

Programmierung des Motorola 6809 auf der Vectrex

Jan Haasch
Humboldt Universität
zu Berlin

Email: jan.haasch@student.hu-berlin.de
Matrikelnr.: 561482

I. EINLEITUNG

„Vectrex, oder kennst Du was Besseres?“ - ist der Werbeslogan mit dem das Unternehmen *Milton Bradley* (auch bekannt als MB Spiele) im Jahr 1983 seine neue Spielkonsole auf dem deutschen Markt präsentierte. Entwickelt wurde die Vectrex¹ bereits im Herbst des Jahres 1981 durch das Unternehmen *General Consumer Electronics* (GCE) und die Entwickler John Ross, Gerry Karr, John Hall und Jay Smith. Obwohl die Konsole insgesamt nur 2 Jahre auf dem Markt vertrieben wurde, stellt sie aufgrund ihrer Einzigartigkeit einen wichtigen Schritt in der Entwicklung der Spielkonsolen dar [4]. So wurde in der Fachpresse die Kategorie „Mini-Arcade“ neu geschaffen, um die Vectrex einordnen zu können. Vor allem durch ihren kompakten Aufbau sowie den integrierten Vektorbildschirm erlangte die Konsole zur ihrer Zeit einen hohen Bekanntheitsgrad [2].



Abbildung 1. Deutsche Vectrex Werbung von Milton Bradley [2]

In den Jahren 1982 bis 1984 wurden allerdings nur 24 Spiele² für diese Plattform entwickelt. Dies gilt neben dem hohen Preis von 499 DM (199\$ in den USA) und der starken Konkurrenz als einer der Gründe für das frühe Aus der Konsole [2].

Nach dem Produktionsende hat es etwas mehr als 10 Jahre gedauert, als 1995 die Programmierer Sean Kelly und John Dodzila begannen erneut Software für die Vectrex zu schreiben. Im März 1996 konnte so, das erste Mal seit 1984, ein neues Spiel (Vectrex Vaders) veröffentlicht werden [5] [4].

¹In einigen Veröffentlichungen wird von der Vectrex aber auch das Vectrex gesprochen, da es sich um ein Phantasiewort handelt ist eine eindeutige Artikelbestimmung nicht möglich.

²Liste der Spiele siehe Anhang A.

Seitdem haben sich weitere Heimentwickler (sog. *Homebrew Programmer*) zusammengeschlossen und auf unterschiedlichen online und offline Plattformen ihre Entwicklungen diskutiert und veröffentlicht. Auf diese Art und Weise wurde der Vectrex ein zweiter Lebenszyklus vergönnt der bis heute anhält.

Die vorliegende Arbeit schließt sich dieser Entwicklung an und hat zum Ziel ein einfach gehaltenes *Jump 'n' Run* Spiel für die Vectrex zu entwickeln und zu dokumentieren.

Um dem Leser den Einstieg in die Vectrex Programmierung zu vereinfachen gliedert sich die Arbeit in das Kapitel I. *Einleitung* mit den Unterkapiteln A. *Historie*, B. *Aufbau* und C. *Programmierung*. In den ersten beiden Kapiteln *Historie* und *Aufbau* soll zunächst ein Blick in die Geschichte und den technischen Aufbau der Vectrex und seinem Mikroprozessor, dem Motorola 6809, geworfen werden. Das Kapitel *Programmierung* geht hingegen auf die Assemblerprogrammierung im Allgemeinen und speziell auf den Befehlssatz des 6809 ein.

Das eigentliche Hauptkapitel bildet Kapitel II. *Das Spiel*. Unterteilt ist es in A. *Spielidee*, B. *Programmierung* und C. *Spielanleitung*. Hier wird das eigentliche Spiel vorgestellt und der Quellcode erläutert. Das Kapitel III. *Probleme und Herausforderungen* sowie das *Fazit* schließen die Arbeit ab.

Diese Arbeit ist im Rahmen des Seminars „*Programmierung des Motorola 6809 auf der Vectrex*“ an der Humboldt Universität zu Berlin entstanden und wurde durch Dr. Stefan Höltingen betreut.

A. Historie

Dieses Kapitel gibt nun einen kurzen historischen Ablauf über die Entwicklung der Vectrex und deren Rolle in der Entwicklung der Heimvideospiele wieder.

Historisch gesehen gehört die Vectrex zur zweiten Generation der Videospiele. Während die erste Generation die Anfänge vor allem zu Beginn der 1970iger Jahren beschreibt, beginnt die Zeit der zweiten Generation zum Ende der 70iger Jahre des 20. Jahrhunderts und führt bis zum ersten großen Videospiele-Crash im Jahr 1983. Diese beiden Generationen wurden vor allem durch das amerikanische Unternehmen *Atari Inc.* beherrscht, welches mit dem berühmten Spiel *Pong* das erste kommerziell erfolgreiche Spiel auf den Markt brachte. In dieser Zeit konnten Unternehmen wie *Atari* und *Magnavox* die aus großen Spielhallen bekannten Spiele auch für Heimanwender zugänglich machen und so einen rasant wachsenden Markt mit immer neuen Spielen und Spielkonsolen versorgen.

Diesem Boom schlossen sich nach und nach auch andere Unternehmen mit ihren Entwicklungen an.

Wie bereits erwähnt wurde die Vectrex nicht von *MB Spiele* entwickelt sondern nur vermarktet. Das bis dahin vor allem für seine Brettspiele bekannte Unternehmen betrat damit erstmals den Videospielemarkt [4].

Die eigentliche Idee einen CRT Monitor zur Darstellung von Vektorgrafiken eines Computerspiels zu nutzen, kam von Mike Purvis und John Ross im Unternehmen *Smith Engineering/Western Technologies*. Geführt wurde das kleine Videospielunternehmen von Jay Smith. Dieser übernahm auch das Projektmanagement und unterbreitete dem Präsidenten des Unternehmens *GCE* im Frühling 1981 die Idee eines Mini Arcade Spieles für Zuhause. *GCE* übernahm das bis dahin erarbeitete Konzept der Vectrex, änderte allerdings einige Spezifikationen. So wurde unter anderem das Display von 5 auf 9 mal 11 Zoll vergrößert. Ab Herbst 1981 begannen die Arbeiten an einem Vectrex Prototypen mit dem Ziel bis Juni 1982 die Hardware sowie 12 Spiele zu entwickeln. Bereits im Januar 1982 wurde klar, dass der gewählte Prozessor Motorola 6502 zu wenig Leistung für die Konsole haben würde, daher entschied man sich für die leistungstärkere Variante 6809. Die Vectrex wurde als reines Vektor-Grafik-System entwickelt, woraus sich auch der Name ableitet. Dies erlaubt der Konsole sehr scharfe Grafiken darzustellen, mit für damalige Verhältnisse sehr fortschrittlichen visuellen Effekten. Als Beispiele zu nennen sind hier die Größenskalierung und Rotation von Objekten. Allerdings wurde aus Kostengründen auf einen Farbbildschirm verzichtet [4].

Am 6. Juni 1982 war es so weit und die fertige Vectrex konnte auf der Sommer-CES in den USA vorgestellt und ab Oktober für einen Preis von 199\$ erworben werden. Erst im März 1983 entschied sich *MB* den Videospielemarkt zu betreten und kaufte das Unternehmen *GCE* auf. Anschließend wurde die Konsole nicht nur in den USA, sondern auch auf dem europäischen und asiatischen Markt vertrieben.

Im Sommer 1983 kam es zum großen Videospiele Crash bei dem die Industrie der Videospiele-Hersteller sehr schnell an Wert verlor. *MB* musste daraufhin deutliche Verluste hinnehmen und hat die Preise der Vectrex zunächst auf 150 und anschließend auf 100 US\$ gesenkt [4].

Später gingen die Rechte der Vectrex zurück an das Unternehmen von Jay Smith, welcher eine „non-profit“ Verbreitung der Anleitungen und der Spiele möglich machte. Diese Entscheidung sowie die Einzigartigkeit der Konsole führte dazu, dass in den 90iger Jahren des 20. Jahrhunderts ein harter Kern von Vectrex Entwicklern einige Emulatoren und neue Spiele für diese Plattform entwickelten [4].

B. Aufbau

Nach dieser historischen Zusammenfassung wird nun der technischen Aufbau der Vectrex erörtert. Abbildung 2 zeigt dabei das äußere Erscheinungsbild der Vectrex mit ihrem integrierten Monitor, einem *Control Pannel* sowie dem *Cartridge* und dem später erhältlichen *Light Pen*.

