Einführung in die Spieleprogrammierung anhand von 8-Player-Panic am VC-20

Arndt Mühlenfeld

25. Juni 2014

Dieser Artikel bietet eine Einführung in die Spieleprogrammierung auf Basis eines einfachen Mehrspielerspiels mit Synchronisierung auf die Bildwiederholrate und diskutiert die dazu benötigten Techniken zur Behandlung von Benutzereingaben, Aktualisierung des Spielzustands, sowie Ausgabe von Grafik und Sound.

1 Einleitung

Die Programmierung eines Spiels lässt sich unabhängig davon, welcher Art das Spiel ist, auf eine einfache Grundstruktur reduzieren. Das ergibt sich aus den Aufgaben, die ein Spiel hat.

- Das Programm verarbeitet Benutzereingaben (Tastatur, Joystick oder ein anderer Controller).
- Der Spielstand wird anhand der Eingaben neu berechnet.
- Der aktuelle Zustand wird angezeigt.

Aktuelle Spiele sind meist umfangreiche Projekte, die ein Team von Entwicklern beschäftigen und auch wenn der Quellcode einsehbar ist, für den interessierten Neueinsteiger nicht so leicht zu durchschauen. Oft wird auch auf bestehenden Projekten aufgebaut, beispielsweise wird man nicht alles im Bereich der Grafik neu erfinden, weil es auch im Open-Source-Bereich brauchbare Pakete zur 3D oder isometrischen 2D-Darstellung gibt.

Ein einfaches Spiel auf einer Plattform die komplett verstanden werden kann und nicht auf zusätzlichen Paketen beruht, die auch erstmal verstanden werden wollen, ist für die Einführung in die grundlegenden Konzepte wesentlich besser geeignet.

Abbildung 1: *8-Player-Panic (Titel) - einfaches Intro mit Rasterzeilensynchronisierter Animation der Hintergrundfarbe. Für die Hintergrundfarbe stehen beim VC-20 16 Farben zur Verfügung, die hier mit Ausnahme von Schwarz und Weiss zyklisch druchlaufen werden. Im Unterschied zum C-64 hat der VC-20 keine Graustufen, dafür aber sieben Regenbogenfarben in zwei Helligkeitsstufen (**Your friendly Colour Computer**).*

Das Spiel *8-Player-Panic* für den VC-20 ist so ein Spiel, weil es die Grundzüge der Spieleprogrammierung relativ einfach implementiert und gut zu verstehen ist, aber auch schon ein paar allgemeingültige Optimierungsmöglichkeiten demonstriert (Abbildung 1). 8-player-panic ist als schnell zusammengestellte Demonstration für den Joystickadapter entwickelt und kann vermutlich effizienter und mit weniger Speicherplatzbedarf implementiert werden. Die Struktur ist dadurch aber noch verhältnismässig klar.

In den folgenden Abschnitten werde ich kurz die Spielidee vorstellen und dann auf die Implementierung der einzelnen Aspekte eingehen und diese erläutern. Abschliessend wird die Fehlersuche anhand eines aufgetretenen Bugs bei der Darstellung des Gewinners untersucht und zuletzt eine Erweiterung des Spiels diskutiert, die von (Test-)Spielern angeregt wurde.

2 Spielidee

Das Spiel basiert auf dem Berzerk-Clone Spiel *AMOK* (Abbildung 2) bei dem der Spieler ein Strichmänchen pixelgenau auf dem zweidimensionalen Spielfeld bewegen kann und amoklaufende Roboter "ausschalten" muss. Wenn der Spieler die Umrandung des Spiels, Mauern oder Gegner berührt, stirbt er. Die Roboter und auch der Spieler schiessen pixelgroße Kugeln, können aber nur eine Kugel fliegen lassen, also kann erst wieder geschossen werden, wenn die Kugel auf ein Hindernis trifft und verschwindet.

Statt der Roboter gibt es nun bis zu neun Spieler, die unter den genannten Regeln gegeneinander antreten. Das ganze soll flimmerfrei umgesetzt werden und ist trotz der einfachen Vorgabe schon leicht herausfordend, da der VC-20 bekanntermaßen keine Sprites kennt. Im Original wird z.b. nur die eine Spielfigur pixelgenau bewegt, während die

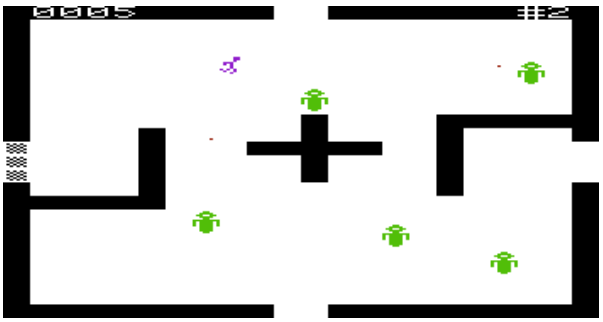

Abbildung 2: *AMOK ist ein Berzerk-Clone aus dem Jahr 1981 für die Grundversion des VC-20 (4K RAM). Der Spieler steuert das violette Strichmännchen und die grünen Figuren sind die gegnerischen Roboter. Die schwarzen Wände sind sowohl für den Spieler als auch die Roboter tödlich, weil sie laut Spielbeschreibung elektrisch sind. Die kleinen Punkte sind fliegende Kugeln.*

Roboter sich nur zeichenweise bewegen können. Bei einer anderen Variante *Super Amok* bewegen sich allerdings auch die Roboter pixelgenau.

