

G A M E D E V E L O P M E N T

# NextGen



VOLUME 18 NUMBER 11 DECEMBER 2011

## POSTMORTEM

### POSTMORTEM: THE GUNSTRINGER

INTERVIEW: BUNGIE'S HAO CHEN

INTUITION AND EXPECTATION: THE PSYCHOLOGY OF  
USER INTERFACES

THE LEADING GAME INDUSTRY MAGAZINE  
COLLECT INSIDE: THE NEW TECHNOLOGY

# Get your game performing like a scripted masterpiece.



## havok™ Script

*The smallest, fastest LUA compatible virtual machine designed specifically for game development*

Havok™ Script armed Relic Entertainment® with a robust toolset, optimized engine, and never-before-seen visibility into their script-related performance and memory changes for the development of Warhammer® 40,000®: Space Marine®.

With these weapons at their disposal, Havok Script was able to help Relic realize their gameplay vision, ensuring each Ork will meet his fate against the chainswords of the Space Marines as efficiently as the last.

Learn more: [www.havok.com/havok-script](http://www.havok.com/havok-script)



Warhammer 40,000: Space Marine – Copyright © Games Workshop Limited 2011. Space Marine, the Space Marine logo, GW, Games Workshop, the Games Workshop logo, 40K, Warhammer, Warhammer 40,000, Warhammer 40,000 Device, 40,000, the Double-headed Eagle device and all associated marks, logos, places, names, creatures, races and race insignia/devices/logos/symbols, vehicles, locations, weapons, units and unit insignia, characters, products, illustrations and images from the Space Marine game and the Warhammer 40,000 universe are either ®, TM and/or © Games Workshop Ltd 2000-2011, variably registered in the UK and other countries around the world, and used under license. All Rights Reserved. Uses RVO2 Library v2.0: © University of North Carolina at Chapel Hill. All rights reserved. Developed by Relic Entertainment. THQ, Relic Entertainment and their respective logos are trademarks and/or registered trademarks of THQ Inc. All rights reserved. All other trademarks, logos and copyrights are the property of their respective owners.



## CONTENTS.1211

VOLUME 18 NUMBER 11

### DEPARTMENTS

- 2 GAME PLAN** *By Brandon Sheffield* [EDITORIAL]  
Virtually Occupied
- 4 HEADS UP DISPLAY** [NEWS]  
Front Line Awards finalists, and top Xbox Live Indie Games
- 26 TOOL BOX** *By Jens Hauch* [REVIEW]  
Allegorithmic Substance Designer 2
- 29 THE INNER PRODUCT** *By Niklas Frykholm* [PROGRAMMING]  
Managing Coupling
- 36 DESIGN OF THE TIMES** *By Damion Schubert* [DESIGN]  
No "I" in Team
- 38 PIXEL PUSHER** *By Steve Theodore* [ART]  
Nurbstalgalia
- 41 GOOD JOB** *By Brandon Sheffield* [CAREER]  
Allen Murray moves to PopCap, who went where, and new studios
- 42 GDC NEWS** *By Staff* [NEWS]  
Top Quotes From GDC Online
- 43 THE BUSINESS** *By David Ederly* [BUSINESS]  
Evolving TRIPLE TOWN
- 44 AURAL FIXATION** *By Jesse Harlin* [SOUND]  
The New Kid in School
- 45 EDUCATED PLAY** *By Tom Curtis* [EDUCATION]  
DIG-N-RIG
- 48 ARRESTED DEVELOPMENT** *By Matthew Wasteland* [HUMOR]  
I'm Almost Done

### POST MORTEM

#### 18 GUNSTRINGER

THE GUNSTRINGER was conceived in one night, in a restaurant, in front of a Microsoft executive. From this curious beginning, the team went on to create a compelling core-friendly Kinect game, complete with a full-motion video downloadable episode, all in 12 months, without an environment artist. Former indie developer Twisted Pixel was purchased by Microsoft post-release, so they must have done something right! *By Bill Muehl*

### FEATURES

#### 6 INTUITION, EXPECTATIONS, AND CULTURE

What makes a game mechanic intuitive? How about a user interface? Game developers have grappled with these questions for years, usually relying on user tests and player feedback. This article proposes that in addition to playtests, developers should brush up on a bit of psychology to measure players' expectations against the final game. *By Ara Shirinian*

#### 13 ALL THAT GLITTERS: AN INTERVIEW WITH BUNGIE'S SENIOR GRAPHICS ARCHITECT

Hao Chen leads Bungie's graphics team, which helped bring the world of HALO to life, and now a brand new IP. We spoke in depth with Chen about trends in graphics, and revamping the company's engine for multiple platforms and the next generation of consoles. *By Brandon Sheffield*



# VIRTUALLY OCCUPIED

WHEN PROTESTS IN MMOS MIRROR REAL WORLD ACTION

**ON NOVEMBER 2ND I STOOD AT** the front of a group of thousands of people, as Occupy Oakland shut down the port in Oakland, California, the fifth-largest in the nation. People across the world are upset about the economy right now, and when they get upset, they can take that anger to the streets. Shutting down the port was a massive show of human power, against those who dictate our economy. This is precisely what happens in in-game protests, as well.

## THE EVE EXODUS

EVE ONLINE developer CCP launched a virtual item store in June, and its rollout was much maligned by EVE users. Nothing was priced reasonably, and some cost over \$60 in real world currency. Players felt as though EVE was moving further away from a skill-based game, and more toward a pay-for-play model, wherein the rich players would rule over the poor, due to having better equipment. They also decried several subtle gameplay changes that changed the user experience.

Upset fans clogged major cities, disrupting the entire game's economy, since EVE is run on a single instance where everyone plays together. This was, in effect, a massive Occupy-style protest. CCP has an elected board of player representatives, which it is supposed to consult on all major decisions. This time around, the representatives were ignored, and players caught wind of the store's impending launch through leaked memos. At the time, CCP CEO Hilmar Petursson was confident in his approach, stating in an internal memo that "Having done this for a decade, I can tell you that this is one of the moments where we look at what our players do and less of what they say."

This is sometimes the correct tactic, but your players are your lifeblood in an MMO, and without them you have no game, and no income. If players start to think that you're operating far more in your own interests than in theirs, you are

going to lose their confidence. EVE's virtual items had little perceived relevance to the game world, but cost an arm and a leg. Since that incident, a CCP source reported to Massively.com that four months later, EVE had lost 8% of its accounts.

In the Occupy movement, one of the major statements these encampments make is, "if you do not fix your system, we will attempt to live outside it." These folks have moved further off the grid, and are moving their money to credit unions instead of large banks. In MMOs, this is analogous to rage-quitting. To slightly pervert William Congreve's famous quote for the game world, "Heaven has no rage like love to hatred turned, nor hell a fury like a warrior scorned." If people didn't love the things they felt were changing for the worse, they wouldn't get so upset. In the real and virtual spaces, they're fighting for what they love.

## DON'T HATE THE PLAYER...

Monetization isn't the only issue fans get up in arms about. In WORLD OF WARCRAFT in 2005, warriors felt their class had been "nerfed" in an update. To protest, they took Ironforge Bridge on the Argent server, and removed their clothes en masse. GMs began kicking any warrior in the region from the game, but players just came back with new level one warriors, in such great numbers that GMs announced that anyone doing so would potentially have their account canceled. But were those warriors really doing anything wrong? They were upset, and they were congregating—but why shouldn't they? The GMs were shutting down a peaceful protest because it was inconvenient. This is a similar pattern to many of the arrests at Occupy protests; someone is standing around holding a sign or camera, and out of nowhere they get shot with a rubber bullet, or teargassed.

## THE BIGGER PICTURE

There's a lot that games can learn from real-world social movements.

Economists have applied their knowledge to MMOs, so why not social psychologists and political analysts?

In the real world, the Occupy protests hope to persist until government action is taken to regulate banks and promote jobs. In games, protest goals tend to be remarkably similar—players want to feel as though they're being treated with fairness. If you make decisions in your games that offer little benefit to the player, they will revolt. The best way to avoid this is prevention, though it is difficult to predict player response. A good yardstick for monetization might be for a designer to think, "would I be happy to pay for this, if this weren't my game?"

Once the damage to player confidence is done, it can be hard to undo. Players invest so much of their lives and money into MMOs that it becomes our duty as game makers to serve their interests as well as, or even above our own. There are ways to make money and please your players, which EVE is trying to do again now. As Petursson admitted in a public apology, "Somewhere along the way, I began taking success for granted. As hubris set in, I became less inclined to listen to pleas for caution." Fans left the game as a result.

There's an even greater issue with games that obfuscate monetization, and target a less tech-savvy crowd. Business persons who have no interest in games are making big bucks off first-time players. While there's not enough space to cover this subject, it may be a matter of time before these players realize what's going on and revolt, as has happened across the world with the Occupy movement. In that case, developers had better take a close look at these real-world protests as they develop and come to conclusions, and figure out how to apply those lessons in their games.

—Brandon Sheffield  
twitter: @necrosoty



UBM LLC.  
303 Second Street, Suite 900, South Tower  
San Francisco, CA 94107  
t: 415.947.6000 f: 415.947.6090

## SUBSCRIPTION SERVICES

FOR INFORMATION, ORDER QUESTIONS, AND ADDRESS CHANGES  
t: 800.250.2429 f: 847.763.9606  
e: [gamedeveloper@halldata.com](mailto:gamedeveloper@halldata.com)

FOR DIGITAL SUBSCRIPTION INFORMATION  
[www.gdmag.com/digital](http://www.gdmag.com/digital)

## EDITORIAL

**PUBLISHER**  
Simon Carless | [scarless@gdmag.com](mailto:scarless@gdmag.com)  
**EDITOR-IN-CHIEF**  
Brandon Sheffield | [bsheffield@gdmag.com](mailto:bsheffield@gdmag.com)  
**PRODUCTION EDITOR**  
Jade Kraus | [jkraus@gdmag.com](mailto:jkraus@gdmag.com)  
**ART DIRECTOR**  
Joseph Mitch | [jmitch@gdmag.com](mailto:jmitch@gdmag.com)  
**DESIGNER**  
Cliff Scorso  
**CONTRIBUTING WRITERS**

Tom Curtis  
Jesse Harlin  
Jens Hauch  
David Ederly  
Steve Theodore  
Niklas Frykholm  
Damion Schubert  
Matthew Wasteland  
**ADVISORY BOARD**  
Hal Barwood Designer-at-Large  
Mick West Independent  
Brad Bulkley Microsoft  
Clinton Keith Independent  
Brenda Brathwaite Loot Drop  
Bijan Forutanpour Sony Online Entertainment  
Mark DeLaura TH0  
Carey Chico Independent  
Mike Acton Insomniac

## ADVERTISING SALES

**GLOBAL SALES DIRECTOR**  
Aaron Murawski e: [amurawski@ubm.com](mailto:amurawski@ubm.com)  
t: 415.947.6227  
**MEDIA ACCOUNT MANAGER**  
John Malik Watson e: [jmwatson@ubm.com](mailto:jmwatson@ubm.com)  
t: 415.947.6224  
**GLOBAL ACCOUNT MANAGER, RECRUITMENT**  
Gina Gross e: [ggross@ubm.com](mailto:ggross@ubm.com)  
t: 415.947.6241  
**GLOBAL ACCOUNT MANAGER, EDUCATION**  
Rafael Vallin e: [rvallin@ubm.com](mailto:rvallin@ubm.com)  
t: 415.947.6223

## ADVERTISING PRODUCTION

**PRODUCTION MANAGER**  
Pete C. Scibilia e: [peter.scibilia@ubm.com](mailto:peter.scibilia@ubm.com)  
t: 516-562-5134

## REPRINTS

WRIGHT'S MEDIA  
Jason Pampell e: [jpampell@wrightsmedia.com](mailto:jpampell@wrightsmedia.com)  
t: 877-652-5295

## AUDIENCE DEVELOPMENT

TYSON ASSOCIATES Elaine Tyson  
e: [elaine@tysonassociates.com](mailto:elaine@tysonassociates.com)  
LIST RENTAL Merit Direct LLC  
t: 914.368.1000



UBM

# with 180 million Arabs under the age of 25\*...



## ...the gaming industry is set to boom in the Arab world.

**twofour54° Abu Dhabi** – the tax-free gateway to a new world of gamers

The MENA region is one of the world's fastest growing media and entertainment markets with 19% growth in recent years. And with 80% of under-25s owning mobile phones\*, strong broadband take-up and new gaming innovations, it's a prime opportunity for gaming businesses. Over 100 leading media companies are already capitalising on the opportunity at **twofour54° Abu Dhabi**.

- 100% company ownership in a stable, tax-free environment
- unique campus environment with facilitated business networking
- the region's only stereoscopic 3D Lab
- **twofour54°** gaming academy in partnership with Ubisoft®
- easy licensing and business set-up services
- guidance and liaison with UAE content regulatory bodies
- dedicated fund for mobile apps development via Apps Arabia™
- full on-site HD production and post-production facilities

Find out how we could help grow your business today.

**twofour54.com/gaming**  
**+9712 401 2454**



**twofour54**  
Abu Dhabi

media & entertainment hub

\*Sources: Arab Media Outlook 2010. Media on the Move 2009. A.T. Kearney. Introduction to Gaming. Michael Moore. Screen Digest. IDC.



## TOP XBLIG OF LATE 2011

EVERY SO OFTEN WE LIKE TO TAKE STOCK OF WHAT'S BEEN GOING ON IN THE WILD FRONTIERS OF XBOX LIVE INDIE GAMES. MANY OVERLOOK THE SERVICE ENTIRELY, SO WE'VE CHOSEN 10 GAMES FROM THE LAST FEW MONTHS THAT ARE AT LEAST WORTH A TRIAL.

### DEAD PIXELS (CANTSTRAFERIGHT)

In **DEAD PIXELS** you're a survivor of the zombie apocalypse, trying to fight your way to freedom while looting shops, collecting money, and upgrading your weapons and character stats. The game is tightly balanced—you can be out of ammo, trying to make it through hordes of zombies while covered in zombie goop, only to die just in front of the shop. The game also has a great DLC model—if the game hits certain sales targets, the developer starts working on new content.

### DLC QUEST (GOING LOUD STUDIOS)

This is a cute play on the nickle-and-dime in some titles, especially in free-to-play games. **DLC QUEST** is an action platformer, but if you want sound, that'll be five coins (in-game currency only, of course). Want the map that gets you through the forest? Well, that'll be 120. The game is good fun, and lasts just an hour. Going Loud decided that since there's a barrier to folks buying \$3 and \$5 games, he'd make the game "worth" \$1.

### ESCAPE GOAT (MAGICALTIMEBEAN)

**SOULCASTER** series creator MagicalTimeBean's newest title surrounds the fate of a goat imprisoned for witchcraft, a magical mouse, and several slumbering sheep. The physics and Rube Goldberg machine-based action platformer mixes tight control with excellent 16-bit-style music and graphics. Many puzzles have multiple solutions, and the robust level editor is deep and intuitive, extending the life of the game.

### FOUR WINDS FANTASY (QUIMDUNG CO. LTD)

This is one of the oddest games I've had the pleasure of playing recently. Graphics are MS Paint

sketches, the soundtrack is utterly bizarre, and the story is no less normal. You'll be talking to aliens (and a giant ogre crotch bulge), fighting plants, and generally being confused about what's happening. You play as an old man whose son may or may not be a jerk, in this very difficult but curiously compelling action RPG.

### LAST DRAGON STANDING (WEREWOLF STUDIO)

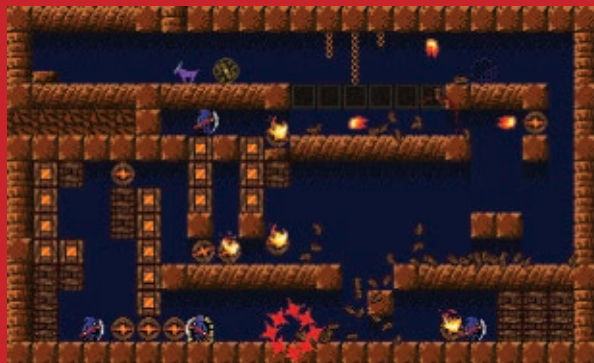
Werewolf Studio has put together a nice spin on the classic **WARLORDS** arcade game. You play as the dragons this time, defending your eggs against villagers, and against each other. Powerups spice up the action, and polished visuals and a dragon-y story keep players going. **LAST DRAGON STANDING** looks to help popularize the trend of four player, single-screen XBLIG.

### MEGA SHOOTER 11 (INFINITEPLAY)

**MEGA SHOOTER 11** is a rather faithful NES-style horizontal shooter with powerups and levels aplenty, but often it behooves you to avoid enemies entirely until you can power up your ship. It's a game that's more about survival than score, and which relies heavily on its pleasing aesthetic to pull players through. Beneath the shell, there's an interesting game of cat and mouse, as you rescue scientists and gain new powers.

### RAVENTHORNE (MILKSTONE STUDIOS)

**RAVENTHORNE** is a testosterone-fueled brawler with **METROID**-style progression. **RAVENTHORNE**'s detailed flash-style 2D graphics and ambient score give it a unique atmosphere, and the combat is deeper than it initially appears, with spells, combos, and dashes to integrate into battles. The whole game has a great energy to it—the only major failing is that the story



ESCAPE GOAT.



DEAD PIXELS.

progresses, and seems to be building toward something, but at some point just...stops.

### SINS OF THE FLESH (SILVER DOLLAR GAMES)

Silver Dollar Games has been maligned in the past (for games like **TRY NOT TO FART**), but everyone starts somewhere. **SINS OF THE FLESH** has the best voice over of any XBLIG I've heard, and has a pleasing visual aesthetic to boot. The player is a man who has recently died, whose sins are recounted as you fight off angels and demons with the left and right sticks.

### SOLAR 2 (MURUDAI)

The game begins with a big bang. You're an asteroid, floating about, trying to absorb other asteroids. Absorb enough and you become a

planet, which can later sustain life. Ships and civilizations spring forth, allowing you to take on rival planets. Ultimately you become a black hole, absorbing everything in sight. There are missions and objectives, but you can ignore those completely if you like, playing vengeful **KATAMARI** among the stars.

### WIZORB (TRIBUTE GAMES)

**WIZORB** takes the **ARKANOID** formula and adds an RPG slant to it. As a traveling wizard, you have to bring prosperity back to a ruined world, controlled by an evil force. Your **ARKANOID** paddle can use magic, so you're not just bouncing a ball about. The game also features charming animations by Paul Robertson.

—Brandon Sheffield



# gdg FRONT LINE AWARDS 2011

## FINALISTS ANNOUNCED!

Every year, *Game Developer* magazine honors the best tools in the game business; those that help us do our jobs faster, better, and sometimes even cheaper. Public nominations were open from October 10–20, 2011, and from that list the finalists were chosen, in the fields of art, audio, middleware, game engines, networking, and programming/production tools. Only five finalists are chosen in each category. After votes from *Game Developer* and Gamasutra readers are tabulated, we'll announce the winners in our January issue, alongside one tool that will receive the prestigious Hall of Fame award. For now, here they are, your 14th annual Front Line Award finalists!

### ART

- >> 3DS Max 2012 (Autodesk)
- >> MotionScan (Depth Analysis)
- >> Modo 501 (Luxology)
- >> Substance Designer 2 (Allegorithmic)
- >> ZBrush 4R2 (Pixologic)

### GAME ENGINE

- >> C4 Engine 2.6 (Terathon Software)
- >> VisionEngine (Havok)
- >> Unity 3.4.1 (Unity Technologies)
- >> Unreal Engine 3 (Epic Games)
- >> CryEngine 3 (Crytek)

### NETWORKING

- >> Photon 3 (ExitGames)
- >> GameSpy Technology (GameSpy Industries)
- >> OpenFeint (Aurora Feint)
- >> ReplicaNet 2.0 (Replica Software)
- >> DeNA Mobage (DeNA)

### AUDIO

- >> Wwise 2011.2 (Audiokinetic)
- >> Fmod Designer 3.4.8 (Firelight Technologies)
- >> Miles Sound System 9 (Rad Game Tools)
- >> Pro Tools 10 (Avid Technology)
- >> Soundminer HDv4.3 (Soundminer)

### MIDDLEWARE

- >> Scaleform GFx 3.2 (Scaleform)
- >> XaitControl 3.4 (Xaitment)
- >> Kontagent kSuite User Analytics Platform (Kontagent)
- >> Kynapse 2012 (Autodesk)
- >> Havok Physics (Havok)

### PROGRAMMING/PRODUCTION

- >> FlashDevelop 4.0.0 (FlashDevelop Project)
- >> RAD Telemetry (RAD Game Tools)
- >> Hansoft 6.6 (Hansoft)
- >> Perforce 2011.1 (Perforce Software)
- >> LUA 5.2 (LUA)

# INTUITION EXPECTATIONS AND CULTURE

A R A S H I R I N I A N

One peculiarity of video games is that we often think of them in terms of “games we are able to play” and “games we are not able to play.” Much like a sport, and unlike most other forms of consumer entertainment, video games typically demand some standard of performance ability before the player can even begin to enjoy their various workings. From the very moment we start playing a game, we develop an impression of how easy or hard a time it’s going to give us.

Some games are quite easy to understand. Regardless of whether there is an explicit tutorial, players instantly intuit what to do, what the basic rules are, what is good, what is bad, and how to go about doing the various things that can be accomplished. They feel like they’re capable of playing the game from the first moment. They don’t really expend a lot of effort figuring out how to operate the basic mechanics of the game, they “just do it,” and find themselves immediately engaged. Any problems and difficulties they do experience are intrinsic to the game.

/// Conversely, other games seem to be bewildering and obtuse. When you play those games, your capabilities are unclear, you find yourself punished for reasons you don’t understand, and you take guesses (often to find out you are wrong) about what various cues and symbols mean. You spend far more time thinking about “how to work” the game. If you’re an experienced gamer, you’ll often ask yourself questions like “What the hell is that?” or, “Why the hell is this here?”—substituting your favorite expletive for “hell.”

Of course, I have just illustrated two extreme situations, and most gameplay experiences have some mix of intuitive-feeling things and counterintuitive-feeling things. In this article, I’ll explore some reasons why certain things feel natural or intuitive, why certain other things don’t, how two people can have very different opinions of what intuitive means, and what the implications are for video game design.

## INTUITION TASTE TEST #1

/// For the sake of communicating in a more universally familiar way, but also to illustrate a more complete picture of how these dynamics work, I’m going to draw on some non-video game examples throughout this piece. The psychological ideas we explore here are indeed the same ones that govern general human interaction with interfaces. Video games just happen to be an application of these ideas, albeit one of primary importance to us.

Suppose you were presented with two different GUI arrangements. In [Figure 1a](#), we have Mystery GUI A where the Cancel button is always displayed to the left of the Save button. In [Figure 1b](#), we have Mystery GUI B where the order of the buttons is exactly reversed: The Cancel button is always displayed to the right of the Save button.

Is one of these arrangements more intuitive than the other? If so, why? You can make a reasonably compelling argument either way. Incidentally, Mystery GUI A is the Mac GUI standard, and Mystery GUI B is the Windows GUI standard. For the moment, all that we will say about this example is that it is interesting that the two most popular computer GUI interfaces use standards that are complete opposites of each other.

## INTUITION TASTE TEST #2

/// Another interesting example, and one that more readily elicits an opinion as far as intuitiveness goes, is a comparison of the QWERTY and Dvorak keyboard layouts. QWERTY, the original and oldest keyboard layout, has been in use for about 140 years. The Dvorak layout, while regarded by many as some kind of unfamiliar, novel alternative layout for hardcore typists, is still a relatively old invention, at 80 years. When I present people with these two layouts and ask them which one is more intuitive, invariably they give their nod to QWERTY. When I ask why they think QWERTY is more intuitive than Dvorak, they typically say that it’s because they are used to it.