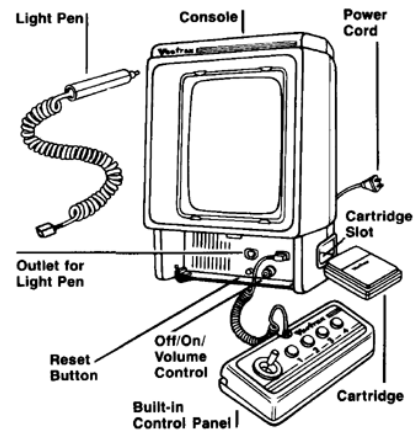


Abbildung 2. Bauteile der Vectrex [1]

Es können bis zu zwei Controller an die Vectrex angeschlossen werden. Wie ebenfalls anhand der Abbildung 2 ersichtlich, besteht der Controller der Vectrex aus einem analogen Joystick, sowie 4 Knöpfen. Seitlich am Gerät können weitere Spielmodule in den *Cartridge* eingeführt werden.

Das Herz der Vectrex bildet der Mikroprozessor 6809 der Firma *Motorola*. Dabei handelt es sich um einen 8 Bit Prozessor mit einer Taktung von 1,5MHz (weitere Daten siehe Tabelle I).

Beschreibung		
Prozessor	8 Bit	
Taktung	1,5MHz	
Datenbus	8 Bit	
Adressbus	16 Bit	bis zu 64KB adressierbar
Datenregister	2x 8 Bit	A und B
Stackpointer	ja	16 Bit, U & S
Indexregister	ja	16 Bit, X & Y
Interrupts	ja	
Transistoren	6800x	
Maschinenbefehle	59x	

Tabelle I

TECHNISCHE DATEN DES MOTOROLA 6809 [3]

Der Prozessor besitzt außerdem einen 8 Bit Daten- und 16 Bit Adressbus. Damit sind bis zu 64 Kilobyte adressierbar. Zur Programmierung stehen 59 Maschinenbefehle und 2 Datenregister mit jeweils 8 Bit zur Verfügung. Die beiden Register des Akkumulators, auch A und B Register genannt, können dabei zu einem 16 Bit Register (D) zusammengefasst werden. Weitere Register sind die Indexregister X und Y sowie der User (U) und System (S) Stack Pointer [3].

Wie alle modernen Prozessoren besitzt der 6809 außerdem eine Reihe von Condition Codes, oder auch Flags genannt. Je nach Bedingung werden diese Flags gesetzt, d.h. mit einer 1 geladen oder enthalten eine 0, falls die nötige Bedingung nicht zutrifft. Ein konkretes Beispiel ist das *Zero Flag*. Es wird nur gesetzt wenn die Rechenoperation die im Akkumulator durchgeführt wurde eine Null zum Ergebnis hat. Weitere oft genutzte Flags sind das *Negative Flag*, *Carry Flag* und das *Overflow Flag* (weitere Flags siehe Abbildung 3).

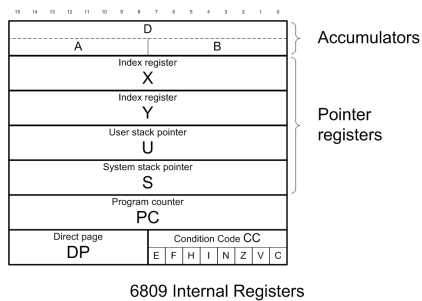


Abbildung 3. Motorola 6809 Register [1]

Neben dem Prozessor ist die Vectrex mit 1 Kilobyte RAM und 8 Kilobyte ROM ausgestattet. Über den *Cartridge* können nochmal 32 Kilobyte ROM adressiert werden. Außerdem besitzt die Vectrex einen Soundprozessor (*Sound General Instrument AY-3-8912*) der als Co-Prozessor zum Einsatz kommt. Die visuelle Ausgabe erfolgt, wie bereits erwähnt, über einen 9x11 Zoll großen schwarz/weiß CRT Monitor der hochkant in das Gehäuse eingelassen ist.

C. Programmierung

Um einen Geist in die Maschine zu bekommen, ist das Schreiben von Software notwendig. Die dafür eingesetzte Programmiersprache nennt sich Assembler. Dabei handelt es sich um eine sehr maschinennahe Programmiersprache, deren Befehle direkt in Prozessorinstruktionen umgewandelt werden können. Wobei zu beachten ist, dass es nicht eine Assembler-Programmiersprache gibt, sondern eine Vielzahl von prozessorabhängigen Befehlssätzen. Die heute wohl bekanntesten Assembler basieren auf der x86 Architektur. Beispiele sind der *Microsoft Macro Assembler* oder der *Netwide Assembler*. Diese beiden Assembler unterstützen die vorherrschenden Assembler Syntaxen von *AT&T* und *Intel*. Dabei werden die gleichen Instruktionssätze, diese sind fest kodiert in den Prozessoren enthalten, in unterschiedliche Schreibweisen übersetzt (siehe Tabelle II).

AT&T Syntax	Intel Syntax
movl %eax, %ebx	mov ebx, eax
addl \$5, %eax	add eax, 5

Tabelle II

VERGLEICH AT&T UND INTEL ASSEMBLER SYNTAX

Der Motorola 6908 ist 1978 in verschiedenen Varianten auf den Markt erschienen. Mit dem *asm6809* ist heute noch ein sogenannter *Cross Assembler* für diesen Prozessortyp verfügbar, welcher auch unter modernen Betriebssystemen lauffähig ist. Zur Zeit seiner Herstellung wurde häufig der Motorola-eigene AS9 als gängiger Assembler genutzt. Als Ausgang einer Assemblierung, d.h. der Umwandlung vom menschenlesbaren Code zu Maschinencode, dient eine .asm-Datei. Im Verlauf dieser Dokumentation werden einige Codefragmente aus der .asm-Datei des Spiels vorgestellt und erläutert. Den vollständigen Inhalt des Spielcodes findet sich im Anhang B. Eine Übersicht über alle 6809 Befehle kann

außerdem auf der Moodle-Seite des Seminars eingesehen werden.

Neben dem Assemblieren und anschließenden Ausführen der Binärdatei auf dem Originalgerät gibt es eine weitere Möglichkeit die .asm-Datei nutzbar zu machen. Mit Hilfe von unterschiedlichen Vectrex Emulatoren, die zum großen Teil frei im Internet zur Verfügung stehen, kann ebenfalls Assemblercode getestet werden. Ein Beispiel ist der *ParaJVE*. Dabei handelt es sich um ein auf Java basierendes und plattformunabhängigen Emulator der auch ohne vorheriges assemblieren .asm-Datei einlesen und wiedergeben kann. Zur Entwicklung des hier dokumentierten Spiels wurde ParaJVE genutzt.

II. DAS SPIEL

Das in dieser Arbeit vorgestellte Spiel trägt den Namen „*J Game*“. Das J steht dabei zum einen für den Vornamen des Autors (Jan) und zum anderen für die Art des Spiels (Jump 'n' Run).

Ziel des Spiels ist es das nächste Level zu erreichen, indem die Spielfigur durch das Drücken des Button 1 (auf dem Emulator die Taste Q) zu einem Sprung über die erscheinenden Hindernisse animiert wird. Da es sich um ein Einstiegsprojekt handelt, wurde auf weitere Aktionen wie sie aus klassischen Jump 'n' Run Spielen bekannt sind, bspw. vor und zurück laufen, verzichtet. Die Spielfigur befindet sich stets auf der selben Spielebene und kann nur eine Art von Sprung durchführen. Die im Spiel erscheinenden Hindernisse können, abhängig vom erreichten Spiellevel, in unterschiedlicher Frequenz und Geschwindigkeit erscheinen.

Ein dediziertes Spielende gibt es nicht. Das höchste erreichbare Level ist Level Nummer 4. Hat der Spieler dieses Level erreicht, kann es nur durch ein Game Over, d.h. der Verlust aller Lebenspunkte beendet werden, bis dahin werden fortlaufend neue Hindernisse generiert.

A. Spielidee

Die dem Spiel zu Grunde liegende Idee ist es, dem Spieler das Gefühl eines klassischen Jump 'n' Runs zu vermitteln, ohne allerdings aufwändige Level entwickeln und abspeichern zu müssen. Damit das möglich wird, müssen die im Spiel erscheinenden Hindernisse fortlaufend neu generiert werden. Dies wird durch die integrierte „Random“ ROM Routine, die einen zufälligen 8 Bit Zahlenwert liefert, möglich (dazu später mehr). Der Spieler wird dabei herausgefordert im richtigen Zeitpunkt abzuspringen um dem herannahenden Hindernis auszuweichen. Schafft er es nicht, verliert er einen Lebenspunkt. Fallen seine Lebenspunkte auf den Wert Null, ist das Spiel verloren.

