GAME DEVELOPER MAGAZINE

THE LEADING GAME INDUSTRY MAGAZINE VOL 19 NO 9

SEPTEMBER 2012

INSIDE: SCALE YOUR SOCIAL GAMES

gd

GAME DEVELOPER MAGAZINE

PIXELJUNK 4AM

METROMTSOP

+ FIGHT THE LAG:

INSIDE GGPO'S NETCODE

UBM

The best ideas evolve.

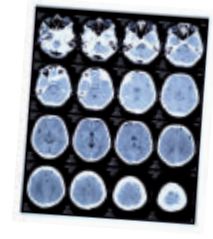
Great ideas don't just happen. They evolve. Your own development teams think and work fast.

Don't miss a breakthrough. Version *everything* with Perforce.

Software and firmware. Digital assets and games. Websites and documents. More than 5,500 organizations and 400,000 users trust Perforce for enterprise version management.

Try it now. Download the free 20-user, non-expiring Perforce Server from perforce.com/try20

Or request an evaluation license for any number of users.



PERFORCE

Version everything.

CONTENTS.0912
VOLUME 19 NUMBER 9

gd

GAME DEVELOPER MAGAZINE

postmortem

features

departments

HATERS GONNA HATE

DEALING WITH CRITICISM, AND WHY PLAYERS LOVE TO HATE WHAT THEY LOVE

MASS EFFECT 3 was the cap to a trilogy of games that players had come to feel emotionally invested in. They had maintained their characters and character alignment through dozens of hours of play, and the internet was rife with speculation about where the story could go. Then they finished the game—and by gum, some vocal people sure hated that ending.

So BioWare read the vitriolic comments, listened to the petitions, and made a new ending that tried to address some of these concerns. And guess what? The haters still hated it. We, collectively, have created a game community that thrives on hate, and sometimes there's not much you can do but grow thicker skin.

I LOVE TO HATE YOU

»The engagement we have with the community is a double-edged sword. It is fantastic (and necessary) to get fan feedback and make people feel involved in a product. But the internet is the "great equalizer," and WeedSmokeBalls187 now feels his voice is as important as that of a work's original creator. The ease of expressing opinion on the internet makes it easy for everyone to feel their voice is as important as any other, and the loudest voice will often become popular opinion.

Then there's how creators react. A comment saying "Wow, I love this!" doesn't get as much attention as "God, this is terrible." Everyone, game developers and journalists included, responds quicker and with greater force to negative reinforcement than positive. The squeaky wheel gets the grease, and to a voice that wants to get noticed, that's a powerful piece of knowledge.

I don't think this is something people do on purpose, but the more reactionary statements are made, and actions are taken based on negative comments, the more we show people that hating on products really works. It gets results, or at the very least reactions, and this has been absorbed into internet and geek culture at large.

Here's another recent example in games. Journalist Patrick Klepek recently lost a close relative, which greatly affected him, and he briefly spoke about it publicly. Not long after, Klepek wrote an article that some people disagreed with, including one person who saw fit to comment, "I'm glad someone close to you died."

Klepek was quite affected by this terrible statement, of course, and wrote about it on his blog. Now, I'm not saying he shouldn't have addressed it, but fewer people would write a blog post about how they had gotten some awesome praise from a random internet denizen. By and large, we as an industry reward negative comments with more attention than we give positive ones.

Very often, when I address a negative sentiment that's been left on my articles, or levied toward me personally, especially if I do it directly via email, the person will backtrack, saying, "Oh my god, I love your stuff. Thanks for replying. I'm sorry if I seemed rude, I was just thinking this and that..." The hate is expressed deeply, but is really only on the surface. It's born out of a deeper need for connection.

People love to hate the things they love. And why not? If they care about something deeply, any minor detail will stick in their craw, and they're more likely to get a response from the people who made the thing they like, if they make a stink about it. We reward it, directly.

THE POWER OF THE PREVIEW

»There's one big exception to this haterade trend in games, and that's the indie darlings of the world. The indie game community is generally much more open and supportive than is the triple-A game community, most likely because many in the indie game community are making games themselves. Nobody wants their first big effort stomped on, and so a culture of support is bred instead of one of negative reinforcement.

But it's interesting to watch these indies as they climb. Self-starters like Zeboyd Games were lauded early on for their revisionist Japanese-style RPGs, but as soon as they started to work on a large property (Penny Arcade's ON THE RAIN-SLICK PRECIPICE OF DARKNESS: EPISODE 3), the criticisms and negativity started streaming in. Once you're rolling with the big boys, it's time to get your shields up.

As an indie, showing your work early and often wins you points with your friends and your fans, and people will be very supportive of you. But when you're BioWare, and you change a character's armor slightly, you're going to get some praise, and a bunch of hate. And as I mentioned before, we're trained to seek out that hate, and to give it greater privilege than the love we receive. It's terribly hard to avoid.

IT'S ALIVE!

»This is a monster that we as professionals have bred and allowed to flourish. Community managers have tried to mitigate it with information leaks to fans, and many developers have had blowups about this very issue. Ignoring your fans is never the answer, and silencing voices of dissent is not very democratic.

I don't think anyone has found the ultimate answer. My advice is to ignore the hate publicly at first, but discuss it internally to see whether it has merit. If it's something you should really address, speak to those concerns as though they were well reasoned and nicely written. But whatever you do, don't feed the trolls.

—Brandon Sheffield
twitter: @necrosofty

GAME DEVELOPER MAGAZINE
WWW.GDMAG.COM

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*
www.gdmag.com/contactus

EDITORIAL

PUBLISHER
Simon Carless e: *scarless@gdmag.com*
EDITOR-IN-CHIEF
Brandon Sheffield e: *bsheffield@gdmag.com*
EDITOR
Patrick Miller e: *pmiller@gdmag.com*
MANAGER, PRODUCTION
Dan Mallory e: *dmallory@gdmag.com*
ART DIRECTOR
Joseph Mitch e: *jmitch@gdmag.com*
CONTRIBUTING WRITERS
Tony Cannon, Joel Poloney, Rowan Parker, Mike De La Flor, Alex Darby, Steve Theodore, Jason VandenBerghe, Damian Kastbauer, David Edery, Matthew Wasteland
ADVISORY BOARD
Mick West **Independent**
Brad Bulkley **Microsoft**
Clinton Keith **Independent**
Brenda Brathwaite **Loot Drop**
Bijan Forutanpour **Sony Online Entertainment**
Mark DeLoura **THQ**
Carey Chico **Globex Studios**
Mike Acton **Insomniac**

ADVERTISING SALES

GLOBAL SALES DIRECTOR
Aaron Murawski e: *amurawski@ubm.com*
t: 415.947.6227
MEDIA ACCOUNT MANAGER
Jennifer Sulik e: *jennifer.sulik@ubm.com*
t: 415.947.6227
GLOBAL ACCOUNT MANAGER, RECRUITMENT
Gina Gross e: *gina.gross@ubm.com*
t: 415.947.6241
GLOBAL ACCOUNT MANAGER, EDUCATION
Rafael Vallin e: *rafael.vallin@ubm.com*
t: 415.947.6223

ADVERTISING PRODUCTION

PRODUCTION MANAGER
Pete C. Scibilia e: *peter.scibilia@ubm.com*
t: 516-562-5134

REPRINTS

WRIGHT'S MEDIA
Jason Pampell e: *jpampell@wrightsmedia.com*
t: 877-652-5295

AUDIENCE DEVELOPMENT

AUDIENCE DEVELOPMENT MANAGER
Nancy Grant e: *nancy.grant@ubm.com*
LIST RENTAL
Peter Candito
Specialist Marketing Services
t: 631-787-3008 x 3020
e: *petercan@SMS-Inc.com*
ubm.sms-inc.com

UBM
WWW.UBM.COM

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
- Guidance and liaison with UAE content regulatory bodies
- Easy licensing and business set-up services
- Dedicated fund for mobile apps development via Apps Arabia™
- Unique campus environment with facilitated business networking
- **twofour54°** gaming academy in partnership with Ubisoft®
- The region's only stereoscopic 3D Lab
- 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

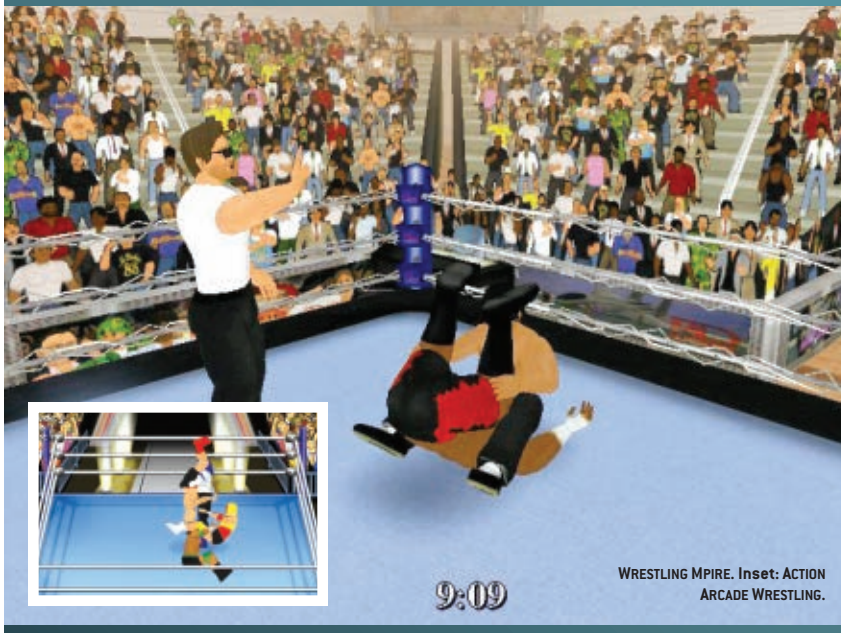
Sources: Arab Media Outlook 2010. Media on the Move 2009. A.T. Kearney. Introduction to Gaming. Michael Moore. Screen Digest. IDC.

twofour54° is an initiative of the Abu Dhabi Government.

IT'S REAL TO ME

INDIE WRESTLING GAME DEVS ADD NEW LIFE TO THE GENRE

//////// Wrestling games are largely a one-company genre; if you have the WWE license, you're making the biggest game in town. But when wrestling fans aren't satisfied with where the big game is going, it's up to the independent developer to turn the match around. *Game Developer* spoke with WRESTLING REVOLUTION developer Mat Dickie about the nascent indie wrestling revival.



WRESTLING MPIRE. Inset: ACTION ARCADE WRESTLING.

Patrick Miller: Who are the main players in this indie wrestling revival?
Mat Dickie: Ten years ago, Dave Wishnowski started a movement called Wrestling Gamers United in response to what he and many others saw as the mainstream developers' collective mishandling of the genre. He put his money where his mouth was and bankrolled a project of his own called PRO WRESTLING X, which he's hoping to release soon. Meanwhile, Dave Horn's ACTION ARCADE WRESTLING is available on the Xbox Live Indie Games Marketplace, and he has started work on a second, more ambitious installment. Dan Hinkles of Serious Parody has secured a £1m ($1.56 million) investment in his WRESTLING MANAGER project for iOS. Finally, I have released the WRESTLING MPIRE series on PC and WRESTLING REVOLUTION for Android, and I have an iOS version in the works.

PM: Do you consider each other colleagues, or competitors?
MD: The way I see it, Dave Wishnowski caters to PC, Dave Horn caters to Xbox, Dan Hinkles caters to iOS, and I'm getting the best out of Android. We're all on the same page, and we're going about it in our own unique ways so there is no competition. Collectively, we're bringing independent content to separate corners of the industry.

PM: How did this indie wrestling revival start?
MD: Well, we are all disappointed with the state of mainstream wrestling games. We thought that THQ had a winning formula with WWF NO MERCY on Nintendo 64, which had inherited a Japanese game engine. When THQ took the license in their own direction, a lot of diehard fans had to endure games that looked the part but lacked substance and playability. Dave Wishnowski has said he is trying to recapture the golden era of Nintendo 64 with his PRO WRESTLING X.

PM: Is there anything you get to do with your wrestling game as an indie that a licensed dev can't?
MD: Our independent games depict a fictitious universe where anything can happen and creativity outranks commercial obligations. I get a kick out of taking things behind the scenes and exposing the realities of the business. A licensed product could never go that far behind the curtain. I can't think of any other sport or license where the contrast between the mainstream view and the independent view is so stark.

PM: Do you think the wrestling market could support more devs and more games?
MD: The wrestling demographic has always been good to me professionally, because you have a small but passionate audience who are prepared to invest in the entertainment they love. That's what first motivated me to make wrestling games for PC. Each platform is dominated by WWE primarily, but there's a heart-shaped hole where an independent can make his presence felt.

PM: Are you working in in-app purchases?
MD: That business model would be the holy grail for us because wrestling games are bursting with content that could be monetized, while still making the fundamental gameplay available to all. It's definitely something I'll be investigating in mobile. That said, I like to keep things simple, so we'll have to see if it's a headache to manage. As a solo developer, I have to pick my battles carefully.

PM: How is it going from a primarily console-based game niche to mobile?
MD: You can't fully recreate a console game on a smaller device, so you have to consider what sacrifices you're willing to make. I made a conscious decision to rediscover my 2D roots and put gameplay at the forefront. A lot of longtime fans felt that 2D was a step backward for me, but I could either make an unplayable 3D game or a playable 2D game. Sometimes you have to go backward to go forward!

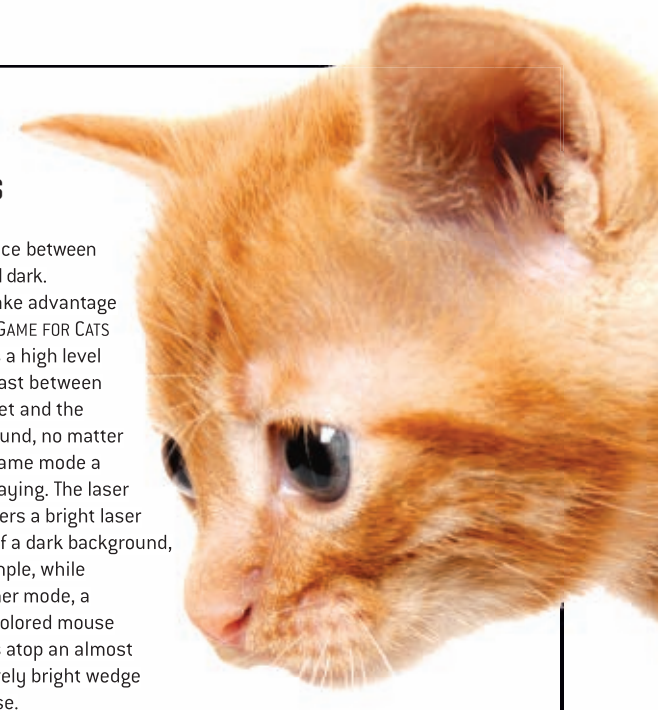
The whole transition has actually been a blessing for me because I seem to have found my place. When I was making big 3D games for PC, all people did was make unreasonable comparisons to mainstream releases. WRESTLING REVOLUTION is an intentionally retro project, so everybody gets it and accepts it for what it is.

Also, players have fewer preconceptions about how things should work. It wasn't easy to squeeze so many commands into a few swipes and pinches, but I feel I've delivered something that goes beyond a gimmick and has become a credible way of engaging with wrestling.

—Patrick Miller

GAMES FOR CATS

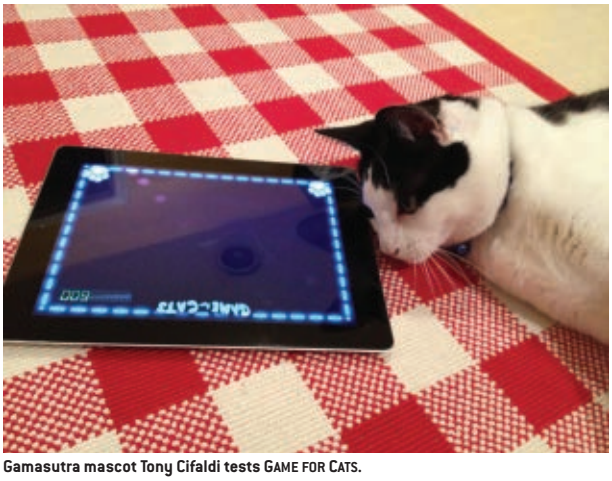
TIPS FOR FELINE-FRIENDLY DIGITAL AMUSEMENTS



The two-man team of artist TJ Fuller and programmer Nate Murray at developer Hiccup had something of a flop with their video game debut: JACOB'S SHAPES, a simple iPad puzzle game aimed at children. Perhaps children weren't their forte. Perhaps they needed a new audience.

Noticing several YouTube videos of cats pawing at iPad games (the human-focused kind), Fuller decided to try to make a game specifically aimed at them—something that could be produced and put on the App Store quickly so they could go back to making "real" games. The resulting game has one simple mechanic: an object on the screen (either a laser light, a mouse, or, in the latest update, a butterfly) moves around the screen in lifelike patterns, enticing the cat to touch it. Cats are scored on their performance, and the game even incorporates Game Center, to make sure kitty's best scores are saved for all to see.

Just three weeks and one playtest at a local animal shelter later (neither Fuller nor Murray are cat owners), GAME FOR CATS debuted, drawing major media attention and enough sales to justify both a sequel and several copycats (pun intended).



Gamasutra mascot Tony Cifaldi tests GAME FOR CATS.

Though it isn't huge (yet), there is now a legitimate market for video games aimed at felines. So we sat down for a quick phone chat with Fuller for a list of best practices behind what makes GAME FOR CATS tick.

1 // SUBTLE, NATURAL MOVEMENTS

Cats can detect subtle movements better than we can. If a cat doesn't sense that its target is "alive," it's going to ignore it. The first attempt at movement in GAME FOR CATS was done purely through code, but everything "felt way too mechanical," Fuller tells us.

The solution was to throw out that code and record finger movements on the iPad. When the laser (or mouse) moves around in the game, creeping around slowly one moment and darting offscreen the next, that's programmer Nate Murray's finger emulating life.

2 // HIGH-CONTRAST VISUALS

There is still some debate as to exactly what cats can and cannot see in terms of color, hue, and saturation, but no one disagrees that they can tell the difference between light and dark.

To take advantage of this, GAME FOR CATS ensures a high level of contrast between the target and the background, no matter which game mode a cat is playing. The laser level offers a bright laser on top of a dark background, for example, while in another mode, a darkly colored mouse scurries atop an almost offensively bright wedge of cheese.

3 // CATS LOVE DLC

Like many mobile games, GAME FOR CATS is a free download that monetizes itself with in-app transactions: specifically, the download includes the laser level for free, and offers the mouse level for an optional 99-cent transaction.

Unfortunately, the initial release of the game made that purchase path a little too easy; in the days immediately following the game's release, cats everywhere were accidentally purchasing the 99-cent level without their owners' permission.

"We got in a lot of trouble," Fuller laughs. "People were accusing us of tricking cats into making purchases. We got a ton of comments on our iTunes page [from] people accusing us of trying to rip them off."

The solution was to implement a test to make sure the purchaser is human before the charge is allowed to go through. Specifically, the purchaser is asked to place a hand on the screen and hold it there while the app "scans" it. In reality, the game makes sure that four fingertip touch points don't move for a few seconds—a test even the craftiest of cats would have a hard time circumventing.

4 // NO PAUSE FOR PAWS

Hiccup wanted to implement a way for humans to pause the game (allowing them to switch levels or, perhaps, tweet kitty's high score), but doing so in a way that prevented cats from pausing the game themselves was a design challenge.

The solution, according to Fuller, "still isn't perfect"; that humans have to tap a specific section of the screen five times in rapid succession to pause the game. It works pretty well, but Fuller says that cats still set it off by accident occasionally.

5 // REWARD WITH A SOUND

Whenever a cat successfully scores in the game (in other words, touches the moving target), its score increases and a sound plays to reward the cat and keep it engaged; the laser chimes, the mouse squeaks in terror.

Fuller warns that while there is temptation to have music in the game, he thinks it would have ruined this effect.

"If there's music playing, they wouldn't hear it," he says. "The sound has to be meaningful, and with purpose."

–Frank Cifaldi

Unreal Engine 3 Kinects with Fable: The Journey

Lionhead Studios is embracing Microsoft's Kinect for Xbox 360 with its latest *Fable* game. *Fable: The Journey* is set a few years after *Fable 3* and unveils a brand new area of Albion to gamers.

Players take control of Gabriel, an outsider who becomes a hero as a result of the seer Theresa, who has been in the background of previous *Fable* games. The new game, which was developed using Unreal Engine 3 (UE3), was designed from the ground up to fully embrace Kinect technology.

"Kinect gave us the chance to let the player feel more involved with the world," said Charlton Edwards, lead level designer, Lionhead Studios. "It gave a physicality to some interactions that a button simply can't replicate.

According to Ben Brooks, senior scripter, Lionhead Studios, the company's level design and gameplay scripting teams – 19 members in total – worked much more closely together on this game than previous *Fable* titles because the Unreal Engine's toolset united these disciplines in a way that the studio's old tools didn't.

"Our level design and gameplay scripting teams have used Unreal Kismet extensively – it's our primary tool for bringing the world to life," explained Brooks. "It's allowed us to very rapidly prototype, and the visual interface has opened up quest creation to people who might view themselves as non-technical. The accessibility of the tools has really helped us be self-reliant as a team."

Brooks said that Unreal Matinee, in particular, was something of a revelation. On previous games, the camera placement was done by noting camera location and facing vectors and interpolating between them with script, which was a slow and clunky process. Being able to craft cutscenes in Matinee gave the team much more control and really empowered them to finesse their work.

Edwards said a couple of the nine level design team members had Unreal Engine experience. For the rest it was a pretty gentle learning curve.

"The nature of the game means we didn't need to get right under the hood and bang our heads on the bonnet," he said.

Edwards said the level design team used Matinee and Kismet to quickly block out puzzles and environmental animations so they could get their vision across and try out ideas without just showing someone something scrawled excitedly on a piece of paper.

"My main reason for choosing UE3 was tools and iteration," said Marcus Lynn, technical director, Lionhead Studios. "I wanted to get a smile back on the faces of content creators again where they wanted to go and improve something and feel like they had the support and tools they needed."

Lynn said his team started the evaluation with a skeletal team and quickly got up-to-speed on most aspects at a basic level. New engineers on the project were given a two-week period to write something fun using UE3, which proved successful.

Lionhead adopted additional Unreal Engine technology tools such as Unreal Landscape and the foliage system, which helped reduce memory overhead and improve performance.

"We have managed to create a world that feels complete and cohesive," said Edwards. "There is a real sense of scale and place with many magnificent vistas and always a sense of traversing a huge land."

The development process to bring Kinect functionality to life was a smooth one. The team ensured that they had Kinect initialized and updated at the correct points and that the interfaces were clear enough for the gameplay and level scripters to use in UE3.

"The Unreal Developer Network (UDN) has been an invaluable resource for us in both discussing common issues among developers and Epic, but also for getting quick answers and resolutions using the forum history and the combined knowledge of the other developers using it," said Lynn.

With the game nearing completion, the team is happy with what they've been able to accomplish.

"It is a lovingly hand-crafted world, and we have tried hard to marry a lot of detail with the need to keep the frame rate high," remarked Edwards.

After three successful RPG adventures in the *Fable* series, Lionhead Studios hopes players will embrace the same type of innovation it pioneered in console games with Kinect technology.

Thanks to Lionhead for speaking with freelance reporter John Gaudiosi for this feature.

UPCOMING EPIC ATTENDED EVENTS

Escapist Expo
Durham, NC
September 14-16, 2012

MIGS
Montreal, Canada
November 13-14, 2012

EPIC GAMES

Please email licensing@epicgames.com for appointments

TONY CANNON

FIGHT THE LAG! THE TRICK BEHIND GGPO'S LOW-LATENCY NETCODE

Some of the best multiplayer arcade games on consoles or PC are absolutely ruined by lag when played online. It's easy to understand why; if you're playing a game that rewards precise timing and control (such as brawlers, shoot-'em-ups, and fighting games, for example), you want your experience to be as close to lag-free as possible, or the game will feel like no one is playing as well as they think they should. In this article, I'll walk through how I designed my GGPO netcode (short for "Good Game, Peace Out"), which was used in SKULLGIRLS, STREET FIGHTER III: 3RD STRIKE ONLINE EDITION, and plenty of emulated arcade games through the FinalBurn Alpha emulator. It's handy for devs looking to mask and minimize the effects of lag in online multiplayer—particularly in games in which timing-based skill is key.

GGPO is compatible with all arcade-style games that meet the following criteria: 1) Each update to the game simulation state must be deterministically derived solely from the player inputs and the previous simulation state, 2) the game must be able to update its simulation state independently of sampling the controller, rendering the video, or rendering audio, and 3) the game must be able to save and load the simulation state and execute on demand. GGPO handles all the details. The game developer simply needs to modify her game loop to allow for speculative execution.

IDENTIFYING THE CHOKEPOINT

The most common method of bringing multiplayer games online is to run a separate simulation on each console and keep them synchronized by ensuring that each simulation gets exactly the same inputs (see **Figure 1**). If the simulations are determined solely by the inputs, playing the same inputs on both consoles will result in the same output. This method has a lot of plus sides to it; computers are by nature deterministic and most arcade games use integer math exclusively, which is very well behaved from processor to processor, and the developer is also completely isolated from the details of the network implementation. Aside from making sure the game runs deterministically, the engineers and designers implementing the gameplay can be isolated from the details of the network engine. This means you can retroactively add online multiplayer support to games that have already been released. (If you can provide a machine emulator for the original hardware the game was written on, you don't even need to recompile it!) Finally, this method has no server-based components aside from a method required for basic matchmaking, which is usually provided by the console manufacturer.

Unfortunately, this method also comes with one major drawback: These games are usually designed to sample the controller input before every simulation state update. A simulation frame in the game cannot execute until all the inputs from remote players have been received. In practice, this manifests itself as an input delay equal to the time it takes to send a packet from one console to the next. In short, the games lag. The instant response the game was designed for is replaced by a sluggish and squishy control that can ruin the players' experience.

