



REALbasic **developer**

www.rbdeveloper.com



THE REALBASIC REVOLUTION

Now **ANYONE**
Can Program!

ANIMATION SECRETS

THREE WAYS TO ANIMATE
BY JOE STROUT

REGEX EXPLAINED

BY MATT NEUBURG

INTERVIEW

A CHAT WITH **ANDREW BARRY**,
CREATOR OF REALBASIC

POSTMORTEM

DEVELOPING WRITER



REVIEWS

UNIHELP
WINDOWSPLITTER
MPCALC

PLUS

Q&A, ALGORITHMS,
BEGINNER'S CORNER,
REALCHALLENGE, &
MORE!

REALbasic® is one of the hottest development environments for the Macintosh. Now there's a **new magazine** that will help you tweak every clock cycle of power from REALbasic.

Whether you're an amateur or a pro, every bimonthly issue of **REALbasic Developer** is packed with over 50 pages of vital information:

- step-by-step tutorials by top authors
- handy tips
- answers from the experts
- fascinating interviews
- "behind-the-scenes" postmortems of popular programs
- reviews and profiles
- REALchallenge programming contest
- REALbasic news
- much more!

REALbasic Developer Magazine:
Now **anyone** can program!



Available in
**PRINT
OR
PDF**

Subscribe today for up to **33% off** the retail price!

www.rbdeveloper.com

REALbasic is a registered trademark of REAL Software, Inc. REALbasic Developer magazine is not affiliated with REAL Software. REALbasic Developer, PO Box 66831, Scotts Valley, CA 95067.

Peachpit Press has the help you need!

Coming Soon!



Sign up for a free PDF excerpt
from *REALbasic for Macintosh:*
Visual QuickStart Guide
Stop by booth #631



**SAVE 20% during
MacWorld NYC 2002.**

Stop by booth #631 or go to:
www.peachpit.com/forms/programming.asp
to purchase these titles and more!

**Save
20%**

Adobe Illustrator Scripting
By Ethan Wilde
ISBN: 0-321-11251-2 • \$35.00

AppleScript for Applications:
Visual QuickStart Guide
By Ethan Wilde
ISBN: 0-201-71613-5 • \$21.99

C#: Visual QuickStart Guide
By Jose Mojica
ISBN: 0-201-88260-4 • \$19.99

Hot Cocoa for Mac OS X
By Bill Cheeseman
ISBN: 0-201-87801-1 • \$44.99

**HTML 4 for the World Wide
Web, Fourth Edition: Visual
QuickStart Guide**
By Elizabeth Castro
ISBN: 0-201-35493-4 • \$19.99

**Java 2 for the World Wide
Web: Visual QuickStart Guide**
By Dori Smith
ISBN: 0-201-74864-9 • \$21.99

**JavaScript for the World Wide
Web, 4th Edition: Visual
QuickStart Guide**
By Tom Negrino & Dori Smith
ISBN: 0-201-73517-2 • \$19.99

**Perl and CGI for the World Wide
Web, 2nd Edition: Visual
QuickStart Guide**
By Elizabeth Castro
ISBN: 0-201-73568-7 • \$19.99

Python: Visual QuickStart Guide
By Chris Fehily
ISBN: 0-201-74884-3 • \$21.99

**XML for the World Wide
Web: Visual QuickStart
Guide**
By Elizabeth Castro
ISBN: 0-201-71098-6 • \$19.99

**Unix for Mac OS X: Visual
QuickPro Guide**
By Matisse Enzer
ISBN: 0-201-79535-3 • \$24.99



Peachpit Press

THE MAGAZINE FOR REALBASIC® USERS

REALbasic Developer is not affiliated with REAL Software, Inc.



Postmortem
11



Revolution
16



Animate
18



Regex
24

FEATURES

11 Postmortem: Writer by Daniel Kennett
Pioneers get the arrows. Learn from the experiences of someone else.

14 Interview: Andrew Barry
REALbasic's creator may no longer guide the software, but his vision still influences us all.

16 The REALbasic Revolution by Marc Zeedar
What makes REALbasic so revolutionary?

18 Three Ways to Animate by Joe Strout
Find out which animation technique is appropriate for your project.

24 Regex Explained by Matt Neuburg
Understand the mysteries of powerful regex commands.

COLUMNS

Source Code 5
A word from the Publisher.

Beginner's Corner 28
Just getting started? This is your column!

Advanced Techniques 30
Tips for professionals.

Ask the Experts 32
Your questions, our answers.

Algorithms 34
The core of all programming.

Object-Oriented Thinking 36
Understanding OOP.

Intel Focus 37
Tips for cross-platform compiling.

The Topographic Apprentice... 38
Step-by-step 3D graphics.

From Scratch 40
Follow a project from concept to completion.

AppleScript 42
Using AppleScripts in RB.

Cocoa 43
Cocoa lessons for the REALbasic programmer.

Interface Design 44
Proper user interface design.

Beyond the Limits 46
Advanced secrets.

REALchallenge 50
Test your programming skill and win prizes!

REVIEWS, PROFILES, ETC.

REALbasic News 6

Carbon Events Plugin 2.5 8

CURLLinkButton 8

MPCalc 9

Stimulus 9

UniHelp 1.1 10

WindowSplitter 10

REAL Ads 49



OPINION & FEEDBACK

GOOD LUCK!

Good luck with the launch of *REALbasic Developer* magazine. I'll be looking forward to reading it.

Michael Swaine
Author, *REALbasic QuickStart Guide*
(forthcoming)

MASCOT GREETINGS

I just wanted to thank you for the honor of being named as the official mascot of *REALbasic Developer*! My family is so proud of me. Now I have something up on my second cousin, Max! (He works for *MacAddict*.)

Arby
arby@rbdeveloper.com

CAN'T WAIT

Thank you so much for launching this magazine. I can't wait for my first issue to arrive!

Frank Palayaman
New York

LONG LIFE

Long life to this new magazine!

Eric

ONE CLICK

On the front page of the RB Dev Mag site, there's a calendar image with a week circled, talking about when the first issue will come out. By any chance, is the calendar shown in the image one of the One Click calendars?

Kevin Ballard

Yes, it is! We're big One Click fans here at RBD: <http://www.designwrite.com/oneclick/>

Send your **Letters to the Editor** to letters@rbdeveloper.com. You must include your full name and valid email address, but we will withhold publishing either on request. All letters may be edited for content or length, and become the property of *REALbasic Developer*.

How to Download Source Code

Article resources available at:
<http://www.rbdeveloper.com/subscriber/>

Every article in *REALbasic Developer* includes an "RBD number" at the end presented like this:

RBD# 1234

To retrieve an article's resources follow these steps:

Step 1: Go to the password-protected *RBD* subscriber website (at www.rbdeveloper.com/subscriber/).

Step 2: Log in using username: **subscriber** and password: **yogurt**. (This password changes for each issue of *RBD*. Please do not share this password with non-subscribers.)

Step 3: In the **Get Article** search field, type in the *RBD number* of the article you need. All resources for that article — source code, graphics, demo projects, etc. — will be available for instant downloading.



is published by DesignWrite,
P.O. Box 66831, Scotts Valley, CA 95067-6831
and has no affiliation with REAL Software, Inc.

Publisher & Editor

Marc Zeedar editor@rbdeveloper.com

Editorial Board

Joe Strout, *REAL Software* jstrout@rbdeveloper.com
Matt Neuburg, Author matt@rbdeveloper.com
Erick Tejkowski, Author etejkowski@rbdeveloper.com

News Editor

Chris Willis news@rbdeveloper.com

Review Editor

Brian Jones reviews@rbdeveloper.com

Copy Editors

Toby Rush trush@rbdeveloper.com
Tim Lisauskas timm@rbdeveloper.com

Advertising Coordinator

Marc Zeedar ads@rbdeveloper.com

Webmaster

Sylvain Le Borgne webmaster@rbdeveloper.com

Layout & Design

Marc Zeedar mzeedar@rbdeveloper.com

Artwork

Scott Melchionda (Cover) scott@scoo.com
Lloyd Colbaugh (Arby) lcolbaugh@rbdeveloper.com

Columnists

Didier Barbas dbarbas@rbdeveloper.com
Sean Beach realchallenge@rbdeveloper.com
Collin Cornaby ccornaby@rbdeveloper.com
Thomas Cunningham tcunningham@rbdeveloper.com
Dean Davis ddavis@rbdeveloper.com
Will Leshner wleshner@rbdeveloper.com
Joseph Nastasi jnastasi@rbdeveloper.com
Matt Neuburg matt@rbdeveloper.com
Thomas Reed thomasreed@rbdeveloper.com
Toby Rush trush@rbdeveloper.com
Christian Schmitz cschmitz@rbdeveloper.com
Seth Willis help@rbdeveloper.com
Charles Yeomans cyeomans@rbdeveloper.com

About the Mascot

Arby™ is the official mascot of *REALbasic Developer* magazine. He's an amateur but determined *REALbasic* programmer, spending late nights trying to make nthField a few milliseconds faster. He's your guide for all things *REALbasic*. Watch for him in the magazine and on the *RBD* website.



REALbasic® is a registered trademark of REAL Software, Inc. It and other trademarks used within this publication are the property of their holders and are used only for editorial purposes with no intent to infringe. *REALbasic Developer*, the *REALbasic Developer* logo, and the Arby mascot name and icon are trademarks of DesignWrite.

Contents Copyright © 2002 by DesignWrite.
All Rights Reserved

Source Code

by Marc Zeedar

Welcome! A dream becomes reality

When I first began using REALbasic, I was amazed: how long had this tool existed without me knowing about it? It seemed cruel I hadn't discovered it earlier.

I'd always longed to program my Macintosh, but I hated the tedious business of learning the Mac's complex Application Programming Interfaces (APIs). I spent hundreds of dollars on massive programming books and *Inside Macintosh* volumes, only to watch them become obsolete before I had time to learn them!

REALbasic was the answer.

But despite its simplicity and elegance, REALbasic is a true programming environment. It can be obtuse, especially for larger projects.

The Internet is an invaluable resource for RB users, but I still longed for a single source for all things REALbasic: a regular magazine, professionally designed and written, with all the quality of *Macworld*. That dream is finally a reality.

A Magazine's Journey

In August 1999 I had the idea for *REALbasic Developer*. I was thinking solely of a subscriber-based PDF "ezine." I posted a survey on my website and the results showed that people would be willing to pay for a quality REALbasic publication.

With a budget of zero dollars, I assembled a team of volunteers and began making plans for the first issue. It soon became clear, however, that the project was more complicated than I'd anticipated. The

volunteers were not always reliable and I was working a full-time job myself; I couldn't make up the slack.

It was a difficult decision, but I put the magazine on hold. The last thing I wanted was to launch and immediately fold, damaging the credibility of the concept in the process.

I had no intention of abandoning *REALbasic Developer* permanently. I hoped at some point in the future I'd be in a position to afford to quit my day job and launch the magazine properly.

The Mac brought computers to
people who weren't (and didn't
want to be) computer literate.
REALbasic brings programming
to people who aren't (and don't
want to be) computer scientists.

REALbasic to the Rescue

During this time, I'd been working on a couple novels, and I'd been horribly frustrated by the complex task. With a spurt of inspiration, I wrote a little program in REALbasic which I called Z-Write. It was a word processor designed to help me keep all the snippets of text associated with my novels organized. Z-Write worked so well, I decided to try selling it.

To my astonishment, Z-Write proved popular: it brought in over \$2,000 in the first month of release!

That may not be much to an Adobe, but for me it was a windfall, and it was the first step toward *REALbasic Developer* becoming a reality.

The next step was the launching of *REALbasic University*, a weekly tutorial series I wrote for the Applelinks website starting in February 2001. I figured writing those columns would be good practice and would draw attention to REALbasic and *REALbasic Developer*.

A Revolution is Launched

REALbasic hasn't made me a millionaire, but it has certainly changed my life. I'm not a programmer by training, but with RB I was able to accomplish a dream.

That's why I chose "Revolution" as the theme for this first issue. REALbasic is a revolutionary product in the same sense the original Macintosh was revolutionary. The Mac brought computers to people who weren't (and didn't want to be) computer literate. REALbasic brings programming to people who aren't (and don't want to be) computer scientists.

It's a match made in heaven.

Give Me Feedback

I hope you enjoy *REALbasic Developer*. I wish this were a venture that could be done for free, but producing a magazine is expensive. Your support means a great deal to everyone involved. It's a statement to us, to Apple, and to the world that Mac programming is for *everybody*, not just those with computer science degrees. Your subscription will ensure that *REALbasic Developer* is around for a long time.

I want this to be *your* magazine, so let me know what you want to see! Write to letters@rbdeveloper.com or feedback@rbdeveloper.com.

I'd love to hear from you!

When RBD publisher Marc Zeedar was a kid he used to create magazines just for fun. Now he's doing it for a living! You may contact him at [<editor@rbdeveloper.com>](mailto:editor@rbdeveloper.com).



REALBASIC 4.5 PRE-RELEASES

REAL Software is working on the next release of REALbasic. Version 4.5 adds sheet support, the PagePanel Control (which works like an invisible TabPanel), virtual volumes, parent and child controls for better organization of controls, lots of database additions and fixes, sounds for tips, new method and properties dialogs, a new build settings dialog, a new icon compositor dialog to add your icons more easily, a brand-new prefs window, an "Open Recent" menu, listbox enhancements, a splittable code editor, a huge number of changes to the Quicktime support, a huge number of changes to the 3D Classes, vector graphic support, Preferences menuitem (in the Application menu) support on Mac OS X, folderitem enhancements, new icons for the code editor and menus, and all those little bug fixes and little new features too. REAL Software is trying to make this release the most stable version ever, with a long beta testing period. If you want to test out prerelease versions of REALbasic, join the RB Developers List. Information on this list is available at <http://www.realbasic.com/support/internet.html>.

SORT LIBRARY 1.8

SortLibrary is a free, open-source REALbasic library providing robust, optimized implementations of several standard sorting algorithms for use by REALbasic developers. The major change in version 1.8 is a substantial rewrite of quicksort which implements the three-way partitioning method of Bentley and McIlroy. This yields substantial improvements in performance in the presence of equal keys. In addition, quicksort is now implemented using recursion. <http://>

www.quantum-meruit.com/RB/SortLibrary.sit.hqx

BARCODE AUTOMATOR 1.0

Intelli Innovations, Inc. has released Barcode Automator 1.0, a barcodingsuite specifically designed for bulk generation and AppleScript automation. Barcode Automator supports 9 different symbologies, includes vectorized EPS and TIFF export capabilities, and comes bundled with a set of OCR fonts. Barcode Automator is now available for \$329.95; a full-featured demo version is also available. <http://www.intellisw.com/barcodeautomator>



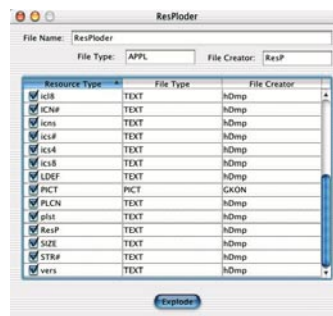
DRAG PROMISE EXAMPLE

This example project creates a DragItem from within a canvas, and when dropped into the Finder will create a file at that location. <http://www.freaksw.com/rb-examples.html>

RESPLODER

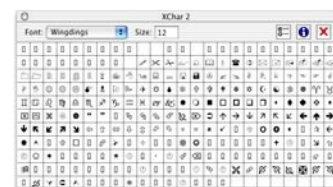
ResPloder explodes the resources of a file into folders. The resource types become folders and the resources become files that can be edited with any hex editor. Some resource types are automatically saved in popular file formats. The data fork of the original file is also preserved. ResPloder can also implode a ResPloded folder back into a file — drop a "ResPloded" folder onto ResPloder and a new file with exactly the same structure as the original file will be created. If any of the resource files in the ResPloded folder have been modified, then the changes will become part of

the new file. <http://ljug.com/sw/resploder.html>



XCHAR 2.0

XChar 2.0 is a new version of the OS X utility that allows you to use uncommon characters even if you don't remember the correct key combination. This new version adds compatibility for applications which do not support Drag and Drop and also allows to copy the font and style along with the character. Moreover, lots of bugs were fixed and interface was improved. <http://www.ziksw.com/software/xchar.html>



CARBON DECLARE LIBRARY

This is a handy collection of enhancements available through the Declare feature of REALbasic. The library is mainly for Carbon applications but includes support for Classic and maybe Windows too. <http://kevin.sb.org/Files/CarbonDeclareLibrary.sit>

CUSTOM ICON PLUGIN V1.0

Custom Icon is a plugin which adds new OS X icons — such as the alert icon, the note icon, the stop icon and other system icons — to any REALbasic

project. <http://www.mac-x-software.com>



DRAW CONTROL PLUGIN V1.0

DrawControl is a plugin which adds new OS X controls — such as the small check box, the small radio button, the ArrowButton, and round BevelButton — and other functionality such as icon and arrow support. <http://www.mac-x-software.com>

MOVIEWORKS PLUGIN V1.1

This new plugin from Alfred Van Hoek enables more powerful features to be used with the QuickTime Movie Player. <http://homepage.mac.com/vanhoek/#movieworks>

DATABROWSER PLUGIN V0.5.1A

This plugin implements the DataBrowser control in Carbon for Mac OS 9 and Mac OS X. It is a more advanced listbox than REALbasic's own control and provides more control and more features to use. <http://www.webprofitable.com/RB>

SETH WILLITS CLASSES/PLUGINS

I updated my REALbasic section and added a lot of my plugins, classes, modules, and examples I have neglected to put up for a while. A total of 16 in all. I also uploaded carbon version of FSSinceWhen which I forgot to do on the initial release. <http://www.freaksw.com/rb.html>

DOCK MENU PLUGIN V1.0

Dock Menu Pro allows users to create dynamic hierarchical Dock tile menus for their REALbasic applications running under Mac OS X 10.1 or higher. Dock Menu





Pro implements virtually all Menu Manager functions to allow users to customize their Dock menu anyway they feel fit. It also allows users to add custom data "tags" to each menu item for added flexibility. Dock Menu Pro plug-in costs \$10 and is royalty free. <http://www.everydaysoftware.net/code/index.html>



FSPLAYQTMovie — MIDI

FSPlayQTMovie can be used to play a QT compatible movie file without using a MoviePlayer control. Because it doesn't have a movie control, it is only useful for hearing the sound track. This plugin will be most useful for developers that want to use MIDI files in their applications. MIDI files are required to be opened by REALbasic as a QuickTime movie and played by a MoviePlayer. <http://homepage.mac.com/freaksoftware/rb/>

SPEECH RECOGNITION PLUGIN V1.1.2

The Speech Recognition plugin allows simple support for Speech Recognition and Speech Synthesis. It also allows users to install speech help in the Speech Commands window under Mac OS X. It works on PPC and Carbon with Speech Recognition 1.5 or higher. <http://www.everydaysoftware.net/code/index.html>



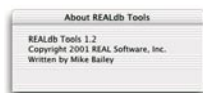
URL ACCESS PLUGIN V2.0

Alfred Van Hoek's URLAccess plugin allows REALbasic applications to upload and download Internet files using the system's

URLAccess manager. The new version of the plugin allows setting and getting of URLproperties and supports HTTPURLMethods such as "GET," "SET," "POST" and others. You can post searches, and above all it works flawlessly with the Van Hoek's HTMLrendering plugin. The versatility of URLAccess Plugin leads to numerous new possibilities, including development of a light-weight web browser application that may compete with other web browsers. URLAccess plugin requires MacOS 8.5 or higher. Users may purchase a professional license for building commercial products (\$39), an academic license for building products you cannot sell (\$12), or an amateur license to get you going (free). <http://homepage.mac.com/vanhoek>

REALDB TOOLS

REALdb Tools gives you the ability to manipulate REAL database files in ways that the built-in database engine does not provide at this time. With REALdb Tools you can compact a database to make it smaller after records have been deleted, drop any table or column from a REAL database, change table names, change column names and data types, and add new tables and columns. <http://www.realsoftware.com/realbasic/about/REALdbTools.html>



CARBON EVENTS PLUGIN 2.5.2

This new version of the Carbon Events Plugin provides a wealth of functionality to Carbon applications: access to a toolbar button, scroll wheel support, live window resizing, access to the application dock tile

menu, standard alert sheet, alert dialog support, Ask Save Changes, Discard Save Changes sheet support, Put File dialog sheet support, full service menu support, access to usable screen size (for respecting the Dock), Quit event notification, complete proxy icon support with icon dragging and window's path menu, the ability to get any folder by type and domain, the ability to set and get long file names, and access to the current theme identifier. Cost: \$15. <http://www.everydaysoftware.net/code/index.html>

RBULK BUILDER 1.2.6

RBulk Builder is a Batch Building companion for REALbasic. With it, you can automatically launch different builds, changing the name, the platform, the language and even constant values for each one. It is designed for people that need to build many different binaries (different platforms, localized, demo/light versions...) from a single source project file. Version 1.2.6 provides compatibility workarounds with REALBasic versions 4.0 and above when building carbon applications. The program is free and open-source. <http://www.liane.net/rb/rbulk>

STRING STUFF PLUGIN 3.1.1

The String Stuff Plugin can return a MemoryBlock pointing a string, allowing a REALbasic program to alter the string "in place" by altering the memoryblock. The plugin allows the use of REALbasic text functions, such as AscB and InStrB with drastically improved performance and provides CharSet searching. String Stuff contains "FastString," a class that can increase the speed of string appends by tens or hundreds of times compared to REALBasic strings. The plugin includes a huge range of UTF32 handling methods,

including an ultra-fast MSR (multi search replacement) object, and FastReplaceAllB method for doing replaceAllB at speeds up to 267 times faster than ReplaceAll. <http://www.elfdata.com/programmer/downloadindex.html>

ROUND BUTTON & ROUNDED BEVELBUTTON PLUGINS

These plug-ins allow users to create real round navigation buttons and rounded bevel buttons under Mac OS X; the buttons are compatible with OS 8.5 and higher. <http://www.everydaysoftware.net/code/index.html>



CALENDAR CLASS

Calendar Class is an easy-to-insert calendar for your projects. Simply drag the class into your next project to add a customizable calendar. Change font, button color, background color and more for a customized look for every project. <http://www.colourfull.com/ã>





Carbon Events Plugin 2.5

Daniel Howard

You use REALbasic because it is the easiest Rapid Application Development tool for the Macintosh. However, it is limited when it comes to OS X (Carbon) events, features, and controls. That is where the Carbon Events Plug-In comes in. It takes over where REALbasic leaves you stranded.

The Carbon Events Plug-In adds several key features for Carbon builds: Scroll Wheel support, Quit events, Sheets support, useable screen space, Proxy Icons, and a lot more!

The *Quit* event is essential for OS X builds. It enables you to use the Quit menu item that is found under the Application menu. REALbasic quits when this menu is selected, but you can't add any code to the event. (i.e. when a user quits your word processor you would usually want to make sure that any open files have been saved. With REALbasic 4.0.2 and lower, this was not an option. The application would quit without warning when Quit was selected from the Application menu.)

A *Sheet* is an OS X dialog box that is attached to a window. They can be translucent or opaque and make it easier for the user to distinguish what window the dialog box is related too. Although it is possible to create a sheet in REALbasic, without this plugin you cannot use sheets to display Save/Open dialogs.

The following may sound like trivial features; however, they are very useful. The first is the inclusion of Proxy Icons. Proxy Icons are displayed in the title bar of a window and have become widely used in OS X. Their inclusion in this plugin allows programmers to make their applications follow OS X appearance conventions. The plugin also allows users to keep track of the usable space on their screen. Under Mac OS X, the user's desktop will usually have a Dock on it, in any of a number of positions. Users will expect that your application recognize the position of their Dock. While nearly impossible in REALbasic, Carbon Events Plug-

in makes that as easy as adding two lines of code!

Although this plug-in has a lot of great features, it still has some minor drawbacks. For instance, it does support Live Window Resizing, but if you do this, no controls can be locked to any of the windows sides (which limits the usefulness of Live Window Resizing). Also, Translucent sheets will not work if anything else in the window is redrawn. These problems are REALbasic-related and are not controllable by the creator of this plugin.

Besides these few minor drawbacks, the Carbon Events Plug-in is a must have for all OS X developers. Hopefully REALbasic will add these features in future releases, but until then this plugin will do the trick.



IN BRIEF

Product
Carbon Events
Plugin 2.5

Manufacturer
Everyday Software

Price
\$15

System Requirements
Mac OS X 10.1+,
REALbasic 3.x

Contact Info
everyday@mac.com
homepage.mac.com/
everyday

Pros
Inexpensive; Easy to
use; Gives essential
OS X controls

Cons
Live Window Resizing
is limited; scrollwheel
support doesn't work
all the time

Rating (1.0 - 5.0): 4.5



RBD# 1024

CURLLinkButton

Mike Richards

At one point in time or another, almost all REALbasic developers will want to integrate a clickable URL inside of their application. And thanks to REALcode's CURLLinkButton control, anybody can do this!

CURLLinkButton is simple to use: you need to drag the class into your project window first, then you create a canvas control, set its "super" class to CURLLinkButton, type a few lines of code (documented in the Read Me), and run the program. Not only does it resize itself to the exact size of your link, it also handles drawing and going to the URL that you indicated.

Several methods let you customize the appearance of your new link. You can set a special cursor when the mouse is hovering over the link, change the font and size of the text, and more. You can even change the text that's displayed to something other than the URL, and it still works perfectly.

You can use any URL with CURLLinkButton, including http, ftp, telnet, or any other URL type that's properly configured in your Internet settings. No declares are used at all, meaning it can work on any REALbasic-supported platform (68K, PPC, Carbon, Win32), and it comes as a REALbasic v1 format project, meaning any REALbasic

user can use CURLLinkButton. In other words, anyone with a Quadra and REALbasic v1.1 up to an OS X machine with REALbasic v4.0.2 can use it. The price, you ask? Entirely free.

However, the total control freak might not like CURLLinkButton. The text is highlighted when you click on the link, rather than the text simply changing colors, as you are probably used to in a web browser. There's no way to alter the text color either, without editing the class yourself. Bold and italic support would have been nice, although would probably not have been used much anyway. But as I mentioned, the class is open source, and you can revise and edit it as you please.

