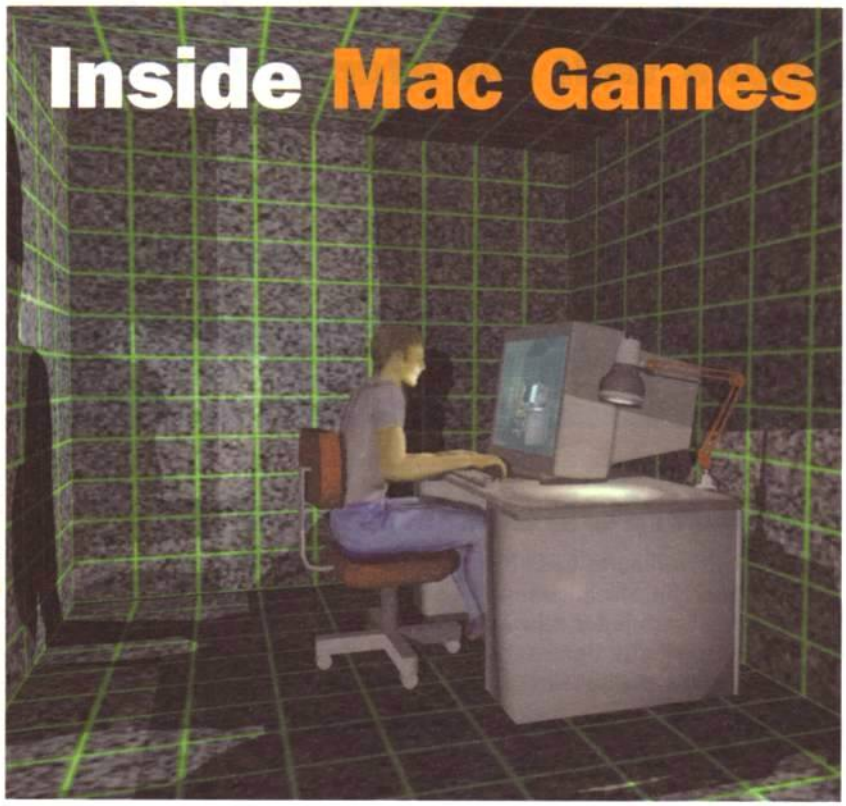


Heute wollen wir uns mal ein wenig mit den Problemen auseinandersetzen, die man als Programmierer von Spielen für den Apple Macintosh so hat. Dazu zählen neben den prinzipiell nötigen MacOS-Funktionen auch Fragen zur Optimierung sowie zur Portierung von Spielen auf den Macintosh.

Ich will hier nicht auf die Frage eingehen, was für ein Spiel man programmieren soll. Solche Ideen muß schon jeder selbst haben. Aber je nach Art des Spiels gibt es natürlich einiges zu beachten: neben der Abfrage von Tastatur, Maus und ggf. Joystick sind da auch noch die Unterstützung von Sound sowie der Aufbau der Grafik – die bei Action-Spielen ja sehr zeitkritisch ist. Wer ein Spiel auf den Markt bringen will, welches von mehreren Spielern gleichzeitig gespielt werden soll, muß sich zudem noch mit der Vernetzung beschäftigen. Hierzu zählen u.a. die Verbindung über Modem, Nullmodem, AppleTalk, TCP/IP und IPX.

Als Programmiersprache für ein Spiel wird man sicher C oder C++ wählen, evtl. Pascal. Bei Bedarf kann man sehr zeitkritische Sachen später immer noch in Assembler implementieren, wobei dies bei nur sehr, sehr wenigen Spielen überhaupt nötig sein wird. Eine Hochsprache hat den entscheidenden Vorteil, daß die Spiele damit deutlich portabler sind, so kann man ohne großen Aufwand das Spiel für 68k- und für PowerMacs rausbringen. Quasi alle auf dem Markt erhältlichen Spiele (egal ob für den Mac oder für PCs) sind in C geschrieben worden.

Fangen wir mit den, technisch gesehen, am einfachsten zu implementierenden Spielen auf dem Mac an, nämlich mit Strategiespielen oder Adventures à la Spaceward Ho und Myst. Natürlich ist mit „einfach" hier nicht der Aufwand für die Implementierung von Algorithmen für schlaue Aliens gemeint, sondern „nur" die verhältnismäßig einfache Einbindung ins MacOS. Man braucht keine besonderen Tricks anzuwenden, um ein brauchbares Spiel zu erhalten. So wird man hier die Fenster und Menüs vom MacOS nutzen. Die Maus- und Tastaturabfrage kann man ebenfalls direkt vom Event Manager erledigen lassen. Bei Bedarf wird man jedoch den Aufbau der Grafik im Fenster nicht mit MacOS-Funktionen bewerkstelligen wollen. Dazu später mehr. Ansonsten könnte man sofort mit dem Programmieren loslegen und den Rest dieses Artikels irgendwann einmal lesen.