# ON, build better game interfaces

# CTIONS,

# TURE

Ironically, Dvorak's layout results in your fingers traveling significantly shorter distances overall, because that's what it was engineered to do (see [References](#).) This ostensibly results in a higher performance ceiling than with QWERTY, and Dvorak's claim to fame has been higher typing speeds, although there have been numerous challenges to its superiority to QWERTY in various contexts.

In any case, to a naive user (someone who doesn't have any experience with any keyboard layout), there isn't much about either alternative that makes it inherently more intuitive than the other. We like QWERTY because we've been using QWERTY all our lives. We have developed expectations from our past experiences with the keyboard, and it's pleasant to know where every key is, particularly when you don't have to look. In this way, we can say that our sense of what is intuitive is at least partially a function of our personal history or training.

This personal history is very powerful; it's so powerful that in any novel situation you encounter, your mind will inevitably produce explicit and implicit expectations about it. I like to call this "cognitive baggage" for its colorful effect, but it's directly related to your personal history, past experience, and training.

In other words, your idea of what makes sense, what you understand, or what you think you can or can't do is a function of

your own personal cognitive baggage. What's more, you bring this baggage with you to every video game you ever play. When a game cooperates with your cognitive baggage nicely, you may say that the game is imminently playable or easy to learn. When a game is inconsiderate of your cognitive baggage, you might say that the game is frustrating or confusing.

## VIDEO GAME COGNITIVE BAGGAGE

/// In 2004, I was a designer on a game called THE RED STAR, which was eventually released in 2007 on PlayStation 2. Often I would present the game to a new player who had never seen the game before, but who had past experience with other games, and observe reactions without actually engaging with them while they played. Of course, it's best to do this while you are making the game instead of after the fact.

THE RED STAR is essentially a character-based melee and projectile action game comprising a linear series of levels. Some levels used a side-view camera, while others have a top-view camera. I would not always present the same starting level to a naive player, but it just so happened that the first level in the game begins with a side-view camera, while the second level in the game introduces the top-view camera.

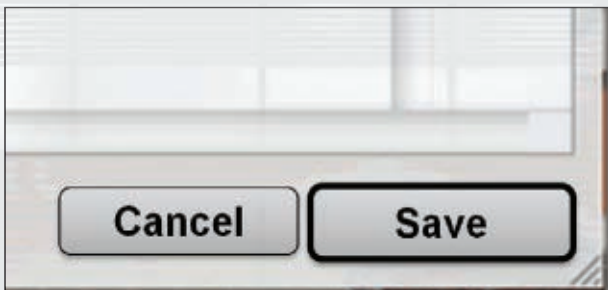


FIGURE 1A Mystery GUI A.

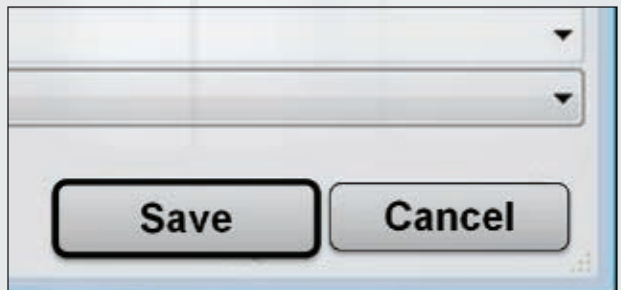


FIGURE 1B Mystery GUI B.



FIGURE 2A THE RED STAR, side-view camera section of gameplay.



FIGURE 2B THE RED STAR, top-view camera section of gameplay.

One of my most interesting discoveries was the clear pattern that emerged from players' desire and expectation to be able to make their character jump; it depended on which level they played first! Why is the expectation of jumping so critical here? Well, in THE RED STAR, there is no conventional jumping action available to the player. A few special melee attacks propel the character into the air as a side effect, but for all practical purposes, this is a game where jumping is not part of the mechanics.

If the subject played the side-view level first (see Figure 2a), it was common to hear reactions like "Where is the Jump button?" or "Why can't I jump?" or perhaps something even more colorful.

However, if the subject played the top-view level first (see Figure 2b), I never heard them make any comments about jumping one way or the other, even if they played the side-scrolling level after that.

So what's happening here? Many gamers, particularly those who grew up with the myriad 8- and 16-bit side-scrollers, carry baggage that says "Character games that are side-scrollers have a jump button." This baggage accumulates over time, after seeing many different examples with the same standard features. Conversely, apart from a few outliers, few games that feature a top-view also allow characters to jump, especially from the 8- and 16-bit eras.

Furthermore, it can be said that the side-view camera affords jumping quite well, while the top-view camera doesn't. Just by looking at the side-view screenshot, you can see the vertical space on the screen and it's very easy to imagine your character jumping. After all, most games that look like this let you jump, so this game looks like one where you

can jump. When you look at the top-view screenshot for the first time, the vertical dimension is squashed on top of itself, so you cannot really see any physical space for jumping, especially if the camera is oriented straight downward. It looks like a game where you cannot jump.

## AFFORDANCES

/// What do I mean I say that something affords something else? This particular usage was popularized by Don Norman in 1990 (see References):

"...the term affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used."

In this context, "the thing" is really any object that has a use, and its affordances are ways it can be used. Don Norman was primarily concerned with perceived affordances, and has since admonished the professional community against the increasing misuse of the word over the years (see References). To be clear, one of the original uses of the word in this context was by J.J. Gibson in 1977, but Gibson did not care about any perceived aspect of affordance. For Gibson, if an object can be used in a certain way, then it has that affordance period, regardless of whether you think it does. For our purposes, we must distinguish between perceived affordance and actual affordance.

Suppose we have a common folding chair. Most everybody is able to perceive its most common affordances: You can sit on it, you can put



FIGURE 3A Sink drawers with more perceived affordances than actual affordances.



FIGURE 3B Sink drawers with the same perceived affordances as actual ones.

things on it, you can use it as a step, and you can fold it for easy storage. Perceived affordances can also be a function of culture or cognitive baggage. If you are a professional wrestler, for example, when presented with the folding chair you are more apt to perceive a different set of affordances, like throwing, a platform for body slamming, and folding it up, but this time for ease of swinging at your opponents.

At the risk of irritating Don Norman further, it can be useful to think of affordance as a point along some continuum, even though it was never intended to be that way. For example, a bicycle affords rolling. If you build the same bicycle with square wheels, you could then say that the bicycle does not afford rolling at all. However, if you build the bicycle a third time with octagonal wheels, it still affords rolling, but perhaps not as well as the original example. In a game like GRAN TURISMO 5, every vehicle in the game affords driving in the strict, actual sense, but there is a huge difference in the drivability of a smooth front-wheel drive Honda Civic Type R versus the powerful and touchy '66 Shelby Cobra 427, or even the insane 1,500 horsepower Red Bull X2010.

## PERCEIVED VS. ACTUAL AFFORDANCE

/// In our chair example, we enumerated several perceived affordances. Coincidentally, these perceived affordances were also the actual affordances of the chair in question. But what happens when a perceived affordance is not an actual affordance?

Let's examine the fine sink installation in Figure 3a. Of interest are the three drawers just below the countertop. We only have one legitimate perceived affordance, that of pulling, because each drawer sports a

handle in the center, and we all know that handles are for pulling. When we consult reality for confirmation, we find that if we pull on either of the two side drawers, the drawers do in fact afford being pulled out. However, when we attempt to pull out the center drawer, it doesn't go anywhere. It's actually not a drawer at all, because there is a sink basin behind it and no room for a drawer. It just looks like a drawer. Its perceived affordance is pulling, but in reality it has no such affordance. There are bound to be problems when perceived affordances are different from the actual ones. In the case of sink drawer design, the problem is solved by creating a surface that does not allow any perceived affordance of pulling. In the case of Figure 3b, no one could confuse a neat relief design on a cover plate for something that could be pulled.

Returning to the camera view examples from THE RED STAR, we find that the sink drawer problem is analogous in terms of the number and type of affordances, if we trade pulling for jumping. Like the fake sink drawer, in the side-view level, a perceived affordance of jumping is frequently reported, but there is no actual such affordance. Similarly, like the relief design cover plate on the second sink, in the top-view level, there is no perceived affordance of jumping, and there is no actual affordance either.

Given the opportunity to correct mistakes, it would have been a much clearer experience for the player if we had started the game in a top-view level, and only later transitioned into a side-view camera style. Once the player has an entire level of experience with how the game works, they no longer carry the expectation or baggage that they should be able to jump in THE RED STAR's side-scrolling levels. After a level of play, the player has already learned a substantial amount about the game's actual affordances, and there is not



# GET YOUR GAME ON!

## EVERYTHING YOU NEED TO KNOW TO GET INTO THE GAME INDUSTRY!

- NEWS and FEATURES FOR STUDENTS and EDUCATORS
- GETTING STARTED section – an INVALUABLE HOW-TO GUIDE
- MESSAGE BOARDS



### DIGITAL COUNSELOR

I'LL MATCH YOUR INTERESTS and GOALS WITH THE RIGHT GAME RELATED PROGRAMS and SCHOOLS FROM AROUND THE WORLD.

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



Download your FREE digital edition of the 2011 *game developer* Game Career Guide online at: [www.gamecareerguide.com](http://www.gamecareerguide.com)





FIGURE 4A Stove burner interface with a natural mapping.



FIGURE 4B Stove burner interface without a natural mapping.

much room left to falsely perceive other affordances from similarly presented games. The big question for us was this: How many players did we lose during the first level, who did not have the patience or resilience to come to grips with the fact that it looked like a game with jumping, yet was not?

## COGNITIVE BAGGAGE VS. NATURAL MAPPINGS

/// Previous experience, training, or baggage is not the only force that gives us expectations about how things work. There is also the idea of a natural mapping, which Norman has expounded upon at length. First, let's look at his definition of regular mapping:

Mapping is a technical term meaning the relationship between two things, in this case between the controls and their movements and the results in the world.

In *SUPER MARIO BROS.*, the right arrow on the D-pad moves Mario to the right. We say that the right button is mapped to Mario moving to the right. Or, on the QWERTY keyboard, we say that the key under your left pinky when your fingers are resting on the home keys is mapped to the letter A. So, what makes a mapping natural?

## NORMAN IDENTIFIED FOUR POSSIBLE CHARACTERISTICS OF NATURAL MAPPINGS

### 1 // NATURAL MAPS MAY USE PHYSICAL OR SPATIAL ANALOGIES

Stove controls are a great example of spatial analogy. Looking at the stove in Figure 4a, how do you know which knobs control which burner? In this case the mapping is natural because the spatial arrangement of the knobs corresponds exactly with the spatial arrangement of the burners. The left knob controls the left burner; the middle knob controls the middle burner, and so on. Figure 4b illustrates an example that does not clearly use a spatial analogy. Which knob controls the upper-left burner? It could be the first knob, or the second. Or is it one of the last two? The only way to know is through trial and error.

### 2 // NATURAL MAPS MAY UTILIZE CULTURAL STANDARDS

Looking at the volume knob in Figure 5, if I asked you to turn the volume up, which way would you turn it? Most everyone would say clockwise, and of course they would be right. Even in the absence of labels, it's a generally accepted cultural standard that turning a knob clockwise means "more" and turning one counterclockwise means "less." [See *References.*] Among PC gamers, it's a cultural standard to map the WASD keys to character movement, and furthermore that particular mapping also takes partial advantage of spatial analogy [the leftmost key, W, moves you left]. However, those not accustomed to that culture, even if they are avid gamers, may have great trouble with WASD. Beyond the spatial analogy, such an input method still requires the player to internalize their muscle memory with sufficient experience before they can perform competently. Video games in general depend more on this aspect than everyday interfaces because of the time constraint: We are not often required to turn a knob in the exact right direction and amount within a fraction of a second to adjust volume, but in *ARKANOID* that's exactly what the game demands from you.

### 3 // NATURAL MAPS MAY UTILIZE "BIOLOGICAL" ASPECTS

This is perhaps the most obscure characteristic of the four. Biological aspects refer to how some mapping analogies have a biological basis—for example, "louder" naturally maps to "more." But this doesn't work with frequency of sound; we do not generally associate higher frequency sounds with "more." However, there has been some esoteric research done on this category, such as Christopher Wickens's finding [see *References*] that aircraft pilots appeared to react faster and with less error when stimulus and response processing occurred in wholly separate hemispheres of the brain [e.g., your visual target appears in your right eye and you use your right hand to control the response].

### 4 // NATURAL MAPS MAY USE PRINCIPLES OF PERCEPTION

This refers to concepts such as physical arrangements of controls where proximity or visual grouping or priority corresponds to function. For example, most car air conditioning systems have temperature controls, fan controls, and vent controls. A natural mapping of such controls in this case would imply that each of those three functional groupings would also be reflected in the physical presentation of the controls themselves; all vent controls should be together, as should all fan and temperature controls.

Now that we understand a few things about natural mappings, let's see how each of the features in Norman's list map to our QWERTY keyboard example.

1. There isn't any physical or spatial analogy to take advantage of, but numbers and letters do have a logical sequence: alphanumeric order.

The order of the numbers at the top row already use this element, and notice that most alternative keyboard layouts use the same order of numbers on the top row. The letters are a different story. Following this idea blindly, we could make a keyboard with just one row of letters, in alphabetic order, but clearly this would be a bad idea. The sheer physical awkwardness of such a result completely outweighs any advantage conferred by sticking to a logical map. The next logical alternative is an alphabetically-ordered keyboard, but spread across three rows in the traditional aspect ratio. This too we find to be suboptimal, although for the absolute naive user, an alphabetic order may confer a very slight performance advantage over an otherwise random layout.

2. The ubiquity of the QWERTY layout is a huge cultural advantage. We've been trained to use QWERTY almost exclusively, and the layout is present almost everywhere a person uses a device

to input text, except perhaps for the occasional portable or entertainment device.

3. There really isn't any biological aspect to take advantage of in this case.
4. There are limited ways to apply functional or perceptual groupings here, since most keys are generically equal in purpose and weight. Interestingly, Dvorak does group all the vowels on the left side of the middle row, although it's for two separate reasons: First, the high frequency of vowel usage coupled with the fingers' natural position on the home row is part of why Dvorak can boast such an efficient usage of finger travel. With Dvorak, roughly 70% of all letters are typed on the home row. Second, since vowels are frequently interlaced between consonants and are rarely consecutive, restricting vowel access to just one hand increases the balance of words typed with one hand versus the other. Generally, it has been regarded as optimal for performance when letters are typed with alternating hands versus on the same hand. As already mentioned, numbers on keyboards are always grouped together, and that is one sensible perceptual grouping. To a lesser extent, both Dvorak and QWERTY do the same with punctuation characters.

## PLAYING TO EXPECTATION

/// Finally, we've arrived at a point where we can make some interesting deductions about natural mappings and cognitive baggage. As we've seen with the keyboard example, not every technique to produce a natural mapping is equally available to exploit. In fact, when you are designing a control interface for a video game, you really have no guarantees that you can devise a natural mapping at all; it just depends on the idiosyncrasies of the design. Because the player's interface is an inseparable component of any game design, you may sometimes have an incredible game design that is severely compromised by the reality that there is just no great way to interface with that design. In these cases it's of paramount importance that you resist the temptation to brush off the consequence of the interface component. Your players will always experience the interface before they experience your vision of the design, and depending on the quality of the interface, they may never even have an opportunity to experience your design in the way you originally intended.

If it has not yet become apparent, a cultural standard (natural mapping characteristic #2) is really nothing more than everybody having the same cognitive baggage or training. Some cultural standards are much broader than others—the entire technologically-minded world is one culture that understands the QWERTY layout. PC gamers, who are able to identify the significance of WASD, comprise a smaller culture. Old-school shoot-'em-up game experts who all know that their avatar's hit detection box is usually very small and in the center of their character are yet an even smaller culture (although this does not diminish the effectiveness of the "small hit box avatar" technique to improve playability). What's more, cultures can change over time on every scale—even on the individual level.

In fact, cultural standards or cognitive baggage can carry so much weight, and be so deeply ingrained in a player's way of doing things, that they can often overpower all other natural mapping avenues. QWERTY's popularity and effectiveness for ease of use is primarily thanks to its cultural advantage. Natural mapping characteristics numbers 1 and 3 don't really come into play, and number 4 is better used by the Dvorak keyboard. And yet, for all Dvorak's advantages, it cannot even come close in ease of use when compared to QWERTY.



**FIGURE 5** A volume knob. Which way do you turn it to turn the volume up, and how?

Part of the reason why "observation only" playtests are so effective for game development is because of this cultural component and its strength. Playtests, when conducted without any outside influence on the subject, reveal the player's cultural knowledge and expectations, or their individual cognitive baggage. When we receive unbiased reports from players about their gameplay experience, we must understand their baggage and culture to really understand the meaning and significance of that feedback. All these things allow us to tweak our game to be more congruent with players' cultures, and that in turn allows them to more readily enjoy our games in the way we intend.

On the other hand, this does not necessarily mean the best way forward is to always maximally align with the largest cultural standards. You may keep more players in the short term, but the novelty of the experience may suffer, or you may be sacrificing another meaningful component of gameplay dynamics. Games like GUNVALKYRIE with its odd dual-analog controls, and STEEL BATTALION with its massive, button-laden controller are arguably poster children for obtuse, unexpected control schemes. Yet, for a player with patience to overcome the interface wall, each offers a uniquely rewarding method of play.

With each game we develop, we inexorably reinforce, nudge, or fight cultural standards in various ways. We are all steering the course of game development together in some direction, the consequences and tradeoffs of which may not be clear for decades. We want everybody to feel that our games are accessible to them, but in some cases in order to show the player new, meaningful, and compelling gameplay, you might have to gently break down a certain part of their culture first. 🗨

**ARA SHIRINIAN** is a game designer, currently working at DreamRift on its latest secret project.

## references

- An analysis of Qwerty vs Dvorak keyboard layouts <http://patorjk.com/keyboard-layout-analyzer/>
- Liebowitz, S.J and Margolis, Stephen E. [1990] The Fable of the Keys. Journal of Law & Economics vol. XXXIII <http://www.pub.utdallas.edu/fliiebowit/keys1.html>
- "The Design of Everyday Things" Norman, Donald A. [1990]
- Norman's further clarification of affordance [www.jnd.org/dn.mss/affordances\\_and\\_design.html](http://www.jnd.org/dn.mss/affordances_and_design.html)
- "Population stereotypes: An attempt to measure and define" Proceedings of the human factors society 25th annual meeting. p.662-665. A discussion of cultural norms: Bergum, B.O. & Bergum J.E. [1991]
- "Principles of S-C-R compatibility with spatial and verbal tasks. The role of display-control location and voice-interactive display-control interfacing" Human Factors, 26(5), 533-543. Wickens, Christopher D., Vidulich, Michael & Sandry-Garza, Diane [1984]
- "Cognitive Aspects of Skilled Typewriting" Edited by William E. Cooper. Norman on typewriting skills: Norman, Donald A. and Rumelhart, David E. [1983] Studies of Typing from the LNR Research Group.



learn / network / inspire

**GAME DEVELOPERS CONFERENCE<sup>®</sup>**

SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATES: MARCH 7-9

**2012**

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

# ALL THAT GLITTERS

BRANDON SHEFFIELD

an interview with Bungie's senior graphics architect

HALO has been a defining series for the Xbox as a major system seller, but also as a series that pushes the limits of what the console can do. A lot of this visual punch is done on the graphics engineering side, subtly decoupled from the art itself. HALO's look is unique and well-defined, but with each iteration is given greater depth and fidelity by the graphics team.

Hao Chen is Bungie's senior graphics architect, and as the team works on its newest project, the first to go multiplatform, the team has encountered new challenges, as well as confronted a host of issues that many companies previously considered solved. As Bungie prepares not only for its leap onto non-Microsoft platforms, but also for a transition into the next generation, we spoke in-depth with Chen about the future of graphics on consoles.

**BRANDON SHEFFIELD: Let's talk about the pros and cons of megatextures. Do you use them?**

**HAO CHEN:** We don't use them right now. There are two main interpretations of "megatexture." One is to store a giant texture, say 16K by 16K, and use a combination of hardware, OS, and client-side streaming to allow you to map that big texture to your world. This is cool, but we already have similar technology in places where that would be needed. For example, this is great for terrain, but we already have a very good terrain solution that uses a great LOD and paging scheme to allow us to do large and high-fidelity terrain. The other interpretation of megatextures, or the so-called sparse texture, is great for things that only have valid data in small parts of a texture. For example, if you paint a mask on terrain, then you will only have valid data where the paintbrushes touched, and everywhere else it is useless data. Sparse textures allow us to represent

these textures very efficiently without wasting precious storage and bandwidth on useless data. Another cool application of sparse texture that we are excited about is shadows. If we can render to sparse textures, then it's possible for us to render very high-resolution shadows to places that need them and still be efficient.

A disadvantage of megatextures is that on a console, we typically already know exactly what the hardware is doing, and we have very, very tight control of where the resources go. So we could do a lot of things [that are similar to the benefits provided by] megatextures already on consoles that you can't do on PC. So I guess there wouldn't really be a disadvantage, it's just that the coolness of having a texture that is automatically managed is less relevant in the console than it is on the PC.

**BS: I've certainly seen how even today people are having trouble rendering shadows without a lot of blockiness or dithering.**

**HC:** That's kind of the problem with computer game graphics these days. A lot of things people consider solved problems are actually quite far from being solved, and shadows are one of them. After all these years we don't have a very satisfactory shadow solution. They're improving; every generation of games they're improving, but they're nowhere near the perfect solution that people thought we already have.

**BS: What do you think might be the answer? Your potential megatexture solution, or something else?**

**HC:** We are still far from seeing perfect shadows. Shadows are a by-product of lighting. All frequency shadows—shadows that are hard and soft in all the right places—are a by-product of global illumination, and these things are notoriously hard in real time. There's just not enough machine power, even in today's generation or the next generation, to be able to capture that kind of fidelity. There are also inherent limitations to the current techniques, such as shadow maps, for example. When the light is near the glancing angle of a shadow receiver, then it is impossible to do the correct thing.

With the current state-of-the-art shadow techniques we can manage the resolution much better, and we can do high-quality filtering, but we still have long way to go to get where we need to be, even if we just talk about hard shadows from direct illumination. I think megatextures could help, but fundamentally there are still things you cannot solve with our current shadow meshes. And until the performance supports real-time ray tracing and global illumination, we're going to continue seeing

hack after hack for rendering shadows. Every year we see a few new hacks of current techniques, and with each hack, we see a little bit of improvement on the quality.

**BS: How about the related issue of removing jaggies from edges? I've seen a lot of discussion about different methodology to remove some of the jagginess as we get into more high-definition displays.**

**HC:** I think that's very, very critical. It's one of the big emphases on our current graphics engine, removing what we call the digital artifacts. Jaggies are definitely one of them. So we place a fairly high emphasis on removing temporal aliasing. And again, this is one area where we see a lot of recent research, and there are some things that look much better, and we're definitely looking at a few of them right now.

**BS: What looks most interesting to you?**

**HC:** I think all of the morphological antialiasing, especially some of the latest variations of MLAA with deferred rendering engines. Some recent techniques are fast enough for even the current generation of consoles. But there's also the notion of decoupling sampling from shading in some of the recent papers from ATI and NVIDIA and other companies, which is also very interesting. That's about having higher-quality AA without having to pay for the extra storage. As you know, storage and bandwidth in a console is always in premium.

There are new techniques that allow us to achieve higher-quality antialiasing, something like the equivalent of 8x or 16x MSAA, but which only requires a 2x or 4x storage. There are also things we can do to make the game look smoother, like motion blur, better filtering, and better lighting and material. I also expect quality and performance improvements to continue, perhaps by taking advantage of a GPGPU. Hopefully with all this, jaggies will be reduced to a point where you have to look very hard to find them in the next generation of games.