There's no revolutionary code here. Just a simple class that does a simple job: create a clickable link in your REALbasic application. Anyone can use it, and it does the job well.



IN BRIEF

Product
CURLLinkButton

Manufacturer
Catalunya Disseny
Informàtic

Price
Free

Contact Info
Pablo Iglesias
63 Local 4
08302 Mataró
(Barcelona), Spain
catdis@catdis.com
catdis.com/realcode/
realcode.htm

Pros
Easy to use; supports
every version
of REALbasic;
customizable; free

Cons
No longer supported;
slightly strange
appearance

Rating (1.0 - 5.0): 3.6



RBD# 1023



IN BRIEF

Product

MPCalc Plugin

Manufacturer

Dr. Robert Delaney

Price

Free

Contact Info

delaneyrm@mac.com

Pros

Very high precision;
Free

Cons

Adds about 700k to
filesize; uses RPN
math

Rating (1.0 - 5.0): 3.8



RBD# 1021

MPCalc Plugin

Jim Rodovich

Chances are if you need to do some math calculations with REALbasic, doubles can hold numbers as large or small as you'll require. However, if you need to get very precise answers, in some cases a double might not be sufficient. Typically, numbers represented by doubles are accurate for about the first sixteen digits. Anything to the right of the sixteenth digit is most likely inaccurate. This means that if you're using a double to store a number that's in excess of one quadrillion, you can expect any values after the decimal point to be incorrect.

Fortunately, you won't need to forsake modern technology and resort to using paper and a pencil when you're dealing with highly precise numbers. Instead, you can use MPCalc, a freeware REALbasic plugin that can perform calculations on numbers with up to thirty thousand digits. You can even set the precision yourself, so if you just need twenty digits, your program won't be chugging along calculating 29,980 unnecessary digits.

The MPCalc plugin operates using what is known as the Reverse Polish Notation (RPN). This is somewhat different from what you're probably used to, but it's a small adjustment. In an expression written in RPN, both numbers are written before the operator. For example, instead of writing $1 + 2$, you would write $1\ 2\ +$ in RPN. It's a different way of thinking, but with a little practice, even expressions with several operations are easy to state in RPN.

Unfortunately, instructing MPCalc to execute various calculations isn't as simple as performing arithmetic on integers or doubles in REALbasic. Instead, you'll need to enter a few commands to tell the plugin to do anything. Adding two numbers requires five commands: two to enter the first number, one to enter the second number, another to tell MPCalc to add the numbers, and a fifth to read the result. Even though it takes five lines of code to perform such a simple calculation, it's still very easy to issue commands. This is in part due

to the simple method names that are easy to remember.

MPCalc gives programmers access to a wide array of mathematical functions. It supports basic arithmetic operations, as well as exponential, trigonometric, hyperbolic, and logarithmic functions. It also supports several more obscure functions, including the Beta and Gamma functions, Bessel functions, and Fresnel Integrals.

Additionally, MPCalc has four storage registers for remembering the results of previous calculations. The registers prove quite valuable since the only other way to store multiple-precision values is by using strings.

Overall, MPCalc is a solid plugin that is fairly well documented. Because of its flexible precision and wide variety of scientific functions, the MPCalc plugin is a worthy addition to your REALbasic toolbox if you do any serious math calculations.



IN BRIEF

Product

Stimulus 2.0

Manufacturer

Electric Butterfly

Price

\$12

System Requirements

Mac OS 8.6+, Mac OS
X 10.1+, Quicktime 4+
(Classic), Quicktime
5+ (OS X), 6 MB RAM

Contact Info

support@ebutterfly.com
www.ebutterfly.com/
stimulus

RBD# 1022

Profile: Stimulus 2.0

Greg Fiumara

If you think the tools that the REALbasic environment gives you to work with digital media are too few, think again. Stimulus 2.0, Electric Butterfly's image viewer and audio/video player, is out.

Stimulus is an all-in-one digital media viewer. If you are tired of switching through tons of applications to view your media files (Photoshop for your pictures, iTunes for your MP3's and iPod, QuickTime Player for your videos), Stimulus is for you. This version of Stimulus has many new features, including the ability to play back from your iPod. (Didn't know you could do that with REALbasic, did you?)

Stimulus supports over fifteen file formats, from Audio CD to WAVE, and still more formats are planned for inclusion in the 3.0 release of the software.

Stimulus has a built-in help system, which is more or less a simple emulation of the Language Reference built into REALbasic. The help demonstrates a good way to use HTML rendering

in your application. This application also takes advantage of the famous CD Control by Toby Rush, which powers its Audio CD playback.

If it were not for custom classes and the sub-classing made available in the REALbasic IDE, REALbasic would probably not have survived. Sub-classing is one of the more powerful features in REALbasic, and, clearly, Stimulus 2.0 shows that. From its amazing color bevel buttons that also have 32x32 icons to the WindowSplitter control, Stimulus 2.0 would not have been released or have been successful without the help of sub-classing and third party class developers.

Electric Butterfly has found an interesting and very clever use for a timer control in Stimulus 2.0. This new version of Stimulus has a slide show feature. This feature activates a timer that, every few seconds, scans through the built-in file browser for media files that it can open. It then displays the files that it finds one at a time for a slide-show effect.

This new release of Stimulus has amazing new error handling techniques, thanks to REALbasic version 4. The new release of REALbasic includes a new event called "Unhandled Exception." This allows programmers to not have to add "exception" lines to each object in their program to prevent from crashes. Instead, the programmer can add in just a few lines of code to the Unhandled Exception event and if there is a crash, it triggers the code in that event. Because of this, Stimulus can also have very light memory requirements to compliment its small file size.

If the user happens to like this shareware program, as long as they have their credit card handy, they can order it without getting out of their chair. Stimulus makes perfect use of the eSellerate plugin for REALbasic for convenient online ordering.

Stimulus 2.0 shows off the power of REALbasic in many impressive areas.





UniHelp 1.1

Brian Jones

When it comes to help systems, the task of recreating the help information in the several different platform-specific formats can occupy important resources in the finishing out process before application release. UniHelp from Electric Butterfly makes these problems a thing of the past with its universal, multi-platform help interface. Configuring the impressive array of customizable attributes for the UniHelp interface is quick and creating the content is as simple as working in your favorite text editor.

UniHelp, when included in a user's application, manages a self-contained help window with a simple interface. On the left side of the window is a ListBox that can be hierarchical if needed. Each element in the ListBox corresponds to a different text clipping which will be displayed in the scrollable EditField on the right side of the window. The help items can be browsed in this list, or the user can search for key words. Easy navigation buttons are accom-

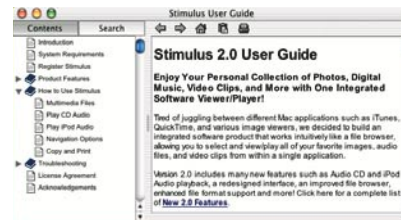
panied by a print button at the top of the window.

Including UniHelp in a project is simple. In the Project window, the user must import the UniHelp module and the UniHelpGraphics folder. A short clipping of code which has been supplied in the UniHelp documentation should be inserted at the Help menu event handler which will create the window. A segment of this code can be adjusted so that certain properties of the UniHelp object can be set to customize the UniHelp interface. All of the text clippings displayed in the help window are written to disk in a directory entitled "Help." The user creates plain text or HTML files and names them according to an easy-to-learn naming convention that determines their display in the UniHelp window.

The customizations available are simple, but they are effective in incorporating UniHelp into the overall look-and-feel of your application. Most

of the options revolve around the display colors of the graphical elements such as buttons and list icons. It also allows the user to set the default language of the user interface elements to one of the following: English, French, Italian, Spanish, German, or Portuguese.

UniHelp provides a simple, multi-platform solution. However, UniHelp costs \$49. At this price, it is likely to be worthwhile for more serious developers, but hobbyists may not be able to justify the expense. It does make things incredibly simple, but other than its powerful search features, it does not provide much functionality that could not be replicated by developers with the necessary time on their hands.



IN BRIEF

Product
UniHelp 1.1

Manufacturer
Electric Butterfly

Price
\$49

System Requirements
REALbasic 3.5.2 or 4.0.2, Mac OS 8.6+, Mac OS X 10.1+, Windows 98+, Quicktime (for image and multimedia features only)

Contact Info
support@ebutterfly.com
www.ebutterfly.com

Pros
Slick; powerful; takes care of tedious work

Cons
May be too expensive for anyone not using it to its full potential

Rating (1.0 - 5.0): 3.8



RBD# 1020

WindowSplitter 4.0

Brian Jones

There are many applications in which a lot of information needs to fit in a small window. A perfect solution for this problem is the use of a bar that can be dragged to resize different areas of the window. Users can expand areas as they need them and then shrink them in order to conserve window real estate. It is precisely this solution that Einhugur Software's WindowSplitter plug-in makes exceptionally easy to implement.

WindowSplitter 4.0 can achieve basic functionality using just one method. Imagine creating a window with a ListBox control on the left and an EditField control on the right with a WindowSplitter control in the middle (similar to UniHelp's window above). In the WindowSplitter control's Open event simply type these lines:

```
me.addControl(myListBox, true)
me.addControl(myEditField, false)
```

That's it. Now, when the application is run, dragging the bar will resize the

ListBox and EditField controls. This simplicity is a result of a great design decision made by Einhugur in a recent version of WindowSplitter. It used to be that a developer had to respond to an event and resize the appropriate controls himself; however, now WindowSplitter maintains a list of controls that it will resize as needed. Controls are added to this list using the AddControl method. The boolean value passed as the second parameter of the AddControl method refers to the relative position of the control to the WindowSplitter. Controls given values of true are understood to be on top of a horizontal bar or to the left of a vertical control, while those given a false value are understood to be below a horizontal bar or to the right of a vertical one. The old resize event has also been retained for developers using custom-defined controls and for the purposes of backwards compatibility.

Several properties allow developers to set things such as minimum sizes for the areas on either side of the separator

and the position of the separator for the purposes of saving interface settings to a preferences file. Also, some special events allow the developer to control the drawing of the separator and to define the clickable region of the separator.

WindowSplitter 4.0 provides a well-polished solution to easily including an important interface element prevalent in many commercial applications. One deficiency is the lack of a catch-all documentation file. The included documentation accessed with Einhugur's Plunger software is useful, but should be supplemented with traditional documentation. The program is paid for (as are other Einhugur products) through their subscription service. For \$49 US, users have access to all of their REALbasic add-ons including a year of upgrades. Those developing applications for which this kind of interface solution would be appropriate would do well to let Einhugur handle it for them.

IN BRIEF

Product
WindowSplitter 4.0

Manufacturer
Einhugur Software

Price
\$49, for Einhugur subscription

System Requirements
REALbasic 2.1+

Contact Info
support@einhugur.com
www.einhugur.com

Pros
Simplicity; surprising flexibility

Cons
Expensive unless you're interested in other Einhugur products; some drawing quirks

Rating (1.0 - 5.0): 3.8



RBD# 1019

Postmortem

by Daniel Kennett

Writer: Behind the application

Writer was my first REALbasic baby. It started off in the REALbasic 2 days, and I learned to program writing it. It slowly evolved until it got to version 4.5, when I decided to change the name from “KennettText” to “Writer,” relabel it version 1.0, and throw it out to the big bad world.

Back then it was really poor. There were no exception handlers at all, and if you tried to open the wrong kind of file it would throw an exception and quit. Despite this, Writer did quite well. Now, I realise this was because the “Made with REALbasic = piece of crap” era hadn’t yet started. As Writer slowly developed, I began to see more and more REALbasic programs on VersionTracker, and I tried many of them to see what others had done. I wasn’t impressed with many; they either looked horrible or looked beautiful but weren’t programmed well, and threw exceptions everywhere. At first, the “Made with REALbasic” bashers were other programmers who worked with complicated languages such as C, and weren’t too happy at seeing how quickly and easily you could make programs in REALbasic. They looked through the programs and blamed every single bug they found on this wonderful development environment. To this day, I see silly mistakes I make blamed on REALbasic by everyone.

By this time, Writer looked fairly decent, was virtually exception-proof, and had started to mature into a fairly useful piece of software. I learned from other people’s mistakes and made Writer

the most stable, good-looking program I could. Now, at version 2.6, it is a successful and popular program that I am proud to show off to everyone I can, especially at work. “Hey, my program can do that much more elegantly!” In the next few pages, I will share my learning experience with you, going through the most important aspects of creating a successful program and what happens when you screw up.

Interface

The first thing people judge your program by is its interface. Even if your program is stable, fast, and the most useful thing on the planet, not many people will use it if it looks ugly. You really have no excuse not to do this, as REALbasic makes it simple

to make a nice interface. Controls are snapped to their proper places when you drag them to your project’s windows, and all you have to do it to set the right font, text size, and size of the control to fit its caption or contents.

After you have made your interface look right, the next things to consider are icons. Nicely done icons that are used for your application’s main, document, and (if appropriate) toolbar icons add the extra sparkle and polish that is needed to make your product shine. However, badly done icons are worse than no icons at all! If you really want icons, but don’t have a drop of artistic talent (like me), you may want to consider getting a specialist to do it for you. This isn’t cheap, and should

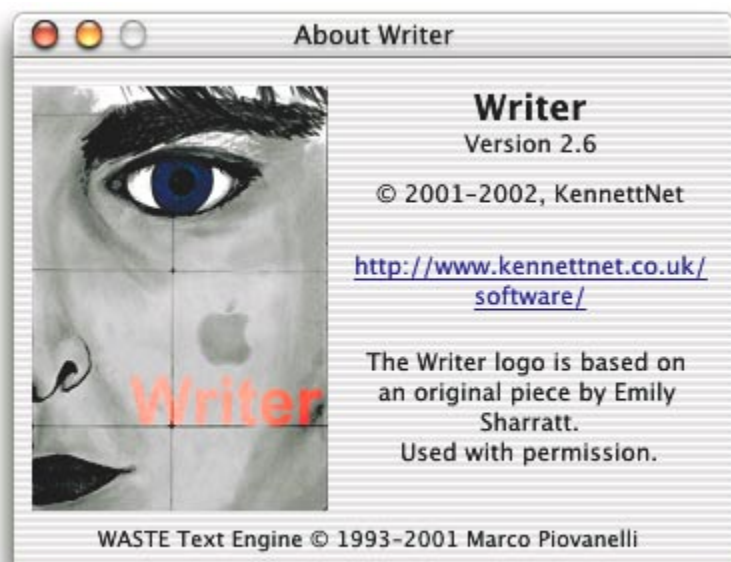


Figure 1: Writer About Box

Daniel Kennett has written several small programs with REALbasic, and successfully released a few of them to the public.

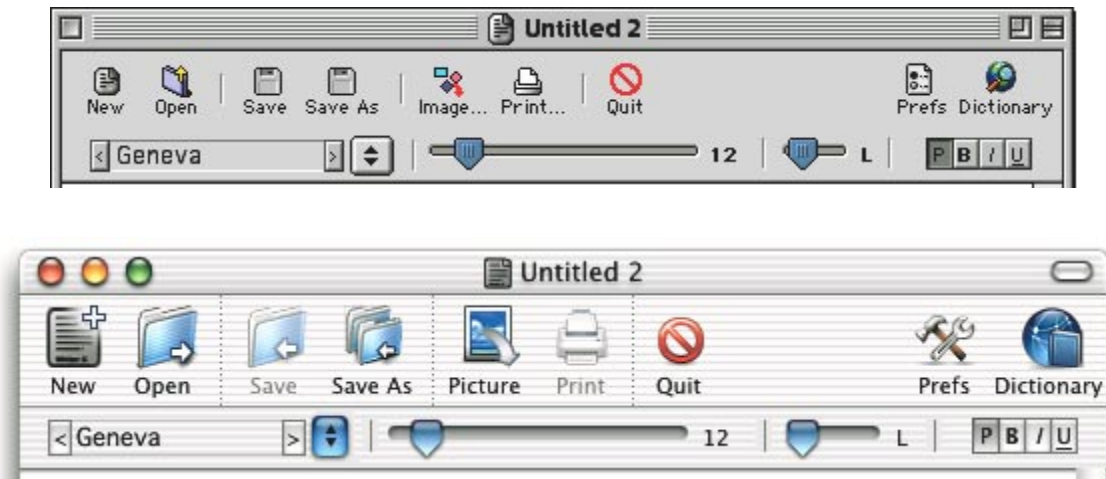


Figure 2: *Writer toolbars for Mac OS 9 and Mac OS X*

only be considered if you are making a big product and expect to get a considerable income from it, or if you are dirty stinkin' rich. My experience with such companies isn't very good, as after much thought and consideration I invested in spiffy icons from Acorn Creative, only to find that they were unable to complete my project due to "funding issues," and I lost my icons and my money. However, don't let that put you down. Take a look at <http://www.iconfactory.com/>. They are one of the industry's best, but are pricey. If you do consider them, bear in mind they designed the icons for the popular MP3 player Audion, and the (not-so-popular among Mac users) operating system Windows XP. Much as I dislike Windows, I love its icons. If I had the money, I'd go for IconFactory in a flash.

One more thing to consider is a logo for your program. Again, you could get a specialist to do this, but doing it yourself is much more fun. Play around with the tools in Photoshop (if you have it), or another graphics program. Often, your program's name with a few effects on it (make sure it's tasteful) will be fine. If you want something fancier, the best thing to do is sit down with a pencil and paper and draw. Once you have something good, scan it in and touch it up in your graphics program. The third option is to look around while you are with friends or colleagues, especially if one of them is artistic. Writer's logo is based on a piece of art my friend Emily drew, which I spotted when she brushed past me on her way home — it was really that quick! Most friends will let you use their work if you give them credit, and if

they're particularly evil, a small cash sum or a present. (See Figure 1.)

Writer's initial interface was absolutely terrible. Badly spaced canvases with icons stolen straight from the system or REALbasic lined the top of the window, which when clicked on could not be cancelled — releasing the mouse button outside the canvas still performed its operation. Now, in the Mac OS X version, the toolbars mimic the beautiful toolbars that are used in Cocoa applications such as Mail and OmniWeb. They aren't customizable yet, but that's something I'm working on. (See Figure 2.)

It is important to tailor the interface for the system on which it runs. Mac OS X applications use slightly bigger fonts and controls than their Mac OS 9 counterparts, so they need a little more room to breathe. Icons are also expected to be bigger, and these large icons look out of place on Mac OS 9, especially if they have been designed to fit in with Mac OS X's Aqua look and feel. In the case of Writer, the interfaces are so different I have split the program into two separate source files, one for "Classic" and one for Mac OS X. It is more work to update, but the effort is worth it as I have a program that looks great in both Mac OS 9 and Mac OS X.

Coding

The code that makes Writer work is terrible. Awful. I'm ashamed of it. It is difficult to update, adapt, and to understand, especially after I haven't looked at it for a while.

When I started Writer, it was to learn how to program. I thought I'd chuck it in the

bin after a few months and start something new. Due to this, I didn't comment my code or make it future-friendly. There was only one way to save a document, which was from the menu bar, so I put the save code into the FileSave menu handler which was fine at the time. Now, there are four ways to save a document — from the menu bar, the toolbar, the "Save Changes?" dialog, and via AppleEvents. Each of those four has its own save code, so when I update the save code in one I have to change it in all four. Now, even when creating the tiniest of programs, I make the effort to comment my code and to use the wonderful invention of global methods. Using these, you can have your save code in one place only, and have your menus and toolbars (or anything else) call it. My save methods usually take the parameters:

```
File as folderitem, askforlocation as Boolean,  
data as variant
```

Then, I can simply call "SaveFile (document, false, editfield1.text)" and the code in that method saves my file, and it works from wherever I call it. This is invaluable for all applications, and absolutely necessary for larger ones. Commenting code is important, too. It takes a bit of extra time and you may think, "I know what my code does, I don't need to comment it!" but it is a godsend for when things go wrong, especially in lengthy or complex pieces of code. Commenting makes it a breeze to tell exactly what your code is doing, where, and when.

Another important thing to do, especially when performing lengthy operations, is to give user feedback. For example, the

Internet Dictionary in Writer used to do its stuff in a loop, with no user feedback. If the operation took a long time, the user would think that my program had crashed and would force quit it, complaining bitterly as much as they could. Now, it works in a thread so the user can do other things while the word is being looked up, and provides feedback in the form of a progress bar, a chasingarrows control, and a small static-text that informs the user what is going on. The operation now takes about three times longer, but the user knows what is going on. I'm sure 95% of your users will prefer that something take longer but know what is going on, rather than it happening quicker with no feedback. This applies especially to novice computer users, who rely on their computer telling them what it is doing.

The last and maybe most important thing to consider is exceptions. Things can go wrong quite easily when programming, and if you miss a potential problem that a user runs into he or she will be presented with a rude box informing them that your program has performed an illegal exception and must shut down. If there was any unsaved work, it will be lost and you will have an angry user. More often than not, exceptions can be caught, fixed, and your program can merrily carry on as if nothing had happened. For example, if your program is instructed to open a corrupt file, a `nilObject` exception may be thrown, which would cause the program to quit. However, after adding these lines to the very bottom of your file opening code, the user will understand what has happened and will be able to try again.

Exception nilobjectexception

```
Msgbox "An unexpected error has occurred. The  
file you selected may be corrupted.  
Please try again."
```

Other possible uses for the exception handler are writing to a log file that an error occurred, or giving the user information about where to report what happened so you can have a go at fixing it. Users that get listened to are very happy users indeed!

Selling/Distributing

You have your beautiful, fast, exception-proof application sitting on your hard drive waiting to be used and abused by your adoring public. Now what? The first thing to do is decide what to charge for it, if anything. If it is a simple app that you have enjoyed making and think others will like, I advise distributing it as freeware. People will love you for it, and you'll get a warm fuzzy feeling for helping out the Mac community. However, if you

do choose to keep it free, never, ever do what I did and decide to start charging for it. People will get used to having it for free, and will complain bitterly when you start to charge. I managed to get away with

**Write a press release and
e-mail it to all the big news
sites, create an entry at
Version Tracker, tell all your
Mac-owning friends, and
put a big sign on top of your
house telling everyone about
your program. Do whatever
it takes to get word out!**

it, but only by the skin of my teeth. The things that saved my hide were charging a ridiculously small amount for it (\$5), and not disabling or restricting the program in any way for unregistered users. The best way to go if you want money for a free program is to politely ask for a small donation toward the development of your program, and to let your users decide what they want to pay.

If you want to distribute your masterpiece as shareware, I recommend using the service provided by eSellerate (<http://www.esellerate.net/>). They don't charge much for their service (10% of each sale to begin with) and provide a web store and an amazing REALbasic plugin that allows people to register from within your program without having to touch a web browser. It can then automatically register your software once payment has been cleared, and a typical transaction takes less than five minutes. To see it in action, download my program Writer and take a look (you don't have to purchase it too see what it does).

Once you have your payment options set up, go ahead and upload your program to your website. If you don't have one, you can use a free service such as Apple's iDisk to put your creation online. Once it's up, tell everyone. Write a press release and e-mail it to all the big news sites, create an entry at [versiontracker.com](http://www.versiontracker.com/)

(<http://www.versiontracker.com/>), tell all your Mac-owning friends, and put a big sign on top of your house telling everyone about it. Do whatever it takes to get word out! Once people know about your program and start using it, you have something else to do: customer support.

Handling your Users

Your users can be very helpful and polite, rude and abusive, and others can be complete novices who need their hands held to do even trivial things. It is important that you remain calm, helpful, and polite at all times. When you reply politely, help users as much as possible, and generally be a nice guy, people will treat you with respect. Listening to people who use your program often is vital to making a successful product. If you fix a bug or add a feature that someone has requested, you'll often get a very nice thank you from that person, and possibly a donation or registration for your program. That person will then tell people what a nice guy you are, and more people will use your program.

Feel free to ignore abusive people, as replying to a message that states "Your program sux! Make it better!" is usually a waste of time. However, computer novices should get help and attention, as they will appreciate it and you will get the same positive image as you would when listening to bug reports and feature requests. Even if you don't have time to write a lengthy tutorial there and then, reply to the person and tell them that you don't have time, but will contact them in a few days with better instructions. Oh, and don't forget to do so, as broken promises are bad. Customer relations are what people judge you and your company by, so don't let it slip. It can be a chore, but it is an important one.

Overview

There we have it. How Writer rose from a pathetic little project to a popular and profitable program. If you follow the advice here, you should be able to produce a program that avoids the "Made with REALbasic = crap" myth, and keep people happy once you've got your program going. Oh, I have one more piece of advice, which is to take your time! Writer is well made, but only because I've had years to work on it. Its bigger brother, Writer PRO, is more powerful, but has been a flop due to the fact that I rushed it and it isn't well built at all. Take a look at <http://www.kennett.net.co.uk/software/> and see how to make a good program, and how not to.

Interview

by Marc Zeedar

Andrew Barry: The inventor of REALbasic speaks with *RBD*

Tell us a little of your background. How did you get into programming? Were you always a Mac fan or did that come later?

I started programming approximately 23 years ago at the age of nine on my father's TRS-80. It originally came with 4k of memory.

My parents purchased me an Amiga when I was 17 — I finally had a real computer with a C compiler — and I taught myself C by looking at sample programs. I wrote my first true compiler that year.

But the Amiga slowly stagnated — and I started learning to program the Mac on a friend's machine. I wrote my first program on my friend's IIci — a mandelbrot generator — I didn't know anything about the Toolbox, but in true Amiga fashion I figured out where the screen buffer was located, and just drew the calculated mandelbrot values directly to the screen. Wheeee!

Once I'd finally managed to afford my own Mac (the just released IIsi), I started writing my first larger scale Mac application with my friends — we released the well received Mac shareware game "Prince of Destruction." This was followed up with a Windows version, since I was also becoming a capable Windows programmer.

But on the whole, I've always enjoyed developing for the Mac far better than developing for Windows — there's always been far fewer cases of pulling my hair out and screaming obscenities when writing Mac programs. The crashing and instability was the only real downside, but that's why I love Mac OS X.