In many cases, the delay introduced by the networking layer can completely change the feel of the game. In STREET FIGHTER, we call them "lag tactics": using the knowledge that by the time your opponent sees your move it will be too late

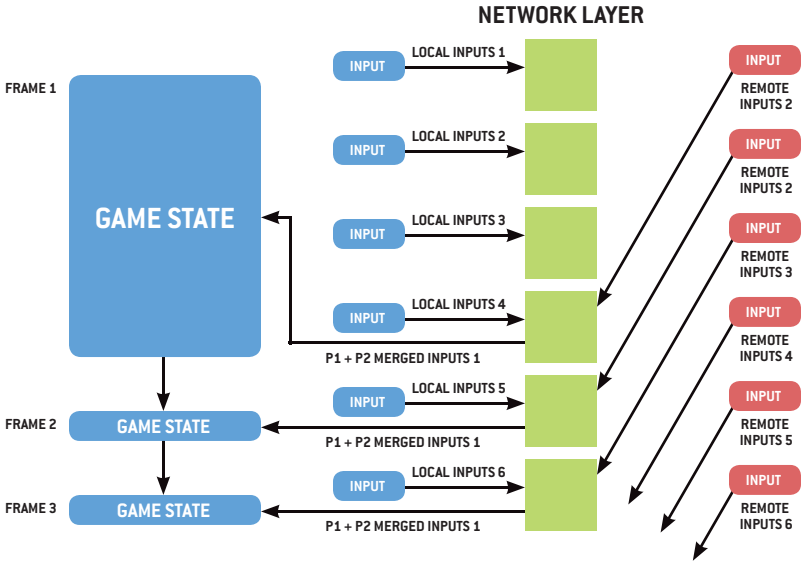


FIGURE 1: Implementing network play with the traditional frame-delay method.

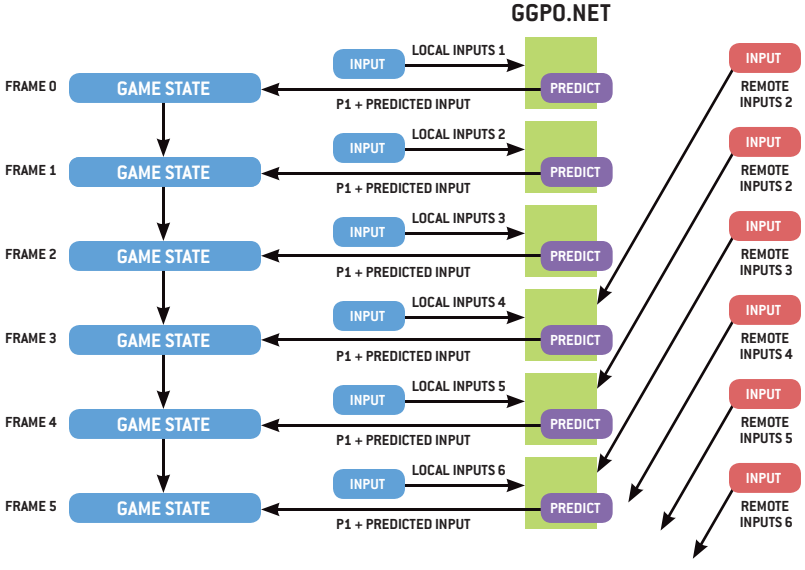


FIGURE 2: Breaking down GGPO's prediction mechanism.

to respond. To many, including myself, lag tactics greatly influence the competitive experience of the game online.

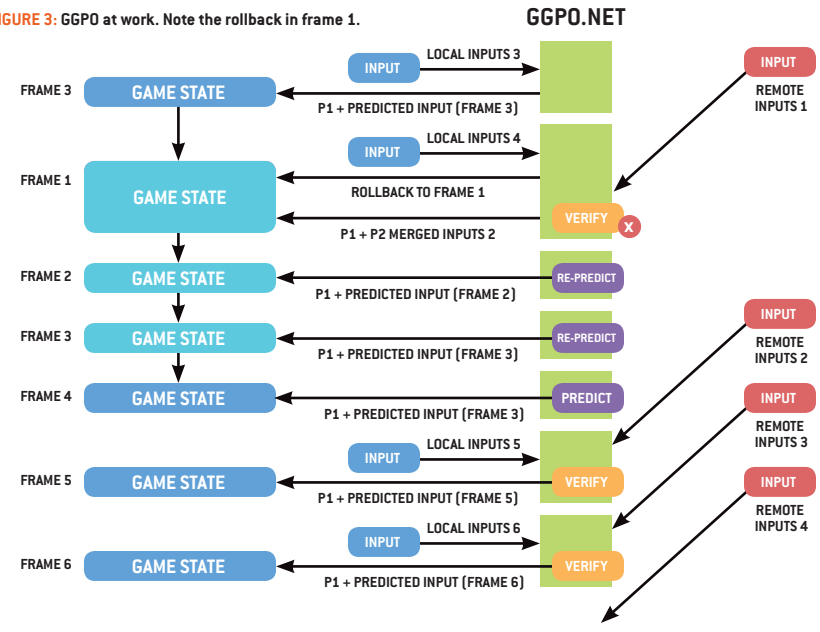
I wrote GGPO to try to do a better job of solving the lag problem, without sacrificing all the other great properties of this method. Instead of adding an input delay to the entire simulation, GGPO gives the local player immediate control over his avatar by hiding the latency of the connection in the start-up of each remote player's action. This does a much better job of faithfully reproducing the offline experience in the online game.

THE SOLUTION: GGPO

GGPO uses speculative execution to eliminate the perceived input delay for each local player. Instead of waiting for all inputs to reach a player's simulation before executing a frame, GGPO will guess what remote players will do based on their previous actions (see **Figure 2**). This eliminates the lag experienced by the local player created by the traditional frame-delay method. The local player's avatar is just as responsive as when playing offline. Though the actions of the other players cannot be known until their inputs arrive, the prediction mechanism used by GGPO ensures that the game simulation is correct most of the time.

When GGPO receives the remote inputs from the network, it compares the predicted inputs to the actual ones. If it finds a discrepancy, it rewinds the simulation back to the first incorrect frame (see **Figure 3**), repredicts the inputs for each player based on the updated input stream, and advances the simulation to the current frame using the new prediction.

FIGURE 3: GGPO at work. Note the rollback in frame 1.



HIDING THE LAG

In GGPO, the local actions performed by the player always happen instantly, and are always correct. This is a great property to have, as the responsiveness of the avatar to the controller is often the most important aspect in the player's enjoyment of the online experience.

Furthermore, any long-term effects caused by remote players whose inputs have already been received are also correct. For example, if your opponent in STREET FIGHTER threw a fireball at you a few frames ago, the behavior of that fireball is completely deterministic and cannot be affected by your opponent's future inputs. Therefore, the fireball will appear to move correctly immediately after your local simulation state has determined

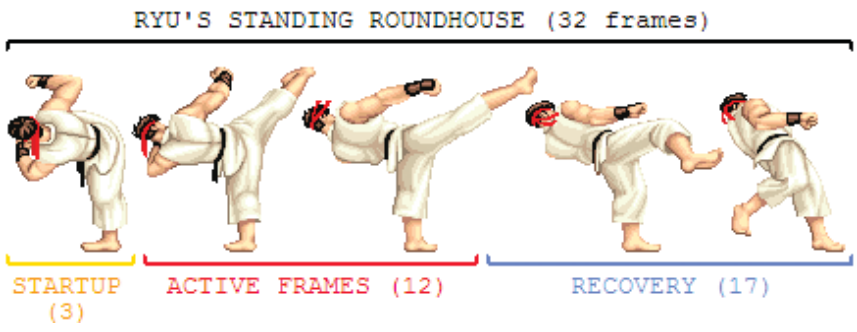


FIGURE 4: On most broadband connections, GGPO can mask the lag in the start-up phase of a Street Fighter move.

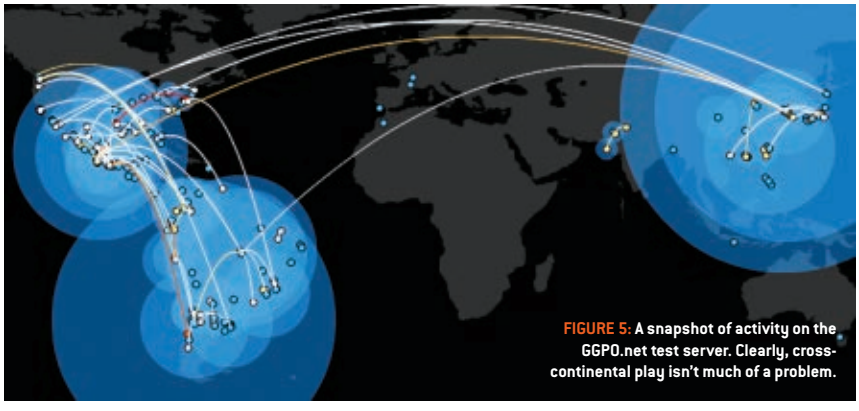


FIGURE 5: A snapshot of activity on the GGPO.net test server. Clearly, cross-continental play isn't much of a problem.

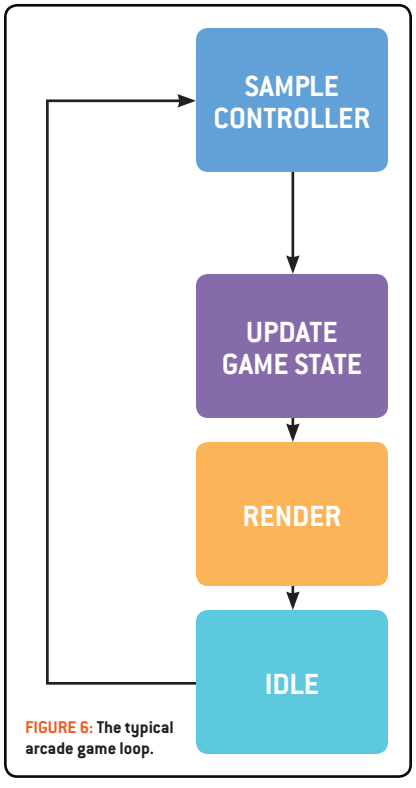


FIGURE 6: The typical arcade game loop.

that the player actually did throw the fireball. This makes the timing and experience of dealing with a fireball online identical to the experience offline, which is important, as dealing with fireballs is a major part of STREET FIGHTER!

Similarly, opponents usually cannot change the arc of their jumps after initiating them, so dealing with your opponent descending from a jump, perhaps with a well-placed Dragon Punch, is identical both online and offline. So if everything appears to be correct all the time, where'd the latency go?

The latency is hidden in the window between when your opponent initiates an action and your simulation realizes that an action was performed. The time lost in that window is effectively skipped to your simulation. For example, suppose both you and your opponent are playing a game of STREET FIGHTER on a network that takes 60ms to send a packet from one console to the next. When your opponent executes a move, his simulation will process the controller inputs immediately. To him, the move comes out right away, since local inputs are sent to the simulation immediately and are always correct. Your simulation, however, will not notice that your opponent performed a move for another 60ms, when a packet arrives from the network carrying that input. When it does, GGPO will instruct the game to rewind 60ms, correct your opponent's input, and fast-forward the game simulation 60ms back to the current time. As a result, on your console you will not see the first 60ms of animation of whatever your opponent did. It is as if the move began 60ms into the animation, from your perspective.

This is not ideal, but the alternative is to delay the entire simulation by 60ms, including local inputs. In practice, losing those 60ms of animation usually results in a greatly preferable user experience. This is partially due to the greatly increased responsiveness of local actions, but is also because most of the time those 60ms just don't matter that much. To illustrate this, let's look at a specific example.

Most attacks in STREET FIGHTER have three phases: start-up, execution, and recovery (see **Figure 4**). The start-up of a move is how long a move takes to become active after the user presses the button. While there is usually some animation state associated with the start-up of a move, the move isn't actually doing any damage yet. The serious business happens in the execution window; if the opponent overlaps your move's active region at any time during the execution window, the game simulation will register it as a hit. A hit causes the simulation to start new animations, play audio, subtract some life from your opponent, and lots of other effects. It's a big deal as far as simulation state is concerned. Recovery is simply the duration after a move executes before you can perform another one.

These numbers are usually measured in frames, and the first thing competitive STREET FIGHTER players do when a new game comes out is to mine the frame data for every move in the game to begin researching tactics. If we look at the frame data for one of the most beloved and fastest STREET FIGHTER games on the market, SUPER STREET FIGHTER II TURBO (http://nki.combovideos.com/flame.html), we see that a vast majority of the moves have a start-up of at least four frames (or 66ms). That's incredibly important to us, because it means that on a 120ms ping connection, it's possible to resolve almost every rollback before the execution of a move, which means that the visual and audio glitches that result from incorrect predictions are almost always limited to the animation of the start-up of a move, not the result of hitting your opponent.

Modern broadband connections from Los Angeles to New York typically have a faster ping than 120ms. In fact, anecdotal evidence

shows that GGPO's techniques scale well up to latency experiences on broadband connections worldwide; see **Figure 5** for a snapshot of the GGPO.net test server activity on a typical evening.

INTEGRATING GGPO

GGPO is written to isolate the developer from the network details as much as possible. **Figure 6** demonstrates a simplified game loop for an arcade game.

The loop samples the controller for the game to generate inputs for the next simulation state of the game. Those inputs are passed into some simulation engine to update the game to the next frame and render the result of each simulation step. Many timing-sensitive arcade games will drive their game-loop at a fixed rate (e.g., 60hz), in which case it may be necessary to idle before sampling the next controller state for the next frame. There are many variations of this game loop. For example, a game may only sample the controller and update the game state at 30hz, but still render up to 120hz by interpolating between the last two game states. GGPO is equally suitable to this scenario as well.

Figure 7 shows a game loop that has been modified to incorporate GGPO. After sampling the controller, the developer should pass the game inputs to GGPO via the ggpo_synchronize_input function; ggpo_synchronize_inputs will transmit all local inputs to remote players. It also merges predicted and actual remote inputs for all remote players in the game into the input stream. The result is a tuple of controller inputs suitable to be passed to the local game engine, with one input for each player in the game.

The bulk of the developer's game engine can remain unmodified, aside from some potential caveats mentioned later. Instead of idling before returning to the top of the game loop, the developer should call the ggpo_advance_frame function, which will compare inputs received for all remote players to their predicted values. When it finds a discrepancy, ggpo_advance_frame loads the last correctly predicted frame and executes the game's update game state function repeatedly, passing the corrected inputs to each subsequent call, to fast-forward the game simulation back to the current frame. The load, save, and execute functions are provided to GGPO by the developer at initialization time.

SYNCHRONIZING THE CLOCK AND INPUTS

GGPO uses a simple and efficient protocol for synchronizing inputs between players in a session (see **Listing 1**). The header of each packet contains a 16-bit, session-specific identifier followed by the type of the payload. The three major packet types are quality_report, quality_reply, and input.

```
STRUCT PACKET {
STRUCT {
     UINT16     MAGIC;
     UINT8      TYPE;
} HDR;
UNION {
          STRUCT {
     UINT32          START_FRAME;
     UINT32     ACK_FRAME;
     UINT16     NUM_BITS;
     UINT8      BITS[MAX_COMPRESSED_BITS];
} INPUT;
STRUCT {
     INT8            FRAME_ADVANTAGE;
     UINT32          PING;
     } QUALITY_REPORT;
          STRUCT {
          UINT32               PONG;
     } QUALITY_REPLY;
     } INPUT;
  } U;
}
```

LISTING 1: A simplified description of GGPO's UDP packet payload.

One input payload is sent for every call to ggpo_synchronize_inputs. The inputs are compressed using a state machine to toggle virtual buttons. The machine begins with an empty input vector at frame 0. Inputs for each frame are encoded into the bits array by encoding the buttons that have changed from the previous frame using the following fairly trivial coding system:

```
1 + <5-BITS OF N> : TOGGLE BUTTON N FROM THE
PREVIOUS INPUT
0 : END INPUT
```

Almost all the inputs in a stream compress down to a single bit (0). Even an extremely twitchy game where the user toggles 10 inputs per second compresses down to around 300 bits per second. This is small enough that all inputs from the previously acknowledged frame are encoded with every packet, which makes handling dropped packets and out-of-order packets extremely simple. GGPO keeps a history of the last x inputs received from a remote player, where x is a window larger than the prediction barrier. Whenever an input is received from a remote player, load the start_frame input from the window and execute the bits state machine to recover all subsequent inputs. If we generate any inputs beyond our window, they are sent to the prediction subsystem for verification and stored in the window. Finally, remember the number of the last generated input so it can be sent to our peer in the ack_frame field of the next input packet.

GGPO will periodically send a quality_report packet to measure the fairness of the connection. Each quality-report packet contains a timestamp generated on the current machine and this peer's measure of its local "frame advantage." When a peer receives a quality_report message, it immediately sends a quality_reply response, copying the ping value from the report into the reply. The originating peer measures the round-trip time for a packet by subtracting the current time from the pong time in the last quality_reply packet.

"Frame advantage" is GGPO's concept of how much of an advantage, in frames, the local player

TESTING GGPO WITH EMULATORS

GGPO was developed, tested, and optimized using the FinalBurn Alpha emulator for PCs. FinalBurn was an ideal choice: Its source license is rather permissive, and it supports a huge library of arcade games. Emulators are also especially good at encapsulating the entire game state in a compact format, which was ideal for GGPO; the SaveState() andLoadState() callbacks for GGPO are based almost entirely on the Save Game and Load Game functions from FinalBurn Alpha.

Emulators also tend to generate an exact amount of audio and video state at some fixed frequency (usually 1/60th hertz to match the average arcade monitor's refresh rate), which lets us avoid many of the video and audio glitches that can occur during mispredictions, since the entire state of the rasterizer and sound hardware is restored in a rollback. And once an emulator is written, debugged, and integrated with GGPO, all games that run on that emulator gain the benefit of networked play, without needing to modify the source of the emulated game.

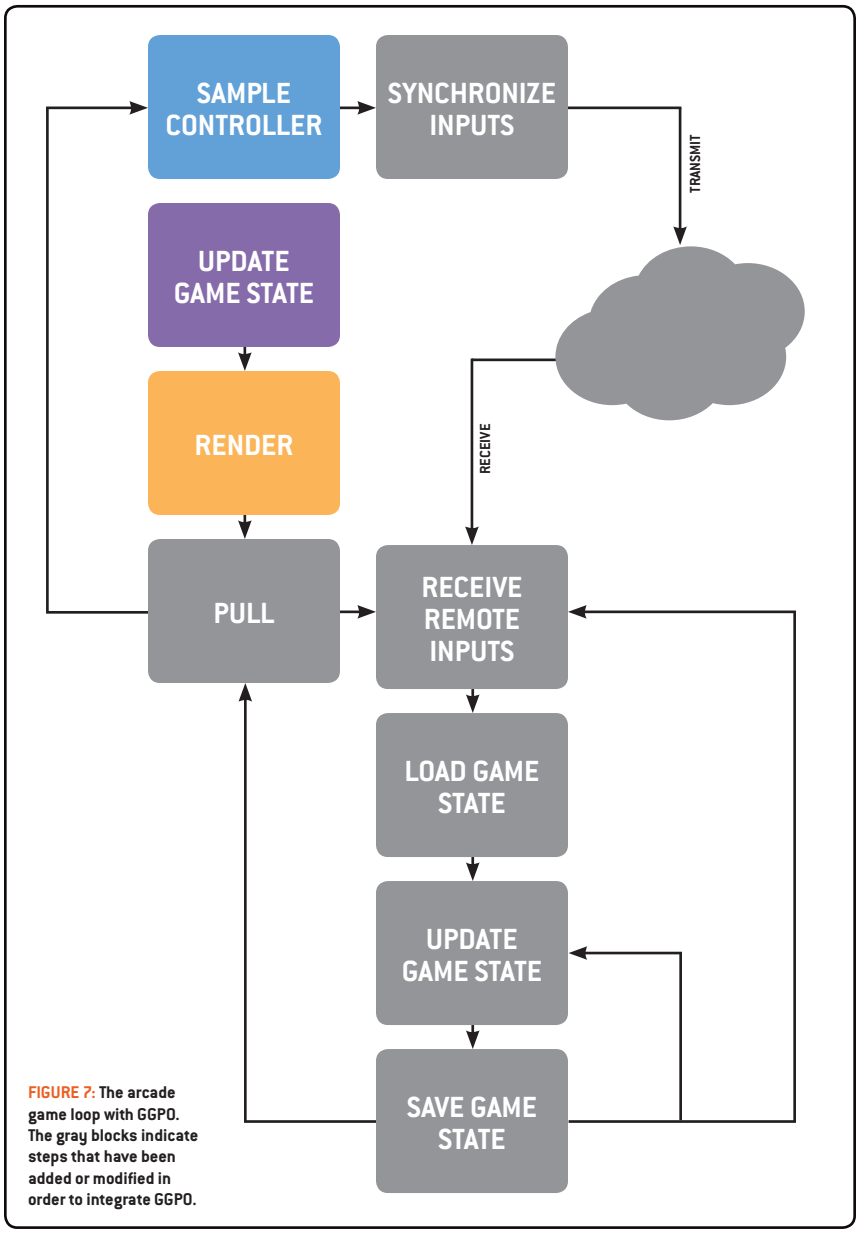


FIGURE 7: The arcade game loop with GGPO. The gray blocks indicate steps that have been added or modified in order to integrate GGPO.

has due to wall clock skew. It is calculated using the following formula:

```
FRAME_ADVANTAGE = (LAST_REMOTE_FRAME + (PING * FRAME_FREQUENCY / 2)) - LAST_LOCAL_FRAME
```

That is to say, we estimate the frame our peer is rendering now by adding half the round-trip packet time to the last packet we received and subtract the frame we are currently rendering. The result will be the number of frames that our peer is ahead of us. For example, suppose we calculate that the frame advantage is 2. This means we believe a neutral observer located equidistant from our two games and equipped with a very good spyglass would see my game rendering frame 20 at the exact instant an opponent's game is rendering frame 22. GGPO considers this an "advantage" because it means our simulation will only need to rollback two fewer frames than our peer's simulation, meaning his version of game reality is, on average, more out-of-date than my own. There are many things that can negatively impact the calculation of an accurate frame-advantage value, including asymmetric packet transmit times, intermittent connection issues, and so on, but this formula seems to work well in practice.

Each connection peer in a GGPO session is always aware of his local frame advantage and receives periodic updates as to his peer's calculation of frame advantage. GGPO will attempt to keep the game "fair" by making sure that the local and remote frame advantages agree to within a one-frame tolerance. For example, if a GGPO endpoint consistently computes a local frame advantage of five while paired to a peer whose local frame advantage is consistently three, GGPO will attempt to equalize the disparity by slowing the execution of the local endpoint down by one frame. This should result in a loss of one frame for the local end and a gain of one frame for the remote end, equalizing the frame advantage of both to four frames. The implementation of exactly how to slow the local endpoint down is handled by a callback provided by the game developer.

SEPARATING GAME STATE FROM RENDERING

It should be clear by now why GGPO requires that you separate your game logic from rendering: For each video frame rendered, the game state may need to be updated many times to re-evaluate the current state from a mispredicted remote input. If the game engine had to rerender those frames in addition to recalculating the game state, GGPO's rollback technique would be prohibitively expensive. For example, while your video renderer may require all sorts of inverse kinematics, soft body simulation, and other expensive calculations for visual effect, to use GGPO you should strive to build your game simulation in such a way that it does not require them.

The game's renderers must also be capable of removing assets from a mispredicted previous frame. For example, a frame that created a fireball and played a whooshing sound might not actually be the correct result when all inputs from remote players are received. This is fairly easily handled for video, but can be tricky for audio. There are two methods for handling audio that have been used successfully.

The first is to treat audio as part of your simulation state. If your game-state is updated 30 times a second, then you should include 33.3ms of audio samples for each channel which should be rendered along with your game state. When it comes time to render audio, render only those 33.3ms per channel rather than the entire effect. This is relatively easy to implement, but can result in some audio popping during long rollbacks.

The second method is to keep in the game-state a list of all previously rendered audio effects and the frame during which they were started. This window need only be as large as the GGPO-configured maximum buffer window. When rendering audio, compare the list of effects in the game state to the currently queued effects in the audio device. Effects that exist in the game state but have not been queued should be sent to the audio device. They may need to be started

several milliseconds into the effect if the effect was created as part of a rollback. (Starting the effect at 0% volume and gradually amplifying it up to 100% can alleviate the popping effect that might occur when this happens.) Effects that are in the audio queue, but not in the simulation state, are ones that have been revoked as a result of a rollback. Playing them to begin with was an error and they should be stopped, potentially after attenuating them down to 0% volume over several frames to remove popping.

GETTING IT ALL RIGHT THE FIRST TIME

While the SDK is very easy to use, very complicated games, or games that were designed and written without GGPO in mind, may run into several stumbling blocks, the first of which was just discussed: The video and audio renderers must be divorced from the game's simulation state. Secondly, the game developers must ensure they have the CPU budget to execute their simulation update step many times per frame. Ideally, you would like your game simulation to run fast enough to execute an extra four to five times per frame, which is enough to mask about 80ms of latency. Finally, the game's simulation must be completely deterministic and determined solely from the player inputs.

When playing the game with local players, there's never any reason to do a rollback, as all inputs are always 100% correct and received immediately from the local controllers. As a result, the code paths implementing these three features are almost always only tested while the networking portion of the game is being tested or implemented. This is usually a small fraction of the total test time a game gets.

To help developers port their game to GGPO, a special debugging mode called "sync test" is included. When initializing GGPO in sync test mode, the SDK will treat every game-frame like a mispredicted remote input frame, even when only local players are playing. Every frame will load, re-execute a configurable number, and save the game state for each call to ggpo_advance_frame. This greatly speeds the development and test time of integrating GGPO into a game, as the performance and correctness of the most difficult bits can be implemented and debugged without requiring a remote session, or even a network.

There is no silver bullet for slaying the lag monster, but GGPO gives game developers another tool for wrestling it into submission. With GGPO, developers can reduce input lag in their game to levels below the one-way packet transmission time without greatly complicating their game-loop or engine design. Speaking as someone whose gaming roots go deep into arcade culture, I hope that developers out there will use GGPO or the techniques described in this article to bring lag-free, online gaming to their titles in the future.

TONY CANNON *graduated from Stanford University in 1995 with a BS in computer science. He has worked at VXtreme Inc., Microsoft, and is currently employed by VMware and Radiant Entertainment. Tony has a passion for fighting games and is a co-founder and tournament director for the Evolution Tournament Series.*

GGPO'S PREDICTION ALGORITHM

A good prediction algorithm can minimize the visual glitches that can occur during a rollback. The prediction algorithm's job is to anticipate the player inputs arriving from the network at frame N + (L / F) given the inputs for all previous frames 1...N, where N is the input last received from the remote player, L is the one-way packet transmission time, and F is the frequency at which the game executes its game state.

The quality of the prediction algorithm is a function of the frequency and severity of the rendering glitches caused by a missed prediction. For example, in a game of PONG, a misprediction will cause the opponent's paddle to jump to a new position on the screen. Getting the paddle position wrong is important, of course, but an algorithm that is always wrong but only off by a few pixels is preferable to one that is only wrong 10% of the time but causes the opponent's paddle to jump by one-quarter of the screen length.

Naturally, the best-possible prediction algorithm is game-specific, and probably player-specific as well. GGPO's built-in algorithm is designed to work well with fighting games and beat-'em-ups, though it should work equally well for shooters, maze-solving games (PAC-MAN, GAUNTLET) and most other arcade games. To date, no developer who has used GGPO has seen the need to implement his own prediction.

