

# Technical Artist Boot Camp



## Introduction

**Adam Pletcher**

Technical Art Director  
Volition, Inc.

[adam@volition-inc.com](mailto:adam@volition-inc.com)

[adam.pletcher@gmail.com](mailto:adam.pletcher@gmail.com)



GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-8, 2012  
EXPO DATES: MARCH 7-9

**2012**

Hi everyone, welcome to the Technical Artists Boot Camp at GDC 2012. I'm Adam Pletcher, Technical Art Director at Volition. You've all seen the schedule, we have an awesome day lined up. Our speakers should have time to do individual Q&A after each talk, and we plan to have a panel-style Q&A session with all the speakers at the end of the day. That was popular last year.



Raise your hand if you were here last year. Another question, how many of you feel like Tech Artists are fully understood and utilized by your studio?

I'm a Technical Artist and I'm a misfit. Nobody knows what to do with me. My project managers don't know how to schedule my tasks. The programmers don't understand what I do, and the artists really really don't understand what I do. I feel like all I do is translate error messages for people. Do those statements ring true to you? If you're at a larger or more established studio it may not, and you should count yourself lucky. But I bet it strikes a chord with anyone at smaller studios.

In 2004, I was working on Volition's first Saints Row title. This was a seriously difficult time of growth at our studio. Our first open world game, on new console hardware, lots of new hires and positions not just being filled but being crafted from thin air. Positions we didn't know existed before.



I was lucky enough to have the title of Tech Artist at this point, along with a handful of others. We were just turning a corner as a discipline, however, learning new crucial skills and finding new ways to be useful. Tech Art was no longer the “3ds Max workaround department”. We were, without fully realizing it, carving out a new identity as a discipline.

It was in that dark middle age when I had a conversation with one of our programmers. He’d recently been hired from a smaller studio, and was working on our first pipeline for getting vehicles into the game. With a harried look, he asked me how to get started with MaxScript. I told him to just send me a list of parameters his exporter required, and I’d take care of all the UI details and make sure the backend received the values it needed. I’ll never forget the look on his face, it was as if the clouds in his world had parted and the sun just came out. “You can do that?” he asked. I put my hand on his shoulder and said “yes I can”. We both shed a few tears. Then we hugged awhile.

Fast forward a few years and our Tech Artists were transformed into a crucial part of the development teams, not only designing but actually implementing in the core of every content pipeline at the studio.

Many of you might be wondering “how can I do that?” I know I can do more important things for my projects, but that’s not my studio’s culture. My advice is simple: Show them what a TA can do. Tech Artists have a unique view into the two major worlds of game development, and nobody is better equipped to bring change than you. Find the slowest, most hated tool or pipeline at your studio, carve out some time, learn what you need to and create something better. Nothing sparks revolution faster than a working prototype. Show it to your artists, have them show it to the art directors. If you’ve made their lives easier, you’ve already won the battle.

Our Tutorial today is all about learning crucial skills TAs can use to pull this off. Experienced industry TAs will cover a variety of techniques and approaches necessary to increase Tech Art involvement on your teams, and give it a solid foundation going forward.

# You Have to Start Somewhere...

Defining the Tech Art Role and Building Their Team

**Arthur Shek**

Technical Art Director

Microsoft Studios | Turn 10



GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-8, 2012  
EXPO DATES: MARCH 7-9

**2012**



GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-8, 2012  
EXPO DATES: MARCH 7-8

2012

FORZA MOTORSPORT 4

# tech art bootcamp

arthur shek, technical art director (microsoft studios)





GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

One day around two years ago, I was sitting in a sweet job at with over 10 years of experience at Disney Animation when I decided I needed a change from working with this gang of characters. 2 months later I was the Technical Art Director at Microsoft Studios working with this very different crew...



It was a big shock to the system.





GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

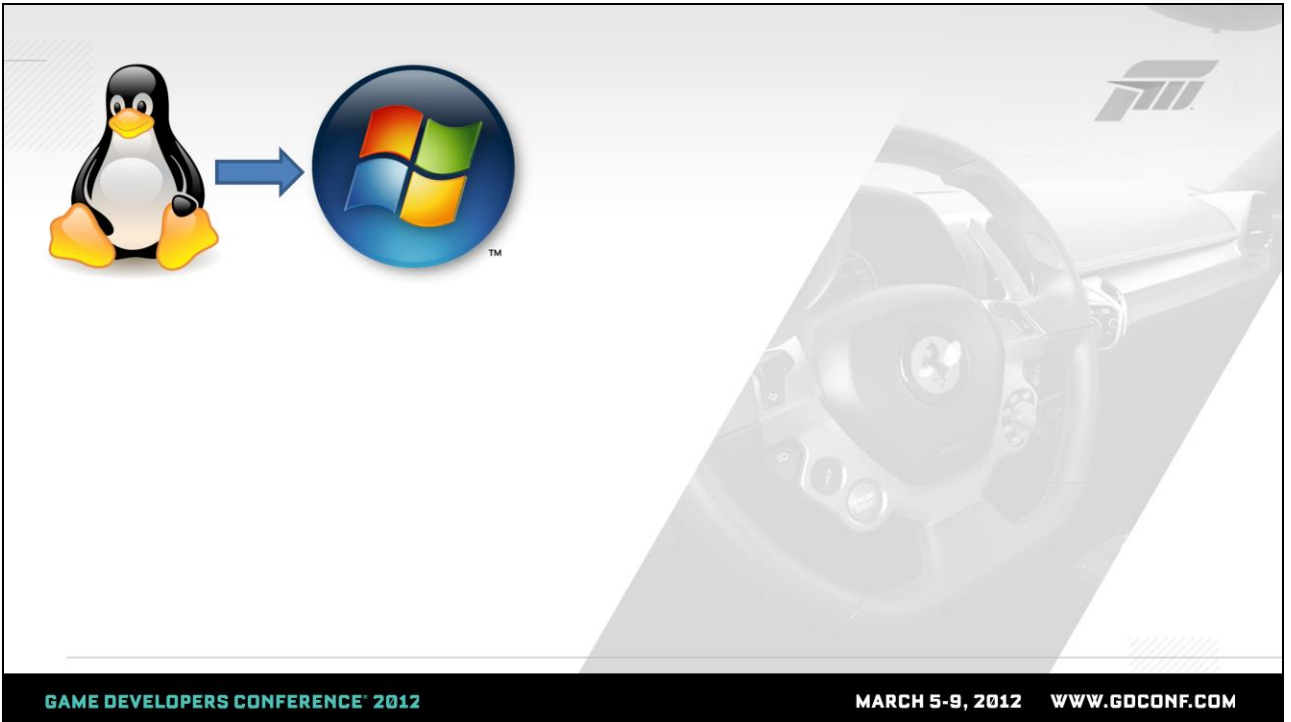
Here's my happy kids before the move enjoying the hot LA sun.



GAME DEVELOPERS CONFERENCE 2012

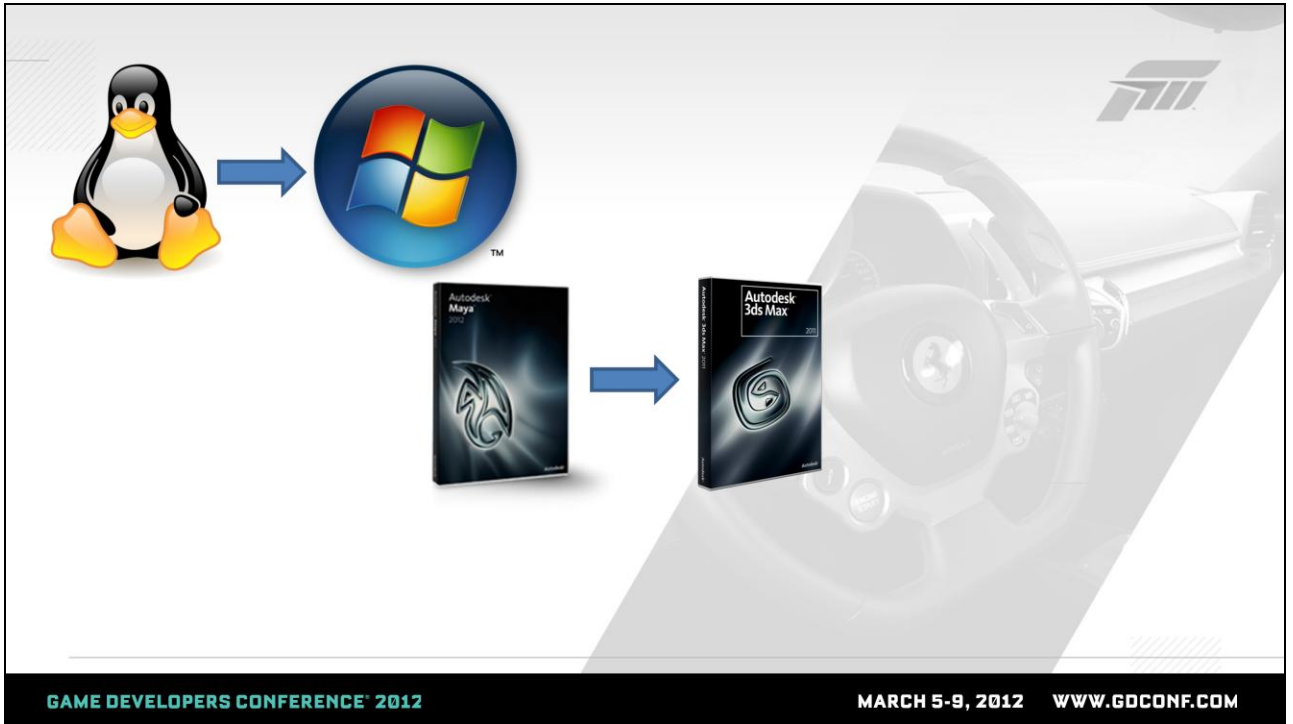
MARCH 5-9, 2012 WWW.GDCONF.COM

Here's the poor miserable kids several months later disgusted by the Seattle weather.

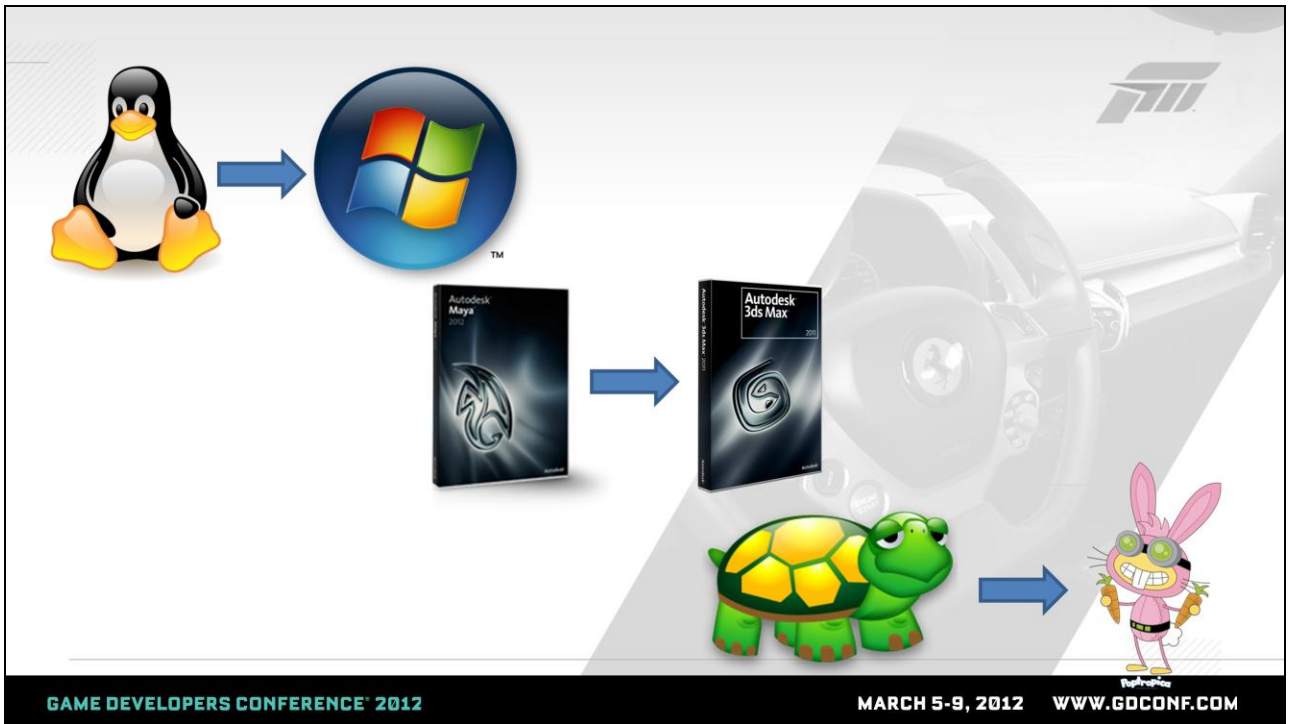


Aside from drastic weather changes, I had to make the big move from Linux to Windows...





Maya to Max...



And go from luxurious 24 hr renders to thinking about how to render every frame in 16.67 ms...



- part I* building out a tech art team
- part II* some technical skills

The remainder of this talk is going to be in two parts. First, I'll quickly talk about some of my experiences in my role, and then I'll drop some technical material on you.

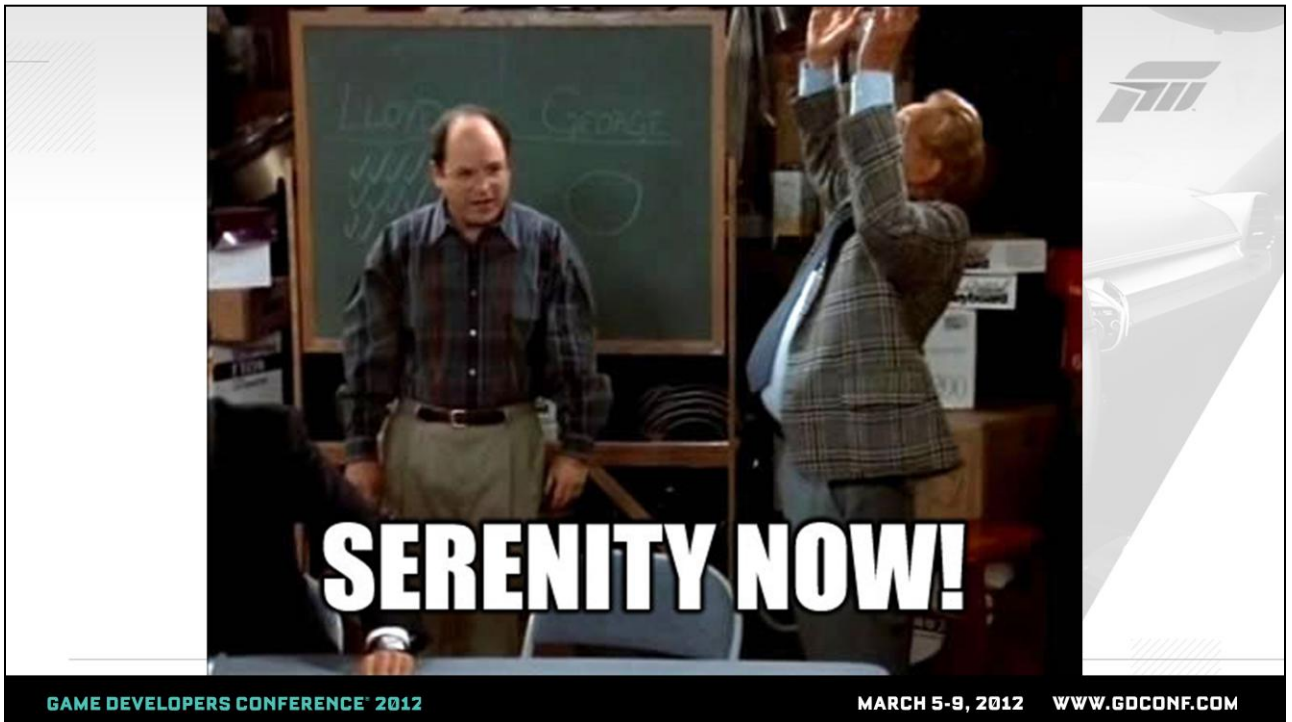


Shutterstock / Shutterstock Life @ Flickr / licensed under Creative Commons

**GAME DEVELOPERS CONFERENCE 2012**

**MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)**

To start in my new role, I sat down and created enormous lists of the tools and new topics I wanted to master in order to succeed at my new role – hardware shaders, DirectX, 3DS Max, Zbrush, real-time workflows, 60 fps rendering techniques. I suppressed an initial instinct to cry – it was overwhelming how much I “needed” to learn. In those early days I sat down and read through the ShaderX and Real-time Rendering books and hated all those guys for being too smart. I flipped through 3D portfolios on conceptart.org and cursed many people.



In a calm moment, I realized I needed to settle down, relax. I was hired by Microsoft to define what a tech artist is at Turn 10, build a team, figure out how our artists want to produce art, and help out with technically challenging problems. It was the same thing I did at Disney, with a different set of problems and inputs. There was no need to freak out.

Adam asked us to spend some time focusing on tech artists who are a rare breed at their studios because they work at smaller places or because their studio hasn't developed a widespread appetite for tech artist work. I wanted to spend a few slides describing how we built up that culture at Turn 10, because we certainly fit that mold just a little bit ago.





Apple.com / Apple.com @ 11/11 / licensed under Creative Commons

**GAME DEVELOPERS CONFERENCE 2012**

**MARCH 5-9, 2012 WWW.GDCONF.COM**

The role of tech artist has no standard definition – in this audience, there are tech artists who would identify themselves as riggers, lighters, VFX artists, animators, or tools programmers. Every studio has different constraints and problems that need to be addressed by artists who can program and programmers who can create art.



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

How do tech artists make an impact? Everywhere I've been, the successful tech art team makes useful stuff for artists over and over and learn as they go. The unifying talent is the ability to present simple and intuitive interfaces to complexity.



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

The first thing on my to-do list was to define what we needed the Tech Art role to be at Turn 10. Across the board, I find that Tech Artists with longevity are the ones who understand the fundamentals of art, process/workflow, programming fundamentals, and how to put any and all these pieces together to solve problems. Tech artists look at artists with hot laptops stuck in their pants and say "no, that is not OK. We can give you an iPod".

Because we make a racing game at Turn 10, we didn't have a vast need for TA's with rigging skills. I spent the first several weeks learning about how artists author cars and tracks and this picture you see on the screen really rang true. Our artists had "settled" for a clunky workflow because of a combination of not having a voice that prioritized their needs and sometimes just not knowing that there were better ways out there. Above all, we needed a team that could serve both those needs.





GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

When I joined Turn 10, there was an issue I've seen at many studios where the dice were loaded so that all developer resources were going to the game. Occasionally, a programmer would pop his head up from the game muck and create an artist tool and be hailed as the hero of the art team. It's easy to see why this is. For our racing game, it's a simple decision when someone says "listen, we can either make the car wheels spin, or the artists can have a fancy tool that shows heatmaps for textures that exceed a certain resolution – choose one".



In building the Tech Art team at Turn 10, I found that in the battle between game features and artist tools, there is only sadness to be had. It's tough to prioritize the two in the same bucket.

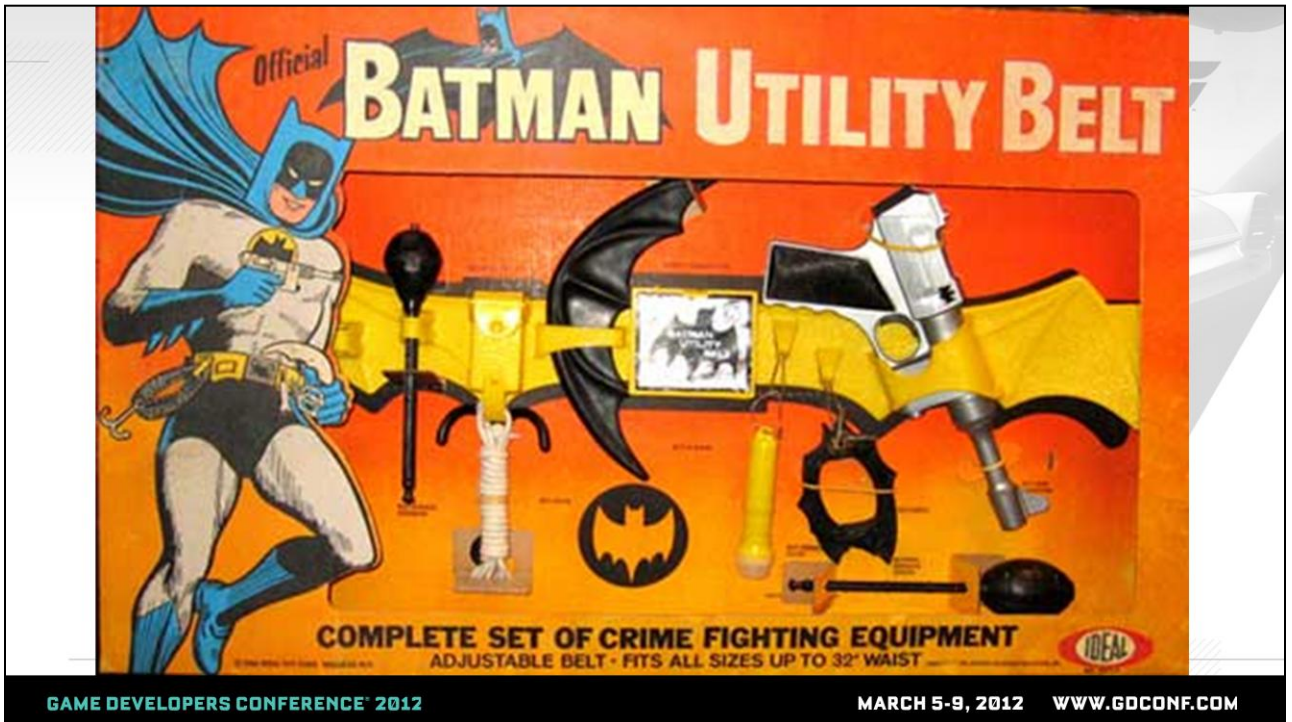


GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

Instead, it is much easier to justify the need for tech artists when you present the fact that you can spend two weeks writing a tool that would save a man year's worth of time over the course of a project. Splitting up the problem by looking at separate cost savings helped justification of my team vs. arguing in the apples to oranges comparison of game features to artist tools. It also helped that allocating some resources to the Tech Art team enabled our studio to take artist support and various existing art problems off developer plates.

In the end, presenting this information to our studio management, we have a Tech Art team that's made up of myself, 2 traditional tech artists and 2 tools developers. I'm sure there will be evolution of this balance, but the important thing is that looking at things with this lens helped us establish a tech art group.

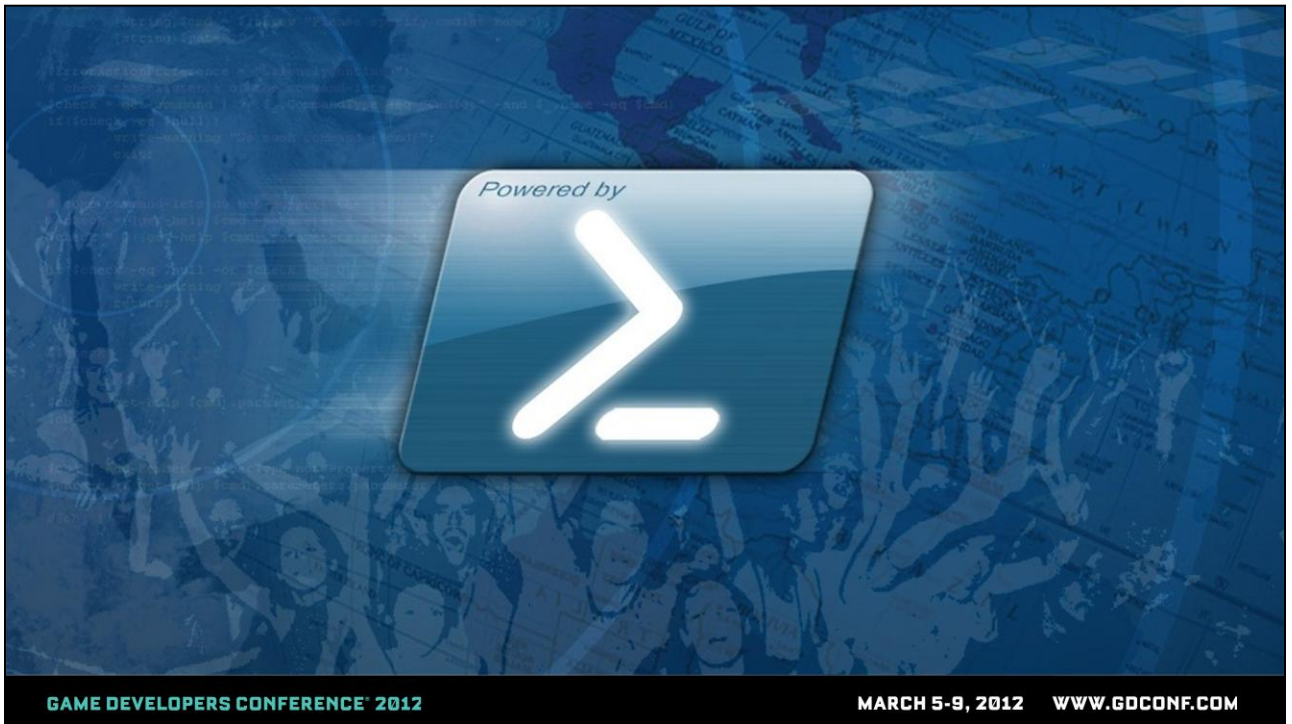


In thinking about what I'd talk about for a more technical contribution, I thought about the diverse mix of skillsets I knew would be represented here. What can a tech artist who calls himself a rigger have in common with a VFX artist? As I mentioned before and you'll hear throughout the day, the common thread binding us all is our ability to solve a wide range of problems with technical fundamentals. For me, having been involved in numerous portfolio reviews for riggers and other TD roles at Disney, what differentiates the great from the mediocre is the demonstrated ability to dive into at least basic programming to enhance your skillset or productivity.

In our industry we have mediocre Batman's and awesome Batman's. You become an awesome Batman by continually adding tricks and new weapons to your toolbelt. In the spirit of my topic, you have to "start somewhere". For the remainder of my section, I wanted to demonstrate how a small investment in basic scripting can make you a more powerful tech artist, regardless of what you consider your primary focus. These fundamental skills are core to the

functionality of my team.



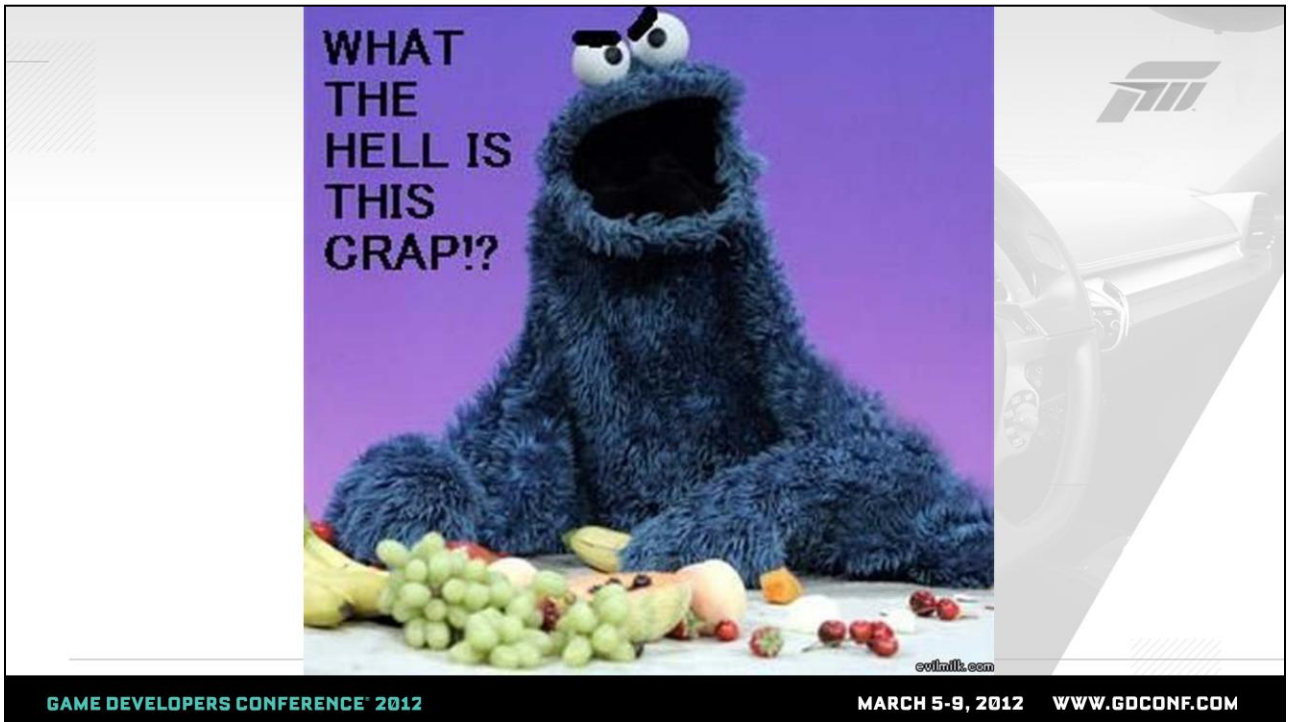


I'm going to walk through some quick topics using a scripting language built into the Windows OS called Powershell.



Before there is a riot, I wanted to point out that I love Python a lot, as evidenced by this picture of me with my shirt off. Python is definitely shaping up to be the universal language of Tech Art. So, why Powershell today?

It is no bootcamp without some pain and misery, and this way the Python experts in the room can't scoff at my pathetic Python "hello world" snippets. In reality, I'm going to give some examples in Powershell because when I first moved to Microsoft and prepared to dazzle the art team with some amazing Python scripting skillz, I found to my dismay that Python wasn't rolled out as a common install. On top of that, the developers on the Pipeline team strongly "encouraged" me to use Powershell.

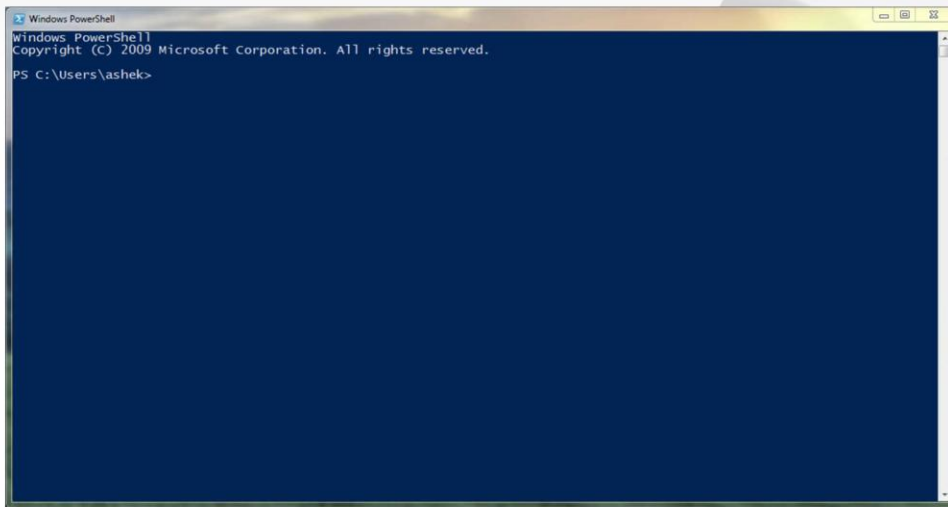


When I was first directed to use Powershell, my first instinct was this. After some calming down, I figured that since many pieces of our pipeline were written in Powershell, I better at least learn enough to tell the Powershell crowd to stuff it. So, what is Powershell? Powershell is a scripting language built into Microsoft Windows. I'd literally never heard of it before starting at Microsoft. I don't know why it's not publicized more – I've actually found it to be a viable substitute for Python for those working in an all Windows environment. As with anything, there are pros and cons involved.



## GDC TECH ART BOOTCAMP

why powershell sucks



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

To talk about some of the strengths of Powershell, I'm going to start by telling you why it really sucks.

This video shows the first user experience to Powershell. The normal use case is to run a PowerShell window by searching for PowerShell in the Start menu. Like Python, you can run commands interactively in a shell, which works fine. For more complex scripts, you can save files with the extension .ps1 and run them. In this video, I'm so excited I can list files in the shell. Look, I found a Powershell script, so let me see what the script does... it looks amazingly useful. Now, try to run it. OK, I get this error. It tells me to type something to get more info, so I do. OK, I'm not reading whatever it just spewed, and... I'm out. That is a tremendously horrible initial experience to a scripting language that you actually want someone to use.

I'm going to save you all time and tell you what that long message just said. It turns out Powershell is SO powerful that you can pretty much write a script that mucks with the OS so

much that you've destroyed someone's computer. So, the Microsoft team added the ability to digitally sign scripts so that users by default can only run signed scripts. To get folks to run scripts, you can read all that and figure out how to digitally sign your scripts. But, I'm a TA and therefore lazy, so I skip all that signing junk and do the easy thing... I can turn off signing for local scripts by running a command buried deep inside that help message one time, and I'm good for that machine's life.



To allow running local scripts, run one time (as Administrator):

```
Set-ExecutionPolicy RemoteSigned
```

or wrap your PowerShell with a batch script like so:

```
powershell.exe -ExecutionPolicy RemoteSigned -File .\test.ps1 %*
```

Here's the fancy command – run this once from the Powershell shell (you may have to run this with Administrator privileges by right-clicking on the Powershell entry in the Start menu and selecting “Run as Administrator”). As you can imagine, it is terribly inconvenient to send someone a script and have to give them these painful instructions the first time someone runs a script.

Luckily, to get around this for artists, you can also deploy your Powershell scripts with a batch script wrapper that disables the harsh execution policy just for that one script. This way, individual artists never have to open up the PowerShell window and step through anything lame.

As I mentioned before, this is a terrible way to expose someone to a scripting language. Once you get over crying about this inconvenience, let me introduce you to the next painful thing I found.



## Non standard

```
print write-output "hello"  
if ($value > -gt 4)
```

Every command is verbose in Powershell and seemingly contains a '-' character. Instead of using the universal "print" statement to write something to the shell, the official command to print lines is "Write-Output", which really sucks. Luckily, Powershell has built-in aliases so instead of using Write-Output, I can use "write" or "echo". If you use "print", it actually tries to print stuff to the printer, which puts you out of the habit of typing "print" really quickly.

Instead of using the '>', '<' and '!= ' conventions for comparing values, you actually have to use '-gt', '-lt', and '-ne'. Actually forgetting and using the '>' sign actually doesn't fail, but instead writes out the left hand side to a file named whatever is on the right side. So, in the example above, the value of the \$value variable actually gets written to a file named '4' without complaining. As a result, I have many files named random integers all over my computer from when I was learning Powershell.

Over and over again, I had to consult documentation for

seemingly easy things, but I guess that's pretty typical when first picking something up.

OK, that's pretty much the big warts of Powershell. There is a third maddening issue that has caused me to lose my mind several times, but I'll point that out in the examples. From this point on, hopefully you'll see why these warts are worth it when all is said and done.



- **Built-in to Windows** – no worries about custom environment management and module deployment
- **Integration into OS and .NET framework** – no modules to juggle to do regular pipeline and tech art tasks
- **Pipes** – shorthand for passing results of commands directly to the next command
- **Object-oriented**
- **Try/catch exceptions**

The subject of why Powershell rocks is the topic of the rest of my talk, but here's a high level view of why Powershell is actually very useful:

- Built into Windows 7 and above – no worry about environment management and deployment like you have to with Python, i.e. what version, do you have the right modules installed, etc.
- Complete first class integration into Windows OS and .NET.
  - no modules to juggle to do regular tech art tasks – shortcuts, permissions, services, file, and plenty more that I'll show
- Pipes: If you've ever used Linux or any UNIX derivative, you get really used to the idea of piping commands together and you miss that when you use almost any other scripting language. Piping commands is the concept of passing the results of one command immediately into another one on the same line to do further processing or filtering. You get that back with Powershell. I'll show some examples.
- Object-oriented: If you don't already know, don't worry about it for this presentation.

- Try/catch for exceptions – robust error handling.

# NINJAS vs. HOMIES

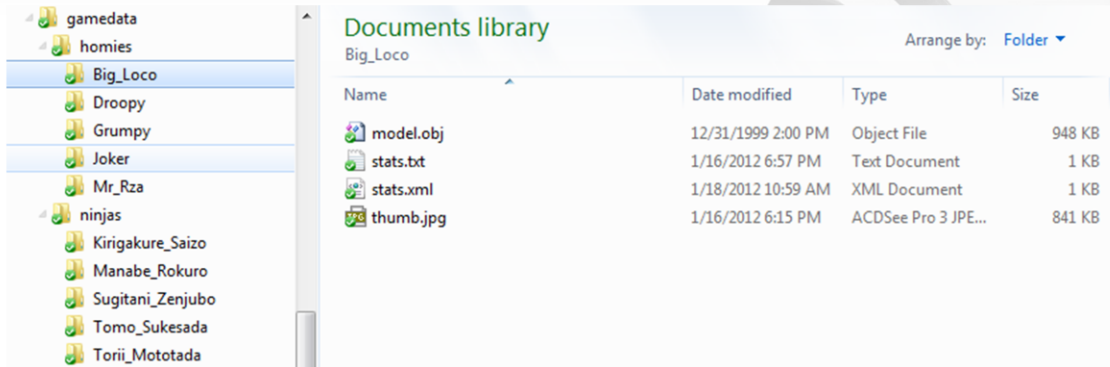


I'm going to walk through some examples of the convenience of Powershell against the backdrop of the greatest theoretical game in the world: Ninjas vs. Homies.



## GDC TECH ART BOOTCAMP

fake game data



For our theoretical game, let's say I've chosen the directory structure indicated above with a root directory and two categories of characters. Each character has their own folder underneath the character type, and each folder contains a stats.txt file which is just a text file containing some character info. Each character folder also contains a stats.xml file (which is an XML file that contains the same info as the .txt file – for later use), a thumb.jpg image of the character and a .obj file representing the character model.

## GDC TECH ART BOOTCAMP

fake game data

```
File Edit Search View Encoding Language Settings Macro Run Plugins
stats.txt
1 color:blue
2 strength:5
3 attack:3
4 defense:2
5 weapon:fist
6 powers:chilling glare,stun flex
```

```
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
stats.xml
1 <character type="homie">
2   <name>Big_Loco</name>
3   <color>blue</color>
4   <stats strength="5" attack="3" defense="2" />
5   <weapon>
6     <weapon name="fist" />
7   </weapon>
8   <powers>
9     <power name="chilling glare" />
10    <power name="stun flex" />
11  </powers>
12 </character>
```

GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

The stats.txt file contains some basic character data in a text format. As our game gets more complex, we might turn this data into an XML file such as you see on the right. I've formatted the XML in a number of different ways for when we get to the XML section.

So with this game data, what are some things I might do with Powershell to flex some Tech Art muscle?



## List directory contents

```
dir gamedata           # returns DirectoryInfo or FileInfo objects
dir -name gamedata      # returns strings
```

## Recursive filter

```
dir gamedata -filter *.jpg -name -recurse
```

## Help

```
help dir
```

Let's start out easy. The first thing you probably want to do in any useful tool is to navigate the file system and see what files exist in a directory. The Powershell command to do this is "get-childitem". Luckily, the developers also aliased this to "dir".

Here, I've listed out some common things we can do with "dir". In its most basic form, you call "dir" on a directory or file and you get a list of items in that directory or the file itself. Without additional flags, this command will return DirectoryInfo or FileInfo objects. I'll get back to this in a moment. What might seem more immediately useful is just to get back a list of strings with the directory contents; you can do that with the -name flag. In the next example, we can filter the returned file names with a cmdline switch that returns a list of files that match the regular expression specifying only files that end with the .jpg extension. I also tossed in the -recurse flag that will walk the entire tree under the specified directory instead of just the files one level deep.

Finally, you can get help on any command by prefixing it with “help”.

## GDC TECH ART BOOTCAMP

paths, files, directories



```
Windows PowerShell
PS F:\My Documents\My Dropbox\Turn10\presentations> dir -name gamedata
homies
ninjas
PS F:\My Documents\My Dropbox\Turn10\presentations> dir gamedata

Directory: F:\My Documents\My Dropbox\Turn10\presentations\gamedata

Mode                LastWriteTime         Length Name
----                -
d-----          1/17/2012   9:08 AM             homies
d-----          1/17/2012   9:08 AM             ninjas

PS F:\My Documents\My Dropbox\Turn10\presentations> dir gamedata -filter *.jpg -name -recurse
homies\Big_Loco\thumb.jpg
homies\Droopy\thumb.jpg
homies\Grumpy\thumb.jpg
homies\Joker\thumb.jpg
homies\Mr_Rza\thumb.jpg
ninjas\Kiripakure_Saizo\thumb.jpg
ninjas\Manabe_Rokuro\thumb.jpg
ninjas\Sugitani_Zenjubo\thumb.jpg
ninjas\Tomo_Sukesada\thumb.jpg
ninjas\Torii_Mototada\thumb.jpg
PS F:\My Documents\My Dropbox\Turn10\presentations>
```

GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

Here's what the results of those commands look like using our data.



## Variables

```
$results = dir gamedata -filter *.jpg -name -recurse # returns array of strings

$results.Count

$results[0]

foreach ($file in $results)
{
    echo $file
    # do something useful
}
```

## Maddening – Powershell tries to be too smart

```
$results = @(dir gamedata -filter *.jpg -name -recurse)
```

We can also stick the results of commands into variables, which are denoted using the \$ convention. Once we have the results, we can get the first item using the common array index notation with square brackets. We can also start doing fancy loops that iterate over the results and do something useful with them.

I wanted to point out the third reason why Powershell sucks. It tries to be too smart. The dir command will always return an array of items *unless the results only have one item*. In that case, it returns a single object. In the top example, \$results is always an array of strings unless the directory only contains one item, in which case \$results will be a string. That really sucks, because very easily, without realizing it, your code might fail. In the first example above, the array operation .Count that returns how many results are in the array would cause your script to fail. You can test the return type in your code, but it makes for ugly code.

In practice, I've found that I have to wrap all my calls that

could return either type in the convention below that forces everything to come back in an array, even if it's only one item.





## Piping

```
dir gamedata -filter *.jpg -recurse | foreach {$_ .Name, " + $_.CreationTime}

dir gamedata -recurse | where {$_ .Length -gt 800KB} | sort-object Length |
    foreach {$_ .FullName, " : " + $_.Length}

dir gamedata | foreach { echo $_.Name; dir $_.PSPath | foreach {"`t" + $_.Name} }

dir gamedata -recurse | where {$_ .LastWriteTime -gt (Get-Date).AddDays(-14)} |
    select-object -Property FullName, LastWriteTime, Length | out-gridview
```

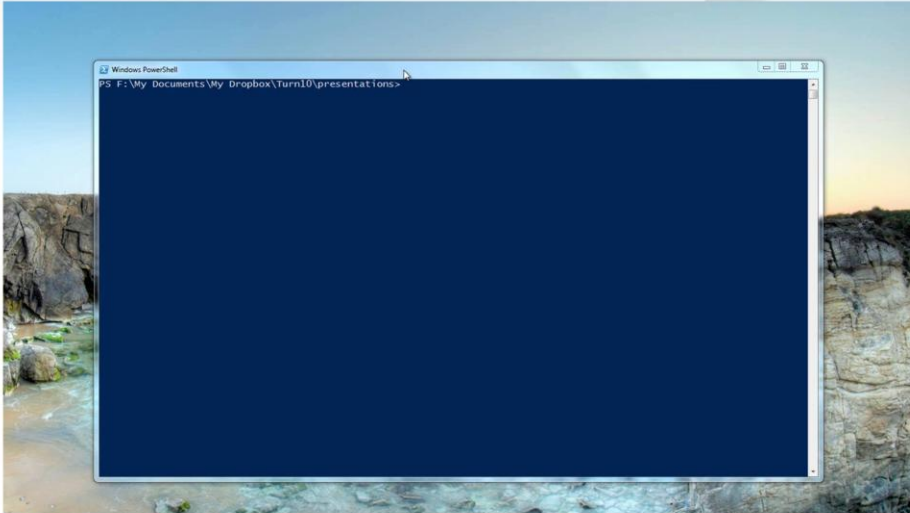
As a final result, I wanted to talk about piping, which I mentioned earlier as something awesome about Powershell. Piping lets you hand the results of a command immediately to the next command.

