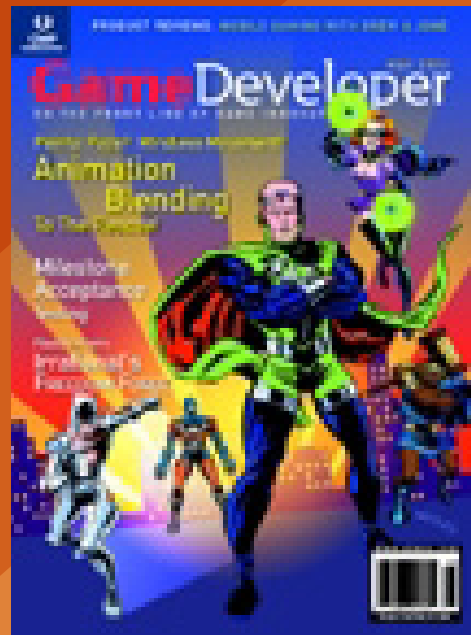


# gd

GAME DEVELOPER MAGAZINE

MAY 2002





# GAME PLAN

LETTER FROM THE EDITOR

## Crossing Over

As much as game-industry cheerleaders tout the apples-to-oranges statistic that game sales figures continue to outpace Hollywood box office receipts, no one would sensibly think that the game industry is even close to leaving Hollywood in its wake, either financially or in the currency of artistic legitimacy. As long as poetry collections can become Broadway musicals, books can become movies, and TV shows can spawn chart-topping singles, enterprising types won't be able to resist brute-forcing a given work into a different medium in the quest for a quick buck.

Most entertainment providers seem to understand how this system of media crossovers works and manage expectations accordingly. For every *To Kill a Mockingbird* that elevates an existing, profound work of art to new levels in a different medium, there are hundreds of *Car 54, Where Are You?*'s racing to a theater near you. The latter are unabashedly products of business enterprises, with decidedly less of an emphasis on the artistic exploration of a given medium.

The game industry is used to being manhandled by Hollywood at the apex of every gaming fad, when people see an opportunity to cash in on a trend. Last year saw what looked like another resurgence of this cyclical games-into-movies trend (as opposed to the opposite status quo of movie and TV licenses migrating to oft-mediocre games), with the release of the *Tomb Raider* and *Final Fantasy* movies. Many brought up the game-to-movie box-office flops of years past and predicted doom for these projects, lamenting the further aspersion that box-office failure might cast on the legitimately good works being turned out by the game software industry, which go largely unnoticed by the general public.

But whereas past abject flops have sent Hollywood recoiling from the game industry as quickly as they pounced on what they thought was a hot trend, last fall's new console releases and the increasing financial momentum of the game industry

seem to have sustained people's interest in bringing games to film and TV. There's still a good business argument to be made for doing so, both for game studios and publishers and for film and television producers. But the inherent challenges that have beset past attempts to cross over have not been eliminated.

The first is that games are successful and appreciated largely for their interactivity. Game developers understand this fact, but they still haven't been able to fully fathom the essence of interactivity and how to exploit this unique facet of our medium to advance games as their own art form (see Chris Hecker's Soapbox this month on page 56 for his take on the issue). It's difficult to help someone translate such a pivotal concept to a medium that utterly lacks it, when we still do so much hand-wringing trying to figure it out for ourselves.

Games also tell interesting stories sometimes, but our being yet so far from divining the ideal relationship between storytelling and interactivity intensifies the problem of translating whatever the essence of this relationship is to completely noninteractive, all-story-based media. Game developers who prematurely fancy themselves Hollywood-style auteurs and fast-buck Hollywood producers have conspired time and again to confound each other.

The crossover momentum is increasing nonetheless, as G4 Media aims to launch a cable channel devoted to games, more game titles than ever are being optioned for film, and even Microsoft has engaged a creative talent agency to pitch its PC and Xbox titles for film and TV. Microsoft Games Studios general manager Stuart Moulder gives me great hope that something good may finally come of these latest efforts with his remark on Microsoft's decision to enlist the agency's help: "We don't understand Hollywood. We're the wrong people to try to make movie or TV deals."

Now who says our industry never learns from its mistakes?

# GameDeveloper

600 Harrison Street, San Francisco, CA 94107 t: 415.947.6000 f: 415.947.6090

## Publisher

Jennifer Pahlka [jpahlka@cmp.com](mailto:jpahlka@cmp.com)

## EDITORIAL

### Editor-In-Chief

Jennifer Olsen [jolsen@cmp.com](mailto:jolsen@cmp.com)

### Managing Editor

Everard Strong [estrong@cmp.com](mailto:estrong@cmp.com)

### Production Editor

Olga Zundel [ozundel@cmp.com](mailto:ozundel@cmp.com)

### Product Review Editor

Daniel Huebner [dan@gamasutra.com](mailto:dan@gamasutra.com)

### Art Director

Audrey Welch [awelch@cmp.com](mailto:awelch@cmp.com)

### Editor-At-Large

Chris Hecker [checker@d6.com](mailto:checker@d6.com)

### Contributing Editors

Jonathan Blow [jon@bolt-action.com](mailto:jon@bolt-action.com)

Hayden Duvall [hayden@confounding-factor.com](mailto:hayden@confounding-factor.com)

Noah Falstein [noah@theinspiracy.com](mailto:noah@theinspiracy.com)

### Advisory Board

Hal Barwood LucasArts

Ellen Guon Beeman Beemania

Andy Gavin Naughty Dog

Joby Otero Luxoflux

Dave Pottinger Ensemble Studios

George Sanger Big Fat Inc.

Harvey Smith Ion Storm

Paul Steed WildTangent

## ADVERTISING SALES

### Director of Sales & Marketing

Greg Kerwin e: [gkerwin@cmp.com](mailto:gkerwin@cmp.com) t: 415.947.6218

### National Sales Manager

Jennifer Orvik e: [jorvik@cmp.com](mailto:jorvik@cmp.com) t: 415.947.6217

### Senior Account Manager, Eastern Region & Europe

Afton Thatcher e: [athatcher@cmp.com](mailto:athatcher@cmp.com) t: 415.947.6224

### Account Manager, Northern California & Southeast

Susan Kirby e: [skirby@cmp.com](mailto:skirby@cmp.com) t: 415.947.6226

### Account Manager, Recruitment

Raelene Maiben e: [rmaiben@cmp.com](mailto:rmaiben@cmp.com) t: 415.947.6225

### Account Manager, Western Region & Asia

Craig Perreault e: [cperreault@cmp.com](mailto:cperreault@cmp.com) t: 415.947.6223

### Account Representative

Aaron Murawski e: [amurawski@cmp.com](mailto:amurawski@cmp.com) t: 415.947.6227

## ADVERTISING PRODUCTION

Vice President, Manufacturing Bill Amstutz

Advertising Production Coordinator Kevin Chanel

Reprints Cindy Zauss t: 909.698.1780

## GAMA NETWORK MARKETING

Senior MarCom Manager Jennifer McLean

Marketing Coordinator Scott Lyon

Audience Development Coordinator Jessica Shultz

## CIRCULATION

Group Circulation Director Catherine Flynn

Circulation Manager Ron Escobar

Circulation Assistant Ian Hay

Newsstand Analyst Pam Santoro



Game Developer is BPA approved

Game Developer is BPA approved

## SUBSCRIPTION SERVICES

For information, order questions, and address changes

t: 800.250.2429 or 847.647.5928 f: 847.647.5972

e: [gamedeveloper@balldata.com](mailto:gamedeveloper@balldata.com)

## INTERNATIONAL LICENSING INFORMATION

Mario Salinas

t: 650.513.4234 f: 650.513.4482 e: [msalinas@cmp.com](mailto:msalinas@cmp.com)

## CMP MEDIA MANAGEMENT

President & CEO Gary Marshall

Executive Vice President & CFO John Day

President, Technology Solutions Group Robert Falertra

President, Business Technology Group Adam K. Marder

President, Healthcare Group Vicki Masseria

President, Specialized Technologies Group Regina Starr Ridley

President, Electronics Group Steve Weitzner

Senior Vice President, Business Development Vittoria Borazio

Senior Vice President, Global Sales & Marketing Bill Howard

Senior Vice President, HR & Communications Leah Landro

Vice President & General Counsel Sandra Grayson

Vice President, Creative Technologies Philip Chapnick



United Business Media

# GamaNetwork

# INDUSTRY WATCH



THE BUZZ ABOUT THE GAME BIZ | *daniel huebner*



Take-Two's Rockstar Games published *STATE OF EMERGENCY* at a time when Take-Two's financial reporting was in something of a similar state.

**Take-Two resumes trading.** Take-Two may have released its long-awaited restated financials — twice — and resumed trading, but many investors fear that this isn't the end of the company's woes. On February 12, the company restated nearly two years of its financials and admitted to overstating 2000 revenues by almost \$20 million. Take-Two also revealed that the Securities and Exchange Commission had issued a formal order of investigation related to the company's financial statements and accounting controls. One day later, Take-Two released its results for the fourth quarter of 2001, posting a net loss of 15 cents per share on sales of \$123 million. In December, the company had projected a profit of 1 cent to 4 cents per share on sales of \$140 million before announcing the decision to delay its reporting. Two weeks later, Take-Two again restated financial results for several quarters, this time because of math errors.

When Take-Two finally resumed trading after being halted for three and a half weeks, investors sent shares tumbling 19 percent to \$15 per share in the first 15 minutes. So far the only executive casualty from the company's financial mess is chief financial officer Albert Pastino, whose resignation was termed due to personal reasons. Pastino had been on the job just two months, having replaced Chip David in December shortly after the company announced the need to restate its numbers.

## Interplay faces Nasdaq delisting.

Interplay has received formal notice that

it will face delisting from the Nasdaq stock exchange unless the company's market value and share price rise. The company has been below a minimum market value based on publicly held shares for over 30 days, in violation of requirements set by the Nasdaq market for listing shares. The February 14 deficiency notice also said that Interplay does not meet Nasdaq's alternative listing requirements, which require stockholders' equity of \$10 million, a minimum market value of publicly held shares of \$5 million, and a minimum bid price per share of \$1. If Interplay fails to meet the market's conditions, the company would face delisting on May 15.

## Ubi Soft wins preliminary victory in Red Storm battle.

Take-Two has agreed to pay more than £1.1 million (\$1.6 million) to Red Storm, a Ubi Soft subsidiary, including reimbursement for some of Red Storm's legal costs. The trial concerns Take-Two's sales of Red Storm games under a European distribution arrangement prior to Red Storm's acquisition by Ubi Soft. Red Storm contends that Take-Two still owes nearly £6.3 million (\$8.9 million) in missing payments. To date, the court has ordered Take-Two to pay Red Storm more than £3 million (\$4.3 million). The court also agreed to Red Storm's request to fast-track the remainder of the case.



Immersion's force-feedback technology, used in games such as Lionhead's *BLACK & WHITE*, is now the subject of a patent infringement case against Sony and Microsoft.

## Immersion files force-feedback patent infringement suit.

Immersion Corporation, a developer and licensor of haptic feedback technology and the owner of more than 150 related patents,



Ghost Recon, a Red Storm Entertainment title.

has filed an infringement lawsuit against Microsoft and Sony. The suit, filed in the U.S. District Court for the Northern District Court of California, alleges that the force-feedback controllers, accessories, and games for the Xbox, PSX, and PS2 infringe on Immersion patents.

**No recovery for Midway.** Midway's fiscal results for its second quarter ended December 31, 2001, showed a sharp drop in revenues, from \$80 million last year to \$44 million this year. The company attributed the fall in revenue to the low number of titles released during the quarter and the company's hasty exit from the coin-operated games business. The net loss for the quarter was \$2.7 million, compared to a net loss of \$3 million last year. 🐛



UPCOMING EVENTS

## CALENDAR

### ELECTRONIC ENTERTAINMENT EXPO

LOS ANGELES CONVENTION CENTER  
Los Angeles, Calif.

Conference: May 21–23, 2002

Expo: May 22–24, 2002

Cost: \$200–\$475

[www.e3expo.com](http://www.e3expo.com)

Expo admission free to game industry professionals

### MEDPI 2002 SOFTWARE

GRIMALDI FORUM

Monte Carlo, Monaco

June 25–28, 2002

Cost: variable

[www.medpi.com](http://www.medpi.com)



## Games on the Run: BREW and J2ME

by ralph barbagallo

Games on mobile phones are nothing new. With NTT DoCoMo's famous iMode service, Japan has gone mobile-game crazy, and even we Stone Age Americans and Europeans have been able to squeeze some entertainment out of primitive text WAP games on our handsets.

A new generation of phones similar to those available in Japan is approaching Europe and the Americas, however, and two major platforms are vying for supremacy over these new handsets. Sun Microsystems' Java 2 Micro Edition (J2ME) and Qualcomm's Binary Runtime Environment for Wireless (BREW) emerged in the last half of 2001 as the leading development environments for mobile gaming.

### J2ME

The concept of Java on embedded and small devices has been around for some time. Sun's efforts to provide a slimmed-down Java have taken many forms over the years, including the Java Card API and PersonalJava, a predecessor of sorts to J2ME; the Japanese handsets used with iMode have a customized version of Java developed before Sun could finalize a J2ME standard.

When we speak of J2ME for mobile phones, we really mean J2ME using the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP) running on top of the K Virtual Machine (KVM). Essentially, the CLDC and MIDP define a set of Java services and language features available for a family of related devices, in this case, mobile phones and other similar technologies. These Java features

```
File Edit Project Help
New Project... Open Project... Settings... Build Run Clear Console
Device: Motorola_i85s
Project "flarb2" loaded
Project settings saved
Building "flarb2"
Build complete
Execution completed successfully
250468 bytecodes executed
15 thread switches
315 classes in the system (including system classes)
889 dynamic objects allocated (53740 bytes)
3 garbage collections (21088 bytes collected)
Total heap size 500000 bytes (currently 461204 bytes free)Execution completed successfully
5124345 bytecodes executed
4879 thread switches
319 classes in the system (including system classes)
27896 dynamic objects allocated (1008628 bytes)
11 garbage collections (979876 bytes collected)
Total heap size 500000 bytes (currently 452796 bytes free)Execution completed successfully
4189 bytecodes executed
```

The J2ME Toolkit's KToolbar is a simple yet powerful MIDP project manager.

are implemented on top of a special virtual machine, KVM, designed to be compact and portable to many small devices. Applets developed for MIDP are called MIDlets. This article uses CLDC/MIDP as the platform to compare with Qualcomm's BREW.

**The development environment.** Most Java IDE vendors have made J2ME/MIDP support a prominent feature in their new Java development tools. However, Sun's free J2ME Toolkit, available at <http://java.sun.com/j2me>, has just about everything you need to begin developing a MIDlet.

Among the most useful tools in Sun's J2ME Toolkit is KToolbar, a simple project manager that allows you to load, compile, and run MIDP projects through a clean, elegant GUI. KToolbar also allows you to run your MIDlet in Sun's standard device emulator while seeing

debug output and error messages in its main window.

The emulator doesn't accurately represent how your code will behave on the real device. First, the emulator doesn't reflect the handset's performance; the emulator's speed depends entirely on your computer's CPU. Your applet will run much slower on actual hardware.

Sun's emulator is also inaccurate in other ways. In the case of Motorola's i85 iDEN handset, simple operations such as the placement of "soft buttons" (tool tips for function buttons on the handset), the behavior and display of GUI components, and basic graphics operations vary wildly from emulator to handset. Even the screen resolution on the emulator is wrong (111×100 on the handset, 105×78 on the emulator). In the end, there is no substitute for testing your MIDlet on an actual handset.

**RALPH BARBAGALLO** | *Ralph (flarb@concentric.net) has most recently worked on Ion Storm's 3D RPG, ANACHRONOX. He now consults for various mobile gaming companies and has developed several titles in both BREW and J2ME with his development company, Flarb Development (www.flarb.com). Ralph is currently working on a book, Wireless Game Development in C/C++, due in July 2002 from Wordware.*

So, how do you actually test your MIDlet on a real handset? This is not part of the J2ME Toolkit, but depends on tools provided by the handset manufacturer. In the case of Motorola, their Java Application Loader (JAL) can be found on the iDEN developers' web site, [www.motorola.com/iden](http://www.motorola.com/iden). This tool allows you to upload .JAR files containing your MIDlets to any of their iDEN handsets.

It's worth mentioning that there is some effect that Java itself has when developing for such small hardware. Java's language conventions make it somewhat difficult to keep code size small. However, using Java also means you generally don't have to worry about strange memory errors and pointer math. Also, the phone OS handles a lot of housekeeping issues for you, such as suspending your applet when an incoming call is received. And MIDP includes handy collection classes such as vectors and hashtables.

Overall, working with the J2ME development environment is a real pleasure. Whether you are using the included KToolbar application or the J2ME editions of any major Java IDE, the process of programming and running MIDlets on hardware is a snap. There are plenty of issues with emulation performance and accuracy, but given the scope of the average mobile game, they are relatively minor. With every release, Sun continues to improve their tools.

**Graphics features.** The structure of J2ME's graphics system is similar to that of the Java 2 Standard Edition (J2SE). The familiar image and graphics classes are available, but with less functionality. J2ME's native file format is .PNG files, with devices commonly supporting 1- to 8-bit color. Palettes are hard-coded in most handsets, so colors are mapped to the closest match. Pixel transparency, often thought to be impossible with MIDP, is actually an optional part of the MIDP standard, but it's up to the handset manufacturer to support this option.