The built-in prediction algorithm assumes future inputs will be identical to the inputs most recently received from the remote player. This caps the number of prediction errors to the number of times the input state changes per interval. While simple, this has proved to work quite well in avoiding the most jarring visual effects. For example, the character moving backward or forward will often cause the screen to scroll left or right. Getting that position wrong will result in the entire screen jumping to the left or right during a rollback. By assuming the joystick remains held in one direction, we ensure that the screen scrolls smoothly in the most common case: running to the right. The worst case (rapidly toggling the joystick from left to right) seldom occurs in actual gameplay.

GAUNTLET.

Time-permitting, it may be preferable to use the game-state of the previous 1...N frames to provide a better prediction algorithm than the one built into GGPO. For example, in a game of Pong, it might be better to predict that the remote opponent will always move their paddle in a manner to intercept the ball, regardless of their previous inputs. In practice, however, incorporating the possibility of a misprediction into the design of the game to minimize the effect of a rollback has a better payoff, as it's impossible to completely eliminate them. For example, if you have a screen-scroll effect when a player moves on the screen, consider basing it on the average position of the player over several frames or the position of the player K frames ago (where K >= the expected latency in a game), which will minimize (or completely eliminate) the screen-jumping glitch during a rollback. If you have any effects that start on the first frame of execution (e.g. a dramatic screen blackout starting a Super Combo in a fighting game), consider starting the effect on the Kth frame instead of the 0th frame so that the effect is always based on confirmed inputs instead of predicted inputs. Obviously, there is some give-and-take between the responsiveness of the game when played offline and the smoothness when played online using GGPO, but it is an option for designers.

JOEL POLONEY

{ OPTIMIZE YOUR SERVER BACK-END TO SCALE QUICKLY, CHEAPLY, AND PAINLESSLY }

SCALE YOUR ONLINE GAME

////////////// When building any type of online game, it's common practice for game developers to have a user-facing client as well as a back-end server component. In a lot of cases, these might be two totally different code bases. For example, most Flash and native iOS games have a server piece written in a common scripting language such as Ruby, Python, PHP, or Node.js, which is used mostly to validate game states and prevent players from hacking their games.

This article is going to assume that you're already familiar with current server-side technologies, especially the LAMMP stack (Linux, Apache, MySQL, Memcached, PHP). Instead of spending a lot of money and time building out your server infrastructure, we will discuss a few interesting ways to use your existing architecture to help scale your game quickly, cheaply, and hopefully with fewer headaches along the way. I highly recommend taking a look at Amitt Mahajan's article titled "Make It Fast!" in the August 2010 issue of *Game Developer* magazine for more background information on this topic. >>>

SCALING SERVER PROBLEMS WITH EXISTING TITLES

/// I've worked on several social gaming titles over the past few years, and one of the most common headaches is dealing with the server side of game development. The first downside to creating separate server architecture is that you have to write your game logic twice. There is currently no clean way to run native iOS/Flash code at the server level, so you're stuck rewriting everything in a second language. In addition to doubling your workload for any given feature in your game, you need to make sure that the client code and the server code are in sync, which can be a huge headache because it's often near-impossible to track down when they go out of sync.

"Out of sync" errors are extremely common in social games. These errors occur when the server comes to a different conclusion about a given player's state than the front-end client does. In this case, it is assumed that the server is the authority, and the server tells the client to reload the player state. This unfriendly user experience is a nasty side effect of having two code bases attempting to run the same logic.

The second downside to creating your own back-end is scaling it. When your game reaches several millions of players every day, your back-end systems will need to grow to accommodate their increased load. Scaling these services is not an easy task and can take teams of dozens of engineers working around the clock to keep them up and running. In this day and age, it's commonplace for a game to go viral and have millions of people playing every day within a few short weeks—so you need to prepare for that before it happens.

I've watched a dozen games go through this growth trajectory. In most cases, these games had a proven, scalable server architecture and a large team to help support it—but despite all that, there were still issues scaling any new game. One of the games I worked on added a million players a week for 30 straight weeks. Its peak usage was close to 33 million people playing every day. Obviously, this required a massive amount of servers, along with midnight phone calls to fix them and keep the game afloat.

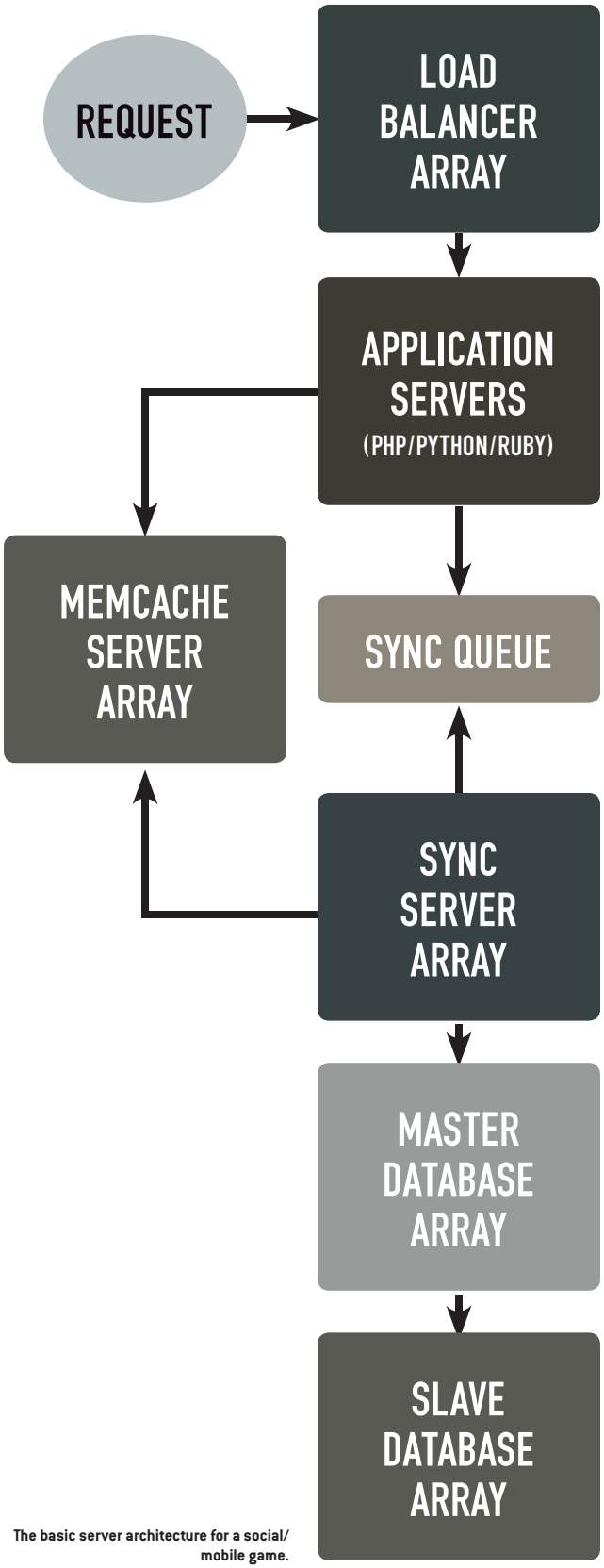
These games validated the player state every five seconds while the game was running; the game would batch up a player's actions and send those requests to our back-end, where they would be routed amongst thousands of servers to validate the game state and send a response back to the client. With over 30 million people playing every day, this was an incredible amount of traffic. We used Amazon's EC2 cloud technology to get new servers quickly and cheaply, but we hit bottlenecks in our architecture on a daily basis, and our server count quickly soared into the thousands.

The functions of these servers ranged from application servers that would actually run the validation logic, to database servers, caching servers, and load-balancing servers. Each group of servers scaled horizontally pretty well, but the sheer size of the server array meant we had to spend lots of time (and money) managing the servers. All told, 1,000 extra-large, high-memory servers on EC2 cost close to $4 million a year. What began as a relatively simple problem (stop players from hacking our game) had grown into a very complicated, costly one.

THINKING OUTSIDE THE BOX

/// I wondered how many players were actively trying to cheat their way through our games compared to number of players who just played them normally. It turned out that the number of players not cheating vastly outnumbered those who are, so why spend all these computing resources checking player state if there's a strong chance they will never actually cheat in the game? This is an optimistic approach to cheating, of course, but it could drastically change what our server architecture looks like. If we don't need to validate player state upon every request, we can significantly streamline our server architecture—especially the application server array, which made up about 75% of the server count.

That doesn't mean we stopped validating player states, of course—



The basic server architecture for a social/mobile game.

APPLICATION CODE:

```
function handle_player_state(header, client_state) {
      // grab and validate our session information
      server_state = Memcache.get(header.user_id);
      if(server_state.session.auth_token != header.auth_token) {
            respond('Not a valid user');
      }

// check to make sure we aren't storing stale data
      if(server_state.last_saved > client_state.last_saved) {
            respond('Received stale data');
      }

      // check the last time we sent data to the server
      if(server_state.session.saves_in_last_minute > BOT_RATE_LIMIT) {
            respond('Rate limited');
      }

      // write data to memcache
      Memcache.save(header.user_id, client_state);

      // add entry to sync queue
      StateQueue.add(USER_STATE_QUEUE, header.user_id);

      respond('Successfully saved');
}
```

VALIDATION CODE:

```
function validate_player_state(user_id) {
      // grab old state and new state
      new_state = Memcache.get(user_id);
      old_state = Database.get(user_id);

      // check to make sure we aren't storing stale data
      if(old_state.last_saved > new_state.last_saved) {
            Memcache.save(user_id, old_state);
            return;
      }

      // prepare list of basic values we want to search
      attributes = ['coins', 'cash', 'xp', 'total_value'];
      averages = Averages.get(attributes);

      flags = old_state.flags;

      for(attribute in attributes) {
            difference = new_state[attribute] Δ old_state[attribute];
if(Math.abs(difference) > averages[attribute]) {
      flags += 1;
}
      }

      // check number of flags and save data
      if(flags > MAX_FLAGS_ALLOWED) {
            old_stage.flags = flags;
Memcache.save(user_id, old_state);
      } else {
            new_state.flags = flags;
            Memcache.save(user_id, new_state);
            Database.save(user_id, new_state);
      }

      return;
}
```

we just didn't need to validate every single action. In this case, we let data from the client flow through to our Memcache array and then validate it in an offline validation process. This setup makes use of the write-back cache technique as well; if you're not familiar, the idea is to write player state to a Memcache array first and then write it to a database when write cycles are free. This removes a tremendous amount of pressure on the database with very minimal risk of losing player state. In our new setup, the validation is moved out of every request and into the syncing process instead. If we are no longer validating state on a per-request basis, we no longer have the need for a complicated and fancy application server setup. Application servers would no longer validate, but simply write the data straight to our Memcache array.

This technique had the added bonus of reducing the size of our server arrays across the board as well, because we didn't have to worry about how often we needed to save player state with the new setup. The reason behind sending requests to our servers every five seconds was to make sure that if the player closed the game at any point, we lost at most five seconds of player state. On a mobile device, you don't have this issue since you typically only have one client. In this case, you could save the state locally on the device and then write it to the servers every few minutes. Flash games are a little different, because you don't have access to store a large amount of data locally. Even if you did, the game could be opened in any browser on any computer, making local storage irrelevant since the changes on one computer couldn't be seen on another. Thus, we had to flush the changes very rapidly.

However, you can use Javascript to detect when a browser has been closed, minimized, or the user has changed tabs, and save your game state to the server before that event finishes. This way, your game only has to make requests to the back-end servers once every few minutes, rather than 10-15 times per minute, which greatly reduces the volume of traffic coming to our servers. Overall, we managed to reduce the load on our load balancing array, Memcache array, and database arrays. It also helped with database lock contention, since we only had to write once every few minutes rather than a dozen times a minute.

What's more, if your clients are only sending data to the server once every few minutes, you can now throttle the number of requests a given player is making. This helps prevent botting and botnet programs from pounding your servers. This type of attack was quite prevalent and was very hard to detect, because valid requests looked identical to invalid ones, and the average player would more often than not look like someone using a botnet to pound our servers. If we're only expecting one hit to our server every minute, then anything significantly over that can easily be picked out.

PUTTING IT ALL TOGETHER

/// The flow for processing and saving user data now looks like this:

•• Player plays your game for some period of time. During this time, you should track the game state solely in the client. All of this data should be zipped up in to a "blob" of data, or a large object that has nested groups of data inside of it.

•• After five minutes have gone by, we zip up the entire state object and send it to our back-end servers.

•• An application server gets the request and writes this zipped-up state object to our Memcache server array without validating anything. It also puts an entry into a queue to validate and sync when we have free write cycles. This entry can be as simple as including the key to the Memcache and a timestamp of when to validate it.

•• A worker server that is processing the queue picks up a user, runs a very quick validity test on the new data, saves it to the database, and removes

it from the queue. If the validity test fails, you simply discard the changes, restore the data from the database, and move on. In this case, the player state would be rolled back to the last known valid state.

• • Rinse and repeat as the player continues to play.

This new approach significantly eases the load on the back-end. Over a five-minute period in the old architecture, we would have 60 requests to our application servers, 60 writes to our Memcache servers, and one sync to our database. In the new setup, we have one request to our application servers, one write to Memcache, and one sync to our database in the same time interval. Current estimates to this new approach are looking at close to a 75-80% savings on server head count, translating to millions of dollars for a large scale game.

BUILDING AN OFFLINE VALIDATION CHECKER

/// If we're going to blindly trust the data we receive from the client, then we will need a quality offline validation checker. But what should it check? We can start with very basic values such as the player's level and experience, the player's currency values, and the player's inventory. We could even look at the value of all of their in-game items combined, or the total net worth of a player. Since the validation checker is running during the sync from Memcache to our database, we can compare values from both the new data we're writing and the old data we're overwriting. In this case, we simply compare the new values for each attribute with our previous values (taking timestamps into account) and see if they make sense. For example, if the player's in-game currency increased from a hundred coins to a million coins in five minutes, that's a red flag that should be more closely examined.

A more complex approach would be to apply statistical analysis to each value against a norm. If your player base is millions of users and only a fraction of them are actually cheating, you can use this to your advantage to figure out who's cheating. For any given level in a game, you can plot out what the values should look like. You will likely end up with a bell curve of values. For example, you would determine at level 10, the average net worth of a player is somewhere between 5,000 and 6,000 coins. At this point, you can place a change in this curve and see where it lies. If it falls several standard deviations outside of the norm, then that should raise a red flag. Don't hesitate to get creative with your validation algorithm. It can vary in complexity and can look at any or all user states.

DEALING WITH CHEATERS

/// When your offline validation algorithm detects strange behavior, there are several things you can do: You can disable accounts until you have a chance to look into them further, you can reset or lock their account entirely, or simply flag the account depending on the magnitude of the infraction. A good approach here would be to have a rating system for flagged users; If an account is flagged once, let it slide, but if it's flagged several times, stop saving the changes to the database. There is also the possibility for a rare false positive where someone who is playing the game normally begins getting flagged for cheating, so you'll need a readily available customer service department to help them get back in to the game.

With this kind of server architecture, you also have the option of taking a hybrid approach to cheating countermeasures. In this case, you would have all the original architecture, but you would start out assuming that all players are not cheating. When your offline validation algorithm flags an account, you flip a switch that would then begin validating their account on every request again. In this case, if a game with a million daily players has a 1% hacking rate, only 10,000 players are checked on every request, which is still a huge difference from the previous million players that we were validating on a per request basis before. This setup has a much stronger level of security and has all the benefits of reducing your server count, but the downside is you have to maintain two code bases.

POTENTIAL CASES WHERE THIS DOESN'T WORK

/// This technique works best for single-player games, such as FarmVille, Words with Friends, The Sims Social, and Angry Birds. In these cases, when a player cheats, they are only cheating for themselves. From what I've seen in the past, most of those who cheat are doing so just to prove a point, rather than to get around having to pay to play the game. Also, since there is no global economy in single-player games, there is no real way for someone to cheat and ruin the game for others.

FarmVille.

Unfortunately, this technique does not work when communicating with other players through trading, auction houses, and so on. When one player's actions affect another significantly, you will want to validate that transaction as it happens. This technique also shouldn't be used when a player is using in-game currency or premium currency to buy goods. This is a rare enough action on its own and should be very secure since it deals with real money. In most social games, only a small percentage of our player base actually spent real money in their games, so this won't be a significant amount of traffic.

FEWER SERVERS, MORE MONEY

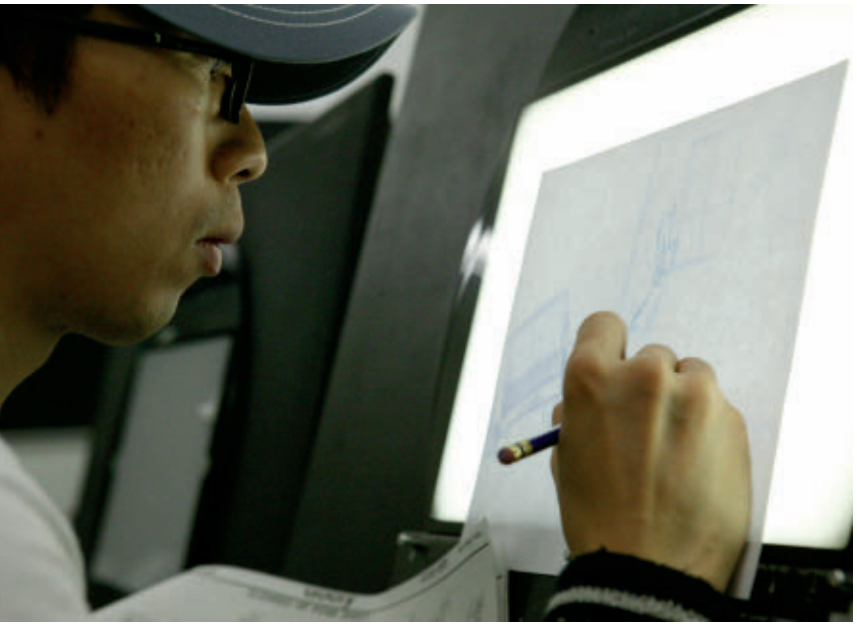
/// Overall, I see three advantages to using this kind of server architecture: You save money on your server bill (which, for a popular game, will most likely outweigh whatever you lose from a few minor cheaters that get away with it), you improve your player experience by making out-of-sync bugs and server outages less frequent, and most importantly, you can focus on building a great game instead of having to worry about how it's going to scale. What's more, it'll be easier for you to build games more quickly because you only have to worry about one code base and you won't need to spend a significant amount of time scaling servers as your game grows.

Scaling the back-end of a mobile or social game is no easy task, and even if you read about a new back-end scaling solution, it's not always the kind of thing you can simply drop in place. This solution uses open source software and existing web technologies, and should be very common to what you've already seen in the past—the only difference is how you're actually using it. The security of this system can be as strong and robust as you like. This architecture can be reused for many titles, with the validation logic swapped in and out depending on which title is using it. In the end, game developers should be focused on building great games, not figuring out how to keep them alive.

JOEL POLONEY *is an entrepreneur who sold his first social gaming startup to Zynga in 2009 where he went on to build FarmVille and lead up central technology. He has since left and is currently working on his second startup, focusing on products that people love to use.*

ACADEMY *of* ART UNIVERSITY®

FOUNDED IN SAN FRANCISCO 1929 BY ARTISTS FOR ARTISTS



TAKE CLASSES ONLINE OR IN SAN FRANCISCO

Acting*
Advertising
Animation & Visual Effects
Architecture
Art Education
Art History
Fashion
Fine Art
Game Design
Graphic Design
Illustration
Industrial Design
Interior Architecture & Design
Landscape Architecture
Motion Pictures & Television
Multimedia Communications
Music Production & Sound Design for Visual Media
Photography
Web Design & New Media

ENROLL NOW

EARN

YOUR AA, BA, BFA, MA, MFA OR M.ARCH ACCREDITED DEGREE

ENGAGE

IN CONTINUING ART EDUCATION COURSES

EXPLORE

PRE-COLLEGE SCHOLARSHIP PROGRAMS

WWW.ACADEMYART.EDU

800.544.2787 (U.S. Only) or 415.274.2200

79 NEW MONTGOMERY ST, SAN FRANCISCO, CA 94105

Accredited member WASC, NASAD, CIDA (BFA-IAD, MFA-IAD), NAAB (M.ARCH)

**Acting degree program is not available online.*

Visit www.academyart.edu to learn about total costs, median student loan debt, potential occupations and other information.

Photo credit: Joseph Taylor, Chris Haejin Chu

KEITH FULLER

LEVEL U STUDIO

"HOW CAN YOUR GAME STUDIO MAKE BETTER GAMES?"

It's a fairly simple question, and the answer—find out what you don't do so well, and fix it—seems equally simple. But when you take a close look inward, you may discover that the strengths and weaknesses your studio had when you started working there haven't drastically changed since. I surveyed a pool of 43 experienced developers across disciplines (see the Methodology sidebar for more details) to determine what we as an industry are good at, what we're bad at, and how important we think it is to shore up our weaknesses. If your studio isn't taking big steps to address its weaknesses, it should be.

IDENTIFYING YOUR STUDIO'S STRENGTH

» When asked to identify their studio's strongest facet (see **Figure 1**), production came out on top (eight respondents), followed by work/life balance and polishing/delivering (six respondents each), and then innovation (five respondents). Interestingly, of the eight people who voted for production, only one of them listed their role as project management. You would expect if a studio's strength was working together as a team that the project managers, who have good oversight of the big picture, would be the ones to comment on it. Also, three people who voted for innovation as their studio's strength occupied studio leadership roles. No other response had such a strong showing from studio leaders, indicating that from the top down more organizations are explicitly targeting new ideas or new fields of endeavor rather than trying to compete in established and often overpopulated genres.

I did a similar survey last year, which you'll see in the charts, but please note that the categories were changed slightly since the prior survey; morale and publishing were added to reflect feedback from the previous survey. The traditional roles of publishing and developing games are, in some cases, combined under one roof in our industry, so I wanted to include both areas in the survey.

In addition to identifying their studio's strength, respondents were also asked to estimate the impact of their studio's strongest point upon overall company success, from 1 (low) to 5 (high). This year's responses (average value of 3.78) are slightly more positive than in the previous year (average value of 3.48). We're going to use this later to compare the impact of a studio's strongest and weakest facets on its success.

IDENTIFYING YOUR STUDIO'S WEAKNESS

» When asked what area of operation people would most like to see improved at their company, the clear winner in 2011 was company leadership, with fairly even votes in the other categories. This year, polishing/delivering took a slight lead over mentoring/training (see **Figure 2**). Intriguingly, the least-critical area of improvement in the 2012 results was last year's most sought-after; possibly because studio leadership has gotten noticeably better in the last year, or possibly because other areas have become more important.

Considering leadership was also ranked third for most-improved area of operation in the past year (see **Figure 3**), I'm inclined to think that overall studio leadership has, in fact, been improving. However, it's worth pointing out that studio leaders are disproportionately represented among the people taking this survey, and said leaders may be more likely to point out the positive aspects of leadership than would the people being led.

HOW DOES YOUR STUDIO'S WEAKNESS AFFECT YOUR SUCCESS?

» You can only improve what you can measure (usually in dollar amounts, when it comes to business), so I asked respondents to quantify the impact of their studio's weakest facet upon their success, from a scale of –5 ("serious negative impact") to 5 ("highest positive impact," meaning even your studio's weaknesses are contributing to its overall success).

Last year the average value of the 24 respondents was –1.71, meaning the area in question was negatively impacting company success. In 2012, the 33 survey participants averaged a –1.79. Taken broadly, this indicates that, year over year, developers believe it's more important for a studio to address its flaws. Essentially, the average assessment made by an industry worker is that the biggest problem area in their organization is having a larger negative impact than it was last year—perhaps upon employee morale, product quality, or other factors—which ultimately translates to less money being made by the worker's studio.

There are a few different reasons why this could be the case; better internal and external

communication could make employees more aware of their studio's weaknesses, or a studio could simply be experiencing overall entropy over time. The ultimate takeaway from this question is that our industry's efforts to continuously improve aren't effective enough. If they were, the answers to this question would be getting increasingly positive, not increasingly negative. From the top down, studios need to examine their improvement efforts and devote more or better resources to this area of operation.

THE CASE FOR FIXING WEAKNESSES

» Now we can take the estimated impact on studio success of both a studio's strongest and weakest facets (3.78 and –1.79, respectively) to determine a studio's potential return on investment for improving its weakest area of operation. When I asked the survey respondents how much fixing the studio's primary weakness would affect their overall success (on a scale from one to five), the average value in 2012 was 3.52 (compared to a 2011 average of 3.5). At this point we have:

a. Impact of studio strength on overall success: 3.78
b. Impact of studio weakness on overall success: –1.79
c. Estimated effect of fixing weakness on overall success: 3.52

If we take the difference between b and c, we end up with 5.31 as the overall impact of fixing a studio's weakness; out of a 5-point scale, that is a 106% perceived impact on success.

Figure 1: What is your studio's strongest point?

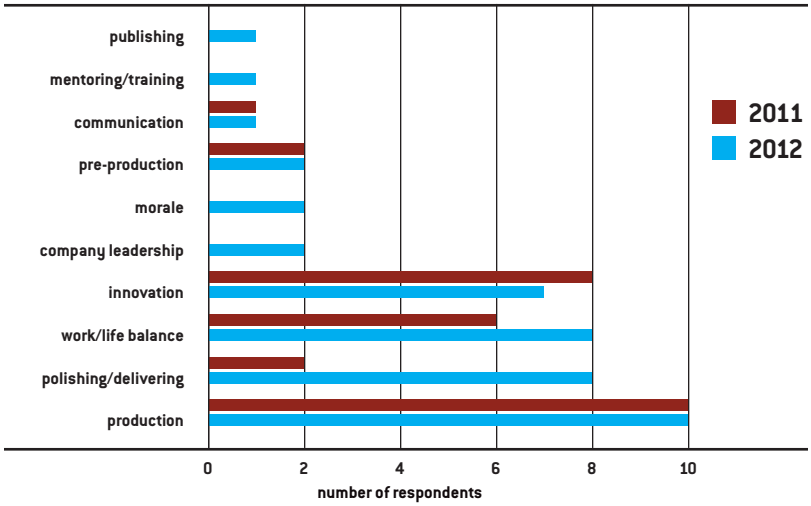


Figure 2: What is your studio's weak point?

