GameDev

Sam Jordan

COLLABORATORS

	TITLE : GameDev	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Sam Jordan	July 19, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

GameDev

1.1 Systemkonforme Spieleentwicklung unter PPC

Systemkonforme Spieleentwicklung unter PPC
1997/98 von Sam Jordan

Tips, Gedanken und Anregungen zum Thema Spieleentwicklung für den PowerAMIGA

1.2 Vorwort

Die erste Diskette, die ein frischgekaufter AMIGA geschluckt hatte, war wohl in den meisten Fällen eine Spiele-Diskette. Bei mir war das jedenfalls der Fall, ich erinnere mich noch gut an das Spiel: Silkworm. Das war Ende 1991.

Spiele bildeten schon immer eines der Standbeine des AMIGA's neben dem Betriebssystem und den berühmten Custom-Chips. Daher bekam der AMIGA wohl auch das Image des Spiele-Computers, das noch heute von unwissenden Computer-besitzern anderer Modelle hervorgebracht wird. Etwas wahres ist sicher daran.

Der Höhepunkt der AMIGA-Spieleentwicklung war wohl so Anfang der Neunziger-Jahre. Dem AMIGA gings noch gut und Spiele waren sehr gefragt. Mit den Problemen und schliesslich mit dem Niedergang von Commodore gings auch mit den Spielen

bergab. Sowohl Quantität als auch Qualität liessen nach. Das kurze Aufbäumen von AMIGA Technologies unter Escom konnte dem Spielebusiness keine Impulse mehr verleiten. Nichtsdestotrotz lebt die Spieleszene noch, allerdings hat die Bedeutung natürlich stark verloren.

Während dieser Zeit waren die Spielehersteller auf den Konkurrenzsystemen nicht untätig. In den letzten Jahren hat die Spieletechnologie einen regelrechten Boom erfahren, was wohl auch hauptsächlich durch die immer schnellere Hardware verursacht wurde. Im Zeichen dieses Technologiesprungs blieb der AMIGA auf der Strecke. Einzig in den Bereichen, in denen er der Konkurrenz seinerzeit um Generationen voraus war, konnte er bestenfalls noch mithalten.

Die Ansprüche des Spielers haben sich grundlegend verändert. Heute stehen vor allem Grafik, Musik und Geschwindigkeit im Vordergrund, während früher wohl grösseren Wert auf Spielbarkeit und Atmosphäre gelegt wurde. Das soll nicht heissen, dass die heutigen Spiele diese Qualitätsmerkmale nicht mehr besitzen. Vielmehr haben sich die Prioritäten gewandelt. Um im hart umkämpften Spielebusiness Erfolg zu haben, muss man zwangsläufig die Ansprüche der Spieler berücksichtigen.

Dem AMIGA fehlte eines: Speed. Genau dieser fehlende Speed wirkte sich höchst negativ auf die Spielebranche aus. Wer Speed wollte, kaufte sich ein Konkurrenzsystem, welches mit dreistelligen Taktfrequenzen protzt, und erfreute sich an dem zusätzlichen Temporausch.

Hier möchte ich jetzt auf das Ziel dieses Dokuments zu sprechen kommen.
Das was dem AMIGA fehlte, das bekommt er jetzt: Speed!

Mit dem Einsatz der PowerPC-Prozessoren kann der AMIGA einen grossen Sprung nach vorne machen und zur Konkurrenz aufschliessen. Der AMIGA bekommt wieder eine Chance, ganz nach oben zu kommen. Das Gleiche gilt auch für die Spiele.

Auf einem PowerAMIGA sind Spiele möglich, welche den selben Qualitätsstandard erreichen wie auf den bekannten Konkurrenzsystemen. Der PowerPC-Prozessor ist auf dem aktuellen Stand der Technik und vermag beispielsweise die Intel-Prozessoren noch zu überbieten.

Um aus den Spielen für den PowerAMIGA das Maximum an Leistung herauszuholen, braucht es aber sehr viel Know-How. Der AMIGA besteht nicht nur aus einem Prozessor, sondern aus vielen Einzelkomponenten, welche eng zusammenspielen. Ebenso wichtig wie der Prozessor ist beispielsweise die Grafikansteuerung. Wenn der Zugriff auf den Grafikspeicher das Spiel völlig bremst, nützt der schnellste Prozessor nichts mehr. Hier gilt es, die entstehenden Flaschenhälse zu erkennen und zu umgehen. In diesem vorliegenden Dokument sind einige sehr wertvolle Tips zur Spiele-Programmierung enthalten. Es werden die grössten Flaschenhälse beschrieben und Lösungsansätze vorgestellt.

Im WarpOS-Archiv befinden sich zwei Demo-Programme, welche hier eine höhere Bedeutung haben. 'Cybermand' und vor allem das 'voxelspace'. Beide Programme zeigen, wie man ein Spiel für den PPC entwickeln kann und auch soll. Und beide Programme demonstrieren eindrücklich, was systemkonform für eine Wahnsinns-Geschwindigkeit erzielt werden kann, wenn genügend Erfahrung vorhanden ist.

Das Dokument geht aber nicht nur auf die Technik ein, sondern auch auf das, was wesentlich wichtiger ist: auf das Game-Design. Nur wenn ein Spiel auch

von der spielerischen Seite überzeugen kann, wird es auch gekauft. Sehr wichtig ist auch in diesem Zusammenhang das Erstellen von tauglichen Demo-Versionen. Der allergrösste Teil der Demoversionen, die ich bisher gesehen und getestet hatte, waren komplett nichts wert. Das ist wirklich sehr schade, wenn durch solche Fehler die grosse Arbeit an einem Spiel zunichte gemacht wird.

Warum mache ich mir eigentlich die Mühe und schreibe solch ein Dokument? Ganz einfach: wir bei HAAGE&PARTNER möchten gute Spiele. Und wir möchten, dass der AMIGA in eine bessere Zukunft geht und dass auch die Spiele diesen Weg gehen.

Sam Jordan

1.3 Einleitung

Dieses Dokument hat zum Ziel, neue Impulse in der Spieleentwicklung für den AMIGA zu geben. Es beschreibt, was bei der Spieleentwicklung speziell für den PowerPC-Prozessor alles beachtet werden muss, wo die Schwächen und Stärken des PPC liegen und wie man die zu seinem Vorteil verwerten kann.

Es geht zunächst in technische Belange ein und stellt die grössten Probleme dar, die auftreten können. Danach kommen auch Aspekte des Game-Designs zum Zug, wo diskutiert werden soll, worauf bei der Spieleentwicklung geachtet werden muss, damit ein Spiel eine hohe spielerische Qualität erreicht.

Was überhaupt mit dem PPC erreicht werden kann, demonstrieren die beiden Demo-Programme 'cybermand' und vor allem 'voxelspace'. Diese beiden Programme demonstrieren, wie ein PPC-Spiel etwa aufgebaut sein kann und zeigen eindrücklich, dass völlig systemkonform eine sehr grosse Geschwindigkeit erzielt werden kann.

Voraussetzung für superschnelle Spiele: sie müssen unter WarpOS lauffähig sein. WarpOS ist die vorerst einzige Chance, auf dem Dual-Prozessor-Board schnelle Games/Demos zum Laufen zu bringen. WarpOS wurde auch im Hinblick auf Spiele speziell optimiert und bietet Features an, welche vor allem der Spieleentwicklung sehr behilflich sein kann.

1.4 Die Grundphilosophie

Zu Beginn möchte ich eine Philosophie vertreten, welche meiner Meinung nach in Zukunft bei der Entwicklung von Spielen angewandt werden sollte:

Spiele sollten in Zukunft völlig systemkonform im Einklang mit dem jeweils vorherrschenden Betriebssystem funktionieren, wobei zusätzliche Möglichkeiten, welche nicht systemkonform sind, als Option dem User nicht vorenthalten werden sollen.

Die AMIGA-Spiele waren seinerzeit schnell, sogar sehr schnell. Das Geheimnis der hohen Geschwindigkeit ist schnell gelüftet: Der Grossteil der Spiele schaltete das Betriebssystem komplett ab und übernahm die Hardware. Folge: Ein Spiel, das auf jedem AMIGA problemlos lief, genoss Seltenheitswert. Vor

allem AMIGA's, die gar noch nicht gebaut wurden, machten in der Folge grosse Probleme, da die Programmierer noch gar nichts über die neue Maschine wussten.

Jetzt könnte natürlich als Gegenargument zu meinem obigen Statement folgen: Systemkonforme Spiele würden zu langsam laufen.

Das erste, das derjenige, der dieses Argument hervorbringt, tun sollte: Die beiden beiligenden Demos 'cybermand' und 'voxelspace' starten. Danach verliert das Argument jegliche Glaubwürdigkeit. Die Programme 'cybermand' und 'voxelspace' laufen zu 100 Prozent systemkonform und trotzdem mit einer enorm hohen Geschwindigkeit. Es IST möglich!

Das 'voxelspace' demonstriert auch den zweiten Teil meiner Philosophie: Es bietet dem User richtig 'böse' Hacks als zusätzliche Parameter an. Der User muss die eigenhändig aktivieren und kann sich danach über erhöhten Speed freuen, sofern das dann überhaupt noch läuft. Genau das ist dann das Problem: Die Hacks laufen unter Umständen nicht mehr überall. Und genau aus diesem Grund ist es auch zwingend notwendig, dass ein Spiel in erster Linie völlig systemkonform laufen muss. Dann läuft es nämlich immer.

Es gibt aber noch mehr Argumente, die für die Philosophie 'systemkonform + optionale Hacks' sprechen. Erstens: Die Hardware wird immer schneller. Wenn ein Spiel jetzt systemkonform läuft, läuft das auf Nachfolge-Hardware immer noch, aber mit erhöhter Geschwindigkeit. Wenn das Spiel auf Hacks abgestützt ist, erhöht sich die Wahrscheinlichkeit, dass es in Zukunft überhaupt nicht mehr läuft.

Systemkonforme Spiele zu entwickeln ist wesentlich effizienter als Spiele zu entwickeln, welche das Rad komplett neu erfinden und dabei nur wenig schneller sind. In der heutigen Zeit muss man effizient arbeiten, damit ein Spiel erstens den Qualitätsansprüchen der User genügt und zweitens noch rechtzeitig herauskommt. Oftmals kommen Spiele erst dann heraus, wenn sie bereits wieder veraltet sind ...

Es ist wichtig, dass die AMIGA-Spiele ein neues Image erhalten. Der potentielle Kunde darf keine Angst haben, dass das Spiel bei ihm nicht laufen könnte. Oft führt solch eine Haltung dazu, dass das Spiel dann doch nicht gekauft wird. Das habe ich selber schon erlebt. Als HighEnd-User war es mir oftmals zu riskant, ein Spiel zu kaufen, welches höchstwahrscheinlich nicht mit meiner Hardware zusammenarbeitet.

1.5 Wahl der Programmiersprache

Waren in der Vergangenheit schnelle Spiele gefragt, war die Wahl der Programmiersprache gar kein Thema. Assembler. Alles andere war zu langsam.

In der heutigen Zeit hat sich das grundlegend verändert. Spiele werden immer komplexer, immer grösser. Mittlerweile konzentriert sich der Teil, welcher tatsächlich für die Geschwindigkeit verantwortlich ist, auf einen winzigen Bruchteil des gesamten Projektes.

Es ist völlig klar, dass es kompletter Irrsinn ist, Projekte dieser Grössenordnung in 100% Assembler durchzuführen. Das ist schlicht und einfach viel zu aufwendig und bringt viel zu wenig Gewinn.

Andererseits hat Assembler als Sprache lange nicht ausgedient. Wenn ein Spiel komplett in einer Hochsprache entwickelt wird, läuft es in der Regel sehr viel langsamer, als wenn einige ausgewählte Unterroutinen in Assembler geschrieben werden. Genau das ist meiner Meinung nach die optimale Mischung.

Als Hochsprache bietet sich selbstverständlich C an, da es eine sehr portable Sprache ist. Ein Projekt sollte also zunächst mal komplett in C durchgezogen werden. Steht das Programm, wird es getestet. Hier leistet ein sogenannter Profiler gute Dienste, welcher die Belastung der einzelnen Funktionen misst und dann eine Statistik ausgibt, wonach festgestellt werden kann, welche Funktionen zeitkritisch sind und welche nicht. Auf der Basis einer solchen Profiler-Messung wird dann entschieden, welche Funktionen in Assembler umgesetzt werden sollen. Es gilt dann, ein gesundes Mass zu finden, so dass man nicht zuviele Funktionen umsetzt, aber auch nicht zu wenige.

Hat man die entsprechenden Funktionen ausgewählt, so sollte zunächst mal ein Assembler-Output des Compilers erzeugt werden. Oftmals muss nämlich die Funktion nicht mehr komplett neu geschrieben werden, sondern nur der Assembler-Output von Hand nachbearbeitet werden. Hat man eine Funktion umgesetzt, wird wieder das Spiel getestet und die Auswirkungen analysiert. Diese Analyse kann unter Umständen ebenfalls nützlich sein, um zu entscheiden, was für Funktionen noch in Assembler geschrieben werden sollten.

Hier sei noch ein etwas unkonventioneller Vorschlag angebracht: Sehr interessant wäre es, wenn solche Kernroutinen als externe Module zur Verfügung gestellt würden bzw. als kleine Programme. Wenn diese Mini-Programme gut dokumentiert sind, können danach Assembler-Spezialisten diese Routinen neu schreiben und einfach das Modul auswechseln. Davon würde bestimmt auch der Hersteller des Spiels profitieren. Dieses Verfahren wurde bereits hin und wieder eingesetzt beim Einsatz der ChunkyToPlanar-Konversions-Algorithmen, welche einfach als Module ausgetauscht werden konnten.

1.6 Struktur eines PPC-Spieles

Wenn Software für das Dual-Prozessor-System entwickelt wird, muss man folgende wichtige Entscheidung treffen: Welche Programmteile werden für 68K und welche für PPC kompiliert?

Ein erster Ansatz ist gewiss der, dass man vor allem rechenintensive Programmteile auf dem PPC laufen lässt, während man Bereiche, welche intensiv Betriebssystem-Funktionen aufrufen und keinen Beitrag zur Geschwindigkeit machen, für den 68K kompiliert. Ist man mit einem guten Entwicklersystem wie beispielsweise 'StormC' ausgerüstet, kann man zu jedem Zeitpunkt einen Quelltext für die jeweils andere CPU neukompilieren ohne dass man am Quelltext etwas ändern muss.

Dieser Ansatz muss aber in einem grösseren Zusamenhang betrachtet werden. Diese Dual-Prozessor-Lösung ist höchstwahrscheinlich nur ein Zwischenschritt auf dem Weg zu einem reinen PPC-AMIGA. Es ist zu erwarten, dass in Zukunft das AMIGA-OS oder nur Teile davon nach PPC portiert werden und 68K-Software mit einem Emulator ausgeführt werden.

Wenn man jetzt noch bedenkt, dass die Entwicklung eines Spieles einiges an Zeit kostet, ist es vermutlich zukunftsweisender, wenn der grösste Teil der Software, wenn nicht gleich alles, für den PPC kompiliert wird. Die Performance-Einbussen

auf dem Dual-Prozessor-Board wegen den vielen CPU-Wechseln bei Betriebssystem-Aufrufen halten sich dank dem Hochgeschwindigkeits-Kommunikations-System von WarpOS sehr in Grenzen.

Sollte sich jetzt zeigen, dass in gewissen Programm-Bereichen die Performance-Einbusse zu gross ist, so kann derjenige Teil für den 68K übersetzt werden. Wenn dann in Zukunft ein portiertes AMIGA-OS vorliegt, läuft das dann immer noch und der Spielehersteller kann dann immer noch ein Update herausbringen, wo die entsprechenden Programmteile für den PPC kompiliert wurden.

Beim Design der innersten Schleifen, welche sozusagen den Takt bei vielen Spielen angeben, ist besondere Vorsicht geboten. Hier muss sehr sorgfältig überlegt werden, wie eine solche Hauptschleife entworfen wird und für welche CPU dass sie übersetzt wird. Im Prinzip gibt es hier folgende Ansätze:

1. Die gesamte Hauptschleife wird für den PPC kompiliert. Das hat zur Folge, dass jeder Betriebssystem-Aufruf (z.B. für das Message-Handling) einen CPU-Wechsel nach sich zieht. Kommt ein Spiel mit sehr wenigen Betriebssystem-Aufrufen aus, ist dieser Weg der optimale.