B. Programmierung

Nun beginnt der eigentliche Hauptteil dieser Arbeit. In den folgenden Kapiteln wird der Quellcode und seine Funktionsweise erläutert. Da es aus Platzgründen nicht möglich ist den vollständigen Quellcode innerhalb dieses Kapitels vorzustellen, wird er in einzelne Funktionsblöcke unterteilt und

Stück für Stück erörtert. Der Text geht nur auf die wichtigsten Programmbestandteile ein. Der vollständige Quelltext kann im Anhang B. eingesehen werden. Neben dem Code werden in diesem Teil der Arbeit auch einige Grafiken genutzt, um das Programm leicht verständlich präsentieren zu können.

So gibt beispielsweise Abbildung 4 einen guten Überblick über den Aufbau des Programmcodes. Zunächst beginnt er mit einem Definitions-Abschnitt (**Define Section**). In diesem Abschnitt werden benötigte Variablen sowie vom Programm genutzte ROM-Routinen definiert. ROM Routinen sind bereits im Speicher der Vectrex enthaltene Funktionen die häufig von anderen Programmierern genutzt werden.

Define Section ... ROM Routinen ... Variablen
Header Section ... Startscreen
Code Section ... Startumgebung ... MAIN() ... Subroutinen > Leveldiff > Life > Player > Field > Level > User_action > Col ... Spielende ... Textanzeige (Daten) ... Zeitschleife ... Vektoren (Daten) ... Sound (Daten)

Abbildung 4. Quelltext Übersicht

In dem als **Header Section** bezeichneten Abschnitt werden zum einen eben diese ROM Routinen mit in den Quelltext eingebunden und zum anderen wird der Startbildschirm des Spiels geladen.

Die ersten Zeilen Quelltext die nun näher betrachtet werden befinden sich in der **Code Section**. Nachdem die oben definierten Variablen mit konkreten Werten in der Startumgebung initialisiert wurden, beginnt die Main des Spiels.

1) *Main*: Wie jedes moderne Programm beginnt der funktionale Teil des Spiels mit einer Main-Funktion. Diese Main wird in der Regel genutzt um einzelne Programmfunktionen aufzurufen und bildet eine Art Inhaltsverzeichnis. Was besonders in der Assemblerprogrammierung hervorsteht, ist hier ersichtlich. Die Main stellt in der Regel eine übergeordnete Schleife dar. Das heißt, alle hier aufgerufenen *Subroutinen* werden abwechselnd der Reihe nach in einer Endlosschleife aufgerufen. Die Main kann nur verlassen werden, wenn eine der Subroutinen nicht zurück zur *main_loop* springt.

Die Main des Programmes *J Game* (siehe Listing 1) ist wie folgt aufgebaut - zunächst beginnt sie damit eine Reihe von ROM Routinen zum Abspielen von Sounddateien aufzurufen.

Der hier geladene Sound wird allerdings nur abgespielt, wenn zuvor ein Flag, das sogenannte „Vec_Music_Flag“ mit einer #1³ geladen wurde. Offensichtlich handelt es sich um einen Sprungton, der nur geladen wird, wenn die Spielfigur zu einem Sprung ansetzt. Die mit dem Befehl **LDU** geladenen Daten **PING1** können im Quelltext im Bereich **Sound (Daten)** gefunden werden. Die dort abgespeicherten Töne werden nun vom Soundprozessor der Vectrex verarbeitet und wiedergegeben.

Ein wichtiger Funktionsaufruf, der nicht nur für die Verarbeitung von Tönen sondern auch für die Darstellung von Abbildungen eminent wichtig ist, findet sich in Zeile 9 des Codeabschnittes. Die hier aufgerufene Subroutine **WAITRECAL** rekaliert die Vectrex und verhindert damit, dass Vektoren aus dem Bild driften. Da das **WAITRECAL** ein Timeout enthält, darf die Funktion nur einmal pro Refresh-Zyklus aufgerufen werden, da es sonst zu einem sehr starken Flackern des Bildes und Störungen im Ton kommt.

Listing 1. Main

```

1  ; MAIN();
2  main:

4  main_loop:
5  ;** Abspielen des Sounds und rekaliertieren der VECTREX
6  JSR  DPTOC8
7  LDU  #PING1      ; Musiknoten fuer Sprung laden
8  JSR  INITMUSICCHK
9  JSR  WAITRECAL    ; max. einmal pro refresh aufrufen!
10 JSR  DOSOUND      ; Sound abspielen
11 ;** Level (Schwierigkeitsgrad)
12 JSR  leveldiff
13 ;** Spielerleben
14 JSR  life
15 ;** Spielfigur
16 JSR  player
17 ;** Spielfeld
18 JSR  field
19 ;** LEVEL Bau (hier wirds kompliziert!)
20 JSR  level
21 ;** Benutzereingabe lesen, Springen und Sound
22 JSR  user_action
23 ;** Kollisionserkennung und Spielende
24 JSR  col

26  LBRA main_loop    ; und zurueck zum Anfang

```

Nach dem Abspielen des Sprungtons, ruft die Main alle wichtigen Subroutinen des Spiels mit dem Befehl **JSR** (Jump to Subroutine) auf. Nachdem alle Subroutinen durchlaufen und jeweils mit dem Befehl **RTS** (Return from Subroutine) abgeschlossen wurden, endet die Main-Schleife mit dem Befehl **LBRA** (Long Branch) und kehrt zum Schleifenanfang zurück.

Der Reihe nach werden nun alle Subroutinen besprochen, allerdings lohnt es, die eigentliche Reihenfolge etwas zu ändern um den Aufbau des Spiels besser zu verstehen. Die nächsten Abschnitte beschreiben die Subroutinen **field** (baut das statische Spielfeld), **player** (baut die Spielfigur) und **user_action** (bewegt die Spielfigur). Erst im Anschluss daran wird die deutlich kompliziertere Funktion **level** (erzeugt und bewegt die Hindernisse) besprochen. Abschließend werden

³Der Syntax des 6809 entsprechend werden Dezimalzahlen in dieser Arbeit mit einem Rautezeichen #, Hexadezimalzahlen mit einem Dollar-Zeichen \$ und Binärzahlen mit einem Prozentzeichen %% maskiert.

die Hauptfunktionen von **col** (Kollisionserkennung und Spielende), **leveldiff** (Schwierigkeitsgrad) und **life** (Spielerleben) erörtert.

2) *Spielfeld*: Die Subroutine **field** ist relativ simpel und schnell erläutert, denn es handelt sich dabei lediglich um ein Rechteck, bestehend aus acht Vektoren mit den Ausmaßen 120 mal 140⁴. Gezeichnet wird dieses Rechteck mit Hilfe der ROM Routinen **MOVEPEN** und **MOVEDRAW**. Zunächst wird der Kathodenstrahl der Vectrex zur Mitte des Bildschirms und dann mit einer Skalierung von #128 und eine Intensität von #30 mit Hilfe der in **sfield** gespeicherten Vektoren über den Bildschirm bewegt. Startpunkt des ersten Vektors ist (60, -70), wobei der erste Wert der Y-Koordinate und der zweite Wert der X-Koordinate entspricht. Warum diese künstliche Verkleinerung des Spielfeldes notwendig ist, wird zu einem späteren Zeitpunkt deutlich.

Listing 2. Field

```
1 ; Spielfeld
2 field:
3 JSR RESETOREF ; Springe zur Mitte
4 LDA #30
5 JSR INTENSITY
6 LDA #0
7 LDB #0
8 JSR MOVEPEN
9 LDY #sfield
10 LDA #8 ; Anzahl der Vektoren
11 LDB #128 ; Skalierung
12 JSR MOVEDRAW
13 RTS ; verlassen der Subroutine
```

Neben dem eigentlichen Spielfeld gibt es natürlich noch weitere Elemente des Spielinterfaces. Abbildung 5 zeigt ein erstes Modell mit Spielfigur, Hindernissen, sowie der Anzeige für Leben, Level und Spielanweisung.

3) *Spielfigur*: Der Quelltext der zur Darstellung der Spielfigur (Subroutine **player**) dient, ähnelt dem des Spielfeldes. Allerdings sind es hier 13 Vektoren die zur Abbildung des „Strichmännchens“ notwendig sind. Die Figur befindet sich nicht, wie zu erwarten auf dem Koordinatenursprung, sondern wird etwas versetzt bei Punkt (0, -10) gezeichnet. Dieser Versatz liegt in der Kollisionserkennung und der Art und Weise der Hindernisgenerierung begründet. Auch dies wird später im Kapitel **Level Design** erörtert.

Listing 3. Player

```
1 LDA posy ; Lade Y Koordinate
2 LDB #-10 ; Lade X Koordinate,
3 JSR MOVEPEN ; Figur steht auf -10
4 LDY #sfigur
5 LDA #13 ; Anzahl der Vektoren
6 LDB #128 ; Scaling
7 JSR MOVEDRAW
```

Ein weiterer Unterschied findet sich in der Variable **posy**. Dieser Wert ist variable und sorgt für eine Verschiebung der Figur in der Höhe, falls der Spieler den Button 1 betätigt und damit einen Sprung auslöst (siehe Listing 3).