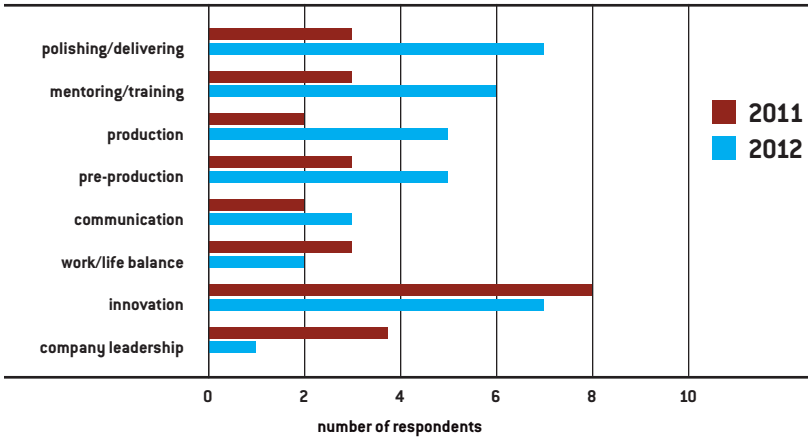


FIGURE 3: What has your studio improved upon most over the last year?

- PRE-PRODUCTION 25.9%
- PRODUCTION 18.5%
- COMPANY LEADERSHIP..... 14.8%
- MENTORING/TRAINING..... 11.1%
- MORALE 11.1%
- POLISHING/DELIVERING 7.4%
- INNOVATION 7.4%
- COMMUNICATION 3.7%
- MORALE 3.7%
- WORK/LIFE BALANCE 3.7%

In other words, the people doing the work and leading the teams at your studio see something that needs to be fixed because it affects your success more profoundly than your greatest strength. And they have reason to believe you could reverse a negative trend by fixing it—so that you are actually not just patching up weaknesses in the leaky boat that represents your studio, but turning those weaknesses into strengths that make your studio more money. And as you'll see in the next section, they even have clear ideas about how to do it.

HOW CAN YOUR STUDIO FIX WHAT'S BROKEN?

» In the latter portion of the survey, I delved deeper into what these folks would suggest doing to bring about the improvements they envision. The questions are more open-ended and don't lend themselves to summary, but the suggestions are wide-ranging and include things like improving communication with QA, professionalizing the management of projects, and listening to the people who make the games and not just the people who sell them. Here are a few of the verbatim responses.

WHAT WOULD YOU FIX?
"Onboarding and training on new tools."
HOW WOULD YOU FIX IT?
"Highly searchable training materials."

WHAT WOULD YOU FIX?
"We have way too many meetings. Sometimes we'll have meetings about previous meetings and post-meeting meetings. Sometimes we'll have weeks where half the team is stuck in back-to-back meetings all day."
HOW WOULD YOU FIX IT?
"There needs to be a limit to the amount of meetings we can have, or a meeting-free day just so people can catch up on work."

WHAT WOULD YOU FIX?
"I'd like to see decisions thought through from the beginning, instead of growing exponentially near the end of the project."
HOW WOULD YOU FIX IT?
" More discussion with different leads at the beginning of the project to determine what is and isn't feasible."

WHAT WOULD YOU FIX?
"I would bring back agile processes."
HOW WOULD YOU FIX IT?
" Hire an agile coach to objectively help us identify and fix the problems we have with agile."

BEHIND THE ANSWERS

» The goal of my annual survey and indeed of this article is to improve our industry—how we do what we do. And that can come from fixing something that exists, or creating something that doesn't. It's not just dependent upon getting the management to listen, or automating your build process, or eschewing Taylorism in favor of Scrum. Sometimes it's about heading off in a new direction, trying a new business model, creating a new company, or having the courage to try just one more time, hoping they've found a studio that remains solvent long enough to ship a product. I plan on running a Third Annual Production Survey in 2013 around GDC and I would greatly appreciate your participation. The more people who participate, the more meaningful the results will be. Hopefully, this survey can play just a tiny bit in helping us get there.

KEITH FULLER *is a production consultant (fullergameproduction.com) with 15 years of game development experience. He's also the author of* Beyond Critical: Improving Leadership in Game Development.

methodology

Around the time of GDC 2011, I invited all manner of production coordinators, producers, and studio leaders to participate in an anonymous 10-question survey about how their companies, projects, and teams operate. This year I repeated the survey, with many of the same questions from 2011, as well as a few new items added in response to comments I received when discussing the previous year's results. These survey results were obtained from a pool of 43 viable respondents, most of whom were either programmers (12 respondents) or project managers (12 respondents). In terms of experience level, only seven respondents described themselves as frontline contributors, compared to 12 senior contributors. All other participants listed their positions as being some form of management or leadership. The demographic responses indicate that programming, project management, and design describe almost 75% of the participants' roles, and that much more than half of all respondents are relatively experienced individuals.

Note that while the survey asks questions about demographic information (the respondent's development discipline and seniority level, as well as their studio's targeted platforms), I did not cross-tabulate any of the production survey questions with the reported demographic information in order to preserve anonymity. From my own experience as a studio developer, I understand that many people fear posting their opinions online because their statements might somehow get linked back to them and result in retribution by an employer. Also, in order to diversify and expand the respondent pool, I made sure to post the 2012 survey invitation in the most discipline-agnostic way possible.


PIXEL

ROWAN PARKER

JUNK 4AM

// At the end of 2010, right after finishing PIXELJUNK SHOOTER 2, Q-Games president and founder Dylan Cuthbert pulls me aside for a chat.

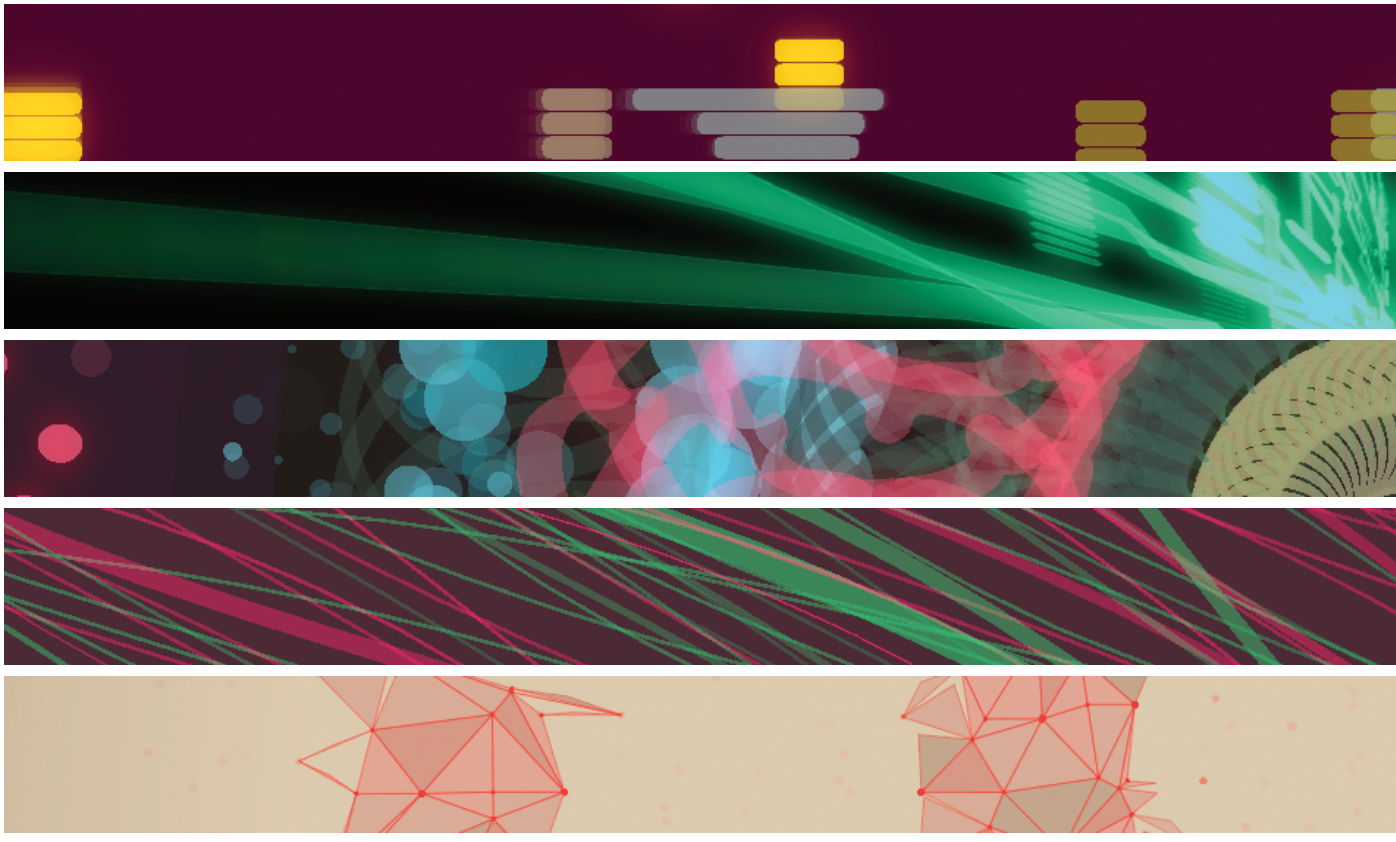
"So we've kind of got this music visualizer using the PlayStation Move called LIFELIKE on the back burner," he says. "You should make it happen."

I asked for some more details about the project. "Well, there's music," Dylan said, "And there's a PlayStation Move. Off you go."

PIXELJUNK 4AM released in spring 2012 on PSN. It's not so much a game in the strictest sense of the word—it's a Move-exclusive audiovisual composer, where all your performances are broadcast live around the world on PSN. The player creates music using the Virtual Audio Canvas, which is an actual physical 3D space carved out in front of the player. It contains more than 190 sound samples, a wide variety of DSP effects, and the ability to dub loops into your own unique groove. We also released a free Live Viewer, allowing anyone on PSN to stream performances live and give real-time feedback.

It's easy for a game designer to say, "Hmm, this boss is still a little easy. It should breathe more magma!" Knowing when a 4AM event is finished, on the other hand, starts venturing into the realm of music production—and there were no other similar games to use as points of reference. The control scheme is completely unique to 4AM, and the experimental social gameplay we included also seemed pretty far-fetched at the time when we were developing.

Normally, those are the points at which someone high up usually says, "Stop smoking so much and make something solid!" Yet throughout development, Dylan supported every new crazy idea by saying, "If it's fun, put it in!" Our U.S. publisher Sony Santa Monica was also super supportive despite the amount of wild new design that 4AM was pushing. The faith (and massive balls of steel!) of these fine people is what ultimately allowed us to release a unique experience that will hopefully be remembered fondly for a long time by its players.



WHAT WENT RIGHT

1. FEARLESS HARDWARE EXPERIMENTATION

/// Going into the project, I didn't exactly have a glowing perception of the Move—it had always felt gimmicky to me. On the first day, though, we played around with some of the SDK samples and were surprised to discover that the Move actually seemed pretty robust and ripe for creating some wild stuff with tricks that neither the Kinect nor the Wii could do. (It occurred to me at the time that it might not be the Move's fault that it wasn't being used for crazy new stuff, but that people weren't making games for it to do crazy new stuff; since we started on 4AM, we've seen some other notable Move experiments, which warms my heart greatly!).

With a skeleton team of two programmers and one designer, we prototyped at least 12 different control methods for 4AM, all utilizing the Move to control music in space. Some of the control schemes varied from casting musical "spells" in the air to replicating an eight-way arcade stick and inputting STREET FIGHTER commands in the air. Regardless of how ludicrous each idea seemed, though, the only test that mattered was whether we could ask, "Can I make music, and is it fun?" and honestly answer yes. Dylan supported our willingness to experiment without regard for whether something felt "normal" or "gimmicky," all the way up until we found 4AM's distinctive Virtual Audio Canvas. Without that support to continue experimenting, we might never have pushed the Move enough to find the Virtual Audio Canvas we have now. It would have been a hell of a lot easier to just implement a menu and pointers, but it wouldn't have been 4AM!

2. DISCARDING OLD TOOLS

/// When I picked up the LIFELIKE project in January, there were already legacy tools and an editor. Unfortunately, this editor was largely unusable (I'll explain why later) and we needed to make a quick decision regarding how to move forward. It was primarily developed and tested on a PC, and the PS3 version wasn't optimized. GameMonkey (our scripting language) was running computationally intensive generic behaviors with a vast array of tweakable properties. The framework gave us flexibility, but came at the cost of unoptimized generic code that was incurring a lot of processing time. Even though the artists and designers were only able to build rather generic-looking visualizers, our frame rate was still suffering.

Shortly after the controls prototype was finished, we scrapped all the generic behaviors and started rewriting the rendering code with specific visuals in mind. It gave us the freedom to optimize each visualizer independently (scalpel over sledgehammer) and expose only the most interesting elements to the artists. Ditching

support for the PC build also let us offload a lot onto the PS3's SPUs, and freed our lone visualizer programmer from the burden of maintaining both versions of the editor. This boosted our frame rate and made the artists feel more empowered to make cool stuff. We also sat the visualizer programmer right by the relevant artists, in order to quickly iterate the visualizer as they figured out what the artists needed to achieve the look they wanted. This paired-nucleus approach was definitely smoother than a programmer trying to write blanket variables to achieve an invisible goal.

3. RENAMING FROM LIFELIKE TO 4AM

/// Around July 2011, we realized we had a completely different beast than we'd started with. There was an internal and external need to redefine the project and leave behind any associations to the old one. We couldn't have known how much of an effect renaming to 4AM would have, though. In the initial meetings a lot of names were thrown around that came very close (PIXELJUNK: ACID, PIXELJUNK: GROOVE) but once it was said, 4AM caught on almost instantly. It encapsulated the identity that we in the office had all come to attribute to the game, and it was the only one we could agree on that would convey that identity to people outside as well. 4AM best captured the atmosphere of a club in the early hours of the morning. The name change didn't just throw off the shackles of old design ideas and tech for us; it gave everyone a common word to congregate around. It's also just plain fun to say.

4. A RADIO STATION ON PSN

/// 4AM broadcasts all performances live around the world using the PlayStation Network. There is also a Free Viewer available for download, which allows anyone to stream current live performances even if they don't own the game itself. It is essentially a free Visualizer-powered radio station available to anyone with a PS3, where the performers create the content. Needless to say, this was some uncharted territory both for us and for Sony. Sony's Title User Storage (TUS) service came to the rescue, helping us avoid P2P networking by storing a buffer of each broadcast per user on their servers. Unlike regular streaming services (UStream, for example), we wanted 4AM to be live without interruption, so we couldn't allow for any buffer latency to occur. This meant that the data sent across the network needed to be heavily packed.

4AM currently creates an initial one-time-only buffer on load, so if the network connection drops packets during a performer's playback, the buffer shrinks permanently. Fortunately, we managed to keep the network data packet size to a minimum, which meant 4AM has an almost-negligible rate of packet loss. The Free Viewer contains all the same assets as the full version, so we only have to send the Move's motion data (gyro and accelerometer) to effectively call all the same functions on the local client, meaning that viewers'

game data

perform create share live

DEVELOPER: Q-Games
PUBLISHER: Sony Santa Monica
RELEASE DATE: May 15, 2012
PLATFORMS: PSN
INITIAL NUMBER OF DEVELOPERS: 3
FINAL NUMBER OF DEVELOPERS: 6 + music by DJ Baiyon
LENGTH OF DEVELOPMENT: 16 months
DEVELOPMENT TOOLS: C++, GameMonkey
HALLUCINOGENS CONSUMED: Minimal.

PS3s are actually replaying the performer's Move motion data in order to replicate the performance, not simply streaming a video. We wanted 4AM to be viewable to all PS3 owners, with thousands of people concurrently streaming live performances, and a P2P network setup simply would not have been able to handle that kind of load. It's also nice to know that our unorthodox use of the TUS didn't overload Sony's servers.

5. SUPPORT YOUR STARS

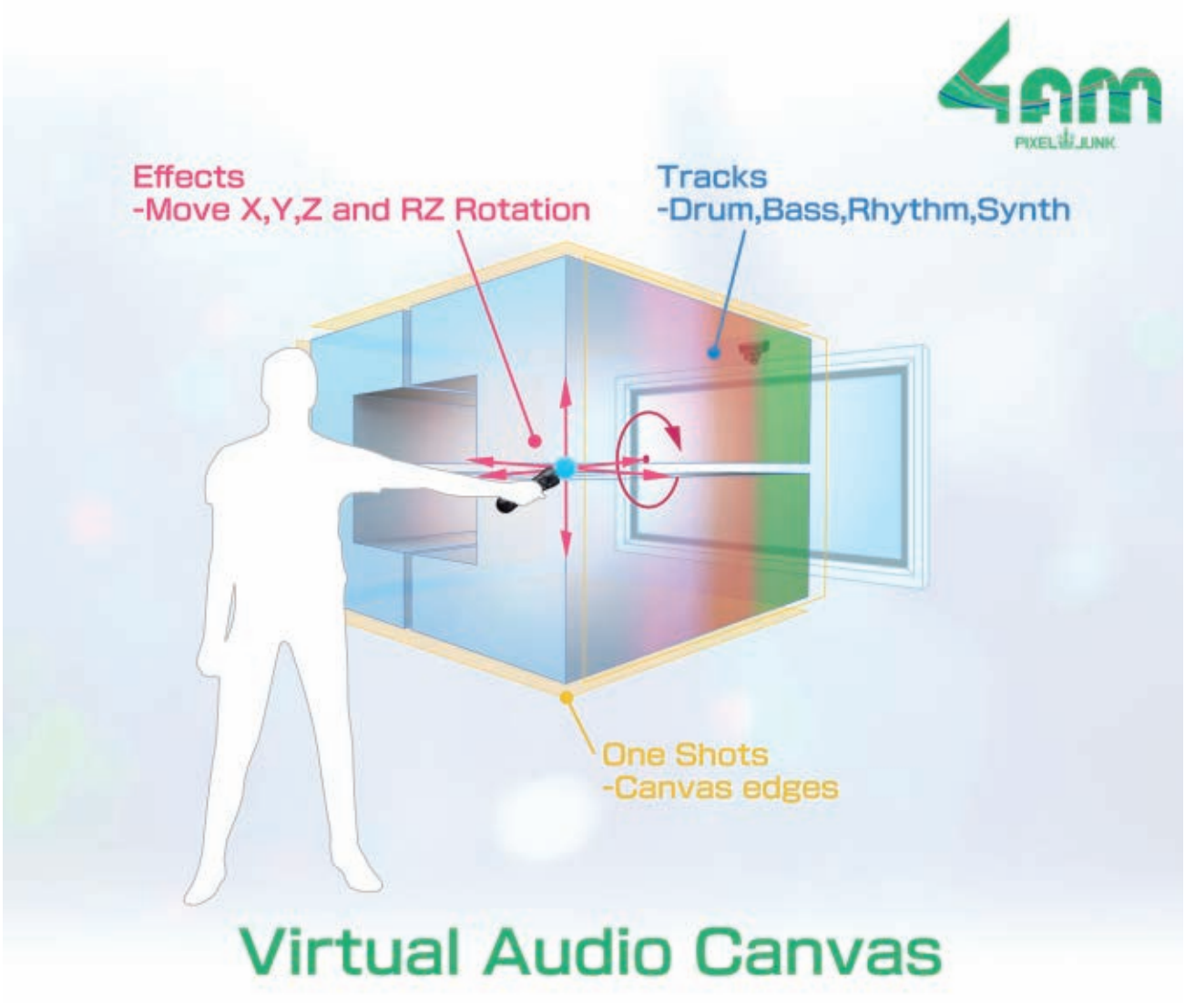
/// PIXELJUNK 4AM was designed to encourage "performance" in every sense of the word, but we only realized what we'd really made once the live beta started. The live beta was an online stage where players could perform. As people started playing with the live beta, they began to post videos and personal streams of themselves performing in 4AM to build their own followings.

We were humbled to see that players were already enjoying themselves with just the featureless beta, and once we released it, the social effect was compounded. One feature shining through is the auto-shuffle attract mode: While on the main menu, 4AM will seek current live streams in the background and put up a random person playing live as a menu background. This attract mode also adds to the head count for that performer's crowd. A stable of other features (Facebook, Twitter, hometown, local time, avatar display) were also designed to let players stamp their own identity on an event; identity really lets performers make the 4AM events their own. Since we've launched and gone live, some performers are amassing crowds with thousands of people in the audience. Most people will go their whole lives and never be able to play in front of a live audience of that capacity. I'm thrilled people are experiencing that.

WHAT WENT WRONG

1. LACK OF EARLY DIRECTION

/// In 2010, when the project was still called LIFELIKE, it was crippled by a confused artistic direction as to where it should go. If we had a strong artistic vision early on, we could have avoided building features we would end up not using, and we would have built tools that let us get the visual effect we wanted. Since we didn't have that artistic vision, we had to build very



generalized tools that produced a rather generic visual style.

In an attempt to make the editor more artist-friendly, we used GameMonkey to expose a lot of in-game models' behaviors. Unfortunately, we exposed too many parameters and variables, which took down the frame rate. A great example was the jump behavior designed to make small circles jump in time with the music. Over time, the jump behavior became bloated with a variety of other requests like interpolating color based on height, rotating on a pivot point, and other small artistic touches. If it so happened that none of them were being used, the cost of one jumping circle could be astronomical for the visual payoff. This was the price though of wanting flexibility in the visuals but never nailing down any one element.

Many of these behaviors were things that could have been (and eventually were) written far more efficiently and optimized for the SPU once we knew exactly what we wanted to make. As a result, early incarnations of the editor on PC were a monstrosity, making it almost impossible to use without an intricate knowledge of each behavior's roots in code. (The original programmer moved on shortly after the prototype, so there are probably still things lurking in there that no one will ever know about now.) The GameMonkey behavior method of using the editor was largely tossed out following the prototype, essentially binning a vast amount of resources and time.

2. TEXT TUTORIAL

/// We knew that 4AM had so many new gameplay concepts and controls that it was going to be hard to explain how to play it. At game shows and events, we found people picked it up extremely easily, given the tactile nature of the Virtual Audio Canvas (vibration feedback). Explaining this idea with just text and images, though, was a completely different story. Even after repeatedly refining the current tutorial and running "blind tests" on new people with every iteration, we still haven't really achieved a 100% understanding rate for the controls after one tutorial play-though. 4AM introduces many new control concepts with no easily available mental reference points—there's nothing like it we can compare it to—so it's kind of like asking players to read a book and then play a saxophone.

The player can really only learn how to control 4AM through play and experimentation, not a tutorial—just like a regular musical instrument. However, since we call it a "game tutorial," people expect to understand everything about the game after they're done with it. While it's nice to see many players just "getting in there" and quickly picking up the rest of the controls, it would still be nicer to have a gentler tutorial experience that gives people time to acclimatize first.

3. ARTIST VS. PLAYER

/// 4AM began as a passive music visualizer. When it started coming together as more of a music creation tool, a fundamental conflict arose: Players want to manipulate the music, but the artists want to preserve their original idealized state of a track. When we began experimenting with DSP effects, we quickly found that people like to warp music and make it their own. The dichotomy within 4AM is that we want players to find their own sound, where traditionally produced music aims to have already achieved that for you by the time you hear it. So when making tracks for 4AM, should they be prepped for "finishing" inside 4AM by the players using DSP effects, or should they be ready to go right out of the box?

This internal tension had a direct impact on the ranges available to players within the DSP effects. A flanger or chorus DSP might have a wide frequency or feedback range to play within, but by carefully setting a min/max range under the hood, we could safely trim the performer's freedom space to something we knew would produce the kind of sound we wanted. Unfortunately, some of these ranges are probably still too subtle for the average user to enjoy, since they can be difficult to hear depending on what track is playing. We had to create effects that were audible and interesting to players, but at the same time stylistically acceptable to the artist. In hindsight, these are two completely opposed ideas, so it was always going to be difficult to satisfy both. The player still needs to have fun, but not at the cost of being a passive appreciation tool for the artist's benefit.

4. COMMUNICATING 4AM TO THE MEDIA

/// 4AM is not a traditional "game." We probably took too long to realize ourselves that it is more of an instrument than a game, and so we slipped into our regular PR habits. The mistake we made was expecting the same response as usual from the traditional game media outlets: Many review sites and journalists clearly didn't know how to approach the title, and either chose to not review because it didn't fall within their definition

of a game, or reached in with a lukewarm reviewing hand without really wanting to shake the boat.

This was in stark contrast to the sharp love/hate response we've seen in comments and forums from people. It's validating to know we made something that generates a passionate reaction from traditional gamers and challenges their definition of a game. Those players who do love 4AM have continued to perpetuate it through word of mouth and online user reviews, which has been a great boon. However, we definitely needed to act earlier and reach out in more lateral PR directions to audio- and music-production-specific sources. We've also been showing 4AM at a variety of music festivals this year, such as Coachella, Electric Daisy Carnival, and very soon at Lollapalooza, giving festival-goers a cool chill-out option.

5. MANAGING BETA EXPECTATIONS

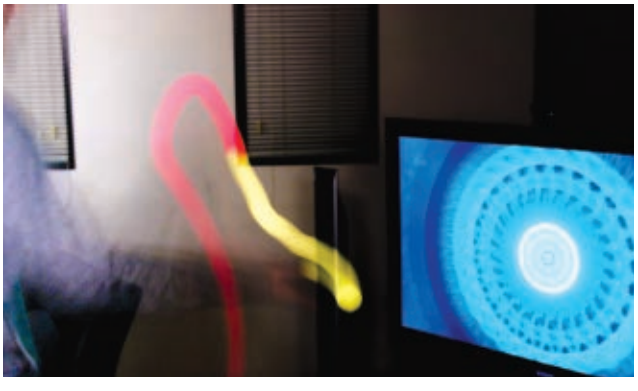
/// There was a degree of miscommunication surrounding the beta for 4AM. We were the first beta to have two packages (a Free Viewer and a Full Version), which made it tricky before we'd even started. Of course, both were critical for testing purposes, but the main problem was people mistaking the Free Viewer as a demo and expecting gameplay. In the context of the full title, the Free Viewer is great and has had a huge uptake, but in a climate where betas are free anyway, that value was lost and people expected playable content.

From a developer standpoint, we wanted to use the beta to test the PSN broadcasting system in a semicontrolled environment and get some player feedback to make some last-minute changes. Prior to the beta, the only broadcasting tests we'd been able to perform were between internal QA with Sony, so we didn't know how 4AM and PSN would handle having thousands of people download the Free Viewer and watch.

The great uptake meant we fulfilled these two goals, but I feel that some people may have been burned on their first impression through the beta. The official beta announcement was a combined effort with our publisher Sony and our own official channels, but ultimately we should have taken more ownership of the message and ensured that word got out about how the beta was going to work (by baking an informative screen right into the package, for example). Unfortunately, with the small team size and time restraints, we really couldn't give the beta the time it deserved. Overall, the beta was successful in satisfying our development requirements, but from a marketing perspective, I feel there may be a few players out there who might never give 4AM a second chance.