REALbasic was originally called CrossBasic. What inspired you to create it? What were your goals?

Like many young programmers writing stuff in their spare time, I was an accom-

plished dilettante. Numerous projects were started, the clever coding completed, and then the project would be dropped since it was just too uninteresting to bother to complete it. I always enjoyed doing things that pushed the limits. For example, in the post-Doom era I was experimenting with writing texturing engines (of course everybody else was doing the same). I came up with a clever technique that did full screen texturing 640x480 at 30 fps, which was around 3x faster than anybody else was managing on a 6100/60. It never got developed past the proof of concept.

One day I realized that I was just wasting my time. What was the point of writing the software if it was never completed? I then resolved to carefully pick my next project and to follow through, no matter how much tedious code had to be written.

As mentioned before, I've always had a soft spot for development environments, and previously I've always had some form of compiler project, or ideal, that I'd been experimenting with. They made perfect sense to me, but it would be a steep learning curve for anybody else to pick up.

Then one day it struck me — people didn't want clever innovative development environments — they wanted something that was approachable and that they could get working with as soon as possible. Visual Basic for the Windows platform was a very handy tool with no real equivalent tool available on the Macintosh platform — and I believed there was an entire market segment that wasn't being served by the available environments.

So that evening at my favorite Chinese restaurant I resolved to create a development tool for the Macintosh that was similar to Visual Basic — to empower a whole new generation of programmers. Of

course, I wanted to make some improvements while I was at it....

Were you wanting CrossBasic to be able to run Visual Basic code or was that even a consideration? Why are there syntax differences between the languages? Were you ever worried about Microsoft being upset at you for making a similar product?

Full source compatibility wasn't much of a consideration. I wanted to maintain some degree of similarity to the BASIC standard, which Visual Basic was by default — but I wasn't looking for a pushbutton recompilation.

This is one of the things that I'm not sure I made the right decision about — though if I'd aimed for 100% VB compatibility this would have brought its own range of problems. If something is meant to be 100% VB compatible, and somebody buys it on that basis, they are going to be upset about anything that gives them trouble.

Contrast this with a product that only aims to be somewhat similar — since it's not being marketed/conceptualized as 100% VB compatible, then people realize they're going to have to do some work.

Perhaps it could have been intentionally more compatible.

I did have some worries that Microsoft would chase me with a litigation stick, but even though I am not a lawyer the legal precedents didn't seem to indicate that I would have a problem. On the other hand, all Microsoft had to do was keep me in court and bankrupt me that way. I was still single back then, and so I wasn't really risking that much by going forward.

What was more significant about CrossBasic: that it was a Mac version of Visual Basic or that it could cross-compile?

The most important thing about CrossBasic was targeting Mac applications. The original feature of cross-compiling to Java was a 'cool' side-project. Two things led to the Java cross-compilation being dropped. One, Java really sucked. Two, continuing to support the Java cross-compilation was limiting the development of CrossBasic. Oh, and the fact that only one percent of my user base was interested in cross-compiling to Java, and even those people were just dabbling.

In fact, a company approached me to buy the product in the early days — but it was made clear from the outset that all they were interested in was the BASIC cross-compiling to Java technology and the Mac targeting would be dropped. This might sound stupid, but I felt strongly about providing Mac programmers with a development tool and so I didn't bother to find out how much money they were offering.

What led to CrossBasic becoming REALbasic?

Geoff Perlman approached me because he liked the product. I knew my limitations (documentation was scant, and I had no time for marketing, etc.). There were a couple of other interested parties. They seemed more interested in taking over complete control, and I was not comfortable with that. I felt that originally Geoff and I shared a similar vision. I then turned over the intellectual property and RB was created out of CB.

You left REAL Software in 1999 right as REALbasic was gaining popularity. Any regrets in leaving?

Leaving REAL was difficult because RB was my "baby." Suffice to say I consider myself an artist in many ways. That left me open to creative differences with the new parent. I suppose I regret giving up my "baby," but in the end it has been a real learning experience and one that I have taken valuable lessons from — I won't make those mistakes again. So I have no regrets in that respect. I do, however, miss the RB community and having relationships with the end users. There was something gloriously fun about being able to fix end user problems and giving them a fix in a short amount of time rather than the extended process that most bugs go through with most applications.

What are you doing now?

Putting up with my wife's remodeling of our house, getting used to living in Australia again, and changing diapers on real babies!!! I have three daughters, who are beautiful, fun, and challenging. Unlike applications there is no way to "fix" them! ;-)

I also own my own company in Australia and we do a lot of custom apps and plugins. You are welcome to check out our company's website <http://www.barrysoftware.com> for current news on our projects, or send an email to say "Hi."

What do you think of the current version of REALbasic? What are the weakest and strongest points of the product?

I think RB is maturing nicely — though I must admit that I only use a small core of the product's features. I'm really happy about the progress of the Mac OS X port. That is the future of the Macintosh and it's good to see RB take a strong role.

The strongest point of the product has always been the ability to whip up a quick utility program that has a first class interface. RB has a lot of nice touches to help the user make sure that their user interface elements are properly aligned, something which can be really fiddly under Visual Basic.

As far as weak points are concerned, this really just applies to the sort of work you're doing. For my purposes, it really doesn't have any weak points. To different people with different requirements, there might be areas that could use further development — but REALbasic has to cover such a broad range of features they can't always make all of the people happy all of the time.

REALbasic's ease-of-use has spawned tons of new software on the market. Are there any particular RB programs that stand out as being most impressive?

Alas, I must admit to being insular. I don't use much software outside CodeWarrior, REALbasic, Mail, and iTunes (got to have my tunes while I work). But when I do my morning web surf, seeing software that's been written with RB brings a smile.

What aspect of REALbasic gives you the biggest sense of accomplishment?

Knowing that in some small way I accomplished what I had set out to do — make the lives of people easier in some way. REALbasic accomplished that in more than one way (ease of use, cross compiling,

etc.). CrossBasic and then REALbasic were created out of the altruistic need to help people (sounds corny, but true). I wasn't in it for the money. I am proud of the fact that what I did has helped people; not changed the world, but in some small way has made a difference. That's all I really wanted. I hope to make that difference again in the future.

Did you ever imagine REALbasic would be as popular as it is today?

I don't think I really envisaged it — it's one thing to imagine and develop the software. It's another thing to go into a local bookstore and see REALbasic programming books.

What do you think of Mac OS X?

I love it. Once I installed it I've never looked back. It strongly reminds me the first time I installed Windows NT after having used Windows 3.11 — everything just becomes "smoother." I'd played around with Linux in the past, but Mac OS X just felt like home.

How would you compare the Cocoa programming tools (Interface Builder, etc.) to REALbasic?

There is a whole mystique about Interface Builder that dates back to the NeXT days, when it was substantially better than the other types of tools available — but it really hasn't advanced much, and I believe that the REALbasic user has a far superior experience.

Apart from the tight integration between the UI editor and the code editor, and a cleaner interface, REALbasic also boasts better support for visually binding objects. It's a pity that nobody uses this feature, since it can take a lot of the irritation out of generating internally consistent UIs. (If you don't know what I'm talking about, try shift-command dragging between a push button and a listbox).

Is there an exciting new product you're working on you'd like to tell us about? Programming via telepathy, perhaps?

For the most part my Australian company has kept me busy with a lot of contracting work (kids are expensive), but I've recently found enough spare time to develop Bug Spray — a killer debugging app! It will be ready for a standard release soon.

Development environments are my forte and my love. I would like to move back to that end of things when I am able to because I feel that is where my greatest contribution lies.

Cover Story

by Marc Zeedar

The REALbasic Revolution

Now anyone can program

As a child, my favorite toy was my Lego set. I had no brothers or sisters, and growing up overseas, I often had to entertain myself. I'd spend hours with those Legos, building entire towns, creating strange new vehicles, and inventing mechanical devices that worked via a kludgy mixture of rubberbands and pulleys. I never had the fancy sets with battery-operated motors and special parts; I was forced to make do with generic pieces. This spurred my imagination and increased my problem-solving skills. I had disdain for my U.S.-resident cousins who always had the special kits and unique pieces. Where was the challenge in that?

Years later, I discovered computers. Computers in the early- and mid-eighties were a lot like Legos. They could be made to do anything you could imagine, with the only catch being that you had to figure out how to control them.

It was during those heady days of discovery that I realized the true beauty and power of a computer. A computer wasn't a single device like a typewriter or calculator. It was hundreds of devices, thousands, millions. By simply running a different program the computer became a different device. What a fantastic concept!

I fell in love with computers not because they let me rewrite my stories or because of a cool game or some neat graphics I saw, but because of their infinite potential.

Today's computers are more like today's Legos: there are special kits for all sorts of projects. You need something to organize your videotape collection? Bingo, buy

Marc's dream project is a virtual Lego set. Just think — no more missing pieces!



With REALbasic, your destination is only limited by your imagination.

a program. You need to retouch photographs? Send Adobe your money. Sure, there's some learning overhead and the program can't always do exactly what you want, but overall, the major work has been done for you.

But where's the challenge in that? What if you need something unique? What if you want to do something no one else has ever done? Perhaps your needs are so specialized there aren't any mass-market solutions for your situation.

The Custom Solution

The solution is simple: write your own software. In the early days of computers

this was standard. Big mainframes arrived empty and dumb: you had to program them to make them do something useful. Every personal computer shipped with a version BASIC, a fairly ease-to-use programming language.

As computers became more mainstream, however, writing your own software became less and less important. The average consumer today has about as much interest in learning to program their computer as a screwdriver owner wants to know how to build a house. Programming computers, like house-building, is complicated and time-consuming.

At least, it used to be.

Now REAL Software has created REALbasic, the most amazing development environment the Macintosh has ever seen. Fans of Apple's innovative HyperCard might protest that statement, but HyperCard hasn't been updated in years and never really took advantage of modern capabilities like QuickTime, or supported the Internet.

Why You Should Care

If you're used to buying shrinkwrapped software packages you might wonder why you'd want to bother learning how to program. Isn't that something better left to the experts?

Absolutely not. Programming is something everyone should learn. Programming skills will enhance every aspect of your life.

First, programming your computer to do something it couldn't do before is one of the most satisfying things in the world. It's like teaching your dog a new trick — you want to demonstrate it to everyone. Except, of course, a computer program is more obedient (and generally more useful). In

short, programming, despite the ominous connotations of the word, is fun!

Second, programming expands your mind. The only limits are your imagination. Want to invent a new computer game? Have an idea for a simple utility that would enhance your real estate business? Need to analyze gobs of data you've collected over the years? Hate how a scheduling program forces you to work? Write your own software! There are no rules when you're in charge.

Third, learning to program teaches you to think logically. No longer can you tell someone "Organize these files." Instead it's, "Sort these files alphabetically by last name, putting any duplicates in chronological order, and trash any obsolete ones." Precision helps with all forms of communication — people will understand you better, you'll understand yourself better, and your whole life will improve.

Fourth, you can save yourself time and money. Let's use a car analogy. Learning a little bit about how your car works, how to change your own oil, and even how to replace a few parts, is invaluable. No longer are you dependent on the price-gouging mechanic up the street. No longer do you shudder when your car makes that strange pinging sound when going up hill, wondering if you're in for another \$1,000 repair bill. Learning programming is the same thing: you're more confident in your computer knowledge, less likely to waste money on consultants or expensive one-shot programs. Sure, there's an initial investment of time and energy, but the pay-off is life-long.

Fifth, your software enhances the community. How much of your computer experience is made more positive by caring, generous people who donated their time and skills to make your life easier? Even if you've never installed a shareware or freeware program in your life, the Mac operating system is full of ideas and technology donated to the Mac community. There's the Internet control panel (formerly

known as the freeware Internet Config), Stuffit Expander and Dropstuff (based on the popular Stuffit compression standard invented by a 15-year-old student), plus Stickies, Windowshow, Boomerang, Apple Menu Options, Launcher, and dozens of

The software industry and ultimately the world will be changed by non-programmers being given the freedom to program.

others. The arrival of Mac OS X is starting this whole process over again.

If you go to any of the popular download sites or open a *MacAddict* CD you'll find thousands of free or practically free programs. Most of these are programs that would never succeed commercially — not because of poor quality, but simply because they appeal to a tiny, vertical market. How many people need a ciceros-to-centimeter calculator or a soccer statistics database? Those that do, of course, are incredibly grateful. But if it weren't for hobbyists or people donating their spare time, many of those programs would never be written.

Sixth, programming can make you money! Perhaps not much money, unless you start your own business and go to work full-time, but many people who originally wrote a program just for themselves discovered others were willing to pay \$10 or \$20 for it. Get a few hundred users and we're talking about some serious mad money. Programming skills could lead to new responsibilities at work or even a new career!

Get Started Now

Don't think that programming's something esoteric and complex and only for

ambitious Bill Gates clones. The fact is, any time you use your computer you are programming it! A computer does nothing without specific instructions. Programming is simply a form of instructing the computer. Double-clicking an icon to open a file is a form of instruction.

In the past decade programming has become much easier. Scripting languages like AppleScript have made basic programming techniques commonplace. People who never thought of themselves as programmers have suddenly found themselves writing Javascripts for the web, or using Lingo in Director.

But scripting systems are always limited. They are slow, don't let you fully access the power of your computer, and rarely provide you with the tools to create true Macintosh applications with standard interfaces.

REALbasic is different. It's revolutionary not because it's easy to use (HyperCard was easy to use) but because it creates Mac applications indistinguishable from programs written in high-level languages like C++ and Pascal. In fact, if you don't tell, chances are people will never know you wrote your program in REALbasic!

Because of REALbasic, thousands of people who never thought of themselves as programmers are programming. Traditional programmers are embracing RB because of its power and speedy development. Teachers are using it as an educational tool. Corporations are writing utilities in-house instead of hiring expensive consultants. High school students are starting their own shareware companies, creating useful and innovative solutions to rare problems. Scientists, graphic designers, artists, musicians, and others in non-programming fields are using REALbasic in creative and unexpected ways.

Who knows where this will lead? Perhaps the next Tetris will be created by someone fooling around with REALbasic. Perhaps interface innovations first seen in REALbasic programs will be adopted by Apple or Microsoft, revolutionizing the way we use our computers.

I believe the software industry and ultimately the world will be changed by non-programmers being given the freedom to program. Instead of suffering silently at the mercy of monolithic corporations who tell us "This is the way it is!" we can take the helm and set our own destination. After all, we're the users — we know how things should work, just not how to make them that way. Until now.

Welcome to the Revolution. Welcome to the future.



Figure 1: The logo for Z-Write. Doesn't it make sense that a writer, not a programmer, should write a word processor?



Three Ways to Animate

Choosing the best animation technique for your program

Nearly every software developer has a need to generate some animated graphics sooner or later. Animation is the bread and butter of game developers, but it can be used effectively in many other places as well, from an eye-catching About Box to a custom progress indicator. REALbasic provides several different ways to produce animation, each with pros and cons.

In this article, we'll pose a moderately simple animation problem: an image of a rocket flying over a starry background, trailing little puffs of smoke. Then we'll solve this in three different ways: using a simple Canvas, using a SpriteSurface, and using an Rb3DSpace. By the end, you should be well-equipped to use the right tool for any animation job.

The Problem

For the sake of this article, assume you need to make a rocket fly over a background of stars. Every now and then the rocket emits a little puff of smoke which should move in the direction opposite that of the rocket, expand a bit, and then disappear. If possible, the smoke puffs should be translucent (because translucency is cool).

So we'll be working with basically three images: the rocket itself, the background, and the smoke puff. The smoke puff actually requires a series of images, so we can make it expand and grow more diffuse. To reduce the number of files we have to work with, we'll combine all the smoke puff images into one picture. Since we want the smoke puffs to be translucent they also

need a mask; we'll put that into the same source picture too.

The three source pictures are shown in Figure 1. If you're creating your own images in order to follow along at home, be sure to use the same names and sizes or else you'll have to make some adjustments to the code.

All three solutions to this problem will involve these pictures, and the easiest way to access them is to simply drag them into your project and refer to them by name. So start with the default new project and add the pictures. The rocket image is designed for white areas to appear transparent, so select RocketImg in the project window and in the Properties window set Transparent to 1.

The downloadable project associated with this article will demonstrate all three techniques in one application, but to keep it simple you might want to use three separate projects. In any case, your project window should look like that in Figure 2.

The Canvas Solution

Let's first tackle this problem by using that Swiss Army knife of all things graphical, the Canvas. A Canvas is basically just a rectangular area of a window where we can draw things, and that's just what we want to do here.

Start by dragging a Canvas from the toolbar onto your window, and via the Properties window, set its Backdrop to BackgroundImg. You could set the width and height to 400 here or do it in code (the sample project does the latter, so we can see all three solutions while working in the IDE). In either case, the Canvas should now look like a starry square.

At this point, I'd be tempted to hit command-R to run the program. It works! It doesn't do anything except

display a bunch of stars, but apart from the rocket and the smoke puffs, it's completely done.

The next step is drawing a rocket. Thinking ahead a bit, we're going to have a timer or a loop that periodically updates the display. So make an "UpdateCanvas" subroutine within Window1. That will have code to grab the size of the rocket and compute its X (horizontal) position within the Canvas:

```
Dim x, w, h As Integer
```

```
x = CanvasDisplay.width / 2
```

```
w = RocketImg.width
```

```
h = RocketImg.height
```

and then draw the rocket into the Canvas' graphics property:

```
CanvasDisplay.graphics.DrawPicture RocketImg,  
x, 200, w, h, 0, 0, w, h
```

Of course, this method won't do any good if we don't call it. So let's go ahead and add that Timer. Drag a Timer control onto your window, and via the Properties window, set its Period to 10 (i.e. 10 milliseconds, which is about as fast as a Timer can go). Double-click it, and in its Action event type:

```
UpdateCanvas
```

Now run the program again. We now have a rocket drawn against a field of stars. That's definite progress, but the article title says "animation," so we'd better get it moving.

Create a new property of the window, "mShipY as Integer." This will be the Y (vertical) position of the ship within the display. Now return to the UpdateCanvas

Joe Strout is a senior software engineer at REAL Software, and likes to dabble in making games with REALbasic.

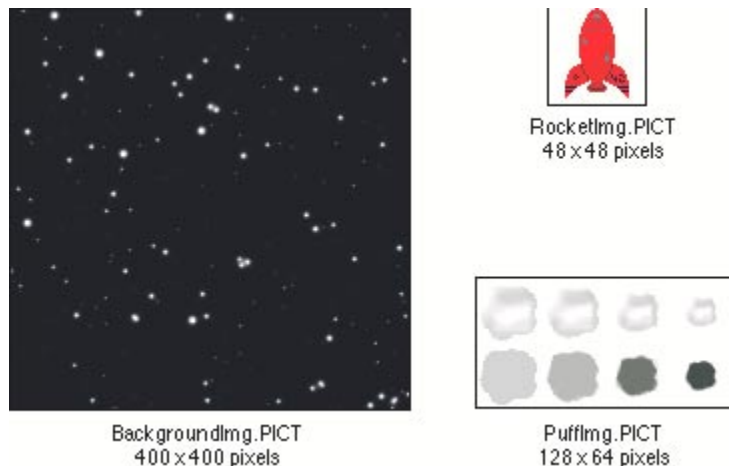


Figure 1: Images needed for the flying-rocket demonstration.

code. The DrawPicture call that draws the rocket needs to be replaced with a few lines that erase the old rocket (by drawing a section of the background where the rocket used to be), update the rocket position, and draw the new rocket, like so:

```
Dim g As Graphics

g = CanvasDisplay.graphics
// Erase the ship at the old position
g.DrawPicture BackgroundImg, x, mShipY, w, h,
x, mShipY, w, h

// Update the ship position
mShipY = mShipY - 4
if mShipY < -h then
    mShipY = CanvasDisplay.height
end if

// Draw the ship in its new position
g.DrawPicture RocketImg, x, mShipY, w, h, 0,
0, w, h
```

Note that to avoid typing “CanvasDisplay.graphics” multiple times, I’ve also declared a new local variable “g” and set it to mean the same thing. This will become even more convenient as our animation gets more complex.

The program at this point should display a rocket that flies up the screen, and when it disappears off the top, it reappears at the bottom. The problem is nearly solved; all that’s left is the puffs of smoke.

The smoke puffs require a little more bookkeeping because there may be several of them, and they change shape as well as move. We also need to carve up that PuffImg composite picture into separate

little pictures. So start by declaring a new property, “mPuffPics(3) As Picture” (allocating space for four pictures, numbered 0, 1, 2, and 3). Then add code like the following to the window’s Open event:

```
Dim i As Integer

// prepare the smoke puffs
for i = 0 to 3
    mPuffPics(i) = NewPicture(32, 32, 32)
    mPuffPics(i).graphics.DrawPicture PuffImg,
    0, 0, 32, 32, i*32, 0, 32, 32
    mPuffPics(i).mask.graphics.DrawPicture
    PuffImg, 0, 0, 32, 32, i*32, 32, 32, 32
next
```

This copies square 32-by-32 pixel sections of the original PuffImg into four separate images that include masks, so they’ll be translucent when drawn.

Now we have the images we want to draw, but we need some more properties to keep track of where to draw them and what state they’re in. So declare two more properties: “mPuffY(-1) As Integer,” and “mPuffAge(-1) As Integer.” Both of these are arrays which are initially empty (upper bound of -1). The first, mPuffY, will be used to keep track of the Y (vertical) position of each puff of smoke. The second, mPuffAge, will keep track of how long each puff has been around so we can make it change shape and eventually disappear as it ages. We’ll keep these two arrays in parallel, so that mPuffAge(i) is always the age of the puff to be drawn at mPuffY(i).

Now we can add a smoke puff by appending to these arrays. In the UpdateCanvas method, right before

updating the ship position, insert this code:

```
// Consider adding a smoke puff
if Rnd < 0.1 then
    mPuffAge.Append 0
    mPuffY.Append mShipY + h - 24
end if
```

This code has a 10% chance of generating a puff of smoke on each frame. When it does add a puff, it’s given an initial age of 0 and a position that is related to the ship position.

Next, we’ll need code to update and draw the smoke puffs; insert this right before we draw the ship:

```
// Update and draw the smoke puffs
for i = UBound(mPuffY) downto 0
    puffNum = 3 - mPuffAge(i) / 5
    if puffNum < 0 then
        mPuffY.Remove i
        mPuffAge.Remove i
    else
        mPuffAge(i) = mPuffAge(i) + 1
        mPuffY(i) = mPuffY(i) + 3
        g.DrawPicture mPuffPics(puffNum), x + 8,
        mPuffY(i)
    end if
next
```

This code appears a little complex, but it’s not doing too much. First we find the puff image number (puffNum) based on the puff age, dividing by five so that the image changes every five frames. The image number is going to be 3 for brand new puffs, and go down to 0 for old puffs; if it gets beyond 0, then we remove the puff entirely. Otherwise, we increment the puff age, move the puff downward (+3 in Y), and draw the chosen puff image.

If you run the project at this point, you’ll see pretty dense smoke as we’ve neglected to erase the smoke puffs once they’re drawn.

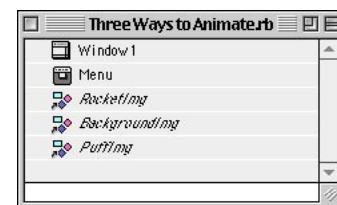


Figure 2: Sample project window, showing elements needed for this project.

So finally, we need to add a bit of erasing code right after erasing the ship:

```
// Erase the smoke puffs
for i = Ubound(mPuffY) downto 0
    g.DrawPicture BackgroundImg, x+8, mPuffY(i),
        32, 32, x+8, mPuffY(i), 32, 32
next
```

The logic here is exactly the same as for erasing the ship, except that we're doing it in a loop to make sure we get all the smoke puffs. (Note that for this particular case we could instead erase the entire column of ship and puffs with a single DrawPicture — this may or may not be faster than erasing each shape individually, depending on how many shapes there are and how much they overlap).

At this point, you should have a rocket that flies over a starry field and emits little animated puffs of smoke. But you may notice that in addition to flying and puffing, your display is doing a fair bit of flickering, too — at least, if you're not running in OS X. That's because we're erasing and then redrawing right on the screen, where everybody can see it (except in OS X, which automatically buffers all drawing). This flicker can be eliminated through a process called "double buffering."

Double buffering means that you do your erasing and redrawing in an off-screen picture first where it can't be seen; then you copy this picture to the screen. This eliminates flicker because the screen never appears in the "erased" state. Start by declaring a new property, "mCanvasBuffer As Picture." In the Open event we'll need to initialize this to have the same size and background as the display:

```
// initialize the canvas display
mShipY = 200
mCanvasBuffer = NewPicture(400, 400, 32)
mCanvasBuffer.graphics.DrawPicture
    BackgroundImg, 0, 0
```



Figure 3: The rocket, flying in an *Rb3DSpace*.

Then, in the UpdateCanvas method, we want to draw to this picture instead of to the Canvas itself. I added a checkbox called "DoubleBufChk" to the demo project so that double-buffering could be toggled at runtime; the graphics-selection code looks like this:

```
if DoubleBufChk.value then
    g = mCanvasBuffer.graphics
else
    g = CanvasDisplay.graphics
end if
```

Finally, after all the drawing is done, we need to copy from the buffer to the Canvas. This will be rather slow if we copy the whole thing, so it's important to copy a smaller region if possible. This particular animation takes place in a narrow column that encloses the rocket and smoke puffs, so we'll copy just that:

```
if DoubleBufChk.value then
    CanvasDisplay.graphics.DrawPicture
        mCanvasBuffer, x, 0, w, 400, x, 0, w, 400
end if
```

That's pretty much it for Canvas animation. There is one remaining snag, which affects you only if you're running on OS X and animating directly from a program loop rather than from a timer. Recall that OS X automatically buffers all drawing for you; when you think you're drawing to the screen you're actually drawing to an invisible window buffer. The system normally "flushes" this buffer to the screen during idle moments, such as in between Timer firings. But if you're animating from a tight loop there's no idling until the loop is over. So you may not see the animation at all; instead, you'd only see the last frame.