Im Unterschied zum Einspielerspiel gibt es eine Startphase, in der sich Spieler zuschalten können und einen Schluß, bei dem der Gewinner bekannt gegegeben wird (Abbildung 2). Alle diese Phasen können mit denselben Methoden arbeiten, die sich im Programm in einer ständig durchlaufenen Abfolge des klassischen Prinzips *Eingabe-Verarbeitung-Ausgabe* niederschlagen: der **Hauptschleife**.

Abbildung 3: *Am Ende eines Spiels darf eine Schlusssequenz oder eine Darstellung des Ergebnisses natürlich nicht fehlen. Bei 8-player-panic wird der Gewinner in aus Blockzeichen zusammengesetzten Buchstaben groß angezeigt.*

3 Die Hauptschleife

Jedes Programm wartet auf Benutzereingaben, verarbeitet diese und gibt die daraus resultierenden Änderungen am Systemzustand aus. In modernen Programmen erfolgt das meistens Ereignis-gesteuert, d.h. man muss den Systemzustand nicht ständig abfragen, sondern wird gezielt über relevante Ereigniss informiert.

Spiele und auch Demos haben allerdings andere Anforderungen, weil sie nicht nur von äusseren Ereignissen abhängen sondern den Programmzustand selbst weiter vorantreiben. Der Spielzustand entwickelt sich auch ohne Benutzereingaben weiter.

Das Grundprinzip eines jeden Spiels ist daher, periodisch nachzusehen, ob eine Benutzereingabe erfolgte, den nächsten Spielzustand zu berechnen und das "Spielfeld" zu aktualisieren. Spielfeld kann in diesem Zusammenhang auch die aktuelle Ansicht in einer dreidimensionalen Spielwelt sein.

Die Hauptschleife von *8-player-panic* ist sehr einfach und übersichtlich:

```
jsr waitframe ; synch frame
jsr nextfrm   ; switch to next frame
jsr readjoy   ; read joysticks
jsr execgame  ; update game state
jsr calcspr   ; prepare objects
jsr drawfrm   ; draw frame
```

Die ersten beiden Zeilen dienen der Synchronisation mit der Bildwiederholung und werden im Abschnitt 6 näher erläutert. Interessant ist an dieser Stelle, dass die Synchronisation auf die Bildwiederholfrequenz bei PAL bedeutet, dass die Schleife 50 mal in der Sekunde durchlaufen wird. Daher sollte die Ausführung eines Durchlaufs nie länger als ca. 20 Millisekunden brauchen (genauer: 22000 Taktzyklen).

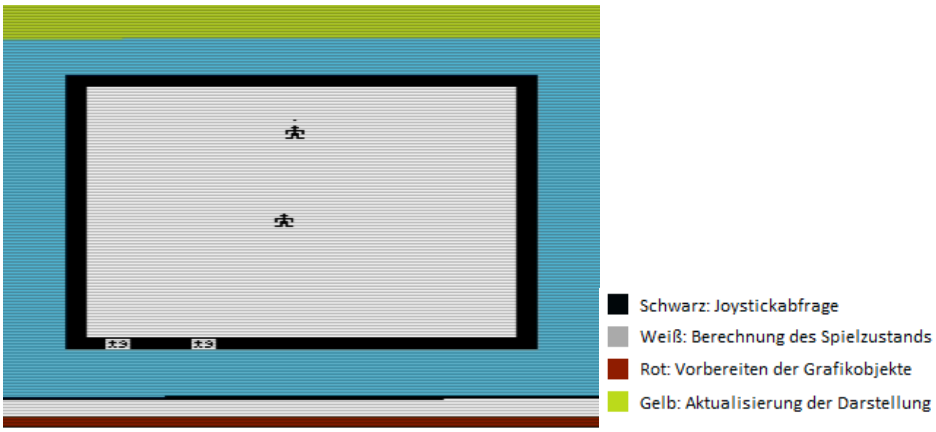

Abbildung 4: *Actionszene mit zwei Spielern. Für diesen Screenshot wurde für jeden Schritt der Hauptschleife die Hintergrundfarbe gewechselt, damit man sieht, wieviel Zeit die einzelnen Programmteile brauchen und ob sich das ganze in einem Bildwechsel ausgeht.*

Abbildung 5: *Für Mehrspielerspiele benötigt man eine nicht geringe Menge an Joysticks. Zum Glück gibt es keinen Mangel an alten digitalen Joysticks, die bis vor einiger Zeit tatsächlich noch in rauhen Mengen auf jedem Flohmarkt zu finden waren.*

Die dritte Zeile fragt die aktuellen Joystickzustände ab (Abschnitt 4), ermittelt also die Spielereingaben. Dann wird der Spielzustand aktualisiert (Abschnitt 5), die Spielobjekte und schliesslich der nächste *Frame* berechnet (wieder Abschnitt 6).

