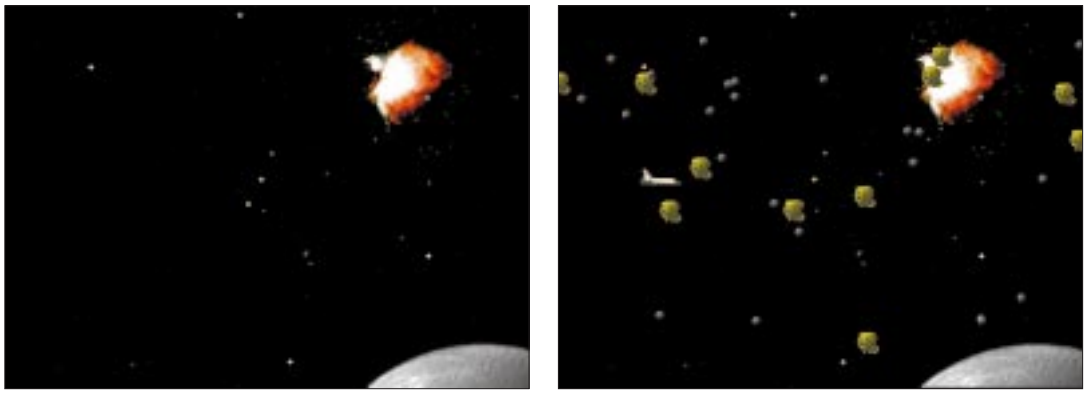


DAS HINTERGRUND-BILD unseres Spiels ist eine starre Collage aus Bildern von Sternen, Nebeln und Monden. Rechts das fertige Spiel mit bewegten Sprites in Aktion.

ACTION SELBST GEMACHT – SPIELE-PROGRAMMIERUNG

NACH DER ARBEIT mit den vielen Anwendungsprogrammen will sich der Mensch auch mal erholen. Aus diesem Grund wollen wir in dieser Folge einen Einblick in die Spieleprogrammierung geben. Wenn alles gut geht, kommt dabei ein Geschicklichkeitsspiel für die Arbeitspausen heraus

VON CHRISTIAN SCHMITZ

VIELE LESER FRAGEN UNS, ob man in Realbasic auch Spiele realisieren kann. Die Antwort lautet: Ja, aber ... Realbasic bietet zwar verschiedene Grundkonzepte zum Programmieren von Spielen an, alle Spielarten, insbesondere die modernen First-Person-3D-Spiele mit Licht- und Schwerkrafteffekten lassen sich jedoch nicht ohne zusätzliche Realbasic-Plug-ins verwirklichen.

DAS SPIEL ALS ANWENDUNGS-PROGRAMM

Die einfachste Möglichkeit, ein Spiel in Realbasic zu programmieren, besteht darin, es als normale Anwendung zu schreiben. Beispiele dafür sind viele Karten- oder Brettspielsimulationen, die in der „PD & Shareware"-Szene verbreitet sind. Wir benutzen die üblichen Systemressourcen, also normale Fenster und Buttons für die Dialoge. Wie in unserem Multimedia-Programm aus dem letzten Heft spielen wir Bilder und Filme per Quicktime ein. Auf dieser Basis lässt sich wunderbar eine Börsensimulation oder ein komplexeres Spiel wie etwa „Sim City Classic" entwickeln.

SPRITES MIT REALBASIC-SPIELE-ENGINE

Spiele arbeiten aber auch mit den so genannten „Sprites". Ein Sprite ist ein kleines Pixel-Bild, das sich mit einer bestimmten Geschwindigkeit und Richtung über den Bildschirm bewegt. Realbasic stellt eine eigene Sprite-Umgebung, das Spritesurface, zur Verfügung, das sich komplett um die Darstellung und Verwaltung der Sprites kümmert. Dabei benutzt das Spritesurface ein bildschirmfüllendes Fenster, das quasi als Bühne fungiert und ein beliebiges Bild als Hintergrund enthält. Alles was sich vor dem Hintergrundbild bewegen soll, definiert man als Sprites. Die Bewegung entsteht, indem man die Bildkoordinaten der Objekte laufend ändert. Animationen, etwa ein laufendes Männchen, erzeugt man, wenn man den Bildinhalt des Sprites entsprechend seiner Bewegungsphase ändert.