Deutlich aufwendiger wird es mit Action-Spielen wie Marathon. Diese Spiele treiben nicht nur immensen Aufwand bei Grafik und Sound. Sie brauchen auch viel Rechenzeit, was die Nutzung des MacOS in vielen Punkten quasi ausschließt. Trotzdem will man natürlich ein Spiel herstellen, welches nicht nur Mac-like aussieht, sondern auch kompatibel zu möglichst vielen Macs ist.

DOS-Spiele auf dem Mac

In letzter Zeit werden immer mehr Spiele auf den Mac portiert. An diesen kann man teilweise sehr schön zeigen, wie ein Spiel nicht aussehen sollte.

Die Installation eines Spiels für den Mac sollte nicht wesentlich aufwendiger als das Kopieren einer Datei sein. Nur in Ausnahmefällen sollte ein Spiel aus mehreren Dateien bestehen, z.B. um neue Levels einfach nachliefern zu können. Man sollte auf alle Fälle vermeiden, daß ein Spiel aus Hunderten von winzigen Dateien besteht. Sowas sieht nicht nur schlecht aus, es belegt auch unnötig Platz auf der Platte des Anwenders.

Ein Spiel sollte sich automatisch an den Rechner anpassen. Dafür notwendig sind neben der Wahl des richtigen Grafikmodus auch die einer sinnvollen Tastaturbelegung und das Erkennen von nötigen Systemerweiterungen. Viele Spiele brauchen für die Nutzung von Sound heute den Sound Manager 3.0 (oder neuer) oder QuickTime 2.0 (für MIDI-Musik im Spiel), dies kann man abfragen. Ein Spiel, das wegen einer fehlenden oder zu alten Systemerweiterung abstürzt, wird nicht viele Freunde finden.

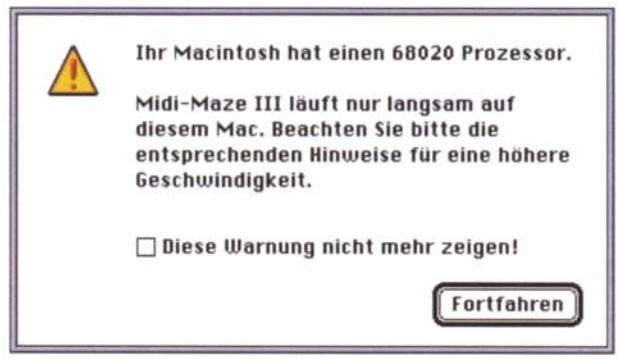
Spiele für den Mac sollten die Human Interface Guidelines beachten. Dazu zählt auch eine Menüleiste, die bei Action-Spielen häufig nicht sichtbar sein wird, um mehr Platz auf dem

Bildschirm zu haben. Aber sie wird automatisch aktiv, wenn der User Command drückt oder mit der Maus an den oberen Bildschirmrand fährt. Hier wird man neben der unvermeidlichen About-Box auch Funktionen für die Einstellungen des Tastatur-Layouts, die Monitornutzung und ggf. auch für das Laden und Speichern von Spielständen unterbringen. Über die Menüleiste kann man natürlich auch andere Programme aktivieren, um dem Chef zu zeigen, daß Excel immer noch rechnet ...

Die Portierung eines Spiels sollte spielbar bleiben. Ein Spiel, das auf einem Mac nur halb so schnell wie auf einem PC ist, wird ebenfalls nicht sehr erfolgreich sein. Da hilft auch nicht die Begründung, daß der Mac eine viel höhere Grafikauflösung habe. Den Spieler interessiert es nicht, daß ein Spiel in 320x200 Pixeln Auflösung ja schneller sein „muß" als ein Spiel in 640x480 Pixeln Auflösung. Ein Spiel in 320x200 Pixel für den Mac rauszubringen ist aber auch keine gute Idee. Briefmarken klebt man auf einen Briefumschlag, aber nicht auf den Bildschirm. Und 320x200 Pixel auf einem 21"-Monitor sind lachhaft. Eine – wenn auch keine gute – Lösung wäre es, die Auflösung hochzurechnen auf 640x400 Pixel, indem man alle Pixel in x- und y-Richtung verdoppelt. Das gibt aber definitiv Abzüge beim ersten Testbericht.

Die Oberfläche des Spiels, also z.B. das Hauptmenü, sollte auch ohne Anleitung bedienbar sein. Wenn der Spieler nicht erkennen kann, was er anklicken kann, oder wenn er nicht versteht, welches Icon was tut, dann hat man etwas falsch gemacht. Auch sollte die Bedienung nicht schlechter als im MacOS sein. Wenn ein Scroll-Balken nur Buttons zum Hoch- und Runterblättern hat, man ihn aber nicht mit der Maus anfassen und direkt bewegen kann, dann hätte man evtl. doch besser den Control Manager vom MacOS nutzen sollen.

