SO PRÄSENTIERT SICH die Entwicklungsumgebung von Real Basic. Links sieht man die Werkzeugleiste.

DIE WERKZEUGLEISTE enthält alle optischen und funktionellen Elemente, die ein Mac-OS-Programm ausmachen. Man zieht ein Objekt einfach per Drag-and-drop in das Programmfenster.

MAC-SOFTWARE SELBST GESTRICKT

TROTZ DER STEIGENDEN Softwareflut für das Mac-OS findet man immer eine Lücke im Markt, und sei es nur ein kleines Hilfsprogramm, das genau das tut, was man gerade von seinem Mac will. Die Lösung: selbst programmieren

VON CHRISTIAN SCHMITZ

WEM C ZU KRYPTISCH und Pascal zu alt ist, der findet seit einiger Zeit in Real Basic eine ernst zu nehmende Alternative, die sich besonders unter Einsteigern wachsender Beliebtheit erfreut. Egal ob man 1000 Dateien um ihre DOS-Dateiendungen erleichtern oder endlich sein Lieblingskartenspiel auf den Mac bringen will, mit Real Basic sollen selbst blutige Anfänger einfach und minutenschnell eigene Programme realisieren können. Doch ganz so einfach geht es nicht. Ein paar Grundlagen zum Thema Programmieren muss man schon kennen. Genau die möchten wir Ihnen in dieser Folge unserer Serie liefern und Sie schrittweise zum ersten eigenen Programm führen.

DA BEKOMMT MAN REAL BASIC

Man erhält Real Basic entweder direkt vom Hersteller Real Software in den USA oder beim deutschen Distributor Application Systems Heidelberg (siehe Kästen „Das kostet Real Basic“ und „Real Basic im Internet“). Dort finden Sie auf jeden Fall eine deutsche und eine englische Version zum Herunterladen, die Sie 30 Tage lang kostenlos nutzen können. Sie liegt auch unserer Heft-CD bei. Zum Einsteigen reicht die abgespeckte LE-Version, die alles enthält, um Mac-Programme für 68K-Rechner und Power Macs zu kompilieren. Die Version 2 von Real Basic erweitert den Sprachumfang erheblich und bietet in der Pro-Variante auch einen Compiler für Windows (95/98/NT/2000) sowie Zugriff auf SQL-Datenbanken. In der neuen 3er Version, die sich zurzeit im Alpha-Stadium befindet, soll man seine Programme recht einfach für Mac-OS X erzeugen können. Somit ist Real Basic ideal für plattformübergreifende Projekte.

Für unseren Exkurs reicht die 30-Tage-Demoversion aus. Aber beachten Sie, dass diese wirklich nur 30 Tage auf jedem Mac läuft. Auch die damit erstellten Kompilate versagen nach 30 Tagen ihren Dienst.

SYSTEMANFORDERUNGEN

Für Real Basic selbst brauchen Sie lediglich einen Mac (ab 68020-Prozessor oder Power-PC) und etwa 5 MB freies RAM, wobei für größere Projekte auch deutlich mehr erforderlich sein kann. Der Hersteller empfiehlt System 7.6.1, aber Real Basic funktioniert auch unter System 7.0. Dort muss man lediglich einige Systemerweiterungen wie den „ThreadManager“ und den „DragManager“ nachträglich installieren. An Festplattenplatz benötigt man etwa 10 MB. Für weitere Beispiele und Hilfen können es auch 100 MB mehr werden. Zum Programmieren brauchen Sie nur die eigentliche Real-Basic-Applikation, in der die komplette Online-Hilfe bereits eingebaut ist. Zudem lässt sich der Befehls- und Funktionsumfang von Real Basic durch Plug-ins erweitern. Diese kommen in einen separaten Ordner.

DER ERSTE BLICK

Nach dem Start der deutschen Real-Basic-Version erscheint das grafische Benutzer-Interface der Entwicklungsumgebung (siehe Abbildung links). Die Werkzeugleiste mit allen Steuerelementen befindet sich auf der linken Seite, das Fenster für die Eigenschaften von Objekten rechts und in der Mitte vor dem Projektfenster das Hauptfenster des neuen Projekts. Real Basic speichert alle Fenster, Klassen, Module, das Menü und den gesamte Programmcode in der Projektdatei. Jedes neue Projekt hat immer eine fertige Menüleiste und ein leeres Fenster.

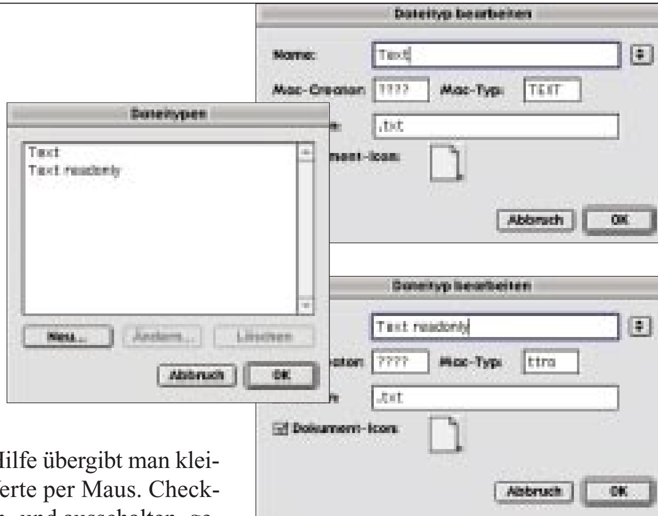
DIE WERKZEUGLEISTE

In Real Basic 2 bietet die Werkzeugleiste insgesamt 33 Steuerelemente, die sich alle per Drag-and-drop ins Fenster ziehen lassen. Alle Elemente sind in unserem Beispielprogramm „Steuerelemente“ einmal in Aktion zu sehen. Dabei bilden die Zierelemente die erste Gruppe. Am häufigsten benutzt man in der Regel „Statictext“, um Text in das eigene Programm einzufügen und um beispielsweise Beschriftungen oder kurze Erklärungen sichtbar zu machen.

Damit sich die mit Real Basic erzeugten Programme möglichst automatisch an jedes Erscheinungsbildthema anpassen, sollte man Placards und Separatoren für ein ansprechendes Layout benutzen und auf reine Linien, Rechtecke und Kreise verzichten.

Die zweite Gruppe stellt einfache Steuerelemente dar. Hier findet man Buttons (Schaltknöpfe), wobei Mac-OS meist einfache, eher schlichte Buttons verwendet. Wer mehr will, kann zu dem Zweck „Bevelbuttons“ benutzen. Diese lassen sich mit einem Bild oder einem Menü schmücken.

Die kleinen Pfeile sind bereits vom Kontrollfeld „Datum & Uhrzeit“ her bekannt, sie eignen sich wunderbar zur Zeit-



IN DIESEN DIALOGEN stellt man die Dateitypen an, die das Real-Basic-Programm laden oder speichern soll. Für unseren Textanzeiger sind das die Typen „TEXT“ und „ttro“.

eingabe. Mit ihrer Hilfe übergibt man kleine nummerische Werte per Maus. Checkboxen kann man ein- und ausschalten, genauso wie die Radio-Buttons, von denen sich allerdings immer nur einer auswählen lässt. Beide Button-Typen eignen sich hervorragend, um Optionen oder Voreinstellungen im Programm zu aktivieren.

Die „Disclosure Triangles“ verwendet man für alle Aufklappoptionen im Fenster, beispielsweise im „Kopieren“-Fenster des Finders. Als Nächstes folgt die Listbox, mit der man alle möglichen Arten von Listen erzeugt, darunter auch Dateilisten, wie man sie vom Finder her kennt. Icons und aufklappende Untermenüs lassen sich ebenfalls einbauen. Die Pop-up-Menüs sind der Platz sparende Ersatz für Radio-Buttons, wenn es darum geht, immer nur eine Option auszuwählen. Beim Kontextmenü handelt es sich um ein unsichtbares Steuerelement, das im laufenden Programm nicht auftaucht, obwohl es seinen Dienst im Hintergrund verrichtet. Damit es erscheint, muss man zusätzlich ein wenig Code programmieren. Wir werden deshalb in einer der nächsten Folgen ein Kontextmenü und dessen Programmierung vorstellen.

Die Scrollbalken kommen nur selten zum Einsatz, da Edit-Felder und Listboxen schon selbst jeweils einen Scrollbalken mitbringen. Schieberegler (Slider) eignen sich prima, um einen bestimmten Wert aus einem klar abgegrenzten Bereich einzustellen. Man benutzt sie häufig, um im Programm zum Beispiel Prozentwerte abzufragen.

Das „Editfield“ ist das umfangreichste Steuerelement, denn man erzeugt damit einfache Eingabezeilen sowie Passwort- und Texteingabefelder bis hin zu ganzen Textverarbeitungen ähnlich wie Simpletext. Mit einem Fortschrittsbalken („Progressbar“) oder den „Little Arrows“ zeigt man an, dass das Programm gerade arbeitet. Die Uhr dient dazu, in regelmäßigen Zeitabständen eine bestimmte Funktion aufzurufen, die beispielsweise eine Zeitanzeige im Programm neu zeichnet. Hiermit ist auch ein gewisser Grad an Multitasking möglich.

Die dritte Gruppe besteht aus den Gruppierungselementen. Man benötigt sie, um etwas Übersicht in einen Programmdialog zu bringen. Dazu fasst man Checkboxen, Radio-Buttons und andere Elemente optisch in Gruppen zusammen. Wenn der Platz im Fenster zu knapp wird, greift man auf die Karteikartenreiter, die so genannten „Tabpanels“, zurück.

AUF DEM WEG ZUR MULTIMEDIA-ANWENDUNG

Mit dem Element „Canvas“ kommen wir zur Multimedia-Ecke. Canvas ist ein Universalelement, denn damit kann man sämtliche Grafiken, Zeichnungen oder Pixel-Bilder anzeigen. So lassen sich auch leicht selbst definierte Steuerelemente erzeugen. Das „ImageWell“ dient dazu, ein Bild optisch versenkt darzustellen – man benutzt es sehr häufig, um zum Beispiel ein Logo im About-Dialog abzubilden.

Einfache 2D-Grafikspiele mit bewegten Animationen programmiert man mit der „Spritesurface“. Auch dies werden wir in einer kommenden Folge kennen lernen.

Das Movieplayer-Logo weist bereits auf die Funktion des nächsten Steuerelements hin. Damit kann man ähnlich wie im Quicktime-Player des Mac-OS alle Arten von Filmen wie etwa Quicktime-Movies, Quicktime-VR-Panoramen, Musik und vor allem auch MP3-Audiodateien abspielen. Damit steht dem MP3-Player nach eigenem Geschmack nichts mehr im Wege.

Durch den „Noteplayer“ wird man zum Musiker, denn damit lassen sich 128 digitalisierte Midi-Musikinstrumente aufrufen und spielen. Dies wird ebenfalls in einer der nächsten Folgen ein Thema sein.

Damit die genannten Elemente funktionieren, ist Quicktime zwingend erforderlich. Die meisten Möglichkeiten bietet Version 4.1, in der Regel reicht aber auch Version 3.0. Einige Funktionen, beispielsweise um Filme zu erstellen, sind allerdings nur mit der Quicktime-Pro-Lizenz möglich.

MIT REAL BASIC KOMMUNIZIEREN

Die letzte Gruppe stellt die Verbindung zur Außenwelt her. Sie beinhaltet eine TCP/IP-Anbindung, die so viele Möglichkeiten bereitstellt, dass sich inzwischen einige Real-Basic-Hobbyprogrammierer an einem eigenen Internet-Browser versuchen. Das Steuerelement „Seriell“ ist besonders für Bastler interessant, denn damit lassen sich alle möglichen externen Geräte über die serielle Schnittstelle des Mac steuern. Das letzte Element namens „Database Query“ dient in der Real-Basic-Pro-Version dazu, eine optionale Datenbank abzufragen.

WO IST DIE HILFE?

Basteln Sie ruhig ein wenig mit den verschiedenen Steuerelementen herum. Verändern Sie unser Beispielprogramm. Nur Mut, so lernen Sie am schnellsten. Wenn Sie Hilfe brauchen, nutzen Sie die Online-Referenz. Sie erreichen sie im Fenstermenü oder mit dem Tastenkürzel Befehl-1. Dort finden Sie alle Befehle und Objekte von Real Basic nach Themen und alphabetisch sortiert. Die Referenz, das Entwicklerhandbuch und ein Tutorial in Deutsch und Englisch kann man im Internet unter den im Kasten „Real Basic im Internet“ angegebenen URLs als PDF-Dateien herunterladen oder als Buch kaufen. Besonders empfehlenswert ist Matt Neuburgs „The Definitive Guide to Real Basic“, das in englischer Sprache vieles detaillierter und praxisnäher erläutert als die Originalhandbücher von Real Basic.

Ein gute Quelle für Beispiele und Quelltexte im Internet ist der Web-Ring (siehe Kasten „Real Basic im Internet“), der einen Index aller Real-Basic-Seiten darstellt. Für Fragen und Probleme zu Real Basic kann man sich immer an die Mailinglisten wenden. Eine deutschsprachige Liste ist ebenfalls vorhanden. Alternativ gibt es auch die Newsgroup comp.lang.basic.realbasic.

TEXTANZEIGER SELBST PROGRAMMIERT

Unser erstes eher einfaches Projekt ist ein Textanzeiger, das heißt ein Programm, das schlicht und einfach den Inhalt von Textdateien in einem skalierbaren Fenster zeigt.

DAS KOSTET REAL BASIC

Produkt	Preis
Real Basic LE Deutsch	DM 100, € 52, S 750, sfr 90
Real Basic 2 Standard Deutsch	DM 300, € 154, S 2250, sfr 265
Real Basic 2 Pro Deutsch	DM 700, € 358, S 5150, sfr 610
Real Basic Buch „The Definitive Guide to Real Basic“	DM 70, € 36, S 550, sfr 65

Info: Application System Heidelberg, Telefon (D) 0 62 21/30 00 02 Internet www.application-systems.de

Nach dem Start von Real Basic wird automatisch ein neues Projekt angelegt. Von nun an können Sie Ihre Programmierfortschritte zu jeder Zeit mit dem Menübefehl „Run“ im Debug-Menü (Befehl-R) testen. Aber speichern Sie das Projekt vorher. Es ist eine sinnvolle Angewohnheit, immer erst Befehl-S (Speichern) und danach Befehl-R zu drücken, denn es gibt kaum etwas Schlimmeres, als seine wertvolle Arbeit durch einen kleinen Tippfehler zu verlieren. Auch ein Real-Basic-Programm kann einmal abstürzen und den Rechner dadurch zu einem Neustart zwingen.

Per Drag-and-drop ziehen Sie ein Editfeld in das leere Hauptfenster und vergrößern es mit der Maus, so dass es das Fenster ganz füllt. Rechts im Eigenschaftenfenster lassen sich die Eigenschaften des ausgewählten Kontrollelements ändern. Klicken Sie auf eine freie Fläche im Fenster. Im Eigenschaftenfenster sollte rechts oben „Fenster1“ stehen. Aktivieren Sie unten im Fenster die Häkchen bei den Optionen „Grow-Icon“ und „Zoom-Icon“, damit Sie Ihr Fenster später im Programm skalieren können. Schauen Sie sich die Änderungen im Programm immer wieder an (Befehl-S und Befehl-R). Vergessen Sie nicht, bei „title“ einen Namen einzugeben oder wenigstens den Text „Namenlos“ zu löschen.

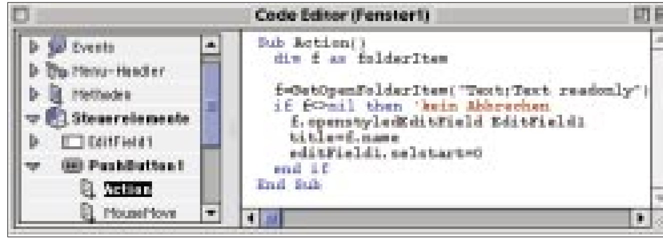
Jetzt klicken Sie auf das Editfeld im Fenster. Als Position des Editfeld empfehlen sich Fenster füllende Koordinaten, das wären: Left: -1, Top: 34, Width: 302, Height: 252. Sie werden sehen, dass das Editfeld das Fenster vollständig ausfüllt. Oben lassen Sie noch etwas Platz für ein paar Knöpfe.

Starten Sie nun Ihr Programm, werden Sie feststellen, dass das Editfeld nicht mitwächst, wenn Sie das Fenster vergrößern. Doch statt dies umständlich von Hand zu programmieren ist es vollkommen ausreichend, bei den Eigenschaften die Häkchen bei „LockLeft“, „LockTop“, „LockRight“ und „LockBottom“ zu aktivieren. Dann klappt es auch mit dem Vergrößern.

Weiter geht es mit den Eigenschaften: „Multiline“ sollte eingeschaltet sein, damit Ihre Texte nicht in eine einzige Zeile gequetscht werden. Nebenbei erscheint nun auch ein Scrollbalken. Ein weiteres Häkchen bei „Styled“ schaltet Stilelemente wie „Fett“ oder „Unterstrichen“ im Text frei.

REAL BASIC IM INTERNET

Real Basic	www.realbasic.com
Real Software	www.realsoftware.com
ASH	www.application-systems.de/realbasic
Real News	www.swssoftware.com/realnews
Web-Ring	www.webring.org/cgibin/-webring?ring=xbasic;List



MIT GANZEN SIEBEN Zeilen Code erzeugen wir einen voll funktionsfähigen Textanzeiger. Der rote Text ist lediglich ein Kommentar und hat keinen Einfluss auf das Programm.

„UseFocusRing“ brauchen Sie, wenn Sie mehr als ein Eingabefeld in einem Fenster benutzen möchten. Da dies in unserem Beispiel nicht der Fall ist, können wir die Option getrost ausschalten.

Da es sich bei unserem Programm lediglich um einen Anzeiger handelt, sollte man den Text ganz unten auf „ReadOnly“ stellen. Damit lässt sich verhindern, dass man Text eingeben oder ändern kann.

Jetzt fehlt nur noch die Funktion zum Laden von Textdateien. Hierfür ziehen Sie aus der Toolbar einen Button in den freien Bereich oben im Fenster und nennen ihn „Öffnen...“. Im Menü wählen Sie unter Bearbeiten den Dialog „Dateitypen...“ und erzeugen einen neuen Dateityp namens „Text“ mit dem Creator-Code „????“ und dem Typ „TEXT“. Für die beliebten Read-Only-Texte legen Sie noch einen zweiten Dateityp an: „Text Read Only“ mit Dateityp „ttro“ und ebenfalls Creator-Code „????“.

Falls Sie später auch eine Windows-Version kompilieren wollen, können Sie dreistellige Dateinamenserweiterungen angeben. Ein Häkchen bei „Dokument-Icon“ ermöglicht es, Dokumente per Drag-and-drop auf das Programm-Icon zu öffnen. Hierzu ist jedoch ein wenig Programmcode notwendig, den wir in dieser Version noch nicht einbauen. Wieder zurück im Layout bringt uns ein Doppelklick auf den Button zum Code-Editor. Kleiner Tipp: Mit der Tastenkombination „Wahltaste-Tabulatortaste“ springen Sie schnell zwischen Fenster und Code-Editor hin und zurück.

Bis hierher haben wir noch keine einzige Zeile Code geschrieben. Das ändert sich aber jetzt, denn im Code-Editor findet die eigentliche Programmierarbeit statt. Bereits vorausgewählt ist der „Action Event“ des Buttons. An der Stelle tippen wir den Programmtext wie in unserer Abbildung oben auf dieser Seite ein.

ERSTER SELBST GESCHRIEBENER CODE

Wer die Syntax der Sprache Basic kennt, für den dürften die meisten Befehle nicht neu sein. Für die weniger versierten Programmierer wollen wir die Kommandos und Steuerbefehle der Reihe nach erklären.

Der DIM-Befehl deklariert eine neue Variable. Dabei verwendet man folgendes Muster: „DIM name AS typ“. Als Typ nehmen wir „FolderItem“. Ein Objekt der Klasse „FolderItem“ dient dazu, eine Datei zu repräsentieren. Die Funktion „GetOpenFolderItem“ zeigt eine Öffnen-Dialogbox an, die alle Dateien auflistet, die unserem Dateityp „Text“ entsprechen. Schließt man nun diesen Öffnen-Dialog mit „OK“, erhält man eine Referenz auf die ausgewählte Datei als „FolderItem“-Objekt und speichert diese in der Variablen „f“. Klickt der Benutzer auf „Abbrechen“, bekommt er keine Referenz, sondern „nil“ zurück. „nil“ ist ein Wert, der anzeigt, dass diese Variable zu keinem Objekt gehört, also sozusagen „leer“ ist. Damit lässt sich überprüfen, ob man „OK“ oder „Abbruch“ gedrückt hat. War es der OK-Knopf, weist man das „FolderItem“ an, sich als „StyledEditfield“ zu öffnen und den Text in unserem Editfeld anzuzeigen. Die Zeile „title=f.name“ kopiert noch schnell den Namen der Datei aus dem „FolderItem“ in die Eigenschaft „title“ des Fensters, was zur Folge hat, dass dieser in der Titelleiste angezeigt wird. Zum Abschluss, vor dem „End If“ setzt man den Cursor auf den Anfang des Textes. Dadurch gewährleistet man, dass der Text an seinem Anfang erscheint.

WENIG CODE MIT GROSSER WIRKUNG

Die erste Lektion in Real Basic ist mit diesem Schritt beendet. Sie haben gerade einmal sieben Zeilen Code geschrieben und dennoch ein vollständiges Programm erzeugt, dass man kompilieren kann. Spielen Sie ruhig ein wenig mit dem Quellcode herum und ändern Sie das Programm wie es Ihren Vorstellungen entspricht. Die vollständigen Projektdateien liegen im Internet unter der Adresse www.macwelt.de zum Herunterladen bereit. In der nächsten Folge werden wir anhand eines kleinen Taschenrechners sehen, wie man komplexere Zusammenhänge programmiert.

FAZIT

Real Basic ist eine mächtige und dennoch sehr leicht zu erlernende Programmiersprache, mit der sich auch Neulinge schnell ein Erfolgserlebnis verschaffen können. In den nächsten Folgen werden wir uns komplexeren Aufgaben zuwenden. *cm*

Serie Real Basic

1 Einführung .... **Heft 9/2000**
2 Komplexe Oberflächen .... **Heft 10/2000**
3 Grafik total .... **Heft 11/2000**
4 Quicktime-Programmierung .... **Heft 12/2000**

TASCHENRECHNER IM EIGENBAU

NACHDEM WIR IN DER LETZTEN FOLGE schon einen ersten Blick auf Realbasic geworfen und dabei gelernt haben, aus sieben Zeilen Code einen kompletten Textanzeiger zu basteln, wollen wir uns in dieser Folge komplexeren Anwendungen widmen. Unser Ziel ist, den mageren Taschenrechner des Mac-OS durch leistungsfähige Eigenentwicklungen zu ergänzen

VON CHRISTIAN SCHMITZ

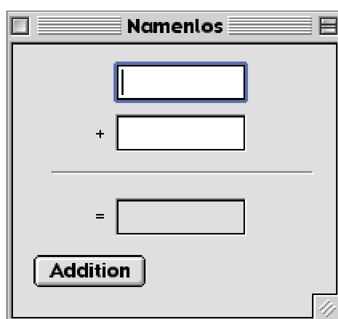
WIR STARTEN mit einem Additionsprogramm, das schlicht und einfach zwei Zahlen addiert und uns die Summe im Klartext in einem Dialog ausgibt.

Nach dem Doppelklick auf Realbasic erscheint wie immer ein neues, leeres Projekt, und Sie sehen ein leeres Fenster mit dem Titel „Namenlos" vor sich. Ziehen Sie jetzt per Drag-and-drop von der Werkzeugleiste nacheinander drei Edit-Felder (Eingabefeld, Sie finden es in der Werkzeugleiste über der „Stoppuhr") in das Fenster. Zwei der drei Edit-Felder dienen uns als Eingabefelder für die Zahlen. Das Dritte zeigt nachher das Ergebnis an, ist also ein reines Ausgabefeld, in das man nichts eintippen kann. Mit ein paar „Statictexten" (links oben in der Werkzeugleiste das Kontrollelement mit dem großen „A") verschönern wir das Fenster noch um ein „+" und ein „=". Ein Separator, also eine Linie, die je nach dem im Mac-OS ausgewählten Erscheinungsbild anders aussieht, dient als dicker Trennstrich. Um Ausgabe- und Eingabefelder zu unterscheiden, ändern wir die Hintergrundfarbe des Ausgabefeldes (Eigenschaft „Backcolor" in der Palette „Eigenschaften" rechts) mit dem Farbpicker. Damit niemand das Ergebnis nachträglich ändern kann, aktivieren wir die Checkbox für „Readonly" (nur lesen). Als Letztes ergänzen wir noch einen „Button" (in der Werkzeugleiste mit „OK" beschriftet), dem wir im Eigenschaftenfenster den Namen (Caption) „Addition" geben.

Nach dem Speichern (Befehl-S) und dem Starten (Befehl-R) sollte das Fenster dann so aussehen wie in der Abbildung links unten auf dieser Seite.

Jetzt kann man noch überall herumklicken, ohne dass etwas passiert. Sie ahnen sicher, dass da noch ein wenig Programmcode fehlt. Also machen Sie bitte im Realbasic-Editor einen Doppelklick auf den „Addition"-Button. Dadurch gelangen Sie automatisch in den Code-Editor. Dort wählen Sie den „Action Event" des Buttons aus und geben rechts Ihren Code ein.

Wenn der Benutzer auf den Knopf „Addition" klickt, wird der Action-Event zur Laufzeit aufgerufen. Dann soll das Programm die zwei Edit-Felder oben auslesen, die Zahlen addieren und das Ergebnis in das untere Feld eintragen und anzeigen.



VIER OBJEKTE REICHEN AUS, um einen kleinen Rechner für die Addition von Zahlen zu programmieren.

KLEINER EXKURS IN DATENTYPEN

Dazu müssen Sie zunächst wissen, was Datentypen sind. Es gibt Variablen, die wie Schubladen mit Namen versehen sind und bestimmte Werte speichern können. Beim Computer gibt es jedoch für jede Art von Werten unterschiedliche Datentypen. Für Zahlen existieren je nach Anzahl der (Nachkomma-) Stellen unterschiedliche Variablen. Die häufigste ist „Integer", die wir für alle ganzen Zahlen von -2147483648 bis +2147483647 wählen (eine Integer-Zahl ist intern also eine 32-Bit-Zahl). Integer-Zahlen reichen jedoch nicht immer aus, und so gibt es den Datentyp „Double", der jegliche Zahlen speichern kann. „Double" reicht zwar für mehrere hundert Stellen große Zahlen, besitzt aber nur eine Genauigkeit von zirka 20 Stellen. Außerdem lassen sich periodische Brüche wie zum Beispiel 1/3 nur ungefähr, also fehlerbehaftet, erfassen.

Der Datentyp „Single" hat eine geringere Zahlenmenge und braucht weniger Speicher, ist aber auf dem Power-PC-Prozessor nicht direkt implementiert und wird jedes Mal zur Berechnung in eine Double-Variable umgewandelt, was ihn langsamer macht als die doppelt so großen Doubles. Daher empfiehlt es sich, nur Integer- oder Double-Variablen zu benutzen.

Um Texte wie beispielsweise „Hallo" zu speichern, gibt es den Datentyp „String". Er dient als Variable für jeglichen Text bis zu einer Größe von (theoretisch) 2 GB. Man beachte den feinen Unterschied zwischen 5 und "5", denn die 5 alleine ist ein Integer, und die "5" in Anführungszeichen ist ein Text, also ein String. Strings kann man einfach mit der Addition „+" aneinander kleben. Der Befehl: me.Text = "Hallo" + " Leute!" im Open-Event eines Edit-Feldes zeigt das Aneinanderketten von Texten. Und ein me.text="5"+"5" ergibt eine "55" und keine 10! Dabei wird der fertige Text in die Variable „Text" des Edit-Feldes gespeichert, die den aktuellen Inhalt darstellt.

WAS SIND EVENTS?

Ein Event ist für den Computer ein beliebiges Ereignis. Wenn der Benutzer eine Taste drückt oder die Maus bewegt und irgendwo hinklickt, bekommt das aktive Programm vom Mac-OS einen Event geschickt, der es über die Art des Ereignisses informiert. In Realbasic werden dadurch so genannte „Event-Handler" aufgerufen. Letztere findet man in jedem Objekt, also in Fenstern, Buttons und Klassen. Man kann in die Event-Handler eines Objekts, zum Beispiel den „Action-Event" des Push-Buttons etwas Code einfügen und damit auf das Ereignis reagieren. Ein Programm wird so lange von diesen Ereignissen gesteuert, bis es zum Ereignis „Quit" kommt, dann beendet es sich selbst, und man landet im Finder.