The usage of the image class allows for off-screen images and, therefore, double buffering. Many handsets have double-buffered displays by default, requiring no additional programming. Although it's



The LCDUI package has control classes suited to the display of this phone.

possible to create an image from an array of bytes as pixel data, it's impossible to do the opposite. This means pixel-level access to bitmaps is impossible, or at least extremely annoying. Aside from images, the graphics class supports the drawing of different geometric primitives such as boxes and lines as well as text.

**GUI.** MIDP has done away with the Abstract Windowing Toolkit (AWT) package and replaced it with a package called LCDUI. This package has a set of GUI control classes that are suited for the display of a mobile device.

MIDP's GUI components are grouped into two types: items and screens. Items are GUI controls in the traditional sense: components that can be added to a screen to build an interface. For instance, you can add a text field, a static text string, and a bitmap image to create a screen of several different GUI components. Or, you can use GUI controls that are screens themselves. For example, the list screen is a simple menu of selectable items. Since this takes up the entire screen, you can't combine the control with bitmap images or other custom graphics. The screen system can be rather inflexible, and both types don't allow you to explicitly position controls in pixel coordinates.

**Cost.** Aside from the cost of a phone, you can begin J2ME development without spending a dime. The prices vary on the different commercial Java IDEs such as Metrowerks' CodeWarrior, Borland's JBuilder, IBM's VisualAge, and others. Sun's J2ME Toolkit also integrates with their own free Forte IDE. The real cost comes with getting your MIDlet tested

with each carrier for distribution. These testing runs can often cost thousands of dollars. Considering the myriad of carriers who support or have announced support for J2ME, this can get expensive.

**Documentation.** J2ME comes with extensive documentation in the form of HTML files. Sun's J2ME Toolkit comes with many sample programs which display features such as GUI controls, graphics, and network communication. There are also a few great books out on the market about J2ME.

**Support.** There are several developer mailing lists and web boards for most J2ME handset manufacturers. With these resources, you will find company representatives and helpful developers answering questions and posting news on new tools, hardware, and events.

**Final comments.** J2ME and MIDP are a relative breeze to get started and develop with. MIDP, while basic, gives enough functionality to develop games. However, the Java language makes writing compact code somewhat of a chore.

J2ME handsets vary in performance. Motorola's iDEN hardware is impressive: with a reasonable blitting speed, it's possible to squeak out about 12 frames per second for a simple action game. Unimpressive perhaps, but I have seen competing J2ME handsets with frame rates in the single digits.

Because J2ME is an open standard, each carrier and handset manufacturer is free to add its own custom packages to the default installation. However, porting code between different flavors of J2ME is relatively painless. Each carrier also has its own testing guidelines and billing standards, so you need to set up a billing relationship and get your applet tested by each carrier before being distributing on their network.

## BREW

Qualcomm introduced Binary Runtime Environment for Wireless (BREW) technology in early 2001. A C/C++ programming API for a specific kind of hardware platform, BREW is also a certification and distribution

model for getting mobile phone applications out to your audience.

Qualcomm invented the CDMA standard, widely used in the United States for digital mobile phone communications. Qualcomm once manufactured CDMA handsets for various carriers, and the internal programming API used to develop software for these phones lives on in the form of BREW.

BREW is more than just a programming API, it also includes a distribution system whereby Qualcomm offers your application to all participating BREW carriers and manages billing services to collect subscription and purchase fees. Once your application goes through the compatibility-testing process with NSTL (an independent testing lab), it's then priced by the developer and made available to BREW carriers, who, having agreed on the pricing and accepted the application, then make it available for users to download. Qualcomm's singular control over the BREW standard may be a real advantage if they can get more carrier support.

This article includes information on BREW 1.0, as 1.1 was released too close to press time to evaluate; however, important new features are noted where relevant, and Qualcomm is already hard at work on BREW 2.0.

**The development environment.** The BREW API can be used natively with either C or C++. The vast majority of the included sample programs are written in C using a bizarre set of macros which seem to convert C++ object references to C macros masquerading as API function calls. Most of these alleged functions take an interface pointer as the first argument, which would be the implicit pointer in a C++ environment. The C++ examples in the SDK still use these macros instead of the objects themselves. However, at least you can encapsulate your own code in an object structure when using C++.

Getting started in BREW is a bit more complicated than J2ME. You simply can't start coding immediately. After retrieving the SDK from [www.qualcomm.com/brew](http://www.qualcomm.com/brew), you must create a project from scratch or edit one of



**BREW's native emulator: Don't use this to judge performance from your compiled binary.**

the existing example projects for a framework (BREW 1.1 adds the AppWizard to get started more quickly). Then there are a few other files associated with a BREW application that must be generated. The first is the BREW ClassID (BID) file. This is a simple C source file that contains a definition of a unique ID value that is included as a way to distinguish your application from others. It's possible to create your own BID value by using a number not present in any of the example application BIDs. This will allow you to run your application locally.

For final release and to pass Qualcomm's QA requirements, you need to create a genuine BID using the BREW developer web site. Another file you need to create is the Module Information File (MIF) using BREW's MIF tools. A MIF contains the icon that is seen on the phone's desktop and the application name, copyright information, and permissions. Once you begin testing your application on a BREW handset, you need to generate a test signature file for your phone using the BREW developer's web site.

Programming for the BREW platform is similar to writing most event-driven C/C++ applications. However, BREW has a few guidelines you must follow: no floating point calculations, no static data or global variables, and no standard library calls. Also, keep the paucity of memory on the average BREW device in mind; some common BREW handsets have a mere 500 bytes of stack space.

Much like J2ME, BREW has an emulator. When you compile a BREW project in Visual Studio, you create a Windows

DLL. This DLL, as well as the MIF and any associated files, is then run through the emulator. The performance of the emulator is also directly tied to the CPU speed of the host PC; developers basing their application performance on the emulator are in for a shock when they compile a native binary.

To compile a native ARM binary, you must spend \$1,500 on the ARM Builder software (which includes the compiler, linker, and assembler components of the ARM BREW developer suite). You will also need a \$400 yearly subscription to Verisign's Authentic Document Service, which allows you to generate up to 100 BID and signature files annually. Larger subscription fees give you more files. You must also obtain the BREW AppLoader, which allows you to upload your compiled application to the handset.

Once you set up the ARM compiler, it's time to test on a real phone. In doing so, you will often find your application crashes or responds with some error message that you didn't get in the emulator. Tracking down hardware bugs is a nightmare descent into medieval debugging techniques; there are no tools included for debugging on actual hardware. You can't so much as get a simple printf from the phone to your host PC's console window. I also found some handsets had various bugs in their implementation of BREW: the Kyocera 3035 handset, for example, would not send resume messages after certain events, and other handsets were not properly interpreting arguments to certain function calls. Because Qualcomm is constantly revising BREW, it's unknown whether these bugs will still be present in the versions of BREW carriers decide to use.

Though the development and debugging process on a handset may be the ultimate in pain, the ability to use C/C++ and Visual Studio, not to mention the freedom of programming without Java's sandbox security model, makes BREW a very game programmer friendly environment. One major complaint I have, aside from hardware debugging, is that Qualcomm doesn't currently provide support for the free GNU GCC ARM compiler.

**Graphics features.** BREW is a bit more gamelike in its graphics functionality. Much like J2ME, BREW provides mechanisms for displaying shapes and vector graphics as well as bitmap images. BREW currently lacks some critical features such as double buffering and low-level pixel access to bitmap data, but Qualcomm promises these and other improved graphics features in future releases.

In version 1.0, only Windows BMPs can be loaded and displayed; 1.1 adds PNG and BREW Compressed Image (BCI) support. BREW contains a high-level image class that can load and display BMPs; it can also animate BMPs that include multiple like-sized frames. However, converting a BMP to a device-dependent bitmap and then using BREW's own bitblt function to copy native image blocks to the frame buffer is the fastest method.

BREW supports four color systems: 1-, 2-, 4-, and 8-bit graphics. Both 4-bit and 8-bit graphics provide color, but only through hard-coded palettes. BREW's hard-coded 8-bit palette is rather garish and makes it quite a challenge for most artists to make use of its rather mismatched array of colors. Also, much like J2ME, manipulating the palette is impossible. Special colors are reserved in the 4- and 8-bit palettes for pixel transparency; masking must be used with 1-bit.

**GUI.** BREW has many of the same GUI components as J2ME. Individual controls can be created in code and positioned in absolute screen coordinates for the ultimate in flexibility, or laid out with the included resource editor.

BREW's ability to position and size GUI controls makes it easy to display graphics and GUI components at the same time. This feature is useful if you want to have a title graphic over the main menu of your game or display a simple menu control on top of the current gameplay screen. However, J2ME's controls are encapsulated in easy-to-use objects, whereas BREW requires a lot of fumbling around with SDK calls and message handlers to get your components to behave.

**Cost.** BREW's basic tools are free to standard authenticated BREW develop-

ers, with additional development costs mentioned previously. You also have to use Microsoft's Visual Studio for compiling on Win32 for use with the emulator. You can also opt to pay for higher levels of developer access which can gain you quicker payment cycles, access to beta SDKs, marketing support from Qualcomm, and various other perks.

**Documentation.** BREW comes with extensive documentation of the SDK and all related tools in a series of PDF files. BREW also comes with quite a few examples demonstrating most major portions of the API. These include a few complete games as well as examples of GUI, graphics, networking, and sound.

**Support.** Qualcomm maintains a series of mailing lists and message boards that provide invaluable help from fellow developers and Qualcomm representatives. The mailing list is archived on Qualcomm's own developer web site. Although you can pay for higher developer status that gains you better technical support, Qualcomm provides excellent information through its own free e-mail support service and official representatives on the mailing list.

**Final comments.** Overall, BREW is a more difficult and expensive platform to develop with than J2ME. Creating a bug-free native binary is a torturous process hindered by a serious lack of tools. Because BREW is currently only used by Verizon Wireless domestically and by a few carriers overseas, handsets are rather uncommon. However, Verizon Wireless is the largest wireless phone carrier in the U.S., and BREW's continuing roll-out in Korea and Japan could be huge.

The BREW API itself has a bit more functionality with graphics and GUI components than J2ME. BREW has great features such as MP3 playback and SMS messaging just waiting for carriers and handset manufacturers to take advantage of them. Because BREW uses C/C++, it's much easier to write compact code using structs and preprocessor tricks. And, with no Java sandbox security model, there's more flexibility in memory and file access.

For game developers, the current BREW hardware is rather unimpressive. The Kyocera 3035, the standard monochrome handset, can barely manage two full-screen 1-bit blits per second. The Sharp BREW color handset is also rather slow; however, Qualcomm continually releases revisions of BREW with faster graphics performance.

It's Qualcomm's distribution and billing system, though, that may push BREW over the top. If Qualcomm can convince more carriers to adopt BREW technology (to date, 17 carriers have announced BREW support), then Qualcomm's centralized billing and certification process not only saves publishers money in testing fees, but makes collecting royalties from various carriers much easier.

## The Last Word

**B**ased purely on API features, BREW has more options, and the low-level C/C++ access to the phone's features is a more familiar environment to the seasoned game programmer. Additionally, Qualcomm continues to update BREW with new SDK and tool releases at a faster pace than J2ME spec revisions.

As a development environment, J2ME is far easier to get started with and much simpler to implement than BREW, but J2ME has been around far longer than BREW. This year will be critical for BREW — if Qualcomm can sign more carriers, produce more powerful handsets, and refine the toolset, it will be a major rival to Sun's early lead in mobile application development platforms. ☛

### J2ME

Sun Microsystems  
Palo Alto, Calif.  
(650) 960-1300  
<http://java.sun.com/j2me>

### BREW

Qualcomm  
San Diego, Calif.  
(858) 587-1121  
[www.qualcomm.com/brew](http://www.qualcomm.com/brew)

## Fresh Games, Taylor-Made

**C**hris Taylor has been successfully exploiting innovative technologies to fuel solid game designs for well over a decade.

Starting out with sports titles at Distinctive Software (now EA Canada), then later moving on to Cavedog and spearheading the seminal 3D RTS *TOTAL ANNIHILATION*, he founded Gas Powered Games in 1998. With the grueling task of starting up a new company behind him, Taylor was busy putting the final touches on GPG's debut, *DUNGEON SIEGE*, when *Game Developer* caught up with him.

**Game Developer.** Your design work has spanned a lot of different genres, from sports to RTS, and now to fantasy RPG. What design principles have you discovered and applied that you've found common to all your games?

**Chris Taylor.** Surprisingly I manage to reuse most of what I learn, even lessons from the first game I ever worked on.

For example, in *HARDBALL 2*, all the baseball team data was stored in data files, and the players could create their own teams. This sort of game architecture where we data-drive the game engine is still how we do things today, whether it's *TOTAL ANNIHILATION* or *DUNGEON SIEGE*. The key is to allow the database to grow over time as new content is added, with no preset limits.

On the gameplay side of the equation, the interface has always been one of the fundamental design elements. With each successive game that I work on, I endeavor to create an interface that is simple and easy to use and that utilizes the latest ideas and popular interface paradigms.

**GD.** Where do you seek out interface inspirations?

**CT.** Other than the obvious method of looking at as many games as we possibly can and blending all the good ideas together, we basically do R&D and brainstorm new ideas. We test these new ideas in a special usability session where we watch exactly how people interact with the game. It's especially important that they have never seen it before, or the results are



Gas Powered Games' Chris Taylor.

compromised. So, it's just a lot of hard work in the end.

**GD.** *DUNGEON SIEGE* aims to blend traditional RPG gameplay with more action-oriented elements. Do you see games heading in the direction of more or fewer distinct genres in the future?

**CT.** I think we are likely to see fewer distinct RPG genres, as even now it seems to be breaking out into two camps. One flavor of the genre is combat and action oriented, whereas the other seems to be heavily story driven with lots of very deep dialogue trees. Having said that, the genre can still surprise us and prove that ultimately there is no way to know where it's all heading.

**GD.** What one piece of advice would you give developers who want to start their own studios?

**CT.** My one piece of advice would be to find the best people, and this means people that are not only talented but who share the same vision. If you hire someone because you are in a hurry and they

turn out to be the wrong person, it could kill the whole project and company. You have to take chances, but don't risk it all by rushing on critical decisions when it comes to team building. This is easily the single biggest piece of advice I could share.

**GD.** How do you suss out potential candidates to determine whether they'd be a beneficial or detrimental addition to a team?

**CT.** Hiring is a critical skill that anyone who builds teams needs to have. It's not a skill that can be learned in 10 minutes, it comes from years of experience, and making a lot of bad hiring decisions and a lot of good ones. There is no substitute for learning this. I'd advise people to interview 10 people for every person you hire, if possible. And checking references is critical, no matter how much you like them. Also, have different people interview the candidate on different days, and don't let the interviewers talk or compare notes with each other until you have debriefed them. People should also consider putting employees on a short contract first, then hiring them full-time if they work out. But perhaps the most important rule is that if you have a bad feeling in your stomach, trust it — it's probably right. *✍*



# Packing Integers

**M**aybe you're writing a console game and you want to fit your save-game data into as small a space as possible for storage on a memory card. Or perhaps you're writing a networked game and you want it to perform well over low-bandwidth connections. Or, if you're making a massively multiplayer game, maybe you want to save your company \$100,000 per month in server bandwidth costs. This month I'm going to start a series of articles about packing information from a 3D game into small spaces.

In his recent article "Distributing Object State for Networked Games Using Object Views" (*Game Developer*, March 2002), Rick Lambricht provided a high-level overview of game networking. Now I'll look at techniques that fill in the low-level side of that picture, discussing what you actually want to send over the wire.

I'll start this month by talking about integers. Later, I'll discuss real numbers, 3D vectors, constrained 3D vectors (like unit vectors), and rotations. Representing rotations in a size-efficient way is an interesting problem that turns out to be pretty deep.

## Compression

In future articles I'll be looking at probability-modeling compression schemes such as Huffman and arithmetic coding, but I'll start with simpler methods. General-purpose compressors have surprisingly limited use when we

try to apply them to games. For example, the best Huffman coders use adaptive encoding techniques, which means the full history of data to be transmitted is used to generate statistics that help make the data small. To uncompress this data, you need to receive the entire stream in order, so that you can retrace the compressor's steps. But if you're writing a networked game, you want to use an unreliable delivery protocol like UDP so that gameplay doesn't stall due to line noise or other packet loss. In other words, you need to process incoming network messages with minimal dependencies between them. But adaptive encoding creates a dependency for each message upon each previous message, which requires reliable, sequential delivery by a system such as TCP, and thus causes poor game performance.

Compressors are not guaranteed to make your data smaller; in fact, they may make it bigger. When you look at the results over all possible input values, compressors don't compress at all; their output on average is as large, or larger, than their input. So if you absolutely must save your game state into a 64K block on a memory card, regardless of the positions of entities in the world, a compressor is not going to help you.

## A Task

**S**uppose we're a game developer, and we're going to make another *QUAKE*-style first-person shooter. We only have five kinds of entities in our universe, for which we have declared some enumerated constants:

```
PLAYER = 0
ROCKET = 1
AMMO_PACK = 2
HEALTH_PACK = 3
CRATE = 4
```

We want to save the types of a bunch of these entities into a file.

We could be extravagant and write a full CPU-register-sized value into the file for each entity type. These days, a CPU register is a 32-bit integer, so we are wasting a whole lot of space. Values from 0 to 4 will all fit into a byte, so we can write one byte per entity. But that still wastes a fair amount of space, as you can see from Table 1: we're leaving five bits per value completely unused.

## Packing Bits

**W**e can reduce our storage size by using only three bits per entity. For example, you can create some helper code that takes as arguments a value and how many bits that value occupies; then



**JONATHAN BLOW** | Jonathan ([jon@bolt-action.com](mailto:jon@bolt-action.com)) still doesn't understand the monkey jump.