Level-Paßwörter sind bei Macintosh-Spielen überflüssig. Man hat im Gegensatz zu Spielekonsolen ein File-System und kann sich den aktuellen Spielstand problemlos merken. Es ist einfach albern, daß der Spieler sich Level-Paßwörter auf einem Blatt Papier notieren muß, wo er doch einen Rechner hat.



Falls ein Spiel doch zu langsam wird, sollte man den User vorher warnen. Das erspart einem einige Beschwerdebriefe.

Die Tastatur

Die Tastaturabfrage ist nicht so einfach, wie man zuerst erwartet. Dies hat verschiedene Gründe. Der wichtigste Grund dürfte das Problem mit der MacOS Funktion *WaitNextEvent()* des Event Managers sein. Diese Funktion ist für Anwenderprogramme ganz wunderbar geeignet, aber innerhalb eines Action-Spiels kostet sie einfach zuviel Rechenzeit! Da bei jedem Aufruf dieser Funktion zwangsweise auf andere Prozesse umgeschaltet wird, verschenkt man rasch Rechenzeit, die man nicht hat. Zudem kann das Programm nicht schneller als mit 60Hz laufen, da die Umschaltung nur innerhalb dieses Zeitrasters erfolgen kann.

Kurzum: diese Funktion sollte man für Action-Spiele meiden! Nur: wenn wir sie meiden, programmieren wir kein sauberes Programm mehr. Aber mit einem großartigen Spiel hofft man auf Nachsicht bei den Usern. Wir können Probleme aber etwas mindern, wenn wir während nicht kritischer Funktionen außerhalb des eigentlichen Spiels, z.B. im Hauptmenü, doch *WaitNextEvent()* aufrufen. Und während des Spiels können wir die Funktion, wenn schon nicht ständig, so doch mindestens einmal pro Sekunde aufrufen. Dies dürfte ein brauchbarer Kompromiß zwischen Performance und Kompatibilität sein. Ich betone nochmals: dies gilt nur für zeitkritische Action-Spiele. Häufig braucht man sich darum nicht zu kümmern und kann *WaitNextEvent()* stets aufrufen.

Wenn man *WaitNextEvent()* nicht aufruft, kann man mit *GetKeys()* den Zustand der Tasten abfragen. Leider muß man hier mit Scancodes statt mit ASCII-Codes arbeiten, und Scancodes sind nicht portabel. Mit den international Utilities kann man sich aber Scancodes in ASCII-Codes umrechnen lassen. Überhaupt sollte man daran denken, daß es mehr als nur eine Tastatur für den Mac gibt. Funktionstasten und Zehnerblock wie z.B. die Page-Up/Down-Tasten gibt es nicht auf allen Tastaturen! In anderen Ländern gibt es andere Belegungen. So ist es immer wieder ein „Genuß", auf die Frage: „Want to quit the game?" mit „Z" zu antworten, weil in den USA ja das „Y" an dieser Stelle auf der Tastatur liegt. Auf dem Mac würde man solche Abfragen allerdings eh mit einem ordentlichen Alert erledigen.

Zudem sollte das Spiel jederzeit mit Command-Q oder Command-Punkt beendet bzw. abgebrochen werden können. Dies wird man üblicherweise mit einer Abfrage in Form eines Alerts absichern, damit der User im hektischen Spielgeschehen sich nicht auf einmal im Finder befindet. Auch sollten Return oder Enter stets für „Ja" oder „Weiter", Escape und Command-Punkt stets für „Nein" oder „Abbruch" stehen. Dies erwarten Benutzer einfach von einem Mac-Programm.

Innerhalb des Spiels sollte die Tastaturbelegung durch den Benutzer editierbar sein. Jeder hat seine eigenen Gewohnheiten, und die sollte er auch in seinen Spielen beibehalten können. Die Standardbelegung von unserem Spiel sollte allerdings so gewählt sein, daß man die Belegung nicht als erstes ändern muß ... Dazu gehört es auch, daß wir uns mal ein paar andere Spiele ansehen und uns Anregungen für die Tastaturbelegung unseres Spiels holen. So kann man in allen Marathon- und Doom II-Spielen mit den Cursor-Tasten durch das Labyrinth wandern.

Achtung: das MacOS kann nicht beliebig viele gleichzeitig gedrückte Tasten verwalten. Wir sollten aber auch von einem Spieler nicht verlangen, in einem Spiel 4 oder 5 Tasten gleichzeitig zu drücken.