Dem aufmerksamen Leser wird auffallen, dass die Tonausgabe hier nicht vorkommt. Es ist natürlich möglich einen Aufruf in eine victracker-basierte Abspielroutine einzufügen[1] und das wird in der Intro des Spiels auch auch gemacht, aber dies würde mit den Geräuscheffekten des Spiels kollidieren, die der Einfachheit halber in einer eigenen Interrupt-Routine behandelt werden.

4 Benutzereingaben

Eingabegeräte die abgefragt werden müssen sind diverse Joysticks und die Tastatur. Das Prinzip ist bei beiden dasselbe, da die Joysticks digital sind, d.h. jede Hauptrichtung (Oben, Unten, Rechts, Links) und der Feuerknopf je einen Taster schliessen und die Tastaturtasten dasselbe tun. Bei mehreren Joysticks und auch der Tastatur ist es nicht

[1] VIC-Tracker-Songs und auch SIDs sind üblicherweise auf die Bildwiederholfrequenz synchronisiert, so dass sie in diese Hauptschleife gut integriert werden könnten.

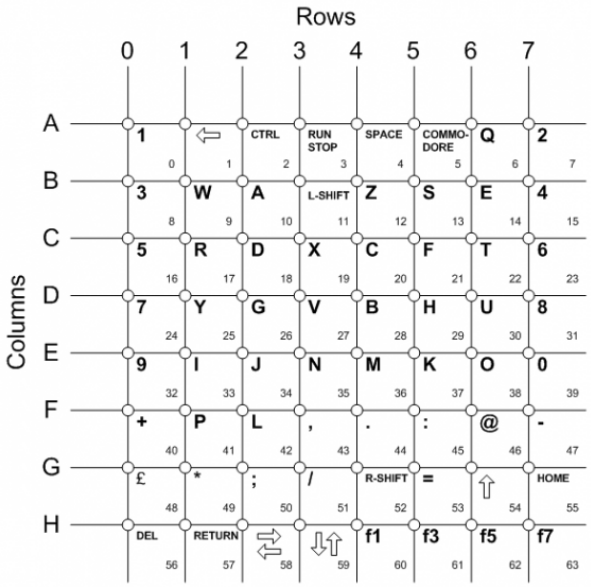

Abbildung 6: *Tastaturmatrix des VC-20 und C-64. Die Tasten sind in einer Matrix angeordnet und schliessen den Kontakt zwischen horizontaler und vertikaler Leitung. Dadurch können die auszulesenden Reihen über die Ausgänge einzeln oder in Gruppen ausgewählt werden und es werden zum Auslesen der 64 Tasten nur 16 Leitungen benötigt. Man könnte die acht Ausgänge mit einem Multiplexer auf drei reduzieren, könnte dann aber auch nur jeweils eine Reihe auslesen. Ohne Multiplexer ist es dagegen einfach festzustellen, ob irgendeine der 64 Tasten gedrückt wurde.*

mehr sinnvoll für jede Taste eine eigene Eingangsleitung zu haben, denn das wären bei der Tastatur des VC-20 oder C-64 65 Leitungen und für jeden Joystick fünf.

Der VC-20 hat aber insgesamt nur 32 digitale Ein- und Ausgänge (beim C-64 40), die aber für alle Peripheriegeräte reichen müssen. Daher arbeiten Tastatur und Mehrfach-Joystickadapter mit Multiplexing. Dabei wird ein Teil der verfügbaren Ausgänge benutzt, um festzulegen, was mit einer Gruppe von Eingängen gelesen werden soll. Dadurch kommt man bei der Tastatur für 64 Tasten mit acht Ausgängen und acht Eingängen aus, um alle Tastenzustände zu ermitteln. Der Trick ist, die Tasten in Achtergruppen zu unterteilen, die auf die acht Eingänge gelegt werden und mit den acht Ausgängen die auszulesende Gruppe zu addressieren (Abbildung 6).

Für den 8-Player-Adapter gilt das analog, d.h. der auszulesende Joystick wird über Ausgänge ausgewählt. Im Unterschied zur Tastatur wird dabei die Auswahl über 1-aus-8 Decoder umgesetzt, so dass statt acht nur drei Leitungen zur Adressierung des Joysticks benötigt werden ($2^3 = 8$).

Unabhängig von den Ein- und Ausgangsleitung muss bei gleichzeitig drückbaren Tasten der Zustand im Programm mit einem Bit je Taste kodiert werden. CBM-Basic tut das nicht, was man sehr leicht überprüfen kann, indem man eine Taste der Tastatur gedrückt hält und dann beliebige andere Tasten drückt und wieder loslässt.

5 Spielzustand

Der Spielzustand bei 8-Player-Panic besteht im wesentlichen aus folgenden Eigenschaften jedes Spielers:

- Position (X,Y)
- Darstellungsform (Steht, Läuft in Richtung R, Stirbt)
- Farbe
- Spieler aktiv
- Spieler getroffen
- Restliche Leben

Dieser Zustand wird in jedem Durchlauf der Hauptschleife neu berechnet. Die Position und Darstellung der Spielfigur sind dabei vom aktuellen Joystickzustand abhängig.