**LISTING 1.** The `Bit_Packer` code that packs integers, their sizes indicated by number of bits, into an array of bytes.

```
typedef unsigned long u32;
typedef unsigned char byte;

struct Bit_Packer {
    int next_bit_to_write; // Initialized to 0
    byte buffer[BUFFER_SIZE]; // All initialized to 0
    ...
};

void Bit_Packer::pack(int num_bits_to_write, u32 value) {
    while (num_bits_to_write > 0) {
        u32 byte_index = (next_bit_to_write / 8);
        u32 bit_index = (next_bit_to_write % 8);

        u32 src_mask = (1 << (num_bits_to_write - 1));
        byte dest_mask = (1 << (7 - bit_index));

        if (value & src_mask) buffer[byte_index] |= dest_mask;
        next_bit_to_write++;
        num_bits_to_write--;
    }
}
```

it packs that value into a string of bytes. We pack in a three-bit number for each entity in the world, and when we're done, we ask it how many bytes of data it has collected (rounding up to the next byte boundary). Then we write those bytes out to disk.

Listing 1 shows some code that writes the input number into the destination buffer, one bit at a time. I wrote this listing for clarity and simplicity, not speed. A faster version, which operates on groups of bits simultaneously, is available in this month's source code on the *Game Developer* web site at [www.gdmag.com](http://www.gdmag.com). I've omitted some implementation concerns for brevity, like range-checking our position in the output buffer.

To successfully decode this buffer and read the values back in, you need to know how many values are stored in the buffer and how big they are, because we didn't store that information. Usually, in a real game situation, we have some set number of fields for an entity for which we know the sizes of the data (its type,

position, orientation, and so on). In cases where not everything is predetermined, we will pack some additional data at the beginning of the buffer that tells us what we need to decode correctly.

To read back the values, we want a routine that grabs bits out of a buffer. I won't show the listing here, because the idea is very similar to our `Bit_Packer`; it's included in this month's sample code, along with some tests that pack and unpack sequences of data.

At this point, just by using the `Bit_Packer`, you have a 90 percent solution for storing integers, in terms of fitting them within a guaranteed small space. Many people stop here. But if we

really care about making things small, we can do better.

## Packing Values with Sub-Bit Precision

Looking back at Table 1, it ought to be obvious that we don't even need three whole bits to store an entity type. We only use the third bit once; if we had one less type of item, we could fit the values into two bits each (though that would mean making a shooter without crates, which would be absurd). As a corollary, while using three bits, we could have up to eight different item types. But since our imagination limits us to five types, for now we are just wasting almost a bit per type that we pack.

The `Bit_Packer` is a very programmer-oriented solution to the value-packing problem. In the past, we as game developers have had to do a lot of low-level stuff to make our games perform well. We've taught ourselves to think about numbers in terms of bits and bytes, and value packing looks like a problem that is well served by that kind of thought. But in fact, restricting ourselves to bit boundaries is problematic — it's causing us to waste that extra space.

We can get that space back if we stop thinking about bits and instead consider what is necessary to encode and decode an integer value. Suppose we want to pack two entity types together. Picture a 5×5 grid of squares, where the x-coordinate of each square in the grid indicates the type of the first entity, and the y-coordinate indicates the type of the second entity. Every combination of entity types maps to some square in this grid; since there are 25 (5×5) different squares, we can store both entity types in five bits

**TABLE 1.** The numeric representations of the different elements in our game.

Entity	TypeValue (Base 10)	Encoding (8 Bits)
PLAYER	0	00000000
ROCKET	1	00000001
AMMO_PACK	2	00000010
HEALTH_PACK	3	00000011
CRATE	4	00000100

**LISTING 2.** The `Multiplication_Packer` code that packs integers, their sizes indicated by maximum value, into a larger integer.

```
struct Multiplication_Packer {
    u32 accumulator; // Initialized to 0
    void pack(u32 limit, u32 value);
    u32 unpack(u32 limit);
};

void Multiplication_Packer::pack(u32 limit, u32 value) {
    accumulator = (limit * accumulator) + value;
}

u32 Multiplication_Packer::unpack(u32 limit) {
    u32 quotient = accumulator / limit;
    u32 remainder = accumulator % limit;
    accumulator = quotient;
    return remainder;
}
```

with room to spare. Before, at three bits per entity type, we would have required six bits.

Any programmer who's ever addressed a screen of pixels knows how to generate a single integer index for this grid: it's  $y * 5 + x$ , where  $x$  and  $y$  range from 0 to 4. To decode this integer, which we'll call  $n$ , we just reverse this process:  $x = n \% 5$ ;  $y = n / 5$ .

This process scales to any number of dimensions and for any range of integer values in each dimension (as a quick exercise, visualize the three-dimensional case). Listing 2 shows code that packs an arbitrary number of values into a 32-bit integer. In my own code, I use the `Multiplication_Packer` to cram as many values together as I can without exceeding the packer's 32-bit limit; then I pass the results to the `Bit_Packer`, which packs it into a long buffer of bytes. This still induces a little bit of waste, but I figure that if I can reduce my memory wastage to a fraction of a bit in every 30, then I'm doing just fine. Extra sweating doesn't seem too worthwhile to me at that point. But if you feel that you must absolutely eliminate all possible memory waste, you can write a version of the `Multiplication_Packer` that automatically stores its results into a byte array when it gets full enough.

The `Multiplication_Packer` is, at its heart, an arithmetic coder without statistical modeling. So understanding this small set of functions may be a good introduction to more sophisticated compression.

By way of quick comparison, packing together 10 of our entity types using the `Bit_Packer` would have required 30 bits; using the `Multiplication_Packer`, we need only 24 bits, a savings of 20 percent. (People who use a byte per value will require a whopping 80 bits.)

## Something to Notice

It's important to see that the `Byte_Packer` and the `Multiplication_Packer` are essentially doing the same thing to stick values together. The `Byte_Packer` uses shifting and a bitwise OR; these are special cases of multiplication and addition for when the range of input values is a power of two. The `Multiplication_Packer` is simpler, more general, and in some sense more pure. But, as game programmers, we tend to think of numbers in a computer as being stored in binary, so we tend to think of the `Bit_Packer` first. It's important not to be trapped in the binary-number mode of thought. Sure, the CPU gives us operations that twiddle individual bits very quickly, and we know that internally, that's how it likes to store things. But so

what? That thing held in the CPU register is a number, not a string of bits.


## Byte Order

Programmers are used to confronting byte-order (or endianness) issues when considering different platforms. But this month's sample code has no such issue; it will work correctly regardless of the endianness of the underlying CPU. This is because we are using mathematical operations to carve chunks off of numbers and store them as units no larger than a byte each. That array of bytes can be written by an Athlon and read back by a Sparcstation.

I have in the past seen bit-packing implementations that did not work this nicely. They would treat an input value (32 bits maximum) as a string of bits in memory and copy that string of bits into the destination buffer. But because the format of bits in memory is not invariant, they have extra work to do when the time comes to port the code (and that extra work can get pretty messy).

## Clarification and Erratum

In my March column ("Hacking Quaternions"), I discussed a method of compensating for the angular distortion induced by linearly interpolating quaternions. I chose to think of this as a multiplication-oriented task, deriving some  $f(t)$  that, when multiplied by the distorted  $t$ , eliminates the distortion. Strictly speaking, what the problem demands is a  $g(f(t))$  approach, not an  $f(t)t$  approach. But we could get away with thinking of this as a multiplication problem because we were using splines that had no constant coefficients, so we could factor out  $t$ .

This can lead to some confusion in implementation. Listing 1 of the article looks like it returns a  $t'$  that you can use for interpolation, but it actually returns  $t'/t$ . Also, Listing 2 is incorrect due to an unintentional deletion. Correct code for that article can be found at [www.gdmag.com](http://www.gdmag.com). 

# Textures: From Source to Screen

**A**s far as great visuals go, texturing is often half the battle. It's all very well modeling a superbly detailed Celtic warrior complete with authentically tasselled sporran and impressively bulky facial hair, but if the texturing is weak, he'll lose some of his menace. Faultless animation is all well and good, but if your character's textures make him look like he's made out of plastic that has been slightly melted, the overall effect will be compromised.

As it stands, even the most overenthusiastic claims from our leading hardware manufacturers as to the number of triangles per frame their system can juggle still mean that it's impossible (and certainly impractical) to create all detail in geometry. Textures are still the paint that gives life to the surfaces of our meticulously crafted gaming worlds.

Still, there is often something of a stigma attached to the role of texture creator in the art pipeline. It's almost as if the shift into three dimensions the game industry experienced several years ago put a premium on the ability of artists to model, relegating the pathetically two-dimensional task of texture creation to the lowliest of art monkeys. How could the arcane skill of making a texture possibly compare to the taxing, multidimensional conundrum that is creating a model?

It has often been my experience that the enormous power of the top 3D packages can allow a competent technician, well versed in the process of building geometry, to create objects and environments with a minimum of problems (characters are a different story). Producing truly effective textures, however, often requires the eye of a genuine artist.

While very few games use hand-painted textures, and most textures originate from a photographic source of one form



FIGURE 1 (left). Good lighting. FIGURE 2 (right). Bad lighting.

or another, the journey from photo to game is not simply one of cut and paste.

Like any discipline that claims to be art, there is no single formula that will produce the desired results every time without fail. Different projects have different requirements, and engines running on different systems have their own advantages and limitations, but there are some basic ideas that are fairly fundamental when creating textures. The best place to start is the beginning, and in the case of textures, this means looking at the source material.

## Traditional Photography

**E**xplaining how to take a good photograph is beyond the scope of this column as well as outside my area of expertise. There are, however, a variety of factors that you need to consider when taking photos specifically intended for texture creation.

**Lighting.** Living in England (home of a

perpetually gray sky) has finally paid off. It's true that there are downsides to being in a country where children stand in the street, point at the sky, and ask, "Mummy, what's that big light?" if the sun happens to appear briefly from behind the clouds, but when it comes to texture sourcing, it's a blessing in disguise. Wherever there's strong sunlight, there are harsh shadows, and this contrast usually makes for a poor source photo.

Even without considering bump mapping, textures need to be relatively low on obvious light-source information. Severe highlights or hard shadows will make the texture less likely to fit within a game, unless you match the local lighting to that which is in the texture. With multiple light sources, or any kind of dynamic lighting, this is very difficult. Figures 1 and 2 show examples of good and bad lighting, respectively.

Gentle, diffuse lighting works much better. This kind of light will generally still have a direction (if nothing else, it will be



**HAYDEN DUVAL** | Hayden started work in 1987, creating airbrushed artwork for the games industry. Over the next eight years, Hayden continued as a freelance artist and lectured in psychology at Perth College in Scotland. Hayden now lives in Bristol, England, with his wife, Leah, and their four children, where he is lead artist at Confounding Factor.



FIGURE 3 (left). Good color. FIGURE 4 (right). Bad color.



coming from above), but this directional information will be subtle enough so as not to interfere with your lighting scheme. On the other hand, climate control isn't generally something that any of us can lay claim to.

**Focus.** It is useful to be aware of the elements within a photo that will be in sharp focus. If a surface is rounded, or if you are photographing extreme detail where the surface has a lot of variation in it, only some of the photo will be in sharp focus. This can restrict the areas that are useful for texture creation. Depth-of-field adjustments can increase the range of features that will be in focus, but choosing the right positioning for the shot ensures that the part of the photo you are most interested in will be evenly focused. You can compensate for uneven focus to a small extent, but it's always better to do as much of the work at the source as possible.

**Color.** Taking photos at either sunrise or sunset might well produce dramatic images with fantastic skies, but these are times to avoid when you are collecting source material for textures. Due to the low angle of the sun, the shadows are likely to be long, and with ambient light levels low, everything will also be less well lit. The reds and oranges from the light at these times of day can also interfere with the quality of the image.

Color manipulation is possible at a later stage, but color information can be permanently compromised if the overall lighting has a strong enough hue. The same is true of most artificial light, which is generally biased toward the yellow end of the

spectrum. Figures 3 and 4 show examples of good and bad color, respectively.

**Film speed.** Use the fastest film possible that will perform adequately when you take conditions into account. A higher-speed film has a finer grain and will hold up better to high-resolution scans.

**Flash.** Using a regular flash, as opposed to a more complex, umbrella-diffuser kind of rig (which won't exactly fit into your jacket pocket) can result in a washed-out photo. Even if this is not

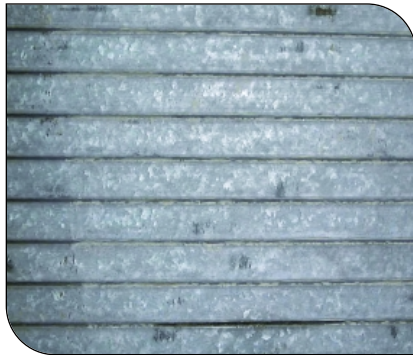


FIGURE 5. A radial pattern of brightness characteristic of flash photography.

the case, a flash produces irregular shadows, as the main light source is from a well-defined point. A flash is also prone to distributing brightness in a radial pattern, darkening towards the edges. Figure 5 shows an example of what can happen to an image if you use a flash.

**Perspective.** Just about any surface has noticeable features that define how it looks. Whether it's wood, rock, or elephant hide, these features appear

larger when they are close to the camera, producing perspective scaling as the surface recedes.

When you try to create a texture to mimic this surface, you need to keep in mind that the game world is in three dimensions, and the surface itself recedes, just as in real life. Creating such perspective is most challenging when photographing surfaces that have some kind of repetitive parallel feature, such as bricks or a metal grid. Figures 6 and 7 show good and bad perspective, respectively.

Thus, eliminating as much perspective as possible from within a photo will make it much more friendly when it comes to using it to produce textures. The best way to do this is to attempt to have the surface being photographed as square as possible to the camera. This is not always possible, and if the texture is on the ground, chances are your feet will get in the way.

## Digital Photography

There is little doubt that the advent of digital photography, which has taken out the slow and expensive chemical processes and removed the need to scan, has helped make the accumulation of photographic source material for textures much quicker and easier. In addition, the fact that most digital cameras allow you to preview your images (albeit on a fairly small screen), gives you a limited amount of filtering at this early stage.

Aside from film speed, the same considerations apply for digital as for traditional photography, although only the most expensive digital cameras let you choose which type of lens to use, so you won't likely be able to make depth-of-field adjustments.

Digital photography does, however, have a few considerations:

**Size.** Most digital cameras allow a choice of image sizes, and (like most electronic devices) performance seems to increase at an almost daily rate. The maximum size of an image (in pixels) varies from camera to camera, but selecting the largest size available is usually the best idea, as this allows small

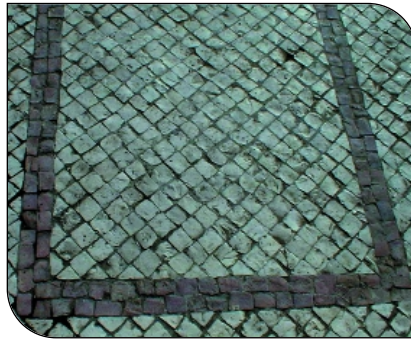


FIGURE 6 (left). Good perspective. FIGURE 7 (right). Bad perspective.

areas of an image to be used while maintaining an acceptably high resolution. All these pixels use up a lot of space in the camera's memory, which brings up the next point.

**Compression.** Again, this depends on the specific camera, but several compression options exist on most models. It's important not to compromise the cleanliness of each image with artifacts associated with high compression levels. At the same time, try to maximize the number of images that you can store. Buying a massive amount of memory can help, as does deleting unwanted images rather than letting them build up. It's also useful to carry a laptop around with you, downloading as you go.

## Scanned Images

**B**efore digital photography was an affordable option, the scanner was a texture artist's best friend, and it still forms an important part of the texture sourcing process.

Scanning actual photos is usually the best option, as the quality of the printed surface is quite high, and you can set a scan to an enormous resolution. Often, due to lighting, color, or feature changes across a source image, only part of an image will be useful for generating a texture. When working with a large scanned image, the artist can pick out small elements from within the image and still retain a good level of crisp detail.

A very large scan produces a very large file size, but with today's enormous hard-drives, storage capacity is

hardly an issue. It is, however, worth considering the speed at which the image can be manipulated when it's extremely large. If you want to scan a photo and use the whole image instead of just a small part, you can reduce the resolution of the scan to a sensible size to make it faster to use.

Printed source material is good because of the vast range of books and magazines that are available. Scanning from a book or magazine, however, is limited by the quality of the print. Magazines are the worst, as their print process is all about being cheap and cheerful. Often, more expensive books have better-quality images, but this isn't always the case.

This reduction in the quality of the images means that the size of the scan is limited, as enlarging a particular area too much exposes the print process and makes it unusable.

## Image Libraries

**T**here are many companies that provide images from within their own vast collections to anyone who wishes to use them. These kinds of images often end up in corporate presentations, or on the back of some idiotic junk mail, but they are generally of very high quality and cover just about any subject you care to mention.

There is a cost involved (some are subscription based, some are priced per image), which quite often rules these services out. Nevertheless, they are worth considering as an option.

## Procedural Textures

**P**rocedural textures, usually generated through a process of parameter adjustment and combination, are better at simulating some surfaces than others. There are ranges of packages that can generate them. Often, you can use procedural textures in conjunction with photographic images to produce interesting results, but unless their mathematically generated nature fits the style of your game, they may not be that useful on their own.

## Rendering Textures

**I**n some cases, you can render textures using a model that has already been textured. This double-texturing method allows you to convert details from what may be a high-polygon model from geometry to texture. More importantly, it can allow you to use complex lighting solutions by having their results render to texture.