Glücklicherweise braucht man sich keinerlei Gedanken über die Darstellung der Sprites auf dem Bildschirm und die Restaurierung (Redraw) des Hintergrundbildes zu machen, denn das gehört zu den Aufgaben der Realbasic-Sprite-Umgebung. Sie sorgt zudem dafür, dass unser Programm benachrichtigt wird, wenn sich zwei Sprites auf dem Schirm berühren (Kollisionsabfrage).

Da man diese Spieletechnik einfach programmieren kann, gab es sie schon in den Heimcomputern und Spielekonsolen der ersten Generation. Der Commodore C64 brachte die Sprite-Spieleprogrammierung mit seinen berühmten Hardware-Sprites (die direkt im Grafikchip integriert sind) auch zu den Hobbyprogrammierern. Heute arbeiten die Grafikkarten und Hauptprozessoren so flott, dass sie die nötige Mathematik selbst übernehmen können, ohne auf einen speziellen Hardwarechip zurückzugreifen.

3D-SPIELE MIT SPEZIELLER ENGINE

Das höchste der Gefühle sind 3D-Engines. Dabei handelt es sich um Umgebungen, die dem Spritesurface ähneln, jedoch drei Dimensionen bieten. Üblicherweise lädt man fertige 3D-Objekte, oder man erzeugt einfache Figuren, wie Kugeln oder Würfel, zur Laufzeit. Zudem gibt es oft eine Kollisionsprüfung und Effekte wie Nebel und Schwerkraft. Die Programmierung eines solchen Spiels würde den Rahmen dieser Serie bei weitem sprengen. Wer sich dennoch dafür interessiert, findet unter www.xs3d.com ein kommerzielles Realbasic-3D-Plug-in inklusive einiger beeindruckender Beispiele im Internet. Unter http://techwind.com/rb3d wird zurzeit ein Plug-in (Freeware) entwickelt, das sich noch im Pre-Alpha-Stadium befindet. Unter www.splitsw.com findet man außerdem die „REAL RPG Engine", mit der sich komplexe Rollenspiele relativ einfach entwickeln lassen.

EINFACHES PRINZIP: UNSER WELTRAUMSPIEL

Bei unserem Realbasic-Spiel handelt es sich um eine einfache Weltraumsimulation. Es läuft in einer Auflösung von 640 x 480

Punkten im Vollbildmodus, also ohne Menüleiste. Wir möchten, dass einige Objekte, etwa Asteroiden oder Meteore, vor einem feststehenden Hintergrund von rechts nach links über den Bildschirm „fliegen“. Am linken Bildrand befindet sich unser Space-Shuttle-Raumschiff, dass der Spieler zwischen den Objekten hindurchmanövrieren muss, ohne mit ihnen zu kollidieren. Trifft eines der Objekte auf unser Raumschiff, löst es sich auf, und das Spiel ist zu Ende.

FRISCH ANS WERK

Bevor wir loslegen, benötigen wir die Objekte, das Raumschiff und die Asteroiden, als kleine Pixel-Grafiken. Wer will, kann sich – beispielsweise in Photoshop – selbst welche zeichnen. Der Einfachheit halber haben wir bereits einige Objekte vorbereitet. Sie befinden sich im Realbasic-Archiv auf der *Macwelt*-Internet-Seite unter www.macwelt.de/_magazin.

Es geht los mit einem neuen Projekt, in dem wir die insgesamt fünf Pixel-Bilder per Drag-and-drop einfügen. Dabei handelt es sich um die Bilder für den Hintergrund, den Rahmen um das Spielfeld, zwei Bilder für die Asteroiden und eins für unser Raumschiff. Alle diese Bilder bekommen von Realbasic automatisch passende Namen, die wir im Programmcode übernehmen.

