

Die Tatsache, dass es nicht viele Spiele dieser Art gibt, ist darauf zurückzuführen, dass die Videospielprogrammierer in den 1980er Jahren nicht über viele Informationen verfügten und in vielen Fällen Amateure waren.

Zu den wenigen Spielen, die einen flüssigen Bildlauf haben, gehören 2 von Firebird:

- "Mission Genocide" (aus Firebird, 1987, von Paul Shirley, einem ausgezeichneten Programmierer, der auch eine ultraschnelle Überschreibetechnik ohne Verwendung von Masken erfunden hat).
- "Warhawk" (von Firebird, 1987)



Abb. 121 Firebird-Spiele mit schnellem und flüssigem Bildlauf

Die Bildlauftechnik dieser beiden Spiele ist die gleiche, bekannt als "vertikaler Bildlauf". Die Bildlauftechnik besteht darin, genau den Zeitpunkt zu steuern, an dem der Bildlauf stattfindet. In diesem Moment tricksen wir den CTRC 6845 aus, indem wir ihm sagen, dass der Bildschirm früher als normal endet. Vor dem Ende dieses Bildschirmabschnitts weisen wir ihn jedoch an, weniger Scanlines einzubauen, als einem Abschnitt dieser Größe entsprechen. Dann, zu einem sehr genauen Zeitpunkt, den wir auf die Mikrosekunde genau kontrollieren müssen, weisen wir den Chip an, einen

neuen Bildschirm zu beginnen, ohne das vertikale Sync-Signal erzeugt zu haben. So können wir einen zweiten

(z. B. die Markierungen) und kompensieren die Anzahl der Scanlines des ersten Abschnitts. Wenn wir den Mechanismus zur Kompensation der Abtastzeilen richtig hinbekommen, können wir einen der beiden Bildschirmabschnitte außerordentlich flüssig bewegen. Das Problem bei der Übertragung dieser Technik auf einen BASIC-Befehl besteht darin, dass die Steuerung der Unterbrechungen aufgrund der Ausführung des Interpreters ungenau ist, während wir hier eine sehr, sehr präzise Steuerung benötigen.

Das Problem beim Hardwarescrolling (das sich auch auf das Softwarescrolling auswirkt, bei dem der gesamte Bildschirm bewegt wird) besteht darin, dass es die vorhandenen Sprites mit sich "zieht", so dass Sie bei der Neupositionierung eine unerwünschte Vibration bei den Feinden und/oder Ihrem Charakter bemerken werden. Um dieses Problem zu lösen, können Sie die doppelte Pufferung verwenden und jedes Mal, wenn ein Bild fertig ist, zwischen zwei 16KB-Blöcken wechseln. Dadurch wird verhindert, dass Sie sehen können, "wie jedes Bild gemacht wird". In 8BP habe ich die doppelte Pufferung verworfen, um dem Programmierer einen guten RAM-Speicherplatz zu lassen, und deshalb wurden diese Techniken nicht implementiert.

Aus all den oben genannten Gründen basiert das Scrollen in 8BP auf einer Weltkarte, die die Sprites bei der Bewegung nicht mit sich zieht und daher effizienter ist, da sie weniger Speicherplatz benötigt und gleichzeitig multidirektionale Bewegungen ermöglicht.

23.6 Migration der 8BP-Bibliothek auf andere Mikrocomputer

Diese Bibliothek wäre leicht auf andere Z80-basierte Mikrocomputer, wie den Sinclair ZX Spectrum, übertragbar. Im Falle des ZX Spectrum müssten die Routinen zum Malen auf dem Bildschirm neu geschrieben werden, da der Videospeicher anders gehandhabt wird. Die ZX-Migration ist bereits ein festes Projekt, nachdem zahlreiche Anfragen von ZX-Benutzern eingegangen sind.

Die Migration der Bibliothek auf einen Commodore 64 wäre ebenfalls machbar, obwohl der Assemblercode nicht wiederverwendet werden könnte, da er auf einem anderen Mikroprozessor basiert. Im Falle des Commodore 64 sollte die Migration der 8BP-Bibliothek außerdem die Vorteile der maschineneigenen Merkmale wie die 8 Hardware-Sprites nutzen, so dass die 8BP-Bibliothek intern einen Sprite-Multiplexer enthalten sollte, der 32 Sprites bietet, aber intern die 8 Hardware-Sprites verwendet.



Abb. 122 Sinclair ZX und Commodore 64, zwei Klassiker

24 Einige Spiele, die mit 8BP gemacht wurden

In diesem Kapitel beschreibe ich, wie einige Spiele, die Sie im Internet unter <https://github.com/jjaranda13/8BP> finden können, mit 8BP gemacht wurden (von den neuesten bis zu den ältesten):

- **Paco the man:** ein puzzleartiges Spiel, das die Technik der fließenden Bewegung (Halbbyte) und eine fortgeschrittene massive Logik nutzt.
- **NOMWARS:** ein Spiel im "Kommando"-Stil
- **Blaster-Pilot:** ein multidirektionales Scrolling-Spiel, das sich an Spielen wie "Time Pilot" oder "Asteroids" orientiert.
- **Happy Monty:** rasantes Spiel im Stil von Mutant Monty
- **Eridu:** ein klassisches Scramble-Spiel mit horizontalem Scrollen.
- **Space Phantom:** inspiriert von Space Harrier
- **Eternal Frogger:** eine Neuauflage des Klassikers Frogger, präsentiert auf der berühmten "amstrad eternal"-Messe.
- **3D Racing one:** das erste Rennspiel, das Pseudo-3D-Fähigkeiten nutzt
- **Fresh Fruits & vegetables:** ein Plattformspiel mit horizontalem Scrollen und fortschrittlichem Sprite-Pfad-Management.
- **Nibiru:** ein horizontal scrollendes Schiffsspiel, das erweiterte Funktionen von 8BP nutzt
- **Anunnaki:** ein Schiffsspiel, Arcade-Genre
- **Mutante Montoya:** ein Spiel, das auf dem Bildschirm scrollt. Man könnte es als Plattformspiel bezeichnen. Es war das erste Spiel, das ich mit 8BP gemacht habe.
- **Minispiele:** Dies sind kurze, einfache, didaktische Spiele, die Ihnen den Einstieg in die Programmierung mit 8BP erleichtern. Es gibt eine Version des Klassikers "Pong", genannt "Mini-Pong" und eine Version des Klassikers "Space Invaders", genannt "Mini-Invaders".

24.1 Mutant Montoya

Eine erste Hommage an den Amstrad CPC, mit einem Titel, der vom klassischen "Mutant Monty" inspiriert ist.

Es ist ein einfaches Spiel mit 5 Ebenen. Es basiert auf der Verwendung des 8BP-Layouts zum Aufbau jedes Bildschirms.



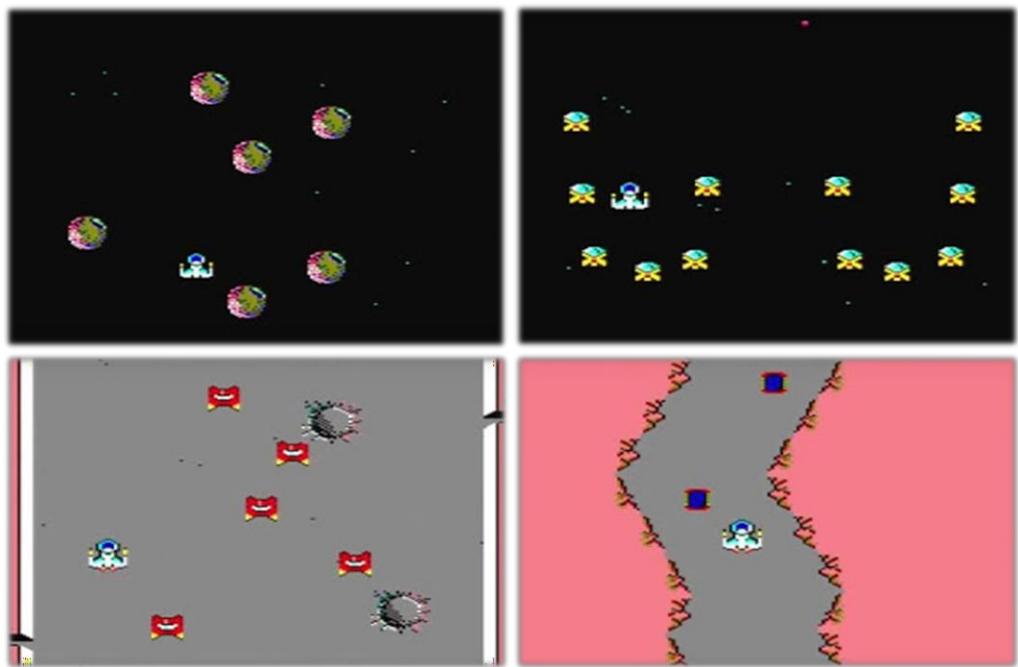


24.2 Anunnaki, unsere außerirdische Vergangenheit

Dies ist ein sehr interessantes Arcade-Videospiel, um es zu analysieren und sich mit der Programmietechnik der "massiven Logik" vertraut zu machen. Als es programmiert wurde, verfügte die 8BP-Bibliothek noch nicht über einen Scroll-Befehl oder Sprite-Routing, weshalb die Programmierung dieses Spiels so interessant ist, da es alles durch massive Logik erreicht.

Im Gegensatz zu "Mutant Montoya" macht das Videospiel "Anunnaki" keinen Gebrauch vom Layout, da es sich um ein Spiel handelt, bei dem es darum geht, voranzukommen und feindliche Schiffe zu zerstören, und nicht um ein Spiel mit Labyrinthen oder vorbeiziehenden Bildschirmen. Dieses Spiel nutzt auch die des "simulierten" Scrollens, sehr interessant.

Sie sind Enki, ein Anunnaki-Befehlshaber, der sich mit außerirdischen Rassen anlegt, um den Planeten Erde zu erobern und so die Menschen ihrem Willen zu unterwerfen. Das Spiel besteht aus 2 Levels. Wenn du ein Leben verlierst, gehst du an dem Punkt im Level weiter, an dem du dich befindest, du gehst nicht zum Anfang des Levels zurück. Der erste Level ist eine Etappe im interstellaren Raum, in der Sie Meteoriten ausweichen und Horden von Schiffen und Raumvögeln töten müssen. Am Ende des Levels müssen Sie einen "Endgegner" zerstören. Der zweite Level findet auf dem Mond statt, wo Sie Horden von Schiffen zerstören müssen. Danach müssen Sie durch einen mit Minen gespickten Tunnel gehen, bis Sie auf drei "Endgegner" treffen, die Sie zerstören müssen.



24.3 Nibiru

Dies ist ein Spiel, das viele der Funktionen von 8BP und der "Massive Logic"-Programmiertechnik testet und Details wie einen ausgefallenen Ladegraphen und drei Melodien im Spiel, sowie eine Punktetabelle, die auch bei einem Neustart des Spiels nicht verloren geht, und andere fortgeschrittene technische Aspekte wie Parallax-Scroll, Routen, Makros usw. enthält. Das BASIC-Listing ist etwas über 16KB groß.