Die Maus

Die Maus kann man auf zweierlei Art in einem Spiel nutzen: als absolutes oder als relatives Eingabegerät. Ersteres wird man bei Menüs oder in Simulationen à la SimCity nutzen. Die Mausposition kann man einfach mit *GetMouse()* bzw. *Button()* abfragen. Wer *WaitNextEvent()* aufruft, bekommt die Koordinaten eh automatisch mitgeteilt.

Die relative Abfrage kann man benutzen, um mit der Maus eine Spielfigur zu bewegen. Die Maus hat hierbei gegenüber der Tastatur den Vorteil, daß man unterschiedliche Geschwindigkeiten gemeldet bekommt. Bewegt man die Maus schnell, so bewegt sich die Spielfigur schnell. Das Prinzip der relativen Abfrage ist dabei recht einfach zu realisieren: man setzt bei jedem Abfragedurchlauf die Mausposition auf die Bildschirmmitte und fragt nach einem Durchlauf die aktuelle Position ab. Die Differenz ist dann die relative Bewegung. Den Mauszeiger darf man natürlich nur dann versetzen, wenn er nicht auf dem Bildschirm sichtbar ist! Ansonsten würde man ganz entschieden gegen die Human Interface Guidelines verstoßen! Aber wenn der Mauszeiger nicht sichtbar ist, kontrolliert der Spieler halt etwas anderes auf dem Bildschirm mit der Maus, und da ist es ihm egal, daß wir den Mauszeiger ständig auf die Bildschirmmitte setzen. Nochmals Achtung: die Mausposition sollte man retten, damit man im Hauptmenü den Mauszeiger wieder an die Stelle setzen kann, wo der Spieler ihn zuletzt gesehen hat.

Man sollte auch nicht vergessen, daß einige Macs gar keine Maus haben! Keine Maus? Ja, ich z.B. habe nur ein Grafiktablett am Mac. Für eine relative Abfrage ist es extrem schlecht geeignet. Deswegen sollte man immer erlauben, daß der Spieler mit der Tastatur spielen kann!

Joystick

Es gibt für den Mac zwei Arten von Joysticks: digitale und analoge. Die digitalen Joysticks sind dabei einfache Schalter, die für die vier Richtungen vier Tasten aktivieren. Für die Richtung links oben können sie wahlweise eine andere Taste benutzen oder beide gleichzeitig. Wir brauchen unser Spiel also normalerweise nicht an diese Joysticks anzupassen, da sie sich wie eine Tastatur verhalten – üblicherweise durch ein Kontrollfeld gesteuert. Wir sollten aber evtl. gleich Settings-Files für diverse Joysticks mitliefern, die ein User einfach installieren kann.

Analoge Joysticks verhalten sich ähnlich wie eine Maus, sie „sagen" nicht nur „nach rechts", sondern auch, wie schnell der Spieler nach rechts will. Diese Joysticks werden sehr gerne bei Flugsimulatoren eingesetzt. Auch für diese Joysticks hier gibt es Treiber-Software, aber auch Developer Kits, so daß man die Joysticks direkt vom eigenen Programm aus ansteuern kann.

Grafik – Farbtiefen

Der Mac hat einen sehr einfach aufgebauten Videospeicher. Er ist im Gegensatz zu dem des ATARI oder AMIGA nicht in Planes unterteilt, sondern streng linear aufgebaut. Dies bedeutet, daß im s/w-Modus jedes Pixel einem Bit im Videospeicher entspricht – ganz so, wie man es auf dem Bildschirm sieht. Im 256-Farben-Modus ist dann ein Byte im Video-RAM ein Pixel auf dem Bildschirm. Bei Truecolor sind es vier Bytes.
Verbreitete Auflösungen für Spiele sind s/w, 16 Farben, 256 Farben und 32768 Farben. s/w ist bei den alten Mac Plus und auch bei den PowerBooks verbreitet. 16 Farben bzw. Graustufen gibt es ebenfalls bei PowerBooks und bei alten Farb-Macs. 32768 Farben oder mehr sind etwas für High-End-Macs mit viel Rechen-Power und Video-RAM oder für Multimediaspiele und -Adventures wie Myst.