**BS: How about that sort of texture pop effect that has continued to plague pretty much every game with more complicated normal maps as they're paged in from the disc? How are you dealing with that?**

**HC:** Well that's more or less a resource management issue. That again is dealing with a limited amount of memory. There are ways to make it less noticeable, and they all have to do with clever blending of the textual transition or morphing into the textures as things are being paged in. And generally higher performance allows you to hide the pops until they're



Chen at work.





HALO REACH.

further away, or have better prediction of textures that are needed. But the cold reality is there's limited amount of space on the console that can store things in memory, so we need to page things in and out.

For a very large world, we also need a level of detail management so we don't draw everything at the same fidelity everywhere; these are the main reasons for "popping." But texture differences are typically only a small part of this; often changing the geometry silhouette and differences in lighting and shading of the different LODs contribute to the most jarring popping. To combat this, we have very clever LOD generation systems that try to preserve both the geometry silhouette and the material appearances. You might have seen our GDC presentation last year on our imposter system that talked about some of this, and we're still improving it. But still, when you want to have much larger worlds with much larger content, you have to page things in. So for now it's just one hack that's probably going to be better than the other hack.

**BS: How much time in your department do you spend being concerned about game performance? Can you push the boundaries as much as possible with your effects and then get reigned back in, or do you have to be constantly on top of memory management and that sort of thing?**

**HC:** You're talking about two things. One is performance, one is memory. But they kind of follow similar patterns. Our teams are typically very involved in both performance and memory, because we are the largest consumer of both. Typically the way we handle performance is we want a game to be within [a performance] ballpark around milestones. So for each milestone we set a performance target and say by this milestone, we have to be within this ballpark of this performance number. And then at the end of the milestone we will have very formal performance reviews, where we go through each level and find the performance bottlenecks and then assign teams and individuals to track those performance bottlenecks and improve performance that way.

That's the typical process we go through. But obviously a lot of performance comes from how things are designed. If you design something to be low-performance, then typically you're going to be low-performance all the way until the end. A lot of our design decisions already factor in performance from very early on. We try to solve the performance issues more as a design problem, not as a hardware optimization problem. So in terms of memory, it follows a similar pattern. We typically have an agreed-upon budget very early in the game, and then we try to make the team live by that budget, especially the content people. And every time

they make a level there's going to be a content report that tells us where they're using memory, and where they exceeded their memory budget. Then we take early enforcement action to make sure we're within ballpark. But performance and memory are some of those things that depending on the stage of your game, you cannot be overly strict about. At the earlier stage you want people to put more content in so they can get a feel for the game and explore the look of the game. As long as they're within ballpark I think we'll be okay. So we try not to be too draconian about performance and memory numbers too early.

**BS: I don't know if you have been keeping up on advances in voxelization...**

**HC:** Voxels are very, very interesting to us. For example, when we take advantage of voxelization, we basically voxelize our level and then we build these portalizations and clustering of our spaces based on the voxelization. And so voxelization, what it does is hide all the small geometry details. And in the regular data structures, it's very easy to reason out the space when it's voxelized versus dealing with individual polygons. But besides this ability, there's also the very interesting possibility for us to use voxelization or a voxelized scene to do lighting and global illumination. We have some thoughts in that area that we might research in the future, but in general I think it's a very good direction for us to think about; to use voxelization to hide all the details of the scene geometry and sort of decouple the complexity of the scene from the complexity of lighting and visibility. In that way everything becomes easier in voxelization. But as far as disadvantages, once you've figured out all this connectivity within your space that's based on voxels, how do you then map that back to the original geometry? In terms of lighting for example, if you've figured out where each voxel should be lit, how do you take that information and bake it back to the original geometry? Because you're not drawing the voxels, you're drawing the original scene. And so that mapping is actually nontrivial. So there's a lot of research that is still needed in order for voxels to be used directly in-engine.

**BS: Any word on the new project?**

**HC:** Our project right now is at a very exciting stage. For the first time we're now shipping on multiple platforms, and then the new consoles are at least not far on the horizon. We have a brand-new game, a brand-new IP. But all of these things actually present a lot of challenges. We already have a very strong graphics team, but because our ambition is so much bigger, we're also looking for very talented graphics people.

**BS: What for you have been the challenges on the graphics side, going multi-platform?**

**HC:** We re-architected our graphics engine, and the primary reason is the need to go multi-platform and get ready for the next gen. The first challenge to abstract out the platform differences and still be efficient for each platform. The other challenge is to have a good architecture for multithreaded, multicore designs that allows us to distribute work across different hardware threads, and have as many things execute in parallel as possible. We found out that even for Xbox 360, we were grossly underutilizing the CPU, mostly due to our multithreading design that doesn't allow us to spread the work and execute them in parallel. So we redid the whole architecture in the new engine. An even bigger challenge is to future proof the engine so when the next generation of consoles is here, we are already pretty good at squeezing the performance out of it. For example, we want the ability to have a particle system to run on the GPU for the 360, SPU on the PS3, and compute shaders on the future hardware, and this requires good design up front.

**BS: How much can you really know about the next platforms aside from the fact that they will generally be multithreaded? Is it difficult to predict, or do you have some insight into it already?**

**HC:** There is some confidence we can have from where the hardware is trending toward. You make educated guesses on the things that you can, and then you have isolated, potentially binding decisions on things where you have no idea what's going to happen. The worst thing you can do is design something that prevents you from being efficient on the future platforms. So on the things where we have sketchy details, we'd rather leave that isolated decision open until later than make the wrong decision. But there are plenty of things we know already, so we try to design an engine that's very efficient based on our knowledge.

**BS: From your perspective as a graphics engineer, what part do you play in the look of HALO? HALO has a unique universe, and I'm wondering how much the art director is actually involved with the graphics programmers on your side.**

**HC:** All graphics features in our game are the result of collaboration between engineers and artists. In the HALO engine, we place great importance on making things physically correct. For example, we used a photon mapping process to compute our global illumination, so in that sense, technology plays a critical role in the realism of our

games. On the other hand, our games are never just a re-creation of the physical world, and art direction plays the dominant role in the look of our games. Our worlds are more colorful than most FPS games; we have fantastic skies that you can't find in reality, and the shapes of our structures and objects conform to a consistent visual language depending on which faction they belong to. In the new engine, we place even more emphasis on giving artists direct control over how the game appears, and the majority of our R&D time was spent making technology that makes artists work more efficiently while lowering the technical requirements.

We play sort of a collaborative role with the art director. Typically, any graphics feature is a combination of the graphics engineer providing the possibilities of what can be done, and then the art director gives their preference against our vision of what should be done. So, the combination of these two typically results in a graphics feature that gets made in our engine. Our role is not necessarily to be the visionary, but we provide the possibility and educated guess of what can be done, and how their vision is best translated in terms of technology.

**BS: What do you see as the benefits of using your own engine, since you all still do that?**

**HC:** First of all, it's a lot more fun to write your own engine than license someone else's, right? But that's more of a joke. The real benefit is the flexibility, because a lot of things as we develop the game will change in the course of the game. A lot of early design decisions will turn out not to be true. A lot of things we didn't anticipate before will become big problems. So having our own engine, where we are intimately familiar with every aspect of that engine, allows us to have the flexibility to change, first of all. Second of all, nowadays we're also using a good mix of middleware to do a lot of the nitty-gritty functionality that typically would take a lot of time. So it makes even more sense for us to focus on the ones we consider to be the bread and butter, and things that give us competitive advantage. And if you adopt an engine wholesale, you lose that competitive advantage.

**BS: Do you see much in the way of middleware that actually helps what you do on the graphics side?**

**HC:** Absolutely. We see more and more roles played by middleware these days. We ended up using a few middleware packages ourselves, even



HALO REACH test.



Chen at work.

though we wrote the rest of the engine. I think good use of middleware allows us to get a feature earlier than we could if we were to engineer it ourselves, and that means the content people will get mature technology and more time to produce content. So that longer iteration time will translate to higher quality that will offset whatever engineering advantage you would have by writing your own. So in a lot of these places where it makes sense, we use middleware.

**BS: My graphics programmer friends are curious to compare how you spend your day-to-day. Are you mostly writing shaders, or doing engine development, or do you develop tools for the artists?**

**HC:** For a typical graphics engineer [at Bungie], we have about sixty-five or seventy percent of the time devoted to coding. The rest of the thirty to thirty-five percent is spent talking to the artist, educating the content people how to properly use the tools, and gathering feedback into what needs to be changed. So we spend about thirty percent of that time doing these tasks. The rest depends on the phase of the project we're in. In the beginning it's very much just core development of the technology itself, but if the feature comes to maturity, part of that coding time is devoted to bug fixing, optimization, and also tools development.

**BS: Do you work on more tools development on the front end to get people implementing stuff faster?**

**HC:** Well, we'd like to. The ideal is before you develop a feature, you figure out exactly how the content people will use it. That's the most ideal case, because then the content people have the maximum amount of time they can spend to get familiar with the tools and give feedback. But it's quite often not that ideal, because we don't know what kind of feature we need to expose until we have enough of the technology to give them a taste of what this feature is about.

Then they will say, "Hey, I really want this and that," and that will factor into the tools. So in the beginning we typically come up with very straightforward, programmerish tools, and we just give them to one or two technical art leads to try out. But before that feature goes to prime time, that's where we spend a lot of time making a very professional tools interface, so the bulk of the artists will then be able to use them.

**BS: This is a broad question, but what do you think are the big problems to solve in the coming years from your perspective, graphics-wise?**

**HC:** That definitely is a big question! I will tell you what our emphasis is for what we think is important for our graphics and our engine. Number one is removing digital artifacts. You mentioned this already—removing all the jaggies, having very clean foliage edges, and awesome-looking hair with no artifacts. Removing these digital artifacts that remind people you are staring at a computer screen is one of our top priorities. The other challenge is selling a dynamic world. In terms of what we think is important, we will even lower some of the quality in order for us to have a more dynamic world. This means dynamic time of day, lots of things that move in the wind, lots of things reacting to players moving through them, and when you walk on soft surfaces like sand and mud, you leave footprints. So basically everything we do to sell that this world is moving and dynamic is important to us.

And then perhaps the last one is believable characters. That's still one of the areas where we spend a huge amount of emphasis in animation, in the rendering of character faces and facial animation, and just characters in general. That's still one of our high emphases. In the end what we're trying to do is deliver the fidelity where it really matters to the end user. All of the stuff that I just mentioned to you is because that's the stuff that makes the player not believe they're in this game. If you see jaggies, if you've seen things that should be moving but are static, if you see a character's face and don't believe it's a real face, then the illusion is broken. So we want to spend all of this energy to try to remove these things and deliver the fidelity where it really matters.

**BS: How far away are we from having realistic hair? I know that's one of the big reasons that we have so many bald space marines in games right now.**

**HC:** Hair is interesting, because it's actually kind of easy to do very long, natural, flowy hair, because of tessellation. We can do lots and lots of strands of straight hair. With a good material map to it, and a good lighting map to it, it looks very real. And it moves very real, because we have plenty of computing power to do the simulation part. But the problem is stylized hair. So we're talking about making many different styles, from a buzz cut to choppy hair to curly hair. So those are still very difficult to do and to make look real, so it will continue to be a difficult problem I think. ☺

postmortem

# GUNSTRINGER

Some game pitches are born from extensive market research and focus group tests. Some are born from months or years of refined pitch materials. THE GUNSTRINGER pitch was born in a desperate moment of luck and creative inspiration.

///Nearly two years ago, our CCO Josh Bear and our CEO Mike Wilford were having dinner at a restaurant with Microsoft to discuss new game pitches for the recently unveiled Kinect. They had a pitch in mind before the meeting, but having recently learned about the technical capabilities of the Kinect, they suddenly realized the idea they had wasn't going to work. Publisher pitch meetings are not to be wasted, but with no backup plan ready, things were looking grim. When Cherie Lutz, one of the Microsoft reps on the greenlight committee, left the table for a break, Josh and Mike scrambled for something new. Mike remembered a marionette idea Josh had several years ago. Josh, seeing a painting of a skeleton cowboy on the wall (see Figure 1), paired the two together. When Cherie returned, Josh pitched a Kinect game based on a skeleton cowboy marionette out for revenge on his posse. In those few minutes, THE GUNSTRINGER was born, greenlit, and we were off and running.



# ER



### ///GAME DATA



**PUBLISHER**  
Microsoft Studios  
**DEVELOPER**  
Twisted Pixel Games

**NUMBER OF DEVELOPERS**

9 full time for the first 8 months, 20 full time for the remaining 4 months

**LENGTH OF DEVELOPMENT**

12 months

**LINES OF CODE**

275,000

**RELEASE DATE**

Sept 13, 2011 (US), Sept 16, 2011 (EMEA)

**SOFTWARE**

BEARD, Visual Studio, Photoshop, 3ds Max, Perforce

**PLATFORM**

Xbox 360 Kinect

**TORCHY'S TACOS CONSUMED**

960

**NUMBER OF FILM TAKES THROWN OUT BECAUSE OF LLOYD KAUFMAN PROFANITY AND/OR NUDITY**

16

**MILES FLOWN BY GUNSTRINGER MARIONETTE**

11,400

**BEST AWKWARD MOMENT**

Microsoft execs' initial reaction to The Wavy Tube Man Chronicles

# GUNSTRINGER

Before I get to the goods on THE GUNSTRINGER, let's do a quick primer on Twisted Pixel. We've shipped four XBLA games since the company was founded in 2006: THE MAW, 'SPLOSION MAN, COMIC JUMPER, and MS.

'SPLOSION MAN. THE GUNSTRINGER continues our tradition of character-driven original IP. All five games have been published by Microsoft for the Xbox 360 and, with the recent Microsoft acquisition of Twisted Pixel, we'll deliver many more for the Xbox platform.

Given the shotgun start for THE GUNSTRINGER and an aggressive dev schedule, we had our work cut out for us. This is the story of Twisted Pixel developing our fourth new IP through the wild ride of company growth, new platform tech, mid-project shifts from digital download to retail, and a couple live-action short films...all in 12 months.

## WHAT WENT RIGHT

### 1 game as a stage play.

///The main character of THE GUNSTRINGER is a mean-looking marionette in his own right, but we put serious effort into surrounding him with tons of details to make the player feel like they are performing in a fully-formed stage play. Our research and reference hunt during pre-production and concept gave us an excuse to revisit all the classic toys we played with as kids. On the art side, everything needed to look like a handcrafted puppet or play prop. Things like outlaws with arms and legs made of yarn, beer can cows, corrugated edges on cardboard trees, and toy model houses with peeling sticker siding extend the theme. The characters needed to look handmade, but it was just as important that they moved in ways that made them believable puppets. As the lead character, the Gunstringer got the most animation love. James Clark, our animator, created over 800 animations for him, nailing the right balance between a puppet's slight clumsiness and a tough cowboy out for revenge.

We also kept our full-motion-video streak alive with some in-game elements to remind players that they were performing on a stage. For example, we shot several dozen variations of hands and arms entering the camera frame in front of a green screen. These shots were then keyed out to make it look like the hands and arms were interacting with in-game characters and assets. In the first Western-themed act, a huge arm slams down across the game path, causing a stagecoach to careen off the road and take out a bunch of innocent puppets. This solid, manly display of muscle belongs to Tony Goodwin, our character artist. If you want more pictures of Tony flexing, you should email him at [tony@twistedpixelgames.com](mailto:tony@twistedpixelgames.com).

We also wanted to make a seamless transition from the intro video, which establishes the play in a real theater with a real marionette, and the first interactive moment for the player. The intro tracks a woman walking into a theater and follows her until she takes her seat. The video then cuts to the backstage area, where the player gets their first look at the Gunstringer character. In this instance, the character is a real marionette, carried from backstage as several stagehands (the Twisted Pixel crew) make last-minute preparations before the show, and the marionette is buried on stage and the lights go down. We then transition from the Bink video to in-engine; the player is prompted to "resurrect" the Gunstringer with a lift of the hand, and then they take control. Creating this intro and connecting it to the first interactive moment took a lot of planning and coordination across all departments, but it was a key part of the thematic setup.

The real marionette was a mini-production itself: it had to look as close as possible to the in-game model. To achieve this level of detail, we worked with Puppet Kitchen, a company in New York City founded by a few puppet design vets, including a guy who designed and built puppets for The Jim Henson Company. Of course, repeatedly burying



Our full-sized Gunstringer puppet.

this \$6,000 marionette in a pile of heavy shredded rubber over multiple takes was pretty nerve-wracking, but thankfully it held up to the abuse without any problems.

Finally, the audio was just as important as the graphical elements to sell the stage-play aesthetic. Chainsaw, who represents our entire audio department, carried all of it. His original score was complemented by a huge variety of crowd samples he recorded at Austin's Paramount Theater, along with a ton of his authentic homebrew foley and narration performed by R. Bruce Elliott. When combined, these pieces created an awesome Western stage-play sound that only an audio master like Chainsaw could create.

### 2 designing to the strengths of Kinect.

///Microsoft took a massive leap when they decided that Kinect would abandon controllers completely. Based on their high sales numbers, the leap seems to have paid off. We now know Kinect's launch was a hit, and many developers have a good understanding of how to develop for it. That said, when Kinect was first announced, it was really a wildcard.

In pre-production, there were a few elements of design that we knew we wanted to accomplish. First, the team wanted to make a motion-controlled game that we would enjoy playing. While some game studios are extremely talented at dancing and fitness, Twisted Pixel is not. We trend more toward the "core" camp. Second, we wanted to make the players feel like they were controlling a real puppet with their hands. Finally, we wanted to support players using Kinect to play THE GUNSTRINGER sitting down (that fitness-oriented thing, again).

When we began prototyping with pre-release Kinect hardware, these three goals shaped a lot of design decisions and gesture controls. The sitting-down goal immediately limited us to use of the upper body, and it helped us focus on simplicity. During the early stages of

development, Dan Teasdale made it his personal design mission to dive into the Kinect hardware and tools to learn what makes it tick. He spent weeks bathed in infrared dots, finding the edges of the new hardware's capabilities to figure out the best balance of ensuring solid gesture detection and maintaining the puppet master gesture feel. Dan's Kinect design work at the beginning of the project provided the foundation upon which we built our core gameplay mechanics in this new controller-free development world. While early prototype tests included complex and overloaded arm movements (pattern matching, repetition, or velocity to differentiate between two similar motions), they quickly fell to the wayside if they didn't "just work" for a casual player.

Core movement and mechanic design also benefited from a long-standing tradition at Twisted Pixel: using other games as references. Everyone at Twisted Pixel has a history with and a deep appreciation for games. The games we talk about most are often the ones that established solid game mechanics, and we're not shy about referencing them if they help the team understand the goals of the game. For *THE GUNSTRINGER*, games that we often talked about included *CRASH BANDICOOT*, for its third-person traversal, and *PANZER DRAGON* or *REZ* for their "painting" targeting system.

After some experimentation, we found that controlling the Gunstringer's movement on one hand and using the other for target/fire gave us the best balance of accessibility. This design also had a strong correlation to the motions used to manipulate a real puppet. For example, the on-screen handle floating above the Gunstringer's head corresponds to the position of the player's left hand and pulls the character to that position. The reticle corresponds to the position of the player's right hand, as if they were guiding the character's pistol-packing hand on a string. Once we had this base mechanic established, it became obvious that things should be mixed up a bit to reduce fatigue. A few of these variations included platforming segments, which allow players to rest one arm, dual-gun sections that eliminate the firing motion, and melee segments that give the player a chance to punch and pummel with reckless abandon. While individual gestures may be simple on their own, we found we could mix and match them with variations in enemies, locations, difficulty, and animations to create a variety of gameplay scenarios, maximize Kinect detection, and reduce fatigue.

### 3 joining forces to hit retail.

///When we first got word that Microsoft wanted us to shift *THE GUNSTRINGER* from XBLA to a retail release, it was huge news for us. Most of us have shipped retail games when we were at other companies, but this was an opportunity for us to ship the first retail game as Twisted Pixel. On top of that, it was a huge vote of confidence for the game from Microsoft. When

this chance is staring you in the face, the knee-jerk reaction is to say, "Are you kidding? Of course we'll do it!" That said, we knew this wasn't a decision to take lightly, considering it would be one hell of an undertaking for a company of our relatively small size. There was no way our core team of nine people was going to finish the game as a retail title in the four months we had remaining to our certification date. We needed the whole company to pitch in.

The majority of the people on the initial *GUNSTRINGER* team were new to Twisted Pixel, and the injection of the guys from the *MS. SPLOSION MAN* team was like a special ops team dropping in from the sky with much-needed firepower. They had just come out of a big push to finish their project but were willing to jump onboard to help. Veteran additions in several areas across art, programming, and design instantly doubled manpower, and having that extra help gave us several benefits.

First, they knew *BEARD* (our internal engine and editor) as well as they could grow one, and they jumped in without missing a beat. Second, there were now fresh eyes on the game. Some elements we originally thought were too tricky or hadn't considered were soon made possible, and they made the game better. The shotgun and flamethrower are two examples of things that wouldn't have shipped if we hadn't gotten the influx of help. Third, together we could simply make more stuff and polish what had already been made, including enemy behaviors, animations, and level design content. The home stretch was far from easy, but we finished *THE GUNSTRINGER* together and the game was a lot better for it.

### 4 content pipeline.

///Starting last summer, we began hiring with the goal of nearly doubling the size of Twisted Pixel to support two projects simultaneously. *THE GUNSTRINGER* and *MS. SPLOSION MAN* were the first games developed with this company setup. Both teams were (and still are) pretty small, with approximately eight to ten people for each project. We keep things lean and always push to have a ton of content in our games, which means our artists often carry an entire content "department" themselves.

*THE GUNSTRINGER* project was allocated one animator, one character artist, and one environment artist. Again, pretty lean stuff for console development, but we had a solid plan in place. Four months into the project, our environment artist left the company (side note: the same thing happened to the *MS. SPLOSION MAN* team two weeks later. I think Tony scares them away). Rather than scramble to look for and ramp up a new hire, I decided to make all our backgrounds into full-motion video, like the Sega CD masterpiece *SILPHEED*. Actually, that's not true, I only thought of that now, and I'm sad I didn't think of that at the time. What really happened is that



# GUNSTRING



Troma studios' Lloyd Kaufman acted in the Wavy Tube Man Chronicles DLC.

we moved forward without an environment artist. Josh made the same call on MS. SPLOSION MAN. One environment artist per team was already pretty aggressive, but going down to zero is close to lunacy. The only way we could pull it off was by doubling down on art outsourcing. To make that work, we'd need awesome partners and an extremely efficient outsourcing pipeline.

Luckily, the first part was easy; we had established relationships with super talented and reliable outsourcers on previous projects, including Virtuos and CGBot, and they were ready to step up and help us out. The second part was where a lot of up-front planning would make the difference between success and a massive content disaster.

We did a first pass on the assets needed for each level in the game and prioritized them based on design and tech dependencies. Once we had these priorities figured out, our two amazing concept artists, Brandon Ford and Ted Pendergraft, started cranking away. Once the concepts were approved, each asset got its own pack of info for the outsourcer, including the concept art, specific measurements, technical requirements, texture references, naming conventions, and a description of how the asset was going to be used in the game. While there were a few times we had to do an iteration on an asset, for the most part, the thoroughness of these packs resulted in quality assets on delivery, saving us a ton of time and headaches. If you've been down the outsourcing road, you know that any time external assets arrive in a state as good as what an internal artist could achieve, outsourcing is a win.

Finally, since we didn't have an environment artist to bridge the gap between outsourced content and our designers working in the BEARD editor, the assets had to "just work" when they came back from the outsourcers. BEARD, by the way, was created by Frank Wilson, our supreme CTO. Frank can shape the engine and toolset like putty in his soft hands, and he rapidly wrote a script that the outsourcers could run to make each asset BEARD-ready. Although there were a few instances where we had to make tweaks to a prop, when assets came in from China, our designers could generally grab them and toss them in the game editor. I don't think Frank knows Chinese, but his magic BEARD script knew no cultural or language boundaries during development.