⁴Bei diesen Werten handelt es sich streng genommen um Zeiteinheiten die je nach Skalierung entscheiden wie lange der Kathodenstrahl sich in eine bestimmte Richtung bewegt. Da maximal ein 8 Bit Register zur Verfügung steht, sind somit die Werte -126 bis 126 adressierbar.

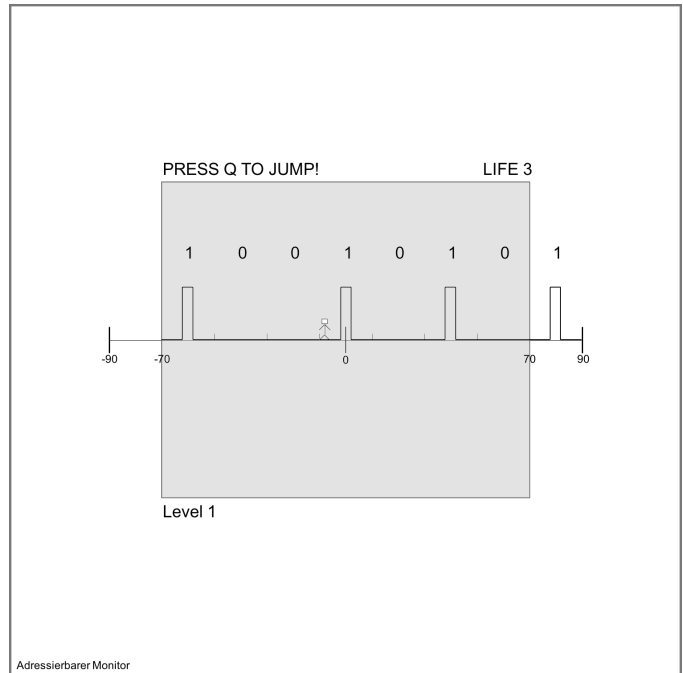


Abbildung 5. Spiel-Interface

4) *Sprung*: Der Sprung der Figur wird durch das Drücken von Button 1 des ersten Controllers der Vectrex oder im Emulator durch das Drücken der Taste Q ausgelöst. Die Prüfung der Buttons erfolgt in der Subroutine **user_action** (siehe Listing 4). Damit der Sprung erkannt wird, muss die ROM Routine **READBUTTONS** verwendet werden. Sie prüft ob einer der Buttons gedrückt wurde und speichert das Ergebnis im Akkumulator.

Listing 4. Jump (Quelltext gekürzt)

```
1 ...
2 JSR READBUTTONS
3 CMPA #$00 ; irgendein Button gedrueckt?
4 LBEQ no_button ; kein Button
5 BITA #$01 ; wird Button 1 gedrueckt?
6 LBEQ false_button ; wenn nicht dann false_button
7 ...
9 ;*** Sprung
10 jump:
11 LDA jumpstate ; fallen oder steigen wir?
12 CMPA #1 ; 0 = wir steigen, 1 = wir fallen
13 LBEQ jumpdown
15 LDA posy ; hoch springen
16 INCA ; ein Stueck steigen
17 STA posy
19 LDA posy
20 CMPA #40 ; Ist Spielfigur schon oben angekommen?
21 LBNE jumpdone
23 INC jumpstate ; Figur ist oben angekommen,
24 ; aendere Jumpstate
26 LBRA jumpdone ; springe ans Ende
28 jumpdown:
29 ...
```


Der im Akku gespeicherte Wert kann anschließend mit dem Befehl **CMPA** ausgewertet werden (siehe Listing 4). Für den Fall, dass kein Knopf, bzw. der falsche Knopf betätigt wurde, kommt es zu einer Textausgabe, die entweder zum Drücken der Taste Q auffordert („PRESS Q TO JUMP!“) oder auf den Fehler hindeutet („WRONG BUTTON!“).

Nur wenn der richtige Knopf betätigt wurde kommt es zum eigentlichen Aufruf der Sprungroutine. Hier wird zunächst geprüft, ob die Spielfigur sich bereits im Sprung befindet und wenn ja ob sie sich nach oben oder nach unten bewegt. Bei jedem Durchlauf wird die Y-Position der Figur um eine Stelle inkrementiert oder zweimal dekrementiert. Die maximale Sprunghöhe ist beim Wert #40 erreicht, danach fällt die Figur durch die zweifache Dekrementierung mit doppelter Geschwindigkeit bis zum Absprungpunkt zurück.

In diesem Code-Fragment ist nicht enthalten wie geprüft wird, ob die Figur sich bereits im Sprung befindet. Dies geschieht durch die Variable **action**, welche wie ein Flag eingesetzt wird. Außerdem wird hier das bereits vorgestellte „Vec_Music_Flag“ gesetzt, um am Anfang der Main einen Ton auszulösen falls die Figur zu einem Sprung ansetzt.

5) *Level Design*: Der folgende Abschnitt dokumentiert das Kernstück und den vermutlich kompliziertesten Teil des Programmes. Die etwas unglücklich bezeichnete Subroutine **level** dient dem Programm zum fortlaufenden Generieren von Hindernissen sowie der Bewegung dieser von rechts nach links.

Grundlage eines jeden Spielabschnittes ist eine zufällig gewählte 8 Bit Zahl. Jedes Bit spiegelt dabei einen Teil des Levels wieder. Eine 0 entspricht einer geraden Linie und eine 1 einem Hindernis bestehend aus fünf Vektoren. Jedes Teilstück ist genau 20 Einheiten lang, d.h. auf das bereits oben erläuterte Spielfeld passen genau sieben Levelteile. Das achte Levelstück ragt jeweils links oder rechts aus dem Spielfeld heraus (siehe Abbildung 5). Dies ist notwendig um bei der Verschiebung des Levels keine Lücke entstehen zu lassen. Aus diesem Grund wurde das Spielfeld auch verkleinert und nutzt nicht den gesamten Bildschirm. Das Modell in Abbildung 5 zeigt exemplarisch einen Levelabschnitt, bestehend aus vier Hindernissen (1) und vier geraden Linien (0). Das vierte Hindernis befindet sich zu dem Zeitpunkt noch außerhalb des Spielfeldes und wird durch eine Schleife im nächsten Schritt in das Spielfeld geschoben. Das führende Hindernis wird dabei links aus dem Spielfeld bewegt und verschwindet sobald es den Punkt (0, -90) erreicht hat. Während dieses Teilstück verschwindet, erscheint auf der rechten Seite ein neues Element und der Zyklus beginnt von neuem. Dieser Shift wird in Abbildung 6 auf der rechten Seite deutlich.

Um diese Funktionalität im Assemblercode umzusetzen sind drei Variablen notwendig. Zum einen die Variable **templ** (für temporäres Level). Sie enthält die 8 Bit die zur Zeit im Spielfeld gezeichnet und vom Punkt (0, -70) bis (0, -90) bewegt werden. Dieser Wert wird während des Zeichnens und der Bewegung nicht verändert. Die Variable **activl** (für aktives Level) enthält das jeweils nächste Levelteil und überträgt es an **templ** sobald dieses den äußeren Rand erreicht hat (siehe

Abbildung 6). Dabei ist zu beachten, dass ein neues Teilstück immer von links in die Variable **templ** geschoben wird, da das höchstwertige Bit (engl. most significant bit, MSB) jeweils als Letztes gezeichnet wird (Rechtssshift der Variable).

Der linke Teil von Abbildung 6 zeigt, wie die Variable **templ** interpretiert und gezeichnet wird. Die ersten sieben Bit enthalten eine 0, daher wird bis zum Spielfeldrand eine gerade Linie gezeichnet (a&b). Das achte Bit enthält eine 1 und ein neues Hindernis entsteht (c). Danach kommt es zur Bewegung des Levels und neue Elemente werden aus der Variable **nextl** (für nächstes Level) geladen.