Dieser wird von der Einleseroutine bit-codiert geliefert. Die vier Hauptrichtungen und der Feuerknopf sind in fünf Bit abgebildet und liefern damit eine Zahl zwischen 0 und 31. Um die neue Situation eines Spielers zu bewerten kann man nun diesen Wert mit verschiedenen möglichen Werten vergleichen oder man macht folgendes:

```
getjoypos
tay
lda actions_lo, y
sta joyptr
lda actions_hi, y
sta joyptr+1
jmp (joyptr)
```

getjoypos liefert den oben erwähnten Wert aus der Joaystickabfrage. Dieser wird nun benutzt, um die Adresse der zuständigen Behandlungsroutine aus einer Tabelle zu ermitteln (*actions_lo/ actions_hi*) und diese dann anzuspringen. Die Tabelle muss Werte für 32 16-bit Adressen beinhalten, benötigt also 64 Byte wertvollen Speicher. Dafür ist die Ausführungsgeschwindigkeit der Auswertung unabhängig vom Joystickzustand und der Programmcode sehr übersichtlich.

Sprungtabellen sind eine oft anwendbare und einfache Methode, um zustandabhängige "Weichen" in Code zu realisieren, der ansonsten gleich durchlaufen wird. Zur Umsetzung derselben gibt es als Alternative zu obigen Code mittels indirektem Sprungbefehl auch die Möglichkeit, die Zieladresse auf den Stack zu legen und ein *return from subroutine (RTS)* auszuführen.

Abhängig von der aktuellen Joystickposition wird jedenfalls die Spielerposition und -erscheinung neu berechnet und muss in Folge dargestellt werden.

6 Grafik

6.1 Einführung

Dieser Abschnitt behandelt den Hauptteil des Programms in dem auch die meiste Prozessorzeit verbraten wird und wird daher etwas länger. Der C-64 und nachfolgende Geräte bis zur heutigen Hardware legen den Schwerpunkt auf die Optimierung der Grafik und ermöglichen dadurch immer ansprechendere Umsetzungen unter ähnlichen Rahmenbedingungen.

Eine wesentliche Bedingung ist, mit allen Berechungen fertig zu sein, wenn dass nächste Bild angezeigt werden muss. Eine weitere nicht weniger wesentliche, die in modernen Systemen aber meist nicht mehr in der Hand des Programmierers liegt, ist, die Anzeige so zu aktualisieren, dass keine Brüche sichtbar werden.

Letzteres bedeutet, dass der Bildschirminhalt so aktualisiert wird, dass alle Bereiche zu jedem Zeitpunkt konsistent sind. Wenn zum Beispiel eine Spielfigur bewegt und dabei nicht auf die physikalische Darstellung synchronisiert wird, kann die obere Hälfte der Figur mit der alten Position und die untere mit der neuen erscheinen. Diese Effekte sind leider sichtbar und für eine saubere Darstellung nicht akzeptabel.

Um das zu umgehen kann, muss man den Bildinhalt aktualisieren, wenn die physikalische Darstellung gerade nicht aktualisiert wird. Das kann wiederum auf zwei Arten machen.

1. Der Bildinhalt wird in der Zeit neu berechnet, in der die physikalische Anzeige nicht aktualisiert wird, bzw. in manchen Demos und Intros parallel dazu, um spezielle Effekte zu ermöglichen, die sonst nicht möglich wären (Abbildung 1).

2. Man arbeitet mit Puffern, d.h. die nächste Darstellung wird in einen Puffer vorberechnet und in einem günstigen Moment in die Darstellung übertragen.

Die erste Variante ist unumgänglich, wenn man hardwarenahe Effekte programmieren will und dazu nicht viel Zeit braucht. Wenn man aber nicht sicher ist, ob die Bildaktualisierung schnell genug möglich ist oder aber sicher ist, dass die Aktualisierung zu langsam ist, muss man die Bildberechnung von der Aktualisierung trennen. Diese (zweite) Variante ist auch als *double buffering* bekannt und bietet den Vorteil, dass die Bildaktualisierung mit der Bildfrequenz nicht unbedingt synchron sein muss. Ich habe die zweite Variante für das Spiel gewählt, weil ich anfangs noch nicht sicher war, die erforderliche Geschwindigkeit für 50 Bilder pro Sekunde mit einem grob zusammengezimmerten Programm zu erreichen. Ein weiterer Vorteil der Pufferung ist, dass man unkompliziert den ganzen Zeitraum der Bildaktualisierung für die Berechnung des nächsten Bildes zur Verfügung hat.