To fix this, you might try to call CanvasDisplay.Refresh. But the Refresh method of any control tells it to erase and redraw itself in its entirety, which is exactly what we don't want to happen here. So instead, we can use Declare statements to access the GetWindowPort and QDFlushPortBuffer function calls in CarbonLib. These tell the system to flush the window buffer to the screen immediately, instead of waiting for some later opportunity. Note that the call works but does nothing if running under Carbon in OS 8/9, so it's safe to call whenever TargetCarbon is true.

Listing 1 shows the UpdateCanvas method in its entirety. You'll also find this in the demo project on the *RB Developer* web site.

The SpriteSurface Solution

The SpriteSurface is a control specifically designed for doing 2D animation. As such, it's often the first (and sometimes the only) approach many RB users think of when considering an animation problem. As we'll see in this section, the SpriteSurface imposes some limitations on what you can do, but within those limitations the animation is considerably easier to code.

The initial set-up is very similar to that for the Canvas approach. Start by dragging a SpriteSurface onto the window from the controls palette. Name it SpriteDisplay, and set its background to BackgroundImg using the Properties window. Notice that unlike the Canvas, the background is not drawn while in the IDE; as soon as you run your project you should see the field of stars.

We'll need the same preparation of mPuffPics in the Open event (although in this case you could skip building the mask since it will be ignored). In addition, we need to create a sprite to represent the ship, as shown below.

```
// prepare the smoke puffs
for i = 0 to 3
    mPuffPics(i) = NewPicture(32, 32, 32)
    mPuffPics(i).graphics.DrawPicture PuffImg,
        0, 0, 32, 32, i*32, 0, 32, 32
next

// initialize the sprite display
mShipSprite = SpriteDisplay.NewSprite(RocketImg,
    , 200, 200)
```

Here, "mShipSprite" is a new window property of type Sprite. The last line creates a sprite at position X=200, Y=200, using RocketImg as its shape. Now if you run again, you should see a rocket stationary against a field of stars.

To make it animate we'll again use a Timer, but have that Timer call a method called "UpdateSprites." Initially, we'll have that method simply move the ship:

```
// Move the ship
mShipSprite.Y = mShipSprite.Y - 4
if mShipSprite.Y < -RocketImg.height then
    mShipSprite.Y = SpriteDisplay.Height
end if

// Update the display
SpriteDisplay.Update
```

Run the program now and the ship should be moving smoothly over the starry background. That's it — no erasing, no buffering, no flushing. Just change the X

Code Listing #1: Window1.UpdateCanvas

```
Sub UpdateCanvas()  
    Dim g As Graphics  
    Dim x, w, h As Integer  
    Dim puffNum, i As Integer  
  
    x = CanvasDisplay.width / 2  
    w = RocketImg.width  
    h = RocketImg.height  
    if DoubleBufChk.value then  
        g = mCanvasBuffer.graphics  
    else  
        g = CanvasDisplay.graphics  
    end if  
  
    // Erase the ship at the old position  
    g.DrawPicture BackgroundImg, x, mShipY,  
        w, h, x, mShipY, w, h  
  
    // Erase the smoke puffs  
    for i = Ubound(mPuffY) downto 0  
        g.DrawPicture BackgroundImg, x+8,  
            mPuffY(i), 32, 32, x+8, mPuffY(i),  
            32, 32  
    next  
  
    // Consider adding a smoke puff  
    if Rnd < 0.1 then  
        mPuffAge.Append 0  
        mPuffY.Append mShipY + h - 24  
    end if  
  
    // Update the ship position  
    mShipY = mShipY - 4  
    if mShipY < -h then  
        mShipY = CanvasDisplay.height  
    end if  
  
    // Update and draw the smoke puffs  
    for i = Ubound(mPuffY) downto 0  
        puffNum = 3 - mPuffAge(i) / 5  
        if puffNum < 0 then  
            mPuffY.Remove i  
            mPuffAge.Remove i  
        else  
            mPuffAge(i) = mPuffAge(i) + 1  
            mPuffY(i) = mPuffY(i) + 3  
            g.DrawPicture mPuffPics(puffNum),  
                x+8, mPuffY(i)  
        end if  
    next  
  
    // Draw the ship in its new position  
    g.DrawPicture RocketImg, x, mShipY, w,  
        h, 0, 0, w, h
```

or Y property of a sprite, call Update on the display, and repeat; sprite animation is as simple as that.

Let's finish the job by adding smoke puffs. First, we must note that the SpriteSurface control does not support masks. Sprites are transparent where the image pixels are white, and opaque where the image pixels are non-white; the mask is completely ignored. This means our smoke puffs are going to look more like solid balls of cotton than diffuse clouds of smoke. Having accepted that, let's get to the code.

We'll use the same concept of puff "age" as we used for the Canvas. So creating a new smoke puff involves appending a "zero" to an array of puff ages, and creating a new sprite on a parallel array of sprites (which we'll imaginatively call "mPuffSprites"). In addition, we want to set the priority of the puff sprites to -1; this causes them to be drawn behind the ship, which by default has a priority of 0. The code to be added to the UpdateSprites method looks like this:

```
// Consider adding a smoke puff  
if Rnd < 0.1 then  
    mPuffAge.Append 0  
    sp = SpriteDisplay.NewSprite(mPuffPics(3),  
        mShipSprite.x+8, mShipSprite.Y + 24)  
    sp.priority = -1  
  
    // If double-buffering, copy from the  
    // buffer to the screen  
    if DoubleBufChk.value then  
        // If possible, copy only the part  
        // that has changed -  
        // though in a less trivial animation,  
        // this can be difficult:  
        CanvasDisplay.graphics.DrawPicture  
            mCanvasBuffer, x, 0, w, 400, x, 0,  
            w, 400  
  
        // If necessary, copy everything -  
        // this always works,  
        // but it can be quite a bit slower:  
        'CanvasDisplay.graphics.DrawPicture  
            mCanvasBuffer, 0, 0  
    end if  
  
    // Finally, in Carbon (OS X), we need to  
    // flush the port buffer  
    #if TargetCarbon  
        Declare Function GetWindowPort Lib  
            "CarbonLib" (window as WindowPtr)  
            as Integer  
  
        Declare Sub QDFlushPortBuffer Lib  
            "CarbonLib" (port as Integer,  
            region as Integer)  
  
        QDFlushPortBuffer GetWindowPort(self),  
            0  
    #endif  
End Sub
```

```
mPuffSprites.Append sp  
end if
```

Running at this point should produce puffs that periodically appear behind the ship, but don't move or disappear. To animate the puffs we'll use a loop which again is very similar to that used by the Canvas. The main difference is that we now need to update the mPuffSprites array as well as mPuffAge.

```
// Update the smoke puffs  
for i = Ubound(mPuffSprites) downto 0  
    mPuffAge(i) = mPuffAge(i) + 1  
    mPuffSprites(i).Y = mPuffSprites(i).Y + 3  
    puffNum = 3 - mPuffAge(i) / 5  
    if mPuffAge(i) mod 5 = 0 then  
        if puffNum < 0 then  
            mPuffSprites(i).Close  
            mPuffSprites.Remove i  
            mPuffAge.Remove i  
        else  
            mPuffSprites(i).image =  
                mPuffPics(puffNum)  
        end if  
    end if  
next
```

So, on every frame we move each puff down 3 pixels and increment the age. On every fifth frame we pick a different shape, or if we've run out of shapes, we delete the puff by closing the sprite and removing the corresponding entries from the two arrays.

And that's all there is for this solution. The complete UpdateSprites method is shown in Listing 2.

The Rb3D Solution

The third control we'll consider for this problem is the Rb3DSpace. This control is a view onto a virtual 3D world. "But wait," I can hear the reader saying, "This is a 2D problem, so of what use is a 3D control?"

As it happens, an Rb3DSpace can be set up to do very nice 2D sprite-like graphics. The trick is to move the virtual camera very far away from the scene, and set the field of view to a very narrow angle. This is equivalent to looking across town with a telescope, and under these conditions there is very little perspective effect. This provides a nice undistorted view for displaying pictures — but as a bonus, we also get to treat them as 3D objects for special effects like scaling or rotation.

Let's start by dragging an Rb3DSpace control from the controls palette onto the window. Name it Rb3DDisplay and set

the SkyColor to a deep blue, matching the background color of the star field. Also set the FieldOfView to 10 (meaning ten degrees), and prepare the lighting by setting AmbientLight to 100 and FloodLight to 0.

In the Open event we'll finish preparing the Rb3DSpace as follows:

```
Dim degrees, angle, distance As Double
Dim bg As Object3D

if Rb3DDisplay.objects <> nil then
    degrees = 0.0174532925 // (radians per
        degree)
    angle = Rb3DDisplay.fieldOfView * degrees *
        0.5
    distance = 200 / sin(angle)
    Rb3DDisplay.camera.position.Z = distance
    Rb3DDisplay.hither = distance - 10
    Rb3DDisplay.yon = distance + 10

    mShipObj = New Object3D
    mShipObj.AddShapePicture RocketImg, 1.0
    Rb3DDisplay.objects.append mShipObj

    bg = New Object3D
    bg.AddShapePicture BackgroundImg, 1.0
    bg.position.Z = -1.0
    Rb3DDisplay.objects.append bg
end if
```

This requires a new property, "mShipObj As Object3D." Note that we check to make sure Rb3DDisplay.objects is non-nil before attempting to use any 3D methods. This is important, because the Rb3D classes require a system library (Quesa or QuickDraw 3D) to be installed. If it is not installed most 3D methods will fail and most properties of the Rb3DSpace will be nil. So a check like the above serves to handle that case gracefully.

The calculations after the check determine how far away the camera should be in order to make one unit of 3D space come out to one pixel on the screen. The camera is then positioned accordingly, and the Hither and Yon values are set to bracket that distance (see the RB language reference for more info on these properties).

Next, we create the ship and the background using the AddShapePicture method to convert our 2D pictures into objects in a 3D world. The final parameter to this call is the scaling factor — by specifying 1.0, we're making one pixel in the picture correspond to one unit of distance in the 3D space; a great convenience for this sort of animation.

If you run your application at this point, you should have a rocket in the middle of the screen against a field of stars. Not very exciting yet, but you know that under the hood that's a 3D rendering, and that lets us do some things that would be very difficult with the other approaches. For starters, let's make the rocket fly in a circle instead of a straight line.

Start by adding a new property, "mShipAngle as Double." This will hold the angle the ship is currently facing, which also corresponds to the ship's position along its circular path. Then make an UpdateRb3D subroutine, with code like the following:

```
Dim radius As Double

radius = 100.0
mShipObj.position.X = radius * cos(mShipAngle)
mShipObj.position.Y = radius * sin(mShipAngle)
mShipObj.orientation.SetRotateAboutAxis 0, 0,
    1, mShipAngle

Rb3DDisplay.Update

mShipAngle = mShipAngle + 0.05
```

This simply does a bit of trigonometry to find the ship position, and uses SetRotateAboutAxis to rotate the ship. We then increment the ship angle by 0.05 radians on each frame, which drives the animation. Run your app now and you should see the ship flying smoothly in a circle — a feat that would leave the SpriteSurface or Canvas dumbfounded.

Let's complete the task by adding some smoke puffs. We'll have to keep track of a bit more data in this case because each puff can be moving in a different direction now, rather than all the puffs moving downward. Also, we'll want to prepare a prototype 3D object representing a smoke puff, which we can then clone for each new puff we need to display. So add two more properties, "mPuffPrototype As Object3D" and "mPuffVelocity(-1) As Vector3D." Now add some code to the Open event as follows:

```
mPuffPrototype = New Object3D
for i = 0 to 3
    p = NewPicture(32, 32, 32)
    p.graphics.FillRect 0, 0, 32, 32
    p.graphics.DrawPicture mPuffPics(i), 0, 0
    mPuffPrototype.AddShapePictureWithMask p,
        mPuffPics(i).mask, 1.0

next
```

This initializes the puff prototype with four different shapes, each created from

the corresponding puff picture. Note that we don't pass mPuffPics(i) directly to AddShapePictureWithMask. The mask used with Rb3D acts a bit differently than it does in other contexts; the color pixels have to be pre-multiplied by the mask. To do that, we draw the image on top of a black background, and then pass this composite image on to Rb3D.

Finally, we need some additional code in the UpdateRb3D routine to add, move, and delete the puffs. I'll just point out a few highlights and refer to Listing 3 for the

Code Listing #2: Window1.UpdateSprites

```
Sub UpdateSprites()
    Dim sp As Sprite
    Dim i, puffNum As Integer

    // Move the ship
    mShipSprite.Y = mShipSprite.Y - 4
    if mShipSprite.Y < -RocketImg.height then
        mShipSprite.Y = SpriteDisplay.Height
    end if

    // Consider adding a smoke puff
    if Rnd < 0.1 then
        mPuffAge.Append 0

        sp = SpriteDisplay.NewSprite(mPuffPics(
            3), mShipSprite.x+8,
            mShipSprite.Y + 24)

        sp.priority = -1
        mPuffSprites.Append sp
    end if

    // Update the smoke puffs
    for i = UBound(mPuffSprites) downto 0
        mPuffAge(i) = mPuffAge(i) + 1
        mPuffSprites(i).Y = mPuffSprites(i).Y
            + 3
        puffNum = 3 - mPuffAge(i) / 5
        if mPuffAge(i) mod 5 = 0 then
            if puffNum < 0 then
                mPuffSprites(i).Close
                mPuffSprites.Remove i
                mPuffAge.Remove i
            else
                mPuffSprites(i).image =
                    mPuffPics(puffNum)
            end if
        end if
    next

    // Update the display
    SpriteDisplay.Update

End Sub
```

full code. First, we get a puff's initial velocity by transforming a vector by the ship's orientation:

```
v = New Vector3D
v.y = -3.0
v = mShipObj.orientation.Transform(v)
mPuffVelocity.Append v
```

We start out with a velocity of -3 in Y, which would move the puff 3 units downward per frame. But then we transform this by the ship's orientation so that this direction rotates to match the ship. We could have instead done some trigonometry like we did for the ship position, but this approach is simpler.

Next, notice the line "obj.position.z = -0.1" where we create the smoke puff. There is no notion of "priority" in a 3D rendering; instead, if you want one object to appear behind another, you move it further away. Our camera position has a positive Z, and the ship is at Z = 0, so by putting the smoke puffs at Z = -0.1 we ensure that they're drawn behind the ship.

Finally, it's worth pointing out how the puff animation works. It's so simple that you might have missed it:

```
obj.shape = puffNum
```

This line tells Rb3D that of all the shapes that have been added to this object, "puffNum" (a number from 0 to 3 in our case) is the one that should be displayed. This is different from the SpriteSurface, where you attach a new image, or the Canvas, where you decide what to draw at drawing time. With Rb3D, you attach all the images (shapes) you might need to the object when you're setting it up and then just flip between them by assigning to the Shape property.

With the code as shown in Listing 3, you should now have a rocket that travels in a circle emitting translucent puffs of smoke that move and dissipate believably, as in Figure 3.

Which to Use?

We've explored the three main approaches to animation in REALbasic. None of them is clearly superior for all situations; you'll want to select the right approach for each job. Here are some of the considerations to keep in mind.

First, simplicity of code. The SpriteSurface generally requires the least code to make things move around over a background without flicker. The Rb3DSpace generally requires a bit more set-up, and a Canvas requires more work to make smooth animation. The Canvas also requires, under OS X, a Declare to force the window to update; the SpriteSurface and Rb3DSpace handle that automatically.

Next, consider features. The Canvas and Rb3DSpace can both do scaling and translucency; an Rb3DSpace can also do rotation. A SpriteSurface supports none of these. However, we didn't consider collision detection in this article. SpriteSurfaces have easy pixel-level collision detection; with a Canvas or Rb3DSpace you'll have to detect your own collisions if you need them.

Performance is harder to quantify because it varies a great deal with the details of the task. Canvas animations can outperform other approaches when the area that changes on each frame is

**[No method]
is clearly
superior for
all situations;
you'll want
to select the
right approach
for each job.**

Code Listing #3: Window1.UpdateRb3D

```
Sub UpdateRb3D()
    // Let's make the ship fly in a circle (because we can!).
    Dim radius As Double
    Dim i, puffNum As Integer
    Dim grp As Group3D
    Dim pos, v As Vector3D
    Dim obj As Object3D

    // Move the ship
    radius = 100.0
    mShipObj.position.X = radius * cos(mShipAngle)
    mShipObj.position.Y = radius * sin(mShipAngle)
    mShipObj.orientation.SetRotateAboutAxis 0, 0, 1, mShipAngle

    // Consider adding a puff
    if Rnd < 0.1 then
        mPuffAge.Append 0
        v = New Vector3D
        v.y = -3.0
        v = mShipObj.orientation.Transform(v)
        mPuffVelocity.Append v
        obj = mPuffPrototype.Clone
        obj.position.x = mShipObj.position.x
        obj.position.y = mShipObj.position.y
        obj.position.z = -0.1
        Rb3DDisplay.objects.Append obj
    end if

    // Update the puffs
    // (which we'll assume are all objects in the Rb3DSpace from 2 on)
    for i = UBound(mPuffVelocity) downto 0
        puffNum = 3 - mPuffAge(i) / 5
        if puffNum < 0 then
            mPuffVelocity.Remove i
            mPuffAge.Remove i
            Rb3DDisplay.objects.Remove(i+2)
        else
            obj = Rb3DDisplay.objects.Item(i+2)
            obj.shape = puffNum
            v = mPuffVelocity(i)
            pos = obj.position
            pos.x = pos.x + v.x
            pos.y = pos.y + v.y
            pos.z = pos.z + v.z
            mPuffAge(i) = mPuffAge(i) + 1
        end if
    next

    // Update the display and ship angle
    Rb3DDisplay.Update
    mShipAngle = mShipAngle + 0.05
End Sub
```

Continued on page 27



Expressions of Delight REALbasic's regex class

The `Regex` class, along with the `RegexMatch`, `RegexOptions`, and `RegexException` classes, is the gateway to REALbasic's implementation of regular expressions. Regular expressions are a way of expressing a textual find or find-and-replace that's too complicated, or too vague, for a function like `InStr` or `Replace`. To see what I mean, let's take an example.

Suppose you've got a string representing some HTML, and you want to remove all the HTML markup from it. An HTML tag starts with a left angle-bracket and ends with a right angle-bracket; the tag consists of both angle-brackets, and everything in between, like this: "`<TAG>`". The trouble is, of course, that you don't know in advance what "everything in between" consists of.

So how would you find and remove an HTML tag using just `InStr`? You'd have to take a piecemeal approach. First you'd have to find a left angle-bracket, and remember where it is. Then you'd look for a right angle-bracket that comes after the left angle-bracket, and remember where it is. Then you'd have to break up the string into three pieces — what precedes the tag, the tag itself, and what follows the tag — and reassemble it without the middle piece, thus deleting the tag. Here's some actual code.

```
dim s, leftPart, rightPart as string
dim starting, ending as integer
```

```
s = // some text with HTML tags

starting = inStr(s, "<")
if starting > 0 then
    ending = inStr(starting, s, ">")
    if ending > 0 then
        leftPart = mid(s, 1, starting - 1)
        rightPart = mid(s, ending + 1)
        s = leftPart + rightPart
    end
end
```

That's not horrible — I purposely chose a simple example to start with — but it's not very pleasant either. The code is fairly illegible, and it feels like we're working much too hard. The notion "a left angle-bracket, the following right angle-bracket, and everything in between" seems simple enough. Yet we can't express it with `InStr`, so we're having to implement it as a succession of finds plus a sort of brute-force replacement, all of which is ugly, error-prone, tedious, and not very general; imagine having to extend this into a loop where we remove *every* HTML tag! Not much fun. Using regular expression syntax, however, we *can* express it, very simply, like this: `<.*>`

Regular expression syntax uses some symbolism that may at first be strange to you. But you can probably guess what's going on in this particular expression. The angle-brackets mean angle-brackets, and the dot-asterisk means "everything in between." That's all there is to it. To demonstrate this expression in action, here's some actual code for removing an HTML tag from a string using the `Regex` class.

Example 1: Regex removal of HTML tag

```
dim s as string, r as regex
r = new regex
s = // some text with HTML tags
r.options.greedy = false // disable greediness
r.searchPattern = "<.*>"
r.replacementPattern = ""
s = r.replace(s)
```

I hope the cleanliness and elegance of that example feels sufficiently compelling that you're encouraged to want to learn more — because, make no mistake, there is a learning curve to using regular expressions. The good news, though, is that regular expression syntax has achieved near universality in the computer world, and REALbasic's implementation of regular expressions is based on a widely used freeware code library called PCRE (Perl-compatible regular expressions). Once you've learned how to use regular expressions in REALbasic, you've also learned how to use them in BBEdit, JavaScript, Perl, Python, and PHP; regular expressions are also used (with a slightly different syntax) in Nisus Writer and Microsoft Word. So they are certainly worth learning about.

This article can't teach you all about regular expressions; the subject is huge. I strongly recommend Jeffrey Friedl's book, *Mastering Regular Expressions* from O'Reilly and Associates, and REALbasic's online help for `Regex` provides a complete guide to the syntactical details. What I'll do here is introduce you to regular expressions, and explain how you use them through REALbasic's `Regex`-related classes.

Search Expression

There are two kinds of regular expressions: the search expression, describing

Matt Neuburg first learned about regular expressions while using the Nisus word processor in 1990, and hasn't had a good night's sleep since.

the text you want to look for; and the replace expression, describing the text (if any) that will replace the found text. The search expression is far more powerful and, since you'll always need one, more important.

Before we start, I have to tell you the basic rule of how regular expression syntax works. In a regular expression, certain characters, such as the dot and the asterisk in Example 1, have special meaning. The basic rule of regular expressions is that if a character has no special meaning, it just represents itself in the normal way, like the angle brackets in Example 1. If a character does have special meaning and you want to use it to represent itself, without that special meaning, you "escape" it by putting a backslash (\) in front of it; for example, * means an ordinary asterisk. Also, if a character does not have special meaning, you can sometimes give it special meaning by putting a backslash in front of it. For example, r is just a normal "r", but \r means a return character. I know this sounds confusing, but it will be clearer when we look at some more examples, and if I don't tell you about it up front we can't get started at all.

The best way to approach an understanding of regular expressions is to consider that when we use InStr, every character in the search expression represents an exact match, whereas the power of a regular search expression lies largely in its ability to be deliberately vague about what we're looking for. There are two main kinds of things you get to be vague about: what individual characters to look for, and how many characters to look for. We'll take these in turn.

We begin with vague individual characters. What we want here is to make a single character in the search expression stand for more than one possible character to look for. To do so, we list the acceptable possibilities inside square brackets. This is called a character set. For example, [aeiou] means a single character that might be a or e or i or o or u. This notation would get tedious if there were lots of acceptable possibilities, so you can invert it by making the first character inside the square brackets a caret; now you've got a list of all the unacceptable possibilities. So, [^aeiou] means any single character that *isn't* a or e or i or o or u.

Also, a range of characters, using ASCII order, can be specified by putting a hyphen between the first and last characters of the range; so [0-9] means any numeric digit, and [0-9A-F] means any character that might be used as a hexadecimal digit. This

is still rather tedious when a character set is very frequently used, so the syntax defines some character sets for you in advance. For example, instead of [0-9] you can just say \d and instead of [0-9a-zA-Z_] you can say \w. And a dot (.) means any character at all. Now let's turn to vague quantities of characters. To specify that a character can occur a vague number of times, you put a metacharacter after the character that is to be repeated. For example, a plus sign (+) means that the preceding character must appear at least once but can occur more times than that, in succession. Note that this doesn't mean that the *very same* character has to appear several times in succession, because the character to be repeated might be a vague character. For example, [aeiou]+ means any stretch of any vowels, and will find the "eau" in "beautiful." A question mark (?) means that the preceding character may appear once or perhaps not at all, and an asterisk (*) means that the preceding character may appear once, not at all, or any number of times in succession. Now you can understand how we found an HTML tag with the expression <.*> earlier; it means a left angle-bracket, a right angle-bracket, and any characters in any quantity between them.

When specifying vague quantities, you must be concerned about "greediness." A greedy search is one that matches the largest stretch it can find. Recall that in Example 1, we disabled greediness before starting the search. To see why, imagine searching a string with two HTML tags in it. A greedy search finds everything from the start of the first HTML tag to the end of the second HTML tag. A non-greedy search finds just the first HTML tag, as desired.

Performing the Search

To perform a search you need three things: a string to look inside, a regular search expression, and an instance of the Regex class. You hand the Regex instance the search expression as its SearchPattern property, and then send the Regex instance the Search message. If you provide just one parameter for the Search message, that parameter is the string to look inside, and the search will start at the first character of the string. If you provide two parameters, the second parameter is the index of the character where the search should start. The character indexing is zero-based! This contrasts unfortunately with InStr, where character indexing is one-based.

The value returned when you send a Regex instance the Search message is an instance of the RegexMatch class.

You consult this instance to learn about the results of the find. If the find failed, the RegexMatch instance will be nil. If the find succeeded, the RegexMatch's SubexpressionString(o) property will contain the matched substring, and its SubexpressionStart(o) property will contain the index (zero-based again) of the first character of the matched substring within the original string. You're probably wondering what the "(o)" means here, but don't worry about it for now; just trust me.

To illustrate, here's a code snippet that counts the number of runs of vowels in a string. We do this by finding vowel runs in successive iterations of a loop until the find fails. Each time through the loop, we use the start position and length of the previously matched substring to determine where to start the next search.

Example 2: Successive searches

```
dim s as string, r as regex, m as regexmatch
dim i, count as integer

r = new regex
s = "beautificalion"
// has 5 vowel runs: "eau", "i", "i", "a", "io"

r.searchPattern = "[aeiou]+"

do
    m = r.search(s,i)
    if m <> nil then
        count = count + 1
        i = m.subExpressionStart(0)
        i = i + len(m.subExpressionString(0))
    end
loop until m = nil

msgbox str(count)
// result is 5, the right answer
```

It seems wasteful to have to supply the search string as a parameter to the Search message every time through the loop when that string isn't changing. To save us from having to do this, the Regex class permits a different way of using the Search message. Having performed the search once, so that the Regex instance knows what the search string is, we set the starting position for the next search using the Regex instance's SearchStartPosition property and send it the Search message with no parameters at all. We can rewrite Example 2 to use this syntax.