Im Hauptfenster legen wir per Drag-and-drop einen Button (Push-Button oder Bevel-Button) an. Danach ziehen wir ein Spritesurface in eine freie Ecke des Fensters. Das Spritesurface ist das Steuerelement mit der Rakete im Symbol. Obwohl das Symbol im Editor auftaucht, sieht der Benutzer später nur den Button, nicht aber das Symbol für das Spritesurface. Da es zur Laufzeit keine Funktion hat, wird es von Realbasic versteckt. Wir können unser Fenster noch mit einigen erklärenden Bildern und Texten schmücken. Auf jeden Fall nennen wir den Button „Start“, denn wenn der Spieler diesen Knopf drückt, beginnt unser Spiel.

DAS SPIEL IST EINE KLASSE

Wir sparen uns einiges an Programmierarbeit, wenn wir das Spiel als Subklasse des Spritesurface anlegen. Deshalb erzeugen wir in unserem Projekt über den Menübefehl „Ablage/Neue Klasse“ eine neue Klasse, die wir in der Eigenschaften-Palette „Mein Spiel“ nennen. Als Super-Klasse wählen wir dort das Spritesurface aus. Damit haben wir eine Klasse geschaffen, in der wir die Eigenschaften, Events und Methoden der vorhandenen Spritesurface-Klasse beliebig ändern und erweitern können. Auf der Basis

dieser Klasse erzeugt unser Fenster ein Objekt und startet das Spiel. Der gesamte Programmcode des Spiels wandert damit in unsere Klasse. Im Hauptfenster findet lediglich der Aufruf zum Start statt.

Innerhalb der Klasse sehen wir die bekannten Events. Im Open-Event benutzen wir die Zeile „backdrop=hintergrund“, um den Hintergrund des Spritesurface festzulegen. Wie man sieht, fehlt hier das „me.“ und das „.self“. Beide Befehle sind überflüssig, denn alle Variablen und Methodennamen beziehen sich direkt auf die Eigenschaften und Methoden des Spritesurface. Jetzt ergänzen wir schnell noch das Hauptfenster, damit unser Programm den Hintergrund anzeigt. Im Event „Action“ des Buttons rufen wir per „spriteSurface1.run“ das Spritesurface auf und starten es direkt. Da das Programm unsere eigene Spritesurface-Klasse benutzen soll, ändern wir beim Spritesurface-Steuerelement die Eigenschaft „Super“ in „MeinSpiel“. Jetzt sollte unser Programm schon ohne Fehlermeldung laufen. Per Klick auf den Start-Button starten wir das Spritesurface. Der Bildschirm wird schwarz, und in der linken oberen Ecke zeigt sich das Hintergrundbild. Durch einen Mausklick lässt sich das Spritesurface wieder beenden.

KAPITÄN AN BORD!

Als Nächstes wollen wir unser Spielfeld auf dem Bildschirm zentrieren. Dazu ändern wir die Eigenschaften „SurfaceLeft“ und „SurfaceTop“. Zunächst ermitteln wir die x- und y-Auflösung des Bildschirms und subtrahieren davon jeweils die Breite und Höhe des Spielfeldes.

In Realbasic kann man zu jedem logischen Bildschirm ein Objekt ermitteln, das einige Informationen, etwa Höhe und Breite in Pixeln, die relative Position zum Hauptbildschirm und die Farbtiefe enthält. Die Funktion „ScreenCount“ liefert die Anzahl der angeschlossenen Bildschirme zurück.