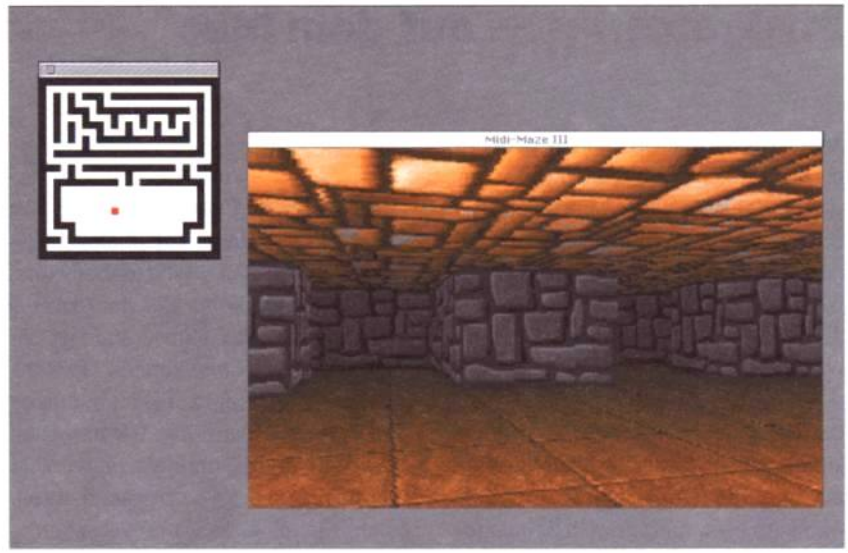
Heute werden fast ausschließlich Spiele für 256 Farben produziert. Einige wenige unterstützen auch den s/w Modus, damit die recht flotten 68030-PowerBooks genutzt werden können. Diese sind gerade bei Spielen, die erst mit mehreren Spielern so richtig Spaß machen, wie Spaceward Ho, sehr verbreitet, da sie leicht zu transportieren sind. Zudem gibt es nichts Witzigeres, als im Hörsaal oder im Flugzeug auf diese Art und Weise zwei Macs zu vernetzen: klick ... klick, und zwei Rechner sind vernetzt. Man kommt dann aber normalerweise kaum zum Spielen, da man alle möglichen Fragen beantworten muß ...

Bildschirmgrößen

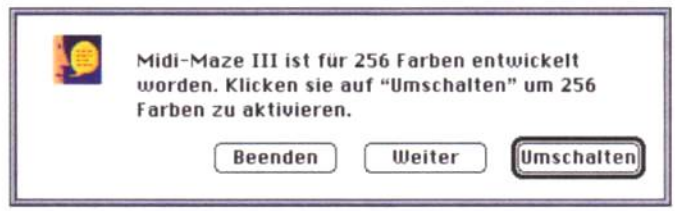
Die Größe des Bildschirms ist beim Mac sehr unterschiedlich, und im Gegensatz zum PC gibt es keine allgemeine Möglichkeit, einfach die Auflösung runterzuschalten. Somit gibt es bei den Farbmonitoren 512x384 Pixel –bei den LCs –, die Standardauflösung 640x480 in 256 Farben und 832x624, 1024x768, 1152x872 bis zu 1600x1200 Pixel bei den neuen PCI-Macs. Man stelle sich nun mal ein Spiel mit einer festen Auflösung von 512x384 Pixeln auf einem solchen Monitor vor: gerade mal 1/10 des Monitors wäre belegt. Hier gibt es zwei Lösungen: Flugsimulatoren wie F/A-18 unterstützen durch ihre Vektorgrafik auch größere Monitore, oder man erwartet vom Benutzer, daß er zumindest auf 640x480 runterschaltet, sonst muß er halt mit einem kleinen Bild leben. Das sind die üblichen Methoden.

Welche Auflösungen und Farbtiefen man unterstützt, hängt sehr vom Spiel und von der Arbeit ab, die man sich mit der Grafik machen will. Standard ist 640x480 in 256 Farben. Ebenfalls sinnvoll kann 640x400 in s/w sein, um auch PowerBooks zu unterstützen. Auflösungen, die kleiner als 640x480 sind, lohnen sich in der Regel nicht mehr, außer bei Lernspielen für Kinder, da Eltern ihren Kindern häufig abgelegte LC-Macs mit der Auflösung 512x384 vermachen. Größere Auflösungen sind häufig etwas für High-End-Macs, da kleinere Rechner kaum in der Lage sein dürften, den Videospeicher 10- bis 20mal in der Sekunde zu füllen. Auch hier gilt: Strategiespiele wie Spaceward Ho oder SimCity wirken erst auf großen Monitoren richtig gut!

Im 256-Farben-Modus wird die Farbpalette über den Paletten Manager des MacOS definiert. Die Farbwerte der 256 Farben sind ein Index auf eine CLUT (Color Lookup Table), in welcher der richtige Farbwert enthalten ist. Dies ermöglicht es auch sehr schön, eine Szene auszublenden, indem man einfach die Farben in der CLUT in Schwarz übergehen läßt. Das Video-RAM braucht man dazu nicht zu ändern. Dies wird sehr gerne genutzt, um Explosionen o.ä. visuell zu verdeutlichen: man läßt einfach die Farben in der CLUT hell aufblitzen. Bei Farbtiefen von 32768 und 16.7 Mio Farben gibt es keine CLUT mehr. Hier wird der Farbwert direkt im Video-RAM ablegt. Obige Effekte sind somit nur durch das Ändern des kompletten Video-RAM möglich. Achtung: bei PowerBooks mit 16 Farben haben wir zwar eine CLUT, können diese aber nicht ändern!