Sie sind der Pilot eines Zerstörerschiffs und müssen den Planeten Nibiru und seinen Anführer "Gorgo", ein fast unbesiegbares uraltes Reptil, besiegen. Sie müssen die galaktischen Vögel zerstören, die auf seinen Monden leben. Sobald Sie den Planeten erreichen, müssen Sie sich seinen Gefahren stellen, bevor Sie Gorgo bekämpfen können.

Das Spiel besteht aus drei Phasen und verwendet den 8BP-Scrollmechanismus, der auf dem MAP2SP-Befehl basiert, sowie Sprite-Routing und Animationsmakros - alles in BASIC! dank 8BP und der Technik der "massiven Logik".



Das Spiel unterhält eine Punktetabelle, die nicht gelöscht wird, auch wenn Sie das Spiel beenden. Dies wird dadurch erreicht, dass sie im RAM mit Pokes gespeichert wird, anstatt sie in BASIC-Variablen zu speichern.

24.4 Frisches Obst und Gemüse

Dies ist ein Plattformspiel, bei dem deine Aufgabe darin besteht, alle Früchte einzusammeln, damit die Bevölkerung nichts mehr zu essen hat und ein armes Schwein opfern muss, um sich zu ernähren.

Die wichtigste Neuerung ist die fortschrittliche Verwendung von Routen, die Verkettung von Routen, um vom "Fallen" zum "Gehen" zu gelangen, und die Verwendung von RINK in Kombination mit MAP2SP als Scrolltechnik.



Die blauen Steine in der Spielpräsentation verwenden synchronisierten Sprite-Druck (PRINTSPALL,0,0,1), während der Sprite-Druck während des Spiels nicht synchronisiert ist (PRINTSPALL,0,0,0). Der Effekt der Synchronisation auf die Darstellung ist, dass alles synchronisiert erscheint, einschließlich der RINK-Tintenanimation (die auf den blauen Steinen verwendet wird). Dadurch entsteht ein flüssiger Bildlaufeffekt. Meiner Meinung nach sollte man ein Spiel nicht synchronisieren, da man dadurch zwar einen flüssigeren Ablauf erhält, aber auch die Geschwindigkeit des Spiels sinkt. Je nachdem, welche Art von Spiel Sie spielen, kann das für Sie interessant sein oder nicht.

24.5 "3D-Rennen eins

Dies ist das erste Spiel, das die Pseudo3D-Fähigkeit von 8BP nutzt. Es hat eine ausgefallene Ladegrafik und 4 Strecken: eine Trainingsstrecke mit Pfützen auf der Straße, eine Strecke, auf der wir mit 4 anderen Autos konkurrieren, eine Strecke, auf der die Geschwindigkeit verdoppelt wird, mit niedrigeren Steigungssegmenten, und eine abschließende Nachtphase. Das Spiel verwendet auch den PRINTAT-Befehl und seinen eigenen Zeichensatz. Darüber hinaus verfügt es über eine Anzeigetafel, einen Hauptmusiktitel, zwei Nebenmusiktitel und Soundeffekte.

Für die Darstellung wird eine 2D-Karte voller Kugeln verwendet und der Befehl MAP2SP mit Überschreiben konfiguriert. Die Kugeln sind "Zoom-Bilder".

Die erste Schaltung wurde mit einer Datei erstellt, in der wir nacheinander alle Elemente (Segmente, Schilder, Bäume, Pfützen...) platziert haben, aber der Rest der Schaltungen wird dynamisch von BASIC erstellt, indem wir die Adresse der Weltkarte

eingeben.

Um Kollisionen und das Verlassen der Straße zu erkennen, wird der Befehl PEEK anstelle von COLSPALL verwendet. Sobald also ein Byte mit einer anderen Farbe als die der Straße auf dem Fahrzeug erkannt wird, wird davon ausgegangen, dass eine Kollision stattgefunden hat und der Motor beschädigt wurde.

Eine weitere interessante Neuerung ist die Verwendung von dynamischen Strecken. Sowohl die Wettkampfstrecke als auch die Auto-Routen werden in BASIC nach dem in diesem Handbuch beschriebenen Verfahren erstellt.



24.6 Weltraum-Phantom

Dies ist ein Spiel, das mit der Version v35 der Bibliothek erstellt wurde. Es nutzt die Pseudo-3D-Fähigkeiten für die Präsentation der Titel im "Star Wars"-Stil. Es verwendet auch transparenten Druck mit Sprites (Münzen), die hinter dem Hintergrund (der Anzeigetafel) vorbeiziehen.

Das Spiel ist vom Klassiker "Space Harrier" inspiriert, und Sie steuern einen Helden, der mit einem Jet-Pack ausgestattet ist und durch den Weltraum fliegt, um Meteore, UFOs, Weltraumvögel und einen Drachen als Endgegner zu töten. Es besteht aus drei Etappen und einem epischen Finale.

Der Zeichensatz ist hausintern, obwohl nur die Zahlen definiert wurden, im Stil einer "Casio-Uhr" für die Marker.

In der ersten Phase werden Routen für die Sterne, die Sprites, sowie für die Meteore und Vögel verwendet.

Das zweite verwendet Sprite-Überschreibung mit 2-Bit-Hintergrund (4-farbig) und Tintenanimation mit RINK. Die Schiffe in einer Reihe werden mit dem verbesserten ROUTESP-Befehl, der in V35 verfügbar ist, platziert.

Obwohl das Spiel 3 Dimensionen simuliert, verwendet es keine Pseudo-3D-Projektion, sondern Sprite-Pfade, bei denen die Version des Gegners in eine größere Version geändert wird, um den Eindruck zu vermitteln, dass er sich nähert. Ein Zusammenstoß

mit einem entfernten Feind hat also keine Auswirkungen auf uns,

ein unbenutztes Flaggenstatusregister wird verwendet, um entfernte Feinde als harmlos zu kennzeichnen.

In der dritten Phase wurde die relative horizontale Bewegung der Steine auf dem Boden in Kombination mit einer beschleunigten vertikalen Bewegungsbahn genutzt.



24.7 Ewiger Frogger

Das Spiel "Frogger Eternal" wurde mit der V36-Version von 8BP erstellt und sein Titel erinnert sowohl an den 1981 erschienenen Klassiker "Frogger" von Konami als auch an die 2019 stattfindende Messe "Amstrad Eternal", auf der dieses Spiel erstmals vorgestellt wurde.

Es handelt sich um ein Spiel, das im Modus 1 programmiert wurde, mit LAYOUT, transparentem Druck auf dem Frosch und Routen verschiedener Art für die Sprites. Einige der Sprites sind unsichtbar, aber kollisionsfähig, wie z.B. 4 "unsichtbare" Flüsse unter Baumstämmen, Seerosen und Schildkröten, über die der Frosch springen muss, wie "unsichtbare Wände" an den Seiten des Flusses, damit der Frosch nicht entkommen kann.



24.8 Eridu: Der Raumhafen

Dieses Spiel ist ein Klon von Konamis Klassiker "Scramble" aus dem Jahr 1981. Es hat viele Unterschiede zum Original, aber im Wesentlichen ist es vom klassischen Spiel inspiriert, da es die Notwendigkeit des Auftankens einbezieht und den Spieler zwingt, Risiken einzugehen, um die Treibstofftanks zu zerstören, um kein Leben zu verlieren. Es ist recht schnell, obwohl es mit einem leistungsstarken Scroller läuft, der an vielen Stellen 32 Sprites auf dem Bildschirm anzeigt. Es erreicht bis zu 18 fps.

Das Spiel hat 5 Etappen, verschiedene Musiken im Spiel und eine sehr ausgefallene Präsentationsgrafik.

Eridu ist ein Videospiel, das an eine alte, "verbotene" Geschichte der Menschheit anknüpft. Eridu war die erste Stadt der Welt, die vor 400.000 Jahren von den "Anunnaki" gegründet wurde, einer außerirdischen Rasse, die laut sumerischen Tafeln in der Wüste des Irak gefunden wurde. Dort errichteten sie einen Raumhafen namens "Erde 1".



Die Karten der verschiedenen Phasen werden in verschiedene Speicherbänke geladen, die jeweils 500 Byte an Welddaten und 200 Byte für die Beschreibung der Feindorte belegen. Das Scrollen wird logischerweise mit dem Befehl |MAP2SP durchgeführt.

24.9 Glücklicher Monty

Es handelt sich um ein rasantes Bildschirmwischspiel, das den Klassiker Mutant Monty fast perfekt imitiert. Es "klont" sogar seinen Startbildschirm. Dieses Spiel wurde mit Version 37 der Bibliothek erstellt und erreicht 25 FPS.

Es macht intensiven Gebrauch vom Layout und natürlich von massiven Logiken. Es kann 25 Ebenen speichern und verwendet dabei eine einfache Technik zur Verdichtung der Layouts (jedes Layout benötigt nur 160 Bytes), die in diesem Buch erläutert wird.



Eine originelle Technik, die in diesem Spiel verwendet wird, ermöglicht es Feinden, ihren Weg zu ändern. Dies sind unsichtbare "Umkehrer"-Sprites. Wenn ein Feind mit einem Inverter-Sprite kollidiert, ändert er seinen Pfad und bekommt den entgegengesetzten Pfad oder sogar einen senkrechten Pfad zugewiesen. So können Pfade beliebiger Länge erstellt werden, ohne mehr als einen vertikalen und einen horizontalen Pfad zu definieren. Es können sogar Schleifen gebildet werden.

Dies vereinfacht auch die Erstellung der Karte (Layout + Feinde) eines jeden Levels, da es bei der Lokalisierung eines Feindes ausreicht, einen Code (ein Zeichen) zu verwenden, der die Geschwindigkeit und die Richtung des Feindes angibt (6 Buchstaben werden für alle Arten von Feinden verwendet und ändern regelmäßig die "Palette" der Feinde, die Puppen, die diesen 6 Buchstaben zugeordnet sind).

24.10 Blaster-Pilot

Es handelt sich um ein Spiel, das mit der Version v39 von 8BP erstellt wurde, in dem es möglich ist, Soundeffekte (erstellt mit SOUND) gleichzeitig mit der Musik abzuspielen, die mit dem Befehl MUSIC abgespielt wird. In diesem Fall wird der SOUND-Befehl für Schüsse und Explosionen verwendet und nutzt den dritten Kanal, während die Musik die ersten beiden Kanäle nutzt.



Das Spiel hat einen multidirektionalen Bildlauf und ist vom Stil von Spielen wie "Time Pilot" oder "Asteroids" inspiriert. Neben der Anzeige von Punktestand und Leben gibt es ein Radar, mit dem man sich im Weltraum orientieren kann, um 3 verirrte Schauspielernauten zu finden, die man retten muss. Das Spiel ist insofern interessant, als die Programmierung der Schiffsbewegung in 12 mögliche Richtungen so effizient wie möglich gehandhabt wird.