Mit den folgenden zwei Codezeilen zentrieren wir die Spielfläche auf dem Bildschirm (Screen(0) ist der Hauptbildschirm):

surfaceleft=(screen(0).width-surfacewidth)/2
surfacetop=(screen(0).height-surfaceheight)/2

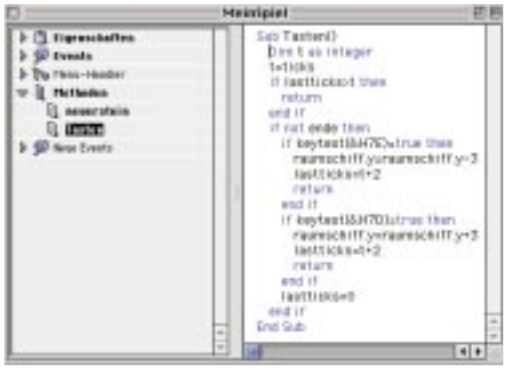
Dann erzeugen wir den Rahmen für das Spiel. Dazu benötigen wir ein Bild mit der Größe der Spielfläche. Die Fläche im Rahmen, durch die wir auf den Hintergrund schauen, muss dabei schneeweiß (RGB: 255, 255, 255) sein. Dann übermalt Realbasic diese Fläche nicht mit dem Rahmenbild, sondern zeichnet sie transparent. Damit das klappt, klicken wir auf das Bild im Projektfenster, und die Eigenschaften erscheinen in der Eigenschaftenpalette. Dort setzen wir die Eigenschaft „Transparent“ auf „white“.

Wenn wir den Rahmen als Sprite definieren, können wir ihn später dazu benutzen, Kollisionen zwischen ihm und anderen Objekten abzufragen.

Wir definieren also die Eigenschaft „rahmensprite as sprite“ in der Klasse (Menüpunkt „Bearbeiten > Neue Eigenschaft ...“). Diese Eigenschaft füllen wir im Open-Event mit der Zeile „rahmensprite=new sprite(rahmen,0,0)“ mit einem Sprite. Letzteres setzt das Bild mit dem Namen „rahmen“ an die Position 0/0. Das Bild und seine Position lassen sich jederzeit über die Sprite-Eigenschaften „picture“ und „x“, „y“ ändern. Um Realbasic mitzuteilen, ob und wie der Rahmen mit anderen Objekten zusammenstoßen soll, müssen wir die Eigenschaft „Group“ des Sprites angeben. Die Zeile „rahmensprite.group=1“ ordnet das Sprite der ersten Gruppe zu.

Bei allen Sprites mit positiven Gruppennummern erzeugt Realbasic beim Zusammenstoß einen Kollisions-Event. Diesen Event erzeugen Sprites mit der Gruppennummer 0 (das ist die Default-Nummer) niemals, während bei einem Sprite mit negativer Gruppe immer ein Kollisions-Event entsteht, egal mit welchem anderen Sprite es zusammenstößt.

Da wir in unserem Spiel keine Kollisionen mit dem Rahmen abfragen (es ist ja kein Ballspiel), setzen wir die Gruppennummer auf 0 oder lassen die Zeile ganz weg. Wichtiger als die Gruppe ist hier die Priorität beim Zeichnen der Objekte.



Realbasic zeichnet die Sprites nach ihrer Priorität oder bei gleicher Priorität in der Reihenfolge, in der sie erzeugt werden. Wir geben also unserem Spielfeldrahmen eine Priorität von 100, damit er immer ganz vorne liegt. Dazu schreiben wir die Zeile „rahmensprite.priority=100“.

Nun setzen wir endlich unser Raumschiff in den Weltraum. Dazu schalten wir im Bild „Raumschiffbild“ genau wie beim Rahmenbild die Eigenschaft „Transparent“ auf „white“ und erzeugen ein neues Sprite bei der Position 100/200. Dafür legen wir

DIESER CODE im Open-Event definiert den Spielerahmen und das Raumschiff als Sprite und zeichnet beide auf den Bildschirm.

erst einmal die Eigenschaft „raumschiff as sprite“ in unserer Klasse an. Diese füllen wir mit der Zeile „raumschiff=me.newsprite (raumschiffbild,100,200)“ im Open-Event. Mit „raumschiff.group=3“ wird das Raumschiff ein Teil der Gruppe 3. Das Bild oben zeigt den gesamten Code im Open-Event. Wenn wir das Programm nun starten, können wir das Raumschiff schon sehen.