Even though the latest hardware and newest engines allow for an impressive range of lighting effects, processes such as the currently popular global illumination are still rooted firmly in the world of high-end graphics. Using software that allows light information to be "baked" into textures is one way to bring visual complexity to a game.

## A Solid Foundation

**C**hoosing source material carefully and generating usable images on which to base your textures are vitally important. In most cases, the finished texture will only be as good as the image from which it was generated. The amount of photos you take that turn out to be useless will diminish as you automatically choose the kind of surface that will work. And when you find that you can't walk down the street without making a mental note to return later to get a couple of good shots of the crumbling brick wall you've just passed, you know that you've got the texture bug. It could be time to take a vacation (just don't forget your camera). 📷

# The 400 Project Continued: Suspending Disbelief

*This month I introduce a rule from Mark Barrett, a freelance designer and storyteller who has led a roundtable at the GDC the past three years called “Creating Emotional Involvement in Interactive Entertainment.” He can be reached at mark@prairiearts.com.*

## The Rule:

### Protect the player’s suspension of disbelief.

In any game that uses or relies on narrative content, players should be encouraged to suspend their disbelief and become imaginatively involved the experience. Once so engaged, players should be protected from other elements which might shatter this imaginative experience.

**The Rule’s domain.** This rule applies to any game involving narrative content, including elements that only provide an imaginative context for play. This means any game that includes characters, locations (real or imagined), story elements, plot, and so on. Only purely rational games along the lines of tic-tac-toe or chess are excluded, provided they are depicted in minimalist form. (BATTLE CHESS, for example, would still be subject to this rule.)

**Rules that it trumps.** This rule is a constant in game design. It does not trump other rules, but exists alongside them.

**Rules that it is trumped by.** This rule is a constant in game design. It is not subordinate to other rules, but exists alongside them.

**Examples from games.** Mark did not supply examples, so I’ll step in here. I’m indebted to Mark for setting me straight years ago on the concept of suspension of disbelief. I had been familiar with the concept from science fiction and fantasy writing, and thought it primarily applied to

keeping your fantasy elements from being so jarring or unbelievable that it reminded readers that they are reading a fantasy. But Mark pointed out to me that it’s a more fundamental principle: suspension of disbelief is necessary to let readers forget that they are reading or hearing a story instead of living it firsthand.

With all of our industry’s focus on technology and high-tech applications like virtual reality, it’s easy to forget that our species’ first VR was invented tens of thousands — perhaps even millions — of years ago. It’s storytelling, or narrative.

We learn to listen to stories from early childhood, and the good ones suspend our disbelief, letting us enter into the world of the story and identify with the characters so deeply that we jump when they are attacked, smile when they are delighted, and cry when they are hurt. We must suspend our natural disbelief that “this isn’t happening to me” or, even more fundamentally, that “this isn’t real.” As media such as epic poetry or theater, then technologies like writing, film, and computers, created new ways to deliver narrative, storytellers have learned ways to use each new medium to tell stories effectively while suspending disbelief.

In games there are many examples of how to maintain suspension of disbelief, and sadly, many grating ones of how to shatter it. One of the more egregious examples of failure in applying this rule comes from the mechanism, once prevalent in some adventure games, of killing

off players when they made even a small misstep, then intruding with a blatant reminder that they were only playing a game — would you like to reload a saved version? I heard of one young boy who refused to play such a game, claiming, “I’m afraid of the evil witch who shows up every time you die,” referring to an awkward intrusion of the designer, picture and all, into the game world. Although the basic mechanism of multiple lives and starting over again is deeply ingrained in many games, it has evolved into a much more elegant fade-out/fade-in, as in any of Shigeru Miyamoto’s recent games, or retreated further into the background, with the use of waypoints or the touching of special in-game icons to keep a player from ever having to reload a saved game at all. Many RPGs now let you recover your “body” in the game and retrieve most or all of your belongings, letting you stay within the (admittedly magical) game world.

To conclude this column, I’d like to clarify a misconception to which a reader alerted me. These rules of game design are not meant to analyze games and derive a rigid system of rules that you can use to churn out new titles. Rather, if applied with an appreciation of the informality, creativity, and flexibility inherent in the game industry, they’re intended as tools that a talented designer can use creatively to break free of a rut, or to improve or expand a design. 🎮



**NOAH FALSTEIN** | *Noah is a 22-year veteran of the game industry. You can find a list of his credits and other information at [www.theinspiracy.com](http://www.theinspiracy.com). If you’re an experienced game designer interested in contributing to The 400 Project, please e-mail Noah at [noah@theinspiracy.com](mailto:noah@theinspiracy.com) (include your game design background) for more information about how to submit rules.*

# Voice Development: Becoming Lord of the Files

If you've never experienced hassles managing a massive voice file set, don't read any further. If, however, you've experienced frustrating, time-consuming, and costly bumps along the way, then read on.

Voice development happens in three phases: preproduction, production, and postproduction. Although the proverbial rubber hits the road when actors are directed and recorded in the production phase, the first and last phases are paramount, as so much of the process hinges on designing a good plan in preproduction and sticking to it in postproduction.

Apart from a good version-control plan, an essential part of preproduction planning is a good file-naming convention. Every file should be named uniquely to express the character and location, and also contain a numerical ID. Also, leave a free space for the appendage of an identifying character that may be necessary in post. A snippet of the verbal content may also be included. For example, if Theseus has lines in the Labyrinth, a good naming convention would be something like "TH\_LAB\_001.WAV," or optionally "TH\_LAB\_Heyyou\_001.WAV."

Being able to isolate a file subset easily is critical — especially to the audio developer who must process all the files. Different character subsets often have problems unique to their original recordings, and all they need is a single batch process such as EQ or compression. If the sound designer can't get a handle on a file subset because of a poor file-naming convention, it's a major drain on quality, efficiency, and morale. If all of Theseus's lines are bass-heavy, the audio pro can isolate all Theseus's lines by doing a find for "TH\*.WAV" and performing a batch EQ. Similarly, to apply special processing such as reverb to every line in the Labyrinth, one can isolate "??\_LAB\*.WAV" and run a batch.

A good database is another key part of the plan. If you're dealing with a couple hundred lines, it's optional; when you're talking about thousands, it's necessary. A good database tracks file name, character, location, level/mission, verbal content, development status, localization information, and processing notes, and has various check boxes or radio buttons for easy finds. A database is only as useful as it's adhered to, though. When you're dealing with localization issues, for example, a strictly accurate correlation with the file set is a must.

A word on hard-processing voice files with special effects: In short, go ahead and process the actual audio files for character enhancements, such as pitch-shifting the evil overlord character, but do everything possible to accomplish environmental processing such as reverb in real time. This is not only more efficient, but it also avoids the hassle of isolating the precisely correct file subset that needs the process. Inevitably, lines are missed, accidentally included, or have to be duplicated because they occur in multiple locations with different sonic properties.

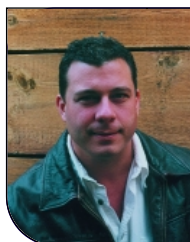
Generally, the two greatest pitfalls occur when non-audio personnel either change file names autonomously or edit verbal content themselves. Both of these practices invariably cause problems, even nightmares for those responsible for the database and master file set as well as those handling localization. Never change file names unless absolutely necessary, and in those rare cases, consult the appropriate staff. Rather than edit

audio files by yourself, request it through the audio pros. A good audio provider will turn around a better edit quickly and maintain all supportive information accordingly.

Khris Brown of KBA Voice Production, a voice production studio for interactive media, believes that one of the best ways to ensure a successful collaboration with the development team is good preplanning. "Preproduction and guideline planning remove contention from the picture," says Khris. "Your energy can then be spent perfecting the creative vision as opposed to chasing down technical inconsistencies."

A clear set of rules and procedures doesn't mean that flexibility or creativity should be restricted. Needing ongoing changes or citing new needs such as pickups is indicative of a designer who cares about his or her project, and a good audio developer will be happy to support those needs. "Characters, puzzles, and dialogue are continually refined throughout the production period," says Khris. "I always build pickups into a budget to ensure designers have the freedom to make adjustments without expensive surprises."

Managing thousands of voice files is a difficult task to begin with, but the last thing it needs is to be further complicated by folks departing from a solid preproduction plan, or worse yet, not even having a decent plan in place at all. When the process works well, everyone can stay more focused on the fun and creative aspects of the project and, as a result, ship a better game. 🎮



**CLINT BAJAKIAN** | *Clint is a composer and sound designer with 11 years of game industry experience. A sound design supervisor at LucasArts for nine years, he now co-manages The Sound Department ([www.thesounddepartment.com](http://www.thesounddepartment.com)), providing original music, sound design, and voiceover production services. He is vice-president of the Game Audio Network Guild (GANG), on the steering committee of the IA-SIG, and a member of NARAS, IGDA, and BBQ.*



# Understanding Animation Blending

**A**nimation blending is one of the latest buzzwords to join the bullet-pointed ranks of real-time engine feature lists everywhere. But even as programmers are busy implementing the technology, many artists and producers are still wondering what benefits and opportunities it offers. For those of you who don't have to worry about the implementation side, this article will focus first on explaining what animation blending can and can't do, and then explain how art staffs and game designers can make best use of it.

Current games excel at animation playback, but no matter how smoothly the animations unfold, the unerring repetition of every cycle and action soon renders even the most satisfying animations stale and lifeless. Picture a typical game crowd scene. The animations may be quite convincing at first glance, but upon closer examination we find that every pedestrian walks down the street with an unvarying stride, eyes fixed straight ahead, oblivious to the outside world. After a few minutes of watching we'll have identified and memorized every one of the cycles being played. However well animated the characters may be, monotony — compounded by our remarkable sensitivity to repeating patterns — soon drains the life out of them and leaves us a street full of walking zombies.

Imagine, though, how much more satisfying this street scene would be if some of the folks on the street were admiring the scenery as they walked; others might stuff their hands in their pockets, nod to passersby, or look at their watches. More subtly, but most importantly, each person's walk cycle could be unique, differing subtly from every other's in ways that suggest different personalities and different moods.

Such a scene would be far more immersive than our avenue of the living dead, and it could also tell us far more about the ongoing state of the game and its inhabitants. Until recently, though, this kind of variety could only be produced with a prohibitive investment in individual animations and detailed planning to make sure that sequences dovetailed with each other.

Animation blending can bring our zombie pedestrians back to life with a much more reasonable outlay of time and

effort. The key advance blending offers is that many animations are played concurrently, with variable influence on the final behavior of the character. In essence, blending also allows the game to create new animations on the fly as circumstances change. Animations can be modified; for example, a walk might become a "tired" walk. They can be combined, so a walk can become a "walk and check watch." And they can be sequenced with greater flexibility and fewer snaps and pops.

Blending widens a character's repertoire of behaviors, but more importantly, it helps preserve the essence of animation — the illusion, spontaneity, and variety of life. And, fortunately for developers, they can derive this breadth of new content from a manageable library of basic sequences, instead of making a vast investment in custom art assets.

## Blending Basics

**S**o what is animation blending? In the simplest terms, animation blending is still just animation.

Any animator knows that animations consist of keyframes — significant points in time and space — and "twens," the stretches between keyframes where the computer supplies interpolated positions. Animation blending works exactly the same way, except that where an ordinary animation tweens between sequential keyframes, a blended animation interpolates between whole animations.

A visual analogy makes this relationship easier to understand. Imagine an old-fashioned cel animation laid out frame by frame like a filmstrip. Each successive cel in the strip is a "tween" — an "in-between," or an interpolation — between the first and last key. A blended animation would look like a grid formed by two parallel strips of cels forming two sides of a square. Each cel in the grid is an interpolation between the corresponding cels in two original films. Moving forward in time would be traversing along the line of original strips, and blending from one animation to the other would be moving sideways across the grid of tweened cels (Figure 1). A simple transition would move diagonally (forward in time and across the grid from one animation to another), while subtle layering effect could be achieved by traveling ahead in time at some slightly varying point in the space between the source animations (Figure 2).

An animation succeeds or fails on the strength of its poses, that is to say, its keyframes. Indeed, early animators dubbed

---

**STEVE THEODORE** | *Steve is an animator and character designer at Valve Software, where he works on online titles such as COUNTER-STRIKE and the upcoming TEAM FORTRESS 2. He can be reached at [stevet@valvesoftware.com](mailto:stevet@valvesoftware.com).*

them “key” frames because they were the critical poses. Keyframes were drawn by the senior animators, whereas tweens were added later by interns and journeymen who needed only to interpolate between the important poses. Animation blending treats entire animations as keyframes and produces new sequences of tweens.

Thus, an important rule to keep in mind is that, like the tweens in an ordinary animation, blended animations contain no new information that was not present in their sources. Certain types of blends, therefore, aren't likely to work out well: blending between a standing animation and a swimming animation isn't going to spontaneously produce a diving animation, since neither animation contains a diving pose. We'll return to the importance of this rule again, because it drives several important aspects of the blended animation process.

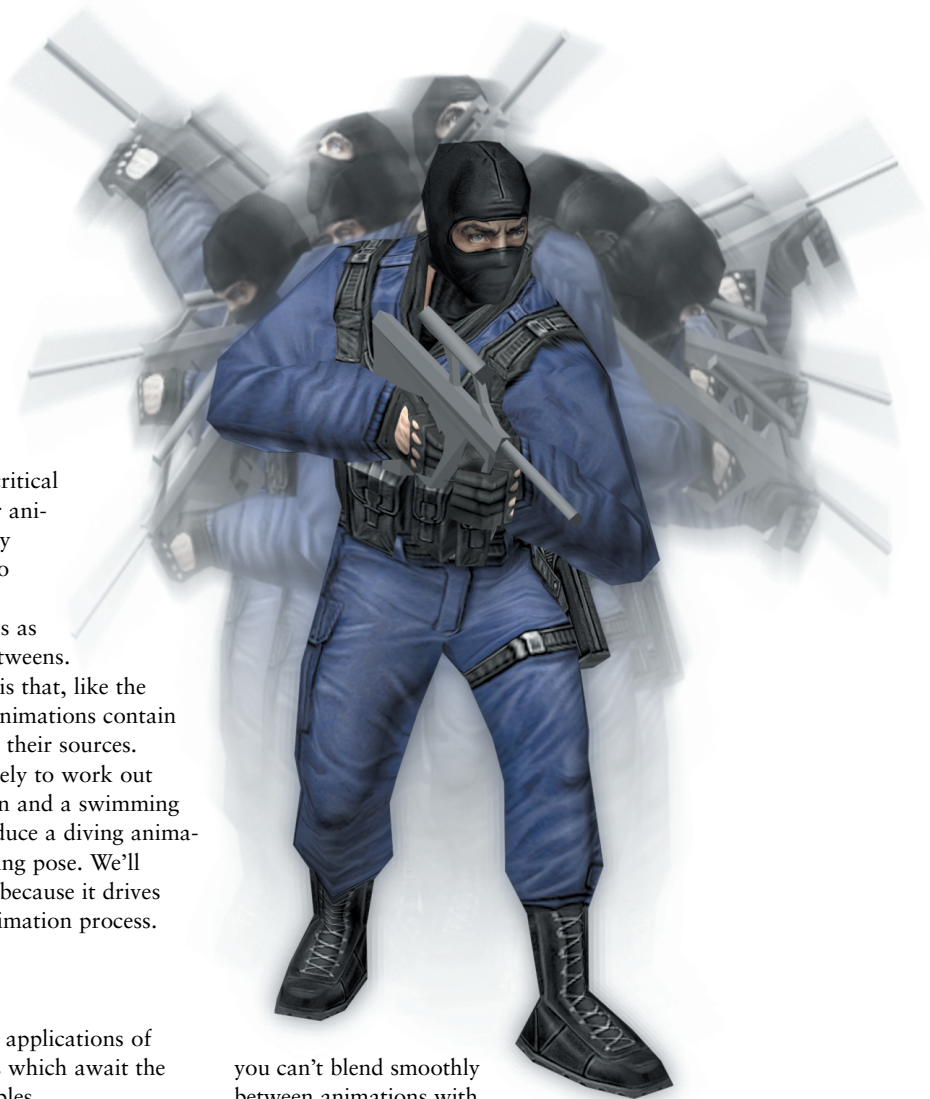
## Some Ground Rules

**T**he best way to understand the practical applications of blending, as well as a few of the pitfalls which await the uninitiated, is to look at some simple examples.

Let's begin with the simplest animation imaginable: a ball bouncing up and down in place. We'll add a second sequence with the ball just resting on the ground. What happens when we blend between them? As we increase the influence of the still sequence, the ball bounces less and less (Figure 3). The interesting lesson here is what happens to the timing of the animation: In this case, nothing. The ball bounces lower but the timing of the bounces doesn't change. In other words, blending affects the amplitude of an animation but not its frequency, which is another important rule to keep in mind.

This rule means that blending between animations with mismatched frequencies can be dicey. If, for example, you blend between a ball bouncing three times per second and a ball bouncing four times per second, the frequency of the bounces does not increase smoothly; instead you see little hiccups as the two patterns overlay each other.

There are in fact many occasions where you may want to break up the underlying frequencies of the source clips, particularly in idles or other cycles that need a little “noise” to add variety. Nevertheless, understanding the interaction between timings is the first key to making effective use of blends. Put simply,



you can't blend smoothly between animations with mismatched cycles. It's possible to build a system that would do this by time-warping the source animations to make them cycle, but few engines currently on the market offer this option.

## Verbs and Adverbs

**A**nyone who has experienced game animation is familiar with idle animations and knows how few repetitions it requires to turn an exquisite little character study into a meaningless cipher. Instead of creating longer idles, or randomly choosing from a list of different animations, we can use blending to create a less repetitious and more flexible set of behaviors.