LATE INTO THE MORNING

/// If someone had asked me to sit down and seriously design a new musical instrument in January 2011, I would have been enthusiastic. I also would have had no idea where to start. (Now, I would probably clutch my knees and rock back and forth underneath my desk.) Yet at the end of our musical-space journey, we now have a new electronic instrument and performance-art platform that people are enjoying online. I feel amazingly fortunate to have landed in something that was backed by so much faith, and I am grateful that we were given the time we needed to make 4AM and not some corner-cutting "missed opportunity." Since then, we've been working in secret fervor on a new PixelJunk IP that will see a return to more traditional gaming roots (but still has that unique PixelJunk twist).



ROWAN PARKER *is the lead designer on PIXELJUNK 4AM. He resides in Japan and is working at Q-Games on future PixelJunk titles. He would like the world to know that waffles are delicious.*

The Cintiq 24HD has a 24.1-inch widescreen 1920 x 1200 display with a 5080 lpi tablet resolution. Compared to the previous Cintiq tablet, the high-definition display is brighter, has more contrast, and covers more of the Adobe RGB color gamut.

WACOM Cintiq 24HD Tablet

BY MIKE DE LA FLOR

//////// Wacom digitizing tablets are ubiquitous in professional computer graphics. Their lower-end Intuos tablets are popular and relatively affordable, but professionals will typically opt for a high-end Cintiq tablet. Unlike the lower-end Intuos tablets, a Cintiq tablet is essentially an upright display you can draw on. This eliminates the dissonance of drawing horizontally on a desktop tablet while viewing an upright display, offering a more natural analogue to traditional drawing. They're pricey, though; the 24HD is the largest Cintiq model available and costs $2,599. If your artistic workflow depends heavily on the quality and versatility of your hardware though, it's worth it.

» Out of the box, the most obvious difference between the 24HD and other Cintiqs is its size. At nearly 64 pounds and 30.29 inches wide x 18.26 inches high, the 24HD is significantly larger than its siblings. The IPS widescreen display is 24.1 inches, with a 1920 x 1200 pixel native screen resolution and 5080 lpi tablet resolution. The 24HD display is brighter, has more contrast, and covers more of the Adobe RGB color gamut than previous Cintiqs or the 21UX.

Another important difference between the 24HD and the 21UX is the stand design. The 24HD features a hinged stand that allows the tablet to operate like a drafting table, which lets you be a little bit more flexible in your workspace. Unlock the hinged arms at the base, and the arms can rotate 75 degrees, making it possible to position the display upright (not possible with the 21UX), flat on the desktop, or beyond the desk edge to rest on the artist's lap. Also, the display itself can pivot 163 degrees along the arm hinges, so between the upper and lower hinges the 24HD is relatively easy to fit into whatever working arrangement you prefer.

21UX users may be surprised to find that the 24HD does not have a rotating display like the 21UX. While this may take some getting used to, the lack of a rotating pivot makes sense because the 24HD is more like a workbench than a sketch pad. While it would have been nice for Wacom to somehow engineer the 24HD display to rotate, its sheer size may have limited this possibility. In practice, once you're accustomed to the 24HD, you won't miss the rotating pivot, especially if you're working with programs that feature rotate view tools (such as Autodesk Sketchbook Pro and Adobe Photoshop CS 4 and higher, for example). On the other

WACOM Cintiq 24HD
www.wacom.com

PRICE
› $2,599

SYSTEM REQUIREMENTS
› Windows 7, Vista or XP (SP3, 32- or 64-bit versions), Mac OS 10.5.8 (or later)

PROS
1. Larger drawing and work area
2. Full HD resolution
3. Innovative stand

CONS
1. No display rotation
2. Surface feels more slippery
3. Heavy

hand, Adobe Illustrator and other important graphics programs don't have a rotate view tool, so you might need to find a workaround.

Of course, when you're working on your tablet, you probably don't want to reach over for the keyboard very often. The Cintiq 24HD has two customizable Touch Rings, each with three presets and 10 Express keys (six fewer Express keys than the current 21UX). The Touch Rings and Express keys reduce the need to reach for the keyboard. However, if you absolutely have to use your keyboard, you can slip it underneath the display, which rests on small, retractable feet that leave about 2 inches of height clearance for a keyboard. The base part of the stand is not just support for the display—it also functions as a balancing counterweight that firmly grounds the tablet while you are positioning it. Note that the stand footprint takes up considerable desktop real estate at 25 inches wide by 15 inches deep. Your experience may vary, but for me, it didn't take long to adjust from the 21UX to the 24HD.

The digitizing technology behind the 24HD is basically the same as that in Intuos 4 tablets. The pen is the standard pressure-sensitive, two-button, cordless, battery-free stylus with an eraser, and it supports 2,048 levels of pressure sensitivity and a 60-degree tilt range. As expected, the pen is lightweight, balanced, and the rubber (latex-free) grip makes it easy to hold. If you're making the jump from an older Cintiq model, you'll immediately notice that the pen is more sensitive. However, you might notice a short lag between when the pen moves and when the stroke appears on screen that you didn't notice before—usually if you're working with very large files or complicated brushes that are taxing your computer's resources. Overall, however, the increased pressure sensitivity lets you draw with more nuance and subtlety.

In the past, Cintiq tablets have had issues with calibrating for the viewing angle and parallax. When you move the display, you change the viewing angle, which causes the tip of the pen to appear to not line up with the cursor. This is compounded by the parallax effect created by the gap between the glass on top of the IPS display, which makes the pen look even more off-kilter. To calibrate the tablet, you have to position the display as desired and then press the handy Wacom control panel button on the bezel and work through the short calibration process that takes a few minutes. Most artists have favorite positions in which they use their tablets, so you probably won't have to do this often, but if you are a bit more nomadic in your artistic endeavors, you'll have to repeat this process each time you move.

One of the natural aspects of drawing on paper or painting on canvas is the resistance provided by the texture of the surface (usually referred to as "tooth"). The 24HD and other Cintiqs feature an antiglare coating on the glass that also provides a subtle tooth to the drawing surface. Compared to older Cintiq or Intuos tablets, the 24HD drawing surface feels a bit more slippery, which might take a little getting used to. Also, even though current Cintiq tablets ship with glass screens instead of the older plastic screens, you still need to keep the drawing surface free of dirt, dust, and skin oils, because the mix may result in excessive wear or permanent scratching.

The Cintiq 24HD features an extensive array of programmable Express keys and Touch strips that reduce the need to reach for the keyboard. The bezel also houses buttons that display an onscreen keyboard and can launch the Wacom control panel.

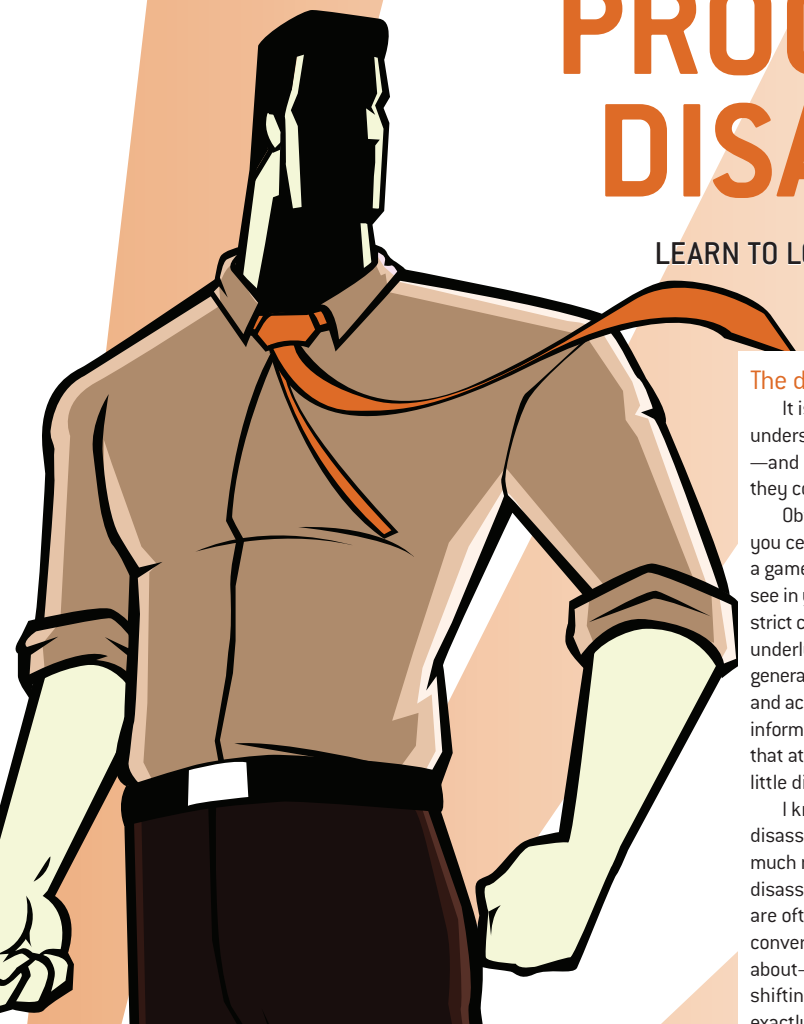
There's no doubt about it: The $2,599 price tag is a pretty big deal, and whether you're upgrading from another Cintiq model or buying a Cintiq for the first time, it might not be worthwhile. CG artists may do well sticking with the smaller 21UX, especially if your workflow relies on the rotating display. However, you might find that the 24HD's stand opens up drawing positions you can't get with the 21UX, and the 24HD's larger size and side bezels give you more drawing space and plenty of elbow room. The 24HD is a drafting table or a workbench, not a sketchpad. Also, the 24HD display supports a full high-definition resolution, includes more of Adobe's RGB color gamut, all at a standard 16:10 aspect ratio (which is more useful for games than the 21UX's 4:3 aspect ratio). For professional CG artists whose productivity depends in large part on hardware, upgrading to the 24HD is a no-brainer.

MIKE DE LA FLOR *is a freelance medical illustrator, animator, instructor and writer. He's the author of The Digital Biomedical Illustration Handbook and other CG books.*

One of the main differences between other Cintiqs and the 24HD is its redesigned stand. The new stand allows the 24HD to be positioned upright, flat, angled, or just rest on the artist's lap.

PROGRAMMERS DISASSEMBLE!

LEARN TO LOVE YOUR DEBUGGER'S DISASSEMBLY WINDOW



The disassembly window in your debugger is your friend.

It is my honest opinion that all programmers should be made to understand and embrace the disassembly, registers, and memory windows; —and by doing so to take a big step toward becoming the programmers they could and *should* be.

Obviously it's not quite that simple, but it's hardly rocket science—and you certainly don't have to become an assembly programmer to do it. As a game programmer, the vast majority of the assembly code that you will see in your disassembly window was generated by a compiler, which follows strict conventions when generating code. Once you understand the principles underlying these conventions, it's not so hard to make sense of compiler-generated assembly on any platform by applying that knowledge (given time and access to the internet or relevant hardware manuals, anyway). Apply this information to the data you get from your debugger, and over time you will find that at the disassembly level, debugging a release build is actually not very little different than from debugging a debug build.

I know this is true, because I once feared what lay behind the veil of disassembly, and having pushed my way through it, I've found that it's much more pleasant and productive on the other side. Understanding disassembly makes it easier to debug release builds and spot issues that are often more or less invisible in high-level code (such as float to double conversions). It also makes it easier to figure out what you *shouldn't* worry about—for example, the kind of traditional by-hand optimizations, such as shifting and adding to try and to optimize multiplication by constants, are exactly the kind of optimization that most compilers will spot and do for you.

Over time, looking at disassembly in the debugger will enable you to gain a solid understanding of how your compiler generates assembly code from C/C++ code, and how the generated assembly code implements behavior that is required by the C/C++ code. You'll finally understand that all the little rules about What Not To Do in C/C++ (e.g., "don't return pointers to local variables," or "don't pass large structures by value") are simply implications of the underlying mechanics of the assembly level implementation of the system—self-evident truths rather than rote-learned rules.

Also, many of the most subtle and insidious bugs (other than those associated with multi-threading) are often only apparent at the level of the disassembly, and are essentially invisible when debugging at the source level.

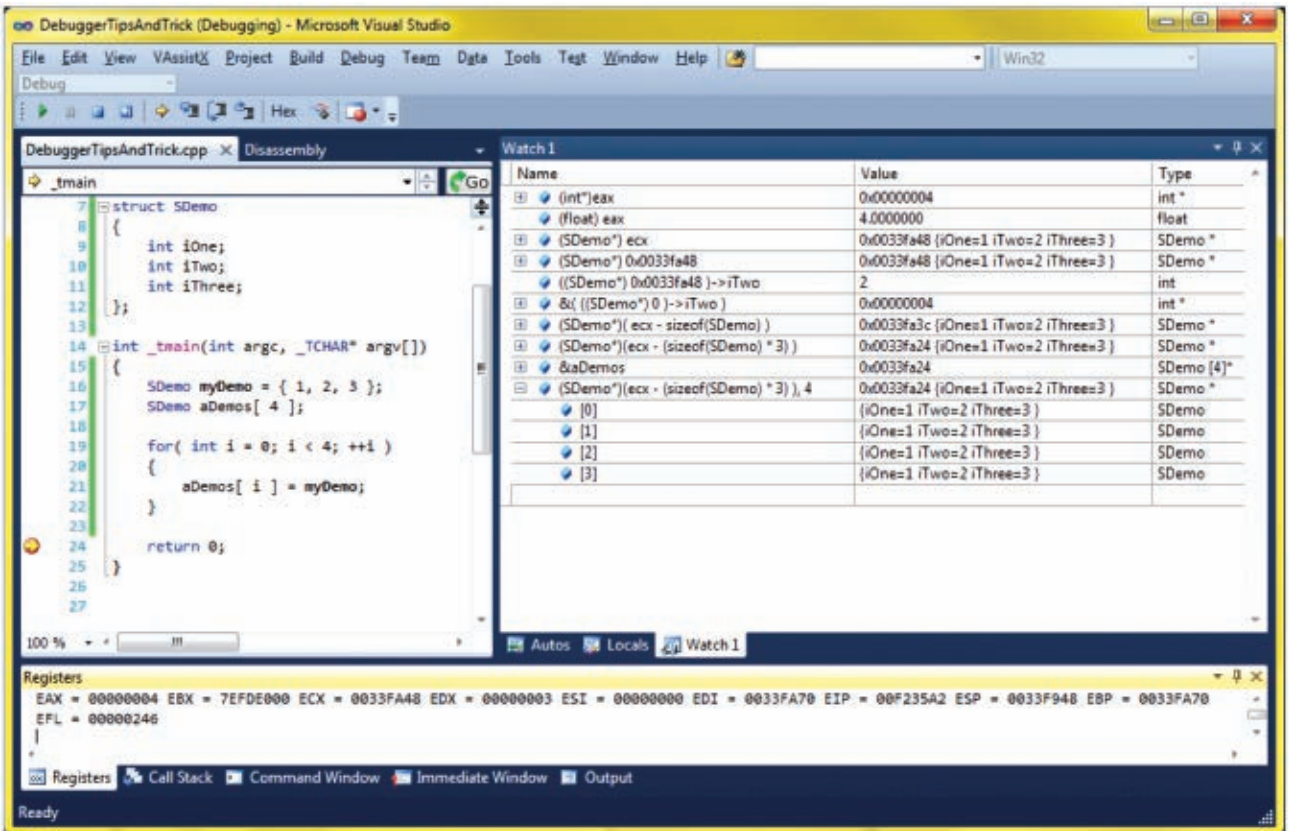


FIGURE 1: Visual Studio 2010 shows the debugger windows that I suggest you have open if you decide to step through this simple snippet of disassembly. The right-click context menu is shown so you can see what display options I have checked.

For example; , it is entirely possible for code from different source files (or more commonly, different libraries) to end up with differing ideas about the size and/or content of a struct or class but to still execute without crashing for the vast majority of the time when accessing instances of the problem type. The same is equally true for data accessed via incorrectly- cast pointers or (heaven forbid) function calls made to incorrectly- cast function pointers.

The final benefit I'm going to mention is only really available to those who devote serious time and effort to understanding disassembly: the crash dump. Most companies have their "crash dump guy" (or gal), the go-to person who can work their way through a crash dump and come out on the other side with the cause, a suggested fix, and the a general smell of heroism about them. None of these people were born able to make sense of crash dumps; they all got there by gaining the knowledge they needed, the determination to apply it, and the raw mechanical practice that made them experts.

I'm not a crash dump guy, but I get by. Given a crash dump, a map file for the build, access to the relevant tools and documentation for the platform I'm working with, and an appropriate amount of time, I can usually find my way to the root of the problem.

SOME ASSEMBLY REQUIRED

» In order to make sense of the workings of the assembly level of C/C++, you'll need a little basic information. First, you need to get your head around the difference in "worldview" between C/C++ code and assembly code. The worldview of assembly code is intimately tied to the CPU and, more or less, an order of magnitude more granular than that of high high-level code. Second, you need to understand the core mechanism employed to implement the "engine" of the C/C++ language—the Stack. While the principles underlying the operation of the Stack are more or less identical across platforms, the specific implementation of the Stack on any given platform is partially defined by that platform's Application Binary Interface (or *ABI*), which is itself determined by the capabilities of that platform's CPU.

THE WORLDVIEW OF ASSEMBLY CODE

» While the overall behavior of compiler-generated assembly code is logically isomorphic with that requested by the C/C++ code (it will produce the same output for a given input), the specifics of how that behavior is achieved are often surprising when you first observe them. I mentioned that the worldview of assembly is an order of magnitude more granular than that of high high-level code; the best way to demonstrate this is to write some code, compile it in a debug build, put a breakpoint on it, and then step through it in a disassembly window using a "mixed" view (i.e., one that shows the source code inline with the disassembly it caused the compiler to generate). Let's start with this simple example:

```
int x = 1;
int y = 2;
int z = x + y;
```

Since this magazine doesn't have a native compiler and IDE, I'll take you through this very simple illustrative snippet of code myself. The disassembly view I'm showing in **Figure 1** shows x86 assembly, and was generated from an executable compiled using the debug configuration of a vanilla win32 console application created using the Microsoft Visual Studio 2010 "New Project" wizard.

So what does it all mean? Let's take a closer look at it in **Figure 2**.

```
      8:     int x = 1;
00DC139E  mov         dword ptr [ebp-8],1
      9:     int y = 2;
00DC13A5  mov         dword ptr [ebp-14h],2
     10:     int z = y + x;
00DC13AC  mov         eax,dword ptr [ebp-14h]
00DC13AF  add         eax,dword ptr [ebp-8]
00DC13B2  mov         dword ptr [ebp-20h],eax
```

FIGURE 2: **Reading some basic disassembly.**

The lines of black text with number prefixes are the original lines of C++ code, and the lines of gray text below each are the corresponding assembly instructions.

The hexadecimal (*hex*) number at the start of each line of assembly (red box) is the memory address of that line of assembly—remember that machine code is just a stream of data in memory that tells the CPU what to do, so logically each instruction must start at a specific address in memory. Next on each line are the assembly code mnemonics (green box). Each mnemonic represents a CPU instruction, and is followed by its arguments, or "operands" (blue box). Only the mnemonics **mov** and **add** are used here because the code is very simple, but there are many more.

Let's take a closer look at the operands. The identifiers **eax** and **ebp** refer to two of the registers of the x86 CPU architecture. Registers are "working area" for CPUs; fragments of memory that are built into the CPU itself, which the CPU can access instantaneously (within one CPU cycle). Rather than having addresses like memory, registers are typically named in assembly code because there are usually only a (relatively) small number of them.

The **eax** register is a "general purpose" x86 register, primarily used for mathematical operations.

The **ebp** register is the "base pointer" register. In x86 assembly, local variables will typically be accessed via an offset from this register; we will cover why this is the case in a second example later.

This just leaves **dword ptr [...]** to explain. Operand values that are intended to be treated as memory addresses rather than values are put in square brackets. The **dword ptr** part is an x86 assembly code size *directive* specifying the size of the value at the address—**dword** is a contraction of "double word" or a 32-bit value (in x86 assembly code a **word** is a 16-bit value).

Now that we know how to make sense of the disassembly we can see here, let's manually step through it.

```
      8:     int x = 1;
00DC139E  mov         dword ptr [ebp-8],1
      9:     int y = 2;
00DC13A5  mov         dword ptr [ebp-14h],2
```

These first two lines of assembly are very simple, and are clearly performing the same instruction with different operands. The first is **mov**ing the 32-bit integer constant value 1 into the 4 bytes (32 bits) starting at memory address **ebp-8**. Since we know that this code corresponds to the line of C++ code directly above it, we can infer that the integer value we refer to at the high level as **x** is stored at **ebp-8**.

Using the same logic we can see that second line of assembly is **mov**ing the 32-bit integer constant 2 into the 4 bytes starting at memory address **ebp-14h**, which is where the value of the high high-level variable referred to as y is stored. Note that the trailing "h" on the offset from **ebp** indicates a hex value (**ebp-8** doesn't require a trailing "h" because 8 is the same value in both decimal and hex).

```
     10:     int z = y + x;
00DC13AC  mov         eax,dword ptr [ebp-14h]
00DC13AF  add         eax,dword ptr [ebp-8]
00DC13B2  mov         dword ptr [ebp-20h],eax
```

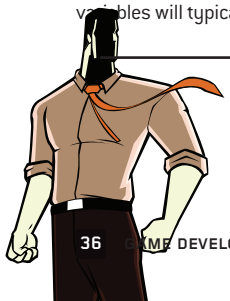
The same approach can be applied equally well to decipher the next three lines of assembly: The value at **ebp-14h** (**y**) is moved into the **eax** register, then the value at **ebp-8** (**x**) is added to the value of **eax**, and the value of **eax** (the sum of **y** and **x**) is moved into the address **ebp-20h** (the high high-level variable **z**).

This is an incredibly simple piece of assembly to look at, but believe it or not this has served to illustrate the difference in worldview between assembly and C/C++ admirably well. We're using x86 disassembly for this particular example, but you will find that the disassembly you encounter on more exotic development hardware (such as a PS3 or Xbox 360 devkit) will behave in much the same way; the mnemonics will differ, but these principles will still hold.

You now know several key principles about the worldview of assembly code which hold on the vast majority of hardware platforms. First, in order to perform a useful operation on some data a CPU will generally require it to be in a register; second, values in CPU registers must usually be explicitly moved to and from memory; and third, there is no machine-level concept of a named variable—the CPU deals with memory addresses, registers, and their contents.

THE STACK AND THE ABI

» Now that we understand the worldview of assembly code in principle we need to look at what is arguably the engine of C/C++—the Stack. If you are a C++ programmer and you're not 100% sure what the Stack is or how it works, you are not alone. *The C++ Programming Language* by Bjarne



"The more you do this, the easier it will get. You'll be a better, more effective, and more productive programmer; and—before you know it—probably one of the people others call over when they break the release build and need help debugging it."

Stroustrup (creator of C++), now on its 3rd third edition, is considered the standard text on C++ (at least until the update for the C++11 standard is published), and while it does from time to time refer to data or objects being "on the stack," it doesn't discuss what the Stack is or how it works. (For the sake of clarity, I'm going to capitalize the Stack to discriminate distinguish it from just any old instance of a stack data structure.)

As C/C++ programmers, we are comfortable that the entry point of our programs is the function `int main( int argc, char** argv )` and that our programs are structured primarily by calling other functions, which call other functions, and so on. In principle, a function can be called from anywhere at any time in a program, and since C and C++ support function pointers and library code, there is no way of knowing at compile time when or even whether a function will be called.

Additionally, the majority of functions need a little working space in memory (for local variables), and since only a relatively small number of nested function calls will generally be made at any given point, it would clearly be very wasteful to just reserve some memory for each function's working space—and it would almost certainly prevent us from using recursion.

So, how can we manage working memory for functions and have a standardized method for functions to call one another? Enter the Stack, stage left. The Stack is a stack data structure, and in a single-threaded program the Stack contains the vast majority of the data relating to the current execution state of the program and all non-global "automatic" memory (i.e., local variables) under the control of the compiler.

In case any of you are unfamiliar with the concept of a stack as a data structure, the concept of a stack requires only two operations: you can **push** data onto the top of a stack, which hides the previous top of the stack, and you can **pop** data off the top of a stack, which exposes the previous top of the stack. This interface gives stack data structures a property that has important implications: **N bytes** of data **push**ed onto a stack followed by ***N bytes*** of data **pop**ped off leaves it essentially unchanged.

Why is this important? Well, let's imagine some arbitrary function **A()** that calls another function **B()**— when **A()** is called it ***push***es a little working space on the Stack, and then while executing it calls **B()**. As long as **B()** ensures that it ***pop***s any working space it ***push***ed onto the Stack before it returns control to **A()**, **A()** can safely assume the state of the Stack is the same as it was before it made the call to **B()**.

Simply by using a stack as the medium for function local storage and enforcing a contract on how functions use the Stack when calling one another (*a Calling Convention*), we gain efficient and "automatic" management of local memory, and a mechanism that allows function calls to be arbitrarily nested. It is such a simple and elegant solution to the problem it solves that it is almost as if it were discovered rather than invented.

The working space used by a function within the Stack is normally referred to as its Stack Frame. Each function puts its own Stack Frame on the Stack when it's called, and removes it before it returns, so at any given point in a program's execution, the current sequence of nested function calls—or Call Stack—can be determined by looking at the Stack Frames currently stored in the Stack.

In fact, when you put a breakpoint in your code in your IDE of choice and use the Call Stack window to discover the sequence of function calls that got you to your breakpoint, the data used to populate that window is almost certainly derived by the debugger from the instantaneous state of the Stack.

While the fine detail of how the Stack is implemented varies from CPU to CPU, machine to machine, and compiler to compiler (and even with the same compiler and different compiler options), its operation is, *in principle*, identical on all hardware I have ever worked with.

In order to properly understand the Stack, we need to look at its operation in a little more detail. **Table 1** details the steps involved in calling a function. The function doing the calling is referred to as *Caller*, and the function being called is referred to as *Callee*:

Note that some of the items in the **Work Done** column are in [square brackets], and some are in italics: . Items in [square brackets] indicate that the work is not always required—for example, if the callee takes no parameters, then they need not be stored—while items in italics may not necessarily store their data on the Stack (this is platform-dependent). Also, "clean up passed function parameters" appears twice, but it will only happen in one place or the other, usually in Step 5 (Caller regains control), because if it happens in Step 3, it precludes the use of variadic functions (for example, functions like `printf()` which can take variable numbers of arguments.).

STEP	WORK DONE	NET STACK OPERATION
1. Caller prepares to call	store current execution state (i.e. CPU registers in use etc.) [store parameters expected by callee] store return address pass control to callee	push N bytes
2. Callee prologue	create Stack Frame allocating space on the Stack for local variables	push M bytes
3. Callee executes	[clean up passed function parameters] possibly call other functions	any zero sum combination of push followed by pop
4. Callee epilogue	de-allocate Stack Frame return control to caller	pop M bytes
5. Caller regains control	[use return value] [clean up passed function parameters — most platforms] reinstate function execution state	pop N bytes

TABLE 1: The steps involved in calling and returning from a function.