Example 3: Successive searches, alternate syntax

```
dim s as string, r as regex, m as regexmatch
dim i, count as integer
r = new regex
s = "beautification"
r.searchPattern = "[aeiou]+"
m = r.search(s)
while m <> nil
    count = count + 1
    i = m.subExpressionStart(0)
    i = i + len(m.subExpressionString(0))
    r.searchStartPosition = i
    m = r.search()
wend
msgbox str(count)
```

Parentheses and Subexpressions

In a regular search expression parentheses have two functions. One is simply to group things. For example, you might want to use a plus-sign to indicate a repetition, not of a single character, but of a more extended regular expression. To do so, you'd group what precedes the plus-sign in parentheses; the plus-sign would then apply to the group as a whole. Thus the expression `(p+e)+r` would match "pepper": first we match the single "p" and the "e" that follows it; then we try to do it again, and we succeed, matching the double "p" and the "e" that follows it; then we try to do it again, and we fail; so we look to see if an "r" follows, and it does, so we stop with a successful match.

The other use of parentheses in a search expression is to demarcate a substring of whatever the search expression finds. This is useful, for instance, when you have to make a search expression where what interests you about the result is not the entire found string but only a certain part of it. For example, suppose you want to find a word starting with "anti," but you don't care about the whole word; you're interested in what's being opposed, so what you really want to know is what follows the "anti." The regular search expression `anti(\w*)` will perform the search; but what do the parentheses do here? They allow us to refer to the relevant substring of whatever is found. The stuff found by what's in the parentheses of a search expression is called a subexpression of the result. In this case, it is subexpression 1, and you can extract it from the resulting `RegexMatch` instance with the `SubexpressionString(1)` property and get its position with the `SubexpressionStart(1)` property.

Example 4: Subexpression

```
dim r as regex, m as regexmatch
r = new regex
r.searchPattern = "anti(\w*)"
m = r.search("The antithesis of synthesis.")
msgbox m.subexpressionString(1)
// result is "thesis"
```

You can now understand what the "(o)" is for in Examples 2 and 3. In a `RegexMatch` instance, subexpression 0 is the entire match result. Any other subexpressions are parts of the result demarcated by parentheses in the search expression. The rule for how they are numbered is simple: just count left-parentheses from left to right. So, for example, if the search expression were `((anti)(\w*))`, then the material following the "anti" would be subexpression 3.

Another interesting use of subexpressions is to refer to them within the search expression. This is done by number, preceded by a backslash; so, `\3` means subexpression 3. You use this to search for material containing repetition at a distance. For example, the expression `\w*(\w)\w*\1\w*` looks for a word containing the same character twice; it matches "metre" which contains two e's, but it doesn't match "metric" where no two letters are the same.

Search and Replace

To perform a search-and-replace, you assign a regular replace expression to the `Regex` instance's `ReplacementPattern` property and send it the `Replace` message. Regular replace expressions can refer to subexpressions, using the same `\3` notation we just talked about; otherwise they are pretty much just ordinary strings. The syntax of the `Replace` message is just like that of the `Search` message. The result is a string where the replacement has been inserted into the original in place of the found match.

Example 1 has already provided an illustration of search-and-replace. Here's another, using a subexpression reference. Suppose we've got a string from an email message where the author indicated emphasis by surrounding words with asterisks. We want to turn this to HTML; we will find a stretch surrounded by asterisks and replace the asterisks with "" and "".

Example 5: Search-and-replace

```
dim r as regex, s as string
r = new regex
s = "This is a *very* important message."
r.searchPattern = "\*(.*)\*"
r.replacementPattern = "<B>\1</B>"
r.options.greedy = false
s = r.replace(s)
// result is "This is a <B>very</B> important message."
```

Example 5 is very typical of a regular expression search-and-replace. We look for a stretch of text consisting of two asterisks and everything in between, using syntax we're now very familiar with (Notice the use of a backslash to show we mean an actual asterisk.). But we're only interested in what's between the asterisks; we want to throw away the asterisks themselves. So in the search expression we demarcate what comes between the asterisks as a subexpression. That way we can refer to it in the replace expression, which consists of "" and "" surrounding whatever turned out to be between the asterisks. The result, in this particular case, is the replacement string "very". That whole replacement string then replaces the whole found string in the original string; thus, we end up with just what we started with except that the asterisks are gone and the HTML markup is inserted.

We can take this even further, finding and replacing with HTML *all* asterisked expressions in a string that contains several of them. It's simply a matter of inserting one line before performing the replace, specifying that the replace should be global:

Example 5a: Search-and-replace, global

```
...
s = "This *is* a *very* important message."
...
r.options.replaceAllMatches = true
...
// result is "This <B>is</B> a
<B>very</B> important message."
```

Another way to do a replace is to perform a search and then send the `Replace` message to the resulting `RegexMatch` instance. You can use a regular replace expression as parameter, or omit the parameter if you already set the `Regex` instance's `ReplacementPattern` property. The result is the replacement string alone, not the replacement string inserted into the original. In this example, we look for

a seven-digit phone number in any of several forms — “555-1234,” “5551234,” or “555 1234” — and render it into canonical form using a hyphen.

Example 6: Search-and-replace, extracted

```
dim r as regex, m as regexMatch
dim s, canonical as string
s = "The phone number is 5437890."
r = new regex
r.searchPattern = "(\d\d\d)([- ]?)(\d\d\d\d)"
m = r.search(s)
canonical = m.replace("\1\3")
// result is "543-7890"
```

Since the search result remains sitting in the `RegexMatch` instance, you can now proceed to perform a different replacement without performing the entire search over again.

Example 6a: Search-and-replace, extracted, repeated

```
dim r as regex, m as regexMatch
dim s, canonical, noncanonical as string
s = "The phone number is 5437890."
r = new regex
r.searchPattern = "(\d\d\d)([- ]?)(\d\d\d\d)"
m = r.search(s)
canonical = m.replace("\1\3")
// result is "543-7890"
noncanonical = m.replace("\1 \3")
// result is "543 7890"
```

You can also take advantage of the Perl operators `$`` and `$'` as metacharacters in a regular replace expression; they stand for the part of the original string respectively preceding and following the found string.

Example 7: Prematch operator

```
dim r as regex, m as regexMatch
dim s, prematch as string
s = "The phone number is 5437890."
r = new regex
r.searchPattern = "(\d\d\d)([- ]?)(\d\d\d\d)"
m = r.search(s)
prematch = m.replace("$`")
// result is "The phone number is "
```

Options

We had occasion in the preceding examples to refer to the `Options` property of a `Regex` instance. This is an instance of the `RegexOptions` class, which has various properties whose values determine the behavior of subsequent searches. You can examine the online help for `RegexOptions`

to study these properties. Besides greediness and whether a search-and-replace should be global, there's a setting for case sensitivity. There are also a number of settings having to do with the treatment of line endings. The reason for these is partly that different platforms use different line-ending characters, and partly that regular expressions were developed in a context of being applied to just one line (or paragraph) of text at a time, which might or might not be the behavior you want.

Exceptions

If you supply a search expression that can't be parsed as a valid regular expression, `REALbasic` will throw an exception when you send the `Search` or `Replace` message. This will be an instance of the `RegexException` class, and you can learn more about the details of the problem through its `Message` property. Of course, you can also get a `NilObjectException` by trying to extract property values from a nil `RegexMatch` instance generated by an unsuccessful search; and trying to extract a `SubexpressionString` or a `SubexpressionStart` using an index too high for the number of subexpressions in the search expression will generate an `OutOfBoundsException`.

Expression Yourself

This discussion has not enumerated every aspect of regular expressions, but it has introduced the high points, and you now know enough to get started with `REALbasic`'s implementation of regular expressions. You shouldn't have much difficulty understanding the list of metacharacters in the online help for `Regex`, and you're well prepared, with a little study, a little thought, and a little experimentation, to do much of what can be done with regular expressions.

Here are a couple of warnings. First, regular expressions take practice. The subject is a deep one, and adepts pride themselves on their ability to construct long, powerful, incomprehensible search expressions. Second, don't imagine that any single regular expression can solve every search problem. There's nothing wrong with breaking down a problem into several searches, and often that's the best solution. Neither of these matters has anything to do with `REALbasic` itself! Remember, `REALbasic` simply implements a standard form of regular expression syntax. Now get out there and search some text!

Three Ways to Animate

Continued from page 23

very small (and you're able to update just that part of the frame). As the animation gets larger or more complex, `Canvas` animation will suffer. `SpriteSurfaces` and `Rb3DSpace`s hold up better under more complex jobs, though with enough objects they will both slow down. In the case of a `SpriteSurface`, it's the number of objects that are actually moving or changing shape that counts; with an `Rb3DSpace` it's the total number of visible objects (and their complexity), even if they are unchanging from frame to frame.

Finally, consider system requirements. `Canvas` animation and `SpriteSurfaces` will work on pretty much any system that `RB` runs on. The `Rb3DSpace` requires support from a system library, either `QuickDraw 3D` (on classic Mac OS) or `Quesa` (on any system). These are free, but many users will not have them by default (except perhaps in classic Mac OS) so you might need to ship your application as an installer that includes the appropriate library.

Conclusion

The project associated with this article includes all the code shown here, plus a few extra perks that were omitted for brevity. It lets you change among the three animation methods, do some extra tricks like scaling the view when rendering with `Rb3D`, and measure the maximum attainable speed. This lets you compare the three approaches very directly. Do keep in mind that details (especially performance) will vary greatly depending on the particulars of the task and on your system (e.g. whether you're using `Quesa` or `QuickDraw 3D` for the `Rb3DSpace`).

If you're looking to explore these animation techniques in more depth, I would suggest you generalize the code to support multiple rockets. This will let you see how each approach scales up with an increasing number of moving objects. Or, if you're interested in game development, you might try connecting each of the update methods to the keyboard state and see which one is most pleasant to “fly.”

There are certainly other approaches to animation in `REALbasic`, especially when plug-ins are involved. But this covers the most important ones; with a good understanding of these you'll be well prepared to handle any animation task.

REFERENCES

Quesa: <http://www.quesa.org/>

Beginner's Corner

by Thomas J. Cunningham

EASY AS PIE!



BEGINNER

Introducing REALbasic For those just getting started

This column is targeted for the new computer programmer. I program with REALbasic (RB) as a hobby. I have tried many other Mac programming applications and RB is far and away the most satisfying. It is a perfect development and learning platform for beginners, students, and experienced programmers. Don't let the BASIC in REALbasic fool you: there is plenty of power to write a wide assortment of fun, useful, and professional applications.

I assume that if you have made the commitment to this magazine, that you are a proficient Mac user and that you have at least spent a few hours going over the QuickStart.pdf and word processing tutorials that come with RB.

This column will back up a bit from the tutorial and explain a few of the beginning computer science issues introduced in these guides. It is intended to be a gentle course that will keep you interested in programming. Make no mistake about it: programming, for most, does not come naturally. It is a discipline that takes time. Like trying to learn to play a musical instrument, it takes patience, a bit of dedication, and lots of practice to obtain a certain level of fun and proficiency.

An Overview Of Programming with REALbasic

Programming a computer. What does this mean to you? The purpose of programming is to tell our computer what we want it to do. We want it to do certain tasks for us in a manner that we, as the programmer, think would be useful to a user, either ourselves

or someone else. Our computer does not comprehend our human language. We need something to take our human desires and interpret these in a language that our Mac understands. REALbasic is our interpreter. Programming a computer can be described by two principles: *logic and math*.

Now, it would be cool if we could just tell RB, "would you please have my Mac balance my checkbook for me?" Sorry, RB is awesome, your Mac is the best, but they are not that good. No, in order to have RB help us program our Mac, we need to learn the language that RB understands. The RB language is not French, German, or even English. It is a version of BASIC, an acronym that stands for Beginner's All-purpose Symbolic Instruction Code. REALbasic is just one of perhaps hundreds of different variations of the BASIC language. Please note that the "B" in BASIC stands for beginners, which is good for us, right?

We give our instructions to RB via code. Code can be thought of as a recipe of sorts. Like baking cookies: "mix 1 cup of flour, add sugar, bake, etc." The computer science word for this recipe is *algorithm*. When we tell our program to "run" (command-R), RB passes our instructions to the computer, translating it into the language our Mac understands: machine code.

In the past, most BASIC interpreters used a "one window" coding approach

(see Figure 1). You would type in your code instructions, line after line, and when you were finished, you ran the program. This interpreter would then read in, line-by-line, each instruction, "printing" its answers in another display window. REALbasic does not use this approach, but instead integrates several windows in to what is called an *Integrated Design Environment (IDE)* (see Figure 2).

REALbasic also has a very different code execution approach than most BASIC compilers. Yes, we will be writing code, telling RB what we would like it to do, but not in a singular code window. Instead, RB reads and runs code segments from *Events*. This event-driven method is a very intuitive and powerful way to program, especially for beginners. It does add a level of complexity, since we have to learn what these events are and when they happen, but in the end this is a very good approach to programming. And luckily, most of these events are intuitive to Mac users.

So what is an RB event? Some examples include: a mouse click, a mouse moving, a window opening or closing, or a button pushed, to name just a few. Each event is associated with a particular item in your program. So when this event occurs, then and only then, is your code executed. The specific events occurring with each item are easy to find, since they are listed in the code editor associated with that particular

```
Print "Hello REALbasic Developers!"  
Hello REALbasic Developers!  
█
```

Figure 1: This is what a programming window looked like on an Apple IIe.

Thomas has been enthusiastically learning to program his Mac using REALbasic since July 1999.

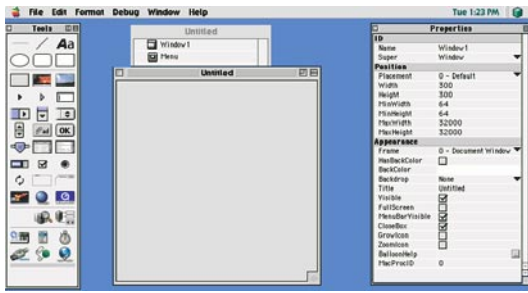


Figure 2: REALbasic's user interface is referred to as the IDE.

control item. Not all of these items have the same events, but many do share common events. Most of these events are self explanatory, but some will take a bit of time to understand. This is part of the learning process of becoming familiar with RB, so be patient.

Another excellent difference in RB from line-by-line interpreters is the ability to design what your user's screen looks like in your program. This is referred to as your *User Interface* (UI). Most all of your programs will have a main window. Much like the words you are now reading need to be printed on a once blank sheet of paper, your program will need to place the items you want to show in a main window. The items that you can add include text, buttons, edit boxes, and pictures, to name a few. You have seen these control items many times in all of the Mac programs you currently use. They help to define the Mac experience.

So where do these control items that we place in a window come from? What are these control items? Again, think of this magazine as a real world example. REALbasic comes with a plethora of built-in "tools" for you to use in creating your programs. They are conveniently listed in a Tools Palette within the IDE. By "built-in," I mean they each have certain jobs or tasks that they perform automatically for us. They have already been programmed by RB and are ready to use. These tools are in computer science parlance referred to as *Classes*. Think of these tools as the raw materials, or building blocks, you will use to construct an application.

You use the tools (classes) just like a drawing program. You select one from the tools palette and drag it into your main design window. This window name is, by default, "Window1." You can drag multiple copies of the same class in to your main application window. Once you place

a tool in Window1, you can position it where you would like it to be, make it bigger or smaller, or otherwise change certain characteristics of the tool. These characteristics are called *properties*. A class has certain properties associated with it. Most of these properties can be changed, via code, while your application is operating (during "runtime"). This is how your program appears to interact with its user in a (hopefully!)

intelligent manner: by changing certain properties as various events occur.

For example, when a user places his or her mouse cursor over a certain word, it turns red; when it moves away, the word returns to black.



This is how events and code execution interact in RB to make your application more engaging to the user. Also note that RB takes care of a lot of the gory details here, like what defines the area of the word and the redrawing of the text.

Let me describe an example. Say we want to create an application that duplicates an advertisement page in a magazine. Lets assume the page has some words, or text with the product's corporate byline, and a picture in the background of the product, BackYard Jets. Launch RB and open *Window1* (the default name of the main window) from the project window by double clicking on its name. Once open, we drag a StaticText class (represented by the "Aa" icon) from our tools palette (which is on the left side of the screen). This class tool building block will display our text. We readjust its position within the window, resize its width and height, and enable its MultiLine property (using the properties window located on the right side of the screen). Enabling the MultiLine property will allow the text to be shown on more than one line. We type into the Text

Appearance area the "text" property we want displayed.

Pretend the jet company has given you a picture they want in the ad. All you have to do is drag and drop the picture file in to our project window, from your desktop, to have access to the picture in your program. We set our Window1 BackDrop property to the name of the picture from the drop down list. The picture will now be displayed when we run our project (Command - R).

We run this little project and see that it looks just like we want. We quit, returning to our Integrated Development Environment (IDE) to add one feature that a print magazine can not offer. We drag a pushbutton class tool (the "OK") from the tools menu to Window1 and change its Caption property to read "Push me for more info." We adjust the button's size properties so we can see this caption and place it where we want. Now, while still in the IDE, double-click the pushbutton, which by default is named "pushbutton1". This reveals pushbutton1's code editor.

In the *action* event of this pushbutton's code editor ("action" means when the user presses this button), we write code that will be executed when the user pushes this button. I won't bother with the actual code here, but it will launch our user's web browser, taking them to this advertiser's web page. We now have a dynamic advertisement with easy access for our user to this product. Very cool, and it took little time and effort, with RB's help of course! (See Figure 3).

I hope this brief overview of the RB development experience has helped you! Next issue we will go in to more depth with specific details of the coding process. 📦



Figure 3: Our fictitious advertisement program.

Using the Declare Statement It's not as difficult as you think

The Mac OS has always been nice to look at, but many folks don't know about the power under the hood. REALbasic sometimes suffers from the same misconceptions, but like the Mac OS, the power is there. In this column I'm going to talk about one way to get "under the hood" in REALbasic: using Declare statements.

A Declare statement allows you to call system routines, enabling you to do things for which REALbasic doesn't have built-in support. For example, a Declare statement can call Process Manager routines from the Mac Toolbox to get information about currently running applications. This information is not available through any native REALbasic classes or methods.

Instead of repeating the information found in the REALbasic documentation regarding Declare statements, I'll assume throughout this article that you're familiar with it. The discussion will also be limited to accessing Mac Toolbox routines; using Declare statements to access system routines under Windows is beyond the scope of this article.

Building a Declare statement is the first step in calling a Toolbox routine. To do this you first need to know which system routine you want to call and how it's called. You can find documentation for the various Toolbox routines on Apple's web site at <http://developer.apple.com/techpubs/>. Raw C function declarations can be found in the "CIncludes" folder of the Universal Interfaces. The Universal Interfaces can be found at <http://developer.apple.com/sdk/index.html>.

Thomas Reed has been programming as a hobbyist for more than 20 years, and fell in love with the Mac in 1984.

Once you have decided which routine to call, you need to translate the C function declaration into a REALbasic Declare statement. Let's start with a fairly simple example: suppose you want to call the GetSysBeepVolume function. Its C function declaration is:

```
OSErr GetSysBeepVolume(long *level)
FOURWORDINLINE(0x203C, 0x0224, 0x0018,
0xA800);
```

The corresponding REALbasic Declare statement for a 68K or PPC build will be:

```
Declare Function GetSysBeepVolume Lib
"InterfaceLib" (byref level as integer)
as Short Inline68K("203C02240018A800")
```

To understand how I arrived at this result, let's start at the beginning of the C declaration, which in this case is the keyword OSErr. This is the return value, and an OSErr is just a short integer. Thus, it needs to be defined as a Function in the Declare statement. In C, a return value of void means the routine doesn't return anything. In that case, it would instead be defined as a subroutine in the Declare statement, using the keyword sub.

Next, we need to specify in which library the routine is found. For almost all Toolbox routines for PPC or 68K builds, this will be InterfaceLib; while for Carbon builds it will be CarbonLib. Note that you can use preprocessor directives to include the appropriate Declare statement for your build. Here's an example that will work properly for all Mac OS builds (68K, PPC or Carbon):

```
#if TargetCarbon
Declare Function GetSysBeepVolume Lib
"CarbonLib" (byref level as integer) as
Short
#else
```

```
Declare Function GetSysBeepVolume Lib
"InterfaceLib" (byref level as integer)
as Short Inline68K("203C02240018A800")
#endif
```

REALbasic 4.5 will introduce the ability to use an OS-specific string constant to specify the library, trimming the above five lines of code down to one.

Next, the parameters are defined. Since GetSysBeepVolume takes a pointer to a long integer, you can handle this one of two ways. The easy way is to make REALbasic pass the parameter by reference using the byref keyword and to define the parameter as an integer. The hard way is to define that parameter as being of type Ptr. We will discuss how to determine parameter types and what kind of variables to pass as parameters in a moment.

If the routine has been defined as a function, the next thing you need to do is define a return type. In the example of GetSysBeepVolume, the return type is OSErr. If you look in the Universal Interfaces or the documentation on Apple's web site, you will learn that OSErr is a fancy name for a short, which is a 2-

RB Declare type	RB variable type
OSType	String
Short	Integer
PString	String
CString	String
WindowPtr	Window
Ptr	MemoryBlock

Table 2: Some data types used with Declares can't be used in variable declarations. This shows how these "Declare-only" data types map to regular REALbasic variable types.

RB Declare type	Toolbox/C type
Integer	long, int, SInt32, UInt32, Size, Handle, Ptr
Short	short, SInt16, UInt16
Single	float
Double	double
Boolean	boolean
Color	same as Integer
WindowPtr [†]	WindowPtr, WindowRef
PString	ConstStr255Param, Str255, Str63
CString [†]	char *
OSType [†]	OSType, ResType, DescType
Ptr	Handle, Ptr, any pointer type
[†] only to be used in parameters, not as return types	

Table 1: This table shows which data type to use in your Declares to represent some common Toolbox/C data types.

byte integer. So, we define the routine as returning a Short.

This is sufficient, unless you also want to support 68K code, in which case you need to include the Inline68K value. (As of version 4.0, RB no longer supports 68K builds.)

Look back at the Declare statement for the GetSysBeepVolume routine and you'll find a long hexadecimal number in the Inline68K parameter. This number is derived by concatenating the numbers found in the FOURWORDINLINE parameter of the C function definition. (Note that the C definitions of other routines may have a ONEWORDINLINE parameter, TWOWORDINLINE parameter, etc.)

Building Declare statements this way may seem difficult, but fortunately, you can get some help from an excellent program called TBFinder, available at <http://homepage.mac.com/everyday/code/downloads/TBFinder.sit>. TBFinder will look up the C function definition and build the Declare statement for you.

Now that you know how to build a Declare statement, you need to know how to use it: how to call it, what data types you can use (yes, we're not done with that yet), and how to interpret complex data structures. Calling a Toolbox routine is simple, as long as you remember a few simple rules. First, note that you must enclose the parameters in parentheses when calling functions while you should omit the parentheses for subroutines. Otherwise, the REALbasic compiler will complain. Second, don't try to use the result of a function call directly as a parameter to the Declared routine. You will first need to store the value somewhere. Properties of REALbasic objects should be considered function calls.

Let's return to the issue of how to translate Toolbox and C data types into REALbasic data types. Table 1 lists the types available for use with Declares in REALbasic and shows how they map to some common Toolbox/C data types. You should note that some of these data types are interchangeable. For example, Integer, Color, OSType, and Ptr are all 4-byte values, so technically you could use OSType where a Toolbox routine uses UInt32. Your choice depends on how you will use the value.

For the most part, the types you use in a Declare statement are the same as regular REALbasic variable types. However, there are some differences. For example, what kind of variable do you pass to a routine that expects an OSType? OSType isn't a REALbasic variable type! Table 2 shows how some of the Declare types map to REALbasic variable types.

Looking at Table 2, we see that if you want to call a Toolbox routine that takes an OSType as a parameter, you can just pass in a REALbasic string. It will be translated into a 4-character OSType before the Toolbox routine is called.

More complex data types offer other challenges. You may notice that there are a LOT of Toolbox data types missing from Table 1! You'll want to use REALbasic's MemoryBlock class for most of these. For example, suppose you want to create an FSSpec, which is the OS equivalent of the FolderItem class in REALbasic. You'll need to create storage in memory for the FSSpec, which requires that you know how large it is. An FSSpec's structure definition in C looks like this:

```
struct FSSpec {
    short vRefNum;
```

```
    long parID;
    Str63 name;
};
```

If you know your Toolbox data types, you'll know this is 70 bytes. However, if you're not so sure, the size can be found in the assembly language header files that are part of the Universal Interfaces. The structure definition in these headers includes offsets for each field and the total size of the structure. Consider the example of an FSSpec:

```
FSSpec RECORD 0
vRefNum ds.w 1    ; offset: $0 (0)
parID ds.l 1      ; offset: $2 (2)
name ds StrFileName ; offset: $6 (6)
sizeof EQU *      ; size: $46 (70)
ENDR
```

Because it's 70 bytes long, you would use a MemoryBlock 70 bytes in size to pass an FSSpec to Toolbox routines. Here's an example that creates an FSSpec from a FolderItem:

```
dim f as FolderItem
dim fsspec as MemoryBlock
dim err as integer
dim fname as string
Declare Function FSSpec Lib "CarbonLib"
    (vRefNum as Integer, dirID as Integer,
    fileName as PString, spec as Ptr) as
    Integer
```

Continued on page 48

Unsigned longs

Since REALbasic doesn't have an unsigned long integer type, it may be difficult to get such a value from an OS routine. When you put a large enough unsigned long in a REALbasic Integer, it will inaccurately be represented as a negative number.