RECHNEN MIT ZAHLEN

Variablen vom Typ „Zahl" lassen sich für verschiedene mathematische Methoden benutzen. Nach einem „Dim a,b,c as double" (drei Variablen anlegen mit dem Namen a, b und c, alle vom Typ Double, also für Zahlen mit Nachkommastellen) kann man inner-

halb des Action-Event von einem Button rechnen. Möglich sind Konstruktionen wie „a=b+c“ oder „a=b*c“. Was es an weiteren möglichen Operationen gibt, können Sie dem Kasten „Mathematik in Realbasic“ auf dieser Seite links unten entnehmen.

ZAHLEN UND STRINGS UMWANDELN

Die Texte in Edit-Feldern sind Strings, aber gerechnet wird mit Double-Variablen. Wollen Sie eine Double-Variable aus dem Text eines Edit-Feldes erzeugen, müssen Sie den Variablentyp umwandeln. Dazu gibt es mehrere Funktionen. Für Zahlen im US-Format, also etwa „1,456.345“ für 1456,345, können Sie die Funktionen „Val()“ und „Str()“ nehmen. Val interpretiert einen String und versucht, daraus eine Zahl zu erzeugen, ignoriert aber alles, was nicht nach Zahl aussieht. Dazu einige Beispiele: Aus "1,234.5" wird 1234.5, aus "1.45" wird 1.45, und aus "1.5hallo" wird 1.5. Die Funktion „Str“ geht den umgekehrten Weg und macht aus einer Zahl einen String im US-Zahlenformat. Auch hierzu einige Beispiele: Aus 1.5 wird "1.5", aus 0.123456 wird "0.123456" und aus 123456789 wird "1.234568e+8". Beide Funktionen können mit der wissenschaftlichen Notation umgehen, wie 1e5 oder 8.643e-56.

Da man hier zu Lande das deutsche Zahlenformat bevorzugt, gibt es dafür die Funktionen „Cdbl()“ und „Format()“. Die Funktion Cdbl() interpretiert einen String und gibt die passende Zahl zurück, wobei sich Cdbl() dabei an die Vorgaben aus dem Kontrollfeld „Zahlenformat“ hält. Ansonsten entspricht diese Funktion der Funktion Val(). Die Funktion Format() dient zum Formatieren einer Zahl. Einige Beispiele: Format(-1.5,"0.0") ergibt "1.5", aus Format (-1.345,"-00.00") wird "-01.35", und Format(123456789,"####,###,##0.0") liefert "123.456.789.0" zurück.

Dabei richtet sich die Formatierung nach einem Formatierungs-String (im oben angeführten Beispiel sind das die vielen Doppelkreuze), wobei Punkte durch Dezimaltrenner und Kommas durch Tausendertrenner ersetzt werden. Eine Null (0) wird durch eine Ziffer ersetzt oder durch ein "0", wenn keine da ist. Aus "#" wird eine Ziffer oder gar nichts, also ein Leer-String. Es gibt noch unzählige weitere Optionen, die den Rahmen dieses Artikels sprengen würden. Hier lohnt sich ein Blick ins Realbasic-Handbuch oder in die Online-Hilfe.

MATHEMATIK IN REALBASIC

Folgende Operatoren gelten in Realbasic für Variablen mit den Typen Integer, Single und Double:

+	Addition
–	Subtraktion
*	Multiplikation
/	Division (mit Nachkommastellen)
\	Division (mit Ganzzahlergebnis)
mod	Division (Rest von \)

UND DER MAC RECHNET...

Jetzt kommt etwas Code in unseren Taschenrechner. Wir sind immer noch im Code-Editor, und der Action-Event ist ausgewählt. Dort deklarieren wir zuerst zwei Variablen für unsere Zahlen: „Dim Zahl1, Zahl2 as Double“. Wie gehabt reserviert „Dim“ ein Stück Arbeitsspeicher und gibt ihm den genannten Namen „Zahl1“ beziehungsweise „Zahl2“ mit dem Typ „Double“.

DIESE SIEBEN ZEILEN Realbasic-Code lesen die Edit-Felder aus, berechnen das Ergebnis und schreiben es zurück in das Ausgabefeld.

Danach haben wir zwei Variablen („Zahl1“ und „Zahl2“) für Zahlen mit Nachkommastellen. Eine weitere Double-Variable für das Ergebnis wird in der nächsten Codezeile mit „Dim Ergebnis As Double“ eingeführt. Man kann auch alle drei Variablen in einer einzigen Zeile deklarieren, aber so wird unser Code übersichtlicher.

Nachdem wir jetzt alle erforderlichen Variablen deklariert haben, fangen wir an, sie zu füllen. Dazu benutzen wir zwei Codezeilen mit: „Zahl1=cdbl(editfield1.text)“ und in der nächsten Zeile: „Zahl2=cdbl(editfield2.text)“. Was in diesen Zeilen passiert, nennt man eine Zuweisung. Der Variablen links vom „=“ wird ein Wert zugewiesen, der rechts ermittelt wird. Die Variable links ist hier einmal „Zahl1“ und einmal „Zahl2“. Um den String einer Double-Variable zuzuweisen, wandeln wir ihn mit der Funktion Cdbl() um. Anschließend wird der Text interpretiert und die passende Zahl in unsere Variable gespeichert.

In der Zeile „Ergebnis=Zahl1+Zahl2“ findet die eigentliche Addition statt, indem der Variable „Ergebnis“ der Wert der Summe von Zahl1 und Zahl2 zugewiesen wird. Wenn Sie diese Zeile später ändern, können Sie leicht aus dem Additionsprogramm ein Programm für Subtraktion, Multiplikation oder Division mit und ohne Rest machen.

Zum Schluss speichern wir das Ergebnis mit der Zeile „Editfield3.text=Format (Ergebnis,"0.0")“ in das Edit-Feld. In diesem Fall müssen wir aus der Zahl, die in der Variable Ergebnis steht, einen formatierten String machen, der dann schließlich im dritten Edit-Feld ausgegeben wird. Mit dem Ausdruck "0.0" geben wir an, dass wir uns die Ausgabe in mindestens einer Vor- und einer Nachkommastelle wünschen.

WEITERE IDEEN

Durch weitere Knöpfe lassen sich relativ einfach Subtraktion, Multiplikation und Division hinzufügen. Das finden Sie auch in unseren Beispielen auf *Macwelt Online* (www.macwelt.de/_magazin). Das Beispiel dort hat zusätzlich einen Reset-Knopf, um alle Edit-Felder auf einmal zu löschen. Hierbei wird jeweils das Textobjekt des Edit-Feldes auf "" (Leer-String) gesetzt. Also EditField1.text="", EditField2.text="" und EditField3.text="".

Natürlich kann man dieses Programm auch dahin gehend erweitern, dass es für andere Aufgaben taugt. Nehmen Sie so viele Edit-Felder zur Eingabe wie nötig, einen Knopf „Rechnen“ und ein paar Edit-Felder zur Ausgabe. Jetzt können Sie ähnlich wie oben eine passende Rechnungszeile erzeugen, wie im Beispiel „Mehrwertsteuer“ (auch auf *Macwelt Online* zu finden).

MENÜS IN REALBASIC

Jede Mac-OS-Applikation bietet eine Menüleiste, auch wenn diese nur den Eintrag „Über dieses Programm...“ enthält, mit dem man eine so genannte Aboutbox aufruft.

Sie haben das Additionsprogramm noch in Realbasic vor sich offen? Dann klicken Sie auf den Eintrag „Menü“ im Projektfenster (man kann das Projektfenster mit Befehl-0 aufrufen, wenn es nicht zu sehen sein sollte). Nun öffnet sich ein Fenster mit dem Menüeditor. Wie Sie sehen, bietet jedes Realbasic-Projekt von Anfang an eine Menüleiste, die bereits voll funktionstüchtige Menüeinträge für „Quit“, „Kopieren“, „Ein-

JEDES NEUE REALBASIC-PROJEKT besitzt bereits eine Menüleiste mit Einträgen für „Ablage” und „Bearbeiten”.

fügen“, „Ausschneiden“ und „Löschen“ enthält. „Undo“ ist vorhanden, funktioniert jedoch nicht ohne zusätzlichen Code.

RECHNEN MIT DEM MENÜ

Unser Programm sollte auch über den Menüpunkt „Rechnen“ arbeiten und auf das Tastenkürzel „Befehl-R“ reagieren.

Dazu klicken Sie aufs Ablagemenü im Menüeditor und dann auf den leeren Eintrag unter dem Beenden-Befehl. Nun geben Sie in der Eigenschaftenpalette oben beim leeren Feld für Text den Text „Rechnen“ ein und drücken Enter. Damit haben Sie den Menüeintrag „Ablage/Rechnen“ erstellt. Nie fehlen sollte ein leicht zu merkendes Tastenkürzel, in unserem Beispiel also Befehl-R. Geben Sie in der Palette bei „CommandKey“ ein „R“ ein. Auch andere Kürzel, die Sie dem Kasten „Tastenkürzel in Realbasic“ entnehmen können, sind möglich. Wenn Sie möchten, geben Sie noch einen Erklärungstext bei „Balloonhelp“ ein. Dieser erscheint dann automatisch, wenn man mit der Maus bei eingeschalteten Erklärungen (findet man im „Hilfe“-Menü) über den Menüeintrag fährt.

EIN BISSCHEN CODE MUSS SEIN

Wir wechseln zum Dialogfenster unseres Programms. Mit Wahl- und Tabulatortaste öffnet sich der Code-Editor des Fensters. Wählen Sie in der Realbasic-Menüleiste unter „Bearbeiten“ den Eintrag „Neuer Menü-Handler“ aus und im Dialog „Ablage/Rechnen“. Hier geben wir als Programmcode lediglich eine Zeile ein: „pushbutton1.push“. Dieser Code simuliert den Mausklick auf unseren Additions-Button, löst also den Rechenbefehl aus. Sollte Sie das Drücken des Push-Buttons optisch stören, können Sie seinen kompletten Programmcode in den Menü-Handler kopieren. Dann rechnet das Programm, ohne den Button zu betätigen. Alternativ lässt sich eine „Methode“ (Unterprogramm) schreiben und aufrufen. Wie das geht, zeigen wir anhand weiterer Beispielprogramme im Internet unter www.macwelt.de/_magazin.

Wenn Sie das Programm nun starten, fällt auf, dass sich der neue Menüeintrag „Rechnen“ nicht auswählen lässt. Er ist grau dargestellt, also „disabled“. Es fehlt noch etwas Code, mit dem man Menüeinträge freischaltet. In den Events vom Dialogfenster (im Code-Editor) gibt es den Event „Enable MenuItems“, den Realbasic immer dann aufruft, wenn Sie in die Menüleiste klicken. Dann müssen Sie im Programmcode entscheiden, welche Menüeinträge Sie aktivieren, denn per Default ist zunächst einmal alles „disabled“. Für unseren Menüeintrag ergänzen wir: „AblageRechnen.enabled= true“. Wir schalten also den „Enabled“-Schalter des Menüeintrags „AblageRechnen“ ein. Damit ist der Eintrag auswählbar.

PLATZ ZUM SPIELEN: ABOUTBOX.

Was noch fehlt, ist die Verewigung. In einem Dialog im Apple-Menü finden sich in der Regel die Namen der Programmierer und die Programmversion. Außerdem lässt sich die Aboutbox mit einem Bild verschönern.

Dazu gehen Sie wieder in den Menüeditor und ergänzen wie oben einen Eintrag für „Über das Additionsprogramm...“. Es ist nicht üblich, dafür ein Kürzel einzubauen. Stattdessen sollten Sie an den Text noch drei Punkte (drei Mal „.“ oder einmal „...“, Wahltaste-Punkt) anfügen, denn bei solchen Punkten sieht der Benutzer sofort, dass sich eine neue Dialogbox öffnen wird. Benennen Sie den Menüeintrag auf jeden Fall in „AppleAbout“ um, damit er nicht den Default-Namen „AppleÜberdasAddierprogramm...“ behält. Deutsche Umlaute und die drei Punkte „...“ sind für Objektnamen nicht zugelassen und führen zu Fehlermeldungen.

IN ZWEI MINUTEN ZUR ABOUTBOX

Für eine ansprechende Aboutbox erzeugen wir ein neues Fenster (Menüeintrag „Neues Fenster“ im Ablagemenü von Realbasic). Geben Sie ihm einen aussagekräftigen Titel und füllen Sie es mit Objekten Ihrer Wahl. Es sollte ein paar Textobjekte (Statictext) bekommen und mindestens einen Push-Button, der einen „Close“-Befehl im Action-Event enthält, um das Fenster zu schließen.

SO KÖNNTE eine einfache Aboutbox mit integriertem Bild aussehen.

Per Drag-and-drop können Sie ein beliebiges Bild (PICT, JPEG, GIF, usw.) in Ihr Projektfenster ziehen. Dazu legen Sie im About-Fenster ein Canvas an (Sie finden es in der Werkzeugleiste. Es ist das Bildobjekt mit dem stilisierten „Sonnenuntergang“.) und wählen als „Backdrop“ für das Canvas das Bild aus. Für den Schließen-Button empfiehlt es sich, die Eigenschaften „default“ und „cancel“ zu aktivieren, damit man den Dialog auch per Zeilenschalter (Return-Taste) und Escape-Taste verlassen kann.

Jetzt müssen wir nur noch den Menüeintrag freischalten, damit er anwählbar ist. Dazu ergänzen wir im „EnableMenuItems“-Event die Codezeile „AppleAbout.enabled=true“ und legen einen Menü-Handler für „AppleAbout“ an, den wir mit dem Code „dialog1.showmodal“ füllen. Durch diesen Befehl zeigt Realbasic unsere Aboutbox auf dem Bildschirm an.

Noch ein kurzer Tipp: Sie müssen das laufende Additionsprogramm nicht immer mit „Befehl-Q“ beenden, um Änderungen im Quellcode oder in der Optik vorzunehmen. Ein Klick auf ein beliebiges Realbasic-Fenster im Hintergrund genügt, um das laufende Programm zu unterbrechen. Es ist dann allerdings noch nicht beendet. Erst durch „Befehl-K“ (Kill) wird es dauerhaft angehalten und aus dem Speicher entfernt.

FAZIT

Ähnlich wie im letzten Teil unserer Realbasic-Serie genügen uns einige wenige Zeilen Code, um einen funktionsfähigen Taschenrechner inklusive Menüsteuerung zu programmieren. Doch nur Text und Zahlen zu bearbeiten ist langweilig. In der nächsten Folge wird es deshalb bunt und grafisch. Wir basteln uns kleine Grafikprogramme, um die Leistung der Grafikkarte im 2D-Bereich zu messen und lernen außer den Grafikbefehlen auch, wie man Zeitmessungen in Realbasic programmiert. *cm*

TASTENKÜRZEL IN REALBASIC

Tasten	Kürzel
X oder ⌘ + X	„X“ oder „command-X“
⇧ + X	„shift-X“
⌥ + X	„option-X“
ctrl + X	„control-X“
⇧ + ⌥ + X	„shift-option-X“
ctrl + ⇧ + ⌥ + X	„control-shift-option-X“

Im Realbasic-Menüeditor kann man einzelnen Menüpunkten ein bestimmtes Tastenkürzel beziehungsweise eine Tastenkombination zuweisen. Dabei tauchen Bezeichnungen wie command (Befehl), shift (Umschalt), option (Wahl) und control (Control) auf, die man exakt so in das Feld „commandkey“ eintippt.

Bedenken Sie außerdem, dass unter Microsoft Windows die Tastenkombinationen mit der Umschalt-, Wahl- und Control-Taste nicht funktionieren, denn dort ist „Control“ gleich „Befehl“, und die Wahltaste ist für die Navigation in der Windows-Menüleiste reserviert.

Stören Sie sich nicht daran, wenn diese Kürzel im Menüeditor seltsam aussehen – sie funktionieren dennoch. Real Software wird eigenen Aussagen zufolge den Menüeditor noch verbessern.

Serie Realbasic

1 Einführung .... Heft 9/2000
2 Taschenrechner im Eigenbau .... Heft 10/2000
3 Grafik total .... Heft 11/2000
4 Quicktime-Programmierung .... Heft 12/2000

GRAFIKPRÜFUNG

NACH UNSEREM TEXTANZEIGER und dem Taschenrechner in den letzten Folgen tauchen wir heute mit einem Programm zum Testen von Grafikkarten in die Tiefen von Realbasic ein. Dabei wollen wir viele Grafikfunktionen ausprobieren, diese in einem Fenster anzeigen lassen und jeweils die Zeit messen

VON CHRISTIAN SCHMITZ

WIE IMMER BEGINNEN WIR mit einem neuen Realbasic-Projekt. Dem Standardfenster, das wir „Ergebnisfenster" nennen („Name"-Eigenschaft in der Eigenschaftenpalette), geben wir den Titel „Ergebnis".

Weiter geht es mit einem zweiten Fenster (Menü „Ablage > Neues Fenster"), das wir „Testfenster" nennen und das den Titel „Bitte warten… Tests laufen." bekommt. Später sollen in diesem Fenster alle Grafiktests sichtbar ablaufen. Deshalb vergrößern wir das Fenster auf eine feste Größe von 600 x 400 Pixel. Nur wenn das Fenster auf allen Testkonfigurationen stets in konstanter Größe erscheint, kann man die Ergebnisse direkt miteinander vergleichen.

Die Eigenschaften „CloseBox", „Grow Icon" und „ZoomIcon" schalten wir aus, denn unser Testfenster darf nicht vergrößerbar sein. Per Doppelklick auf das leere Fenster oder mit der Tastenkombination Wahl-Tabulator kommt man in den Code-Editor. In den „OpenEvent" des Fensters fügen wir einige Zeilen Code ein, um es auf dem Bildschirm zu zentrieren und auf ein wenig Abstand zur Menüleiste zu halten. In der Zeile „left=(screen(0).width-width)/2" wird die Breite des Hauptbildschirms (der mit Menüleiste; screen(1) ist der zweite Bildschirm, der keine Menüleiste hat) ermittelt. Davon ziehen wir die Fensterbreite ab und teilen das Ganze durch zwei, denn der Abstand links und rechts soll gleich groß sein. Mit der zweiten Zeile „top=50" hält unser Fenster einen Abstand von 40 Pixeln zum oberen Ende des Bildschirms. Dabei sollte man darauf achten, dass das Fenster nicht von der Menüleiste, der Kontrollleiste oder anderen Tools wie „Dragthing" überlagert wird, denn das macht die Messung ungenau.

In das Testfenster setzen wir per Drag-and-drop ein Canvas (findet man in der Werkzeugpalette über dem Movieplayer), das das Fenster komplett ausfüllen und später die Grafiktests anzeigen soll.

Dafür bekommt es den Namen „Ausgabe" und die Maße „Left=0", „Top=0", „Width=600" sowie „Height=400".

Zurück zum Ergebnisfenster. Dort fügen wir unten rechts einen Button ein und nennen ihn „Start". Die Eigenschaft „Default" setzen wir auf „true" (Häkchen in der Checkbox), damit man den Button per Eingabetaste betätigen kann. In den Action Event des Buttons (Doppelklick auf den Button) schreiben wir „testfenster.show". Startet man jetzt das Programm und drückt auf den Button, sollte das Testfenster zentriert erscheinen. Zurück kommt man nur mit einem Klick in ein Realbasic-Fenster (Code-Editor oder Projektfenster), denn in unserem Programm wird das Testfenster bis-

IM ERGEBNISFENSTER listet das Benchmark-Programm mittels 18 Statictext-Kontrollelementen die neun Testergebnisse auf.

lang an keiner Stelle geschlossen. Dies ist ein häufiges Problem in Realbasic, da dort alle neuen Fenster zuerst einmal modale Dialoge sind. Für unser Projekt ist das allerdings weniger wichtig, weil das Testfenster ja keine Bedienungselemente enthält. Wenn man jedoch in den Eigenschaften unter „Frame" von „1 – Moveable Modal" auf „0 – Document Window" umschaltet, lässt sich das Programm auch bei geöffnetem Testfenster per Befehl-Q beenden.

Am linken Rand des Ergebnisfensters platzieren wir neun „Statictexte" für die Titel der einzelnen Tests. Hier kann man noch Platz für weitere eigene Tests lassen. Die neun Statictexte bekommen als Titel: „Leerlauf:", „Linien:", „Rechtecke:", „Vollbild Rechtecke:", „Gefüllte Rechtecke:", „Text (9 Punkt)" , „Text (12 Punkt)", „Text (24 Punkt)" und „Macwelt-Bild:". Für eine optimale Darstellung erhalten alle neun Statictext-Kontrollelemente die feste Breite von 130 Pixeln (Eigenschaft „Width").

Rechts daneben platzieren wir neun weitere Statictext-Kontrollelemente, in denen unser Programm später die Ergebnisse darstellen soll. Die Eigenschaft Text setzen wir auf einen einfachen Bindestrich („-"). Als Breite wählen wir 157 Pixel (Eigenschaft „Width"), als Abstand von links (Eigenschaft „Left") 150 Pixel und als Fensterbreite schließlich 320 Pixel. Dann geben wir jedem Statictext für die Ergebnisse noch einen aussagekräftigen Namen, wie zum Beispiel „LinienFeld" oder „Text12Ausgabe". Wie man die Textfelder nennt, bleibt jedem selbst überlassen, nur sollte man die Namen parat haben, wenn man Code schreibt, der dort etwas anzeigen soll.

Noch ein Tipp dazu: Um eine Eigenschaft bei mehreren Kontrollelementen in einem Rutsch zu ändern, selektiert man per Auswahlrechteck mit der Maus oder mit Umschalttaste-Klick mehrere Kontrollelemente gleichzeitig. In der Eigenschaftenpalette lassen sich dann die Eigenschaften sämtlicher selektierten Elemente in einem Arbeitsgang setzen. Doch Vorsicht, es werden nie Eigenschaften angezeigt! Übrigens: Kontrollelemente können Sie mit den Cursortasten auch pixelgenau positionieren.

DIE GRAFIKTESTS IM EINZELNEN

Der Test „Leerlauf" soll herausfinden, wie viele Durchläufe machbar sind, ohne dass irgendeine Grafik ausgegeben wird. Mit diesem Test verfolgen wir das Ziel, einen Wert zu ermitteln, der die Leistung des Prozessors widerspiegelt. Gleichzeitig ist dieser Wert auch das Maximum für die Grafikmessungen, denn keine Grafikroutine kann schneller ablaufen, als wenn sie gar nicht erst aufgerufen wird.

Der Test namens „Linien" zeichnet in zufälligen Farben Linien mit zufälligen Koordinaten in das Testfenster.

Wenn wir „Rechtecke" testen, wird auf dem Bildschirm ähnlich wie beim Linientest an zufälligen Koordinaten in einer zufällig ausgewählten Farbe ein Rechteck aus vier Linien gezeichnet.

Im nächsten Test werden diese Rechtecke mit einer Farbe gefüllt wieder an zufälliger Stelle positioniert. Dagegen zeichnen wir die „Vollbild Rechtecke" immer über die ganze Fläche des Fensters, was hauptsächlich den Pixel-Durchsatz der Grafikkarte belastet.

Die Texttests zeichnen den String „Macwelt“ in den Schriftgrößen 9, 12 und 24 auf den Bildschirm, und zwar auch jeweils in einer zufällig ausgewählten Farbe an einer ebenso zufälligen Position. Dabei sollte man beachten, dass die Zeichengeschwindigkeit stark abnimmt, wenn der Text vom Mac-OS geglättet wird. Dies lässt sich verhindern, indem man im Kontrollfeld „Erscheinungsbild“ die Zeichenglättung ausschaltet.

WAS DAHINTER STECKT: DER PROGRAMMCODE

Ein Doppelklick auf den Button „Start“ bringt uns wieder in den Code-Editor. Dort fügen wir unter dem vorhandenen Befehl „testfenster.show“ die Zeile „Run“ ein. Diese Zeile enthält den Namen eines Unterprogramms (Methode), das wir jetzt programmieren. Mit dem Menüeintrag „Neue Methode…“ (Befehl-Wahl-M) erstellen wir ein neues Unterprogramm. Als Namen geben wir „Run“ ein. Auch hier kann man die Methode nennen, wie man möchte, solange man denselben Namen im Action-Event des Pushbuttons benutzt. In die Methode „Run“ schreiben wir mal ein „Beep“. Durch den Befehl Beep lässt das System einen Warnton erklingen. Starten Sie testweise Ihr Programm und drücken Sie den Button. Wenn kein Ton erklingt, ist vermutlich der Lautsprecher ausgeschaltet.

Nun zu unserem Programmcode in der Methode. Zunächst besorgen wir uns ein Grafikobjekt, mit dem wir unsere Grafik ausgeben wollen. Zudem speichern wir die Werte für die Breite und die Höhe der Ausgabe. Die ist zwar nun festgelegt, aber wenn man das Programm zum Beispiel auf 800 x 600 Pixel erweitern möchte, ist es praktisch, wenn man das jetzt schon berücksichtigt. Je größer der Bildschirmausschnitt, desto relevanter wird die Leistung der Grafikkarte im Vergleich zum Overhead des Systems. Unser Testprogramm soll aber auch auf älteren Macs laufen, die teilwesie nur 640 x 480 Pixel darstellen können.

HIER SEHEN SIE das Testfenster in Aktion. Das Programm testet gerade die Ausgabe von Pixel-Bildern anhand eines verkleinerten Macwelt-Titels.

Zur Deklaration der Variablen schreiben wir folgenden Code: „dim g as graphics“ für das Grafikobjekt und in die nächste Zeile „dim w,h as integer“ für die Breite (Width) und die Höhe (Height). Es ist also möglich, mehrere Variablen vom selben Typ direkt in einer Zeile anzulegen. Um aber den Quellcode übersichtlich zu halten, benutzt man das nur bei eng verbundenen Variablen, wie zum Beispiel bei: „dim Nachname, Vorname as string“ und nicht bei folgendem Beispiel „dim NetzwerkPasswort, DruckerStatus, MonitorHerstellerName as string“.

Da Bindestriche oder Leerzeichen in Variablennamen innerhalb von Realbasic generell nicht erlaubt sind, empfiehlt es sich, die Namen übersichtlicher zu gestalten, indem man einzelne Teilwörter groß schreibt.

Mit dem nächsten Befehl „g=testfen ster.ausgabe.graphics“ holen wir uns in „g“ eine Referenz auf das Grafikobjekt des Canvas im Testfenster. „g“ ist nicht besonders aussagekräftig, aber das Canvas lässt sich auch anders benennen, dazu ändert man im Code einfach den Namen. Sinn des „g“ ist es allerdings, die Zugriffe auf das Grafikobjekt zu verkürzen und jede Menge Schreibarbeit zu sparen. Wenn unser Prozessor immer erst im Fenster nach dem Kontrollelement suchen muss, um dann dort die Referenz auf das Objekt zu erhalten, bremst das den ganzen Rechner ein paar Prozent herunter, und die können das Ergebnis unserer Tests beeinflussen.

Zudem sollte man beachten, dass man auf Objekte nur Referenzen erhält und nie das Objekt selbst. Das lässt sich mit einer Datei vergleichen, die auf einem Server (dem Arbeitsspeicher) liegt und auf die nur das System einen direkten Zugriff hat. Wir dagegen haben lediglich Aliasse auf diese Datei, und wenn wir wie hier ein solches Alias kopieren, dann kopieren wir eben nicht das Original, sondern nur die Referenz.

Mit den Zeilen „w=g.width“ und „h=g.height“ speichern wir noch die Werte für Breite und Höhe in lokalen Variablen.

TEXTAUSGABE ZENTRIERT MIT KONSTANTEN

Die erste Messung erzeugt keine Grafik und misst nur die CPU-Leistung. Während einer Zeit von fünf Sekunden könnte man allerdings meinen, dass der Mac abgestürzt oder eingefroren ist. Damit dieser Eindruck nicht aufkommt, schreiben wir vor den eigentlichen Test den Text „Bitte warten…“ in großer Schrift zentriert in unser Testfenster.

Wenn man einen String zentriert ausgeben will, muss man ihn im Programmcode an zwei Stellen eingeben. Um uns die Tipparbeit zu sparen und den Text später einfach ändern zu können, deklarieren wir dafür eine Konstante. Diese wird lokal arbeiten, also nur in diesem Unterprogramm gelten, und sie soll „warten“ heißen. Wir fügen oben in der Run-Methode vor den „Dim“-Befehlen folgende Zeile ein „const warten="Bitte warten…"”. Solche lokalen Konstanten lassen sich überall im Code deklarieren, um den Code jedoch übersichtlich zu gestalten, setzen wir Konstanten immer ganz oben an den Anfang. An jeder Stelle im Code, wo jetzt ein String erwartet wird, können wir den Namen dieser Konstante verwenden. Beim Kompilieren wird Realbasic den Namen der Konstante automatisch durch ihren Inhalt ersetzen.