ASTEROIDEN IM ANFLUG

Natürlich brauchen wir für unser Spiel ein paar angreifende Gegner. Diese Objekte erzeugen wir auch als Sprites, und sie sollen sich auf das Raumschiff zu bewegen. Zudem muss unser Programm auf das Verschwinden der Gegner reagieren.

Wir fügen nun als Erstes der Klasse einen Zähler in Form der Eigenschaft „count as integer“ hinzu. Im Nextframe-Event erhöhen wir diesen dann mit der Zeile „count=count+1“ pro Durchlauf um jeweils 1. Damit haben wir eine unbestechliche Uhr, die bei jedem dreißigsten Durchlauf einen großen Meteor und bei jedem fünfzehnten einen kleinen Asteroiden erzeugen und auf unser Raumschiff zu bewegen soll. Da ergibt sich ein Problem: Wie merkt unser Programm, dass genau 30 Durchläufe vorbei sind? Also schauen wir einmal ins Mathematikbuch. Dort finden wir die Möglichkeit, eine ganze Zahl durch eine andere ganze Zahl zu teilen und einen Rest zurückzuerhalten. Wir teilen also durch 30 und möchten den ganzzahligen Rest dieser Division ermitteln.

Der passende Realbasic-Befehl heißt „mod“ (von Modulo). Mit dem Vergleich „if count mod 30=0 then“ wird dieser Wenn-dann-Block immer bei Zahlen aufgerufen, die genau durch 30 teilbar sind, also 0, 30, 60, 120, aber auch 369129475394520. Um den Programmieraufwand abzukürzen, rufen wir für neue Gegner und neue Meteore dasselbe Unterprogramm auf. Wir nennen es hier „neuerstein“ und geben als Parameter ein Bild für das Sprite an. Im Dialog für die neue Methode (Menüpunkt „Bearbeiten > Neue Methode ...“) tragen wir bei Name „neuerstein“ und bei Parameter „p as pictu re“ für das Bild ein. Mit der Zeile „neuer stein feindbild“ rufen wir das Unterprogramm auf und übergeben als Parameter das Bild des Gegners, der auftauchen soll. Bis hierher sieht der Code im Event „Next Frame“ so aus:

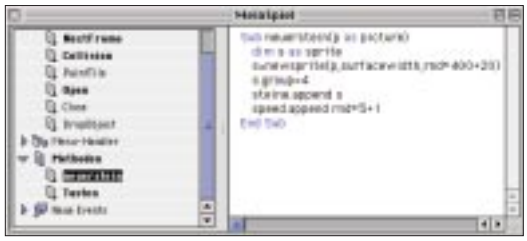
```
count=count+1
if count mod 30=0 then
        neuerstein feindbild
end if
if count mod 15=0 then
        neuerstein steinbild
end if
```

NEUE STEINE BRAUCHT DIE WELT

In der Methode „neuerstein“ erzeugen wir uns ein Sprite. Dafür brauchen wir die Variable „s“ vom Typ „Sprite“. Die Zeile „dims as sprite“ erledigt das für uns. Für das Sprite benutzen wir das Bild, das in der Variablen „p“ als Parameter übergeben wird. Wir positionieren alle Objekte zunächst bei der x-Koordinate „SurfaceWidth“, also ganz am rechten Rand noch außerhalb des sichtbaren Bereichs. Als y-Koordinate generieren wir einen Zufallswert. Die Funktion „rnd“ multipliziert mit der Maximalhöhe der Sprite-Umgebung erledigt das.

Wir schreiben „s=newsprite(p,surface width,rnd*400+20)“. Für die zweite Koordinate benutzen wir einen Maximalwert von 400 Pixeln, da wir wegen unseres Rahmens noch ein paar Pixel Abstand nach oben und unten halten müssen. Von den 480 Pixeln ziehen wir 20 für den oberen und 20 für den unteren Rand ab. Weitere 40 Pixel gehen uns für die Objekte selbst verloren, bleiben also insgesamt noch 400 Pixel übrig.