## 5 fmv—bringing back the '90s.

///I'm sure some readers will say anything related to full-motion video (FMV) in a video game belongs in the "What Went Wrong" section, which is fine, but they're wrong. I personally have more love for Sega CD and 3DO than is healthy or rational. Anyone familiar with our games knows we're fans of using FMV, and it was a natural fit for the premise of THE GUNSTRINGER, because it opened up options to reinforce the idea that the player is performing.

Despite the intentionally campy quality of the FMV elements in the game, these productions take massive amounts of logistics and preparation, especially since they are all shot on location. While green-screen elements take a lot of work, such as the hands that drop into the main gameplay sequences, the Paramount Theater and Wavy Tube Man Chronicles DLC productions were absolute beasts. These all had to be written, planned, cast, set up, shot, edited, and integrated into the game while we were making the main game itself. In retrospect, I can't believe we pulled it off, because they added a huge amount of complexity and risk, but they all made the final game a lot stronger.

The Paramount Theater is a historic building in downtown Austin, and it was far and away our first choice to introduce the player to the stage-play theme at the start of the game. The theater is nearly 100 years old, with a fantastic classic look and feel inside and out. Once we had all the pre-production work settled for this shoot, the biggest challenge was figuring out how to recruit and organize hundreds of volunteer extras to make the theater look populated. I initially sent open casting calls to fellow Austin game developers and local universities, but I only got a few dozen responses. I needed more. A lot more. So I turned to the ultimate bastion of high risk/high reward to reach a massive number of candidates: Craigslist. Once I posted the vague casting call as part of an unannounced Twisted Pixel game, the floodgates opened. In a matter of two days, I received over 200 emails from strangers, some who lived hours away, and all were excited to be part of our production.

We had no idea what to expect on the day of the shoot, but everyone who came was impossibly helpful and polite, especially during the downtime between takes. We are extremely grateful for everyone who showed up for the Paramount Theater shoot; the GUNSTRINGER intro, finale, and crowd-reaction shots would not have been possible without them.

The Wavy Tube Man Chronicles DLC was the second big FMV production for GUNSTRINGER. Josh and I talked about going all-in with a modern interactive FMV game for a long time, but we knew the odds of being able to make one for a





# ER

wide audience would be next to impossible. Who in their right mind would fund a game genre that's been dead for over 10 years? Once the retail option and budget for a few DLC packs were in the cards, I knew this might be our only chance to pull it off, and I thought an homage to MAD DOG MCCREEE was a natural fit [or a close enough fit] to the Western theme of THE GUNSTRINGER.

The Wavy Tube Man Chronicles production was insane. We shot a 40-minute movie in three days, and based on when the DLC had to be ready for cert, I had to schedule the shoot two weeks before our GUNSTRINGER zero-bug release (ZBR) milestone. The timing couldn't have been worse, but it was the only way it was going to happen. With this compressed timeframe, I only had five weeks of pre-production to write a script, plan all the shot setups and stunts, cast the talent, find and rent the locations, and hire a bunch of crews. We needed a licensed pyro FX crew, stunt crews with horses, a helicopter and pilot, as well as hair, makeup, and costume departments. Of course, the spot-on weather forecast for all three days was a brutal 102 degrees.

It can be tricky to make a game or film deliberately campy without driving it into the ditch but, believe it or not, having seasoned actors, stuntmen, costume people, and a veteran hair and makeup crew played a big part in pulling it off. They were all extremely professional and took every direction Josh and I gave them, even if they thought we were out of our minds based on the bizarre content and pace we were shooting.

Two quick stories about the shoot: First, I was talking to Darylin Nagy, our makeup artist, about her experiences on film sets she had recently worked on. She told a couple short stories and casually mentioned that she had most recently been the head of the makeup department for *The Tree of Life*. Darylin proceeded to talk about her team doing hair and makeup for Brad Pitt and Sean Penn. I had seen her IMDB profile so I knew her professional history, but hearing her say that only minutes before she made up the face of one of our \$20 inflatable dolls was surreal. Your move, Terrence Malick!

Second, Laura King, one of the awesome stunt people on set, told Josh and me that she had worked a set like ours before. We thought she meant some Western

movie, but she said no, it was a similar set and that they filmed victory—and death—reaction shots for most of the actors, just like we were doing. Then she asked us if we had heard of MAD DOG MCCREEE. It turned out that she was on-set in 1989 while her ex-husband was a cowboy actor in the original interactive laser disc. Unbelievable.

## WHAT WENT WRONG

### 1 the double punch of a new team + new platform.

///As mentioned, Twisted Pixel doubled in size since the summer of 2010. There was a slim window of time between projects when it would have been perfect to increase full-time headcount. However, recruiting, interviewing, and hiring is never predictable, so we knew we'd have to roll with the punches until we found the right candidates. On my official start date, I needed to fill three positions, or a third of the team. The hunt was on for our animator, an environment artist, and a third programmer.

We found an environment artist a month into the project, but as I mentioned earlier, he left after a few months, so the time spent getting him up to speed was largely lost. We had many applicants for the animator and programmer positions, but none were the right blend of talent and personality. It wasn't until nearly three months into our twelve-month dev cycle that we found the two guys who were perfect for the animator and programmer positions. Time spent developing without key roles filled was a huge drag, but in the long run, waiting for the right person was much better than hiring out of desperation. Once we had all the positions filled, we knew we had a strong team. Still, the vast majority of the people on the team were new to Twisted Pixel, and it would take time for everything to gel.

The challenge of late hires and a new team was compounded by development for a new platform. Even though most of us had shipped several titles, we were all accustomed to starting out with the same thing: a controller. Kinect wasn't just a graphical or memory upgrade, it was a radically new way to make games. It was a double-edged sword, because we were excited about the chance to work with something so new and different, but it came at the price of increased iteration time for gameplay mechanics, gesture detection, difficulty balance, and UI navigation on an already tight schedule.

### 2 late vertical slice.

///The term vertical slice sounds corporate and out of place at our company, but whatever you want to call it, every game eventually needs to get to a point where one of the levels, tracks, or maps sets the bar for how the rest of the game will play, look, and sound. To put it bluntly: when the calendar hit what was supposed to be our vertical slice milestone date, playing the game felt like a series of functional core systems rather than a cohesive gameplay experience. I had been overly focused on managing the massive content push. This meant that the general art and audio content were tracking well, but these things came at the price of neglecting important design elements. All the little things that add up to make a game engaging, like gameplay transitions, pacing, cameras, and threat balance, needed a lot of work. Josh's creative direction at this stage was critical to reach a true vertical slice that would be the basis for building the rest of the game.

Once the vertical slice stabilized, we were several weeks behind. Of course, every week the vertical slice milestone slips equals one less week for the production, alpha, beta, or cert milestones. Since there's only so much compression that could be done on the last three milestones, the production milestone ended up taking the hit. While the switch to retail bought us an extra six weeks or so, it carried the burden of additional content to justify the retail price. Shifting the whole company to work on the project helped immensely, but in the end, the last few months of the project were a pretty mean crunch. Not fun.



# GUNSTRINGER



Figure 1: The skeleton that inspired *The Gunstringer*.

### 3 two teams with lock-step development schedules.

///It was a big step for Twisted Pixel to grow to support two development teams, from both from business and creative standpoints. While the company is now built for two teams, it doesn't mean they operate independently. Each game team has a core group that works exclusively on one project, and each core group is supported by a handful of people who split time across both. Prior to the retail shift, the original milestones of *MS. SPLISION MAN* and *THE GUNSTRINGER* were very closely aligned. This arrangement put a lot of pressure on us to figure out a reasonable balance for people working on both projects.

At the early and middle stages of the projects, our all-star concept artists, Brandon and Ted, had to work like mad to keep up with our internal and outsourced content pipelines. Even though they're both supremely talented, we kept raising the bar for content, and there was too much work hitting them simultaneously. Each project went through phases where we had to either tough out a dry spell so the other project could get fed, or simplify some of the asset designs. Similarly, the double project slam put Chainsaw in a really tough spot. He's an absolute beast, but he was creating, managing, and implementing all audio for two very ambitious games. In a perfect world, each team's dev schedule would be staggered so that their critical milestones are outside the blast radius of each other. While it will never be possible to even things out perfectly, the experience of going through the madness of such closely timed projects will help us plan better on future milestone schedules.

### 4 prototyping with the first level.

///When we're developing our games, the level that comes out of the prototyping period is functional. Under the surface, however, it's pretty much cobbled together because it's the test bed that helps us figure out what the hell we're doing. The art, design, tech systems, and features that rise to the top during prototyping are built on earlier experiments and carry the baggage of iteration.

As we started prototyping *GUNSTRINGER*, we started on the Western-themed play because it was the easiest style for everyone to wrap their head around; the other plays were themed on the less traditional bayou, samurai western, and Day of the Dead, and they needed to be fleshed out more. We got to work on the first Western level to prove our gameplay mechanics, animations, level design rules, scripted moments,

cinemas, gameplay transitions, and everything else that we needed to figure out for our prototype. While the prototype moved forward, Frank was upgrading our terrain system with amazing new BEARD powers. Unfortunately, once the upgrade was done, there was no easy way to convert the terrain and everything else in the prototype level to the new and improved BEARD system. Of course, by this point, the prototype level was taking shape as a "real" level, so we'd either have to live with its legacy issues or rebuild it in the new system.

Time was not on our side, so we decided we'd live with it, knowing full well it would be the level that would give us the most headaches when we needed to change anything or started pushing it hard during QA. This held true. The icing on the cake was that this Western-themed *Frankenstein* turned out to be the best choice to ease players into the game's style, themes, and narrative, which meant it should be the first [non-tutorial] level of the game. Although this level shipped with some polish, our later levels were built with a better foundation and were much easier to work with. Throwing away the prototype level or, at the very least, prototyping with one of the later levels in the game is the easiest "what went wrong" issue to fix, and we're not falling into this trap again.

### 5 boss battles slipped to the end.


///When we knew we were making the shift to retail, we had to make a choice that would impact scope. We had enough time and resources to either add more content and polish to the main game or blow out our boss battles, but not both. The choice to add more content and polish was necessary, but it was painful to have such awesome boss character designs and lose the opportunity to showcase each of them in a unique way.

Although we had extremely rough prototypes of a unique battle for each boss, they were way too ambitious for the remaining schedule and had to be scrapped. Josh came up with a novel MST3K-style presentation for the boss battles that would look and feel different than any other part of the game. The downside was that, other than the Wavy Tube Man battle at the beginning and the end, the boss battles are all pretty similar. The fact that the boss battles that shipped in the final game are functional and cool in their own stylized way is a minor miracle considering how little time was left to make them.

### THE NEXT RIDE

///Like most game studios, we all poured everything we had into *THE GUNSTRINGER*. Although our studio growth and path to retail was challenging and unorthodox, it was another big step for us as a company.

Speaking of big steps, I'm writing this only two weeks after the ink dried on the paperwork that formalized Microsoft's acquisition of Twisted Pixel. We're all extremely excited about what this relationship is going to allow us to do next. The additional support and resources that come along with becoming a first-party studio will help us make better games and refine our development process, but it won't solve everything. The way we see it, the best way to improve as a studio is to keep making games and continue to try to fix what's broken each time.

And to all those calling for a new *NIGHT TRAP*, be careful what you wish for. 

**BILL MUEHL** joined Twisted Pixel Games as game director for the *GUNSTRINGER* project in 2010. He's been in the game industry for nearly 10 years and worked with the Twisted Pixel founders back when the company was just a twinkle in their eye. They sustained a long distance relationship for several years until Bill finally stopped fighting destiny and moved to Austin to join them.

# GAME DEVELOPER MAGAZINE

the best of  
postmortems,  
product reviews,  
and standout  
columns

# gd

NOW AVAILABLE FOR  
DIGITAL DOWNLOAD AND  
FOR IOS DEVICES.

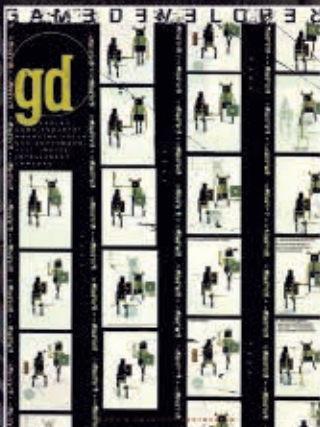
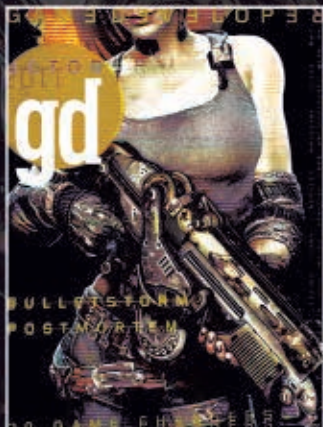


DOWNLOAD THE GAME  
DEVELOPER APP

[bit.ly/gdmag\\_ios](http://bit.ly/gdmag_ios)

SUBSCRIBE TODAY!

[WWW.GDMAG.COM/SUBSCRIBE](http://WWW.GDMAG.COM/SUBSCRIBE)





# ALLEGORITHMIC Substance Designer 2

**TEXTURE MAPPING HAS COME A LONG WAY SINCE DR. EDWIN CATMULL DEvised THE PROCESS** in 1974. Its usage in real time is now commonplace, and the tools and techniques are very sophisticated. There are a few exceptions, though, one of which is the way textures are animated. That is still mainly accomplished using the same visual trickery zoetrope employed almost two thousand years ago. We are due for an upgrade. The latest version of Substance Designer from Allegorithmic attempts to do just that—and a whole lot more.

For those unfamiliar with Substance Designer, it creates “substances,” which Allegorithmic refers to as smart textures. These differ from bitmap textures in that they are parametric and dynamic (you can completely change the look of a texture by adjusting or animating exposed parameters), lightweight (usually only a few KB), resolution-independent (32 to 2048), and can include inputs that turn them into “smart filters” (think Crazy Bump or Filter Forge). It is a mature product, and it is the current evolution of Allegorithmic’s procedural texturing software that was previously called MapZone.

## WHAT’S NEW

» With the arrival of version 2, which now supports Mac OSX, Allegorithmic has rewritten the scalable vector graphic (SVG) editor, and added a layer-based compositing system. Also included in the release are a host of smaller features and bug fixes.

The rewritten SVG editor now includes multiple colors, a new brush tool, and the ability to import SVG files. The brush tool is a nice addition, but it falls a bit short. While it gives you the ability to paint and perform Boolean quick shapes, don’t expect to create complex masks without a lot of fiddling. This is mainly due to the fact that the shapes the brush creates are fairly inaccurate and that the software lacks basic vector editing capabilities. While it is unfair to expect Designer to do everything that Illustrator or other dedicated vector editors can, it certainly needs more of the fundamentals. As an example, you can’t quickly switch between handle selection and editing mode.

The SVG-import feature is a welcome addition that allows complex shapes to be imported, but again, Illustrator format support would be nice to have since most of the popular image editing packages can’t export SVG.

New to Designer is a layers-based interface that lets you composite substances. This allows you to compartmentalize

complex substance graphs into more workable chunks, and then blend them all together using masks and layer blending modes. The graph gets noticeably slower when you start combining multiple substances, so you will want to plan accordingly when using this feature.

The UI continues to be very pleasant to interact with, and anyone familiar with Maya’s Hypergraph or Unreal’s material editor will feel right

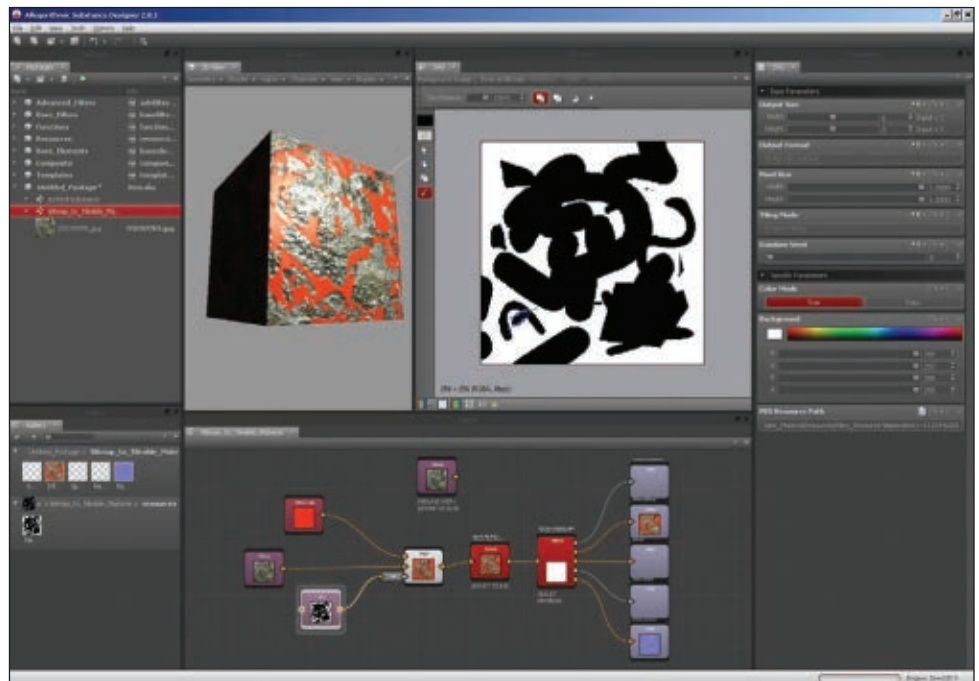
at home. The normalizing sliders are really nice, especially since there are so many value ranges that substances can use. The ability to toggle tiling view on and off makes it easy to check timing in the 2D view. Newly created nodes in the graph auto-connect to what you have selected, which speeds up creation time. The UI does have a couple oddities, however. First, you need to press tab twice to move between

value fields, since the focus moves out to the UI control. Second, the color picker needs to be closed to dedicate a color choice. These are small irritations in the big picture, which will hopefully be resolved in future releases.

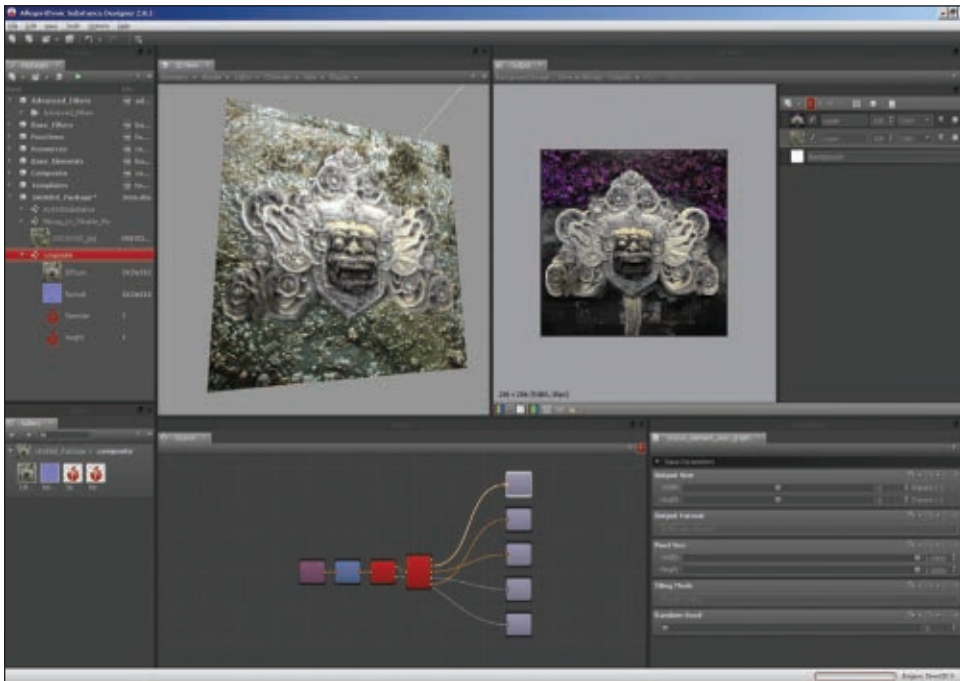
## NOVEL TEXTURES

» Designer’s learning curve has always been pretty steep, but for technical artists who are used to editing shaders in a graph environment, it shouldn’t take too long to get comfortable.

Designer includes a vast array of filters, functions, and elements (patterns and noises) with which to create or alter existing substances. A lot of these effects are very difficult or expensive to achieve with High Level Shader Language (HLSL) vertex or pixel shaders. This is one of the main things that makes Designer so powerful and intriguing.



Designer’s main view.



## ALLEGORITHMIC Substance Designer 2 (Windows, 2.01 build 6946)

www.allegorithmic.com

### PRICE

> \$990

### SYSTEM REQUIREMENTS

> OS: Windows XP, Vista, 7 (all editions), CPU: x86 with SSE2 GPU (Windows only): DX9, DX10 - shader model 3.0 minimum Video RAM: 512MB minimum RAM: 4GB recommended

### PROS

- 1 Organic tiled noise and pattern creation is unmatched
- 2 Layering expands the flexibility and dynamic nature of substances
- 3 Space savings of substances continues to be astonishing

### CONS

- 1 Expensive. \$990 puts it above Photoshop and Genetica Pro
- 2 SVG editor is cumbersome to use
- 3 Hard to justify using without the Air texture engine

Also included is a library of 130 sample substances (more are available from Allegorithmic's store), and a vast array of extensions can easily be dropped into Designer, such as Seamless Random Noise and the Grunge Maps Toolkit.

Substance Designer can also create textures from scratch. Everything done in Designer is nonlinear, nondestructive, and except for external bitmaps, all results are stored in the substance itself. Substances are kept in packages and can be instanced into other substances, which allows you to manage a very large network of textures that can be updated en masse if needed. The non-linear workflow allows upstream changes to cascade down to all your outputs, so changes can be made very quickly. Once you get comfortable with the graph, you can create variations of existing substances in no time at all.

### SUBSTANCE SUPPORT

>> Outside of game engines, substances are currently supported in Maya, Max, and Flame. This review was done with Maya,

but features are very similar in Max. Using substances in Maya is straightforward but a little cumbersome. CgFx shaders are provided that allow you to attach substances to file 2D nodes, when partnered with mel scripts. These are then attached to the CgFx shaders, allowing you to see the results in Maya's viewport. While this is not the most optimal workflow, it does show that substances can be used with hardware shaders in DCC apps for eventual export into a game engine. Since Flame is not typically used to create in-game assets, I didn't evaluate its substance support.

### MATTER MATTERS

>> Now that we have our substances, there are effectively two options: export textures as traditional bitmaps using the free Allegorithmic player, or use the substances directly in an engine that's running Allegorithmic's Air texture engine. The latter is where substances really shine. With Air, substance textures are generated at run time and can be edited in-game by updating their parameters to

create dynamic effects. Also, because substances are so compact, games using Air can reduce their texture package size by up to 99%.

Using substances in a game engine is done by exporting asset files from a supported 3D application and/or directly assigning them to shaders in a game editor. There is a catch, though: you must license Allegorithmic's Air texture engine or use a commercial engine that supports the substance format. At the time of this review, substances drop right into Unity 3.4 and ShiVa, with additional unannounced engine support forthcoming in the next few months. Be sure to check out the "Airstream" tech demo that showcases substances in Unity.


### DECIDING ON DESIGNER

>> Unless you leverage the run-time benefit of substances with Allegorithmic's Air, the cost and learning curve of Designer won't seem worthwhile, since you will need to eventually save everything as traditional bitmaps before you can get

them into your game. You are probably better off just using Allegorithmic's B2M (Bitmap to Material) and à la carte substances purchased from Allegorithmic's online store.

On the other hand, if you're using an engine that runs Air, Designer is a must-have, since you are going to want to customize substances to fit your needs.

Things could also get really exciting for substances if they supported a UV-less workflow (like Ptex). I put this question to Allegorithmic, and they said it is in their "mid-term" plan and is currently in a prototype stage.