The only way to represent a full unsigned long integer is to store it in a Double, which is an 8-byte value. However, a little number crunching is needed to get the real unsigned number from the signed REALbasic Integer:

```
dim i as integer
dim d as double
i = DoSomethingToGetAnUnsignedInteger()
if i < 0 then
    d = BitwiseAnd(i, &hFFFFFFF)
    d = d + 2147483648.0
else
    d = i
end if
```

Ask the Experts

by Seth Willits
with Thomas Reed & Sean Beach

In a program I'm writing, I'd like to include a menu where the users can add their own AppleScripts and execute them. How can I implement this?

A fully implemented Scripts menu is seemingly involved, but easily implemented through the use of available third party plugins and modules. The two main parts of a Scripts menu are handling the dynamic menu items, and executing the script as it is selected from the menu. To add a little flavor, we can also add icons to the menu title as well as the individual menu items.

Dynamic Menu Items

This portion of the menu is by far the most involved step in creating a working Scripts menu. The idea is to create a menu that can hold a variable number of menu items pointing to scripts. The menu itself will have one menu item called `ScriptItem` which will have an index value of 0. With this menu instated we can create two properties in the Application class; `Scripts(-1)` as folderitem and `ScriptItems` as integer. Next we can use the code in the listing to dynamically create the menuitems in the `EnableMenuItems` event. The code depends on folderitems in the `Scripts` array that point to the scripts themselves. You can choose how to handle adding those folderitems on your own.

```
dim si as MenuItem
dim i as integer

// Scripts Menu
ScriptsAddScript.enable
```

Seth Willits, the President and Head Developer of Freak Software, is slowly working his way to becoming a distinguished programmer for the Macintosh. Thomas and Sean are columnists for REALbasic Developer. Send your "Ask the Experts" questions to help@rbdeveloper.com. Due to the volume of email, personal responses are regretfully not possible.

```
// Hide the index
ScriptItem(0).visible = false

// Delete all items
If ScriptItems > 0 then
    for i = ScriptItems downto 1
        ScriptItem(i).close
    next
end if

// Add items
scriptItems = 0
if UBound(scripts) > -1 then
    for i = 0 to UBound(Scripts)
        si = New ScriptItem
        si.text = Scripts(i).name
        si.enabled = true
        ScriptItems = ScriptItems + 1
    next
end if
```

Executing the Scripts

Executing the scripts is a simple matter. We will use Doug Holton's AppleScript plugin (http://www.geocities.com/d_h_1_h/) to do this for us. By examining the code in the `EnableMenuItems` event you will see that the lower bound of the `ScriptItem` menu item is not used. This means we'll have to subtract one from the menuitem's index property (in the `ScriptItem` menu handler) to get the correct `Scripts()` folderitem.

Once we have the folderitem pointing to the script we can determine if it is compiled or not by its `MacType`. A `MacType` of "TEXT" means the script has not been compiled while a `MacType` of "osas" means that it has. An uncompiled script keeps the script code in the datafork making it easy to simply open the file in a text input stream, read everything, and execute it using the `RunScript` method of the AppleScript plugin. If the AppleScript has been compiled we need to extract the data from resource 128 of type "scpt" and pass that as the parameter to the `RunCompiledScript` method.

I have a few fields in my program that should only contain numbers. Unfortunately REALbasic doesn't offer a simple checkbox like for password fields so I have to write my own. How should I go about this?

Number-Only fields simply filter out keystrokes from keys other than numbers, arrows, and the delete key. In REALbasic we can do this filtering inside of the `KeyDown` event of an editfield by returning true if the key is one we do NOT want to appear in the field. As you can see in the code listing I chose to use the `InStr` function instead of using a series of conditions on the ASCII value for simplicity (using `InStr` will not noticeably slow down the speed of typing in the field unless it contains hundreds of characters).

In this implementation, the field allows the numbers 0-9, the delete, return, enter, tab, and arrow keys to pass through. Letting the tab key pass through prevents the interruption of tabbing from field to field. Just make sure that you don't have the `AllowTabs` property of the field enabled, otherwise the tab will appear in the field!

```
Function KeyDown(Key As String) As Boolean
    if InStr("0123456789" + chr(8) + chr(13) + chr(9) + chr(3), key) > 0 then
        return false
    elseif asc(key) > 27 and asc(key) < 32 then
        return false
    else
        return true
    end if
End Function
```

My son is into racing go-karts and I'd like to write a program to keep track of all of his lap times and other race specific information. Since each race or practice session can have a different number of laps the difficult part of writing the program is how to handle those lap times. The only way I can think

of is to have a separate folder for every race or practice session which contains separate text files for lap times and race info. While this would work, it would be very messy. What is a more elegant solution?

This is a problem that arises and is solved everyday using a relational database. If you have not yet learned to use REALbasic's built-in database tools I highly recommend that you do. Not only will it enable you to write an elegant solution to this problem, but it will help you with many other similar problems in the future.

To learn about databases in REALbasic, I would recommend that you read the REAL Database guide written by Seth Willits of Freak Software. The instructions below assume that you know how to create a database and understand how to use it.

The general concept of the solution to this problem is to have two tables in a database. One table (RaceInfo) will hold the race specific information (track, date, etc.) and the other (LapTimes) will only hold lap times. Each record in the RaceInfo table will have an ID number which is specific to that race event. Each record of the LapTimes table will contain an ID number which is the same as the ID number of the race that the lap time was recorded on.

Take a look at Figure 1. By matching the RaceID from a lap time to a record in the RaceInfo table with the same RaceID or vice versa we can determine which lap times

belong to which race event. For example, the longest lap time is a 01:49:58 which has a RaceID of 001. In the RaceInfo table we search for the record with the RaceID of 001 and that's the race that the lap time belongs to. The slowest lap time occurred at the California track on 01/02/02.

The table schemas for the database are as follows:

```
RaceInfo (
    RaceID integer,
    Track varchar,
    Date varchar,
    Primary Key(RaceID)
)
```

```
LapTimes (
    ID integer,
    RaceID integer,
    LapTime varchar,
    Primary Key(ID)
)
```

Note that the RaceID field for the RaceInfo table is the primary key, but in the LapTimes table it is not. Every record in the RaceInfo table will have a unique RaceID but in the LapTimes table several records will have the same RaceID.

Adding records to the database can be handled anyway you'd like. Just keep in mind that when you add lap times to the LapTimes table they must have the same RaceID as the corresponding record in the RaceInfo table.

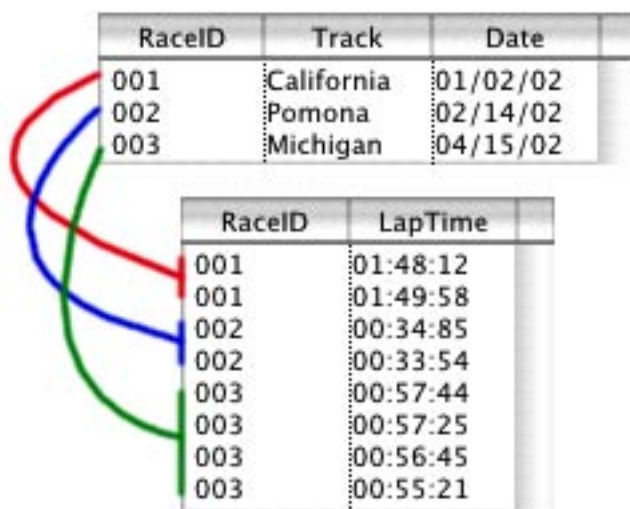


Figure 1: This is a graphical representation of the two tables that will be in your database. Top: RaceInfo. Bottom: LapTimes.

To extract all of the information from the database *at one time* we'll need to use an instance of a custom class for each race event. You can define the class as follows:

```
<Class>
    <Name>Event</Name>
    <Property>LapTimes(-1) as string</Property>
    <Property>Track as integer</Property>
    <Property>RaceDate as String</Property>
</Class>
```

The code to read the data is as follows.

```
dim e as Event
dim curs, lapcurs as DatabaseCursor
dim rid as integer

// Select all race dates in the database
curs = Database.SQLSelect("Select * from RaceInfo")

// For every race event
do
    e = New Event
    e.Track = curs.Field("Track").getString
    e.RaceDate = curs.Field("Date").getString

    // Get the RaceID of the event
    rid = curs.Field("RaceID").integerValue
    // Select every lap time with the RaceID of this event
    lapcurs = Database.SQLSelect("Select LapTime from LapTimes where RaceID = " + str(rid))

    do
        // Add the lap time to the laptimes array of the event's Event class instance
        e.laptimes.append lapcurs.field("LapTime").getString
        lapcurs.moveToNext
    loop until lapcurs.eof
    lapcurs.close

    // Add Event class instance to the global Events array
    Events.append e
    curs.moveToNext
loop until curs.eof

curs.close
lapcurs.close
```

REFERENCES

REAL Database Guide:

<http://www.freaksw.com/rb-papers.html>

Race Example Project: <http://www.freaksw.com>



Linked Lists

Mind your stacks and queues

The term “algorithm” can mean instructions for carrying out some typically useful operation, but it can also mean a typically useful data structure and the methods for working with it. A linked list is often one of the first data structures treated in books on algorithms, and it makes a good subject for this first Algorithms column.

A linked list is a simple data structure that’s useful chiefly because it’s dynamic. This means you don’t have to set aside storage for the data beforehand; instead, the linked list grows and shrinks as the program runs, as when data is added or removed. In REALbasic this fact is not very compelling, though, because arrays are dynamic already! You can add data with Insert or Append, and remove it with Remove. Nevertheless, you can also very easily implement a traditional linked list in REALbasic, and it’s instructional to see how. The principles involved are applicable to more complicated and powerful data structures, such as binary trees, that we can examine in future columns.

The chief characteristic of a linked list is that from each piece of data it is possible to reach the next piece of data. The pieces of data are connected by pointers; to reach the next piece of data, follow this piece’s pointer (see Figure 1).

To see why this is dynamic, consider how you’d insert a new piece of data, “hi,” after “hey”: just disconnect the pointer from “hey” to “ho”, and make it run to “hi” instead, and connect the pointer from “hi” to “ho” (see Figure 2).

Matt Neuburg is the author of *REALbasic: The Definitive Guide*, and a member of the editorial board of *REALbasic Developer*.

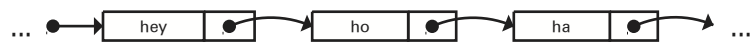


Figure 1: A linked list

Let’s implement this in REALbasic. In REALbasic, object references are pointers. (See pp. 94-104 of my book for much more about that.) So the problem is solved if each piece of data is an object, and consists of two things: the value to be stored, and a reference to the next piece of data. An object consisting of two things is merely an instance of some class that has two properties. Let’s call the class `ListItem`, and let its two properties be called `Value`, a variant (for generality), and `NextItem`, another `ListItem`. It’s easy to see that, with appropriate values for the properties of each instance, we end up with a linked list (see figure 3).

We can now implement `Insert`, handing a `ListItem` the `Value` we want attached to a new `ListItem` that will follow it.

ListItem.Insert:

```
Sub Insert(v As variant)
    dim newItem as ListItem
    newItem = new ListItem
    newItem.value = v
    newItem.nextItem = self.nextItem // (a)
    self.nextItem = newItem // (b)
End Sub
```

The last two lines, (a) and (b), perform the disconnection and reconnection of pointers that we talked about earlier. After (a), for an instant, both the new item’s `NextItem` and the target’s `NextItem` point at the same thing; then in (b) we repoint the target’s `NextItem` at the new item. The order of operations is important! If (b) precedes (a), then after (b), the target’s `NextItem` no longer points at what was the next item in the list — in fact, nothing at all points to it, the item is lost, and the list is ruined.

In REALbasic, an object reference that has never been set explicitly is `nil`. This fact makes it easy to obtain the last item in the list, by following the chain of pointers item by item until we come to one whose `NextItem` is `nil`. This approach works no matter what item of the list we target initially. Following the whole chain of pointers in this way is called traversing the list, and can be neatly expressed in REALbasic.

ListItem.LastItem:

```
Function LastItem() As ListItem
    dim x as ListItem
    x = self
```

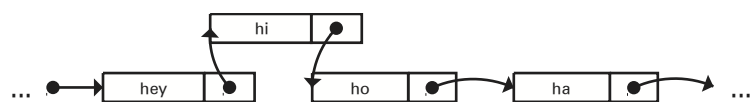


Figure 2: Insertion into a linked list

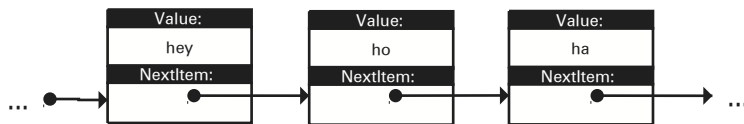


Figure 3: *Linked list made of REALbasic objects*

```
while x.nextItem <> nil
  x = x.nextItem
wend
return x
End Function
```

As an exercise, implement Append. This accepts a value and adds it at the end of the list, no matter what item of the list we target initially. Hint: combine LastItem and Insert.

Finally, we implement RemoveNext. When sent to an item of the list, it removes the next item from the list. We first check for a NextItem value of nil, since the target might be the last item already; in that case, obviously, we do nothing. Otherwise, we just point the target's NextItem at the next item's NextItem (see figure 4). The instant we do this, nothing points at what used to be the next item, whose existence is promptly terminated by REALbasic's delightful memory management system.

ListItem.RemoveNext:

```
Sub RemoveNext
  if self.nextItem <> nil then
    self.nextItem = self.nextItem.nextItem
  end
End Sub
```

Stacks

A stack is a data structure where access to the pieces of data works like a pile of plates: you can add or remove an item only at the top of the pile. Items can thus be retrieved only in the reverse of the order in which they were added; for this reason, a stack is often described as LIFO (last in, first out). The two fundamental operations on a stack are called Push (add a new item at the top of the stack) and Pop (remove the topmost item from the stack and return its value).

Under the hood, stacks are fundamental to many computer operations, such as calling subroutines. They also come in handy during certain kinds of parsing operations, and wherever backtracking is involved. For example, a stack would be a good way to implement an Undo list. In

the case of an EditField, every time the EditField changes, you could Push its current state (its text and style) onto the stack; to Undo one level, just Pop the stack to retrieve the previous state.

A stack is easily implemented as a linked list. We might, indeed, have a class Stack that is a subclass of ListItem. Suppose we maintain somewhere a property or variable called TheStack, which is a Stack. TheStack represents the entire stack; Push and Pop are sent only to it. But in reality it is the start of a linked list, and points to the topmost item of the actual stack.

Stack.Push:

```
Sub Push(v as variant)
  self.insert v
End Sub
```

Stack.Pop:

```
Function Pop() as variant
  dim v as variant
  if self.nextitem <> nil then
    v = self.nextitem.value // (a)
    self.deleteNext // (b)
    return v
  end
End Function
```

Again, order of operations is important. We use a local variable to capture the value to be returned, in (a), before deleting the item that contained it, in (b). If (b) were performed before (a), the value would be destroyed before it could be returned; there would be nothing to return.

Since, as mentioned earlier, a REALbasic array is already a dynamic data structure, Stack could just as well be implemented by wrapping a variant array. Let's start over. Stack now has no superclass, and has a

single property, MyArray, a variant array initially dimmed to size -1. We'll treat the end of the array as the top of the stack. So Push just means Append.

Stack.Push:

```
Sub Push(v as variant)
  MyArray.append v
End Sub
```


As an exercise, implement Pop.

To the programmer, you'll observe, it doesn't matter whether Stack is implemented as a linked list or an array. This is in keeping with the principle of abstraction: lower-level manipulation of data is hidden inside a controller object. As much as possible, you should try to program this way. For one thing, other objects in your program have no need to concern themselves with the underlying implementation details; they just say Push or Pop and everything simply works. This greatly reduces the chances for error. Also, you could change your mind about the underlying implementation, changing it from a linked list to an array for example, and all the calls to Push and Pop throughout the program would go right on working.

Queues

A queue is a data structure where access to the pieces of data works like patrons lined up to enter a movie theater: the first patron to get on line will be the first patron to enter the theatre, and so on. A queue is often described as FIFO (first in, first out). The fundamental operations on a queue are called Queue (append an item to the tail of the queue) and Dequeue (retrieve an item from the front of the queue).

Queues often arise in threaded situations, where pending tasks can pile up while a previous task is being performed. For example, in an email client program, we might allow the user to press the Send button at any time, but we won't actually perform any new sending until we finish with whatever message we may be sending at the moment.

Implementation of Queue and Dequeue are left as an exercise to the reader; do it both ways, with a linked list and with an array. 

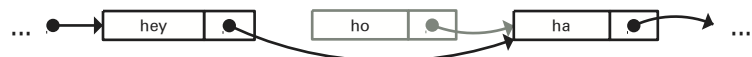


Figure 4: *Deletion from a linked list*



Favor Composition Over Inheritance

Designing a validating editfield

The Problem

You are writing a Quicken-killer in REALbasic. You need to do a lot of input validation — dates, currency input, names, etc. You think, “Ah, RB has made it easy for me — all I need to do is slap some validation code in the LostFocus event handlers of the editfields.” But writing a Quicken-killer is a big job, and cut-and-paste is especially efficient at propagating errors. It occurs to you that you can subclass Editfield. So you define DateEditField, NumberEditField, CurrencyEditField, NonEmptyEditField, TextLengthAtMostNEditfield, and other subclasses. But as the subclasses begin piling up — for instance, you need an EditField subclass that accepts valid currency input and must be nonempty — the project begins to crumble under its own weight. Then you decide that some editfields must be able to change the validation method at runtime. Next, you need to write an editable listbox requiring validation. And then Real Software introduces a Text class....

The Solution

The common theme of this problem is the need to validate text input. This is a task that can be handed off to an object whose sole purpose is to validate text. In other words, we can create validating editfields by *composing* an editfield and a validator object; the editfield *delegates* the task of validating input to a validator object.

We'll still need to use inheritance to write an EditField subclass which can use validators. But it will be just one subclass requiring simple coding and offering a consistent interface; it's very easy to intro-

duce small but annoying variations in the interfaces of several similar subclasses.

There are two actors here, TextValidator and ValidatingControl. Our next task is to specify their roles; that is, we need to declare their interfaces.

The main task I want to ask of a TextValidator is to tell me whether a string I pass it is valid. It would also be nice to be able to ask a TextValidator for an error message; if users enter invalid data, they'll want feedback more informative than a beep. Any class implementing TextValidator can supply a default error message (“Invalid Date”), but it would probably be nice to be able to set an error message more appropriate to the context.

```
Class Interface TextValidator
Method
IsValid(text as String) as Boolean
SetErrorMessage(msg as String)
GetErrorMessage() as String
```

It's easy to see that probably every class implementing TextValidator will have a property holding the error message, and the implementations of SetErrorMessage and GetErrorMessage will be the same. So we'll define a class (AbstractValidator, in the associated project) implementing basic functionality and let our TextValidators inherit from it. For special cases we can choose not to inherit from AbstractValidator; this is an advantage of factoring the validators into class interface + abstract class + concrete classes.

We'll use the same idea for the editfield subclass; that is, we'll define a class interface ValidatingControl which a subclass ValidatingEditField will implement. Using a class interface here is mostly for reasons of clarity and convenience. Other controls,

like Listbox, can also implement this interface; having the interface explicitly defined helps to document the design.

Also, a window can maintain references to ValidatingControl objects it contains, instead of needing to loop through controls looking for ValidatingControls to test.

```
ClassInterface ValidatingControl
Methods
AddValidator(validator as TextValidator)
RemoveValidator(validator as TextValidator)
IsValid() as Boolean
ValidationErrorMessage() as String
```

The associated project contains an implementation of a ValidatingEditField and some TextValidators.

The Buzzwords

We've illustrated two principles of object-oriented design:

Favor object composition over class inheritance; and *Program to an interface, not an implementation.*

By looking first to composition, our design is built from objects focused on clearly defined tasks. But composition only works if we are careful to minimize the extent to which these objects depend on each other. The use of class interfaces allows us to specify how these objects can communicate; this allows us to make internal changes to one class without affecting others.

Further Reading

Design Patterns: Elements of Reusable Object-Oriented Software contains an excellent discussion of inheritance v. composition starting on page 18. ■

Charles Yeomans is a software developer in Lexington, Kentucky.



REALbasic Windows Support

What works and what doesn't

About Windows Versions

In this column we will discuss REALbasic's capability to create applications for the Win32 platform. Windows 95, 98, ME, NT4, 2000, and XP all implement a growing subset of the Win32 API (Application Programming Interface). But these different implementations of Win32 are not equal, so you'll find functions in Windows 2000 which are in NT4 but not in the newer 98 version of Windows.

We'll mostly discuss functions which are available in Windows 95 and newer. Windows NT4 misses some of these functions but Windows 2000 (which is a renamed NT5) has all the functions from 95 and 98.

If readers request it, we can even discuss using a Win32 subsystem on Linux (named Wine) or Windows 3.11 (named Win32s). If you use one of these versions of Windows, send me an e-mail at cschmitz@rbdeveloper.com and let me know.

REALbasic's Mac-Only Features

Using REALbasic version 2.0 or later, you can compile applications for Win32, but obviously you need the Professional version if your applications need to run longer than five minutes.

People often ask for a list of functions and objects in REALbasic which are Mac-only and won't compile for Windows. Before getting to this, however, we need to learn how to handle Mac-only features.

#if target

Using the keyword `#if`, you can tell REALbasic to use conditional compiling. For example:

```
#If TargetWin32 then
msgbox "This application runs on Windows!"
#else
```

```
msgbox "This application runs on Mac OS!"
#endif
```

As you can see, this looks like a normal if-then block but with the number sign ("`#`") in front of the keywords. Everything which is not for the current platform will be ignored by the compiler, which can reduce application disk space (for example, an icon for Windows is not compiled into the Mac version). You can even build larger blocks with nested `#ifs` like this one:

```
#If TargetWin32 then
msgbox "This application runs on Windows!"
#else
#If TargetCarbon then
msgbox "This application runs on Mac OS
Carbon!"
#else
#If targetPPC then
msgbox "This application runs on Mac OS Classic
on PPC!"
#else
#If target68k then
msgbox "This application runs on Mac OS Classic
on 68k!"
#else
msgBox "Mac OS on something other than 68k or
PPC? Can't be!"
#endif
#endif
#endif
#endif
```

Sadly there is no `#elseif` and REALbasic doesn't indent the lines like in normal if-

Christian Schmitz has written several articles for the German magazine Macwelt and has made many cross-platform applications using REALbasic.

then blocks. Table 1 describes the target constants.

AppleEvents and AppleScript

This first thing to note is that the AppleEvent object on Windows is completely non-functional. REALbasic will compile it but it will just waste memory and it does nothing for you. This is a good case where the `#if TargetMacOS` directive can be used to prevent the code from being compiled on Windows.

You can add compiled AppleScripts into your project and call them from inside your applications using their name like in the example below... but use `#if TargetMacOS` to hide this from the Win32 compiler.

```
#if TargetMacOS
dim result as string
result = MyAppleScript("some value as
parameter")
#endif
```

Resource Forks

All functions to open a resourcefork will refuse to work. Depending on your REALbasic version, they will crash, return nil or won't even compile, so you should again use a `#if TargetMacOS` directive.

Type and Creator codes

As a means of being user friendly, Macintosh applications will sometimes find files by their type and creator code combination. For example, the Finder finds Sherlock in Mac OS Classic by its creator code. However, creator and type codes are not supported on Windows and any function which returns a code will return an empty string. This may force you to rewrite code to use a file extension to find the files.

Continued on page 48

Welcome

Three-dimensional graphics for everyone

Before you can raise your hand to knock, the wooden door creaks open. The ancient one nods. "I've been expecting you." The dark chamber is filled with dozens of geometric objects: spheres, cubes, pyramids; all spinning and changing color. "The road to real-time 3D can have a treacherous topology, my young apprentice," he warns. "But I have faith in you."

Welcome to the *Topographic Apprentice*. The goal of this column is to provide a painless introduction to real-time 3D programming for games and other applications. I (otherwise known as "the ancient one") will make the assumption that you are a REALbasic beginner and have never worked with real-time 3D before.

A Bit of History

When I started to use REALbasic to build my spacecraft simulator, *A-OK! The Wings of Mercury*, I had no idea how I would handle the requirement for real-time 3D graphics. After trying an as-yet-unreleased plugin and struggling to work directly with the OpenGL libraries, I discovered Rb3D, a plugin that was written by REAL Software developer Joe Strout.

While Rb3D had (and still has) limitations, it was the fastest and easiest way to create real-time 3D applications with REALbasic. A small but active user community rallied around Rb3D and REAL Software decided to make it an integrated feature with the release of REALbasic 3.5.

Joe Nastasi is the developer of a spacecraft simulator, A-OK! The Wings of Mercury, created with REALbasic. Joe, who has been a programmer since 1977, has been a full-time REALbasic consultant since version 1.0.

Rb3D Architecture

Rb3D is not a stand-alone 3D engine, but a programming interface (API) that runs on top of several 3D technologies. Under Rb3D is Quesa, which is an open-source version of Apple's now-defunct 3D engine, QuickDraw 3D (QD3D). Under Quesa is OpenGL, a low-level 3D engine. Rb3D objects, methods, and properties are mapped to equivalent Quesa objects. Quesa handles a lot of the organization, math, and other high-level issues associated with real-time 3D and then calls the appropriate routines to actually draw the scene. Quesa can draw using a variety of rendering engines, but the most common one is based on OpenGL. Unlike Quesa, QD3D handles both the high-level functions and low-level drawing routines (drawing is via RAVE), so nothing else is required (see Figure 1).

In order to use Rb3D, you will need one of two sets of software libraries. Even though Apple has decided not to support QD3D, it is actually the best option for Mac OS Classic. With QD3D nothing else is required. QD3D comes with QuickTime and can be installed by doing a Full or Custom QuickTime 5 install.

Unfortunately, QD3D does not work on OS X and does not provide support for hardware accelerator boards on Windows. Quesa is a call-by-call replacement for QD3D that runs on Mac OS and Windows (and other platforms as well). It works, but has not been optimized yet, so it is slower than QD3D. However, it is the future, so drop a line to the Quesa folks and let them know that it is important to you.