2. Die Hauptschleife wird für den 68K kompiliert und die rechenintensiven Funktionen auf dem PowerPC durchgeführt. Hier ist es in der Regel so, dass schon mit wenigen CPU-Wechseln gearbeitet werden kann. Dies wird wohl oft der optimalste Weg sein. Dieser Weg wurde auch bei den beiden Demo-Programmen 'cybermand' und 'voxelspace', welche mit zwei bzw. einem PPC-Call auskommen.

3. Die Hauptschleife und die rechenintensiven Funktionen laufen mit zwei verschiedenen Tasks auf zwei verschiedenen CPU's. Mittels Signalen können die beiden Tasks synchronisiert werden. Dieser Ansatz ist sehr heikel, denn Multiprozessing hat oft kontraproduktive Auswirkungen.

Im Bereiche der innersten Schleifen sollte also immer der optimalste Weg eingeschlagen werden, damit ein Maximum an Performance zu erreichen ist. Wenn dieser Weg auf einem reinen PPC-System nicht der optimalste ist, kann dann immer noch ein Update erstellt werden, welches dann wiederum das Maximum an Performance auf dem jeweiligen System erzielt.

Prinzipiell ist folgendes zu sagen: Ein grosser Performance-Faktor sind die Betriebssystem-Aufrufe, wenn diese einen CPU-Wechsel verursachen. Bei wenigen Calls wirkt sich das sehr wenig aus, wenn aber sehr viele CPU-Wechsel geschehen, wirkt sich das sehr schlecht auf die Performance aus. Es ist also eine wichtige Aufgabe, dass die Struktur eines Spiel-Kernes so aufgebaut ist, dass die CPU-Wechsel minimiert werden.

1.7 Grafikprogrammierung

In diesem Abschnitt wenden wir und jetzt der technischen Seite der Spiele-Programmierung zu, insbesondere der Grafikprogrammierung. Wenn Sie direkt zum technischen Teil gehen wollen, gehen sie zum Grafik-Hauptmenu.

Die grösste Schwäche des AMIGA's in der heutigen Zeit ist die Grafikansteuerung, also genau das, was einst seine Stärke ausgemacht hatte. Die berühmten Custom-Chips des AMIGA wurden in erster Linie für Schrolling und Sprite-Handling ausgelegt, Bereiche, in welchen der AMIGA sogar heute noch Ansehnliches zu leisten vermag. Im Zeitalter der 3D-Spiele sind die

Custom-Chips leider nicht mehr zu gebrauchen.

Dazu kommt noch, dass die Custom-Chips nur im Chip-RAM operieren können. Das heisst, wenn der Prozessor Daten den Custom-Chips zur Verfügung stellen sollte, musste er alle Daten in das unglaublich langsame Chip-RAM schreiben. Der Zugriff auf das Chip-RAM war anno dazumal für den 68000 optimal ausgelegt. Ueberflüssig zu sagen, dass für einen Prozessor, der mehr als 100 mal so schnell ist, dieser Zugriff komplett unakzeptabel ist.

Wo liegt denn der Ausweg aus diesem Problem? Wenn hohe Performance gewünscht ist, ist eine Grafikkarte vonnöten. Und genau hier liegt zur Zeit die Hauptschwäche des AMIGA's:

1. Der sogenannte 'Standard'-AMIGA verfügt heute noch über keine Grafikkarte.

2. AMIGA-Grafikkarten sind ungleich teurer als vergleichbare Grafikkarten auf Konkurrenzsystemen und bieten gleichzeitig schlechtere Performance. Der höhere Preis geht wohl auf das kleinere Marktvolumen zurück, denn Grafikkarten-Besitzer gehören heute noch zu einer Minderheit.

3. Das normale AMIGA-Grafiksystem arbeitet mit Bitplanes. Viele Spiele (vor allem 3D-Spiele) sind aber bei der internen Darstellung von Grafiken auf das Chunky-Format angewiesen, welches von allen Grafikkarten verwendet wird. Daraus folgt, dass solche Software sogar noch zusätzlich gebremst wird, da die Grafikdaten ins Bitplane-Format konvertiert werden müssen. Der Geschwindigkeitszuwachts mit dem PPC-Prozessor ist hier ebenfalls minimal, da der Algorithmus sehr speicherlastig ist und vor allem natürlich, weil er dauernd in das Chip-RAM schreibt.

4. Der allergrösste Teil der Spiele unterstützt keine Grafikkarten. Das hat natürlich zur Folge, dass die Spiele selbst auf schnellen 68K-Systemen, welche mit einem 68040 oder 68060 ausgestattet sind, hoffnungslos langsamer sind als auf bekannten Konkurrenzsystemen, welche nicht einmal über viel schnellere Prozessoren verfügen. Ein PowerPC-Prozessor ist bei solchen Spielen überhaupt nicht mehr sinnvoll.

Mit dem Einsatz der PowerPC-Prozessoren ist eine Grafikkarte ein absolutes Muss. Deswegen müssen auch Spiele, welche für PPC geschrieben werden, Grafikkarten möglichst optimal unterstützen. Im folgenden wird die Ansteuerung von Grafikkarten ebenso beschrieben wie die systemkonforme Programmierung von ECS/AGA-Grafik.

An dieser Stelle sei ein Hinweis auf eine Software gemacht, welche einen Grossteil der unten beschriebenen Vorgänge bereits unterstützt und dem Programmierer entsprechende Funktionen anbietet: RTGMaster von Steffen Haeuser (zu finden im Aminet unter gfx/board). Spieleprogrammierer sollten ernsthaft erwägen, die Grafikprogrammierung über RTGMaster zu machen, anstatt alles neu zu codieren. RTGMaster wird immer noch intensiv weiterentwickelt und betreut und unterstützt laufend neue Grafik-Hardware.

Die folgenden Erklärungen sind also für diejenigen von Interesse, welche aus irgendwelchen Gründen RTGMaster für eine bestimmte Anwendung nicht verwenden können.

ECS/AGA-Programmierung
Grafikkarten-Programmierung
CyberGFX+

TurboGFX

1.8 ECS/AGA-Programmierung

Ein Grossteil der AMIGA's verfügt noch immer über keine Grafikkarte. Wenn also Spiele entwickelt werden sollten, muss immer auch die Zielgruppe anaylsiert werden und wenn ein Grossteil der Zielgruppe über keine Grafikkarte verfügt, dann muss ECS und AGA unterstützt werden. Wichtig ist, dass ein Spiel nicht nur die LowEnd-AMIGA's unterstützt, sondern auch die High-End-AMIGA's. Ein modernes Spiel muss also auf verschiedenen Grafiksystemen lauffähig sein.

Es wird im folgenden nicht zwischen ECS und AGA unterschieden, da bei system-konformer Programmierung dieser Unterschied fast nicht ins Gewicht fällt.
Es wird diese Art der Programmierung als PAL-Programmierung bezeichnet, da solche Spiele dann im PAL-Modus laufen.

Spiele für ECS/AGA waren zum allergrössten Teil nicht systemkonform programmiert worden, da das AMIGA-OS keine allzugute Unterstützung der Spieleprogrammierung

anbot. Es wird im folgenden eine Möglichkeit gezeigt, wie systemkonforme Spiele mit laufendem Betriebssystem entwickelt werden können.

Im folgenden sind auch einige Beispiele in Assembler dargestellt. Der Sinn und Zweck der Funktion kann natürlich problemlos auf Hochsprachen übertragen werden. Die Assembler-Ausschnitte sind direkte Auszüge aus dem Quelltext zum Voxelspace-Demo.

Grundsätzlich funktioniert ein systemkonformes ECS/AGA-Spiel mit Double-Buffering, genau so wie ein hardwarenahes Spiel. Deswegen wird in den folgenden Beispielen immer mit zwei Bitplanes operiert. Die Reihenfolge der folgenden Menupunkte entspricht auch der Reihenfolge in der Programmierung.

All diese Ausführungen gehen davon aus, dass das Spiel die ganze Bildfläche ausnützt und dass nicht mehrere Fenster vorhanden sind. Spiele, wie z.B. Strategiespiele oder Wirtschaftssimulationen können natürlich mit mehreren Fenstern arbeiten. In diesen Fällen ist aber Darstellungs-Geschwindigkeit sowieso kein Problem. Deswegen wird hier auch nicht darauf eingegangen.
Die folgenden Ausführungen sind vor allem für 3D-Spiele und ähnliches von Bedeutung.

Allozieren der Bitplanes
Oeffnen eines Screens
Zuweisung der Farben
Oeffnen eines Fensters
Löschen des Mauszeigers
Erstellen der Grafik
Umschalten der Bildpuffer
Schliessen des Fensters
Schliessen des Screens
Freigeben der Bitplanes

1.9 Allozieren der Bitplanes

Zunächst müssen zwei Bildpuffer alloziert werden, welche wiederum aus mehreren Bitplanes bestehen. Die Anzahl der Bitplanes hängt von der Anzahl Farben ab, welche dargestellt werden sollen. Bei AGA sind das in der Regel acht.

Es wird jetzt ein Verfahren gezeigt, welches auch mit frühen Betriebssystem-Versionen zusammenarbeitet. Es gibt auch noch alternative Wege, welche eine höhere Version erfordern.

Zunächst mal muss Platz für zwei Bitmap-Strukturen gemacht werden. Dies kann sowohl statisch wie auch dynamisch erfolgen. Der Aufbau der Bitmap-Strukturen ist in der Include-Datei 'graphics/gfx.i' beschrieben.

Danach werden beide Bitmap-Strukturen mittels 'graphics/InitBitMap' initialisiert. Dieser Funktion wird die Adresse einer Bitmap-Struktur übergeben, sowie Höhe, Breite und Tiefe der Bitmap.

Anschliessend wird für jede zu allozierende Bitmap die Funktion 'AllocRaster' aufgerufen, welche den nötigen Speicher für die Bitplanes alloziert. Die Rückgabewert werden anschliessend in die Bitmap-Struktur als PlanePtrs eingetragen. Man kann auch mehrere Bitplanes auf einmal allozieren, wie unten

beschrieben.

Von nun an werden die beiden Bitmap-Strukturen wie folgt bezeichnet:

ActualBitmap : Die Bitmap, welche gerade aktiv ist. Die Bitplanes, welche in
der Bitmap-Struktur eingetragen sind, werden gerade dargestellt.
HiddenBitmap : Die Bitmap, welche gerade inaktiv ist. Die Bitplanes, welche in
der Bitmap-Struktur eingetragen sind, stehen für das Erstellen
der Grafik zur Verfügung.

Es folgt ein Ausschnitt aus dem Voxelspace-Quelltext, welcher die Bitplanes alloziert:

```
*****************************************************************************
*
*       d0 = SetupBitmaps
*
*       prepares two bitmaps for double buffering
*
*       Out:
*       d0 = error code
*
*       error codes:    -1 = success
*                        4 = not enough memory
*****************************************************************************
SetupBitmaps
                movem.l d1/d5-a2,-(sp)
                moveq   #2-1,d7                 ;zwei Bitmap-Strukturen
                lea     bitmap1,a0              ;a0 -> erste Bitmap-Struktur
                move.l  a0,ActualBitmap         ;bitmap1 ist die ActualBitmap
.loop
                move.l  a0,a2
                lea     bm_Planes(a2),a2        ;a2 -> Bitplane-Zeiger-Array
                moveq   #8,d0                   ;Farbtiefe (256 Farben)
                move.l  #320,d1                 ;Breite der Bitplanes
                move.l  #256,d2                 ;Höhe der Bitplanes
                CALLGRAF        InitBitMap      ;Bitmap-Struktur initialisieren
                move.l  #320,d0                 ;Breite einer Bitplane
                move.l  #256*8,d1               ;Höhe einer Bitplane * 8
                CALLGRAF        AllocRaster     ;Allokation von 8 Bitplanes
                tst.l   d0                      ;genug Speicher vorhanden?
                beq.b   .error                  ;nein -> Fehlermeldung
                move.l  d0,a0
                move.l  #(256*8*320/8/4)-1,d5
.clear
                clr.l   (a0)+                   ;Ein Bildpuffer wird gelöscht
                subq.l  #1,d5
                bne.b   .clear
                moveq   #8-1,d6                 ;Alle 8 Bitplane-Ptr werden in
.loop2                                          ;die Bitmap-Struktur eingetragen
                move.l  d0,(a2)+
                add.l   #320/8*256,d0           ;springe zur nächsten Bitplane
                dbra    d6,.loop2
                lea     bitmap2,a0              ;jetzt alles noch einmal für
                move.l  a0,HiddenBitmap         ;die 2. Bitmap (HiddenBitmap)
                dbra    d7,.loop
```

```
                moveq   #-1,d0                  ;Funktion erfolgreich
                bra.b   .end
.error
                moveq   #4,d0                   ;Fehlercode zurückgeben
.end
                movem.l (sp)+,d1/d5-a2
                rts
```

1.10 Oeffnen eines Screens

Nach dem Allozieren der Bitmaps wird ein Screen geöffnet. Dazu bietet sich die Funktion 'OpenScreenTagList' der intuition.library an. Diese Funktion erwartet eine NewScreen-Struktur oder eine Tagliste. In unserem Beispiel verwenden wir nur eine Tagliste.

Wenn die Tagliste in Assembler realisiert wird, kann sie folgendermassen aussehen (Beispiel aus dem Voxelspace-Quelltext):

```
ScreenTags
                dc.l    SA_Left                 ;linker Rand des Screens
                dc.l    0
                dc.l    SA_Top                  ;oberer Rand des Screens
                dc.l    0
                dc.l    SA_Width                ;Breite des Screens (hier 320)
                dc.l    320
                dc.l    SA_Height               ;Breite des Screens (hier 256)
                dc.l    256
                dc.l    SA_Depth                ;Tiefe des Screens (hier 256
                dc.l    8                       ;Farben)
                dc.l    SA_BitMap               ;Zeiger auf eine Bitmap-Struktur
                dc.l    bitmap1
                dc.l    SA_Quiet                ;Verhindert, das Intuition selber
                dc.l    TRUE                    ;am Screen herumbastelt
                dc.l    SA_Type                 ;ScreenTyp
                dc.l    CUSTOMSCREEN
                dc.l    TAG_DONE
```

Wichtig ist, dass das Langwort nach SA_Bitmap einen Zeiger auf eine initialisierte Bitmap-Struktur enthält. Wurde die Bitmap dynamisch alloziert, muss dieser Zeiger noch hier eingetragen werden.

Jetzt kann 'OpenScreenTagList' aufgerufen werden. Diese Funktion liefert einen Zeiger auf eine Screen-Struktur zurück, welche noch gebraucht wird. Aus der Screen-Struktur lässt sich nämlich der Zeiger auf den ViewPort ermitteln, der im späteren Verlauf noch benötigt wird. Folgender Code zeigt das Oeffnen eines Screens:

```
                sub.l   a0,a0                   ;Keine NewScreen-Struktur
                lea     ScreenTags,a1           ;Zeiger auf obige Tagliste
                CALLINT OpenScreenTagList       ;Screen öffnen
                moveq   #5,d1                   ;Fehlercode bereitstellen
                move.l  d0,_Screen              ;Screen-Zeiger speichern
                beq.w   .error                  ;Ist er Null? -> Fehler
```

```
                move.l  d0,a0
                move.l  d0,ScreenAddress        ;In Window-Tagliste eintragen
                lea     sc_ViewPort(a0),a0      ;Adresse des Viewports ermitteln
                move.l  a0,_VPort               ;Viewportadresse speichern
```

Auf die Zeile 'move.l d0,ScreenAddress' kommen wir im Kapitel
'Oeffnen eines Fensters' noch zu sprechen.

1.11 Zuweisung der Farben

Dieser Abschnitt gilt sowohl für ECS/AGA wie auch Grafikkarten-Programmierung.