So kann ein Action-Spiel mit Fenstern und Nutzung von *CopyBits()* aussehen. Natürlich fehlt es hier noch an einer schöneren Oberfläche ...]



Wählt der User „Weiter" an, so wird das Spiel deutlich langsamer, läuft aber trotzdem, denn *CopyBits()* setzt die Farbtiefe dann automatisch um!

Probleme beim Start

Nehmen wir also an, daß wir ein Spiel mit einer Auflösung von 640x480 Pixeln im 256-Farben-Modus programmieren wollen. Was gilt es zu beachten?

1. Wir öffnen ein Fenster, welches den gesamten Bildschirm umfaßt.
Die Funktion *CopyBits()* kann nur in ein Fenster blitten. Zudem bekommen wir dann bei unseren Aufrufen von *WaitNextEvent()* auch automatisch Update-Events, so daß wir bei Alerts und Dialogen den Hintergrund wieder ordentlich updaten können. Dies ist insbesondere nötig, wenn der Benutzer von unserem Spiel auf ein anderes Programm umschaltet, oder wenn er eine Diskette einlegt, um sie zu formatieren, und das System nach dem Namen für diese Diskette fragt. An dieses Fenster können wir dann auch eine Palette binden, so daß CLUT mit den richtigen Werten geladen ist, wenn unser Programm aktiv ist.

2. Der Monitor könnte zu klein sein
In diesem Fall könnten wir nur einen Teil vom Bild darstellen, damit das Spiel trotzdem – wenn auch nicht so schön – läuft. Oder wir beenden das Programm mit dem freundlichen Hinweis, daß ein neuer Mac fällig wäre.

3. Der Monitor könnte zu groß sein
In der Mitte unseres Fensters stellen wir unser 640x480 Pixel großes Bild dar, den Rest färben wir schwarz. Denkbar wäre eine automatische Umschaltung der Auflösung innerhalb unseres Spiels, wenn der Display Manager dies erlaubt und ein Monitor angeschlossen ist, der dies unterstützt. Achtung: wenn der User den Prozeß wechselt, wird er sehr überrascht sein! Das ist nicht gut! Zudem hat der Display Manager die unangenehme Eigenschaft, die Icons auf dem Desktop neu zu sortieren, wenn der Bildschirm kleiner wird. Dies kann sehr nervig sein.

4. Die Farbtiefe könnte unpassend sein
Hier gilt es im System zu testen, ob der Monitor in der Lage ist, die gewünschte Farbtiefe zu unterstützen. Wenn das der Fall ist, wird man mit einem Alert nachfragen, ob man die Farbtiefe umschalten soll! Dies erspart dem User das manuelle Umschalten der Farbtiefe. Achtung: beim Ende unseres Spiels sollten wir die Farbtiefe natürlich wieder auf den ursprünglichen Wert zurücksetzen!

5. Der Benutzer hat mehrere Monitore
Dies ist der kompliziertesteste Fall! Viele Spiele haben damit Probleme, daß der Benutzer einen 19"-s/w-Monitor hat, auf dem sich die Menüleiste befindet und einen kleinen 13"-Monitor mit 256 Farben. Diese Spiele meinen dann: auf einem s/w-Monitor können wir nicht laufen und beenden sich. Dabei müssen sie einfach nur überprüfen, ob nicht ein weiterer Monitor vorhanden ist, der eventuell eher ihren Wünschen entspricht. Ein gutes Programm sollte dem Benutzer immer die Möglichkeit geben, zwischen seinen Monitoren zu wählen! Der Flugsimulator F/A-18 unterstützt sogar drei Monitore gleichzeitig, um verschiedene Ansichten darzustellen. Achtung: Wenn der Benutzer zu Hause das Bild eines Spiels auf den zweiten Monitor gelegt hat und nun mit Rechner und Monitor zu einer Party zieht, ist er ziemlich erstaunt, wenn er kein Bild bekommt ... Ein gutes Spiel sollte also abfragen, ob sich die Systemkonfiguration geändert hat.

Während eines Spiels kann der Platz, den die Menüleiste belegt, eventuell benötigt werden. Vielleicht stört einen die Menüleiste auch nur. Jedenfalls wird man sie häufig wegschalten wollen. Man sollte aber daran denken, sie wieder anzuschalten, wenn der User

Command drückt oder wenn er die Maus (so sie denn angeschaltet ist) an den oberen Bildschirmrand bewegt.