As mentioned previously, the details of the Stack's operation on a given platform are defined by a platform-specific standard called an Application Binary Interface (or *ABI*). The ABI specifies how functions should allocate their working space on the Stack, how parameters are passed to functions, how the various CPU registers are to be used, how various data types are represented, and so on. This standard is partly required to ensure compatibility between library code generated by different compiler vendors, but also typically ensures that best practices for using the hardware are followed.

Like the worldview of assembly, this information is a lot easier to take in and understand by working through an example. Just as before we'll be looking through the disassembly of a very simple piece of code (see **Figure 3**). This time, however, we'll be looking at it in a release build, since there are less instructions and use of the Stack is much more efficient, and therefore easier to follow in a memory window.

Most platforms have one ABI. Unhelpfully, there are several ABIs for x86, or (depending on your point of view) you might say there is one ABI with several not entirely compatible calling conventions. The disassembly shown in this article uses the "standard" x86 **cdecl** calling convention (which is default for a win32 project in Visual Studio 2010).

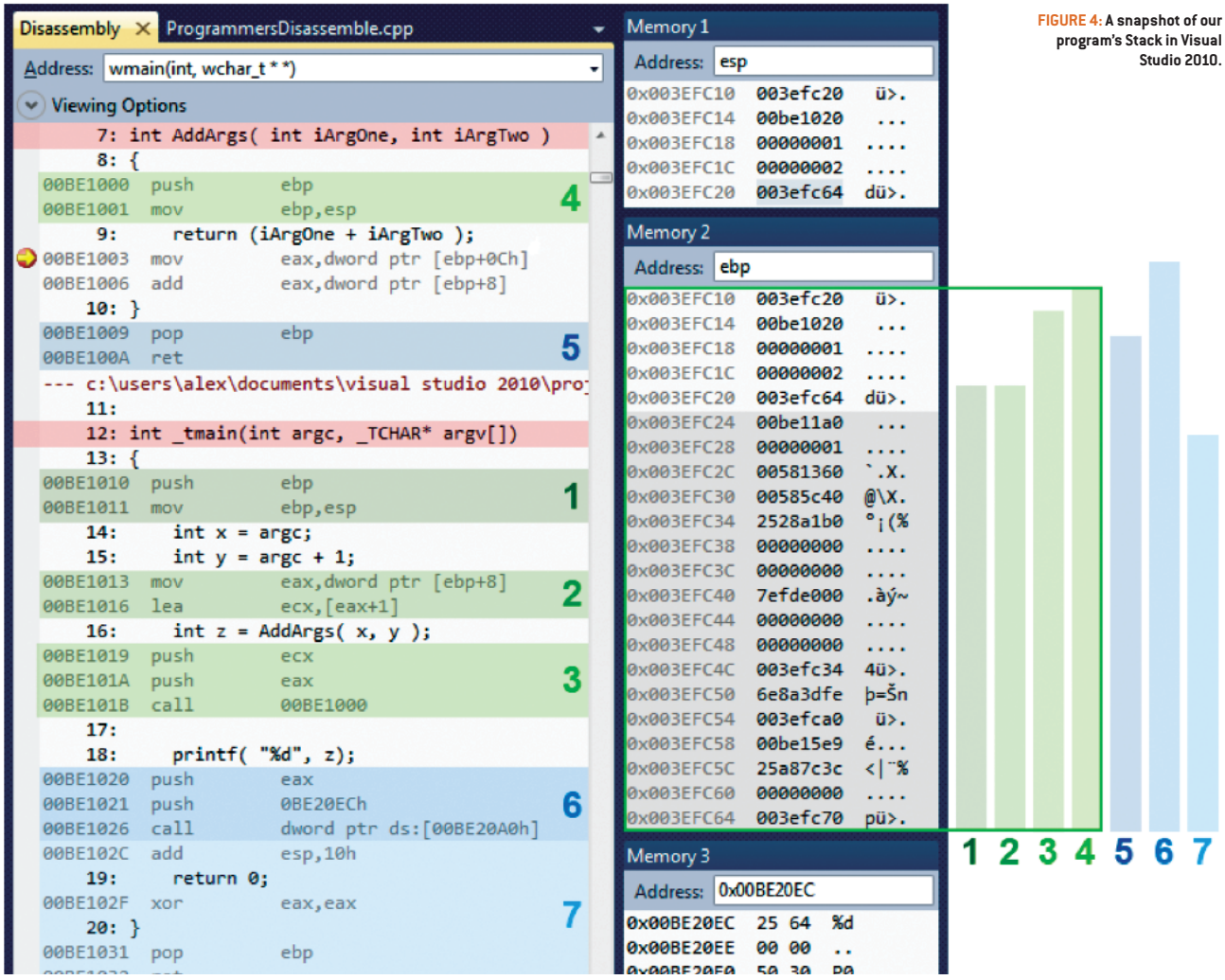


FIGURE 4: A snapshot of our program's Stack in Visual Studio 2010.

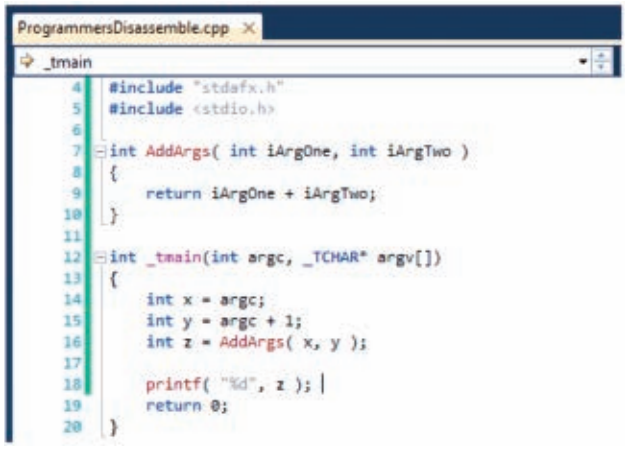


FIGURE 3: Our second sample code fragment.

Astute readers will be wondering why this code uses `argc` and `printf()`. This external input and output is necessary to prevent the optimizing compiler from optimizing the entire program code away in a release build. In fact, in order to demonstrate the Stack in action, I also had to turn off link time code generation, and all function inlining in the compiler options. It is a testament to the tireless efforts of compiler programmers over the years that it is non-trivial to prevent simple snippets of code from boiling away into nothing when compiled in a release build configuration!

Now let's take a look at **Figure 4**, which was originally a screenshot from Visual Studio 2010, but has been edited fairly extensively to highlight specific areas of interest. It certainly looks pretty intimidating, but once the meanings of the various annotations are explained it should appear less so.

Figure 4 represents a snapshot of the program's Stack state at the point of the deepest Stack use within our code—that is, immediately after the block of assembly numbered 4 in the diagram has been called. The diagram is broken up into several smaller areas, each of which holds different information.

The panel on the left with the tab named **Disassembly** is the disassembly of the simple C++ code snippet above. The next assembly instruction that will be executed is shown by the yellow arrow in the red circle located in the left left-hand margin of the window.

Memory 1 is a memory window whose top address is the value stored in the ***esp*** register. This window is set up to represent memory as 32 32-bit unsigned integers in hex format, and it shows the top of the Stack. **Memory 2** is a second memory window whose top address is the value stored in the

ebp register. This pane contains the entire content of the Stack belonging to our code and also (in the grey area) the Stack Frame of the function that called `main()`. **Memory 3** is set up to show bytes, and its top address is the memory containing the string constant "%d" that we pass to `printf()`.

The colored areas in the **Disassembly** pane delimit the significant blocks of assembly we will examine: Red blocks highlight the function names in code, green blocks show code that executed before this Stack snapshot, blue blocks show code that will execute after the Stack snapshot, and green and blue blocks are numbered in order of execution. The green and blue bar graph on the rightmost edge of the image show the Stack depth immediately after each of the corresponding colored blocks.

To make sense of this concrete example of the Stack in action we're going to step through the numbered blocks looking at each order, but before we do this we are going to have a quick look at two special-purpose x86 registers called **ebp** and **esp**, and at the x86 assembly instructions **push** and **pop**, which are all instrumental in the implementation of the Stack in the standard x86 assembly generated by the Visual Studio 2010 compiler.

The register **esp** is normally referred to as the Stack Pointer, and it holds the address of the top of the Stack. The register **ebp** is normally referred to as the Base Pointer or the Frame Pointer, and it holds the base address of the current function's Stack Frame; local variables and function arguments are typically accessed by an offset from the **ebp** register (as we saw in our first example).

The assembly instruction **push** is responsible for **push**ing its operand onto the top of the Stack; it does this by affecting the value of **esp** so that the size of the Stack is increased and then writing to the new address of **esp**. The assembly instruction **pop** is responsible for **pop**ping its operand off the Stack. It does this by setting its operand to the value at the address stored in **esp**, and then reducing the size of the stack.

A crucial fact about the x86 Stack is that it grows downward in memory address space. When data is ***push***ed onto the top of the Stack the value of **esp** is decreased, and when data is **pop**ped off the Stack the value of **esp** is increased. **Table 2** shows an example use of **push** and of **pop**, and the equivalent assembly code that would have the same effect:

EXAMPLE USE	EFFECT	EQUIVALENT ASSEMBLY
push eax	Increase the size of the Stack by 4 bytes (i.e. the size of an x86 register). Store the value of the *eax* register on the Stack.	*sub esp, 4* *mov [esp], eax*
pop eax	Gets the value off the top of the Stack into the *eax* register. Reduces the size of the Stack by 4 bytes.	*mov eax, [esp]* *add esp, eax4*

TABLE 2: **Explaining the use of push and pop.**

Now we're ready to walk step-by-step through the Stack.

```
    12: int _tmain(int argc, _TCHAR* argv[])
    13: {
00BE1010  push        ebp                    1
00BE1011  mov         ebp,esp
```

BLOCK 1: prologue of `main()`

As we saw from **Table 1**, the function's prologue is primarily responsible for creating its Stack Frame; the space in which its local variables are stored. Let's look at the assembly that does this.

First, the instruction **push ebp** "closes" the caller's Stack Frame, storing the base address of the calling function's Stack Frame on the Stack so it can be reinstated before `main()` returns. Next, **mov ebp, esp** stores the value of **esp** in **ebp**, which sets the base address of `main()`'s Stack Frame to the current top of the Stack.

Now we're ready to take the last step in creating a Stack Frame: growing the stack to make room for the function's local variables by **sub**tracting the size required by the local variables from the current value of **esp**. However, in the code block we're looking at, the compiler has already optimized the code by using registers to store the values for ***x*** and ***y***, so `main()` has no local variables and we don't have to subtract anything. Essentially, the **sub esp, 0** instruction isn't actually there, but it's implied by the ABI, so it would be remiss of me not to mention it.

Looking at the **Memory 2** pane in **Figure 4**, we can see (from the gray area representing the Stack before `main()` was called) that the top of the Stack was 0x003EFC24 when we entered `main()`.

Looking at vertical bar on the right labeled 1 (which represents the Stack depth after **block 1**) we can see that the top of the Stack is now memory address 0x003EFC20, which holds the value 003efc64 that the prologue has **push**ed onto the Stack; , which was the value of **ebp** when `main()` was called.

We also know that **esp** and **ebp** both currently have the value 0x003EFC20.

```
    14:     int x = argc;
    15:     int y = argc + 1;
00BE1013  mov         eax,dword ptr [ebp+8]  2
00BE1016  lea         ecx,[eax+1]
```

BLOCK 2: initializing `x` and `y`

As we have briefly discussed already, the optimizing compiler has managed to do away with our local variables, so let's look at what it's done with them. First we have the **move eax, dword ptr [ebp+8]** instruction: note Note that positive offsets from **ebp** are *below* the current function's Stack Frame, which typically indicates a function parameter is being accessed. As it happens, the **dword ptr**-sized value at **[ebp+8]** is **argc**, and its value is being **mov**ed into the general purpose **eax** register, so we now know that ***eax*** holds the value defined as ***x*** by the C++ code.

After that comes the **lea ecx, [eax+1]** instruction. ***lea*** (***L***oad ***E***ffective ***A***ddress) is another assembly instruction we've not yet seen. It evaluates its second operand as an address, and stores the result of that evaluation in the register specified by its first operand. As I understand it, ***lea*** was originally intended to help optimize array access, but you will very commonly see it used like it is here, as an optimization for combined addition and multiplication. In this case, it is clearly adding one to the value in **eax** (which we know is `x`) and storing the result in **ecx**, which is another general purpose x86 register. So we now know that the value we defined as `y` in the C++ code is in the ***ecx*** register. Looking at the vertical bar on the right for **block 2**, we can see that this has left the Stack unchanged.

```
    16:     int z = AddArgs( x, y );
00BE1019  push        ecx                    3
00BE101A  push        eax
00BE101B  call        00BE1000
```

BLOCK 3: calling `AddArgs()`

This block shows how the Stack is used by the default x86 calling convention to pass arguments to functions. Under the standard x86 **cdecl** calling convention, arguments are passed to a function by pushing them onto the Stack in the reverse order that they are declared in the high high-level code, and the 'return address' for the callee to return execution to after it finishes executing is pushed onto the Stack after the arguments. This behavior is specified by the calling convention, and the assembly of the callee expects the return address to be on the top of the Stack when it is called so that it knows where to return to.

In this block, we start with **push ecx**, which puts the value of ***ecx***—which we know is `y`—on the top of the Stack; this can be seen at address 0x003EFC1C in **Memory 2**. Next comes **push eax**, which pushes **eax** (which we know is `x`; this can be seen at address 0x003EFC18 in **Memory 2**. After that we see **call 00BE1000**.

The instruction ***call*** is new to us, like **push** and **pop**, it is a special instruction that combines work which can be done with multiple other assembly instructions. When **call** is executed it **push**es the address of the next instruction onto the Stack (this is the return address), then it jumps the execution to the address specified by its operand, which will be the address of the first instruction in the prologue of the callee function.

Comparing the vertical bar for **block 3** with **Memory 2** in the diagram, we can see that immediately before the first instruction in `AddArgs()` is executed; address 0x003EFC14 is the top of the Stack, and holds the return address 00be1020, which is the address of the next instruction after the call to `AddArgs()`; address 0x003EFC18 is 4 bytes from the top of the Stack, and holds the value of `x`; and address 0x003EFC1C is 8 bytes from the top of the Stack, and holds the value of `y`.

```
      7: int AddArgs( int iArgOne, int iArgTwo )
      8: {
00BE1000  push        ebp
00BE1001  mov         ebp,esp                    4
```

BLOCK 4: prologue of `AddArgs()`

The prologue of **block 4** is identical to the prologue of `main()`, so there is no need to cover the assembly instructions or their effects in detail. By consulting the diagram, we can see that the base pointer of `main()`'s Stack Frame (0x003EFC20) is now stored on the top of the Stack; **ebp** is now the base address of `AddArgs()`'s Stack Frame which has replaced `main()`'s as the topmost Stack Frame on the Stack; and `AddArgs()`, like `main()`, has no local variables and so its Stack Frame has a size of 0 bytes (i.e., ***ebp*** == ***esp*** == 0x003EFC10). The values corresponding to `AddArgs()`'s parameters ***iArgOne*** and ***iArgTwo*** are now at the memory addresses ***[ebp+8]*** and ***[ebp+0Ch]*** respectively.

```
      9:     return (iArgOne + iArgTwo );
00BE1003  mov         eax,dword ptr [ebp+0Ch]
00BE1006  add         eax,dword ptr [ebp+8]
     10: }
00BE1009  pop         ebp                        5
00BE100A  ret
```

BLOCK 5: epilogue of `AddArgs()`

Before we consider the epilogue, we should note that the x86 ABI specifies that the **eax** register should be used to return integer values from functions (note: floating Floating point is another story entirely).

Looking at the assembly instructions between **block 4** and **block 5**, we can see that the parameters ***iArgOne*** and ***iArgTwo*** (on the Stack at ***[ebp+8]*** and ***[ebp+0Ch]*** respectively) are being added together in **eax**, so `AddArgs()`'s expected return value is in **eax** immediately before the start of the function's epilogue.

The first instruction in this line is **pop ebp**, which takes the value on the top of the Stack and stores it in **ebp**. The last thing pushed was the value of **ebp**, which we know was the base address of `main()`'s Stack Frame at the point when the function was called. Restoring this value into **ebp** removes `AddArgs()`'s Stack Frame from the Stack and reinstates `main()`'s as the topmost Stack Frame. This reduces the size of the Stack by 4 bytes (by adding 4 to **esp**).

The second instruction in this line is **ret**, which is new to us. It is essentially the opposite of the instruction **call**—when ret is executed, it **pop**s the return address off the top of the Stack, and then jumps execution back to the return address.

```
     17:
     18:     printf( "%d", z);
00BE1020  push        eax
00BE1021  push        0BE20ECh                   6
00BE1026  call        dword ptr ds:[00BE20A0h]
```

BLOCK 6: call `printf()` from `main()`

Now, since we only called `printf()` to prevent the compiler from optimizing the C++ variable `z` away, this assembly code doesn't really matter too much for the purpose of illustration. That said, it does push parameters to `printf()` onto the Stack, which makes the size of the Stack relevant to the function prologue of `main()`. It also does a couple of things we've not seen before, so it's worth looking at.

This block begins with **push eax**; we know ***eax*** contains the return value of `AddArgs()`, so this code is just passing the C++ variable `z` as the 2nd second parameter to `printf()`. Next is **push 0BE20ECh**; looking in the diagram at **Memory 3** we can see that this address is the first byte of the string constant "%d". This makes sense, since we know from the previous call to `AddArgs()` that parameters are pushed onto the Stack in reverse order. Then we have **call dword ptr ds:[00BE20A0h]**. This is clearly calling a function (`printf()`), but why does it use a different syntax for the address operand? Well, **ds:[...]** indicates an offset from the **d**ata **s**egment register; this register is used in an x86 addressing mode that was inherited from 16-bit windows, and which is now primarily used to call functions that are linked via .dlls. Since `printf()` is part of the standard C library, and the default option for a win32 console app is to load this library as a .dll, this is as one would expect.

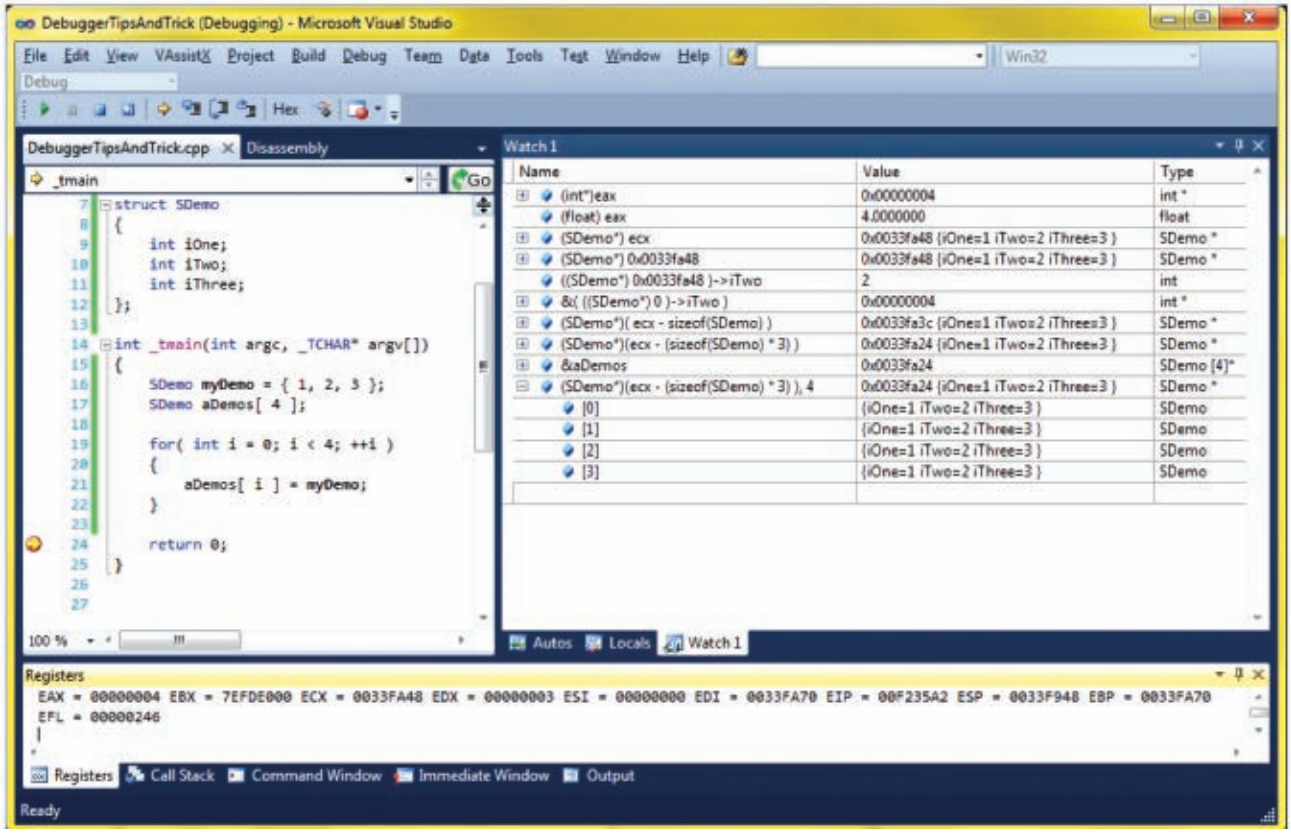
The important thing to note about this block is that it doesn't clean up any of the parameters that were pushed onto the Stack in order to call `AddArgs()`. It simply pushes more parameters onto the Stack in order to call `printf()`. Looking at the vertical Stack depth bar corresponding with the end of **block 6** in the diagram, we can see that immediately after the **call** instruction the Stack is now deeper than after the prologue of `AddArgs()`.

```
00BE102C  add         esp,10h
     19:     return 0;
00BE102F  xor         eax,eax                    7
     20: }
00BE1031  pop         ebp
00BE1032  ret
```

BLOCK 7: clean up the Stack and exit

We're almost there! This block of assembly is typical of the epilogue of a non-leaf function (i.e., a function that calls other functions). **add esp, 10h**

FIGURE 5: **A few examples of handy watch window casts..**



resets the Stack to the state it was in immediately after the prologue of `main()`, essentially undoing all **push**es for passing function parameters. **xor eax**, **eax** sets the return value (**eax**) to 0—**xor** is a self-explanatory assembly instruction, and the XOR of anything with itself is 0. **pop ebp** removes `main()`'s Stack Frame from the top of the Stack, reinstating the Stack Frame of whatever called `main()` as topmost. Finally, **ret** pops the return address (00be11a0) off the top of the Stack and jump execution to that address. That's it!

SO, WHAT NEXT?

» You should now have a good idea of what the worldview of assembly code is like, and have a good basic grasp of how the Stack works both in principle, and via a (relatively) simple concrete example of that principle in action on the x86 architecture. So, what next?

The next step is to take what you've learned from this article and apply it, and then apply it some more until you really understand it. The next time you're debugging and you're not immediately pressed for time, go and turn on the disassembly view and try to make sense of what's going on at the assembly level.

If you find an assembly instruction you don't know: , Don't don't panic! Just do what the pros do; search the internet, check the hardware manuals, post on Stack Overflow, or post on your hardware vendor's support forums. There are plenty of places where experts are more than willing to share their knowledge with people who want to learn.

The more you do this, the easier it will get. You'll be a better, more effective, and more productive programmer; and—before you know it—probably one of the people others call over when they break the release build and need help debugging it.

MAKING YOUR DEBUGGER WORK FOR YOU

» Now that we've worked through this simple piece of assembly code, it should be pretty clear that once you're armed with an understanding of the worldview of assembly code, and the way the Stack works to manage local variables and function calls, it's nowhere near as scary as it looks.

This section covers a few invaluable hints and tips that I've picked up about debugging in release mode—since there's no debugging information in your executable, you need to make up for that by knowing how to get the debugger to tell you what you want to know whenever it possibly can.

SYMBOLS PLEASE!

» When you compile code in a debug configuration, it will generate a file of some sort that contains the symbols in your code. This data is part of the data that allows your debugger to match the source code files up with the assembly and let you debug your code at the source level.

Make sure your release build is generating symbols (or your platform's equivalent—a .map file, for example). If you can't see a call stack in your debugger when debugging a release build, then there is a good chance you're not building symbols in release. Your debugger will be able to make use of the symbols to display the names of functions and global variables (singletons and such), and you will have a far, far easier time debugging with symbols than without them.

When you're working on a console, you should be archiving symbol files along with the other temporary data needed to make a disc as part of your build process for discs that are in any kind of test process—debugging a core dump without symbols is not an easy task.

MEMORY WINDOWS

» Since the code is optimized, the debugger doesn't necessarily have all the information available that it needs in order to work out the correspondence between the high-level code and the assembly code you're looking at, so the usual debugging staples of watch windows, autos, and local panes are unlikely to be of much use.

Memory windows obviously allow you to get around this issue by looking directly at memory yourself. As long as you have an address, you can see exactly what value is stored in it. But there is another less-obvious use for memory windows: You can easily keep track of the Stack by opening a memory window and using the name of the Stack Pointer register and Stack Frame pointer register (if your system has one) where you would normally put a memory address. On x86, as we know, the Stack Pointer is **esp**, and the Frame Pointer (or Base Pointer) is **ebp**.

To make the memory window in Visual Studio 2010 update as the value of the register changes, you will need to click the Reevaluate Automatically button just to the right of the "Address:" edit box. With most debuggers, you can also ask the memory window to interpret and display the memory it's viewing as if its content were of a specific type (int, float, char, etc.) and control how those values are laid out in the window. This is how I got the Stack to be displayed in a manner that was useful for following the second simple example of assembly code, so you should definitely play with the options available in your debugger until you're happy that you know what it allows you to look at.

WATCH WINDOWS

» Most of the debuggers out there will support way more functionality than you expect in their watch windows.

The `sizeof()` trick: You should be able to use the `sizeof()` operator to find out the size of any type that's in the executable you're debugging. It generally uses the standard syntax without the semicolon. You also usually need to give the type name fully qualified with namespaces.

View contents of registers and memory addresses: Most debuggers also allow you to use the name of the register you want to watch in a watch window and see the content of it just like any variable name, and the same also goes for (valid) memory addresses. This is often a lot more convenient than using the Registers or Memory windows, since you usually only want to see one or two registers, or a small range of memory addresses at any given time.

The casting types trick: Most debuggers support C-style casts in watch windows these days. This is incredibly useful because you can ask the debugger to treat any value as a pointer to any type in your codebase and to evaluate it and display it as the type you ask it to in the watch window. You typically use the standard C / C++ syntax, and as with `sizeof()`, you'll need to use type names fully qualified with namespaces.