Zusätzlich zu den 6 Stufen gibt es am Ende jeder Stufe eine Bonusphase, in der Sie Punkte sammeln und ein Leben zurückgewinnen können.

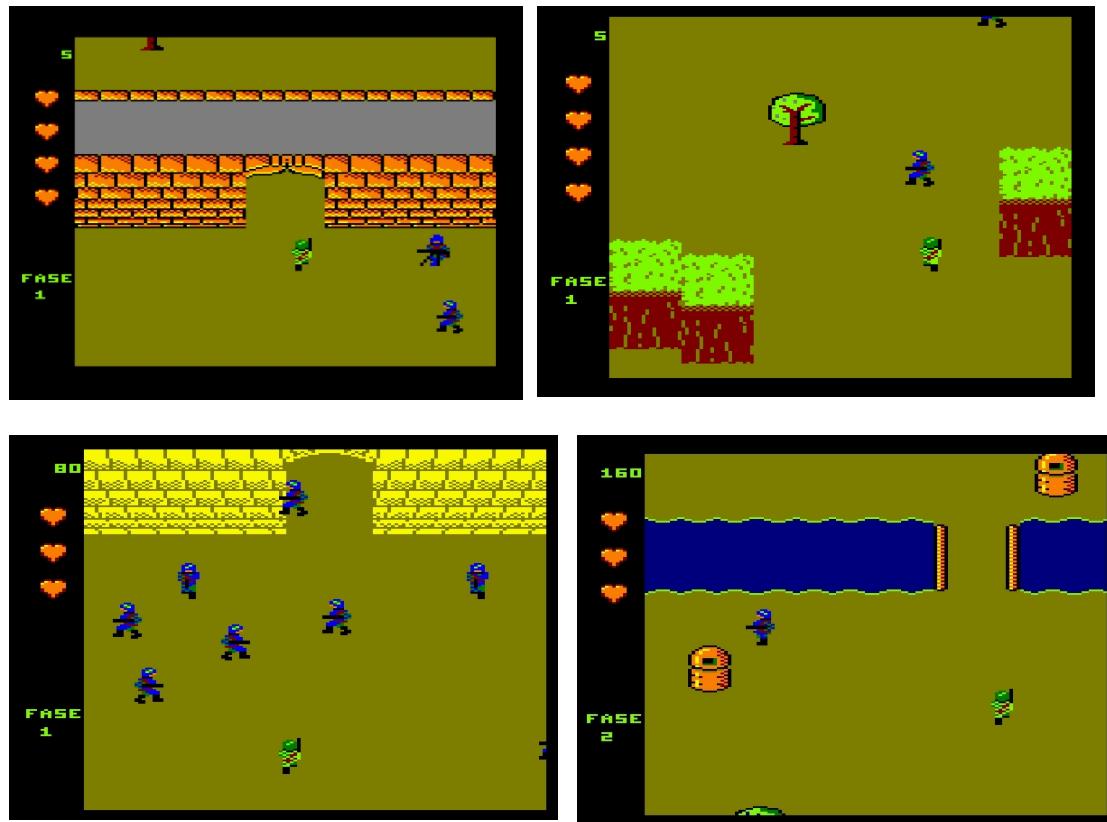
24.11 NOMWARS

Es ist ein Spiel im Stil des klassischen "Commando", das 1985 von Capcom entwickelt wurde. Ein klassisches vertikales Scroll-Shoot 'em up.

Das Spiel besteht aus 4 Etappen und einem "ausgefallenen" Intro, mit einer Geschichte über den Krieg der neuen Weltordnung. Es ist im DES-Format veröffentlicht worden.

Diese Version nutzt die Scroll-Fähigkeit von 8BP (MAP2SP-Befehl) und enthält die berühmte Brücke, unter der Joe hindurchfährt, mit einer Technik, die auf dem SETLIMITS-Befehl von 8BP basiert.

Das Spiel wird in zwei Versionen angeboten: die reine BASIC-Version und die kompilierte Cycle-Version (Cycle übersetzt in die Sprache C unter Verwendung des 8BP-Wrappers und des 8BP-Minibasic). Beide Versionen sind identisch, da die C-Übersetzung eine totale Replik, fast ein Spiegel der BASIC-Version ist. Das Spiel wurde in BASIC programmiert und am letzten Tag mit Hilfe der 8BP-Funktion in C konvertiert.

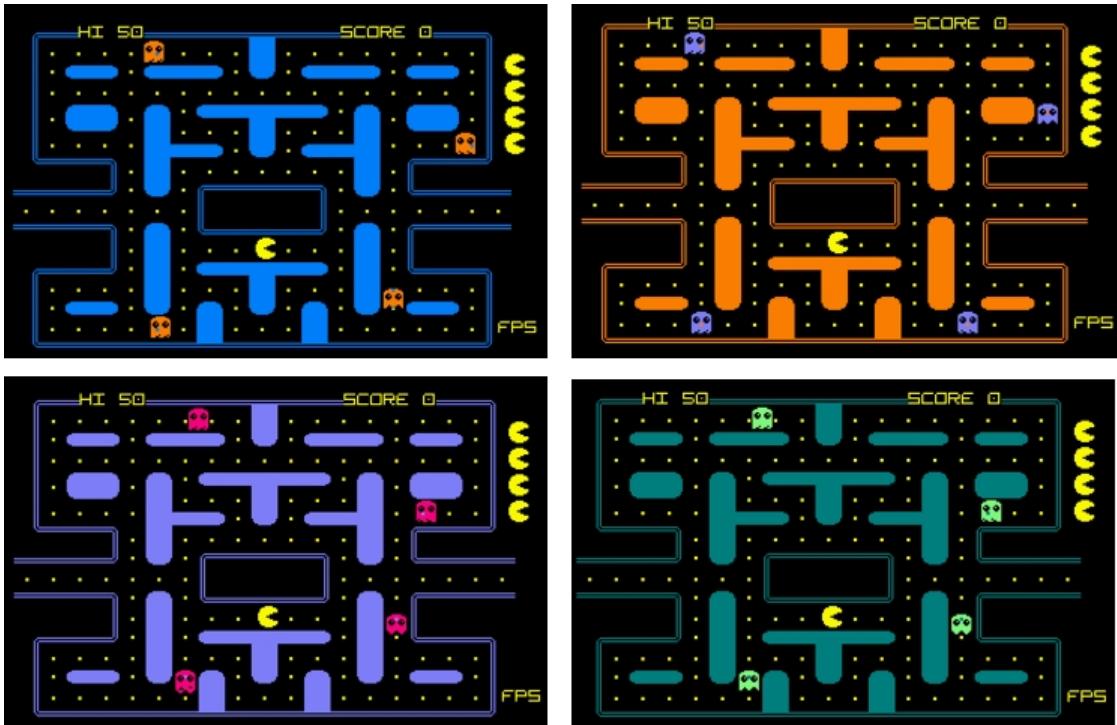


24.12 Paco, der Mann

Ein Spiel im reinsten Pac-Man-Stil. Das Spiel enthält zwei Phasen in BASIC und zwei mit kompiliertem Zyklus. In BASIC erreicht es 19 FPS mit Labyrinth, Kollisionen und 4 Geisterlogik und in C erreicht es 33 fps.



Jeder Paco-Level ist durch seine Farben gekennzeichnet. Die ersten beiden laufen in BASIC und verwenden "vorberechnete" Geisterentscheidungen, um 19 fps zu erreichen. Die dritte und vierte Stufe verwenden C und erreichen 33 fps.



24.13 Mini-Spiele

Diese Spiele sind für Bildungszwecke gedacht. Einfach zu verstehen und kurz, um den Programmieranfängern bei der Entwicklung ihrer eigenen Spiele zu helfen.

24.13.1 Mini-Pong

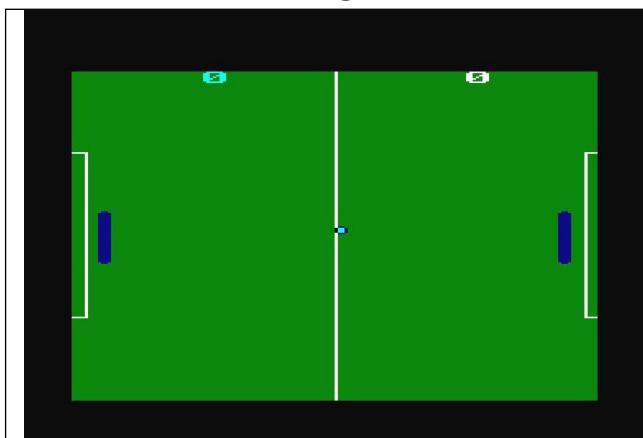


Abb. 123 Videospiel "Mini-Pong"

Es ist ein sehr einfaches und didaktisches Videospiel. Es basiert auf dem Klassiker "Pong" von Atari (1972).

Der gegnerische Balken (der Computer) fängt an, Entscheidungen zu treffen, wenn der Ball über die Hälfte des Spielfeldes geht, also ist es möglich, ihn zu schlagen. Wenn wir ihn früher Entscheidungen treffen lassen, ist es irgendwann unmöglich zu gewinnen.

Ein paar kurze Fakten zum Spiel:

- Verwendung von **|COLSPALL**: um Kollisionen zwischen dem "Collider" (dem Ball) und den "Colliders" (den Stäben) zu erkennen. Im Statusbyte der Sprites ist das Collider-Flag für den Ball (Sprite 29) und das Collider-Flag für 31 (unser Balken) und 30 (der gegnerische Balken) aktiviert.
- Verwenden Sie das Überschreiben auf dem Ball, um den weißen Streifen auf dem Spielfeld zu respektieren und ihn beim Passen nicht zu löschen. Verwenden Sie dazu eine Palette mit Überschreiben und aktivieren Sie das Überschreiben-Flag im Statusbyte des Ball-Sprites (das 29.).
- es gibt nur zwei Bilder (der Ball=17 und der Balken=16), die den 2 Sprites zugeordnet sind (Sprite 30 und 31 haben Bild 16 und Sprite 29 hat Bild 17).
- Die Sprites bewegen sich automatisch. Dazu haben sie das automatische Bewegungsflag aktiviert und der Befehl **|AUTOALL** bewegt sie (ändert ihre Koordinaten) entsprechend ihrer Geschwindigkeit.
- Alle Sprites werden mit **|PRINTSPALL** in jedem Spielzyklus gedruckt.

24.13.2 Mini-Invaders

Wie "Mini-Pong" ist dies ein Spiel für pädagogische Zwecke, inspiriert vom Klassiker "Space Invaders" von Taito (1978).



Abb. 124 Videospiel "Mini-Invasoren".