Is Designer 2 worth checking out? Absolutely. At the very least, you should be aware of what Designer is capable of, and try to leverage its abilities at your studio. While many of the things it does are not yet ubiquitous, it has the potential to set a new standard for how game textures are created. 

**JENS HAUCH** has been cranking out pixels in the game industry since 1997. He is currently the technical art and FX lead at ArenaNet, working on GUILD WARS 2.



take  
control  
of your  
future

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

the art and business of making games



# MANAGING COUPLING

## TIPS FOR KEEPING SYSTEMS AS INDEPENDENT AS POSSIBLE

The only way to stay sane while writing a large, complex software system is to look at it as a collection of smaller, simpler systems. And this is only possible if the individual components are properly decoupled.

Ideally, each system should be completely isolated. The effect system should be the only system manipulating effects, and it shouldn't do anything else. It should have its own `update()` call just for updating effects. No other system should affect how the effects are stored in memory, or what parts of the update happen on the CPU, SPU, or GPU. A new programmer wanting to understand the system should only have to look at the files in the `effect_system` directory. It should be possible to optimize, rewrite, or drop the entire system without affecting any other code.

Of course, such complete isolation is not really possible. If anything interesting is going to happen, different systems will at some point have to talk to one another, whether we like it or not.

The main challenge of keeping an engine healthy over time is in finding ways to allow these necessary interactions to take place while still keeping the systems as decoupled as possible. When a system is properly decoupled, adding features is simple. Want a wind effect in your particle system? Just write it. It's just code. It shouldn't take more than a day. However, if you are working in a tightly coupled project, such seemingly simple changes can stretch out into nightmarish day-long debugging marathons.

If you ever get the feeling that you would prefer to test out an idea in a simple toy project rather than in "the real engine," that's a clear sign that you have too much coupling.

Sometimes, engines start out decoupled, but as deadlines approach and features are requested that don't fit the well-designed APIs, programmers get tempted to open back doors between systems and introduce couplings that shouldn't really be there. Slowly, through this "coupling creep," the quality of the code deteriorates and the engine becomes less and less pleasant to work with.

Still, programmers cannot lock themselves in their ivory towers. "That feature doesn't fit my API" is never an acceptable answer to give a budding artist. Instead, we need to find ways to manage the challenges without destroying our engines.

In this article, I'll look at some different ways in which coupling tends to sneak into a project and suggest alternative ways of handling the issues.

### FOUR QUICK IDEAS

#### 1 // BE WARY OF FRAMEWORKS

By "framework" I mean any kind of system that requires all your other code to conform to a specific worldview, such as a scripting system that requires you to add a specific set of macro tags to all your class declarations.

Other common culprits are root classes that every object must inherit from, RTTI/reflection systems, serialization systems, and reference-counting systems.

Such global systems introduce coupling across the entire engine. They rudely enforce certain design choices on all subsystems—design choices that might not be appropriate for them. Sometimes the consequences are serious. A badly thought-out reference system may prevent subsystems from multithreading. A less-than-stellar serialization system can make linear loading impossible.

Often, the motivation for such global systems is that they theoretically increase maintainability. With a global serialization system, we just have to make changes in one place. Thus, it is claimed, refactoring is much easier.

In practice, however, the reverse is often true. After a while, the global system has infested so much of the code base that making any significant change to it is virtually impossible. There are just too many things that would have to be changed, all at the same time.

You would be much better off if each system just defined its own `save()` and `load()` functions. Then, you could incrementally move them over to a new implementation, starting with the most important ones.

Frameworks also make code sharing harder. You can't take someone else's code and plug it into your engine, because it doesn't use your frameworks the right way. And you can't share your code with someone else without also giving them the 200 framework files that your code depends on.

#### 2 // USE HIGH-LEVEL SYSTEMS TO MEDIATE BETWEEN LOW-LEVEL SYSTEMS

Instead of directly coupling low-level systems, use a high-level system to shuffle data between them. For example, handling footstep sounds might involve the animation system, the sound system, and the material system; however none of these systems should know about the others.

So instead of directly coupling them, let the gameplay system handle their interactions. Since the gameplay system knows about all three systems, it can poll the animation system for events defined in the animation data, sample the ground material from the material system, and then ask the sound system to play the appropriate sound.

Make sure that you have a clear separation between this messy gameplay layer (that can poke around in all other systems) and your clean engine code (that is isolated and decoupled). Otherwise, there is always a risk that the mess will propagate downward and infect your clean systems.

In BitSquid Tech, we put the messy stuff in either Lua or Flow (a visual scripting tool, similar to Unreal's Kismet). The language barrier acts as a firewall and prevents the spread of chaos.

Sending information *down* to a low-level system is simple; you can just call the functions of the system. Sending information *back up* is trickier, because the low-level system shouldn't know about the high-level systems. Later in this article, I'll look at some possible approaches to this problem.

#### 3 // DUPLICATING CODE IS SOMETIMES OK!

Avoiding duplicate code is one of the fundamentals of software design. Entities should not be needlessly multiplied. However there are instances when you are better off breaking this rule.

I'm not advocating copy-paste programming or writing complicated algorithms twice. I'm saying that sometimes people can get a little overzealous with their code reuse. Code sharing has a price that is not always recognized: an increase in system coupling. Sometimes, a little judiciously applied code duplication can be a better solution.

A typical example is the `String` class (or `std::string` if you are thusly inclined). You see the `String` class used almost everywhere in some projects. If something is a string, it should use the `String` class, the reasoning seems to be. But many systems that handle strings do not need all the features that you find in your typical `String` class: `locales`, `find_first_of()`, and so forth. They are fine with just a `const char *`, `strcmp()` and maybe one custom-written (potentially duplicated) three-line function. So why not use that? The code will be much simpler and easier to move to SPUs.

Another culprit is `FixedArray<T>`. Sure, if you write `int a[5]` instead, you



will have to duplicate the code for bounds checking if you want that; however your code can be understood and compiled without `fixed_array.h` and template instantiation.

Also, if you have any method that takes a `const Vector &v` as argument you should probably take `const T *begin, const T *end` instead. Now, you don't need the `vector.h` header, and the caller is not forced to use a particular `Vector` class for storage.

Sometimes, you want to be even more flexible and write code that doesn't know anything about the objects it manipulates. Later in this article, I'll look at how that can be achieved.

As a final example, I recently wrote a patching tool that manipulates our bundles (aka pak-files). That tool duplicates the code for parsing the bundle headers, which is already in the engine. Why? Well, the tool is written in C# and the engine in C++, but that is beside the point in this case. The point is that sharing that code would have required a significant effort.

First, it would have to be broken out into a separate library, together with the related parts of the engine. Then, since the tool requires some functionality that the engine doesn't (it needs to parse bundles with foreign endianness), I would have to add a special function for the tool, and probably a `#define TOOL_COMPILE`, since I don't want that function in the regular builds. This means I would also need a special build configuration for the tool, and debug and release versions of that configuration. And the engine code would forever be dirtied with the `TOOL_COMPILE` flag. And I wouldn't be able to rearrange the engine code as I wanted in the future, since that might break the tool compile.

In contrast, rewriting the code for parsing the headers was only 10 minutes of work. It just reads a vector of string hashes. It's not rocket science. Sure, if I ever decide to change the bundle format, I might have to spend another 10 minutes rewriting that code, but I think I can live with that.

Writing code is not the problem. The messy, complicated couplings that prevent you from writing code are the problem.

## 4 // USE IDS TO REFER TO EXTERNAL OBJECTS

At some point, one of your systems will have to refer to objects belonging to another system. For example, the gameplay layer may have to move an effect around or change its parameters. I find that the most decoupled way of doing this is by using an ID. Let's consider the alternatives.

### Effect \*

A direct pointer is no good, because it will become a dangling pointer if the target object is deleted, and the effect system should have full control over when and how its objects are destroyed.

### shared\_ptr<T>

A standard `shared_ptr<T>` won't work for the same reason; it puts the lifetime of `Effect` objects out of the effect system's control, since they will be forced to live as long as the reference exists.

### weak\_ptr<T>

By this, I mean some kind of reference-counted, indirect pointer to the object. This is better, but still too strongly coupled for my taste. The indirect pointer will be accessed by both the external system (for dereferencing and changing the reference count) and by the effect system (for deleting the `Effect` object or moving it in memory). This has the potential to create threading problems.

Also, this construct kind of implies that external systems can dereference and use the `Effect` whenever they want to. Perhaps the effect system allows that only when its `update()` loop is not running and it wants to `assert()` that. Or perhaps the effect system doesn't want to allow direct access to its objects at all, but instead double buffer all changes.

In contrast, using IDs as external references allows the system to freely reorganize its data and processing in any way it likes. The IDs are just integers that uniquely identify a particular object and that can be thrown away when the user is done with them. They don't have to be "released" like a `weak_ptr`, which

removes a point of interaction between the systems. It also means that the IDs are PODs. We can copy and move them freely in memory, juggle them in Lua, and DMA them back and forth to our heart's content. All this would be a lot more complicated if we had to keep reference counts.

Later in this article, I'll demonstrate how you can create a data structure that allows you to quickly look up objects based on their IDs.

## SIGNALING

Signaling is the general problem of how a low-level system can notify a high-level system that something of interest has happened. For example, the animation system may want to notify the gameplay system that a character's foot has touched the ground, so that a footstep sound can be played.

Note that, as mentioned previously, we don't allow direct communication between two low-level systems. The communication is always mediated by a high-level system that sits above the low-level systems.

Also, the low-level systems cannot call high-level systems directly, because they shouldn't know about them. That would introduce strong coupling and break the hierarchical decomposition of the engine into simpler and simpler systems. Instead we want to use some form of indirect notification. There are three common techniques for doing so: polling, callbacks, and events.

### // POLLING

In a polling solution, the high-level system calls some function every frame to check if the event it's interested in has occurred. Has the file been downloaded yet? What about now? Are we there yet?

Polling has a pretty bad rep. It's considered ugly and inefficient. And indeed, in the desktop world, polling is very impolite, since it means busy-waiting and tying up 100% of the CPU with nothing.

However, the situation is completely different in game development. The CPU is already up and spinning. We are already doing a ton of stuff every 33 ms (or half a ton of stuff every 17 ms). As long as we don't poll a huge number of objects, polling won't have any impact on the framerate.

Code that uses polling is often easier to write, and it ends up better designed than code that uses callbacks or events. For example, it's much easier to just check if the A key is pressed inside the character controller, than to write a callback that gets notified if A is pressed and somehow forwards that information to the character controller.

So, in my opinion, polling is actually preferable to other solutions, when you can get away with it from a performance perspective (i.e., when the thread is awake anyway, and you don't have to monitor a huge number of objects). Some areas where polling works well include file downloads, server browsing, game saving, and controller input.

An area less suited for polling is collision notification, since there are  $N^2$  possible collision pairs that you would have to poll for. You could argue that, rather than polling for a collision between two *specific* objects, you could poll for a collision between *any* two objects. In that case, I would say that you are no longer strictly polling but are actually using a rudimentary event system..

### // CALLBACKS

In a callback solution, the low-level system stores a list of high-level functions to call when certain events occur. An important question when it comes to callbacks is whether the callback should be called immediately or whether it should be queued up and scheduled for execution later in the frame.

I much prefer the latter approach. If you do callbacks immediately, you not only thrash your instruction and data caches, but you also prevent multithreading (unless you use locks everywhere to prevent the callbacks from stepping on each other); plus, you open yourself up to nasty bugs when a callback through a chain of events ends up destroying the very objects you are looping over.

It's much better to queue up all callbacks and only execute them when the high-level system asks for it (with an `execute_callbacks()` call). That way you always know when the callbacks occur. Both the high-level system



and the low-level system are in safe, well-known states. Side effects can be minimized, and the code flow is clearer. Also, with this approach, there is no problem generating callbacks on multiple threads and merging the queues.

The only problem with delayed callbacks is that the world state can change from the time when the event happens to the time when the callback is called. In some cases, such changes can invalidate the callback; for example, if one of the objects involved in a collision is destroyed before the callback is made. Luckily, we can handle that by using the ID reference system described in this article to determine whether the objects are still alive.

Note that the callback system outlined here has some similarities to a polling system in that the callbacks only happen when we explicitly poll for them. It also shares similarities with the event system; the callback queue and the event queue are close cousins.

It's not self-evident how to represent a callback in C++. You might be tempted to use a member function pointer. Don't! The casting and typing rules make it nigh impossible to use them for any kind of generic callback mechanism. Also, don't use an "observer pattern," where the callback must be some object that inherits from an `AnimationEventObserver` class and overrides `handle_animation_event()`. That just leads to tons of typing and unnecessary heap allocation.

There is an interesting article about fast and efficient C++ delegates at [www.codeproject.com/KB/cpp/FastDelegate.aspx](http://www.codeproject.com/KB/cpp/FastDelegate.aspx). It looks solid, but personally, I'm not comfortable with making something that requires so many platform-specific tricks one of the core mechanisms of my engine.

Instead, I use regular C function pointers for callbacks. This means that, if I want to call a member function, I have to make a little static function wrapper that forwards the call to the class. That's a bit annoying, but it's better than the alternatives.

To be useful, C callbacks need some context data. The typical approach is to pass a "user data" `void *` to the callback function. I actually prefer a slightly different mechanism, since I sometimes want to pass more data than a single `void *`. I use something like this:

```
struct Callback16
{
    void (*f)(void);
    char data[12];
};
```

There aren't a huge number of callbacks, so using 16 bytes instead of 8 matters little. You could even go to `Callback32` if you needed room for more data. I like to think of this as a "poor man's closure."

When calling the callback, I cast the function pointer to the appropriate type and pass a pointer to its data as the first parameter.

```
typedef void (*AnimationEventCallback)(void *, unsigned);
AnimationEventCallback f = (AnimationEventCallback)callback.f;
f(callback.data, event_id);
```

I'm not worried about casting the function pointer back and forth between a generic type and a specific one, nor about casting the data in and out of a raw buffer. Type safety is nice, but there is an awful lot of power in juggling blocks of raw memory. You don't have to worry that much about someone casting the data to the wrong type, because doing so will cause a huge spectacular crash. Huge spectacular crash bugs are not scary. "I only show myself in release builds with at least 16 networked players"-bugs are scary.

## // EVENTS

As I mentioned, event systems are very similar to callback systems. The only difference is that, instead of storing a direct pointer to a callback function, they store an event enum. The high-level system that polls the events decides what action to take for each enum.

In my opinion, callbacks work better when you want to listen to specific notifications, such as "Tell me when this sound has finished playing." Events work better when you process them in bulk: "Check all collisions to see if the forces involved are strong enough to break the objects." But it is largely a matter of taste.

For storing the event queues (or callback queues), I just use a raw buffer (`array<char>` or `char[FIXED_SIZE]`) where I concatenate all events and their data:

```
[event_1_enum] [event_1_data] [event_2_enum] [event_2_data] ...
```

The high-level system steps through this buffer, processing each event in turn. Note that event queues like this are easy to move, copy, merge, and transfer between cores. (This shows, again, the power of raw data buffers.)

In this design there is only a single high-level system that polls the events of a particular low-level system. It understands what all the events mean, what data they use, and how to act on them. The sole purpose of the event system (it isn't even much of a "system," just a stream of data) is to pass notifications from the low level to the high.

This is, in my opinion, exactly what an event system should be. It should not be a magic global switchboard that dispatches events from all over the code to whoever wants to listen in on them. That kind of spooky "action at a distance" tends to lead to code that is hard to debug, cache inefficient, and strongly coupled in unexpected ways.

## DUCK TYPING

Some systems need to manipulate objects whose exact natures are not known. For example, a particle system has to manipulate particles that sometimes have mass, sometimes have full 3D rotation, sometimes have only 2D rotation, and so on.

(That's what a *good* particle system does, anyway; a bad particle system could use the same struct for all particles in all effects, with "magical" fields called things like `custom_1`, `custom_2` used for different purposes in different effects.)

Another example is a networking system tasked with synchronizing game objects between clients and servers. A very general system might want to treat the objects as open JSON-like structs, with arbitrary fields and values:

```
{
    "score" : 100,
    "name": "Player 1"
}
```

Such systems need to be able to handle these "general" or "open" objects in C++ in a nice way. Since we care about structure, we don't want the system to be strongly coupled to the layout of the objects it manages; and since we are performance junkies, we would like to do it in a way that doesn't completely kill performance. That is, we don't want everything to inherit from a base class `Object` and define our JSON-like objects as below:

```
typedef std::map<std::string, Object *> OpenStruct;
```

Generally speaking, there are three possible levels of flexibility with which we can work with objects and types in a programming language:

### 1 // EXACT TYPING—IF IT IS A DUCK

We require the object to be of a specific type. This is the typing method used in C and for classes without inheritance in C++.

### 2 // INTERFACE TYPING—IF IT SAYS IT'S A DUCK

We require the object to inherit from and implement a specific interface type. This is the typing method used by default in Java, C#, and in C++ when inheritance and virtual methods are used. It's more flexible than the exact



approach, but it still introduces a coupling, because it forces the objects we manage to inherit from a type defined by us.

Side note: My general opinion is that, while inheriting *interfaces* (abstract classes) is a valid and useful design tool, inheriting *implementations* is usually little more than a glorified hack—a way of patching parent classes by inserting custom code here and there. You almost always get a cleaner design when you build your objects with composition instead of with implementation inheritance.

### 3 // DUCK TYPING—IF IT QUACKS LIKE A DUCK

We don't care about the type of the object at all, as long as it has the fields and methods that we need. For example:

```
def integrate_position(o, dt):
    o.position = o.position + o.velocity * dt
```

This method integrates the position of the object *o*. It doesn't care what the type of *o* is, as long as it has a "position" field and a "velocity" field.

Duck typing is the default in many scripting languages, such as Ruby, Python, Lua, and JavaScript. The reflection interface of Java and C# can also be used for duck typing, but the code tends to become far less elegant than in the scripting languages, as below.

```
o.GetType().GetProperty("Position").SetValue(o, o.GetType().
GetProperty("Position").GetValue(o, null) + o.GetType().
GetProperty("Velocity").GetValue(o, null) * dt, null)
```

This is fixed by the *dynamic* type in C# 4.0, which behaves more like the duck types in Ruby and Python. What we want for systems of this type is some way of doing duck typing in C++.

Let's look at inheritance and virtual functions first, since that is the standard way of generalizing code in C++. It's true that you could do general objects using the inheritance mechanism. You would create a class structure looking something like this:

```
class Object {...};
class Int : public Object {...};
class Float : public Object {...};
```



Figure 1

You'd then use either `dynamic_cast` or perhaps your own hand-rolled RTTI system to determine an object's class.

But there are several drawbacks to this approach. It is quite verbose, and the virtual inheritance model requires objects to be treated as pointers, so they (probably) have to be heap allocated. This makes it tricky to get a good memory layout, and that hurts performance. Also, they are not PODs, so we will have to do extra work if we want to move them to a coprocessor or save them to disk.

So I prefer something much simpler. A generic object is just a type enum followed by the data for the object (see Figure 1).

To pass the object you just pass its pointer. To make a copy, you make a copy of the memory block. You can also write it straight to disk and read it back, or send it over network or to an SPU for off-core processing.

To extract the data from the object you would do something like this:

```
unsigned type = *(unsigned *);
o += sizeof(unsigned);
if (type == FLOAT_TYPE)
    float f = *(float *);
```

You don't really need that many different object types: *bool*, *int*, *float*, *vector3*, *quaternion*, *string*, *array*, and *dictionary* are usually enough. You can build more complicated types as aggregates of those, just as you do in JSON.

For a dictionary object, we just store the name/key and type of each object (see Figure 2).

I tend to use a four-byte value for the name/key and not care if it is an integer, float, or a 32-bit string hash. As long as the data is queried with the same key it was stored with, the right value will be returned. I only use this method for small structs, so the probability of a hash collision is close to zero and can be handled by "manual resolution."

If we have many objects with the same "dictionary type" (i.e., the same set of fields, just different values) it makes sense to break out the definition



Figure 2



Figure 3

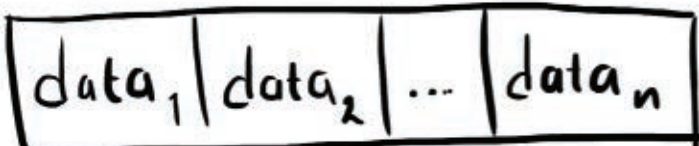




Figure 4

of the type from the data itself to save space (see Figure 3).

Here, the *offset* field stores the offset of each field in the data block. We can now efficiently store an array of like data objects with just one copy of the dictionary type information, as seen in Figure 4.

Note that the storage space (and thereby the cache and memory performance) is exactly the same as if we were using an array of regular C structs, even though we are using a completely open free-form JSON-like struct. And extracting or changing data just requires a little pointer arithmetic and a cast.

This would be a good way of storing particles in a particle system.

Note that this is an array-of-structures approach. You can also use duck typing with a structure-of-arrays approach. I leave that as an exercise to the reader. If you are a graphics programmer, all of this should look pretty familiar. The “dictionary type description” is very much like a “vertex data description” and the “dictionary data” is awfully similar to “vertex data.” This should come as no big surprise. Vertex data is generic, flexible data that needs to be processed fast in parallel on in-order processing units. It is not strange that with the same design criteria we end up with a similar solution.

## THE ID LOOKUP TABLE

I have made several mentions of IDs in this article, and how they are a better way of referring to objects than pointers or references.

By an ID, I simply mean an opaque data structure of *n* bits. It has no particular meaning to us, and we'll just use it to refer to an object. The system provides the mechanism for looking up an object based on it. Since we seldom create more than four billion objects, 32 bits is usually enough for the ID, so we can just use a standard integer. If a system needs a lot of objects, we can go to 64 bits.

To be able to efficiently use IDs, we need a way of looking up system objects based on IDs. There are several requirements that such a data structure needs to fulfill.

- There should be a one-to-one mapping between live objects and IDs.
- If the system is supplied with an ID to an old object, it should be able to detect that the object is no longer alive.
- Lookup from ID to object should be very fast (this is the most common operation).
- Adding and removing objects should be fast.

Let's look at three different ways of implementing this data structure, with increasing degrees of sophistication.

### // THE STL METHOD

The by-the-book object-oriented approach is to allocate objects on the heap and use a `std::map` to map from ID to object.

```
typedef unsigned ID;

struct System
{
    ID _next_id;
    std::map<ID, Object *> _objects;

    System() { _next_id = 0;}
```

```
inline bool has(ID id) {
    return _objects.count(id) > 0;
}

inline Object &lookup(ID id) {
    return *_objects[id];
}

inline ID add() {
    ID id = _next_id++;
    Object *o = new Object();
    o->id = id;
    _objects[id] = o;
    return id;
}

inline void remove(ID id) {
    Object &o = lookup(id);
    _objects.erase(id);
    delete &o;
}
};
```

Note that the `_next_id` counter will wrap around if we create more than four billion objects, and we risk getting two objects with the same ID.

Apart from that, the only problem with this solution is that it is really inefficient. All objects are allocated individually on the heap, which gives bad cache behavior; plus the map lookup results in tree walking, which is also bad for the cache. We can switch the map to a `hash_map` for slightly better performance, but that still leaves a lot of unnecessary pointer chasing.

### // ARRAY WITH HOLES

What we really want to do is to store our objects linearly in memory, because that will give us the best possible cache behavior. We can either use a fixed-size array `Object[MAX_SIZE]` if we know the maximum number of objects that will ever be used, or we can be more flexible and use a `std::vector<Object>`.

If you care about performance and use `std::vector<T>` then you should make a variant of it (call it `array<T>` for example) that doesn't call constructors or initialize memory. Use that for simple types, when you don't care about initialization. A dynamic `vector<T>` buffer that grows and shrinks a lot can spend a huge amount of time doing completely unnecessary constructor calls.