Here are some examples of varying complexity:

- (1) Handling the results of a file listing to a "foreach" command, which loops through each result and does something to each of them. In this case, printing out the name and creation time of the file. I'll speak a bit later to how we know that each result has a .Name and .CreationTime property.
- (2) Handling the results to a "where" command that filters on file size greater than 800 K, sorts all the objects by file size, and then prints out file names and sizes to the shell.
- (3) The next example gets even more crazy, where I have nested pipe action happening. Here, I get the results of the dir command ("homies" and "ninjas"), and for each of them I print out the name, then get a directory listing for each of

those categories and print out the indented name below. This command will list all the characters in game grouped by their type.

- (4) In the final example, I loop through the entire gamedata directory looking for files that were modified in the past 2 weeks, select a certain number of properties to display, and then call the awesome out-gridview command that lets me visually see any results in a grid table. This one's worth a video.



The nice thing about piping is that we can build up the commands as we go and inspect the results at each stage.

At the last step, I make use of the out-gridview command, which loads results in a visible grid. We can use this command on any custom object with personalized properties, so you can hopefully easily imagine places where you'd write a script that loads some sort of debug data into this grid with no UI code. The grid view is pretty basic, but we can sort by columns, filter to easily narrow down the results graphically, or add custom filter criteria.



## Basic reading/writing files

```
dir gamedata -filter *.txt -recurse | select-string -pattern "color"

get-content gamedata\homies\Grumpy\stats.txt

add-content gamedata\homies\Grumpy\stats.txt "`nmagic:poison"

$test = get-content gamedata\homies\Grumpy\stats.txt
$test += "beltcolor:none"
$test | out-file -encoding ASCII gamedata\homies\Grumpy\stats.txt
```

Of course, we also want the ability to read and write files.

The first example shows how we can search through files looking for some particular text – we don't actually have to open the files.

Next, we can just read the contents of a file, or add a line at the end without cracking open the file. That little backtick n indicates a newline character – in other languages it's typically backslash n.

Finally, we can use the get-content command to return an array of strings (one entry per line). Then we append to that array, and use the out-file command to stream everything back out.

This is basic ascii file manipulation; of course there are many other techniques to crack open files and jump all around them. There are also other techniques to write out more special files, like XML and Excel files that I'll be poking into

later.



## Using file data

```
$matches = dir gamedata -recurse -filter *.txt | select-string "color"
foreach ($m in $matches)
{
    $file = $m.Path
    $color = $m.Line
    $split = $file.split('\')
    $character = $split[-2]
    $split = $color.split(':')
    $color = $split[1]
    $character + ": " + $color
}
```

A lot of the previous examples have relied on a lot of piping with crazy “select” and “foreach” statements. Here’s an example that might be more recognizable to the Python fans in here. This snippet of code reads every string containing “color” from every text file as a “MatchInfo” object, then loops through all of them to print out a nicely formatted list of characters and colors from the stats text files. Like the “dir” command that output FileInfo and DirectoryInfo objects, I’m now talking about MatchInfo objects and using magically conjured attributes like .Path and .Line to get at properties of the object. It’s time to clarify and talk about .NET quickly and how useful a combination that is with Powershell.

## GDC TECH ART BOOTCAMP .NET



<b>* .NET Framework 4</b> <b>.NET Framework Class Library</b> <ul style="list-style-type: none"><li>System</li><li>System.Activities.Namespaces</li><li>System.AddIn.Namespaces</li><li>System.CodeDom.Namespaces</li><li>System.Collections.Namespaces</li><li>System.ComponentModel.Namespaces</li><li>System.Configuration.Namespaces</li><li>System.Data.Namespaces</li><li>System.Deployment.Namespaces</li><li>System.Device.Location</li><li>System.Diagnostics.Namespaces</li><li>System.DirectoryServices.Namespaces</li><li>System.Drawing.Namespaces</li><li>System.Dynamic</li><li>System.EnterpriseServices.Namespaces</li><li>System.Globalization</li><li>System.IdentityModel.Namespaces</li><li>System.IO.Namespaces</li><li>System.Linq.Namespaces</li><li>System.Management.Namespaces</li><li>System.Media</li><li>System.Messaging.Namespaces</li><li>System.Net.Namespaces</li><li>System.Numerics</li><li>System.Printing.Namespaces</li><li>System.Reflection.Namespaces</li><li>System.Resources.Namespaces</li><li>System.Runtime.Namespaces</li><li>System.Security.Namespaces</li><li>System.ServiceModel.Namespaces</li><li>System.ServiceProcess.Namespaces</li><li>System.Speech.Namespaces</li><li>System.Text.Namespaces</li><li>System.Threading.Namespaces</li><li>System.Timers</li><li>System.Transactions.Namespaces</li><li>System.Web.Namespaces</li><li>System.Windows.Namespaces</li><li>System.Workflow.Namespaces</li><li>System.Xml.Namespaces</li><li>System.Xml.Linq.Namespaces</li></ul>	<b>* .NET Framework Class Library</b> <b>System.Windows.Namespaces</b> <ul style="list-style-type: none"><li>System.Windows</li><li>System.Windows.Annotations</li><li>System.Windows.Annotations.Storage</li><li>System.Windows.Automation</li><li>System.Windows.Automation.Peers</li><li>System.Windows.Automation.Provider</li><li>System.Windows.Automation.Test</li><li>System.Windows.Baml2006</li><li>System.Windows.Controls</li><li>System.Windows.Controls.Primitives</li><li>System.Windows.Converters</li><li>System.Windows.Data</li><li>System.Windows.Documents</li><li>System.Windows.Documents.DocumentTitle</li><li>System.Windows.Documents.Serialization</li><li>System.Windows.Forms</li><li>System.Windows.Forms.ComponentModel</li><li>System.Windows.Forms.DataVisualization</li><li>System.Windows.Forms.Design</li><li>System.Windows.Forms.Design.Behavior</li><li>System.Windows.Forms.Integration</li><li>System.Windows.Forms.Layout</li><li>System.Windows.Forms.PropertyGridEditor</li><li>System.Windows.Forms.VisualStyles</li><li>System.Windows.Input</li><li>System.Windows.Input.Manipulations</li><li>System.Windows.Input Stylus Pluggins</li><li>System.Windows.Interop</li><li>System.Windows.Markup</li><li>System.Windows.Markup.Localizer</li><li>System.Windows.Markup.Primitives</li><li>System.Windows.Media</li><li>System.Windows.Media.Animation</li><li>System.Windows.Media.Converters</li><li>System.Windows.Media.Effects</li><li>System.Windows.Media.Imaging</li><li>System.Windows.Media.Media3D</li><li>System.Windows.Media.Media3D.Convert</li><li>System.Windows.Media.Media3D.Format</li><li>System.Windows.Navigation</li><li>System.Windows.Resources</li><li>System.Windows.Shapes</li><li>System.Windows.Shell</li><li>System.Windows.Threading</li><li>System.Windows.Xps</li><li>System.Windows.Xps.Packaging</li><li>System.Windows.Xps.Serialization</li></ul>
---	---

GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

.NET is a software framework on Microsoft Windows. It exposes a core class library to numerous programming languages. If you're using Visual Studio to write tools or apps (basically anything but a rendering engine), chances are you're using the .NET library. In the screenshots from the documentation above, you can see the scope of what this library contains. On the left is a high level view of the different pieces supported, including some I've clicked on to show media support, XML files – there are numerous others including math and drawing. On the right is a dive into just one of those namespaces - the System.Windows namespace, which shows a whole bunch of more useful libraries, including all Windows UI components, 3D support, ink drawing, threading, etc.

The best part of it is that Powershell is one of the supported languages with first class .NET integration. What this means is that with little effort, you can write tools that do really useful stuff.





## Reflection

```
dir gamedata\homies\Grumpy\stats.txt | get-member  
  
dir gamedata | get-member  
  
dir gamedata -filter *.txt | select-string "color" | get-member  
  
"hi" | get-member
```

We started out this shallow dive into Powershell by looking at the "dir" command, which lets you list the contents of a directory. In there, I was able to get some file or directory objects and access some properties on them, like LastModifiedTime. It turns out that the objects returned by the "dir" command are .NET objects that represent files or directories. Using a command "get-member", we're able to inspect those objects and find out more about what we can do with them – this is a concept in .NET known as reflection, where we every object has discoverable methods and properties.

In the snippets above, we can figure out that the commands return a FileInfo, DirectoryInfo, MatchInfo and String objects, respectively. The "get-member" command also lets us discover properties we can access on each of them, as well as functions we can call.

```
PS F:\My Documents\My Dropbox\Turn10\presentations> dir gamedata\homes\Grumpy\stats.txt | get-member

TypeName: System.IO.FileInfo

Name      MemberType Definition
-----
Mode      CodeProperty System.String Mode{get;Mode;}
AppendText Method      System.IO.StreamWriter AppendText()
CopyTo    Method      System.IO.FileInfo CopyTo(string destFileName), System.IO.FileInfo CopyTo(s...
Create    Method      System.IO.FileStream Create()
CreateObjRef Method      System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
CreateText Method      System.IO.StreamWriter CreateText()
Decrypt   Method      System.Void Decrypt()
Delete    Method      System.Void Delete()
Encrypt   Method      System.Void Encrypt()
Equals    Method      bool Equals(System.Object obj)
GetAccessControl Method      System.Security.AccessControl.FileSecurity GetAccessControl(), System.Secur...
GetHashCode Method      int GetHashCode()
GetLifetimeService Method      System.Object GetLifetimeService()
GetObjectData Method      System.Void GetObjectData(System.Runtime.Serialization.SerializationInfo in...
GetType   Method      Type GetType()
InitiaizeLifetimeService Method      System.Void InitiaizeLifetimeService()
MoveTo    Method      System.Void MoveTo(string destFileName)
Open      Method      System.IO.FileStream Open(System.IO.FileMode mode), System.IO.FileStream Op...
OpenRead  Method      System.IO.FileStream OpenRead()
OpenText  Method      System.IO.StreamReader OpenText()
Overwrite Method      System.IO.FileStream Overwrite()
Refresh   Method      System.Void Refresh()
Replace   Method      System.IO.FileInfo Replace(string destinationFileName, string destinationBa...
SetAccessControl Method      System.Void SetAccessControl(System.Security.AccessControl.FileSecurity fil...
ToString  Method      string ToString()
PsrchIdName NoteProperty System.String PsrchIdName=stats.txt
PSDrive   NoteProperty System.Management.Automation.PSDriveInfo PSDrive=F
PSIsContainer NoteProperty System.Boolean PSIsContainer=False
PSParentPath NoteProperty System.String PParentPath=Microsoft.PowerShell.Core\FileSystem::F:\My Docu...
PSPath    NoteProperty System.String PPath=Microsoft.PowerShell.Core\FileSystem::F:\My Documents\...
PSProvider NoteProperty System.Management.Automation.ProviderInfo PSProvider=Microsoft.PowerShell.C...
Attributcs Property      System.IO.FileAttributes Attributcs {get;set;}
CreationTimeUtc Property      System.DateTime CreationTimeUtc {get;set;}
Directory Property      System.IO.DirectoryInfo Directory {get;}
DirectoryName Property      string DirectoryName {get;}
Exists    Property      bool Exists {get;}
Extension Property      string Extension {get;}
FullName  Property      string FullName {get;}
IsReadOnly Property      bool IsReadOnly {get;set;}
LastAccessTime Property      System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc Property      System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime Property      System.DateTime LastWriteTime {get;set;}
```

For example, running it on a output of a "dir" command of a file shows us that the result is of type System.IO.FileInfo, and that we can access Properties of that object like "LastWriteTime" or "CreationTime", as well as call Methods (or functions) on the result, some of which look handy, like "Delete()" or "AppendText()".

We can also use the robust online .NET documentation to learn more about these properties.

### Using included libraries/modules/assemblies

```
Add-Type -AssemblyName System.Drawing

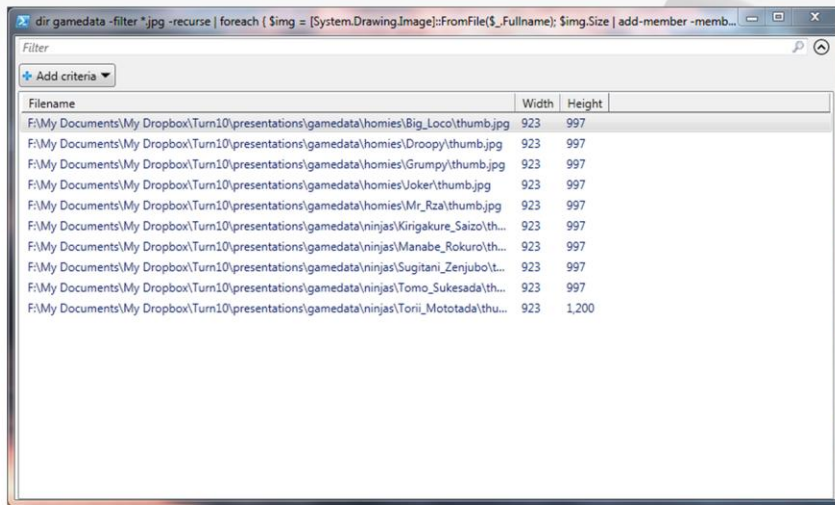
dir gamedata -filter *.jpg -recurse | foreach { $img =
[System.Drawing.Image]::FromFile($_.FullName); $img.Size | add-member -
membertype noteproperty -name Filename -value $_.FullName -passthru} |
select-object Filename, Width, Height | out-gridview
```

.NET includes a ton of useful libraries; here's an example of using the basic image library to get some file size info on the thumbnails we have in our game.

The System.Drawing.Image library isn't loaded by default in Powershell, so the first thing we need to do is call the "add-type" command to load that library, or assembly.

Now we can use it at will. I want to display a grid that has image name, width and height for all character thumbnails. The snippet provided finds all the thumbnails buried in the gamedata folder. For each one it'll create an Image object and get the size property (which includes width and height). The ".Size" property only includes the width and height, but I don't want to lose the filename as I continue passing data down the pipe, so I call the "add-member" command which lets me add arbitrary properties to objects. Here, I add a Filename property with the correct value and specify the -passthru command to indicate that I want to pass this data on down the pipe. If you just run the command up to this point,

you'll notice there's an unnecessary property in there, so I get rid of it by selecting specific properties of Filename, Width and Height. Finally, the whole thing is piped out to a grid view and we get the following...



You can use the grid view to filter and sort as before. Doing this, I see that one of my thumbnails is sized differently.

At Turn 10, all of our automated testing tools for the game, including all button presses and debug menu fiddling, are sent to the Xbox are wrapped in a convenient .NET library. We can write a Powershell script that simply loads our game library and presses any button or sets any debug menu programatically, which is really handy for artist tools.

You can even find .NET libraries written by third parties and download and include those in your scripts if desired, sort of the Python methodology. We recently worked on a tool that needed a robust color picker that also accounted for alpha. If you've worked with the built-in Windows color picker, you know how lame it is. I was able to find a great .NET color picker that someone at Microsoft published open source, and just use the .dll library in my tool without any misery on my end.

## GDC TECH ART BOOTCAMP

### XML

```
stats.txt
1 color:blue
2 strength:5
3 attack:3
4 defense:2
5 weapon:fist
6 powers:chilling glare,stun flex
```

```
stats.xml
1 <character type="human">
2   <name>Big_Loco</name>
3   <color>blue</color>
4   <stats strength="5" attack="3" defense="2" />
5   <weapon>
6     <weapon name="fist" />
7   </weapon>
8   <powers>
9     <power name="chilling glare" />
10    <power name="stun flex" />
11  </powers>
12 </character>
```

GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

For my next example, I wanted to talk about XML. XML is a common way to store game data for larger games. The ascii data on the left is showing the same data as we see in the XML file on the right, so as tech artists we might be tempted to come up with custom file formats like you see on the left. For the love of game development, do not make that mistake! It's OK for prototyping and reading files, but tough for the scale of production.

At some point, people will hand-edit text files, whether ascii .txt or XML. So, the first thing we want to do with our games is have a data validator that makes sure that the format is valid and that someone hasn't hosed an important property. We'll probably also want the ability to add weapons or powers to our characters at some point.

To do this with a custom ASCII format like on the left, you need to write a lot of non-reusable code to make sure that the file has a line that contains "weapon:" and that it has a "powers:" line where expected. If you find the need to change



the file format, it's a huge pain. In contrast, almost all scripting languages have libraries to easily do the same thing with XML files because data is represented in a node object model. With Python and XML, there are a lot of different modules that you can use, so you download a bunch, test them out to find a comfortable style, write code against it and make sure that the module is deployed to all users. With Powershell, you just use the built-in XML objects and there is no additional misery.



## XML validation

```
function validate-XML ([string] $xmlpath)
{
    $xd = new-object System.Xml.XmlDocument
    try
    {
        $xd.load($xmlpath)
        echo "`tYour XML is so beautiful."
    }
    catch [System.Xml.XmlException]
    {
        echo $("`t"+$xmlpath+": Invalid XML ("+$_.Exception.Message+")")
    }
}

dir recurse -filter *.xml | foreach { validate-XML $_.FullName }
```

This is an example showing a Powershell function that validates generic XML and how we might use it. The first thing we want to do with XML files is make sure they formatted correctly.

As you can see, validating our XML data is as easy as trying to load the file. If it succeeds, it'll make it to the success print. Otherwise, it'll throw an XmlException error before getting to the success message and we can print out what the error message is – this will point out the exact line where the format is incorrect.



## Validating/modifying specific XML

```
[System.Xml.XmlDocument] $xd = new-object System.Xml.XmlDocument
$xd.load("stats.xml")
$xd.SelectNodes("/character/powers/power") | foreach { $_.name }
$xd.SelectSingleNode("/character/name") | foreach { $_.InnerText }

$stats = $xd.SelectSingleNode("/character/stats");
$stats.Strength
$stats.SetAttribute("intelligence", "1")

$powers = $xd.SelectSingleNode("/character/powers")
$newpower = $xd.CreateElement("power")
$newpower.SetAttribute("name", "mama")
$powers.AppendChild($newpower) | out-null

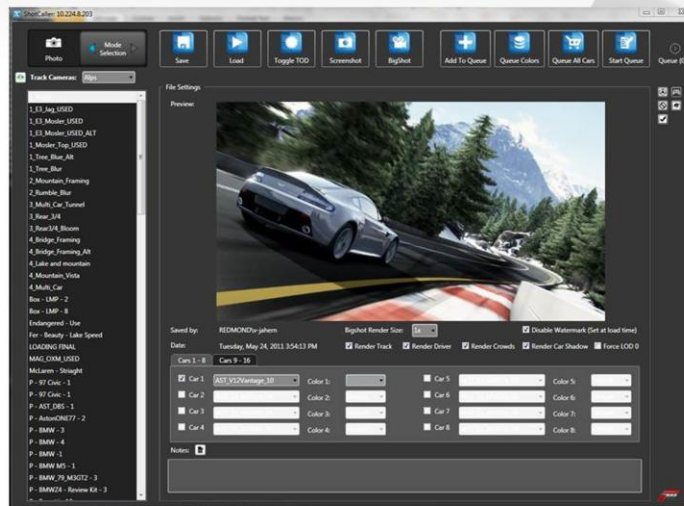
$xd.Save("stats.xml")
```

For further validation of specific XML formats, we can use the `SelectNodes` or `SelectSingleNode` methods for an `XmlDocument` object; if it finds results, you'll get those back, otherwise the result set will be empty.

We can also get specific attributes of a node just by calling the property name, i.e. `.Strength`. We can add attributes, like a new one for intelligence.

We can also create new powers by adding a new "power" node to the `XmlDocument` and parenting it to the "powers" node.

Before we lose the results, we can just save the XML file with the handy "Save" method. As you can see, it's much easier to generically modify XML game data than a specifically formatted ASCII file.



We now know the various fundamentals of pieces required to write some useful tools. When dealing with artists, that counts for exactly nothing unless you can toss a snazzy UI on top of things.

Luckily, again without having to research and purchase UI packages like QT, we have the handy power of WinForms or WPF (the old and new UI tech) that developers use to author Windows UI's available to us in Powershell via .NET.

WPF is a good option because it's the direction that UI programming is headed from a Windows perspective. It has hardware acceleration and really powerful styling and binding capabilities. The UI you see above is a WPF Powershell script that James O'Donnell and I wrote at Turn 10 to deal with media capturing from the Xbox.



To run WPF Powershell scripts, launch them with a batch script:

```
powershell.exe -STA -ExecutionPolicy RemoteSigned -File .\test.ps1 %*
```

Before moving on, I wanted to mention that in the current released version of Powershell (2.0), WPF UI's can only be created in STA mode (single-threaded apartment). This is an inconvenience because it means you can't just run this scripts with powershell.exe; you need to launch powershell with the -STA flag. You can do this by launching a Windows command prompt and running "powershell -STA" before you step through any script lines, or you can wrap your Powershell script with a batch file that runs the script with the STA flag. I will also mention that in the preview of Powershell 3.0, which is publicly available with a quick search, you don't have to worry about this at all so that's how I'm running my demos.

## GDC TECH ART BOOTCAMP

### UI

```
Add-Type -Assembly PresentationFramework
Add-Type -Assembly PresentationCore
Add-Type -Assembly WindowsBase

function typeChanged
{
    $type = $comboBox.SelectedItem
    Write-Host "Ha ha, this should fill the list with '$type' characters, but instead nothing happened!"
}

function launchClicked
{
    $char = $listbox.SelectedItem
    Write-Host "Ha ha, you wanted to launch '$char', but instead nothing happened!"
}

$grid = New-Object Windows.Controls.Grid
$grid.Background = "#FF3636"

$label = New-Object Windows.Controls.Label
$label.Content = "Type:"
$label.Margin = "12,12,0,0"
$label.FontWeight = "Bold"
$label.Foreground = "White"

$comboBox = New-Object Windows.Controls.ComboBox
$comboBox.Margin = "51,12,0,0"
$comboBox.Width = 160
$comboBox.Height = 23
$comboBox.VerticalAlignment = "Top"
$comboBox.SelectedIndex = 0
$comboBox.Items.Clear()
$comboBox.Items.Add("Homies") | out-null
$comboBox.Items.Add("Ninjas") | out-null
$comboBox.Add_SelectionChanged($function:typeChanged)

$saveLabel = New-Object Windows.Controls.Label
$saveLabel.Content = "Characters:"
$saveLabel.Margin = "12,52,0,0"
$saveLabel.FontWeight = "Bold"
$saveLabel.Foreground = "White"

$listbox = New-Object Windows.Controls.ListBox
$listbox.Margin = "19,60,0,10"
$listbox.Width = 200
$listbox.Height = 300
$listbox.HorizontalAlignment = "Left"
$listbox.VerticalAlignment = "Top"
$listbox.SelectionMode = "Single"
$listbox.Items.Add("Big_Loco") | out-null
$listbox.Items.Add("Grumpy") | out-null

$goButton = New-Object Windows.Controls.Button
$goButton.Content = "Launch"
$goButton.Width = 200
$goButton.Height = 30
$goButton.VerticalAlignment = "Top"
$goButton.Add_Click($function:launchClicked)

$grid.Children.Add($label) | out-null
$grid.Children.Add($comboBox) | out-null
$grid.Children.Add($saveLabel) | out-null
$grid.Children.Add($listbox) | out-null
$grid.Children.Add($goButton) | out-null

$window = New-Object Windows.Window
$window.Title = "Character Loader"
$window.Content = $grid
$window.SizeToContent = "WidthAndHeight"

$null = $window.ShowDialog()
```



GAME DEVELOPERS CONFERENCE 2012

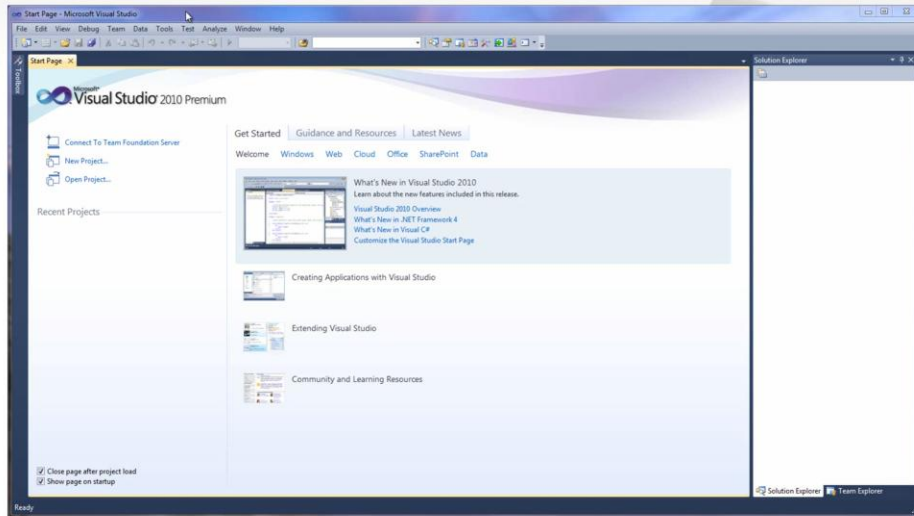
MARCH 5-9, 2012 WWW.GDCONF.COM

I'm going to quickly go through two ways of authoring UI's. One is completely programatically, as those who might be used to Python/Tk might be familiar with. I apologize for the small font, but this is a complete Powershell script (lacking in functionality due to space restrictions) that actually creates the stunning WPF user interface you see on the right.

The first 3 lines of the script load the WPF .NET assemblies. The two functions are handlers for events when the type pulldown changes value and for when the Launch button is clicked. Finally, we actually create a bunch of UI stuff and load them into a new window. It's pretty self-explanatory.

The application of this for our Ninjas vs. Homies game might be to have a pulldown to select a character type, populate a list of characters that fit belong to that type, and then launch them into our game. You can use what we learned about file and directory management to actually fill in the functions with something that works as expected.





The second way of creating WPF UI's is even more useful. If any of you have played with UI development using Visual Studio, you know there is a built-in drag and drop UI designer. For WPF applications, this creates what is known as a XAML file, which is an XML file that describes the UI. Through a lot of banging around, we've been able to author UI's using the Visual Studio designer and use those XAML files directly from a Powershell script to get away from having to generate all the components ourselves.

In this quick clip, I go through the process of laying out a UI using the WPF designer. As the UI is built, at the bottom you can see an XML file being built-up – this is the XAML file that contains all the info about a UI look and feel.

```

Design 1: XAML
<?xml version="1.0" encoding="utf-8" ?>
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="452" Width="774">
    <Grid Background="#FF363636">
        <ComboBox Height="23" HorizontalAlignment="Left" Margin="62,16,0,0" Name="comboBox1" VerticalAlignment="Top" Width="131" />
        <Label Content="Type:" Height="28" HorizontalAlignment="Left" Margin="35,16,0,0" Name="label1" VerticalAlignment="Top" Foreground="White" FontWeight="Bold" />
        <ListBox Height="332" HorizontalAlignment="Left" Margin="12,65,0,0" Name="listBox1" VerticalAlignment="Top" Width="203" />
        <GroupBox Header="Character Properties" Height="353" HorizontalAlignment="Left" Margin="231,11,0,0" Name="groupBox1" VerticalAlignment="Top" Width="509" Foreground="White" FontWeight="Bold" BorderThickness="0.5">
            <Grid Height="330" Width="507">
                <Label Content="Thumbnail:" FontWeight="Bold" Foreground="White" Height="28" HorizontalAlignment="Left" Margin="15,13,0,0" Name="label2" VerticalAlignment="Top" Width="76" />
                <Label Content="Strength:" FontWeight="Bold" Foreground="White" Height="28" HorizontalAlignment="Left" Margin="24,231,0,0" Name="label3" VerticalAlignment="Top" Width="67" />
                <Label Content="Weapons:" FontWeight="Bold" Foreground="White" Height="28" HorizontalAlignment="Left" Margin="21,265,0,0" Name="label4" VerticalAlignment="Top" />
                <Image Height="216" HorizontalAlignment="Left" Margin="102,16,0,0" Name="image1" Stretch="Fill" VerticalAlignment="Top" Width="175" Source="/WpfApplication2;component/Images/thumb.jpg" />
            </Grid>
        </GroupBox>
        <Button Content="Launch" Height="29" HorizontalAlignment="Left" Margin="231,370,0,0" Name="button1" VerticalAlignment="Top" Width="507" />
        <Label Content="Character:" FontWeight="Bold" Foreground="White" Height="28" HorizontalAlignment="Left" Margin="10,40,0,0" Name="label5" VerticalAlignment="Top" />
    </Grid>
</Window>

```

Here's what the XAML file looks like when done – pretty compact compared to the previous method of generating components with Powershell commands.



```

Design 11 83 XAML
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="452" Width="774">
    <Grid Background="#FF363636">
        <ComboBox Height="23" HorizontalAlignment="Left" Margin="82,16,0,0" Name="comboBox1" VerticalAlignment="Top" Width="131" />
        <Label Content="Type:" Height="28" HorizontalAlignment="Left" Margin="35,16,0,0" Name="label1" VerticalAlignment="Top" Foreground="white" FontWeight="Bold" />
        <ListBox Height="332" HorizontalAlignment="Left" Margin="12,65,0,0" Name="listBox1" VerticalAlignment="Top" Width="203" />
        <GroupBox Header="Character Properties" Height="353" HorizontalAlignment="Left" Margin="231,11,0,0" Name="groupBox1" VerticalAlignment="Top" Width="509" Foreground="white" FontWeight="Bold" BorderThickness="0.5">
            <Grid Height="338" Width="507">
                <Label Content="Thumbnail:" FontWeight="Bold" Foreground="white" Height="28" HorizontalAlignment="Left" Margin="15,13,0,0" Name="label2" VerticalAlignment="Top" Width="76" />
                <Label Content="Strength:" FontWeight="Bold" Foreground="white" Height="28" HorizontalAlignment="Left" Margin="24,231,0,0" Name="label3" VerticalAlignment="Top" Width="67" />
                <Label Content="Weapons:" FontWeight="Bold" Foreground="white" Height="28" HorizontalAlignment="Left" Margin="21,265,0,0" Name="label4" VerticalAlignment="Top" />
                <Image Height="216" HorizontalAlignment="Left" Margin="102,16,0,0" Name="image1" Stretch="Fill" VerticalAlignment="Top" Width="175" Source="/WpfApplication2;component/Images/thumb.jpg" />
            </Grid>
        </GroupBox>
        <Button Content="Launch" Height="29" HorizontalAlignment="Left" Margin="231,370,0,0" Name="button1" VerticalAlignment="Top" Width="507" />
        <Label Content="Character:" FontWeight="Bold" Foreground="white" Height="28" HorizontalAlignment="Left" Margin="10,40,0,0" Name="label5" VerticalAlignment="Top" />
    </Grid>
</Window>

```

In order for us to use this in Powershell, you have to get rid of the Class reference at the top (this is a helper for C# and not applicable to Powershell). Also, the thumbnail image source was a placeholder to make sure the thumbnail looked good. Since this is also referencing something from the Visual Studio project, we need to get rid of it or change the source to a blank black thumbnail here. We'll fill this in depending on what character is clicked in the app itself.

## GDC TECH ART BOOTCAMP

### UI

```
Add-Type -Assembly PresentationFramework
Add-Type -Assembly PresentationCore
Add-Type -Assembly WindowsBase

function trackChanged
{
    # do something on change
}

# Load WPF UI
$xml = [xml] (Get-Content ".\MainWindow.xaml")
$window = [Windows.Markup.XamlReader]::Load((New-Object System.Xml.XmlNodeReader $xml))
$window.Title = "CharacterLauncher"

$typeComboBox = $window.FindName("comboBox1")
$typeComboBox.add_SelectionChanged($function:trackChanged)

$thumbnailImage = $window.FindName("image1")
$thumbnailImage.Source = "file:///path/to/images/black.gif"

# Show UI
$window.ShowDialog() | out-null
```

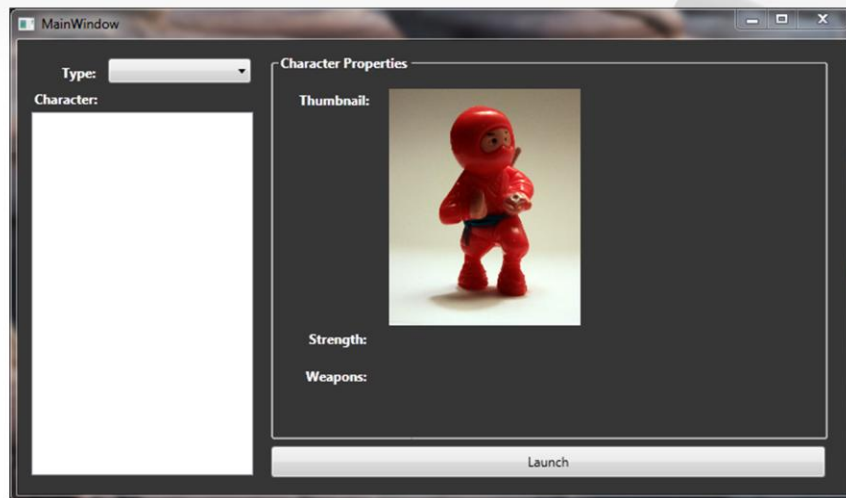


GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

Now, to launch the app, this is the minimum set of code needed. As before, we need to make sure the WPF .NET libraries are loaded. We have a small snippet of code that reads the XAML file we created in Visual Studio and loads it – that's all you need to generate the UI.

The rest of the code I've put in as placeholders to show how you can now access specific UI components from Powershell, add event callback functions and manipulate the thumbnail image source.



To complete the app, we'll need to populate the types and characters when appropriate (using skills we learned previously). You'll need to parse some XML and possibly use some image data using the XML and .NET skills we picked up earlier. You can see how these seemingly idiotic skills pile on quickly to create useful tools.



At many points during game development, you hit situations where there is too much data. For our Ninjas vs. Homies game, we keep detailed win/loss stats for every multi-player match to make sure that balance is consistent. Because there are 12.5 million matches played a day, we have a lot of data – you can be sure this data isn't stored in text files somewhere. If we have a load of data that we want to mine for info, we'll use a database.

Database is hit or miss in Python depending on what database you're using. Anyone who's looked for a database module to work with MS SQL or any other flavor can testify that there are too many solutions out there – it's like finding a good contractor.

Once again, .NET to the rescue; in Powershell, we have built-in DB access without installing anything. Disclaimer: since I work at Microsoft, we use MS SQL and things just work out for us with the built-in DB libraries.

## GDC TECH ART BOOTCAMP

### database

```
$db = new-object System.Data.SqlClient.SqlConnection
$db.ConnectionString = "server=DBSERVERNAME;database=MyDatabaseName;Integrated Security=sspi"
$db.Open()

$cmd = new-object System.Data.SqlClient.SqlCommand
$cmd.Connection = $db

$cmd.CommandText = "SELECT WinnerName FROM AllMatches ORDER BY WinnerName"
$winners= @()
$res = $cmd.ExecuteReader()
while ($res.Read()) { $winners += $res.GetValue(0); }

$cmd.CommandText = "INSERT INTO AllMatches (WinnerName, LoserName) VALUES ('Kirigakure_Saizo', 'Joker')"
$cmd.ExecuteNonQuery() | out-null

$db.Close()
```

GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

In Powershell to talk to databases, first we open a database connection in the first three lines. Then we create a SqlCommand (SQL is the language to talk to databases) and attach it to the database connection.

In the first command, we query the database to “select” a list of all the winners from the database of all matches. Once we have this, we can plot the data in a tool that shows win/loss balance. We can generate a heat map of where players commonly die on a level. There are many tool possibilities with just reading from a database.

In the second instance, we aren’t running a read-only query; we’re interested in modifying the database. So, we just create the appropriate SQL command and run ExecuteNonQuery() on it to update the database.

Obviously, this is merely scratching the surface of what we might do with databases; the interest is in having you see the tiny bit of code necessary to do the basics and you can



extrapolate that to larger tools or usage.

## GDC TECH ART BOOTCAMP

com objects – excel

```
$xl = new-object -com Excel.Application
$xl.Visible = $false
$xl.ScreenUpdating = $false
$xl.DisplayAlerts = $false

$wb = $xl.Workbooks.Add()
$ws = $wb.Worksheets.item(1)
$ws.Name = "Progress by Character"
$xlcol = 1

foreach ($char in $characters)
{
    $xlrow = 1
    $ws.Cells.Item($xlrow, $xlcol) = $char
    $progress = Get-CharProgress $char
    foreach ($file in $progress)
    {
        $xlrow++
        $ws.Cells.Item($xlrow, $xlcol) = "$file.FullName, $file.LastModifiedDate"
    }
    $xlcol++
}

$ws.UsedRange.EntireColumn.AutoFit() | out-null
$wb.SaveAs("C:\Temp\progress.xlsx")
$xl.Workbooks.Close()
$xl.Speech.Speak("Give your tech artist a raise")
$xl.Quit()
$xl = $wb = $ws = $null
[GC]::Collect()
```



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 WWW.GDCONF.COM

The last thing I wanted to cover is to show how Powershell allows us to easily interface with other third party applications that don't support native Python scripting. Many applications that we touch peripherally as Tech Artists offer a COM object interface to programatically drive the apps. In this example, I wanted to show you how we might drive an application such as Excel using Powershell – if any of you Tech Artists has to work with Project Manager's or Producers, you'll know this is a useful skill. Photoshop offers a COM interface as well, which theoretically allows you to script Photoshop operations externally from Powershell (as opposed to the built-in Javascript or VBScript support from within the app).

I'll also note that this same COM object scriptability is also available from within Python via the pywin32 module.

In this example, from a Powershell script I can open up Excel in hidden mode, write progress info on our game character development to a new worksheet, and then save the file as a native Excel file instead of juggling comma-delimited text or

some other mess.



Wow, that was a lot to cover.

To conclude, I'm excited to be a part of us folks covered by the "Tech Art" umbrella. It's an exciting and challenging field, and maturing quickly. We have such a diverse skillset, but I am confident that as we progress on, more and more will be expected of us technically. We've gotten to a point where the more effort we put into tools and workflow with the artist in mind, the easier it is to become an "artist" with a few button pushes. So, to stand out in the future and not just be a commodity, you need to either be a tremendously talented artist or a great technical artist, with emphasis on the technical.

I hope the quick examples I tossed together using Powershell have convinced you either to dive more into the technical side, for those who are still getting by with strictly 3D chops, and secondly that Powershell is worth a look as a tool for your tech artist Batman belt.

Better  
Faster  
Stronger

Teaching Tech  
Artists to Build  
Technology.

Rob Galanakis • CCP Games • [rob.galanakis@gmail.com](mailto:rob.galanakis@gmail.com)

My name is Rob Galanakis, I'm a Lead Technical Artist at CCP Iceland, I'm also a founder of [tech-artists.org](http://tech-artists.org), and today I'm going to talk about teaching Tech Art teams to build technology.



Before I begin in earnest, before I tell you what and how to teach and train, hell before I define exactly what I mean by “building technology”, I think you all deserve a pat on the back. Since the Tech Art community has come into its own over the past few years, I’ve seen the type of work we do change. The type of discussions heard at the Tech Artist Roundtable and read on tech-artists.org four or five years ago was a lot different from what I see now. A few years ago we were asking about how to rig a knee, now we’re making our own IK solvers. We were asking about half lambert shading, now we’re writing our own graphics engines. We were asking about writing to text files, now we’re setting up cloud clusters. I don’t think there’s a domain in game development that I haven’t heard of a Tech Artist infiltrating. We’re no longer holding things together from the shadows- we’re taking on mission critical roles and features from start to finish.

**“I’m having breakfast with the  
CEO of Tencent Boston to explain  
why he needs Tech Artists.”**

**- Jeff Hanna, 2009**

This growth has been the product of a success I’m not at all surprised to see. I’ve been, all of the speakers here today have been, loud and vocal evangelists for Tech Art because we know you all can provide better. Better tools, better pipelines, better workflows, a better type of development for artists and all members of our teams. And over the last few years our efforts have been a resounding success. I get the feeling for the first time that studios are by and large starting to understand us. **This is important.** Part of the reason Jeff Hanna organized the first TA Bootcamp last year was to address this issue of developers in general not knowing what to make of this uncategorizable phenomenon called Tech Art.



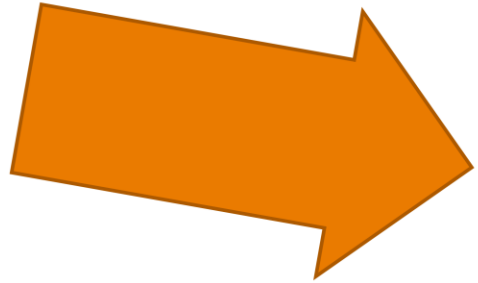


So I believe that management and our teams are starting to understand us, but, do we understand us? The side effect of this success and growth has been a very rapid change of roles and responsibilities. It's a double edged sword; I see more and more Tech Artists developing the types of hard technical skills I've been advocating for, the type that make us an even more effective weapon, but I'm not always sure we're aware of the repercussions. I've seen more than one otherwise competent Tech Artist fall victim to our own version of the Peter Principle. Based on our success, we get more and more responsibility until we are put in a position where we just aren't able to execute.



An analogy. Years ago, my brother and I remodeled a bathroom as a gift for my mother for her birthday. We estimated it'd take 3 weeks and cost \$1000 dollars. It ended up taking about 3 months and \$4000. Now in hindsight this is obvious. We had no idea what we were getting ourselves into. It was ultimately the plumbing that killed us, because though my brother could fix and replace pipes, laying the plumbing for an entire bathroom with a jacuzzi is quite a different story. So we ended up with an overbudget construction, rough edges, and nagging problems like horrendous water pressure. Now, we did get those problems fixed, and many a bubble-bath has been taken, but to finish it, we needed to call in professionals to redo a lot of our job properly.

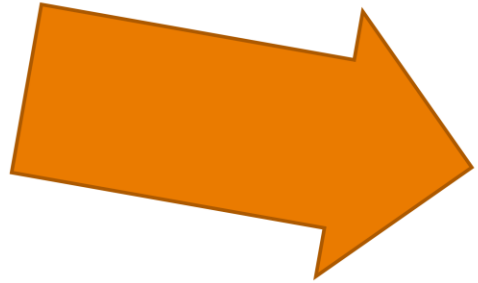
**Geek,**  
**not Plumber**



Now, I learned something from this. If you need something plumbed right, hire a plumber. We weren't snaking a drain here, we were installing the pipes for a new bathroom, and covering that with new walls and tiles and paint. This wasn't something we could afford to get wrong, but we got it wrong, because it wasn't our competency and we didn't have someone to teach us. We got it wrong and we paid dearly for it.

And this is just what worries me about the developments I've seen in Tech Art.

# **Tech Artist, not Software Developer**



We're developing software but we don't know how to be software developers. We're using cutting edge tech to work on some great ideas and we don't realize, there's a big difference between writing code and building technology.

I'm not pointing fingers. You're talking to the mother of bathroom remodeling fuckups. See, I had these delusions that I was a software developer long before I actually was, so I used to try to build technology. It took a lot of trial and error, but I finally learned. But that was not the end of the screw ups. In fact it was just the beginning. I thought because I knew how to build technology, every Tech Artist knew how to build technology. But they aren't trained to do it. Some because they haven't yet learned, some because they think they should never have to learn. But the ability to build technology, like a professional, as opposed to just hacking on scripts, is a skill every Tech Artist needs to have. Whatever you think of my views on what a TA is or should be, I think we can all agree that improving your programming skills improves you as a Tech Artist.

# **Tech Artist, not Software Developer**



When success makes you seem awesome, and no one notices failure, how we've been doing things is fine. But we are working on bigger, more important projects now and the flipside of the greater awareness and importance of our craft is, we can no longer afford these types of failures. We get to build great things and people notice. But it also means we can fail harder. And people notice.



So that's what 'technology' is for me. It is the stuff at the pointy top of the pyramid. The exciting stuff we're think we can swallow, but we choke and die. It is the bathroom we don't have the skills to remodel. It is the complex systems we want to build, and we can either suffer through it, or we can train ourselves properly.

Over the past couple years I like to think that I've finally gotten good at creating technology and training teams to develop and maintain it. Like I said, a lot of this comes from failing so hard at first. Some of it was the luck of having great people to work with. But those things aside, I can pretty clearly point to **three** distinct areas that are necessary to build a Tech Art team that can build and maintain this sort of technology.