Ein paar Hinweise, wie das geht:

- Das Spiel verwendet 32 Sprites
- Das Schiff ist Sprite 31
- Die Schüsse, die Sie mit dem Schiff abgeben können, sind 29 und 30.
- Invasoren schießen mit Sprite 28
- Die Angreifer verwenden die Sprites 0 bis 27 (insgesamt 28 Angreifer).
- Die Sprites 31, 30 und 29 haben ein aktives Collider-Flag.
- Die übrigen Sprites sind "kollidiert" und haben eine aktive Kollisionsmarkierung.
- Eindringlinge haben eine aktive automatische Bewegungsflagge und sind mit der Route "0" verbunden, die sie von rechts nach links und nach unten bewegt, typisch für Eindringlinge.
- Die Auslöser für Schiffe und Eindringlinge nutzen eine Funktion des V27, sie durchlaufen den Bildschirm und werden am Ende ihres Weges automatisch mit einem definierten Zustandswechsel deaktiviert, was die BASIC-Logik vereinfacht und damit das Spiel beschleunigt.

25 APPENDIX I: Organisation des Videospeichers

25.1 Das menschliche Auge und die Auflösung des CPC

Der Videospeicher des Amstrad CPC hat 3 Betriebsmodi. Der am häufigsten verwendete Modus für Spiele ist Modus 0 (160x200), weil er mehr Farben bietet, aber auch Modus 1 (320x200) wurde häufig für Programmierspiele verwendet. Modus 2 (640x200) wurde aufgrund seiner begrenzten 2 Farben selten oder nie für Spiele verwendet.

Da die Menge des Videospeichers gleich ist, wird die Auflösung geopfert, um die Anzahl der Farben zu erhöhen, aber seltsamerweise ist die horizontale Auflösung, die die längere Seite des Bildschirms ist, geringer als die vertikale (160 horizontal und 200 vertikal). Sie werden sich fragen, warum. Im Übrigen ist dies nicht der einzige Mikrocomputer, der dies getan hat, viele andere Computer haben die gleiche Strategie für die horizontale Seite verwendet.

Das hat mit der Funktionsweise des menschlichen Sehsystems zu tun. Das Auge nimmt vertikal mehr Details wahr als horizontal, so dass eine "Beschädigung" der Auflösung auf der horizontalen Achse nicht so gravierend ist wie eine Beschädigung auf der vertikalen Achse. Subjektiv ist das Ergebnis akzeptabler. Das menschliche Sehsystem ist wie der Modus 0: sehr breite Pixel. Aus diesem Grund ist das horizontale Sichtfeld größer als das vertikale.

25.2 Video-Speicher

Die vollständigsten und klarsten Informationen finden Sie im Amstrad-Firmware-Handbuch. Diese Informationen werden nützlich sein, wenn Sie einen verbesserten Sprite-Editor bauen wollen oder in Assembler einsteigen und Drucküberschreibroutinen oder irgendetwas anderes programmieren wollen.

25.2.1 Modus 2

Im Modus 2 wird jedes Pixel durch ein Bit dargestellt. Ein Byte steht also für 8 Pixel. Wenn wir ein beliebiges Byte aus dem Videospeicher nehmen, ist seine Entsprechung mit den Pixeln 1 Bit für jedes Pixel, in dieser Tabelle werden die Bits dargestellt und welchen Pixeln (p) sie entsprechen.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

In einem Byte ist Bit 7 als das ganz linke Bit nummeriert. Pixel 0 ist auch das ganz linke Pixel, d.h. hier steht nichts "auf dem Kopf". Alles ist richtig

25.2.2 Modus 1

Im Modus 1 haben wir 4 Farben (dargestellt durch 2 Bits). Ein Byte steht also für 4 Pixel. Die Entsprechung zwischen Pixeln und Bits ist etwas komplexer. Pixel 0 wird zum Beispiel mit den Bits 7 und 3 kodiert.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p2(0)	p3(0)	p0(1)	p1(1)	p2(1)	p3(1)

25.2.3 Modus 0

Hier herrscht ein ziemliches Durcheinander. Jedes Byte stellt nur zwei Pixel dar, die mit den Bits des Bytes wie folgt korrespondieren: Pixel 0 wird mit den Bits 7,5,3 und 1 kodiert.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p0(2)	p1(2)	p0(1)	p1(1)	p0(3)	p1(3)

Das folgende Bild soll Ihnen dies verdeutlichen:

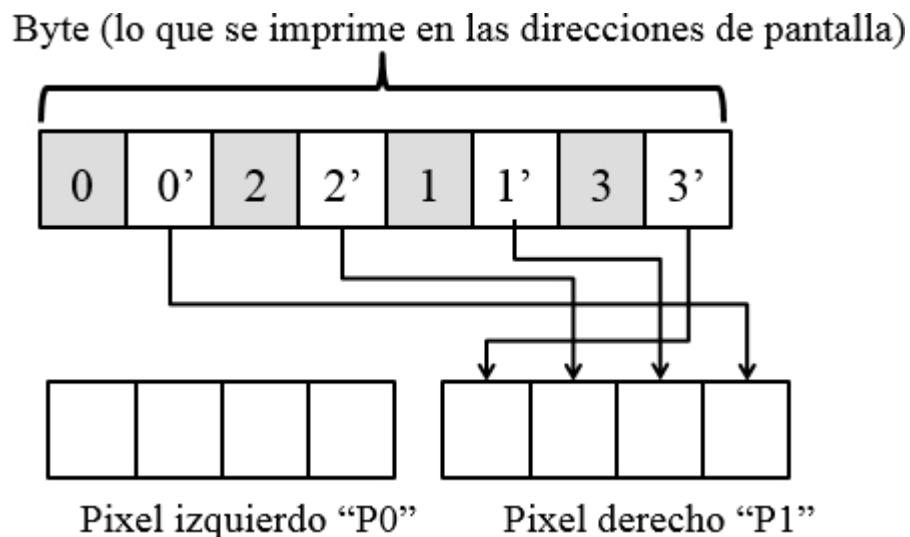


Abb. 125 Pixel und Bits im Modus 0

Ich kann Ihnen nicht sagen, was der obskure Grund dafür ist, dass der Speicher so organisiert ist, aber ich vermute, die Ursache liegt im GATE ARRAY, dem Chip, der diese Bits in ein Videosignal umwandelt. Ich kann mir vorstellen, dass der Designer mit diesem verdrehten Design die Schaltkreise reduziert hat.

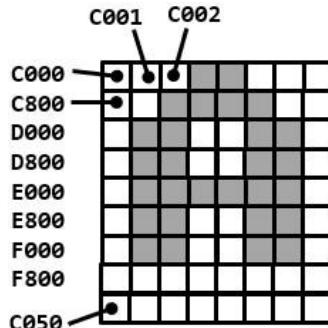
25.2.4 Speicher anzeigen

Die zur selben Zeile gehörenden Bildpunkte werden in Bytes kodiert, die ebenfalls zusammenhängend sind. Allerdings gibt es Sprünge von einer Zeile zur anderen.

Wenn wir uns in Speicheradressen vorwärts bewegen, springen wir, wenn wir das Ende einer Zeile erreichen, zu einer Zeile, die 8 Zeilen weiter unten liegt. Und wenn wir in der nächsten Zeile fortfahren wollen, müssen wir in den Speicheradressen bis zu 2048 Positionen springen.

Die folgende Tabelle zeigt den Videospeicher. Auf der linken Seite steht die Zeile mit den Zeichen (von 1 bis 25) und für jede Zeile die Anfangsadresse jeder der 8 Scanlines, aus denen sie besteht (benannt als ROW0 ...ROW7).

CHARACTER LINE	R0W0	R0W1	R0W2	R0W3	R0W4	R0W5	R0W6	R0W7
1	C000	C800	D000	D800	E000	E800	F000	F800
2	C050	C850	D050	D850	E050	E850	F050	F850
3	C0A0	C8A0	D0A0	D8A0	E0A0	E8A0	F0A0	F8A0
4	C0F0	C8F0	D0F0	D8F0	E0F0	E8F0	F0F0	F8F0
5	C140	C940	D140	D940	E140	E940	F140	F940
6	C190	C990	D190	D990	E190	E990	F190	F990
7	C1E0	C9E0	D1E0	D9E0	E1E0	E9E0	F1E0	F9E0
8	C230	CA30	D230	DA30	E230	EA30	F230	FA30
9	C280	CA80	D280	DA80	E280	EA80	F280	FA80
10	C2D0	CAD0	D2D0	DAD0	E2D0	EAD0	F2D0	FAD0
11	C320	CB20	D320	DB20	E320	EB20	F320	FB20
12	C370	CB70	D370	DB70	E370	EB70	F370	FB70
13	C3C0	CBC0	D3C0	DBC0	E3C0	EBC0	F3C0	FBC0
14	C410	CC10	D410	DC10	E410	EC10	F410	FC10
15	C460	CC60	D460	DC60	E460	EC60	F460	FC60
16	C4B0	CCB0	D4B0	DCB0	E4B0	ECB0	F4B0	FCB0
17	C500	CD00	D500	DD00	E500	ED00	F500	FD00
18	C550	CD50	D550	DD50	E550	ED50	F550	FD50
19	C5A0	CDA0	D5A0	DDA0	E5A0	EDA0	F5A0	FDA0
20	C5F0	CDF0	D5F0	DDF0	E5F0	ED50	F5F0	FD50
21	C640	CE40	D640	DE40	E640	EE40	F640	FE40
22	C690	CE90	D690	DE90	E690	EE90	F690	FE90
23	C6E0	CEE0	D6E0	DEE0	E6E0	EEE0	F6E0	FEE0
24	C730	CF30	D730	DF30	E730	EF30	F730	FF30
25	C780	CF80	D780	DF80	E780	EF80	F780	FF80
spare start	C7D0	CFD0	D7D0	DFD0	E7D0	EFD0	F7D0	FFD0
spare end	C7FF	CFFF	D7FF	FFFF	E7FF	FFFF	F7FF	FFF



*Direcciones de La
esquina superior
izquierda de La
pantalla*

C000 = comienzo de pantalla
= 49152 , es decir 48KB

FFFF= fin de pantalla
= 65535

La pantalla mide:
65535 – 49152 = 16384 =16KB

Abb. 126 Bildschirm-Speicherplan

Der Amstrad-Bildschirm ist 200 Zeilen x 80 Bytes breit, also beträgt der angezeigte Bildschirmspeicher $200 \times 80 = 16.000$ Bytes. Der Videospeicher ist jedoch 16384 Bytes groß. Es gibt 384 Bytes, die in 8 Segmenten von je 48 Bytes "versteckt" sind, die nicht auf dem Bildschirm angezeigt werden, obwohl sie Teil des Videospeichers sind. Diese 8 Segmente werden in der obigen Tabelle als "Spares" bezeichnet. Jedes Segment ist 48 Bytes lang, weil man, wie gesagt, 2048 Bytes hinzufügen muss, um von einer Zeile zur nächsten zu springen, aber in Wirklichkeit belegen die 25 zusammenhängenden Speicherzeilen, die sie trennen, nur 25×80 Bytes = 2000 Bytes.