Nach dem Screen geöffnet wurde, steht auch die Adresse des ViewPorts zur
Verfügung. Damit lassen sich jetzt die gewünschten Farben dem Screen zuordnen.
Hier gilt es zwei Fälle zu unterscheiden:

1. ECS/OCS

Wenn das Spiel ECS/OCS unterstützen soll, muss folgender Weg eingeschlagen
werden:

Die Zuweisung der Farben geschieht mit der Systemfunktion 'graphics/LoadRGB4'.
Diese Funktion erwartet einen Zeiger auf den Viewport, eine Tabelle aller
Farben und die Anzahl aller Farben. Die Farbtabelle ist ein ganz normales
Array of USHORT (2 Bytes), dessen Elemente den RGB-Wert der Farbe angeben.
Beispiel für eine Farbtabelle mit 4 Farben:

```
ColorTable
                        dc.w    0               ;schwarz
                        dc.w    $f00            ;rot
                        dc.w    $00f            ;blau
                        dc.w    $fff            ;weiss
```

Der Code zum Zuweisen der Farben sieht dann folgendermassen aus (Beispiel
für 32 Farben):

```
                move.l  _VPort,a0               ;a0 -> ViewPort
                lea     ColorTable,a1           ;a1 -> Farbtabelle
                moveq   #32,d0                  ;d0 = Anzahl Farben
                CALLGRAF        LoadRGB4
```

2. AGA / CyberGFX

Wenn AGA/CyberGFX unterstützt werden soll, sieht der Vorgang ein wenig anders aus:

Die Zuweisung der Farben geschieht mit der Systemfunktion 'graphics/LoadRGB32'.
Diese Funktion erwartet einen Zeiger auf den ViewPort und einen Zeiger auf
eine Farbtabelle. Diese Tabelle hat aber einen anderen Aufbau wie bei der
ECS/OCS-Variante:

Das erste Wort der Tabelle stellt die Anzahl der Farben dar, welche geladen
werden sollen.
Das zweite Wort die erste Farbnummer, welche geladen werden soll (üblicherweise
Null).
Dann folgt die eigentliche Tabelle. Jede Farbe besteht aus 3 Langworten, wobei

das erste Langwort den Rot-Anteil, das zweite Landwort den Grün-Anteil und das dritte Langwort den Blau-Anteil angibt. Noch eine wichtige Besonderheit: Die Farbwerte müssen LINKSBUENDIG angegeben werden! Bei 8-Bit-Farbwerten muss der eigentliche Farbwert um 24 Bits nach links verschoben werden.
Den Abschluss der Tabelle markiert ein Null-Langwort.

Beispiel für eine Farbtabelle mit 4 Farben:

```
ColorTable
        dc.w    4                               ;4 Farben
        dc.w    0                               ;erste Farbe ist Farbnummer 0
        dc.l    $ff000000,$ff000000,0           ;gelb
        dc.l    0,0,0                           ;schwarz
        dc.l    $7f000000,$7f000000,$7f000000   ;grau
        dc.l    0,0,$40000000                   ;dunkelblau
        dc.l    0                               ;Abschluss der Tabelle
```

Der Code zum Zuweisen der Farben sieht dann folgendermassen aus:

```
                move.l  _VPort,a0               ;a0 -> ViewPort
                lea     ColorTable,a1           ;a1 -> Farbtabelle
                CALLGRAF        LoadRGB32
```

1.12 Oeffnen eines Fensters

Dieser Abschnitt gilt sowohl für ECS/AGA- wie auch für CyberGFX-Programmierung

Jetzt kann das Fenster geöffnet werden. Dazu bietet sich die Funktion 'OpenWindowTagList' der intuition.library an. Diese Funktion erwartet eine NewWindow-Struktur oder eine Tagliste. In unserem Beispiel verwenden wir nur eine Tagliste.

Wenn die Tagliste in Assembler realisiert wird, kann sie folgendermassen aussehen (Beispiel aus dem Voxelspace-Quelltext):

```
WindowTags
                dc.l    WA_Left                 ;linker Rand des Fensters
                dc.l    0
                dc.l    WA_Top                  ;oberer Rand des Fensters
                dc.l    0
                dc.l    WA_Width                ;Breite des Fensters (PAL: 320)
WinWidth
                dc.l    320                     ;CyberGFX : dynamisch
                dc.l    WA_Height               ;Höhe des Fensters (PAL: 256)
WinHeight
                dc.l    256                     ;CyberGFX : dynamisch
                dc.l    WA_Activate             ;Fenster soll gleich aktiviert
                dc.l    TRUE                    ;werden
                dc.l    WA_Borderless           ;Fenster soll keinen Rahmen haben
                dc.l    TRUE
                dc.l    WA_RMBTrap              ;rechte Maustaste soll abgefangen
                dc.l    TRUE                    ;werden
                dc.l    WA_ReportMouse          ;Mausbewegungen sollen mitgeteilt
                dc.l    TRUE                    ;werden
                dc.l    WA_IDCMP                ;zu unterstützende IDCMP-Flags
```

```
                dc.l    IDCMP_MOUSEBUTTONS!IDCMP_RAWKEY!IDCMP_MOUSEMOVE! ↩
                   IDCMP_DELTAMOVE!IDCMP_ACTIVEWINDOW!IDCMP_INACTIVEWINDOW
                dc.l    WA_CustomScreen         ;Adresse des übergeordneten  ↩
                   Screens
                dc.l    0
                dc.l    TAG_DONE
```

Das Langwort nach WA_CustomScreen muss mit der Adresse des Screens gefüllt werden, wie im Kapitel 'Oeffnen eines Screens' beschrieben.

Diese Window-Struktur ist natürlich nur ein Beispiel. Je nach Anwendung sieht sie unterschiedlich aus. Es hängt z.B. ganz davon ab, welche Interaktion des Benutzers behandelt werden soll. Wenn keine Mausbedienung vorgesehen ist, macht es auch keinen Sinn WA_ReportMouse und die entsprechenden IDCMP-Flags anzugeben.

Auf die IDCMP-Flags soll noch näher eingegangen werden. Folgende Flags könnten für Spiele von Interesse sein:

```
IDCMP_MOUSEBUTTONS      : Der User hat eine Maustaste gedrückt.
IDCMP_MOUSEMOVE         : Der User hat die Maus bewegt. IDCMP_MOUSEMOVE sollte
                          immer gleichzeitig mit IDCMP_DELTAMOVE verwendet
                          werden.
IDCMP_RAWKEY            : Der User hat eine Taste gedrückt. Bei diesem Modus
                          werden die Tasten unverarbeitet übergeben. Damit
                          können Spezialtasten wie Ctrl, Alt, Shift usw.
                          getestet werden.
IDCMP_ACTIVEWINDOW      : Kann ev. gebraucht werden, wenn jemand durch
                          Umschalten der Screens ein Spiel in den Pausen-
                          Modus gebracht hat. Durch Aktivieren des Fensters
                          mit der Maus kann das Spiel fortgesetzt werden.
IDCMP_INACTIVEWINDOW    : Kann gebraucht werden, um das Spiel in den Pausen-
                          Modus zu schalten, wenn jemand die Screens umschaltet
                          und das Fenster inaktiviert.
IDCMP_DELTAMOVE         : Sollte immer verwendet werden, wenn IDCMP_MOUSEMOVE
                          verwendet wird.
IDCMP_VANILLAKEY        : Der User hat eine Taste gedrückt. Die Daten werden
                          hier schon überarbeitet übergeben. Damit können
                          nicht alle Tasten überwacht werden.
```

Jetzt kann 'OpenWindowTagList' aufgerufen werden. Diese Funktion liefert einen Zeiger auf eine Window-Struktur zurück, welche noch gebraucht wird. Aus der Window-Struktur lässt sich nämlich der Zeiger auf den RastPort ermitteln, der im späteren Verlauf noch benötigt wird. Folgender Code zeigt das Oeffnen eines Windows:

```
                sub.l   a0,a0                   ;Keine NewWindow-Struktur
                lea     WindowTags,a1           ;Zeiger auf obige Tagliste
                CALLINT OpenWindowTagList       ;Fenster öffnen
                moveq   #6,d1                   ;Fehlercode bereitstellen
                move.l  d0,_Window              ;Window-Zeiger speichern
                beq.b   .error                  ;Ist er Null? -> Fehler
                move.l  d0,a0
                move.l  wd_RPort(a0),_RPort     ;Adresse des Rastports speichern
```

1.13 Löschen des Mauszeigers

Es soll hier noch eine Möglichkeit gezeigt werden, wie der Mauszeiger zum Verschwinden gebracht werden kann. Er kann nämlich unter Umständen ziemlich lästig werden.

Zunächst muss ein kleiner Speicherbereich aus dem Chip-RAM alloziert werden (z.B. 16 Bytes). Dieser wird dann als Mauszeiger verwendet, welcher dann komplett transparent und somit unsichtbar wird.

Danach wird die System-Funktion 'intuition/SetPointer' aufgerufen. Das kann nun folgendermassen aussehen:

```
move.l  #16,d0                  ;16 Bytes zu allozieren
move.l  #MEMF_CHIP!MEMF_CLEAR,d1
CALLEXEC        AllocVec        ;Chip-RAM allozieren
move.l  d0,NullPointer          ;Adresse speichern
move.l  d0,a1                   ;nach a1 für SetPointer
move.l  _Window,a0              ;Adresse des Fensters nach a0
moveq   #1,d0
moveq   #1,d1
moveq   #0,d2
moveq   #0,d3
CALLINT SetPointer              ;Mauszeiger löschen
```

Soll der Mauszeiger wieder erscheinen, kann dies mittels 'intuition/ClearPointer' vollzogen werden.

Wenn das Fenster geschlossen wird, kommt der Mauszeiger automatisch wieder zum Vorschein.

1.14 Erstellen der Grafik

Nach dem alle Vorbereitungen getroffen worden sind, gelangt das Spiel in die Hauptschleife, wo Iteration für Iteration das nächste Bild berechnet und dargestellt wird.

In den modernen Spielen, wie bzw. 3D-Spielen wird die Grafik zunächst im Fast-RAM erstellt und zwar im Chunky-Format (d.h. ein Byte pro Pixel). Danach wird dieses Bild mit Hilfe eines Konvertieralgorithmus konvertiert. Diese sogenannten C2P oder ChunkyToPlanar-Konverter existieren mittlerweilen zuhauf. Es ist darauf zu achten, dass ein Algorithmus ausgewählt wird, der zur Aufgabenstellung passt. Oftmals funktionieren solche Algorithmen nur für bestimmte Farbtiefen.

Eine solcher C2P-Funktion sollte für den PPC umgesetzt werden. Der Algorithmus ist zwar überhaupt nicht gut geeignet, um die Leistung des PPC zu demonstrieren, da er sehr speicherlastig ist und in das Chip-RAM schreibt. Andererseits lassen sich kleine Performance-Steigerungen gegenüber einem 68K-Prozessor erzielen. Das kann ev. doch sichtbare Auswirkungen auf das Resultat haben.

Nachdem die eigentliche Hintergrund-Grafik erstellt worden ist, werden eventuell noch Statusanzeigen und Texte dargestellt. Diese können dann mit den

konventionellen Funktionen der graphics.library (z.B. Move, Text, ...) in den unsichtbaren Bildpuffer gezeichnet werden, bevor die Puffer ausgetauscht werden und die Veränderungen sichbar gemacht werden. Ein Beispiel dazu ist beim Demo 'voxelspace' zu sehen, wo nach Berechnung der Landschaft noch mittels 'Move' und 'Text' die Statusanzeige erstellt wird.

Wichtig: Wenn die Standard-Funktionen der graphics.library benützt werden, muss vor deren Gebrauch die Rastport-Struktur sowie die RasInfo-Struktur innerhalb des ViewPorts angepasst werden, da diese Funktionen direkt auf den aktuellen Bildpuffer angewendet werden. Details dazu im folgenden Kapitel 'Umschalten der Bildpuffer'.

1.15 Umschalten der Bildpuffer

Eine typische Hauptschleife könnte etwa folgendermassen strukturiert sein:

- Bild berechnen und ins FAST-RAM legen
- Bild in den unsichtbaren Bildpuffer kopieren/konvertieren
- Sichtbaren und unsichtbaren Bildpuffer austauschen

Auf diese Art und Weise können flüssige und ruckfreie Animationen erzeugt werden (Double Buffering). Jetzt geht es darum, wie die beiden Bildpuffer ausgetauscht werden können.

Wie im vorherigen Kapitel 'Erstellen der Grafik' erwähnt, müssen einige Vorbereitungen getroffen werden, wenn Standard-Funktionen der graphics.library verwendet werden. Diese Funktionen wirken immer auf den aktuellen Bildpuffer, wir wollen aber, dass diese Ausgaben in den unsichtbaren Puffer gemacht werden, damit die Animation sauber vonstatten geht.

Der Vorgang des Umschaltens sieht nun folgendermassen aus:

1. RastPort und RasInfo-Struktur anpassen
2. Eventuell Funktionen der graphics.library benützen
3. Obige Veränderungen mittels 'ScrollVPort' sichtbar machen
4. Bitmap-Zeiger vertauschen

Zunächst mal muss dafür gesorgt werden, dass die Funktionen der graphics.library auf den unsichtbaren Puffer wirken. Dies wird erreicht, indem die unsichtbare Bitmap für diese Funktionen sichtbar gemacht wird, ohne dass die Bitplanes aber effektiv umgeschalten werden. Dies sieht dann folgendermassen aus:

```
move.l  _RPort,a0               ;Adresse des RastPorts besorgen
move.l  HiddenBitmap,a1         ;Adresse der Hiddenbitmap nach a1
move.l  a1,rp_BitMap(a0)        ;eintragen in RastPort
move.l  _VPort,a0               ;Adresse des ViewPorts besorgen
move.l  vp_RasInfo(a0),a0       ;Adresse des RasInfo besorgen
move.l  a1,ri_BitMap(a0)        ;eintragen in RasInfo
```

Danach können Funktionen wie 'Move' und 'Text' angewendet werden. Diese Funktionen wirken somit auf den noch unsichtbaren Puffer.

Als drittes werden die Veränderungen im RastPort und RasInfo sichtbar gemacht. Damit werden die beiden Bildpuffer effektiv umgeschaltet:

```
                move.l  _VPort,a0               ;Adresse des ViewPorts besorgen
                CALLGRAF        ScrollVPort     ;Bildpuffer umschalten
```

Zuletzt werden die beiden Zeiger Actualbitmap und Hiddenbitmap miteinander vertauscht, so dass das Spiel jetzt das nächste Bild jetzt im anderen Bildpuffer erstellt, welches jetzt unsichtbar geworden ist:

```
                move.l  ActualBitmap,d0
                move.l  HiddenBitmap,ActualBitmap
                move.l  d0,HiddenBitmap
```

1.16 Schliessen des Fensters

Nach dem das Spiel beendet wurde, sollte ein möglichst sauberer Ausstieg vollzogen zogen. Das heisst, dass z.B. aller Speicher wieder freigegeben werden sollte. Das Fenster sollte natürlich wieder geschlossen werden. Dies geschieht mittels der Funktion 'intuition/CloseWindow':

```
                move.l  _Window,d0              ;Hole Adresse der Window-Struktur
                beq.b   .nowindow               ;War Window geöffnet?
                move.l  d0,a0                   ;Adresse nach a0
                CALLINT CloseWindow             ;Window schliessen
.nowindow
```

1.17 Schliessen des Screens

Genauso wie das Fenster, sollte auch der Screen geschlossen werden. Dies geschieht mit 'intuition/CloseScreen':

```
                move.l  _Screen,d0              ;Hole Adresse der Screen-Struktur
                beq.b   .noscreen               ;War Screen geöffnet?
                move.l  d0,a0                   ;Adresse nach a0
                CALLINT CloseScreen             ;Screem schliessen
.noscreen
```

1.18 Freigeben der Bitplanes

Die Bitplanes, welche alloziert wurden, sollten wieder freigegeben werden. Zunächst wird für jede Bitplane die Funktion 'FreeRaster' aufgerufen werden, welche das Gegenstück zu 'AllocRaster' darstellt.

Wenn die Bitmap-Strukturen dynamisch alloziert wurden, sollten diese ebenfalls freigegeben werden.

Es folgt ein Ausschnitt aus dem Voxelspace-Demo, welches die Bitplanes freigibt:

```
*****************************************************************************
*
*       FreeBitmaps
*
*       frees all the memory allocated by 'AllocRaster'
*
*****************************************************************************
FreeBitmaps
                movem.l d0/d1/d6-a2,-(sp)
                moveq   #2-1,d7                 ;zwei Bitmap-Strukturen
                lea     bitmap1,a0              ;a0 -> 1. Bitmap
.loop
                move.l  a0,a2
                lea     bm_Planes(a2),a2        ;a2 -> Array of Bitplane-Zeiger
                moveq   #8-1,d6                 ;8 Bitplanes freizugeben
.loop2
                move.l  (a2)+,d0                ;Lies Bitplane-Zeiger
                beq.b   .next                   ;Null? -> keine Freigabe
                move.l  d0,a0
                move.l  #320,d0                 ;Breite der Bitplane
                move.l  #256,d1                 ;Höhe der Bitplane
                CALLGRAF        FreeRaster      ;Bitplane freigeben
.next
                dbra    d6,.loop2
                lea     bitmap2,a0              ;nocheinmal für 2. Bitmap
                dbra    d7,.loop
                movem.l (sp)+,d0/d1/d6-a2
                rts
```

1.19 Grafikkarten-Programmierung

Die systemkonforme Grafikkarten-Programmierung unterscheidet sich ein wenig von der ECS/AGA-Programmierung. Ein wesentlicher Unterschied: es findet kein Double-Buffering statt. Normalerweise beginnt ein Spiel/Demo ohne Double-Buffering an zu flackern, bei richtiger Programmierung von CyberGFX treten solche Effekte fast gar nicht auf.