Diese und weitere später benötigte Informationen zu den Grafikfähigkeiten des VC-20 werden im folgenden Abschnitt kurz vorgestellt.

```
Bit  7 6 5 4 3 2 1 0
    +-+-+-+-+-+-+-+-+
    | | | |*|*| | | |  =  2^4 + 2^3  =  16 + 8  =  24
    +-+-+-+-+-+-+-+-+
    | | |*| | |*| | |  =  2^5 + 2^2  =  32 + 4  =  36
    +-+-+-+-+-+-+-+-+
    | |*| | | | |*| |  =  2^6 + 2^1  =  64 + 2  =  65
    +-+-+-+-+-+-+-+-+
    | |*|*|*|*|*|*| |  =  2^6 + ...  =  64 ...  =  126
    +-+-+-+-+-+-+-+-+
    | |*| | | | |*| |  =  2^6 + 2^1  =  64 + 2  =  65
    +-+-+-+-+-+-+-+-+
    | |*| | | | |*| |  =  2^6 + 2^1  =  64 + 2  =  65
    +-+-+-+-+-+-+-+-+
    | |*| | | | |*| |  =  2^6 + 2^1  =  64 + 2  =  65
    +-+-+-+-+-+-+-+-+
    | | | | | | | | |  =  0
    +-+-+-+-+-+-+-+-+
```

Abbildung 7: *Zeichendefinition in 8x8 Pixel am Beispiel des Zeichens **A**. Jede Zeile entspricht einem Byte und die Vordergrundpixel entsprechen in der Binärdarstellung des Bytes der 1, während Pixel in der Hintergrundfarbe durch die 0 repräsentiert werden. Die acht Zahlenwert für das Zeichen ergeben sich aus den Werten jeder Zeile, also hier 24,36,65,126,65,65,65,0.*

6.2 Grafikfähigkeiten des VC-20

Grundsätzlich kann der VC-20 nur Text darstellen und das in der Standardauflösung von 22 Zeichen in 23 Zeilen, was sich bei der PAL-Ausgabe bis zu maximal 28 Zeichen in 32 Zeilen erweitern lässt. Das es dennoch Spiele gibt, die nicht auf den eingebauten Zeichensatz beschränkt sind, liegt daran, dass man den Zeichensatz für die Textdarstellung vollständig umdefinieren kann (Abbildung 7).

Der Video-Chip konstruiert das dargestellte Bild dabei aus drei Speicherbereichen:

1. das Video-RAM mit den Zeichencodes
2. das Farb-RAM mit den Zeichenfarben
3. und das Zeichengenerator-ROM bzw. -RAM mit der Definition der Zeichendarstellung.

Das Umschalten zwischen mehreren parallelen Puffern geht beim VC-20 zum Glück ganz einfach, weil sowohl Video- als auch Zeichenspeicher auf verschiedene Positionen gelegt werden können. Das Farbram bietet zwar nur zwei mögliche Bereiche zum Umschalten, das ist für zwei Puffer aber völlig ausreichend.

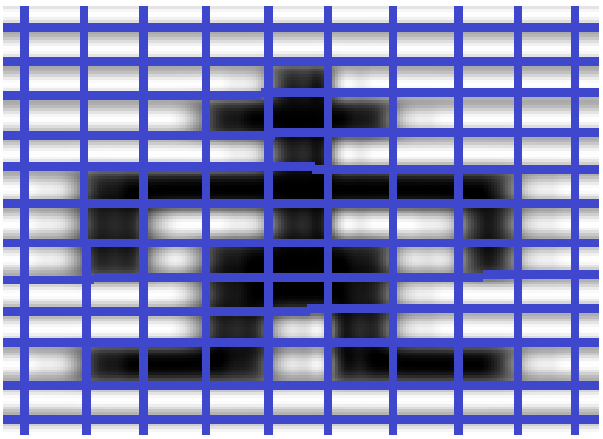

Abbildung 8: *Darstellung eines Spielers im Stillstand.*

6.3 Vorbereitung der Bildobjekte

Bewegliche Bildobjekte sind die bis zu neun Spieler und die dazugehörigen Schusspixel. Da der VC-20 keinen echten Grafikmodus und auch keine Sprites kennt, bedeutet die Anzeige dieser Objekte etwas Programmierarbeit. Die Objekte werden an die richtige Position eines Sonderzeichen gesetzt, welches dann an die entsprechende Bildposition geschrieben wird. Dabei ist für die Schusspixel ein Zeichen ausreichend, da es nie an eine Zeichengrenze geteilt werden kann. Für die Spielfiguren sind 2x2 Zeichen nötig, weil sowohl horizontal als auch vertikal die Zeichengrenze überschritten werden kann. Dadurch ergibt sich eine maximale Grösse füer die Figur in allen Bewegungsformen mit 9x9 Pixeln. Im Stillstand wird das in der Höhe bereits voll ausgenutzt (Abbildung 8).

Die Position eines Objekts wird zunächst in die Zeichenposition und die Position im Zeichen zerlegt. Da jedes Zeichen aus 8x8 Pixeln besteht ist das eine einfache Bit-Operation. Nun muss dass Objekt an die richtige Position im Zeichensatz geschoben werden. Das lässt sich mit den Bitschiebeoperationen (Abbildung 9) leicht machen, ist aber sehr zeitwaufwändig. Im Intro wird die Laufschrift auf diese Weise in jedem Frame um einen Pixel nach links verschoben, für die Positionierung der Spielobjekte ist diese Methode jedoch zu langsam. Darum werden alle möglichen Positionen vor dem Spiel vorberechnet und im Spiel nur das zur aktuellen Position passende Objekt ausgewählt.