Mit der Zeile „s.group=4“ verhelfen wir den Objekten zu einem Kollisions-Event beim Zusammenstoß mit dem Raumschiff. Unser Sprite ist jetzt fertig, aber wir würden es gerne noch länger aufheben, um die Objekte später zu bewegen. Dazu speichern wir die Steine dauerhaft in einer Variablen, indem wir für sie ein „Array“ anlegen (große Meteore werden dabei genau so behandelt wie die Asteroiden). Wenn wir eine normale Variable anlegen, schreiben wir „dim stein as sprite“. Mehrere Steine bekommen wir mit „dim stein(0) as sprite“. Damit haben wir ein Variablenfeld (Array) mit einem Element, nämlich dem „nullten“. Mit „dim stein(10) as sprite“ legt man ein Array mit insgesamt elf Elementen (zehn plus das Element „0“) vom Typ Sprite an.



DIESE METHODE erzeugt Asteroiden und Meteore, die an zufälligen vertikalen Positionen am rechten Spielfeldrand erscheinen.

Wie aus der Mathematik (Analysis) gewohnt, greift man auf die einzelnen Elemente zu. Das fünfte Element erreichen wir mit „stein(5)“, das achte mit „stein(8)“ und so weiter, der Index steht einfach innerhalb der Klammern. Auch mehrdimensionale Felder sind möglich, „dim steine3D(5,10,3) as sprite“ erzeugt ein dreidimensionales Feld, das 150 Elemente enthält (5 x 10 x 3).

Für unsere Steine reicht aber ein einfaches eindimensionales Feld aus, das wir wie im folgenden Absatz beschrieben als neue Eigenschaft in unsere Spieleklasse einfügen.

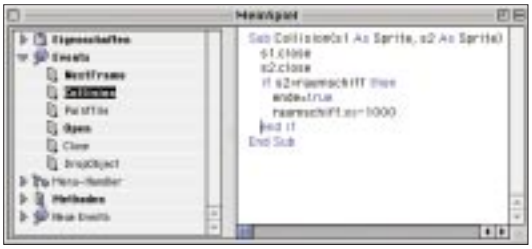
Im Dialog „Neue Eigenschaft“ (Menüpunkt „Bearbeiten > Neue Eigenschaft ...) geben wir „steine(0) as sprite“ ein. Damit legen wir zunächst das kleinstmögliche Feld an. Später im Programmcode erweitern wir das Array einfach, indem wir neue Sprites hinten anhängen. Im Unterprogramm erledigt das die Zeile „steine.append s“. Um unsere Steine zu bewegen, speichern wir zu jedem Stein auch seine Geschwindigkeit, die wir in Pixeln pro Bild (Pixel per Frame) messen. Passend dazu legen wir ein weiteres Variablen-Array mit der Deklaration „speed(0) as integer“ an. Dieses Feld füllen wir dann im Programm durch die Zeile „speed.append rnd*5+1“ mit einem Zufallswert zwischen eins und fünf.

FLIEGENDE STEINE

Nun geht es daran, unseren Steinen das Fliegen beizubringen. Dazu kehren wir in den „Nextframe“-Event unserer Spieleklasse zurück. Zunächst brauchen wir eine Zählvariable für eine Schleife. Dazu deklarieren wir „dim i as integer“. In der Schleife gehen wir die Liste der Steine vom Anfang bis zum Ende durch. Doch zu dem Zeitpunkt wissen wir noch nicht, aus wie vielen Elementen das Feld „Steine“ besteht, da wir zur Laufzeit immer wieder Elemente hinzufügen. Die Lösung ist die Funktion „Ubound“, die zu einem Feld den höchstmöglichen Index ermittelt. Unsere Schleife läuft also vom ersten (das nullte Element verwenden wir nicht) bis zum letzten Element, also „ubound(steine)“. Dafür ergänzen wir folgende Zeile „for i=1 to ubound(steine)“. Jetzt bewegen wir das Sprite, indem wir die

x-Koordinate, also die Eigenschaft „x“ um den Wert der Geschwindigkeit verringern. Wir schreiben dazu „steine(i).x=steine(i).x-speed(i)“. Diese Zeile ermittelt zunächst den Wert der Eigenschaft „x“, holt dann die Geschwindigkeit aus der Variablen „speed(i)“ und subtrahiert die Werte voneinander. Das Ergebnis landet dann wieder direkt in der Eigenschaft „x“, wodurch sich das Sprite um genau den Geschwindigkeitswert nach links bewegt.