Wieder ganz unten in der Methode fügen wir die Zeile „g.textsize=48“ ein. In unserem Grafikobjekt, in das wir unsere „Bitte warten…“-Meldung schreiben wollen, setzen wir damit die Eigenschaft für die Textgröße auf 48 Pixel. Alle folgenden Textausgaben finden nun in dieser Größe statt.

Für die eigentliche Ausgabe greifen wir auf den „DrawString“-Befehl der Graphics-Klasse zurück. Er erwartet eine x- und eine y-Koordinate innerhalb des Grafikbereichs. Als Nächstes wollen wir den Text horizontal zentrieren, wozu wir zunächst die Breite des Textes mit „g.stringwidth(warten)“ ermitteln. Als Ergebnis erhalten wir die Brei-

WIE MAN METHODEN mit Parameterübergabe deklariert, zeigt die Deklaration der Methode „RunBild“. Man trennt die verschiedenen Parameter einfach durch Kommata.

DIESE DEKLARATION der Methode „Run” zeigt, wie man ein Unterprogramm mit dem Namen „Run” anlegt. Nur der Name ist Pflicht.

te des Strings in Pixeln relativ zu den Schriftstilen im Graphics-Objekt „g". Da nicht anders gewünscht, ist die Schrift bei uns die Systemschrift in der Größe 48 Punkt. Fett (g.bold), Kursiv (g.italic) und Unterstrichen (g.underline) sind ausgeschaltet, lassen sich aber zum Beispiel mit der Zeile „g.bold=true" einschalten.

Um den Text jetzt noch zu zentrieren, ziehen wir die Breite des Strings von der Breite des Grafikbereichs ab und dividieren durch zwei (Mittelwert). Bei der Höhe ziehen wir die Textgröße von der Grafikhöhe ab und dividieren wieder durch zwei. Am Ende sieht die nächste Zeile wie folgt aus: „g.drawstring warten,(g.width-g.string Width(warten))/2,(g.height-g.textsize)/2".

Anschließend legen wir für die Aufrufe der einzelnen Tests die Methoden an. Wir fangen mit den Messroutinen für die CPU-Messung und die Linien an. In den Zeilen „RunLeerlauf g,w,h" und „RunLi nien g,w,h" rufen wir jeweils die Methode auf und übergeben ihr die drei Parameter (Grafikumgebung, Breite und Höhe). Ein abschließendes „testfenster.close" schließt unser Testfenster nach dem Test wieder.

TEST FÜR DEN LEERLAUF

Wir deklarieren eine Methode mit folgender Zeile: „Sub RunLeerlauf(g as graphics,w as integer,h as integer)". Dazu rufen wir im „Bearbeiten"-Menü die Funktion „Neue Methode…" auf und verwenden als Namen „RunLeerlauf". Für die Parameter geben wir „g as graphics,w as integer,h as integer" ein. In diesem Fall werden die drei Variablen so übergeben, dass sie alle Methoden lokal verfügbar haben. Lokale Variablen sind im Zugriff schneller als globale.

Nun braucht die Methode einen Inhalt. Für die Zeitmessung benutzen wir hier die Funktion „Microseconds", die uns die Anzahl der seit dem Rechnerstart vergangenen Mikrosekunden liefert. Da sich Mikrosekunden nicht in Integervariablen speichern lassen – die Zahl wird einfach zu groß – benutzen wir eine Variable vom Typ „double". Mit der Zeile „dim z as double" deklarieren wir für die Zeitmessung eine Variable namens „z", die den Endwert für unsere Messung speichert.

Der Benutzer soll stets darüber informiert sein, welcher Test gerade aktuell läuft. Dazu schreiben wir mit „testfenster.tit le="Nix Tun"" den Namen des aktuellen Tests in die Titelzeile.

DAS GEHEIMNIS DER ZEITMESSUNG

Alle Tests beruhen auf dem Verfahren, fünf Sekunden lang zu zählen, wie oft eine bestimmte Grafikfunktion ausgegeben wird. Je schneller die Grafikkarte ist, desto mehr Aufrufe schafft sie in dieser Zeit. Im „Bearbeiten"-Menü erzeugen wir dazu per „Neue Eigenschaft…" (Befehl-Wahl-P) eine globale Variable mit „count as integer". Diese Ganzzahlvariable dient im Programm als Zähler für alle unsere Tests. Wir setzen sie vor jedem Test mit der Codezeile „count=0" zunächst auf den Anfangswert Null.

Dann bestimmen wir mit der „Microseconds"-Funktion den Endzeitpunkt des Zählvorgangs. In der Variable z speichern wir den Wert des Mikrosekunden-Zählers plus fünf Sekunden (also fünf Millionen Mikrosekunden) Testdauer. Das erreichen wir mit folgender Codezeile: „z=microse conds +5000000.0".

Da das Programm bei unserem Leerlauftest innerhalb der Messschleife nichts anderes macht als zählen, sieht der restliche Code deshalb so aus:

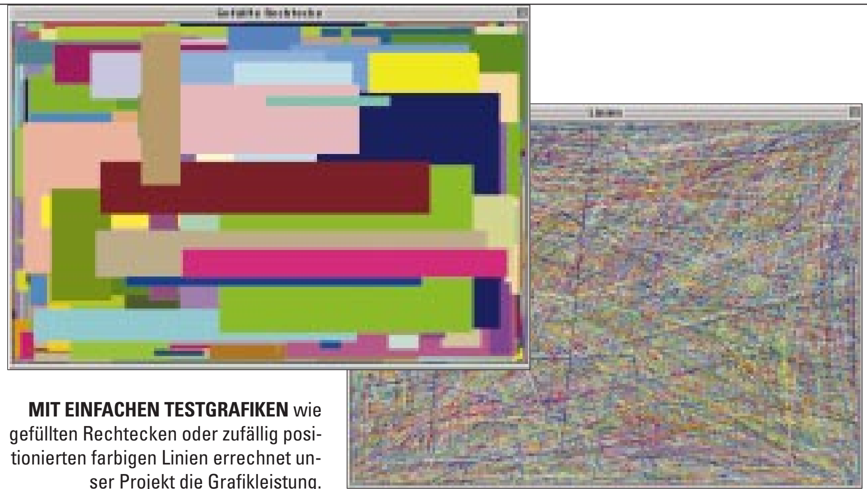
```
Do
    count=count+1
    //kein Test, da Leerlauf.
loop until microseconds>z
leerlauffeld.text=zeit
```

Der Aufbau ist einfach. Mit „Do" leiten wir die Schleife ein. Anschließend erhöhen wir den Zähler „count" um eins. In dem nachfolgenden Kommentar steht, dass eigentlich nichts im Leerlauf gemacht wird. Wenn die aktuelle Zeit im Mikrosekunden-Zähler den vorher festgelegten Endzeitpunkt überschreitet, verlässt das Programm die Schleife. Zum Schluss gibt es im Statictext für das Ergebnis den Wert des Leerlauftests aus, wozu es eine selbst geschriebene Funktion benutzt, die wir „Zeit" nennen.

DIE FUNKTION ZUR ZEITFORMATIERUNG

Damit unser Programm bis hierhin funktioniert, müssen wir erst noch die Funktion „Zeit" ins Leben rufen, die uns den unansehnlichen Zeitwert übersichtlich in Mikrosekunden formatiert.

Mit Hilfe des Menübefehls „Neue Methode…" (Befehl-Wahl-M) erzeugen wir eine Funktion. Diesmal ist der Name „Zeit". Damit es eine Funktion wird, müssen wir allerdings einen Ergebniswert angeben. Dazu wählen wir in dem Pop-up-Menü rechts unten den Eintrag „String" aus oder schreiben selbst „String" als Ergebnistyp hinein.



MIT EINFACHEN TESTGRAFIKEN wie gefüllten Rechtecken oder zufällig positionierten farbigen Linien errechnet unser Projekt die Grafikleistung.

Als Code für die Funktion schreiben wir folgende Zeile: return format(count/5000, "0.0")+" pro ms ("+format(count,"0")+")". Mit dem „Return"-Befehl geben wir das Ergebnis der Funktion an den aufrufenden Code zurück. Den Formatbefehl kennen wir schon aus der letzten Folge. Hier teilen wir den Wert der globalen Variable „count" durch 5000 (für 5 Sekunden bei 1000 Millisekunden pro Sekunde), denn wir geben unser Ergebnis in Frames (Grafikereignis) pro Millisekunde an. Mit „0.0" sorgen wir für eine Nachkommastelle. In Klammern hinter dem Text „pro ms" geben wir noch den genauen Inhalt der Count-Variable aus.

Jetzt sollte unser Programm schon funktionieren und den Leerlauf testen.

JETZT KOMMT GRAFIK: LINIEN

Für den Test von geraden Linien kopieren wir einfach den kompletten Leerlauftest in eine neue Methode und ändern ein paar Zeilen. In die Titelleiste schreiben wir jetzt „Linien" hinein.

Anstatt des Kommentars hinter dem „count=count+1" fügen wir die Zeile „g.fo reColor=rgb(rnd*256, rnd*256, rnd*256)" ein. Diese Zeile macht Folgendes: In die Eigenschaft „ForeColor", die als Color-Objekt definiert ist (zur Erinnerung: Eine Klasse ist die Definition und ein Objekt das Konkrete und Anfassbare. Die Wörter werden jedoch häufig wie Synonyme gebraucht) schreibt das Programm jeweils für die Farben Rot, Grün und Blau drei zufällig ausgewählte Werte zwischen 0 und 255. Insgesamt ergibt das eine zufällige Farbe aus 16,7 Millionen möglichen Farben.

Mit „g.drawline rnd*w,rnd*h,rnd*w, rnd*h" entsteht dann eine zufällige Linie. Dazu werden zwei zufällige x- und zwei y-Werte aus den Maßen für Breite und Höhe des Testfensters gebildet, und die Linie wird gezeichnet. Zum Schluss sichern wir mit „li nienfeld.text=zeit" den Wert im passenden Statictext-Kontrollelement.

ANDERE GRAFIKTESTS

Indem wir den Linientest abermals duplizieren, erzeugen wir sehr schnell das Codegerüst für den Test von Rechtecken.

Zuerst ermitteln wir in einer lokalen Variable („Dim l,t as integer“) mit der Zeile „l=rnd*w“ einen zufälligen Abstand des Rechtecks vom linken Fensterrand und mit der Zeile „t=rnd*h“ den Abstand von oben. Durch „g.drawrect l,t,rnd*(w-l),rnd*(h-t)“ zeichnen wir dann in unserem Grafikbereich ein Rechteck an den Koordinaten l / t. Als Breite nehmen wir wieder eine zufällige Zahl, die aber nie größer als die Breite abzüglich des linken Abstands sein darf (rnd*(w-l)). Wenn wir einfach nur „rnd*w“ benutzen würden, sind die Rechtecke nicht mehr nur im Fenster, sondern ragen unter Umständen rechts heraus. Die gleiche Berechnung machen wir für die Höhe.

Für gefüllte Rechtecke kopieren wir noch einmal denselben Programmcode, nehmen aber statt „g.drawrect“ den Befehl „g.fillrect“. Für die meisten Grafikfunktionen gibt es einen Befehl mit „g.draw…“ für Linien und „g.fill…“ für gefüllte Flächen. Probieren Sie ruhig andere Befehle wie etwa „drawoval“ oder „filloval“ aus.

Im nächsten Test bringen wir die volle 2D-Beschleunigung der Grafikkarte zur Geltung, indem wir mit „g.fillrect 0,0,w,h“ die ganze Fensterfläche füllen.

Auf einem G4/450 schafft unser Programm 1,8 Fillrects pro Millisekunde, also 1800 Rechtecke pro Sekunde. Für jedes Fillrect bewegen wir 600 x 400 Pixel, was einer Füllrate von 432 Megapixel pro Sekunde entspricht. Intern arbeitet die Grafikkarte mit 32 Bit pro Pixel, das ergibt eine Füllrate von über 3 GB pro Sekunde im Bildschirmspeicher. Zum Vergleich: Würde der Hauptprozessor die Arbeit erledigen, müssten diese Daten komplett über den AGP-Bus wandern. Der AGP-Bus (Apple verwendet AGP 2x) schafft aber nur maximal 533 MB/s (das ergibt sich aus 133 MHz AGP-Bustakt, mal 4 Byte pro Takt). Nun kann man sich vorstellen, warum eine 2D-Beschleunigung der Grafikkarte den Mac so stark aufwertet.

DREI TEXTTESTS IN EINEM

Zu einer Methode fassen wir die drei Tests für den Text zusammen. Deshalb deklarieren wir die Testmethode „RunText“ mit den Parametern „g as graphics,w as integer,h as integer,size as integer,feld as statictext“. Die Variablen g, h und w verwenden wir genauso wie bei den anderen Tests. Zusätzlich übergeben wir in „size“ die gewünschte Textgröße in Pixeln (9, 12 oder 24 Punkt). In „feld“ übergeben wir das Statictext-Kontrollelement, in das später die Ausgabe des Ergebnisses erfolgt.

Man sollte sich merken, dass auch ein Kontrollelement nur ein Objekt ist, das sich wie jeder andere Variablentyp bei Unterprogrammen als Parameter oder als Ergebnistyp verwenden lässt. Als Beispiel könnte man eine Methode deklarieren, die fünf Buttons und eine Nummer übergeben bekommt und den passenden Button zurückliefert.

Für die „TextTest“-Methode kopieren wir nochmals den Quellcode eines der anderen Tests. Damit die Texte perfekt in den Grafikbereich passen, müssen wir die Breite des Textes ermitteln und speichern. Dafür deklarieren wir die Variable Breite als Integervariable („dim breite as integer“). Als Nächstes folgt eine Konstante für den Text, da wir darauf mehrmals zurückgreifen: „const text="Macwelt"“. Die Titelleiste füllen wir abhängig von der Textgröße mit der Zeile: „testfenster.title="Text "+str(size)+"Punkt"“. Wie im Taschenrechner müssen wir auch hier die Zahlenvariable in einen String umwandeln. Da „size“ sicher betragsmäßig unter einer Million liegt, reicht die Funktion „Str()“ dafür aus. Noch bevor wir „count“ auf Null setzen, weisen wir der Textgröße vom Grafikobjekt mit „g.text size=size“ den gewünschten Wert zu. Anschließend messen wir mit der Funktion „stringwidth“ die Länge des Texts in unserer Konstante und speichern den Wert in der Variable „Breite“. Folgende Zeile erledigt dies: „breite=g.stringwidth(text)“.

HIER DIE ERGEBNISSE eines Power Mac G4/450 AGP (Frühjahrsmodell) mit 256 MB RAM, Mac-OS 9.0.4 und deaktiviertem virtuellen Speicher.

In der Schleife berechnen wir für die Textausgabe einen Abstand vom linken Fensterrand. Dazu ziehen wir von der Fensterbreite die Breite des Textes ab und erzeugen mit „l=rnd*(w-breite)“ einen zufälligen Wert. Für den Abstand von oben müssen wir die Textgröße von der Höhe subtrahieren. Wir benutzen dafür nicht „g.textsize“, sondern unser „size“, denn die lokale Variable ist wesentlich flotter als eine Eigenschaft eines Objekts, und wir wollen möglichst wenig Zeit im Test verschwenden, um nicht ungenau zu werden. Da bei der Textausgabe immer die Linie unter dem Text angegeben wird, müssen wir noch „size“ addieren: „t=rnd*(h-size)+size“. Jetzt zeichnet unser Programm per „g.drawstring text,l,t“ den Text aus der Konstante an der Stelle l / t.

Nach der Schleife speichern wir in dem Statictext, den wir in „Feld“ bekommen, den Zeitwert wie folgt ab „feld.text=zeit“. Aufgerufen wird der Test jeweils für 9, 12 und 24 Pixel große Schrift:

„runtext g,w,h,9,text9feld“
„runtext g,w,h,12,text12feld“
„runtext g,w,h,24,text24feld“

BUNTE BILDER

Als Letztes wollen wir die Darstellung von Pixel-Bildern messen. Der Code ist dem Linientest sehr ähnlich, so dass wir eine Kopie davon benutzen. Den Titel ändern wir mit „testfenster.title="Bild"“. Das Bild wird an eine zufällig ausgewählte Position gemalt, und wie in den anderen Tests auch ziehen wir die Breite und die Höhe des Bildes ab.

Das Bild selbst wird einfach per Drag-and-drop ins Projektfenster gezogen. Danach lässt es sich mit dem Namen, den es dort hat, vom Programm aus ansprechen. Im folgenden Code haben wir das Bild „macwelt.jpg“ genommen, das im Projektfenster dann „macwelt“ heißt. Zum Darstellen von Bildern benutzt man den Befehl „drawpicture“, mit dem man nicht nur Bilder ausgibt, sondern die Ausgabe auf einen Ausschnitt des Bildes beschränken und diesen auch noch skalieren kann. Die fertige Zeile in der Testschleife lautet dann „g.drawpicture macwelt,rnd*(w-macwelt.width),rnd*(h-macwelt.height)“. Mit dem obligatorischen „bildfeld.text=zeit“ speichern wir abschließend noch das Endergebnis.

WEITERE IDEEN

Anhand dieser einfachen Beispiele kann man leicht weitere und komplexere Grafiktests hinzufügen. Wir empfehlen, unser Beispielprogramm genau anzuschauen und mit verschiedenen Grafikfunktionen herumzuexperimentieren. Hier schon mal ein paar Ideen: Denkbar wäre ein Test für Ovale beziehungsweise Kreise („g.drawoval“ und „g.filloval“) oder Text von 5 bis 50 Pixel mit einer kleinen grafischen Kurve, die anzeigt, wie sich die Ausgabegeschwindigkeit im Vergleich zur Textgröße verhält.

Das fertig kompilierte Programm sowie den Quellcode des kompletten Projekts können Sie im Internet unter http://www.macwelt.de/_magazin herunterladen.

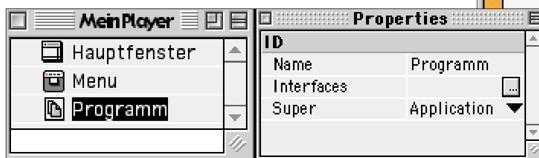
FAZIT

Realbasic ist mächtig genug, um auch komplexe Grafik-Benchmark-Tests zu erzeugen. Damit wir in puncto Grafik und Multimedia noch weiter vorankommen, wollen wir uns im nächsten Teil dem Thema Quicktime-Programmierung widmen und Audio- und Videodateien abspielen. *cm*

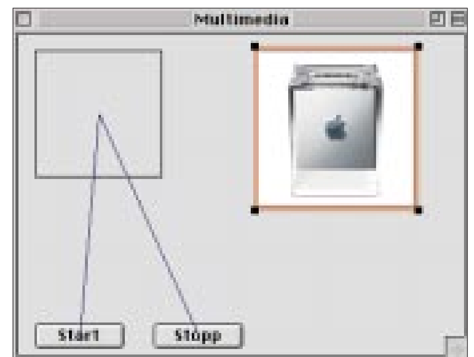
Serie Realbasic

1 Einführung .... **Heft 9/2000**
2 Taschenrechner im Eigenbau .... **Heft 10/2000**
3 Grafikprüfung .... **Heft 11/2000**
4 Quicktime-Programmierung .... **Heft 12/2000**

IM PROJEKTFENSTER des Drag-and-drop-fähigen Movieplayer sieht man die selbsterzeugte Klasse namens „Programm“.



SO SIEHT unser bunter Movieplayer aus, der gerade einen iMac-TV-Spot abspielt.



BUTTON MIT BILD Buttons müssen nicht grau sein. Man kann in Realbasic auch aus beliebigen Grafiken Knöpfe erzeugen.

ALLES SO SCHÖN BUNT HIER: MOVIEPLAYER IM EIGENBAU

IN DIESEM MONAT wollen wir uns mit Multimedia-Programmierung, also Bildern, Tönen und Filmen beschäftigen. Als Ziel setzen wir uns ein kleines Multimedia-Programm, das man sehr leicht zu einer einfachen Präsentationssoftware oder einer interaktiven Grafikanwendung, wie man sie von vielen Multimedia-CDs her kennt, ausbauen kann

VON CHRISTIAN SCHMITZ

UNSEREN ERSTEN KLEINEN Movieplayer erstellen wir mit lediglich zwei Objektverknüpfungen, ohne eine einzige Programmzeile zu schreiben. Dazu starten wir wie immer mit einem leeren Projekt. Für unser Multimedia-Programm benötigen wir etwas Bild- und Filmmaterial. Dazu suchen wir uns einen Quicktime-Film und ziehen ihn per Drag-and-drop in das Projektfenster. Später brauchen wir noch zwei Pixel-Bilder mit identischen Abmessungen. Noch ein Hinweis: Bitte beachten Sie, dass „Objektverknüpfungen“ eine spezielle Funktion von Realbasic 2.x ist und daher nicht mit Realbasic 1.0 beziehungsweise Realbasic Light funktioniert. Für diese Realbasic-Versionen haben wir unsere Beispielprogramme etwas modifiziert. Auf der Heft-CD finden Sie die Quellcodes zu den modifizierten Versionen.

Wie Sie im Screenshot vom Projektfenster weiter vorne auf der Seite 154 sehen, hat Realbasic die Dateinamen zurechtgestutzt. Das muss auch so sein, denn diese Bezeichnungen dienen im Programmcode als Variablennamen, damit man direkt auf die Bilder und Filme im Projekt zugreifen kann. Alle Film- und Sounddateien, die der Quicktime-Player abspielt, lassen sich direkt in das Projekt ziehen und als Movie-Objekt unter dem jeweiligen Namen ansprechen. Es ist außerdem möglich, alle Bilder, die der Quicktime-Picture-Viewer anzeigen kann, ins Projekt zu ziehen und als Picture-Objekte zu verwenden. System-7-Tondateien (zum Beispiel Warntöne oder Effekte) landen direkt als Soundobjekte im Projekt.

QUICKTIME IST VIEL MEHR als nur ein Movieplayer-Alias auf dem Schreibtisch. Apple stellt mit Quicktime eine umfangreiche Funktionenbibliothek zur Verfügung, die jeder Programmierer für seine Zwecke nutzen kann. Auch unter Realbasic ist es möglich, einfach auf den mächtigen Quicktime-Befehlsumfang zurückzugreifen. Wenige Zeilen Programmcode reichen für einen eigenen Movieplayer aus

MIT DEM EIGENEN MOVIEPLAYER HEISST ES: FILM AB!

Da nun alle benötigten Dateien im Projekt als Alias vorliegen, gehen wir zum Hauptfenster über. Aus der Werkzeugpalette ziehen wir das Movieplayer-Steuerelement in unser Hauptfenster. In der Eigenschaftenpalette wählen wir im Pop-up-Menü bei der Eigenschaft „Movie“ unseren Film aus. Jetzt können wir das Programm schon starten und den Film abspielen. Einige der Eigenschaften des Movieplayer-Steuerelements sind bereits per Default eingeschaltet. So sorgt die Option „AutoResize“ zum Beispiel dafür, dass der Movieplayer genau so groß ist wie der Film. Wenn man diese Option ausschaltet, skaliert Realbasic den Film entsprechend unserem Kontrollelement. Der Film lässt sich also beispielsweise in doppelter Größe abspielen, indem wir die „AutoResize“-Eigenschaft ausschalten und das Kontrollelement im Fenster auf die doppelte Filmgröße aufziehen.

Mit der Eigenschaft „Speaker“ erscheint automatisch der Lautstärkeregler unter dem Film, und die Option „HasStep“ zeigt die Vor- und Rückspultasten. Diese Funktionen

sind fest in Apples Quicktime eingebaut, und Realbasic setzt komplett darauf auf. Dank an Apple, denn das spart uns eine Menge Programmierarbeit.

INTERFACE-PROGRAMMIERUNG EINFACH PER DRAG-AND-DROP

Ziehen Sie noch zwei Buttons (Push-Buttons oder Bevel-Buttons) in das Fenster. Wenn Sie mit der Maus einen Button selektieren, können Sie direkt losschreiben und die Eigenschaft „Caption" des Buttons ändern. Auf jeden Fall nennen wir die Buttons „Start" und „Stop", denn diese Funktionen sollen Sie später im Programm haben.

Jetzt wollen wir eine Verknüpfung vom Movieplayer zu dem Start-Button erzeugen. Wir drücken die Tasten Befehl-Wahl-Umschalt und klicken auf den Startknopf. Mit gedrückter Maustaste ziehen wir den Mauspfeil auf das Movieplayer-Kontrollelement. Während wir ziehen, folgt eine Linie von der Maus zum Startknopf, und das Movieplayer-Steuerelement lässt sich erfassen (siehe Abbildung Seite 150). Wenn wir die Maustaste loslassen, erscheint der Dialog „New Binding" und zeigt uns die Verknüpfungen. Wie Sie sehen, kann ein Button, wenn er gedrückt wird, den Film entweder starten oder stoppen. Für den Button „Start" wählen wir also die Verknüpfung „Play Movie Player1 movie …" und für den „Stopp"-Button dementsprechend die Verknüpfung „Stop Movieplayer1 movie …".

Starten wir unser Programm jetzt, können wir den Film mit unseren eigenen Steuerknöpfen abspielen und wieder stoppen.

Dieses Beispiel finden Sie unter dem Namen „Multimedia 1" auf der Heft-CD. Auch für Realbasic Light gibt es dort eine Version, die jedoch auf die Verknüpfungen verzichten muss und stattdessen mit den Befehlen „movieplayer1.play" und „movieplayer1.stop" in den Action-Events der jeweiligen Buttons arbeitet.

BUNT ANIMIERTE BUTTONS MIT DEM ROLLOVER-EFFEKT

Als Nächstes basteln wir zusammen einen eigenen animierten Multimedia-Button, der sich verändert, wenn man mit der Maus darüber fährt. Dazu ziehen wir ein Canvas-Steuerelement in das Fenster und setzen die Eigenschaft „Backdrop" in der Palette auf das erste Bild im Projekt. Bei einem Teststart des Programms sollte nun schon das erste Bild im Fenster zu sehen sein.

Wieder zurück im Fenstereditor öffnen wir den Code-Editor des Canvas per Doppelklick. Unter den vielen Events, die ein Canvas anbietet, finden wir auch den Event „MouseEnter". Dort tragen wir als Programmcode die Zeile „me.backdrop=imac blau" ein, wobei „imacblau" hier für das zweite Bild steht. Falls Sie ein anderes Bild benutzen wollen, passen Sie den Namen einfach an. Im Event „MouseExit" kommt parallel dazu der Befehl „me.backdrop=cube", wobei „cube" hier der Name des ersten Bildes ist. „me" bedeutet, dass der Programmcode eine Eigenschaft seines eigenen Objekts anspricht. Man könnte stattdessen den Namen des Objekts benutzen, aber „me" ist kürzer und eleganter.

Die Eigenschaft „Backdrop" von einem Canvas oder auch einem ganzen Fenster enthält das Bild, das als Fensterhintergrund angezeigt wird. Fährt man nun zur Laufzeit mit der Maus über das Bild, ruft Realbasic unseren Eventhandler auf, und der Programmcode ändert blitzschnell das Hintergrundbild. Verlässt die Maus den Canvas wieder, passiert das Gleiche, nur dass jetzt das ursprüngliche Bild erscheint.

Um den Redraw der Objekte brauchen wir uns glücklicherweise nicht zu kümmern, denn sobald die Backdrop-Eigenschaft im laufenden Programm geändert wird, führt Realbasic den Redraw des Canvas automatisch durch. Auf der aktuellen Heft-CD finden Sie dieses Beispiel unter dem Namen „Multimedia 2".

JETZT KOMMT BEWEGUNG INS SPIEL: BILDWECHSEL BEIM MAUSKLICK

Damit der Benutzer beim Mausklick auf den Multimedia-Button auch ein visuelles Feedback bekommt, fügen wir in den Events „MouseDown" und „MouseUp" noch etwas Code ein (Beispiel: „Multimedia 3" auf der Heft-CD). Dafür benötigen wir erst einmal ein weiteres Bild. Also noch schnell ein drittes Bild suchen und per Drag-and-drop auf das Projekt ziehen. In den „MouseDown"-Event des Canvas schreiben wir die Zeile „me.backdrop=ibook", damit das Bild mit dem Namen „iBook" angezeigt wird. Im Event „MouseUp" müssen wir passend dazu ein „me.backdrop=imacblau" einfügen, um das vorherige Bild wieder zu restaurieren. Wenn man jetzt das Programm startet, klappt das nicht richtig. Genau genommen wird „MouseUp" nie aufgerufen. Warum es nicht aufgerufen wird, ist eine Standardfrage von Realbasic-Neulingen. Aber schauen wir uns den Event doch einmal an. Da steht „Function MouseDown() as boolean". Der Event ist eine Funktion, liefert also einen Wert zurück. Wir ergänzen in diesem Event hinter unserer Zeile die Zeile „Return true". Damit sollte auch das „Mouse Down" klappen.