Aufwändige Animationen lassen sich mit dem am VC-20 knappen Speicher so zwar nicht realisieren. Die Vorberechnung der im Spiel verwendeten 9 Posen benötigt aber nur 2367 Byte (s.u.), die mit den grösseren Speichererweiterung durchaus verfügbar sind.

```
shapes2x2: .res (7+8*32)*NUM_SHAPES_2x2
```

Die Zahlen in obiger Formel resultieren aus der Art der Speicherung der 9x9 Objekte.

- Ein Objekt belegt 4 Zeichen mit 8 Byte für die Zeichendefinition, also **32** Byte.
- Jedes Objekt kann an **8** Pixelposition beginnen, also werden 8*32 Byte zur Speicherung der vorberechneten horizontalen Verschiebungen benötigt.

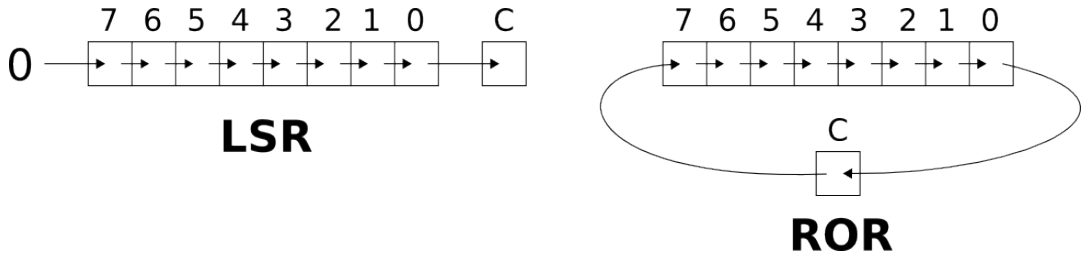

Abbildung 9: *Rechtschiebeoperationen der 6502.* ***LSR*** *(logical shift right) schiebt eine 0 am höchstwertigsten Bit hinein und schiebt das niederwertigste in das Carry-Flag (Überlauf).* ***ROR*** *(rotate right) rotiert den Operanden durch das Carry-Flag. LSR entspricht damit einem ROR mit gelöschten Carry-Flag. Die entsprechenden Befehle zum* ***Linksschieben*** *sind* ***ASL*** *(Arithmetic shift left) und* ***ROL*** *(rotate left). Mittels ROR bzw. ROL lässt sich eine ganze Pixelzeile nach rechts bzw. links schieben, weil der aus dem Byte entfernte Pixel durch das Carry-Flag mit dem Folgebefehl ins nächste Byte übertragen wird.*

- Die Zeichen sind nach Spalten abgelegt, d.h. das zweite Zeichen im Speicher liegt unter dem ersten. Dadurch und durch die Tatsache, dass die letzten 7 Pixelzeilen im unteren Rand leer sind, kann die vertikale Verschiebung durch Verschieben der Startadresse nach vorne realisiert werden. Beim ersten Objekt wird dazu zusätzlich ein Vorlauf von **7** Leerzeilen benötigt. Daher werden 7+8*32 Byte zur Speicherung einer Darstellung (Pose) benötigt [2].

6.4 Aktualisierung der Anzeige

Die Anzeigeaktualisierung erfolgt im Bufferverfahren durch Wechseln des aktuell angezeigten Buffers.

Der Rechenaufwand steckt also in der Vorbereitung des nächsten anzuzeigenden Bufferinhalts. Dazu muss folgendes erledigt werden:

1. Löschen des alten Bufferinhalts
2. Zeichnen des Hintergrunds im Buffer
3. Zeichnen der beweglichen Objekte

Die ersten beiden Punkte können zusammengefasst werden, wenn der Hintergrund unveränderbar ist und als vollständiger Bildinhalt in den Buffer kopiert werden kann.

Das Löschen des Bildinhalts sieht bei der Anzeige des Gewinners z.B. so aus:

[2] Genaugenommen werden die sieben führenden Leerzeilen nur ganz am Anfang benötigt, so dass sich hier nochmal ein paar Byte (56) sparen liessen.

```
                ; clear screen
                ldx #0
@clrloop:       lda #$20
                sta VIDEO_BASE, x
                sta VIDEO_BASE+$0100, x
                lda #0
                sta COLOR_BASE, x
                sta COLOR_BASE+$0100, x
                inx
                bne @clrloop
```

Die Schleife wird 256-mal durchlaufen und benötigt zur Ausführung **6621** Taktzyklen. Sie ließe sich zwar noch optimieren[3], aber eine Kopierschleife für den Hintergund ist immer noch langsamer. Wir haben für einen Durchlauf der Hauptschleife ca. **22000** Takte Zeit. Obige Löschroutine würde bereits mehr als ein Viertel der verfügbaren Zeit verbraten und da stellt sich doch die Frage, ob es nicht eine bessere Möglichkeit gibt, den Bildinhalt vorzubereiten.