For a simple example, consider two simple animations: one in which the character has his weight on the left leg and is looking off in to the middle distance, and another in which he has his weight on his right foot and is looking down. Blending between them will cause the character to appear to shift both his weight and his attention. Simply blending across from one animation to the other effectively creates a third animation, in the form of the weight shift. This is already an advance; but more importantly, we can create ongoing, subtle variations in the

stance by blending only fractionally. We can randomly blend in order to add variety to the cycle, or we can use the blend to communicate information about the character's mood or status.

It's important to note that we're not restricted to just two blends. We can create a huge range of variations for very little extra effort in authoring by blending between only a few sources. When working with sets of animations like this it's often a good idea to think of one animation as the "verb" and the others as "adverbs" — modifications on the basic action. In this example we could have not only a default idle, but also a nervous idle, an alert idle, a depressed idle, and so on. We could then vary the mood of the idling character by blending in various small amounts of the various "adverb" animations. (See Bodenheimer, Rose, and Cohen's paper in *For More Information* for more on verbs and adverbs.)

Grammatically, we also use adverbs to indicate directional information. An important application for blending is enabling characters to look at the player, aim guns, and so forth. Many games do this now by directly controlling character's heads or torsos — for example, this was how Valve made the scientists and security guards look at the player in *HALF-LIFE*. Blending improves on direct bone control by giving a more fluid and natural orientation to the whole character, since the blend can affect the entire body at once. Well-chosen poses can make even the simple act of aiming a gun a real study in character. Moreover, the control of the character remains in the hands of the animator, where it can be critiqued and tweaked without having to recompile any code.

## Understanding Blend Space

As the number of source animations increases (particularly in applications such as directional blends), it becomes increasingly important to think clearly about how the sources relate to each other. A useful tool for visualizing these relationships is what I call a blend space, a graphic representation of the range of behavior a given blend is meant to describe.

A blend space can be thought of as a simple graph whose axes are the parameters or variables that control the blend. A look animation might have a "left-right" and an "up-down" axis, for example. Each source animation is fixed at a particular location in the grid formed by the relevant axes of the blend space. Thus the "look left" and "look right" animations, for example, would be at opposite ends of the "left-right" axis.

The axes are not always spatial, however. They can be used to represent whatever abstractions are important to the character's animations — not unlike the alignment diagrams found at the end of old *Dungeons & Dragons* manuals. An idling character might have a "confident-nervous" axis, an "alert-relaxed" axis, and so on. A blend can have three or more of these abstract dimensions. The character's state — how "left-right," "happy-sad," and so on — will be a point somewhere on the grid that moves as the state changes. The relative influence of source ani-

mations on the blended result can be seen in their distance from the character's "parameter point."

Blend spaces are important for understanding the rule that blending doesn't create new information; they help to visualize what conditions a given set of sources will, or will not, cover. The blend's range is represented by a hull drawn around the sources' points on the grid of the blend space. If, for example, you attempted to make a look blend using only four sources — up, down, left, and right — the blend space diagram would warn you that the "corners" of your blend space (look up and left at the same time, for example) were outside the area covered by your sources.

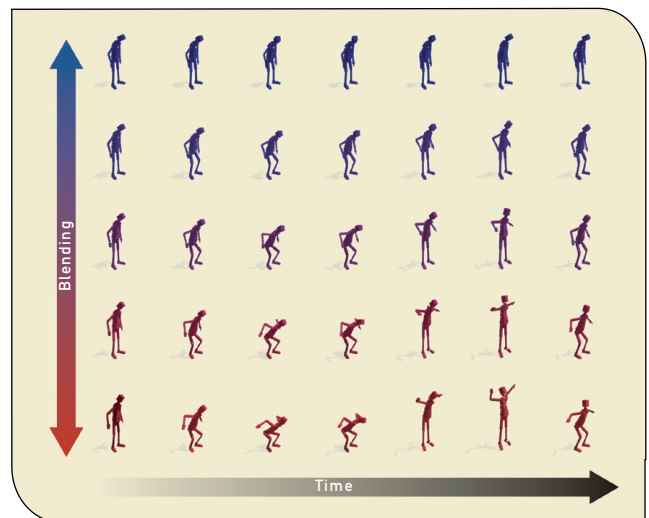


FIGURE 1. Blending two animations is just interpolating between the matching frames.

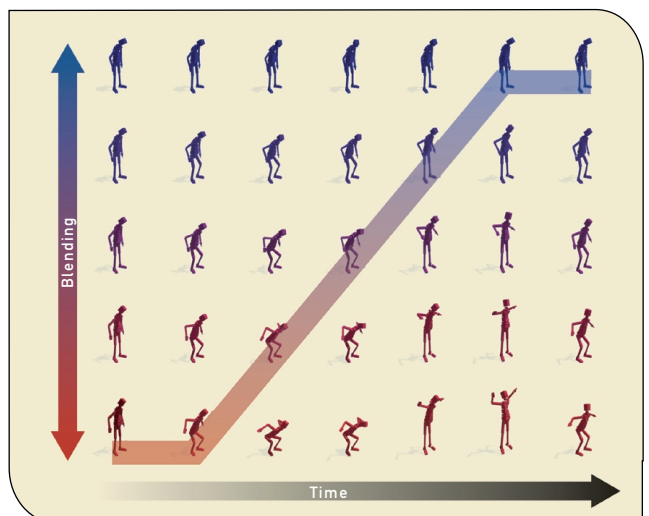


FIGURE 2. Blending from "happy" to "sad" while both animations play produces a smooth transition.

Blending between look up and look left would be looking 50 percent up and 50 percent left, as you can see in Figure 4. If you need to look all the way up and all the way left simultaneously, you will need another source animation. It's mathematically possible to fake a missing source by extrapolating based on a blended point on the hull, but the results are rarely satisfactory.

It's easy to assume that the best way to cover a blend space is with a regular grid of samples. However, the analogy between source animations and keyframes suggests that some useful effects can be achieved by the clever placement of sample animations in the blend space. Any feature in the behavior being blended that would deserve a keyframe in a traditional animation probably should be a source clip in the blend.

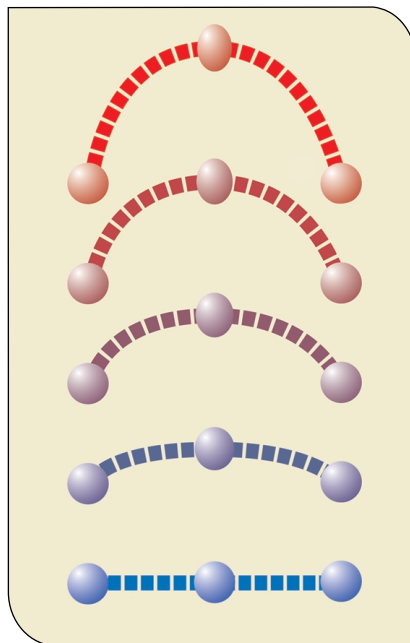
Consider a simple blend between a standing and a squatting pose. A straight blend between these two poses will play similarly to a deep knee bend, with the character's back remaining mostly vertical and his center of gravity moving almost straight downward. Most people, however, start a squat by leaning forward and pushing their hips back. If we add a source animation about one-third of the way from the standing pose to the squat with fairly straight legs and a lean, the resulting blend becomes much more natural and realistic.

## How Many Sources?

**Y**ou'll still want to use as few source clips as you can get away with. If you have two extreme poses that are perfectly symmetrical — say, a character who is looking exactly 45 degrees right in one sequence and 45 degrees left in another — you may be able to omit an intermediate pose and blend straight across. But when the sources are asymmetrical or the motions between them extreme, the tweened results are less predictable and the likelihood that you'll need additional source poses grows.

For example, a character who's aiming a rifle in the classic cross-body shooting stance will have to treat his right hand and left hand arcs differently (turning left involves mostly the upper body, whereas turning right requires turning the hips as well). It's unlikely that a straight blend of the rightward and leftward aims would produce a perfect straight-ahead pose, so in this case you probably need a third source for the blend.

The last but most important rule for placing samples is that (in the absence of specific code that says otherwise) blended animations are always forward kinematics animations. A limb



**FIGURE 3.** Blending between a bouncing ball (red) and a nonmoving ball (blue) produces a lower bounce but does not change the timing.

animated with IK will follow a straight line in space between two keyframes to generate tweens, whereas an FK animation between the same keys will typically travel along a compound curve defined by the changing rotations of all the joints involved in the motion.

Animators typically use IK for parts of the body that interact with the world: feet that stay planted on the ground, hands that grasp objects, and so on. FK, on the other hand, is better for motions with a pendulum-like arc, such as the relaxed swing of an arm during a walk or the tossing of a ponytail. Animators who prefer FK generally end up setting many more keyframes than IK animators do if they need exact control of positional data such as foot placement.

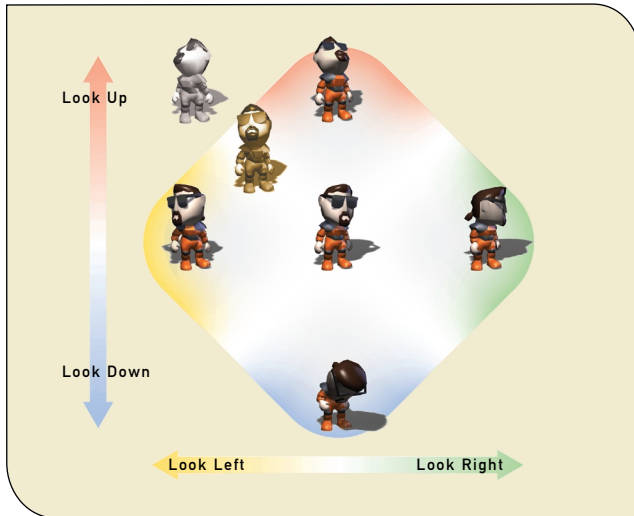
The difference between the types of tweens becomes more and more pronounced as the interval between keys gets larger. For small movements it may be unnoticeable to the untrained eye, but it's very obvious in larger motions. Since blending is generally done as FK, blended animations in which positional relationships are critical — those with locked feet, hands holding weapons, and the like — will show slippage during the blend (Figure 5).

Animations where these relationships are critical need more sources in their blend space to minimize the error. Any animator familiar with FK and IK keying will have a pretty good intuitive sense for what will happen and what trade-offs will be acceptable. The lack of IK blending is in many ways the biggest handicap in working with blends right now. The technology for doing real-time IK blending exists but is not very widespread. It will hopefully become standard in the near future.

Animations where these relationships are critical need more sources in their blend space to minimize the error. Any animator familiar with FK and IK keying will have a pretty good intuitive sense for what will happen and what trade-offs will be acceptable. The lack of IK blending is in many ways the biggest handicap in working with blends right now. The technology for doing real-time IK blending exists but is not very widespread. It will hopefully become standard in the near future.

## Masking Animations

**A**s we have already seen, blending gives us very powerful tools for manipulating characters. By far the most powerful is the ability to blend selectively on only some parts of the body. Localized replacement of animations is fairly simple from a technical standpoint. If regular blending can be likened to simple 2D-image blending, then replacing only part of a body is analogous to compositing with an alpha mask. The “mask” in this case targets only some bones in the skeleton for blending. By replacing only the transforms on one arm, we can let a walking character make a wave gesture, or throw a ball, without having to author complete “walk and wave” or “walk and throw” animations. Masking is equally useful for animating characters who have a large variety of equipment or weapons, while reusing basic cycles



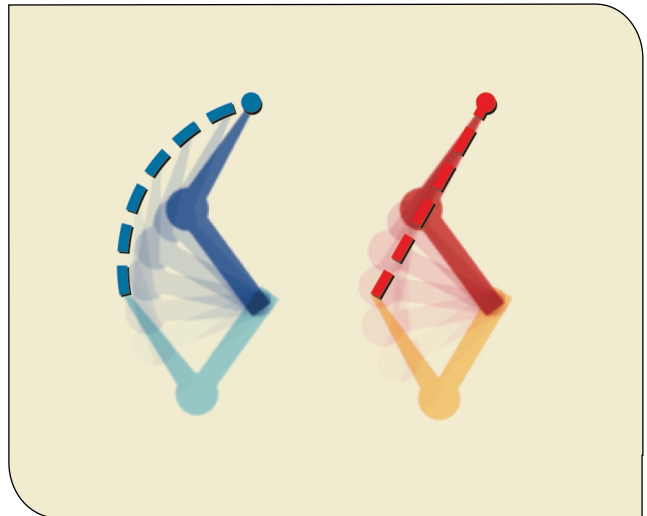
**FIGURE 4.** Blend space for a two-dimensional look blend. The interpolated position (orange) is inside the area covered by the sample; compare with the full "look up and left" pose in gray.

such as walks and runs.

We've already considered the common cases of looking and aiming blends. Shooting and running, however, can't be blended because of the rule presented previously about mismatched frequencies. (Even if the run and shoot cycles have the same duration, the firing can occur at any time.) With masked blends, however, the character's upper body can be replaced when the character shoots. The frequency of the moving legs won't interfere with the fire animation and — as long as the animations are properly planned — the two should work together regardless of the length of the animations involved, which weapon is used, and when the shot occurs.

Masked blends are fairly simple to implement in code, but they demand careful attention from animators and character designers. Most actions are part of a complex dynamic system involving almost the entire body, and the parts cannot be isolated from each other without a certain degree of artificiality and stiffness. For example, even an athlete or a soldier trying carefully to keep his or her upper body still while moving needs to use the entire torso to compensate for the movement of the hips; although this movement is fairly subtle, its absence is easily detectable.

Moreover, the arrangement of the skeletal hierarchy can seriously affect the resulting animation. In *TEAM FORTRESS CLASSIC*, we used masking to give the characters' upper bodies aiming and shooting animations. Unfortunately, our animation package treated the pelvis as the root of the skeleton. Thus, the pelvis transmitted all of the rotation of the hips to the model's upper body when we masked out the compensating animation on the torso. We were forced to choose between unnaturally flat hips and a "Frankenstein walk" in



**FIGURE 5.** FK (blue) and IK (red) will follow different paths between the same keys.

which the whole upper body jerked back and forth.

If you know you're going to have two body sections with unrelated animations, the point where the two halves meet should most likely become the root of your skeleton. If this is impossible, you can try to minimize the Frankenstein effect by differing the amount of blending (in traditional 2D compositing, this would be "feathering the mask") along the bones of the torso so that the compensating animation that keeps the torso from swinging is retained as much as possible. Even without the Frankenstein problem, starting your mask farther up the torso, rather than at the waist, helps preserve the original body dynamic and gives a more natural feeling to the result.

The ability to reuse basic cycles with multiple overlays is both an excellent way to add variety to characters and a very efficient tool for leveraging content. It is, however, rather tricky from an authoring standpoint for both aesthetic and technical reasons. Good planning and a lot of testing are the keys to getting the most out of masked blends.

## Creating Source Art

In general, game engines don't care how a character was animated. Ordinary keyframes, inverse kinematics, motion capture, or even dynamics simulations can all provide good starting points for blends. Unfortunately, in almost every case, the engines will know only about the transforms resulting from the various animation tools; unless the animation system is specifically designed to recognize and reproduce IK or other kinds of controls, the original driving mechanisms will be lost. Moreover, when blending begins, the blends will be taking place through FK regardless of how the original animations were created.

The real difficulty in authoring for a blended animation system is in visualizing and previewing the nearly infinite possible interactions between the various source animations. The current versions of the major 3D packages now include some form of nonlinear animation editor that can be used for prototyping blended sequences. Maya's Trax Editor, Lightwave's Motion Mixer, the layering system and Motion Flow module in Character Studio, and most recently the Mixer in Softimage XSI (perhaps the most powerful of the current crop) all allow animation blending, as do many motion capture processing packages. Most artists will probably have encountered the nonlinear animation functions in the unglamorous roles of asset management, content recycling, and retargeting animations from one character to another, but as the use of blending in games becomes more widespread, these tools are bound to become more popular (and I hope more capable) as laboratories for tinkering with blends.

For the time being, though, most of the nonlinear animation systems are designed around light-duty editing and performing simple transitions rather than real-time, free-form blending and masking. In fact, these systems are often too good — they preserve information such as IK relationships that will be lost in the game engine. Artists, therefore, must have a quick, low-overhead method of testing and previewing the blends. A good viewer application is an unbeatable investment for a team that wants to use blending; the availability of such a tool should be a key feature in deciding whether to license an existing animation system.

The real difference between working with a blended animation system and a traditional, playback-oriented system first becomes apparent in the planning stage. Higher granularity — storing more narrowly-targeted animations and gestures — is fundamental to effective use of blending. Unfortunately, higher granularity may inflate the initial asset list to alarming proportions, as multiple versions of base cycles, masked actions, and so forth are added up.

Fortunately, most components tend to be easier and cheaper to create than complete, unblended action animations. Full-body blends often consist of a single base animation and a number of modified derivatives (our verbs and adverbs). The derivative animations are much cheaper to build, particularly if the character rigs are well designed or tools such as Character Studio's Layers interface are available. Masked animations cut down on the number of combinations that need to be authored.

Masked blends are especially important to asset management, because they essentially offer the flexibility to trade quality for speed of authoring and asset reuse. A one-fire animation, for example, could be added onto walking, running, and standing animations. While it's sometimes painful to have to make such a trade-off, it's nevertheless an invaluable option available to the typical overstressed and understaffed art team.

## FOR MORE INFORMATION

Ashraf, Golam, and Kok Cheong Wong. "Constrained Framespace Interpolation." In *Computer Animation*, November 2001. pp. 61–72.