**Von &C7D0 bis C7FF beide einschließlich Von
&CFD0 bis CFFF beide einschließlich Von &D7D0
bis D7FF beide einschließlich Von &DFD0 bis
DFFF beide einschließlich Von &E7D0 bis E7FF
beide einschließlich Von &EFD0 bis EFFF beide
einschließlich Von &F7D0 bis F7FF beide
einschließlich Von &FFD0 bis F7FF beide
einschließlich Von &FFD0 bis FFFF beide
einschließlich Von &FFD0 bis FFFF beide
einschließlich**

Sie können dies überprüfen, indem Sie mit POKE auf diese Speicheradressen zugreifen, und Sie werden sehen, dass Sie den Inhalt des Bildschirms nicht verändern werden.

Es ist verlockend, diese "versteckten" Speicherbereiche zum Speichern kleiner Assemblerroutinen oder Variablen zu verwenden. Dies ist jedoch gefährlich, da ein

MODE-Befehl, der von BASIC aus ausgeführt wird, löscht diese Speichersegmente vollständig, daher sollten Sie sich dessen bewusst sein, wenn Sie ihn verwenden. In der 8BP-Bibliothek werden diese Segmente verwendet, um lokale Variablen einiger Funktionen zu speichern, deren Wert sicher gelöscht werden kann.

25.3 Berechnung einer Bildschirmadresse

Wenn Sie die Speicheradresse wissen wollen, der bestimmte 8BP-Koordinaten entsprechen, müssen Sie folgende Operation durchführen

$$\text{Dir} = \&C000 + \text{INT}(y/8)*80 + (y \bmod 8)*2048 + x$$

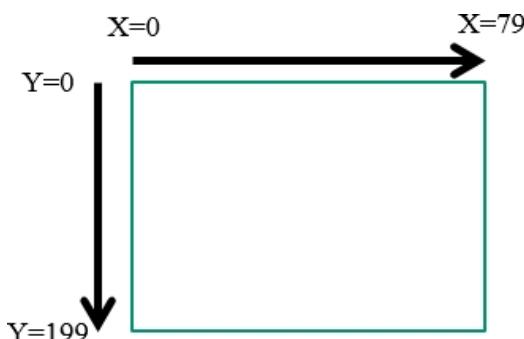


Abb. 127 Bildschirmkoordinaten in 8BP

Dies ist sehr nützlich, wenn Sie z. B. PEEK verwenden möchten, um herauszufinden, ob sich ein bestimmtes Element oder eine bestimmte Farbe in einem bestimmten Byte auf dem Bildschirm befindet, und dies als Mechanismus zur Kollisionserkennung verwenden möchten. Diese Technik wird in dem Videospiel "**3D-Racing One**" verwendet.

Wenn Sie die Richtung einiger grafischer Koordinaten wissen wollen (die vom BASIC-Befehl PLOT verwendet werden), müssen Sie zuerst $y2=(200-y)/2$, $x2=x/8$

$$\text{Dir} = \&C000 + \text{INT}(y2/8)*80 + (y2 \bmod 8)*2048 + x2$$

25.4 Bildschirmabtastungen

Der Amstrad erzeugt 50 Bilder pro Sekunde. Das bedeutet, dass etwa alle 20ms ein neuer Bildschirmscan erzeugt werden muss.

Man könnte meinen, dass der Screen-Swipe, also das Malen des Bildschirms, nur einen Bruchteil dieser 20ms verbraucht, aber das stimmt nicht. Für das Malen des Bildschirms benötigt der Amstrad die vollen 20 ms. Selbst wenn Sie also Ihren Sprite-Ausdruck mit dem Screen-Swipe synchronisieren, ist es sehr wahrscheinlich, dass er Sie erwischt, was zu zwei bekannten Effekten führt:

- **Das Flackern** tritt auf, wenn Sie das Sprite löschen, bevor Sie es an seiner neuen Position drucken. Um dies zu vermeiden, gibt es eine sehr einfache Lösung: Löschen Sie es nicht. Lassen Sie das Sprite einfach seine eigene Spur löschen, wobei ein Rand auf dem Sprite verbleibt, um diese Funktion zu erfüllen. Das Sprite ist größer, aber es flackert nicht, selbst wenn es in der Mitte hängen bleibt, weil es nicht verschwindet.

- **Tearing:** tritt auf, wenn wir vom Sweep in der Mitte des Sprites erwischt werden. Die Hälfte wird mit der neuen Position (Kopf und Rumpf) gedruckt und die andere Hälfte nicht.

Zeit gibt (die Beine). Dann wird das Sprite "falsch" gedruckt, obwohl es im nächsten Frame korrigiert wird, aber für einen Moment ist es, als ob es verformt oder gebrochen ist. Tearing ist ein schlechter Effekt, aber viel akzeptabler als Flackern. Die perfekte Lösung besteht darin, jede Millisekunde zu kontrollieren, wo das Flackern auftritt, um jedes Sprite zu drucken, ohne dass es uns einholt.

Eine typische Empfehlung ist, Sprites von unten nach oben zu drucken, um diese Effekte zu minimieren. Auf diese Weise ist es möglich, den Scan nur einmal auf einem der Sprites zu erhalten, während man beim Drucken von oben nach unten den Scan auf mehreren Sprites erhalten kann, da beide (CPU und Kathodenstrahl) in dieselbe Richtung arbeiten. Leider ist das Interessanteste, was man tun kann, von oben nach unten zu sortieren, um den Sprites einen Tiefeneffekt zu geben (nützlich in bestimmten Spielen wie "Golden axe", "Double dragon", "Renegade", usw.).

Die vom Bildschirm verbrauchten Zeiten sind wie folgt. Beachten Sie, dass Sie ab dem Zeitpunkt der Unterbrechung des Sweeps 3,5 ms Zeit haben, um zu malen, ohne dass Sie abgefangen werden können. In dieser Zeit können Sie jedoch höchstens 2 kleine Sprites drucken.

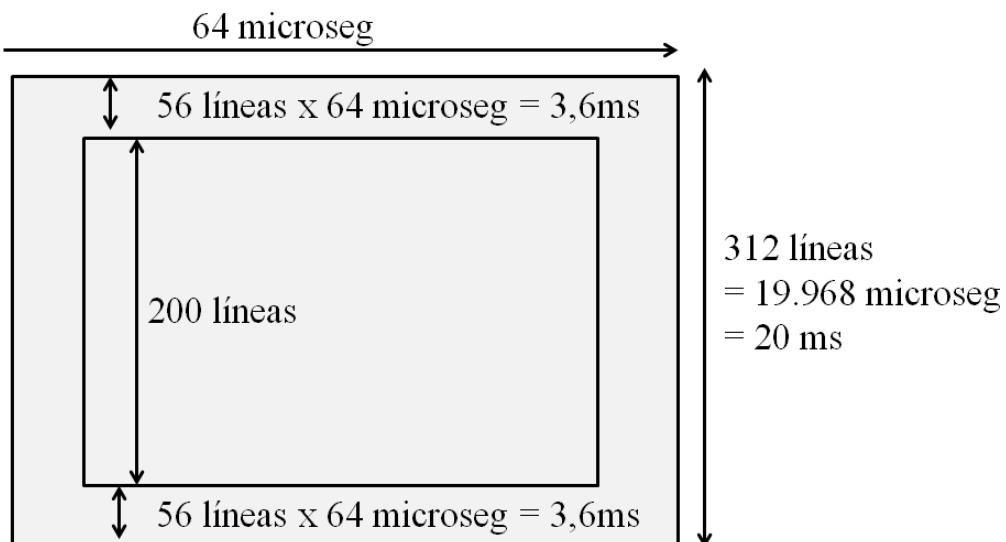


Abb. 128 Mal bei einem Bildschirmdurchlauf

25.5 Wie man einen Ladebildschirm für sein Spiel erstellt

Es gibt viele Möglichkeiten, dies zu tun. Eine sehr einfache ist, eine Grafik mit dem Programm SPEDIT zu erstellen, das so modifiziert wurde, dass man über den ganzen Bildschirm malen kann, ohne Menüs anzuzeigen, und am Ende eine Taste zu drücken, um einen SAVE-Befehl wie diesen zu starten

SAVE "mipantalla.bin", b, &C000, 16384

Wie Sie sehen können, speichert der Befehl 16KB ab der Startadresse des Bildschirms, die &C000 ist.

Der Weg zum Laden wäre

LOAD "mipantalla.bin", &C000

Wenn Sie nicht die Standardpalette verwenden, müssen Sie vor dem Laden des Bildschirms zunächst die INK-Befehle ausführen, die der von Ihnen verwendeten

Palette entsprechen. Beim Laden des Bildschirms sehen Sie, wie der Bildschirm beim Laden langsam auf den Bildschirm gezeichnet wird, denn genau dort, im Videospeicher, wird er geladen.

Eine andere Möglichkeit ist, ein gut ausgearbeitetes Layout zu erstellen und es mit dem obigen Befehl SAVE zu speichern. Es geht darum, eine Zeichnung zu erstellen, und wie Sie sehen können, gibt es viele Möglichkeiten.

Schließlich können Sie ein Tool wie ConvImgCPC (es gibt auch andere) verwenden, einen Bildkonverter/-bearbeiter, der unter Windows funktioniert. Mit diesem Tool können Sie ein beliebiges Bild (z. B. den Scan einer Ihrer Zeichnungen) in eine für den CPC geeignete Binärdatei (mit der Erweiterung .scr) umwandeln. Mit diesem Tool können Sie das Bild auch Pixel für Pixel bearbeiten und retuschiieren, bis es perfekt ist. Sie können es sogar von Grund auf bearbeiten, ohne etwas zu scannen. Meiner Meinung nach ist dies die beste Option.

Um diese Datei auf einem Datenträger zu speichern (in einer .dsk-Datei), müssen Sie CPCDiskXP verwenden, ein weiteres Tool, mit dem Sie Dateien in .dsk-Dateien speichern können.

Einmal in der .dsk-Datei, können Sie sie mit LOAD "mipantalla.bin",&C000 laden. Die Farben werden jedoch nicht richtig aussehen, da ConvImgCPC die Palette so anpasst, dass sie den Originalfarben so nahe wie möglich kommt. Um sie richtig zu sehen, müssen Sie die Routine ausführen, in der ConvImg die Palettenänderungsroutine platziert, also CALL &C7D0.

Diese Routine ist im ersten der 8 versteckten Segmente des Videospeichers "versteckt", so dass das .scr-Bild nicht mehr Platz einnimmt, weil es diese Routine enthält. Das Schlimme daran ist, dass man die Routine nicht ausführen kann, solange der Bildschirm nicht geladen ist, und deshalb wird man sehen, wie das Bild mit falschen Farben geladen wird und am Ende kann man diesen Aufruf aufrufen und die Farben ändern. Was Sie tun können, ist, eine spezielle Palettendatei vorzubereiten. Tun Sie dies:

**Lade "image.scr", &c000
Speichern von "palette.bin", b, &c7d0, 48, &c7d0**

Jetzt haben Sie eine 48-Byte-Datei, die die Palette enthält. In Ihrem Game Loader würden Sie dies tun:

**Lade "!palette.bin"
Aufruf &c7d0
Lade " !image.scr", &c000**

Ich empfehle Ihnen, einfach einige INK-Befehle vor dem LOAD-Befehl zu platzieren, der das Bild lädt.