Viele Events, zum Beispiel alle Menü-Handler und auch der „Keydown"-Event erwarten, dass man mit „return true" meldet, ob man den Event verarbeitet hat. Ansonsten wird er eine Instanz höher, also an das Steuerelement, das Fenster oder das ganze Programm, weitergereicht.

PER KNOPFDRUCK ALLES UNTER KONTROLLE: „FILM AB!"

Wir möchten gerne, dass der Film per Mausklick auf unseren eigenen Multimedia-Button startet. Dazu ergänzen wir im „Mouse Up"-Event die Zeile „movieplayer1.play".

DIE REALBASIC-SPEICHERVERWALTUNG

Bei dem vielen Hin- und Herkopieren von Bildern stellt sich sicher die Frage, ob und wann Realbasic mal in Speichernot kommt. Dazu ein paar beruhigende Worte zur internen Speicher- und Variablenverwaltung von Realbasic:

Wenn wir ein Bild laden, reserviert Realbasic intern dafür Speicher im RAM. Jedes Objekt im Speicher verfügt über einen Zähler, der festhält, wie viele Variablen als Referenzen auf das Objekt existieren. Bei dem Befehl „me.backdrop=bild" wird zuerst einmal der Zähler von dem Bild, auf das „backdrop" zeigt, um eins verringert. Da jetzt keine Referenz mehr existiert, wird das Bild komplett aus dem Speicher entfernt und anschließend das neue Bild eingeladen. Um die Speicherverwaltung brauchen wir uns als Realbasic-Programmierer also erfreulicherweise nicht zu kümmern, das macht Realbasic von selbst im Hintergrund. Sollte der Speicher dennoch einmal knapp werden, bekommen wir statt des Objekts den Nullwert „nil" zurück, und der Benutzer sieht einfach nichts. Es empfiehlt sich, im Hinterkopf zu behalten, dass Realbasic alle Bilder im Projekt beim Programmstart direkt einlädt und bis zum Programmende im Speicher hält. Wer Speicher sparen möchte, sollte also möglichst kleine Bilder verwenden oder sie zur Laufzeit aus Bilddateien nachladen.

Damit teilen wir dem Movieplayer-Steuerelement mit, dass es den Film starten soll. Unser Beispiel „Multimedia 4“ auf der Heft-CD macht genau das.

Wir haben nun gelernt, wie man mit drei Pixel-Bildern multimediale Buttons erzeugt. Unser Beispiel „Multimedia 5“ geht noch weiter und stellt einen richtigen Stand-alone-Movieplayer dar.

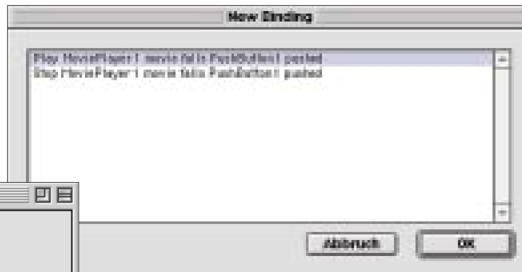
FARBE MUSS HER: DER BUNTE MOVIEPLAYER

Auf der Heft-CD finden Sie neben dem Programmcode die Grafiken, die für einen bunten Movieplayer sorgen. Konkret sind das jeweils drei Bilder für den Start- und den Pauseknopf, eins für den Knopf im Ruhezustand, eins für den Knopf, wenn die Maus darüber fährt und eins, wenn man die Maustaste drückt. Zusätzlich benötigen wir ein weiteres Bild, das als allgemeiner Fensterhintergrund dient – der Einfachheit halber erzeugen wir mit Apple Works einen Farbverlauf, jedes andere Bild funktioniert auch.

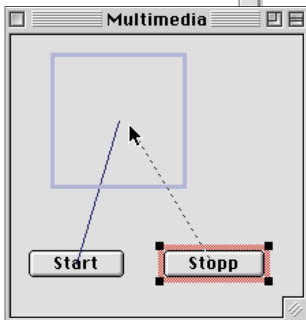
Wir importieren zunächst alle sieben Grafiken und den Film per Drag-and-drop in das Projektfenster eines neuen Projekts. Damit um die Bilder kein weißer Rand sichtbar bleibt, wählen wir im Projektfenster die Bilder für die Knöpfe einzeln an und setzen jeweils in der Eigenschaftenpalette die Eigenschaft „Transparent“ auf „white“. Damit wird die Farbe Weiß (im RGB-Farbmodell 100% Rot, 100% Grün und 100% Blau) transparent gezeichnet.

Im Hauptfenster platzieren wir wieder ein Movieplayer-Kontrollelement für den Film. Es sollte genau so groß sein wie unser Film (der iMac-Film ist 240 x 180 Pixel groß). Die Eigenschaft „Controller“ des Movieplayer-Kontrollelements setzen wir auf „None“, damit der Benutzer unsere Multimedia-Buttons für die Bedienung nimmt und nicht die Knöpfe von Quicktime. Die Eigenschaft „Autoresize“ können wir wahlweise ein- oder ausschalten. Sie wird nicht mehr gebraucht, stört aber auch nicht. Klicken wir in den leeren Fensterteil, lässt sich in der Palette der Titel des Fensters ändern. Jedes Fenster hat die Eigenschaft „Backdrop“, in der festgehalten wird, welches Bild im Fensterhintergrund erscheint. Dort wählen wir unser Bild „Hintergrund“ aus. Damit bekommt das Fenster einen bunten Hintergrund. Auf Wunsch kann man zudem die Eigenschaft „Frame“ des Fensters ändern. Dort wird angegeben, ob das Fenster als normales Dokumentenfenster („Document Window“) oder beispielsweise als randloses Fenster („Plain Window“) erscheinen soll. Das Fenster muss für unser Beispiel mindestens 266 Pixel breit und 276 Pixel hoch sein.

Wieder im Fenster selbst setzen wir zwei Canvas-Steuerelemente unter den Movieplayer. Sie dienen uns als Buttons. Dem linken Fenster weisen wir in der Eigenschaftenpalette das Bild „PlayButton1“ als Hintergrund (Eigenschaft „Backdrop“) zu. Beim rechten Nachbarn ist es das Bild „PauseButton1“. Bei beiden Elementen tragen wir die Größe der Bilder genau ein (für „Play“ sind es 58 x 58 Pixel und für „Pause“ 57 x 58 Pixel).



DIESES FENSTER zeigt an, welche Verknüpfungen für Buttons und den Movieplayer im Projekt verfügbar sind.



DURCH ZIEHEN mit der Maus stellt man eine Verknüpfung von einem Button zu einem Movieplayer-Objekt her.

In den Events unserer Buttons müssen wir nun noch den Programmcode wie im Beispiel „Multimedia 4“ eintragen. Also schreiben wir jeweils bei „MouseEnter“ „me.backdrop=playbutton2“ und „me.back drop=pausebutton2“ und im Event „Mouse Exit“ „me.backdrop=playbutton1“ sowie „me.backdrop=pausebutton1“ hinein. In den „MouseDown“-Events laden wir das jeweils dritte Bild per „me.backdrop=playbutton3“ und „me.backdrop=pausebutton3“.

Bei den „MouseUp“-Events schreiben wir „me.backdrop=playbutton2“ und „me. backdrop=pausebutton2“. Vorsicht: hier nicht die Zeile „return true“ vergessen. Durch „movieplayer1.play“ beziehungsweise „movieplayer1.stop“ in den „Mouse Down“-Events der Buttons starten und stoppen wir dann den Film. Nun ist unser Movieplayer bereits einsatzbereit.

DRAG-AND-DROP AUF DAS MOVIEPLAYER-ICON: DAS STECKT DAHINTER

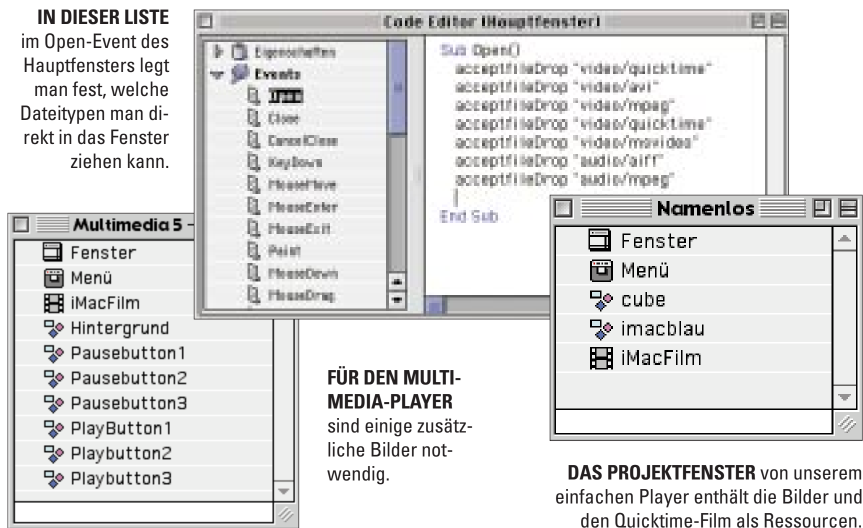
Bislang spielt unser Movieplayer nur einen bestimmten, bereits vorgeladenen Film ab. Viel schöner wäre eine Lösung, die beliebige Filme verarbeitet und sie lädt, wenn man sie per Drag-and-drop im Finder auf den Movieplayer schiebt. Sinnvoll wäre zudem ein Fenster, in dem man die Lautstärke regeln und den Film starten und unterbrechen kann. Dieses Beispiel programmieren wir nun in Realbasic 2 Standard/Pro. Es sollte aber – mit kleinen Änderungen – auch in der Light-Version machbar sein.

Wieder beginnen wir mit einem leeren Projekt und benennen das Standardfenster (Fenster1) in „hauptfenster“ um. Über das Ablagemenü fügen wir eine neue Klasse hinzu. In der Eigenschaftenpalette ändern wir die Eigenschaft „Super“ auf „Application“ und nennen die Klasse „Programm“. Diese Klasse beinhaltet automatisch ein paar Events. Als Erstes schreiben wir in den Event „NewDocument“ die Zeile „msgbox "Kein Dokument aufgerufen"“. Wenn man jetzt das Programm startet, wird ein Warndialog mit dieser Meldung ausgegeben. Nun öffnen wir im Menü „Bearbeiten“ den Dialog „Dateitypen...“. Dort erstellen wir für jeden Dateityp, den unser Programm öffnen können soll, einen Realbasic-Dateityp. Klickt man auf den Button „Neu…“, lassen sich links in einem Pop-up-Menü vorgefertigte Dateitypen auswählen. Wir legen also für jeden Audio- und jeden Videodateityp einen in Realbasic an. Bitte denken Sie an das Häkchen bei „Dokument Icon“, denn ohne es funktioniert Drag-and-drop auf das Programmsymbol nicht.

Wenn wir eine Datei auf das kompilierte Programm im Finder ziehen, schaut der Finder nach, ob unser Programm diesen Dateityp erlaubt. Falls ja, übergibt er die Datei an das Programm, es empfängt eine Meldung des Finders und ruft in der Klasse „Programm“ den Event „OpenDocument“ auf. Als Parameter hat dieser Event ein „DragItem“. Darin enthalten ist unter anderem auch ein „FolderItem“-Objekt, das uns als Alias auf die eigentliche Datei dient. Wir ergänzen also im Event „OpenDocument“ noch eine Zeile „run item“. Den Parameter des Event, nämlich das Folder-Item „item“, das auf die Datei verweist, übergeben wir dem Unterprogramm beziehungsweise der Methode „Run“, die wir später noch programmieren wollen. Vorsicht: Ist in der Dateitypenliste auch nur ein Eintrag mit „????“ für den Typ und „????“ für den Creator vorhanden, akzeptiert das Programm alle Dateien. Das wäre für unseren Player allerdings in Ordnung, denn Dateitypen, die er nicht kennt, ignoriert das Programm einfach.

Nun legen wir noch schnell eine neue Methode namens „Run“ an (Menüpunkt „Neue Methode…“ im „Bearbeiten“-Menü anklicken). Dabei tragen wir als Übergabeparameter „datei as folderitem“ ein. Die Methode soll diese Datei später laden, im Hauptfenster anzeigen und abspielen.

Dazu müssen wir erst einmal das Hauptfenster modifizieren. Wir ändern die Fenstergröße auf einen relativ kleinen Wert von beispielsweise 200 x 100 Pixel. Darüber hinaus schalten wir die Eigenschaften „Grow Icon“ und „ZoomIcon“ in der Palette aus, denn das Fenster ändern wir automatisch vom Programm aus. Im Fenster platzieren wir anschließend ein Movieplayer-Steuerelement in Fenstergröße, jedoch ohne Rand, also left= -1, top=-1, width=202, height= 102. Dann schalten wir die vier Lock-Eigenschaften (Left, Top, Right, Bottom) des Movieplayer-Objekts ein, damit er sich automatisch an die Fenstergröße anpasst. Das „MoviePlayer“-Kontrollelement benennen wir um in „Player“. Ein Doppelklick auf das Fenster (oder den Player) öffnet wie immer den Code-Editor, und wir können eine Methode namens „lade“ anlegen, die als Übergabeparameter die Zeile „datei as folder item“ bekommt.



IN DIESER LISTE im Open-Event des Hauptfensters legt man fest, welche Dateitypen man direkt in das Fenster ziehen kann.

FÜR DEN MULTIMEDIA-PLAYER sind einige zusätzliche Bilder notwendig.

DAS PROJEKTFENSTER von unserem einfachen Player enthält die Bilder und den Quicktime-Film als Ressourcen.

Zurück in der Programmklasse öffnen wir das Unterprogramm „Run“ und rufen dort mit der Programmzeile „hauptfens ter.lade datei“ das Unterprogramm des Hauptfensters auf.

Nun muss das Unterprogramm „lade“ noch die Datei öffnen. Dazu benutzen wir die Funktion „openasmovie“. Vorher legen wir mit dem Befehl „dim film as movie“ eine Variable ab, in der wir den Film festhalten. Wir laden den Film aus der Datei mit der Zeile „film=datei.openasmovie“. Doch zu diesem Zeitpunkt wissen wir noch nicht, ob die Datei überhaupt einen Film enthält. Zum Glück nimmt uns auch hier Quicktime die Arbeit ab, denn es meldet uns „nil“ zurück, wenn es sich bei der Datei nicht um ein Quicktime-Dokument handelt. Andernfalls erhalten wir ein Movie-Objekt in der Variable zurück. In der nächsten Zeile brauchen wir also nur zu prüfen, ob wir „nil“ zurückbekommen oder nicht. Dazu benutzen wir: „If film<>nil then“, und weiter unten kommt schon mal ein „end if“ hin, um die Bedienungsabfrage zu schließen.

Zwischen den beiden Zeilen ist jetzt gesichert, dass es sich bei der Datei um einen Film handelt. Wir setzen die Höhe des Fensters mit „player.height=m.movieheight +14“ auf die Filmhöhe plus 14 Pixel für die Quicktime-Steuerleiste (eigentlich müssten es 16 Pixel sein, aber wir arbeiten hier ohne den Rand). Danach ermitteln wir die Breite. Wenn die Datei keinen Videokanal hat (beispielsweise bei einer reinen Audiodatei im MP3-Format), ist die Breite Null. Das ist unangenehm, da nichts mehr ins Fenster passt, wenn dessen Breite kleiner als 50 Pixel wird. Mit dem Vergleich „if film.movie width<50 then“ prüfen wir, ob die Breite zu klein ist und setzen sie danach mit „width=150“ auf 150 Pixel fest. Dabei wächst der Player automatisch auf die passende Größe, denn mit den Eigenschaften „LockLeft“ und „LockRight“ achtet Realbasic selbst automatisch darauf. Nach einem „else“-Befehl setzen wir die Alternative: „width=film.moviewidth“ ändert die Fenstergröße auf die Filmbreite. Nach dem obligatorischen „end if“ müssen wir den Film nur noch per „player.movie= film“ zuweisen. Durch diesen Befehl erscheint der Film im Fenster. Die Videokassette wird sozusagen in den Videorekorder eingelegt, nachdem wir sie aus dem Schrank (Festplatte) geholt haben.

Damit unser Programm funktioniert, müssen wir in den Projekteinstellungen („Bearbeiten“-Menü) noch einen Creator-Code vergeben. Spontan schreiben wir mal „test“ hinein, möchte man das Programm jedoch öffentlich verbreiten, muss man bei Apple einen freien Creator-Code beantragen. Den Code kann man zunächst frei wählen und dann auf der Apple-Internet-Seite http://developer.apple.com/dev/cftype/ überprüfen, ob er noch frei ist. Wenn ja, sollte man ihn schleunigst anmelden, bevor ihn jemand anderer wegschnappt. Aber Achtung, alle Codes, die nur aus Kleinbuchstaben bestehen, sind explizit für Apple reserviert.

Noch sind wir nicht ganz fertig. Fügen wir die Zeile „player.play“ ein, spielt unser Programm den Film direkt nach dem Ladevorgang ohne weiteres Zutun ab.

SO WIRD ES MAC-LIKE: DRAG-AND-DROP INS FENSTER

Eine schöne und Mac-typische Funktion ist Drag-and-drop auf Fenster. Unser Player soll also auch auf Dateien reagieren, die wir direkt in das Movieplayer-Fenster ziehen. Dazu müssen wir im Open-Event des Hauptfensters mit dem Befehl „AcceptFileDrop“ jeden Dateityp, den man ziehen darf, anmelden. Auszugsweise sieht das so aus:

```
AcceptFileDrop "video/quicktime"
AcceptFileDrop "video/avi"
AcceptFileDrop "audio/mpeg"
```

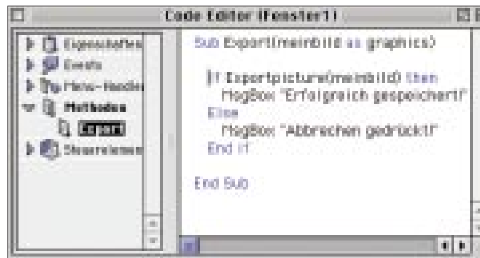
Wichtig ist, dass wir jeden Dateityp mit seinem genauen Namen einzeln nennen.

FILME HERAUSFILTERN

Der Event „DropObject“ bekommt beim Aufrufen einen Parameter „Obj“ vom Typ „DragItem“ übergeben. Dieser enthält das Folder-Item für die Datei. Da wir aber auch andere Dinge als nur Dateien per Drag-and-drop ziehen können, etwa Bilder, Texte oder weitere programmspezifische Daten, müssen wir uns unser Folder-Item selbst heraussuchen. Dazu gehen wir alle Objekte der Reihe nach durch und schauen nach, ob es sich dabei um ein Folder-Item und damit um unsere Datei handelt.

Wir benutzen eine „Do-Loop“-Schleife. Am Anfang steht die Zeile „Do“ als Einleitung einer Do-Loop-Schleife und am Schluss die Zeile „Loop until not Obj.Next Item“. Am Schleifenende rufen wir die Funktion „NextItem“ des Objekts auf. Diese Funktion liefert den Wert „true“ (wahr) zurück, so lange noch ein Objekt übrig ist, und lädt es dann. Gibt es kein weiteres Objekt mehr, bekommen wir den Wert „false“ (falsch) geliefert. Damit unser Programm die Schleife bei „false“ beendet, müssen wir über die logische Verknüpfung „not“ dafür sorgen, dass „true“ und „false“ quasi gegeneinander vertauscht werden (wir merken uns: „not false“ ergibt „true“ und „not true“ ergibt „false“). Und wenn wir jetzt als Ergebnis der Schleife insgesamt ein „true“ erhalten, beendet das Programm die Schleife, und wir können sicher sein, dass wir alle Objekte durchgesehen haben.

Innerhalb der Schleife müssen wir schauen, ob sich im aktuellen Stand des „DragItems“ ein „FolderItem“ befindet. Das fragen wir durch „if obj.folderitemavailable



MIT QUICKTIME lassen sich Pixel-Bilder und Grafiken auch wieder speichern. Die Funktion „ExportPicture" sorgt für die nötigen Speicherformate.

then" ab, denn wenn die Funktion „FolderitemAvailable" des „Drag Item" „true" meldet, haben wir eine Datei gefunden. Dann rufen wir per „lade obj.folderitem" unser Unterprogramm „lade" auf und übergeben ihm diese Dateireferenz als Parameter.

Wir wissen nun, dass wir mit dem „MoviePlayer"-Steuerelement Quicktime-Filme jeder Art abspielen können. Dabei ist das Format des Films weit gehend unerheblich, solange Quicktime es unterstützt. Darunter fallen sowohl die vielen „echten" Quicktime-Filme (Dateityp „MooV" oder Dateiendung „.mov"), als auch alle unterstützten Fremdformate, wie zum Beispiel „AVI"- oder „MPEG"-Filme. Ein Film muss aber nicht zwingend einen Videokanal haben, sondern kann auch aus einer reinen Audiokomponente bestehen. Das hilft uns, um etwa die beliebten MP3-Dateien abzuspielen. Diese sind genau genommen nichts anderes als Filme ohne Bildkomponente.

WEITERE IDEEN: STANDBILDER MIT QUICKTIME LADEN

Mit Hilfe von Quicktime können wir auch Standbilder in vielen Formaten laden und darstellen, indem wir uns ein „FolderItem" (beispielsweise über die Funktion „open asfolderitem") zu der Bilddatei besorgen. Die Zeile „dim bild as picture" reserviert eine Variable für ein Bild, die anschließend mit „bild=datei.openaspicture" gefüllt wird. Auch hier bekommen wir ein „nil" zurück, wenn die Datei kein Bild enthält. Ansonsten können wir mit dem Bild genauso umgehen, wie mit denen im Projektfenster (als Beispiel: „backdrop=bild").

Eine Liste der über 30 Formate, die Quicktime unterstützt, finden Sie unter http://www.apple.com/quicktime/authoring/fileformats.html.

STREAMING MIT REALBASIC: FILME ÜBER DAS INTERNET

Seit der Version 4 gibt es in Quicktime die Möglichkeit, Filme als Streams live über das Internet anzusehen. Realbasic unterstützt dies ebenfalls und bietet die Funktion „Open URLMovie". Damit öffnet man eine Internet-URL als Quicktime-Film und kann den Film anschließend genau so wie in unserem Beispiel in einem Fenster anzeigen. Als Parameter muss man lediglich die exakte Internet-URL angeben „film=OpenURLMovie("http://www.apple.com/quicktime/showcase/live/bbc")". Um den Rest kümmert sich wie immer Quicktime selbst. Wenn Sie unseren Movieplayer noch erweitern und diese Funktionen integrieren, können Sie ihn schon als vollständigen Ersatz für den von Apple mitgelieferten Quicktime-Movieplayer verwenden.

BILDER KONVERTIEREN LEICHT GEMACHT: GRAFIKEN EXPORTIEREN

In Realbasic kann man auch einfach Bilder in anderen Formaten speichern und so einen einfachen Bildkonverter aufbauen. Dazu dient der Befehl „ExportPicture", der viele Bildformate zur Verfügung stellt. Dabei übergeben wir der Funktion lediglich das Bildobjekt. Der Benutzer darf sich dann in einem Speicherdialog selbst aussuchen, in welchem Format er das Bild gerne gesichert hätte. Auf dieser Seite oben rechts sehen Sie ein kleines Programmcodebeispiel dazu.

TAUSENDSASSA QUICKTIME: MUSIK MACHEN MIT MIDI-INSTRUMENTEN

Was wir bislang noch nicht angesprochen haben, ist die Möglichkeit, die in Quicktime integrierten Midi-Instrumente in Aktion zu hören. Quicktime bietet 128 verschiedene Instrumente, von denen sich 32 gleichzeitig mit einem Tonumfang von 128 Tönen spielen lassen. Realbasic unterstützt auch dies.

Zudem bietet Realbasic die Möglichkeit, Quicktime-Filme zu bearbeiten. Das fängt bei MP3-Dateien an, bei denen man den Interpreten und den Titel des Stücks anzeigen und ändern kann, und geht bis hin zu einem aus einzelnen Bilddateien selbst erzeugten und animierten Quicktime-Film.

Für Diaschauen oder eigene Filme liefert Realbasic in Verbindung mit Quicktime auch eine Vielzahl von Überblendeffekten, wie man sie von iMovie her kennt.

FAZIT

Wegen der guten Resonanz erweitern wir die Realbasic-Serie um einen fünften Teil. Im nächsten Monat werden wir uns ausführlich mit der Spieleprogrammierung beschäftigen und wollen lernen, wie man mit der integrierten Sprite-Engine umgeht. *cm*

Serie Realbasic

1 Einführung ....**Heft 9/2000**
2 Taschenrechner im Eigenbau ....**Heft 10/2000**
3 Grafikprüfung ....**Heft 11/2000**
4 Quicktime-Programmierung ....**Heft 12/2000**
5 Sprites und Spiele ....**Heft 01/2001**

DAS HINTERGRUNDBILD 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 ANWENDUNGSPROGRAMM

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

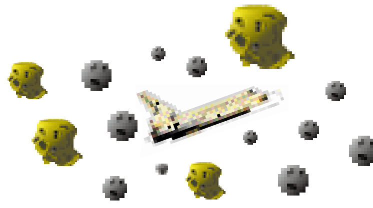
Bevor wir loslegen, benötigen wir die Objekte, das Raumschiff und die Asteroiden, als kleine Pixel-Grafiken. Wer will, kann sich – beispielsweise in Photoshop – selbst welche zeichnen. Der Einfachheit halber haben wir bereits einige Objekte vorbereitet. Sie befinden sich im Realbasic-Archiv auf der *Macwelt*-Internet-Seite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

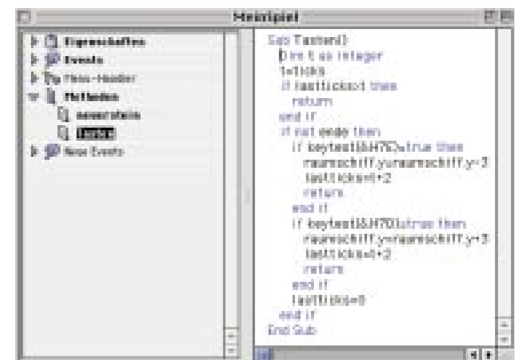
Dann erzeugen wir den Rahmen für das Spiel. Dazu benötigen wir ein Bild mit der Größe der Spielfläche. Die Fläche im Rahmen, durch die wir auf den Hintergrund schauen, muss dabei schneeweiß (RGB: 255, 255, 255) sein. Dann übermalt Realbasic diese Fläche nicht mit dem Rahmenbild, sondern zeichnet sie 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.

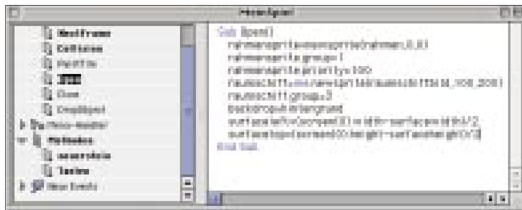
Bei allen Sprites mit positiven Gruppennummern erzeugt Realbasic beim Zusammenstoß einen Kollisions-Event. Diesen Event erzeugen Sprites mit der Gruppennummer 0 (das ist die Default-Nummer) niemals, während bei einem Sprite mit negativer Gruppe immer ein Kollisions-Event entsteht, egal mit welchem anderen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

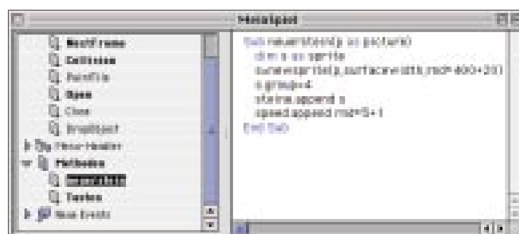
Wie aus der Mathematik (Analysis) gewohnt, greift man auf die einzelnen Elemente zu. Das fünfte Element erreichen wir mit „stein(5)“, das achte mit „stein(8)“ und so weiter, der Index steht einfach innerhalb der Klammern. Auch mehrdimensionale Felder sind möglich, „dim steine3D(5,10,3) as sprite“ erzeugt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

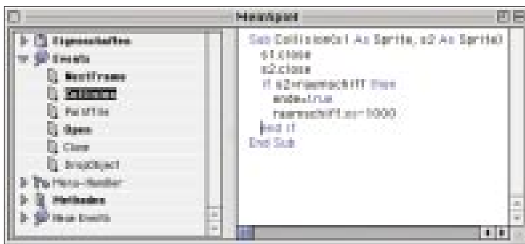
Nun geht es daran, unseren Steinen das Fliegen beizubringen. Dazu kehren wir in den „Nextframe“-Event unserer Spieleklasse zurück. Zunächst brauchen wir eine Zählvariable für eine Schleife. Dazu deklarieren wir „dim i as integer“. In der Schleife gehen wir die Liste der Steine vom Anfang bis zum Ende 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



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

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.