Der Ablauf der Programmierung sieht ähnlich aus wie beim ECS/AGA-Modell:

Wahl des Bildschirm-Modus
Oeffnen eines Screens
Zuweisung der Farben
Oeffnen eines Fensters
Löschen des Mauszeigers
Erstellen des temp. RastPort
Erstellen der Grafik
Schliessen des Fensters
Schliessen des Screens
Freigeben des temp. RastPort

1.20 Wahl des Bildschirm-Modus

Grafikkarten erlauben es in der Regel, den Bildschirm-Modus relativ frei zu definieren. So haben viele AMIGA-Benützer mit Grafikkarten unterschiedliche Bildschirm-Modi eingestellt. Es ist gerade für Spiele sehr wichtig, dass sie mit allen vernünftigen Bildschirm-Modi zusammenarbeiten.

Die Wahl des Bildschirm-Modus hat auch eine direkte Auswirkung auf die Performance des Spiels. Je kleiner die Auflösung, desto schneller läuft ein Spiel. Auf diese Weise ist es sogar möglich, bestehende 68K-Spiele superflüssig zu spielen (z.B. mit einem Screenmodus von 192*128). Die Grafik sieht dann zwar grob aus, aber in der Animation fällt das oftmals gar nicht auf.

Ein Spiel sollte also dem Benützer die Möglichkeit geben, seinen bevorzugten Bildschirm-Modus auszuwählen. CyberGFX bietet hier eine Funktion an, welche einen Screenmode-Requester anzeigt und das Resultat der Wahl wieder zurückgibt. Voraussetzung für deren Benützung ist, dass die cybergraphics.library erfolgreich geöffnet wurde.

Die Funktion 'CModeRequestTagList' erwartet eine Tagliste und liefert die DisplayID des gewählten ScreenModus zurück, welcher in der Folge noch verwendet wird. Mit der Tagliste lassen sich die darzustellenden Screenmodi im Requester filtern. Wenn das Spiel beispielsweise nicht mit gewissen Dimensionen zusammenarbeitet, sollte das so angegeben werden. Näheres dazu steht in den Autodocs zur cybergraphics.library.

In der Regel wird man die ScreenMode-Eintragungen nach Farbtiefe filtern. Spiele werden in den seltensten Fällen mit 16- oder 24-Bit-Modi zusammenarbeiten. Die Tagliste sieht dann folgendermassen aus (die Include-Datei zur cybergraphics.library muss ebenfalls eingebunden worden sein):

```
CyberModeTags   dc.l    CYBRMREQ_CModelArray
                dc.l    ColorModel
                dc.l    TAG_DONE
ColorModel
                dc.w    PIXFMT_LUT8
                dc.w    -1
```

Somit kann jetzt der Screenmode-Requester dargestellt werden:

```
                sub.l   a0,a0                   ;muss NULL sein
                lea     CyberModeTags,a1        ;Zeiger auf Tagliste
                CALLCYBERGFX    CModeRequestTagList     ;Requester darstellen
                moveq   #8,d1                   ;Fehlercode bereitstellen
                tst.l   d0                      ;war Funktion erfolgreich?
                beq.w   .error                  ;nein -> Fehler
                moveq   #0,d1                   ;Fehlercode bereitstellen
                cmp.l   #-1,d0                  ;Hat der User 'Abbruch' gewählt?
                beq.w   .error                  ;dann -> verlasse Spiel
                move.l  d0,DispID               ;speichere DisplayID
```

1.21 Oeffnen eines Screens

Im Prinzip funktioniert das Oeffnen des Screens, wie gehabt, mittels 'OpenScreenTagList'. Nur muss jetzt neu beachtet werden, dass der Screenmodus vom Benutzer gewählt wurde, die Daten wie Höhe, Breite und DisplayID sind

dynamisch. Diese Wert müssen zunächst ermittelt werden und danach in die TagListe für 'OpenScreenTagList' eingetragen werden.

Die DisplayID wird von 'CModeRequestTagList' zurückgegeben (siehe voriges Kapitel 'Auswahl eines Bildschirm-Modus'
ermittelt werden. Dazu kann eine weiter CyberGFX-Funktion benützt werden: 'GetCyberIDAttr'. Sie verlangt die DisplayID und einen Modus als Parameter. Der Modus gibt an, welche Information gewünscht wird. Die Funktion wird jetzt zweimal aufgerufen. Das erste mal mit dem Parameter 'CYBRIDATTR_HEIGHT', um die Höhe zu ermitteln, danach mit 'CYBRIDATTR_WIDTH', um die Breite zu ermitteln. All diese Werte werden jetzt in die Screen-Tagliste eingetragen.

Nach dem Oeffnen des Screens sollten die Breite und Höhe aus der Screen-Struktur ausgelesen und in die Window-Tagliste eingetragen werden. Zusätzlich wird aus der Screen-Struktur noch die Adresse des ViewPorts ermittelt und abgespeichert.

Die Screen-Tagliste kann folgenden Aufbau haben:

```
ScreenTags_C
                dc.l    SA_Quiet                ;verhindert, dass Intuition selber
                dc.l    TRUE                    ;am Screen herumbastelt
                dc.l    SA_Width                ;Breite des Screens
ScreenWidth
                dc.l    0
                dc.l    SA_Height               ;Höhe des Screens
ScreenHeight
                dc.l    0
                dc.l    SA_Depth                ;Tiefe des Screens
                dc.l    8
                dc.l    SA_DisplayID            ;DisplayID des Screens
DispID
                dc.l    0
                dc.l    TAG_DONE
```

Der Code sieht dann folgendermassen aus:

```
                move.l  DispID,d1               ;DisplayID holen
                move.l  #CYBRIDATTR_HEIGHT,d0   ;Höhe soll ermittelt werden
                CALLCYBERGFX    GetCyberIDAttr  ;Höhe wird ermittelt
                move.l  d0,ScreenHeight         ;und in Tagliste eingetragen
                move    d0,AreaHeight           ;Höhe speichern
                move.l  DispID,d1               ;DisplayID holen
                move.l  #CYBRIDATTR_WIDTH,d0    ;Breite soll ermittelt werden
                CALLCYBERGFX    GetCyberIDAttr  ;Breite wird ermittelt
                move.l  d0,ScreenWidth          ;und in Tagliste eingetragen
                sub.l   a0,a0                   ;keine NewScreen-Struktur
                lea     ScreenTags_C,a1         ;Zeiger auf Tagliste
                CALLINT OpenScreenTagList       ;Screen oeffnen
                moveq   #5,d1                   ;Fehlercode bereitstellen
                move.l  d0,_Screen              ;Zeiger auf Screen speichern
                beq.w   .error                  ;Ist er Null? -> Fehler
                move.l  d0,a0
                move    sc_Width(a0),AreaWidth  ;Breite speichern
                move    sc_Width(a0),WinWidth+2 ;Breite -> in Window-Tagliste
                move    sc_Height(a0),WinHeight+2 ;Höhe -> in Window-Tagliste
```

```
move.l  d0,ScreenAddress        ;Zeiger auf Screen in Window-TL
lea     sc_ViewPort(a0),a0      ;Adresse des ViewPort ermitteln
move.l  a0,_VPort               ;Adresse speichern
```

1.22 Erstellen eines temp. RastPort

Das eigentliche Kopieren der Bilddaten in den Grafik-Speicher wird von der Funktion 'graphics/WritePixelArray8' übernommen. Nähere Details zu diesem Thema folgen später. Wichtig ist jetzt, dass diese Funktion einen temporären RastPort benötigt.

Zunächst muss Speicher für eine Rastport-Struktur bereitgestellt werden (die Struktur ist in der Include-Datei 'graphics/rastport.i' definiert). Dies kann statisch oder dynamisch geschehen. Im folgenden Beispiel heisst dieser RastPort 'tempRP'.

Jetzt wird mit der Funktion 'graphics/InitRastPort' der Rastport initialisiert. Danach wird eine Bitmap-Struktur und die zugehörigen Bitplanes alloziert, mittels 'graphics/AllocBitMap'. Diese Funktion verlangt nach AMIGA-OS V3.0. Bei Unterstützung von Grafikkarten sollte OS V3.0 vorausgesetzt werden können.

Der Code sieht dann folgendermassen aus:

```
lea     tmpRP,a1                ;a1 -> Platz für temp. RastPort
CALLGRAF        InitRastPort    ;Initialisiere RastPort
moveq   #0,d0
move    AreaWidth,d0            ;Breite des Screens ermitteln
moveq   #1,d1                   ;Höhe = 1 Zeile
moveq   #8,d2                   ;Tiefe = 256 Farben
move.l  #BMF_MINPLANES,d3       ;spezielles Flag
move.l  _RPort,a0               ;Window-RastPort ermitteln
move.l  rp_BitMap(a0),a0        ;Bitmap als 'Friend' übergeben
CALLGRAF        AllocBitMap     ;Bitmap + Bitplanes allozieren
lea     tmpRP,a0                ;Adresse des temp. RastPort -> a0
move.l  d0,rp_BitMap(a0)        ;neue Bitmap eintragen
```

1.23 Erstellen der Grafik

In der Hauptschleife wird, wie gehabt, das Bild zunächst mal im FAST-RAM erstellt. Jetzt muss es auf irgendeine Weise in das Grafik-RAM kopiert werden, damit es dargestellt werden kann.

Es gibt hier prinzipiell zwei Funktionen, welche diese Kopierarbeit übernehmen können:

1. cybergraphics/WritePixelArray
2. graphics/WritePixelArray8

Die erste Funktion hat den Vorteil, dass man auch Teile des Bildschirms kopieren kann. Der Nachteil: Diese Funktion ist UNENDLICH langsam.

Die zweite Funktion ist so schnell, wie es eben möglich ist. Die hat aber leider das Problem, dass sie bei den Parametern relativ eingeschränkt ist. Sie wird in der Regel verwendet, um das komplett erstellte Bild in den Grafikspeicher zu kopieren.

Das allergrösste Problem bei dieser Art von Grafikkarten-Programmierung: Da kein Double-Buffering gemacht werden kann, ist es nicht mehr möglich, nach Erstellen der Grafik im Grafikspeicher weiter grafische Elemente mit den Standard-Funktionen der graphics.library zu erstellen, ohne dass das Spiel aufs Heftigste zu flackern beginnt. Alle zusätzlichen grafischen Elemente müssen bereits im 'ChunkyBuffer' (Bild im FAST-RAM) erstellt werden, was relativ mühsam sein kann, da es nicht mehr mit den Standard-Funktionen der graphics.library gemacht werden kann.

Der Code für das Kopieren des ganzen Bildes vom 'ChunkyBuffer' (Fast-RAM) in den Grafikspeicher kann folgendermassen aussehen:

```
move.l  _RPort,a0               ;Adresse des Rastport ermitteln
moveq   #0,d0                   ;xstart = 0
moveq   #0,d1                   ;ystart = 0
move    AreaWidth,d2            ;xstop = Breite - 1
subq    #1,d2
move    AreaHeight,d3           ;ystop = Höhe - 1
subq    #1,d3
move.l  ChunkyBuffer,a2         ;a2 -> Bilddaten im Fast-RAM
lea     tmpRP,a1                ;a1 -> temporärer RastPort
CALLGRAF        WritePixelArray8 ;Bild kopieren
```

1.24 Freigeben des temp. RastPort

Der temporäre RastPort, welcher für die Funktion 'WritePixelArray8' erstellt worden war, sollte wieder freigegeben werden. Dies lässt sich ganz einfach mit der Funktion 'FreeBitMap' erledigen:

```
lea     tmpRP,a0                ;Adresse des RastPorts -> a0
move.l  rp_BitMap(a0),a0        ;Adresse der Bitmap -> a0
CALLGRAF        FreeBitMap      ;Bitmap freigeben
```

Anschliessend sollte der temp. Rastport selber noch freigegeben werden, wenn er dynamisch alloziert wurde.

1.25 CyberGFX+

Ein sehr grosser Nachteil der oben beschriebenen Grafikkarten-Programmierung ist, dass kein Double-Buffering oder Multi-Buffering realisierbar ist. Das hat auch zur Folge, dass nach dem Erstellen der berechneten Grafik keine zusätzlichen Elemente mit den Standard-Grafik-Funktionen mehr erstellt werden können, was eine sehr grosse Einschränkung darstellt.

Im folgenden ist eine Erweiterung der systemkonformen Grafikprogrammierung

beschrieben, welche mit ein paar Tricks echtes Multibuffering erzeugt. Auch das Erstellen von zusätzlichen Grafik-Elementen ist somit kein Problem mehr.

Diese Technik ist im Prinzip systemkonform, wenn man davon ausgeht, dass die Neupositionierung eines ViewPorts mittels 'ScrollVPort' auch funktioniert und das ist mit CyberGFX der Fall. Wenn aber immer möglich sollten Spiele die Grafikkarten-Ansteuerung ohne Multi-Buffering ebenfalls unterstützen, um eventuellen Problemen aus dem Weg zu gehen.

Der Ablauf der Programmierung sieht ähnlich aus wie beim Modell der normalen Grafikkarten-Programmierung:

Wahl des Bildschirm-Modus
Triple Buffering
Oeffnen eines Screens
Zuweisung der Farben
Oeffnen eines Fensters
Löschen des Mauszeigers
Erstellen des temp. RastPort
Erstellen der Grafik
Schliessen des Fensters
Schliessen des Screens
Freigeben des temp. RastPort
Probleme

1.26 Oeffnen eines Screens

Im Prinzip funktioniert das Oeffnen des Screens, wie gehabt, mittels 'OpenScreenTagList'. Nur muss jetzt neu beachtet werden, dass der Screenmodus vom Benutzer gewählt wurde, die Daten wie Höhe, Breite und DisplayID sind dynamisch. Diese Wert müssen zunächst ermittelt werden und danach in die TagListe für 'OpenScreenTagList' eingetragen werden.

Wie im Abschnitt 'Triple Buffering' beschrieben, wird der Screen mit der dreifachen Höhe geöffnet.

Die DisplayID wird von 'CModeRequestTagList' zurückgegeben (siehe voriges Kapitel 'Auswahl eines Bildschirm-Modus'
ermittelt werden. Dazu kann eine weiter CyberGFX-Funktion benützt werden: 'GetCyberIDAttr'. Sie verlangt die DisplayID und einen Modus als Parameter. Der Modus gibt an, welche Information gewünscht wird. Die Funktion wird jetzt zweimal aufgerufen. Das erste mal mit dem Parameter 'CYBRIDATTR_HEIGHT', um die Höhe zu ermitteln, danach mit 'CYBRIDATTR_WIDTH', um die Breite zu ermitteln. All diese Werte werden jetzt in die Screen-Tagliste eingetragen.

Nach dem Oeffnen des Screens sollte die Breite aus der Screen-Struktur ausgelesen und in die Window-Tagliste eingetragen werden. Zusätzlich wird aus der Screen-Struktur noch die Adresse des ViewPorts ermittelt und abgespeichert.

Zusätzlich zur normalen Grafikkarten-Programmierung werden jetzt die vertikalen Positionen der drei Puffer bestimmt. Die Position des 1. Puffers ist immer Null, die Position des zweiten Puffers ist gleich der Pufferhöhe und die Position des 3. Puffers ist gleich der doppelten Pufferhöhe.