Sobald der Stein am linken Bildrand angekommen ist, sollten wir ihn auflösen und damit etwas Arbeitsspeicher freigeben. Damit die Steine nicht schon innerhalb des sichtbaren Bereichs verschwinden, lösen wir sie erst auf, wenn sie die x-Koordinate -100 unterschreiten. Diese Koordinate befindet sich auf jeden Fall links außerhalb des sichtbaren Spielfeldes. Die Zeile „if steine(i).x< -100 then“ überprüft, ob die x-Koordinate kleiner als -100 wird, wenn ja, löschen wir den besagten Stein, in dem wir die Methode



EINE KOLLISIONSABFRAGE entscheidet darüber, ob und wann unser Raumschiff zerstört und damit das Spiel beendet ist.

„Close“ des Sprites aufrufen, „steine(i).close“ erledigt dies. Mit dem „Remove“-Befehl entfernen wir noch schnell das Sprite und dessen Geschwindigkeit aus den dazugehörigen Variablenfeldern und machen somit Platz für neue Sprites. Der Code dazu lautet „steine.remove i“ und „speed.remove i“. Jetzt fehlen uns nur noch ein „End If“ für den „If-Then“-Block und ein „Next“ für das Schleifenende der „For-Next“-Schleife. Wenn wir nun das Programm starten, erhalten wir den unumstößlichen Beweis: Steine können fliegen.

DER STEUERMANN SETZT DEN KURS

Als Nächstes steht die Raumschiffsteuerung auf dem Plan. Dazu wollen wir die Pfeiltasten auf der Tastatur verwenden. „Pfeil nach oben“ lässt das Raumschiff höher fliegen, „Pfeil nach unten“ steuert es dementsprechend tiefer. Wir müssen aber auch eine kleine Bremse einbauen, damit das Raumschiff gut steuerbar bleibt und nicht zu schnell auf die Kursänderungen reagiert.

Außerdem wollen wir verhindern, dass man weiterspielen kann, wenn das Spiel nach den Regeln zu Ende ist. Hierzu verwenden wir eine Variable „ende as boolean“, die innerhalb der Klasse anzeigt, ob das Spiel beendet ist. Wenn wir später ein neues Spiel starten wollen, setzen wir lediglich die Variable „ende“ auf „false“.

Doch zunächst zu der Tastenabfrage. Wir erzeugen eine neue, parameterlose Methode namens „Tasten“, die wir im „NextFrame“-Event aufrufen.

Nun benötigen wir zwei weitere Variablen, die wir als Eigenschaften deklarieren. Die Eigenschaft „lastticks as integer“ speichert den Zeitpunkt, an dem zuletzt eine Taste aufgerufen wurde. Dabei greifen wir auf den System-Timer im Mac-OS zurück, der jede sechzigstel Sekunde um eins erhöht wird. So läuft unser Programm unabhängiger, als wenn wir unseren eigenen Frame-Zähler benutzen würden, dessen Geschwindigkeit davon abhängt, wie schnell die Grafikkarte Bilder auf den Bildschirm zeichnet. Als lokale Variable benötigen wir noch „t as integer“, um eine weitere Zeit zwischenzuspeichern, damit wir die Systemfunktion nicht zu oft aufrufen müssen. Dadurch sparen wir erheblich viel Zeit.

In der ersten Programmzeile des Unterprogramms ermitteln wir den Stand des Systemzählers. Das geschieht einfach durch „t=ticks“. Der Test „if lastticks>t then“ prüft, ob das Programm noch vor unserem Zeitlimit für einen Tastendruck liegt. Dann beenden wir das Unterprogramm mit dem Befehl „return“. Solange „lastticks“ größer als „t“ ist, passiert nichts. Später brauchen wir also nur noch „lastticks“ mit einer passenden Pausenzeit zu besetzen.