Zugriff auf das Video-RAM

Generell gilt: man sollte nie direkt in den Videospeicher schreiben! Und wenn man es tut, dann sollte man zumindest wissen, wie man es richtig macht. Ansonsten wird man immer wieder Rechner finden, auf denen das eigene Spiel nicht läuft. Also: vorzugsweise die Möglichkeit bieten, auf MacOS-Routinen für den Zugriff auf den Bildschirmspeicher umschalten zu können!

Normalerweise wird man nicht direkt ins Video-RAM schreiben, da die *CopyBits()*-Funktion vom MacOS schnell genug ist. Dies setzt aber voraus, daß man sie optimal einsetzt. Dazu wird man normalerweise folgenden Ansatz wählen: man alloziert einen Offscreen-Bildschirmspeicher, in dem man seine Grafik komplett aufbaut, und blittet diese am Ende mit der *CopyBits()*-Funktion ins Fenster. Selbst Spiele wie Marathon nutzen *CopyBits()* auf diese Art und Weise. Es gibt also keinen Grund, sich mit Performance-Argumenten rauszureden. Ein Beispiel für diese Art der Programmierung findet sich im angehängten Source-Fragment.

Eine Offscreen-Bitmap alloziert man seit System 7 üblicherweise als GWorld. Wer kompatibel zu System 6 programmieren muß, sollte sich die entsprechende Offscreen-Technote von Apple besorgen, da wird ein ziemlich langer Beispielsource mitgeliefert.

Sound

Was wäre ein tolles Spiel ohne die entsprechenden Soundeffekte und Hintergrundmusik? Der Mac unterstützt einem dabei sehr. So kann man die Musik mit einem Sequencer komponieren und dann als Standard-Midi-File über den in QuickTime 2.0 eingebauten Synthesizer abspielen. Die so erzeugte Musik kostet nur sehr wenig Plattenplatz. Spiele wie Marathon und Doom II nutzen diese Funktion in QuickTime.

Soundeffekte werden beim Mac über Soundkanäle abgespielt, die man vorher alloziert. Auf jedem Kanal kann zur Zeit ein Effekt oder eine Musik abgespielt werden. Die Anzahl der Kanäle hängt jedoch von der Rechenleistung des Macs ab, so daß man die Musik und die Soundeffekte zumindest getrennt abschaltbar einrichten sollte – zum einen wegen der eben erwähnten Einbußen bei der Rechenleistung, zum anderen, um dem Spieler den Umweg über das Sound-Kontrollfeld zu ersparen, wenn er die Musik wider Erwarten doch nicht mag. Der Sound Manager ist vor Version 3.1 nocht nicht native, so daß PowerPC Spiele wie das bereits mehrfach erwähnte Marathon besonders darunter leiden. Also: Anzahl der Soundkanäle begrenzen oder Sound und Musik ganz abschalten.

Netzwerkfähigkeit

Viele Spiele machen noch einmal soviel Spaß, wenn man sie mit mehreren im Netz spielt. Dazu bieten sich auf dem Mac viele Möglichkeiten an, die ich hier aber aus Gründen des Platzes nur anreißen kann:

Modem

Hierbei können zwei Spieler zusammen spielen. Üblicherweise wird mit Hilfe der Communication Toolbox eine Verbindung zwischen den beiden Modems aufgebaut. Alternativ könnte man mit AppleTalk Remote Access eine Vernetzung aufbauen, dies kostet jedoch extra Geld und zudem mehr Performance, als wenn man die Modemverbindung selbst aufbaut.

Nullmodem

Mit einem einfachen Kabel kann man zwei Macs über die serielle Schnittstelle verbinden. Dies dürfte bei Zwei-Spieler-Spielen sehr verbreitet sein und ist programmtechnisch sehr einfach zu erledigen. Da viele Macs zwei serielle Schnittstellen haben, wäre einen Midi-Maze-ähnliche, ringförmige Vernetzung per Nullmodem ebenfalls denkbar. Der serielle Device-Treiber ist sehr leicht zu benutzen.

AppleTalk

Die Standardmethode zur Vernetzung von zwei oder mehr Macs! Sie funktioniert sowohl über LocalTalk als auch über ein bestehendes EtherNet-Netzwerk. Bei Strategiespielen kann man AppleEvents durch das Netz schicken, bei Action-Spielen wird man DDP-Pakete direkt verschicken. Dann muß man sich aber um die Fehlerkorrektur selbst kümmern. Näheres findet sich hierzu in den NIMs „Interapplication Communication" für die AppleEvents und in „Inside AppleTalk" für DDP.

TCP/IP

Erlaubt das Vernetzen von Spielen über das Internet. Hierzu müssen das Kontrollfeld MacTCP und in der Regel MacPPP installiert sein. Vorteil: man kann in der ganzen Welt mit Leuten spielen. Nachteil: sehr unterschiedliche Performance. Hauptsächlich für Strategiespiele oder Simulationen gedacht. Die MacTCP-Dokumentation hilft dem Programmierer weiter.