# **1. Support Process**

## **2. Code Review**

## **3. Collaboration**

The first key is to get a process for support tasks into place. Unless your Tech Artists can focus and have room to breathe, they are never going to learn any of the other necessary skills.

The second key is to set up code review. I don't think any subject has caused so many headaches for me but it is absolutely essential, and I'm going to talk about the how and why of reviewing code.

The last area is collaboration. Collaboration both in working on common projects, and team cohesiveness. To develop those big tools and systems effectively, you need to be able to apply more than one or two people to a single problem.

And I'm going to close by going over some of the pitfalls and problems I've run into while implementing these changes. Learning from my mistakes should give you a head start in driving these improvements.



# **Support** Process

So let's start by talking about that process for support. Over the years I've learned that process is a dirty, dirty word.

# Support Pr@%\*SS

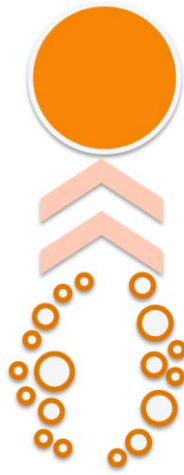
Nobody likes process. In fact, lots of us hate process. I don't want you to like process. For most Tech Artists, 'process' as a concept goes against our very core. We shouldn't have processes because we want raw, direct access to the people with problems. We shouldn't have processes because we should write a script for that. A process implies a workflow we can't automate, and we believe we can automate any workflow.

However I can't simply ignore the fact that how TA's handle support is broken. I don't think there's a happy programming job in the world that works how we work. To successfully build technology, you need to be deliberate. You need to plan and have stability to execute. Tech Artists often get the role of firefighter and garbage man. We have to understand this is part of our lot in development, but we also need to aspire to minimize that role. To handle it in a way so we can address larger issues and tools as well. We have great ideas and we need to create an environment that we can build those ideas in.

Support process isn't about creating something rigid and inflexible, it is about making your development and support more deliberate. Process isn't the answer, but it does allow us to find an answer.

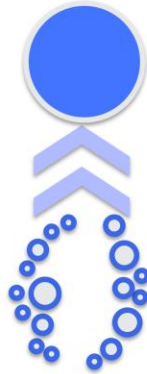
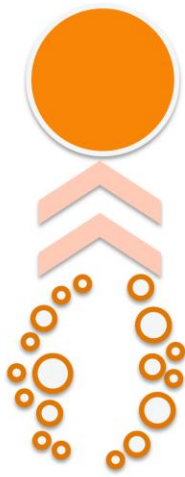
TA Fixes Defect/  
Adds Feature

Users Find Defect/  
Request Feature



## Idealized Support Process

So let's talk about how we think of our current process. This isn't much of a process- defects are found and feature requests are made by users, and fixed or added by TA's. This is a tight feedback loop, which is good. We want to be able to get raw feedback from the user, and act on that feedback quickly. I want to introduce process, but I want to keep the responsiveness of this type of support.



## Actual Support Process

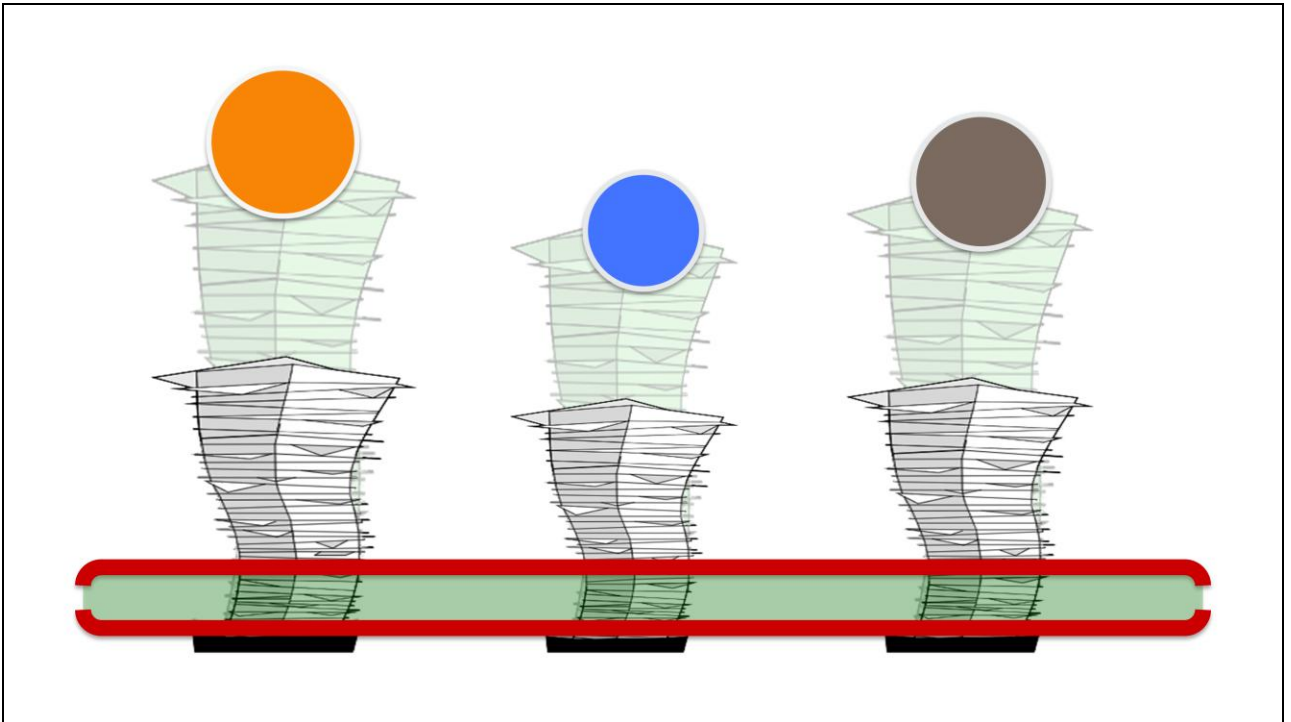
But we actually have that process performed, in parallel, by each Tech Artist. A tight feedback loop is great, but when we find every TA has his own feedback loop and way of doing things, which is usually different from every other TA, you create several problems.



Providing support has side effects. The codebase grows. It grows much faster than it should, because code is rarely shared properly and is often copied and pasted, or existing code is unknown and unused. And the relationships TA's have with customers is different for every group. What we develop when we work like this is not shared property, and we never develop a shared identity. We don't develop cohesive practices, a unified style, a vision or architecture. So every step further down the individualized support path we go, the more redundant work we do, the harder things are to unwind. The stack of debt grows and grows.

This works as long as your TA's have time and success. But I've seen it with every competent TA I've worked with. You build up this stack so you can work effectively. And eventually it turns to shit, and takes all your effort to support it. And now you can't really develop new features properly. Your truck factor becomes high. **Your code becomes brittle. And**

your genius and ethic can keep things together for a while. But at some point, you run out of hours in the day. Until then, you can give high quality support, which is expected- you're a skilled bunch and ours is a labor of love. But as soon as you hit that wall, you start to stumble, you fail hard. The end goal of a support process- what all the changes I propose hope to achieve- is to have the most productive Tech Artists possible.



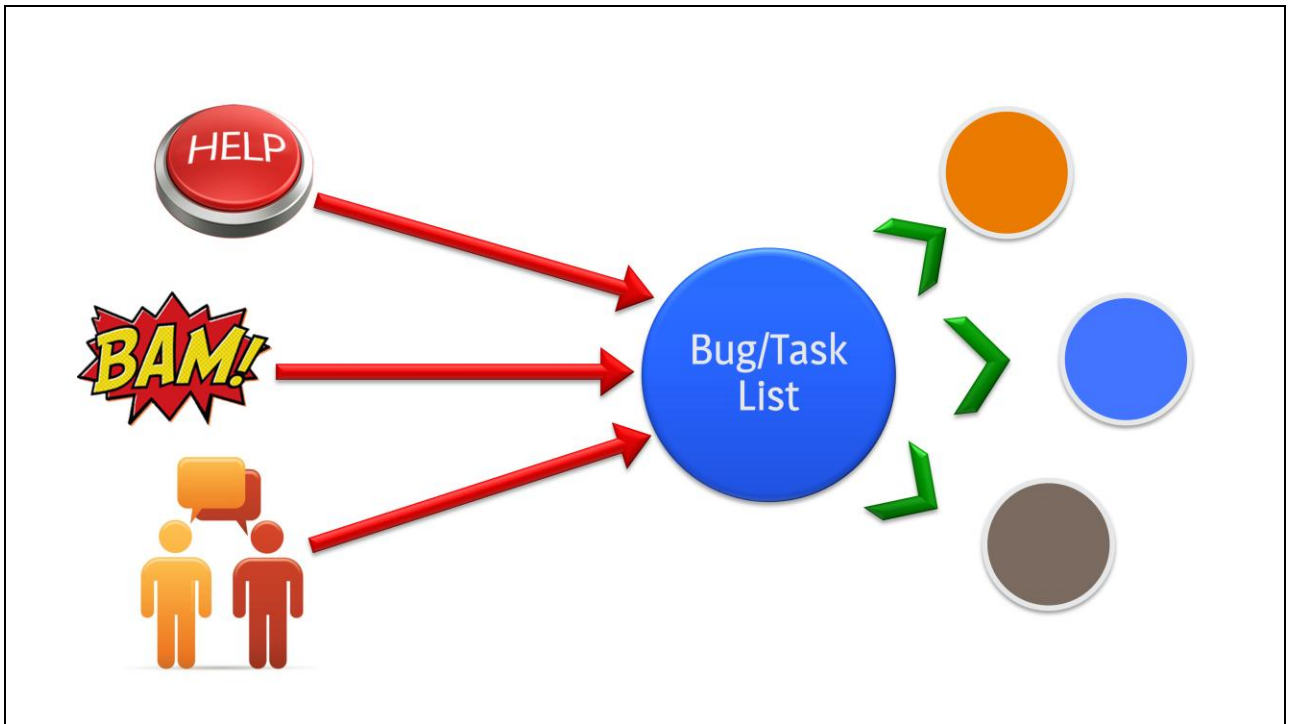
In order to do that, we need to start by collapsing the stack (no pun intended, Max users). Putting a support process in place is the first step towards understanding the work we do, sharing it, paying down debt, developing better. You're not going to do that via a support process alone. The real goal of whatever support process you end up with is this:



**Realize** how **much**  
**support** you do, and  
**trust others** to do it.

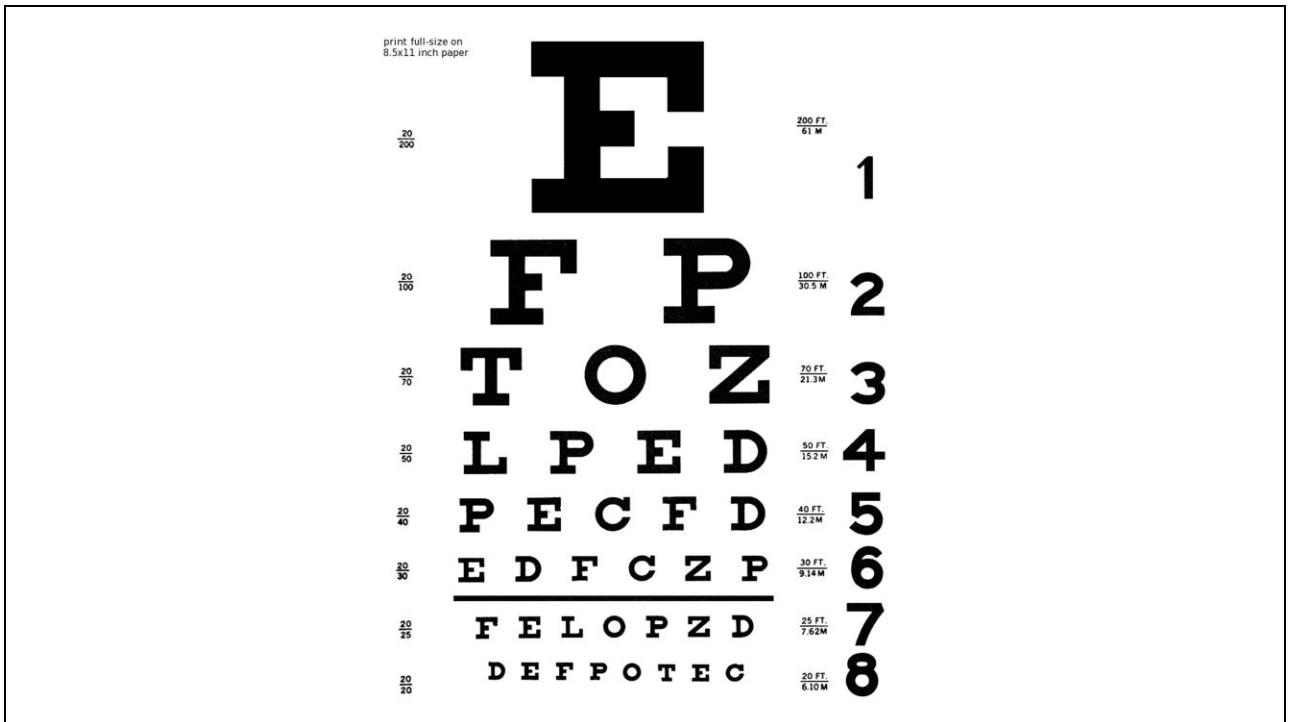
Realize how much support you do, and trust others to do it. That's it. Most people don't realize just how much time they spend supporting artists and designers, answering programmer questions, adding features, fixing bugs, twiddling around with stuff. A support process forces a reckoning with this behavior.

And the idea is, once you realize how much time you actually spend in support, and once you have a process where other people can actually cross over and help you, you can start to breathe again. It is the first step down a long road to building technology.

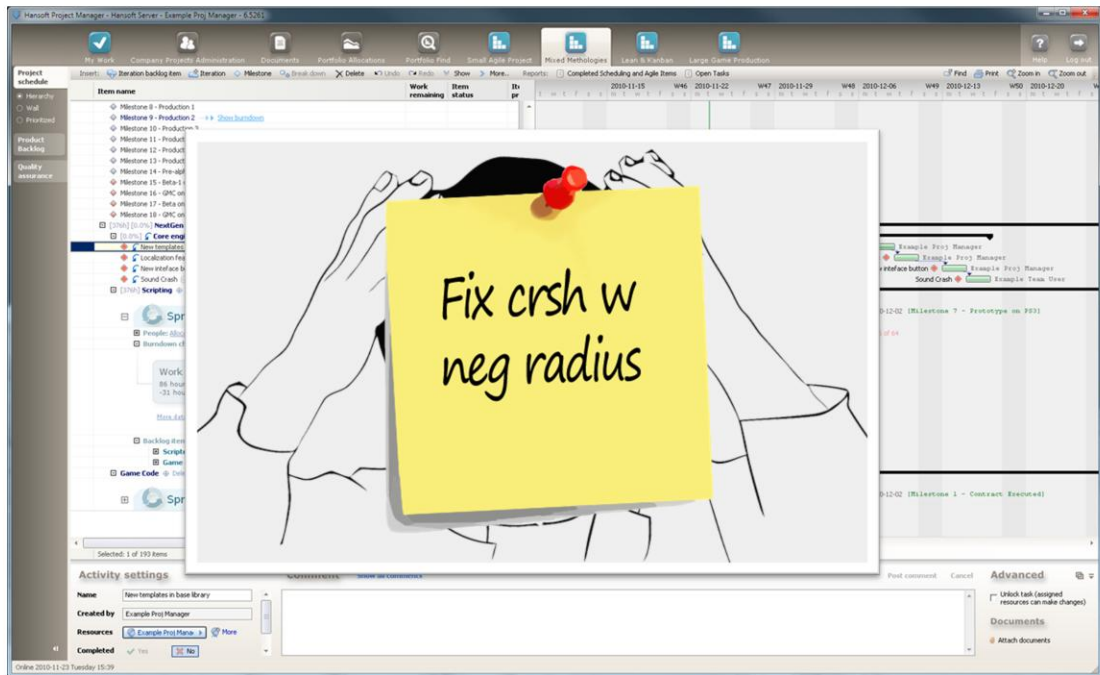


My recommended support structure is something like this. There is of course the direct feedback, via face-to-face or other channels. That is not going away. But important here are the technical mechanisms. Build a way to catch when errors happen in your tools, and get those errors, along with callstack and logs, reported to your team as emails, or into a bug tracker, or whatever. I'd also suggest giving users a big shiny 'HELP' button when you can't do it automatically. These things are not hard to set up.

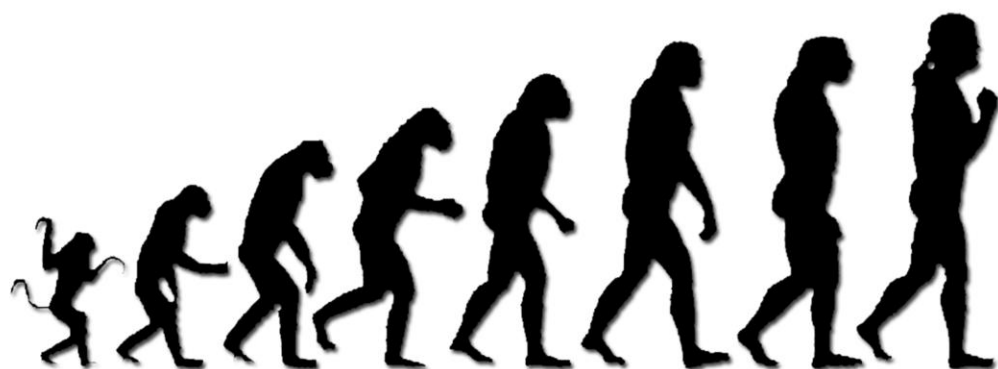
Everything goes into a single task list. Let the Tech Artists draw from this list, and manage it internally. Do not involve management or gate tasks by assigning them out in some bureaucratic fashion. And use daily standups and the task list to keep track of what people are doing.



One reason I love this system is everyone on the Tech Art team can see when things go wrong. You can see where the problematic tools are because they generate the most crashes. You can see who is not testing properly because there are always a flurry of reports after they check in. You can see who isn't specing out work well because they generate the most feature tasks. You can see the exact errors in a tool and everyone can offer suggestions, or take it on themselves, even if it isn't their core responsibility. For this to happen, you need automated reporting, and processes that are simple enough for everyone to participate.



On the other hand, I've seen people try to use project management software like Hansoft, to manage tasks and support. Don't do this. Keep it simple, stupid. Don't require people to log into a chatroom, or send an email to a special address they need to remember. Manage this via familiar, simple tools, such as email or Outlook Tasks, and automate, automate, automate.



Figuring out a support process is evolutionary and not a science. Keep it simple, grow it organically. Try lots of things, throw out what doesn't work, keep what does, and sometimes you need to switch out what works just so you can find something better. Those optimal mechanisms are unique for each team and studio, but every team has them, and you just need to keep trying until you find them.

So if support processes are designed to be flexible, helpful, and non-intrusive, the next goal does not have the same fluidity and it is always disruptive.

# Code Review

Code review. I imagine a good number of you have tried out or at least read about code review. There are lots of benefits of code review, but the best way I can sum it up is this.



A successful culture of review creates a mind meld between your team. You have shared domain knowledge, you feed off of each other's skills, you develop a single cohesive set of standards and idioms. There are a hundred benefits to code review; and being here at GDC, at the forefront of the industry, probably means these benefits are obvious.

More than that, pretty much all software development houses, and game studios of any size, have some form of code review on programming teams. Some do it better than others, but everyone's doing it. The benefit is without question. But it took many years for code reviews to get implemented, and usually there are lots of aborted attempts because it causes so much friction and is seen as adding a lot of overhead.

# Mandatory & Thorough Code Review

So if I were to stand here and just tell you why code review is good and how to run effective reviews, I'd be doing you a disservice. Just saying 'Huzzah! We're doing code reviews!' is not good enough.

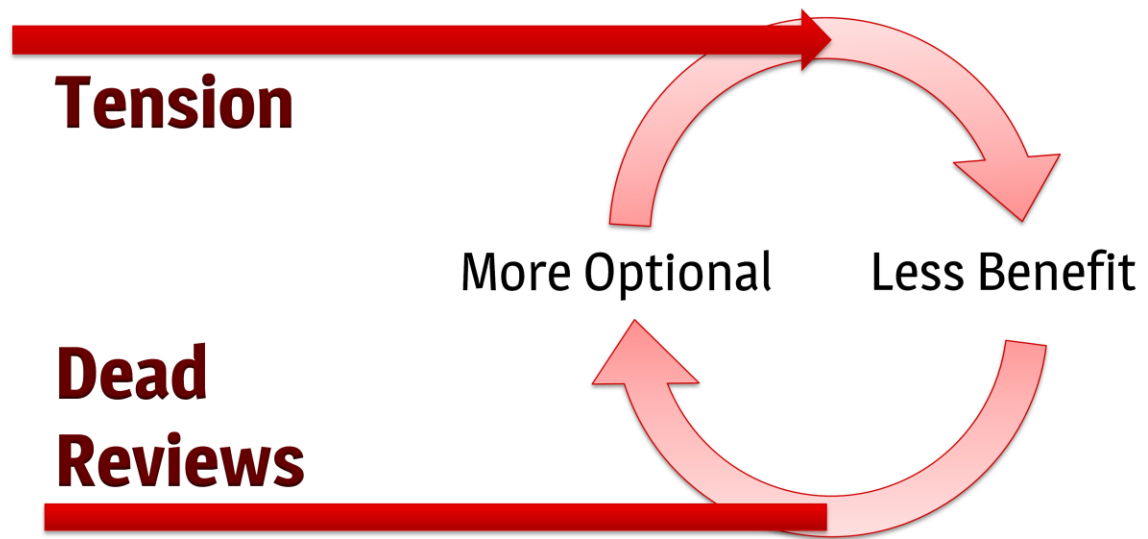
So not just any code review. **Mandatory** and **thorough** code review.

Mark my words: The road to effective code review is perilous and will cause tension on the team. Which is sort of nice because you know if they're not causing tension at first, you're doing it wrong. As opposed to support process, which can be continually tweaked and be made non-invasive, there are some absolutes with code review in a Tech Art environment that need to be set down from the start.

But why are code reviews so difficult? Well, you're basically putting two competent people into an arena, often a public arena, and having the reviewer attack the person being reviewed. Now, this is obviously not how successful code reviews work, but it's often how they can feel, especially early on.







Code reviews sound and work great in theory, and you think you're ready to do them, but reality of course proves differently. The primary obstacle to code review is **commitment**. Whereas art reviews are an accepted part of life, code reviews are not. And while you can tell your team they need to be done to ensure quality, if you don't want to have a morale problem you need to demonstrate that the reviews have benefits.

And herein lies the problem. It takes **at least** a few weeks for code reviews to start reaping benefits. It takes even longer when you have a team that has never built technology properly and, quite honestly, don't care much for software and project management theory. So code review sounds great, people start doing it, but it takes time, and causes tension. Because they cause tension,

someone decides to make them optional as a way to diffuse the tension. Now because they're optional, people don't do them thoroughly or at all, so they have much less benefit. And if they have less benefit, they become more optional. This is a vicious cycle that results in people not doing reviews at all.

# An effective review process requires:

- A Single Arbiter
- Total Compliance

And there is a cure but it is a hard pill to swallow. **Code reviews are mandatory, and a single Tech Artist has the final word.** Let me be clear about this and state in no uncertain terms: I have never seen or heard of code reviews being successfully implemented on a Tech Art team without them being mandatory and without a single arbiter of correctness. It doesn't matter who the arbiter is- it can be the lead, or a senior member, but there needs to be a single person responsible for the process and the quality of the reviews, at least at first. And it doesn't matter whether you do over the shoulder reviews, or have software, or use email, you just need to make sure every Tech Artist is getting their code reviewed, and reviewing other people's code. Just remember though that making code reviews mandatory **allows** them to be successful, it does not **cause** them to be successful.

So I'm not going to go deep into how you perform good code reviews. The truth is the subject warrants its own lecture. I'm going to have some links for you in the presentation notes, which I hope you download. Read them. You need to do your research, study, treat it as its own process and skillset. The performance of the review can, must, be tweaked and adjusted- how many people review, who reviews, what type of things are flagged, etc. But these two things- a single arbiter, and total compliance- cannot be negotiable.

## Dictator



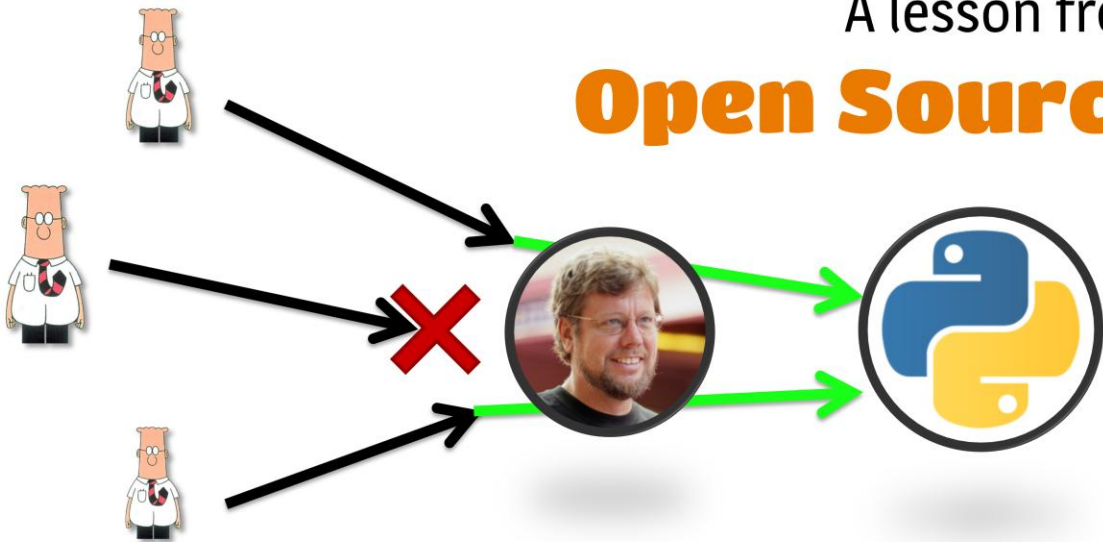
## BDFL



This sounds like a harsh dictatorship, and in some ways it is. And you need to get over it if you want effective review. If you want to run things like Barney the Dinosaur, you are going to fail. Very few people who haven't been part of a review culture want to submit to code review. But very few people who have been part of a successful code review culture regret it. The key is to figure out how to make it successful, and just like support process, needs to grow and change with the team. So you need to establish a dictatorship, and inside of that box, you have the ability to adjust your review process however you see fit. No one is going to get it correct right away. It will require constant revision, and the team will grow and improve so what works now may not be optimal in six months.

The 'dictator', so to speak, must be someone who can change their minds and solicit feedback from the team. It needs to be less a military dictatorship and more an open source software-like Benevolent Dictator. We can definitely take some guidance from open source project management. And as Guido van Rossum was the original Benevolent Dictator for Life (BDFL), we'll use his picture. Now your review dictator is obviously not 'for life', but the roles are the same.

## A lesson from **Open Source**



In open source, you tend to have a benevolent dictator, the 'father of the language', who at least initially is responsible for all the code that makes its way into the main branch, and often does all reviews to the core systems. He or she is a 'gatekeeper' of sorts. So if they don't like something, it doesn't go in. In practice, BDFL's cannot afford to be fickle, tyrannical, or anything less than excellent, because if they are, contributors will just fork the code.

For better or worse, we don't have the option to fork our code if our gatekeeper is not cut out for the responsibility. So we need to choose wisely, and make sure she is honoring their part of the bargain, by listening to the team and by educating herself on her own time as well. She's not going to be an expert out of the box- everyone will grow as time goes on.

And for those of us who have become the dictators of our Tech Art codebases, a good way to judge is, how compelled are people to hide away their own code where only they will see it? How comfortable are people contributing to the core,



putting things where they should go, or are they really just forking by carving out subdomains of the larger codebase?

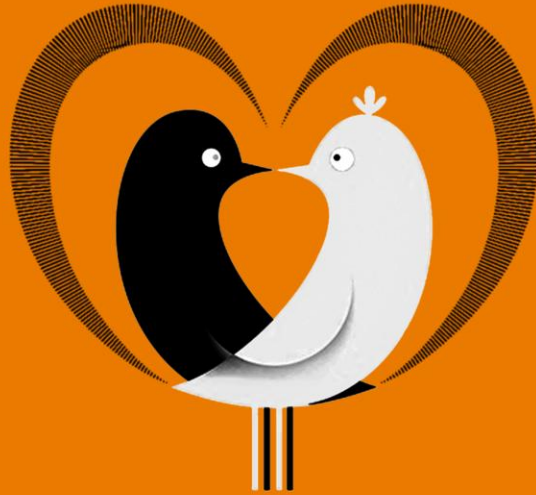
# **Code Review** **is the mark of a** **Professional**



And of course the best metric is, after a few months, are people learning? Are their reviews better, do they have less defects?

Everything about code reviews get's back to being a professional, to being able to develop software and build technology. There's a difference between just writing code, and writing code professionally. Anyone can write code that does x or y. Slightly fewer can actually work with other people's code. Many fewer take their code seriously enough that they require it to go through peer review. Just like a professional artist has his work reviewed and critiqued, so must you as a Tech Artist have your code reviewed in the same way.

# Collaboration



A support process will assemble you into a team. Code review will allow you to gel, work, and grow together as a cohesive coding unit.

Ultimately both of those things are working towards an environment that fosters collaboration. Collaboration allows us as Tech Artists to build technology. It is vital. Without tight, effective collaboration between TA's we are simply not equipped to develop the big projects we increasingly need to.

# What is it?



What specifically do I mean by “collaboration”? It doesn’t mean pair programming, though that’s a good indication you collaborating.

At its most basic it means, you need a coding buddy. You need another person on the team who can not just read your code, but understands the systems you are working in. You need someone you can go to with ideas and who can give meaningful feedback. You need someone who will come to you with problems and you can provide solutions. You need someone who you can sit with at a computer and pair program together. Collaboration in all these forms is a way to improve the effectiveness of all your team members, that is the goal we’re interested in.



These are the traits of successful programmers and successful programming teams. It wasn't so long ago, though, that the myth of the genius programmer was still prevalent and teams didn't have the sort of collaboration and practices I'm encouraging you to emulate.

But over the past few years, this myth has largely disappeared in the professional programming world. But we have our own version of it in the Tech Art world. We have the animation guy, the character guy, the environment guy. The guy that wrote system A and the girl that wrote system B. Each of us love the idea of being a badass Sherlock Holmes in our field of expertise. We love having the freedom to do what we want to do, to not really have to answer to anyone, to do amazing things and get the silent glory. But as fun as it is, we need to be the police force as well. We need to be clever like Holmes, yes, but we also need to give out parking tickets and we need to be able to respond to crises or operate in force. There is a time and a place for us to act as a solitary genius, but it must become the exception, and not the rule.



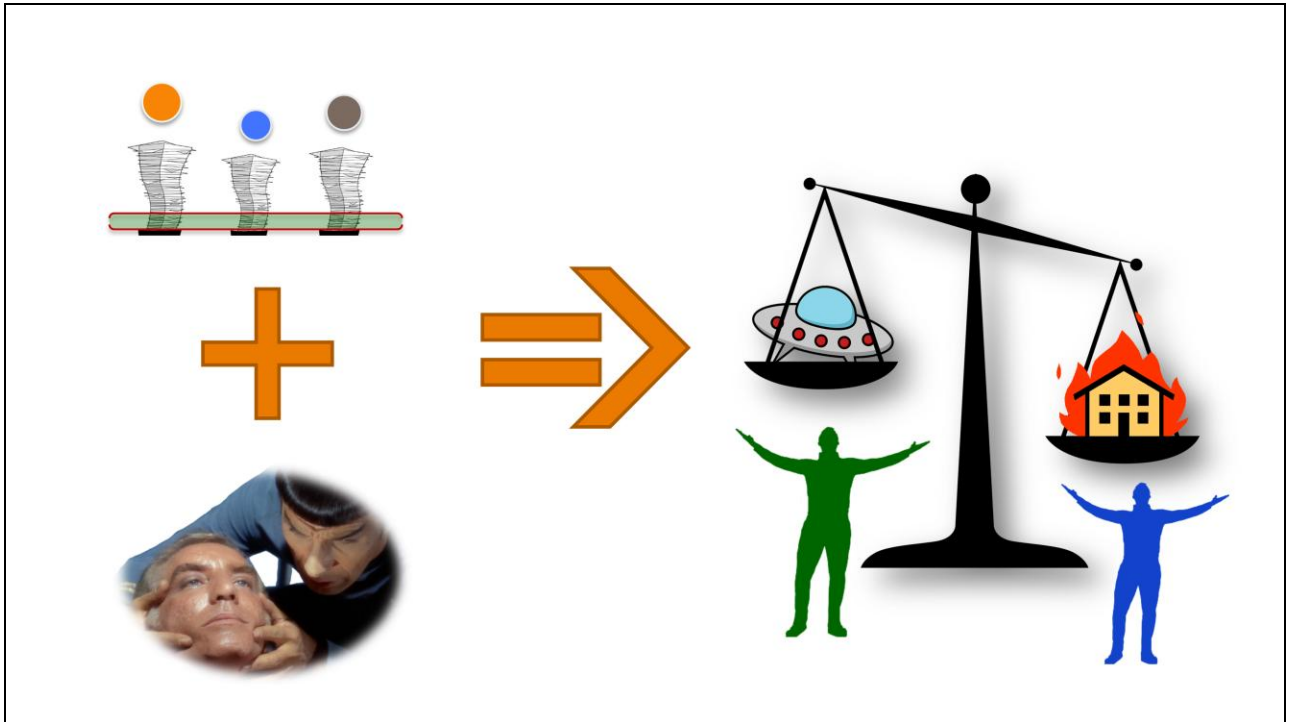
Why. Well it must be acknowledged that support is a large part of the job of anyone that develops tools. And like I said earlier, Tech Artists provide the best support in the business. We pride ourselves on it.

But the support splits our focus- we need to provide support, *and* develop new tech. And when it comes down to it, support is going to win every time.

So we're stuck providing full time support, but we all love doing those bigger things as well. So we work 50, 60, 70 hour weeks, and develop those bigger things in our spare time. And then something terrible happens. The new tech we develop comes with its own support burden, and suddenly we're working overtime *just* to provide support.

I've been guilty of it and I've seen it time and time again. So not only is collaboration a requirement for building technology, it is also a requirement for keeping your own sanity. Being able to share responsibility means you can load balance your

team effectively.



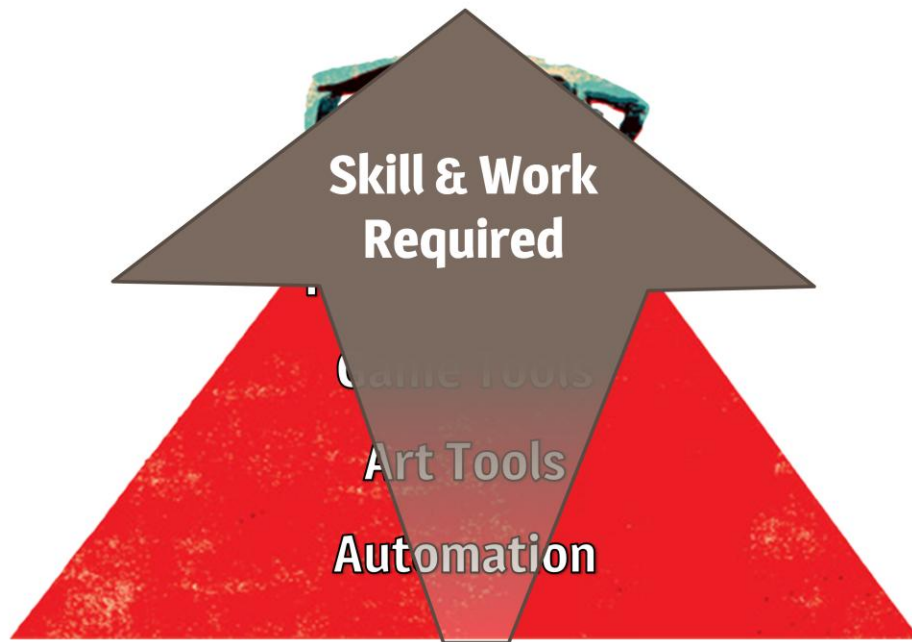
So we come back to fixing our support process. Remember this unique stack-per-TA? This is an impediment to collaboration. So we need to get rid of it. Now we have people that are actually sharing code. This is a form of collaboration, to be sure, but it is a reactive, not creative, form. It finds code that should be shared, and shares it. When problems come in, they can be fixed or informed by multiple people. It is collaboration for support. This is important but this isn't the creative form of collaboration I'm talking about here.

But if we combine that sort of reactive collaboration with code review, we reinforce the ability of the team to work together, grow and learn together. In my experience, I've found that it isn't until a team establishes code review that they establish a 'team identity,' where they turn from a



group of individuals who have the same job title, into a team that can collaborate together to create new things, as a team, and handle the burdens of tech art, as a team.

So we combine these two things- reactive collaboration, and team unification- and something marvelous happens. We move past the point where we can only load balance support, and we can start to load balance new features, new tools, internal work, everything. Suddenly we have the bandwidth to build technology.



Collaboration is your ultimate weapon against the likelihood of failing as your team succeeds. As you get higher on this pyramid of responsibility, there is a rapid increase of the skills and amount of work required in the tasks you are asked to do, and want to do. And a Tech Art team, as a rule, rarely has the experience with the sort of tasks that are at the tip of this pyramid. These are studio-wide features and tools that usually have very little to do with the specific expertise of Tech Art. Or they are super-critical, paradigm-shifting, fundamental changes to art pipelines, that can hose a team if not executed properly.



We all know the analogy about the bundle of twigs, and it works here. We're more effective in the long run when we work together. The key to building technology isn't having a team of programmers. It is to have experts that can program who collaborate like a team.

# **Avoid** the **Pitfalls**



I've been through this regimen at three separate offices and I've learned a lot each time. Now I hope I've stressed enough how the specifics of the implementations will be unique for each team, but there are some pitfalls to avoid. Now I say avoid but what I really mean is, these things are going to happen and you need to be ready.

# Morale

**will** **suffer** ...temporarily



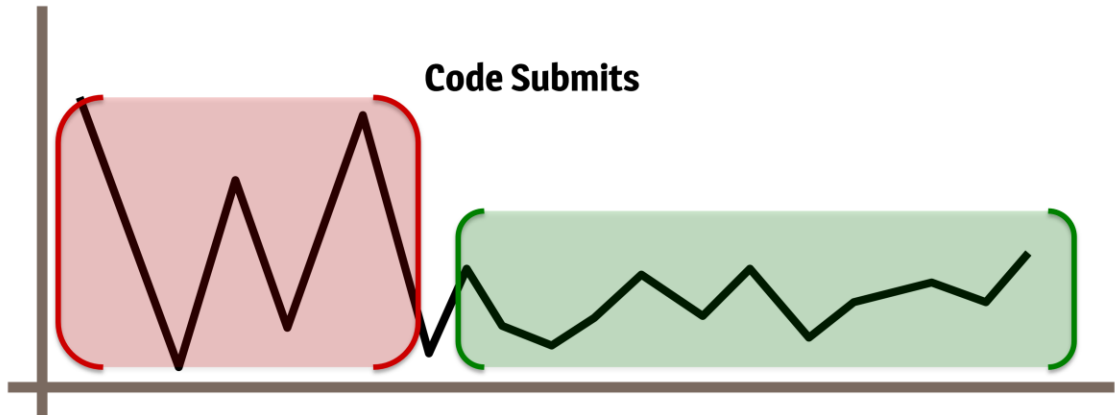
I'm going to be straight with you. Making the changes I'm advocating for is going to rub some people the wrong way. Changing how they interact with artists, forcing them through procedural hoops for code reviews, putting them into collaborative projects where they feel awkward- there's no way to get around the fact that this is going to have a negative impact on morale. You have to go into this with the belief that the suffering is temporary. Making these changes has led to huge gains, but I can say personally they've caused hardships, for me and for others. I'll talk about some of those difficulties in a bit, but in all cases, we came through it together and were better off for it.

# Rigor or Failure



But we only got through it because we had the will to succeed. We knew that we had to eat our vegetables to make us strong, even if we hated the taste at first. We refused to compromise on the changes we knew we had to make, because if you aren't rigorous, the whole thing is going to fail and you'll be far worse off. You need to follow through with process improvements, rather than just letting bad habits creep back in. You need to be absolute in your commitment to reviews and not let whining throw them off track. You need to create opportunities to collaborate, and not just wait for them to occur. Find a way to address morale problems without compromising the integrity of what you're doing.

# Find a Metric for Success



If you are being rigorous, you can measure your success. This doesn't have to be complex, or official. For me, it was simple—the number of Perforce submits for the team and individuals over time. Too many checkins, and it's clear we're crunching or behind on work. Too few checkins, and we're not developing effectively, either because we're not doing much or we're doing too much without checking in. I wanted a stable line. And without fail, a few months after you put a support process in place, are doing reviews, are collaborating on projects- I bet you'll get a stable line.

**Change  
won't happen  
overnight.**



Now that said, don't expect changes overnight. I'd suggest doing these things in order- starting with improving your support process, then implementing code review, then finding collaborative projects. Do one at a time, but do not half ass anything. Implement, and stick to it. Change won't happen overnight but it doesn't mean you're doing it wrong, as long as you're rigorous. Give it a month, then measure and adjust. You're slowly evolving into the team you know you can become. It takes longer than a month but should not take a year. As long as the team makes some progress, every week, be content and keep pushing.



And **not**  
**everyone**  
will **fit**  
**a mold.**



**THE BORG**  
They can assimilate THIS!

But no matter how long you wait, there will be people who won't change. This is the worst cause of morale problems. This is what I have struggled with most personally.

There are two types of people who don't fit in, and yes I realize the irony of that statement on this slide.

People of the first type are just not cut out for the level of coding your team will be doing. You need to find them a place in the organization where they can contribute. It cannot be writing production code. It may change constantly- maybe it is to plug a hole until the rest of your team can fix it properly. But there is always a place these people can add value. Maybe not as much value as the ideal employee you'd want to replace them with, but in the immortal words of Donald Rumsfeld: "You go to war with the army you have, not the army you wish to have."

The second and more difficult type are people who don't want to or can't add value in non-coding roles. Maybe they have a

software background or have been scripting for a while. There's no easy way to deal with them. They need to be put into the first category, or they need to assimilate, or they need to ship out. They can quickly become an open wound for morale, or will compromise the rigor these changes need. Give them the opportunity to change roles, or offer the extra help for them to improve. But their poor fit cannot be allowed to fester.

# Building Technology is as easy as...

**A. Support Process**

**B. Code Review**

**C. Collaboration**

And D, E, F, G, H, I, J...

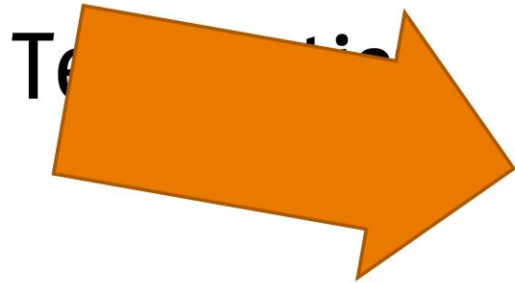
I've made all of this seem a lot simpler than it is. We're talking about significant culture changes. Cultural changes are the hardest ones to make. Computers are easy to deal with. People are not. This is a process, and a journey. The thing of paramount importance for training your Tech Artists is that you establish principles and stick to them.

Now I want to close with this.

The goal is **not**  
to become a  
programmer.