To find an object in the array, we need to know its index. But just using the index as ID is not enough, because the object might have been destroyed and a new object might have been created at the same index. To check for that, we also need an ID value, as before. So, we make the ID type a combination of both:

```
struct ID {
    unsigned index;
    unsigned inner_id;
};
```

Now, we can use the index to quickly look up the object and the `inner_id` to verify its identity.

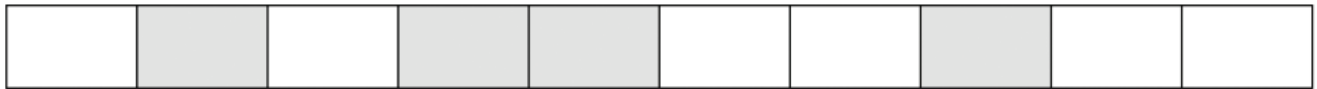


Figure 5

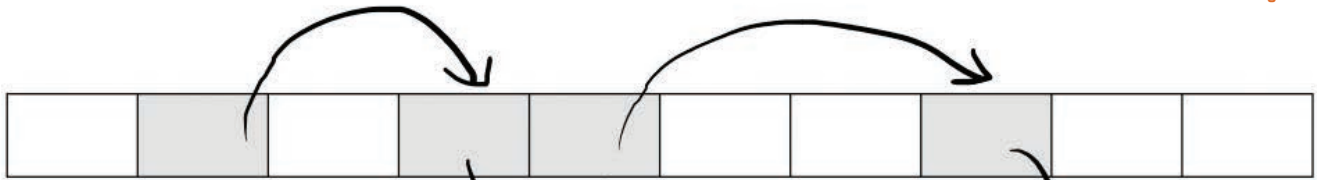


Figure 6

freelist

Since the object index is stored in the ID, which is exposed externally, an object cannot move once it has been created. Objects will, however, leave holes in the array when they are deleted [see Figure 5].

When we create new objects we don't want to just add them to the end of the array; we want to make sure that we fill the holes in the array first.

The standard way of doing that is with a free list. We store a pointer to the first hole in a variable. In each hole we store a pointer to the next hole. These pointers thus form a linked list that enumerates all the holes [see Figure 6].

Interestingly, we usually don't need to allocate any memory for these pointers. Since the pointers are only used for holes (i. e., dead objects), we can reuse part of the objects' own memory for storing them. The objects don't need that memory, since they are dead.

Listing 1 shows an implementation. For clarity, I have used an explicit member *next* for the free list rather than reusing the object's memory:

This is a lot better than the STL solution. Insertion and removal is  $O(1)$ . Lookup is just array indexing, which means it is very fast. In a quick-and-dirty-don't-take-it-too-seriously test this was 40 times faster than the STL solution. In real life, of course, it all depends on the actual usage patterns.

The only part of this solution that is not an improvement over the STL version is that our ID structs have increased from 32 to 64 bits.

There are things that can be done about this. For example, you can get by with 16 bits for the index, if you never have more than 64 K objects live at the same time, which leaves 16 bits for the *inner\_id*. Note that the *inner\_id* doesn't have to be globally unique; it's enough if it's unique for that index slot. So a 16-bit *inner\_id* is fine if we never create more than 64 K objects in the same index slot.

If you want to go down that road you probably want to change the implementation of the free list slightly. The code above uses a standard free-list implementation that acts as a LIFO stack. This means that, if you create and

delete objects in quick succession, they will all be assigned to the same index slot, so you'll quickly run out of *inner\_ids* for that slot. To prevent that, you want to make sure you always have a certain number of elements in the free list (allocate more if you run low) and rewrite it as a FIFO. If you always have *N* free objects and use a FIFO free list, then you are guaranteed that you won't see an *inner\_id* collision until you have created at least *N* \*64K objects.

Of course, you can slice and dice the 32 bits in other ways if you have different limits on the maximum number of objects. You have to crunch the numbers for your particular case to see if you can get by with a 32-bit ID.

// PACKED ARRAY

One drawback with the approach sketched above is that, since the index is exposed externally, the system cannot reorganize its objects in memory for maximum performance.

The holes are especially troubling. At some point, the system probably wants to loop over all its objects and update them. If the object array is nearly full, no problem; however, if the array has 50% objects and 50% holes, then the loop will touch twice as much memory as is necessary. That seems suboptimal.

We can get rid of that by introducing an extra level of indirection, where the IDs point to an array of indices that point to the objects themselves, as seen in Figure 7.

This means we pay the cost of an extra array lookup whenever we resolve the ID. On the other hand, the system objects are packed tight in memory, so they can be updated more efficiently. Note that the system update doesn't have to touch or care about the index array. Whether this is a net win depends on how the system is used, but my guess is that, in most cases, more items are touched internally than are referenced externally.

To remove an object with this solution, we use the standard trick of swapping it with the last item in the array. Then, we update the index so it

Index array with "holes"

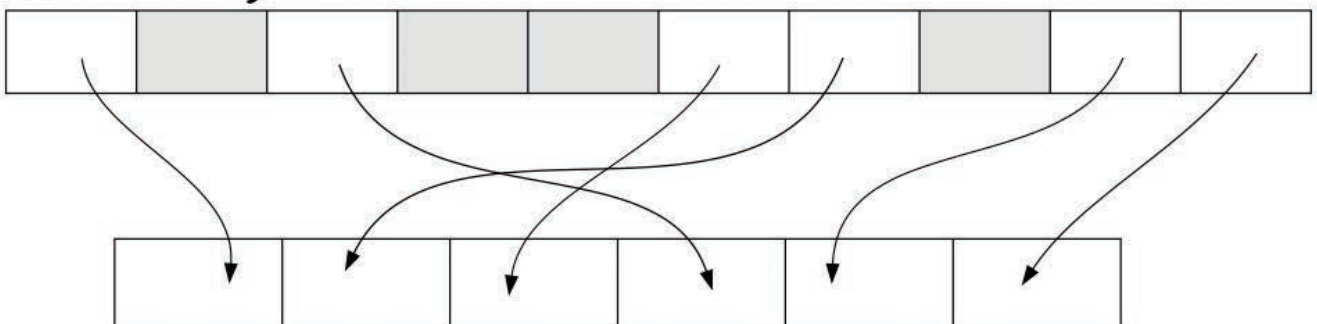


Figure 7


Packed object array

points to the new location of the swapped object.

I've provided an implementation in Listing 2. To keep things interesting, this time there's a fixed array size, a 32-bit ID, and a FIFO free list.

## COUPLES THERAPY

As I've shown in this article, writing modular, reusable, and decoupled code doesn't require heavy use of templates, virtual inheritance, or other advanced C++ constructs. In fact, I think such solutions can often be counterproductive. A "too pure" approach to object-oriented design can lead to tightly coupled class systems where nothing can be moved, changed, or replaced.

Certainly, if I'm going to incorporate some third-party code in my engine, I would like it to be isolated, free of dependencies and inheritances, and preferably just a few files with a simple no-nonsense pure C or barebones C++ interface. Shouldn't I apply the same standards to my own code? 

**NIKLAS FRYKHOLM** is a founder of BitSquid AB and the system architect of BitSquid Tech, a licensed game engine. Prior to starting BitSquid he worked at Grin AB for six years where he served as lead programmer and technical director on multiple titles for PC, Xbox 360, and PS3.

### LISTING 1

```
struct System
{
    unsigned _next_inner_id;
    std::vector<Object> _objects;
    unsigned _freelist;

    System() {
        _next_inner_id = 0;
        _freelist = UINT_MAX;
    }

    inline bool has(ID id) {
        return _objects[id.index].id.inner_id == id.inner_id;
    }

    inline Object &lookup(ID id) {
        return _objects[id.index];
    }

    inline ID add() {
        ID id;
        id.inner_id = _next_inner_id++;
        if (_freelist == UINT_MAX) {
            Object o;
            id.index = _objects.size();
            o.id = id;
            _objects.push_back(o);
        } else {
            id.index = _freelist;
            _freelist = _objects[_freelist].next;
        }
        return id;
    }

    inline void remove(ID id) {
        Object &o = lookup(id);
        o.id.inner_id = UINT_MAX;
        o.next = _freelist;
        _freelist = id.index;
    }
};
```

### LISTING 2

```
typedef unsigned ID;

#define MAX_OBJECTS 64*1024
#define INDEX_MASK 0xffff
#define NEW_OBJECT_ID_ADD 0x10000

struct Index {
    ID id;
    unsigned short index;
    unsigned short next;
};

struct System
{
    unsigned _num_objects;
    Object _objects[MAX_OBJECTS];
    Index _indices[MAX_OBJECTS];
    unsigned short _freelist_enqueue;
    unsigned short _freelist_dequeue;

    System() {
        _num_objects = 0;
        for (unsigned i=0; i<MAX_OBJECTS; ++i) {
            _indices[i].id = i;
            _indices[i].next = i+1;
        }
        _freelist_dequeue = 0;
        _freelist_enqueue = MAX_OBJECTS-1;
    }

    inline bool has(ID id) {
        Index &in = _indices[id & INDEX_MASK];
        return in.id == id && in.index != USHRT_MAX;
    }

    inline Object &lookup(ID id) {
        return _objects[_indices[id & INDEX_MASK].index];
    }

    inline ID add() {
        Index &in = _indices[_freelist_dequeue];
        _freelist_dequeue = in.next;
        in.id += NEW_OBJECT_ID_ADD;
        in.index = _num_objects++;
        Object &o = _objects[in.index];
        o.id = in.id;
        return o.id;
    }

    inline void remove(ID id) {
        Index &in = _indices[id & INDEX_MASK];

        Object &o = _objects[in.index];
        o = _objects[--_num_objects];
        _indices[o.id & INDEX_MASK].index = in.index;

        in.index = USHRT_MAX;
        _indices[_freelist_enqueue].next = id & INDEX_MASK;
        _freelist_enqueue = id & INDEX_MASK;
    }
};
```



# NO “I” IN TEAM

## THE INCREASING IMPORTANCE OF COOPERATIVE MULTIPLAYER

**LEFT 4 DEAD** has many things going for it. It's got tight mechanics, a compelling atmosphere, and great characters, all of which make Valve's zombie-pulping, low-budget masterpiece a must buy for serious gamers. Without a doubt, though, the center pillar of the game is the focus on cooperative play—the idea that all players are working for a common goal.

Cooperative play used to be an afterthought in games, except those made to be played on the same couch. For a while, designers focused more on direct conflict [i.e., player-vs-player combat or deathmatch] as the default way to play with others. In spite of this, cooperative play has survived, and indeed now thrives, frequently as a fulcrum to a multiplayer game's design.

### CAPTURE THE FLAG

» The shooter market seemed to almost abandon the concept of co-op for many years, implementing token modes while spending more and more time making the other multiplayer modes more fluid. But along the way, a funny thing happened; the two gameplay styles merged. As id moved from *DOOM* to *QUAKE* and *QUAKE II*, capture the flag slowly emerged as a gameplay mode far preferable to straight-up deathmatch, and almost every gameplay mode that has emerged since then has focused on various team-vs-team structures.

There are a lot of reasons for this. Capture the flag offers a lot more strategy and depth than straight-up deathmatch, and there are many more ways to vary the game. One could play defender of the home base, or the kamikaze flag runner, or the sniper taking potshots across no man's land. But I think most of all, it is the sense of teamwork and camaraderie that enriches the game experience and keeps players coming back.

Consider the various psychological emotions that happen in a good game of capture the flag. Leaders get a chance to shine. The very skilled get a chance to display mastery over other players, but the lesser skilled can still contribute and feel positive about a group victory. Winning team members congratulate each other. Losing members console each other. The odds that a player will have a positive interaction in a game with a cooperative element are far higher than in one where everyone is trying to crush each other—especially online, where the other driving factor is anonymity.

### COOPERATIVE BOARD GAMES

» While it is by no means the first, the excellent board game *Pandemic* ushered in a wave of cooperative dice-throwers (other games like *Forbidden Island* and *Defenders of the Realm* offer very similar gameplay styles). It's not difficult to see why this became a trend, even ignoring the fact that they offer different game mechanics from the usual fare.

In most of the great board games, gathering six people means that after a couple of hours of play, one player will be the winner and five will be losers; and of course, the math is even worse on a 32-person *QUAKE* deathmatch server. But in *Pandemic*, either everyone wins or everyone loses. As a person who runs a lot of board games, this comes in very handy when, for example, you have more than one person who cares a little too much about winning. Or more crucially, when you have a new person at the table who is unsure of the rules and concerned about making foolish decisions. The tone of the table changes considerably when everyone has a vested interest in the new guy's success.

*Pandemic* is not without its flaws. The nature of the game means that it's possible (and indeed likely) that one domineering player may effectively run the game, controlling everyone's turns. And some designers, such as the team that created the excellent *Battlestar Galactica* board game, have managed to find success creating tension and dynamic social mechanics with the introduction of a traitor concept. Board games have improved dramatically as a whole since some designers have taken cooperative gameplay to heart.

### FORCE MULTIPLIERS

» When many people imagine the possibilities inside an MMO like *ULTIMA ONLINE* or *EVE*, they tend to gravitate toward the “massive” part of the equation. Getting hundreds or thousands of people in the same space is interesting because there are possibilities of doing something much larger than yourself, whether it's attacking an enemy's city with 50 teammates in *SHADOWBANE* or killing the Lich King with 25 close friends in *WORLD OF WARCRAFT*. These spaces are interesting largely because of the uniqueness of the experience, and what adventuring with other players brings to the table.

Even on PvP servers, MMOs are largely all about cooperative play, and the cutting edge of that play is typically dominated by guilds that

have embraced the three great force-multipliers: leadership, teamwork, and communication. Guilds with strong, charismatic leaders can motivate and drive their players through conflict. Players acting in concert can be devastating to their in-game enemies. And the degree of coordination and responsiveness that can be achieved with strong communication tools like voice chat can dramatically increase a team's effectiveness.

One of the great challenges of an MMO designer is to find ways to test those players who have embraced these tenets of cooperative play while also keeping the game playable for those who can't find these guilds. But the interesting thing about these principles of leadership, teamwork, and communication is that they take hard work to achieve. Whether it be hardcore PvP or top-level raiding, excelling requires players to get to know each other, learn how to work well with each other, and depend on each other. And these dependencies work to build strong communities inside your game space.

### ASYNCHRONOUS PLAY

» One of the principal knocks against Facebook games is that they are typically profoundly asocial, despite being called social games. Most Facebook games are remarkably solitary, and the play patterns are very short. A player may spend 15–20 minutes getting a group together in an MMO like *WORLD OF WARCRAFT* or *RIFT*, while few *MAFIA WARS* game sessions last more than 10 minutes for the entire experience.

Some games, like *FRONTIERVILLE*, allow the player to visit another player's lot, but the time cycles are so brief that the odds of actually running into the owner of that lot are fairly low—and considering many people are tending their crops when they're supposed to be at work, they may not be in the mood for a prolonged conversation anyway.

At the same time, Facebook games lean heavily on cooperative play in order to build virality into their products, and they do so with asynchronous game concepts, such as finding



ILLUSTRATION BY JUAN RAMIREZ

ways for players to assist each other even when they're not playing at the same time. They like to do this with both carrots (offering rewarding mechanics for giving gifts) and sticks (putting in roadblocks that can only be overcome by getting help).

Facebook games are still hitting their stride in terms of finding the best ways to do this. Spamming someone's wall provokes a fair amount of backlash from players, which I think not enough Facebook developers worry about. Asking for help is often socially awkward, and introverts in particular may resist. But logging in to find that your high school girlfriend gave you a rusty pump handle while you were offline is a surprisingly powerful emotional event.

### OTHER PLAYERS AS CONTENT

» Playing GUITAR HERO and ROCK BAND alone is one thing. Playing it with a full group of four is quite another. The former is a test of one's own personal skill, and little more. The latter is more social, and more fun. Suddenly, new concerns come up, such as maximizing star power


bonuses or saving a weaker member of the band. Players play songs outside their comfort zones. Virtuosos have an audience to show off to. And like most cooperative gameplay, the sense of shared triumph is even more intoxicating than beating the game alone.

One critical aspect of multiplayer game design is that, when designed correctly, other players are the content. Few things drive this home like handing the microphone around in ROCK BAND and hearing your mother sing Metallica, often while reading the lyrics for the first time. Like all cooperative games, the presence of other people makes old content new again, and the presence of different people brings new strengths and challenges.

### A GROUP EFFORT

» Making great cooperative content isn't easy, but if done right, it can result in powerful gameplay elements that strike strong emotive notes in a larger group of players. Principles like cooperation, teamwork, and leadership become very important. Designers need to work to

account for these, and to encourage players to bond and sympathize with each other to achieve loftier goals inside the game space.

One of the best ways to make cooperative gameplay interesting is to elevate the other players to the status of interesting actors inside the space, who can bring different skills, talents, and personality to a task. Designers who succeed may find themselves rewarded with games that have greater replayability, stronger communities, and memories that resonate in the players' minds long after the game is gathering dust on the shelf. Other people are interesting. Cooperative gameplay should embrace that. 

---

**DAMIAN SCHUBERT** is the lead systems designer of STAR WARS: THE OLD REPUBLIC at BioWare Austin. He has spent nearly a decade working on the design of games, with experience on MERIDIAN59 and SHADOWBANE as well as other virtual worlds. Damian also is responsible for Zen of Design, a blog devoted to game design issues. Email him at [dschubert@gdmag.com](mailto:dschubert@gdmag.com).



# NURBSTALGIA

## REDISCOVER THE LOST ART OF NURBS MODELING

Given the game industry's penchant for turnover and the endless drumbeat of technological change, it's a good bet that most artists don't remember the days when the phrase "NURBS modeler" was a high-status bullet point on a resume. In the current era of ZBrush and Mudbox, when every kid fresh out of art school has a portfolio of zillion-poly characters with lovingly detailed wrinkles and pores, it's hard to find a modeler who remains devoted to the esoteric tools and workflows of NURBS, even though that's the tech that gave us *Jurassic Park*, *Terminator 2*, and *Independence Day*. Nowadays, it's a lot easier to find Max and Maya users who have never even explored the NURBS tools than it is to find a dedicated Birail junkie.

The lost world of NURBS is more than just a historical curiosity, though. There remain a lot of modeling tasks where the old-school techniques can pay off handsomely. If you're modeling vehicles, architecture, or almost any kind of hard surface, NURBS can be a valuable addition to your toolkit. This month, we're going to delve through the modeling archives in search of the forgotten lore of NURBS modeling. We'll have to stick to theory, since implementations in Max, Maya, and XSI (not to mention Rhino, Cinema 4D, or form.Z) differ greatly in their details. The basic toolset is similar across all the packages, though, so we can cover the highlights that make this overlooked set of tools attractive.

/// You probably already know the basics. NURBS models are built using smooth curves, rather than polygons. NURBS surfaces are just extruded, lathed, or swept curves. NURBS curves and surfaces smoothly interpolate between control points, much as a subdivision model interpolates between polygon verts [see [Figure 1](#)]. When you work on a NURBS surface with its control points turned on, you can almost imagine you're working on a subdivision model. However, once you spend some time pushing and pulling verts in both systems, you'll notice that NURBS curves and surfaces are slightly "stiffer" than their subdivision counterparts.

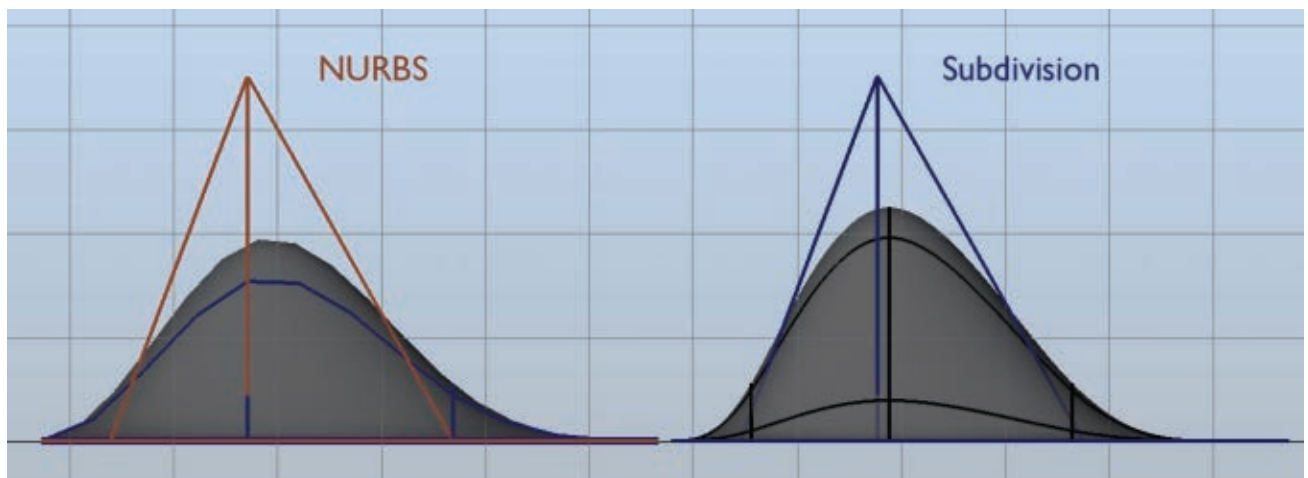
### WHAT'S IN A NAME

» NURBS stands for "non-uniform rational B-splines," a lighthearted mathematician's way of saying "curves." The verbiage also includes some useful hints about what's really going on. In particular, it explains why NURBS and subdivisions respond differently to similar vertex pulls. It also explains where NURBS get their unique strengths for mechanical modeling.

The "non-uniform" part of the acronym means that the control points of a curve can have different degrees of influence on the final curve. This is what allows NURBS to represent everything from flowing freehand curves to carefully controlled Beziers. It also means that

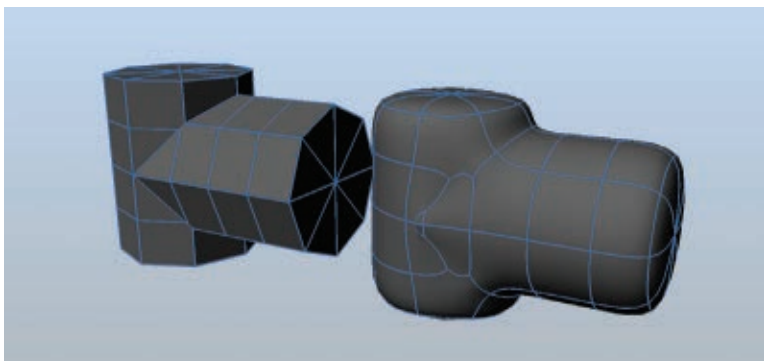
any NURBS surface or curve can be rebuilt with different degrees of flexibility: you can take a curve with 20 control points and rebuild it to have 10 (or 100) without changing the appearance of the curve. This makes it easy to find the balance between point-by-point control and simplicity when modeling.

The really important keyword, though, is "rational," which in this context means that the underlying math of a NURBS curve or surface is precise enough to allow for all sorts of useful geometric calculations. While the subdivision models we're familiar with from, say, Mudbox look pretty and smooth, they are ultimately just approximations of mathematically smooth surfaces.

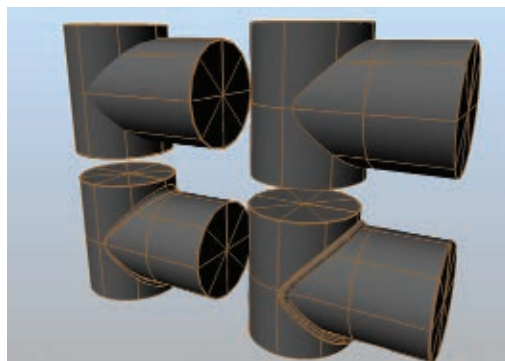


**FIGURE 1:** NURBS and subdivision surfaces work in similar ways. In both cases, you work with a low-res mesh of control points and get a smooth, high-resolution surface. NURBS math, however, is not the same as subdivision math. Here, identical control cages produce noticeably different results from NURBS (left) and subdivision (right). It's the differences in the math that make NURBS such a powerful tool for hard-surface modeling.