Bodenheimer, B., A. V. Shleyfman, and J. K. Hodgins. "The Effects of Noise on the Perception of Animated Human Running." Eurographics Workshop on Animation and Simulation, 1999.  
<http://citeseer.nj.nec.com/bodenheimer99effects.html>

Bodenheimer, B., C. Rose, and M. Cohen. "Verbs and Adverbs: Multidimensional Motion Interpolation Using Radial Basis Functions." *IEEE Computer Graphics and Applications* (Sept. 1998): pp. 32–40.

<http://citeseer.nj.nec.com/rose98verbs.html>  
see also <http://research.microsoft.com/graphics/hfap/>

Bruderlin, A., and L. Williams. "Motion Signal Processing." *Proceedings of Siggraph 95*. pp. 97–104.  
<http://citeseer.nj.nec.com/bruderlin95motion.html>

Grassia, F. Sebastian. "Believable Automatically Synthesized Motion by Knowledge-Enhanced Motion Transformation." Ph.D. thesis, Carnegie Mellon University, 2000.  
<http://citeseer.nj.nec.com/cache/papers/cs/19972/>  
<http://zSzSreports-archive.adm.cs.cmu.edu/zSz2000/zSzCMU-CS-00-163-bw.pdf/grassia00believable.pdf>

Lander, Jeff. "To Deceive Is to Enchant: Programmable Animation" *Game Developer* (May 2000).  
online at [www.darwin3d.com/game/dev/articles/col0500.pdf](http://www.darwin3d.com/game/dev/articles/col0500.pdf)

Perlin, K., and A. Goldberg. "Improv: A System for Scripting Interactive Actors in Virtual Worlds." *Proceedings of Siggraph 96*. pp. 205–216.  
<http://citeseer.nj.nec.com/perlin96improv.html>

Unuma, M., K. Anjyo, and R. Takeushi. "Fourier Principles for Emotion-Based Human Figure Animation." *Proceedings of Siggraph 95*. pp. 91–96.  
[www.cs.wisc.edu/graphics/Courses/cs-838-1999/Papers/UNUMA.PDF](http://www.cs.wisc.edu/graphics/Courses/cs-838-1999/Papers/UNUMA.PDF)

## Into the Blender

**A**nimation blending is much more than a buzzword. Even in its current state, which is far from mature, it has revolutionary potential for bringing freshness, variety, and a wider range of behaviors to game characters. As new options become available, such as the ability to blend IK targets and time-warps for blending between mismatched cycles, the power of blending will only become greater.

While games have only just begun to harness the power of this new tool, it's clear that blending will soon be an indispensable weapon in the designer's arsenal. Working with blends does involve new challenges for coders, artists, and producers, but mastering them is far from impossible, and the rewards of doing so will be great for both developers and our players. 🎮

# Taking Your Project to the MAT: Implementing Milestone Acceptance Tests

All roads lead to Rome, or so goes the expression. In classical times, Roman engineers placed stone markers every mile by the side of their highways, each one indicating the distance to the capital city, the center of the empire. In software development, the release-to-manufacturing (RTM) date is Rome and the schedule is the road leading to this destination. Every project can use periodic milestones to gauge progress and set attainable objectives on a long and often difficult journey. Milestones also provide convenient places to pause to get one's bearings or to recuperate before moving on.

This article assumes that you already use milestones, but possibly not as consistently or efficiently as you would like to. You may not use a Milestone Acceptance Test (MAT) document, at least as it is presented here. It is my hope that the methodology my group has developed over a number of projects at Microsoft will prove useful to others.

## Setting Milestones

In the Microsoft Games Studios (MGS), we use MATs to help set goals for each milestone, define the success criteria, and verify that these criteria have been met before a team can move on to the next milestone. These documents also help break the master schedule (usually a large Microsoft Project file) into convenient chunks that are easier to understand at a glance. As a test lead, the act of creating a project's MATs helps me become intimately familiar with the feature set, uncover problems with the schedule, and establish the quality bar for each milestone.

How frequent should milestones be? At MGS, we usually set milestones six to 10 weeks apart. If they're farther apart, you risk losing focus, which defeats the purpose of milestones. If they're closer together, you risk putting your project into a perpetual state of milestone preparation and verification, and nothing really gets accomplished.

---

**CHRIS HIND** | *Chris is a test lead in the role-playing, adventure, and technology (RAT) group within Microsoft Games Studios. He is currently working on a future Xbox title. Contact him at [chind@microsoft.com](mailto:chind@microsoft.com).*

In addition, you can use one or more interim drops between milestones to track progress and show developers what to expect from final acceptance. In my current project, we decided to use the interim drops as a forcing function to improve our level-completion process. At each interim drop, the developers deliver new engine code. This allows testing and stabilization to happen before the level designers need to access this code base for their own deliverables. It also means that during the final drop, developers are available to address level-specific problems that may block level designers from resolving bugs assigned to them. Similarly, modelers complete their deliverables for the interim drop so that animators have time to finish their work by the final drop. Finally, artists and animators deliver out-of-game art assets (for example, AVIs showing animations in-progress or an updated art bible) during the interim drops. This allows the product manager (PM) and me to verify these deliverables outside of the often-hectic milestone acceptance period.

Having decided upon the frequency of your milestones, set specific due dates for each milestone candidate drop. I prefer that these drops be scheduled for Fridays, with an optional extension to 9:00A.M. Monday morning. This provides a week-end buffer and makes it easier to calculate and remember the end of our 10-day acceptance period (explained later under "Testing a Milestone Candidate Build"). It often makes sense to set deadlines around important dates such as management reviews, E3, or hard-stop localization deadlines.

Whatever schedule you set, double-check everyone's understanding. If you expect the milestone drop in the morning and others plan to prepare it in the evening (or vice versa), better to discover this disconnect early. Clarify whether this time represents the launch of the build process or the start of testing. There could be several hours' difference between the two when you consider the time necessary to compile a build, install it on a test machine, and perform a smoke test (the preliminary run performed immediately after delivery).

In a publisher-developer relationship, be sure to consider additional factors such as Internet connection speed (for FTP processes), physical distance (mailing CD-ROMs), and time zones. While working with a developer in France, for example, there was a nine-hour time difference and sub-optimal Internet

connection to consider. The Redmond-based test team at Microsoft had to initiate the download by 5:00A.M. so that we would have the option of requesting a second build that same day should our smoke test fail. Working backward, the development team had to start their build process in the morning (France time; midnight in Redmond) to have time to compile, smoke-test, and upload the build to our FTP site.

## The Contents of a Milestone

**F**undamentally, a milestone contains whatever tasks in your schedule fall within a range of dates. While this is the simplest definition, you can do more to make each milestone cohesive and worth striving toward. At MGS, we try to give each milestone a theme or focus, or at least make them a logical grouping of deliverables. For instance, “first playable mission” is a great milestone focus, as it clearly indicates the objective of pulling together a sample level. This becomes a launching point for visualizing tasks that must be completed and may actually be missing from the schedule. A focused milestone is more likely to generate demonstrable progress and excitement (with management, marketing, and the development team) than a milestone that simply sets out to complete a series of scheduled items.

In my last two projects, the PM and I pushed our development team to define their milestones around complete levels (where possible), which could be delivered in two states: alpha and beta. In alpha state, the test team could validate geometry, AI, and basic gameplay. In beta state, a level was supposed to be essentially finished, with the understanding that additional polish and play-balancing would happen. This strategy produced a half-dozen playable levels one year before the final “release to manufacturing” (RTM) milestone. No previous project received as much integration testing, play-test time, or usability feedback.

Figure 1 shows a simplified milestone structure where each milestone has a clearly defined goal. (The contents of each milestone do vary from project to project.)

## Creating a MAT

**F**or most projects at MGS, the test lead is responsible for creating the first draft of the MAT. It shouldn’t take more than one or two days to generate a first draft. This draft then gets sent to the development team (designers, coders, and artists) to review and comment on. The test lead and PM review the comments and then decide whether to incorporate or reject them. This process of give-and-take continues until everyone agrees on the final result, which takes anywhere from a few days to a week. In contrast, on my current project, the producer at the external development house generates the first draft. The benefit here is that she knows the deliverables better than anyone and has better access to the leads. There are trade-offs

MILESTONE	DELIVERABLE
M0 – Kick-off	Contract, contact info
M1 – Vision/Prototyping	Vision document, high-level design and determine project scope
M2 – Design/Engine	Project schedule initial level designs, functional specification, art bible, and concept art
M3 – Proof-of-Concept	Rendering engine, scripting engine, camera control, collision system, character in game, “money shot” of expected final visualization. Refine character control, run intellectual property though focus group, schedule management review
M4 – First Playable	Rudimentary UI, sample mission, basic attacks, basic camera, representative gameplay. Perform play-test
M5 – Production	Levels 1 to 4, game shell (select mission), inventory system, finished camera
M6 – Production	Levels 5 to 8, tutorial, save/load, game shell (options)
M7 – Feature Complete	Levels 9 to 12, all game features in and functioning, including cheat codes, finalized lighting and shadows, finalized special effects
M8 – Code Complete	Levels 13 to 15, code optimization and/or rewrites complete
M9 – Content Complete	All game content in and functioning. Bonus levels, all cinematic cut-scenes in place. Release to beta and localization
M10 – RC0 to RC?	Release candidate(s)
M11 – RTM/RTC	Release to manufacturing / release to certification (for console titles)

**FIGURE 1.** This sample milestone overview reveals how and when the game is going to come together.

to this arrangement, not the least of which is that the PM and I must be extra diligent during our subsequent review of the MAT to ensure that it represents all milestone and schedule goals. Whoever creates the MAT, the most important element is collaboration between those who create the deliverables and those who verify them.

If your schedule is in excellent shape or you have worked on a similar project before, you may be able to create all of the MATs (or one meta-MAT) early in the project. More likely, the required level of detail is simply not present until each mile-



stone's start date approaches. To ensure that MAT creation does not drag on too long and that the test team has sufficient time to prepare for the next milestone, you might add "Sign off on the next milestone's MAT" as an exit criterion to each MAT.

When creating a MAT, you will likely follow one of two approaches: the schedule-driven (or bottom-up) approach or the goal-driven (top-down) approach. In the schedule-driven approach, you comb through the schedule for tasks that fall between the milestone's start and end dates and add each one to the MAT. Figure 2 shows typical tasks in a fictitious project schedule. Additional clarification is usually needed to make sense of the often-arcaic notation used in schedules (particularly in the Tech/Dev section). Be sure to specify how a successful implementation would manifest itself and behave in-game. This approach works best if you have a reasonably complete schedule and the developers are good at answering questions.

In the goal-driven approach, your team first generates a milestone showing when all of the features and levels will appear in game. This is similar to the outline provided in Figure 1 but is usually more detailed. The milestone outline differs from the schedule in that it's only concerned with deliverables as they

will appear in-game. This outline becomes the foundation for your MATs and drives the revision of the schedule to ensure that these deliverables are met. This approach works well if your project has a weak or incomplete schedule, or if you find it difficult to get answers from the development team.

In the end, you might consider a mix of these two approaches. You can use the goal-driven approach to set high-level expectations and ensure that everything can be finished by the end of the project, then use the schedule-driven approach to fill in the details on each MAT.

The biggest challenge when creating a MAT is getting the right level of detail. If the MAT is too vague, it fails to enforce the schedule and standards of quality. If the MAT is too granular, you will spend a lot of time negotiating and revising its content right up to the lockdown date (see the following section, "Locking Down a MAT"), frustrate the developers by constraining their implementation, and overwhelm the testers as they attempt to verify such a daunting list during the short acceptance phase (more under "Testing a Milestone Candidate Build"). Figure 3 shows a portion of three hypothetical MAT outlines, built from the schedule in Figure 2. The too-vague example speaks for itself. In the too-granular example, note how the "refined chase camera" tasks are not really testable. Under "animations, player character," the MAT squashes creativity should an animator think up better idle animations. The Enemies and Gameplay sections are too tied to a design that is almost guaranteed to change. This too-granular MAT loses focus of what the project is about (creating a great game) and instead acts as a sterile checklist or process dictator. In contrast, the good MAT example shows how you can set measurable objectives without constraining creativity.

To help you get you started creating your own MAT, download the MAT template from the *Game Developer* web site at [www.gdmag.com](http://www.gdmag.com).

## Locking Down a MAT

Once everyone is comfortable with a MAT, lock it down. In this state, changes are no longer allowed. If you fail to formalize this lockdown process, the temptation to tweak the MAT continually (right up to the milestone due date) remains. Not only is constant change costly in terms of analyzing the impact of the change and updating the MAT document, it's also bad practice. The MAT is supposed to drive development, not reflect a slipping schedule. This is particularly critical in a developer-publisher relationship, in which delivery of specific content is a matter of contractual agreement and often tied to payment.

For projects where signing off on next milestone's MAT is included as an exit requirement in the current MAT, the lockdown date is obvious. Otherwise, specify a date that makes sense. For instance, you might require that each MAT be locked down two to four weeks before the milestone due date, depending on the length of your milestones and other project-specific factors. This time frame gives a team more than enough time to draft, dis-

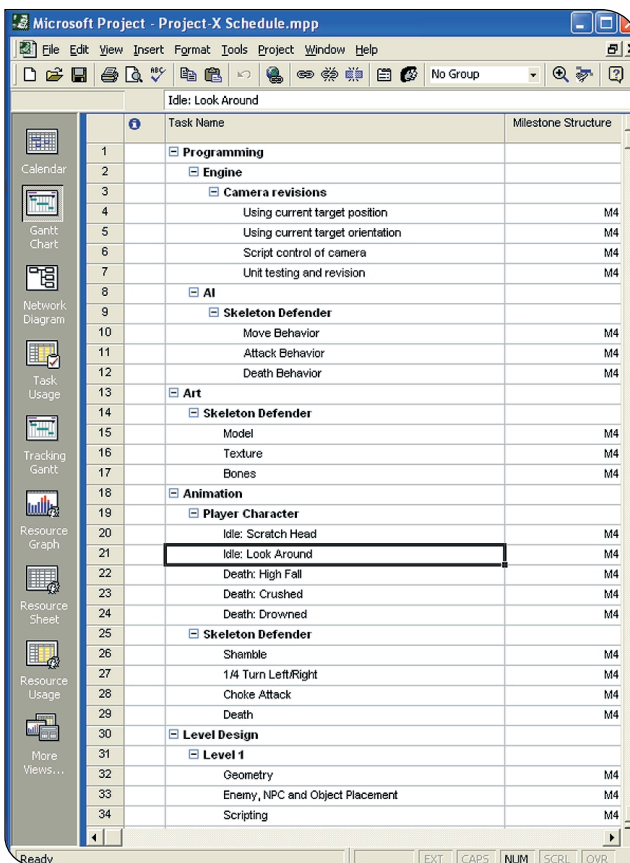


FIGURE 2. The tasks in this fictitious schedule (created in Microsoft Project) constitute the basis of the project's M4 MAT document.

## MILESTONE ACCEPTANCE

**POOR MAT: Too Vague**

- Refined Chase Camera (tweaking)
- New Player Character Animations
- Level 1 Complete

---

**POOR MAT: Too Granular**

- Refined Chase Camera (tweaking)
  - Using current target position
  - Using current target orientation
  - Script control of camera
  - Unit testing and revision
- Animations
  - Player Character
    - Idle: Scratch Head
    - Idle: Look Around
    - Death: High Fall
    - Death: Crushed
    - Death: Drowned
- Level 1 Complete
  - Landscape
    - <details omitted for space>
  - Enemies
    - Three "Skeleton Defenders" in Hall of Bones
    - Skeleton Defender
      - Model
        - Model
        - Texture
        - Bones
      - Animation
        - Shamble
        - 1/4 Turn Left/Right
        - Choke Attack
        - Death
      - AI
        - Move Behavior
        - Attack Behavior
        - Death Behavior
  - Gameplay
    - NPCs
      - Failed tomb robber standing next to entrance provides hint #37.
    - Puzzle
      - Tomb of the Ancient Defender, secret door puzzle: stand on pressure-sensitive floor tile while singing the song of opening.
    - Experience Points
      - Player character gets 200 experience points for escaping the tomb.

**GOOD MAT: Measurable Objectives**

- Refined Chase Camera
  - When character remains in place, camera could slowly drift around behind character.
  - When character stands still, his head should be a little lower than the center of the frame.
  - When character moves to the right or left, camera should rotate in place slightly to show more of what is ahead.
  - When character moves away from camera, camera should tilt to show more of what is ahead of him.
  - When character moves toward camera, camera should tilt to show more of what is between the camera and him.
- Animations
  - Player Character
    - Idle Animations (2)
    - Death Animations (3)
- Level 1
  - Landscape
    - Geometry
      - Entire map is walkable
      - Entire map has correct collision
      - Player character cannot step off map anywhere
    - Textures
      - All surfaces textured
      - No visible seams
      - Skybox present (if applicable)
    - Object Placement
      - No floating objects on map
      - No objects embedded in geometry on map
    - Lighting
      - Landscape "lit" with correct lighting model
  - Enemies
    - All Enemies present, per level design document (model and finalized textures)
    - All animations complete and in-game
    - AI complete
  - Gameplay
    - NPCs
      - All NPC encounters per level design document
    - Puzzles
      - All puzzles fully described in level design document
      - All puzzles finalized, functional, and fun
    - Experience Points
      - Experience points allocated per level design document

**FIGURE 3:** The first MAT outline (top left) is too vague, the second (bottom left) is too granular. The third MAT (right) contains clear, measurable objectives without restricting the ultimate implementation.

cuss, and revise the MAT. Most potential problems (such as overly optimistic scheduling or unforeseen dependencies) should be obvious before then.