It is to **learn**  
to program,  
to be a **better**

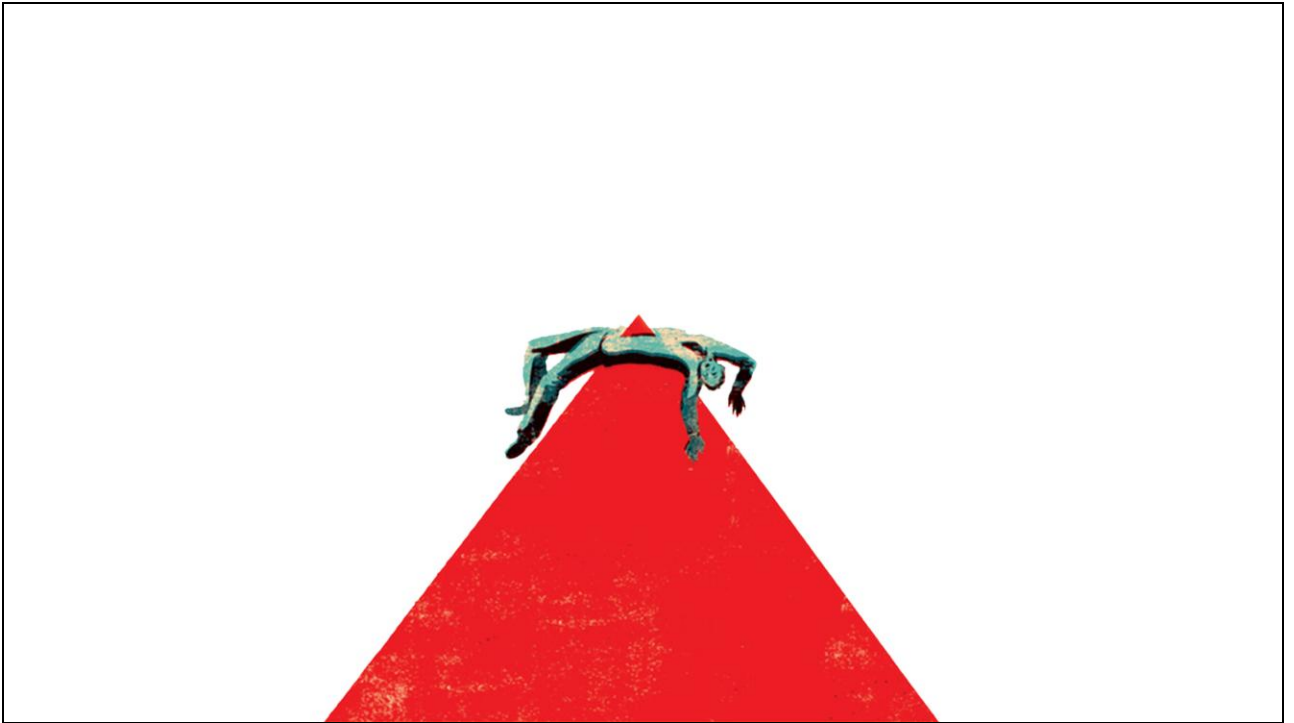


I call myself a Tech Artist, but I am a programmer. I don't have creative visual ideas anymore. I avoid Maya as much as possible. I know, in skills and interests, I am on the fringe of Tech Artists. This is a fact I've come to be OK with.

So please do not take away from this presentation that I am saying, being a programmer is the most evolved form of Tech Art. It isn't. I have made myself a programmer because I enjoy the tools and pipeline aspect of Tech Art most. Other people enjoy different things.

I'm not trying to ruin your life. I'm trying to make it better.

But just because you do not want to write your own compiler one day, does not mean you don't need to know how python importing works. Just because you only write simple tools, doesn't mean you don't need to know how to serialize UI settings properly. If you are going to consider writing code part of your job, it is incumbent upon you to write good code.



But the goal of this presentation isn't to encourage you to become a programmer. We. Are. Tech Artists. We need to respect that history and who we are. If you're uneasy with these changes, I'm not asking you to forsake what you love. I'm asking you to sharpen or relearn your programming skills, so that you may become a better Tech Artist. You can know how to program without being a programmer. And knowing how to program well will allow you to get stuff done quickly, which makes you a better Tech Artist.



And that makes everyone happy.

Thanks. Questions?

# Build It On Stone

Seth Gibson  
Rapid Experience Development



GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATES: MARCH 7-9  
**2012**

Alright well, welcome to Tech Art bootcamp 2012, for those of you who were here last year, welcome back, and let me extend a personal thanks to all of you who upvoted us on the review sheet, enough so that we get to do this again. I know I don't speak solely for myself when I say it's been a privilege, an honor, and just downright fun doing this, and I hope we can keep doing this for years to come

(So Fill Out Your Review Sheets please, and same as last year, bribes go out the day I get back from GDC)

The thing I'd like to talk to you guys about today is sort of a hybrid of two things that have become near and dear to my heart over the last couple of years, drawing on some of the situations I faced and experiences I gained in my brief stint as a Tech Art Lead/Manager last year. Industrywide, it's safe to say, insert your platitude of choice for change here, you know, change is on the horizon, winds of change are blowing, OMG zombies are coming we're all going to die, the sky is falling, you get what I'm saying, but I think

that puts us in a really great position to step back for a bit, take a look at what we've been doing and think about how we can change all that, and of course the logical place to start...



# Begin At The Beginning



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9  
**2012**

...is at the beginning, and I definitely feel that Tech Art is uniquely qualified to observe from this point. You know, I remember a while back getting with my parents and looking at some videos of Halo ODST, because, well Wes can back me up on this now, you work on Halo and all of a sudden all your little cousins, nieces, nephews who play video games want to tell their friends, which is flattering, don't get me wrong, it instantly moves you from creepy uncle who's only a head on skype to cool uncle who plays video games all day!

So anyway, as we were running through the level, I was pointing out all the bits of content I had touched and at one point my mom remarked "Wow, you do everything, they're not working you too hard are they, I mean, are you getting enough sleep?" As moms tend to do, and of course me being the good son, I didn't have the heart to tell her that you exchange your sleep privilege for the working in games privilege, but the thing it really hammered home is that we touch every discipline and most of the content that goes into a game, in some form or fashion, and this gives us a really unique and almost global understanding of how the production got where it is, be that good or bad. So now it's time for us to

take that step back, start looking at what all the commonalities are between the different phases of production and working paradigms, abstract those into a set of general ideas and concepts, and figure out how we can use those ideas to make us more efficient as Tech Artists, and in turn make our productions much more solid, starting from the beginning, because the reality is...

# "Work Smarter, Not Harder"



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9  
**2012**

...we need to work smarter not harder. We keep hearing it, and from my recent experience, it's true, the requirements and demand for content are increasing, cycles are decreasing, and overall the workload is just getting bigger, whether you're working on a small F2P online game or the next big console FPS. No coincidence that I mention those, because it's the disparity between those two projects that really crystallized for me how important it is for Tech Artists to be laying the foundation that enables this. There had come a point in the development of said next big console FPS where we were really starting to just be able to churn on Tech Art tools, you know, we had this great set of python apis, we had a really robust software warehouse, unit test suite, all these things were we could just write code, everything from big tools to one off scripts, and we knew it would work! To the point where I almost forgot how hard Tech Art could be...you know, got a little comfortable, put on some extra weight, you know how it is...

A few months later I found myself on the opposite end of the spectrum working on a PC F2P online title and was faced with that age old Tech Art task of batching every

character and animation in the game. SO of course I thought to myself, well how hard can this be, hell I used to do this in my sleep, actually still do sometimes just for fun, so I set off to writing the same python script I'd written a million times over you know import, constrain, bake save, wrap that all up nicely in a batch, profit right? Yeah so it was probably about 2 the next morning when I finally decided that, maybe things weren't going quite as well as I had imagined they were going to.

# Change Starts With Tech Art



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9  
**2012**

And you can imagine my shock, I mean I'd just come from a pipeline where people would batch whole animation sets of several thousand animations multiple times a day, to spending a whole weekend trying to reliably batch 1600 files, you can imagine I probably wasn't feeling great about myself...

So the disparity between these two pipes should be fairly obvious:

Good tools infrastructure, solid data and object models, common frameworks, etc,

vs

Character rigs all outsourced

But as I said, this was actually a big positive because it did in fact cement the need for infrastructure and how left unchecked, little oversights turn little issues into huge unmanageable issues for Tech Art. Now therein lies the rub, this sort of work that "only benefits" tech art immediately, sure that's true, but this is the sort of work that will continue to pay dividends throughout the project and in fact can

absolutely affect future productions. I feel like at this point in Tech Art, we all know how to write code, we all know how to build tools, we know how to setup the artist toolbox, but what seems missing to me is that we don't talk about how we setup the tech artist toolbox, and like I said, I feel like it's time to start stepping outside the specifics of our production experience and start not just generating abstract ideas and paradigms from that experience, but leveraging it as well...



# Solid Infrastructure==Solid Productions



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9  
**2012**

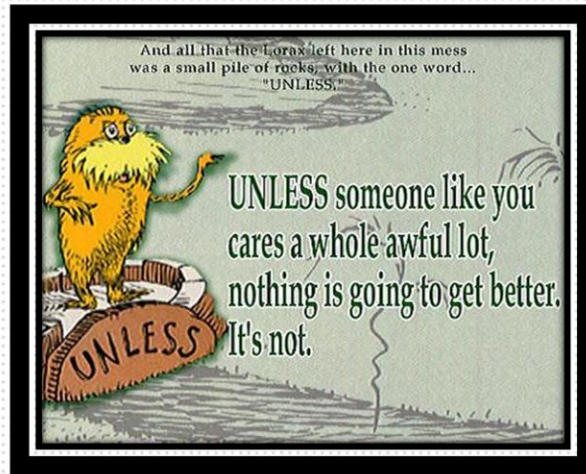
...and the way we do that is by using those ideas to set down a foundation for Tech Art to build the requisite tools and tech for our production. Now, I don't want us to confuse infrastructure with specific tools and pipelines, in fact, I'm not actually going to talk about much production code at all. What I hope you guys all take away from this is some ideas and tech that you can use to write your own code.

The reason I feel like this is a valuable topic is that well, everyone is writing foundation code in the Tech Art world today, from students and jr Tech Artists, all the way up to old men like myself, so I get the impression that everyone understands how to write tools and build pipelines for artists, but what I feel like we're missing is conversation around how to build tools and pipelines for Tech Artists. This can be a bit of an uncomfortable topic, because it really is the drab, unsexy work that you can't really convince a producer to let you do, and you're not gonna get that appreciative feedback from artists for writing the Big Red Button, and I know as Tech Artists, we like to dive in, we like to just get our hands dirty and go. But having worked on both ends of the spectrum, I can tell you right now that I don't think I could ever work in

an environment that DIDN'T have a Tech Art infrastructure, and I hope this is something we can all start to take to heart as a discipline if we aren't already, because otherwise...



# Improper Use Risks Slow Change

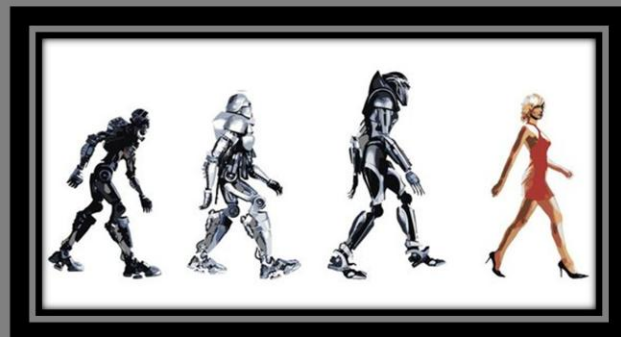


**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATES: MARCH 7-9  
**2012**

...we're doomed to repeat the same production missteps, but in a climate where mistakes are just continuing to get more costly. Make no mistake, when I say Improper Use, I'm putting the onus to correct that on US, that is not a statement of implied blame on production's part, management's part, it's all us, and I don't say that too piss anyone off too much, but just enough that maybe you're saying to yourself by now, "Well look at the big brain on Seth, fine smartass, what do we do?" Well, I'm glad you ask...

# Stop Scripting, Start Programming



Flashback to about 2007 and you might remember that this little technology called “cobra” or...“viper” or...Python that’s it had been introduced into to Maya a little earlier and was starting to get a bit more uptake in games. I feel like this sort of threw a bit of a wrench into the whole idea of what Tech Art was as a discipline and for a little while there, it really felt like there were going to be battle lines drawn between the Art Tech Artists and the Tech Tech Artists, though that could have just been my flair for the dramatic, but, I know Rob and I have ranted to each other about this a lot, that whole idea that “Oh I’m a tech artist, not a programmer, so I don’t need to learn the Python standard library, I’ll just write my own string functions”...no seriously, this is a conversation I’ve had...

Jump back to today though, and I feel like things like this have been pretty well settled and the idea of Tech Artists as programmers is definitely not quite the foreign concept it was back in those dark times. So That’s the big overarching premise here is that we are very much software engineers nowadays, and we should start thinking, acting, and working like software engineers.

# The Need For Tech Art Infrastructure

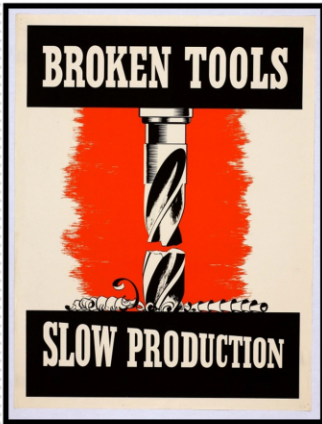


GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

So my story about the character batching incident is a good testament to the need for Tech Art infrastructure, but as I've mentioned, we need to broaden our FoV a bit and look at how the lack of infrastructure can really affect production at all stages and all levels. To be honest, not being able to batch characters in a specific instance isn't horribly egregious, and really I was just feeling sorry for myself, right, but if we think about the implications of that, the issues probably tend to manifest themselves in your minds, yeah? For example, not having that infrastructure means we can't quickly iterate on character rigs, which means if we need to add features or recover animation, it's a time consuming process, and what happens when we're up against a deadline and we have a show-ish stopper? Not a situation people aren't unfamiliar with, but certainly one that could be solved by...proper infrastructure.

# Bad Infrastructure Creates Bad Tools



One-off tools and scripts can often obfuscate both the pipeline and content.

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
EXPO DATES: MARCH 7-9

**2012**

At some point in your career you may have written a really quick, dirty, one off tool, in the privacy of your own cubicle and of course you washed your hands afterwards, all the while thinking to yourself “ah, it’s just a one off, we’ll just fix this content and call it good.” This in and of itself isn’t a bad thing, and it’s not even a bad thing if you write a lot of one-off tools and scripts. What differentiates between good and bad one-offs is the underlying framework on which the tools are built. Much in the same way that we like to think of/design UIs in our tools to be layers on top of separate core functionality, we can abstract that idea out to tools in general, what we call tools should really just be layers on top of our infrastructure and frameworks that manipulate content and data, and it’s bad infrastructure that keeps you from writing those one-off tools while preserving process and data integrity...



# Bad Tools Create Bad Pipelines



Poor pipelines increase complexity while reducing flexibility and scalability

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
EXPO DATED MARCH 7-9

**2012**

...what ends up happening instead is that without that roadmap of infrastructure to follow, each tool potentially becomes its own little mini-pipeline with its own data model and execution patterns, so as we write more of these little one-off sovereigns, we end up diverging the content and data paths so much that when we try and wrangle things back in, we end up with these huge tools that are trying to account for every one of those little forks in the content path we created and so we get this thing that's more akin to a rube goldberg device than...say a waterslide, we're standing at the top dropping content into one end and hoping it comes out one of the chutes that we can see the end of. And while that's all well fun, or at least it sounds fun, the real tragedy here is that we've totally engineered flexibility out of the pipeline because our content has to be conditioned so specifically, and likewise scalability, because even small changes require massive amounts of code, again to account for each one of these diverging content paths from all of our pipelines-in-a-tool

# Bad Pipelines Create Bad Content



At worst, we end up with bad content at both ends of the pipe

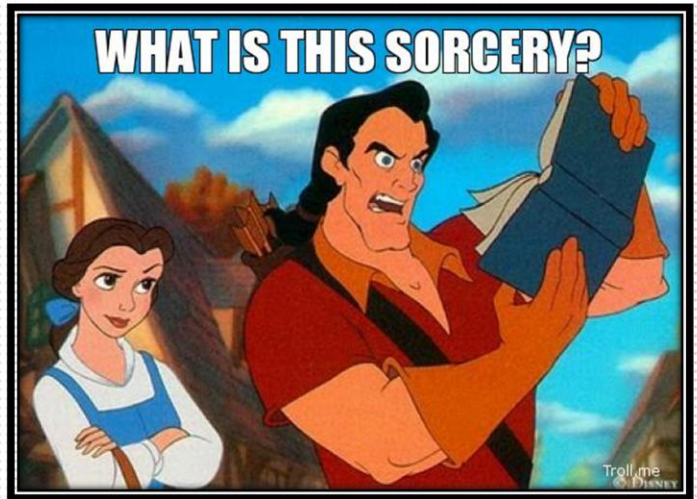
GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
EXPO DATED MARCH 7-9

**2012**

The worst case scenario is that we end up with pre-pro content that we can't recover or otherwise use, which isn't too terrible, you want to have that sort of flexibility in pre-pro, but once we move into production sans infrastructure, we get content that's a bear to debug because we can't be sure of every thing each individual tool introduced to the content path, and at the end of the cycle, we end up with content we can't optimize because again, there's no way to trace back through the steps other than to read the code, and I know, none of this is a really huge deal for now, but remember what we're facing, we're facing a new hardware cycle where our ideas of what content is may completely change, for example, suppose we do make the big jump to procedural authoring. Now we've got "content" that doesn't exist in a state that remotely resembles what we think of as content today, and if we try and wrangle that with a toolchain that has no common frameworks, well...I'd prefer not to imagine what that might look like.

# So, What Is This "Tech Art Infrastructure"?



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

Ok so now that we hopefully have a compelling case for why we need Tech Art infrastructure, we should talk a bit about what it is that we're actually trying to build, and I think that can be best illustrated by talking about some of the benefits. And I know, this is one of those things I feel like as Tech Artists we all know, but the reason I want to touch on this point is because having these conversations are an integral of building Tech Art at a studio. You may at some point have to sit down with Art Directors, production leads, etc and talk about why it is that you need to go dark for a 3 month period wherein you're not doing anything immediately beneficial to the art team.

So that said, I'd like to frame this more by discussing some of the benefits versus trying to directly answer the question, because ultimately we want to be able to spin this not as a "what" but more as a "why" or a "how", that is why should we do this, how does this benefit the production as a whole. Taking this sort of approach presents compelling arguments versus simply saying "This is a thing, we need to do it", because when I present it like that, what's the first thing you're asking? Why...



# Good Infrastructure Begets Good Pre-Pro



Good infrastructure creates a good  
GAME

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA

MARCH 5-6, 2012

EXPO DATES: MARCH 7-9

**2012**

So all the bad news, fire, and brimstone from the repercussions of bad infrastructure behind us, now we can focus on happier things, like how a good infrastructure can not just make life for Tech Art, but can in fact affect the whole production all the way through ship. This actually has to do with how we think about/approach pre-production, as well as what we actually end up implementing, and the way to think about that is to never lose sight of production. One of the...I don't want to say mistakes, but maybe one of the less pragmatic approaches we tend to take in tech art sometimes is to get caught up in that whole headiness of pre-production along with our friends on the art team, so we do things like create these amazing shaders for example that in no way are going to run at frame rate, but it's cool because it's pre-pro and it's throwaway...until it's alpha and we have to ship with it so were stuck trying to optimize this thing that...really wasn't built to be optimized. But if instead, we approach everything with that infrastructure first mentality, sure we can build those crazy shaders, but, we'll build them in such a way that we can take it apart and reconfigure it when we need to get some cycles back. The same high-level paradigm should apply to our pipelines, and of course that starts with...good



infrastructure.

# Good Pre-Pro Begets Good Production



Proper pre-pro means we've answered our production questions

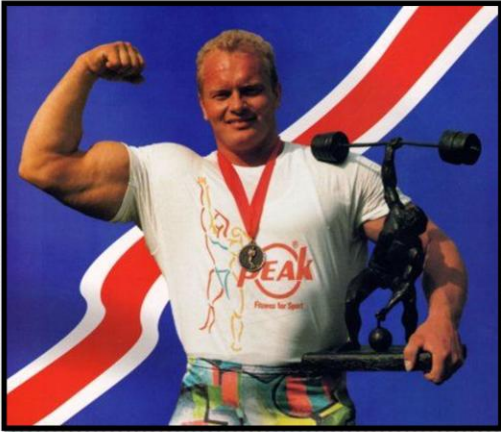
GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
EXPO DATED MARCH 7-9

**2012**

Now, if we've gone through pre-pro infrastructure first, we set ourselves to go into production in such a manner that we should have a pretty good idea of what kind of game we're making, which means we know what content we're going to be building and from this we should have an idea of what the pipelines might look like, and while that might not be enough for us to dive in and start building tools yet, it puts us in a position where all the big questions should be answered and we can at least start asking the smaller questions now, like...what should the tools look like. And if we've taken the opportunity in pre-production to start putting down some very rough infrastructure, that being gathering external libraries, experimenting with different SDKs and patterns, I think we'll find that developing our production infrastructure becomes a fairly simple task, along with our tool and pipeline development.

# Good Production Begets STFG



Good production means we finish strong and start again stronger

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-9, 2012  
DVD DATED MARCH 7-9

**2012**

And if we've gone through production properly, again with that infrastructure-first mentality, not only do we Ship The Freakin Game, hopefully with minimal casualties, but we leave ourselves in a position to hit the ground running on the next game. Building a new pipeline and toolset just becomes a matter of...writing more one-off scripts against our battle tested infrastructure, and if our infrastructure was such that our content pipeline wasn't the divergent, branching nightmare from our previous conversation, we even set up art and design to start in that same strong position, since they'll have tons of content that they can easily strip down and re-purpose for the next project.

# And Who Owns The Product?



GAME DEVELOPERS CONFERENCE 2012

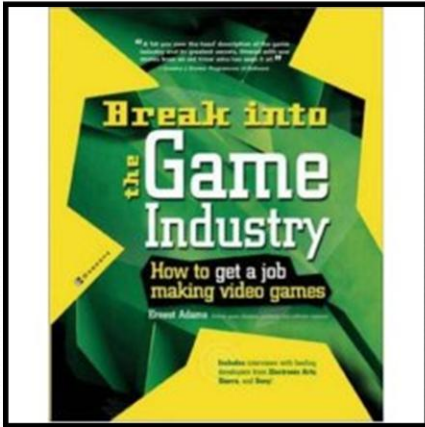
MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

Alright so before we dive right in, there's one more sort of...I guess we could call it a logistical issue we need to address and that's where the accountability is going to fall. Someone needs to be the go-to guy for all this development, but more than that, someone needs to be a vision holder. Vision is definitely not just heady stuff for artists and designers, we can certainly benefit from having our own visions of what not just what Tech Art should be, but again, why we're doing it, and how it's going to help us ship the game, see, there's that why and how again.

The key point for all this is leveraging proper experience properly. Now, I'm not going to promise that this going to guarantee that as Tech Artists, we're always going to get to do all the work we want to do all the time, but I do believe having an advocate in place on the team can help foster proper advocacy outside of the team, so hopefully as your team moves forward, you WILL be doing more of the work you want to be doing more of the time.



# "But Mom, Technical Art Director IS A Real Job..."



We should really be managing ourselves, just like every other discipline.

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA

MARCH 5-6, 2012

EXPO DATES: MARCH 7-9

# 2012

Now, this whole effort starts with the Technical Art Director. From my own experience, Tech Art Directors is one of the most understood roles in Game Development, evidenced by the fact that a lot of studios don't have Tech Art Directors, and a lot of studios hire junior Tech Artists before hiring a Tech Art Director...which is a whole other point of contention, and I don't mean to disparage anyone's experience or effort, like I said, it's actually knowing some of the work that junior and student Tech Artists are doing now that motivated me to move away from a hard skills technical talk to more of a soft how-to-deal-with-producers talk.

But it's also my personal experience seeing how the lack of a properly respected Tech Art Director can adversely impact production. I've certainly seen small problems become large problems simply by virtue of the fact that they weren't allowed to be addressed when they were small and manageable by the people who were best suited to address them...Tech Art.

# The Tech Art Director Is Not...



...Just More Experienced  
...Just A "Bigger Hammer"  
...A Job For Engineering Leads

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9

**2012**

So what IS a Tech Art Director? Well, let's first look at what a Tech Art Director shouldn't be. The biggest fallacy I've seen is putting the Tech Art "Director" in a role that's really more akin to just a more experienced Tech Artist. You tend to see this a lot at companies that start by hiring junior Tech Artists to fill the gaps, not to disparage junior Tech Artists at all, but the expectation becomes that a more experienced Tech Artist is just someone who can solve bigger problems faster, but is never really given that authority to set down the parameters of those problems. The issue there is that Tech Artists are suited to tackle a very unique set of problems, which like Tech Art itself, doesn't fall entirely in the domain of art or engineering, so what ends up happening is a potentially production-changing resource is not leveraged properly often times at the expense of the production. And that doesn't make anyone happy.

# The Tech Art Director Is...



...Part Production Artist, Part Software Engineer  
...A Peer To The Art And Engineering Directors  
...Focused On The Needs Of The Production Before  
The Needs Of The Art Team

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
EXPO DATED MARCH 7-9

**2012**

So we still haven't answered the question, and sadly it's not as simple as just taking what a Tech Art Director is not and flipping it around. A good place to start is with the idea that Tech Art is a bridge, that Tech Artists have feet in both the disciplines of art and engineering. That said, a good Tech Art Director needs to be both production artist and software engineer. Now, within the lower ranks, it's probably permissible to be more focused in one direction, but by the time one gets up to the directorial level, you really need to be able to not just understand the conversations on both sides of the fence, but you also need to be able to contribute and even push back. The corollary to that is that the Tech Art Director needs to be seen as an equal in management to both Art and Engineering Directors, as opposed to this catch all for the whims the two. Ultimately, the Tech Art Director needs to understand that the needs of the many outweigh the needs of the few, or in this case, sometimes art has to take a back seat to the overall scope of production.



# Building Blocks



As we're all probably aware, a solid foundation begins with good construction materials, and a Tech Art infrastructure is, of course, no different. It should go without saying that one of the biggest points we've gained from moving to more widely accepted programming languages like Python and C# is that now we can leverage both experience and code from other developers. And I'll admit, I do sorta miss those heady days of MEL development where we were all writing our own big libraries and sharing stuff on highend3d, not to date myself horribly, but overall I feel like what we've gained from no longer being a sovereign nation far outweighs that old national pride, and certainly benefits our artists and productions to a far greater degree than we ever could have done by ourselves...(not to slag on anyone's individual abilities or work ☺)

So let's take a look at some of the tools and tech we can use to build up our infrastructure, and standard disclaimer applies here, this is not the best way to build up an infrastructure, these aren't the only resources you could use, these are just a few things I've picked up in my travels that I



think are useful, and in fact, I'd love to hear from anyone else during or after the conference who's got their own takes on infrastructure, because after all, we get better as a discipline.

# Always Use Protection

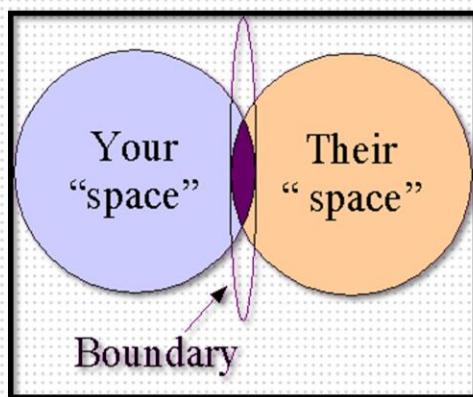


GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

So the first thing we need to consider is how we're going to manage development. By this I mean how are we going to keep multiple developers working on different tools and features in sync with each other, and moreso, how are we going to keep artists in sync with all those changes, or the flipside, how do we keep them off the bleeding edge until we're ready? The time honored pattern of just having a custom folder somewhere in source control that we manually point our apps and scripts towards has served us well, but current software development practices and technology certainly afford us better, and we shouldn't be afraid to explore that brave new world. So let's take a look at some of these technologies and see how we might be able to leverage them to the construction of our dream infrastructure.

# Keep Proper Filters In Place...



"Sandboxing" provides a secure means for Tech Artists to iterate "off-line"

GAME DEVELOPERS CONFERENCE

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

**2012**

## ...And Open The Valve Slowly



We can also control who gets what changes when

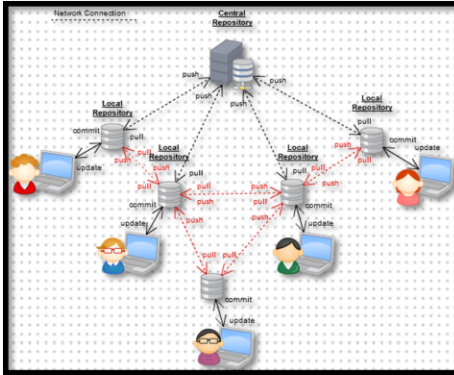
GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9

**2012**

Another advantage of sandboxing is that now we can selectively push changes to one or several “trusted partners” on the art team when we want to test changes or iterate on a new tool. Rather than just submitting a big changelist, hoping that our own personal testing covered all the bases or at least the common use cases, and making the whole art team drink from the firehose, we can choose to push a file, a folder, or a whole sub-structure to any artist. Given that, you can imagine how having multiple tech artists iterate on multiple features in different parts of the pipeline now becomes quite feasible, so this is absolutely an idea worth exploring. There are several source control packages that allow this sort of workflow, and in fact, if anyone wants to take some of the ideas here and test them in perforce’s new sandbox tech, I’d love to hear about your experience.

# Case Study: DVCS And Symlinks



- Use A Branching Scheme To Develop And Test Features In Isolation
- Symlink Individual Artists To These Changes For Test
- Use Hooks (if supported) To Manage Distribution

GAME DEVELOPERS CONFERENCE

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

**2012**



## "Don't Let It Go To The Judges"



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

Now that we have this pristine sandbox architecture to start playing in, let's do that! One of the major tenets of any infrastructure is that it should be flexible, or it should scale, or it should be able to evolve with the needs of the production, preferably all of the above. That means we shouldn't be afraid to adopt new technologies or development paradigms, and in fact in my experience, this sort of sandboxing paradigm actually lends itself to that really well, for example, it's no probably no surprise to anyone that I'm still a registered-card carrying Pymel user and I remember when it first came out, the big argument against was its lack of official support. Softwarehousing really lets you mitigate that sort of fear, because it makes it very easy to pull in and push off external libraries and tech.

That's an extreme and specific example, but there's definitely a high level gain we can take from this. As my buddy Dana White likes to say, it's all about imposing your will, don't leave it in anyone else's hands, don't let it go to the judges, and that parallels this, the idea being that, don't leave your development efforts in someone else's hands if you don't have to, and this isn't NIH advocacy at all, what I'm saying is

own your own bugs and fixes, and having a softwarehouse lets you do that, so you don't need to be as apprehensive about polluting a working environment with external code, and in fact, you're setting your infrastructure up for growth.

# Don't Reinvent The Wheel...



Tech Art should focus on solving the problems that HAVEN'T been solved yet

GAME DEVELOPERS CONFERENCE

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

**2012**

The most obvious way we can leverage softwarehousing is by bringing the libraries we want to use in development in house and maintaining them ourselves. Like I said, we don't need to be writing XML parsers, or math libraries, or string management functions, this all exists, but at the same time, as I said, we want to put ourselves in a position where we own our bugfixes and features. A fun side effect of this is that it can be use as an educational practice as well as being a good development practice, you know, one of the things I like to tell people who are looking to improve their chops in any programming language is to read other people's code, or add features, or port stuff to between languages, otherwise get your hands into code that isn't yours. And I'll warn you in advance, it won't all be good code, but at least it'll get you thinking. And as Tech Artists, we should always be learning and thinking, maybe not in that order.



# ...Build A Better Car Instead



"Softwarehousing" gives us a solid base to build our own foundation

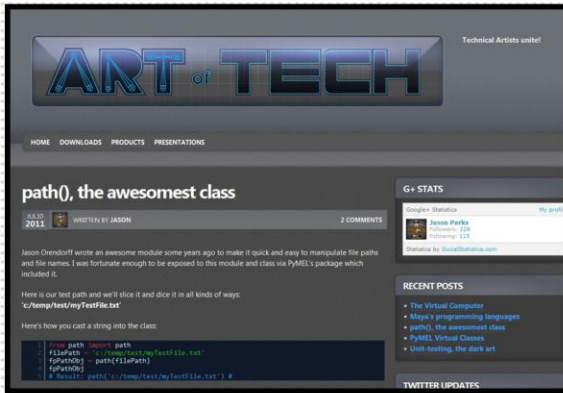
GAME DEVELOPERS CONFERENCE

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

**2012**

Another benefit of softwarehousing is that we have a good base of production tested libraries to build our own libraries upon. This is that idea of scaling, customizing, and otherwise adapting our infrastructure to the specific needs of the project as we get further along, for example, maybe there's a specific pattern you use a lot that you need to optimize better, or maybe there's a specific feature in an alpha or beta branch that you want to merge and build on. By keeping your own cut of whatever libraries you're using, you've got that freedom now. And I won't lie, this is real software engineer work, not necessarily for the faint of heart, and again, this is some of that unsexy work you're probably going to have to do on your own dime, but I feel that the benefits to the production, the Tech Art team, and you as a developer far outmatch the cost of adoption and upkeep.

# Case Study: Orendorff's path



path is a simple module that's no longer actively supported but provides some powerful path manipulation tools. Grab it from github and modify it to your needs

For more on path, see Jason Parks' post on ArtOfTech:  
<http://www.jason-parks.com/artoftech/>

GAME DEVELOPERS CONFERENCE

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

2012

# Choose Your Weapon



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

This next topic is a little weird because I almost feel like at this point it should go without saying, but at the same time, I can recall as little as 8 months ago trying to convince people to use a full-blown IDE to write code. Of course, therein lies the issue, in that, we all think that everyone just knows why you should be using an IDE, but we don't often explain compelling reasons. Things like autocomplete and debuggers aside, there are quite a few other features that really make modern IDEs attractive to Tech Artists, and these are probably things you're already aware of but may not have given too much thought to, but with careful planning, your IDE can actually become your one-stop development shop, so to speak...

# Use The Right Tools...



notepad is only useful because it's simple  
and you already know how to use it

GAME DEVELOPERS CONFERENCE

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

**2012**

Let's be honest guys and gals, notepad is only useful because it's simple and you know how to use it.  
We provide specialized tools to art teams to create content, we should follow our own lead!



## ...To Get The Job Done Right



A proper IDE lets us shorten our code iteration cycle and do away with extra tools

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA

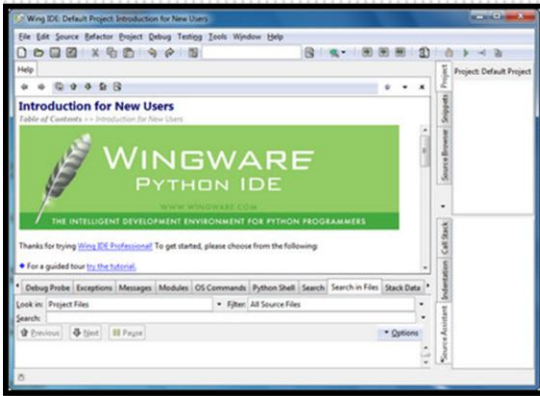
MARCH 5-9, 2012

EXPO DATES: MARCH 7-9

# 2012

A proper IDE lets us shorten our code iteration cycle and do away with extra tools. Why not just test your Maya code in the IDE instead of having to fire up Maya or deal with restarting it because of that infinite loop you totally meant to take out before testing?

# Case Study: IDE Features (Wing)



- Sharing Project Files
- Source Control Integration
- Unittesting
- Documentation Builds

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA

MARCH 5-9, 2012

EXPO DATES: MARCH 7-9

# 2012

Project files, etc

# Laying The Foundation



Alright, so continuing my clumsy use of construction metaphors, now that we've got our building materials, it's time to start laying some foundation. In this case, we have some sort of ecosystem that we can start putting our infrastructure into, so similar to the preceding topic, "foundation" can be thought of less as specific code and tools and more like a suite of technology that we can use to build said code and tools, in fact, I want to keep impressing on you guys that when we think about "infrastructure", we need to be very careful to not confuse infrastructure with tools or pipeline.

I like to use the term foundation because in Tech Art infrastructure land, the sort of ideas we're talking about really do provide that layer for solid, unambiguous development, but at the same time, it's the sort of thing that can continue to evolve along with your infrastructure, while evolving your infrastructure itself...



# So How Does This All Work?



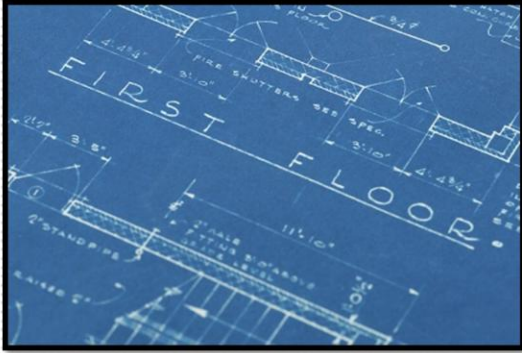
GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

Alright, so rubber meets the road moments here, let's say you're the Tech Art Director, and you convinced your peers in management that you need a few months of blackout time to lay some pipe for the upcoming project, and you've got everything in place, so you're good to go yeah? You're just gonna check a bunch of code in, get everyone set up on the IDE project, and start assigning tasks! Well, hopefully it really does go that simply, and with just a little more forethought and work, you can ensure that happens.

As I've mentioned a few times, one of the things we really want to try to do with our infrastructure development is minimize ambiguity, and one of the first steps to making sure that every developer who's going to be working in our sandbox is on the same page is good documentation.

# Teach Yourself...



Writing documents can often show you gaps in your own knowledge

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
EXPO DATED MARCH 7-9

**2012**

Now when I talk about writing documentation, I don't mean putting it on the wiki or whatever other systems that often get set up with the idea that people are going to update it and of course people are going to read it, right? Because everyone reads the freakin manual, and I'm sure you guys who have worked with Wikis or any sort of other communal documentation know that they come with...let's say varying degrees of success. No, I'm talking about dedicated documentation systems like Robodoc, or doxygen, serious documentation generators that use markup languages and hook into IDEs and build processes but produce professional looking documentation. As a Tech Art Director, lead, or otherwise an infrastructure builder, writing documentation should be a required task, if nothing else for the educational benefit. I remember when I started writing a style guide about 9 months ago, my thought was, "Oh this'll be easy, I'll take some of the google style, some PEP-8, change a few things that I don't like, and we'll be good to go"...and it's when you actually step back and try writing code against your style guide, you start to realize it's not that easy, so...you know if you ever feel like you need some humbling and want to know exactly what you don't know about a language, try writing a

style guide.

# ...As You Teach Your Team



Proper documentation makes all the difference in adoption and shaping opinion.

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATED MARCH 7-9

**2012**

The benefit to the Tech Art team is obvious, now they don't have to read code if they don't want to, although I'm a huge proponent of the idea that good code documents itself. Documentation however, is more than just function signature, arguments, description, save that for the docstring. Documentation ensures that everyone on the team is using everything the same way, which means anyone can jump into anyone's code if need be and maybe not own it, but at least maintain it in a pinch. Another advantage of documentation is that it smooths the introduction of new ideas and paradigms to the team. I've actually heard in really severe cases of people who had formed poor opinions of whole paradigms like markup schemes or metadata because they were forced to work inside of a system that relied on a very cryptically implemented feature with no documentation, and I'll admit I've been guilty of this myself in the last couple of months. And therein lies the downside, once you invest in becoming a good documenter, you have much less tolerance for poor documentation, which sadly seems to run rampant in software land...

# Case Study: Sphinx



- Fully customizable
- rST based, minimal coding required
- Supported by many IDEs
- If you've seen the python docs, you've seen Sphinx in action

GAME DEVELOPERS CONFERENCE

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

2012



# How Do I Know It Will Work?



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

Now I've written a lot of tools over the course of my career and I've debugged all kinds of code and content, and I don't know about you guys, but given some of the things I've seen and had to deal with, there have been times and pipelines in my life in which I was genuinely afraid to release tools to the wild, I believe it was sometime month.

In all seriousness, it can definitely be nerve wracking to push tools out, and it seems like that grows along with the size of your checkin, right? Well, fear not, because we have another weapon in our arsenal aimed specifically at alleviating this situation, especially when combined with some of the other paradigms we've already discussed, for instance, we've talked about how to keep external code from polluting our working environment until it's been properly sanitized, we've talked about how we can keep test and iteration to a limited number of trusted partners on the art team, so now we can start thinking about how we keep error overall down when we're developing, and like everything we've talked about here, we start at the beginning, we start when things are small and manageable by testing our code little bytes at a time, or in what they call units, hence the term...unittests, exactly.

# Catch Bugs In Your Design...



By testing simple code units, we squash bugs when they're small

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-6, 2012  
DVD DATED MARCH 7-9

**2012**

Ah unittesting...if there was one practice that I could say changed the way I write software, it would absolutely be unittesting, and the way I came to unittesting was actually what really exemplified to me why you should use unittest. You see, normally when you write unittests, you do it at the beginning of development, where you can use it to design your interfaces, prototype common use cases, that sort of thing. I, on the other hand, started by writing unittests for code that was already in production and I had a good working knowledge of. So it was one day when I was writing these unittests that I realized I was writing tests that I knew would pass, because I knew the code worked, and about an hour into writing a test, I realized exactly what I was doing, that was I was writing my setup methods to create specific environments in which the test would pass, and the complexity of my test cases made me realize that...aha!...this code is too complicated. So from an infrastructure standpoint, this is one of the things we use unittests for, we don't necessarily use it to catch bugs (well, not all the time), but instead we use to keep our interfaces simple, and in doing so we provide an extra layer of documentation, in the form of common use cases.



# ...And You'll Have Fewer Bugs In Your Tools



Unitesting gives us the opportunity to design the bugs out of our code

GAME DEVELOPERS CONFERENCE

SAN FRANCISCO, CA  
MARCH 5-9, 2012  
DVD DATED MARCH 7-9

**2012**

They say it's always good to end on a high note and not leave your audience with negative thoughts, so I'd like to wrap this up by talking about what happens when things go wrong. Best laid plans and all, you know the drill. Reality of development is sometimes things just don't go the way you think they should, it worked on your machine, you checked everything in, but for some reason now the art team is blocked and your only recourse is to roll back and go through your change line by line until you find the proper debug spew...

# Case Study: Unittest Content

**You Got Art...**



**...In My Test Suite!**

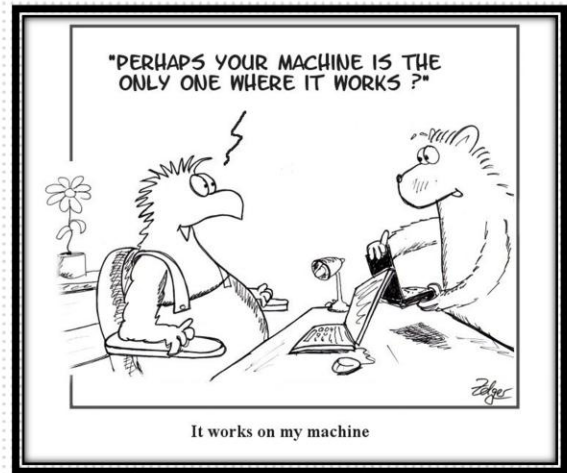
- Leverage your DCC app's ability to access scene objects and files through Python
- Build a "known good" set of content and pull it into your TestCases
- Come to my poster session and I'll tell you more!

GAME DEVELOPERS CONFERENCE

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

**2012**

# What Happens When It Doesn't Work?



GAME DEVELOPERS CONFERENCE 2012

MARCH 5-9, 2012 [WWW.GDCONF.COM](http://WWW.GDCONF.COM)

They say it's always good to end on a high note and not leave your audience with negative thoughts, so I'd like to wrap this up by talking about what happens when things go wrong. Best laid plans and all, you know the drill. Reality of development is sometimes things just don't go the way you think they should, it worked on your machine, you checked everything in, but for some reason now the art team is blocked and your only recourse is to roll back and go through your change line by line until you find the proper debug spew...

Or is it?

Well hopefully it isn't. Everything we've discussed up to this point feeds into the idea that when accidents happen, we can move through them very quickly, because we've set ourselves up to build tools in a known environment, and part of that known is knowing what's going to go wrong. I know that sounds crazy, but think about what we've done up till now, we've effectively wrangled our development effort such that we've made it very easy for use to isolate bugs to code or content by using things like unittesting and sandbox development, so the last little thing we need to do is setup

some infrastructure to funnel errors properly and make sure that all the issues we've worked so hard to suppress up till now remain little issues.