Die Screen-Tagliste kann folgenden Aufbau haben:

```
ScreenTags_C
                dc.l    SA_Quiet                ;verhindert, dass Intuition selber
                dc.l    TRUE                    ;am Screen herumbastelt
                dc.l    SA_Width                ;Breite des Screens
ScreenWidth
                dc.l    0
                dc.l    SA_Height               ;Höhe des Screens
ScreenHeight
                dc.l    0
                dc.l    SA_Depth                ;Tiefe des Screens
                dc.l    8
                dc.l    SA_DisplayID            ;DisplayID des Screens
DispID
                dc.l    0
                dc.l    TAG_DONE
```

Der Code sieht dann folgendermassen aus:

```
                move.l  DispID,d1               ;DisplayID holen
                move.l  #CYBRIDATTR_HEIGHT,d0   ;Höhe soll ermittelt werden
                CALLCYBERGFX    GetCyberIDAttr  ;Höhe wird ermittelt
                move    d0,AreaHeight           ;Höhe speichern
                move.l  d0,d1                   ;Höhe mit drei multiplizieren
                add.l   d0,d0
                add.l   d1,d0
                move.l  d0,ScreenHeight         ;und in Tagliste eingetragen
                move.l  DispID,d1               ;DisplayID holen
                move.l  #CYBRIDATTR_WIDTH,d0    ;Breite soll ermittelt werden
                CALLCYBERGFX    GetCyberIDAttr  ;Breite wird ermittelt
                move.l  d0,ScreenWidth          ;und in Tagliste eingetragen
                sub.l   a0,a0                   ;keine NewScreen-Struktur
                lea     ScreenTags_C,a1         ;Zeiger auf Tagliste
                CALLINT OpenScreenTagList       ;Screen oeffnen
                moveq   #5,d1                   ;Fehlercode bereitstellen
                move.l  d0,_Screen              ;Zeiger auf Screen speichern
                beq.w   .error                  ;Ist er Null? -> Fehler
                move.l  d0,a0
                move    sc_Width(a0),AreaWidth  ;Breite speichern
                move    sc_Width(a0),WinWidth+2 ;Breite -> in Window-Tagliste
                move    sc_Height(a0),WinHeight+2 ;Höhe -> in Window-Tagliste
                move.l  d0,ScreenAddress        ;Zeiger auf Screen in Window-TL
                lea     sc_ViewPort(a0),a0      ;Adresse des ViewPort ermitteln
                move.l  a0,_VPort               ;Adresse speichern
                move    AreaHeight,d1           ;Pufferhöhe lesen
                clr     ActualOffset            ;y-Pos. Puffer 1 = 0
                move    d1,HiddenOffset         ;y-Pos. Puffer 2 = d1
                add     d1,d1
                move    d1,ThirdOffset          ;y-Pos. Puffer 3 = d1*2
```

1.27 Erstellen der Grafik

In der Hauptschleife wird, wie gehabt, das Bild zunächst mal im FAST-RAM erstellt. Jetzt muss es auf irgendeine Weise in einen unsichtbaren Puffer innerhalb des Grafik-RAM kopiert werden, damit es dargestellt werden kann.

Es gibt hier prinzipiell zwei Funktionen, welche diese Kopierarbeit übernehmen können:

1. cybergraphics/WritePixelArray
2. graphics/WritePixelArray8

Die erste Funktion hat den Vorteil, dass man auch Teile des Bildschirms kopieren kann. Der Nachteil: Diese Funktion ist UNENDLICH langsam.

Die zweite Funktion ist so schnell, wie es eben möglich ist. Die hat aber leider das Problem, dass sie bei den Parametern relativ eingeschränkt ist. Sie wird in der Regel verwendet, um das komplett erstellte Bild in den Grafikspeicher zu kopieren.

Nachdem die Grafik-Daten kopiert wurden, können jetzt mit den Standard-Funktionen der graphics.library zusätzliche grafische Elemente im unsichtbaren Puffer erzeugt werden.

Jetzt werden die Puffer umgeschaltet. Dies geschieht mit der Funktion 'graphics/ScrollVPort'. Nach dem Umschalten werden noch die Positionswerte (ActualOffset, HiddenOffset und ThirdOffset) rotiert, damit nächstes Mal der richtige Puffer ausgewählt wird.

Der Code für das Kopieren des ganzen Bildes vom 'ChunkyBuffer' (Fast-RAM) in den Grafikspeicher kann folgendermassen aussehen:

```
move.l  _RPort,a0               ;Adresse des Rastport ermitteln
moveq   #0,d0                   ;xstart = 0
moveq   #0,d1                   ;ystart = 0
move    AreaWidth,d2            ;xstop = Breite - 1
subq    #1,d2
move    AreaHeight,d3           ;ystop = Höhe - 1
subq    #1,d3
add     HiddenOffset,d2         ;unsichtbaren Puffer auswählen
add     HiddenOffset,d3         ;unsichtbaren Puffer auswählen
move.l  ChunkyBuffer,a2         ;a2 -> Bilddaten im Fast-RAM
lea     tmpRP,a1                ;a1 -> temporärer RastPort
CALLGRAF        WritePixelArray8 ;Bild kopieren
```

An dieser Stelle können jetzt die zusätzlichen grafischen Objekte erstellt werden. Danach werden die Puffer umgeschaltet und die Positionswerte rotiert:

```
move.l  _VPort,a0               ;Adresse des ViewPorts ermitteln
move    HiddenOffset,d0         ;Position des unsichbaren
neg     d0                      ;Puffers ermitteln und negieren
move    d0,vp_DyOffset(a0)      ;in ViewPort eintragen
CALLGRAF        ScrollVPort     ;Puffer umschalten
move    ActualOffset,d0         ;Positionswerte der Puffer
move    HiddenOffset,ActualOffset ;rotieren
move    ThirdOffset,HiddenOffset
move    d0,ThirdOffset
```

1.28 TurboGFX

Vermutlich hat sich der eine oder andere schon gefragt, warum eigentlich bei der Grafikkarten-Programmierung das Bild zunächst im FAST-RAM erstellt wird und danach in den Grafikspeicher kopiert wird. Im Prinzip sollte man doch das Bild direkt im Grafikspeicher erstellen können. Damit sollte ein Spiel sogar noch schneller funktionieren.

In diesem Abschnitt geht es jetzt darum, zu zeigen, dass so etwas möglich ist und wie das zu programmieren ist.

Hinweis: Diese hier vorgestellte Technik ist im Prinzip systemkonform. Der direkte Zugriff auf das Grafik-RAM bzw. auf die Bitmap des Screens wird durch Locking-Mechanismen geschuetzt, wie es auch von CyberGFX verlangt wird. Diese Technik ist allerdings als LowLevel zu betrachten und es ist ueberhaupt keine gute Idee, sich völlig darauf abzustuetzen. Zudem kann es durchaus sein, dass diese Technik auf anderen Systemen mit anderer Gfx-Software nicht funktioniert. Deswegen sollten Spiele und Demos diese Technik als Ergänzung zu den anderen Techniken anbieten.

Der Begriff TURBOGFX stammt vom einem CLI-Parameter des Voxelspace-Demos. Dieses Demo war mein Prototyp für diese neue Technik. Das Voxelspace-Demo unterstützt das direkte Reinschreiben in den Grafikspeicher, sowohl mit dem 68K wie auch mit dem PowerPC.

Wenn ein Spiel mit TurboGFX läuft, dann ist es wichtig, dass die Screens nicht umgeschaltet werden, weil sonst Grafik-Fehler auf der Workbench erscheinen können. Bei richtiger Programmierung von TURBOGFX sollte das theoretisch nicht passieren, beim Voxelspace-Demo ist es aber schon vorgekommen.

Folgende Themen werden im folgenden behandelt:

Triple Buffering
Die Adresse des Bildes
Modulo-Probleme
Implementation
Optimierungen
Probleme

1.29 Triple Buffering

Ein Teil-Bild innerhalb eines Spiels wird in der Regel nicht linear von oben links nach unten rechts aufgebaut. Deswegen ist bei dieser Technik unbedingt notwendig, dass mit mehreren Puffern gearbeitet wird. Um mögliche Flackereffekte komplett auszuschalten, benützt man hier Triple-Buffering, d.h. drei Puffer, die nach jeder Iteration rotiert werden.

Leider gibt es keinen direkte Möglichkeit, wie man ein solches Triple-Buffering programmieren kann, z.B. mit einer Library-Funktion. Man muss hier zu einigen Tricks greifen, damit das funktionieren kann:

Beim Oeffnen des Screens wird die Höhe verdreifacht. Jetzt wird dieser überhohe Screen in drei Teile unterteilt und zwar in vertikaler Richtung. Jeder dieser

Teile wird im folgenden als eigenständiger Bildpuffer betrachtet. Zudem kann die Tatsache ausgenützt werden, dass diese Bildpuffer direkt untereinander liegen.

Jetzt wird immer einer der drei Bildpuffer dargestellt und das neue Bild in einem anderen, unsichtbaren Puffer erzeugt. Das Umschalten geschieht, ähnlich wie bei ECS/AGA, mittels graphics/ScollVPort.

1.30 Die Adresse des Bildes

Eine wichtige Frage muss noch beantwortet werden: Wo ist die linke, obere Ecke des Bildes im Grafikspeicher?

Und jetzt folgt das Problem: Ich habe bis heute noch keine 100prozentig sichere Methode gefunden, wie man diesen Zeiger ermitteln kann. Im folgenden werde ich einige Methoden vorstellen und ich möchte die Programmierer dazu ermutigen, möglichst viele der Methoden zu implementieren und dafür zu sorgen, dass der User die Methoden über Schalter auswählen kann.

Es folgen alle Methoden, die ich kenne:

1. Ermitteln des Zeigers auf das Grafik-RAM mittels cybergraphics/
   GetCyberMapAttr (Parameter CYBRMATTR_DISPADR)

Diese Methode kommt im Voxelspace-Demo zum Einsatz, wenn die Option 'MODE2' aktiviert ist. Sie hat bisher nahezu immer funktioniert.

Ich hege allerdings die Vermutung, dass der Zeiger auf den Beginn des Gfx-RAM's und der Zeiger auf den Beginn des Bildes im Gfx-RAM nicht identisch sein müssen. Ich habe auch schon von Verschiebungen des dargestellten Bildes erfahren, welche eventuell darauf zurückgehen.

2. Ermitteln des Zeigers mittels cybergraphics/LockBitmapTagList

Hier wird die angesprochene Funktion mit dem Parameter LBMI_BASEADDRESS aufgerufen.

Diese Methode ist diejenige, die von CyberGFX authorisiert ist und sollte immer als Defaulteinstellung vorhanden sein, wie es auch im Voxeldemo der Fall ist.

Bei gewissen älteren CyberGFX-Versionen war die benötigte Funktion allerdings kaputt. Deswegen sollte man sich nicht 100prozentig auf diese Methode festlegen und sollte mindestens eine andere der Methoden zusätzlich ueber eine Option zur Verfuegung stellen.

3. cybergraphics/DoCDrawMethodTagList

Diese Methode wurde von mir noch nie getestet. Damit sollte der Zeiger auch zu ermitteln sein. Diese Funktion war ebenfalls bei älteren CyberGFX-Versionen völlig kaputt.

4. Man benützt eine Library-Funktion, welche den Zeiger zurückgibt.

Es soll bereits solche Funktionen geben. Wenn eine solche Library gut unterstützt ist und gewartet wird, ist das eine Methode, die unbedingt

zu empfehlen ist. Wenn das auf einmal nicht mehr läuft, braucht dann nur die Library ausgewechselt zu werden.

1.31 Modulo-Probleme

Ein weiteres Problem ist noch zu lösen. Die zweite Zeile eines Bildes liegt nicht zwangsläufigerweise direkt nach der ersten Zeile im Speicher. Ein Bild kann also auch über einen horizontalen Modulo verfügen. Wenn das nicht berücksichtigt wird, wird die Grafik-Darstellung unter Umständen total verzerrt.

Es besteht die Möglichkeit, herauszufinden, wie gross die Distanz zwischen zwei Zeilen in Bytes ist. Dazu wird die Funktion 'cybergraphics/GetCyberMapAttr' mit dem Argument CYBRMATTR_XMOD aufgerufen. Diese Funktion liefert die Breite der verwendeten Bitmap zurück, also die gewünschte Differenz zwischen zwei Zeilen.

Der Algorithmus der eigentlichen Bildberechnung muss also in jedem Fall davon ausgehen, dass ein Modulo-Wert existiert.

1.32 Implementation

Es folgen nun einige Codefragmente, welche die Programmierung des TurboGFX-Modus veranschaulichen.

Als erstes muss beim Oeffnen des Screens die Screenhöhe mit drei multipliziert werden.

Dann folgt der Code, um die Adresse des Bildes sowie den Modulo-Wert zu erfahren. Im folgenden Beispiel werden zwei Methoden gezeigt, wie die Adresse der Bitmap ermittelt werden kann.

Die Variablen haben folgende Funktion:

```
BitmapWidth     : Breite der Bitmap (Distanz zwischen zwei Zeilen)
ActualOffset    : Anzahl Zeilen zwischen Beginn des Bildes und 1. Bildpuffer
HiddenOffset    : Anzahl Zeilen zwischen Beginn des Bildes und 2. Bildpuffer
ThirdOffset     : Anzahl Zeilen zwischen Beginn des Bildes und 3. Bildpuffer
BitMapAddr      : Adresse der Bitmap
ActualBitmap    : Adresse des 1. Bildpuffers
HiddenBitmap    : Adresse des 2. Bildpuffers
ThirdBitmap     : Adresse des 3. Bildpuffers
ChunkyBuffer    : BildPuffer, welcher zum Erstellen der Grafik verwendet wird
AreaHeight      : Höhe eines Bildpuffers
Mode2           : Ist 0, wenn die BitmapAdresse mit 'LockBitMapTagList' ermittelt
                  werden soll, und -1, wenn die BitmapAdresse mit
                  'GetCyberMapAttr' ermittelt werden soll.
```

```
                move.l  _RPort,a0               ;Adresse des RastPort ermitteln
                move.l  rp_BitMap(a0),a0        ;Adresse der Bitmap-Struktur
                tst.b   Mode2                   ;welcher Modus ist eingeschaltet?
                bne.b   .nomode2                ;GetCyberMapAttr -> Sprung
```

```
                lea     CyberLBTLTags,a1        ;Zeiger auf Tagliste für LBTL
                CALLCYBERGFX    LockBitmapTagList ;Bitmap locken
                move.l  LBMI_Addr,d3            ;Bitmap-Adresse ermitteln
                move.l  d3,BitMapAddr
                move.l  d0,a0
                CALLCYBERGFX    UnLockBitmap    ;Bitmap unlocken
                bra.b   .mode2
.nomode2
                move.l  #CYBRMATTR_DISPADR,d0
                CALLCYBERGFX    GetCyberMapAttr ;Startadresse ermitteln
                move.l  d0,d3
.mode2
                move.l  _RPort,a0               ;Adresse des RastPort ermitteln
                move.l  rp_BitMap(a0),a0        ;Adresse der Bitmap-Struktur
                move.l  #CYBRMATTR_XMOD,d0
                CALLCYBERGFX    GetCyberMapAttr ;Bitmapbreite ermitteln
                move    d0,BitmapWidth          ;und speichern
                move    AreaHeight,d1           ;Höhe eines Bildpuffers ermitteln
                clr     ActualOffset            ;vert. Position des 1. Puffers
                move    d1,HiddenOffset         ;vert. Position des 2. Puffers
                move    d1,d2
                add     d1,d1
                move    d1,ThirdOffset          ;vert. Position des 3. Puffers
                mulu    d0,d2
                move.l  d3,ActualBitmap         ;Adresse des 1. Puffers
                add.l   d2,d3
                move.l  d3,HiddenBitmap         ;Adresse des 2. Puffers
                move.l  d3,ChunkyBuffer         ;2. Puffer als erstes verwendet
                add.l   d2,d3
                move.l  d3,ThirdBitmap          ;Adresse des 3. Puffers
```

Die Tagliste, welche der Funktion 'LockBitMapTagList' übergeben wird, kann folgendermassen aussehen:

```
CyberLBTLTags   dc.l    LBMI_BASEADDRESS
                dc.l    LBMI_Addr               ;-> Platz für Bitmap-Adresse
                dc.l    TAG_DONE
LBMI_Addr       dc.l    0                       ;hier steht danach die Adresse
```

Das Voxelspace-Demo geht davon aus, dass zum Zeitpunkt des Aufrufs von 'LockBitMapTagList' der Screen im Vordergrund ist und die Startadresse, welche zurückgegeben wird, in das Grafik-RAM zeigt. Da der Screen erst gerade geöffnet wurde, sollte das immer der Fall sein.