Lade "image.scr", &c000

Und kurz danach, bevor 8BP zum Drucken von Sprites usw. verwendet wird, ist es ratsam, das versteckte Segment zu löschen, in dem Convimg die Routine verlässt, denn es ist ein Bereich, den 8BP für Variablen verwendet, und wenn diese nicht anfänglich auf Null gesetzt sind, können sie stören

für i = &c000+2000 bis &c000+2000+48: poke i,0:NEXT

kurz und bündig:

10 <mehrere INK-Befehle>

20 Lade " !image.scr", &c000

30 for i = &c000+2000 to &c000+2000+48: poke i,0:NEXT

Um die Palette auf ihren Standardwerten zu belassen, verwenden Sie CALL &BC02, eine Firmware-Routine.

Und um den Bildschirm auf einem Band zu speichern?

Wir haben gesehen, wie man das auf der Festplatte macht, aber CPCDiskXP lässt die .scr-Datei nicht auf dem Band, also müssen wir so vorgehen:

```
|DISC  
SPEICHER 15999  
LOAD "image.scr", 16000  
|TAPE  
SCHNELLES SCHREIBEN 1  
SAVE "imagen.scr",b,16000,16384
```

Und wenn wir sie laden, laden wir sie auf dieselbe Weise wie auf der Festplatte:

Lade " !image.scr", &c000

26 APPENDIX II: Die Palette

Die folgende Tabelle zeigt die AMSTRAD-Palette. Innerhalb jeder Farbe und in Klammern ist die Farbnummer angegeben, die dieser Farbe in der Standardpalette zugewiesen ist. Die 27 Farben sind:

0 - Negro (5)	1 - Azul (0,14)	2 - Azul claro (6)	3 - Rojo	4 - Magenta	5 - Violeta	6 - Rojo claro (3)	7 - Púrpura	8 - Magenta claro (7)
9 - Verde	10 - Cyan (8)	11 - Azul cielo (15)	12 - Amarillo (9)	13 - Gris	14 - Azul pálido (10)	15 - Anaranjado	16 - Rosa (11)	17 - Magenta pálido
18 - Verde claro (12)	19 - Verde mar	20 - Cyan claro (2)	21 - Verde lima	22 - Verde pálido (13)	23 - Cyan pálido	24 - Amarillo claro (1)	25 - Amarillo pálido	26 - Blanco (4)

Die Standard-Palettenwerte in jedem Modus sind:

Modo 2:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
--------------------	---------------------------------

Modo 1:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)

Modo 0:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)	2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)
4: Blanco (paleta 26)	5: Negro (paleta 0)	6: Azul claro (paleta 2)	7: Magenta claro (paleta 8)
8: Cyan (paleta 10)	9: Amarillo (paleta 12)	10: Azul pálido (paleta 14)	11: Rosa (paleta 16)
12: Verde claro (paleta 18)	13: Verde pálido (paleta 22)	14: Parpadeo Azul/Amarillo	15: Parpadeo azul cielo/Rosa

Die Palettenwerte in jedem Modus werden mit dem INK-Befehl verwaltet, weitere Informationen finden Sie im Amstrad BASIC Referenzhandbuch. Um zum Beispiel die Nulltinte als rot einzustellen, sehen wir in der Palette 27 nach, dass rot die sechste Farbe ist und schreiben

TINTE 0,6

Und wir haben die Nulltinte bereits als rot konfiguriert. Wie Sie sehen, ist eine Tinte keine bestimmte Farbe, sondern kann in jeder gewünschten Farbe konfiguriert werden.

Die Amstrad-Palette ist deshalb so gut, weil die 27 Farben so viele Möglichkeiten bieten, auch wenn nur 16 gleichzeitig verwendet werden können. Die Farben sind nach Helligkeit sortiert.

Wenn wir die Farben des Amstrad in der 24-Bit-RGB-Skala (8 Bits pro Komponente) darstellen, stellen wir fest, dass es 3 Rot-, 3 Grün- und 3 Blauwerte gibt (diese Werte sind 0, 127 und 255), was der Anzahl der Kombinationen $3 \times 3 \times 3 = 27$ entspricht. Die graue Farbe befindet sich genau in der Mitte der Palette, wo die Werte von R, G, B gleich sind.

(R=127,G=127,B=127). Die 3 Komponenten sind auch in Weiß (R=255,G=255,B=255) und Schwarz (R=0,G=0,B=0) gleich.

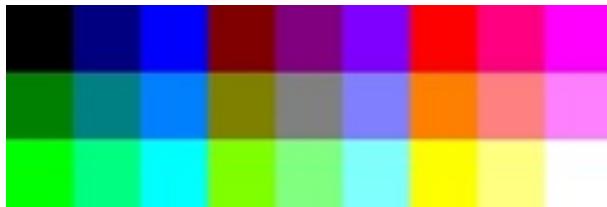


Abb. 129 Amstrad-Palette

Eine Palette von 27 Farben bedeutet, dass wir zwar nur 16 auswählen können, aber immer eine Farbe zur Verfügung steht, mit der wir Übergänge und Überblendungen erstellen können. Andere Computer der damaligen Zeit, wie der C64 (eine großartige Maschine), hatten 16 Farben aus einer Palette von 16. Der C64 hatte 3 Grautöne in einer solch reduzierten Palette, was, obwohl es kritisiert wurde, meiner Meinung nach nicht schlecht ist, da es sich um nicht sehr gesättigte Farben handelt, die sich gut mit dem Grau kombinieren lassen, und sie lassen sich auch gut miteinander kombinieren, das heißt, es sind 16 Farben, die "nahe" beieinander liegen.

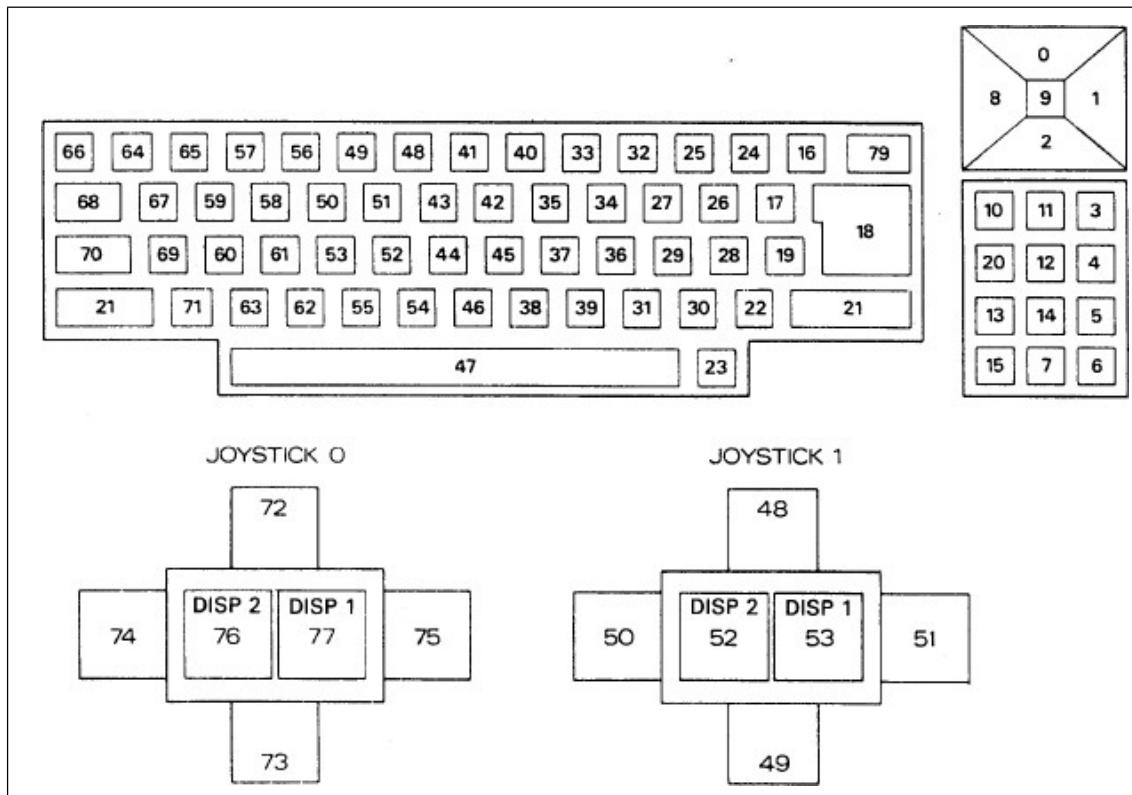


Abb. 130 C64-Palette

Kurz gesagt, auf dem Amstrad können Sie aus mehr Farben wählen und so immer die richtige Farbe zum Verwischen, Schattieren oder einfach zum Finden des gewünschten Farbtöns finden. Der große Erfolg von Amstrad bestand darin, eine Palette von 27 Farben zu schaffen, obwohl man nur 16 Farben gleichzeitig verwenden kann. Man kann auch 16 Farben wählen, die nicht gut zueinander passen, und erhält dann sehr kitschige Grafiken (was auf dem C64 unmöglich ist, da man nicht wählen kann).

Die Palette von 27 erlaubte es vielen Amstrad-Ladegrafiken, wahre Kunstwerke zu sein.