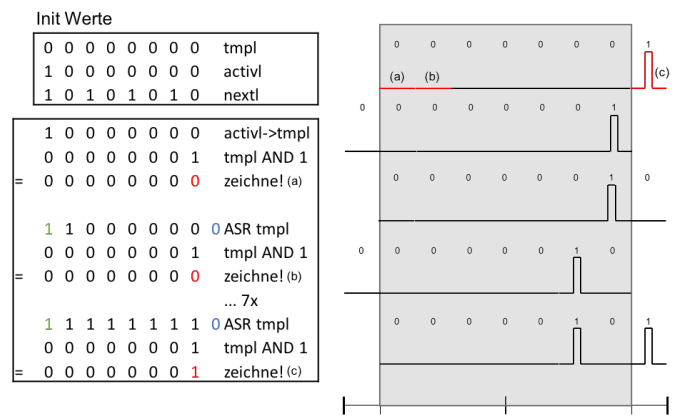


Abbildung 6. TempLevel und ActivLevel Shift

Dies ist die dritte benötigte Variable. Sie wird mit Hilfe der ROM Routine *Random* initialisiert und enthält jeweils die neuen Bauelemente, die ins aktive Level (**activl**) geladen werden. Zusammengefasst werden die Variablen **templ** und **activl** zum Zeichnen des Levels und **nextl** zum Generieren neuer Elemente benötigt.

Der Quelltext in Listing 5 zeigt zunächst die Art und Weise wie das Programm entscheidet, ob ein Hindernis oder eine gerade Linie gezeichnet wird. Geprüft wird, wie in Zeile zwei zu sehen, ob das niedrigstwertige Bit von **templ** eine 1 oder eine 0 ist. Je nachdem wird dann im nächsten Schritt die Zeichenroutine für eine Linie oder ein Hindernis angesprochen.

Listing 5. Level zeichnen (Quelltext gekürzt)

```

1 ...
2 LDA templ ; pruefe die Bits auf Hindernisse
3 ANDA #%00000001 ; ist das erste Bit eine 1
4 BNE on ; wenn ja springe zum Hop

6 LDX #line ; wenn nicht,
7 LDA #1 ; dann zeichne eine gerade Linie
8 LDB scale ; Skalierung
9 JSR MOVEDRAW
10 LBRA 12 ; springe danach zur
11 ; abschliessenden Linie
12 on:
13 LDX #hop ; und zeichne ein Hindernis
14 LDA #3
15 LDB scale ; Skalierung
16 JSR MOVEDRAW
17 ...

```

Das Quellcodefragment in Listing 6 kümmert sich um die Verschiebung des Levels vom rechten zum linken Rand. Auch

hier spielen wieder die Grenzen des Spielfeldes eine wichtige Rolle. Die Position des Levelabschnittes wird in der Variable **levelx** gespeichert. Außerdem wird an dieser Stelle eine Zeitschleife (**time**) aufgerufen. Sie ist dafür verantwortlich ob das Level sich schneller oder langsamer bewegt.

Listing 6. Level bewegen (Quelltext gekürzt)

```

1  ;* Level Bewegung (levelx)
2  DEC levelx      ; Level um 1 nach links schieben
3  JSR time        ; Zeitschleife fuer Speed aufrufen

5  LDA levelx      ; pruefe ob Levelteil
6  CMPA #-90       ; Level verlassen hat
7  LBNE leveldone  ; wenn nicht dann springe zum Ende
8                  ; wenn doch, verwirfe
9  LDA #-70        ; altes Levelteil und
10 STA levelx      ; springe wieder an den Rand
11 ...

```

Das letztes Codefragment in diesem Abschnitt beschreibt wie aus dem Zufallswert **nextl** ein Element des Levels wird. Zunächst muss **activl** um eine Stelle nach rechts verschoben werden. Dies geschieht mit dem Befehl **ASR** (siehe Listing 7) und schafft Platz für ein neues Element. Allerdings muss dieser freigewordene Platz mit einer 0 überladen werden, da **ASR** den Wert an der höchstwertigen Stelle unangetastet lässt. Nachdem **activl** vorbereitet ist, wird **nextl** geladen und auf ähnliche Weise vorbereitet. Im Unterschied zu **activl** werden hier aber alle Stellen bis auf die Höchstwertige verworfen (**ANDA #10000000**). Mit einer logischen ODER Funktion können nun beide Werte verbunden und ins aktive Level abgespeichert werden. Die anschließende Linksverschiebung von **nextl** sorgt dafür, dass im nächsten Durchlauf ein neues Element geladen wird. Erst nachdem acht Elemente auf diese Weise neu geladen wurden, wird eine neue Zufallszahl mit **JSR RANDOM** gewählt.

Listing 7. Levelstück einarbeiten (Quelltext gekürzt)

```

1  ;* neues Levelstueck einarbeiten (activl)
2  ASR activl      ; rechtssshift um Platz
3  LDA activl      ; fuer neuen Wert zu schaffen
4  ANDA #01111111
5  STA activl
6  LDA nextl      ; lade generierte Zufallszahl
7  ANDA #10000000 ; loesche alles, ausser MSB
8  ORA activl     ; MSB ins aktive Level
9  STA activl     ; speichere neues aktive Level
10 ASL nextl      ; shift um naechstes Bit zu erhalten
11 ...
12 rl:
13 JSR RANDOM      ; und neue Zufallszahl waehlen
14 ANDA difficulty ; Schwierigkeitsgrad einbeziehen
15 LBEQ rl
16 STA nextl

18 leveldone:
19 RTS             ; verlassen der Subroutine

```

6) **Schwierigkeitsgrad**: Diese Prozedere enthält eine Besonderheit. So wird die zufällig gewählte Zahl zunächst mit einer Variable namens **difficulty** und einem logischen UND maskiert. Dies ist neben der Geschwindigkeit der erscheinenden Hindernisse (siehe **time** im Quellcode), der zweite eingebaute variable Schwierigkeitsgrad. Je nach Level sorgt diese Variable dafür, dass mehr oder weniger Hindernisse erscheinen können (siehe Tabelle III). Der Ablauf sieht dabei vor, dass jedes Level

langsam beginnt und seine Geschwindigkeit nach und nach steigert bis das nächsten Level erreicht ist.

Level 1	##00010001	bis zu zwei Hindernisse
Level 2	##10010010	bis zu drei Hindernisse
Level 3	##11001100	bis zu vier Hindernisse
Level 4	##11101110	bis zu sechs Hindernisse

Tabelle III
SCHWIERIGKEITSGRADE

7) **Leben**: Ein weiteres zentrales Element eines jeden Jump 'n' Run Spiels sind die Leben des Spielers. Auch das *J Game* bietet diese Funktion. Dem Spieler werden dabei 3 Lebenspunkte eingeräumt, die er verliert, wenn er es nicht schafft über ein Hindernis zu springen. Gespeichert werden diese in der Variable **lifecount**. Dieser Wert wird in der Subroutine **life** ausgewertet und in der Routine **col** im Fall einer Kollision dekrementiert.

8) **Kollision und Spielende**: Die Subroutine mit dem Namen **col** beinhaltet zum einen die Kollisionserkennung zwischen der Spielfigur und den Hindernissen und zum anderen den Aufruf des Spielendes. Dies ist die letzte Subroutine, die in der Main aufgerufen wird.

Die Routine prüft, ob ein Hindernis im Bereich der Spielfigur auftaucht (siehe Listing 8, Zeile 2 und 3) und entscheidet, ob eine Kollision möglich ist. Hier liegt auch die Verschiebung der Figur auf die Position (0, -10) begründet, denn in diesem Bereich ist das fünfte Bit von **activl** auszuwerten. Wenn ein Hindernis sich der Figur annähert, sollte diese mindestens eine Y-Position von größer 20 haben, da es sonst zu einer Kollision und damit zum Verlust eines Lebens kommt.

Listing 8. Kollisionserkennung (Quelltext gekürzt)

```

1  ; Kollisionserkennung
2  col:
3  LDA activl      ; pruefe auf Hindernis
4  ANDA #00001000
5  LBEQ coldone

7  LDA levelx      ; in diesem Bereich koennen wir
8  ADDA #74        ; auf ein Hindernis stossen
9  LBGT coldone    ; linke Grenze der Spielfigur

11 LDA levelx
12 ADDA #82        ; rechte Grenze der Spielfigur
13 LBLT coldone

15 LDA posy        ; hoffentlich sind wir hoch genug
16 SUBA #20        ; wenn nicht verlieren
17 LBGE coldone    ; wir ein Leben

18 ...
19 DEC lifecount   ; Mist, Leben verloren :(
20 LDA lifecount   ; haben wir noch welche?
21 LBGT coldone    ; wenn ja geht es weiter
22 LBRA gamelost   ; wenn nicht ist an dieser
23                ; Stelle Schluss, GAME OVER

24 coldone:
25 RTS             ; verlassen der Subroutine

```

Wenn der Lebenszähler auf den Wert Null gefallen ist, wird mit dem **LBRA** Befehl ein **gamelost** aufgerufen. Dieser Codeabschnitt befindet sich außerhalb der Main-Schleife und kann nur an dieser Stelle aufgerufen werden. Der Akku wird gelöscht und der Schriftzug „GAME OVER!“ erscheint. Außerdem wird eine Abschiedsmelodie gespielt und die Vectrex mit einem **WARMSTART** neu gestartet.