# Keep Your Error Logs Close...



Custom error handling goes a long way to removing ambiguity in debug

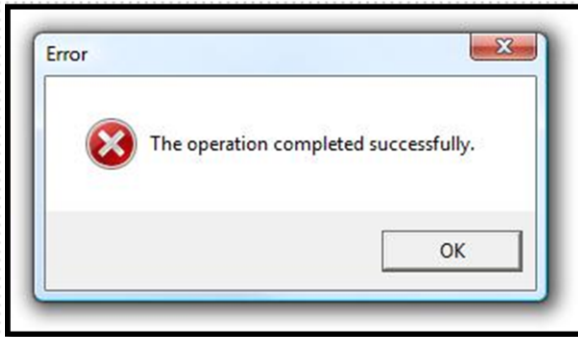
GAME DEVELOPERS CONFERENCE

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

**2012**

The easiest way to know for sure how to handle an error is to setup a system wherein you always know exactly what the error you're handling is. Sorcery? Blasphemy? Well, not quite. By coupling a good logging API with custom exceptions, we can pretty much catch, handle, and redirect any undesirable results in such a manner as to be able to provide USEFUL feedback to both the user and the developer. Since we went to the lengths of setting up unittests, we should also be able to pare out the more common cases of built-in exceptions, which we could also handle ourselves. So for instance, we have a bit of functionality that we know is going to raise a `ValueError` sometimes in situations that may be beyond our control (just for the sake of this conversation), based on our unittest. Since we know what the case is that raises that exception, we could create our own subclass of `ValueError` that handles our specific case and returns useful data. Obviously this is a very naïve and ideal situation, but you get the idea.

# ...And Make Debugging Easier For EVERYONE



Knowing what we're handling makes  
for easier debugging

GAME DEVELOPERS CONFERENCE

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

**2012**

...And with all that in place, we've now created an error reporting structure that starts with a user knowing exactly what's happened, and maybe even how to talk to Tech Art about it. Once Tech Art steps in, we can very easily look at the traceback and know that we've caught one of our own exceptions. Couple this with our carefully built development sandboxes, and we can iterate with the affected artist directly, off-line, to resolve the issue while the rest of the team continues to work. We fix the problem, we merge that fix into the head branch, and production rolls merrily along.



# Case Study: Email On Exception



- Setup custom logging levels
- Customize distribution lists and message content based on each level
- Creep artists out when you show up at their desks unannounced right after they error

GAME DEVELOPERS CONFERENCE

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

**2012**



# "Work Smarter, Not Harder"



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATES: MARCH 7-9  
**2012**

So to wrap things up, we've talked about a lot of different ideas today, but the reality is, all the specifics of infrastructure building and whatnot really don't carry the weight they should if we don't keep our overarching premise in mind, so let's never forget that the main impetus to all this is to do more work with less effort, and I don't mean to keep bringing up the spectre of 4<sup>th</sup> gen, next-next gen, 720land, whatever you know it as, to scare you guys, but the reality is, more content, more production, less time, but if we as Tech Art get in front of this at the right times in production with the right ideas, that idea of more content in less time doesn't necessarily have to translate to more work for more bodies with more crunch...

# Change Starts With Tech Art



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATES: MARCH 7-9  
**2012**

And that's really where it needs to hit you guys, in Tech Art departments all across the industry. I hope if anything I've put the seed in your minds that, we really can be the keepers of production and we really are in the best place to start or continue to affect change in our pipelines and productions, by stepping back and taking a look at the pipeline as a whole and figuring out those big high-level points we need to address, and really, we do that by...

# Begin At The Beginning



**GDC**  
12

GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-9, 2012  
EXPO DATES: MARCH 7-9  
**2012**

...going back to square one if need be, and hopefully we aren't afraid to do that, hopefully we're not afraid to say, "No, we need to pull this up by the roots", and hopefully we have some ideas about why we need to do this, and how it's going to affect our production and how we're going to do it better. This way when we go to our peers in development and management, we can instill in them the same confidence we have in why we're trying to do what it is we're trying to do, and then maybe just maybe, that idea of beginning at the beginning so we can come out of it by working smarter not harder won't just seem like a clever statement full of silly middle-management buzzterms.

That's all I've got, thanks for listening, questions?

# Questions?

[seth.gibson1@gmail.com](mailto:seth.gibson1@gmail.com)



[linkedin.com/in/sethgibson1](https://www.linkedin.com/in/sethgibson1)



[facebook.com/djTomServo](https://www.facebook.com/djTomServo)



[twitter.com/voMethod](https://twitter.com/voMethod)



GAME DEVELOPERS CONFERENCE

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

**2012**



# JOINING THE DARK SIDE

How Embedded Tech Artists Can Unite Artists and Programmers

Ben Cloward  
Senior Technical Artist  
BioWare

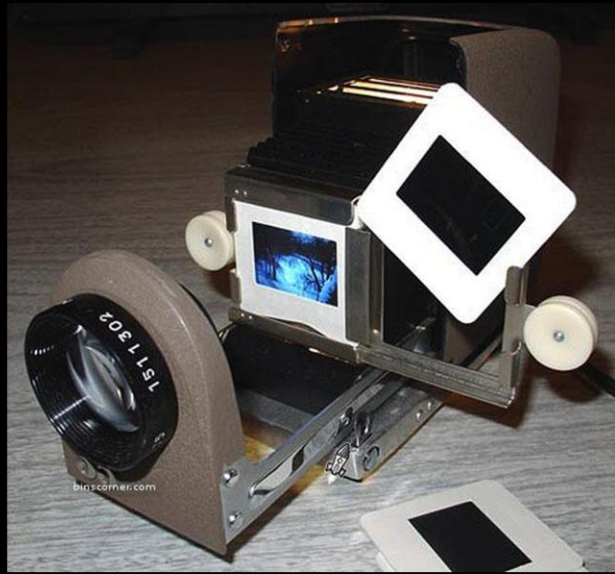
For the last couple of years at GDC, I've given talks that focused on technical challenges that the team at Bioware faced as we created Star Wars: The Old Republic. Last year I talked about the Cinematic Lighting in our game, and the year before that I talked about our automated system for facial animation and lip sync. Technical talks like that are really exciting to give and they're also exciting to listen to. This year, I've chosen to talk about a different kind of challenge that we faced at Bioware - and this is more of a social problem than a technical challenge. While a talk about relationships between people may not seem as

cool at first, I hope that you'll stick with me and appreciate by the end that what I'm going to talk about this year had just as big an impact on our project - if not greater - than some of the technical hurdles we faced.

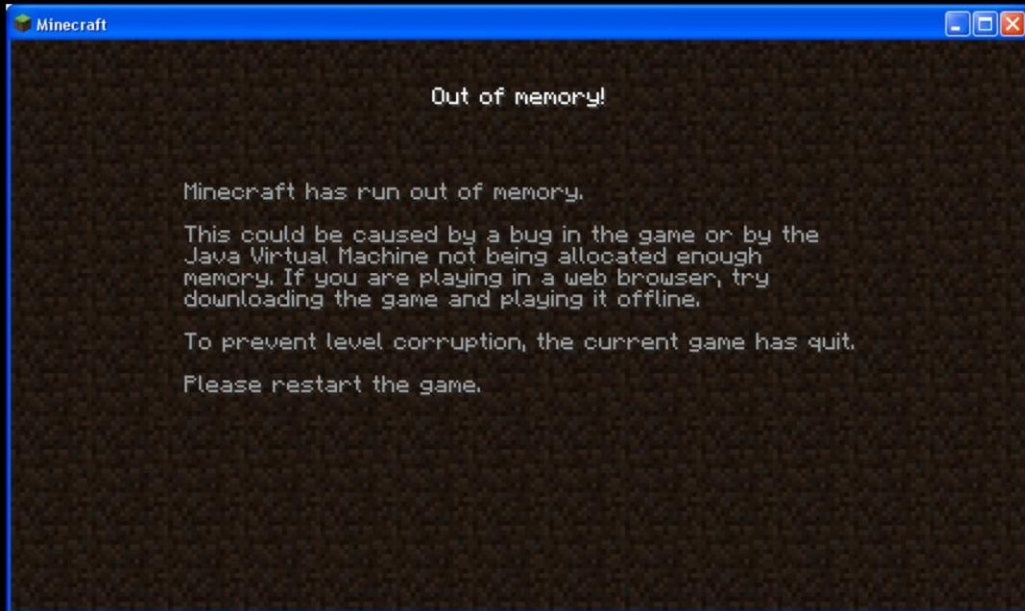
# The Problems

So we were a couple of years into our project and we had several problems. First of all, the performance of our game was pretty poor.





The frame rate was low



and the memory usage was high.



# BLAME

THE SECRET TO SUCCESS IS KNOWING WHO TO BLAME FOR YOUR FAILURES.

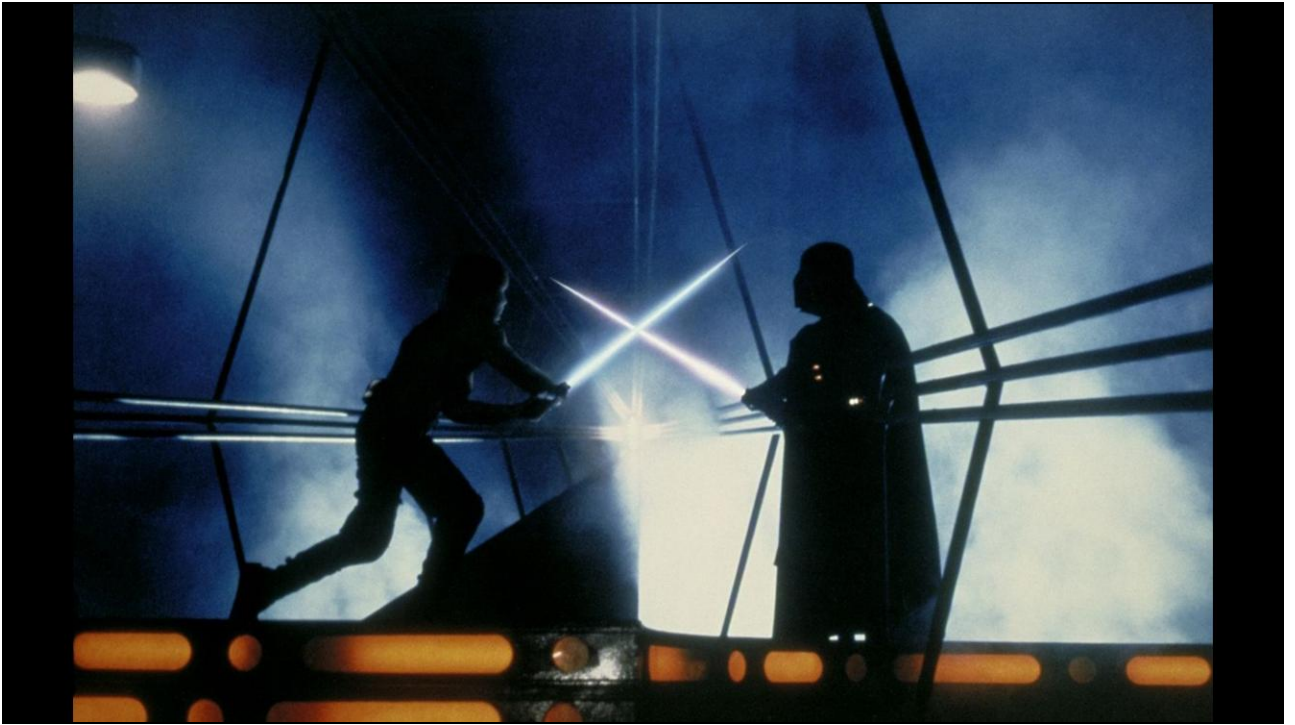
Compounding that technical problem was the trickier issue that our artists and our programmers both considered these issues to be the responsibility of the other department.



Our programmers considered the art in the game to be the main cause of the performance issues because it was implemented in an inefficient way.



The artists, on the other hand, felt that getting the frame rate up was the job of the programmers, and they were frustrated at the programming department for delivering tools that were difficult to use and that caused a lot of headaches and lost time.



As you can probably imagine, this type of a relationship



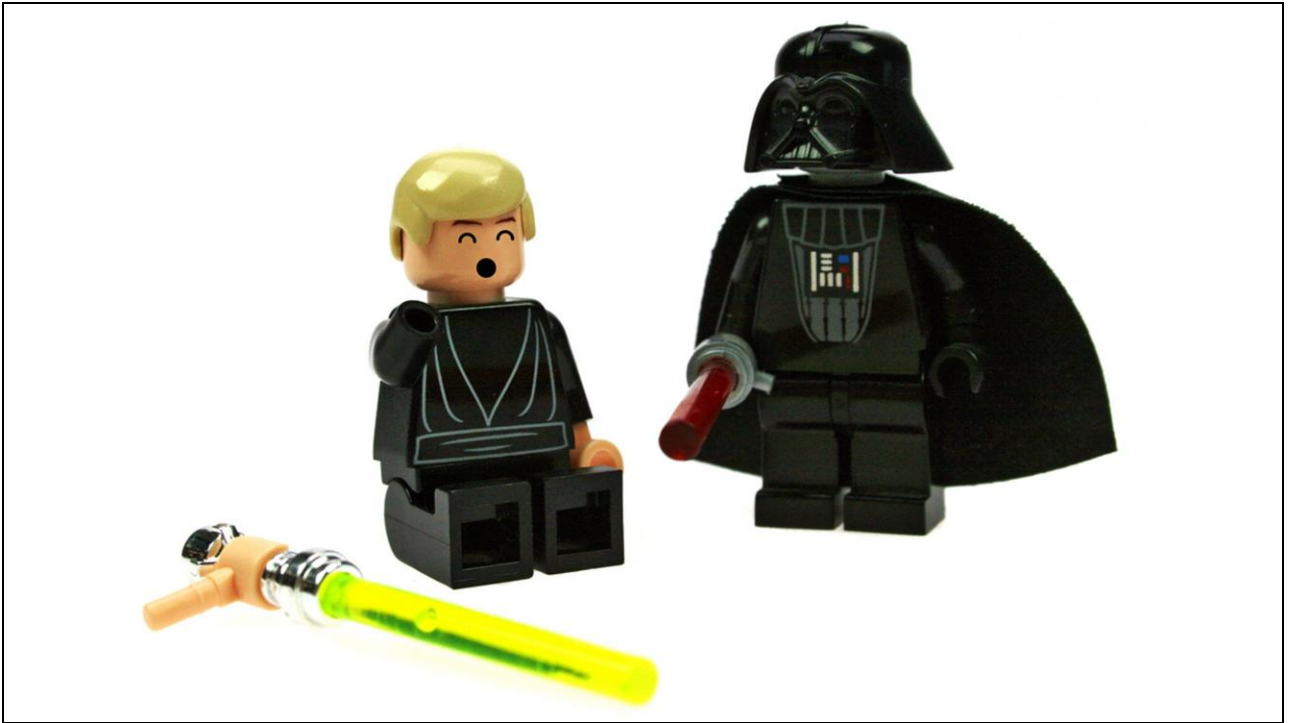


between departments





was not very conducive



to creating a triple A MMO.

# The Solution



## THE SOLUTION

Finally

So, what was our solution to fix these problems?



We joined the dark side! Two technical artists were chosen to move their desks into the room where the programmers worked and to work together with them in adding new features and tools, and in optimizing the game. In this talk, I'm going to share my experiences as an artist living and working among the programmers on our team - and show how this simple act of moving into the programmer space was a major part of the solution to our social issues.





As a technical artist, you might think that this type of change - sitting with the programmers instead of with the artists - would be easy. After all, we're supposed to be able to speak the same language as the programmers and it is often said that our role is to act as a bridge between the two groups. However, after mostly sitting with artists for the past ten years, it did take a little getting used to. I often felt like "One of these things is not like . . ." especially after listening to long conversations about quantum physics or arguments about the relative merits of Common LISP over Scheme. It's pretty amazing how distinctly different it is to listen to

the casual conversation among coders vs. artists.  
The cultural differences are very real - and I had a  
bit of culture shock.



# Helping Artists Trust Programmers



Let's get back to the problems that we had. The main problem that the artists had was with tools. Frequently, the artists would run into a problem that they needed to solve. They would envision a tool to solve the problem, write up a document describing the tool and pass it along to the programming team. After a long wait the tool would be completed. While the artists had originally envisioned something like this



what they would get back ended up being more like this. This was a pretty major source of frustration for the art team. This is a pretty common problem. Designing and building complex systems requires constant communication and collaboration between art and programming.

## Tool Development Process

- Collaborate with the artists to create a prototype
- Document the tool requirements and demo to programmers
- Guide the tool implementation process and give feedback
- Write artist documentation and teach the artists how to use the tool

With tech artists embedded with the programmers, we changed this process around. First, I would collaborate with the artists to create a prototype tool that met all of the artists' requirements. Through this collaboration with the artists, I would become familiar with what they really wanted and how they intended to use the tool. I was also able to control (to a certain extent) the size and scope of the system to make sure that it wouldn't hurt the performance of the game and that the artists' expectations for the tool didn't get so high that the concept would be too complex or require too much programmer time.

Once the artists were happy with the prototype, I would document the requirements for the tool and show it to the programmers.

Once development started, since I was sitting with the programmers, it was a very natural for me to watch and guide the process, ensuring that the original vision was maintained. Often, the programmer would come up with his own ideas about what the tool should be. Sometimes these ideas were improvements, but sometimes they changed the nature of the tool. We would discuss changes to the original prototype and I would make sure that the original intent was maintained and that the artists would get the tool that they asked for.

Once the tool was nearing completion, I wrote documentation for the tool and helped the artists learn how to use it. Since I had been involved with the original prototype of the tool and involved during the tool's development, I was the best qualified to write the documentation and it was easy for me to teach.



I'd like to stress that this step - educating the artists - is important. It needs to be more than just an email that says "Hey, we have a new tool. Go read this document that I wrote about it." You really need to sit with the artists and show them the tool, and then watch them use it. By doing this, you can make sure that the tool is doing what the artists need AND that the artists are using the tool correctly.

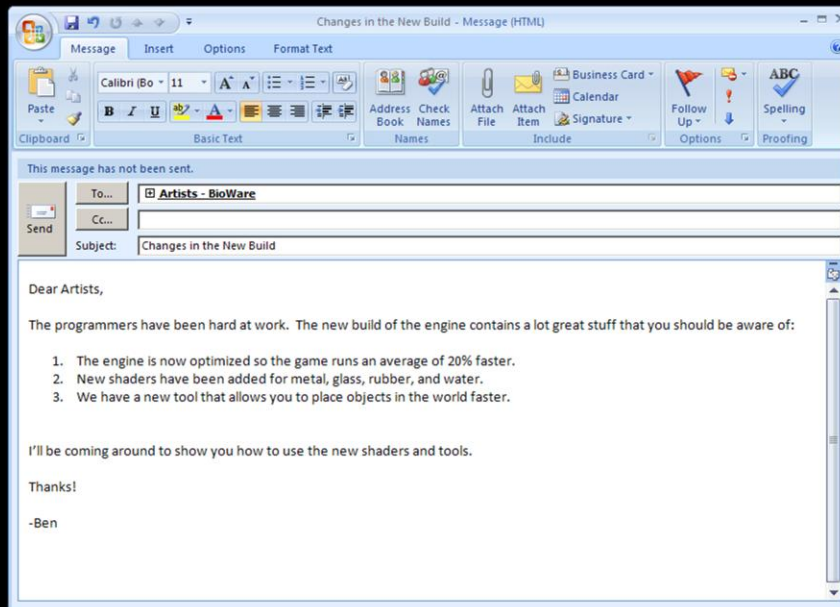


No matter how well a tool is designed, if you don't teach the artists how to use it properly, they'll find a way to make a mess with it - and it's always easier to teach first than to clean up the mess later.





Having an art representative in the programmer area helped the artists in a couple of other ways too. When the programmers wanted to make an optimization to the game that might make an impact on the art, I was there to stand up for the artists and help the programmers know when an optimization was going too far or when the performance benefit out-weighed the small loss in quality.



Also, since I worked with the programmers every day, I was aware of all of the projects they were working on to improve performance and quality. When new builds of the game engine went out, I sent out an email to the art team to help them understand the new improvements in the build that would impact them. This increased visibility helped the artists to see that the programmers were working hard to improve the game.



As a result of these changes, the artists got tools that matched and sometimes exceeded their expectations and their ability to create game art was improved. They also had more information about what the programmers were doing to improve the game. Their trust in the programmers increased.

# Helping Programmers Trust Artists



Now I'd like to switch gears and talk about how the programmers benefited from having embedded tech artists join them. As I mentioned earlier, the major concern that the programmers had is that the artists were creating art that was wasteful and inefficient. They believed that the main cause for the low frame-rate was that the art was too heavy. One thing that I had noticed before moving into the programmer area was that the programmers often found specific examples of bad art and said things like, "Man, this texture is huge. They need to fix this." or "This model is super dense and you only ever see it from a distance.

There are a lot of wasted triangles here.”



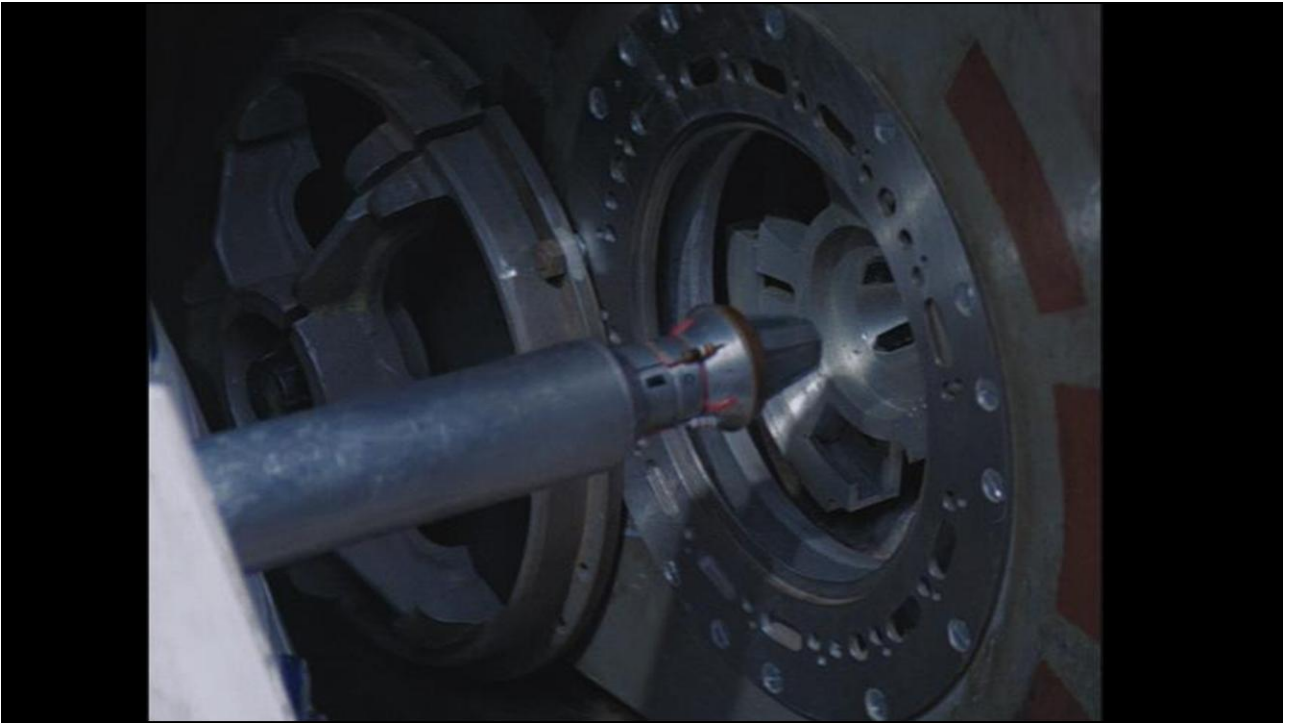


Since no one from the art department was present, these off-the-cuff complaints were basically just getting thrown out there with no one to respond to them. The programmers had a lot of built-up frustration at seeing in-efficient assets and no one seemed to be fixing them. After moving in with the programmers, I made it a point to jump up and respond when ever I heard a programmer complain about the art. I'd make a note about the asset in question and either fix it myself or pass it along to the right artist to optimize.

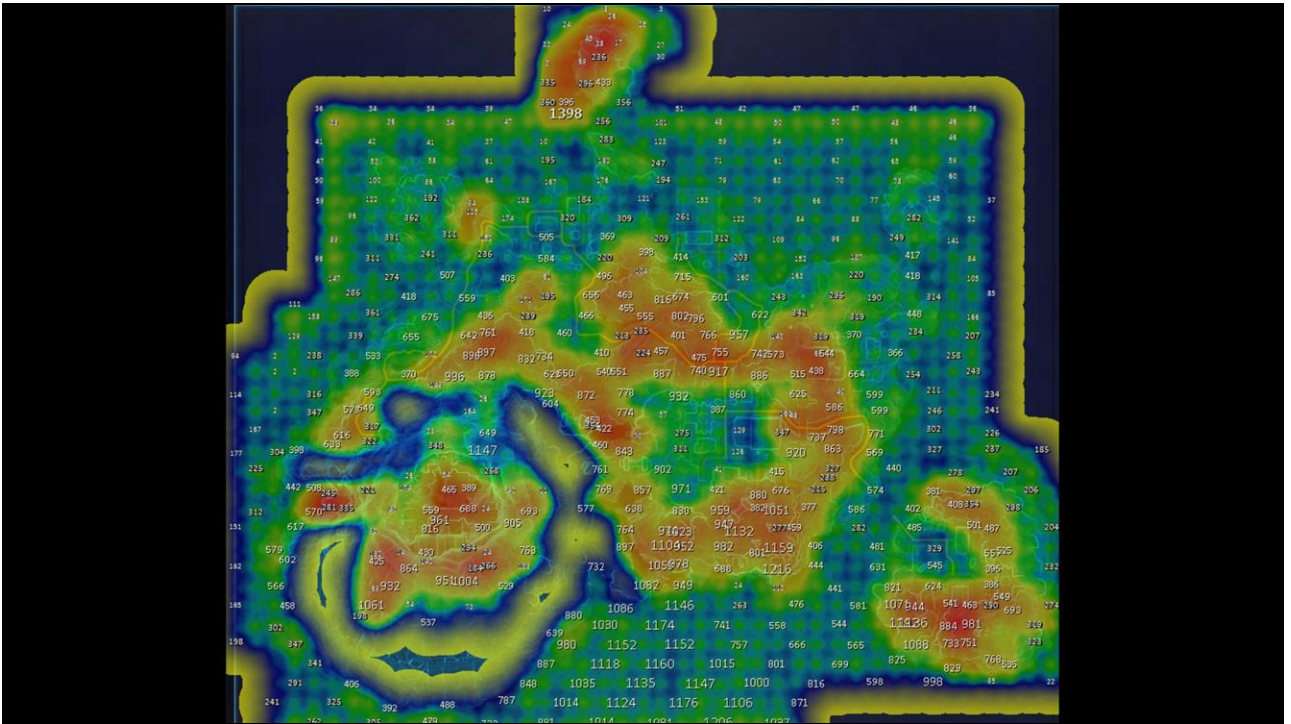




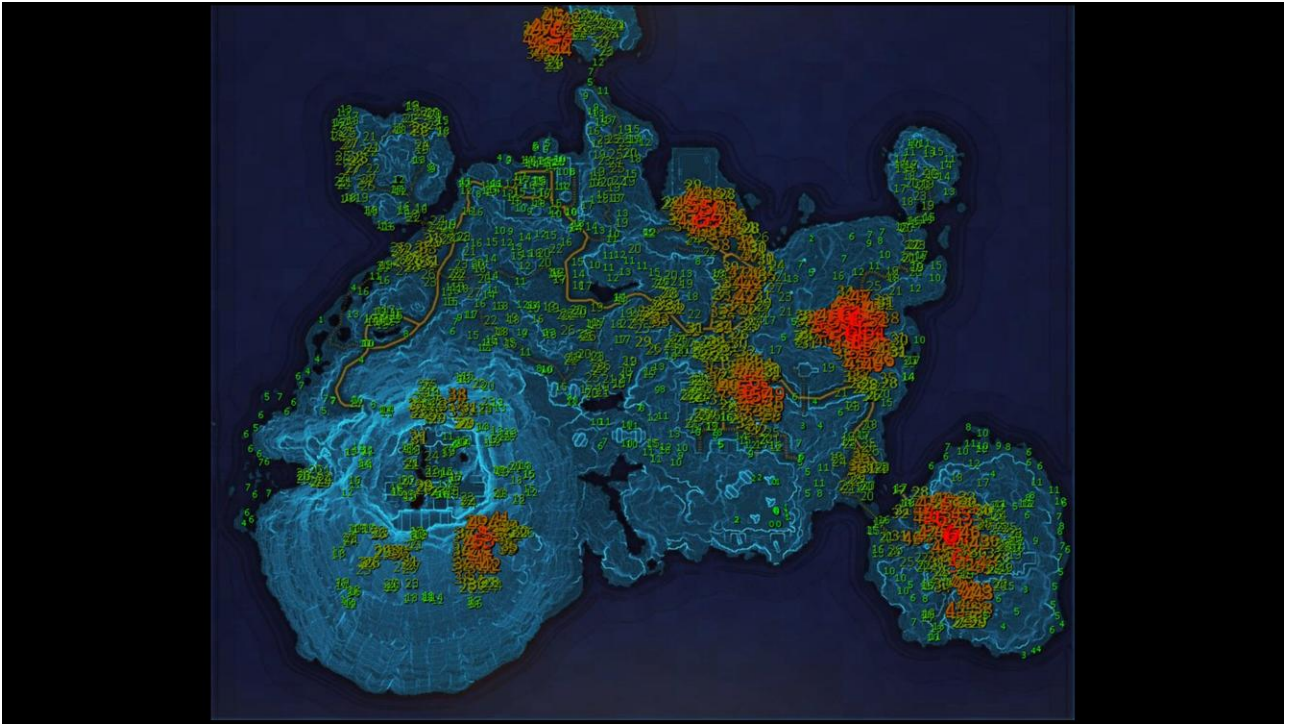
I basically become the programmer's complaint department. Even if things didn't get fixed right away, the programmers at least felt like someone was listening and responding to their complaints. This served to ease a lot of the tension that the programmers felt toward the artists.



We went several steps beyond just responding to off-the-cuff complaints. One of our major initiatives was a full audit of all the assets in the game.

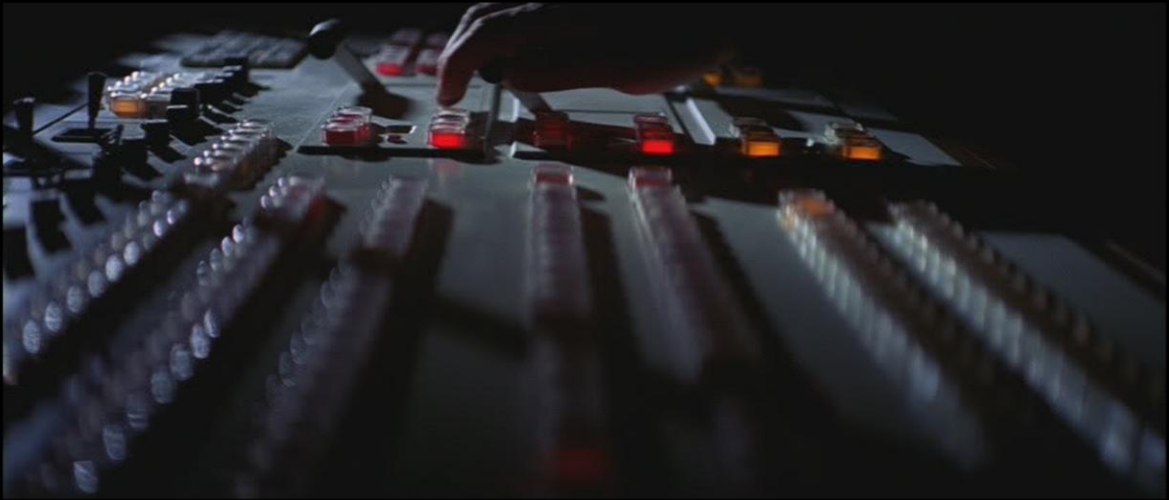


We created frame-rate and memory usage heat maps of all of the planets, created lists of textures that were too large, and models that used too many triangles. We put a lot of effort into gathering all of the information that the art team needed to make the game run faster using less memory. Some of the items we were able to go in and fix ourselves but most of the time we would create a report of actionable items and pass it along to a lead artist so that the work could be divided up among his team.





With the right information in hand, the art team was able to reduce texture memory usage significantly and increase frame rate.



Beyond texture size and triangle counts, we also investigated other resource drains such as the density and clip distance of terrain details, the complexity of our cloth simulations, and LOD settings on our character skeletons. In one case, I found that reducing the draw distance on the grass by about half raised our frame rate by 10 frames per second with no noticeable visual difference. It's pretty exciting when you can find that kind of improvement.

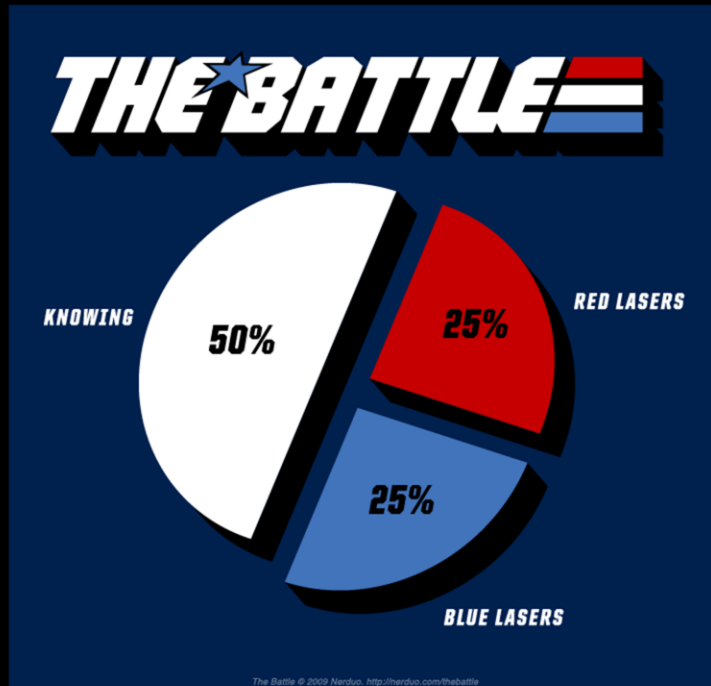




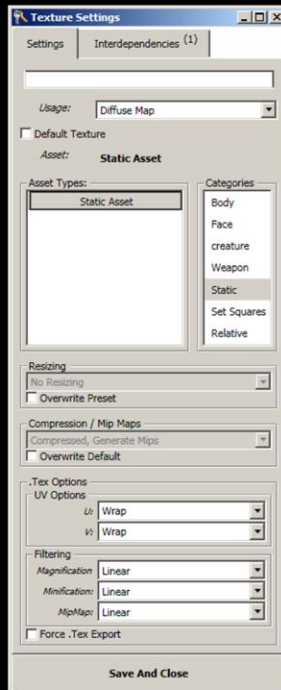
I once saw that the tech artists at Crytek created t-shirts for themselves that said “Digital Janitor” on them. I think this describes this project pretty well.

After all of our work optimizing the game’s art, we wanted to make sure these optimizations would remain that way and that future assets would be created in an optimal way. Basically we wanted to avoid having to do this type of clean-up project again.





The first thing that we did was to educate the artists. This mostly took place during the optimization process. As we passed list of assets along to artists to optimize, we would also take some time to explain the metrics that we were hoping to improve. We would show them the heat maps and help them understand what it was that caused the problem. Teaching the artists how to make efficient art, and how to check the on-screen metrics to make sure that their work was within budget was half of the solution.



The other half of the solution was to build smarter export tools. Our tech art team is responsible for the tools that export all assets into the game including models and textures. This means that we have a point in the pipeline where we can add checks to see if assets are optimal. We took advantage of this opportunity mostly with texture maps. Our exporter already had some context for how each map would be used, so we taught it the dimensions that a texture should be for each usage case. When a texture is exported, the size defaults to our optimal size or the original size, whichever is smaller. We also have a maximum size built

into the exporter. This prevents textures from getting exported at insane resolutions.



In addition to optimizing the art assets, I was also given the task of writing the low-end shaders for the game. In the options, the players can select to use low or high quality shaders. The low-end shaders are mostly for people that have weaker hardware. This was an ideal task for me, for several reasons. First of all, as an artist, I had a strong understand of what features of the shaders were core to the look and style of the game, so I could remove the right set of things without the art team going up in arms against me. Second, I was able to off-load this large task from the programming team and free them up to do other

optimizations.



Finally, on a personal note, I had a laptop at home that barely met our minimum hardware requirement and I wanted to be able to run the game on it. I set a personal goal to improve performance enough so that the game could run on my laptop, but without losing the distinct artistic style that we had defined.



**BEFORE**

250 Pixel Shader  
Instructions



**AFTER**

53 Pixel Shader  
Instructions

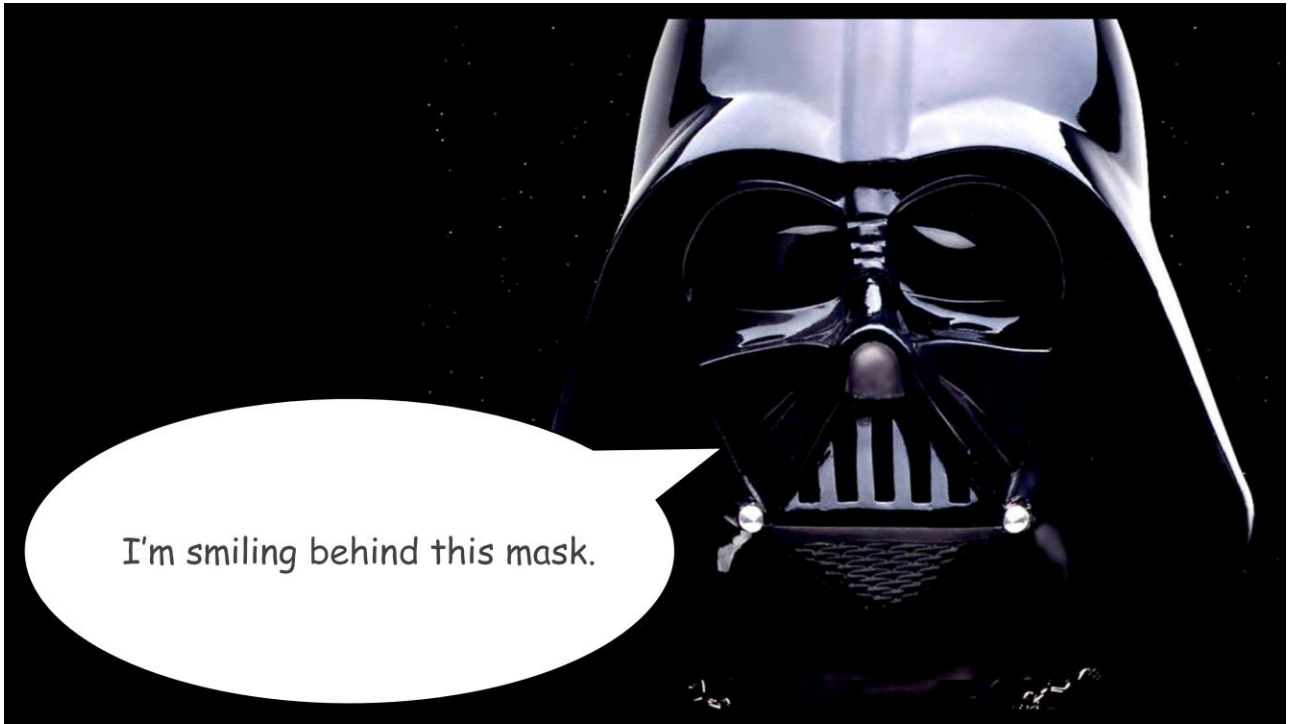
After working on the project for awhile, I was able to reduce many of the shaders to around a quarter of their original instruction count by removing optional features, moving some math into the vertex shaders, and simplifying core functions. On our low-end target hardware, the game ran an average of 20 frames per second faster when using my optimized shaders. And most importantly, it ran well on my laptop.





Here, I want to pause for a minute and talk about the importance of measuring things. Your worth as a tech artist is determined by your ability to solve problems. Making the game run faster and making the artists' work more efficient are two examples of the types of problems we solve. If you want to show your worth - and thus build the value of tech art as a discipline within your studio, you need to be able to put a number on it and use cold, hard facts. Before you start any project to solve a problem, try to measure the results as they currently stand. Figure out how long it takes the artist to accomplish his task with his current

tool. Make a list of how much texture memory each level is using. Write down what the instruction counts are on all the shaders. Once you have this initial data, go to work to make things more efficient. When you're done, take your measurements again. With this data in hand, you can show the team and the company - hey, I saved a week of artist time. Or - I increased the frame rate by 50%. Or - All of the levels are now within the memory budget - and the visuals still look as good as they did before. Having this type of fact-based information to share goes a long way toward increasing the respect that your company will have for the tech artists.



I believe that the programming team learned to respect the art team more when they saw all of the effort that we were putting in to optimize the art and the shaders in the game. The artists earned the respect of the programmers by being an active part of the solution.

## Conclusion

- Artists trust programmers because their tools work and they see the performance gains.
- Programmers trust artists because art assets are efficient and because they see the team's dedication to performance.
- The game is better because both teams are working together.

In summary, moving technical artists in to work with the programmers had several key benefits. First of all, the artists gained direct representation on the programming team. This was beneficial because we were able to participate in and guide the development of new tools, help the artists see what the programmers were doing to improve the game, and protect the interests of the art team from over-optimization. As a result of these changes, the artists now have a greater respect for the programming team and are more willing to work together and collaborate with them.

Second, the programmers gained direct access to the art team. Since I was sitting right there with them, whenever they had a question about the art, they could just ask. As we worked to improve the efficiency of the art assets and optimize our memory usage, the programmers saw the work that we were doing and experienced the performance gains that we achieved. I was also able to complete a couple of shader projects that the programmers would have had to do and free them up for other tasks. These changes helped increase the level of respect and trust that the programmers had for the artists.



This increased level of confidence and trust that the teams had for each other is important - because it means that they are more willing to work together. I'm not going to say that the relationships between the teams are perfect. We still have room for more improvement in some areas. But we have made significant progress - and that progress shows in the product that we shipped.

I also don't want to take the credit for these improvements. I was in the middle of the process, but it was really the willingness of the programmers to allow an artist to be a part of

their team, and the willingness of the artists to work hard to improve performance that made this happen.

So unlike in the movies where joining the dark side leads to hate, misery, and suffering, in our case joining the dark side led to greater trust, confidence, collaboration, and in the end - a better game.