27 APPENDIX III: INKEY-Codes



Wenn Sie die Tastatur auslesen wollen, versuchen Sie zuerst, den Lesepuffer für die letzten Tastenanschläge zu löschen. Es kommt sehr häufig vor, dass sich in einem Bildschirm, in dem nach dem Namen des Benutzers gefragt wird, weil er eine hohe Punktzahl erreicht hat, die letzten Tastenanschläge des Spiels (Bewegungen und Schüsse) "einschleichen" und Dinge wie "OPPPOQQAAA" im INPUT-Befehl angezeigt werden. Um dies zu vermeiden, führen Sie etwas aus wie:

```
10 B$=INKEY$: wenn B$<>"" THEN 10
20 INPUT "Name:";$name
```

Zusätzlich zu dieser Empfehlung sollten Sie daran denken, dass die Tastatur am schnellsten wie folgt bearbeitet werden kann:

```
10 IF INKEY(keycode) THEN 30: REM springt auf 30, wenn nicht gedrückt
20 <Hinweise für den Fall, dass der Tastencode
gedrückt wird> 30
```


29 APPENDIX IV: AMSTRAD CPC ASCII-Tabelle

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	P	^P	·	^	α	/	-	Ω	↑					
1	Γ	Φ	!	1	Α	Q	α	q	■	Ι	β	＼	I	Θ	↓	
2	Τ	Φ	"	2	Β	R	b	r	■	-	„	γ	„	Φ	←	
3	Λ	Φ	#	3	C	S	c	s	■	£	€	„		♦	→	
4	₩	Φ	*	4	D	T	d	t	■	,	®	€	^	♥	▲	
5	₪	₪	×	5	E	U	e	u	■		π	Θ)	♣	▼	
6	✓	Π	&	6	F	V	f	v	■	r	§	λ	√	○	▶	
7	Ω	Γ	'	7	G	W	g	w	■	τ	‘	ρ	〈	◀	●	
8	←	X	(8	H	X	h	x	■	-	14	π	✓	❀	□	✗
9	→	†)	9	I	Y	i	y	■	12	σ	ς	▀	█	♂	
A	↓	Ω	*	:	J	Z	j	z	■	-	34	δ	○	❀	♂	
B	↑	Θ	+	;	K	[k	€	■	±	δ	X	▀	♀	♂	
C	Ψ	¶	,	<	L	\	l	l	■	¬	÷	X	/	▀	♂	
D	↔	▀	-	=	M]	m	»	■	¬	ω	＼	▀	▀	▀	
E	⊗	▀	.	>	N	†	n	~	■	†	δ	Σ	▀	▀	▀	
F	Θ	¶	/	?	O	_	o	▀	■	+	i	Ω	▀	▀	▀	

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

30 APPENDIX V: einige Soundeffekte

Zunächst einmal müssen Sie wissen, dass der Amstrad 3 Kanäle hat, die mit 1, 2 und 4 bezeichnet sind. Um zwei Kanäle oder alle 3 gleichzeitig zum Klingen zu bringen, müssen Sie sie einfach addieren

Verwendung des SOUND-Befehls:

SOUND Kanal, Tonhöhe, Dauer, Lautstärke, Hüllkurve Tonhöhe, Hüllkurve Lautstärke, Rauschen

WICHTIG: Die Lautstärke reicht von 0 bis 7 auf dem CPC464 und von 0 bis 15 auf dem 6128. Auf dem CPC464 können die Werte 8..15 verwendet werden, aber sie sind eine Wiederholung der Werte 0..7 (d.h. 8 bedeutet Lautstärke 0). Dies ist ein grundlegender Unterschied zwischen den beiden Modellen. Infolgedessen ist die Lautstärke 10 auf dem CPC6128 hoch, während sie auf dem 464 sehr niedrig ist.

Der Rauschparameter reicht von 0 bis 31

Beispiele:

SOUND 1,2000,10,7 : Töne Kanal 1

SOUND 1+2,2000,10,7 : Kanäle 1 und 2 werden abgehört

Hier finden Sie einige Beispiele, die Sie direkt in Ihren Programmen verwenden oder die Sie zu anderen Klängen inspirieren können.

Sammeln Sie einen Diamanten oder eine ENT-Münze 1,10,-100,3: Ton 1,638,30,30,15,15,0,1	Sie wurden von einem Stein oder einem Projektil getroffen ENT 1.10, 100.3: Ton 1,638,30,30,15,15,0,1
Boing ENV 1,1,15,1,15,-1,1: TON 1,638,0,0,0,1	Boing 2 ENT 2,20,-125,1: TON 1,1500,10,12,12,,2
Tod HNO 3,100,5,3: TON 1,142,100,15,0,3	
Explosion TON 7,1000,20,20,15,,,15	Explosion 2 ENV 1,11,-1,25:ENT 1,9,49,5,9,-10,15 TON 3,145,300,12,1,1,1,12
Feuern ENT -5,7,10,1,1,7,-10,1: TON 1,25,20,12,12,,5	

31 APPENDIX VI: Interessante Firmware-Routinen

In diesem Abschnitt werde ich einige Firmware-Routinen vorstellen, die von BASIC aus aufgerufen werden können und die in Ihren Programmen interessant sein können.

CALL 0 : setzt den Computer zurück

CALL &bc02 : initialisiert die Palette auf ihren Standardwert. Es ist ratsam, diese Funktion am Anfang des Programms aufzurufen, falls sie geändert wird.

CALL &bd19 : Synchronisierung mit dem Bildschirmscrollen. Wenn Sie nur sehr wenige Sprites bearbeiten, können Sie eine flüssigere Bewegung erreichen, aber bedenken Sie, dass diese Anweisung Ihr Programm sehr verlangsamt.

CALL &bb48 : Deaktiviert den BREAK-Mechanismus, so dass das Programm nicht angehalten werden kann, wenn es bereits läuft.

CALL &bd21 , &bd22, &bd23, &bd24, &bd25 : erzeugt einen Blitzeffekt auf dem Bildschirm

Um den TIMER des AMSTRAD zurückzusetzen:

Bei einem 6128

POKE &b8b4,0: POKE &b8b5,0: POKE &b8b6,0: POKE &b8b7,0

In einem 464

POKE &b187,0: POKE &b188,0: POKE &b189,0: POKE &b18a,0

Um zu unterscheiden, auf welchem Gerät Ihr Programm läuft, müssen Sie die Musik deaktivieren und eine Adresse mit PEEK abfragen.

**| MUSIC: Wenn peek(&39)=57 dann model=464 sonst
model=6128 Wenn model=464 dann ...**

CALL &bca7 : Beenden des Klingelns eines Tons, der gerade abgespielt wurde

Der Befehl GRAPHICS PAPER existiert in CPC6128, aber nicht in CPC464. Es gibt jedoch eine Möglichkeit, ihn zu erhalten, indem man den **CALL &BBE4** und so viele Parameter mit dem Wert 1 wie z.B. die gewünschte Tintenfarbe verwendet:

CALL &BBE4,1,1: wie "GRAPHICS PAPER 2", funktioniert aber auf cpc464

CALL &BB18 : wartet darauf, dass Sie eine Taste drücken

32 APPENDIX VII: Tabelle der Sprite-Attribute

Die folgende Tabelle enthält die Speicheradressen, an denen die Attribute der einzelnen Sprites gespeichert sind, sowie die Länge der einzelnen Attribute in Bytes.

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Tabelle 7 Adressen der Sprite-Attribute

7 ROUTEALL lo ruta	6 Sobre- escritura	5 COLSPALL collider	4 MOVERALL lo mueve	3 AUTOALL lo mueve	2 ANIMALL lo anima	1 COLSP collided	0 PRINTSPALL lo imprime
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Tabelle 8 Merker im Statusbyte

33 APPENDIX VIII: Speicherplan des 8BP

```
AMSTRAD CPC464 SPEICHERKARTE von 8BP

;
; &FFFF +-----+
; | Anzeige + 8 versteckte Segmente von je 48 Byte
; &C000 +-----+
; | System (umdefinierbare Symbole, Stapelzeiger, usw.)
; 42619 +-----+
; | Bank mit 40 Sternen (von 42540 bis 42619 = 80 Bytes)
; 42540 +-----+
; | Zeichen-Layout-Map (25x20 =500 Bytes)
; | und Weltkarte (bis zu 82 Elemente passen in 500 Bytes)
; | Beide sind im selben Speicherbereich gespeichert.
; | weil man entweder das eine oder das andere benutzt.
; 42040 +-----+
; | Sprites (fast 8,5KB für
; | Zeichengrafiken-8440 Bytes, wenn es keine Sequenzen und
; | | keine Routen gibt)
; | +-----Auch-Alphabetbilder werden hier gespeichert.
; | | Routendefinitionen (jeweils mit variabler Länge)
; | +-----+
; | | Animationssequenzen mit 8 Bildern (je 16 Byte)
; | | und Gruppen von Animationssequenzen (Makro-
; | 33600 Sequenzen)
; | | Lieder
; | | | (1500 Bytes für Musik, die mit WYZtracker 2.0.1.0
; | 32100 +-----bearbeitet wurde)
; | | 8BP-Routinen (8100 Bytes oder 7100 Bytes)
; | | Hier sind alle Routinen und die Sprite-Tabelle
; | | enthält den Musik-Player "wyz" 2.0.1.0
; 25000 +-----+
; | |
; | | IHRE BASIS- oder C-LISTE
; | | 24KB, 24,8 KB oder bis zu 25KB frei für BASIC oder C, je
; | | nachdem, welche Assembleroption Sie für 8BP
; | | verwenden
; | |
; | 0 +-----+
```


34 APPENDIX IX: Verfügbare Befehle 8BP

Liste der verfügbaren Befehle in alphabetischer Reihenfolge:

3D, <Kennzeichen>, #, Offsetdruck 3D, 0	Aktiviert den Pseudo-3D-Projektionsmodus.
ANIMA, #	Ändert den Rahmen eines Sprites entsprechend seiner Sequenz
ANIMALL	Ändert den Frame von Sprites mit aktiviertem Animationsflag (muss nicht aufgerufen werden, ein Flag in der PRINTSPALL-Anweisung reicht aus, um es aufzurufen).
AUTO, #	Automatische Bewegung eines Sprites entsprechend seiner Vy,Vx
AUTOALL, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>.	Bewegung aller Sprites mit aktivem Flag für automatische Bewegung
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	Erkennt Kollisionen mit dem Layout und gibt 1 zurück, wenn eine Kollision vorliegt. Akzeptiert eine variable Anzahl von Parametern (immer in der gleichen Reihenfolge) von 4 bis keine.
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, Ende COLSP, 33, @collided% COLSP, 34, dy, dx COLSP, #	Gibt das erste Sprite zurück, mit dem # kollidiert. Der Befehl kann mit den Codes 32, 33 und 34 konfiguriert werden.
COLSPALL, @who%, @who%, @mitwho%. COLSPALL, Kollider COLSPALL	Gibt zurück, wer kollidiert ist (collider) und mit wem er kollidiert ist (collided).
LAYOUT, y, x, @String\$, @String\$, @String\$, @String\$, @String\$, @String\$.	Druckt 8x8 Bildstreifen und füllt das Kartenlayout
LOCATESP, #, y, x	Ändert die Koordinaten eines Sprites (ohne es zu drucken)
MAP2SP, y, x MAP2SP, Status	Erzeugt Sprites, um die Welt in scrollenden Spielen zu zeichnen. Sprites werden mit state = status erstellt
MOVER, #, dy, dx	relative Bewegung eines einzelnen Sprites
MOVERALL, dy,dx MOVERALL	Relative Bewegung aller Sprites mit aktivem Flag für relative Bewegung
MUSIC, C, Flagge, Lied, Geschwindigkeit MUSIC, Flagge, Lied, Geschwindigkeit MUSIK	Eine Melodie beginnt zu spielen. Kanal C kann bei Bedarf für die Verwendung mit Effektgeräten ausgeschaltet werden. Keine Parameter beenden das Klingeln
PEEK, dir, @Variable%	Liest einen 16-Bit-Wert (kann negativ sein)
POKE, dir, wert	einen 16-Bit-Wert eingeben (der auch negativ sein kann)
PRINTAT, flag, y, x, @string	Druckt eine Zeichenkette aus umdefinierbaren "Mini-Zeichen".
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, Bits	druckt ein einzelnes Sprite (# ist seine Nummer) unabhängig vom Statusbyte. Wenn 32 angegeben ist, werden die Hintergrundbits gesetzt
PRINTSPALL, ini, fin, anima, sync PRINTSPALL, Auftragsmodus PRINTSPALL	Druckt alle Sprites mit aktivem Druck-Flag. Beim Aufruf mit einem einzigen Parameter wird der Ordnungsmodus festgelegt.
RINK,tini,Farbe1,Farbe1,Farbe2,...,FarbeN RINK, Sprung	Rotiert einen Satz von Druckfarben nach einem definierbaren Muster, das aus einer beliebigen Anzahl von Druckfarben besteht
ROUTESP, #, Schritte	Lässt dich N Schritte der Sprite-Route auf einmal

	durchlaufen
ROUTEALL	Ändern Sie die Sprite-Geschwindigkeit mit dem Pfad-Flag (Sie brauchen es nicht aufzurufen, nur das Flag in AUTOALL).
SETLIMITS, xmin, xmax, ymin, ymax	Definiert das Spielfenster, in dem das Clipping durchgeführt wird.
SETUPSP, #, param_number, value SETUPSP, #, 5, Vy, Vx	Ändert einen Parameter eines Sprites. Wenn Parameter 5 angegeben wird, kann Vx optional angegeben werden.
STARS, initstar, num, colour, dy, dx	Schriftrolle aus einer Reihe von Sternen
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Aktualisiert Weltkartenelemente mit einer Teilmenge von Elementen aus einer größeren Karte