Figure 5 shows examples of the sort of watch window casts I use frequently, which include register as pointer to type, memory address as pointer to type, offset from register as pointer to type, any of the above as an array of arbitrary size specified in the watch window, byte offset of a member within a struct, and my personal favorite, which gets the byte offset of a member within a struct / class. This is really useful for following assembly code inside libraries for which you can only access the headers.

Note that the code in **Figure 5** uses **ecx** to contain the pointer to the current element in `aDemos` in the **for** loop, so at the point shown it is pointing to the last element in `aDemos`.

ALEX DARBY *is technical director at Birmingham City University's Gamer Camp Pro (MSc in Video Game Development). He has been working in the game industry since 1996, and is very proud to have been one of the founding members of the FreeStyleGames team and to have been involved with the creation of* DJ HERO. *He is also a husband, dad, part-time indie developer, and skateboarder.*

who went where

/// Former Sega West CEO Mike Hayes has joined up with a small, England-based mobile/smart TV startup called Caperfly. Caperfly makes interactive titles for iOS, Android, and smart TV platforms.

/// Ex-Yahoo! CFO Blake Jorgensen has left for a similar role with Electronic Arts, filling the gap left by Eric Brown, who resigned earlier this year.

/// Relic Entertainment co-founder Ron Moravek has rejoined THQ as the executive vice president of production, and will be focusing on the company's new digital business model strategy. He initially joined THQ when it purchased Relic in 2004, and left in 2006 to work at Electronic Arts Canada.

new studios

/// Several ex-Epic Games staffers, led by GEARS OF WAR 3 gameplay designer Lee Perry, have formed a studio called BitMonster Games. Their first game will be a mobile adventure RPG called LILI, which will be built in Unreal Engine 3 and feature a "noncombat" battle system.

/// Former Zynga lead designer Dave Pottinger (CASTLEVILLE), Robot Entertainment senior programmer John Evanson, and id Software senior character artist Jason Sallenbach are forming a new mobile game development studio called BonusXP. All three worked together on the AGE OF EMPIRES series at Ensemble Studios.

/// Electronic Arts is merging its two Melbourne, Australia-based internal mobile studios (iOS DEAD SPACE developer IronMonkey Studios and FLIGHT CONTROL developer Firemint) into one studio called Firemonkeys. In a public statement, EA said that the aim of the merger is to create a more focused team, though both studios will continue to work on separate projects.

/// Konami has announced plans to open a London-based studio devoted to its PRO EVOLUTION SOCCER franchise. The new studio will focus on PES titles for consoles and PC, and will look to "adopt and recreate local football culture" as a move to keep the series fresh.

Street Fighter All-Star

SETH KILLIAN LEAVES CAPCOM FOR SONY

LONGTIME CAPCOM STRATEGIC MARKETING DIRECTOR (AND STREET FIGHTER SPECIALIST) SETH KILLIAN JUMPED SHIP TO SONY SANTA MONICA IN EARLY JULY TO WORK AS A LEAD DESIGNER ON (AMONG OTHER THINGS) PLAYSTATION ALL-STARS: BATTLE ROYALE BY SUPERBOT ENTERTAINMENT. GAME DEVELOPER CAUGHT UP WITH SETH TO TALK TO HIM ABOUT HIS CAREER SHIFT.

Patrick Miller: *Didn't you start your professional career teaching philosophy? How'd you get into game dev?*

Seth Killian: I got into game dev by playing and thinking about games for a lifetime, and then having a bit of good old "right time, right place" luck. I grew up in arcades, and while I was in graduate school, I was also one of the best STREET FIGHTER players in the U.S. I've always played a lot of games—from poker, to chess, to the endless games I would make up for me and my friends as a kid—and even when I wasn't playing games, I would study them.

PM: *Your new job title is lead game designer for the external group—exactly how does this work?*

SK: Sony is the publisher, and pays the bills for SuperBot's work on ALL-STARS, so at the end of the day, my responsibility is to make sure we're on board with their vision. In practice, however, it's simply collaborative.

PM: *You seem to create job roles rather than fill existing ones. How do you pull that off?*

SK: I do have a history of titles built around me, rather than the other way around. I'm not sure that's always a good thing, but even at Sony, it was more of "we want you here—how about this job and then whatever else you want to do" than it was an existing opening that needed to be filled.

It may seem strange from the outside, but to me, philosophy and game design have a lot in common. Both deal pretty directly with balancing and manipulating complex, abstract systems. Sliding around variables, making up new variables, or simply inverting entirely the way people had approached a subject previously. There's a surprising amount of overlap. I loved teaching, but also wanted to try something new, so when I got the chance to join Capcom help revive STREET FIGHTER, I had to do it.

I think I get oddball job titles because I tend to worry less about what I'm officially supposed to be doing, and focus on what needs to be done the most. I care a lot about the details, but I always come back to the big picture, so I gravitate toward whatever I feel isn't getting the attention it needs, irrespective of my official duties. I'm like a free safety in football.

Sometimes people welcome that and it's an easy win, other times it strains the bounds of the corporate organization. I just try and make everything better for the people that play our games. When I say that out loud, now it seems strange that there isn't already a staff job to do that.

PM: *Before working with Sony Santa Monica, you were with Capcom USA. Think you've developed any specialized skills that help with U.S.–Japan collaboration?*

SK: I think I was able to have some success at Capcom working with the Japanese teams for two reasons. First, I knew what I was talking about—I had studied their games deeply for most of my life, and played some of them at a world-class level.

Beyond that, I always try to listen carefully to what people are really saying. I think some Westerners struggle when dealing with Japan because the most important things being said are often implied rather than stated directly—similar to the way artists can use negative space in powerful ways. In the West, implying things rather than stating them can be seen as confusing or obtuse, so lots of struggles start there. To get past it, listen to what people really mean, rather than simply looking at the words they are using. I think much of the nuance of Japanese communication can come through clearly even in translation, so if you can get a sense for that, it will help you immensely and not just with Japanese counterparts—it's good training for communicating effectively with anyone.

Ideally, you would spend the last few weeks of any project adding the little tweaks and contextual details that take your work from good to great. All too often, though, you will instead be interrupted by frantic bouts of cuts; no matter how good you are, how efficiently you use your textures, how carefully you lay down your keys, or how tightly controlled your poly budgets are, it's a given that at some point you will find yourself and your work trembling at the foot of the chopping block. You'll have to see characters you've known for several months at a crisp high resolution be degraded with smudgier textures and cruder silhouettes, and you'll probably see the big, important models and animations take the blow so less-important assets can escape deletion. Cutting is no fun.

Any pro knows the importance of good technique and efficiency. Too many of us, though, don't look ahead to the inevitable bloodbath at the end of the project and prepare for it ahead of time. Planning is to games what "just fixing things" is to political problems—everybody wants it to work, but we're all just a little fuzzy on how, exactly, it's supposed to happen. As with politics, it helps if you get more actively involved. Here are a few tips for making sure that the art-cutting process is as painless as possible.

FOR WHOM THE BELL TOLLS

» The first things on the chopping block are the ones that haven't been done yet. If an asset (or a cluster of assets) is still just a gray box when the budget ax starts to swing, it's probably a goner. This means that you need to prioritize your scheduling a long time before the final sprint. You don't want big holes in your game because some anxious producer decided to "simplify" things by nixing a bunch of models or animations you have already planned out but haven't worked on yet.

It's safer to group assets in your schedules into logical units so that interrelated models, animations, and shaders come online—or get chopped—together. It's painful enough to lose a bit of the game you've been planning for months or years, but it's far more painful to lose those parts when you've already sunk precious time into subordinate pieces that don't make sense on their own. For example, if you're building a set of areas with diverse art styles, it's better to schedule similarly styled pieces together so that you don't find yourself with many megabytes of jungle-themed models and textures that will all get cut because the jungle animations never got done and the grim gods of scheduling have decreed that the jungle must go.

The exact boundaries of a logical unit are going to vary from one project to the next, of course. In games with a broad geographical reach, each unique landscape or climate might be a logical unit. In games with a tighter focus on story, it might be characters or scenes that are the primary scheduling chunk. The art team needs to think hard about how the pieces of the initial design and scope fit together, so that smart and relatively safe schedules can be negotiated early.

Designing your content and scheduling your work to keep things grouped logically will make the cutting process less brutally random when things get crazy. The last thing you want to do is start with an alphabetized Excel sheet and work thorough it from A to Z (especially if you're making a game in which zebras, yaks, or wildebeests are critical to gameplay). Revisit the schedule frequently, make sure you're on track, and revise plans as things come up. Keep the things that are most important to the game at the top of your list to protect them from late-game panics. Don't be afraid to revise your priorities as the game evolves. Above all, artists should be actively keeping an eye on how much content is done, how polished it is, and what's expendable. When the inevitable scoping meetings start, the art team should be able to walk in with good, up-to-date numbers and ready recommendations when the other disciplines start looking for savings.

A LIVING WILL

» A good plan recognizes the need for fallback positions. Plan as much of your content as possible in sets, so that you have a good plan in place for replacing assets that fall before the executioner's blade. If you've already got six different kinds of single-family houses in your game, cutting the seventh is not a bitter loss, particularly if the gameplay and setup for all of the house types are fairly similar and the swap is not so laborious. Ideally, you can get the game working with a subset of your pieces, and go on to refine its looks and gameplay progressively by swapping in more varied content as the project goes on. For example, it's probably better to start with one tree and add more gradually, instead of creating a large library of trees right at the outset of the project before the limits on memory and processor power define the foliage budget.

Unique individual pieces, on the other hand, present more difficulties when the ax is swinging—they will have idiosyncrasies that can't be easily mapped onto other content, but also they tend to have a significant impact on the way the game feels to the players. These capstone pieces are tricky to plan for. One school of thought says that the precious artist time devoted to one-offs is better spent on assets with high reuse value that get more playtime. Unfortunately, this pushes many of the game's most interesting and identifiable visuals onto the back end of the schedule, where they are exposed to the cruelty of the budgeteers. If things go badly, the result is a dry, functionalist game that's lacking in character and memorable moments. On the other hand, finishing the capstone pieces early can make for awkward gaps or sparsity when the game is scoped down—a beautiful fantasy fortress loses a lot of its impact when it's placed in a town that contains only two different cottages. If you're careful to find logical scheduling units, this problem is somewhat simplified because it's not an all-or-nothing choice—the capstones are set in the context of a unit that stands or falls together.

DEATH BY INCHES

» Of course, the death of a thousand cuts that we all fear doesn't only come in the form of models or animations that don't make the final game. The most wrenching cuts are those last-minute grabs for memory or performance that degrade assets that are already finished and looking good. There's nothing more disheartening that firing up a playtest build to find a familiar character suddenly turned into a MINECRAFT model by a last-minute downsizing, or a finely crafted cityscape suddenly turned into a screenshot from THE SIMS circa 2003. To add insult to injury, it is all too often the most important assets—the ones that get the most screen time and form the biggest impression on the players—that get cut the most brutally. To a desperate producer or engineer looking for quick savings, the character with 16 megs of special detail textures looks a lot more immediate than a folder full of insignificant props that weigh in at 384k apiece. The fact that halving the texture resolution on that character will torture the art staff may not matter much to the lead who's frantically scrounging for RAM.

To some degree, this is an inescapable problem. It's certainly made more galling by the way graphics hardware works—the powers-of-two rule is a hard taskmaster when it comes to saving memory. So you can understand why the budget mavens swoop down on the most gorgeous assets when things get tight—but that doesn't make it hurt less.

Sometimes, you can finesse your way past the budgeting problem. For example, you can get around the powers-of-two rule if you atlas several assets together into larger textures. This allows you more fine-grained control over memory use. Imagine, for example, that you have a set of six assets, each with its own 512-by-512 textures. If you atlas the textures for six of these assets together onto a single 1024-by-1024 texture, you could go from two megs of texture memory to 1.33 megs, and the losses would be spread evenly over all six assets. The typical "across-the-board" cut would instead pick three of those six assets and halve their resolutions, which is a far more noticeable change.

Unfortunately, atlassed assets are a pain to work with—artists are notoriously bad at sharing files, and having six artists fighting for control of a single texture would be a nightmare, especially while the assets were still evolving. Manual atlassing is also fairly labor-intensive compared to simply choosing Resize Image in Photoshop. Nonetheless, it's a valid option to consider when the headsman is at the door, particularly if you know the assets in question are in their final visual and gameplay forms. It's a good trick when you really can't stand to see all your pretty texture work go poof two weeks before gold master.

Like death and taxes, cuts are inevitable. If you manage to finish a project without major cuts, you are either a planning genius or just not ambitious enough. The rest of us should be ready to deal with the cuts when they arrive. Even if nothing can console you for the loss of a favorite bit, at least you can use a little foresight to make sure that the last-minute budgetary crises don't flush months of solid, completed work down the drain.

If you're working on the next CRYSIS, you probably don't want your bad guys to look like this.

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.*

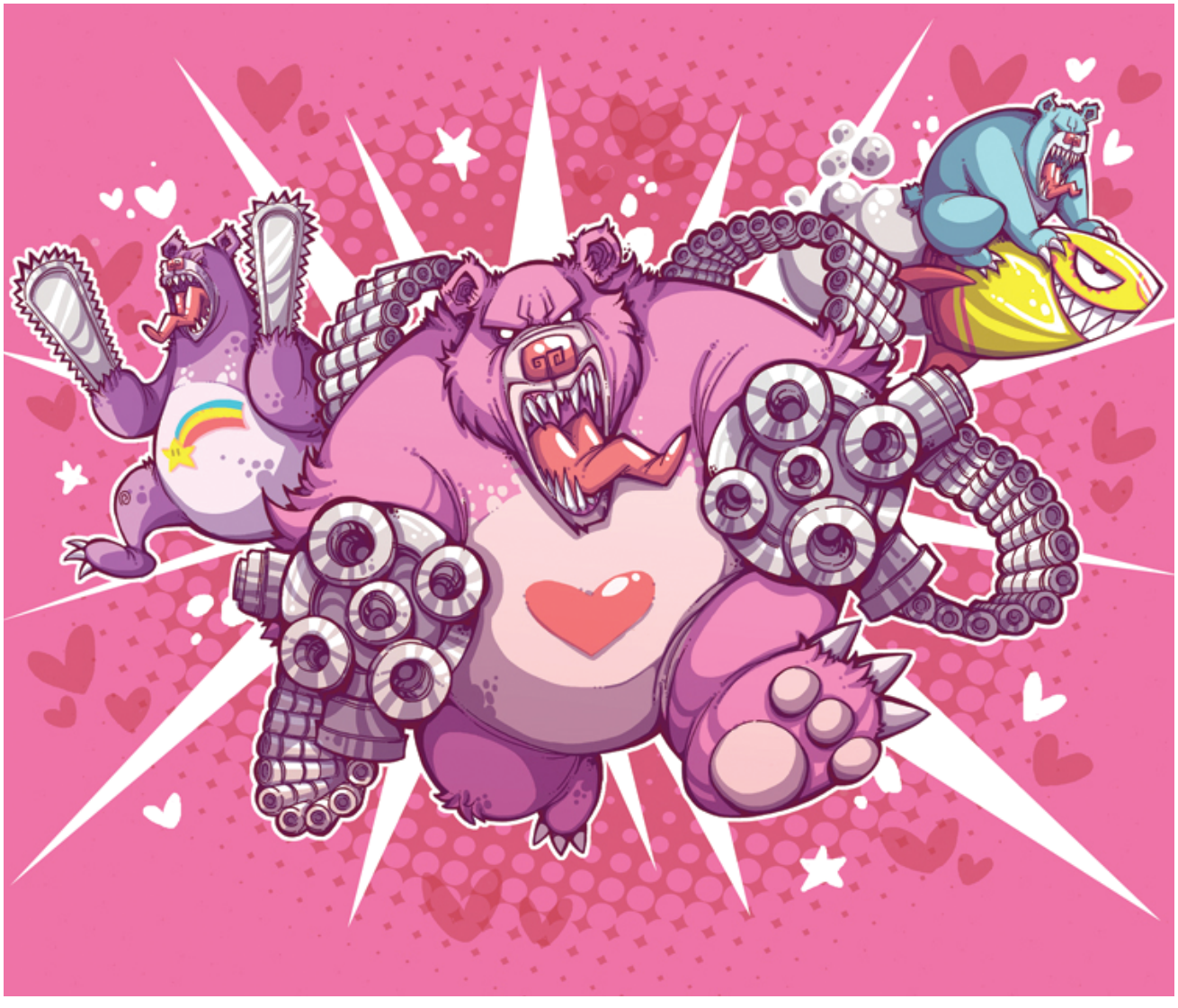


ILLUSTRATION BY JUAN RAMIREZ

THE CARE BEAR MYTH

DEBUNKING A GAME DESIGN URBAN LEGEND

Most game designers think that human play-style preferences can be mapped to a spectrum with player vs. player (PvP) on one side, and player vs. environment (PvE) on the other. Players who prefer PvP are aggressive and like competition. Players who prefer PvE are more passive and don't want to compete—like Care Bears. Most game designers are wrong.

While this PvP / PvE split may well be the best way to categorize some of the features of an MMO server, the research I have been doing over the past few years seems to indicate that this model has nothing to do with actual human motivations. From what I can tell, there is no such thing as someone who doesn't enjoy winning in one form or another. Care Bears are a myth, and thinking about our players as PvP or PvE completely misses how humans are actually motivated—which in turn makes it harder for us to design games that appeal to them. The opposite of player vs. player is actually team vs. team. At least, that's what my research subjects are telling me.

In the May issue of *Game Developer*, I wrote an article called "The Five Domains of Play," in which I explained how I have been using the Big Five model from modern motivation psychology to study how game design elements can appeal to different kinds of players. In doing so, I've been conducting "qualitative interviews" with players, where I give them the Big Five test (you can take it for free here: www.personal.psu.edu/ffj5j/IPIP), and then I talk

to them about what satisfies them when they play games. Before I explain why Care Bears don't exist, I'll need to explain a bit more about the Big Five model, and how we think of competition as players.

WHAT'S A FACET?

» In Big Five terms, a facet is one of the measurable atomic spectra of personality. Take adventurousness, for example: If you score high in adventurousness, you are more inclined to be interested in what's over the next hill, around the next corner, inside the box, and so on. If you score low in adventurousness, you're more interested in what you know, your stomping ground, the familiar cycle of routines, and the known. Humans, as it turns out, show something like a normal standard distribution along the spectrum of those two extremes.

The core of my research has been to correlate facets like that with game elements. Unsurprisingly, the adventurousness facet correlates remarkably well with, for high scorers, a preference for exploration and discovery in gameplay. Low scorers in this facet show a marked preference for base building and exploration through conquest. In the words of one of my interviewees, "I only explore something after it is part of my domain."

Let's jump to the competitiveness facet. Different researchers call this facet different names, most of which focus on the opposite end of the motivation spectrum: compliance, accommodation, and cooperation, for example. The International Personality Item Pool (seen at http://ipip.ori.org, where much of the core data for this research is stored) describes the competitiveness facet thusly:

Individuals who score on one end of this scale dislike confrontations. They are perfectly willing to compromise or to deny their own needs in order to get along with others. Those who score on the other end of this scale are more likely to intimidate others to get their way.

Sounds pretty straightforward, right? On one side, we should have nonconfrontational HARVEST MOON and ANIMAL CROSSING players, and on the other hand we should have tea-bagging HALO campers. Right?

That's certainly what I expected from this facet when I started this research, so I formulated my questions accordingly: "Do you seek games with competitive play in them?" "Do you prefer PvP, PvE, or something else?" "Do you play online multiplayer deathmatch?" Each of these questions would trigger conversations about respondents' play styles and preferences.

I knew my theory was off-target in some of my earliest interviews. One of my very first interviewees was a low-competitive-scoring player who loved HARVEST MOON, ANIMAL CROSSING... and HALO deathmatch. Loved it. As in, one of her top five games of all time.

PLAYER VS. TEAM

» After the sixth or seventh person with a score on the cooperative side of things regaled me with tales of how awesome CALL OF DUTY is, it was pretty clear that something was wrong with this theory. So I started interrogating my victims in new ways, looking for insight.

Looking at my surveys, I noticed something: While my cooperative players were often happy playing games like CALL OF DUTY, HALO, and LEAGUE OF LEGENDS, my competitive players were playing a lot more STARCRAFT. So I dug in and made the connection that would help fix my survey questions. (By fix, I mean that I started getting answers to my questions that lined up respondents' personality scores with my predictions.)

Players with a high score in competitiveness often have little to no preference about whether they are on a team. They generally don't care about the rest of the team, as long as they personally have an opportunity to crush their opponents. These players tend to prefer STARCRAFT 1v1 and CALL OF DUTY free-for-all. On the other end of the spectrum we have cooperative players, who rarely report getting much satisfaction from the personal victory of a STARCRAFT 1v1 match, but report a huge sense of satisfaction from their team winning. From what I can tell so far, there are few (if any) humans with a strong motivational desire for both of those experiences. They are motivational opposites.

BUT WHAT ABOUT CARE BEARS?

» Earlier, I said that Care Bears are a myth. That's a slight exaggeration; if we reduce our examination to just this competitiveness spectrum, there is next to no one out there who doesn't enjoy competition in some way. Care Bears are supposed to be wilting, super-friendly sweethearts who just don't appreciate the finer points of a good win.

They don't exist—or if they do, I haven't found any. What I have found is Don't-Care Bears.

Don't-Care Bears are usually people with a competitiveness score somewhere in the middle of the spectrum. They are neither strongly competitive nor strongly cooperative, so they'll be happy to take competition if it's there (and enjoy it), but they just don't seek it out as one of their primary sources of satisfaction. They may find greater satisfaction in other kinds of things, depending on their other motivation scores. For example, people with a high score in the thrill-seeking facet have reported that they like the "thrill-ride" of a competitive experience, but such a player may be just as happy to get that particular kind satisfaction from a single-player action ride as from winning a competitive multiplayer game.

So how does this revelation affect the way we design games? Well, people who are playing primarily single-victor competitive experiences want to feel like they are kicking ass at other people's expense. People who are playing primarily team-based games want to feel like their personal contribution mattered to other people's victory, regardless of whether they got the highest score on their team.

ORGANIZE YOUR FEATURES ACCORDINGLY.

» Of course, the best games already do this. This dilemma has been the primary constraint in the evolution of the way modern shooters are scored. In their online competitive modes, most modern shooters blend a "PvP" score (kills) with a "TvT" score (match score). Each is exquisitely balanced against the other: Personal kills have features like the "brag tag" and kill streaks to celebrate the solo warrior, yet the kill streak rewards often benefit the team, and the overall game is generally won and lost by some kind of team score. It wasn't always this way. Originally, competitive multiplayer had only one mode: deathmatch. The idea of a team deathmatch was exotic until designers experimenting with human satisfaction quickly discovered that it rocked, and each new game has brought better and better ways of balancing these conflicting motivations together in the same game.

This isn't accidental. We wouldn't be such a wildly successful industry if we weren't able to satisfy a broad spectrum of motivations for something as fundamental to playing games as the desire to win. But while we're getting better and better at this every day, we still end up with games that have failed to accomplish this. As amazing as the online PvP play is in DARK SOULS, its team play consists mostly of ganging up on people. It is a satisfying PvP experience, but the way the team mechanic has been implemented puts a team player almost exclusively into the role of Bully #2. Oops.

If you believe that the people who like being nice to each other and cooperating together on tasks (PvE) do not also enjoy soundly thrashing their enemies in combat, you need to stop believing that, and organize your features according to the idea that they do want to compete, but together, against a common foe.

If you have one takeaway from this article, take the PvP / PvE mindset and delete it from your mind. Replace it with this: PvP / Don't-Care Bears / TvT.

JASON VANDENBERGHE *is a creative director at Ubisoft, which he has to admit doesn't exactly suck. You can read his intermittent blog and various scribblings at www.darklorde.com. He can be reached by email at jason.vandenberghe@ubisoft.com.*

GDC ONLINE PREVIEW

CAN'T-MISS SESSIONS AT GDC ONLINE

The stage is set for this year's GDC Online, October 9–11 in Austin, Texas. Want a sneak peek of what you'll see at the show (or what you're missing)? Here is just a small selection of the 100+ sessions that will be available to game industry professionals. Please see www.gdconline.com for more information!

GAME NARRATIVE SUMMIT

A FOUR-HOUR STORY IN 400 SIMPLE STEPS: *FALLOUT* DLC

Chris Avellone (Obsidian Entertainment, Inc.)

» Using FALLOUT: NEW VEGAS DLC DEAD MONEY as a test case, creative director Chris Avellone runs through the process of creating a small RPG game narrative from pitch to completion, detailing all the tasks along the way: pitch docs, narrative standards docs, thematic design, maps and level layouts, bug fixing, and more, until the game story is finally in a player's hands. Focus and questions include specific studio tools and techniques, such as story flowchart creation, building emotional vistas, script proofing, and text editing and lockdown sessions. Participants will come away with a better understanding of specific layouts, formats for conversation editors, localization info, level and character design templates, approaches to character concept creation, research, as well as a "how to" and "what needs to be done" in terms of game narrative for an RPG.

WRITING THE ROMANCE-ABLE NPC: ICING THE CONTENT CAKE

Heidi McDonald (Schell Games)

» McDonald's "ICING" model for writing satisfying NPC romances is based on her survey data from more than 500 game players, in combination with scholarly works in games, writing, and psychology. McDonald calls the model "ICING" because NPC romance is to the single-player RPG what icing is on a cake: a delicious addition. Learn how to make your NPC romance writing more delicious!

THE MUPPETATIONAL GAME WRITING CRITIQUE WORKSHOP

Richard Dansky (Red Storm Entertainment/ Ubisoft)

» Not every writer working in games can count on having other writers or an editor on site to help critique, polish, and edit their work. This workshop is intended to give game writers experience in both giving and receiving professional critiques, and developing the skills to critique both themselves and others. (See GDC session listing for details.)

IMMERSIVE STORYTELLING FOR A MISUNDERSTOOD AUDIENCE

Lisa Brunette (Big Fish Games)

» With more than 1.5 billion downloads to date, Big Fish Games made its mark in the casual space by delivering the right games to an often ignored, often misunderstood segment of gamers: women over 35. Brunette describes how to write game stories for an audience whose tastes are polar opposite to those in hardcore games and how you can transition from writing for a mostly young male audience to writing for women over 35.

DESIGN TRACK

WHAT WOMEN WANT: HOW AND WHY A WOMAN'S TASTES IN GAMES CHANGE AS SHE AGES

Terry Redfield (Real Life Plus)

» Traditional games targeted at men have proven to appeal to a wide variety of ages, and are often grouped by genre. But casual games have proven that what may appeal to a woman in her 20s may change drastically as she enters her 40s. Using the latest research on female hormone and neuroscience studies, this session will examine popular mechanics in the casual game market and examine how and why these appeal to three distinct female age groups: adolescent, middle-aged, and menopausal.