Because Quesa doesn't handle drawing, you need the latest version of OpenGL which is available from Apple. Note that at the time of this writing, there is a serious bug in OS X concerning OpenGL and threads. If

you use Rb3D within a REALbasic thread, it causes a crash. This has been traced to the OpenGL/OS X combination.

The good news is that, since both Quesa and OpenGL are open source, they are unlikely to be orphaned. Additionally, even if a different rendering engine (e.g. Microsoft's Direct3D) is needed, Quesa can make use of that with no changes required to either REALbasic or your code.

Rb3D Components

Rb3D contains five classes and one control. If you are familiar with REALbasic, you'll feel right at home with the syntax.

Right out of the box, Rb3D does not have the large feature set of commercial 3D engines. However, with additional REALbasic programming, more deluxe features like a particle generation system (great for smoke and explosions) can be added. In fact, there will be some feature articles right here in *REALbasic Developer* over the coming year that will discuss the advanced techniques you'll need to turbocharge Rb3D!

Rb3DSpace

The basic control of Rb3D is called Rb3DSpace. It exists on the REALbasic Controls Palette (called the Tools Palette prior to Rb 4.5) and is dragged on to the window of your choice, as you would the canvas control. An Rb3DSpace functions as a window that your 3D world will be drawn into and also provides basic camera, light, and environment controls.

Initial Properties

All of Rb3DSpace's Initial Properties can be set in the IDE or under program control. *Note: properties marked with an asterisk (*) are new to RB 4.5.*

Hither and Yon - These properties define how close and how far away an object can be from the "camera" inside Rb3DSpace and still be visible. You can save lots of processing time by making this range as small as possible for your application. More importantly, Hither and Yon affect rendering quality; if the yon/hither ratio is too large, you get unwanted visual effects.

FieldOfView - This property defines the "camera" lens. A lower value will narrow the viewable field, acting like a telephoto lens. Conversely, a larger value will create a larger viewable field, simulating a wide-angle lens.

SkyColor - This property allows you to set the background color for this particular Rb3DSpace. This is sufficient for simple games and object editor applications.

AmbientLight and AmbientColor* - These properties set the intensity and color of the global non-directional lighting source.

FloodLight, FloodDirection*, and FloodColor* - These properties set the intensity, direction, and color of the global directional light source. Like the Sun, the FloodLight is an infinitely strong light source that is infinitely far away.

FogVisible and FogStart - These properties toggle the fog effect on and off, and define how far away from the camera the fog will start. The fog gets progressively thicker until Yon, where it is completely opaque.

Wireframe - This is a boolean property that allows you to toggle how the models get rendered by Rb3DSpace: shaded or wireframe. Wireframe is sometimes useful for a 3D object editor or to create a "Sci Fi" 3D look.

DebugCube - This boolean property creates a background with all the axes marked. When you move objects or rotate your Rb3DSpace camera, you can see exactly what section of the coordinate grid you are pointing at. *Note: this feature only works within the IDE, not built apps.*

Object Properties

Rb3DSpace has a Camera, Background, and Object property associated with it. These three items are specialized versions of Rb3DSpace's core classes, Object3D and Group3D.

Camera - This object can be positioned and rotated, which affects what you see within Rb3DSpace. There is only one Camera for each Rb3DSpace in your application. In addition to position and orientation, the

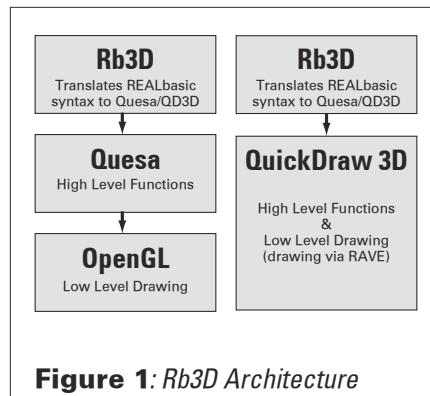


Figure 1: Rb3D Architecture

Camera is affected by the Hither, Yon, and FieldOfView properties.

Object - This property is basically a group to which you can add objects. Every object that you append to Rb3DSpace's Object property will be visible in that Rb3DSpace, provided that the object's Visible flag is true and it is in the Rb3DSpace's camera's field of view.

Background - This object works exactly like the Object property except for two differences: the Ambient and FloodLights do not affect any object in the Background group and the Background's position relative to the camera is fixed. These characteristics are useful for things like a sky box; you don't want shading on air, nor do you want the camera to go through your sky!

Events

Rb3DSpace can handle all the usual events directly. MouseUp, MouseDrag, MouseDown, MouseEnter, and MouseExit can be used to select and move objects or as a way of implementing camera movement. The Open and Close events can handle the initial loading of objects and any cleanup when an Rb3DSpace is closed.

Methods

Three methods are associated with an Rb3DSpace, two of which are typically called from a MouseUp or MouseDown event.

FindObject - This method takes an X,Y coordinate within an Rb3DSpace and returns any Object3D that lies on that point.

In a shooting game this method can be used to verify if a target has been hit.

FindPoint - This method is similar to FindObject: pass it an X,Y coordinate and it will return the equivalent 3D point. Useful for 3D editors as you can know the 3D position the user is pointing to.

Update - This method redraws an Rb3DSpace without erasing or triggering a Refresh event. This is what you call to update a change you've made to the 3D environment.

Vector3D Class

How does one know where an object is in a 3D world? The Vector3D class contains that information. You can assign it any 3D location by setting its X, Y, and Z properties. Other Rb3D classes use the Vector3D class to define their position in 3D space or their vector. I'll explain this in the future.

Quaternion Class

Which way is an object facing? A Quaternion is a way of representing an object's orientation. There are other ways of representing orientation: matrix, Euler angles, etc., but the quaternion is the most stable. As with Vector3D, other Rb3D classes inherit the Quaternion class.

Object3D Class

Object3D is the core class that most of Rb3D is built upon. As stated before, an Object3D has a Vector3D (the Position property) and a Quaternion (the Orientation property) associated with it. Other properties allow you to scale, control visibility, etc. Several methods are provided to load a model, rotate, move the object, etc.

Group3D Class

The next Rb3D class is Group3D. As the name suggests, this class allows you to treat a whole bunch of Object3D classes as one. Since it is a sub-class of Object3D, you can do all the same things to an entire Group3D as you can to one individual Object3D. You can load parts of a car (wheels - remember to clone three of them, body, etc.) separately, then load them all into a Group3D that represents the entire car. You can then move or rotate the entire car by moving or rotating the Group3D.

Light3D Class

At the time of this writing, Real Software was developing the Light3D class for inclusion with REALbasic 4.5. This class creates one of two types of lights

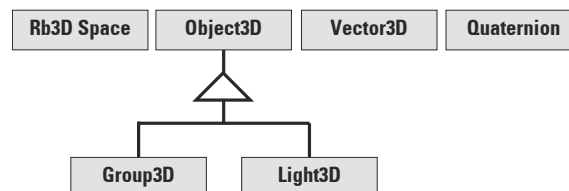


Figure 2: Rb3D Classes

From Scratch

by William Leshner

EASY AS PIE!



BEGINNER

Day 1: The Mission

Designing and building a complete application from scratch

Thank you for coming to this staff meeting. We don't have a lot of time, so I'll get right to the point.

Our marketing department has come up with a product they want us to design and build from scratch, and management has given us only six days to do it. In a nutshell, this new product helps developers release software by automatically generating the various documents usually found in a release, such as a readme, a user's guide, a web page describing the software, and an email announcing the software to mailing lists. These documents all share information about the software being released. For example, the name of the product, the company name, copyright information, and a brief description of the product are some of information you might find on the documents included in a software release. It is a tedious and time-consuming job to keep this information up-to-date in all of the documentation for every release. Our new product solves this problem by gathering and maintaining release-related information and using that information to generate release documents from templates. We are calling this new product ShipIt!.

Let us quickly go over our marketing requirements. ShipIt! must enable the user

William Leshner has been programming for twenty years and programming Macs for ten. He has spent a good deal of the last several years working in REALbasic and has come to the conclusion that REALbasic is the best development environment available anywhere.

Name	Text
product	ShipIt!
company	RBApps Inc.
version	1.0
copyright	Copyright (c) 2002 RBApps Inc.

Figure 1: A product information ListBox.

to enter information about the product to be released. This information must include the name of the software being released, the name of the developers, the company name, copyright information, and a short description. ShipIt! must provide templates for the documents it generates, and allow the user to edit these templates. Each template is used to generate one document by substituting release information into user-specified fields. ShipIt! must be able to save the release information and templates as a document file that can be opened and reused. Standard UI elements such as unique icons and an about box should be implemented, as well as the ability to run under Mac OS 8, 9, and X. A Windows version of ShipIt! is highly desirable, but not required. All versions of ShipIt! should have the look and feel of the platform on which they are running. In other words, a Mac OS 8 version of ShipIt! should look like a Mac OS 8 application, and a Windows version of ShipIt!, if there is to be one, should look like a Windows application.

The first order of business is choosing our development tool. We need a tool that is fast and easy to use, but powerful enough to finish the job. This tool must be able

to build versions of ShipIt! that will run under older versions of the Mac OS (OS 8 and OS 9) as well as Mac OS X. A tool that can also build a Windows version of ShipIt! is preferred, and all the versions must be built from one set of code. We don't have enough time to write different versions of ShipIt! for each of our target platforms. After considering a number of development environments, we have come to the conclusion that REALbasic by REAL Software Inc. is the best tool for the job. It meets all of our requirements, and it's fun to use.

Before we build ShipIt! we must design it. A good design, taking into account our marketing requirements, is essential in creating a useful application. The first marketing requirement is that ShipIt! gather release-related information from the user. A simple form could satisfy that requirement, with fields for each piece of information we have decided will be needed in our templates. This approach assumes we have correctly anticipated every piece of information about a release that any user might want. Instead of a form, a two-column list would be more flexible. Each row in the list would contain a template field. The first column would contain the

```
ShipIt! v1.0
Copyright © 2002 RBApps
```

```
What is ShipIt!?
```

```
ShipIt! helps you release your software by
taking over the tedious job of maintaining
and generating release documents.
```

Figure 2: A readme snippet.

name of the field, so that we can refer to it in our templates. The second column would contain the replacement text, which will be substituted in our templates. Such a list is easy to implement with REALbasic's ListBox control. See Figure 1 for what the ListBox might look like.

Our next requirement is that ShipIt! provide templates for the release documents it generates and that these templates be editable so that a user can customize them for each software product. First we should consider what a template is and then we should figure out how to let the user edit one. Let us consider a typical release document: the readme. A snippet of a readme is shown in Figure 2.

The parts of this readme that we might want to share with other release documents are the product's name, version number, copyright, and description. Each of these pieces of information should become substitution variables in a readme template. These variables will tell ShipIt! how to perform a substitution by naming the information the user entered in the product information ListBox we discussed previously. We just need to tell ShipIt! which pieces of text in a template are variables so that it knows what to substitute where. Our convention will be to put template variables between '#' characters. For example, a variable for the name of a product might appear as "#product#" in our templates, and "product" will also appear as the name of one of the rows in our product information ListBox (see Figure 1). Figure 3 shows how we have made a template out of the readme snippet.

A release will require a number of templates like the readme snippet in Figure 3. One of our requirements is that the user be able to edit them, which can be satisfied with a ListBox and an EditField. The user will select templates with the ListBox and edit the selected template

with the EditField. Figure 4 shows how that might look.

ShipIt! now has a total of three UI elements: a product information ListBox, a templates ListBox, and an EditField for editing one template. The templates ListBox and EditField will be together in one window, but where will the product information ListBox be? We could put it in a separate window, but a better approach is a single window with a TabPanel control. The product information ListBox would be under one tab, and the templates ListBox and EditField would be together under another tab. This design keeps each ShipIt! document in one window, which simplifies the UI and reduces window clutter.

The last ShipIt! requirement that we need to consider is the ability to save and reuse ShipIt! documents. Manipulating files in REALbasic is simple, but we need to design the ShipIt! file format. We could just invent a proprietary binary format, but unless we design it very carefully a proprietary format could be very inflexible. If, in a future version of ShipIt!, we need to change the file format, we might not be able to open new files in older versions of ShipIt!. A better approach is to design a

file format in XML. XML is easily extended and older versions of ShipIt! will simply ignore tags and attributes in the document it doesn't understand. To parse our XML files we will need an XML parser. Although we could use a third-party XML parser, we propose to write a very simple one that will handle just enough XML to read ShipIt! files. We will cover the details of the ShipIt! XML tags and the parser in a future column.

Our remaining marketing requirements are solved by our selection of REALbasic as our development tool. REALbasic will produce working applications with a platform-correct look-and-feel for all of our target platforms. An about box and menus are easy to construct and the only other

```
#product# #version#
#copyright#
```

```
What is #product#?
```

```
#shortdescription#
```

Figure 3: A template for the readme snippet in Figure 1.

consideration is icons. We will discuss these issues in the meetings that we will hold in the days ahead.

That pretty much wraps things up for today's meeting. Next time we will begin to layout and code the ShipIt! application in REALbasic, starting with the main window and its two tabs. We will add the ListBoxes and EditField and write the code to manipulate them. ■

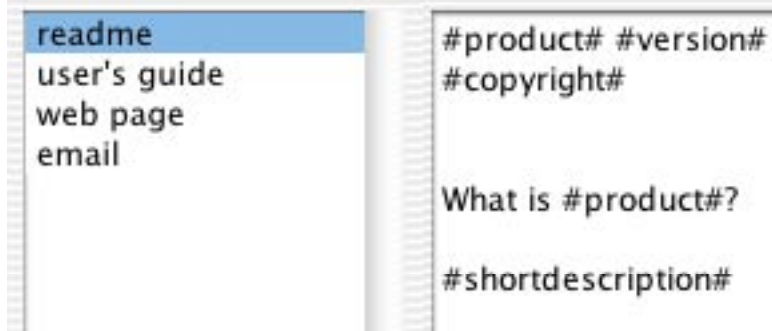


Figure 4: Templates ListBox and EditField.



Revealing AppleScript

Controlling other applications using AppleScripts

AppleScript has always been one of the best-kept secrets of the Macintosh world. Most users know it exists but haven't taken the steps to harness its powers. REALbasic does a very good job at integrating AppleScript technology into its architecture.

AppleScript allows your application to control, or be controlled, by other applications. With proper programming, this interaction between programs can work across a network, even the internet. Letting other applications control a REALbasic program is done using AppleEvents and is a topic for another article.

Controlling other programs can be accomplished in two different ways. The controlling program can create and send AppleEvents to a target program, or the controlling program can call a compiled AppleScript "program." The former is more powerful but more difficult to program; the latter method is limited but easy to program. We will concentrate on the latter method with this column.

A program can utilize three different types of scripts. The first type are simple, where the script simply accomplishes a task, like Emptying the Trash.

```
tell application "Finder"
    empty trash
end tell
```

Save this code as a compiled AppleScript and drag it into your project window in RB. The name of the script now becomes a subroutine that can be called in code. (Hint: Double click the script icon in the project window to open ScriptEditor.)

Dean Davis is the author of the shareware program WeatherManX and a huge AppleScript enthusiast.

When this subroutine is called the script will be executed and the trash emptied. When the program is compiled, the script is incorporated into the application so the actual script file doesn't have to be distributed with the application.

The second type of script can return a value to your program. iTunes will serve as an excellent target application.

Here is a script that will make iTunes return a list of playlist names.

```
tell application "iTunes"
    set PlaylistList to the name of every playlist
end tell
return PlaylistList
```

A script, when called by REALbasic, always returns a string regardless of the data type that would have been passed back by the actual script. This script is called as any other function would be called.

```
Sub Action()
    Dim PlayLists as string
    PlayLists = NamesOfPlaylists()
End Sub
```

The returned string is a list of playlist names separated by commas. It can then be parsed using the CountFields and NthField functions. The problem is that if any of the playlist names contain a comma, the returned list will be useless. So the script code needs to be modified to return a list with a different delimiter.

```
tell application "iTunes"
    set PlaylistList to the name of every playlist
end tell
set WhatImReturning to ""
repeat with i from 1 to (count of items in PlaylistList) - 1
```


```
    set WhatImReturning to WhatImReturning & item i of PlaylistList & "|"
end repeat
set WhatImReturning to WhatImReturning & last item of PlaylistList
return WhatImReturning
```

Now the returned string's playlist names are delimited by "pipe" symbols.

In the last type of script, a value will be passed to the script and then a string returned. Here is the code that will return a list of tracks in a playlist.

```
on run {playlistname}
    tell application "iTunes"
        set TrackNamesList to the name of every track of playlist playlistname
    end tell
    set WhatImReturning to ""
    repeat with i from 1 to (count of items in TrackNamesList) - 1
        set WhatImReturning to WhatImReturning & item i of TrackNamesList & "|"
    end repeat
    set WhatImReturning to WhatImReturning & last item of TrackNamesList
    return WhatImReturning
end run
```

The example program (AppleScript Demo.rb) utilizes these scripts to populate a ListBox with the playlist names, and when a playlist is chosen it will populate the other list box with the track names via the other script. I leave it as an exercise for the reader to write a script and the code to make iTunes play the selected track.

AppleScript is a somewhat overlooked tool that can be very powerful in solving many problems and helping extend the functionality of REALbasic. 



Instant Cocoa

by Colin Cornaby

The History of Cocoa

Cocoa for REALbasic programmers

When Apple released Mac OS X in March last year they did something remarkable. Included with the operating system (or available for a free download) were powerful development tools that rival the costly commercial C++ development tools of other platforms. This opened many new possibilities not only for companies wanting to port existing code to OS X, but also for shareware and freeware developers, and hobbyists. Now, amateurs and smaller developers could try programming without paying for development tools. Not only that, but Cocoa is one of the easiest varieties of C. It is a powerful Mac OS X-only language that allows programmers to use advanced portions of OS X.

In this column I'll be exploring Cocoa from a REALbasic developer's standpoint. I will also highlight the main differences between Cocoa and REALBasic. I'll be assuming that you are an intermediate RB programmer, and you know the basic vocabulary of RB (such as a method, a class, an argument, etc). I believe that having a firm understanding of RB is important before trying Cocoa, as RB and Cocoa are built on similar concepts. I'll assume, however, that you know nothing about Cocoa, Mac OS X, or C.

You might get the feeling from my introduction that Apple invented Cocoa recently. In fact, they didn't. Cocoa is probably about as old as that LC II most of us have wasting away somewhere. When Steve Jobs was exiled from Apple back in the 80's he took some of his engineers and founded NeXT, a company dedicated to

creating powerful new technology. They built futuristic machines called NeXT Cubes and a new operating system called NeXTStep, which shared the Cube's spiffy technology. It was based on UNIX and was a wonder of its time. NeXT Cubes were powered by a new variation of C++, which was "object oriented." Objects can send and receive messages as you've probably found in REALBasic. NeXT's new variation of C++ made programming a lot easier. NeXT programmers quickly and easily built interfaces, cutting development time down to a fraction of what it had been. This new language became known as Object-Oriented C.



Unfortunately, the NeXT cubes faced the same fate as their more recent incarnation, the G4 Cube. Too overpriced and overpowered for the average home user, NeXT Cubes only caught on in high tech labs and businesses. NeXT cut back their hardware division and began pushing NeXTStep as an alternative OS for (ironically) x86 based PCs.

NeXT continued with software development and became fairly popular with developers for its advanced programming language, but still never caught on in a market that was built for Windows. However, it was around this time that Apple was looking for a new OS. Apple needed a product that would compete

with Microsoft's new Windows NT OS and saw NeXTStep as exactly what it needed. Apple bought NeXT and began the process of bringing the code over to the Mac. Apple began to experiment with NeXT's programming language (now dubbed Cocoa). They actually ported Cocoa so it could run in Windows along with other Windows applications.

Apple also developed Carbon. Carbon was simply a version of OS 9's libraries that ran under Mac OS X. Because Cocoa was so different from other programming languages on the Mac, Apple realized that developers could not easily port their code from OS 9. Carbon is the version of C++ that developers use to port projects from "Classic" Mac OS to Mac OS X. Many developers still write code in Carbon so that both OS 9 and OS X users can run their programs (especially games). REALbasic and programs compiled in REALbasic are also Carbon.

The history of OS X is important because the development tools have not really changed. In fact, there are even variations of NeXTStep other than OS X that exist on other platforms. One example is OpenStep; an open source project with an OS that is heavily based on NeXTStep and Cocoa. Apple had an early version of OS X that ran on PCs called Rhapsody.

So should you expand on the familiar surroundings of REALBasic and venture into the realm known as Cocoa? Read this column. Every issue, I will be exploring Cocoa from the perspective of an RB user.

Next issue I'll compare RB and Cocoa side by side. We'll compare interface elements and general design principals. After I show you the differences and advantages of RB and Cocoa, we'll begin doing some coding so you can decide whether or not Cocoa is something you want to learn.

Colin Cornaby is an OS X developer. Current projects include "Duality," a theme changer for Mac OS X, written in REALbasic.



Dealing with Dialogs in Aqua

The low-down on designing dialog boxes for Mac OS X

Aqua, the interface component of Mac OS X, obviously has a lot of differences from Platinum, its OS 9 predecessor. Some of them are obvious changes: the Dock, the Finder windows, the Apple menu. But there are quite a few more subtle things that are new, many of which are a result of some fundamental interface changes.

One place where Aqua has introduced changes is in the field of dialog boxes — those helpful windows that appear when the application needs to communicate with the user. Dialogs designed in Platinum will almost always work perfectly when ported to Aqua, but this approach ignores a lot of design guidelines established for the new OS, and in some cases these dialogs may cause confusion for the user. Taking the extra steps to redesign the dialogs can help avoid this confusion and give your OS X application a very professional, polished look.

Some Things Never Change

Before we talk about the changes in the new interface, let's cover some of the aspects of dialog box design that apply no matter what system you're using.

Language. Text in dialog boxes should always be non-threatening and helpful. "Don't close the window until you are finished entering your preferences!" is short and to the point, but something like "Closing the window now will cause your preferences to be lost. Are you sure you want to continue?" makes the user

feel less like they've made a mistake and more like the computer is helping them from doing something they didn't mean to do. Buttons should be clearly labelled, and, if the dialog is asking a question, the buttons should give possible answers to that question. This means "Yes" and "No" are often preferable to "OK" and "Cancel." Aqua interface standards appear to encourage longer texts in buttons — like "No, don't close the window" — than Platinum standards did.

Fonts. Dialog boxes should always use the appropriate system font — REALbasic does this automatically if you enter "System" as the font name. To be honest, there are a couple of small changes: Aqua dialogs should use 13-point text instead of the 12-point text used in Platinum, and Aqua alert boxes (small, one- or two-line messages used for warning or alerting the user) should use the bold version of the system font. But that's it for variety: don't use a different font or size just because you think it looks good.

Arrangement of controls. Good dialog boxes are designed to be read from top-left to bottom-right. For example, in an alert box the icon is in the top left to catch the user's attention. The message appears to the right of this icon, and the buttons (which represent the user's response) appear in the lower right, with the most probable choice appearing to the farthest right.

Control spacing. While the numbers are different for each operating system, both Platinum and Aqua have specific pixel measurements for placement of buttons, icons, and text. You'll find them specified in detail in the Dialogs sections of the *Aqua Human Interface Guidelines* and the *Mac*

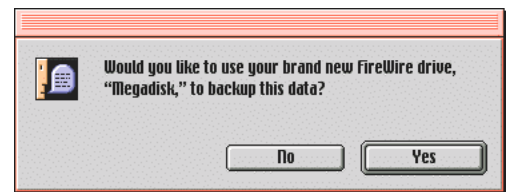


Figure 1: The Note Icon

OS 8 Human Interface Guidelines, both of which are available from Apple's web site (<http://developer.apple.com/techpubs/macos8/HumanInterfaceToolbox/HumanInterfaceGuide/humaninterfac eguide.html>). The "snap-to" guidelines that REALbasic provides are a good start, but they don't cover everything, such as distances in between buttons and other objects.

Let's take a look now at a few interface guidelines that have been changed for Aqua.

Icons

Both Platinum and Aqua alert boxes display an icon in the upper left-hand corner of the window, but the two interfaces have different criteria for choosing the icon.

In Platinum, there are three icons which can be used: the Note icon, the Caution icon, and the Stop icon. The Note icon is used for messages to the user that do not involve any risk of data loss (see Figure 1).

The Caution icon is used to warn the user of possible data loss (see Figure 2).

The Stop icon is used when an action cannot be completed due to a problem (see Figure 3).

These icons can be displayed by creating a Canvas control, 32 pixels high and 32 pixels wide, and using the

Toby Rush, a music instructor, consultant, and freelance software and web designer, has been using REALbasic since before version 1.0. His current projects include *The Interface Mafia* (www.interfacemafia.org) and his newborn son.

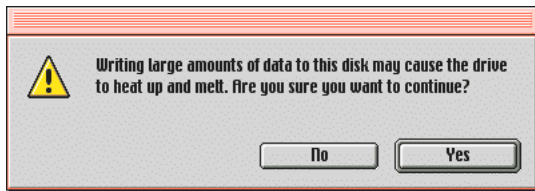


Figure 2: The Caution Icon

graphic.drawNoteIcon, graphic.drawCautionIcon or graphic.drawStopIcon methods.

In Aqua, however, the rules are a little bit different: because windows and dialogs from different programs can be layered in between one another it is important that a dialog box indicate what program it belongs to. To do this, the application's icon itself is used as the icon for the alert box. The only exception to this is when the alert box is warning about potential *irreversible* data loss; in this case, the caution icon is used, with the application icon placed as a small "badge" over the lower right part of the icon:



Unfortunately, REALbasic doesn't yet have a built-in way to display this caution icon. It does exist deep within the OS X System Library, but getting at it will take some extra code (see "The Aqua Caution Icon" sidebar).

Window Types

In Platinum, we have five different types of windows which dialogs can use:

Modal. The modal dialog has no titlebar, so it can't be moved around the screen. It puts the user in a mode (hence the name), meaning the user can't do anything else in the program until he or she dismisses the dialog box. Use of this window is generally discouraged for almost all cases.

Moveable modal. This window is still modal — the user has to dismiss the dialog before doing anything else in the program — but it has a title bar and can be moved around the screen. When your program requires a modal dialog (for a Preferences window or About box, perhaps), this is the one to use.