Während der eigentlichen Hauptschleife sollte aber nach jedem 'Locken' der Bitmap die zurückgegebene Adresse mit 'BitMapAddr' verglichen werden. Wenn beispielsweise die Screens umgeschaltet werden, so wird die Bitmap ins FAST-RAM verlegt. Ein Spiel/Demo sollte jetzt üblicherweise in so einem Fall in einen Wartezustand gehen, bis der Screen wieder im Vordergrund ist.

Die Hauptschleife sieht jetzt schematisch etwa wie folgt aus:

1. Bitmap mit 'LockBitmapTagList' locken.

2. Bild berechnen in den Bildpuffer, wo 'ChunkyBuffer' hinzeigt
3. zusätzliche Elemente mit der graphics.library im unsichtbaren Bildpuffer erstellen
4. Bitmap mit 'UnLockBitmap' freigeben
5. Bildpuffer mittels 'graphics/ScrollVPort' rotieren
6. Die nötigen Zeiger und Offsets ebenfalls rotieren

1. Zunächst sollte die Bitmap mittels 'LockBitMapTagList' gelockt werden. Wenn man dies nicht tut, so funktioniert das Spiel/Demo ebenfalls noch, nur ist das nicht mehr legal im Sinne von CyberGFX. Ein Spiel/Demo sollte aber die Möglichkeit, die Bitmap nicht zu locken, als Option zur Verfügung stellen, für den Fall, dass das Locking Probleme machen sollte.

   Im folgenden Beispiel wird auch überprüft, ob die Bitmap-Adresse unterschiedlich ist zu der Adresse, welche kurz nach dem Oeffnen des Screens ermittelt wurde. In diesem Falle wurde der Screen umgeschaltet und das Voxeldemo geht in einen Wartezustand und überprüft periodisch, ob der Screen wieder im Vordergrund ist.

   Hinweis: Das Voxeldemo hängt sich in der nachfolgenden Funktion auf, wenn das Locking aus irgendwelchen Gründen prinzipiell nicht funktioniert. Ein Spiel/Demo sollte hier eine Möglichkeit zur Verfügung stellen, wie das Programm mit einer Fehlermeldung beendet werden sollte.

```
                tst.b   Mode2
                bne.b   .noLBTL
.retry
                move.l  _RPort,a0               ;Adresse des RastPorts ermitteln
                move.l  rp_BitMap(a0),a0        ;Adresse der Bitmap ermitteln
                lea     CyberLBTLTags,a1        ;Zeiger auf Tagliste für LBTL
                CALLCYBERGFX    LockBitmapTagList ;Bitmap locken
                move.l  d0,d2
                beq.b   .delay                  ;wenn Fehler, dann Sprung
                move.l  LBMI_Addr,d0            ;Bitmap-Adresse ermitteln
                move.l  BitMapAddr,d1           ;originale Bitmap-Adresse laden
                cmp.l   d1,d0                   ;Adresse vergleichen
                beq.b   .noLBTL                 ;wenn gleich, dann normal weiter
                move.l  d2,a0
                CALLCYBERGFX    UnLockBitmap    ;Bitmap unlocken
.delay
                moveq   #5,d1
                CALLDOS Delay                   ;warte kurze Zeit
                bra.b   .retry                  ;überprüfe ein weiteres Mal
.noLBTL
```

2. Diese Punkt ist natürlich abhängig vom jeweiligen Spiel. Es ist aber zu beachten, dass der erwähnte Modulo-Wert mitberücksichtigt wird.

3. Mit der TurboGFX-Technik ist es wieder möglich die Standard-Funktion der graphics.library zu verwenden. Es ist aber darauf zu achten, dass die vertikale Position so angepasst wird, dass der richtige Bildpuffer ausgewählt wird. Die vertikale Position 0 ist immer die erste Zeile des ersten Bildpuffers.
   Solch eine vertikale Verschiebung wird am einfachsten durch Addition des Wertes 'HiddenOffset' erreicht.

4. Nachdem alle Zugriffe auf die Bitmap erfolgt sind, muss die Bitmap wieder 'unlocked' werden, wenn sie bei Punkt tatsächlich 'gelockt' wurde. In diesem Beispiel hier wird davon ausgegangen, dass der 'Handle', welcher von 'LockBitmapTagList' zurückgegeben wurde, in d2 steht.

```
                tst.b   Mode2
                bne.b   .noLBTL2
                move.l  d2,a0
                CALLCYBERGFX    UnLockBitmap
.noLBTL2
```

5. Das Umschalten der Bildpuffer geht wie folgt vonstatten:

```
                move.l  _VPort,a0               ;Adresse des ViewPort ermitteln
                move    HiddenOffset,d0         ;vert. Offset des 2. Bildpuffers
                neg     d0                      ;negieren
                move    d0,vp_DyOffset(a0)      ;und in Viewport eintragen
                CALLGRAF        ScrollVPort     ;Bildpuffer umschalten
```

6. Nun müssen noch die nötigen Zeiger und Offsets rotiert werden, damit das nächste Bild wieder im richtigen Puffer erzeugt wird:

```
                move.l  ActualBitmap,d0
                move.l  HiddenBitmap,ActualBitmap
                move.l  ThirdBitmap,HiddenBitmap
                move.l  d0,ThirdBitmap
                move.l  HiddenBitmap,ChunkyBuffer
                move    ActualOffset,d0
                move    HiddenOffset,ActualOffset
                move    ThirdOffset,HiddenOffset
                move    d0,ThirdOffset
```

1.33 Optimierungen

Wenn die TurboGFX-Technik verwendet wird, müssen noch einige Dinge beachtet werden, damit die optimale Performance erzielt werden kann. Der Zugriff auf das RAM der Grafikkarte geschieht in der Regel über den Zorro3-Bus (früher Zorro2). Genau dieser Zugriff vom Prozessor über den Z3-Bus zur Grafikkarte ist wesentlich langsamer als der Zugriff des Prozessors auf das Fast-RAM, welches oftmals auf dem Prozessor-Board selbst zu finden ist.

Dazu kommt noch, dass das Fast-RAM in der Regel im Copyback-Modus angesprochen wird, während das RAM auf der Grafikkarte immer 'noncachable', also mit ausgeschaltetem Cache angesprochen wird.

Daraus folgt jetzt direkt, dass es eine sehr grosse Rolle spielt, ob die Grafik byteweise oder langwortweise im Grafik-RAM erzeugt wird. In ersterem Fall geht die Performance unter Umständen stark zurück.

Als anschauliches Beispiel sei hier der Voxelspace-Algorithmus erwähnt. Dieser

Algorithmus projiziert die Landschaftsdaten streifenweise auf den Bildschirm. Der Code schreibt also vertikale Spalten von links nach rechts direkt im Grafik-RAM. Sind die Spalten 4 Pixel breit, so ergibt das Langwort-Zugriffe, welche optimal sind. Bei Spalten von 2 oder nur 1 Pixel sind diese Zugriffe überhaupt nicht mehr optimal.

Der Zugriff kann jetzt aber optimiert werden. Im Voxelspace-Demo werden, im Falle von 1 Pixel breiten Streifen, Gruppen von 4 Streifen gebildet und diese in einem Fast-RAM-Puffer erzeugt. Danach wird dieser Puffer langwortweise in das Grafik-RAM kopiert und danach die nächsten 4 Streifen erzeugt. Da der Puffer relativ klein ist, profitieren diese Zugriffe noch vom Datencache des Prozessors. Die Byte-Zugriffe auf das Fast-RAM werden vom Copyback-Modus des Cache abgefedert.

Es ist bei der TurboGFX-Technik also sehr darauf zu achten, dass der Zugriff auf das Grafik-RAM möglichst langwortweise geschieht. Solche 'Umwege' über das Fast-RAM können oftmals eine erhebliche Verbesserung der Performance nach sich ziehen.

1.34 Probleme

Es sei hier noch auf bekannte Probleme mit CyberGfx+/TurboGFX hingewiesen.

Es kann vorkommen, dass bei Betätigung der Maustasten Aufhänger oder Abstürze stattfinden. In diesem Falle empfiehlt es sich, alle im System laufenden Commodity-Programme zu deaktivieren (wenn bekannt ist, welches Programm das Problem verursacht, kann auch nur dieses deaktiviert werden). Als Beispiel seien hier die verschiedenen Blanker-Programme erwähnt.

1.35 Interaktion

Jedes Spiel lebt von der Interaktion mit dem Spieler. Der Spieler benützt Tastatur, Maus oder Joystick, um das Spiel in die gewünschte Richtung zu lenken.

Es soll im folgenden darauf eingegangen werden, wie man systemkonform die Eingaben des Spielers auswerten kann. Auf die Joystick-Programmierung wird dabei nicht eingegangen, da ich das selbst noch nie in der Praxis angewendet habe. Es sei auf die Dokumentation des 'gameport.device' verwiesen, welche dabei verwendet wird.

Die Eingaben des Benützers finden in der Regel im aktivierten Fenster statt. Das Betriebssystem registriert diese Eingaben und sendet dem Programm Meldungen, welche es ihm erlauben, die Art der Eingabe auszuwerten. Das Spiel muss sich beim Oeffnen des Fensters darauf festlegen, welche Eingaben ausgewertet werden sollen, indem es in der Window-Tagliste die entsprechenden IDCMP-Flags angibt.

Die Hauptschleife eines Spieles sieht dann etwa folgendermassen aus:

1. Window-Message abholen
2. Window-Message auswerten und wenn nötig, Aktionen

festlegen
3. Window-Message beantworten
4. Aktionen ausführen
5. Bild berechnen und darstellen

1. Window-Message abholen

Eine Message wird mit der Systemfunktion 'exec/GetMsg' gelesen. Sie erwartet als Parameter einen Message-Port. Es wird der UserPort des Fensters übergeben. Das sieht dann folgendermassen aus:

```
move.l  _Window,a0              ;Window-Adresse ermitteln
move.l  wd_UserPort(a0),a0      ;Adresse des UserPort ermitteln
CALLEXEC        GetMsg          ;Message abholen
tst.l   d0                      ;Message vorhanden?
beq.w   .loop                   ;nein -> keine Auswertung
move.l  d0,d4                   ;Message speichern für ReplyMsg
move.l  d0,a0                   ;Message nach a0 für Auswertung
```

2. Window-Message auswerten

Eine Window-Message hat eine definierte Struktur (zu finden in der Include-Datei 'intuition/intuition.i' unter der Struktur 'IntuiMessage'). Einige Elemente dieser Struktur können nun für die Auswertung verwertet werden. Zunächst wird das Feld 'im_Class' ausgewertet, welches die Art der Eingabe klassifiziert.

Nach dem Lesen der Message werden also zunächst einmal alle interessanten Elemente der Message ausgelesen:

```
move.l  im_Class(a0),d0         ;Message-Klasse holen
move    im_Code(a0),d1          ;Message-Unterklasse holen
move    im_MouseX(a0),d2        ;Maus-Delta-Postionen holen
move    im_MouseY(a0),d3        ;und für Y-Richtung
```

Jetzt wird die Message-Klasse überprüft (die möglichen Werte entsprechen den IDCMP-Flags, welche beim Oeffnen der Fensters angegeben wurden). Beispiel:

```
cmp.l   #IDCMP_MOUSEBUTTONS,d0  ;Maustasten gedrückt?
beq.b   .checkmouse
cmp.l   #IDCMP_RAWKEY,d0        ;Taste gedrückt?
beq.w   .checkrawkey
cmp.l   #IDCMP_MOUSEMOVE,d0     ;Maus bewegt?
beq.w   .checkdeltamove
cmp.l   #IDCMP_ACTIVEWINDOW,d0  ;Fenster aktiviert?
beq.b   .activewindow
cmp.l   #IDCMP_INACTIVEWINDOW,d0 ;Fenster inaktiviert?
beq.b   .inactivewindow
bra.w   .reply                  ;sonst beanworte Message
```

Abhängig von der Message-Klasse werden jetzt noch weitere Informationen extrahiert. Dies geschieht meistens mit dem Feld im_Code, welches jetzt im Beispiel in d1 liegt:

IDCMP-Flag MOUSEBUTTONS:

```
                cmp     #IECODE_LBUTTON,d1      ;linke Maustaste gedrückt?
                beq.b   .leftdown
                cmp     #IECODE_LBUTTON+IECODE_UP_PREFIX,d1 ;losgelassen?
                beq.b   .leftup
                cmp     #IECODE_RBUTTON,d1      ;rechte Maustaste gedrückt?
                beq.b   .rightdown
                cmp     #IECODE_RBUTTON+IECODE_UP_PREFIX,d1 ;losgelassen?
                beq.b   .rightup
                bra.w   .reply
```

IDCMP-Flag RAWKEY:

Das Feld im_Code enthält den Tasten-Code der gedrückten Taste. Dieser Code entspricht NICHT dem ASCII-Code (dazu müsste das IDCMP-Flag IDCMP_VANILLAKEY verwendet werden). Die Codes müssen aus einer Tabelle entnommen werden oder durch Ausprobieren ermittelt werden.

Bit 7 des Keycodes gibt an, ob die Taste gedrückt oder losgelassen wurde. Das kann mit dem Flag IECODEF_UP_PREFIX getestet werden.

Beispiel:

```
                cmp     #$45,d1                 ;ESC gedrückt?
                beq.w   .esc
                btst    #IECODEB_UP_PREFIX,d1
                bne.w   .keyup
                cmp.b   #$50,d1                 ;F1 gedrückt?
                beq.w   .F1pressed
                cmp.b   #$51,d1                 ;F2 gedrückt?
                beq.w   .F2pressed
                ...
                bra.w   .reply
.keyup
                bclr    #IECODEB_UP_PREFIX,d1
                cmp.b   #$55,d1                 ;F6 losgelassen?
                beq.w   .F6released
                cmp.b   #$56,d1                 ;F7 losgelassen?
                beq.w   .F7released
                ...
                bra.w   .reply
```

IDCMP-Flag MOUSEMOVE:

Dieses IDCMP-Flag sollte immer zusammen mit IDCMP_DELTAMOVE verwendet werden. Die Felder 'im_MouseX' und 'im_MouseY' enthalten jetzt den Wert, um wie viele Einheiten die Maus bewegt worden ist. Die Grössenordnung dieser Werte muss durch Ausprobieren ermittelt werden. Oftmals werden diese Werte für die weitere Bearbeitung im Spiel noch skaliert.

IDCMP_Flags ACTIVEWINDOW und INACTIVEWINDOW:

Hier können keine zusätzlichen Informationen mehr gewonnen werden.

3. Window-Message beantworten

Nach der Auswertung der Message muss sie beantwortet werden. Dies geschieht mit der exec-Funktion 'ReplyMsg':

```
move.l  d4,a1                   ;Message nach a1
CALLEXEC        ReplyMsg        ;Message beantworten
```

Die Punkte 4 und 5 sind programmspezifisch und gehören im Prinzip nicht mehr zur Interaktion dazu.

1.36 RAM is slow

In den letzten Jahren ist die Prozessorleistung nahezu exponentiell gestiegen. Im Gegensatz dazu hat sich die Zugriffszeit der konventionellen RAM-Bausteine, wie sie in herkömmlichen Computern eingesetzt ist, nur sehr schwach verbessert. Der Zugriff auf das RAM hat sich immer mehr zu einem Flaschenhals entwickelt.

Dieser Flaschenhals wird mit den immer grösser werdenden Cache-Speichern bekämpft, mit Erfolg. Die Caches sind ein wichtiges Leistungsmerkmal bei konventionellen Applikationen geworden.

Spiele sind aber keine konventionelle Applikationen. Spiele sind sehr speicherlastig, da sie mit immer grösser werdenen Datenmengen umgehen muss. Oftmals wirken sich die Caches in gewissen Bereichen kontraproduktiv aus. Deswegen muss bei der Spieleentwicklung darauf geachtet werden, dass Speicherzugriffe minimiert werden.

In diesem Zusammenhang müssen auch Programmierphilosophien, wie sie von den alten Prozessoren her bekannt sind, völlig umgekrempelt werden. Bei der Programmierung eines 68000 optimierte man Spiele, indem man möglichst viele Berechnungen im Voraus machte und in Tabellen ablegte, welche dann im Echtzeit-Teil ausgelesen wurden.

Da viele Spiele mit dieser Philosophie entwickelt wurden, führte dies dazu, dass diese Spiele auf stärkeren Prozessoren nicht die Leistungssteigerung erbrachten, welche eigentlich zu erwarten war. Denn selbst der allerschnellste Prozessor greift auf das RAM etwa gleich schnell zu wie ein konventioneller 68000.