Der Hintergund ändert sich in diesem Spiel eigentlich nicht und es werden je vier Zeichen für die Spielfiguren und je ein Zeichen für die fliegenden Pixel dargestellt. Das sind insgesamt also nur $5*9 = 45$ von 506 Bildpositionen, die verändert werden. Es könnte sich daher lohnen, die zuletzt eingezeichneten Figuren und Pixel aus dem Bild einfach wieder rauszulöschen anstatt den ganze Hintergund neu zu zeichnen. Diese Methode bedeutet zwar etwas zusätzlichen Aufwand, weil man sich für jeden Buffer die letzten Positionen merken muss und die Löschroutine wird auch etwas komplizierter, aber in diesem Fall lässt sich das tatsächlich in maximal **2682** Takten erledigen (Abbildung 10, erster Teil).

Der Rest der Aktualisierung besteht aus

- Kopieren der vorberechneten Zeichen in das Zeichen-RAM,
- Schreiben der Zeichen an die vorgesehenen Stellen im Video-RAM und
- Setzen der Spiele-Farbe zu den vorher geschriebenen Zeichen im Farb-RAM

Beim Schreiben der Zeichen in das Video-RAM wird auch die Kollisionsabfrage durchgeführt. Dazu wird vor dem Schreiben des aktuellen Zeichens geprüft, ob an der Stelle schon ein Zeichen steht, das kein Hintergrundzeichen ist. Wenn es kein Hintergrundzeichen ist und daher eine potentielle Kollision vorliegt, werden die Zeichendaten auf Überlappung geprüft (AND), um eine Kollision zu erkennen. Ein weiterer Grund, den Fall extra zu behandeln, dass ein Zeichen über ein anderes geschrieben werden soll, ist, dass diese beiden Zeichen zu einem verschmolzen werden müssen (OR).

Die Bildaktualisierung inklusive Kollisionsabfrage belegt den größten Teil der im Frame verfügbaren Zeit und ist aus Abbildung 10 nicht wirklich nachvollziehbar. Was man aber sieht ist, dass die Hauptarbeit der Adressberechnung an den Assembler delegiert wird, in dem die Bildpufferadressen als Makroparameter angegeben werden und die jeweiligen Schleifen über die Spielerobjekte durch *.repeat* Befehle echte Schleifen ersetzen.

[3]Dies sei dem geneigten Leser als Übungsaufgabe überlassen

```
.macro draw_frame char_base, video_base, color_base, frame
    ; clear background
    .repeat MAX_OBJECTS_2x2, I
        lda active2x2+I+MAX_OBJECTS_2x2*(frame+1)
        bne :+
        jmp :++
    :   clr_object_2x2 video_base, I+MAX_OBJECTS_2x2*(frame+1), bkvid
        clr_object_2x2 color_base, I+MAX_OBJECTS_2x2*(frame+1), bkclr
    :
    .endrep
    .repeat MAX_OBJECTS_1x1, I
        lda active1x1+I+MAX_OBJECTS_1x1*(frame+1)
        bne :+
        jmp :++
    :   clr_object_1x1 video_base, I+MAX_OBJECTS_1x1*(frame+1), bkvid
        clr_object_1x1 color_base, I+MAX_OBJECTS_1x1*(frame+1), bkclr
    :
    .endrep
        ; moving objects
    .repeat MAX_OBJECTS_2x2, I
        ldy #0
        lda active2x2+I
        sta active2x2+I+MAX_OBJECTS_2x2*(frame+1)
        bne :+
        jmp :++
    :   lda #0
        sta coll2x2 + I
        copy_object_2x2 char_base, I
        show_object_2x2 video_base, I, frame, char_base
        color_object_2x2 color_base, I
    :
    .endrep
    .repeat MAX_OBJECTS_1x1, I
        lda active1x1+I
        sta active1x1+I+MAX_OBJECTS_1x1*(frame+1)
        bne :+
        jmp :++
    :   lda #0
        sta coll1x1 + I
        copy_object_1x1 char_base, I
        show_object_1x1 video_base, I, frame, char_base
        color_object_1x1 color_base, I
    :
    .endrep
.endmacro
```

Abbildung 10: *Der zentrale Programmteil zur Aktualisierung eines der Puffer (Frame).*

Dadurch wird einerseits die Verwendung von zeitintensive Befehlen mit indizierten Operanden vermieden und andererseits Loop-Unrolling in einfacher Form umgesetzt, wodurch wiederum enorm Zeit gespart wird.

6.5 Synchronisierung auf die Bildwiederholrate

Im Gegensatz zum C-64 hat der VC-20 keinen Rasterzeileninterrupt. Es gibt aber ein Register im Video-Chip, das die aktuelle Rasterzeile angibt. Um auf eine bestimmte Rasterzeile zu synchronieren muss man also das Rasterzeilenregister abfragen, bis der gewünschte Wert erscheint. Da die Warterei etwas Zeitverschwendung ist, kann man einen Timer mit dem Wert der Bildfrequenz laden und Interrupts generieren lassen. Wenn dieser synchron mit der gewünschten Rasterzeile gestartet wird, simuliert der Timer-Interrupt den Rasterzeileninterrupt genau genug, um das Bild im Strahlrücklauf ruckelfrei zu aktualisieren.

6.6 Zusammenfassung

Die in diesem Abschnitt beschriebenen Optimierungen waren:

- Löschen der Bildobjekte aus dem Bild statt jedesmal den Hintergund komplett zu kopieren.
- Vorberechnung der möglichen Bildobjektpositionen im 8x8-Pixel-pro-Zeichen-Raster.
- Loop-unrolling und Code-duplizierung für die beiden Bildpuffer durch Makroprogrammierung.