**FIGURE 2:** Traditional Boolean techniques produce messy intersections between poly meshes, and they don't make for clean subdivision modeling. Here, intersecting two tubes and subdividing the result doesn't produce a nice clean T-junction. Cleaning this up would require a bunch of hand work to add edge loops and fix the divots.



**FIGURE 3:** NURBS make clean cuts in intersecting surfaces. NURBS also know exactly where the intersection occurs, allowing for neat fillets (lower left) or extruded beads (lower right) along the intersected geometry.

They're good enough for eyeballs, but not for computers. You can't, for example, ask the computer to produce a neat radius fillet between two subdivision meshes, because it doesn't have enough information. With NURBS, on the other hand, a computer can resolve the math exactly. A NURBS circle is an actual circle in the true Platonic sense, where a subdivision circle will always be a bunch of almost-but-not-quite-perfect line segments that approximate a circle.

This cleanliness and precision is the reason manufacturers and industrial designers still use NURBS tools rather than polygons and subdivisions: NURBS are as precise as you need them to be. This fact makes them invaluable to the builders of cars and aircraft, who need to know that their CAD models can be accurately embodied in aluminum or fiberglass. They can analyze their surfaces to be sure that they don't exceed the tolerances of sheet steel or the aerodynamic demands of a wind tunnel.

In our profession, we obviously don't worry much about these things, but even game models can benefit from that mathematical precision. To give a concrete example, think about intersecting shapes. We're all familiar with Boolean operations on solid meshes. We're also all too familiar with the sliver polygons and stray vertices that result from the intersection of oddly angled shapes, and so on. Not only do the limitations of polygonal intersections demand a lot of irritating cleanup work, they also

get in the way of clean subdivision modeling.

### FIT AND TRIM

» Check out the example in [Figure 2](#). Getting a good Boolean to start the mesh requires careful snapping, and even then the resulting mesh isn't subdivision-friendly. With NURBS, on the other hand, the intersection between two surfaces can be calculated perfectly. This makes it easy to create clean intersections, as in [Figure 3](#). This is all due to the inherent "rationality" of NURBS, which allow the computer to know exactly where the surfaces hit without any worries about triangulation or mesh tessellation.

Another really important aspect of the NURBS toolkit is its ability to selectively remove bits of complex surfaces. In the example above, intersecting two shapes is only interesting if you can discard the intersected geometry. Fortunately, NURBS offer the unique ability to lop off arbitrary regions of a surface without affecting the underlying surface curvature. Imagine, for example, the way a wheel well is cut out of the fender of a car: Even though the quarter panel has a complex swooping form, you want to be able to remove a well-defined semicircular part.

If you've ever tried to achieve the same effect in a subdivision or even a poly model, you know how tricky it is to maintain both the large scale curvature of a surface and also the clean outlines of a cutout. When using NURBS, on the other hand, this is easy to do, and more

importantly, it is easy to iterate on. The trimming operation doesn't change the underlying structure of the surface the way a Boolean operation would affect a poly mesh. Instead, it simply turns off unwanted areas of the quarter panel, as if they had been masked out in a Photoshop layer. This makes it easy to move or reshape the trimmed areas without affecting the form, or to re-sculpt the form while keeping the same trimming. (See [Figure 4](#).) This ability to separate the sweep of a surface from its outlines makes trims one of the most powerful and unique features in NURBS modeling. You can approximate the effect with Booleans, Max's ShapeMerge tool, or Maya 2012's new Projected Curve feature, but only NURBS can create cuts that are precise, flexible, and easy to iterate on.

NURBS trims are flexible because of the way the computer sees them. Every NURBS surface is kind of like a perfect subdivision mesh: these surfaces are required to use nicely gridded quad topology. This creates some important limitations, which we'll address shortly, but when it comes to trims, it's a huge plus. Knowing the surface is ultimately a quad, the computer also knows it has inherent U and V directions. When you trim a NURBS mesh, you're really finding an intersection with the surface and then capturing it as a 2D spline in that UV space. If you move the control points and edit the mesh, you'll move the trim curve as well, since it's defined in the UV space of the mesh you're editing. That's why it's so easy to preserve the contours

of your object while cutting out any bits you don't need.

### SIDEBAR: U AND V

» Our familiar "UV mapping" actually gets its name from U and V, the two parametric directions on a NURBS surface. Many early poly modelers had no way of permanently mapping a poly mesh onto 2D textures. Workstation jockeys of the early '90s, running PowerAnimator or the original Softimage, used to sneer at the poor PC drones who had to do everything with planar and spherical projections. For some applications, the inherent UVs of NURBS are still a big advantage. If you want to get a repeating texture to flow precisely along an extruded surface, for example, NURBS are your friends.

Trimming isn't the only way to exploit the inherent UVs of a NURBS surface. In [Figure 3](#), we showed how the same trim curve that cuts the intersection of two surfaces can be used to create a fillet or run a bead around the join. A NURBS surface curve can do anything that a regular curve could do; you can use it for extrusions, lathing, or as part of a loft. This is great for embossed details and surface features, like the flow lines on a car hood or the expressed piping of an engine block. Even better, this ability makes it easy to create complex surface-to-surface relationships that are tedious to model in polys or subdivisions.

For example, imagine you need to model the complex fillet that joins an aircraft wing to a fuselage (see [Figure 5](#)). In a subdivision model, you'd need



**FIGURE 4:** Trimming is a unique and powerful feature of NURBS modeling. This is the ability to mark out areas of a smooth surface to be discarded. It's a very powerful tool for handling tough modeling jobs where an object has continuous curvature but is divided by doors, windows, or other cutouts.

to carefully add edge loops to both the wing and the fuselage, spending a lot of time hiding the inevitable pole points and trying to line up verts for a smooth join. With a NURBS model, on the other hand, the process is hardly more complicated than lofting between two surface curves: one on the wing and another on the fuselage. The NURBS math includes all the information the system needs to ensure that the ends of the loft are smoothly integrated into the surrounding surfaces.

### THE INEVITABLE DISCLAIMER

» With all this good stuff, you might wonder why NURBS tools ever fell out of favor. The problem is the flip side of something we touched on earlier, which is the fact that NURBS surfaces are required to have nicely gridded topology. This has some great properties; it enables trimming and surface curve modeling, for example. Experienced subdivision modelers love quad meshes, because they behave predictably and smoothly.

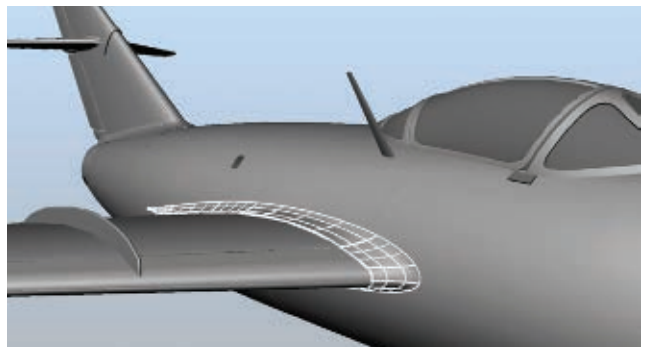
Unfortunately, the world isn't made of neat grids, as any subdivision modeler knows all too well. Non-quad meshes and intersections where more than four edges come together are irritating for subdivision modelers because the computer doesn't really know how to smooth them out properly. ZBrushers and Mudboxers spend much of their lives hunting down the irritating divots and bumps that come from non-quad meshes. The armpits, groins, and inner ears of game characters everywhere are the graveyard of countless pole points, stuffed away by modelers trying to keep the topology elsewhere neat and clean.

NURBS models suffer from a worse version of the same affliction. NURBS tools can ensure that two NURBS surfaces that share an edge appear smooth, but there's nothing in NURBS math that can ensure smoothness across a corner where three or more surfaces come

together. It was this limitation that set the stage for the explosion of subdivision modeling tools and workflows 10 years ago. Even the most dedicated NURBS modelers got heartily sick of trying to divine the correct web of surfaces to capture the folds of a face or the musculature of a body.

Starting with an arbitrary polymesh and simply smoothing it out into a single continuous model was exhilarating after years of worrying about exactly how to keep those pesky corners hidden. Sure, subdivision models had little artifacts where they weren't neatly quadded, but basically, they "just worked," while NURBS modeling of complex organic shapes was too often "just work."

Unfortunately, the passage of time has done little to address this issue, as it's inherent to the math. Maya and XSi include tools that can minimize it by tweaking the continuity of adjacent surfaces, but ultimately, it can't be eliminated from the NURBS workflow. This means that some kinds of modeling, particularly organic subjects



**FIGURES:** NURBS allow you to draw curves directly onto surfaces, and then use those curves as the basis for new surfaces. Here, two surface curves define the complex junction of an aircraft fuselage and wing.

like creatures or characters, are frustrating to do in NURBS. Managing the topology crowds out creativity and experimentation. Only diehards would rather build a portrait head in NURBS than in tactile sculpting tools like Mudbox or ZBrush.

On the other hand, mechanical modeling remains an excellent application for this powerful toolset. Manufactured objects are often designed with NURBS, so it just makes sense to use a similar toolset to recreate them digitally. Trims and surface curves are irreplaceable when addressing the complexities of industrial design. If your job involves a lot of mechanical modeling, invest the time to learn some of the lost art of NURBS. 📌

**STEVE THEODORE** has been pushing pixels for more than a dozen years. His credits include *MECH COMMANDER*, *HALF-LIFE*, *TEAM FORTRESS*, *COUNTER-STRIKE*, and *HALO 3*. He's been a modeler, animator, and technical artist, as well as a frequent speaker at industry conferences. He's currently the technical art director at Seattle's *Undead Labs*.



# Business Casual

## ALLEN MURRAY JOINS POPCAP

As social games continue to make market moves and triple-A teams continue to increase in size, many developers are making the move to smaller teams. And so it is that Allen Murray, formerly a producer at Bungie and director of online strategy at En Masse, has come to make the leap to PopCap.



**BRANDON SHEFFIELD:**  
*How have you found the transition from triple-A games to high-end casual?*

ALLEN MURRAY: It's been wonderful. As a player I found that the games I was spending most of my time obsessing over were more casual in nature, especially on my phone. And I was impressed with the deep gameplay, fidelity and polish that I was experiencing. It was also an area that I hadn't worked in and I wanted to broaden my professional experience.

**BS: How does the difference in scale and scope impact you as a producer?**

AM: Immensely. Working on a game like HALO you are a member of a team of hundreds of people so your direct impact on the game is limited to the specific area you are responsible for. You end up specializing your skill set over time, which as Heinlein says, is for insects. It's a great experience, but ultimately not as fulfilling if you want to have a bigger direct impact on the game. With a smaller game and a smaller team of maybe 4–5 people that potentially grows to dozens, you have a much bigger influence across all disciplines, and you also have the opportunity to

understand and mold the vision of the game. With teams this size the work is much more collaborative and iterative than it is when you are managing the assembly line production processes that are absolutely necessary to deliver a game with a traditional triple-A scope.

**BS: You've worked in jobs that interface with users more directly at places like En Masse, and more abstractly at Bungie—how does working on Facebook games feel in comparison?**


AM: Working at PopCap on games for social and mobile platforms is a perfect merging of everything I learned about making games at Bungie with everything I learned at Amazon.com, where I worked as a programmer for 5+ years during the dot-com boom (and bust). From Bungie: First and foremost we need to make a fun and compelling game that is polished and makes people want to play and play—it needs to kick ass! And we need to ship it. From Amazon: It needs to run on an online infrastructure that can scale well, with all the commerce systems, and full integration with Facebook's communication infrastructure. It also needs

to have the pipeline in place for updating content at the speed of the web. To me, making a game on Facebook is basically making a casual MMO.

**BS: What made you choose PopCap?**

AM: Because it's awesome! Seriously, it's rad here. I really wanted to be at PopCap because I was a fan of their games, saw the growth in mobile and social gaming, and that PopCap was one of the few developers in my opinion with a consistent track record of fun game experiences. I knew that I could learn a lot from this studio full of veteran developers, and hopefully contribute something as well. Another compelling reason is that they are located near one of my favorite bakeries in downtown Seattle.

**BS: Have you seen any changes after the EA acquisition, or is it business as usual?**

AM: Business as usual. EA has been really supportive by allowing us to continue to operate the way we choose, while giving us the opportunity to learn from and share with our colleagues across their entire organization. 

## who went where

Zynga vice president of studios Louis Castle has left the company less than eight months after his arrival, citing work/life balance issues as the driving factor for his departure.

Lloyd Melnick, formerly general manager at Disney's social games branch Playdom, has taken up a CEO role at social start-up 519 Games, and is looking to release the company's first games in early 2012.

Social app and game monetization company Tapjoy has hired Jim Jones, formerly vice president at online corporation Yahoo! as its new vice president and general manager of sales.

Activision Blizzard chairman and former Activision publishing president and CEO Michael J. Griffith has been named to the board of directors for arcade operator Dave & Buster's.

## new studios



Industry veterans from Insomniac, THQ, and Liquid Entertainment have formed LA-based Hidden Variable Studios. The studio's first title, BAG IT!, was released in November on the App Store and Android market.

West Pier Studio, which will focus on online and mobile games, has been formed in Brighton, UK by five former staffers from recently closed SPLIT/SECOND studio Black Rock.

Mob Rules, a new studio formed by veterans from CHAMPIONS ONLINE developer Cryptic Studios, is letting contributors to its Kickstarter fund have a say in the company's direction, including the subject of its first game project.

Australian developer and publisher Halfbrick, maker of popular iOS game FRUIT NINJA, is opening a second studio in Sydney.



# GDC Online 2011's Most Memorable Quotes

\\ This year's GDC Online, organized by the UBM TechWeb Game Network (which also owns this magazine) featured more than 145 lectures, panels, keynotes, and roundtable discussions presented by more than 225 speakers. There was also a bustling expo floor with more than 100 exhibitors and sponsors, and the conference boasted record-breaking attendance numbers of 3,350.

Game Developer sister site Gamasutra covered the event in full, and while we couldn't possibly give proper attention to all of the lectures, we've rounded up a few of our favorite quotes from the show.

*"A few Kotaku articles and IGN front pages do not make a hit game."*

BioWare San Francisco's **Ethan Levy**, from an insightful and open talk about how the studio's social game DRAGON AGE LEGENDS attracted a lot of temporary Facebook likes, yet wasn't a big hit.

*"That's bullshit. Are we going to start hiring 10-year-old kids to make games for 10 year old kids?"*

Veteran MMO developer and former FREE REALMS creative lead **Laralyn McWilliams** (who recently joined iWin) discusses the flaw in thinking the only path to attracting more female gamers is to hire more female developers. Instead, she says, stop making games for yourself and learn to give your audience what it needs.

*"What they're doing looks a lot more like e-commerce than game design."*

EA Playfish's **Tom Mapham** on how analysts and product managers are running usability tests and market research on over a terabyte of daily data generated by players of THE SIMS SOCIAL.

*"We also ignored MySpace."*

PopCap cofounder **John Vechev** lists several of the reasons PopCap has remained a success, and how he sees the company going for at least another 30 years.

*"I'm inherently super-duper lazy, so if I think of something, it's going in."*

Valve writer **Eric Wolpaw** responds when asked if he has a larger vision of his games' worlds than what players experience on-screen. Teammate **Marc Laidlaw** agreed, saying that creating things that don't make it into the game is "kind of counterproductive."

*"Rarely waste an opportunity to fire a brilliant jackass."*

**Jeremy Gaffner**, executive producer at WILDSTAR MMO developer Carbine Studios, on how an "irreplaceable" employee who does solid work but is abrasive to everyone in the studio might not be so irreplaceable after all.

*"Designers are really worried that their great idea is going to be misunderstood or unfairly judged if it's seen too early."*

LEAGUE OF LEGENDS design director **Tom Cadwell**, stressing that providing feedback early and often, even if something isn't "ready," is a great way to avoid design pitfalls.

*"My job is to get everyone on the internet to want to have a beer with me."*

Sega community manager **Kellie Parker** who, along with Microsoft's **Kathleen Sanders**, provided an insightful overview on the role of community manager, the differentiation between community and PR, and how to best deliver bad news.

*"They would be insane not to."*

Gaikai's **David Perry** responds when asked if the next generation of consoles would implement cloud gaming capabilities.

*"If you're engrossed in the minutiae of running your team, who's actually saying, 'Well what are we going to do next? What are the threats, and how are we going to mitigate them?'"*

LEAGUE OF LEGENDS lead producer **Travis S. George** warns that one of the worst things a game producer can do when a game is in trouble is dive in and take control instead of delegating.

*"If you don't make it fun in the first three minutes, you've failed."*

Blizzard SVP of creative development **Chris Metzen** on engaging your players through proper storytelling.

*"I tell you, it's not easy."*

Eidos Montreal's **Mary DeMarle** discusses the incredible Excel spreadsheet that contained the entirety of the studio's multifaceted DEUS EX: HUMAN REVOLUTION in one document.

*"Players are genius at missing stuff."*

Playdom's **Gordon Walton** discussing with community experts how there's "always an opportunity to help" new players.

*"Theme is the oxygen of narrative. You don't need to see it to notice when it's missing."*

BioWare Austin writer **Hall Hood** (STAR WARS: THE OLD REPUBLIC) on how a story's theme can be even more important than characters, plot, and dialogue.

*"Last year, we came here and told you that everything had to be a farm."*

Playdom's **Dave Rohrl** who, along with coworker **Steve Meretzky**, provided their annual humorous talk on social gaming trends. Unfortunately, it seems like too many people listened to them, as they said that the mechanic has been overdone now.

*"You click through everything until it explodes with blood and treasure."*

Blizzard's **Kevin Martens**'s mantra for the upcoming DIABLO III. He, along with several other writers and designers, provided a fast and off-the-cuff talk about their inspirations and what makes a great gaming moment.

*"If you dock a ship and you're wearing a monocle, people come and fight you."*

CCP Games' **Ben Cockerill** outlines the strong pushback from EVE ONLINE players after the company introduced controversial in-game items.

*"On iOS you have 10 phones you have to care about. On Android it's over 100."*

PopCap's **Giordano Bruno Contestabile** on mobile development essentials. One hundred may sound like a big number to someone new to mobile development, but it's nothing like the bad old J2ME days!

*"Writers don't often get to sit at the adults table."*

Game writer and Extra Lives author **Tom Bissell** calls for writers to be ingrained deeper in the development process.

*"It's pretty easy for a sandbox to turn into a desert."*

Telltale Games's design director **Dave Grossman** finds the balance between giving a player total freedom and having their fates dictated to them.

*"Everyone who's had a shower has had a good idea."*

Atari cofounder **Nolan Bushnell** reinforces his belief that creativity is not about having great ideas, but owning them. ☞



# EVOLVING TRIPLE TOWN

## TALES OF A SINGLE-PLAYER SOCIAL NETWORK GAME

My company, Spry Fox, just recently released a game called TRIPLE TOWN on Facebook and Google+. It is, in its current incarnation, primarily a single-player game that is made social by the prominent inclusion of an in-game leaderboard a la BEJEWELLED BLITZ. TRIPLE TOWN had the distinction of being featured by Facebook within days of its launch and of being the 20th game (and first indie-developed title) on Google+. This month, I'll talk about why we chose to make TRIPLE TOWN our first game for social networks, and how we have been surprised by the game thus far.

### WHY TRIPLE TOWN ON SOCIAL NETWORKS?

» There are many other games we could have chosen as our first social network title, all of which are probably more "social" than the current iteration of TRIPLE TOWN. However, TRIPLE TOWN was a proven game—it first launched on the Kindle, and has consistently held one of the highest user ratings on that platform. We were nervous about the high failure rate of independent game developers on Facebook, and wanted to do whatever possible to reduce our risk; having a proven game design was helpful in that regard.

TRIPLE TOWN also has a relatively small scope. We knew it would not be too expensive to bring to social networks, which was another way we could control our risk. Additionally, we were inspired by the success of BEJEWELLED BLITZ and surprised by the relative scarcity of games like it on Facebook; we felt that Triple Town represented an opportunity to compete in a relatively underserved category of games with a unique twist of our own invention.

An original, fun, proven gameplay mechanic, a low up-front investment cost, and an undersaturated genre do not guarantee success; however they do tilt the odds in your favor and increase the likelihood that platform managers will notice and choose to feature you.

### OUR FEARS AND EXPECTATIONS

» TRIPLE TOWN originally launched on Kindle as a paid game (\$3.99) with no in-app purchases, and it was unclear to us how best to transform the game into a free-to-play experience suitable for social networks. The example set by BEJEWELLED BLITZ was to charge for power-ups, so we adopted a somewhat similar strategy with the TRIPLE TOWN item store, but it simply didn't feel like enough. TRIPLE TOWN in its current incarnation is basically a single-player experience with a leaderboard, and it is difficult to imagine many players spending cash just to best their friends' scores. So, we did something scary and added turn limits to the game.

I call turn limits "scary" because players of puzzle games are not used to them, even though many other social games incorporate limits in the form of "energy" and similar concepts. We were nervous that turn limits would put people off, but we were even more nervous about the significant likelihood that the game simply wouldn't generate enough revenue to sustain itself. So we decided to experiment with turn limits, understanding that if it seemed like they were really upsetting our players, we would ultimately remove the limits and try something else.

### WHY TESTING MATTERS

» It is worth noting that almost every game designer I've spoken to has questioned our use of turn limits, and they all assumed that the limits would destroy the game. As it turns out, turn limits are the least of our problems. While some players have made negative comments about them, very few players actually quit TRIPLE TOWN after hitting the limit for the first time (or second or third for that matter). Our retention numbers are not great, but it has nothing to

do with limits. The vast majority of players who churn out of TRIPLE TOWN do so within the first few hundred turns of the game—well before they even realize there are turn limits.

If you would have asked me three months ago what our top three problems in TRIPLE TOWN would be, I would not have included retention during the first 30 minutes of gameplay on the

list. After all, Kindle users loved TRIPLE TOWN, and everyone who beta-tested the game complimented us on our supposedly excellent tutorial. I was much more concerned about the fact that we were launching without a metagame and without any significant viral functionality. And of course, we launched with those scary turn limits! All these things remain concerns, but they are secondary to a short-term retention issue that we never even predicted. It wasn't predicted by all those experienced game designers who rolled their eyes at our use of turn limits and predicted it would be the game's downfall, either.

Moral of the story? Don't make assumptions. Have the courage to experiment, but be diligent enough to monitor the results of your experiments carefully.

### WHERE TO GO FROM HERE?

» We have pretty big plans for TRIPLE TOWN. We're revamping the tutorial in order to address some of the short-term retention issues I mentioned earlier. We're going to enable players to see each other's cities, and we're going to enable players to view previously built cities. We've got a metagame planned that should add a real sense of progression to the game, and it will afford us opportunities to make the game more truly social, in addition to helping with retention. We're also going to keep experimenting with monetization; maybe we'll find something better to sell than turn limits, or maybe not. One way or another, TRIPLE TOWN will be a very different game in three months, and that's a wonderful thing! Rest in peace, "fire and forget" model of game development...we don't miss you one bit. 🙏

“ It is worth noting that almost every game designer I've spoken to has questioned our use of turn limits, and assumed that the limits would destroy the game.”

DAVID EDERY is the CEO of Spry Fox and has worked on games such as REALM OF THE MAD GOD, STEAMBIRDS, and TRIPLE TOWN. Prior to founding Spry Fox, David was the worldwide games portfolio manager for Xbox LIVE Arcade.



# THE NEW KID IN SCHOOL

## PROFESSIONAL GAME AUDIO EDUCATION

**All art forms wrestle with legitimacy.** New forms of art first struggle to be seen as legitimate among peer artists, then by their audience, and lastly by both the art world and society at large. Compared to ballet, portraiture, or even our closest cousin, post-production for film and television, game audio is a nascent upstart in the art world, that snuck on stage just before the curtain came down on the 20th century.