Moderne Prozessoren können in der Zeit, die ein Speicherzugriff etwa braucht, 50-100 oder sogar noch mehr Rechenbefehle ausführen. Sehr oft können also Daten schneller in Echtzeit berechnet als aus einer Tabelle ausgelesen werden. Dazu kommt noch, dass grosse Tabellen sehr Cache-unfreundlich sind. Sie führen dazu, dass die Effizienz des Caches sehr stark nachlässt und damit die Performance stark gebremst wird.

Im Voxelspace-Demo wurde diese neue Philosophie in die Tat umgesetzt. Viele Voxelspace-Algorithmen, welche in der Fachpresse beschrieben wurden, waren für alte Prozessoren optimiert, indem viele Strukturen im Voraus angelegt wurden. Im Voxelspace-Demo wurden jetzt nahezu all diese Berechnungen in Echzeit durchgeführt. Die Hauptschleife enthält dann nur noch einen einzigen Speicherzugriff: das Auslesen der Höhen/Farbdaten der Landschaft. Natürlich kommen dann noch die Zugriffe zum Erstellen der Grafik hinzu. Solche Algorithmen, welche fast nur noch über Rechenbefehle verfügen, fördern auch

das Pipelining und somit auch den Durchsatz der Befehle.

Es ist für die optimale Programmierung von PPC-Spielen entscheidend, dass die Algorithmen in erster Linie auf ein Minimum von Speicherzugriffen optimiert werden, damit die volle Leistungsfähigkeit dieser Prozessoren genutzt werden kann.

1.37 MMU und Cache

In diesem Abschnitt geht es darum, aufzuzeigen, wie man durch geeignete Ausnützung von Cache und MMU grosse Performanceverbesserungen erzielen kann.

Die Caches sind dann am effizientesten, wenn sehr oft auf die gleichen Speicherbereiche zugegriffen wird. Dann lassen sich zeitraubende Speicherzugriffe vermeiden. Das beste Beispiel ist hier der Prozessor-Stack.

In Spielprogrammen ist es oft so, dass auf riesige Datenmengen zugegriffen wird, dass aber höchst selten die gleiche Stelle mehrmals angesprochen wird. In solchen Fällen führt dies dazu, dass die Caches kontraproduktiv werden und das Spiel zusätzlich verlangsamen. Caches werden intern in Stücken zu 32 Bytes verwaltet. Sobald auf ein Datenelement zugegriffen wird, wird eine solche Cacheline als Ganzes in den Cache geladen. Dieser eine Speicherzugriff dauert also solange, wie 8 konventionelle Speicherzugriffe (bei Prozessoren mit 32-Bit-Bus). Dieser zusätzliche Aufwand wird in der Regel dadurch kompensiert, dass nachfolgende Zugriffe auf diese Cacheline nur noch auf dem Cache stattfinden.

Bei Spielen, welche solche Datenelemente nur einmal benützen, führt das dazu, dass die Speicherzugriffe langsamer werden, als wenn der Datencache ausgeschaltet wird.

Jetzt folgt das Problem: Wenn man nun den Datencache global ausschaltet, so hat man zwar dieses Teilproblem beseitigt, nun verliert man aber wieder Performance dadurch, dass die Bereiche, welche vom Cache profitierten, verlangsamt werden. Die technisch einzige Lösung ist hier die Erstellung eines optimierten MMU-Setups, welche den einzelnen Speicherbereichen einen separaten Cache-Modus zuteilt. Das AMIGA-OS bietet leider absolut keinen MMU-Support an. Die einzige Möglichkeit ist hier, 'böse Hacks' anzuwenden, also die Hardware direkt anzusprechen, was ja eigentlich nicht mehr getan werden sollte.

Mit dem Einsatz des PowerPC und des Betriebssystem WarpOS hat sich das jetzt grundlegend verändert. WarpOS bietet Applikationen die Möglichkeit an, Speicherbereiche mit gewünschten Cache-Modi zu allozieren. Damit steht also ein systemkonformer Weg zur Verfügung, um die MMU und den Cache möglichst optimal auszunützen.

Die Funktion 'AllocVecPPC' der powerpc.library unterstützt zusätzliche Speicherattribute, womit der Cache-Modus angegeben werden kann. Spiele können jetzt also einen Speicherbereich allozieren und diesen als 'noncachable' markieren (mit dem Attribut MEMF_NOCACHE). Für eine detaillierte Beschreibung dieser Funktion seien die Dokumente 'WarpOS.guide' und 'powerpc.doc' empfohlen.

Allerdings sei hier noch folgender Hinweis angegeben: Die lokale Abschaltung des Datencache kann je nach Prozessor ein Programm beschleunigen oder auch

bremsen. Je grösser der Cache und je schneller der Speicherzugriff, desto eher kann das der Fall sein, dass das Programm dann gebremst wird. Es sollte also immer auf verschiedenen Systemen getestet werden und eventuell verschiedene Versionen unterstützt werden. Das Voxelspace-Demo setzt die lokale Abschaltung des Datencaches beim 603E ein, nicht aber beim 604E (bei der CyberstormPPC), weil der 604E tatsächlich gebremst wurde.

Nun sei noch auf ein weiteres Problem hingewiesen. Die MMU hat die Aufgabe, Adressübersetzungen durchzuführen. Sie schaut in einer Tabelle nach, welche logische Adresse zu welche physikalischen Adresse passt und führt die Uebersetzung durch. Bei allen 68K-Prozessoren mit MMU wurde das automatisch per Hardware durchgeführt. Diesen Vorgang nennt man den Tablesearch oder Tablewalk.

Von den PowerPC-Prozessoren unterstützen nicht alle den hardwaremässigen Tablesearch. Der PPC603 und der PPC603E kennen den nicht. Der Tablesearch wird auf diesen Prozessoren per Software erledigt. Solch ein Software-Tablesearch ist natürlich extrem langsam. In der Regel spielt das aber keine Rolle, da ein 'MMU-Cache' dafür sorgt, dass diese Tablesearches selten genug auftreten, so dass keine Performance-Probleme auftauchen.

Bei Spielen kann das aber ganz anders werden. Beispiel Voxelspace: Dem Voxel-space-Algorithmus liegt eine Landkarte zugrunde, welche 2MByte gross ist. Der Algorithmus liest jetzt nach einem bestimmten Verfahren die Daten der Landkarte aus. Dabei bewegt er sich durch einen äusserst grossen Adressraum. Dies hat jetzt zur Folge, dass, ähnlich wie beim Cache, die Effizienz des MMU-Cache gegen Null geht. Somit wird also für die meisten Speicherzugriffe ein Table-search durchgeführt, was ein Spiel de facto zum Zusammenbruch führen kann.

Dieser Effekt kann sogar bei Systemen mit hardwareunterstütztem Tablesearch beobachtet werden, nur ist die Auswirkung dort nicht allzu gross.

Auch hier ist eine Lösung bei 68K-Systemen nicht systemkonform möglich. Das Voxelspace-Demo bietet hier für 68K-Systeme eine zusätzliche Option an, welche als 'Hack' zu betrachten ist und somit komplett nicht-systemkonform ist. Dieser Hack setzt die MMU mit Hilfe der Transparent Translation Register neu auf und verhindert auf diese Weise, dass Tablesearches auftreten können. Dies kann Performance-Steigerungen von bis zu 40-50 Prozent bringen.

Beim PPC unter WarpOS gibt es eine systemkonforme Möglichkeit, dieses Problem zu lösen. WarpOS unterstützt die sogenanten BAT-Register, welche ähnlich wie die Transparent Translation Register funktionieren. Jeder PPC-Task hat unter WarpOS die Möglichkeit, die BAT-Register mit einem beliebigen Speicherbereich zu belegen. Es stehen 4 BAT-Register zur Verfügung, wobei einer davon oftmals vom System belegt ist, um das Grafik-RAM abzudecken. Durch den Einsatz der BAT-Register können Tablesearches für den angegebenen Speicherbereich verhindert werden.

Die Funktion 'AllocVecPPC' der powerpc.library kennt das Speicherattribut MEMF_BAT. Wird dieses angegebenen, so wird der allozierte Speicherbereich von einem BAT-Register kontrolliert.

Das Voxelspace-Demo macht Gebrauch von diesen Zusatz-Features, wenn die Option TURBOPPC eingeschaltet ist. Es alloziert sowohl den Speicher für die Landkarte als auch den Speicher für den Himmel mit den Attributen MEMF_NOCACHE und MEMF_BAT.

Um die Mächtigkeit dieser Features zu demonstrieren, sei auf folgendes hingewiesen:

Das Voxelspace-Demo läuft auf einem PPC603E/150 bei gewöhnlichen Einstellungen etwa doppelt so schnell wie auf einem 68060/50, wenn beide Prozessoren möglichst optimal angesteuert werden. In diesem Fall läuft der 68060 nicht-systemkonform.

Wenn der 'MMU-Hack' für den 68060 ausgeschaltet ist, läuft der PowerPC bis zu dreimal so schnell. Das heisst, systemkonform programmierte Spiele können aus dem PowerPC noch mehr an Leistung herausholen.

1.38 Multiprozessing

Um es vorweg zu nehmen: Dieser Abschnitt behandelt nicht, wie Multiprozessing zur Performanceverbesserung verwendet werden kann, sondern warum Multiprozessing nicht zur Performanceverbesserung verwendet werden kann.

Ein Dual-Prozessor-Board weckt natürlich die Hoffnung, dass man beide Prozessoren parallel laufen lassen kann, um eine höhere Leistung zu erzielen. Das Problem: Beide Prozessoren benützen denselben Bus. Bei jedem Speicherzugriff blockiert der eine Prozessor den Bus für den andern. Algorithmen, welche also viele Speicherzugriffe beinhalten, führen dazu dass die Gesamtperformance beider Prozessoren nach unten geht.

Angenommen, beide Prozessoren führen Algorithmen aus, welche nicht auf den Speicher angewiesen sind, dann könnte tatsächlich eine Leistungssteigerung erzielt werden. Aber gerade in diesem Fall ist der Leistungsunterschied zwischen den beiden Prozessoren so gross, dass es sich gar nicht mehr lohnt, beide Prozessoren parallel laufen zu lassen. Zudem sind rein rechenintensive Algorithmen äusserst selten. Der Mandelbrot-Algorithmus ist so ein Beispiel. Und selbst beim 'cybermand' führte Parallel-Processing nicht zu einer sichtbaren Leistungssteigerung.

Beim Design der Spiele ist also darauf zu achten, dass sie sequentiell auf beiden Prozessoren laufen. Es sollten auch nicht mehrere Tasks verwendet werden, welche wiederum ähnliche Probleme auslösen können.

1.39 Scheduling / Optimierungen

Scheduling ist die Kunst, Befehle so anzuordnen, dass sie die internen Ausführungseinheiten eines Prozessors optimal ausnützen. Scheduling erhält bei modernen Prozessoren einen erhöhten Stellenwert. Allerdings sollte das Scheduling nicht überbewertet werden. Einzig systematische Anwendung der allgemeinen Scheduling-Regeln (wie beispielsweise in den PPC User-Manuals beschrieben) kann eine Performance-Steigerung bringen. Und das ist in der Regel Sache von Hochsprachen-Compilern.

Es sei hier trotzdem noch auf Möglichkeiten hingewiesen, wie man Befehle optimal plazieren kann. Im Voxelspace-Demo ist ein solcher Fall vorhanden. In der innersten Hauptschleife kommen sowohl eine Fliesskomma-Division wie auch ein Speicherzugriff vor. Beide Befehle haben eine äusserst grosse

Ausführungszeit, sind also sehr langsam. Jetzt kann die Performance so optimiert werden, indem man den Fliesskomma-Befehl direkt vor den Speicher-Zugriff plaziert (also FP-Befehl, gefolgt von Speicherzugriff). Auf diese Weise kann der PPC beide Befehle parallel ausführen und so kann einiges an Zeit gespart werden.

Die effizienteste Scheduling-Massnahme, welche auch am besten eingesetzt werden kann, ist das Vermeiden von Abhängigkeiten. Der PowerPC arbeitet erst dann am Effizientesten, wenn seine Pipelines möglichst gut gefüllt sind. Wenn jetzt aber direkt aufeinanderfolgende Befehle das Resultat ihres Vorgängers benötigen, muss solch ein Befehl warten, bis das Resultat zur Verfügung steht. Durch kluges Plazieren der Befehle kann hier zum Teil sehr viel Performance gewonnen werden.

Grundsätzlich ist es ratsam, Algorithmen auf mehrere Ausführungseinheiten zu plazieren. Wenn es möglich ist, sollten Integer- und FPU-Befehle abwechslungsweise eingesetzt werden, damit sie parallel ausgeführt werden können. Dies macht dann Sinn, wenn der Algorithmus eine gewisse Grösse erreicht. Bei kleinen Algorithmen kommt das Problem zum Zug, dass durch die Fliesskomma<->Integer-Konvertierung die Zeit, die gewonnen wird, wieder verloren geht.

Wenn man in Assembler programmiert sollte man also immer versuchen, Integer- Fliesskomma- und Speicherzugriffs-Befehle abwechslunsgweise zu verwenden.

Wenn sich dem Programmierer die Wahl stellt, ob ein Algorithmus bzw. ein ganzes Programm vor allem auf Integer- oder FPU-Operationen basieren soll, dann wird in der Regel die FPU-Variante die Optimalere sein. Die FPU des PowerPC ist extrem leistungsfähig und bietet sehr starke Befehle an, z.B. die kombinierten Multiply-Add/Sub-Befehle.

Demgegenüber muss natürlich die Tatsache gestellt werden, dass der PPC604[E] mehrere Integer-Ausführungseinheiten kennt. Wenn ein Integer-Algorithmus diese Einheiten optimal ausnützt, kann das wiederum die bessere Performance abgeben.

Dann folgen hier noch ein paar weitere Tips zum Optimieren. Dank der vielen Register des PPC können Speicherzugriffe oft vermieden werden. Vor der eigentlichen Funktion können also Konstanten wie auch Variablen in Register geladen werden und in der Hauptschleife mit den Registern gerechnet werden. Das erfordert natürlich eine saubere Dokumentation, damit die Zuordnung der Register noch durchschaubar ist.

1.40 Konfigurierbarkeit

Jetzt verlassen wir den technischen Bereich und wenden uns dem Game-Design zu. Zunächst mal soll die Konfigurierbarkeit von Spielen diskutiert werden.

Spiele gewinnen oft an Reiz, wenn sie viele Optionen und Schalter haben, welche es erlauben, das Spiel aus einer anderen Perspektive zu spielen oder mit anderen Rahmenbedingungen. Dazu gehört beispielsweise die Auflösung, welche von Spieler frei gewählt werden sollte (wenn eine Grafikkarte vorhanden ist).

Die Steuerung, sollte ebenfalls möglichst flexibel einsetzbar sein. Sowohl

verschieden Steuerungsarten sollten unterstützt sein wie auch Parameter, welche für Geschwindigkeit, Trägheit usw. zuständig sind.

Die Tastaturbelegung sollte auch möglichst frei definierbar sein, da jeder Spieler hier andere Vorlieben zeigt. Zuviele Spiele sind schon verrissen worden, nur weil die Tastenbelegung völlig unmöglich zusammengestellt worden war.

Oftmals sind technische Parameter gern benützte Spielzeuge des Spielers. Auflösung, Detailgenauigkeit, Bildgrösse, Approximationen, Landschafts-Parameter, Schwierigkeitsgrade, Darstellung der Framerate, um nur einige wenige zu nennen. Je mehr solcher Spielereien vorhanden sind, desto interessanter wird es, das Spiel mehrere Male zu spielen.

Generell sollte möglichst viel Fantasie eingesetzt werden, um dem Spieler die Möglichkeit zu geben, das Spiel auf viele verschiedene Arten zu spielen.

1.41 Steuerung

An der Steuerung sind schon viel zu viele Spiele mit guten Ansätzen gescheitert. In den meisten Fällen war die Steuerung fest im Spiel verankert, ohne Einstellungsmöglichkeit durch den Spieler.

Prinzipiell sollte ein Spiel möglichst viele Steuerungs-Arten unterstützen, z.B. Joystick, Maus und Tastatur. Jeder Spieler hat seine Vorliebe für eine dieser Steuerungsarten.