ZUSÄTZLICHES MATERIAL

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.

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*

Serie Realbasic

1 Einführung .... **Heft 9/2000**
2 Taschenrechner im Eigenbau .... **Heft 10/2000**
3 Grafikprüfung .... **Heft 11/2000**
4 Movieplayer im Eigenbau .... **Heft 12/2000**
5 Spiele-Programmierung .... **Heft 01/2001**

Simulieren mit Realbasic

von Christian Schmitz

Serie Realbasic, Folge 1 Nachdem unsere Realbasic-Serie für Einsteiger (ab Macwelt 9/2000) großen Anklang gefunden hat, wollen wir nun mit der Computer-Simulation des Spiels „Mensch ärgere Dich nicht!" tiefer einsteigen

EINSTIEG

Realbasic ist eine leicht zu erlernende und dennoch sehr mächtige Programmiersprache für den Mac. Auch absolute Programmieranfänger erhalten nach wenigen Minuten schon ein Erfolgserlebnis. In dieser Serie führen wir den fortgeschrittenen Realbasic-Programmierer Schritt für Schritt zu einer kompletten Brettspielsimulation.

✚ Wir beginnen mit einem neuen leeren Projekt. Vor uns sehen wir das noch leere Hauptfenster. Mit dem Menübefehl „Fenster:Eigenschaften einblenden" blenden wir die Palette „Eigenschaften" ein. Wir geben bei der Eigenschaft „Title" des Fensters den Text „Mensch ärgere Dich nicht!" ein.

Wie bei den meisten Brettspielvarianten möchten wir den Spielfeldhintergrund gelb einfärben. Dazu klicken wir auf die Checkbox bei der Eigenschaft „HasBackColor". Jetzt sollte der Hintergrund weiß sein. Damit er gelb wird, klicken wir auf die Farbe bei „BackColor" und wählen im Farbdialog ein helles Gelb. Am besten im Modus „RGB" mit 100 Prozent rot, 100 Prozent grün und 50Prozent blau.

Als Fensterbreite stellen wir 600 Pixel und als Höhe 458 Pixel ein. Dies geben wir bei den Eigenschaften Width und Height ein.

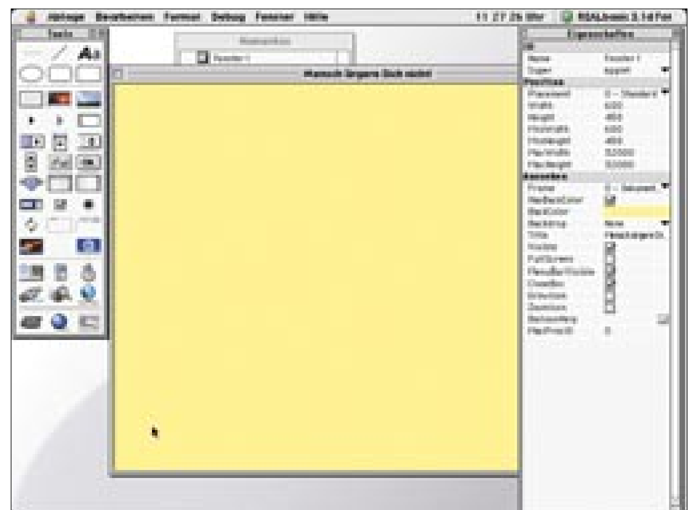
Ein Feld!

Für dieses Spiel brauchen wir verschiedene Felder auf dem Spielbrett. Zum Beispiel einen Feldtyp für den Weg, einen für das Haus und einen für das Ziel. Wir werden das Problem objektorientiert lösen. Es wird also eine Klasse für ein Feld geben, die es zeichnen soll. Alle weiteren Felder werden von dieser Klasse abgeleitet und können sich damit selber zeichnen. Wir vermeiden dadurch die Routine zum Zeichnen eines Feldes mehrmals zu programmieren. So spart man Arbeit!

Wir legen über den Menüpunkt „Ablage: Neue Klasse" eine neue Klasse an und nennen dieses „Feld" (Eigenschaft „Name"). Unterhalb der Eigenschaft „Name" gibt es in der Eigenschaften-Palette die Eigenschaft „Super", bei der wir in einem Pop-up-Menü nun den Eintrag „Canvas" auswählen. Dadurch wird unser Feld eine Erweiterung des normalen Canvas-Steuerelements.

Serie: Realbasics für Profis

1	Interface	Heft 12/2001
2	Computerspieler	Heft 01/2002
3	Mensch spielt mit	Heft 02/2002
4	Netzwerk	Heft 03/2002
5	Kompilieren und Release	Heft 04/2002



Spielbrett in Gelb Wie beim realen Vorbild „Mensch ärgere Dich nicht!" geben wir dem virtuellen Spielbrett eine gelbe Farbe.

Mit einem Doppelklick auf den Namen öffnen wir die Feld-Klasse im Projektfenster und landen im Code-Editor. Unter den Events finden wir die Events der Canvas-Klasse. Diese benutzen wir fortan. Jedes Feld soll eine Farbe als Hintergrund erhalten, wobei die Felder der Häuser eine andere Farbe bekommen als die Spielfelder. Wir fügen also der Klasse eine neue Eigenschaft hinzu. Mit dem Menübefehl „Bearbeiten:Neue Eigenschaft…" rufen wir den passenden Dialog auf und geben „Farbe as color" ein.

Feldlein, zeichne Dich!

Es gibt in der Feld-Klasse den Event (Ereignis) „Paint", den das System aufruft, wenn sich das Feld zeichnen soll. Das passiert zum Beispiel, wenn das Fenster zum ersten Mal erscheint, oder wenn ein anders Fenster darüber hinweg bewegt wird. Wir könnten aber auch einmal das Bedürfnis bekommen, diesem Feld sagen zu wollen, dass es sich neu zeichnen soll. Dafür wäre die Methode „Refresh" der Control

Klasse, von der das Canvas abstammt, geeignet. Wir raten aber davon ab, weil es stark flackert. Bei einem Refresh malt Realbasic zuerst den Fensterhintergrund und dann das Feld. Besser wäre, wenn wir das neue Feld direkt über das alte zeichnen. Wir benutzen daher eine eigene Methode namens „Draw“ um unser Feld zu zeichnen. Im Paint-Event schreiben wir deshalb nur

```
Draw g
```

Wir rufen dort unser Zeichnenunterprogramm auf und geben den Parameter „g“ für die Grafikumgebung weiter.
Für den Fall, dass wir selber neu zeichnen wollen, fügen wir über den Menübefehl „Bearbeiten:Neue Methode…“ eine neue Methode namens „Redraw“ ein. Diese ruft zwar auch „Draw“ auf, aber da wir kein „g“ zur Hand haben, nehmen wir die Grafikumgebung des Canvas selber. Mit der Zeile

```
Draw Graphics
```

greifen wir daher auf die Grafikumgebung unter dem Namen „Graphics“ zu.
Nun fehlt nur noch das eigentliche Unterprogramm „Draw“. Über den Menübefehl „Bearbeiten:Neue Methode…“ fügen wir ein Unterprogramm namens „Draw“ mit dem Parameter „g as graphics“ ein. Das Unterprogramm wird also in die jeweilige Grafikumgebung zeichnen. Das g im Paint-Event hat nichts mit der Graphics-Eigenschaft des Canvas zu tun! Wir übergeben dem Unterprogramm die jeweils aktuelle Umgebung.
Als erstes weisen wir mit der Anweisung

```
g.foreColor=Farbe
```

die aktuelle Zeichenfarbe auf die Farbe, die wir in der Variable „Farbe“ gespeichert haben. Mit der Zeile

```
g.fillOval 0,0,width,height
```

zeichnen wir anschließend ein gefülltes Oval in der Größe des Feldes. Die zwei Zeilen

```
g.foreColor=rgb(0,0,0)
```

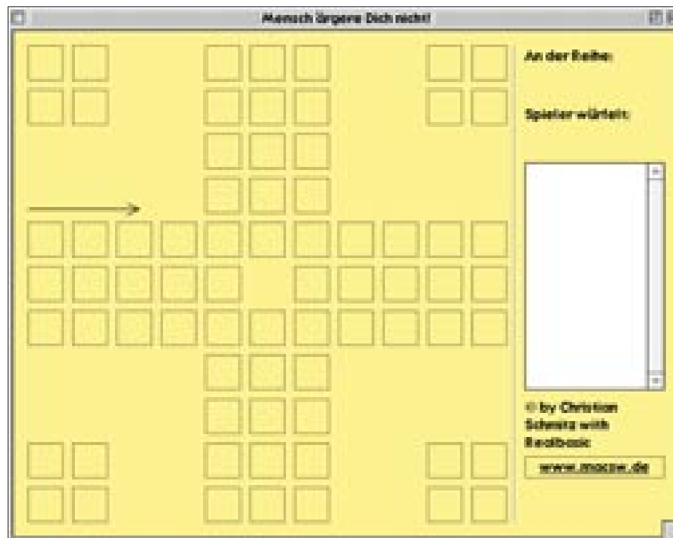
und

```
g.drawOval 0,0,width,height
```

zeichnen dann noch einen schwarzen Rand um das Feld.

Wir wollen was sehen!

Natürlich soll unser Feld auch mal in Erscheinung treten. Wir wechseln ins Hauptfenster und ziehen ein Canvas-Steuerelement aus der Palette hinein. Jetzt ändern wir die Eigenschaft mit dem Namen „Super“ von „Canvas“ auf „Feld“. Alternativ können wir auch direkt die Klasse aus dem Projektfenster in das Hauptfenster ziehen. Realbasic legt dann automatisch ein neues Canvas an und ändert die Super-Klasse. Wir drücken „Befehl-R“ (Run) und sehen einen schwarzen Kreis. Schwarz, weil die Variable Farbe von dem Feld keinen Wert bekommen hat und der Defaultwert aller Farben RGB(0,0,0) also schwarz ist.



Spielbrett im Editor Wenn wir alle Felder korrekt positioniert haben, sollte unser Spielfeld im Editor so aussehen.

Häuslebauer

Wir schieben unser erstes Feld nach links oben bis zu den Hilfslinien. Die Left- und Top-Eigenschaften betragen dann jeweils 13. Nun ändern wir die Feldgröße auf 32 mal 32 Pixel (Width und Weight auf 32 setzen). Wir positionieren alle Felder so, dass sie bei einer Koordinate liegen, die die Ziffer drei am Ende hat. Außerdem nehmen wir einen Abstand von 40 Pixeln von einer Ecke zur nächsten. Dazu ziehen wir ein Feld mit gedrückter Alt-Taste (Option) nach rechts. Dabei kopieren wir es und schieben es so, dass es bei left= 53 liegen bleibt. Nun kopieren wir beide Felder zusammen nach unten, so dass diesmal top= 53 wird. Jetzt sollten wir vier Canvas-Steuerelemente mit der Super-Klasse „Feld“ haben. Allen geben wir den Namen „bhaus“ (für blaues Haus). Realbasic fragt uns beim Zweiten, ob wir ein Controlarray anlegen wollen. Diese Frage beantworten wir mit „ja“.

Nun haben wir vier Steuerelemente, die alle unter demselben Namen ansprechbar sind. Um sie zu unterscheiden, benutzt man einen Index. Dieser läuft hierbei von „bhaus (0)“ bis „bhaus(3)“. Realbasic zählt die Index-Eingenschaft jedes Steuerelements automatisch durch.

Wir klicken ein Feld doppelt an und landen im Code-Editor im Mousedown-Event des Feldes. Wie wir sehen, gibt es dort nur einen Eintrag für „bhaus“. Wir klicken weiter unten auf den Open-Event des Feldes. Rechts oben im Programmcode steht „sub open(index as integer)“. Realbasic ruft diesen Event vier mal auf und übergibt jedes Mal die Nummer des Feldes. Hier können wir das Feld noch vor seinem Erscheinen verändern. Das nutzen wir und fügen die Zeile

```
me.farbe=rgb(100,100,255)
```

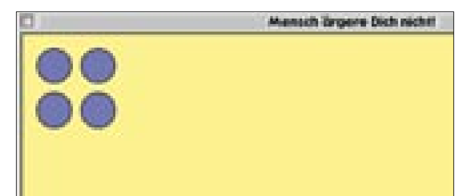
ein. Mit „me“ verweisen wir auf das jeweils aktuelle der vier Steuerelemente. „Farbe“ ist unsere Variable für die Farbe des Feldes. Wir ändern sie hier auf ein helles Blau. Wenn wir nun unser Programm starten sollten wir das unten stehende Bild sehen.

Nachbarhäuser

Nun selektieren wir die vier Felder im Hauptfenster und ziehen sie mit gedrückter Optionstaste (Alt) nach unten links in die Ecke. Damit kopieren wir alle vier Felder. Sie sollten bei der Position (left= 13, top= 373) liegen. Wir benennen vier neuen Häuser um in „rhaus“. Mit einem Doppelklick auf ein solches Feld kommen wir in den Codeeditor und ändern den Event „Open“. Da diese Felder rot werden sollen, ergänzen wir dort die Zeile

```
me.farbe=rgb(255,100,100)
```

Nun kopieren wir auf die gleiche Art alle acht Felder nach rechts, so dass die Left-Eigenschaft der am weitesten links liegenden Felder 373 beträgt. Jetzt liegen in allen



Aller Anfang ist schwer Noch ist nicht viel zu sehen auf unserem Spielfeld. Wir beginnen mit den blauen „Haus“-Feldern in der linken oberen Ecke.

vier Ecken jeweils vier Felder. Die Felder unten rechts nennen wir „ghaus“ für „Grünes Haus“ und die Felder oben rechts „yhaus“ für „Yellow Haus“. (yellow = englisch für Gelb). Die Farben der Felder ändern wir wie gehabt mit

```
me.farbe=rgb(100,255,100)
```

in ein Grün, beziehungsweise mit

```
me.farbe=rgb(255,255,100)
```

in ein Gelb.

Das Spielfeld

Das Spielfeld setzt sich aus 40 aneinandergelegten Feldern zusammen. Alle Felder heißen „Brett“ und sind der Reihe nach von 0 bis 39 durchnummeriert. Für Höhe und Beite nehmen wir wieder 32 Pixel. Die Felder bilden eine Art Stern mit vier Schlaufen. Anschließend fügen wir wie beim Originalspiel die Zielfelder ein. Dazu setzen wir vier Felder in jede der vier Schlaufen. Diese nennen wir analog zu den Haus-Feldern „Ziel“ (gZiel, rZiel, bZiel, yZiel).
Für die korrekten Farben schreiben wir in die Open-Events noch jeweils die Zeilen:

```
bZiel.open: me.farbe=rgb(100,100,255)
gZiel.open: me.farbe=rgb(100,255,100)
yZiel.open: me.farbe=rgb(255,255,100)
rZiel.open: me.farbe=rgb(255,100,100)
```

Dekoration im Fenster

Das Spielfeld ist noch recht kahl. Damit man weiß in welche Richtung sich die Figuren bewegen, ergänzen wir einen Richtungs-Pfeil.

Dazu ziehen wir dreimal eine Linie aus der Toolbar in unser Fenster. Passende Koordinaten für die Linien wären (13, 113, 161, 161), (102, 112, 166, 161) und (103, 113, 156, 161). Die drei Linien ergeben dann einen langen Pfeil, der nach rechts zeigt.

Den Pfeil kann man durch eine schöne Grafik ersetzen. Rechts auf die Seite des Fenster kommen noch einige Informationsanzeigen. Um sie optisch abzutrennen, nehmen wir eine Trennlinie, in Realbasic der Separator. Als Koordinaten nehmen wir left= 449, top= 13, width= 4 und height= 432. Separator-Steuerelemente sind immer vier Pixel hoch oder breit, damit das OS die Linie je nach eingestelltem Erscheinungsbild erkennen kann.

In dem abgetrennten Bereich positionieren wir oben vier statische Texte. Zunächst den Text „An der Reihe:“ und darunter ein leeres Feld mit dem Namen „spielername“. Analog dazu dann „Spieler würfelt:“ und das Feld „wurfel“.

Ergebnisse zum Mitschreiben

Jeder möchte gerne wissen, was während des Spielablaufs passiert, deshalb ergänzen wir eine Protokollfunktion, die alle Spielzüge mitschreibt. Dazu brauchen wir eine Listbox unter den Texten. Diese bekommt eine Spalte und einen vertikalen Scrollbalken (keine Überschrift). Wenn wir hierzu die rechte Listbox aus der Toolbar benutzen, sind diese Eigenschaften bereits voreingestellt, während die linke Listbox aus der Toolbar andere Einstellungen beinhaltet.

Unsere Listbox bekommt den Namen „Liste“ und die Koordinaten left= 461, top= 119, width= 126, height= 206 zugeteilt.

Nun brauchen wir ein Unterprogramm, das einen Text in die Protokollliste einfügt. Wir erzeugen dazu eine neue Methode namens „log“ mit dem Parameter „logtext as string“. Als einzige Codezeile ergänzen wir in dieser Methode

```
liste.insertRow 0,logtext
```

Dadurch fügt Realbasic den Text in der Liste ganz oben an der ersten Position (Realbasic zählt hier ab der Ziffer Null!) ein.
Um die Methode zu testen, können wir bei

> **Aufruf | Ihr Spiel auf CD**
>
> Das in dieser Folge programmierte Interface für „Mensch ärgere Dich nicht!“ lässt sich mit eigenen Grafiken je nach Geschmack deutlich aufbessern. Wenn Sie mögen, schicken Sie uns Ihre eigene Kreation per E-Mail an: redaktion@macwelt.de Wir prämieren die gelungensten Variationen, präsentieren sie online und packen sie auf die jeweils aktuelle *Macwelt* Abo-CD!

der Feld-Klasse im Event „MouseDown“ eine Zeile ergänzen

```
window1.log me.name+" "+str(me.index)
```

Wir rufen aus dem Feld heraus die Log-Methode des Fensters „window1“ auf. Als Text übergeben wir den Namen von dem Feld und daran angehängt ein Leerzeichen und die laufende Nummer des Steuerelements. Die Nummer wird als Zahl gespeichert, daher müssen wir sie erst durch die Funktion Str() in einen Text umwanden.

Copyright und Link ins Web

Ein Copyright-Vermerk sichert die Rechte an dem Programm, dazu setzen wir einen statischen Text unter der Listbox. Er enthält einen Text ähnlich zu „© 2001 by Christian Schmitz und Macwelt“. Das Copyright Zeichen erreicht man mit Option-G.

Praktisch ist ein anklickbarer Link zur *Macwelt*-Homepage. Dazu benutzen wir ein Canvas mit einem darüberliegenden statischen Text. Wir ziehen das Canvas rechts unten in die Ecke und dazu einen Statiktext, so dass beide genau übereinander liegen. Der Text bekommt den Inhalt „www.macwelt.de“ und passend dazu setzen wird die Eigenschaft „underline“ auf true. Im Canvas unter dem Text schreiben wir in den „MouseDown“-Event

```
return true
```

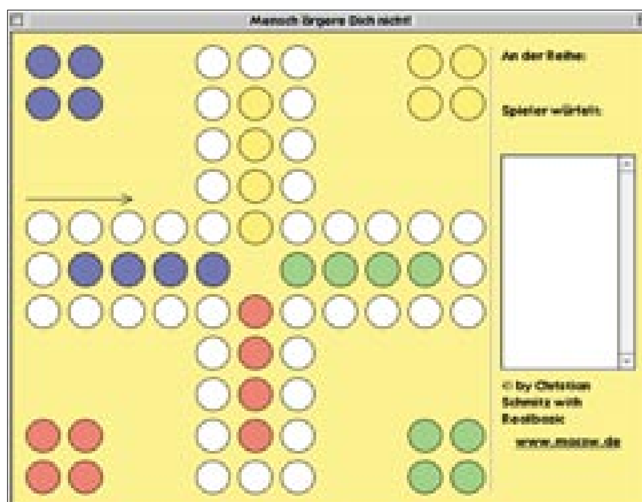
Dadurch erklären wir dem System, dass wir uns selbst um diesen Mausklick kümmern. Genau wie bei KeyDown und MenüEvents wird der Event solange weitergereicht, bis sich jemand verantwortlich fühlt.
Im MouseUp Event ergänzen wir die Zeile

```
showurl "http://www.macwelt.de"
```

Dieser Befehl zeigt eine URL an. Das System ruft automatisch den Webbrowser auf, der im Internet-Kontrollfeld eingestellt ist.

Ausblick

In der nächsten Folge kümmern wir uns eingehend um die künstliche Intelligenz des Computergegners. *cm*

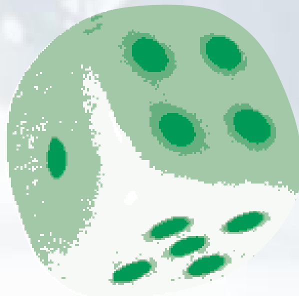


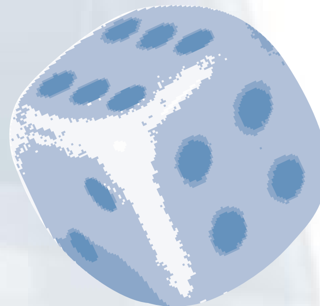
Spielbrett zur Laufzeit Wenn wir nach getaner Arbeit unser Programm zum Test starten, erscheint dieses fertige Spielfeld.

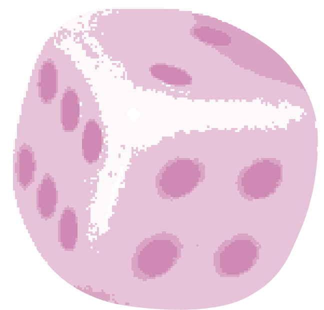
Der Computerspieler

von Christian Schmitz

Serie Realbasic, Folge 2 In dieser Folge programmieren wir den Computergegner für die Computer-Simulation von „Mensch-ärgere-Dich-nicht“. Am Schluss spielt diese komplett automatisch mit allen vier Spielern gegen sich selbst







EINSTIEG

Die aktuelle Serie zeigt, wie man in Realbasic ein Brettspiel à la „Mensch ärgere Dich nicht“ erstellt. In der ersten Folge haben wir die Spieloberfläche entworfen, in dieser Folge entwickeln wir vier Computerspieler, die gegeneinander „Mensch ärgere Dich nicht“ spielen.

ZUNÄCHST SORGEN wir dafür, dass wir die Felder auf dem Spielbrett im Programmcode besser unterscheiden können. Dafür legen wir unter der Superklasse „Feld“ drei Klassen an „Hausfeld“, „Zielfeld“ und „Brettfeld“. Im Fenster ändern wir passend die Superklassen der Felder für die Häuser, die Zielfelder und die Felder des Bretts.

Wenn jetzt alle Felder nicht mehr von der Klasse „Feld“ abgeleitet werden, dann kann man später im Programmcode einfach testen zu welcher Art ein Feld gehört. Nehmen wir an, dass wir eine Variable „f“ vom Typ Feld haben, in der irgendein Feld abgelegt ist. Mit der Zeile „If f isa Zielfeld Then“ kann man testen, ob dieses Feld von der Klasse Zielfeld stammt. Natürlich stimmt „If f isa Feld Then“ auch.

Figuren-Klasse

Für die Daten der Spielfiguren legen wir eine Klasse namens „Figur“ an. Sie bekommt eine Eigenschaft „Farbe as Color“ für die Farbe der Spielfigur. Dazu dann noch eine Eigenschaft „Feld as Feld“ für einen Verweis auf das aktuelle Feld, auf dem die Figur steht.

Wenn wir im Programmcode eine Figur erzeugen, dann sollten wir direkt beim Erzeugen der Figur eine Farbe angeben. Für eine Zeile wie „f = New Figur(Farbe)“, bei der wir direkt die Farbe übergeben, brauchen wir einen so genannten Konstruktor. Ein Konstruktor ist ein kleines Unterprogramm, dass aufgerufen wird, wenn ein Objekt aus einer Klasse erzeugt wird. Passend dazu wird ein Destruktor aufgerufen, wenn das Objekt aus dem Speicher aufgelöst wird. In Realbasic benennt man den Konstruktor genauso wie die Klasse. Vor den Namen des Destruktors stellt man zusätzlich ein „~ “. Für unsere Figur heißen damit der Konstruktor „Figur“ und der Destruktor „~ Figur“.

Nun erstellen wir in der Klasse „Figur“ eine neue Methode mit dem Namen „Figur“ und dem Parameter „c as Color“ (Menübefehl „Neue Methode…“ im Menü „Bearbeiten“). Mit der Zeile „Farbe = c“ in dieser Methode sorgen wir dafür, dass man keine Figur erzeugen kann, ohne Ihr eine Farbe zu geben.

Es ist eine gute Angewohnheit, Variablen, die nur lokal beschränkt verwendet werden, einen kurzen Namen zu geben. Dabei orientiert man sich am Namen des Variablentyps und zählt bei Bedarf im Alphabet weiter. Wer also ein „i“ oder ein „j“ sieht, kann davon ausgehen, dass es ein Integer ist oder ein „c“ eine

Serie: Realbasics für Profis

1	Interface	Heft 12/2001
2	Computerspieler	Heft 01/2002
3	Mensch spielt mit	Heft 02/2002
4	Netzwerk	Heft 03/2002
5	Kompilieren und Release	Heft 04/2002

Farbe (Color) und ein „s“ ein Text (String). Dabei sollte man aber direkt auf einen Blick sehen können, wo sie deklariert wurden. Bei Variablen mit größerer Reichweite nimmt man längere beschreibende Namen wie zum Beispiel „Farbe“ in der Klasse „Figur“.

Spieler-Klasse

Um alle Spielerdaten zusammen zu halten, erzeugen wir eine neue Klasse namens „Spieler“. Diese Klasse bekommt mehrere Eigenschaften. Eine Farbe mit „Farbe as Color“, der Name des Spielers mit „Name as String“ und drei Zahlenwerte mit „Würfel as integer“, „Runden as integer“ und „Offset as integer“. In „Würfel“ speichern wir die zuletzt gewürfelte Zahl des Spielers. In der Eigenschaft „Runden“ halten wir die noch verbleibenden Würfelversuche für den Spieler fest. Am Anfang ist das eine drei, bis der Spieler eine Figur im Spiel hat. Bei einer gewürfelten sechs bekommt er einen Versuch mehr. Mit „Offset“ vermerken wir, ab welchem Index in den Feldern des Spielbretts das jeweilige Heimfeld anfängt.

Immer der Reihe nach!

Die Spieler sollen der Reihe nach spielen. Dazu setzen wir einen Timer in das Hauptfenster. Wir ziehen also das Steuerelement mit dem Bild einer Uhr aus der Toolbar in das Hauptfenster. Ein Doppelklick auf dieses Steuerelement im Fenster bringt uns in den Code-Editor. Dort öffnen wir den „Action“-Event des Timers. Als Code fügen wir nur eine Zeile mit dem Inhalt „Nextplayer“ ein. Die Methode „Nextplayer“ wird den nächsten Spieler auswählen und seinen Spielzug veranlassen.

Wir legen nun eine Methode „Nextplayer“, eine Eigenschaft „Currentplayer as Spieler“ und eine Eigenschaft „SpielerCounter as integer“ an. Currentplayer hält einen Verweis auf den aktuellen Spieler. SpielerCounter speichert, welcher Spieler an der Reihe ist.

Die erste Aufgabe von Nextplayer ist es, herauszufinden, ob ein Spieler schon im Spiel ist und noch eine Runde zu spielen hat. Das passiert zum Beispiel nachdem er eine sechs gewürfelt hat. Mit der Zeile „If Currentplayer = Nil or currentplayer.runden< = 0 Then“ prüfen wir, ob noch kein Spieler ausgewählt wurde. Der Vergleich „currentplayer.runden< = 0“ kontrolliert, ob der aktuelle Spieler keinen Wurf mehr hat. In beiden Fällen gehen wir zum nächsten Spieler über.

Falls noch kein Spieler dran war (Currentplayer ist noch Nil), nehmen wir einfach den Spieler 0 („Currentplayer = Spieler(0)“). Ansonsten zählen wir den SpielerCounter eins höher. Wenn SpielerCounter größer als die Liste der Spieler wird, setzen wir den Zähler wieder auf 0. Die Zeile „Currentplayer = Spieler(SpielerCounter)“ wählt den aktiven Spieler aus und mit der Zeile „Spielername.Text = Currentplayer.Name“ schreiben wir den Namen des aktiven Spielers in das Fenster.