35 APPENDIX X: 8BP Montageoptionen

Seit Version V42 verfügt die 8BP-Bibliothek über mehrere Assembler-Optionen, die es Ihnen ermöglichen, die gewünschten Kapazitäten für Ihr Spiel auszuwählen und somit mehr Speicher für Ihr Spielprotokoll zur Verfügung zu haben.

Die Assembler-Option muss in der Datei **Make_all_mygame.asm** angegeben werden, die eine spezielle Zeile zur Zuweisung des Wertes des Parameters "**ASSEMBLING_OPTION**" enthält.

Option	Beschreibung der Option	Beispiel für ein typisches Spiel
0	<p>Sie können jedes Spiel spielen Alle Befehle verfügbar Sie müssen MEMORY 23499 verwenden</p> <p>Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP0.bin",b,23500,19119</p>	jeder
1	<p>Labyrinth- oder Screen-Passing-Spiele Sie müssen MEMORY 24999 verwenden In diesem Modus nicht verfügbar: MAP2SP, UMAP, 3D</p> <p>Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP1.bin",b,25000,17619</p>	
	<p>Für Rollenspiele müssen Sie MEMORY 24799 verwenden. In diesem Modus nicht verfügbar: LAYOUT, COLAY, 3D</p> <p>Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP2.bin", b,24800,17819</p>	
	<p>Für Spiele mit Pseudo-3D müssen Sie MEMORY 23999 verwenden. In diesem Modus nicht verfügbar: LAYOUT, COLAY</p> <p>Zum Speichern von Bibliothek + Grafiken + Musik: SAVE "8BP3.bin", b,24000,18619</p>	

36 APPENDIX XI: RSX/CALL-Zuordnungen

Liste der verfügbaren Befehle in alphabetischer Reihenfolge und der zugehörigen Adresse, um den CALL &XXXX-Aufruf zu verwenden, wenn dies zur Erhöhung der Geschwindigkeit erforderlich ist. Ich habe nur einige Beispiele zur Veranschaulichung angeführt, da die Verwendung mit dem RSX-Befehl identisch ist, mit der Ausnahme, dass Sie den Befehl durch CALL <Adresse> ersetzen müssen.

WICHTIG: Von einer Version von 8BP zur anderen kann diese Adressliste variieren. Vergewissern Sie sich, dass Sie die neueste Version von 8BP verwenden, wenn Sie die Adressen in dieser Tabelle nutzen wollen.

KOMMANDO	ADRESSE	BEISPIEL
3D	&6BDE	
ANIMA	&6BB7	
ANIMALL	&7479	
AUTO	&6BC9	
AUTOALL	&6B9C	CALL &6B9C,1
COLAY	&6BA8	
COLSP	&6BBA	
COLSPALL	&6B99	
LAYOUT	&6BD5	
LOCATESP	&6BAE	
MAP2SP	&6BA2	
MOVER	&6BC0	
OVERALL	&6B9F	
MUSIK	&6BD8	AUFRUF &6BD8,0,0,0,0,6
PEEK	&6BB1	CALL &6BB1,dir,@var
POKE	&6BB4	
PRINTAT	&6BC6	CALL &6BC6,0,y,x,@c\$ CALL &6BC6,0,y,x,@c\$
PRINTSP	&6BC3	CALL &6BC3,31
PRINTSPALL	&6B96	AUFRUF &62A6,0,0,0,0,0,0
RINK	&6BBD	
ROUTESP	&6BCC	
ROUTEALL	&6BD2	
SETLIMITS	&6BDB	
SETUPSP	&6BAB	
STARS	&6BA5	
UMAP	&6BCF	

37 APPENDIX XII: 8BP-Funktionen in C

RSX	C-Prototyp
3D, 0 3D, <Kennzeichen>, #, Offsetdruck	void _8BP_3D_1(int flag); void _8BP_3D_3(int flag, int sp_fin, int offsety);
ANIMA, #	void _8BP_anima_1(int sp);
ANIMALL	void _8BP_animall();
AUTO, #	void _8BP_auto_1(int sp);
AUTOALL, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>, <Flag geroutet>.	void _8BP_autoall(); void _8BP_autoall_1(int flag);
COLAY, umbral_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	void _8BP_colay_3(int threshold, int* collision, int sp); void _8BP_colay_2(int* collision, int sp); void _8BP_colay_1(int sp); void _8BP_colay();
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, Ende COLSP, 33, @collided% COLSP, # COLSP, 34, dy, dx	/* Operation 32, ini,fin oder Operation 34,dy,dx*/ void _8BP_colsp_3(int operation, int a, int b); /*Operation 33 oder sp*/ void _8BP_colsp_2(int sp, int* collision); void _8BP_colsp_1(int sp);
COLSPALL,@who%,@who%,@mitw hom% COLSPALL, Kollider COLSPALL	void _8BP_colspall_2(int* collider, int* collided); void _8BP_colspall_1(int collider_ini); void _8BP_colspall();
LAYOUT, y, x, @String\$, @String\$, @String\$, @String\$, @String\$.	void _8BP_layout_3(int y, int x, char* cad);
LOCATEESP, #, y, x	void _8BP_locatesp_3(char sp, int y, int x);
MAP2SP, y, x MAP2SP, Status	void _8BP_map2sp_2(int y, int x); void _8BP_map2sp_1(unsigned char status);
MOVER, #, dy, dx	void _8BP_mover_3(int sp, int dy,int dx); void _8BP_mover_1(int sp);
MOVERALL, dy,dx	void _8BP_moverall_2(int dy, int dx); void _8BP_moverall();
MUSIC, C, Flagge, Lied, Geschwindigkeit MUSIC, Flagge, Lied, Geschwindigkeit MUSIK	void _8BP_music_4(int flag_c, int flag_repetition,int song, int speed); void _8BP_music();
PEEK, dir, @Variable%	void _8BP_peek_2(int address, int* data);
POKE, dir, wert	void _8BP_poke_2(int address, int data);
PRINTAT, flag, y, x, @string	void _8BP_printat_4(int flag,int y,int x,char* cad);
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, Bits	void _8BP_printsp_1(int sp) ; void _8BP_printsp_2(int sp, int bits_background) ; void _8BP_printsp_3(int sp,int y,int x) ;
PRINTSPALL, ini, fin, anima, sync PRINTSPALL, Auftragsmodus PRINTSPALL	void _8BP_printspall_4(int ini, int fin, int flag_anima, int flag_sync); void _8BP_printspall_1(int order_type); void _8BP_printspall();
RINK,tini,Farbe1,Farbe1,Farbe2,...,FarbeN RINK, Sprung	void _8BP_rink_N(int num_params,int* ink_list); void _8BP_rink_1(int step);
ROUTESP, #, Schritte	void _8BP_routesp_2(int sp, int steps); void _8BP_routesp_1(int sp);
ROUTEALL	void _8BP_routeall();
SETLIMITS, xmin, xmax, ymin, ymax	Void _8BP_setlimits_4(int xmin, int xmax, int ymin, int ymax)

SETUPSP, #, param_number, value	void _8BP_setupsp_3(int sp, int param, int value);
SETUPSP, #, 5, Vy, Vx	void _8BP_setupsp_4(int sp, int param, int value1,int value2);
STARS, initstar, num, color, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int color, int dy, int dx); void _8BP_stars();
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	void _8BP_umap_6(int map_ini, int map_fin, int y_ini, int y_fin, int x_ini, int x_fin);

38 APPENDIX XIII: MiniBASIC in C

BASIC	C-Prototyp
BORDER	<code>void _basic_border(char Farbe);</code> <i>//Beispiel _basic_border(7)</i>
ANRUFEN	<code>void _basic_call(unsigned int address);</code> <i>// Beispiel _basic_call(0xbd19)</i>
ZEICHNEN	<code>void _basic_draw(int x, int y);</code>
TINTE	<code>void _basic_ink(char ink1,char ink2);</code>
INKEY	<code>char _basic_inkey(char key);</code> <i>//dauert etwa 0,3 ms. langsam aber einfach</i>
LOCATE	<code>void _basic_locate(unsigned int x, unsigned int y);</code> <i>// Beispiel: _basic_locate(2,25);_basic_print("TEST");</i>
MOVE	<code>void _basic_move(int x, int y);</code>
PAPIER	<code>void _basic_paper(char ink);</code>
PEEK	<code>char _basic_peek(unsigned int address);</code>
GRAFIKEN	<code>void _basic_pen_graph(char ink);</code>
PEN	<code>void _basic_pen_txt(char ink);</code>
POKE	<code>void _basic_poke(unsigned int address, unsigned char Daten);</code>
PLOT	<code>void _basic_plot(int x, int y);</code>
DRUCKEN	<code>void _basic_print(char *cad);</code> <i>//Beispiel: _basic_print("Hallo")</i>
RND	<code>unsigned int _basic_rnd(int max);</code> <i>//Beispiel: num=_basic_rnd(50)</i>
TON	<code>void _basic_sound(unsigned char nChannelStatus, int nTonePeriod, int nDuration, unsigned char nVolume, char nVolumeEnvelope, char nToneEnvelope, unsigned char nNoisePeriod);</code>
STR\$	<code>char* _basic_str(int num);</code> <i>//ähnlich wie STR\$</i> <i>//Beispiel: _basic_print(_basic_str(num))</i>
ZEIT	<code>unsigned int _basic_time();</code> <i>//Rückgabe eines unsigned int,(0..65535). Als Ganzzahl, wenn</i> <i>// bei Erreichen von 32768 gehen Sie auf -32768</i>