## Take-Away Ideas

- Guide
- Teach
- Measure
- Build Trust



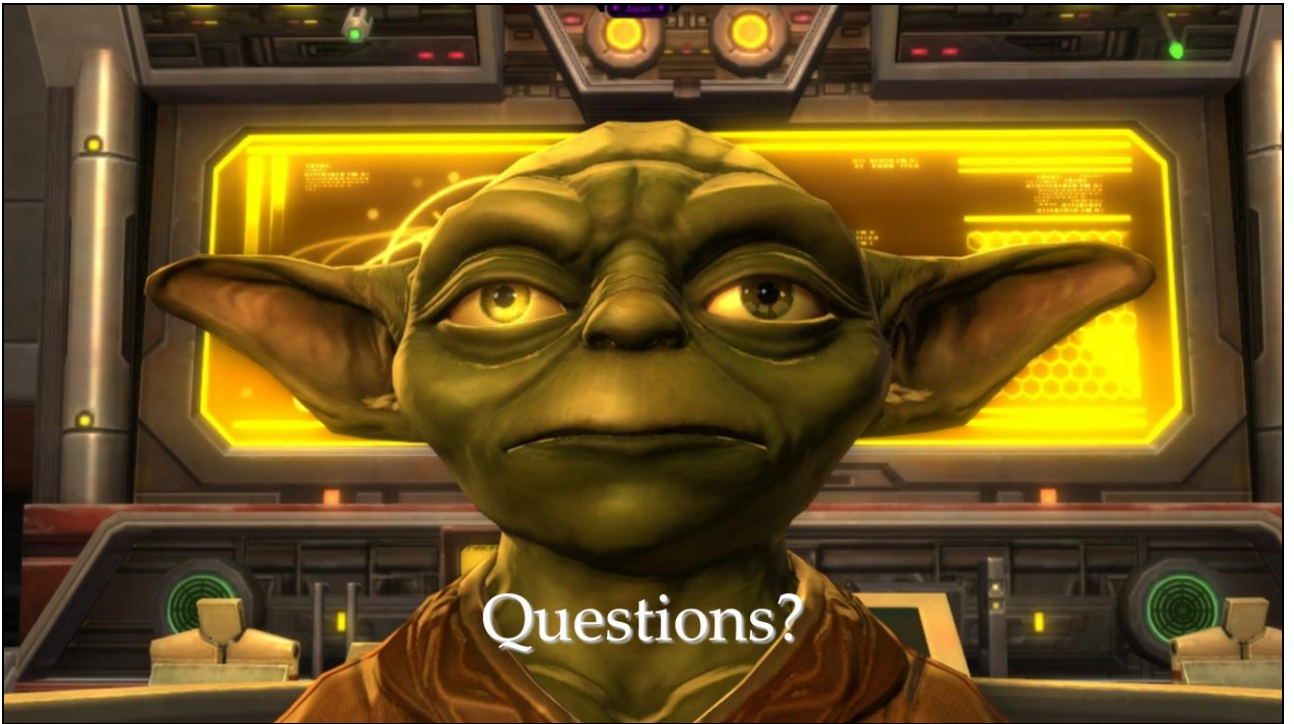
So, after sharing some of my experiences working together with the programmers for the past two years, I'd like to leave you with four key ideas. The first is guide. When working to design a new tool or solution to a problem, you need to guide the artists so that their expectations for what the tool will be able to do don't get out of control. Then you need to guide the process of creating the tool so that the end results matches the original vision and so that the tool accomplishes the intended task.

The second is teach. As a tech artist, it's your job to make sure that the artists on your team

know the pipeline, know the tools, and know what's acceptable and what isn't - in terms of creating art that will perform well. If you end up with a mess to clean up at the end, you can only blame yourself for not being a better teacher.

Next - measure. Capture data both before you begin a project and once you have completed it so that you can show the company - in hard facts - what benefit your efforts have achieved.

And finally, build trust. This one is the most important. Artists and programmers that trust each other, will work together and collaborate to build incredible games. Artists and programmers that don't trust each other, won't work together - and your project will suffer. Do all you can to help each team see what the other is doing. Bring the teams together and help them collaborate.



# Lessons in Tool Development

Jason Hayes

Technical Art Director – THQ / Volition

Hi everyone, my name is Jason Hayes. I am a Technical Art Director at Volition working on Guillermo Del Toro's Insane video game. Just to give a little bit of history on myself- I've been in the video game industry for over 14 years, and have worked on a bunch of titles: Madden NFL franchises, Lord of the Rings Online, Saints Row series and the Core Technology Group at Volition.

# Overview

## What to expect

- Concepts that **enable** you to develop better tools.
- **Not** about how to write a good **for** loop.

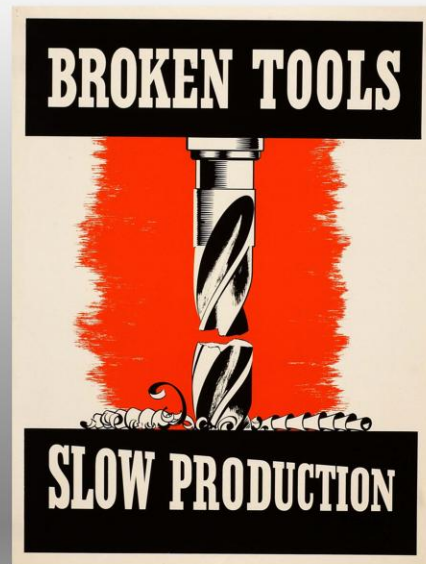
## Intended Audience

- **New** Technical Artists

Today, I'm going to give a talk about lessons in tool development. These aren't all of the lessons I'd like to talk about today, I could go on for hours and this talk is probably the least sexy out of all of the talks you will and have seen presented today. I also think your mileage will vary since the role of a Technical Artist varies from company to company. At Volition, we don't license a game engine or editor, we primarily build everything ourselves, which keeps our Tech Artists pretty busy.

A lot of what I will present today is probably common sense to many of you, however in a lot of cases, common sense doesn't always prevail during the development process. My talk primarily stays at a high level in terms of software and pipeline development for Technical Artists. I'm not going to get into the level of detail of how to write a for loop, or best practices on code optimization-there is a ton of literature out there already on the subject. Instead, what I'm going to talk to you about today are some of the principles I've experienced and learned over the years that have helped guide me and the team at Volition when building tools and content pipelines. My target audience for this talk is primarily geared towards Technical Artists who are new to the industry and companies who are trying to grow their Tech Art departments.

# Overview



To help frame the discussion, I thought it would be useful to start off by talking about something we've all experienced during game development: bad tools and pipelines. Why and how in the world does this happen? In my experience, a lot of the time we end up doing it to ourselves because we are under pressure to meet a deadline and just need to crank a tool out. Sometimes we can't always avoid these situations. But, most of the time it happens because we enjoy building tools and typically skip a lot of planning and just jump right in. So we don't stop and take the time to really think things through, because that's boring! Or if we did stop and look, we didn't have the right information because we asked the wrong questions.

Today, I'm going to present some basic and fundamental lessons in tool development. When approaching tool development, I tend to think about it in two different ways: Strategic and Tactical. Let me explain.

# Strategic



Strategic is the high-level vision of a tool. At this level, you should be looking at the big picture of how the tool fits into your overall pipeline, and how it affects the user workflow. This is also the most effective point at which you can save your company money and increase the productivity of your team. As our industry moves into another cycle of next generation hardware, I believe pipelines will become larger and more complex, so we need to build things that are scalable and most importantly, save your company money. I don't know how common it is for Technical Artists to think about the tools they write in this way: your company is paying you to write tools to make the content pipeline an efficient machine, and so you must look at the best way to spend that money.



# Tactical



Tactical is the low-level view of a tool. This is where the architectural design, implementation and code reviews of tools happen.

I'd like to start the presentation by talking about the Strategic level of tool development, and the first part of that process for us is what I call Tool Briefs.

## Strategic - Tool Briefs

- Describe the need and scope.
- Strategic document.
- Build consensus.
- Made up of 3 simple questions.



Tool briefs are short documents, typically one page or less that describe the need, criteria and scope of the proposed tool. They are a strategic document, and don't delve into the details and logistics of how the tool will be implemented. At Volition, tool briefs are written and approved prior to any technical spec being drafted.

The primary purpose of the tool brief is to make sure that everyone is on the same page about what will be delivered. They give your Manager the opportunity to assess the cost of implementing the tool, and they also provide an easy point to make course corrections early in the design process. Moreover, it gives everyone involved an opportunity to ask questions, and provides your Director a window into how you are thinking.

At Volition, our tool briefs are made up of three simple questions. These questions are intentionally designed to be kept short and focused to make the person writing the tool brief really think and question what they are about to build and write it in such a way that it communicates the tool to a wide audience.

# Strategic - Tool Briefs

- The Description
  - What is it?
- The Function
  - How might the end-user use the tool?
- The Justification
  - Why does it need to exist?

## **The Description (What is it?)**

This one is pretty straightforward. Here, you describe the needs, criteria and scope of the tool.

## **The Function (How might the end-user use the tool?)**

Sometimes it's easy to overlook how the tool might affect the end-users productivity. It's important to keep this in mind and how the tool fits into the big picture of the overall process. It's very easy to add new tools to a pipeline, but it's very difficult to keep it running smoothly for your team.

## **The Justification (Why does it need to exist?)**

This is the part of the brief that provides the rationale for why we should be creating the tool. The following is a quote that I feel fits the bill of a Tool Brief perfectly:

## Strategic - Tool Briefs

"The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double-barreled penalty because you also don't solve the right problem."

Steve McConnell, *Code Complete*

*"The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double-barreled penalty because you also don't solve the right problem." - Steve McConnell, Code Complete*

To illustrate the importance of a tool brief, I'm going to talk about a scenario at work that happened several months ago. I was talking to one of our Technical Artists for a little while about a system where we would allow artists to define what I was calling a "lighting diorama". The basic premise was that we would allow artists to setup different lighting situations in which to test their content against on the console. I'm not going to go into all of the details of the system, but suffice to say, leaving the conversation, I thought we were both on the same page, and asked him to write up a tool brief. What I got back was a tool brief that talked about how the material pipeline would be leveraged to verify content on the consoles. So there was a complete disconnect there. How did this happen? Well, in this particular instance, it was because we had a similar system on Red Faction Armageddon (albeit a different engine and codebase), but that system primarily focused on material tweaking. It illustrated how easily things get missed or misunderstood in a conversation where you say one thing, but the other person hears something else and vice versa. Having the tool brief written probably saved the company a lot of money by not allowing the TA to start heading down the wrong path. At the end of the day, that's what we are really talking about-saving your company money.

# Strategic - Tool Briefs

## Cost of Solving the Wrong Problem



$$\$10\text{k per month} * \frac{5 \text{ incidents}}{\text{development}} = \$50\text{k loss}$$

Take the story I just gave you as an example. In that story, the TA and myself were on two different pages, and if I just let him start going to work, he very well could have spent a month writing the wrong tool. If the employee overhead of your company has a burn rate of \$10k per month, and miscommunication like this happened 5 times over the course of development for that one person, that's a potential loss of \$50,000.

# Strategic - Tool Briefs

## Cost of Solving the Wrong Problem



$$\begin{array}{l} \$10\text{k per month} * 5 \text{ Tech Artists} * \\ \frac{5 \text{ incidents}}{\text{development}} = \$250\text{k loss} \end{array}$$

On Insane, I have a team of 5 Technical Artists, and if each of them had the same issue 5 times over the course of development, that's a loss of \$250,000.

The tool briefs are the one tool in our arsenal as Technical Artists that can turn a \$250,000 loss into a \$250,000 savings.

For the next strategic topic, I'm going to talk about understanding the user and their workflow.

## Strategic – Understand the User



Seems obvious, right? This is one of the most important parts of the job of being a Technical Artist, and one that is the easiest to overlook. A lot of the time, we are building tools that support or expand existing pipelines. But what happens over time is the pipeline gradually becomes more complex and ultimately slows down the artists.

There are several methods to understanding the end user and how they work.



# Strategic – Understand the User

## Map out the Workflow



Mapping out the workflow as a flow chart is a good way to get a high level perspective on the end users process. If your pipeline is creating bottlenecks for your users, then mapping it out should reveal where the problem areas are, then you can create a plan of action to address the issue. A good approach to how to map out the pipeline is to associate how long they spend on each part of the process.

# Strategic – Understand the User

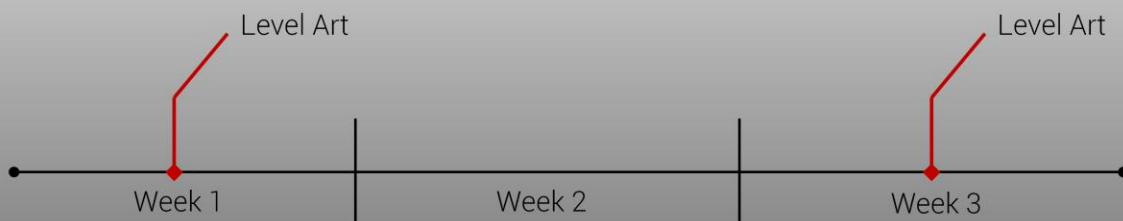
## Face Time



Another way to familiarize yourself with the end-user is to sit down with them and watch how they are working. It's very important to build those working relationships with the people we are supporting. They need to feel confident that the tools we are developing are there to make their lives better. Besides, if the person you are watching doesn't get creeped out over this, it can be an eye-opening experience for a lot of Tech Artists. Sometimes, you'll see artists using your tools in ways you didn't expect and would have probably only surfaced by watching them work.

# Strategic – Understand the User

## Bi-Weekly Discipline Dependency Meetings



Something I like to do is hold bi-weekly dependency meetings with each art discipline on the team. The meetings are fact-finding missions to discover what the artists are working on and what they have coming up. At Volition, we use Hansoft to manage the project's tasks and backlog, but trying to determine dependencies in that software is nearly impossible, so I avoid it altogether and meet for 30 minutes to talk.

The meetings are also great opportunity for the artists to bring up any other issues they are having and have become my most valuable meetings. I would encourage you guys to implement these into your process if you don't already.

For the next strategic lesson in tool development, I'd like to talk about overengineering.

# Strategic – Overengineering

## Definition

Making things more complex rather than simplifying them.

Just like art, overengineering is really subjective. It's a term that's widely used, but very difficult to define. My definition of overengineering is when you are making things more complex rather than simplifying them. In a lot of cases, overengineering a tool or pipeline happens because we want to make the code we are writing able to work with any other piece of code, and that we want to make it the greatest thing since sliced bread. Or, while we are in some other code, we see something we don't like and want to refactor it to make it "better". As long as the set of requirements are clearly defined, this is okay and there is nothing inherently wrong with trying to make things better and shareable, but if not careful, it can easily lead to situations where the code ends up getting too complex and some other person trying to use your code doesn't really understand what's going on. One reason why this can be bad is it typically leads to dependency issues for other disciplines who are depending on your tool or system. This usually results in delays and can have a cascading effect on the schedule and ultimately cost your project and company money.

So how do we know when a tool is being overengineered? As a Technical Art Director, these are the warning signs I look for of a system or tool that's being potentially overengineered:

# Strategic – Overengineering

## Warning Signs

- Running behind schedule



Running behind schedule doesn't always mean that something is getting overengineered. In most cases, we just underestimate how long something will take, usually because we discover things over the course of development. But it is a warning sign and if the person is running behind schedule, it's probably a good time to step in and take a look at what's going on.

# Strategic – Overengineering

## Warning Signs

- Making the code too generic



Making the code too generic is another one of those tricky and subjective aspects of software development. One way to tell if something is being too generic is to look at the set of requirements. If you are expanding a focused set of requirements into something that can be a "jack of all trades", you are probably overengineering something. For example, say you are tasked to build a simple tool that is supposed to track how shaders are used on content. Well, you decide that you want to turn it into a generic system that can track how *any* piece of data is being used on content. It would become incredibly complex and stop being good at showing how shaders are used on your content.

# Strategic – Overengineering

## Warning Signs

- Code is complicated to follow
- Difficult to maintain



Another indicator of when something is being overengineered is when the code has become difficult to follow, which usually means it has also become difficult to maintain. A warning sign for this is when someone who is using your code is frequently asking questions about how it works. When this happens, the best method I've found to surface overengineering is to go to a white board and have the TA(s) write out what they are trying to do. Sometimes this takes the form of a flowchart, sometimes it's pseudo-code. I've had countless situations like this with TA's over the years where they started to get lost in their code and only realized it when I put them in front of a white board.



# Strategic – Overengineering

## Warning Signs

- Designing too far into the future



Probably the best way of detecting overengineering is when someone is designing too far in to the future. The problem with this is that it can lock you into a particular design without knowing if it's really the right solution. I'm not saying you shouldn't design for the future, but keep the design at a high-level. This will give you the flexibility to course correct over the course of the tool's development.

# Strategic – Overengineering

## Warning Signs

- Little-by-little



Overengineering doesn't always have to happen in big stages either. More often than not, overengineering occurs in tiny little stages, function by function. Before you know it, you've got a mess on your hands. This usually happens when someone is working on a tool and sees a snippet of code that looks like it could be abstracted out. This is a fine practice, but it's just something to be conscious of.

The reverse of what I'm talking about is also true, you can underengineer a tool or pipeline by not architecting it in such a way that it allows for later expansion.

## Strategic – Overengineering

Good design practice = code that is easy to follow and maintain.

Bad design practice = code that is too difficult to understand and maintain.

But at the end of the day, good design practice equals code that is easy to follow and maintain. Bad design practice ends in code that is too difficult to understand and maintain, and ends up costing you time and money.

# Strategic – Phased Development

Developing in phases is just common sense, but you'd be surprised at how often it gets overlooked—at least in my experience. The reason why working in phases is important is it allows you the ability, at a high level, to make course corrections in the design process and set of requirements. While developing anything, you always discover new things that impact what you are going to do in the future. In a sense, you answer what you didn't know. Obviously, it's easier to make these course corrections in the earlier set of phases. If you have to make a major course correction in later phases, then something earlier in the process broke down.

One distinction we make at Volition about phases is that they aren't tied to any particular milestone. A phase can expand multiple milestones if necessary.

How many phases are appropriate? Well, it really depends on how you operate, but we've found that breaking things down into four phases is a good place to start. The general framework looks like this:

# Strategic – Phased Development

## General Framework

- Phase 1
  - Basic design and implementation.
- Phase 2
  - First pass of iteration.
- Phase 3
  - Second pass of iteration.
- Phase 4
  - Final polish pass.

### **Phase 1 - Basic design and implementation.**

At this phase, the basic system is functional. Verification takes place based on the goal/task lists and associated expectations. The system is not polished, but sufficiently fleshed out to allow iteration work to start.

### **Phase 2 - First pass of iteration.**

In this phase, the system or tool is sufficiently fleshed out that a decision can be made as to whether it fulfills its promise or exhibits fundamental issues. Validation is focusing on whether the feature is "on track". It's also the point where the determination is made as to whether this is a tool or system that could potentially be cut.

### **Phase 3 - Second pass of iteration.**

This is the first polish phase and the tool or system should be fully implemented. Features at this point are locked down and only minor additions are acceptable (subject to approval). An evaluation is made to assess whether or not the tool is successful and whether it should be kept or cut. You can only enter the final phase if the tool or system is being kept.

### **Phase 4 - Final polish pass.**

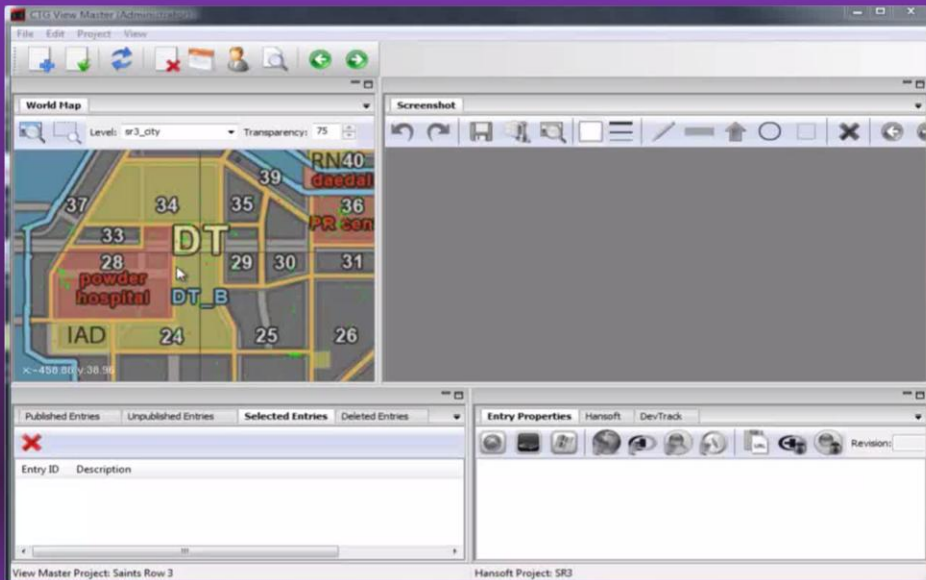
Here, the features of the system or tool are locked down and it's essentially production ready.

# Tactical

In this next section of the presentation, I'm going to talk about what I consider to be the tactical approaches to tool development. Primarily, I'm going to talk about approaches for Interface and Architectural Design.

To illustrate the lessons in this part of the presentation, I'm going to use two tools that I designed and help write at Volition.

# Tactical – View Master



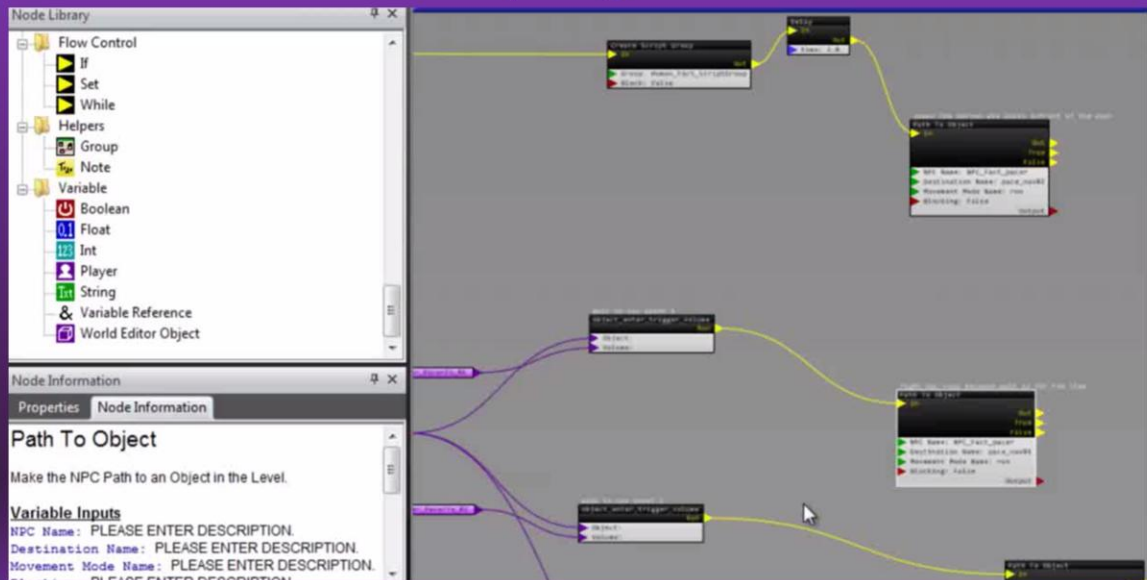
This is a tool that myself and Adam Pletcher wrote with Python over the course of several months. View Master is based off of a tool that Will Smith wrote on the Red Faction: Armageddon title. The tool has three primary goals in life:

- Allow the team and QA to easily record bugs directly from the game.
- Provide art an easy way to provide direction.
- Allow the team and QA to easily navigate to bugs in the game.

Adam developed all of the Hansoft integration and database backend, and I created the rest of the framework including the GUI and OpenGL renderer. In fact, this tool has become so valuable to our process that other THQ internal studios have begun integrating it into their pipelines.



# Tactical – Visual Scripting Editor



The other tool I'm going to use to illustrate some of these principles is our Visual Scripting Editor. This is a node-graph based editor, similar to Unreal Kismet, that we use to do the scripting in our games and is a fairly new tool. I developed the framework and initial implementation, as well as the OpenGL renderer for the tool, using only Python.

# Tactical – Interface Design

## Separate Core Functionality

```
def on_mouse_left_down( self, event ):  
    screen_pos = event.GetPosition( )  
    self.selected_node = None  
  
    nodes = self.graph_manager.get_nodes( )  
  
    for node in nodes:  
        if node.hittest( screen_pos ):  
            self.selected_node = node  
            node.select( )  
            break
```

```
def on_mouse_left_down( self, event ):  
    screen_pos = event.GetPosition( )  
    self.selected_node = None  
  
    self.graph_manager.handle_left_mouse_down( screen_pos )
```

```
print  
node.select( )  
self.selected_node = node
```

From an architectural point of view, separating core functionality from the interface allows you to work in a non-destructive manner to the rest of your program. When done well, you should be able to completely change around the design of your interface and the program still work. Architecting your code in this way also opens the door for your tool to be run in headless mode, i.e. as a command line tool.

For example, in the Visual Scripting Editor, I have a bunch of mouse event handlers in the OpenGL viewport. I could have written something like this in the interface's mouse event handler to select a node in the graph:

```
def on_mouse_left_down( self, event ):  
    screen_pos = event.GetPosition( )  
    self.selected_node = None  
  
    nodes = self.graph_manager.get_nodes( )  
  
    for node in nodes:  
        if node.hittest( screen_pos ):  
            self.selected_node = node  
            node.select( )  
            break
```

Instead, I have a graph manager class that handles node selection. So all I do is call that handler function from the interface while passing in the current screen position of where the user clicked.

```
def on_mouse_left_down( self, event ):  
    screen_pos = event.GetPosition( )  
    self.graph_manager.handle_left_mouse_down( screen_pos )
```

Not only does it reduce the code complexity in the interface, but this allows me to completely change around how the interface looks, what functions are bound to event handlers, etc. and the program will still function-because the core functionality hasn't changed.

# Tactical – Interface Design

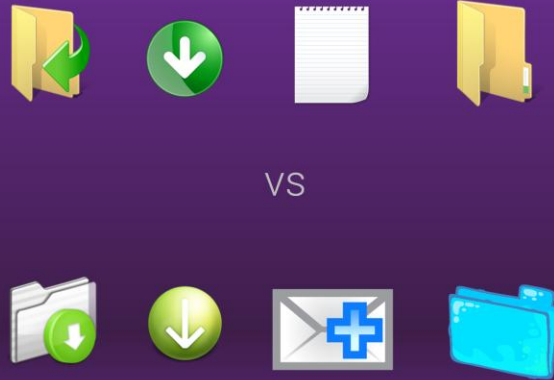
Keep clicking to a minimum

In order to make the end-user more productive, keep the number of clicks in your interface to a minimum. Using keyboard shortcuts are also good, but just be careful that they aren't the only way to do something. I've actually used editors and tools in the past where this was the case and certain functionality of how to do something became lore of the tool, absent in any documentation.

# Tactical – Interface Design

## Keep Interface Clean and Simple

- Consistent icons

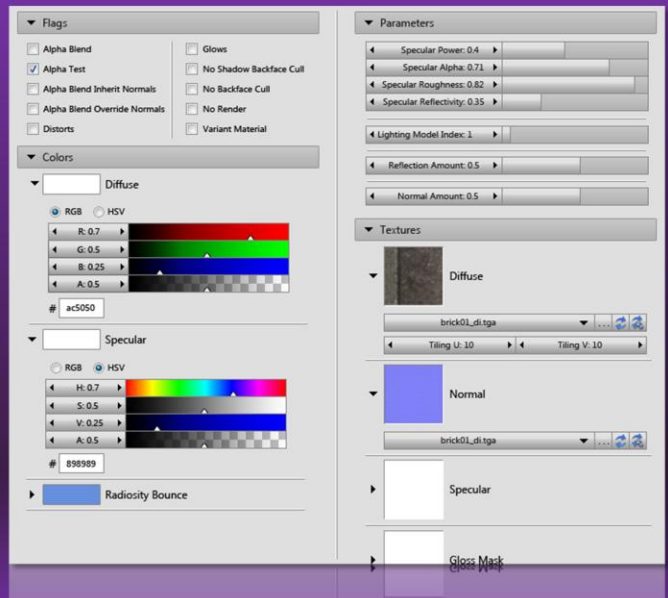


Having consistent icons is a good way to keep things clean in your interface. What I mean by consistent is that the icons should look like they exist in the same world. Inconsistent icons can make the user interface feel unprofessional and cluttered.

# Tactical – Interface Design

## Keep Interface Clean and Simple

- Intelligent grouping
- Control alignment



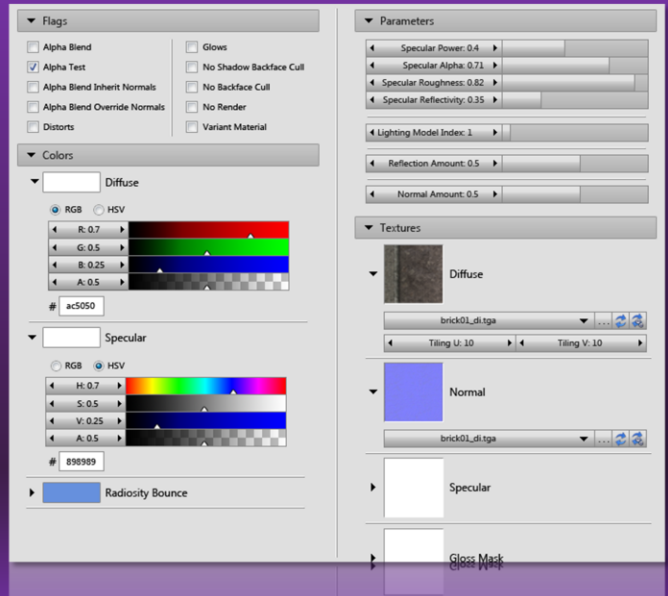
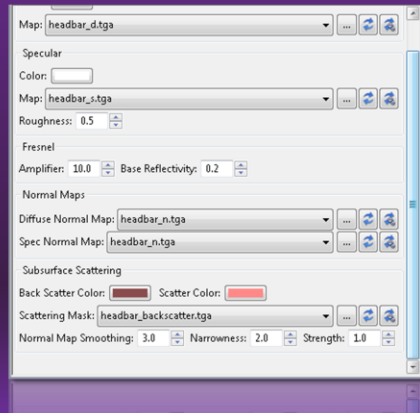
Intelligent grouping and keeping alignment of controls in mind and consistent will also make your interface appear clean and simple.

The image here showcases part of the design of Volition's Material Editor.

# Tactical – Interface Design

## Polished Interface

- Should make the user feel creative



## Provide a polished interface that makes the user feel creative

Out of all my years of designing user interfaces, this is in my opinion, the most important thing to keep in mind in terms of visual representation. Regardless of what the tool does, it should make the user feel creative when they are using it. Take for instance the following tool:

Image 1

With this tool, the user can do their job just fine. But replace it with this:

Image 2

Even though the functionality of the tool is the exact same, the user might feel more creative while using this version of the tool because of its visual design and more intuitive controls. This simple approach can lead to more productive end users and higher quality results.



# Tactical – Architecture Design

## Managing the Complexity

- Software is inherently complex.



Next, I'm going to talk about one principle I've learned over the years when it comes to designing the architecture of a tool or pipeline. This principle lesson I'm going to discuss is about managing the complexity.

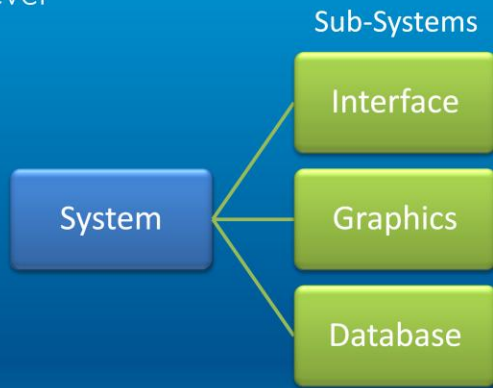
Software design is inherently complex, and in order to develop successful software we must build a solid foundation. If you were constructing a house, it would be a very bad idea to jump right in and start building the walls without first building a solid foundation, because the consequences would be dire. This same fundamental principle applies to software construction.

Just as a house is complicated to build, so is software. Like a construction manager who needs to manage the complexities of a construction site, we have to do the same in the tools we develop. When designing software, it's important to start by keeping the design at a high level, minimal in complexity, and easy to maintain.

# Tactical – Architecture Design

## Managing the Complexity

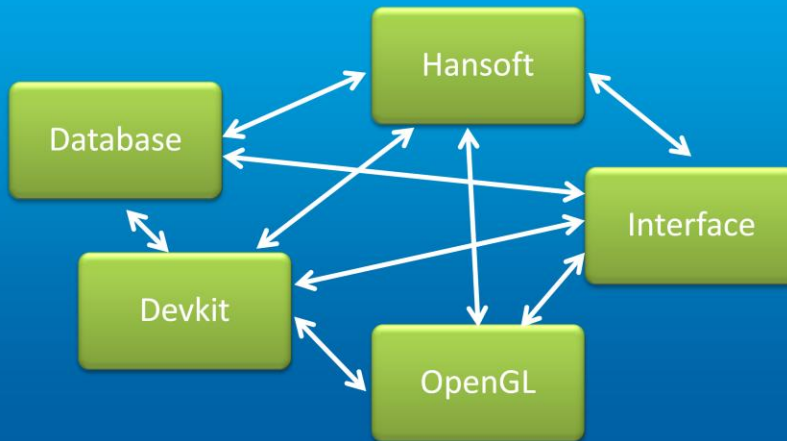
- Design at sub-system level



I've found that designing at the subsystem level is a good first step to managing the complexity. If it isn't clear, subsystems are the components that make up a system.

# Tactical – Architecture Design

## View Master Sub-System Design



For Viewmaster, the tool's subsystems are broken up as follows:

**Database:** Read and write bugs to a database

**Hansoft:** Communicate with Hansoft, create new bugs and retrieve bug information

**Devkit:** Create bugs from the game and navigate to those bugs in the game

**OpenGL:** Viewports to view bugs on a game map and display screenshots with the ability to add annotation.

**Interface:** The gui needed to be flexible enough for each user to customize their layout as well as the projects adding in their own views.

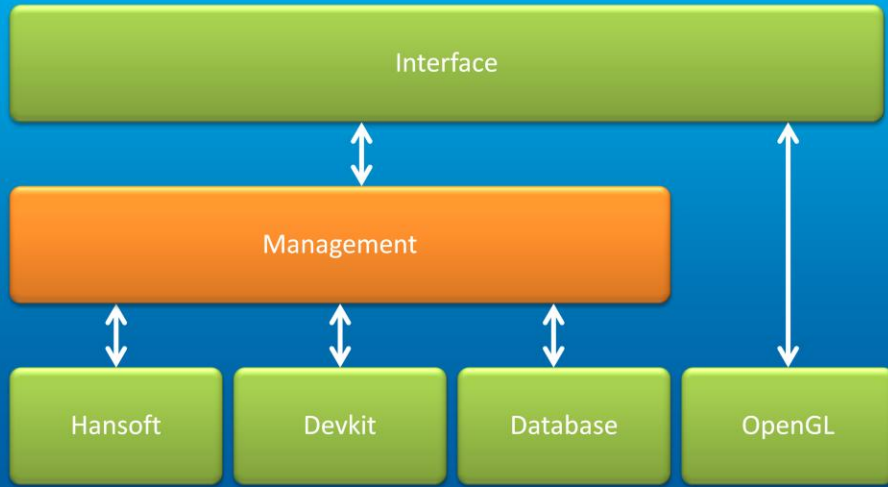
Those are the major subsystems of the tool. As you can see, these are some pretty major and broad systems for a tool. But where to go next? Communication between the subsystems is a critical step in the design process, and one that should never be skipped. If I designed the inter-communication of the subsystems while coding them, I could easily get into a situation where each subsystem was trying to communicate to each other because I'm not sure how it all fits together. I could end up in a mess and get lost.

It's important to keep the communication between subsystems simple, clear and straightforward. If it isn't, you open yourself up to code paths

that become confusing to follow, you make it more difficult for extensibility and maintenance becomes more of a headache.

# Tactical – Architecture Design

## View Master Sub-System Design



In Viewmaster, to make sure communication was kept simple and in clear paths, I created a Management layer.


The Management layer is how subsystems interact with each other. It becomes important at the Class or Object level of the code. Instead of having to pass every subsystem to each other, I only need to pass the primary manager class.

# Tactical – Architecture Design

## View Master Sub-System Design

```
class Devkit( object ):  
    def __init__( self, Hansoft, Database, OpenGL ):  
        pass
```

```
class Devkit( object ):  
    def __init__( self, Manager ):  
        pass
```



The management layer becomes important at the Class or Object level of the code. Instead of having to pass every subsystem to each other, I only need to pass the primary manager class.

So instead of this:

```
class Devkit( object ):  
    def __init__( self, Hansoft, Database, OpenGL ):  
        pass
```

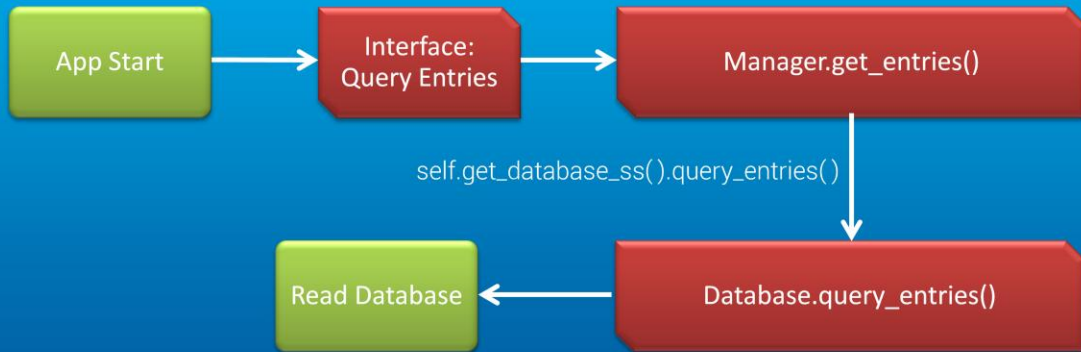
I can simply do this:

```
class Devkit( object ):  
    def __init__( self, Manager ):  
        pass
```

This keeps your Class design simple and straightforward. It also allows maximum flexibility and extendibility. The only issue you have to be aware of in a design framework such as this are circular dependencies.

# Tactical – Architecture Design

## View Master Program Flow Example

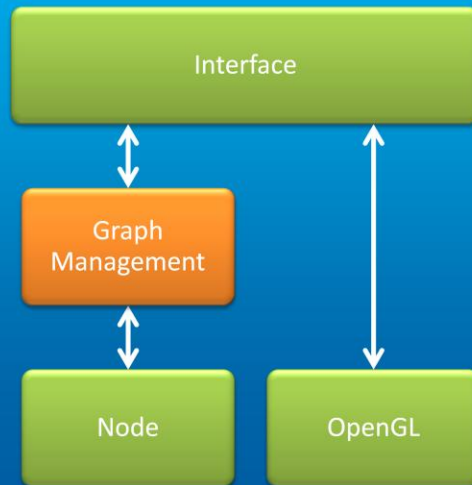


With View Master, whenever the interface subsystem needs information, it has to go through the manager to get the information it wants. For example, when the tool first loads up, it asks the database manager which project the user has set, and then populates and caches the entries manager with all of the entries for that project. It was designed this way for performance reasons of hitting the database. The Viewmaster database stores screenshots and annotation data which can cause performance problems for our remote users. It also gives a nice interface for each view of Viewmaster to simply ask the entries manager for the entries to display, based on the users current set of criteria.



# Tactical – Architecture Design

## Visual Scripting Editor Sub-System Design



In the Visual Scripting Editor, the subsystems were set up as follows:

- Interface
- Node
- OpenGL renderer
- Graph Management

Once you've determined how the general framework of your tool works at a high level, then you can begin to design your classes and so on.

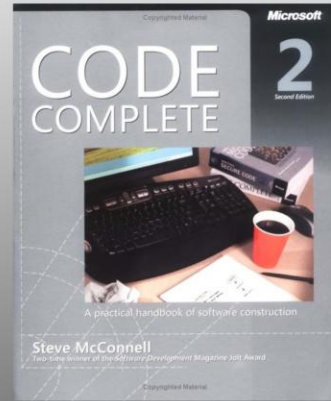
The take away here is that as long as you can manage the complexity of your design by keeping things simple and straightforward, put restrictions on how subsystems communicate with one another, your tools and pipelines will improve.

## Key Takeaways

- Utilize **Tool Briefs** to clarify and build consensus on the work you are doing.
- Determine the **right** problem to solve, because solving the **wrong** problem will cost you engineering time and money.
- Watch out for **overengineering**.
- Develop in **phases**.
- Manage the **complexity** of software design by **simplifying** subsystem communication.

# Recommended Literature

- Code Complete (2<sup>nd</sup> Edition)



Questions?

<http://www.jason-hayes.com/>

# Shady Situations: Real-time Rendering Tips & Techniques

**Wes Grandmont III**  
Senior Technical Art Director  
Microsoft Studios | 343 Industries



GAME DEVELOPERS CONFERENCE  
SAN FRANCISCO, CA  
MARCH 5-8, 2012  
EXPO DATES: MARCH 7-9

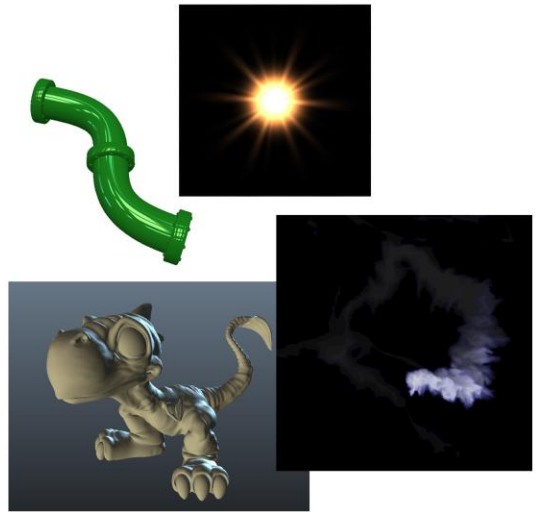
**2012**

Hello my name is Wes Grandmont III. I'm Senior Technical Art Director at Microsoft Studio's 343 Industries. The talk I'm giving today, Shady Situations: Real-time Rendering Tips & Techniques is meant to be a spring board for those of you looking to add shader development to your skill set. It collects a bunch of really useful techniques into one place so that you'll have a set of building blocks that you can use in your own work.



# Session Summary

- GPU Overview
- HLSL Overview
- Applied Techniques\*



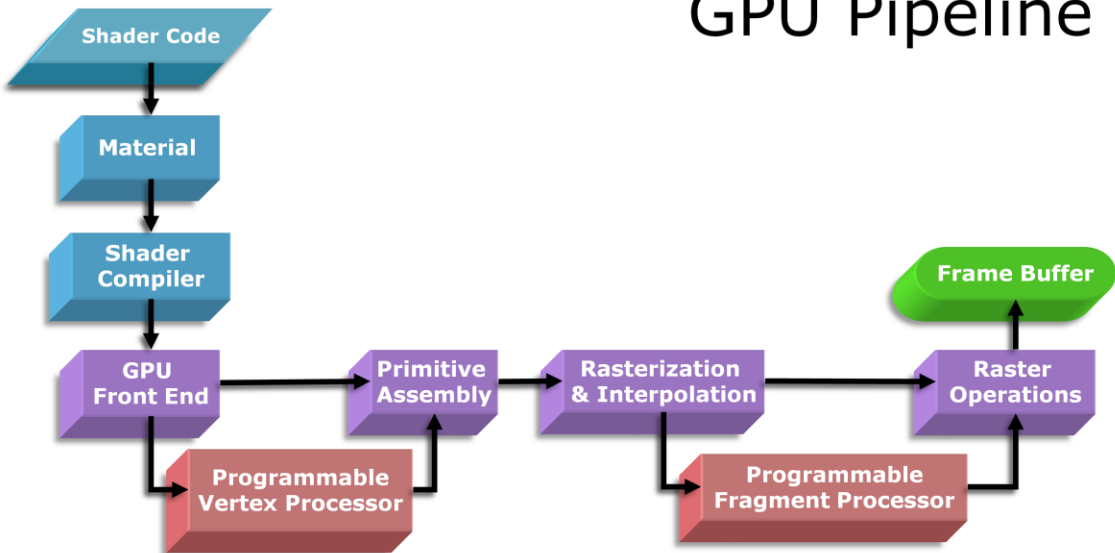
\*This is not a discussion of rendering techniques being used in Halo 4

The first part of this session will be focused on some big picture concepts. It's really important to understand what's going on inside the GPU when writing shader code so we'll do a very quick overview of how the GPU pipeline works. After that it's essential to understand the basic code structure of a high level shader, so we'll briefly discuss the different parts of an HLSL shader. Finally, I'll dive into a series of techniques and go over how they are implemented.

I'd like to note that this talk is not a discussion of techniques being used in Halo 4. Everything I'll be presenting today is based on projects I worked on prior to joining 343 Industries and my current role at 343 is not focused on maintaining the shader portion of our pipeline.