Abbildung 7 zeigt den Game-Over Schriftzug mit einem Bild der gestorbenen Spielfigur.



Abbildung 7. GAME OVER

C. Spielanleitung

Die Regeln des Spiels sind denkbar einfach. Ziel ist es den erscheinenden Hindernissen auszuweichen. Nachdem das Spiel gestartet ist, erscheint eine Weile kein Hindernis, diese Zeit wird dem Spieler gegeben um sich an das Sprungverhalten der Figur zu gewöhnen. Testsprünge sind an dieser Stelle ausdrücklich empfohlen.

Während die ersten beiden Level noch relativ einfach zu meistern sind, kann es spätestens im dritten Level zu zwei aufeinander folgende Hindernisse kommen. Um diese ohne den Verlust eines Lebens zu überspringen muss der Absprungpunkt sehr genau getroffen werden. Weiterhin kann es zu einer Kombination von Hindernissen kommen, die die Figur nicht ohne Verlust eines Lebens überspringen kann. Beispielsweise zwei Hindernisse mit einer Lücke in der Mitte. Hier muss der Spieler abwägen um höhere Verluste zu vermeiden.

Spätestens im vierten Level sollte der Spieler an der hohen Anzahl von Hindernissen scheitern. Abbildung 8 zeigt ein Bildschirmfoto des Emulators ParaJVE mit dem fertigen Spiel im Level 1.

III. PROBLEME UND HERAUSFORDERUNGEN

Eine generelle Herausforderung bei der Entwicklung mit maschinennahen Programmiersprachen ist es, die Denkweise höher Programmiersprachen abzulegen. So führte die Begrenzung auf 8 Bit Register für die Berechnungen immer wieder zu Problemen, zum Beispiel bei der Aufteilung des Spielfeldes. Somit war es vor allem die Einarbeitungszeit, unter anderem auch in die Syntax des 6809, die zu Verzögerungen in der Entwicklung führte.

Grundsätzlich stehen einige sehr gute Einstiegstutorials für die Vectrex im Internet zur Verfügung, allerdings decken diese nur die ersten Schritte ab. Außerdem ergibt sich aus der Tatsache, dass es sich um eine ca. 30 Jahre alte Konsole handelt deutlich weniger Möglichkeiten für Support durch

Online-Communitys, wie sie beispielsweise bei höheren und aktuellen Programmiersprachen Gang und Gäbe sind.

Die begrenzten Möglichkeiten der Register machen oft kreative Lösungen notwendig, wie anhand der Subroutine **level** ersichtlich.

Eine weitere Hürde stellen die Skalierungsfunktionen des Monitors dar. Da die Steuerung von Kathodenstrahlen deutlich von der Programmierung eines auf Pixel basierten Bildschirms abweichen und heute kaum noch gelehrt werden.

Eine sehr große Hilfe bei der Entwicklung des Spiels stellten die von anderen Entwicklern zur Verfügung gestellten Programme dar. So konnte beispielsweise das Problem der Rekalibrierung anhand des Spiels *VPONG* von Christopher Salomon gelöst werden. Für Einsteiger in die Vectrex Programmierung sollten die auf der Internetseite „http://www.playvectrex.comdesignit_f.htm“ zur Verfügung gestellten Quelltexte der erste Anlaufpunkt sein, um Struktur und Aufbau eines Vectrex Spieles zu erlernen.

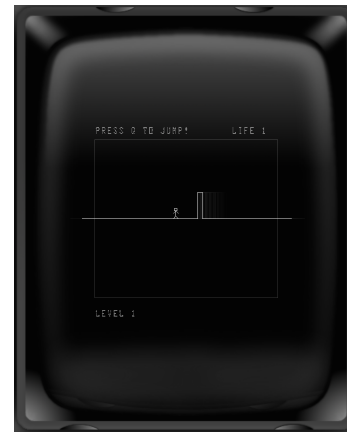


Abbildung 8. Spiel im Level 1

IV. FAZIT

Als Fazit der Arbeit lässt sich zusammenfassen, dass die Programmierung mit maschinennahen Programmiersprachen wie bspw. Assembler zu einem besseren Verständnis für höhere Programmiersprachen wie *JAVA*, *C++* oder *Perl* führt. Die Ziele der Arbeit konnten weitestgehend erfüllt werden und das Ergebnis ist ein funktionstüchtiges kleines Jump 'n' Run mit interessanten Funktionen wie einem zufälligen Leveldesign, einem Lebenszähler und einem ansteigenden Schwierigkeitsgrad.

LITERATUR

B. Quellcode

- [1] classickmateria. Retro Video Game system of the moment: Vectrex (1982). <http://coldslitherpodcast.com/2013/02/11/retro-video-game-system-of-the-moment-vectrex-1982/>, September 2015.
- [2] A. Eymann. Vectrex - Vorhang auf. <http://videospiegelgeschichten.de/vectrex.html>, September 2015.
- [3] Motorola. Motorola 6809. <http://www.classiccmp.org/dunfield/r/6809e.pdf>, September 2015.
- [4] vectrexmuseum.com. Vectrex System History - The Mini Arcade. <http://www.vectrexmuseum.com/vectrexhistory.php>, September 2015.
- [5] vectrex.wikia.com. Vectrex. <http://vectrex.wikia.com/wiki/Vectrex>, September 2015.

ANHANG

A. Liste der offiziellen Vectrex Spiele

- Armor...Attack
- Bedlam
- Berzerk
- Blitz (US Football)
- Clean Sweep
- Cosmic Chasm
- Flipper Pinball
- Fortress of Narzod
- Hyper Chase
- Minestorm
- Minestorm II
- Polar Rescue
- Pole Position
- Rip Off
- Scramble
- Soccer
- Solar Quest
- Space Wars
- Spike
- Star Castle
- Star Hawk
- Star Ship
- Web Warp
- Dark Tower

```

;*****
;
; The J-GAME by Jan Haasch 2015 @ HU Berlin
;
; http://vectrexmuseum.com/share/coder/html/bios.htm#F511 -> Link zu ROM Routinen
;
;*****

;*****
; DEFINE SECTION
;*****
; ROM Routinen:
WAITRECAL equ $f192
INTENSITY equ $f2ab
PRINTSTR equ $f37a
RANDOM equ $f511

RESET0REF equ $f354
MOVEPEN equ $f2fc
MOVEDRAW equ $f3b7

READBUTTONS equ $f1ba
DOSOUND equ $f289
INITMUSICCHK equ $f687
DPTOC8 equ $f1af
WARMSTART equ $f06C
PRINTSTRYX equ $f378
PRINTSTRHWYX equ $f373

; Einige Variablen:
posy equ $c882 ; y-position der Figur
jumpstate equ $c883 ; Status des Sprunges hoch/runter
action equ $c884 ; Status der Aktion an/aus
nextl equ $c885 ; Zufallszahl für Level 1
activl equ $c886 ; aktives Level das im Kasten gezeichnet wird
tmpl equ $c887 ; kopie vom aktiven Level
runcount equ $c888 ; Durchlaufcounter für Levelbau
levelx equ $c889 ; X-Koordinate für den Levelbau
scale equ $c890 ; Skalierungsfaktor für Levelbau
randomcount equ $c891 ; bestimmt ob neue Zufallszahl generiert werden muss

gameover equ $c892 ; Gameoverloop zum Anzeigen des GAME OVER Schriftzuges
speedcount equ $c893 ; Counter setzt die Zählschleife time und wird immer schneller
lifecount equ $c894 ; Anzahl der Leben sind initial 3
dead equ $c895 ; wir sterben nur einmal!
difficulty equ $c896 ; Schwierigkeitsgrad, Wert wird mit Randomzahl verundet
levelcount equ $c897 ; zeigt das aktuelle Level

;*****
INCLUDE "VECTREX.I"
; Start of vectrex memory with cartridge name...
ORG 0
;*****
; HEADER SECTION
;*****
fcb $67,$20
fcc "GCE 2015"
fcb $80 ; All text ends with $80

```

```

fdb $fd0d      ; Play song "$fe38" from ROM
fdb $f850      ; Width, height
fdb $30b8      ; y-position, x-position
fcc "THE J GAME"
fcb $80,$0     ; Init block ends with $0