Attendees who are targeting a female demographic for casual games will take away exclusive knowledge about what is happening inside a woman's brain and body as she ages and how this may apply to what she is looking for in an online experience.

BEHIND THE CURTAIN: USING MMO GAME SYSTEMS TO TELL BIOWARE STORIES

Damion Schubert (BioWare)

» BioWare has a unique way of telling (and crafting) stories. So what happens when these concepts are introduced to an MMO? In this talk, Damion Schubert discusses how MMO mechanics both helped and hindered the development process of STAR WARS: THE OLD REPUBLIC and the startling discoveries made along the way.

PSYCHOLOGY VS. STRUCTURE: THE POWER OF NUMBERS IN GAME DESIGN

Dave Mark (Intrinsic Algorithm)

» Numbers, visible or not, are often at the core of game design. They are the expression of the designer's vision of how the world works. Through the selection of numbers such as scores, abilities, damage ranges, and even prices, designers are often crafting what a player perceives, believes, and even feels. This talk will demonstrate what the numbers may be

conveying and explore ways that designers can leverage the psychology of numbers to build more engaging games.

SOCIAL GAMES: THE YEAR IN REVIEW

Steve Meretzky (Playdom) and Dave Rohrl (FunSockets)

» With their exhaustive knowledge and up-to-the-minute analysis of social games, Rohrl and Meretzky distill a year of turbulent progress into a single hour. This talk focuses on the most interesting or significant social games of the past year, the lessons those games teach us, and the trends they portend. Whether you are unfamiliar with social gaming and want to quickly get up to speed on what's happening, or you are deeply immersed in the space, this talk has something for you!

PRODUCTION TRACK

STACKING TALENT: GROWING THE LEAGUE OF LEGENDS TEAM

Travis George (Riot Games)

» It's fairly easy for development teams to get bigger, but it's really hard for development teams to get better at the same time. Travis George talks about how Riot Games has successfully balanced scaling the team while simultaneously leveling up its performance and processes and maintaining its company culture.

WARHAMMER ONLINE: TAKING A TRIPLE-A MMO FREE-TO-PLAY THE OTHER WAY

Carrie Gouskos (BioWare Mythic)

» Plenty of big subscription MMOs have gone free-to-play, but what happens when your game doesn't seem conducive to the model? With WARHAMMER ONLINE: AGE OF RECKONING, the answer was simple: Build a completely free-to-play game (WARHAMMER ONLINE: WRATH OF HEROES) from the ground up, using the best from the full-scale MMO (and applying the lessons learned from making it). Gouskos shares why she thinks there is more than one road for MMOs to go free-to-play.

BUSINESS AND MARKETING TRACK

BET ON IP AND PLATFORM

Kristian Segerstrale (Electronic Arts)

» Success in the video game industry isn't about delivering your game on the right platform; it's about creating entertainment that resonates with consumers so they'll play on any device, in any location. From THE SIMS to FIFA, EA has focused on taking big brands and turning them into their own platforms to deliver a unified experience, while trying to create a direct relationship with the consumer. Digital EVP Kristian Segerstrale will cover how EA has turned the industry's biggest titles into a 365-day-a-year service.

PUBLIC RELATIONS 101: MAKE YOUR OWN MEDIA MAGIC

Valerie Massey (CLARA)

» Small studios often lack the budget for public relations staff, but that doesn't negate the need. For the uninitiated or those who've been burned by bad press in the past, the idea of striking out on your own can be daunting. CLARA's Valerie Massey will introduce some essential tools of the trade that will prepare you for some low-cost but successful do-it-yourself media outreach. Prepare to be prepared and launch your own mini-media blitz with confidence. Attendees should leave the session armed with key information for putting together a basic media kit and contact list, kicking off their PR efforts, and gaining awareness of common mistakes to avoid when dealing with the press.

START-UP SUMMIT

EXITING / CASHING OUT: STRATEGIES TO BE ACQUIRED, MERGE, GO PUBLIC

Jim Charne (Law Offices James Charne), James Niesewand (Illyriad Games), Rob Shillingsburg (Jetbolt Games), Dan Offner (Loeb & Loeb), David Rosenbaum (Law Offices of David S Rosenbaum), David Baszucki (ROBLOX), Bill Graner (Crater House), Don Daglow (Daglow Entertainment), and Gary Gattis (Spacetime Studios)

» Every business needs a goal. The time to start thinking about this is at the formation stage. The time to start planning for it is when you begin to take outside equity financing. How you finance will have a strong influence on how and when you may look to cash out. The panel will discuss expectations of founders, angels, venture, and institutional investors, how those expectations may influence decisions made by the board of directors or managers, and what founders can expect when the time comes for the endgame.

PROGRAMMING TRACK

HTML5 REALITY CHECK

Jiri Kupiainen (Disney Interactive)

» This talk is a hype-free look at HTML5 as a game development platform today. Common pitfalls and ways to avoid them, real-world adoption numbers and estimates based on long-term trends, and silly anecdotes from over two years of building games with only HTML5. If you're considering using HTML5 for your next project, this is the session for you. After this session, you will be better equipped to decide whether HTML5 is the right choice for your next project. And if you decide to go with HTML5, you'll have a good understanding of what the common pitfalls are and how to work around them.

CITYVILLE: LESSONS LEARNED AND TOOLS USED TO RUN A LARGE SOCIAL GAME

Kartik Ayyar (Zynga)

» CITYVILLE at its peak was one of the largest social games of all time as measured by monthly active users. All along, the team made multiple code pushes a day. A little-known fact about CITYVILLE: Many of our smartest engineers don't work on the game itself, and instead work on parts of the product that are invisible to the user. Specifically, a lot of the work behind CITYVILLE has been about just building the right set of tools and improving the performance and server efficiency of the game. This session will go behind the scenes of the tools, performance, infrastructure, and scaling work that has been silently been happening behind the scenes of CITYVILLE, and show what happens behind the curtains to keep the show running.

POP WILL EAT ITSELF

COULD GAMES MIX IN POPULAR MUSIC SAMPLES?

For people of a certain age, the characteristic video game sound is inexorably bound to the echoing arcade caverns of hardware synthesis, in just the same way that using gated reverb on drums recalls Phil Collins's pop music dominance in the '80s. But now game tech is advanced enough that we can include the same music we're used to listening to in our daily lives as part of the games we play. While retro-inspired tunes call to mind a certain era, we can use music from any era to inspire a certain mood. These days, any genre can be turned interactive, and at a certain point it becomes a matter of choice.

INVISIBLE TOUCH

» Is it any wonder that the extreme sounds of early arcade game music have grown up, left the living room, and found their way into popular music? Everything from chart-toppers Ke$ha, Beck, and Nelly to Crystal Castles, Daft Punk, and Owl City have brought the iconic style of vintage game audio to the masses. What was once the exclusive pursuit of composers living on the edge of game hardware technology has now become knowingly referenced and nostalgically mined via chip tunes, trackers, and hardware hacking. These days, it may be a shade easier to cue up a sampler or soft-synth than it was to program a sequence using assembly language back in the day, but the real virtue of classic game music came from the overcharged composition techniques that emerged from the technological constraints of the time. Wildly cycling arpeggios flying off the rails in a syncopated flood of operatic musicality, noise-as-percussion relentlessly driving emotional epiphanies—these techniques are immediately recognizable as coming from a brief moment in time. Regardless of whether you grew up during its initial explosion, the chip tune brings with it the power of the past, in the same way that string instruments can't help but carry classical connotations.

When these techniques and iconic sounds are applied to a game with a retro-leaning art style, the combination can be both futuristic and nostalgic. Polytron's FEZ does this by combining a clear vision of a pixelated utopia with the progressive chip-tune stylings of Rich Vreeland. Vreeland is a graduate of the esteemed Berklee College of Music, and brings an intelligent progressive-rock flavor that, when coupled with old-school technique and tonality, grounds the mind-bending puzzle-game experience with an expansive, retro-futurist soundtrack that seems to have emerged fully formed. The symbiosis comes from finding the right "voice" to support the game's design intentions. Other examples that nail the aesthetic connection include Lifeformed's soundtrack for DUSTFORCE, Anamanaguchi's for SCOTT PILGRIM VS. THE WORLD: THE GAME, and Jim Guthrie's singular SWORD & SWORCERY E.P. space opera.

IN TOO DEEP

» While maintaining a connection to the roots of game sound honors the groundbreaking work of those that came before us, it's all too easy to get wrapped up in the way things were. While every soundtrack's coupling must begin with the game, it needn't follow in the familiar footsteps of past pairings. Take, for example, the "acoustic frontier trip-hop" of BASTION, which swept across the gaming community like a breath of fresh air last year. It was the first game for composer/sound designer Darren Korb, and the aesthetic for the music developed during production. Following the Kid's progress in the game, the soundtrack feels more like a concept album when listened to outside the game due to the way song lyrics are used to convey parts of the story.

BASTION is by no means the first game to cross-pollinate music genres, nor is it the first to weave story concepts into the music, but it is noteworthy for how easily it could rub shoulders with the likes of Ben Harper or the Black Keys. (Imagine it cranking out of speakers at your next party.) This is where things get interesting. Having been involved in a few projects that employ one or many music remixers, it seems there would be no stopping popular music from performing the reverse crossover into game soundtracks. Today the potential to take any music and apply adaptive techniques such as generative, state based, or intermittency is just too great as potential not to explore. The question is whether there are games that would be well served by using existing music as part of a fluid and dynamic gaming experience. Other games that have successfully leveraged this paradigm include Les Claypool and Midlake for the MUSHROOM MEN soundtrack, New Orleans swamp-funk band Galactic/über-drummer Bryan "Brain" Mantia on INFAMOUS 2, SSX, and the NEED FOR SPEED: SHIFT 2 UNLEASHED remixes.

BASTION.

AGAINST ALL ODDS

» Sometimes, you won't know the right musical style for your game until you see them together. Take the standout soundtrack for BOTANICULA, which features Czechoslovakian music group Dva. The game fits into the Amanita Design mold that has been evolving since SAMAROST, and continues the company's tradition of integrating unique and appropriate music with whimsical storytelling. Dva's characteristic Bjork-meets-cocktail-era-Stereolab exotica sound gives BOTANICULA a skittering, organic, and playful score, which complements the game's focus on illuminating the joy of discovery and interaction. This is music that already exists on the fringe of popular culture, but found its perfect match in the independent adventure game. It's inspiring to hear such creative pairings. As the interactive landscape continues to embrace new experiences, it's good to know that music in all forms can exist in the same space. For examples of exotic game music couplings, listen to Play Dead's LIMBO soundtrack from Martin Stig-Andersen, and Mona Mur's "Industrial Terror Ambience" for IO Interactive's KANE AND LYNCH 2.

"Everybody's talkin' 'bout the new sound. Funny, but it's still rock 'n' roll to me" —Billy Joel

DAMIAN KASTBAUER *is a technical sound design time-traveler who has been tortured by a jukebox of '80s songs looping in his head at LostChocolateLab.com and on Twitter @LostLab.*

FREE-TO-PLAY PITFALLS

HOW TO AVOID MESSING UP YOUR FIRST FREE-TO-PLAY GAME

If you've never made a free-to-play game, you can find dozens of articles describing how to do it "right." Most of those articles harp on the same handful of issues: Make sure you're properly employing analytics and A/B testing, do everything you can to maximize your one-day and seven-day retention, and so on. Those issues are important, but in my limited experience, I've observed a whole set of major errors made by developers (including my company, Spry Fox) that rarely get talked about. So let's talk about them.

DON'T ASSUME OTHER GAMES ARE PROFITABLE

» TRIPLE TOWN was Spry Fox's first serious attempt at making a F2P game. We were inspired by the success of BEJEWELED BLITZ, which had rocketed up the charts on Facebook and was supposedly raking in the dough. Except at the time, it really wasn't raking in the dough! In reality, BEJEWELED BLITZ had a very low ARPU that was only offset by an enormous population of players that most games could never hope to match. Had we simply bothered to ask any of our friends at PopCap about BEJEWELED BLITZ, they would have honestly told us the game wasn't performing as well as we believed. But we didn't ask, and so we based our monetization design in large part on faulty assumptions.

I wish we were the only studio making this kind of mistake, but I've met plenty of indies who were in process of building games inspired by Game X, where Game X was something popular but not necessarily profitable. Unfortunately, a game's popularity doesn't necessarily correlate to revenue. If, for example, Apple or Google feature a mobile title a couple of times, that's more than enough to give it a sizable audience—but that doesn't mean you can assume the game is profitable!

DON'T DESIGN YOURSELF INTO A CORNER

As of today, TRIPLE TOWN only has two ways to generate revenue: We sell you turns, and we sell you items that help improve your performance in the game. Some in-game items are only available for cash, and some can be purchased with freely earned currency. Unfortunately for us, it turns out that very few people are willing to spend real money for any of the in-game items in TRIPLE TOWN. More people are willing to spend money for turns (or unlimited turns in the mobile version of the game), but the percentage of paying users is still lower than we expected.

All of that would be okay if we could easily come up with additional things to sell. Unfortunately, because of the nature of the game, we can't. TRIPLE TOWN, as it stands today, is a single-player game with a very simple economy, limited social interactivity, and no meaningful persistence. Individually, each of these things make TRIPLE TOWN harder to monetize effectively; together, they make it nearly impossible.

We've been working on making the game more social, and we'll soon unveil an update that adds meaningful persistence...but these changes have taken a tremendous amount of time and effort, and their payoff is as-yet unproven. Had we started with a more spacious and fertile design, we wouldn't have hit this wall so quickly.

DON'T EXPECT RECOGNITION FOR YOUR RESTRAINT

» We are proud of the fact that we chose to limit how many in-game items a player can purchase during a session of TRIPLE TOWN. We made that decision in part because we wanted it to be clear to everyone that TRIPLE TOWN was a game of skill, not a game you could pay to win. And certainly there have been some people who have recognized this. Unfortunately, countless others have bashed us for being a mini-Zynga and for nickel-and-diming them.

We've unquestionably traded away revenue, but it's unclear what (if anything) we received in return. Most players who hate F2P games still hate what we do in TRIPLE TOWN. Everyone else seems to be okay with the concept of the in-game store, regardless of whether it has limited items. In fact, plenty of players have asked us to remove the store limits because they find them annoying!

In the future, we're going to keep trying to do right by players and keep trying to make games that you can't pay to win. But we won't make the mistake of assuming that we'll be recognized or rewarded for it. Make no mistake: Most people buy things in a game because they really want those things—not because they are interested in rewarding your good behavior as a game designer. The latter is called charity, and hoping for it won't get you very far.

DON'T EXPECT MIRACLES

» Right now, the mobile F2P game space is brutally competitive. Consider this: TRIPLE TOWN was featured three separate times by Apple, received tons of positive press, and was generously promoted by our friends at Halfbrick in their mobile games (thanks guys!). And yet TRIPLE TOWN has never broken into the top 50 free apps on iOS.

This isn't the good old days, when simply being new and noteworthy could drive you into the top 20 all by itself. (If it does, it is because you got very, very lucky.) Cross-promoting with other developers won't get you there. Nor will great press. It takes all of that, simultaneously, and more, whether that's paid user acquisition, driving traffic via a web-based version of your game, or any other promotional strategies you employ.

Some of your competitors in the F2P space are spending hundreds of thousands of dollars over a very short period of time to push their games to the top of the mobile charts. If you want to see your game at the top of the charts, you need to be prepared to push equally hard, or find markets that aren't quite so competitive.

THE LIST GOES ON...

» There are many other common mistakes that we fortunately avoided with TRIPLE TOWN, but that I often observe other developers making. For example: not having consumable items as a source of revenue, excessively relying on a single platform (which is a potentially fatal flaw whether you're making paid games or F2P games), emphasizing aesthetic virtual goods instead of functional virtual goods (for more on this, see my recent GDC lecture: http://gdcvault.com/play/1015659/Realm-of-the-Counter-Intuitive), and so on.

Making a F2P game is difficult! If you've never done it before, there's a very good chance you'll blow your first attempt. Take the time to talk to folks who have bitten the dust before you. Take advantage of the many online resources available to you. And most of all, make sure you've given yourself plenty of time to experiment and to fail gracefully! Even the best of us need that.

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.*

SOUVENIR

WWW.SOUVENIRGAME.COM

YOU PROBABLY REMEMBER THE DAY YOU MOVED YOUR STUFF OUT OF YOUR PARENTS' HOUSE, SORTING THROUGH OLD GAMES AND KNICKKNACKS AND RELIVING THEIR ASSOCIATED MEMORIES. THROW IN A RATHER FLEXIBLE FORM OF GRAVITY IN AN M.C. ESCHER-INSPIRED WORLD, AND YOU'VE GOT SOUVENIR, AN EXPERIMENTAL NARRATIVE GAME PRODUCED AS PART OF A THESIS PROJECT FOR A DESIGN AND TECHNOLOGY MFA AT PARSONS.

***Patrick Miller:* How'd you come up with the idea for SOUVENIR?**
Ben Norskov: We intended to create a game that would inspire some awe in the player and have strong narrative elements. Storytelling in games is one of the most unexplored areas of game development, so we wanted to tell a compelling story. We knew that the gravity-shifting mechanic would disorient players and put them in a magical or dreamlike space, where the possibilities seem endless. We also needed a mechanic that would be fun, because we knew that simply walking around could get tedious. We originally planned on more of a hard narrative, but realized through early prototypes that it wouldn't fit well with the game.

***PM:* Tell me about the team. Who did what?**
BN: Mohini, Robert, and I all graduated from Parsons MFA in the Design and Technology program. Our animator and concept artist, Shin Huang, was visiting for a year from China, and Alejandro Ghersi is an incredible DJ and sound engineer I had the pleasure of working with once before.

***PM:* Whose story are you telling in SOUVENIR?**
Mohini Dutta: The protagonist in SOUVENIR is a young woman about to leave home to begin the next chapter of her life. However, transitions are never without baggage, and the game is a representation of all the things that tie her to her old life: her high school, her family, her relationship to religion. To truly begin anew, she needs to look back and choose what to take with her into the future and what to leave behind.

When I left home after my undergraduate degree to work and move out of my parents' home, I was amazed at how much junk I had collected over the years. Each little souvenir—an old book, a broken CD player, some old tapes—each of them was attached to a memory or an event, and I didn't want to let go of any of it, but I couldn't move out if I took all of it with me. The experience of sorting out all my things to pick what to save and what to let go was very rewarding for me, making me finally deal with a lot of loose ends, and eventually allowing me to move on with confidence and space in my life for the new souvenirs to come.

***PM:* What did you draw inspiration from while making SOUVENIR?**
***BN*:** Our first and strongest influence is Terry Cavanagh's VVVVVV, which is simply a stunning game. We were searching for a mechanic, and Robert had suggested VVVVVV's mechanic in 3D, but then rejected it quickly for some reason. Then he showed up after a weekend with a basic prototype of the mechanic and it felt really awesome. Other game influences are PSYCHONAUTS, PORTAL (for level design), and PROTEUS. Art influences are Remedios Varo, Giorgio de Chirico, and many other surrealist architectures, both physical and painted.

***PM:* The Escher-inspired level design is really neat. Did you have any problems modeling such an abstract environment and turning it into a playable game space?**
Robert Yang: We had millions of problems. We kept iterating the player physics—trying to make it smoother, adjusting friction and drag, changing controls—which would make movement feel better, but would alter the relationship to the environment. We couldn't just say, "These are the player physics, now they're done," because game design can't really work like that; you don't know whether something works until it's done. It was a chicken-and-egg problem, like designing platformer levels when you don't know how high or far the player can jump. The only way through is to just accept how much waste you'll have to throw away.

***PM:* At the moment, the game is still incomplete. Are you planning on developing any of the concepts in SOUVENIR further?**
RY: We want to add an ending of some sort. We've had conversations about what would be appropriate, but it's hard to go back to this project because we're all kind of moving on to our own things—we graduated, so why go back to this? Emotionally, we have very different mind–sets now. So we want to, but I'm not sure if we will. I imagine every student project encounters this shift.

***PM:* What's with the crows?**
MD: The crows have been a crowd favorite throughout our development! One of our older prototypes had a stronger boss-fight element in it, and we had devised a few anthropomorphic animal bosses that represented oppressive characters from the protagonist's past. Eventually our game design evolved out of that phase, but the crow still remained.

The crow represented a bastardization of the wise raven character trope from myths; a crow is almost a raven, but represents all the base and vile elements of its character. I imagined the mean teacher character to be a crow to the general ravenness of teachers who represent wisdom and the bigger things in life. The crow began to represent the passive-aggressive antagonists who don't do anything overtly terrible, but cause damage and fear nonetheless.

Release Date: May 2012 for proof of mechanic and narrative prototype
Development Time: 1 year
Development Budget: <$1,000 (not counting the price of Parsons' tuition)
# of lines of code in the game: about 4k–5k
Fun Fact: A 12-year-old playtester told us that "You know you're inside a girl's mind because everything's all messed up."

ADVERTISER INDEX

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: $59.95; all other countries: $69.95 (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.

THE GAMEMASONS

THE SECRET ORDER OF GAME DEVELOPERS WELCOMES YOU

Who posts in forums at night about how other game developers don't understand anything? Who knows the real story behind the publisher drama a few years back that surrounded that WiiWare title that most people forgot exists? Who represents leading lights of the video game developer community, collected together in a rarefied fraternal atmosphere that excludes the chaff of game journalists, managers, entry-level employees, and anybody we don't like?

We do.

/// Welcome to The Gamemasons, acolyte. We're a select group of game industry elites who have banded together in order to hobnob with each other in clandestine, highly secretive events, sharing gossip and laughing over the finest caviar and contraband whale sashimi hors d'oeuvres. We also have an online forum!

Most don't know that we exist. Those who do live in the quivering hope that we'll take notice of them and bestow upon them membership in this hallowed group. It's not that easy, of course—to be considered for our organization in the first place, one must first be known to me, the Grand Admin, as a cool person who'll fit in here (or else you can be famous).

That's why you, newly anointed with the sacred "user" and "password" that grants you access to our storied and long-running phpBB installation, must take our vows as seriously as your life. And you must do more than simply safeguard the knowledge of our Great Board from any and all who would seek to escape with our precious secrets. You must also work constantly to maintain the trust that binds you to our ranks.

Understand that any violation, no matter how slight, can result in a little thing I like to call "excommunication." Don't let the religious overtones of that term scare you, of course. All I would really do is permanently deny you access to the True Knowledge and forever shut you away from bathing in the illuminating light of the Chosen Few ever again.

THE RULES

» Play by my rules, acolyte, and I think you'll come to like the life here.

The first ironclad rule of the Gamemasons is you don't talk about the Gamemasons. Never at all to anyone. It's true that sometimes we have looked past a few exceptions—for example, after members have had a couple of drinks at the hotel bar at an industry conference, or if they wanted to seem "in" to their co-workers at lunch, or if their parents asked them what they've accomplished lately.

But generally, please take this rule very seriously. The last thing we need is anyone with any relation to the sniveling rumormongers of the press finding out about our secret forum for information exchange! Remember, we share highly privileged, high-level discussion, and you are not to repeat what you see or read here to anyone, under the punishment of excommunication.

(Of course, this rule does not apply to me, the Grand Admin. I'll occasionally publicly broadcast the information I hear from acolytes such as yourself!)

How do you recruit new members if you can't talk about us, you ask? Well, if you know a supercool, elite, famous game developer person, you can formally "propose" that person to me by filling out this form and asking the applicant to perform the Ritual of Obeisance. And I'll decide if I feel like accepting the application. Also, be sure to tell that person not to get his or her hopes up. Membership feels more elite if you make applicants wait. We're like that hip downtown nightclub with the really long line outside...except inside our club, we party it up and talk about games!

Anyway, the second rule of the Gamemasons is that all members must contribute to the discussion. There is no freeloading here. Gossip about where you work. Give us secret info and insider stories. Tell us about the time a guy who worked on a major FPS did that crazy thing. Did you get screwed by your publisher? Tell us.

Questioning other studios' strategies is also a great way to contribute to the discussion. Go on and be snarky about games you don't like or other popular things you "never understood." Dis and stomp on famous game industry people that aren't here, too! Remember, the press isn't invited, so you can be as petty, snarky, small-minded, and spittle-flingy as you want!

THE INNER CIRCLE

» Now, I hereby bestow upon you the Badge of Membership. Look at that cool-looking pin with the picture of Vishnu on it. People might think you're in a secret religious cult wearing that. And they'd be right! Co-opting another culture's religious imagery for our group is just the kind of bad-ass behavior you should expect from us. So go on and affix that to your collar as soon as you can—you'll need it to get into the party later tonight.

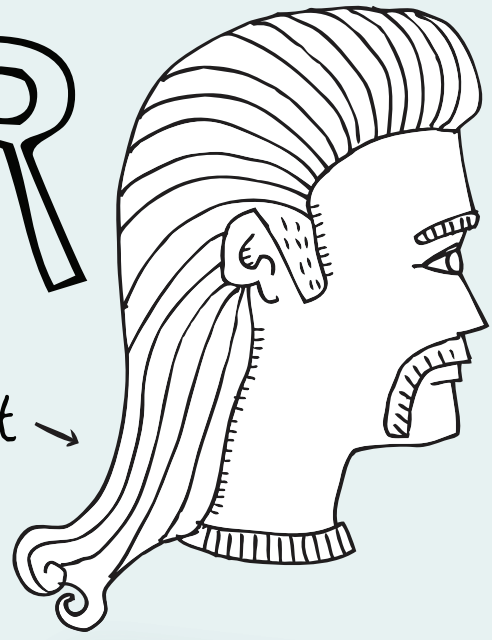
What's that? You don't like this? What's the matter with you? Look, it's just a fun little thing. If you don't like it, you can leave. Why are you taking this all so seriously, anyway? Wow...what a jerk.

MATTHEW WASTELAND *writes about games and game development on his blog, Magical Wasteland (www.magicalwasteland.com). email him at* ***mwasteland@gdmag.com.***

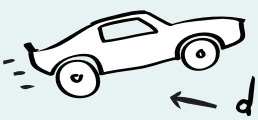
ILLUSTRATION BY JUAN RAMIREZ

THERE ARE LOTS OF WAYS TO BE AN EVEN

COOLER GAME DEVELOPER



like grow a mullet →



← drive a sweet old Camaro

or you can use **BINK VIDEO**.

You get an amazing, super =FAST video and audio codec - all in a simple, clean API. And Bink Video runs on **EVERY PLATFORM**!

BINK VIDEO

USING BINK VIDEO NOT ONLY MAKES YOU cooler, IT MAKES YOU rad!

RAD GAME TOOLS ©

www.radgametools.com
(425) 893-4300