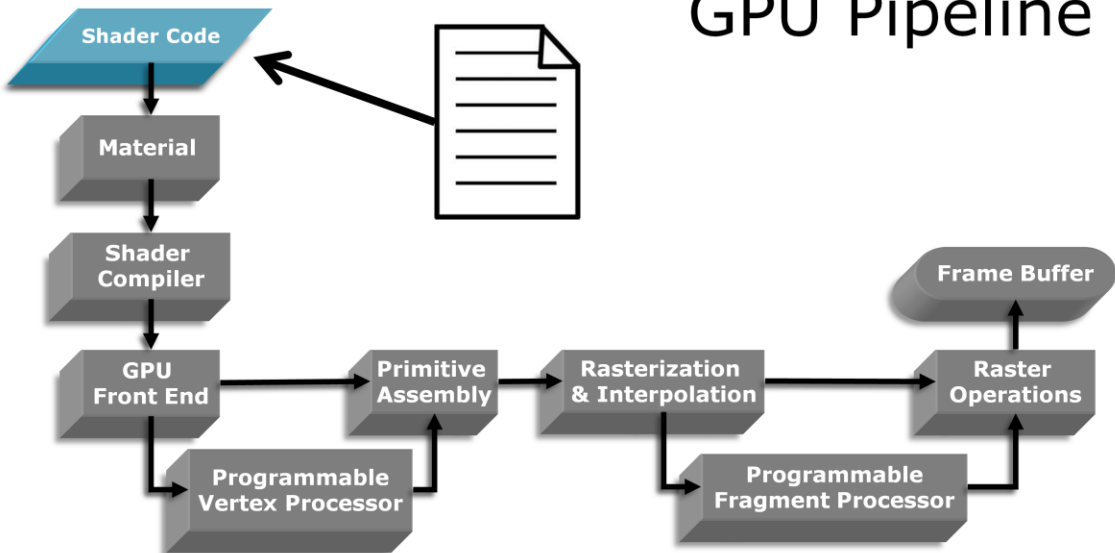
# GPU Pipeline



This is the GPU pipeline as it looks for GPUs that support Shader Model 3. The more recent Shader Model 5 has some more components that I'll discuss very briefly.

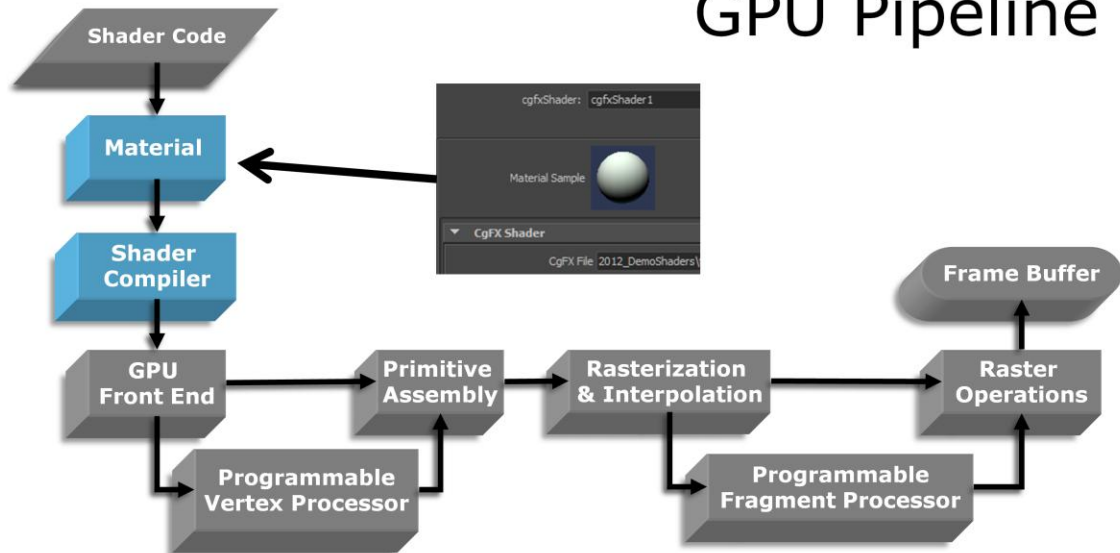
Shader code flows through this pipeline in order to become a pixel in a render target. The only parts we touch directly are the shader code and material, everything else is handled by our application and the graphics hardware.

# GPU Pipeline



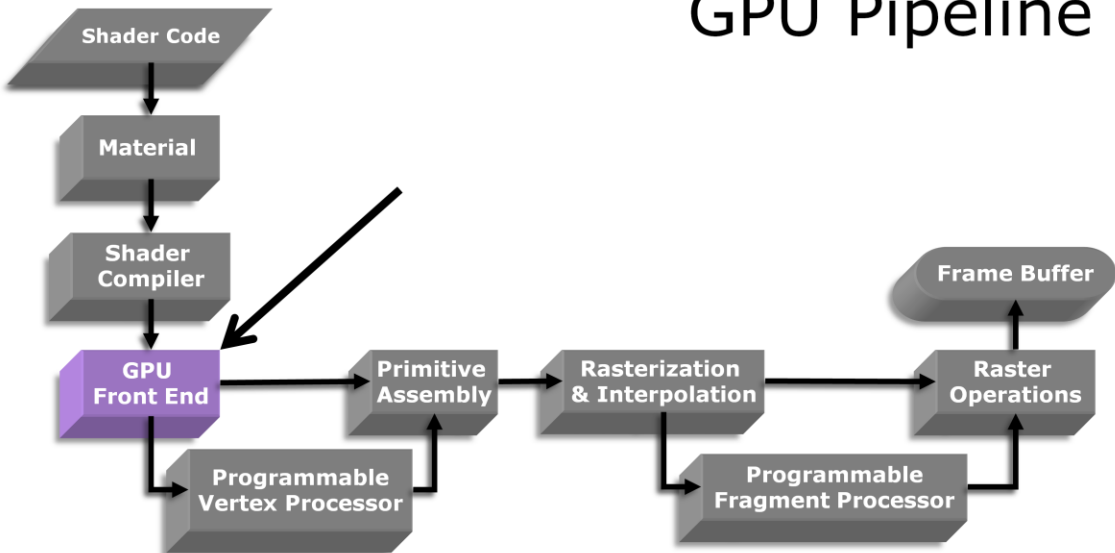
The first step is having some shading code written in a high level shading language (HLSL). For this talk, I'll be using Nvidia's Cgfx language. The Cgfx runtime supports OpenGL and DirectX and more importantly for this talk, it's supported in Maya via the cgfxShader plugin. Cgfx as well as the .fx file format allow us to encapsulate multiple rendering techniques, embed GUI parameters, create multipass shaders and specify render and texture states in one file.

# GPU Pipeline



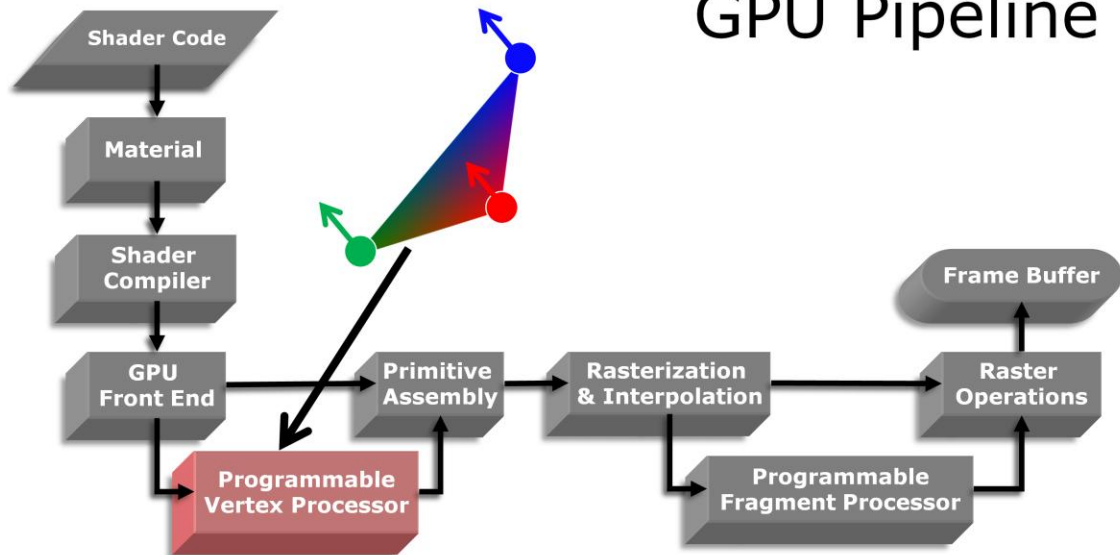
The application (3D program using the shader code) will have some sort of material implementation that will point to your shader code. In Maya this is an instance of the `cgfxShader` node. The shader code is referenced by the material which supplies instance specific values for various inputs in the shader (colors, texture paths, scalars). The shader code is compiled by the shader compiler and uses the material parameters as inputs that are passed to the GPU.

# GPU Pipeline



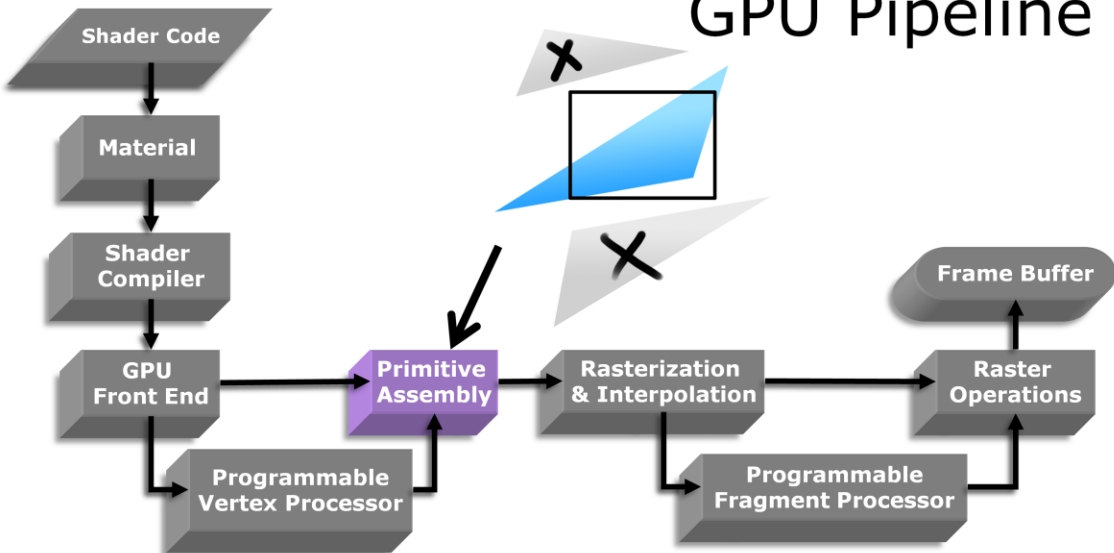
The GPU front-end receives data from the application which includes compiled shader code, vertex positions, normals, tangents, binormals, uvs, vertex colors, matrices and material parameters. These are all the ingredients the GPU needs to proceed with rendering.

# GPU Pipeline



The GPU processes the vertex data before passing it to the primitive assembly stage. At this stage, the shader code has the first opportunity to alter the data coming from the application at the vertex level. This could include moving vertices, changing vertex colors, or changing transform spaces. It's usually much cheaper to do things at this stage since there's usually far fewer vertices to process than fragments that need to be processed further down the pipe.

# GPU Pipeline



At the Primitive Assembly stage the vertices are assembled into triangles, points or lines and clipping and culling operations are performed to limit the primitives being processed to only those that are needed.

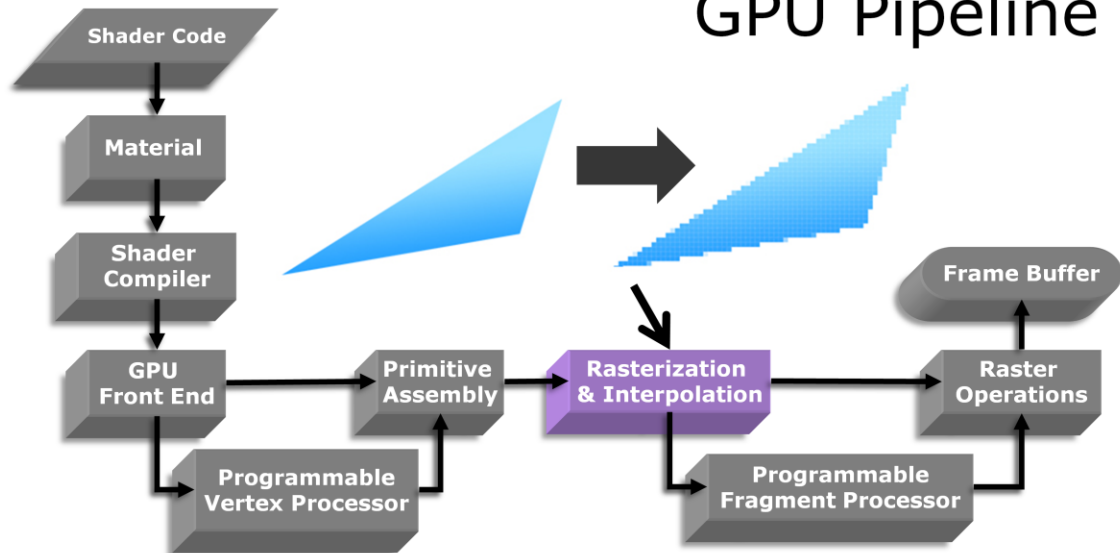
In modern GPU's that support DirectX 11 this stage is replaced by the following:

- **Hull Shader:** Performs operations on sets of patch control points, and generates additional data known as patch constants.
- **Tessellator:** Subdivides geometry to create higher-order representations of the hull.
- **Domain Shader:** Performs operations on vertices output by the tessellation stage, in much the same way as the vertex shader.
- **Geometry Shader:** Processes entire primitives and allow them to be discarded or to have new ones generated.
- **Stream Output:** We also have the opportunity to write out

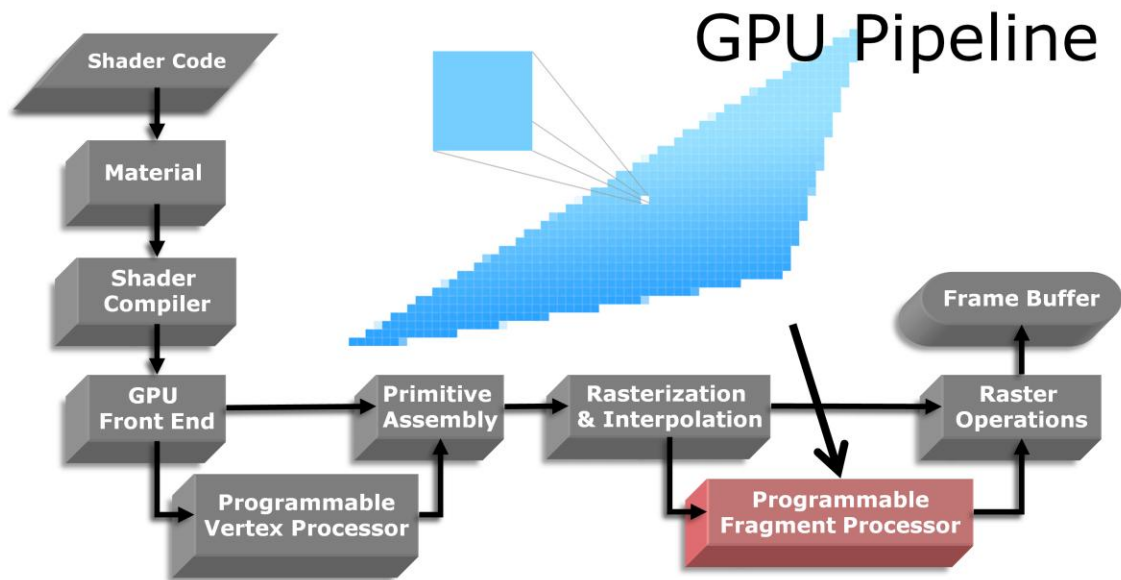
the previous stage's results to memory so we can use it again later



## GPU Pipeline



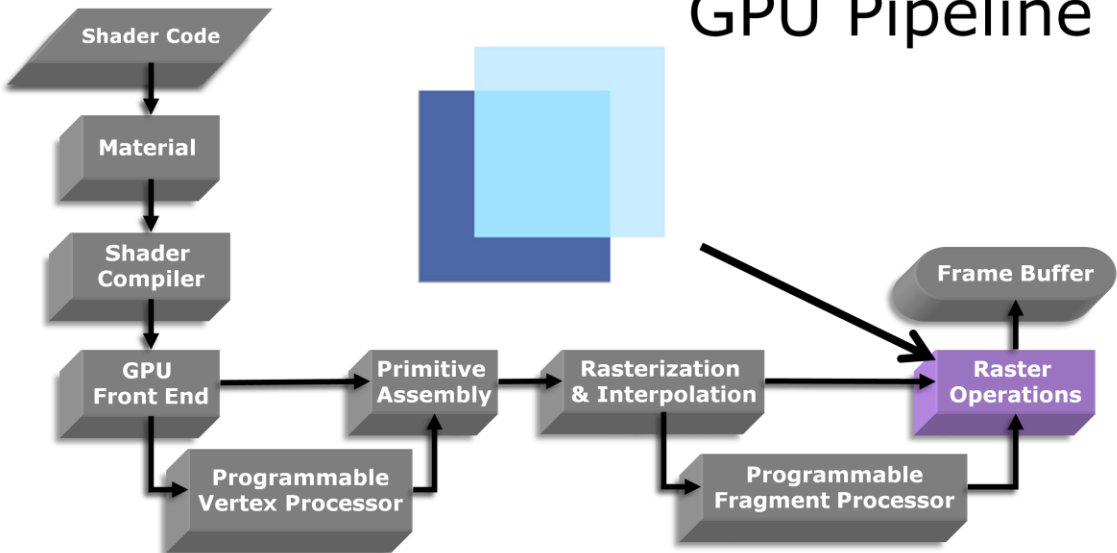
Next, the primitives are “rasterized” to produce a set of pixel locations and fragments (“potential pixels”) in the framebuffer. At this stage we call them fragments because the GPU hasn’t decided whether or not the fragment will actually get written to the framebuffer. Vertex colors, UV and depth information are interpolated at each fragment location.



The fragment processor (pixel shader) Determines the final fragment/pixel color to be written to the frame buffer or render target and can also calculate a depth value to be written to the depth buffer.

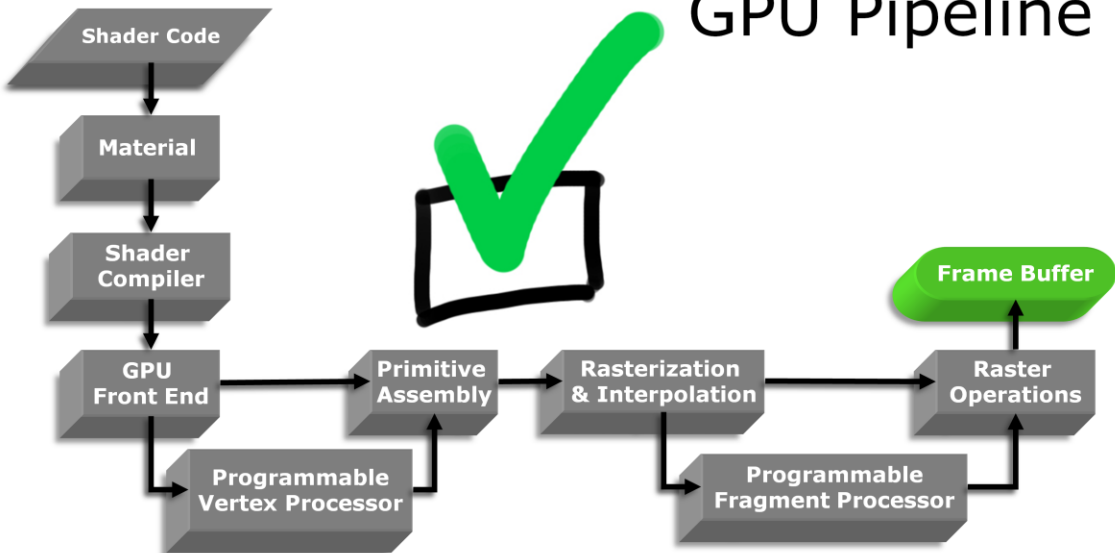
In modern GPU's that support DirectX 11 this also includes passing data from the pixel shader to a compute shader for additional processing.

# GPU Pipeline



Raster operations are performed to determine if and how the fragment should be added to the frame buffer/render target. This includes things like alpha test, depth test, and what blending operation to use.

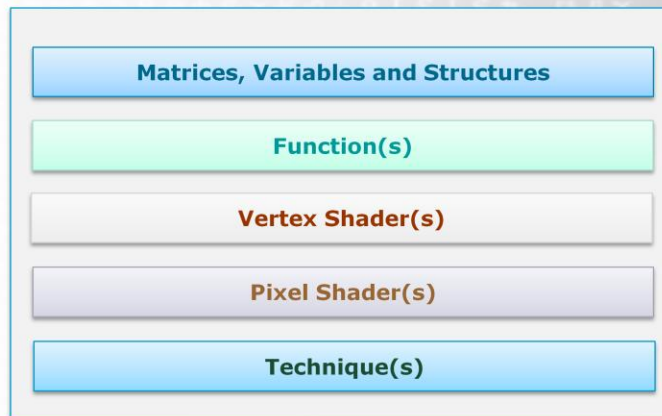
# GPU Pipeline



If the fragment passes all the raster operations then it is promoted to a final pixel and is written into the frame buffer / render target.

And that's the whirlwind tour of what's happening inside the GPU. Now let's take a look at the anatomy of a high level (Cgfx) shader.

# HLSL Code Structure



This diagram shows an abstract view of the order in which blocks of code are defined in the shader. Each block depends on the ones above it so the order is very important! Variables need to be defined first so that the functions and shaders can refer to them farther down in the code. The following examples will show what each of these blocks look like in a Cgfx shader text file for Maya.

# HLSL Code Structure

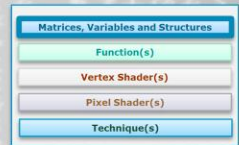
- Maya Cgfx Matrix Semantic Examples

```
//<variable name> : SEMANTIC FROM APP

//world space
float4x4 wMatrix : World;

//World View Projection
float4x4 wvpMatrix : WorldViewProjection;

//contains camera position
float4x4 vitMatrix : ViewInverseTranspose;
```

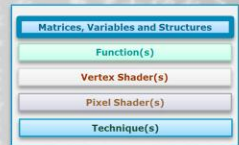


Our top block of code includes all the matrices, structures and variables we'll want to use in our functions, vertex and pixel shaders. The first part of this input data that we'll discuss are the matrices that are passed from the application to the GPU. We can access these through identifiers called SEMANTICS. Using the colon operator (:) we can map these keywords to float4x4 matrix variables that we can use later.

# HLSL Simple Example

- Matrix Transformation Spaces

- Frame of reference / point of view
- Passed from Application
- `mul()` to move vectors into different spaces
- Very important to understand what space things are in!
  - Vector calculations will be wrong if vectors aren't in same space



Matrices allow us to evaluate vectors from different points of reference (transform spaces). It's very important to perform shader calculations in the right transform space. Using the `mul()` operator, a vector can be multiplied by one or more of the transform matrices in order to move it from one frame of reference to another.

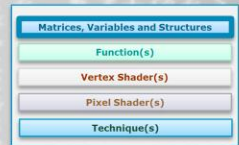
When performing math operations between vectors, always double check to see whether they are both in the same space and that that space is the most efficient place to be doing the calculation.



# HLSL Simple Example

- Data Structures

```
//Data passed from the application to the vertex program
struct a2v
{
    float4 Position    : POSITION;
    float4 Color       : COLOR;
    float2 TexCoordA   : TEXCOORD0;
    float2 TexCoordB   : TEXCOORD1;
    float3 Tangent      : TEXCOORD2;
    float3 Binormal     : TEXCOORD3;
    float3 Normal       : NORMAL;
};
```

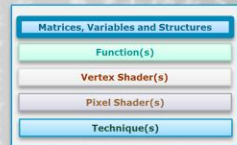


The next bit of code is the data structures that will be used to pass data from the application to the vertex shader. We use keyword semantics to help fill out our data structure. The struct has been given the name "a2v" which is shorthand for "application To vertex shader".

# HLSL Simple Example

- Data Structures

```
//Data passed from the vertex program to the fragment program
struct v2f
{
    float4 Position : POSITION;
    float4 color : COLOR;
    float4 TexCoord : TEXCOORD0;
    float3 Normal : TEXCOORD1;
    float3 Binormal : TEXCOORD2;
    float3 Tangent : TEXCOORD3;
    float4 worldPos : TEXCOORD4;
};
// TexCoord.xy = UV1, TexCoord.zw = UV2
//(we stuff 2 sets of UVs in a single float4 to save space)
```



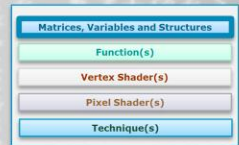
We also need a structure for passing data from the vertex shader to the pixel shader. Here we've called it "v2f", shorthand for vertex shader to fragment shader". We have up to 8 TEXCOORDs that we can pack values into (TEXCOORD0-TEXCOORD7). Since there's limited TEXCOORDs available for passing data to the pixel shader, it's common practice to pack smaller values together. In this example we'll be packing two sets of UV's into the float4 called "TexCoord". TexCoord.xy will be UV1 and TexCoord.zw will be UV2.

# HLSL Simple Example

- Variables

```
float3 Ambient : Ambient
< string UIName = "Ambient"; string UIWidget = "Color3";> = {0,0,0};

float3 Diffuse : Diffuse
< string UIName = "Diffuse"; string UIWidget = "Color3";> = {1,1,1};
```

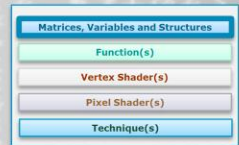


Next we define one or more variables that can be modified on each material instance. With Cgfx we can also specify how we want the UI widget to be created by supplying additional parameters within the <> braces. Finally we can specify a default starting value.

# HLSL Simple Example

- Texture Sampler

```
texture DiffNormalPackMap : Diffuse
<
    string UIName = "DiffNormalPackMap";
    string UIWidget = "texture";
    string ResourceName = "C:/white.tif";
    string ResourceType = "2D";
>;
sampler2D diffNormMapSampler = sampler_state
{
    Texture = <DiffNormalPackMap>;
    MinFilter = LinearMipMapLinear;
    MagFilter = Linear;
};
```

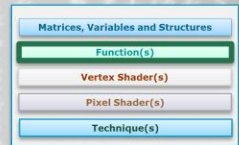


Texture samplers are the last piece of input data that we need to define in this top section of the shader code. A texture sampler consists of two parts. A texture resource that references a texture file and a sampler that we can use later to access values from the texture resource.

# HLSL Simple Example

- Functions

```
float getFogFactor(float4 worldPos)
{
    float fogDistance = length(mul( wvpMatrix, worldPos));
    float fogFactor = exp2(-abs(fogDistance * (fogDensity/10000.0f)));
    return fogFactor;
}
```



Once all the input data has been defined, the next big block of code are the function declarations that we want to use in our vertex and pixel shaders. Functions are small re-usable bits of code that make our shaders easier to maintain by allowing us to use the same code across multiple shaders. This is an example of a simple fog function that returns a fog value based on an input world position.

# HLSL Simple Example

- Vertex Shader

```
v2f mainVS(a2v IN)
{
    v2f OUT;

    OUT.Position      = mul(wvpMatrix, float4(IN.Position.xyz,1.0));
    OUT.worldPos      = mul(wMatrix, IN.Position);
    OUT.TexCoord.xy    = IN.TexCoordA;
    OUT.TexCoord.zw    = IN.TexCoordB;
    OUT.Normal         = IN.Normal;
    OUT.Binormal       = IN.Binormal;
    OUT.Tangent        = IN.Tangent;
    OUT.color          = IN.Color;

    return OUT;
}
```



Once all the functions are defined we can add our vertex shader code. The vertex shader allows us to alter the vertex data coming from the application before it gets sent to the pixel shader.

# HLSL Simple Example

- Pixel Shader

```
float4 mainPS(v2f IN) : COLOR
{
    //Super simple example...usually there's a lot more happening here
    float4 finalcolor    = float4(1,0,0,1);
    return finalcolor;
}
```



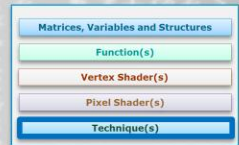
After defining our vertex shader we can now define the pixel shader. This is a very simple example that doesn't use any of the incoming data from the vertex shader. It simply returns the color red as the final color for the each fragment. Usually there's a lot more happening here.



# HLSL Simple Example

- Technique

```
technique Default
{
    pass p0
    {
        BlendEnable      = true;
        BlendFunc         = int2(SrcAlpha, OneMinusSrcAlpha);
        VertexProgram     = compile vp40  mainVS();
        FragmentProgram   = compile fp40  mainPS();
    }
}
```

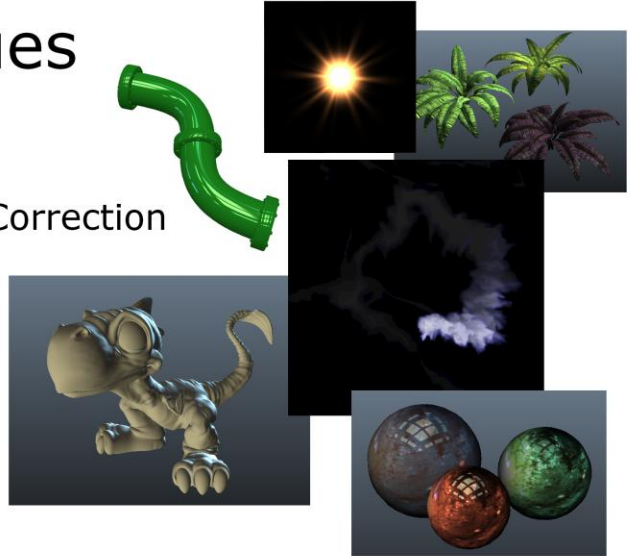


Finally we get to the very last section of code which is where we define one or more techniques. A technique is recipe of sorts where we can setup various render states and tell the GPU what vertex and pixel shader to use for each render pass, and what profile to use when compiling. With techniques we can do rendering in multiple passes as well as supply different fallback rendering methods.

And that concludes the super quick overview of the GPU and HLSL. If you have questions, check out the recommended reading slide at the end of this deck to get more information on these subjects. Now let's dive into the techniques!

# Applied Techniques

- Memory Efficient Plants
- Art-Director-Friendly Color Correction
- Auto Rim Light
- Procedural Gradients
- Bulging Pipe
- Pseudo-Fluids



For the remainder of this talk I'll be covering six applied techniques. Each one touches on a set of core concepts that can be used as building blocks for meeting a variety of rendering goals. I've sequenced the order to ramp from the simpler to the more complex.

# Memory Efficient Materials

- Texture Packing
- Swizzling
- lerp()



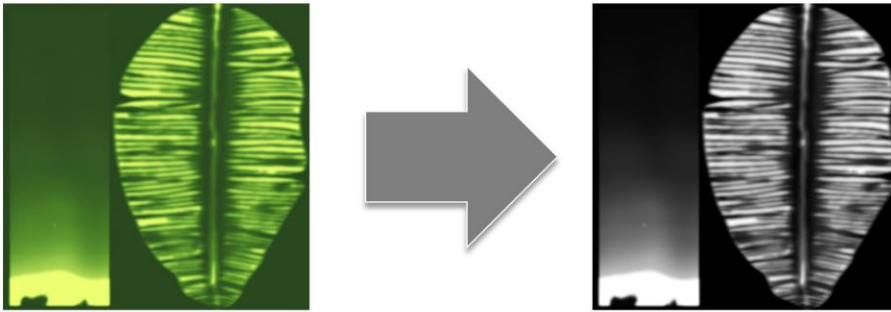
This technique can be used to reduce texture memory while maintaining visual fidelity. It will touch on some basic topics including texture packing, swizzling and linear interpolation.

# Memory Efficient Materials



The technique starts with surveying the game's content to identify assets whose diffuse maps are primarily mono or duo chromatic. In practice this technique works the best on things like plants and rocks, since these tend to consist of two main colors. We then pre-process the textures of those assets and do the following:

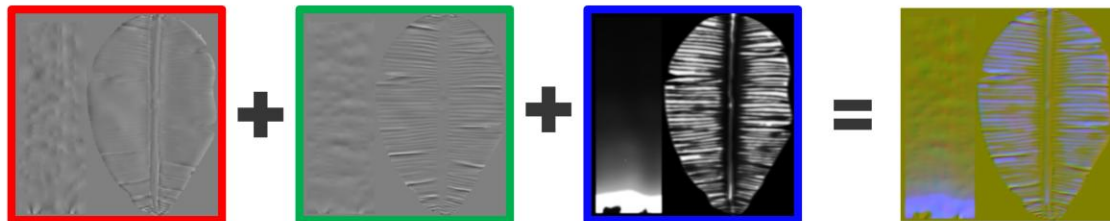
# Memory Efficient Materials



## 1. De-saturate Diffuse & Expand histogram

1. Convert diffuse to greyscale by de-saturating and then expand the histogram to get a full black to white range.

# Memory Efficient Materials



## 2. Copy into B channel of Normal Map

2. Copy grayscale diffuse into the B channel of the normal map

In most cases the B component of the normal maps is all or mostly a value of 1.0. We take advantage of this and throw it out so that we can use that channel to store a simplified version of our diffuse. Once the packing is done we can throw out the original diffuse, resulting in a significant memory savings.

# Memory Efficient Materials



## 3. Setup shader:

- re-construct normal and diffuse

3. Setup a shader that will re-construct the normal and diffuse.

In this example I've created a plant in Zbrush knowing that I intended to use this technique. I sculpted the plant, used cavity masking to darken the crevices and extracted that into a monochrome diffuse map. I then extracted the normals and performed the previously described packing in Photoshop.



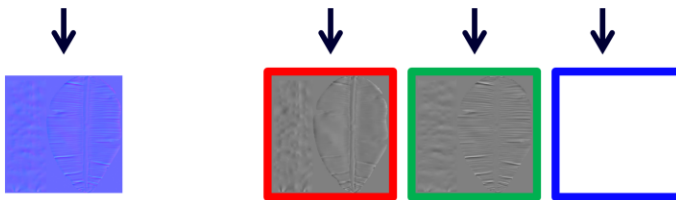
# Memory Efficient Materials

## Re-construct the normal



```
//Inside Pixel Shader ...  
//Sample the DiffNormPackMap  
float3 pack = tex2D(diffNormMapSampler, float2(UV.x, UV.y)).rgb;
```

```
//Reconstruct the normal  
float3 normal = float3(pack.r, pack.g, 1.0);
```



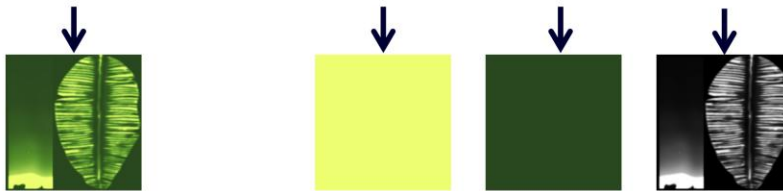
In the pixel shader we sample the diffNormPack map to get an RGB value, then reconstruct the normal by combining the R and G components with 1.0 for B. There is some loss of normal precision, but it's usually not too noticeable.

# Memory Efficient Materials

Re-construct the diffuse



```
//Define some user tweakable colors
float3 Start_Color : Diffuse = {0,0,0};
float3 End_Color : Diffuse = {1,1,1};
//... Skipping code for brevity
//Inside Pixel Shader ...
//Remap greyscale diffuse to Start/End color gradient
float3 diffuse = lerp(Start_Color, End_Color, pack.z);
```



We then use the B value to lerp between a user defined start color and end color. This allows us to add color back into the diffuse and also means that we have an easy way to make lots of material variations!

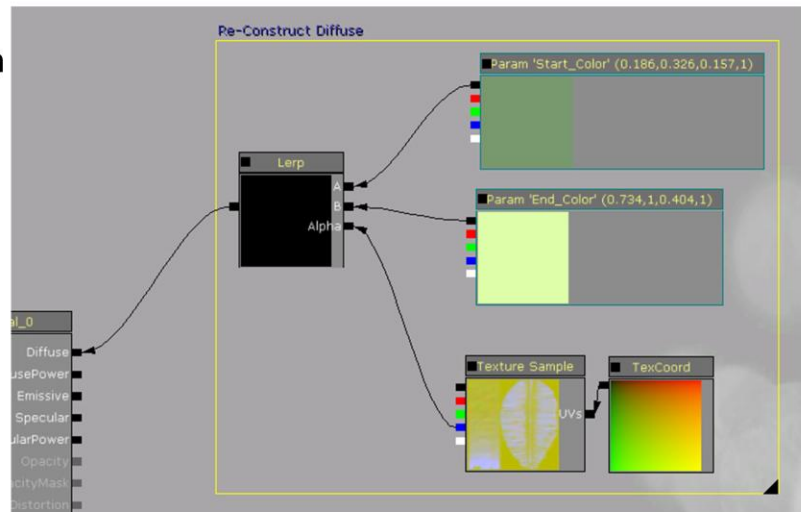
## UDK Node example – Normal reconstruction



273

# Memory Efficient Materials

UDK Node Example:  
Diffuse reconstruction



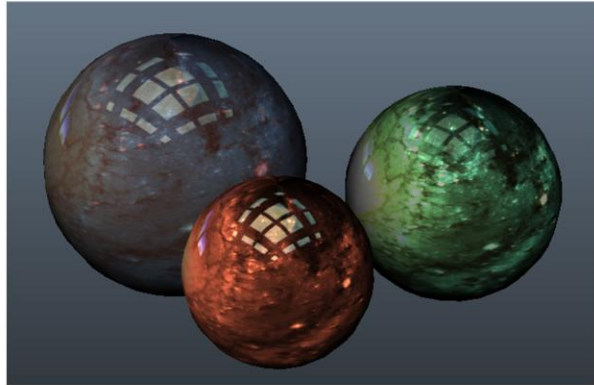
The diffuse reconstruction as a node network is fairly straight forward. A lerp between two vector parameters using the B component of our packed texture.



Here are some examples of the visual variety we can get with this approach and as you can see in the surface shading, the normal hasn't been noticeably degraded.

# Art Director Friendly Color Tweaking

- `pow()`
- Calculating Length



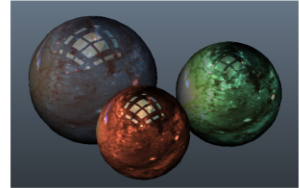
In this section I'll show a quick method for adding brightness/contrast, saturation and color tint controls to any shader. I've found this to be a useful addition in the past since it's allowed me to help the Art Director make quick adjustments without editing textures.

# Art Director Friendly Color Tweaking

```
float Brightness = 1.00; //Multiplier
float Contrast = 1.00;   //Exponent
float Saturation = 1.00;
float3 colorTint = {1.0f, 1.0f, 1.0f};

float3 colorCorrect(float3 inColor)
{
    float3 brightened = inColor*Brightness; //Multiply to Brighten
    float3 bc = pow(brightened,Contrast);

    ... function continued on next slide ...
```



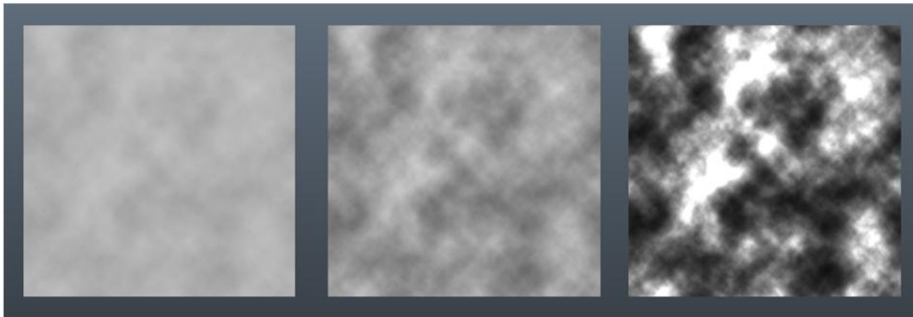
First we declare the user controllable parameters. To fit these on the slide I've removed the UI part of their declaration (you'll want to add that back in if you want the parameters to appear in Maya).

For example: `float Brightness <string UIName = "brightness"; string UIWidget = "slider"; float uimin = 0.0; float uimax = 2.0; float uistep = 0.01; > = 1.00;`

There's different ways to calculate brightness/contrast, but I've gotten a lot of mileage out of the simplicity of the following approach. I treat brightness as just a scalar multiply. For contrast I use `pow()`, with the user tune-able "Contrast" variable as the exponent input.



# Art Director Friendly Color Tweaking



Brightness=0.5  
Contrast=0.2

Brightness=1.0  
Contrast=1.0

Brightness=1.4  
Contrast=5.0

If you aren't familiar with what the `pow()` function can do for you visually, it's basically taking your incoming value and raising it to some power. This has the effect of either eroding or dilating your values depending on whether you use a value greater or less than 1.0. The combined result of the multiply and `pow()` gives us a final brightness/contrast result. The values don't match what you would use in Photoshop, but by tweaking them together you can have good control over the brightness and contrast range.

# Art Director Friendly Color Tweaking

```
//Previous bc variable from last slide for reference
```

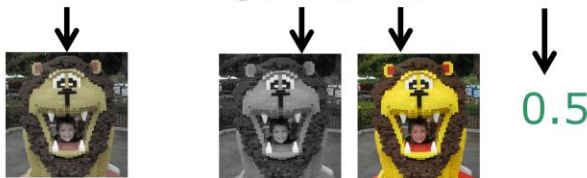
```
float3 bc = pow(brightened, Contrast);
```

```
//Get RGB Intensity
```

```
float3 mono = length(bc).xxx;
```

```
//Calculate Saturation Amount
```

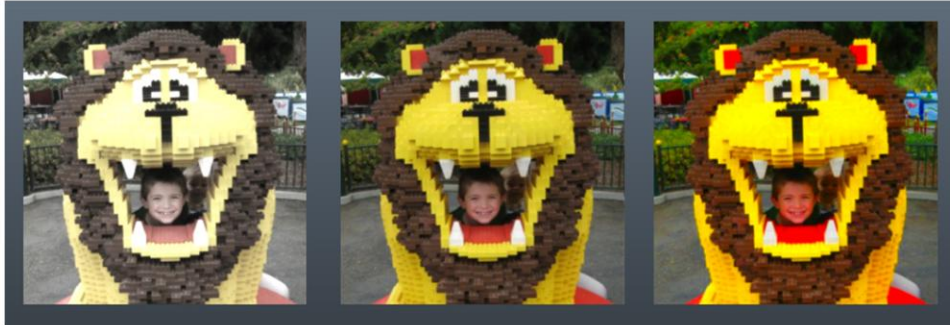
```
float3 satColor = lerp(mono, bc, Saturation);
```



For the saturation control we use `length()` to get the overall intensity of the brightness/contrast corrected color. This works because we can treat our RGB value like any other vector (such as normal), the shorter the length, the darker the color is. Since `length` is a single float value, we swizzle the result to get a greyscale float3 value. We can then `lerp()` between the greyscale version and the original brightness/contrast corrected color to allow varying degrees of saturation. As a side note, the `length()` command is equivalent to dotting a vector with itself and taking the square root of the result.

For more great color correction approaches, check out Chapter 22 of the Nvidia's GPU Gems. For example you could replace the `length()` approach above for calculating greyscale with the weight-based approach described in 22.3.1 of GPU gems.

# Art Director Friendly Color Tweaking



Saturation=0.5

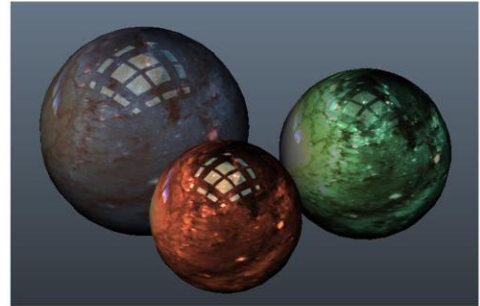
Saturation=1.0

Saturation=1.5

As you can see in the images above, this gives us good control over the saturation level. You can even overdrive the values past 1.0 to boost saturation.