ALLE KOMMANDOS ERHALTEN – DIE TASTATURABFRAGE

Kommen wir nun zu den eigentlichen Tasten. Bevor wir die Tastatur abfragen, müssen wir sicherstellen, dass der Spieler auch wirklich spielen darf und nicht bereits verloren hat. Die Zeile „if not ende then“ löst dieses Problem auf einfache Weise.

Wir fragen die Tastatur mit der in Realbasic integrierten Funktion „keytest(Tastaturcode as integer) as boolean“ ab. Den Tastaturcode erfährt man aus den Tabellen im Realbasic-Handbuch (gedruckt oder als PDF). Die Taste „Pfeil nach oben“ hört auf den Code „7E“, für die Taste „Pfeil nach unten“ benutzen wir „7D“. Dies sind hexadezimale Zahlen, die wir in Realbasic mit dem Prefix „&H“ angeben.

Wenn „KeyTest“ für eine der beiden Tasten ein „true“ meldet, bewegen wir das Raumschiff nach oben oder unten, indem wir die Eigenschaft „raumschiff.y“ um drei Zähler erhöhen oder verringern. Anschließend setzen wir „lastticks“ auf „t+2“. Mit einem „return“ beenden wir das Unterprogramm und verhindern so, dass man das Raumschiff gleichzeitig in beide Richtungen bewegt. Zum Schluss der beiden Tests setzen wir „lastticks“ auf „0“. In diesem Fall wurde keine Taste gedrückt. Den kompletten Code für die Tastaturabfrage finden Sie auf Seite 161 in der Abbildung rechts unten.

RAUMSCHIFF IM CRASH-TEST

Damit das Spiel irgendwann endet, müssen wir auf Kollisionen reagieren. Das tun wir im Event „Collision“. Wir erhalten dort zwei an der Kollision beteiligte Sprites als Parameter. Da sich hier das Raumschiff mit Objekten streitet, sollen beide bei einem Zusammenstoß zerstört werden. Das geht einfach mit „s1.close“ und „s2.close“. Sollte unser Raumschiff zerstört sein, beenden wir das Spiel. Wir prüfen das mit „if s2=raumschiff then“ und setzen die Variable „ende“ auf „true“. Dann verschieben wir das Raumschiff mit „raumschiff.x=-1000“ aus dem Bildschirm. Um das Spiel zu reaktivieren, wenn der Spieler zum Beispiel mehrere Leben hat, braucht man lediglich die Befehle „raumschiff.x=100“ und „ende=false“.

Unser Spiel funktioniert jetzt schon zufrieden stellend. Als Ergänzung kann man das Raumschiff wehrhafter machen und es mit Torpedos ausstatten, die die gegnerischen Objekte aus dem Weg räumen. Auch ein Punktezähler ist sinnvoll. Im Archiv zu diesem Artikel gibt es eine Version, die sowohl mit Torpedos, als auch einem Punktezähler ausgestattet ist. Es empfiehlt sich, die entsprechenden Codepassagen genau anzusehen. Der Phantasie für weitere Funktionen sind keine Grenzen gesetzt.

FAZIT

Dies war der fünfte und vorerst letzte Teil unserer Realbasic-Serie. Wer alle fünf Teile verfolgt und unsere Beispielprogramme nachprogrammiert hat, verfügt bereits über ein gutes Basiswissen in der Programmiersprache Realbasic. Für das Jahr 2001 planen wir einen weiteren Kurs, der sich an fortgeschrittene Programmierer wendet. Ideen und Vorschläge dazu sind uns willkommen. *cm*

ZUSÄTZLICHES MATERIAL

Macwelt Online

Auf unserer Homepage finden Sie unter www.macwelt.de/_magazin die vorbereiteten Objekte zu diesem Spiel, Quellcode sowie die bisherigen Folgen dieses Workshops im PDF-Format.

Serie Realbasic