IPX

Will man Macs mit PCs vernetzen, so kann man mit dem Kontrollfeld MacIPX (kommt von Novell und liegt unter anderem auch FileMaker Pro bei) das Novell-Netzwerkprotokoll auch auf Macintosh Rechnern nutzen. Näheres wird man wahrscheinlich bei Novell erfahren. Soweit mir bekannt ist, nutzt nur Doom II für den Mac diese Art der Vernetzung. Sie macht auch nur Sinn, wenn man sein Spiel für Mac und PC anbietet.

Literatur

Jeder, der sein Spiel für den Mac programmieren will, sollte sich unbedingt „Tricks of the Mac game programming gurus" von Hayden Books (ISBN 1-56830-183-9, US$ 50) besorgen. Auf knapp 900 Seiten und einer CD-ROM wird alles beschrieben, was man als Programmierer so wissen muß – selbst QuickDraw 3D wird schon beschrieben. Dieses Buch ist eine echte Pflichtlektüre!

So, an dieser Stelle verabschiede ich mich von all den mutigen Lesern, die wieder einmal bis zum Ende meines Artikels durchgehalten haben. Übrigens: Wer eine Idee für einen Artikel zum Thema Programmieren hat, kann mir ja auch gerne eine EMail schicken (*<URL:mail://mf@maushh2.hh.provi.de>* oder *Markus Fritze @ HH2* im MausNet)

MFR

```
1 /***
2  *   Beispiele für den kompatiblen Zugriff auf den ¬
   Videospeicher
3  *  (c) 1995 MAXON Computer
4  * Autor: Markus Fritze
5  ***/
6 WindowPtr            view3DWindow;
7 GWorldPtr            view3DGWorld;
8 PixMapHandleview3DPixMap;
9 CTabHandle           AktCLUT;
10
11     /***
12      *      Window mit Farbpalette öffnen und passende ¬
   GWorld dazu
13      ***/
14     void            InitWindow()
15     {
16             // Maze-Window holen
17             view3DWindow = GetNewCWindow(128, nil, ¬
       (WindowPtr)-1L);
18             SetPort(view3DWindow);
19             ShowWindow(view3DWindow);
20
21             // Farbpalette an das Fenster hängen
22             AktCLUT = GetCTable(129);
23             PaletteHandle   hPalette = NewPalette(256, ¬
       AktCLUT, pmExplicit | pmTolerant, 0x100);
24             NSetPalette(view3DWindow, hPalette, ¬
       pmAllUpdates);
25             ActivatePalette(view3DWindow);
26
27             // Offscreen Bitmap
28             if(NewGWorld(&view3DGWorld, 8, ¬
       &view3DWindow->portRect, AktCLUT, nil, 0L) != ¬
       noErr)
29                     ExitToShell();
30
31             // Bitmap löschen
32             CGrafPtr        savePort;
33             GDHandle        saveHandle;
34             GetGWorld(&savePort, &saveHandle);
35             SetGWorld(view3DGWorld, NULL);
36             EraseRect(&view3DGWorld->portRect);
37             SetGWorld(savePort, saveHandle);
38
39             view3DPixMap = GetGWorldPixMap(view3DGWorld);
40             LockPixels(view3DPixMap);
41     }
42
43     /***
44      *      Window schließen
45      ***/
46     void            ExitWindow()
47     {
48             DisposeGWorld(view3DGWorld);
49             DisposeWindow(view3DWindow);
50             DisposeCTable(AktCLUT);
51     }
52
53     /***
54      *      Unsere Zeichenfunktion
55      ***/
56     void DrawIt()
57     {
58      SetPort(view3DWindow);
59
60      // Calculate the address of the first byte of ¬
   the destination.
61      char   *screenMemPtr = ¬
   GetPixBaseAddr(view3DPixMap);      // Pointer to ¬
   video memory
62
63      // High bit of pixMap rowBytes must be cleared.
64      long   stripRowBytes = 0x7FFF & ¬
   (**view3DPixMap).rowBytes;
65
66      // Change to 32-bit addressing mode to access ¬
   video memory. The previous addressing mode
67      // is returned in mmuMode for restoring later.
68      char   mmuMode = true32b;
69      SwapMMUMode(&mmuMode);
70
71
72      // Zugriff auf unseren Videospeicher
73      ...
74
75      SwapMMUMode(&mmuMode);         // Restore addressing ¬
   mode back to what it was.
76
77      ForeColor(blackColor); // und in das 3D Window
kopieren
78      BackColor(whiteColor);
79      CopyBits((BitMap*)*view3DPixMap, ¬
   &view3DWindow->portBits, ¬
   &(*view3DPixMap)->bounds, ¬
   &view3DWindow->portRect, srcCopy, nil);
80     }
```