Spieler rekrutieren

Noch existiert kein Spieler. Das ändern wir aber schnell, indem wir eine neue Eigenschaft „Spieler(3) as Spieler“ anlegen. „Spieler“ ist jetzt ein Feld von 0 bis 3 für insgesamt 4 Spieler. Eine neue Methode „NeuesSpiel“ im Hauptfenster mit folgenden Zeilen legt vier Spieler an:

```
Spieler(0)=New Spieler("Blau", RGB(0,0,255),0)
Spieler(1)=New Spieler("Gelb",
RGB(255,255,0),10)
Spieler(2)=New Spieler("Grün", RGB(0,255,0),20)
Spieler(3)=New Spieler("Rot", RGB(255,0,0),30)
```

Damit existiert nun pro Farbe ein Spieler mit dem Namen der Farbe als Spielername, der eigentlichen Farbe und der jeweiligen Position im Spielfeld. Doch den Konstruktor der Klasse „Spieler“ gibt es noch nicht. Dazu legen wir in der Klasse Spieler eine neue Methode mit dem Namen „Spieler“ und den Parameter „n as String“, „c as Color“, „o as Integer“ an. Die übergebenen Werte speichern wir dann in den Variablen Name, Farbe und Offset ab.

Im Open Event vom Fenster sollten wir jetzt noch zum Starten des Spieles die Methode „NeuesSpiel“ aufrufen. Außerdem sollte man dort noch festlegen, dass die jeweiligen Startfelder der Spieler die selbe Farbe wie die Hausfelder haben:

```
Brett(0).Farbe = bHaus(0).Farbe
Brett(10).Farbe = yHaus(0).Farbe
Brett(20).Farbe = gHaus(0).Farbe
Brett(30).Farbe =
rHaus(0).Farbe
```

Protokollant Unser Protokoll speichert alle Ereignisse im Spiel in einer Liste und zeigt diese chronologisch an. Das aktuelle Ereignis steht dabei immer an der obersten Stelle.

Wenn wir das Programm nun starten, wechselt oben rechts in dem Textfeld sekündlich der Name des Spielers, der an der Reihe ist.

Alea iacta est – Der Würfel ist gefallen

Nun kommt der Zufall ins Spiel. Zunächst bekommt die Klasse Spieler eine neue Methode namens „Würfeln“. Diese Methode enthält die Würfelformel: „Würfel = rnd*6+ 1“. Rnd generiert eine Fließkomma-Zufallszahl, die größer gleich 0 und kleiner 1 ist. Wir nehmen diese Zahl mal sechs und addieren eins dazu. Da Würfel vom Typ „Integer“ ist, schneidet Realbasic die Nachkommastellen ab. Wir bekommen so gleichförmig verteilte Zufallszahlen von Eins bis Sechs.

Im Hauptfenster in der Methode „Nextplayer“ schreiben wir nun weiter. Zuerst versichern wir uns mit einem „If Currentplayer < > Nil Then“, dass ein Spieler an der Reihe ist. Dieser Spieler bekommt als erstes eine Runde abgezogen „currentplayer.runden= currentplayer.runden-1“. Anschließend rufen wir die Würfeln-Routine dieses Spielers auf und lassen ihn würfeln.

Das Ergebnis veröffentlichen wir zuerst im Textfeld unter dem Spielername mit der Zeile „Würfel.Text = Str(Currentplayer.Würfel)“. Als nächstes ergänzen wir das Protokoll um dieses Ergebnis. „Log Currentplayer.Name+ ” würfelt “+ Str(Currentplayer.Würfel)“.

Wer eine Sechs würfelt, sollte noch mal würfeln. Dazu testen wir, ob „Currentplayer. Würfel“ gleich sechs ist. Wenn ja, setzen wir „Currentplayer.Runden“ wieder um eins rauf.

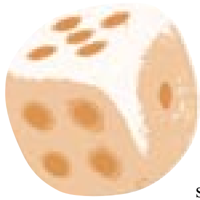
Jedem Spieler seine Figuren

Jeder Spieler braucht vier Figuren, die wir am Anfang in die vier Häuser setzen. Zuerst bekommt die Klasse Spieler eine neue Eigenschaft namens „Figur(3) as Figur“. Im Konstruktor füllen wir die Figuren mit den Zeilen:

```
Figur(0)=New Figur(Farbe)
Figur(1)=New Figur(Farbe)
Figur(2)=New Figur(Farbe)
Figur(3)=New Figur(Farbe)
```

Im Hauptfenster in der Methode „NeuesSpiel“ fehlen noch einige Variablen, denen wir Startwerte übergeben. Vor dem bisherigen Code sollten wir für gleich schon einmal eine Zählvariable für die For-Next-Schleife deklarieren: „Dim I as integer“. Anschließend löschen wir die Protokoll-Listbox mit „Liste.DeleteAllRows“, löschen den Verweis zum aktuellen Spieler mit „Currentplayer = Nil“, setzen den SpielerCounter auf Null.

Dies alles dient dazu, einen Menüeintrag „Neues Spiel“ zu erzeugen und damit die Reste vom letzten Spiel zu löschen. ►



Figuren ins Haus!

Nun schreiben wir noch einen weiteren Code, um jedem Spieler seine Häuser zu zeigen und dort seine Spielfiguren hinzustellen. Mit einer Schleife von null bis drei bekommt jeder Spieler seine Häuser und die Zielfelder zugewiesen:

```
For i=0 to 3
  Spieler(0).haus(i)=bHaus(i)
  Spieler(1).haus(i)=yHaus(i)
  Spieler(2).haus(i)=gHaus(i)
  Spieler(3).haus(i)=rHaus(i)
  Spieler(0).ziel(i)=bZiel(i)
  Spieler(1).ziel(i)=yZiel(i)
  Spieler(2).ziel(i)=gZiel(i)
  Spieler(3).ziel(i)=rZiel(i)
Next
```

Als nächstes stellt jeder Spieler seine Figuren ins Haus. Dazu rufen wir die Methoden „MoveHaus“ der Spieler auf, die wir noch anlegen.

```
Spieler(0).MoveHaus
Spieler(1).MoveHaus
Spieler(2).MoveHaus
Spieler(3).MoveHaus
```

Nun legen wir in der Klasse Spieler eine Methode „MoveHaus“ an. Diese Methode benutzt eine For-Next-Schleife, um allen Figuren den Spieler und das Haus zuzuweisen.

```
Dim i as integer
For i=0 to 3
  Figur(i).Spieler = me
  Figur(i).MoveTo haus(i)
Next
```

In der Klasse Spieler fehlt noch die Eigenschaften „Haus(0) as Hausfeld“ und „Ziel(3) as Zielfeld“. Damit kann jeder Spieler direkt auf seine Felder zugreifen.

Figur, setz dich da hin!

Wir übermitteln der Figur zuerst die Information, dass sie zu einem bestimmten Spieler gehört. Dafür deklarieren wir in der Klasse Figur die Eigenschaft „Spieler as Spieler“ und die Eigenschaft „Feld as Feld“, um einen Verweis auf das aktuelle Feld zu speichern.

Nun folgt die Methode „MoveTo“ mit dem Zielfeld als Parameter „f as Feld“. Dank der objektorientierten Programmierung akzeptiert MoveTo auch ein Ziel- oder Hausfeld.

Wenn das Zielfeld nicht undefiniert ist („If f < > Nil Then“), sehen wir nach, ob diese Figur schon auf einem Feld ist („If Feld < > Nil Then“). In diesem Fall löschen wir den Verweis auf die Figur in diesem Feld und lassen es neu zeichnen („Feld.Redraw“). Jetzt setzen wir die aktuelle Figur mit „f.Figur = me“ in das neue Feld. Mit „f.Redraw“ zeichnen wir das Feld neu und speichern es mit „Feld = f“. Eben haben wir eine Eigenschaft vom Feld benutzt, die es noch nicht gibt. Wir ergänzen also in der Klasse Feld die Eigenschaft „Figur as Figur“. So weiß jedes Feld, was auf ihm sitzt.

Jedes Feld besitzt eine Draw-Methode und in dieser darf die Figur gezeichnet werden. Wenn also eine Figur zum Feld gehört („If Figur < > Nil Then“), dann setzen wir die Malfarbe der Grafikumgebung auf die Farbe der Figur „g.ForeColor = Figur.Farbe“. Wir zeichnen ein gefülltes Oval mit dem Befehl „g.Fill Oval 3, 3, width-6, height-6“. Dieses Oval ist kreisrund und hält zum Rand vom Feld einen Abstand von 3 Pixel. Wir wechseln die Malfarbe nun mit „g.ForeColor = RGB(0,0,0)“ auf Schwarz und zeichnen mit „g.DrawOval 3, 3, width-6, height-6“ über das gefüllte Oval ein ungefülltes als Rand.

Methoden für die Figur

Unsere Figuren sind jetzt noch recht unbeweglich. Manchmal muss eine Figur zurück ins Haus. Dazu geben wir der Klasse Figur eine Methode „MoveHaus“. Wir deklarieren eine Laufvariable mit „Dim i as integer“. In der For-Next-Schleife prüfen wir die vier Hausfelder des Spielers („For i= 0 to 3“), ob dort Platz ist („If Spieler.Haus(i).Figur = Nil Then“). Stoßen wir auf ein Hausfeld, das keine Figur hat, bewegen wir die Figur mit der Zeile „MoveTo Spieler.Haus(i)“ in das Haus. Mit dem Befehl „Exit“ brechen wir die Schleife ab.

Eine weitere Hilfsfunktion ist „ShowAt“. Wenn die Figur später wandert, sollen für einen kurzen Moment die Felder auf dem Wanderweg aufleuchten. Dies erreichen wir, indem wir auf dem Feld kurz (1/6tel Sekunde) die Figur gegen unsere austauschen. Wir deklarieren eine Methode „ShowAt“ mit dem betreffenden Feld als Parameter „f as Feld“.

In der Methode brauchen wir eine Variable, um die Figur des Feldes für den Tauschvorgang zwischen zu lagern („Dim old as Figur“). Zusätzlich benötigen wir noch eine Variable für die Startzeit („Dim I as integer“). Bis das Zielfeld definiert ist („If f < > Nil Then“), speichern wir den Verweis auf die aktuelle Figur des Feldes in der Variable Old („Old = f.Figur“). Danach setzen wir unsere aktuelle Figur mit „f.Figur = me“ in das Feld und zeichnen es mit „f.Redraw“ neu.

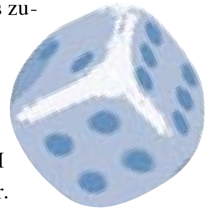
Eine Pause von 1/6tel Sekunde erhalten wir mit einer leeren Schleife. Wir messen die aktuelle Zeit in Ticks (ein Tick entspricht einer 1/60tel Sekunde) und zählen 10 Ticks hinzu. Wir benutzen eine While-Schleife, die so lange läuft, bis I größer als Ticks wird. Der Code dazu sieht so aus:

```
I = Ticks +10
While I > Ticks
Wend
```

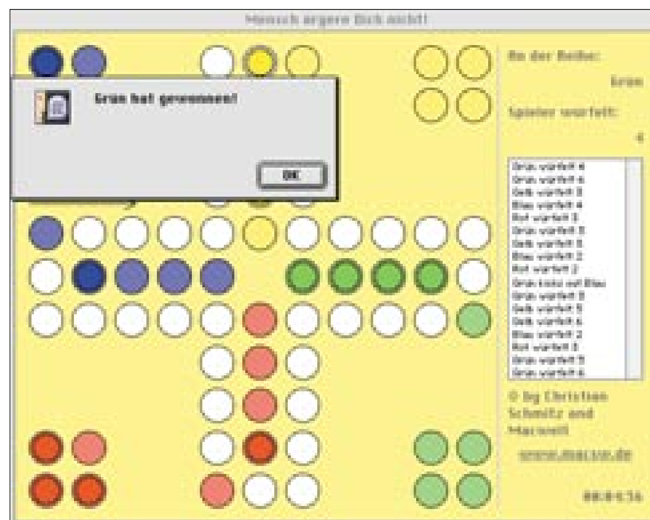
Zum Schluss restaurieren wir wieder mit „f.Figur = Old“ und „f.Redraw“.

Methoden für den Spieler

Auch die Klasse Spieler benötigt noch ein paar Befehle. Hilfsroutinen, die wir später aufrufen. Zuerst wäre da die Funktion „Alleimhaus“, die ein Boolean-Ergebnis zurückliefert. Es wird True, wenn der Spieler alle Figuren im Haus hat, sonst bleibt es False. Wir deklarieren zwei Variablen I und Z vom Typ Integer. I für die Schleife und Z als Zähler.



Für jede Figur („For I = 0 to 3“) prüfen wir, ob ihr Feld ein Hausfeld ist „If Figur(i).Feld isa Hausfeld Then“. Der Test mit isa klappt immer. Wenn das Feld nicht definiert ist (Wert Nil), dann kommt Falsch als Ergebnis zurück. Aber wenn es stimmt, zählen wir den Zähler um eins weiter („Z = Z + 1“).



Selbstläufer Das in dieser Folge fertiggestellte Programm läuft vollständig mit allen Regeln des Spiels. Vier Computerspieler spielen dabei gegeneinander. In der nächsten Folge kommt dann auch ein Mensch ins Spiel.

Mit der Zeile „Return Z = 4“ melden wir, ob alle vier Figuren im Haus sind.

In der Spieler-Klasse benötigen wir noch eine Funktion, die eine Figur aus dem Haus zurückgibt. Wir legen also eine Methode „FigurImHaus“ mit dem Rückgabetyp „Figur“ an.

In einer Schleife von 0 bis 3 gehen wir die Hausfelder durch und schauen, ob eine Figur drin ist („If haus(i).Figur < > Nil Then“). Wenn ja, melden wir diese als Ergebnis zurück: „Return Haus(i).Figur“.

Außerdem deklarieren wir schon für später eine Variable „ZielCount as integer“ und eine Methode „GeheFelder“ mit einem Parameter „c as integer“. Diese Methode wird die Figur auf dem Spielbrett bewegen. Die Variable „ZielCount“ zählt dann die Anzahl der Figuren im Zielbereich für jeden Spieler.

Bewegung auf dem Spielbrett

Den Code zum Würfeln schreiben wir in die Methode Nextplayer des Hauptfensters. Wir deklarieren dazu zwei Variablen „n“ und „war“ vom Typ „Figur“.

Wenn eine Sechs gewürfelt wird, setzen wir „Currentplayer.Runden“ auf eins. So lange ein Spieler noch Figuren im Haus hat, muss er bei einer Sechs eine Figur aufs Spielfeld setzen. Dazu rufen die Funktion „FigurImHaus“ des Spielers am Zug auf, lassen uns einen Verweis auf eine Figur im Haus geben und speichern diesen in „n“. Wenn wir keine Figur im Haus mehr finden, rufen wir „Currentplayer. GeheFelder 6“ auf und bewegen so die Figur um sechs Felder weiter.

Falls wir eine neue Figur auf das Brett stellen müssen, prüfen wir vorher, ob auf dem Startfeld vor dem Haus schon eine Figur steht. Die Zeile „War = Brett (Currentplayer.Offset).Figur“ fragt die Figur des Startfelds ab und kopiert einen Verweis in die Variable War. Wenn War ein gültiger Verweis auf eine Figur ist und diese Figur nicht dem aktuellen Spieler gehört („If War < > Nil and War.Spieler < > Currentplayer Then“), schreiben wir ins Protokoll: „log Currentplayer.Name + “kicks out“ + War.Spieler.Name“, schicken die Figur mit „War.MoveToHaus“ heim und leeren das Feld mit. „Brett(Currentplayer.Offset).Figur = Nil“.

Nun prüfen wir noch einmal, ob das Feld leer ist „If Brett(Currentplayer.Offset).Figur = Nil Then“, und bewegen unsere Figur N mit „n.MoveTo Brett(Currentplayer.Offset)“ dort hin. Falls vorher auf dem Feld noch eine Figur steht, muss es die eigene sein und wir lassen den Spieler mit „Currentplayer.GeheFelder 6“ einen normalen Zug machen.

Falls keine Sechs gewürfelt wurde, soll die Figur die gewürfelte Anzahl an Feldern vorrücken. Die Zeile „Currentplayer.GeheFelder CurrentPlayer. Würfel“ erledigt dies. Der Methode „GeheFelder“ fehlt noch eine Funktion, die prüft, wie das nächste gültige Feld heißt.

Mit Uhrzeit In der finalen Fassung unserer Simulation bauen wir auch noch eine Zeitanzeige ein, für die wir in dieser Ausgabe leider keinen Platz mehr haben.

Suche nach dem nächsten Feld

Wir erzeugen dazu eine neue Methode im Hauptfenster mit dem Namen „NextFeld“, den Parametern „S as Spieler, F as Feld“ und dem Rückgabetyp „Feld“. Wir wollen hiermit für den Spieler S vom Feld f aus das nächste freie Feld für die Figur suchen.

Auch hier brauchen wir eine Laufvariable („Dim i as integer“). Wir prüfen, ob das Feld f ein Brettfeld ist („If f isa Brettfeld Then“). Innerhalb der 40 Brettfelder zählen wir nun mit „I = f.Index + 1“ hoch. „Index“ ist dabei der Index des Steuerelements.

Hier stoßen wir auf ein Problem. I kann den Wert 40 erreichen, doch Brett(40) existiert nicht mehr. Falls das I also 40 erreicht, setzen wir es einfach wieder auf Null.

Wenn I mit dem Offset des Spielers identisch ist, also der Nummer des Feldes, bei dem der Spieler anfängt, dann hat diese Figur eine Runde geschafft und wir schauen in die Zielfelder. In einer Schleife gehen wir die Zielfelder durch und suchen einen leeren Platz:

```
For i=0 to 3
    If s.Ziel(i).Figur = Nil Then
        Return s.Ziel(i)
    End If
Next
```

Falls wir ein Feld finden, auf dem keine Figur steht, geben wir dieses Feld als Ergebnis der Funktion zurück. Wenn der Spieler noch nicht fertig war, dann melden wir mit „Return Brett(i)“ das nächste Brettfeld zurück. Falls F jedoch kein Brettfeld ist, geben wir mit „Return F“ das selbe Feld wieder zurück.

Achtung! In dieser Funktion benutzen wir die Variable i für zwei unabhängige Dinge. Normalerweise schafft man sich damit sehr schnell Bugs, aber es spart nun mal ein paar Bytes Arbeitsspeicher.

Gehe-Felder

Jetzt endlich legen wir die Methode „GeheFelder“ mit dem Parameter „C as integer“ in der Klasse Spieler an. Mit dieser Methode bewegt der Spieler seine Figur um C Felder.

Wir legen zuerst zwei Variablen vom Typ Figur mit den Namen „F“ und „War“ an. Dann noch ein Feld mit dem Namen „B“ und ein Integer „i“ für die Schleifen.

Nun gehen wir alle Figuren des Spielers durch und suchen eine Figur auf dem Brett. Falls wir eine finden, legen wir sie in F ab.

Haben wir eine Figur gefunden, kopieren wir in B das aktuelle Feld der Figur mit „B = F.Feld“. Dann rufen wir die Methode NextFeld von 1 bis C im Hauptfenster auf rücken mit „B = window1. nextfeld (Me, B)“ jedesmal um ein Feld weiter. Wir kennen damit in B das Zielfeld unserer Wanderung. Falls dieses Zielfeld definiert ist („If B < > Nil Then“), legen wir zuerst die Figur, die dort steht, in der Variable War ab. Wenn War leer ist bzw. nicht zu einer der Figuren des Spielers gehört („If War = Nil or War.Spieler < > me Then“), können wir den Spielzug zu Ende führen.

Dazu starten wir wieder mit „B = F.Feld“ und gehen wie oben die Felder entlang. Diesmal rufen wir jedoch nach dem NextFeld-Aufruf die Methode „F.ShowAt b“ auf und zeigen damit kurz die Spielfigur auf dem Feld.

Wenn das letzte „B“ Feld ein Zielfeld sein sollte („If b isa Zielfeld Then“), dann hat der Spieler einen Punkt gemacht, den wir in der Variable „ZielCount“ mitzählen.

Falls nun in der Variable War eine Figur definiert ist („If War < > Nil Then“), dann dürfen wir diese Figur Heim schicken. Mit „window1.log me.Name+ “kicks out“ + War.Spieler.Name“ dokumentieren wir das und führen es mit „War.MoveToHaus“ aus. Schließlich stellen wir die Figur mit „f.MoveTo b“ an Ihren Zielort.

Falls der Spieler alle vier Figuren im Ziel versammelt hat, dann hat er gewonnen! Wir schreiben „If me.zielcount= 4 Then“ und zeigen den Sieger mit. „Msgbox me.name + “hat gewonnen!“ “ in einer Dialogbox an.

Fazit

Wir sind fertig mit dem Computerspiel. Mit der 3D-Umgebung aus Realbasic 3.5 könnte man dem Spiel auf Wunsch auch räumliche Tiefe verleihen. In der nächsten Folge darf dann der Mensch mitspielen. *cm*

Der Mensch spielt mit!

Serie Realbasic, Folge 3 In diesem Teil spielen wir als Programmierer und Anwender endlich selbst mit. Bis es soweit ist, sollte unser Spiel etwas freundlicher werden *von Christian Schmitz*

EINSTIEG

Realbasic ist eine einfach zu erlernende Programmiersprache, in der auch Anfänger ihre ersten Programmiererfahrungen sammeln können. Mit dieser Serie können Sie ein Brettspiel à la „Mensch ärgere Dich nicht" programmieren. In dieser Folge binden wir den Anwender als Spielpartner ein.

✚ **WIR BEGINNEN DAMIT**, die Menüleiste etwas aufzumöbeln. Wir hätten gerne einen Eintrag für eine neues Spiel, einen Eintrag für ein neues Spielbrett und einen Eintrag zum Schließen eines Fensters.

Wir öffnen das Menü im Projektfenster. Dort steht „File" beziehungsweise „Ablage" für das erste Menü. Für Programme, die keine Dateien erzeugen oder öffnen, empfiehlt Apple in den Human Interface Guidelines (Richtlinien für die Gestaltung von Benutzeroberflächen) dieses Menü anders zu nennen. Wir nennen es einfach „Spiel", denn dort soll man ein neues Spiel starten können beziehungsweise ein Laufendes beenden.

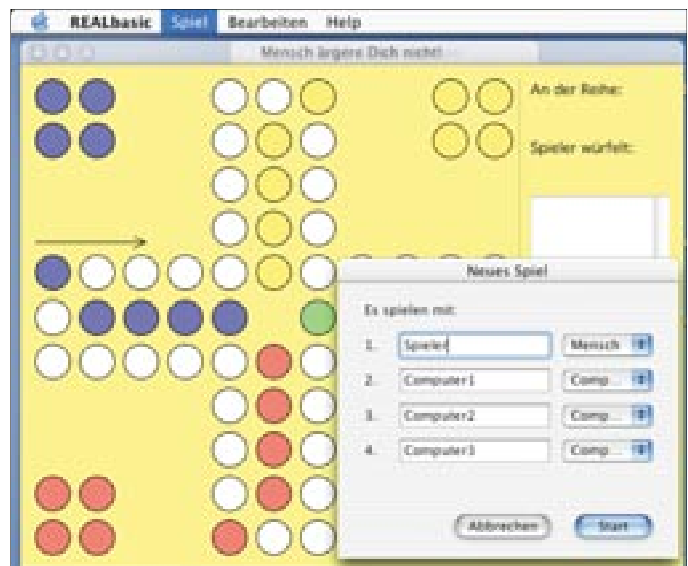
Wir klicken auf „File" bzw. „Ablage" und in der Eigenschaften-Palette erscheinen die Eigenschaften des MenuItems. Dort geben wir bei Text „Spiel" ein. Die Konsequenz sehen wir direkt im Menüfenster.

Wir legen nun einen neuen Menüeintrag an. Dazu klicken wir unten im Menü auf den leeren Eintrag und tippen den Text „Neues Spiel…" ein. Als Name sollte Realbasic automatisch „SpielNeuesSpiel" eintragen. Sollten bei Ihrer Realbasic-Version Punkte am Namen auftauchen, müssen Sie diese auf jeden Fall entfernen, denn die Namen für Variablen dürfen keine Punkte enthalten. Beim Commandkey geben wir als Kurzbefehl „N" ein. Bitte immer nur Großbuchstaben dort eingeben, denn bei Befehlskürzel unterscheidet man nicht zwischen Groß- und Kleinschrift.

Der nächste Menüeintrag wird „Neues Spielbrett..." lauten und heißt im Code dann analog „SpielNeuesSpielbrett". Im Spiel darf man ein Spielfenster jederzeit schließen und ein neues Spiel öffnen. Für dieses MenuItem tragen wir beim Commandkey „Shift-N" ein. Der Benutzer muss also zusätzlich noch die Umschalttaste drücken. Falls wir mit einer englischen Realbsic-Version arbeiten, sollten

Serie: Realbasics für Profis

1	Interface	Heft 12/2001
2	Computerspieler	Heft 01/2002
3	Mensch spielt mit	Heft 02/2002
4	Netzwerk	Heft 03/2002
5	Kompilieren und Release	Heft 04/2002



Nomen est omen In dieser Ausgabe programmieren wir unter anderem einen Einstellungsdialog, mit dem alle Spieler Namen bekommen.

wir nun alle Menüeinträge eindeutschen (Bearbeiten: Widerrufen, Ausschneiden, Kopieren, Einfügen, Löschen). Außerdem benennen wir das Spielfenster in „SpielFenster" um. Zunächst ändern wir den Namen in der Eigenschaften-Palette und anschließend ersetzen wir ihn im gesamten Code. Dazu benutzen wir den Suchen-Dialog. Wir suchen nach „Window1." und ersetzen es durch „SpielFenster."

Neues Spiel auf Wunsch

Ein neues Spiel soll starten, wenn man im Menü „Spiel" den passenden Menüeintrag auswählt. Wir öffnen das Spielfenster im Codeeditor und dort den Open-Event. In diesem Event steht bereits die Zeile „NeuesSpiel", die wir jetzt löschen.

Stattdesssen legen wir einen neuen Menühandler für den Menüeintrag „SpielNeues Spiel" an. Dorthin schreiben wir „NeuesSpiel" als Befehl und anschließend „Return true" um Realbasic zu sagen, dass dieser Menüeintrag von uns bearbeitet wurde.

Im Event „EnableMenuItems" müssen wir diesen Menüeintrag noch aktivieren. Dazu ergänzen wir dort die Zeile „SpielNeues Spiel.enable".

Nun kontrollieren wir noch einmal, ob alle Timer im Fenster auf Mode 0 stehen, denn sonst gibt es Laufzeitfehler im Programm. Damit man ein laufendes Spiel schließen kann, legen wir einen weiteren Menühandler für

„SpielSchliessen“ an und ergänzen die Zeilen „Close“ und „Return true“. Im EnableMenuItems-Event schreiben wir „SpielSchliessen.enable“.

Neues Fenster

Für die Menübefehle, die auch ohne Fenster funktionieren sollen, benötigen wir eine Klasse vom Typ „Application“ in der wir die Menühandler versorgen. Dazu legen wir eine Klasse an, geben ihr den Namen „App“ und wählen als Superklasse „Application“ aus. Im NewDocument-Event der Klasse legen wir ein neues Spielbrett mit den folgenden Zeilen an:

```
Dim f as SpielFenster
F = New SpielFenster
```

Starten wir nun das Programm, entdecken wir zwei geöffnete Spielbrett-Fenster, da in den Projekteinstellungen das Spielfenster als Default ausgewählt ist. Das ist jetzt nicht mehr nötig, weshalb wir dort „none“ auswählen.

Global und lokal

Nun müssen wir noch eine größere Änderung vornehmen, denn an mehreren Stellen im Code sprechen wir das Fenster über den Namen „Spielfenster“ an. Das funktioniert gut, solange es nur ein Fenster gibt aber bei mehreren Fenstern ist es nicht eindeutig. Realbasic legt beim Zugriff auf ein Fenster über dessen Namen automatisch dieses Fenster an, weshalb wir bisher problemlos Methoden wie SpielFenster.NextFeld aufrufen können. Um aber gleichzeitig mit mehreren Fenster spielen zu können, statten wir einige Objekte mit Verweisen auf ihr eigenes Fenster aus.

In der Spieler-Klasse legen wir dazu eine neue Eigenschaft „Fenster as SpielFenster“ an und ändern alle Zugriffe auf das Spielfenster innerhalb dieser Klasse um auf „Fenster“. Der Constructor Spieler bekommt einen zusätzlichen Parameter f mit dem Verweis auf das Spielfenster. Die Parameterzeile sieht dann so aus: „n as String, c as Color, o as integer, f as SpielFenster“. Mit der Zeile „Fenster = f“ speichern wir den Verweis für später. In unserem Spielfenster ändert sich dadurch in der Methode „NeuesSpiel“ der Aufruf des Constructors für die neuen Spieler. Ein „„self“ als weiterer Parameter übergibt dem Spieler bei seiner Erschaffung einen Verweis auf sein Fenster.

Zwei Spiele nebeneinander

Um ein neues Spielbrett anzulegen, erzeugen wir in unserer App-Subklasse einen neuen Menühandler „SpielNeuesSpielbrett“ mit folgendem Inhalt:

```
NewDocument
Return true
```

Wir rufen hier lediglich den Event „NewDocument“ auf, der dann eigenständig das neue Fenster anlegt.