```

```

;*****

```

```

; CODE SECTION

```

```

;*****

```

```

; Startumgebung setzen

```

```

LDA    #0
STA    posy      ; setze Y-Startkoordinate
STA    jumpstate ; setze Sprungstatus auf 0, d.h. steigen
STA    action    ; setze action auf 0, d.h. gerade wird nicht gesprungen
STA    runcount  ; setze den Durchlaufcounter auf 0
STA    tmp1      ; TempLevel, das gezeichnet wird
STA    activ1    ; ActiveLevel das in Temp geladen wird
STA    randomcount ; Counter für neu zu generierende Zufallszahl
STA    gameover  ; Schleifenwert für gameoverloop
STA    dead      ; binärwert für einmaliges sterben

LDA    #40
STA    speedcount ; gibt die Startgeschwindigkeit an, siehe auch Zeitschleife

LDA    #3
STA    lifecount  ; setze die Spielerleben auf 3

LDA    #1
STA    levelcount ; zeigt die Level von 1 bis 4

LDA    #-70
STA    levelx     ; setze Levelstart

LDA    #%00010001
STA    difficulty ; Schwierigkeitsgrad, steigt pro Level, mehr Hindernisse
möglich

JSR    RANDOM      ; initiale Random Zahlen bestimmen
ANDA   difficulty  ; um Level zu vereinfachen, Doppelsprünge vermeiden
STA    next1

LDA    #128        ; Skalierung für Levelbau wählen
STA    scale

```

```

; ROM Routinen für Schrift initialisieren

```

```

LDD    #$FC20      ; HEIGHT, WIDTH (-4, 32)
STD    Vec_Text_HW ; store to BIOS RAM location

```

```

;*****

```

```

; MAIN();

```

```

main:

```

```

main_loop:

```

```

;*** Abspielen des Sounds und rekalisieren der VECTREX

```

```

JSR    DPTOC8
LDU    #PING1      ; Musiknoten für Sprung laden
JSR    INITMUSICCHK
JSR    WAITRECAL    ; max. einmal pro refresh aufrufen!

```

```

        JSR     DOSOUND                ; Sound abspielen

;*** Level (Schwierigkeitsgrad)
        JSR     leveldiff

;*** Spielerleben
        JSR     life

;*** Spielfigur
        JSR     player

;*** Spielfeld
        JSR     field

;*** LEVEL Bau (hier wirds kompliziert!)
        JSR     level

;*** Benutzereingabe lesen, Springen und Sound (siehe auch Buttonhandling)
        JSR     user_action

;*** Kollisionserkennung und Spielende
        JSR     col

        LBRA    main_loop            ; und zurück zum Anfang

;*****
; Subroutinen:
;*****
; Levelhöhe
leveldiff:
        LDA     levelcount            ; prüfe aktuelles Level
        CMPA    #1
        BNE     ld2

        LDU     #level_1              ; Schrift anzeigen mit Leben
        JSR     PRINTSTRYX
        BRA     leveldiffdone

ld2:    LDA     levelcount
        CMPA    #2
        BNE     ld3

        LDA     #%10010010            ; difficulty Level 2
        STA     difficulty

        LDU     #level_2
        JSR     PRINTSTRYX
        BRA     leveldiffdone

ld3:    LDA     levelcount
        CMPA    #3
        BNE     ld4

        LDA     #%11001100            ; difficulty Level 3
        STA     difficulty

        LDU     #level_3
        JSR     PRINTSTRYX
        BRA     leveldiffdone

ld4:    LDU     #level_4
        JSR     PRINTSTRYX

        LDA     #%11101110            ; difficulty Level 4

```

STA difficulty

leveldiffdone: RTS ; verlassen der Subroutine

; Spielerleben

life:

LDA lifecount ; prüfe auf Anzahl der verbleibenden
CMPA #3 ; Leben
BNE li2

LDU #life_3 ; Schrift anzeigen mit Leben
JSR PRINTSTRYX
BRA lifedone

li2:

LDA lifecount
CMPA #2
BNE li1

LDU #life_2
JSR PRINTSTRYX
BRA lifedone

li1:

LDU #life_1
JSR PRINTSTRYX

lifedone: RTS ; verlassen der Subroutine

;*****

; Spielfigur

player:

JSR RESETOREF ; Spring zur Mitte
LDA #127
JSR INTENSITY ; Setze INTENSITY auf 127
LDA posy ; Lade Y Koordinate
LDB #-10 ; Lade X Koordinate, Figur steht auf -10
JSR MOVEPEN
LDX #sfigur
LDA #13 ; Anzahl der Vektoren
LDB #128 ; Scaling
JSR MOVEDRAW
RTS ; verlassen der Subroutine

;*****

; Spielfeld

field:

JSR RESETOREF ; Springe zur Mitte
LDA #30
JSR INTENSITY
LDA #0
LDB #0
JSR MOVEPEN
LDX #sfeld
LDA #8 ; Anzahl der Vektoren
LDB #128 ; Skalierung
JSR MOVEDRAW
RTS ; verlassen der Subroutine


```
*****
```

```
; LEVEL (hier wirds kompliziert!)
```

```
level:
    JSR    RESET0REF    ; Springe zur Mitte
    LDA    #127
    JSR    INTENSITY    ; Setze Intensity

    LDA    #0            ; Y-Koordinate
    LDB    levelx        ; X-Koordinate, dieser Wert wird geschoben
    JSR    MOVEPEN      ; springe zum Start

    LDA    activl        ; lade das generierte Stück Level zur
    STA    tmp1          ; Ausgabe ins TempLevel
```

```
; * Level aus Lines und Hops zeichnen (tmp1)
```

```
l1:    LDX    #dline      ; zeichne immer erst eine gerade Linie
    LDA    #1
    LDB    scale          ; Skalierung
    JSR    MOVEDRAW

    LDA    tmp1           ; prüfe die Bits des Random Bytes
    ANDA   #%00000001     ; ist das erste Bit eine 1
    BNE    on             ; wenn ja springe zum Hop

    LDX    #line          ; wenn nicht, dann zeichne eine gerade Linie
    LDA    #1
    LDB    scale          ; Skalierung
    JSR    MOVEDRAW
    LBRA   l2             ; springe danach zu abschließenden Linie
```

```
on:    LDX    #hop        ; und zeichne ein Hindernis
    LDA    #3
    LDB    scale          ; Skalierung
    JSR    MOVEDRAW
```

```
l2:    LDX    #dline      ; zeichne am Ende immer eine gerade Linie
    LDA    #1
    LDB    scale          ; Skalierung
    JSR    MOVEDRAW
```

```
; * Levelcounter für tmp1
```

```
    ASR    tmp1           ; rechts shift um Levelteil zu zeichnen

    INC    runcount       ; zähle den Durchlaufzähler bis 8 hoch
    LDA    runcount
    CMPA   #8
    LBNE   l1             ; ist 8 nicht erreicht, wird weiter gezeichnet

    LDA    #0             ; wenn 8 erreicht Counter zurück setzen
    STA    runcount
```

```
; * Level Bewegung (levelx)
```

```
    DEC    levelx         ; Level um 1 nach links schieben
    JSR    time           ; Zeitschleife für Delay aufrufen, ggf. anpassen

    LDA    levelx         ; prüfe ob Levelteil Level verlassen hat
    CMPA   #-90
```

```

    LBNE    leveledone    ; wenn nicht dann springe zum Ende

    LDA     #-70           ; wenn doch, verwirfe altes Levelteil und
    STA     levelx         ; springe wieder an den Rand

    LDA     #0             ; setze deadflag wieder auf 0
    STA     dead           ; jetzt können wir wieder sterben,
                           ; aber nur einmal pro Hindernis!

```

```

; * neues Levelstück einarbeiten (activl)

```

```

    ASR     activl         ; rechtsshift um Platz für neuen Wert zu schaffen
    LDA     activl
    ANDA    #%01111111
    STA     activl
    LDA     nextl          ; lade generierte Zufallszahl
    ANDA    #%10000000     ; lösche alles bis auf höchstwertiges Bit
    ORA     activl         ; lade höchstwertiges Bit ins aktive Level
    STA     activl         ; speichere das neue aktive Level
    ASL     nextl          ; linksshift um nächstes Bit zu erhalten

    LDA     speedcount     ; Speedcount um das Spiel
    CMPA    #1             ; zu beschleunigen, von init Wert bis 1
    LBEQ    s1             ; siehe time

```

```

    DEC     speedcount
    LBRA    s2             ; Level noch nicht geschafft, weiter nach unten

```

```

s1:    LDA     levelcount
    CMPA    #4             ; prüfe ob wir schon in Level 4 sind
    LBEQ    s2             ; wenn ja nichts weiter tun

    INC     levelcount     ; wenn nicht, erhöhe das Level um 1
    LDA     #40            ; und setze die Geschwindigkeit wieder auf 40
    STA     speedcount

```

```

s2:

```

```

; * neue Zufallszahl generieren (nextl)

```

```

    INC     randomcount    ; wir zählen wieder die Durchläufe
    LDA     randomcount    ; um nach 8 Bit eine neue Zufallszahl zu generieren
    CMPA    #8
    LBNE    leveledone     ; wenn Grenze noch nicht erreicht zum Ende

    LDA     #0             ; wenn erreicht, dann zurück setzen
    STA     randomcount