Enforcing lockdown is simply a matter of establishing ownership and version control. At MGS, the test team owns the MAT and keeps the master copy checked into Visual SourceSafe. A read-only copy is placed on a file share for reference purposes. Change requests — or even subsequent drafts of the MAT — are submitted via an agreed-upon mechanism, such as e-mail or an

entry in a defect-tracking database. The test lead ensures that all approved changes are applied to the master copy.

Occasionally, you may need (or just think you need) to make changes to the MAT after the lockdown date. Carefully consider the reasons for this potential change. If they are legitimate — for example, approved redesign, unforeseen dependencies that require the swapping in of tasks from later milestones, or unexpected complexity or delays — then go ahead with the change, but only after identifying any potential consequences or risks and dis-

curring these with the other leads. The likelihood that a MAT would need to be changed is greater early on in the project and should decrease as people become more comfortable using them.

## Testing a Milestone Candidate Build

At MGS, we are contractually bound to evaluate milestone candidate builds and provide feedback — acceptance or rejection — within 10 business days. In theory, rejecting one build and requesting another resets this 10-day deadline. In practice, we strive to limit the entire acceptance period to 10 days to minimize impact on the schedule. Though this deadline is based on the needs of a contractual relationship between a publisher and a developer, internally managed projects should also set an acceptance deadline. Ten days should be adequate, if used wisely.

Remember, this stage involves verifying that a milestone meets its intended goals, not performing the same comprehensive test pass that should be expected in the months leading up to the ship date.

I break our 10-day acceptance period into four phases: initial review (one to two days), testing (three to four days), regression testing (two to three days), and final review (one to two days).

Initial review begins with verifying the presence of all assets: art files, documents, build (debug, release, instrument versions), build notes, and so on. With regards to nonbuild assets, the test lead (or sometimes the PM) merely verifies that these are present, leaving the actual evaluation to other leads. At MGS, we have an art liaison review concept art, models, and animation; the PM checks the schedule and full design document; a usability expert performs heuristic evaluation of the UI, control scheme, inventory system, menu structure, and other such aspects. As test lead, I place either a check mark or an X in the “Present” column and then follow up with these other leads during the final review phase. Once the initial review is complete, I e-mail the development team with a list of any items that appear to be missing and a list of questions (for example, “Where can I find an example of bumpmapping in this build?”). Since missing assets can delay verification, I expect a response as soon as possible.

In the three or four days devoted to the testing phase, don’t expect to test every feature completely. That’s not the goal. Aim to verify that every feature works and does not block further testing. (This further testing is performed in the weeks and months following milestone acceptance, while developers, artists, and level designers work on the next milestone’s deliverables.) The test team should divide the most important features equally so that every area gets a similar depth of coverage. Failing that (perhaps it’s early in the project, and there’s only a test lead), perform a series of test passes, each one deeper than the last. This ensures at least minimal coverage of every feature. It’s also more likely to spread bug fixes among the development team. In contrast, if a tester focuses on one feature or level for too long before moving on to the next one, a single member of the development team may become swamped with work while other members (especially artists and animators) sit idle. In my current project, our MATs

Category	Item	Present?	Pass?	Comments
Development Tasks	New renderor design	✓	✓	See "Technical Spec.doc", p.31
Level	New renderer implementation	✓	✓	
	Re-implement chase camera	✓	X	Bug 93: blocks milestone acceptance.
	Script controlled Chase Camera	✓	✓	
	Static Light Maps	✓	✓	Implemented in Separate Hit Interval Level 1
Test Hooks	Test Hook: Player / Display	✓	✓	
	Test Hook: Fillamentary	✓	✓	
	Test Hook: PichupItem: <ID>	✓	✓	
	Test Hook: Power	X	✓	Task incomplete due to unexpected illness, schedule revised
	Test Hook: Collision	✓	✓	
	Test Hook: JumpToLevel:<ID>	✓	✓	
Character Control	Sneak	✓	✓	
	Walk	✓	✓	
	Run	✓	✓	
	Climb Ladder	✓	✓	

FIGURE 4. Your MAT should look something like this by the final review phase of Milestone Acceptance, replete with check marks, Xs, and comments.

specify that all Priority 1, Severity 1, and Severity 2 bugs be fixed before milestone acceptance. Priority 1 bugs are test blockers: all features must be in the build, basically working, and fully testable. Severity 1 bugs are crashes and things that prevent continuing gameplay: by running the game entirely in debug mode, I can catch most crashes and locks; I also seek out bad collision geometry that causes the playable character to fall through the world. Severity 2 bugs signify a broken feature: this involves checking each MAT item against the latest spec to verify that it behaves correctly.

As soon as a fair number of defects have been logged, perform bug triage to determine which ones must be fixed immediately in order to pass milestone acceptance and which can be punted to a later milestone. You definitely want to take this opportunity to stabilize the code base as well as avoid adding to the workload and stress of the final crunch that can result from postponing too many bug fixes. At the same time, stubborn insistence can jeopardize the project schedule, lead to poorly considered “quick-fix” solutions, and damage your relationship with the team. As a member of the triage team, I enforce each MAT’s requirement that we fix all Severity 1, Severity 2, and Priority 1 bugs immediately. For each remaining bug, the triage team checks if there is another milestone where the corresponding feature or code is scheduled for revision or additional work. If so, we assign the bug to that milestone with the assumption that grouping bug fixes with scheduled tasks is more efficient and reduces risk. If not, then the bug must be resolved immediately. Besides fixing the bug, possible resolutions include reworking the schedule, redesigning the feature (with team approval), or cutting the feature completely. To further mitigate risk, we have an “entry requirement” in our MATs that states that all bugs punted to a given milestone must be fixed before that milestone’s candidate build can be submitted.

Once the development team fixes all milestone-blocking bugs and submits a new build, the test team enters the regression-

testing phase. Testers concentrate on verifying bug fixes and ensuring that previously working features still work, but new milestone-blocking bugs sometimes crop up. This phase may last considerably longer than the recommended three days if subsequent builds are required.

During final review, the test lead meets with other leads or compiles their written feedback, and reviews each MAT item with the testers responsible for the corresponding test area. For each item, he or she notes its status as “present” or “not present,” “passed” or “failed.” Optionally, I record active bugs next to each item and add a comment, such as “Bugs are minor and do not block milestone acceptance,” or “Fails due to bug 134.” At this point, you are ready to meet with the team leads and discuss whether the milestone has passed or failed.

## Accepting a Milestone

**A**t the end of the acceptance period, you should have a completed MAT document, new bugs in your defect-tracking database opened against milestone features, and feedback from various functional leads regarding the deliverables they were responsible for evaluating. This data helps determine whether or not the team has met the milestone’s goals.

At MGS, the test lead, product manager, and product planner are responsible for signing off on milestone acceptance. In an electronic workplace, this signature usually takes the form of an e-mail thread wherein everyone clearly states whether the MAT passed or failed:

- The test lead looks to the MAT document and to the contents of the defect-tracking database: Are all items present? Do all items function per specification? Have all bugs found during the current milestone been resolved in some manner, either fixed or reassigned to a later milestone?
- The test lead also represents the art, development, usability, and localization liaisons, ensuring that their feedback is being considered.
- The product manager has similar concerns, but also considers the big picture: What risks are involved? Are those risks acceptable? How can we lessen risk? How will the schedule be impacted? How might this decision affect our relationship with the development team?
- The product planner is responsible for releasing payment, so the test lead and PM must keep this individual informed regarding the status of a milestone.

Limiting sign-off authority to three people works well, reducing the chance of someone becoming a bottleneck, especially since at MGS only the PM and test lead work full-time on a single project. Additionally, we find it difficult to justify to an external developer that an art or play-test concern, for instance, can block milestone acceptance and payment.

In contrast to the publisher-developer situation described in the preceding example, an integrated environment might require sign-off from the producer, test lead, and development lead. A test lead in this situation requires more authority and more tact,

as he or she may need to argue points against the development lead or possibly even push back on both the development lead and the producer. This is where a MAT really pays off as supporting evidence; after all, everyone contributed to and agreed upon its contents in advance.

Can a milestone pass even if the MAT document contains failed items? While your team should discuss and decide upon this issue (and do so early on in the project), allow for the possibility of a “yes.” Say the usability liaison reports that the new UI has tested abysmally and should be completely reworked, or the art liaison insists that the new character models are constructed of too few polygons, or the development liaison finds serious flaws in the rendering engine’s architecture. Do you hold up the milestone for what could be weeks? Do you force the developers to implement a hack just to pass the letter of the MAT? Probably not. Better to reschedule these items and assign them to a later milestone. Whatever your decision regarding MAT failures, record it somewhere so that the debate does not reappear at a bad time, such as in the middle of a contested milestone acceptance period. You might want to record this decision in the introduction or appendices of every MAT document.

By suggesting that the team can sign off on a “failed” milestone, I don’t mean to endorse leniency. For the most part, MAT items should meet or exceed the predetermined goal for a given milestone. If your MAT contains mostly failures, there is likely a disconnect somewhere in terms of acceptance criteria; review the MAT to determine if it’s too vague (in that it doesn’t set clear expectations) or too specific (allowing no flexibility in terms of implementation). If the failures are legitimate, hold the development team to strict standards: milestone acceptance is the best tool you have to enforce schedule and quality.

## Signing off

**O**nce everyone has signed off on the MAT, that milestone is officially over, and everyone should congratulate one another for their hard work. A little goodwill helps soften any hard feelings that may have developed during the often-tense acceptance phase. Some teams celebrate with a milestone dinner. The PMs I have worked with usually send a little milestone acceptance gift, such as a big jar of jellybeans or a box of assorted toys. Finally, encourage team members to take a couple of well-deserved days off, schedule permitting. Better yet, plan ahead and schedule vacations for just after the expected acceptance date, leaving a one-to-two-week buffer to account for potential last-minute delays.

Enjoy this feeling of accomplishment while it lasts. Soon, you’ll be starting work on another milestone, driving towards the requirements in the next MAT. 🎉

### ACKNOWLEDGEMENTS

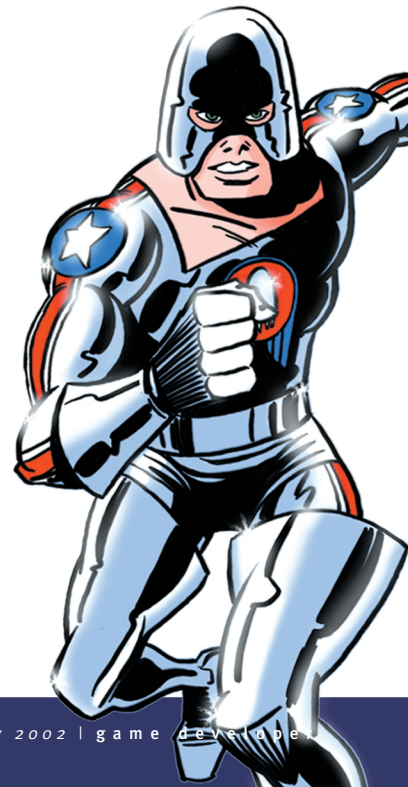
Thanks to Scott Amis for helping brainstorm the article idea and to Richard Thames Rowan for providing feedback.

# Irrational Games' Freedom Force



## GAME DATA

**PUBLISHER:** Crave Entertainment/Electronic Arts  
**FULL-TIME DEVELOPERS:** 24  
**PART-TIME DEVELOPERS:** 3  
**CONTRACTORS:** 1  
**LENGTH OF DEVELOPMENT:** 18 months  
**RELEASE DATE:** April 2002  
**PLATFORM:** PC  
**DEVELOPMENT HARDWARE:** 600MHz Pentium IIIs with 256MB RAM and 32MB GeForce 2s (average)  
**DEVELOPMENT SOFTWARE:** MS Visual C++, cvs, BoundsChecker, 3DS Max, Photoshop, Paint Shop Pro, ACDSee, MS Access, and ColdFusion  
**NOTABLE TECHNOLOGIES:** NDL NetImmerse




In November 1999, after Irrational finished SYSTEM SHOCK, I wrote a Postmortem for *Game Developer*. Did we apply the lessons we learned from SYSTEM SHOCK 2 to our new project, FREEDOM FORCE? If not, what new lessons did we learn?

In answering these questions, I'll avoid discussing source code control systems and resource management. Instead, I'll discuss problems we encountered motivating and organizing game developers on a new team and new project, and how design issues forced us to think about risk management as opposed to innovation and ambition.

SYSTEM SHOCK 2 presented a number of major challenges that were overcome only with hard work and good luck. It was my ambition not to face the same challenges on the next project, or, if these challenges were to be faced, then to outflank them and not confront them head on. Unfortunately, most of these problems did reoccur on FREEDOM FORCE, and the fact that we did, in the main, surmount them again resulted more from hard work and good fortune than clear thinking.

## Starting Over Again



At the time I wrote the SYSTEM SHOCK 2 Postmortem, I was about to return to Australia from Boston, where Irrational is based. Our development team in Boston moved on to a new project, THE LOST, a survival horror RPG that shares many features in common with SHOCK 2 but moves them into the console environment. It was my job to organize a new development project to be run from a new office located in Australia.

After several months back in my home town in Australia, we managed to organize FREEDOM FORCE for the

new studio. Although design elements mutated during its early development, the core idea for FREEDOM FORCE has always been very clearly defined: a superhero tactical combat game aiming to re-create comic-book battles on a PC. There is a lot that goes along with this notion: RPG elements, user-created characters, homage to 1960s comic conventions, and so on. But the game's core is superhero tactical combat and all that goes along with it.

So we established our new office in Canberra, a sleepy town of 300,000 that happens to be Australia's capital, and I set about finding a team to complete the game in around 18 months. What followed was a wild ride with lots of ups and downs. Most of these experiences don't fall neatly into the categories of "what went right" or "what went wrong"—"what mostly went right" or "what mostly went wrong" would be more accurate.

## What Went Right

**1** ● **Finding a new team.** "The SYSTEM SHOCK team was frighteningly young and inexperienced, especially for such a high-profile title," I wrote in the SYSTEM SHOCK 2 Postmortem. Despite this forewarning, we committed ourselves to the same problem on our new project.

The initial problem was simple: to fully staff the project in the shortest time period possible while simultaneously setting up the new office and proceeding with development. The many administrative problems associated with establishing a new office were time consuming, but could always be overcome with enough money. Renting office space, buying PCs and software, setting up Internet connections, and establishing payroll are all relatively well-understood problems; recruiting game developers is another problem altogether.

Any developer, new or growing, faces recruiting problems. We were in a slightly unusual position with some unique advantages and disadvantages. On the one hand we had dug ourselves into a hole by committing to a quick start-up, a technically difficult project, and a relatively short development schedule. On the other hand, we had many advantages not enjoyed by a completely new company: as a result of our work on SYSTEM SHOCK 2, we had a high profile; we had adequate cash reserves, and we had a core of experienced developers available to help out, though I was the only one of them physically present in Australia.

These advantages ultimately enabled us to successfully build a new team. We raised our profile in the local development community as fast as possible, as we knew that our reputation and the project would attract staff if they could be made aware of what was going on. Articles in the local press and on the local TV station made Canberra residents aware that we were established in their town; developer diaries and previews in Australia's gaming press and interviews on local web sites made us known to the local developer community.

Developing in Australia is an attractive proposition: Australians are well educated and technologically minded. They speak English (of a sort) and are familiar with just about every aspect of American culture. Despite the presence of several well established and quality developers, there are still a lot fewer opportunities for game developers than in the U.S., and we counted on not having to face the same savage competition for talented staff that we were familiar with in Boston. However, there is not a

**JONATHAN CHEY** | Jon is a co-founder of Irrational Games and currently managing director of Irrational Games Australia. He served as project manager and lead designer on FREEDOM FORCE and was project manager on SYSTEM SHOCK 2. Prior to working at Irrational he spent time at the late Looking Glass Studios and received his Ph.D. in cognitive science from Boston University.



great pool of experienced staff in the country. This would be our biggest problem — finding that core group of experienced people to manage and direct the talented but relatively inexperienced staff who would form the bulk of the team.

When recruiting, we faced choices between several competing alternatives: expensive and distant (and therefore poorly evaluated) staff from overseas, inexperienced but talented local staff, or experienced local developers from outside the industry. In almost all cases we settled for inexperienced local developers with great potential. This caused us many short-term headaches but set us up to be in a good position provided we could weather these problems.

**2. Supporting the fans.** Very early on in the development we realized that it would be impossible to make a game with enough content to satisfy comic-book fans. Clearly, we needed to make a big effort to enable players to incorporate their own content into the game. We tackled this at several different levels to support casual users who just want to make their own characters, more

committed players who want to create their own skins, and hardcore enthusiasts who want to create new meshes, animations, or missions.

To support the casual users, we created a flexible system for generating characters and powers that we would use to both build the precreated characters and expose to users in the game. Our power system allows users to construct new characters not just by changing their stats but also by creating new powers from scratch. In addition to setting the parameters for these powers, users can choose an animation and an effect to go with that power. Each created power has a cost value and then goes to form part of the overall cost of that character.

While this flexibility created balancing nightmares for us, it opened up an immense array of possibilities for user-created content in the game. How well users ultimately respond to this remains to be seen, but we have already had an excellent response to the already released character tool — a viewer that allows users to cycle through animations and skins on game character meshes. With nothing more than this to work

with the online community has already generated hundreds of new skins, meshes, and animations.

This kind of user support is clearly important to the extended life of the product. Minimal effort on our part has paid off many times already. What we could have achieved with greater levels of support — releasing the full character and power generation system, for example — we can only imagine.