Ein Bug bei NeuesSpiel!

Lassen wir das Spiel nun laufen, fällt uns beim Neustart eines Spiels ein kleiner Bug auf. Die „alten“ Figuren auf dem Spielbrett werden nicht entfernt, bevor das neue Spiel startet. Also ergänzen wir in der Feld-Klasse die Methode „Clear“ die folgenden Code beinhaltet:

```
Figur = Nil
Redraw
```

Hierdurch löschen wir den Verweis zur Figur und zeichnen das Feld neu. In der Methode „NeuesSpiel“ ergänzen wir zudem:

```
For i=0 to 39
    Brett(i).Clear
Next
For i=0 to 3
    yZiel(i).Clear
    rZiel(i).Clear
    gZiel(i).Clear
    bZiel(i).Clear
Next
```

Damit haben wir den Bug beseitigt.

Jedem Spieler seinen Namen

Jeder Spieler sollte einen eigenen Namen bekommen. Dazu konstruieren wir ein spezielles Dialogfenster. Wir benötigen in diesem Fenster vier Eingabefelder für die Namen der vier Spieler und dahinter jeweils ein Pop-up-Menü für „Mensch“ oder „Computer“. Das Ganze sollte mit Texten (und eventuell auch Bildern) geschmückt werden. Rechts platzieren wir einen Knopf mit der Beschriftung „Start“. Links daneben einen „Abbrechen“-Knopf. Im Codefenster dieses Dialogs ergänzen wir eine Eigenschaft „Okay as Boolean“. Der Code für den Action Event vom OkayButton lautet:

```
Okay = true
Hide
```

Beim Abbrechen-Knopf schreiben wir:

```
Okay = false
Hide
```

In beiden Fällen speichern wir zunächst, ob der Anwender den Okay-Knopf gedrückt hat und blenden den Dialg dann wieder aus. Würden wir den Dialog ganz schließen (Close), wüssten wir später nicht mehr, ob die Variable Okay true oder false ist. Genau genommen schließen wir diesen Dialog erst beim Programmende, denn wenn wir ihn beim nächsten Aufruf von NeuesSpiel wieder zeigen, besitzt er immer noch alle Eingaben vom letzten Mal. Wir benennen unser neues Dialogfenster „NeuesSpielDialog“.

Der Eingabedialog verursacht ein paar Probleme

Die Abfrage der Spielernamen gehört in die Methode „NeuesSpiel“. Wir setzen dort direkt hinter den Dim-Zeilen die Zeile „NeuesSpielDialog.showmodal“. Dadurch erscheint der Dialog. Spätestens beim zweiten Mal sieht man, dass das Spiel im Hintergrund weiter läuft. Wir müssen also vor dem Aufruf des Di- ►

Tipp | Ideen für den Ausbau der Simulation

Im momentanen Zustand ist das Spiel zwar spielbar, aber man könnte einiges verbessern, was jedoch den Rahmen unserer Serie sprengen würde. Hier ein paar Anregungen:

- Unsere Beispiele sind mit einer Uhr als nette Dekoration ausgestattet. (Tipp: das Date Objekt liefert die aktuelle Zeit)
- Man könnte die Spielfarben frei wählbar machen. (Tipp: der SelectColor-Befehl hilft)
- Die Computergegner könnten zufällig Namen aus einer (editierbaren) Liste von Namen bekommen. (Tipp: Rnd- und nthField-Funktion helfen)
- Alle Einstellungen wie die Spielernamen könnte man für das nächste Spiel in einer Preferencesdatei speichern (Tipp: BinaryStream über ein Child-Folderitem vom PreferencesFolderitem)
- Das Spiel ist nicht 100%nach den Regeln. Am Ziel kommt man mit jeder möglichen Würfelzahl ins Häuschen. Ändern Sie das ab, so dass es nur bei passender Würfelzahl funktioniert.
- Den Würfelvorgang könnte man grafisch als eine Art Glücksrad gestalten, das man per Maus anhält oder das einfach langsamer wird.
- Eine Anzeige der möglichen Züge und deren Risiken beziehungsweise Vorteile. Ähnlich zu den Tipps im Schachprogramm von Apple.

alogs das Spiel anhalten. Die zwei verantwortlichen Timer halten wir mit folgenden zwei kurzen Code-Zeilen an:

```
Timer1.Mode = 0
Timer2.Mode = 0
```

Weiter unten stehen folgende zwei Zeilen, die wir direkt hinter dem Aufruf des Dialogs benötigen. Indem man den Modus eines Timers auf 2 setzt, startet man ihn und genau das brauchen wir, wenn der Dialog wieder verschwindet.

```
Timer1.Mode = 2
Timer2.Mode = 2
```

Das alte oder das neue Spiel läuft nun weiter, unabhängig davon, ob der Anwender „Ok“ oder „Abbrechen“ drückt. Beim Abbruchknopf sollte jedoch kein neues Spiel starten, sondern das alte weiterlaufen. Wir prüfen also die gespeicherte Okay-Eigenschaft des Dialogs und beenden die Methode mit Return, falls Abbrechen gedrückt wird:

```
If Not NeuesSpielDialog.Okay Then
    Return
End If
```

Damit stoßen wir wieder auf ein kleines Problem: Die Timer werden auch dann aktiviert, wenn noch gar kein Spiel läuft, was später zu Laufzeitfehlern führt. Wir müssen also feststellen, ob der Anwender das erste Spiel starten will oder ob er einen Spielneustart auslöst. Dazu definieren wir in dieser Methode eine Variable „ErstesSpiel“ als Boolean mit: „Dim ErstesSpiel as Boolean“.

Bevor wir die Timer auf 0 setzen, sollten wir über die Zeile „ErstesSpiel = Timer1.mode = 0“ den aktuellen Timerzustand abfragen und in dieser Booleschen Variable sichern. Kommt True heraus, dann lief vorher kein Timer und damit auch kein Spiel.

Nun müssen wir noch die Zeilen, mit denen wir die Timer starten, ändern, damit sie beim ersten Spiel nicht aufgerufen werden. Aber Achtung! Wenn wirklich ein Spiel gestartet werden soll, dann müssen die Timer auch laufen, weshalb die folgende Bedingung etwas komplizierter ist: Auf Deutsch lautet die Bedingung: „Wenn es nicht das erste Spiel ist oder ein Spiel gestartet werden soll, dann“. In Realbasic sieht das so aus:

```
If Not ErstesSpiel or
NeuesSpielDialog.Okay Then
    Timer1.Mode = 2
    Timer2.Mode = 2
End If
```

Die Namen ins Spiel übernehmen

Ganz unten in dieser Methode sehen wir noch einige Zeilen aus der letzten Folge. Diese ändern wir durch folgenden Code so ab, dass sie die Spielnamen aus dem Dialog übernehmen.

```
Spieler(0) = New Spieler(NeuesSpielDia
log.Editfield1.text, rgb(0,0,255), 0, self)
Spieler(1) = New Spieler(NeuesSpielDia
log.Editfield2.text, rgb(255,255,0), 10, self)
Spieler(2) = New Spieler(NeuesSpielDia
log.Editfield3.text, rgb(0,255,0), 20, self)
Spieler(3) = New Spieler(NeuesSpielDia
log.Editfield4.text, rgb(255,0,0), 30, self)
```

Die Namen wollen wir im Spiel auch anzeigen, weshalb wir im Hauptfenster vier Static-Texte in die Nähe der Häuser setzen und sie „NameFeld“ nennen. Die Zeile: „NameFeld(i). Text = Spieler(i).Name“ tragen wir in eine der Schleifen (von 0 bis 3) in der Methode „Neues Spiel“ ein. Dadurch füllen wir die Spielernamen mit dem Inhalt aus dem Dialog.

Spieler: Mensch oder Computer?

Beim Programmstart ist jeder Spieler zunächst automatisch ein Computerspieler. Mittels einer zusätzlichen Eigenschaft der Klasse Spieler mit dem Namen „Mensch“ und dem Typ „Boolean“ ändern wir dies. Wenn also ein Spieler ein Mensch ist, soll diese Eigenschaft den Wert True bekommen.

In der Methode „NeuesSpiel“ holen wir uns die Information, „Mensch oder Computer“ aus dem Dialog und halten sie in der Eigenschaft fest. Man kann die Wahl zwischen Mensch und Computer im Dialog durch Checkboxen, Radiobuttons oder Pop-up-Menüs realisieren. Wir entscheiden uns hier für vier Pop-up-Menüs und benutzen zur Abfrage folgenden Code den wir ganz am Ende der Methode „NeuesSpiel“ eintippen:

```
Spieler(0).Mensch =
NeuesSpielDialog.PopupMenu1.listindex = 0
Spieler(1).Mensch =
NeuesSpielDialog.PopupMenu2.listindex = 0
Spieler(2).Mensch =
NeuesSpielDialog.PopupMenu3.listindex = 0
Spieler(3).Mensch =
NeuesSpielDialog.PopupMenu4.listindex = 0
```

Wählt man nun im Pop-up-Menü den Eintrag 0 für „Mensch“ aus, wird dieser Spieler zum Menschen erklärt.

Interaktive Spielzüge

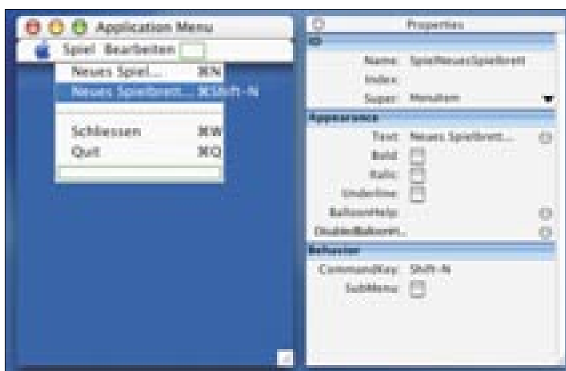
Bis zu vier Menschen dürfen mitspielen. Dafür müssen wir kurz überlegen, wie das Mitspielen von statten geht: Der Computer würfelt für alle. Wenn man eine Sechs würfelt und eine Figur ins Spiel muss, dann braucht der Spieler nichts zu tun, der Rechner erledigt das für ihn. Lediglich die Figuren, die schon im Spiel sind, muss der menschliche Spieler „von Hand“ bewegen. Wird bei einem Spieler die Routine „GeheFelder“ aufgerufen, muss im Falle, dass dieser Spieler ein Mensch ist, das Spiel stoppen (Timer ausschalten).

Wir warten, bis der Spieler auf ein Feld vom Brett klickt. Hat dieses Feld eine Figur (Feld.Figur <> Nil) und gehört die Figur zum fraglichen Spieler (Feld.Figur.Spieler = Currentplayer), dann soll der Computer für den Spieler einen Zug mit der betreffenden Figur machen. Damit wir unnötige Arbeit vermeiden, teilen wir die GeheFelder-Methode auf und schaffen eine zweite Methode namens „GeheMitFigur“. Sie führt einen Zug für Computer oder Mensch mit einer vorgegbenen Figur durch. In GeheFelder rufen wir dann diese Methode für eine Computerfigur auf oder halten das Spiel für den Menschenzug an.

Die Umsetzung ist nicht besonders schwer, findet aber in dieser Ausgabe keinen Platz mehr. Auf der Abo-CD beziehungsweise im Internet finden Sie unseren Lösungsvorschlag. Damit läuft das Spiel schon vollständig und inklusive menschlichen Spielern.

Fazit

In dieser Folge haben wir gelernt, wie man menschliche Spieler in unsere Simulation einbaut. In der nächsten Ausgabe wollen wir uns mit der Vernetzung der Simulation befassen. Man kann unser Spiel dann mit mehreren Partnern an verschiedenen Orten übers Internet gleichzeitig spielen. *cm*



Tagesmenü In Realbasic konstruieren wir nicht nur Fenster, sondern auch das Programmmenü inklusive der Tastatur-Befehlskürzel (hier Befehl–Umschalt–N).

Multiplayermodus für das Internet

Serie Realbasic, Folge 4 Internet-Spiele sind Kult, und warum nicht mal eine Partie „Mensch ärgere Dich nicht" mit einem Partner in Asien oder in den USA spielen? In dieser Folge programmieren wir die dafür nötigen Netzwerkgrundlagen nach dem Client/Server-Prinzip

von Christian Schmitz

EINSTIEG

Realbasic ist eine leicht zu erlernende Programmiersprache, mit der auch Anfänger schnell zurecht kommen. In dieser Folge lernen wir, wie wir die „Mensch ärgere Dich nicht"-Simulation für das Internet anpassen, um es mit Spielern rund um die Welt zu spielen.

FÜR UNSER NETZWERKSPIEL beschränken wir uns zunächst auf zwei Spieler. Wir legen dazu ein neues Fenster mit dem Titel „Netzwerkspiel" an, in das wir einige Steuerelemente, wie in der Abbildung auf Seite 67 zu sehen, platzieren. Der Schließknopf unten rechts führt lediglich den Befehl „hide" aus, um das Fenster verschwinden zu lassen.

Wir dienen dem Imperium

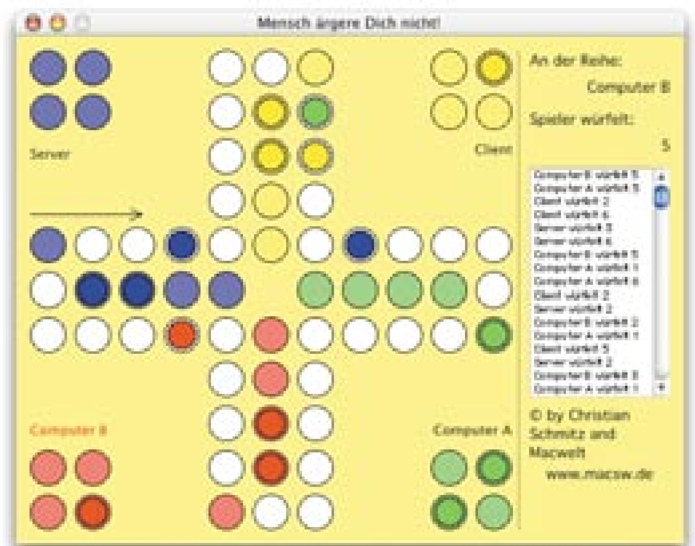
In unser Spielfenster legen wir ein Socket-Steuerelement und geben ihm den Namen „sock". In der Eigenschaftenpalette stellen wir für das Socket den Port 20 000 ein, damit die Kontaktaufnahme nur zwischen unseren Spielen funktioniert und wir keinen anderen Serverdienst (zum Beispiel: Webserver auf Port 80) behindern. Im Netzwerkdialog brauchen wir eine weitere Zugriffsmöglichkeit auf dieses Socket, weshalb wir eine Eigenschaft „sock as socket" anlegen und diese später mit einem Verweis auf das Socket „sock" im Spielfenster verbinden. Wir erhalten dadurch zwei Referenzen auf ein Objekt.

Der Startknopf für den Server ruft den Realbasic-Befehl „Sock.Listen" auf und teilt damit dem Socket mit, dass es auf eingehende Anforderungen warten soll. Jedes andere Gerät im Netzwerk, das unsere IP-Adresse erreicht, kann nun eine Verbindung zu diesem Socket aufbauen.

Damit wir im Spiel wissen, ob wir Server oder Client sind, legen wir zwei Eigenschaften „client as boolean" und „server as boolean" im Codeeditor des Netzwerkdialogs an.

Um den Server im Netzwerkdialog starten zu können und automatisch die anderen Buttons zu deaktivieren, fügen wir den Code aus Listing 1 in den Action-Event des Startknopfs ein. Passend dazu folgt im Action-Event des Stoppbuttons der Code aus Listing 2.

Das Socket lässt sich damit nun schon auf Empfang einstellen und wieder beenden. Damit der Dialog im Programm erscheint, ergänzen wir im Menü einen Eintrag für das Netzwerkspiel, den wir im Spielfenster mit dem Netzwerkdialog verbinden (siehe Listing 3). Die Zeile „netzwerkspiel.sock = sock" übergibt dabei eine Referenz auf das Socket im Spielfenster an den Einstelldialog. Der Server soll in einem Textfeld seine eigene Adresse



Server und Client Hier sieht man, wie zwei Computergegner, ein Server und ein Client, gemeinsam über das Internet ein Spielchen wagen.

Serie: Realbasics für Profis

1	Interface	Heft 12/2001
2	Computerspieler	Heft 01/2002
3	Mensch spielt mit	Heft 02/2002
4	Multiplayer	Heft 03/2002
5	Kompilieren und Release	Heft 04/2002

(IP) anzeigen. Die aktuelle IP des Computers findet man in der Eigenschaft „Localaddress“, so dass die Zeile „EigeneIP.text= sock.LocalAddress“ die IP-Adresse des Mac in das Textfeld „EigeneIP“ schreibt. Da aber der Netzwerkdialog kein eigenes Socket hat, müssen wir das Gleiche noch mal im Menü-Handler machen.

Wir sollten nicht vergessen, den Menü-Handler im Event „EnableMenuItems“ des Spielfensters durch eine Zeile „spielnetz.enabled= not isnetzwerkSpiel and not isnetzwerkSetup“ zu aktivieren.

Der Client meldet sich an

Wenn der Server läuft, wollen wir uns mit einem Client anmelden. Dazu brauchen wir ein paar Zeilen Code im ConnectButton-Action-Event, wie uns Listing 4 zeigt. Hier deaktivieren wir zunächst alle Buttons und das Eingabefeld, dann leiten wir die eingegebene Adresse an das Socket und starten einen Verbindungsversuch. Nun müssen wir mit einer Reaktion des Sockets rechnen, weshalb wir im Spielfenster ein paar Variable für den aktuellen Status benötigen. Wir nehmen die Variablen „IsNetzwerkSetup as boolean“, um zu vermerken, dass der Netzwerkdialog mit korrekten Daten ausgefüllt ist. Die Variable „IsNetzwerkSpiel as boolean“ wird wahr, wenn ein Netzwerkspiel läuft. Vor der Zeile „netzwerkspiel.showmodal“ setzen wir die Variable „IsNetzwerkSetup“ auf True und nach dem Aufruf des Dialogs wieder auf False. Man beachte den Unterschied zwischen .show und .showmodal, denn bei .show folgt direkt die nächste Programmzeile während bei .showmodal der Rechner darauf wartet, dass der Anwender den Dialog schließt. Wie diese Routine aussieht finden wir in Listing 5.

Schauen wir jetzt einmal auf das Socket im Spielfenster. Dieses Socket verfügt über vier Events wobei wir den Error-Event zuerst bedienen. Es gibt über 100 Gründe (Onlinehilfe) warum der Error-Event aufgerufen wird, uns interessiert jedoch nur, dass bei einem Fehler keine IP-Verbindung zu Stande gekommen ist. Dies passiert, wenn entweder auf der Gegenseite kein Server-Socket auf eine Verbindung wartet oder die IP nicht stimmt. Tritt also ein Fehler auf, während der Netzwerkdialog geöffnet ist, dann soll unser Programm den Fehler verarbeiten und anzeigen. Wie das im Code aussieht erfahren wir in Listing 6.

Bei geöffnetem Netzwerkspiel-Dialog reagieren wir auf den Fehler, indem wir als Server auf weitere Clients warten oder als Client eine Fehlermeldung anzeigen. In allen anderen Fällen interessiert uns der Error-Event nicht, was wir aus Listing 7 entnehmen.

Im Spielfenster kann über das Socket-Steuerelement auch der Event „Connected“ kommen, der uns anzeigt, dass eine Verbindung besteht. Sind wir jedoch kein Server, bleibt unsere Spielhölle für den Kunden geschlossen und dessen Verbindung wird beendet. Wir schreiben also in den Connected-Event des Sockets den Code aus Listing 8

Wir rufen hier die Methode „connected“ im NetzwerkSpiel-Dialog auf, die noch nicht existiert. Wir erzeugen sie und füllen sie mit dem Code aus Listing 9.

Wir haben nun eine Verbindung und sind Server. In diesem Fall läuft ein Spiel, wir schließen den Dialog jedoch nicht sofort, denn wir müssen dem Spielfenster noch mitteilen, dass wir der Server sind. Dies machen wir im Spielfenster beim Menü-Handler für ein neues Netzwerkspiel. Vor dem „return true“ ist noch Platz und wir ergänzen den Code, der den aktuellen Server/Client-Status setzt. Danach schließen wir den Dialog.

Nun teilt der Server dem Client mit, dass der seinen Netzwerkdialog schließen und sich auf das Spiel vorbereiten soll. Wie der Befehlscode dazu aussieht ist bedeutungslos. Wir schicken der Einfachheit halber den Text „Get Ready!“+ chr(13). Mit chr(13) ist das Zeichen für die Eingabetaste gemeint, die auf dem Mac die Bedeutung eines Zeilentrenners hat. Wir ergänzen also den Menü-Handler „Spiel-Netzwerkspiel“ um die Zeilen aus Listing 10.

Zum Schluss rufen wir die Methode „CreateNetzwerkSpiel“ auf, die wir als leere Methode anlegen. Auch die Eigenschaften „Server as boolean“ und „Client as boolean“ erzeugen wir vorab im Fenstercode.

Das Programm lernt lesen

Fließen Daten über eine Netzwerkverbindung, ruft Realbasic den Event „DataAvailable“ auf. Dort kann man über die Socket-Methoden „Lookahead“ und „Read(anzahl)“ beziehungsweise „Readall“ den Datenpuffer bearbeiten.

Der Code für den DataAvailable-Handler, der ungefähr alle 1,5 KB (Größe eines TCP/IP Pakets) aufgerufen wird, sollte möglichst schnell arbeiten, damit die Übertragung ohne Datenstau und Wartepausen abläuft. ►

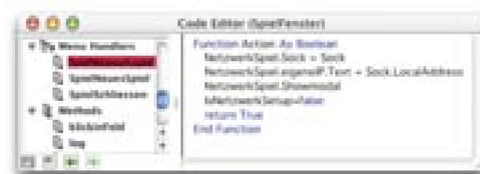
Tipp | Ideen für Erweiterungen

Programmieren lernt man, indem man es tut. Nehmen Sie sich Zeit und bauen Sie das Spiel weiter aus:

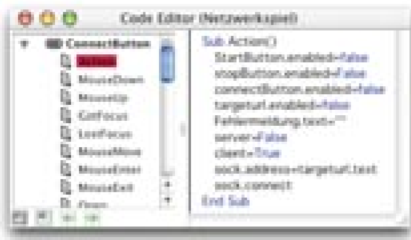
- Chat und Cheat? Wie wäre es mit einem kleinen Edit-Feld, das beliebigen Text über das Socket abschickt. Doch Achtung: Stellt man dabei einer Chat-Meldung keine besondere Kennung voran, dann wird sie die Gegenseite als Befehl interpretieren, was sich letztlich auch gut zum Mogeln eignet.
- Sicherheit? Unser Programm ist nicht gegen das Empfangen von Datenmüll gesichert. Eingegangene Befehle werden bislang auch noch nicht bestätigt.
- Zu schwer? Wer meint, dass der Computergegner zu schwer zu besiegen ist, kann den Computer so programmieren, dass er alle möglichen Züge ermittelt und dann den besten aussucht. Dabei kann der Computer versuchen, maximal viele andere Figuren rauszuwerfen oder möglichst wenige.
- Mehr Mitspieler? Was mit zweien geht, geht auch mit mehreren. Erweitern Sie das Spiel doch auf drei oder vier Netzspieler.
- Zu spät und Server zu? Man könnte auch erlauben, sich später einzuloggen und einen Computerspieler zu übernehmen. Genau so gut kann man sein Spiel abgeben, wenn die Lust mitten im Spiel nachlässt.

Listing 1

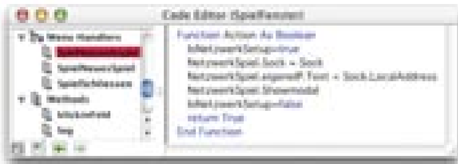
Listing 2



Listing 3



Listing 4



Listing 5

Listing 6

Tipp | TCP/IP in Realbasic

In Realbasic gibt es ein Steuerelement namens „Socket“, mit dem man über das Netzwerk-Protokoll TCP/IP Daten zwischen zwei Computern austauschen kann. Dazu wartet ein Socket beim Server auf eine Verbindung zu einem Client. Jeder Computer in einem lokalen Netzwerk oder im Internet hat eine eindeutige Identifikations-Nummer wie beispielsweise 129.168.1.10 oder einen symbolischen Namen dafür wie zum Beispiel „www.macwelt.de“. Innerhalb von TCP/IP verfügt jeder Computer über 65536 logische Ports. Vergleichbar mit einer Firmentelefonnummer und der Durchwahl. Viele dieser Ports sind fest vergeben. Bei www.macwelt.de findet man unter Port 80 die Internet-Seite der Macwelt und unter Port 110 und Port 25 den E-Mailserver der Macwelt-Redaktion.

Zuerst kopieren wir den Puffer in eine Stringvariable ohne ihn zu löschen. Wir durchsuchen den String dann auf ein Zeilenende. Wenn wir eines finden, befindet sich mindestens eine Meldung im Puffer, die wir inklusive des Zeilenendes auslesen. Nun überprüfen wir, ob die Meldung der „Get Ready!“-String ist, falls ja, wird unser Mac zum Client erklärt (der Server wird diese Meldung hoffentlich nie bekommen!) und blendet das Netzwerkspielfenster aus. Den Realbasic-Code hierzu sehen wir in Listing 11.

Sobald der Dialog geschlossen wird, führt Realbasic den Code hinter „showmodal“ aus und sowohl der Server als auch der Client starten die Routine „CreateNetzwerkSpiel“, der wir uns jetzt zuwenden.

Schnell entwickeln durch Kopieren und Einfügen

Kreatives Kopieren und Einfügen erleichtert uns das Tippen der Methode „CreateNetzwerkSpiel“. Wir kopieren den kompletten Code aus „NeuesSpiel“ und füllen damit die noch leere neue Methode. Alles was der Dialog „NeuesSpiel“ braucht, löschen wir. Die Spieler benennen wir wie folgt:

```
Spieler(0) = new Spieler(“Server”,
rgb(0,0,255), 0, self)
Spieler(1) = new Spieler(“Client”,
rgb(255,255,0), 10, self)
Spieler(2) = new Spieler(“Computer A”,
rgb(0,255,0), 20, self)
Spieler(3) = new Spieler(“Computer B”,
rgb(255,0,0), 30, self)
```

Am Ende der Routine legen wir fest, wer als menschlicher Spieler agiert:

```
Spieler(0).Mensch = true
Spieler(1).Mensch = true
Spieler(2).Mensch = false
Spieler(3).Mensch = false
```

Zum Schluss stellen wir ein, dass es sich um ein Netzwerkspiel handelt und starten, falls wir nicht der Client sind, den Timer.

```
isnetzwerkSpiel=true
if not client then
nexttimer.mode=2
end if
```

Das Spiel läuft nun! Jetzt kann allerdings noch der Fall auftreten, dass unser Mac weder Client noch Server ist. In diesem Fall spielt der Computer allein.

Der Client dient als reiner Befehlsempfänger

Da wir an dieser Stelle nicht viel Platz für lange Code-Listings im Heft haben, gestalten wir unseren Client etwas weniger intelligent. Als braver Dienstmann wird er die Befehle vom Server ausführen und fleißigst die Reaktionen der Mitspieler an seinen Meister schicken.

Im Folgenden bauen wir viele kleinere Änderungen ein und beschreiben diese aus Platzgründen nur kurz. Zunächst schicken wir in der Methode „log“ im Spielfenster alle Einträge als Kopie zum Client:

```
Sub log(s as string)
  liste.insertRow 0,s
  if server then
    sock.write “log”+s+chr(13)
  end if
End Sub
```

In der Methode „Nextplayer“ sind ein paar Änderungen notwendig. Wir schützen den ganzen Befehlsblock mit einem „if not client then“ davor, dass er auf dem Client ausgeführt wird.