Alert Boxes, moveable and stationary. These two dialog box types are just like the Modal and Moveable Modal varieties, except that the borders and title bar have red highlights. They should be used for alert boxes, regardless of which icon is being used. To create one of these in REALbasic,

simply set the window's MacProcID to 1044 (Stationary) or 1045 (Moveable).

Modeless. This type of dialog doesn't put the user in a mode; that is, the user can switch to another window and do other work without closing the dialog box. Dialog boxes of this type are rare but they are gaining popularity — seasoned interface designers like them because they allow the user more flexibility. To create a modeless dialog in REALbasic, set the window's Frame parameter to "Document Window" and make sure the CloseBox item is not checked.

In Aqua, the number of dialog types has been reduced to three:

Modeless. This type is just like its counterpart in Platinum, and it's still the dialog type of choice when appropriate.

Application Modal. This type of dialog is just like the Moveable Modal dialog box in Platinum, but it's only used when the alert doesn't pertain to a specific document window. For example, the Open File dialog box isn't connected to any particular document (since it can be displayed when no documents are open), while the Save File dialog box is connected to the document being saved. In this case, the Open File dialog box should be an Application Modal type.

If you use an Application Modal dialog and include a title in the title bar (like "Open"), it's a good idea to precede it with the name of your application (like "SurfWriter: Open") so it's apparent to which application the window belongs.

Document Modal. This type of dialog is a new addition; it is attached to an existing window and appears to emerge from just under the titlebar of the existing window. This type of dialog is also known as a *sheet*, and unfortunately REALbasic applications cannot properly take advantage of them yet (windows set to MacProcID 1088 will appear to be sheets, but there are some limitations to using this method).

The OS X Caution Icon

In OS 9, icons like the alert box caution icon are easily available as system resources which can be retrieved using app.resourceFork (although this is unnecessary thanks to the graphic.drawCautionIcon method). In OS X, however, it's not quite as easy. If you're up for a challenge, here's how to find the file that contains the OS X caution icon. Starting from the root level of your OS X startup disk, navigate through the following folders:

```
System:Library:Frameworks:
Carbon.framework:Versions:A:
Frameworks: HIToolbox.framework:
Versions:A:Resources
```

Inside that last folder, you'll find a file called "HIToolbox.rsrc". A great many OS X icons are hidden in there, stored in 'icns' format. In order to use any of the images, you'll have to convert that 'icns' data into something usable by REALbasic. That's outside the scope of this article, but if you have a way to do it, e-mail me at trush@rbdeveloper.com and I'll mention the solution in a future article. (Thanks to ResExcellence for the location of this resource file!)

Pay Attention to the Details

There are a few other small differences between Platinum and Aqua dialog box guidelines, and it's well worth reading the appropriate Human Interface Guidelines documents published by Apple. Because OS X is still pretty new, many of these subtle differences are ignored by designers rushing to get their applications compiled for the new system. However, the differences add a level of professionalism and style that is well worth the effort. If you make your program feel at home in Aqua, your users will feel at home with your program. 🍎

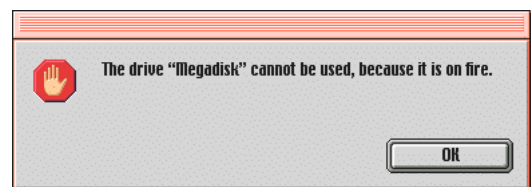


Figure 3: The Stop Icon

Regular Expressions Overdrive

Taking regular expressions to the next level

This article assumes that you already have covered the basics of regular expressions (RegExes), and have read Matt Neuburg's introduction on page 24.

Just don't bother

A discussion on one of the REALbasic discussion lists was how to suppress extra spaces in a text. The pattern that will come up immediately to most people is `[\t]+`, to be replaced with a single space. It was argued that the correct pattern should be `[\t][\t]+`, since RB's RegEx engine should start matching only when there are at least two tabs or spaces. However, the speed difference on average-sized texts was negligible (applied to this article `[\t][\t]+` is six times faster than `[\t]+`).

While a fascinating discussion, it's academic since a) we're talking microseconds or milliseconds, not seconds, and b) another fellow came up with an example using `replaceAll`, which was faster. I made it even faster by changing `inStr` to `inStrB`, and by adding a line of code to first remove odd-numbers of spaces.

Compiled, the subroutine executes in 3.8 milliseconds (5.5 milliseconds without the two optimizations), versus 18+ milliseconds with the regular expression engine. In this case, the use of regular expressions is actually slower, so they shouldn't be used.

In the sample program provided on RBD's website (see page 4 for downloading instructions), you can see that the longer the text, the more efficient `ReplaceAll` becomes (actually `ReplaceAll` is efficient,

Didier has been a dilettante programmer and linguist for more than 20 years. Unusual for a Frenchman, he speaks 11 languages, including Korean and PowerPC machine-language.

it is just that the initialization overhead makes it slow on small chunks of text).

Bionic Vision... Not!

One common assumption made by people who have played with regular expressions is that, being powerful and cryptic tools, they have some kind of bionic vision that gives them the capacity to "see" all matches at once and apply any kind of transformation to the source, from case adjustment to styling. One of the factors that contributes to this confusion is the `.ReplaceAllMatches` option, which does not, on the user/developer side, involve any looping. Snap your fingers and Bobby the RegEx will do it. Not quite so. The `.ReplaceAllMatches` is just one facility of the regular expression engine that does the looping for you. But it does need to loop.

So what if I want to have all my matches neatly tucked away in an array or something similar? Roll up your sleeves, m'dear; it's time to work. It doesn't mean you will be doing it all the time: this is a good example of a COUP (code once, use plenty). Subclass a `RegEx` and provide a `searchAll` method. In the project window, add a class and subclass it from `RegEx`. Name it `wrappedRegEx`, for instance. Add a `SearchAll` method as follows:

wrappedRegEx.searchAll:

```
Sub searchAll(source as string, byref a() as
    string, subEx as integer)

    dim m as regexMatch
    dim s as string
    dim i as integer

    s = source
    redim a(-1)
    m = me.search(s)
    while m <> nil
```

```
        a.append m.subExpressionString(subEx)
        i=len(m.subExpressionString(0))+m.subExpres
            sionStart(0)
        m = me.search(s,i)
    wend
End Sub
```

Now, let's test that. Imagine you want to catch the last word of each sentence in a text. Here's what you'd do:

```
dim r as wrappedRegEx
dim result(-1) as string
dim i, j as integer

r = new wrappedRegEx // Our new subclass!
r.options.caseSensitive = false
r.searchPattern="([-a-zA-Z']+)[.!?](\s\
r)]$)"

// the $ sign is to make sure you catch
// the last word of the last sentence
// which may not be followed by anything
// Note that result is passed as byRef and
// its value may change
r.searchAll(editField1.text, result, 1)
j = ubound(result)
listBox1.deleteAllRows
for i = 0 to j
    listBox1.addRow result(i)
next
```

This is a bare-bones RegEx; don't expect it to catch everything. The `[\s\r]` at the end of the `searchPattern` prevents the RegEx engine from matching things like `listBox1.deleteAllRows`, for instance. Cosmetic, really.

`SubEx` enables the user to chose what will be stored in the array, since we often care only about a piece of the match, and not the whole match.

Another frequent issue connected to this bionic-vision problem is applying versatile changes to a source with regular expressions, like changing the case of matches. To the beginner, it appears to be one single issue, whereas in reality it is an array of 26 possible issues. You have to keep in mind that regular expressions don't describe things; they describe the way things may look under certain conditions. So, matching words that start with a lower-case letter is easy, but asking the regular expression engine to then capitalize these letters doesn't make sense.

Looping. Again and again.

Since there are 26 letters (at least in English), we can loop through them and change each lowercase to an uppercase letter. The code would look like this:

```
dim i As integer
dim r As regEx
dim m As regExMatch
dim a, b, source As string
dim t1, t2 As double

source = editField1.text // Our source text
t1 = microseconds
r = new regEx
r.options.CaseSensitive = true
r.options.ReplaceAllMatches = true
For i = 1 to 26
    a = chr(96+i) // a-z = ascii 97 to 122
    b = chr(64+i) // A-Z = ascii 65 to 90
    r.SearchPattern="\b"+a+"([a-z]*)\b"
    r.replacementPattern=b+"1"
    m = r.search(source)
    if m <> nil then
        source = r.replace(source)
    end if
next
t2 = microseconds
editField1.text = source
staticText1.text = format(t2-t1,"###,###")
+ " microseconds"
```

Originally, I had tried with a slightly different version, using (W|^) and (W|\$) as word delimiters, but this proved to be two to three times slower.

Styling

Many programs today provide syntax coloring services, which makes editing code much easier.

There are two ways of doing syntax coloring: real-time or offline. Real-time coloring is neat, but it doesn't involve regular expressions. Let's start with off-line

coloring. For our example, we will colorize all instances of certain words in a text.

Off-line syntax coloring:

```
dim r as regex
dim m As regexMatch
dim c, c2 as color
dim s, t As string
dim i, j as integer


c = rgb(0, 0, 200) // Matches to be blue
r = new regex
r.options.caseSensitive = false
r.searchPattern="(W|^)(W+)(W|$)"
s = editField1.text // Our source text
j = ubound(myList)
m = r.search(s)
while m <> nil
    t = m.subExpressionString(2) // that's (W+)
    for i = 0 to j
        if t = myList(i) then
            editField1.selStart =
                m.subExpressionStart(2)
            editField1.selLength =
                lenB(m.subExpressionString(2))
            editField1.selTextColor = c
            exit
        end if
    next
    m = r.search(s, m.subExpressionStart(0) +
        lenB(m.subExpressionString(0)))
wend
```

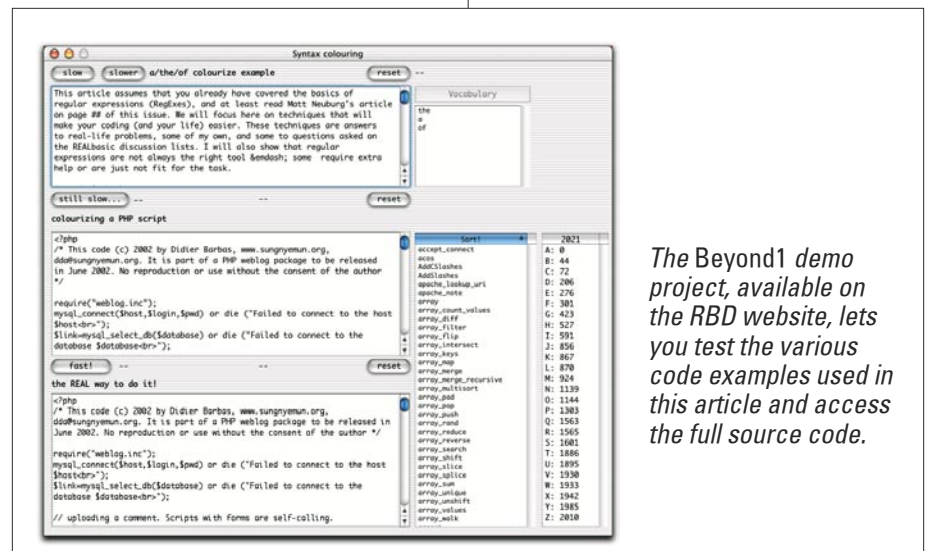
Here again, there is no way we can avoid looping or trust the grunt work to the RegEx engine. However, applied on this article, the process is quite slow. myList() is a global array that contains the list of words we are looking for; in this example a, the, of. This list relates heavily to a dictionary

of keywords, like HTML or PHP keywords. In the case of PHP, for instance, literal strings (between double quotes) are set to a different color for better discrimination. Moreover, C-style comments (/* ... */) can spread over several lines too. The problem is that both comments and literal strings also can contain keywords. The conflict is solved by styling comments first, then strings, and finally keywords, if these are not already colored. Download the demo project for the source code.

However, this is not really fast; it took my Titanium PowerBook 1.8 seconds to colorize a 3.5 KB PHP script (see example project). And there is no easy way to cut down on processing time. Here again, we have to look around for other tools to help us achieve the task. In this case switching from the standard editField to WasteField will do the trick. WasteField, designed by Marco Piovaneli and adapted for RB by Doug Holton, has many more features than the standard editField, among which are two useful functions for syntax coloring: findIt and SetAttColor. With these tools we will take a slightly different approach, since findIt does not find patterns, but words. Regular expressions will still be used where patterns are needed. Again, the source code is in the demo project.

This executes in around 60 milliseconds flat, and almost a third of that is due to the fact I have to back-pedal every time there is a match on a keyword, for fear there could be instances of this keyword preceded by a \$, i.e. variable names like \$date, which, while being legal, would be seen by WasteField as the PHP keyword date.

Here again, a combination of regular expressions and something else did the trick. 



The Beyond1 demo project, available on the RBD website, lets you test the various code examples used in this article and access the full source code.

Intel Focus

Continued from page 37

XCMDs und XFCNs

XCMDs and XFCNs are external functions originally designed to extend Apple's Hypercard development environment. You can use them in REALbasic applications running on Mac OS Classic like this:

```
#If targetwin32
// can't work on Windows
#Else
#If TargetCarbon
// can't work on Carbon
#else
CallMyXCMD
#endif
#endif
```

Colored Mouse Cursors

If you drag a resource file into your project that contains a 'CURS' resource

(a black and white mouse cursor), RB will import it and you can use it on Mac OS and Windows. After compiling your application, you can open your application's file using ResEdit and take a look at what ID this 'CURS' resource is assigned. If you add a 'curs' resource (a colored cursor, note the capitalization) to your app using a resource file it will be colored on Mac but not on Windows.

Special chars

When REALbasic compiles for Windows it will translate all char codes from Mac to Windows. But some characters, like the ellipsis "...", can't be translated and will result in a small rectangle on Windows.

RBD# 1017

	Mac OS PPC	Mac OS 68k	Mac OS Carbon	Win32
TargetMacOS	True	True	True	False
TargetPPC	True	False	True	False
Target68k	False	True	False	False
TargetCarbon	False	False	True	False
TargetWin32	False	False	False	True

Some Notes:

- None of these constants can tell you if you are running on Mac OS X or not.
- In some earlier REALbasic versions there was a constant TargetJava, but the Java compiler was never released and this constant was removed.
- You can define your own boolean constants in a module. For example:

```
// The "then" after the boolean constant is ignored like everything else on the line.
// This means that you can't combine constants in an expression
#If GermanVersion then
msgbox "Guten Tag."
#endif
```

Table 1: Target Constants in REALbasic.

Advanced Techniques

Continued from page 31

```
// Get a text file from the user.
f = GetOpenFolderItem("text/plain")
if f <> nil then
// Create the storage for the FSSpec
fsspec = NewMemoryBlock(70)
if fsspec <> nil then
// Store the result of f.name
fname = f.name
// Create the FSSpec
err = FMakeFSSpec(f.MacVRefNum, f.MacDirID,
fname, fsspec)
end if
end if
```

Note that I had to put the file name into the fname variable, since the Name property of the FolderItem class is actually a function call, making it impossible to pass f.name directly.

Accessing data from a MemoryBlock is a very important part of calling Toolbox routines. As an example, to get the filename from the FSSpec created above, you need to look back at the definition for the FSSpec data structure to determine where the data you want is located. According to the C or assembly language structure definitions, the file name is found at offset 6 in the MemoryBlock. It can be accessed as follows:

```
dim fsspec as MemoryBlock
dim filename as string
fsspec = DoSomethingToGetAnFSSpec()
filename = fsspec.PString(6)
```

These tips should give you a good start using Declare statements in your own programs. After some experimentation and reading of Apple's documentation (if you haven't already), you'll be accessing the power of Toolbox calls in no time!

RBD# 1007

The Topographic Apprentice

Continued from page 39

that can be added to Rb3DSpace. This gives you more control over lighting the 3D scene than using Rb3DSpace's built-in light sources.

The Next Step

In the next column, I'll go over the 3D Meta File (3DMF) format that Rb3D uses

to load 3D models and present techniques for loading, placing, and viewing 3D objects. I've provided a project, Rb3DSpace Demo, that you can download from the RBD website (see page 4 for instructions). It allows you to experiment with Rb3DSpace's properties and see their affect on the 3D scene.

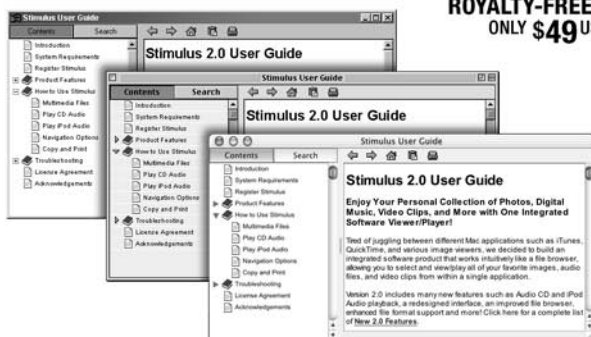
RESOURCES

Mac OS OpenGL Libraries: <http://www.apple.com/opengl/>
Windows OpenGL: <http://www.opengl.org/users/downloads/>
Quesa Libraries: <http://www.quesa.org/info/download.html>
Quesa Forum: http://www.quesa.org/quesa_forum.html
Rb3D FAQ: <http://www.strout.net/info/coding/rb/>

RBD# 1011

Apple Help Viewer, Microsoft HTML Help, WinHelp.... Why waste your precious development time designing and compiling a different Help system for each platform when you can easily use UniHelp for all of them?

NO additional plugins or classes needed. Just drag the UniHelp module and its graphics folder into your REALbasic project, add a few lines of code, create your external help pages, and that's it — instant online help for your compiled REALbasic applications that looks and functions the same on **Classic Mac, Mac OS X** and **Windows 98/NT/ME/2000/XP**.



ROYALTY-FREE!
ONLY \$49 US

KEY FEATURES:

- Displays Both HTML and ASCII Text Files.
- Supports URL Links, Relative Links, Anchors & Email Links.
- Parses More Than 20 HTML Tags, including .
- Supports Images, Video and Audio.
- Built-in Search Engine.
- Hierarchical Listbox Displays Your Help Table of Contents.
- Customizable Listbox Icons, Window Title, Start Page, Font, Contents Sequencing, etc.
- GUI Support for 6 Languages: English, Spanish, French, Italian, Portuguese and German.
- Small Footprint: Since Your Help Pages are External, only the UniHelp Module is Compiled with Your REALbasic Applications.
- Familiar Help-style Interface with Back, Forward, Home, Copy and Print buttons.
- Easy to Use & Royalty-Free!

"UniHelp" and "Electric Butterfly" are never mentioned anywhere on the UniHelp interface, providing you with a full-featured generic Help solution — your customers will think you built it yourself!



**electric
butterfly**

FREE DOWNLOAD!
<http://www.ebutterfly.com>



FOR SALE

Made with RB

Organize your programming life with Z-Write, the non-linear word processor for writers and creative types. Take control over your text! Free trial at <http://www.z-write.com>.

You can't change the weather but you can see what it is and what it's going to be with WeatherManX. Check it out at <http://www.weathermanx.com>

midnite.liteman offers creative solutions for your Mac. ideaSpiral, our idea organizer, has received a great deal of attention, and Reference Worker is highly anticipated. www.midnite-liteman.com

SERVICES

RB Consultant

Full-time REALbasic consultant (since 1.0) with over 20 years experience. Let me create your entire project or supply you with a needed class or troubleshooting. Joseph Nastasi Pyramid Designs <http://www.pyramiddesign.tv> 732 458-3738

RB Websites

Need more REALbasic tutoring? Go to REALbasic University every week at <http://www.applelinks.com/rbu/> — sponsored by REALbasic Developer!

Read by Erick Tejkowski's REALbasic column on ResExcellence: <http://www.resexcellence.com/realbasic/>

REALbasic Developer classified advertisements are a **great** value! They start at just \$0.15/byte (character). (A typical four-line ad runs about \$20 u.s.) The cost goes down the more times you run the ad!

Purchase online at:

[<http://www.rbdeveloper.com/classifieds.html>](http://www.rbdeveloper.com/classifieds.html)

Extend REALbasic with Advanced Plugins

We offer a wide range of controls for REALbasic: Grid, StyleGrid, CustomGrid, DataGrid, DateControl and WindowSplitter



Cryptography:

Provides data encryption, encoding, compression and hashing for REALbasic.

Supported Algorithms:

- Encryption: e-CryptIt, BlowFish (448 bit), AES (Rijndael) (256 bit)
- Encoding: e-CryptIt Flexible, Base 64, BinHex, MacBinary III, AppleSingle / Double UUCoding
- Compression: Zip on Strings and streams (.gz)
- Hashing and Checksums: CRC32 / Adler32 MD5 / HMAC_MD5, SHA / SHA1 / HMAC_SHA1

We offer various other plugins for REALbasic, including plugins for advanced MacOS Toolbox tasks and highly specialized utility plugins.

Check www.einhugur.com to download free demo versions.



Einhugur Software
sales@einhugur.com
www.einhugur.com



Our First Challenge

Win fame and cool prizes!

In REALchallenge, I'll have contests that will appeal to both beginners and experienced users. These challenges may consist of entire programs, games, classes, or just algorithms. One issue I'll do a beginner challenge; the next may be an intermediate or advanced challenge. Regardless of difficulty level, each issue I will present a new programming task and give you a couple months to write, debug, and submit your solution.

Your submission will be reviewed on the following criteria: code style, user interface (if applicable), features, and overall function. *REALbasic Developer* will choose the best submission and I'll feature it in a future column. I may also show my own solution to the task, and explain the logic behind the code.

Winners will be presented in the following categories: Most Efficient Code, Best Interface (if applicable), Coolest Feature, and Best Overall. Each winner will receive a book of their choice, courtesy of **Peachpit Press** (we'll contact you with a list of books to choose from), and possibly other prizes as well!

This Month's Challenge

Difficulty Level: Beginner

Your challenge for this month is to create a simple rock/paper/scissors game with a stellar interface. The emphasis is on the interface, so focus on that but do not leave out playability and features, as those are almost as important! Also don't forget, quirky features will help your score, so don't be afraid to give it Internet play,

a scoring system, etc. Just make sure that anything you add does not impede the gameplay. The programming aspects should be easy if you know how to play the game. Rock/Paper/Scissors is a two-player game where each player simultaneously chooses a "weapon" — either rock, paper, or scissors. See Figure 1 for "The Idiot's Guide to Rock/Paper/Scissors" which provides the win/lose scenarios for each possible combination.

The deadline for this challenge is September 30th. Winners will be announced in the December issue of *REALbasic Developer*.

Submission Rules and Requirements

When you feel your work is ready for submission, check it for bugs, compile it, and make sure it runs compiled. Make sure you have not used any plugins, and that any classes/modules are unprotected. All code must be your own; you may not use any 3rd party classes or modules. Save your project as a *REALbasic* project file. Place the source in a folder, and don't forget to include any necessary files such as pictures, sounds, movies, resource files,

and cursors. Compress the folder in StuffIt format (.sit) and email it to realchallenge@rbdeveloper.com with the subject "REALchallenge Submission August 2002." Make sure to include your name, company (if applicable), mailing address (for sending you your prize), and telephone number in the email. Show the world your talent!

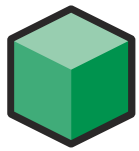
Copyright Agreement

By submitting your source code to us you grant us the right to distribute your source code to all of our print and internet subscribers. This means printing your project's source code and/or screenshots in *REALbasic Developer* or on the *RBD* website. We may release your software in either compiled or uncompiled form to our subscribers at our discretion. *REALbasic Developer* does not reserve any rights to your code (you still own your code), merely the right to distribute it. You will always receive full credit as the author and copyright holder. Readers of the magazine are not permitted to use your code in their own projects without your written permission.

	Rock	Paper	Scissors
Rock	draw	paper wins	rock wins
Paper	paper wins	draw	scissors wins
Scissors	rock wins	scissors wins	draw

Figure 1: The Idiot's Guide to Rock/Paper/Scissors

Sean has been programming since he was eleven and has used *REALbasic* since version 1.0.



REAL Software

**Congratulates REALbasic Developer
for the inaugural issue!**

Partner with us Today!



Join the Made with REALbasic program:

- Free space on the REALbasic CD
- Free distribution of announcements to thousands of subscribers of REALnews
- Free promotion on the REALbasic website

www.realbasic.com/mwrb

SCOTT MELCHIONDA

SCOTT@SCOO.COM

770-934-2550

WWW.SCOO.COM



ILLUSTRATION



MOTION
GRAPHICS



USER
INTERFACE



CORPORATE
IDENTITY



Get **REAL...** with **FRONTBASE**

FrontBase™ is a robust, fully scalable, high-performance, relational database that strictly adheres to international standards. Together with your favorite development environment, you have an unbeatable combination!

FrontBase features

- Easy to embed for standalone applications
 - Zero restart time
 - Backup of live databases
 - Standards compliance — SQL92, Unicode, TCP/IP...
 - Instant Versioning™ for zero downtime
 - Advanced security features
 - Low Total Cost of Ownership
 - Virtually zero administration
 - License and support programs to suit standalone to Enterprise-wide solutions
-
-

We've had a lot of requests for links to high-end databases, and the FrontBase Plug-in is a great example of what our users have been asking for. We believe the power of FrontBase's database server technology is a perfect match for REALbasic's powerful, but easy-to-use development environment. — Geoff Perlman, CEO REAL Software Inc.



FrontBase plug-in features

- Incorporate into Mac OS (9/X) or Windows-based applications
- Full support for REALbasic database classes
- Simple administration using the built-in database management class

Using the FrontBase REALbasic database plug-in, you can create killer REALbasic applications that utilize the full power of an industrial-strength database — as a client, or with the database embedded in your application.

FrontBase Inc. — www.frontbase.com — sales@frontbase.com
Los Angeles • Seattle • Copenhagen • Distributors worldwide

All product names and company names and logos mentioned herein are the trademarks or registered trademarks of their respective owners. Other products and corporate names mentioned which are trademarks of a third party are used only for explanation and for the owners' benefit and with no intent to infringe.