**3. Not starting from scratch: NetImmerse.** We successfully made use of the NetImmerse technology from Numerical Design Ltd. to provide the core rendering systems for the game. This proved to be a wise decision that gave us an early leg up and provided a stable rendering platform. Third-party rendering technology has matured enough to make a lot of sense; the remaining issues come down to price and flexibility.

**4. Cool scripting language: Python.** We made good use of Python, which we used for a scripting language. This was an admirable choice whose only downside was that it required our designers to be fairly competent programmers. Because Python is already a complete language with a relatively easily understood syntax, we were tempted to provide little on top of it and require the designers to write real Python code. Scripting at this level essentially is programming and requires designers who wish to write scripts to master programming as well as conceptualizing game elements. We encountered problems initially because we didn't realize the level of technical expertise required when we hired our script designers. But ultimately Python proved to be a flexible and powerful scripting tool.

**5. Breaking the curse.** FREEDOM FORCE is not a unique concept. Similar kinds of games have been attempted before, most noticeably Microprose's GUARDIANS: AGENTS OF JUSTICE, a CHAMPIONS licensed game, and Bullfrog's INDESTRUCTIBLES. None of these games has made it onto the shelves thus far,



The team in action. From left to right: Minuteman, El Diablo (flying), Manbot, Iron Ox, Man O' War (flying), Liberty Lad, Mentor.



leading to the perception of a hex on this genre. How did we avoid this curse?

Perhaps it's more instructive to consider why such a curse might exist. Perhaps a good superhero game has only recently become possible. Superheroes are about smashing stuff up, and making clever use of the environment as much as extravagant numbers of powers and abilities. This kind of stuff doesn't lend itself well to the sort of static environments that most games have been able to portray up to now.

X-COM, the initial touchstone for our game design, included a destructible environment that was rich enough to portray superhero combat, but its turn-based mechanics would be hard pressed to do justice to the dynamics of comic-book combat. What we needed was X-COM levels of destruction and interaction but in real time, something that has only recently become possible.

The destructible and interactive nature of the FREEDOM FORCE environment creates major headaches for the physics and AI systems as well as the renderer. The CPU requires considerable grunt to handle these systems properly. While this kind of stuff was clearly possible a few years ago, it's only recently that a game can handle it without a major investment in experimental technology (or indeed, with a third-party rendering engine).

So has FREEDOM FORCE then avoided the superhero curse? To the extent that the game is on shelves, yes. Certainly there are aspects of comic books that could be realized better, and this will hopefully be addressed in future products. The richness of the character interactions is less than it might be. Proper dynamic character interactions await the attentions of future developers.

## What Went Wrong

**1** ● **Organizing the team.** As we slowly grew the team, the Canberra Irrational office changed from a start-up into a large development studio. This process created problems familiar from our previous project. However, due to the larger scale of the project, these problems were more severe this time around.

The initial work on FREEDOM FORCE was done in my living room by the first four staff members. Now, 18 months later, 25 developers occupy a large office in the downtown business area. We are actively looking to our next larger office space and considering how to manage multiple simultaneous projects. Naturally, this rapid growth has created changes and points of stress.

One of the greatest stresses is simply the change in organizational structure required by growth. Four people working in a living room don't require hierarchical management structures or formal management systems. Imposing these kind of systems can lead to resentment and bad feelings, particularly among game developers who have come to regard informality and freedom as an essential part of working in the industry. As a result, we have had to be very careful about making changes along these lines; but changes are necessary to accommodate greater numbers of people.

For example, when we were working in a living room, we had no official policy or rules about such things as work hours, the use of office computers for game playing, and the use of Internet services for downloading movies or music. As we grew, we had to introduce standards for all of these, if for no other reason than that with 25 people there will always be outliers who test

the bounds of any unregulated system. One developer who prefers to start work in the afternoon, for example, or another who spends the afternoon playing LAN games, can cause problems not only through personal behavior but also by undermining other people's work practices.

Regulations and restrictions that might seem reasonable or lax in a big business can seem harsh and authoritarian when they are introduced into a less-regulated environment. Managing the process requires slow change and providing proper justifications to accompany any rule system. It may be better to impose these rules from the start, even if they are only required once the team has grown.

**2** ● **Motivation and self-belief.** Most games developers are in the industry because they love the work. Game development is an act of creation, where a small group of people bring into being a complex and original piece of software. A very important part of the reward is the recognition of one's peers and the public. That's why game credits, while unimportant to the general public, are an important part of any game.

But before the game is finished, developers, like most people, not only want to be recognized for their work, but also want to know that they are doing something worthwhile. The path from initial conception of a game to gold master and box on the shelves is long and hard. An experienced team that has worked together and achieved good results can rely on their past to sustain their belief in themselves and the eventual results; a new team needs to make a leap of faith.

Perhaps the biggest difficulty we faced in the development of FREEDOM FORCE was getting the team to believe in themselves and the product. Initial progress



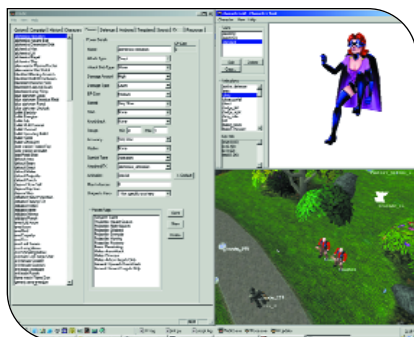
on the game was slow, and a year into development there were many on the team who were still unclear about what the game was about or who were not confident in their abilities or the abilities of other team members. This situation was complicated by the fact that there was no existing game that we could identify as a template for what FREEDOM FORCE would finally be like.

For this reason, milestones along the way, such as our E3 demo, were very important, not only for their external PR potential but also to prove to our own team that we were making progress. Nevertheless, it was not until a couple of months before the game was complete that it was really clear to everyone what standard was going to be achieved and whether or not the core gameplay mechanics worked. It was only at this stage that enthusiasm and productivity and enthusiasm peaked.

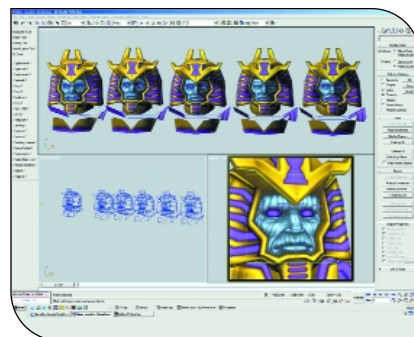
Our catch-22 was that we required self-belief to create a good product, but self-belief was contingent upon creating a good product. There were several moments in the development process where incremental progress toward a good product produced positive feedback to self-belief. Had these moments not been present, destructive negative feedback would have resulted instead.

What factors helped create positive feedback? The most important elements were the key people who bought into the project ideas early on and sustained that buy-in until completion. Without those people, things could have turned out very differently. Finding and keeping these people can therefore mean the difference between success and failure.

**3 • Inter-office politics.** A difficulty we encountered in developing SYSTEM SHOCK 2 was coordinating development between Irrational and our co-



The editor and character tool in action.



Character creation in 3DS Max.

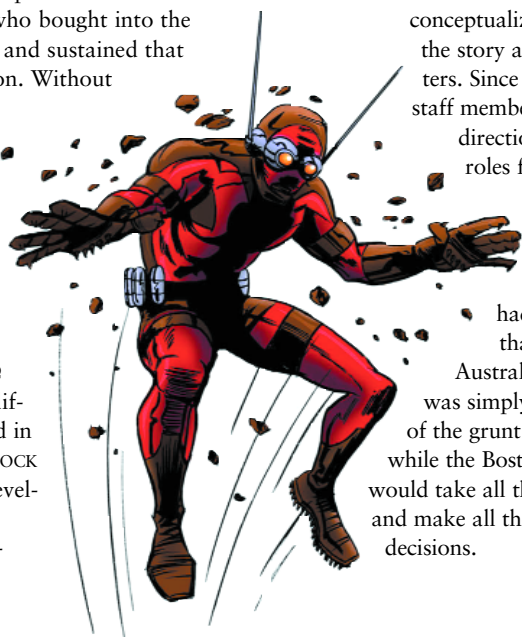
developer, Looking Glass Studios. Although we were the sole developer of FREEDOM FORCE, two teams were involved: one in Australia and a smaller group in Boston. Although both were a part of the same company, this arrangement created problems, made worse by the fact that they were unforeseen and that we were slow to respond to them once they became apparent.

I've already identified the root causes of these problems: passionate interest in being recognized as a creator of the product combined with lack of self-belief. Our Australian team felt very strongly that they were responsible for creating the game, yet publicity often ascribed development of the product to Irrational Games, commonly understood to be based in Boston, not Australia. In addition, Boston staff members filled two key roles,

conceptualization of the story and characters. Since Australian staff members required direction in these roles from

Boston, some people in Canberra had the feeling that the

Australian office was simply in charge of the grunt work, while the Boston office would take all the credit and make all the interesting decisions.



Resolving this issue took time and energy — and it wasn't resolved until we recognized that this problem would not just go away on its own. There's no simple solution to these kinds of issues, only an ongoing process of encouraging more communication in the form of moving staff from one office to the other and ensuring that appropriate public exposure is provided to all parts of the company. For example, team members in the Australian office had no real idea what roles were being filled by U.S. staff. Once they became aware of these people's efforts, the problems ebbed.

Companies that rely heavily on the passion and enthusiasm of staff must be prepared to deal with the negative as well as the positive aspects of those feelings. Most employees in large companies accept that they are faceless elements of a process, but many game developers want more than that. It's key that the company recognize and respond to this feeling.

**4 • Ambition and risk.** Games are hard to design from the scratch. Some of the most successful developers espouse iterative game design processes in which games are polished and changed over time — the end result of which is a "It'll ship when it's ready" policy. Most developers cannot afford this luxury.

For many games, a fixed schedule is no greater a problem than any group of people creating a complex product faces, and it has been suggested that game design should proceed as would any major software project, with a specification document followed by an implementation

phase. However, games that employ new gameplay or test out new ideas have to find room in their schedules for experimentation in a way that developing a spreadsheet program does not. Irrational's game designs do not rely on exact analogues from existing products. This leads to scheduling strains as the teams implement complex technology, create ambitious amounts of content, and test out new gameplay at the same time.

Certain tools and processes can help with this. The NetImmerse engine helped us get our renderer and basic game shell running in the first month of the project. Artists were able to create content and view it in the game in the same time period. Monthly "releases" provided us with a steady stream of prototypes through which we were able to explore gameplay and UI concepts. Nonetheless, we had a tough job prototyping and experimenting — as well as actually shipping the product — in a short period of time.

Our major difficulty stemmed from lack of clarity about how gameplay issues would be resolved. Our initial specification for the game described it as a turn-based tactical combat game involving multiple player-controlled characters with a deep set of combat options.



However, we quickly decided that turn-based gameplay was no longer appropriate for this kind of title, and therefore committed to creating a real-time game. Although a sensible decision, it increased our risk, as we didn't clearly understand how to create complex tactical gameplay in a real-time game.

Constructing a simulation environment and UI that realized this design concept became a major risk. The solution we settled on made use of elements drawn from other games put together in a unique way. We utilized a contextual command menu with auto-pausing whenever the menu was open as well as other contextual UI elements. Camera controls, combat mechanics, and many other elements were balanced carefully to find the right amount of information and decision making that would neither make the game too simplistic nor overload the player. The balance we settled on was not what we wrote our initial design proposal; we introduced major changes and new elements as we worked the problem through. Without our regular monthly prototypes, we would have had no hope of solving these problems in time. Even with them we were lucky to do so. Because our gameplay was not completely established until close to the end of the project, we didn't have time to tune the very complex combat and

RPG systems properly.

Although the results are not perfect, we are happy with them and intend to use these same solutions for further tactical combat games. Reusing this design gnosis is the key to avoiding project slippage and crunch periods in the future.

Experimental game design requires a generous schedule. If new game design concepts are to be explored in a short period of time, then technology and content requirements must be low. FREEDOM FORCE contains gameplay innovations and experiments, sophisticated technology (physics and AI), as well as large amounts of content. This is a recipe for overwork and scheduling problems that we hope to avoid in the future.

**5 • Multiplayer blues.** The only other technology issue of note was the continued difficulty developing good multiplayer systems, another thing we had previously struggled with on SYSTEM SHOCK 2. Again, this probably resulted from a failure to recognize multiplayer as a core game element from the beginning of the project. This lack of interest inevitably results in problems as multiplayer systems take a backseat to the development of the single-player technology. As a result, our multiplayer components are less than they can or should be.

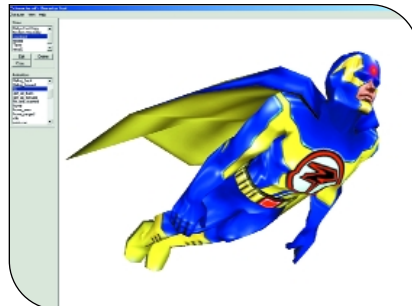
## Lessons Learned (and Relearned)

Although FREEDOM FORCE was Irrational's second published title, the process of building it felt very much like a first project. Many of the problems we encountered have been identical to those encountered in developing SYSTEM SHOCK 2. These are not the kinds of problems that are enjoyable to encounter more than once.

One core lesson I have taken home from this project is the tremendous importance of psychological factors. Developers are human beings, not faceless automatons. Keeping people happy and motivated is difficult, but it is also the most important task for anyone



A city layout in 3DS Max.



The character tool with a generic model released entirely for fan-made custom skins.

# Art, Game Design, Programming, and Technology

**W**hat is the role of technology in games? There is conventional wisdom in the game industry that games and game designs should not be based on technology, especially if we are ever going to be seen as a true art form. I'd like to challenge this wisdom.

I take it as a given that games are — or at least have the potential to be — an art form on par with literature, film, music, visual arts, and the rest. We might squander this opportunity, but I am confident we have the potential. As many have said before, we resemble film in the early 1900s: disrespected as mindless popular entertainment. Sadly, we live down to that expectation all too often. However, someday, someone will

design the game equivalent of D.W. Griffith's *Birth of a Nation*, and the world — like it did with film in 1915 — will take notice of what's really possible with games.

Music is completely different from painting, and games will be different from the other art forms as well. This is why the Hollywood people who occasionally try to overrun us always fail: they don't understand our medium and its strengths and weaknesses.

What is so different about games? The answer is clichéd and appears in a lot of dot-com business plans, but it's the truth: interactivity. The computer underlying all of our games offers us the ability to dynamically respond to the player, and that's something no other art form can touch.

If interactivity is the key differentiator of our form, then it behooves us to understand it. I'm not going to step into the morass of defining interactive and noninteractive, and certainly not good interactivity versus bad interactivity, but it's clear that interactivity is inherently algorithmic at its core. There is a set of inputs (the current state of the world, the actions of the player, and so on), a system that decides what outputs to produce from the inputs (have an NPC kiss the player, have an NPC shoot the player), and a set of those outputs (the rendering of the smooch, a bit of text saying, "You're dead").

*continued on page 55*



Illustration by Ben Fishman

*continued from page 56*

The algorithm that decides what to do at any given moment has to be described to the computer. Depending on the complexity of the algorithm, there are many ways of doing this, ranging from having some high-level visual flowchart system where designers connect boxes with inputs and outputs, all the way down to writing assembly code on a DSP.

However simple these descriptions may seem, they're all computer programming, and this is the inescapable fact of embracing interactivity and games: You must tell a computer how to respond to the various inputs, and you tell a computer how to do things via programming.

Getting back to the conventional wisdom that games shouldn't be founded on technology, if I regard "technology" as "programming," then games are founded on technology, as they must be. I'm not saying games should be founded on fancy graphics tricks, as is so often the case these days. However, designing and tuning the algorithms at the heart of the game — how it reacts to the player and how proactive it is — should be the fundamental task of game design. These algorithms determine how the game feels to play.

Which leads me to my next heretical statement: Game designers must learn to program.

Still with me? No, I don't mean that every game designer has to be a C++ wizard who dreams in curly braces, but game designers must be able to think and solve problems algorithmically and have a very clear understanding of what's easy and

*Game designers must be able to think and solve problems algorithmically and have a very clear understanding of what's easy and what's hard to do on a computer.*

what's hard to do on a computer. Sure, game designers can tell programmers what to implement, but, as all programmers know, there's a world of difference between "what to implement" and "how to implement." The more vague the description of the "what," the more decisions get made during the "how." Those decisions directly affect the gameplay. The more precise the designer is with the description, the closer he or she comes to real programming, and the more control he or she has over the game design itself. Learning to think algorithmically is not a burden for game designers. On the contrary, it liberates designers from the whims of programmers.

As my colleague Jon Blow once put it, programming is the last mile of game design. It grows to be much more than just the last mile when we begin to consider game designs that are not built of simple, hard-coded paths but are procedural by their very nature.

Eventually, the graphics engine will not be a differentiating factor. All that will matter is the algorithms that make one game play differently from another. Those algorithms must be designed. To the game designers who want to do more than tune the damage a weapon inflicts, to designers who want to take games into the realm of interactive storytelling and richer emotional experiences, I say: Study up. The art form needs you. 🎮

**CHRIS HECKER** | *Chris (checker@d6.com) is editor-at-large of Game Developer.*

#### FOR MORE INFORMATION

I am certainly not the first person to say all this stuff. Chris Crawford, for example, has been talking about this since 1987, as you can see in the excellent library of his writings from the old Journal of Computer Game Design:

[www.erasmatazz.com/Library.html](http://www.erasmatazz.com/Library.html)

Also, Richard Rouse did an excellent Soapbox in April 2001 about the advantages of being a designer/programmer; you can read that at:

[www.gamasutra.com/features/20020222/rouse\\_01.htm](http://www.gamasutra.com/features/20020222/rouse_01.htm)