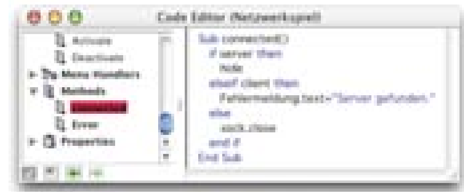
Hinter der Ausgabe des Spielernamens mittels „Spielername.Text = Currentplayer. Name“ schicken wir ein Paket an den Client, damit dieser den Namen des Spielers anzeigt:

```
if isnetzwerkSpiel and server then
   sock.write “next“+str(spielercounter)
   +chr(13)
end if
```

Weiter unten schicken wir hinter der Zeile „war.movetohaus“ eine Meldung an den Client, damit er die Spielfigur an der Startpo-

Listing 7

Listing 8



Listing 9

sition des aktuellen Spielers hinauswirft:

```
if server then
  sock.write "kick"+str(currentplayer.
  offset)+chr(13)
end if
```

Hinter der Zeile „n.movetobrett(currentplayer.offset)" melden wir dem Client eine neue Figur:

```
if server then
  sock.write "neuefigur"+chr(13)
end if
```

In der Methode „menschzug" verschicken wir einen Auftrag an den Client dort einen Zug auszuführen, wenn der Client an der Reihe ist.

```
if spielerCounter=1 and server then
  sock.write "zug"+chr(13)
end if
```

Die Methode „klickinFeld" wird sowohl beim Client als auch beim Server aufgerufen. Aber man darf die Figur eines anderen Spielers nicht verschieben, weshalb wir da etwas komplizierter prüfen müssen, bevor wir einen Klick erlauben. Wenn ein Netzwerkspiel läuft, müssen wir dem Client den Zug mitteilen, damit dieser ihn bei sich ausführen kann. Falls das System nicht der Client sein sollte, starten wir anschließend den Timer zum Spielerwechsel. Den neuen Code der Methode „KlickInFeld" sehen wir in Listing 12.

Jetzt wird es ernst! Die Befehlsausführung

Weiter geht es im Steuerelement „Sock". Wir erweitern den Event „DataAvailable" um neue Variablen: „dim f, war as figur", „dim w as integer", „dim t as string" und „dim b as brettfeld". Bisher haben wir nur einen Befehl pro Paket zugelassen. Durch unsere Log-Funktion kommen aber wesentlich mehr Pakete an und das Mac-OS wird zur Steigerung der Geschwindigkeit mehrere Pakete zusammen ausliefern. Mit einer „Do-Loop"-Schleife:

```
do
loop until n=0
```

gehen wir solange auf Paketsuche, bis wir keines mehr im Puffer finden. Im weiteren Verlauf wertet unser Programm alle Befehle aus und führt sie durch. Sie finden den kompletten Code im Projekt-File auf der Abo-CD beziehungsweise im Internet unter www.macwelt.de/_magazin. Die Kommentare sollten zur Erklärung ausreichen.

Die Figuren brauchen einen Index

Im Netzwerkspiel müssen wir die Figuren durch eine eindeutige Nummer auswählen können. Bis jetzt befinden sich die Figuren in einem Feld, und wissen ihren Index nicht. Daher bekommt die Klasse Figur eine neue Eigenschaft „index as integer" und wir ändern den Konstruktor, indem wir die Methode Figur wie folgt erweitern:

```
Sub Figur(c as color,i as integer)
  farbe=c
  index=i
end sub
```

Dadurch müssen wir auch den Konstruktor des Spielers anpassen, indem wir den neuen bei jedem Aufruf Index mit übergeben. Folgende Zeilen leisten dies:

```
figur(0)=new figur(c,0)
figur(1)=new figur(c,1)
figur(2)=new figur(c,2)
figur(3)=new figur(c,3)
```

In der Methode „würfeln" schicken wir nun die gewürfelte Augenzahl nach der Zeile „Würfel= rnd*6+ 1" zum Client:

```
if Fenster.isnetzwerkSpiel then
  Fenster.sock.write "würfeln"
  +str(würfel)+chr(13)
end if
```

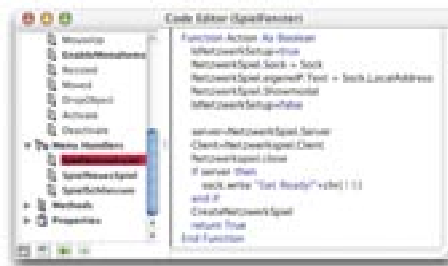
Weiterhin fehlt in der Methode „gehefelder" noch eine Benachrichtigung an den jeweils anderen Rechner. Wir schreiben hinter der Zeile „gehemitFigur f,c":

```
if fenster.isnetzwerkSpiel then
  if (fenster.server and fenster.spieler
  counter<>1) or (fenster.client and
  fenster.spielercounter=1) then
    fenster.sock.write"Move"+str(f.Index)
    +" "+str(würfel)+chr(13)
  end if
end if
```

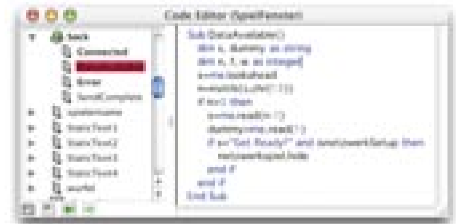
Wenn wir unser Projekt jetzt kompilieren, sollte die Netzwerkfähigkeit funktionieren. Probieren Sie ein Spiel über das Internet, vielleicht mit dem Onkel aus Amerika?

Fazit

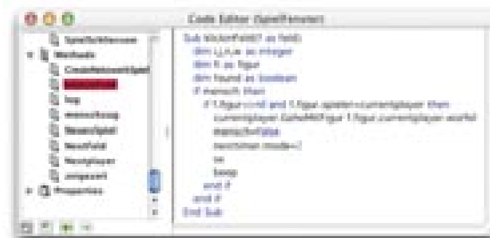
Der aktuelle Stand unserer Simulation zeigt, wie man ein einfaches Server/Client-Prinzip unter Realbasic implementiert. Das öffnet nicht nur Spielern Tür und Tor zum Internet. Auch ernsthafte Anwendungen sind denkbar. In der nächsten Folge widmen wir uns dem finalen Schliff unseres Programms sowie dem Kompiliervorgang und der Optimierung. *cm* ✖



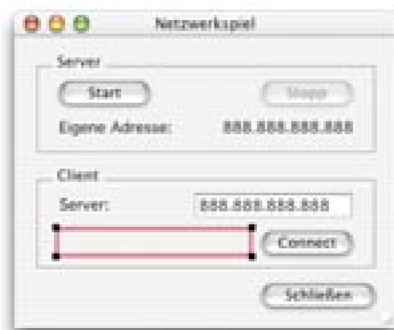
Listing 10



Listing 11



Listing 12



Neuer Dialog im Spiel Der Netzwerkdialog erlaubt es, Server und Client in einem einzigen Fenster zu steuern.

ONLINE....

Alle Quellcodes zu dieser Serie finden Sie im Internet unter www.macwelt.de/_magazin und auf der Heft-CD, weitere Programmierbeispiele für Realbasic bei der Firma ASH unter www.applicationsystems.de

Optimieren und Compilieren

von Christian Schmitz

Serie Realbasic, Folge 5 In der letzten Folge haben wir den Code unseres Programms bereits fertig gestellt. Damit daraus auch ein auslieferungsfähiges Produkt wird, wollen wir aber noch ein wenig an der Optik feilen

EINSTIEG

Unsere Serie erklärt Schritt für Schritt, wie man in Realbasic ein Brettspiel à la „Mensch ärgere Dich nicht" erstellt. In dieser letzten Folge zeigen wir, wie wir unser Programm zur Veröffentlichung fertig stellen und für Mac-OS X verschönern.

✚ **UNTER MAC-OS X** kommt vor allem dem Programm-Icon eine wichtige Bedeutung zu, denn die Icons sind unter Mac-OS X deutlich größer und fallen im Finder gut auf. Unter Mac-OS 9 und Mac-OS X bekommt ein Programm jedoch nur dann ein eigenes Icon, wenn es mit einem Creator-Code versehen ist. Über diesen Code ordnet der Finder die Icons den Programmen zu. Ein Creator-Code besteht aus vier beliebigen Zeichen, wobei alle Codes, die ausschließlich aus Großbuchstaben bestehen, bereits für Apple reserviert sind. FNDR ist beispielsweise der Creator-Code für den Finder, BOBO für Apple Works. Andere Beispiele sind nwSP für Mac Soup, VtPC für Virtual PC und MSWD für Microsoft Word.

Den Creator-Code darf man sich jedoch nicht beliebig ausdenken. Alle Codes müssen bei Apple registriert sein, um Doppler zu vermeiden. Unter der Adresse http://developer.apple.com/dev/cftype/information.html kann man seinen Creator-Code prüfen und registrieren lassen. Dieser Code ist, wenn man ihn denn bekommt, einmalig auf der Welt. Für unser Spiel „Mensch ärgere Dich nicht" haben wir bereits den Creator-Code „MäDn" registriert, der bislang noch ungenutzt war. Wir tragen ihn im Fenster „Projekteinstellungen" im Bearbeiten-Menü von Realbasic ein.

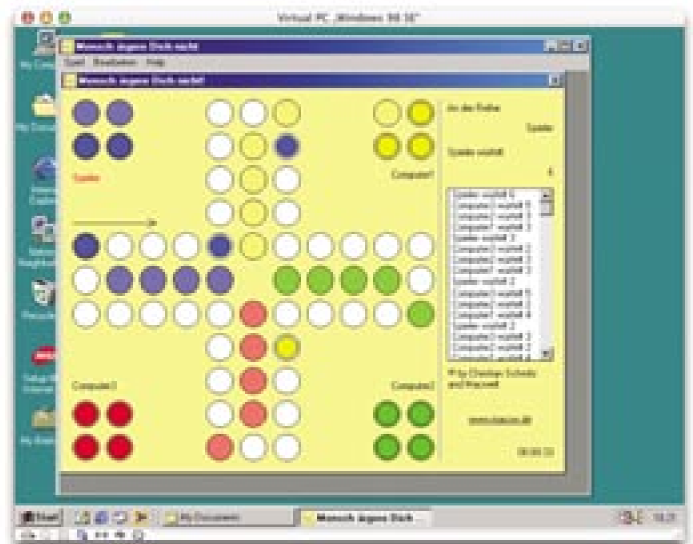
Icons organisieren

Es gibt mehrere Möglichkeiten an ein hübsches und aussagekräftiges Icon zu kommen. Apple hat dafür extra ein Team von Grafikern eingestellt und auch Sie können sich ein Icon bei einem Grafiker malen lassen. Das kann allerdings schon mal 100 Euro oder mehr kosten. Nicht jeder kann sich das leisten.

Für unser Programm machen wir es uns etwas einfacher. Wir fertigen ein Icon aus einem verkleinerten Screenshot unseres Programms an. Dazu lassen wir das Spiel ein paar Minuten lang automatisch laufen (vier Computer spielen gegeneinander) und wenn es interessant aussieht, schieben wir die Maus beiseite und lösen mit den Tasten Befehl-Umschalt-3 einen Screenshot aus.

Diesen Screenshot öffnen wir zum Beispiel mit dem Programm Grafikkonverter. Nun ziehen wir einen quadratischen Bereich über der Spielfläche auf und achten dabei darauf, dass Breite und Höhe des Ausschnittes, die rechts oben bei den Mauskoordinaten angezeigt werden, gleich groß sind. Anschließend wählen wir im Menü Bearbeiten den Punkt „Auswahl freistellen" (Befehl-Y).

Diesen quadratischen Ausschnitt müssen wir nun noch auf 128 mal 128 Pixel herunterrechnen. Dazu wählen wir im Menü Bild unter



Fremdgänger Auch unter Windows macht unser Spiel eine gute Figur. Hier sieht man Virtual PC unter Mac-OS X mit Windows 98.

Serie: Realbasic für Profis

1	Interface	Heft 12/2001
2	Computerspieler	Heft 01/2002
3	Mensch spielt mit	Heft 02/2002
4	Multiplayer	Heft 04/2002
5	Compilieren und Release	Heft 05/2002

Tipp | Hilfsprogramme im Internet

Alle in dieser Folge erwähnten Hilfsprogramme und Utilities finden Sie im Internet. Zumeist handelt es sich um kostenlose Software oder Shareware, für die man ein paar Euro oder Dollar Sharewaregebühr zahlen muss.

Programm	Vertrieb	Preis	Internet
Grafikkonverter	Lemkesoft, Shareware	25 €	www.lemkesoft.de
Res-Edit	Apple	kostenlos	ftp.apple.com/Apple_Support_Area/Apple_Software_Updates/English-North_American/Macintosh/Utilities/ResEdit_2.1.3.sea.bin
Iconographer	Mscape, Shareware	15 US-Dollar	www.mscape.com/products/iconographer.html
Icon Paste Mac-OS 9	Realsoftware	kostenlos	ftp.realsoftware.com/REALbasic/release/IconPaste.sit
Icon Paste Mac-OS X	Realsoftware	kostenlos	ftp.realsoftware.com/REALbasic/release/IconPasteX.sit
Plug-in-Sammlung	Monkeybread Software	kostenlos	http://monkeybreadsoftware.de/realbasic

Bildgröße die Option „Skalieren…“ aus (Befehl-Control-Y). Wir geben für die Breite 128 Pixel ein und aktivieren die drei Häkchen für Proportional, Bild Skalieren und Fehlerkorrektur. Das nun skalierte Bild speichern wir im Format PICT.

Icon zusammensetzen

Um das Icon in eine Mac-Resource zu wandeln gibt es verschiedene Möglichkeiten. Eine ist das Programm Res-Edit von Apple (siehe Kasten „Hilfsprogramme im Internet“)

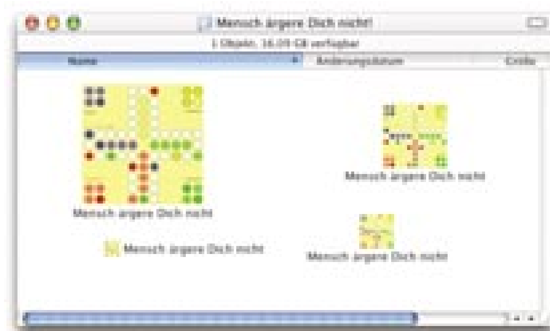
Wir legen in Res-Edit eine neue Datei an. Dort ergänzen wir eine Resource vom Typ „icl8“ mit der ID 128 und fügen in dem Icon-Editor Icons für die verschiedenen Auflösungen über die Zwischenablage ein. Diese Resource-Datei speichern wir nun.

Die zweite Möglichkeit ist entgegenkommender, kostet aber Geld. Gemeint ist das Programm Iconographer. Es ist für viele Entwickler das Icon-Malprogramm der Wahl, denn es erstellt auch Icons für Mac-OS X. Für reine Mac-OS Classic-Programme reicht Res-Edit aus.

In Iconographer legen wir ein neues Icon an. Zuerst sehen wir uns im Menü „Icon“ das Infofenster an. Die ID sollte 128 sein, den Namen wählen wir beliebig und stellen „Mac OS Universal“ als Format ein. Unter der Karteikarte „Member“ können wir noch detailliert festlegen, welche Icon-Typen die Datei enthalten soll. Da der Finder fehlende Icons bei Bedarf selbst berechnet, könnte man beispielsweise die Huge-Icons weglassen, denn diese sind nur interessant, wenn die Icons beim Skalieren, zum Beispiel im Dock unter Mac-OS X, nicht schön aussehen.

Wir kopieren nun unser Icon per Zwischenablage von Grafikkonverter in Iconographer in das Icon „Thumbnail 32bit Icon“. Die Maske füllen wir mit Schwarz, denn nur die Icon-Pixel, die in der Maske schwarz sind, werden später auch vom Finder angezeigt. Mit der 8-Bit-Maske kann man auch wunderschöne Schatten erzeugen wie man es an den Realbasic-Standard-Icons sieht.

Mit dem Menübefehl „Add Member...2“ im Icon-Menü erzeugen wir automatisch aus dem großen Icon die anderen Icon-Typen. Von dieser Funktion machen wir Gebrauch, sie spart uns eine Menge Arbeit. Trotzdem sehen wir uns jedes so generierte Icon noch einmal kritisch an. Die ein oder andere Retusche ist meist notwendig.



Viele bunte Icons Diese Montage zeigt die Icons für unser Spiel in vier verschiedenen Skalierstufen. Mac-OS X zeigt das 128 Pixel große Icon sehr detailliert.

Einbauen ins Projekt

Der Dialog „Compiler-Einstellungen“ im Ablage-Menü von Realbasic ist bei Icons nicht besonders ausgereift. Ein großes Mac-OS X Icon können wir hier leider nicht einfügen. Wir müssen sie als Mac-Resource ins Projekt hineinziehen und für das Windows-Binary zusätzlich noch ein kleines und mittleres Icon in diesen Dialog einfügen.

Iconographer speichert eine einzige Datei, die all unsere Resourcen beinhaltet. Realbasic akzeptiert nur Resource-Dateien mit dem Type „rsrc“ und dem Creator „RSED“. Anders gesagt: Realbasic akzeptiert nur Resource-Dateien von Res-Edit, obwohl es Res-Edit für Mac-OS X gar nicht gibt. Damit das klappt, müssen wir den Type und den Creator der Datei ändern und Realbasic vorspiegeln, dass sie von Res-Edit stammt. Erfreulicherweise kann man dies mit Res-Edit selbst erledigen. ►

Häusle baue Im Build Dialog tragen wir die Dateinamen und Programminformationen für die compilierten Versionen ein.

Die modifizierte Resource-Datei ziehen wir nun per Drag-and-drop ins Realbasic-Projektfenster, wo sie als Alias verbleibt.

Im Build Dialog

Werfen wir einen Blick auf den Screenshot (oben) „Build-Settings". Er zeigt den fertigen Dialog mit allen Einstellungen, wie er in Realbasic 4 unter Mac-OS X aussieht. Wenn wir jetzt das Programm compilieren, wird Realbasic die Warnung bringen, dass Resourcen überschrieben werden. Wir ignorieren das, denn das ist genau das, was wir wollen!

Wir können unser Programm für Mac-OS Classic und für Mac-OS Carbon erzeugen. Classic-Programme laufen ab System 7 bis Mac-OS 9 und passen sich automatisch dem jeweiligen Erscheinungsbild des Mac-OS an.

Mac-OS-Carbon-Programme sind in erster Linie für Mac-OS X gedacht. Sie laufen mit installierter Carbon Lib jedoch auch unter Mac-OS 8.6 bis Mac-OS 9.2.2.

Carbon als Schicht zwischen dem Programm und der jeweiligen Mac-OS Version benötigt aber unter Mac-OS Classic (nicht bei Mac OS X!) etwa 3% mehr Rechenzeit für Funktionsaufrufe, so dass man vorzugsweise sein Programm für beide Umgebungen anbieten sollte. Da unser Spiel nicht sehr zeitkritisch arbeitet, würde uns auch eine Carbon-Version für den Classic-Betrieb genügen.

Die Speicherzuteilung für die Classic-Umgebung muss man selber grob schätzen oder raten. Es gibt wenig konkrete Hilfe den Speicherbedarf eines Realbasic-Programms herauszufinden. Einem Carbon-Programm sollte man mindestens 2 MB zuteilen. Je nach Anzahl der Grafiken, Sounds und Filmen im Programm erhöht sich der Speicherbedarf, so dass schnell auch mal 10 MB zusammen kommen. Für dieses Spiel in der Variante ohne viele Bilder (wir haben keine Bilder als Resourcen eingebunden und erzeugen auch keine zur Laufzeit) sollten 2 MB reichen. Wir schreiben also 2048 KB in die zwei Felder für die Speicherzuteilung.

Einstellungen für Windows

Besitzer von Realbasic Pro können das Programm auch für Windows compilieren. Der Dateiname muss dabei mit „.exe" enden. Umlaute im Programmnamen können sich wegen der unterschiedlichen internen Kodierung beim Kopieren zu Windows hin verändern. Der Einfachheit halber nennen wir das Spiel schlicht „Mensch.exe".

Unter Windows gibt es zwei mögliche Programmtypen. MDI steht für „Mulitple Document Interface" und SDI für „Single Document Interface". Das Windows-Programm Notepad ist ein Beispiel für ein SDI-Programm, denn es kann immer nur ein Dokument in einem Fenster öffnen. Einige Word und Excel Versionen können aber auch mehrere Dokumente in unterschiedlichen Fenstern öffnen. Bei einem solchen MDI-Programm umschließt ein großes Fenster mit der Menüleiste alle Dokumentenfenster.

In unserem Spiel kann man mehrere Spielbretter öffnen. Wenn wir uns jedoch für SDI entscheiden, müssen wir dies unterbinden, denn nur das erste Spielfenster bekommt eine Menüleiste. Schließt man diese, hat man gar keine Menüleiste mehr.

Wir aktivieren also das Häkchen bei „Multiple Document Interface" und bekommen eine Menüleiste in einem übergeordneten Fenster. Vergrößert man dieses Parent-Fenster, hängt die Menüleiste wie beim Mac am oberen Bildschirmrand.

Wir wollen uns verewigen

In Deutschland gilt das Urheberrecht, wonach unsere Arbeit als Programmierer automatisch gesetzlich geschützt ist. Wir dürfen selbst entscheiden, ob und wie wir unser Programm veröffentlichen möchten. Richtig verkaufen können wir es aber nicht, sondern nur lizenzieren. Wir räumen also zum Beispiel einer Firma Vertriebs- und Nutzungsrechte ein oder vergeben einfach, à la Open Source, eine Lizenz zum uneingeschränkten Weiterverwenden des Programms an alle.

Die meisten Hobbyprogrammierer werden die Programme, mit der Erlaubnis sie kostenlos privat einzusetzen, einfach weitergeben. Wer das Programm dann vertreiben oder aus-

Tipp | Icons für Mac-OS und Windows einbinden

Wir wollen unser Programm auch für Windows compilieren. Ein Icon für Windows bekommen wir über den Realbasic-Dialog „Compiler-Einstellungen". Wenn wir ein Icon dort einfügen, müssen wir in einem Rutsch alle sechs möglichen Icon-Typen über die Zwischenablage hineinclippen. Realsoftware hat für diesen Zweck ein kleines Programm names Icon Paste entwickelt, das dies deutlich einfacher macht. In Icon Paste fügen wir alle Icon-Typen einzeln ein. Dazu öffnen wir in Res-Edit unser fertiges Mac-OS-Programm und kopieren die Resourcen über die Zwischenablage Stück für Stück in Icon Paste. Wir benötigen:

Resource icl8, ID 128 (Icon in 32x32 Pixel und 256 Farben)
Resource icl4, ID 128 (Icon in 32x32 Pixel und 16 Farben)
Resource ICN#, ID 128 (Icon in 32x32 Pixel in S/W mit der Maske)
Resource ics8, ID 128 (Icon in 16x16 Pixel und 256 Farben)
Resource ics4, ID 128 (Icon in 16x16 Pixel und 16 Farben)
Resource ics#, ID 128 (Icon in 16x16 Pixel in S/W mit der Maske)

In Icon Paste wählen wir anschließend im Menü Edit den Befehl „Copy Icons" aus und fügen den Inhalt der Zwischenablage in Realbasic im Dialog „Compiler-Einstellungen" ein. Dazu klicken wir einmal auf das kleine Icon links unten, um es auszuwählen und fügen die Icons per Befehl-V ein. Wenn alles geklappt hat, sieht man im compilierten Windows Programm nach dem Starten links oben im Fenster das kleine Icon.

bauen will, muss sich beim Programmierer melden. Damit jeder weiß, wo er sich melden soll, sollte man sich im Programm verewigen und zwar so vorausschauend, dass man auch nach fünf Jahren noch Kontakt bekommt und auch dann, wenn alle Readme-Dateien verloren gegangen sein sollten. Was unserem Spiel also noch fehlt, ist Dekoration und vor allem eine Dialogbox mit Informationen über den, der das Programm entwickelt hat.

Die Aboutbox

Wir erstellen ein neues Fenster namens „AboutWindow" und geben ihm den Titel „Über Mensch ärgere Dich nicht". Wir möchten gerne in diesem Fenster links oben das aktuelle Programm-Icon als Logo anzeigen. Dafür gibt es mehrere Möglichkeiten: Wir können das Icon noch einmal als Bilddatei speichern, in das Projekt legen und in einem Canvas als Backdrop auswählen.
Alternativ dazu kann man unter Mac-OS mit der Zeile

```
me.backdrop = app.resourceFork.geticl(128)
```

das Icon mit der ID 128 aus der Resourcefork des Programms selber laden. Das klappt auch in unserem Spiel, hat jedoch zwei Nachteile: Erstens bekommt man nur ein Icon mit maximal 32 Pixel Breite zu Stande und zweitens funktioniert das unter Windows nicht, denn dort gibt es generell keine Resourcefork.
Besorgt man sich eine spezielle Plug-in-Sammlung für Realbasic von Monkeybread Software kann man über die Zeile

```
app.applicationfile.DrawwideIcon
g,20,14,128
```

im Paint-Event des Fensters das Programm-Icon in voller Größe zeichnen. Nur unter Windows sollte man statt der 128er Größe besser die 32er Icons nehmen, da es in diesem System keine so großen Icons gibt.

Internet-Links im Fenster

Wir nehmen einen Statictext, schreiben eine Internet-Adresse wie zum Beispiel „macwelt.de" hinein, setzen die Farbe auf Blau und die Schrift auf Times 14 mit dem Attribut Unterstrichen – schon sieht das aus wie ein Internet-Link im Browser. Ab Realbasic 4.0 hat ein Statictext-Object auch MouseDown- und MouseUp-Events. Benutzt man eine ältere Realbasic-Version, so muss man erst ein Canvas darüber legen um an diese Events zu kommen. Egal ob Canvas oder Statictext, wir schreiben in den MouseDown-Event des Links:

```
me.textcolor=Rgb(255,0,255) // lila text
return true
// ja, wir verarbeiten diesen Event
```

Sechs auf einen Streich Icon Paste erleichtert die Zusammenstellung der richtigen Icon-Resourcen für das Programm-Icon unter klassischem Mac-OS.

Alle Infos So sollte ein sinnvoller About-Dialog aussehen: mit Logo und Kontaktmöglichkeit zum Programmierer.

Beim MouseUp-Event tippen wir analog dazu den folgenden Realbasic-Code ein:

```
me.textcolor=Rgb(0,0,255)
// Wieder auf blau
showURL "http://www.macwelt.de"
// Webseite im Browser öffnen
```

Der Befehl „showURL" akzeptiert je nach Betriebssystem unterschiedlich tolerant viele verschiedene URLs. Wir sollten hier also besonders auf die Vollständigkeit der URL achten. „macwelt.de" oder „info@macsw.de" reicht nicht aus. Bei der URL müssen wir das Protokoll mit angeben, also „http://www.macwelt.de" oder „mailto:info@macsw.de".

Ein Menüeintrag im Apple Menü

Ein Doppelklick auf das Menü im Projektfenster bringt uns in den Realbasic-Menüeditor. Dort klicken wir einmal auf das Apfelmenü und geben im noch leeren Menüeintrag „Über Mensch ärgere Dich nicht..." ein. Realbasic weist dem Menüeintrag automatisch einen langen Namen zu, aber wir nennen ihn einfach „AppleAbout". In unserer Application Subklasse („App") ergänzen wir einen Menühändler für „AppleAbout" und rufen dort den Dialog mit folgenden Zeilen auf:

```
aboutWindow.show
return true
```

Unter Windows gibt es kein Apfelmenü, sondern nur ein Startmenü zum Beenden. Realbasic schiebt die Menüeinträge in diesem Fall an das Ende des letzten Menüs ganz rechts. Bei uns ist das also das Menü „Bearbeiten". Wir verbessern das, indem wir ein Menü namens „Help" anlegen. Beim Mac-OS werden alle Einträge in diesem Menü in das jeweilige Hilfe-Menü geschoben, so dass aus „Help" unter einem deutschen Mac-OS automatisch „Hilfe" wird. Unter Windows klappt das leider nicht immer, was man aber mit einer Konstante ändern kann, die dann unter Mac-OS den Wert „Help" und unter Windows den Wert „Hilfe" bekommt. Als Text schreiben wir dann in den Menüeintrag ein Doppelkreuz „#" gefolgt vom Konstantenname, also zum Beispiel „#Helpmenutitle".
Damit dieses Hilfemenü aber nicht so allein da steht, bekommt es als weitere Einträge noch „Homepage" und „Email an den Autor". Diese Einträge verknüpfen wir dann über die Application-Subklasse und rufen sie wieder mit ShowURL auf. In Realbasic 4 gibt es da einen kleinen Bug, denn die Einträge im Hilfemenü sind in der Entwicklungsumgebung niemals aktiviert, im fertig compilierten Programm klappt jedoch alles wie es soll.
Als letztes bleibt uns nur noch ein paar Freunde zu suchen, die das Programm noch einmal auf Herz und Nieren prüfen und dann, wenn wirklich die Version 1.0 final daraus wird, das Programm auf eine Homepage zu legen und es in der Öffentlichkeit bekannt zu machen.

Fazit

In dieser Serie haben wir die Entwicklung eines Brettspiels in Realbasic von Anfang bis Ende verfolgt und viele neue Programmiertechniken gelernt. Bei entsprechender Resonanz sind wir gerne bereit in einer der nächste Ausgaben der *Macwelt* mit einem neuen Projekt zu starten. Schreiben Sie uns! *cm*

ONLINE....

Sie finden die Projektdateien und die fertig compilierten Programme im Internet unter www.macwelt.de/_magazin oder auf der aktuellen Abo-CD. Eine 30-Tage-Demo-Version von Realbasic gibt es unter www.application-systems.de/realbasic