Die Steuerungs-Geschwindigkeit spielt ebenfalls eine zentrale Rolle. Zunächst einmal ein paar negative Beispiele von bekannten Spielen:

Ein bekanntes Auto-Rennspiel verwendete ein System mit Zeitlimits, d.h. eine bestimmte Strecke musste in einer bestimmten Zeit absolviert werden. Die Zeit verstrich absolut, also wie eine echte Uhr. Wenn jetzt das Spiel auf einem langsameren Rechner gespielt wurde, so vermochte die Hardware weniger Bilder pro Zeit darzustellen. Konsequenz: Der Wagen kommt in der gleichen Zeit mit der gleichen Fahrgeschwindigkeit weniger weit. Das ist dann natürlich äusserst schlecht, da es die Benützer langsamerer Maschinen benachteiligt und da es sowieso unfair ist.

Ein bekanntes 3D-Ballerspiel verwendet einen ausgeklügelten Mechanismus zum Rotieren des Spielers, wenn er die Cursortasten links und rechts betätigt. Die effektive Rotationsgeschwindigkeit soll aber konstant sein, d.h. der Spieler soll sich, unabhängig von der Spielgeschwindigkeit, immer um den gleichen Winkel in derselbe Zeit rotieren können. Konsequenz: Um auf langsamen Maschinen denselben Winkel in derselben Anzahl Teilbildern zurückzulegen, muss der Winkel dramatisch vergrössert werden. Die Folge davon war, dass nur ein kurzer Druck auf die Cursortaste genügte, um den Spieler um eine halbe Drehung zu rotieren. Damit ist eine präzise Steuerung praktisch verunmöglicht und das Spiel wurde auf den meisten Systemen nicht mehr vernünftig spielbar (ausser wenn die Spielgeschwindigkeit durch Herabsetzen der Bildgrösse/Auflösung künstlich beschleunigt wurde).

Diese Beispiele zeigen, wie wichtig es ist, dass die Steuerung der jeweiligen Hardware angepasst wird. Gute Beispiele existieren ebenfalls schon, gewisse

Spiele erlauben es, Parameter wie Geh- und Rotations-Geschwindigkeit und sogar die Trägheit bei den Bewegungen in Stufen einzustellen.

Wichtig ist auch, dass die Steuerung möglichst präzis ist. Viele Spiele sind schon deswegen unspielbar geworden, weil gewisse Aufgaben im Spiel durch unpräzise Steuerung nur durch Glück zu meistern waren.

Ein Spiel sollte auch die eingeschränkte Bewegungsgenauigkeit des Spielers berücksichtigen. Wenn z.B. Kollisionen erkannt werden sollten, sollten solche Abfragen nicht haarklein schon reagieren, wenn nur ein Pixel des einen Objektes mit einem Pixel des anderen Objekts in Kontakt kommt. Hier ist eine grosszügigere Abfrage vonnöten, damit nicht Kollisionen stattfinden, welche gar keine sind (wenn zwei Autos sich streifen, ist das etwas anderes, als wenn sie frontal zusammenstossen).

1.42 Schwierigkeitsgrade

Man kann die Spieler in mehrere Kategorien einteilen: solche, die hin und wieder ein Spiel spielen, andere, welche mehr oder weniger regelmässig spielen und die Profis. Ein Spiel sollte jetzt möglichst alle diese potentiellen Kunden ansprechen und dazu braucht es verschiedene Schwierigkeitsgrade.

Sehr viele Spiele kennen Schwierigkeitsgrade - und die allerwenigsten davon setzen dieses Mittel vernünftig ein. Wenn ein bekanntes Action-Spiel vier Schwierigkeitsgrade anbietet und den härtesten mit 'Maniac' anschreibt, dann darf es einfach nicht sein, dass ein professioneller Actionspieler mit 'Maniac' im ersten Ansatz bis zur Hälfte des Spiels kommt und danach selber abbrechen muss, damit er beim nächsten Mal überhaupt noch was Neues sieht.

Genauso übergangen werden oft absolute Neulinge. Der einfachste Schwierigkeitsgrad ist oft immer noch zu schwer für diejenigen, die sich mit solchen Spielen praktisch nicht auskennen. Regel:

Der Programmierer sollte den einfachsten Schwierigkeitsgrad so wählen, dass er ihn für kindisch einfach hält und den Schwierigkeitsgrad anschliessend um die Hälfte reduzieren.

Für den schwersten Schwierigkeitsgrad sollte analog vorgegangen werden. Je höher die Unterschiede, desto besser. Dann erhalten selbst Top-Profis noch eine zusätzliche Herausforderung.

Oft werden 'einfach' und 'schwierig' völlig falsch interpretiert. Ein Spiel wird nur allzuoft schwieriger gemacht, indem einfach die Portion Glück, die benötigt wird, nach oben geschraubt wird. Oft wird auch einfach und billig die Stärke der Spielfigur reduziert und das dann als separater Schwierigkeitsgrad verkauft.

Ein einfacher Schwierigkeitsgrad sollte so implementiert werden, dass Fehler des Spielers weniger stark ins Gewicht fallen. Es macht keinen Sinn, die Anzahl der Gegner extrem stark zu reduzieren, wenn der Spieler aus lauter Ungeschick dauernd die Umgebung rammt und dabei jedesmal ein ganzes Leben verliert. Hier wäre es angesagt, solche Kollisionen weniger stark zu gewichten.

Schwierigkeitsgrade sollten nicht nur Spielparameter ändern. Es sollten auch neue Elemente im Spiel eingebracht werden, damit der fortgeschrittene Spieler

auch wirklich nachdenken muss und seine bisherige Taktik ändern muss. Hier kann durch Einsatz von viel Kreativität die langfristige Attraktivität eines Spiels enorm ansteigen.

Es ist auch dringend zu überprüfen, ob Besitzer von unterschiedlich schnellen Systemen bevorteilt oder benachteilt werden. Dies sollte nie der Fall sein und sollte durch geeignete Massnahmen kompensiert werden.

1.43 Die nötige Abwechslung

Immer öfter kommt es vor, dass in der Werbung oder auf der Verpackung zu Spielen geworben wird mit Sätzen wie: über 33 Levels, über 100 Aufgaben, usw.

Es ist eigentlich sehr zu begrüssen, wenn ein Spiel gross genug ist, um den Spieler lange zu unterhalten. Leider ist es aber oftmals so, dass die Spiele-Hersteller in den ersten 10 Prozent des ganzen Spiels ihr gesamtes Pulver verschiessen. Während dieser Zeit ist der Spieler mit Freuden am Spielen und danach flacht das völlig ab, weil einfach nichts Neues mehr kommt, keine Abwechslung mehr da ist.

Wichtig ist, dass man neue Elemente auf das ganze Spiel regelmässig verteilt. Man darf nicht zu Beginn des Spieles schon die meisten Elemente vorstellen und noch oft den Spieler damit überfordern. Besser ist es, wenn der Spieler Schritt für Schritt mit neuen Aspekten des Spiels konfrontiert wird und dass er genügend Zeit bekommt, um sich auf dieses neue Feature einzustellen.

Ein Spiel sollte auch die Motivation des Spielers aufrechterhalten, das Spiel noch weiterzuspielen. Es sollten in regelmässigen Abständen spezielle Ereignisse stattfinden, z.B. Obergegner, welche den Spieler richtig beeindrucken sollten, oder Bonussequenzen, möglichst immer wieder Verschiedene. Es ist oft auch interessant, eine richtige Hürde einzubauen, um den Spieler zu fordern, was sehr oft mit Obergegnern realisiert wird. Das Besiegen eines starken Obergegners ist ein sehr eindrückliches Erlebnis und stellt eine sehr hohe Motivationskraft dar, da man oftmals viele Anläuft braucht, bis er endlich geschafft ist. Es erfordert auch vom Spieler, dass der Rest des Spieles mit äusserster Konzentration in Angriff genommen wird, damit genug Resourcen für den Obergegner noch zur Verfügung stehen.

Die allermeisten Obergegner, die ich bisher gesehen habe, sind äusserst einfallslos und durchschaubar. Meistens sind sie durch eine äusserst primitive Taktik ausser Gefecht zu setzen. Es wäre am besten, wenn neben innovativen Angriffsstrategien, welche nach dem dritten, vierten Mal durchschaut sind, auch Geschicklichkeits-Elemente drin sind, welche den Spieler dazu veranlassen, jedes mal mit derselben Konzentration ans Werk zu gehen. Es ist sowieso für das ganze Game-Design zu empfehlen, reichlich von Geschicklichkeits-Sequenzen Gebrauch zu machen, da Geschick etwas ist, was man nicht eines Tages 'auswendig' kann. Die meisten Spiele verflachen mit der Zeit, weil man schon 'alles gesehen' hat.

1.44 Spielbarkeit / Fairness

Spielbarkeit ist wohl das, was vielen Spielen besonders fehlt. Ein Spiel ist dann spielbar, wenn es auch tatsächlich Spass macht. Deswegen die Spielbarkeit einen hohen Stellenwert in der Spieleentwicklung erhalten.

Eines der klassischen Beispiele von unspielbaren Elementen sind die berühmten Angriffe von hinten. Oftmals werden solche Taktiken angewandt, um den Schwierigkeitsgrad künstlich zu erhöhen. Gerade solche Attacken haben aber langfristig keinen Effekt, ausser dass sie ein Spiel abflachen, weil der Spieler nach dem dritten Mal auswendig weiss, wo unfaire Attacken zu erwarten sind.

Viele Spiele protzen auch mit der Geschwindigkeit, mit der sich die Objekte bewegen. Nur allzuoft wird hier sehr kurzfristig ein toller Effekt erzielt, welcher dann aber dazu führt, dass nach dem zweiten, dritten Mal der Effekt nicht mehr vorhanden ist. Was bleibt, ist meistens eine Szene, welche spielerisch nichts mehr zu bieten hat.

Ein Fehler, der oft gemacht wird: Ein Spiel wird so konstruiert, dass das Spiel jedesmal praktisch genau gleich abläuft. Meistens wird der Spieler auf einen Weg gezwungen, wo immer die gleichen Gegner sind, die immer gleich angreifen, die natürlich auch immer gleich besiegt werden. Langfristig ist das natürlich höchst uninteressant.

Moderne Spiele sollten dem Spieler möglichst viel Handlungsfreiraum überlassen und auch Zufalls-Elemente ins Spiel bringen, damit eine gewisse Abwechslung eintritt. Gerade beim Einsatz des Zufalls ist aber höchste Vorsicht geboten, der Schuss kann auch nach hinten los gehen, wenn unfaire Situationen entstehen.

Spiele sollten auch immer wieder Geschicklichkeits-Szenen anbieten, welche dafür sorgen, dass der Spieler sich konzentrieren muss. Das Ziel eines Spieles sollte es sein, den Spieler zu fesseln, seine Konzentration voll in Anspruch zu nehmen, damit auch eine spannende Atmosphäre entsteht. Sobald der Spieler sich zurücklehnt und das Spiel aus der Distanz mit Leichtigkeit überwindet, ist die Motivation, dieses Spiel zu spielen, praktisch schon verflogen.

1.45 Demo-Versionen

90 Prozent der Demo-Versionen, die ich bisher gesehen und getestet hatte, waren kompletter Schrott.

Es ist kaum zu glauben, wieviel Energie verloren geht, nur weil die Präsentation eines Spieles mit einer Demo-Version mangelhaft vollzogen wird. Demo-Versionen sollten einen Einstieg in das Spiel bieten und auch die Vorzüge des Spieles demonstrieren. Demo-Versionen werden auch sehr oft als Massstab angewendet, ob ein Spiel gekauft werden soll oder nicht. In Zeiten, wo in einschlägigen Fach-zeitschriften praktisch alle Spiele als 'gut' bezeichnet werden, ist das wohl die einzige Möglichkeit, sich ein Bild von einem Spiel zu machen.

Demos, welche eine äusserst kurze Aufenthalts-Genehmigung auf meiner Harddisk erhalten:

- Demos, wo ich nach 10 Sekunden sterbe, ohne dass ich weiss, was überhaupt passiert ist.
- Demos, welche bereits nach 2 Sekunden tonnenweise Objekte auf den Bildschirm

bringen, die mich erledigen wollen.
- Demos, welche alle Elemente des gesamten Spiels auf einmal präsentieren wollen und mich völlig überfordern.
- Demos mit einer schlechten, unpräzisen Steuerung.
- Demos ohne vernünftige Dokumentation.
- Demos, die nicht oder instabil laufen.

Damit ist schon einiges dazu gesagt, wie Demos aussehen sollten. Demos werden oft mit Prototypen verwechselt. Ein Prototyp darf alle obigen Punkte verletzen. Er ist aber nicht für die Oeffentlichkeit bestimmt. Die potentiellen Kunden haben anderes verdient.

Demos sollten eine Tutorial-Funktion übernehmen. Sie sollten den Spieler langsam in das Spiel hineinbringen und einige Elemente vorstellen und dem Spieler Aufgaben stellen, die er bewältigen kann.

Demos sollten genug Spiel-Zeit zur Verfügung stellen, damit sich der Spieler ein Bild vom Spiel machen kann.

Es ist wichtig, dass genügend Zeit in das Erstellen von Demo-Versionen investiert wird. Diese Zeit rendiert mit Sicherheit, da eine gute Demo-Version den potentiellen Kunden zum Kauf motiviert.

1.46 Gedanken zu 3D

Hier möchte ich noch speziell auf die 3D-Thematik zu sprechen kommen. Die 3D-Spiele erlebten schon vor einiger Zeit einen regelrechten Boom, da die Hardware der Konkurrenzsysteme die benötigte Geschwindigkeit erreichte. Auf dem AMIGA war sehr lange Zeit überhaupt nichts in der Richtung zu finden.

Heutzutage ist die Technik der 3D-Spiele auf den Konkurrenzsystemen schon sehr weit fortgeschritten. Und jetzt beginnt man auf dem AMIGA langsam, die alten 3D-Spiele zu kopieren.

Die meisten 3D-Spiele auf dem AMIGA verwendet noch die antiquierte Boden-Wand-Technik, welche den Spieler immer noch stark in der Bewegungsfreiheit einschränkt. Auf Konkurrenzssytemen ist die völlige Bewegungsfreiheit schon seit langer Zeit der Standard.

Ich möchte hier an dieser Stelle all diejenigen, welche sich mit der 3D-Technik auskennen, auffordern, nicht einfach permanent den Rückstand zu den Konkurrenz-systemen aufrechtzuerhalten, indem halt alte Quelltexte einfach ohne jegliche Kreativität auf den AMIGA gezogen werden, sondern die neuesten Techniken anzu-wenden, damit der Rückstand bald nicht mehr existiert.

Mit dem PowerPC-Prozessor erhalten wir die Möglichkeit, dieselbe Geschwindigkeit zu erhalten, wie auf Konkurrenzsystemen, also sollten wir auch dieselbe neueste Technik anwenden. Zu diesem Zwecke sollten sich also alle 3D-Koryphäen zusammen-schliessen und neue 3D-Techniken definieren, welche das Ziel haben müssen, die Konkurrenz zu überbieten. Heute sollten die Spiele von morgen gemacht werden und nicht die Spiele von gestern!

1.47 Adresse des Autors

Ich habe viel Zeit investiert, um alle Gedanken und das ganze Know-How in dieses Dokument zu bringen. Ich möchte damit zeigen, dass es Zeit wird, dass das Know-How im technischen Bereich offengelegt werden soll, damit der AMIGA mit vereinten Kräften wieder den Anschluss zur Konkurrenz finden kann.

Es macht überhaupt keinen Sinn, technisches Know-How vor der 'bösen' Konkurrenz im AMIGA-Sektor zu verstecken. Ich möchte noch einmal daran erinnern: Die Technik ist nur ein Werkzeug. Die Qualität eines Spieles wird am Spiel selbst gemessen. Es ist also anzustreben, dass die Spielehersteller die technischen Belange offenlegen und sich danach auf die spielerische Seite der Spieleentwicklung konzentrieren, wo sie natürlich wieder ihre Betriebsgeheimnisse pflegen können.

Ich würde mich auch sehr über Kontakte zu Leuten freuen, welche im Sinne haben, einzigartige und innovative Spiele für den AMIGA zu entwickeln. Durch Diskussion und durch Offenlegung von technischen Finessen kann somit der Abstand zu den Konkurrenzsystemen verringert und die spielerische Qualität erheblich gesteigert werden, da nicht mehr so viel Zeit in die Technik investiert werden muss.

Wer mich kontakten will, kann das folgendermassen tun:

normale Briefpost:	eMail:
HAAGE&PARTNER GmbH z.Hd. Sam Jordan Mainzer Straße 10a D-61191 Rosbach Germany	s.jordan@haage-partner.com warpup@haage-partner.com