```

```

r1:    JSR     RANDOM      ; und neue Zufallszahl wählen
    ANDA    difficulty     ; Schwierigkeitsgrad einbeziehen
    LBEQ    r1
    STA     nextl

```

```

leveledone:    RTS          ; verlassen der Subroutine

```

```

; *****

```

```

; Benutzereingabe lesen, Springen und Sound

```

```

user_action:    JSR     RESET0REF

```

```

    LDA     action          ; prüfe ob bereits Q gedrückt wurde

```

```

    CMPA    #1
    LBEQ    jump                ; wenn ja dann Button nicht testen

    JSR     READBUTTONS

    CMPA    #$00                ; irgendein Button gedrueckt?
    LBEQ    no_button           ; kein Button
    BITA    #$01                ; wird Button 1 gedrueckt?
    LBEQ    false_button        ; wenn nicht dann false_button

    INC     action               ; wir beginnen action Sprung
                                ; ab jetzt gibt es kein Zurück mehr!

;*** Sound initialisieren
    LDA     #1                  ; Musikflag schalten für Sprungmusik,
    STA     Vec_Music_Flag      ; damit Musik gespielt wird, siehe Beginn
                                ; der mainloop

;*** Sprung
jump:
    LDA     jumpstate           ; fallen oder steigen wir?
    CMPA    #1                  ; 0 = wir steigen, 1 = wir fallen
    LBEQ    jumpdown

    LDA     posy                ; hoch springen
    INCA    posy                ; ein Stück steigen
    STA     posy

    LDA     posy
    CMPA    #40                 ; Ist Spielfigur schon oben angekommen?
    LBNE    jumpdone

    INC     jumpstate           ; Figur ist oben angekommen,
                                ; ändere Jumpstate

    LBRA    jumpdone            ; springe ans Ende

jumpdown:
    DEC     posy                ; runter fallen
    DEC     posy                ; ein Stück runter fallen

    LDA     posy                ; sind wir unten angekommen?
    LBNE    jumpdone            ; wenn nicht dann ans Ende

    DEC     jumpstate           ; wenn doch, dann Jumpstate = 0
    DEC     action              ; wir beenden den Sprung und aktivieren
                                ; die Eingabe

jumpdone:
    RTS                        ; verlassen der Subroutine

;*****
; Buttonhandling:
false_button:
    LDU     #false_button_string ; Falscher Button
    JSR     PRINTSTRYX           ; String schreiben
    LBRA    jumpdone            ; Zum Ende springen

no_button:
    LDU     #no_button_string
    JSR     PRINTSTRYX
    LBRA    jumpdone            ; Zum Ende springen

```

```

;*****
; Kollisionserkennung
col:      LDA      activl          ; prüfe auf Hindernis
          ANDA     #%00001000
          LBEQ     coldone

          LDA      levelx          ; in diesem Bereich können wir
          ADDA     #74              ; auf ein Hindernis stoßen
          LBGT     coldone          ; linke Grenze der Spielfigur

          LDA      levelx          ;
          ADDA     #82              ; rechte Grenze der Spielfigur
          LBLT     coldone

          LDA      posy            ; hoffentlich sind wir hoch genug
          SUBA     #20              ; wenn nicht verlieren wir ein Leben
          LBGE     coldone

          LDA      dead            ; ein Hindernis kann uns aber nur ein Leben
          LBNE     coldone          ; kosten, daher merken wir uns den Tot für
          LDA      #1              ; die letzten 20 Durchläufe
          STA      dead            ; wird weiter oben zurück gesetzt

          DEC      lifecount        ; Mist, Leben verloren :(
          LDA      lifecount        ; hoffentlich haben wir noch welche?
          LBGT     coldone          ; wenn ja geht es weiter
          LBRA     gamelost        ; wenn nicht ist an dieser Stelle schluss, GAME OVER

coldone:   RTS                    ; verlassen der Subroutine

;*****
; Spielende:
gamelost:  CLRA                    ; Lösche den Akku
          LDA      #1
          STA      Vec_Music_Flag  ; Lade 1 für neue Musik

gameoverloop: JSR      DPTOC8
          LDU      #musicb          ; lade Musik
          JSR      INITMUSICCHK     ; and init new notes
          JSR      WAITRECAL        ; sets dp to d0, and pos at 0, 0
          JSR      DOSOUND

          LDU      #game_over_string ; zeige GAME OVER Schriftzug
          JSR      PRINTSTRHWYX

;*** Spielfigur Abschiedsbild
          JSR      RESET0REF        ; Spring zur Mitte
          LDA      #127
          JSR      INTENSITY        ; Setze INTENSITY auf 127
          LDA      #0               ; Lade Y Koordinate
          LDB      #0               ; Lade X Koordinate
          JSR      MOVEPEN
          LDX      #sfigurt1
          LDA      #8               ; Anzahl der Vektoren
          LDB      #250             ; Scaling
          JSR      MOVEDRAW

          JSR      RESET0REF        ; Spring zur Mitte

```

```

LDA    #127
JSR    INTENSITY        ; Setze INTENSITY auf 127
LDA    #0                ; Lade Y Koordinate
LDB    #0                ; Lade X Koordinate
JSR    MOVEPEN
LDX    #sfigurt2
LDA    #5                ; Anzahl der Vektoren
LDB    #250              ; Scaling
JSR    MOVEDRAW

```

```

DEC    gameover
LDA    gameover
BNE    gameoverloop

```

```

reset:    JSR    WARMSTART        ; Starte die Vectrex neu ohne Reinitialisierung
                                   ; des OS (nicht schön, vll andere Lösung finden)

```

```

;*****

```

```

; Textanzeige:

```

```

no_button_string:

```

```

    DB 70,-75,"PRESS Q TO JUMP!", $80

```

```

false_button_string:

```

```

    DB 70,-75,"WRONG BUTTON!", $80

```

```

game_over_string:

```

```

    DB -10,50,50,-45,"GAME OVER!", $80

```

```

life_1:

```

```

    DB 70,30,"LIFE 1", $80

```

```

life_2:

```

```

    DB 70,30,"LIFE 2", $80

```

```

life_3:

```

```

    DB 70,30,"LIFE 3", $80

```

```

level_1:

```

```

    DB -70,-75,"LEVEL 1", $80

```

```

level_2:

```

```

    DB -70,-75,"LEVEL 2", $80

```

```

level_3:

```

```

    DB -70,-75,"LEVEL 3", $80

```

```

level_4:

```

```

    DB -70,-75,"LEVEL 4", $80

```

```

;*****

```

```

; Zeitschleife:

```

```

time:    LDA    speedcount        ; Speedcount beeinflusst Levelgeschwindigkeit

```

```

t1:      LDB    #0                ; und damit den Schwierigkeitsgrad

```

```

t2:      DECB    t2

```

```

        BNE     t2

```

```

        DECA    t2

```

```

        BNE     t1

```

```

        RTS                    ; verlassen der Subroutine

```

```

;*****

```

```

; Spielfigur:

```

```

sfigur:  fcb 0,0

```

```

        fcb 2,2

```

```

        fcb -2,2

```

```

        fcb 2,-2

```

```

        fcb 4,0

```

```

        fcb -2,2

```

```

        fcb 2,-2

```



```

fcb -2,-2
fcb 2,2
fcb 0,-1
fcb 2,0
fcb 0,2
fcb -2,0
fcb 0,-1

```

```

; Spielfigur Körper:

```

```

sfigurt1:fcb 0,0
          fcb 4,4
          fcb -4,4
          fcb 4,-4
          fcb 8,0
          fcb -4,4
          fcb 4,-4
          fcb -4,-4
          fcb 4,4

```

```

;Spielfigur Kopf:

```

```

sfigurt2:fcb 0,20
          fcb 0,-2
          fcb 4,0
          fcb 0,4
          fcb -4,0
          fcb 0,-2

```

```

; Spielfeld:

```

```

sfeld:  fcb 60,-70
        fcb 0,70
        fcb 0,70
        fcb -60,0
        fcb -60,0
        fcb 0,-70
        fcb 0,-70
        fcb 60,0
        fcb 60,0

```

```

; LEVEL:

```

```

line:   fcb 0,0
        fcb 0,4

```

```

dline:  fcb 0,0
        fcb 0,8

```

```

hop:    fcb 0,0
        fcb 20,0
        fcb 0,4
        fcb -20,0

```

```

;*****

```

```

; Sound

```

```

PING1:                                     ; die Musik ist "geliehen" aus Patriots von "John Dondzilla"

```

```

        fdb    $FD69
        fdb    $FD79
        fcb    $20,$0A
        fcb    0, $80

```

```

;*****

```

END main

;*****