Die letzten beiden Verfahren blähen den Speicherbedarf des Programms natürlich immens auf. Das Ergebnis findet sich im der Linker-Map wie folgt:

```
Name                   Start   End     Size
-------------------------------------------
ZEROPAGE               0000D8  0000F0  000019
BASIC                  0011FF  00164D  00044F
LOCODE                 00164E  001DB0  000763
TEXT                   001E00  001F6F  000170
JMPTBL                 002000  002003  000004
CODE                   002004  00568F  00368C  14.0K
RODATA                 005690  005AB0  000421   1.1K
DATA                   005AB1  005F23  000473   1.1K
BSS                    005F24  006A16  000AF3   2.8K
```

Die vorberechneten Bildobjekte werden im Segment **BSS** abgelegt und sind nicht einmal der grösste Brocken. Dieser findet sich im **CODE**-Segment, das durch die Makros auf stolze 14K angewachsen ist. Der Speicherbereich ab 002000 ist im VC-20 für Speichererweiterungen vorgesehen und wird hier mit 18966 Byte belegt. In Summe benötigt

```
sndsteps:   inc sndvol
            ldy #SOUND_STEPS
            jmp sound2

sndthud:    inc sndvol
            ldy #SOUND_THUD
            jmp sound1

sndblip:    inc sndvol
            ldy #SOUND_BLIP
            jmp sound3

sndnoise:   inc sndvol
            ldy #SOUND_NOISE
            jmp sound4
```

Abbildung 11: *Aufrufe der Toneffekte zu einem Spieler. Jeder Effekt benutzt einen anderen Kanal, so dass die verschiedenen Geräusche gleichzeitig abgespielt werden können. Die Lautstärke kann beim VC-20 nur für alle Kanäle gemeinsam angeben werden und wird hier durch die Anzahl der Spieler, die einen Toneffekt auslösen bestimmt.*

das Programm beim VC-20 also mindestens eine 24K-Erweiterung, welche in den besten Zeiten des VC-20 auch eher selten war.

Der Code für das Intro befindet sich im Segment **BASIC**, **LOCODE** und **TEXT**, der während des Spiels durch die Grafikdaten überschrieben wird und kostet also keinen zusätzlichen Speicher.

7 Toneffekte

Der VC-20 ist im Vergleich zum C-64 leider sehr limitiert, was die Tonausgabe betrifft. Diese wird vom Video-Chip nebenbei miterledigt und bietet drei Rechteck- und einen Rauschgenerator, die intern über eine Schieberegister realisiert sind. Die Schiebefrequenz wird über einen 7-Bit-Wert eingestellt, der 4 Oktaven abdeckt und daher zum Teil die verfügbaren Noten etwas ungenau abbildet. Dazu kommt, dass die Lautstärke nur für die Gesamttonausgabe reguliert werden kann, was zu Clipping-Effekten führen kann.

Immerhin gibt es vier Kanäle, die im Spiel für Schritte (steps), Schussgeräusch (blip), Spielerneustart (thud) und Treffer (noise) verwendet werden (Abbildung 11).

Die abzuspielenden Geräusche sind als Sequenz von Werten füer den Schiebetakt abgelegt und werden im simulierten Rasterzeileninterrupt abgespielt. Ein Geräuschmuster besteht aus einer Sequenz von Wertepaaren, wobei der erste Wert die Dauer einer *Note* in Vielfachen von 20ms angibt (wir erinnern uns: eine Bildwiederholrate von 50Hz entspricht 20 ms pro Bild) und der zweite Wert den Schiebetakt (Frequenz-Äquivalent).

8 Fehlersuche

9 Erweiterung

Nach einigen Testspielen im Rahmen des Commodore Treffen Graz hat sich herausgestellt, dass das Spiel mit geübten Spielern gegegn Ende etwas langweilig wird. Wenn nur noch zwei oder drei Spieler auf dem Spielfeld sind, bieten sich genug Möglichkeiten den Angriffen der oder des Gegner(s) auszuweichen und es kann etwas dauern, bis das Spiel entschieden ist.

10 Ausblick

Mit den verwendeten Algorithmen sind Erweiterungen leicht möglich. Beispielsweise kann ein Team-modus realisiert werden, bei dem die Spieler in zwei oder mehr Gruppen gegeneinander spielen. Dabei können verschiedene Rand- und Gewinnbedingen realisiert werden. Der Hauptaufwand ist daher, sich geeignete und interessante Spielmodi zu überlegen.

Wodurch wird ein Team zum Sieger? Wenn alle Spieler den anderen Teams ausgeschieden sind? Oder gibt es ein Punktesystem? Kann man eigene Spieler auch treffen?

Es wäre auch möglich ein Capture-the-Flag Szenario umzusetzen, bei dem ein Gegenstand aus dem feindliche Lager geholt und zur eigenen Basis gebracht werden muss.

Man sieht schon, dass auch mit einer grafisch einfachen Umsetzung einige interessante Ideen realisierbar sind.