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=testfenster.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

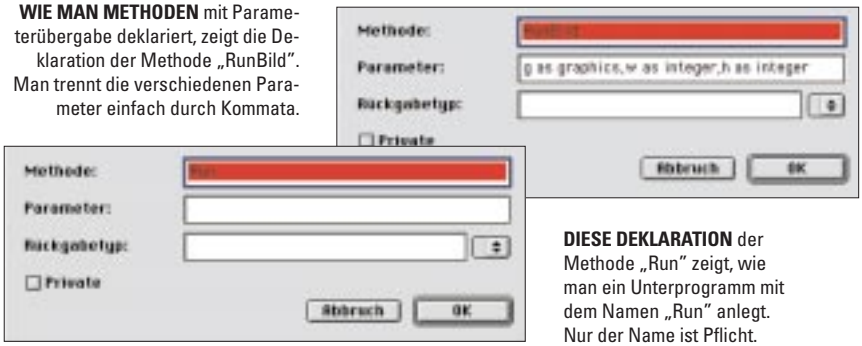
Die erste Messung erzeugt keine Grafik und misst nur die CPU-Leistung. Während einer Zeit von fünf Sekunden könnte man allerdings meinen, dass der Mac abgestürzt oder eingefroren ist. Damit dieser Eindruck nicht aufkommt, schreiben wir vor den eigentlichen Test den Text „Bitte warten…“ in großer Schrift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

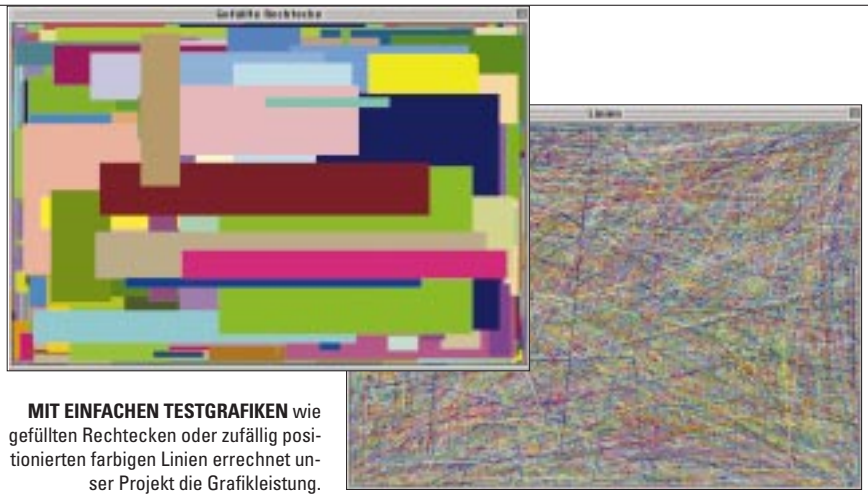
Wir deklarieren eine Methode mit folgender Zeile: „Sub RunLeerlauf(g as graphics,w as integer,h as integer)“. Dazu rufen wir im „Bearbeiten“-Menü die Funktion „Neue Methode…“ auf und verwenden als Namen „RunLeerlauf“. Für die Parameter geben wir „g as graphics,w as integer,h as integer“ ein. In diesem 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.title="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.



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

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=microseconds +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.

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.foreColor=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 „linienfeld.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

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

„stringwidth“ die Länge des Texts in unserer Konstante und speichern den Wert in der Variable „Breite“. Folgende Zeile erledigt dies: „breite=g.stringwidth(text)“.

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 „mac welt.jpg“ genommen, das im Projektfenster dann „macwelt“ heißt. Zum Darstellen von Bildern benutzt man den Befehl „drawpic ture“, 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