Nothing legitimizes like the ivy-clad halls of academia, though. Across the globe, programs that focus on interactive audio as an official field of study, career path, and artistic endeavor are springing up in trade schools and universities. This means an ever-growing crop of young sound designers and composers are being trained and readied for assistant and entry-level jobs, and an ever-growing list of potential teaching gigs are becoming available to more experienced audio professionals.

### THE FIELD

» Game audio programs can be found in cities across the country. They come with a variety of available degrees, a mixture of topics, and a wide range of experience levels from the instructors. With one primary exception, game audio courses are part of larger degree programs and come in two distinct flavors: those focusing on sound design, and those focusing on interactive music. Few programs seem to blend the two into one all-inclusive program.

Those programs focusing on sound design and implementation for games are usually part of recording arts degrees. Schools like California's Ex'pression College for Digital Arts, and Florida's Full Sail University all offer Bachelor's of applied science degrees. Meanwhile, Chicago's Tribeca Flashpoint Media Arts Academy offers an Associate's degree in applied science for recording arts students. Each of these programs is fundamentally about recording engineering. Students focus on everything from mastering audio to the use of large consoles and microphone placement.

Game audio in these programs tends to be represented by a single unit on interactive audio. As such, students learn the basics, such as file name management, introductions to the concepts of interactive implementation, and the beginnings of working with audio middleware engines. Students are often exposed to either the Unreal or Unity engines as well as Max/MSP.

Just like game sound design programs, programs that teach game music composition have a variety of degree levels available. Boston's notable Berklee College of Music offers a number of classes on interactive scoring as part of its Bachelor's of music degree. Some of the game music programs available are at a graduate level, since apart from smaller trade schools and conservatories, even mainstay institutions are getting into the act. The University of Southern California's graduate scoring for motion pictures and television program, for instance, has a game scoring component that was taught for years by Lennie Moore and is now helmed by BIOSHOCK composer Garry Schyman. Moore himself now teaches game music composition at UCLA's Extension campus as part of its graduate film scoring program. BROTHERS IN ARMS composer Stephen Harwood Jr. currently teaches "Composing for Video Games" as part of New York University's masters of music in film scoring program.

Like the recording arts degrees, interactive composition students learn about the specific challenges related to the field, including nonlinear



composition and an introduction to middleware such as XACT, FMOD, and Wwise. While the larger focus for each of these programs remains film scoring, each gives the distinct separation of game composition its due.

### THE PINNACLE

» As mentioned above, there is currently one exception to the general model wherein game audio is a component of a larger degree program. Los Angeles' Pinnacle College offers the world's first dedicated video game sound design program. In a year of concentrated study, students emerge from Pinnacle College with an occupational Associate's degree in game sound.

Pinnacle's program is in many ways the reverse of most others, in that recording engineering is a component of its game audio curriculum. In addition to basics in recording console operation and introductions to Cubase

and Pro Tools, students in Pinnacle's program have game-specific classes on the history of interactive audio, recording and producing game dialogue, and hands-on experience with both the Unity and Unreal engines. In addition to voice work, Pinnacle is also one of the rare programs that teaches both interactive music composition and field recording, allowing students to gain experience in all three of the main game audio disciplines.

The course is run by Eitan Teomi, and it is quite intensive. It's focused on current game technology like FMOD and Wwise, and its curriculum has been guided by a board of advisors from within the game audio world that includes folks like Activision's Don Veca and THQ's Victor Rodriguez.

Knowledge that once passed from sound designer to assistant like a medieval apprenticeship is now being standardized and refined, focused and institutionalized. Students are finding a well-marked path into what has traditionally been a difficult industry to enter. Sound designers and composers are also finding that the option exists to move from the role of audio professional to audio professor. While Pinnacle is the first dedicated game audio program, it will not be the last. As time moves on, interactive audio will only continue to find itself increasingly legitimized. <sup>10</sup>

**JESSE HARLIN** has been composing music for games since 1999. He is currently the staff composer for LucasArts. You can email him at [jharlin@gdmag.com](mailto:jharlin@gdmag.com).



# DIG-N-RIG

<https://www.digipen.edu/?id=1170&proj=24629>

SYNTACTIC SUGAR'S DIG-N-RIG IS A BLEND OF RESOURCE MANAGEMENT AND CREATIVE STRATEGY, IN WHICH PLAYERS TAKE CONTROL OF A FUTURISTIC MINING ROBOT WHOSE ONLY TASK IS TO DIG INTO THE EARTH TO COLLECT VALUABLE ORE. AS PLAYERS DIG FURTHER INTO THE GROUND, THEY MUST CONSTRUCT INCREASINGLY ELABORATE SYSTEMS OF CONVEYER BELTS TO BRING THEIR SPOILS TO THE SURFACE. WITH THIS ORE, PLAYERS CAN BUY UPGRADES TO TUNNEL EVEN FURTHER INTO THE DEPTHS OF THE EARTH. HERE, WE TALK TO THE TEAM OF DIGIPEN STUDENTS AT SYNTACTIC SUGAR TO DELVE INTO HOW DIG-N-RIG CAME TO BE.

**Tom Curtis:** *Let's start by talking a bit about the origin of the project. How did you all come up with the premise and basic design for the game?*

**DIG-N-RIG Team:** We decided that we wanted a game in which players use creative building mechanics to traverse their environment. We found that most games that involved building typically had the player building upward. Andrew Colean (lead producer) joked about the idea of building downward—and it sounded silly at first—but after Kirk Barnett (project manager) drew a quick concept on a whiteboard, we actually found the idea quite compelling. From there, we just went with it.

**TC:** *How much did the game change between the initial concept and the final version?*

The core mechanics of DIG-N-RIG were always about collecting resources and building creatively. The gameplay was designed to be heavily item-based. We did this in order to have the ability to cut content if necessary. This was a very important decision for us because, as a student game team with limited development time, we needed to keep scope under control. Aside from cutting some features, the concept of DIG-N-RIG changed very little from initial design to release.

**TC:** *What would you say was the biggest challenge you faced during development?*

As a DigiPen student game team, we had to balance making a game along with the homework for the rest of our classes. Most of the time the biggest challenge was just that: finding time. DIG-N-RIG,

being our first game project, was the largest coding task that any of us had ever encountered. In fact, most of us were just learning how to program for the first time. This resulted in many hours of bug fixing that we wish we could have spent developing additional content. In the end it was actually quite surprising to us that DIG-N-RIG came together as well as it did.

**TC:** *Looking back on the project, is there anything you wish you had done differently?*

Absolutely. One feature we all wanted was an interactive tutorial—a step-by-step guide to assist the player with their journey. At the start of the game an overwhelming amount of information is presented to the player. Introducing our gameplay in smaller chunks would have worked much better. DIG-N-RIG could also have benefited from additional play testing. It was only through player feedback that we discovered our game's mineral economy and layer progression were both completely unbalanced. Unfortunately we began play testing late in the development process, leaving us with very little time to fix those issues. The most important thing to us is the players' experience, and going forward we plan on learning from our mistakes and using the knowledge we've gained to build better games.

**TC:** *DIG-N-RIG is a very UI-heavy game with a number of menus and resources to keep track of. Can you describe your process for UI design, and did you encounter any challenges communicating information to the player?*

We embraced the idea of a UI-intensive game very early on. We rearranged its composition many



times, adding menus as necessary and reducing screen clutter where possible. This made our game more user-friendly by transitioning an otherwise overwhelming number of key presses to simple point-and-clicks. This also, however, created a lot of menu-based information that needed to be displayed compactly

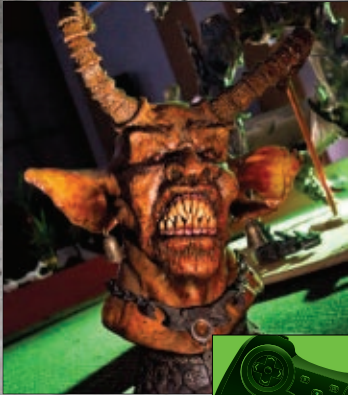
and meaningfully. The vertical UI flow was chosen as a complement to the vertical gameplay flow. This was done intentionally to keep the player on-screen as much as possible. Some UI problems still exist in DIG-N-RIG, and to this day plenty of players still aren't sure how to use the Vac Pak. 🙄

## the DIG-N-RIG development team

Kirk Barnett project manager  
Andrew Colean lead producer

Brandon Stenen tech director  
Derek Opitz lead designer

# Start Living The Dream!



## A.S. Degree in Game Production

Learn to create the future of games with an Associate's Degree in Game Production from The Los Angeles Film School. Your education will give you the knowledge to view every piece of a game artistically, analyze its programming and learn the tools & techniques to create the worlds we play every day.

Learn the Science of Game Production and have the following skill set:

- Create Game Art
- Design Characters, Objects & Environments
- Develop Game Programming Skills
- Discover the Art of Storytelling

Learn from The Los Angeles Film School's experienced industry professionals.

- On-site housing coordinator
- Accredited College, ACCSC, VA-Approved
- Financial Aid & Military Education Benefits (including BAH) available to those who qualify

Scan for more Information



THE  
**LOS ANGELES**<sup>®</sup>  
FILM SCHOOL

ANIMATION + AUDIO + FILM + GAMES

Create Your Future Today. Call:

**800.406.7485**

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



\*Length of program and start dates are dependent on course of study and degree option. For more information on our programs and their outcomes visit [www.lafilm.edu/disclosures](http://www.lafilm.edu/disclosures).

© 2011 The Los Angeles Film School. All rights reserved. The term "The Los Angeles Film School" and The Los Angeles Film School logo are either service marks or registered service marks of The Los Angeles Film School. Accredited by ACCSC.

## TURN YOUR PASSION FOR GAMING INTO A CAREER

### Game Art

Bachelor's Degree Program  
Campus & Online

### Game Design

Master's Degree Program  
Campus

### Game Development

Bachelor's Degree Program  
Campus

### Game Design

Bachelor's Degree Program  
Online



### Campus Degrees

#### Master's

- Entertainment Business
- ▶ Game Design

#### Bachelor's

- Computer Animation
- Creative Writing for Entertainment
- Digital Arts & Design
- Entertainment Business
- Film
- ▶ Game Art
- ▶ Game Development
- Music Business
- Recording Arts
- Show Production
- Sports Marketing & Media
- Web Design & Development

#### Associate's

- Graphic Design
- Recording Engineering

### Online Degrees

#### Master's

- Creative Writing
- Education Media Design & Technology
- Entertainment Business
- Internet Marketing
- Media Design
- New Media Journalism

#### Bachelor's

- Computer Animation
- Creative Writing for Entertainment
- Digital Cinematography
- Entertainment Business
- ▶ Game Art
- ▶ Game Design
- Graphic Design
- Internet Marketing
- Mobile Development
- Music Business
- Music Production
- Sports Marketing & Media
- Web Design & Development



**FULL SAIL**  
UNIVERSITY

[fullsail.edu](http://fullsail.edu)

Winter Park, FL

800.226.7625 • 3300 University Boulevard

Financial aid available to those who qualify • Career development assistance • Accredited University, ACCSC

To view detailed information regarding tuition, student outcomes, and related statistics, please visit [fullsail.edu/outcomes-and-statistics](http://fullsail.edu/outcomes-and-statistics).



VFS student work by Aldo Martinez Galzadilla

# MAKE MORE ENEMIES

**Game Design at VFS** lets you make more enemies, better levels, and tighter industry connections.

In one intense year, you design and develop great games, present them to industry pros, and do it all in Vancouver, Canada, a world hub of game development.

The LA Times named VFS a top school most favored by game industry recruiters.



**VFS** Find out more.  
[vfs.com/enemies](http://vfs.com/enemies)

“VFS prepared me very well for the volume and type of work that I do, and to produce the kind of gameplay that I can be proud of.”

DAVID BOWRING, GAME DESIGN GRADUATE  
 GAMEPLAY DESIGNER, SAINTS ROW 2

## ADVERTISER INDEX

| COMPANY NAME                        | PAGE | COMPANY NAME                | PAGE |
|-------------------------------------|------|-----------------------------|------|
| EPIC GAMES .....                    | C3   | RAD GAME TOOLS .....        | C4   |
| FULL SAIL REAL WORLD EDUCATION..... | 46   | TWOFOUR54.....              | 3    |
| HAVOK .....                         | C2   | VANCOUVER FILM SCHOOL ..... | 47   |
| LOS ANGELES FILM SCHOOL .....       | 46   |                             |      |

*gd Game Developer* (ISSN 1073-922X) is published monthly by UBM LLC, 303 Second Street, Suite 900 South, South Tower, San Francisco, CA 94107, (415) 947-6000. Please direct advertising and editorial inquiries to this address. Canadian Registered for GST as UBM LLC, GST No. R13288078, Customer No. 2116057, Agreement No. 40011901. **SUBSCRIPTION RATES:** Subscription rate for the U.S. is \$49.95 for twelve issues. Countries outside the U.S. must be prepaid in U.S. funds drawn on a U.S. bank or via credit card. Canada/Mexico: \$69.95; all other countries: \$99.95 (issues shipped via air delivery). Periodical postage paid at San Francisco, CA and additional mailing offices. **POSTMASTER:** Send address changes to Game Developer, P.O. Box 1274, Skokie, IL 60076-8274. **CUSTOMER SERVICE:** For subscription orders and changes of address, call toll-free in the U.S. (800) 250-2429 or fax (847) 647-5972. All other countries call (1) (847) 647-5928 or fax (1) (847) 647-5972. Send payments to *gd Game Developer*, P.O. Box 1274, Skokie, IL 60076-8274. Call toll-free in the U.S./Canada (800) 444-4881 or fax (785) 838-7566. All other countries call (1) (785) 841-1631 or fax (1) (785) 841-2624. Please remember to indicate *gd Game Developer* on any correspondence. All content, copyright *gd Game Developer* magazine/UBM LLC, unless otherwise indicated. Don't steal any of it.



# I'M ALMOST DONE!

## WHY THAT TASK TOOK LONGER THAN I THOUGHT

Hey, thanks for checking in with me on that feature request! I know I told you that adding an “effectiveness” stat to all the weapons in the game was going to be a snap, so I understand that it might seem like I’ve been spending an unreasonably long time on it.

Thing is, when you emailed me to ask for that, I didn’t know if you meant, like, “effectiveness” as a function of damage, or something like apparent effectiveness to different kinds of enemies, or maybe some kind of recommendation system that would tell players what weapons would be most effective for them.

I called your name a couple times but you must not have heard me over the cubicle wall, so I decided I would try to code up something that would be flexible enough for you to be able to do any one of those things. I thought, well, the damage stat isn’t the only thing that determines effectiveness: there’s how fast the weapon is, if you can dual-wield them, your own character’s level, skill trees, and all that crazy stuff.

It seemed pretty clear to me that you’d at least want access to all of those parameters so that you could iterate on the design of the “effectiveness” stat, right? So, I started to implement a generalized suite of classes and other tools that would anticipate all of the ways you and the other designers might want to use them. It’s all contained in this new top-level category on the debug menu. I call it the EFFECTIVATOR.

Basically speaking, the EFFECTIVATOR is an abstract, component-based analyzer that takes well-formed inputs and measures them against a configurable set of criteria that are then saved as Effectiveness Evaluation Parameters [I’ve created a file format, too—.eep files]. You can see that I now have an EEP set up to evaluate the weapon’s color and its number of rune slots as a function of how long the player has had at least three friends online! Pretty cool, huh?

Then I thought, well, you probably need some kind of interface to put those criteria together, since I didn’t want the designers to have to learn another new scripting language (you remember what happened the last time we introduced those six new languages for you guys to use). Plus, the hallmark of a good game dev tool is that it doesn’t just expose values for people to tweak: it makes authoring content just as fun as playing it!

With that in mind, I spent a few days gathering information on various GUI libraries that we could leverage. Hey, I’m no “not invented here” guy. I’m all about saving as much time and being as efficient as possible with all our tool efforts! It’s results that matter, not how we got there! Anyway, I eventually settled on creating the graphical interface to my new data format using Swing.

Surely you remember the Java Swing library? That thing really came in handy, because I know the music guy who sits in the corner uses a Mac, and



I wasn’t sure if he might need to use the graphical editor too. So I thought I’d make it cross-platform, you know, just in case. As soon as we get that good ol’ Java Runtime Environment installed on every computer in the studio, we’ll all be able to create our own EEPs in an easy-to-use, drag-and-drop environment! I’m so stoked!

Oh yeah, there’s one more thing I thought I should tell you about. As I was wrapping up the interface work, I noticed that there were a couple of bugs that cropped up whenever I tried to actually pull down those damage values from each weapon object. It turns out that you don’t really know what damage a weapon will do until you try it in-game because of the way different materials dampen the hit.

To solve for that, I took the gameplay engine and wrapped that up inside the front end, so that the front end can run a simulation of the damage code anytime it wants to pull the damage value! I went and built a tiny level environment off-screen

that actually tested the weapon against a variety of targets for over 65,536 iterations to calculate extremely accurate average damage values.

Actually, I guess I shouldn’t call that a “simulation,” since it really just runs the game in the background! Ha ha! So those values it generates are 100% true to the game. Damn, that’s even more awesome than I thought.

I see the look on your face now—I hope you aren’t worried about speed. It’s all very functional. It only takes a couple of seconds to load everything, and then another couple seconds to run the simulation and get the result. Right now, I have that test run whenever the game grabs the stats of any weapon for display so that it’s always up to date.

Good stuff, huh? And the best part of it is that I’m really for reals all done now. Yup, I’m ready to check this bad baby in. You’ll finally have not just the rudimentary effectiveness stat you asked for, but the best stat-computation framework in the industry!

Actually—let me send it to a couple other people for a code review first. That should only take a couple more days. Then it’ll be done. ☺

**MATTHEW WASTELAND** writes about games and game development at his blog, *Magical Wasteland* ([www.magicalwasteland.com](http://www.magicalwasteland.com)). Email him at [mwasteland@gdmag.com](mailto:mwasteland@gdmag.com).



## BATMAN: ARKHAM CITY TAKES UNREAL ENGINE 3 TO NEW HEIGHTS

In the wake of the mega hit, *Batman: Arkham Asylum*, Rocksteady Studios, in collaboration with Warner Brothers Interactive Entertainment, has released the game's highly anticipated sequel, *Batman: Arkham City*. Like its predecessor, *Batman: Arkham City* uses Unreal Engine 3 to push game consoles to their graphical limits, and the result is heart-stopping excitement in an immersive, massively explorable world.

Critics and players alike are heralding the game as an innovation in the genre, and even more impressive than the original, with Game Informer calling it "The Best Licensed Video Game Ever Made."

Paul Denning, senior programmer at Rocksteady Studios, attributes much of the success to the fact that the studio sets a very high bar in all they do. Their aim is to hire extremely talented individuals, helping to contribute to the company's overarching desire to deliver the best possible game from every perspective. And Unreal Engine 3 was a big part of doing just that with *Arkham City*.

"Unreal Engine 3 is an amazingly powerful toolset," said Denning. "There are always new things to learn and new features being added. We had been using the Unreal Engine for about a year before we started *Batman: Arkham Asylum* and it really helped us learn how to tackle some of the key challenges in making a game of that style.

"However, without the experience of making *Batman: Arkham Asylum*, we'd never have been able to push the engine further for the sequel. System refinements coupled with the talent we have in the studio has allowed us to make a much richer world with more detail, characters and features than any other title we've ever made."

Part of the challenge of Rocksteady's overall vision for *Arkham City* was to add a living, breathing, open city as the centerpiece of the gaming experience.

Denning reports that this team worked right up to deadline in order to perfect the world. Their goal was to deliver a complex game world, replete with detail, from neon lights in every district to dazzling effects lighting up all corners of the world—all while maintaining 30 fps. The result, says Denning, isn't about the size, but about the level of detail, which he believes may be the most sophisticated in gaming so far.

"For *Batman: Arkham City* we spent quite some time on the Detective mode visuals," said Denning. "In *Batman: Arkham Asylum*, we had the Detective materials draw behind the post-process layer and used base colors to allow them to punch through the blue tint. This led to a very limited color palette and blocked out models. In *Batman: Arkham City*, we moved all the models that were highlighted into newly engineered layers that have given us so much more control over look and color."

According to Denning, one of the essential resources during the build was the Unreal Developer Network (UDN), which allowed the Rocksteady team to keep a close eye on key insights and solutions to challenges they encountered as well as offer their own advice and solutions when possible.

"Unreal Engine 3 makes so many areas of the game much simpler, yet at the same time more powerful, that we forgot how we even made games before learning some of the core areas of the engine," explained Denning. "The time I save writing code on previous projects that a designer can now simply hook up in Kismet means I can focus my efforts writing those key little moments that really set our title apart from our competition. I'd not recommend anything else."

In addition to UDN, Denning found the monthly updates

and upgrades to Unreal Engine 3 integral to the final success of the game. "While no single innovation or improvement led to unique gameplay aspects, I can say that the multitude of upgrades we saw over the development cycle all went towards truly pushing the envelope of what is possible in a third-person action adventure. We hope to be remembered as an outstanding game among the high-quality releases of the year. I'm most proud of delivering a fully-realized city at a solid 30fps that's more detailed and alive than any other game on the market with a graphical fidelity to rival any current generation game."

Judging from the overwhelmingly positive response to the game, that mission is accomplished.

At the time of this writing *Batman: Arkham City* is the second highest MetaCritic-rated PS3 game of all time (96) and the fifth highest MetaCritic-rated Xbox 360 game of all time (95). In addition, four out of the top 10 MetaCritic-rated Xbox 360 games of all time and three of the top 10 MetaCritic-rated PS3 games of all time use Unreal Engine technology.

WWW.UNREAL.COM



Canadian-born Mark Rein is vice president and co-founder of Epic Games based in Cary, NC. Epic's Unreal Engine 3 has won Game Developer magazine's Best Engine Front Line Award seven times, including entry into the Hall of Fame. UE3 has won four consecutive Develop Industry Excellence Awards.

Epic is the creator of the mega-hit "Unreal" series of games and the blockbuster "Gears of War" franchise. Follow @MarkRein on Twitter.

### UPCOMING EPIC ATTENDED EVENTS

Game Connection Europe  
Paris, France  
December 6 - 8, 2011



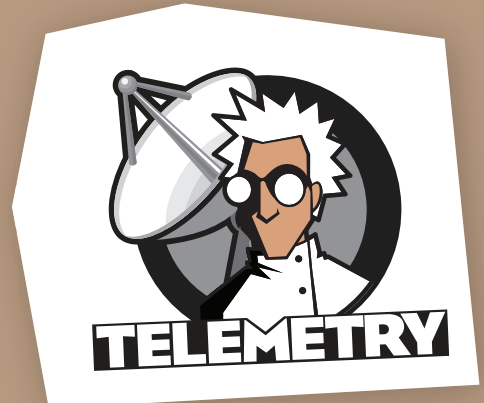
Please email [licensing@epicgames.com](mailto:licensing@epicgames.com) for appointments

THE NEWEST RAD TOOL IS

yes, it's **NEW**

**NOW SHIPPING**

**OUT**  
OF THIS  
**WORLD**



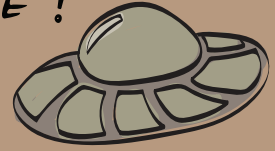
**TELEMETRY**

42

is a programming library and set of tools for instrumenting, profiling, tuning and visualizing application **PERFORMANCE!**



VISUALIZE real-time game performance,



see **WHEN** things happen — not merely **WHAT** happened!



PROBE the hierarchical display —



see thread interactions, context switches, and mutex locking!

THIS ISN'T JUST ROCKET SCIENCE, THIS IS **rad!**



[www.radgametools.com/telemetry](http://www.radgametools.com/telemetry)  
(425) 893-4300