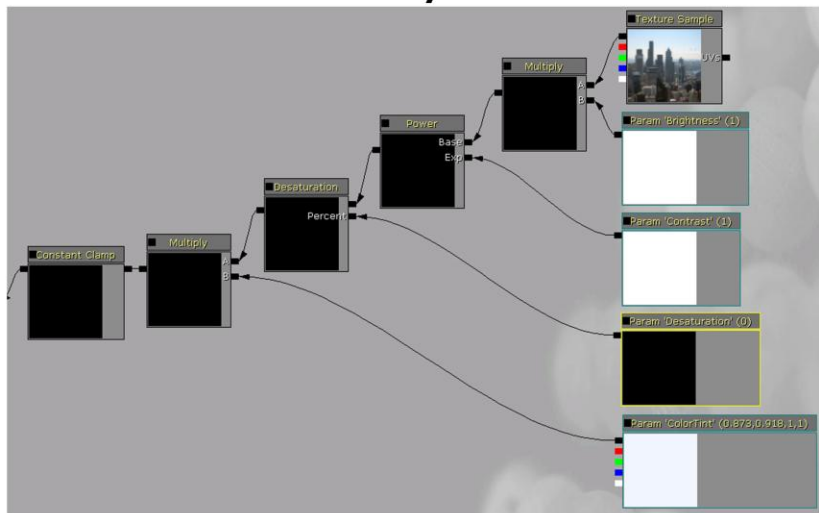
# Art Director Friendly Color Tweaking

```
//Clamp our final color to 0.0-1.0  
return min(max((satColor*colorTint),0),1.0);  
}  
//End of colorCorrect function
```



Finally we take the output of the saturation operation and multiple it by a user controllable color tint. This allows us to cheaply adjust the hue. The combination of these four tweakable parameters gives plenty of control for making variations on base diffuse textures and for giving your Art Director knobs to tweak that can vary asset to asset.

## Art Director Friendly Color Tweaking



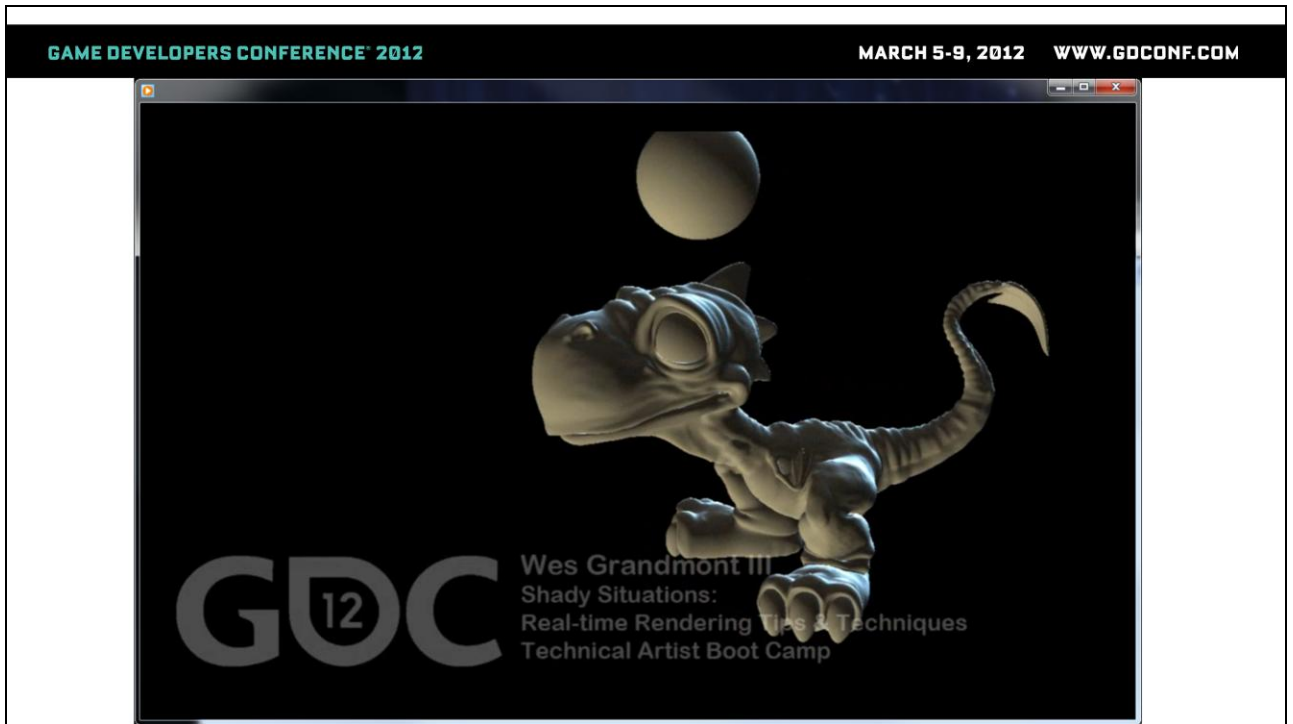
As a node network in UDK this technique looks similar to the following example. The main difference is that I used Unreal's built-in desaturate node instead of rolling my own in a custom node.

# Auto Rim Light

- dot()
- Basic Lighting
- Fresnel
- Lighting Customization



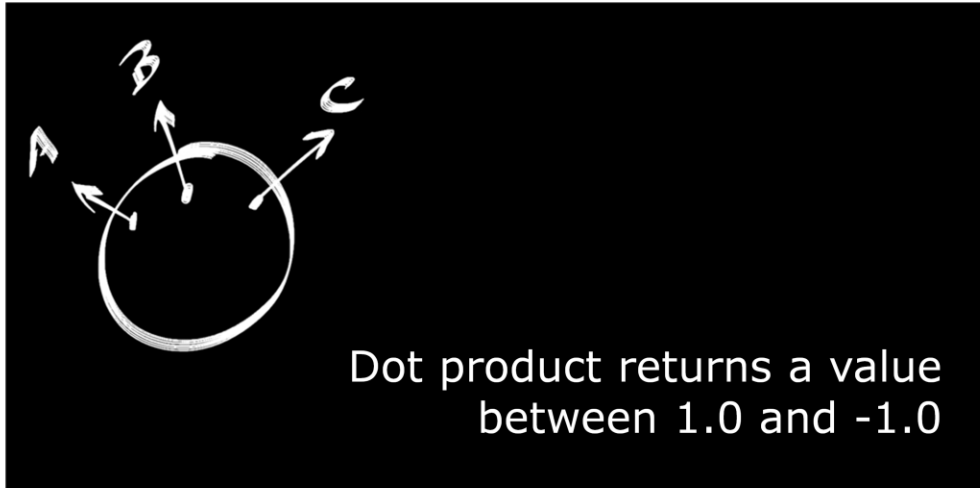
For this example, I'm going to demonstrate how to add a rim light that fades on as the camera's view vector faces the same direction as the key light and fades off as the view moves towards the back-side of the model. While not physically correct, this is a useful tool for helping to separate characters from dark backgrounds. Since it's view-dependent, the rimlight won't look blown out when the character is viewed from behind. The technique can also be adapted to achieve other view dependent effects.



Demo Movie



# Auto Rim Light

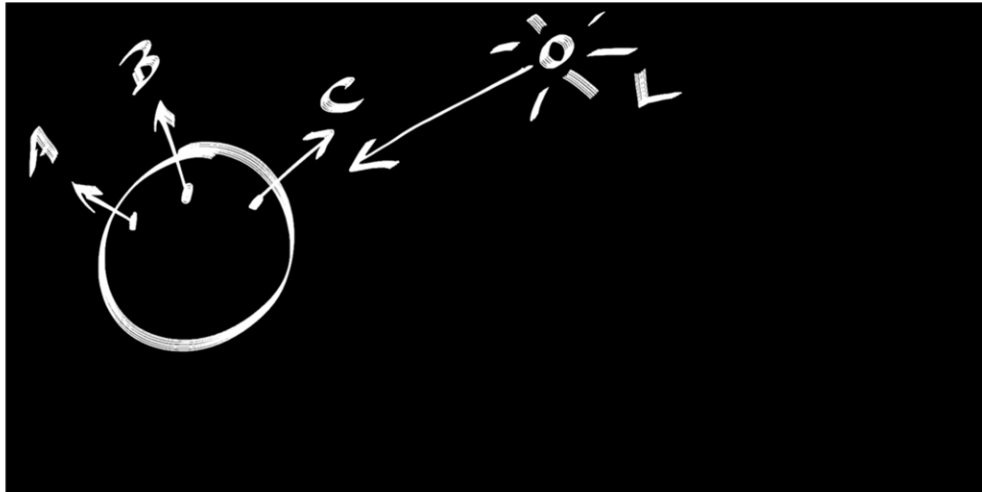


Understanding basic lighting calculation is essential in order to control the look of your shaders. Central to the standard lighting model is the dot product operator which tells us how much one vector is facing towards another vector. It returns this comparison as a value between 1 and -1.

1 means the first vector is facing the second vector, -1 means it is facing away. In the case of a simple directional light, we want to compare the surface normal at the point we are shading to the light vector to determine how much it is facing the light and shade it accordingly.

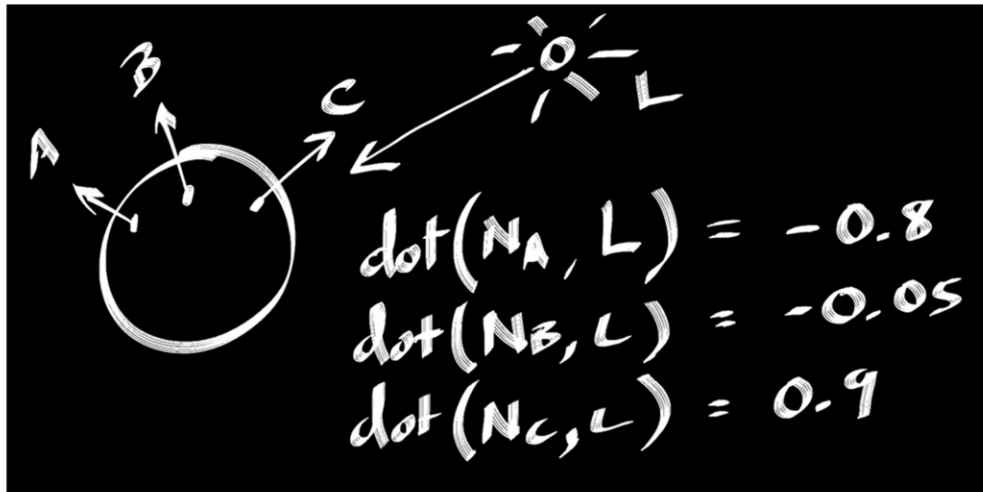
In this example we have a sphere with a surface normal at each point we will be shading. Normals are vectors whose length is always 1 and that are orthogonal (perpendicular) to the surface at each point. Here we've got three of them each facing in a slightly different direction based on the curvature of the sphere.

# Auto Rim Light



Here's our sun light vector. The light vector also has a length of 1.0. In dot product calculations all participating vectors must be normalized (meaning they must have a length of 1.0 in order to get an accurate result from the dot product calculation). You can use `normalize()` to insure that your vectors are 1 unit in length.

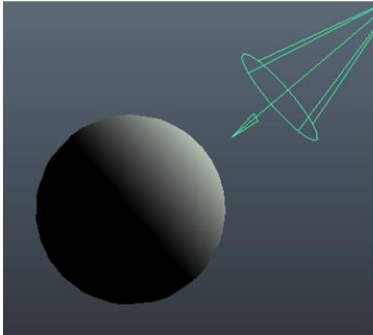
# Auto Rim Light



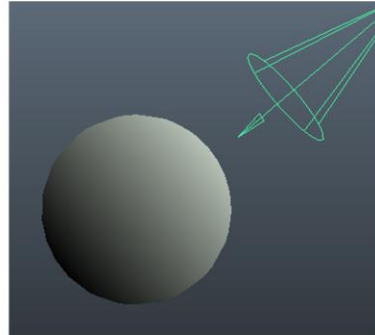
At each point that is to be shaded, we use the dot operator to compare the normal to the light vector. As you can see, Normal C is almost facing the light vector so the result of that dot product calculation is close to 1.0. The other normals are starting to face away from the light vector so they have started to go towards -1.0.

NOTE: By drawing this as a picture the math has become more "fun"...really.

# Auto Rim Light



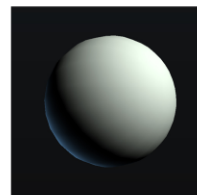
$\max(\text{dot}(N, L), 0.0)$   
Standard Lighting



$(\text{dot}(N, L) + 1.0) * 0.5$   
"Wrap" Lighting

We can do a lot of interesting things with the result of the `dot()` operator. For example we can clamp the value so that anything below zero is just zero (standard lighting math) or we can do a "wrap light" where we add 1.0 to the result and then multiply by 0.5 so that instead of a value between 1 and -1 the value will range from 1 to 0 (or any variation on these). Once we have the clamped result of our dot product calculation we can either immediately apply it via a multiply operation or use it to drive something else such as interpolating between two colors.

# Auto Rim Light



```
//In the pixel shader...
float3 N      = IN.Normal;
float3 L      = normalize(LightKeyDir.xyz);

//Get view position
float3 viewPos      = viewInverseMatrix[3];

/* Could also be written long-form as ...
float3 viewPos      = float3(      viewInverseMatrix[3][0],
                                viewInverseMatrix[3][1],
                                viewInverseMatrix[3][2]);
*/
```

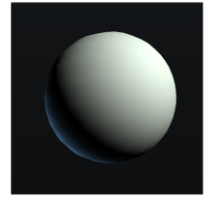
So now that everyone understands the role of the dot product lets look at how we can use this great bit of math to add a rimlight to our initial lighting calculation. First we're going to need a few vectors to work with. The ones we're interested in are the light vector, surface normal and the camera view vector. As I previously mentioned these need to all be in the same transform space or the math won't work out.

We get the normal and key light vector from the application, but we need to calculate the view vector. We do this by subtracting the world position of the point being shaded from the camera world position. We passed the world position of the point in from the vertex shader so we just need the camera world position.

We can get the camera world position from the ViewInverseTranspose matrix which is passed in by the application. This is a 4x4 matrix. In OpenGL this means that the last column of the matrix contains the camera's world space translation values. (0 would be the first column, so the

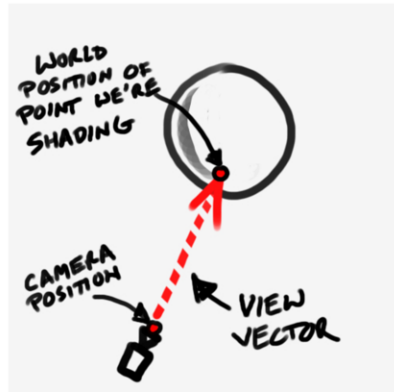
fourth column of the matrix is found at index 3).

# Auto Rim Light



```
//Calculate the view vector
```

```
float3 V = normalize(viewPos.xyz-IN.worldPos.xyz);
```

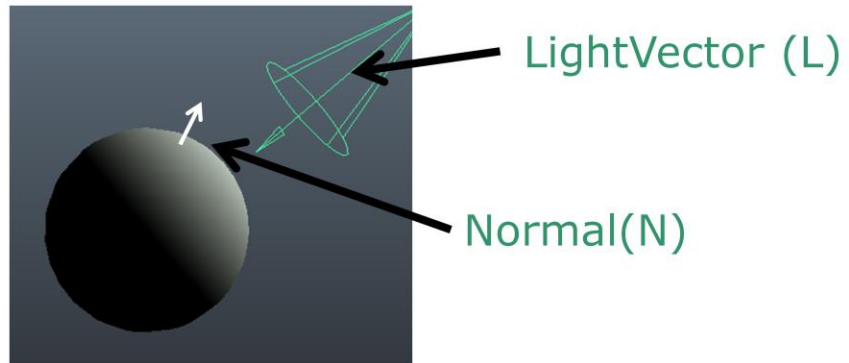


We subtract the camera position from the world position and normalize to find the view vector.



# Auto Rim Light

```
//Get dot between normal and keylight to get keylight contribution  
float3 keyLight = max(dot(N,L),0) * keyColor;
```

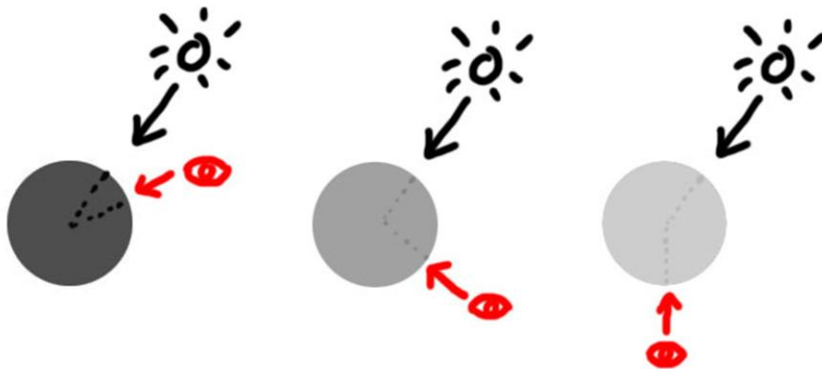


We've got all the vectors we need so now we can proceed with the lighting calculations.

First we calculate the lighting contribution for the keylight using the standard lighting I described earlier and color tint it with a user-tunable keyColor parameter.

# Auto Rim Light

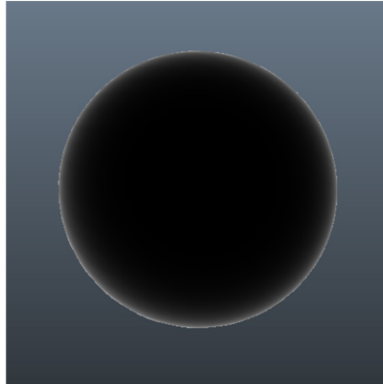
```
//Get the clamped dot between viewVector and keyLight  
float vDotL = min(max(0, dot(V,L)), 1);
```



Next we're going to calculate the clamped dot between the view vector and the light. This tells us how much the camera is facing in the same direction as the light. As the camera moves around the object, this value will dynamically change which will cause the rimlight to fade off as we face toward the light and back on as we face away from the light. We'll be inverting the result of this value to get the result we want

# Auto Rim Light

```
//Get the clamped and inverted dot between view and normal  
float fresnel = min(pow(1-max(dot(N,V),0),3),1.0);
```



Finally we're going to calculate a fresnel term which tells us how much the surface being shaded is facing the camera. We do this by getting the dot product between the view vector and the normal. We then use the `pow()` operation. As I mentioned in the last technique, the `pow` operation acts as a contrast function, which will allow us to control how fast the fresnel effect falls off. This will help isolate the rim light to the edges of the surface.

# Auto Rim Light

```
float3 rimColor = {0.6f, 0.6f, 1.0f};
```

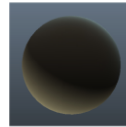
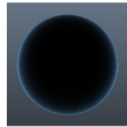
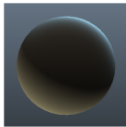
```
//Use vDotL to interpolate between min/max rim intensity.
```

```
float3 rimLight = lerp(2.0,0.2,vDotL) * rimColor * fresnel;
```



```
//Add rimLight to the final output color and your done!
```

```
float 3 finalColor = rimLight + keyLight;
```



We can now use vDotL to interpolate between a minimum and maximum intensity for the rim light. When the camera is facing towards the light the max value will be used and as it faces away the rimLight value will fade towards the minimum value. I try to keep the minimum value greater than 0.0 to prevent it from completely disappearing, but you can of course tune this to suite your needs.

Normally when filling in a lerp(A,B,C)

A=min

B=max

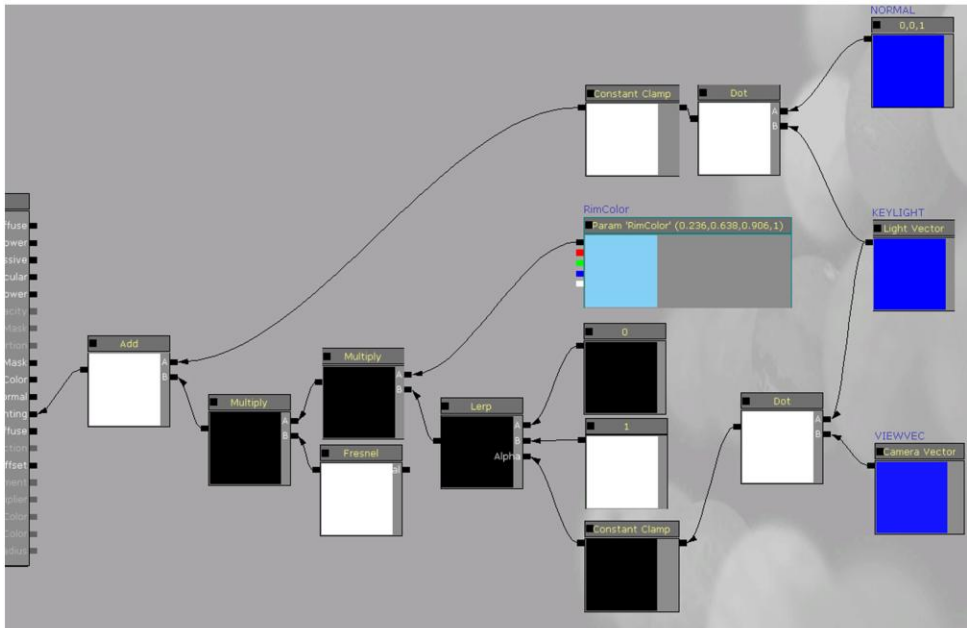
C=interp value

In our case we're switching A and B since when we are in shadow we actually want the rimlight to be at full brightness

Finally we multiply by a user tune-able rim color to tint the final light.

We finish up by adding our rim to the initial keylight contribution we calculated at the beginning.

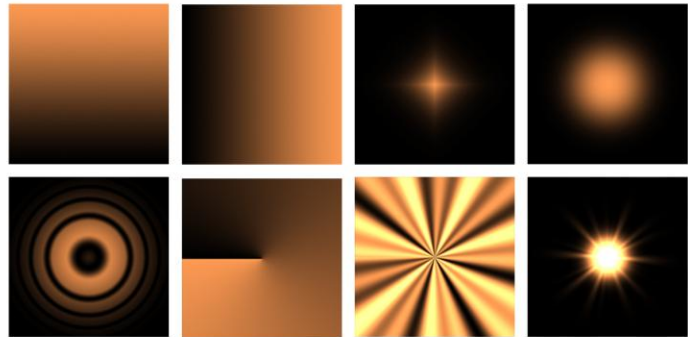
# Auto Rim Light



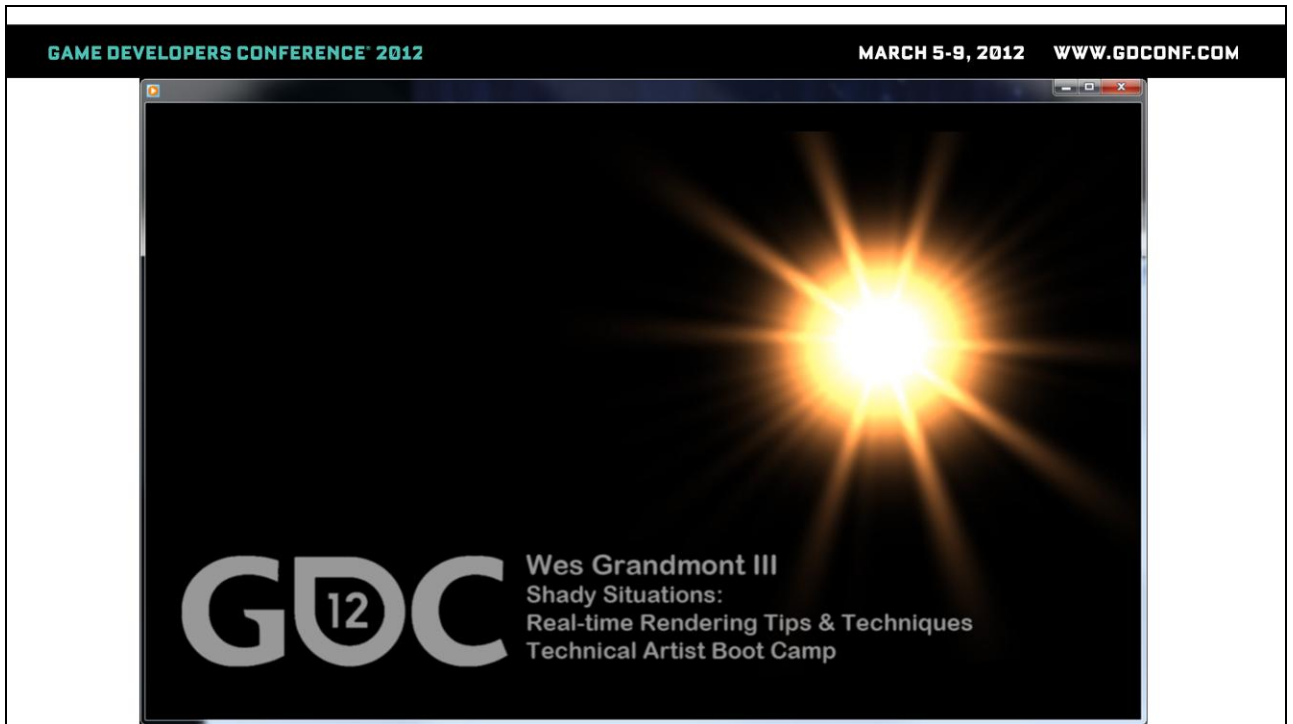
For those of you who work in the land of material networks, here's what this setup looks like in Unreal. This network is plugged into the custom lighting input on the material block.

# Procedural Gradients

- UV Texture Synthesis
- TIME
- $\sin()$
- $\text{atan2}()$



Many times in order to achieve subtle gradients, flares and other effects you get better results if you use a procedurally generated gradient rather than a texture map. In this next section we discuss how to use some simple math to create a variety of procedural texturing effects that will give you great results, real-time interaction and zero hit to your texture memory.



Demo Movie



# Procedural Gradients

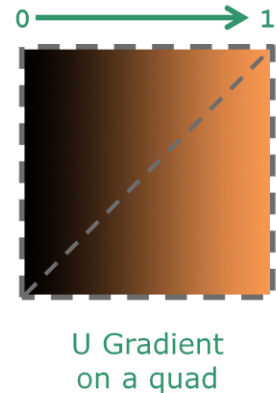
## Linear

```
float3 start = float3(0.0,0.0,0.0); //Black
float3 end   = float3(1.0,0.6,0.3); //Orange
float falloff = 1.0;

//Use U as blend value
float blend = IN.UV.x;

//contrast control
blend = pow(blend, falloff);

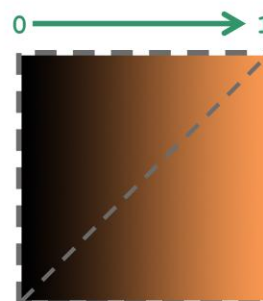
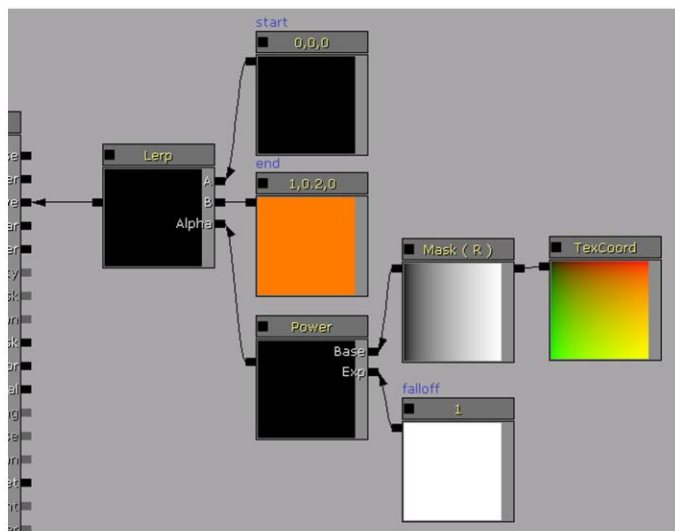
float3 color = lerp(start, end, blend);
```



Okay so we know how a `lerp()` works and that if we have a value between 0.0 and 1.0 we can use that to blend between two colors. We also know that UV coordinates are often normalized between 0.0 and 1.0 in UV space (and even if they aren't we can normalize them so that they are). Armed with these two bits of information we can easily write shading code to render a gradient on any polygons that have UV's.

We can create a simple horizontal gradient by using the U component of the incoming UVs as the blend value in a color `lerp()`. We can also add a power function to allow the user to adjust how quickly the color ramp falloffs as it blends to the second color.

# Procedural Gradients



U Gradient  
on a quad

Here's what that looks like as a material network

# Procedural Gradients

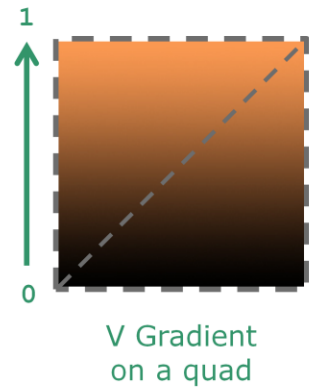
## Linear

```
float3 start = float3(0.0,0.0,0.0); //Black
float3 end   = float3(1.0,0.6,0.3); //Orange
float falloff = 1.0;

//Use V as blend value
float blend = IN.UV.y;

//contrast control
blend = pow(blend, falloff);

float3 color = lerp(start, end, blend);
```



Similarly we can use the V component of the incoming UVs as the blend to create a vertical gradient.

# Procedural Gradients

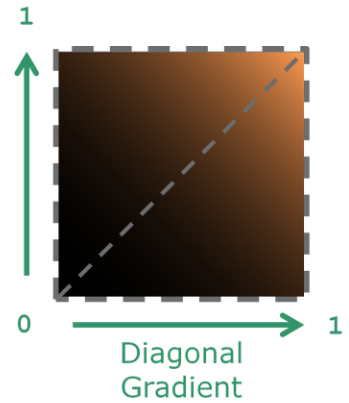
## Diagonal

```
float3 start = float3(0.0,0.0,0.0); //Black
float3 end   = float3(1.0,0.6,0.3); //Orange
float falloff = 1.0;

//Average U and V
float blend = (IN.UV.x + IN.UV.y) * 0.5;

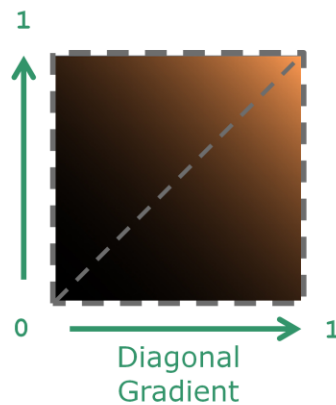
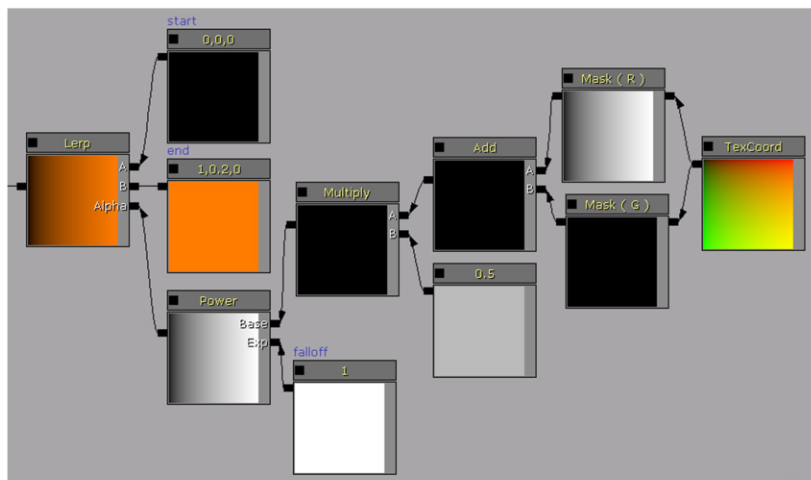
//contrast control
blend = pow(blend, falloff);

float3 color = lerp(start, end, blend);
```



And we can make a diagonal gradient by adding U and V together and then multiplying by 0.5 (averaging them).

# Procedural Gradients

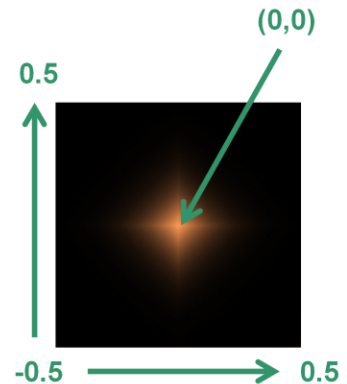


Here's what that looks like as an Unreal shading network

# Procedural Gradients

## Tartan

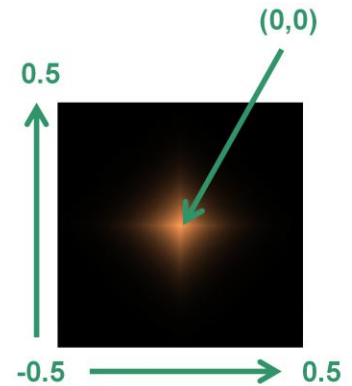
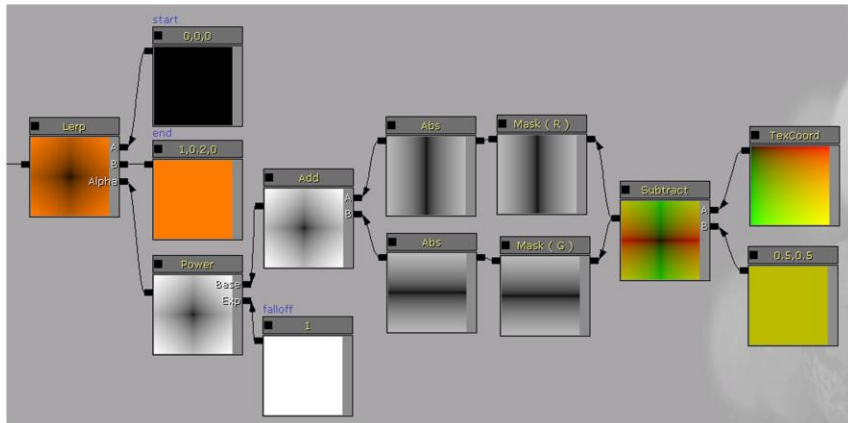
```
//Shift UV origin  
float2 uv = IN.UV.xy-float2(0.5,0.5);  
  
//Add abs value of u and v  
float blend = abs(uv.x) + abs(uv.y);  
  
//contrast control  
blend = pow(blend, falloff);  
  
float3 color = lerp(start, end, blend);
```



We can extend the diagonal gradient example further and create a tartan or diamond gradient. First we shift where the origin of UV space to the center of the quad by subtracting - 0.5 from both U and V. We then take the absolute value of the U and V components and add them together. This gives us four diagonal linear gradient that falloff in each of the four quadrants.

# Procedural Gradients

## Tartan

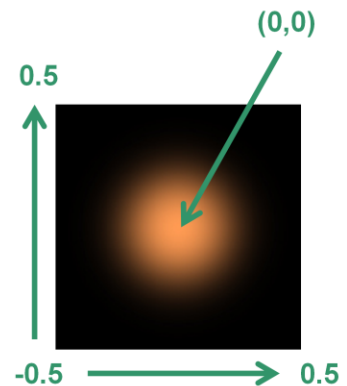


Here's what that looks like as an Unreal shading network

# Procedural Gradients

## Spherical

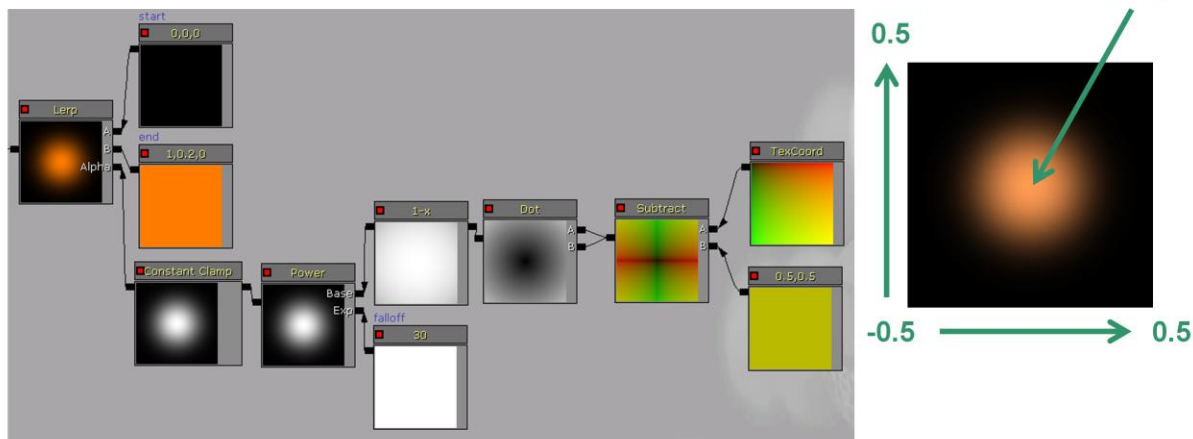
```
//Shift UV origin  
float2 uv = IN.UV.xy-float2(0.5,0.5);  
  
//Dot uv with itself  
float blend = dot(uv,uv);  
  
//contrast control  
blend = pow(blend, falloff);  
  
float3 color = lerp(start, end, blend);
```



From the tartan example we can make a minor change and we'll have a spherical gradient. Instead of adding the U and V components together we dot() it with itself. Getting the dot product of a 2D or 3D vector with itself returns the length of the vector squared. In this case we're going to skip the square root operation and instead rely on our pow() function to allow us to contract the radius of the spherical gradient.



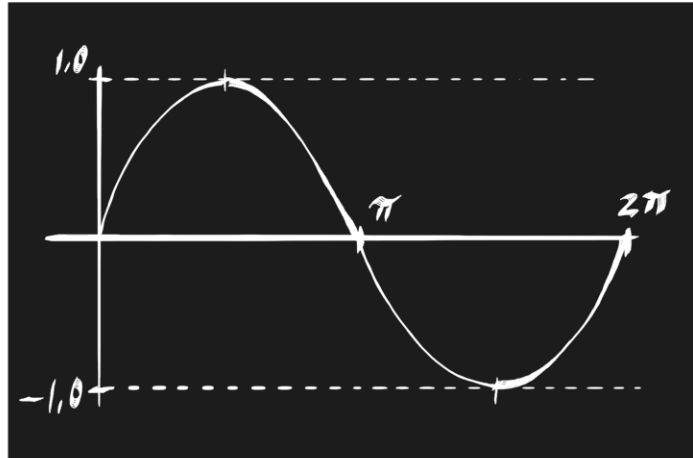
# Procedural Gradients



Here's what that looks like as an Unreal shading network

# Procedural Gradients

`sin()`  
Sine Wave

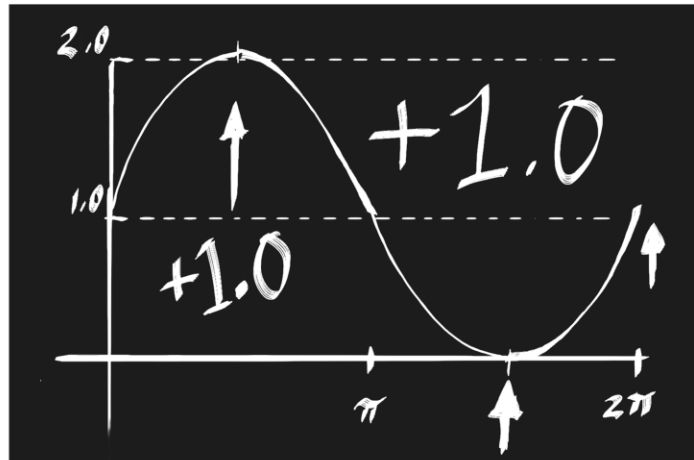


Let's take quick break and talk about the `sin()` function. Sine returns a value that ranges between 1 and -1 and repeats every  $2\pi$  ( $\sim 6.28$ ). Half the time the value we get back from `sin()` is negative. If we want it always to be a positive value we can do the following:

# Procedural Gradients

## Normalization

1.) Add 1.0

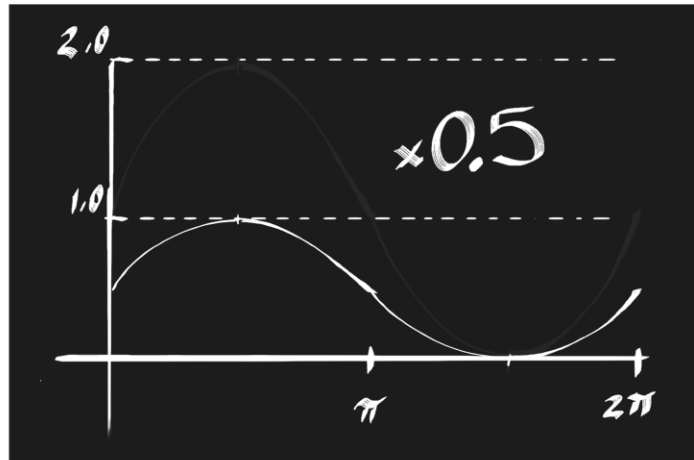


First we add 1.0 to whatever we get back from the `sin()` function. This shifts the entire graph. It now ranges from 0.0 to 2.0 instead of from -1.0 to 1.0.

# Procedural Gradients

## Normalization

2.) Multiply by 0.5



Then we multiple the value by 0.5 which scales down the graph by half which means it now ranges from 0.0 to 1.0. Now we have a normalized value from the `sin()` function that we can use for rendering.

# Procedural Gradients

## Spherical with Wave

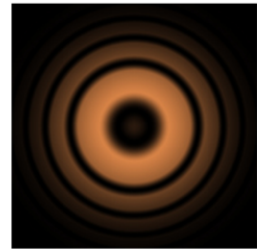
```
float time : TIME; //Get time from the app
time = time * 0.2; //Use a small fraction of it

float2 uv = IN.UV.xy-float2(0.5,0.5);
float blend = dot(uv,uv);
blend = pow(blend, falloff);

float freq = 20.0; //Controls number of rings

float wave = (sin((blend + time)* freq) + 1)*0.5;

float3 color = lerp(start, end, wave );
```



So to create a simple animated radio beacon or shockwave effect we can use a `sin()` in our spherical gradient calculation. First we use the `TIME` semantic provided by our application to get a time value that will drive our animation. We do our previous spherical gradient calculation to generate a blend value.

We then create a wave value by adding time to our blend value, scaling that with a frequency value to control how many rings we're going to get and then pass that to the `sin()` function. We then normalize the value we get back from `sin()` using the two step process I just described: Add 1.0 and then multiply by 0.5.

Final we do our `lerp()` with the wave to get the final color.

Here's what that looks like as an Unreal shading network

# Procedural Gradients

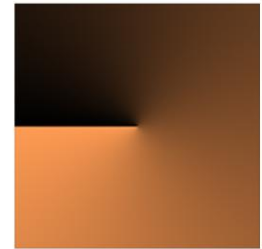
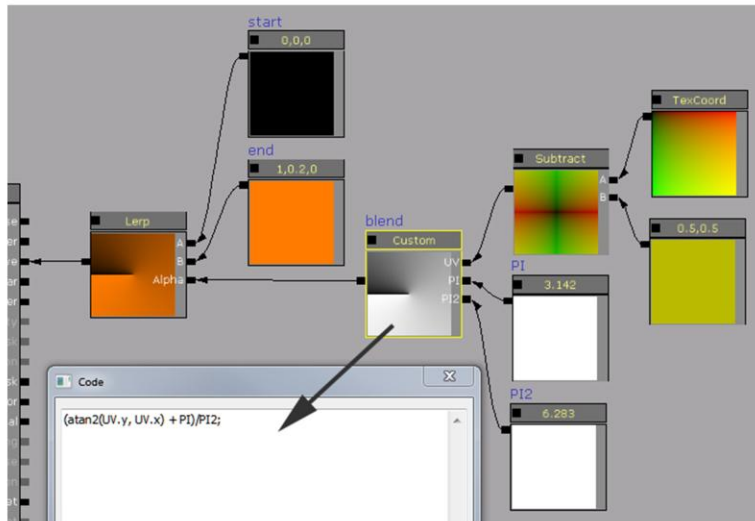
## Radial

```
float PI = 3.141592653589793238462643383279;  
float PI2 = 6.283185307179586476925286766559;  
  
//Offset the UVs  
float2 uv = IN.UV.xy-float2(0.5,0.5);  
  
//Use atan2 to create radial gradient  
float blend= (atan2(uv.y, uv.x) + PI)/PI2;  
  
blend = pow(blend, falloff);  
  
float3 color = lerp(start, end, blend);
```



Creating a radial gradient requires the use of the `atan2()` function. The `atan2` function returns the angle in radians between the +x axis of a plane and an input 2D vector. Our input vector is our offset UV value. The radian angle is returned as a value between  $-\pi$  to  $\pi$ . In order to normalize this value, we perform steps similar to what is needed to normalize the `sin()` value. In this case we add  $\pi$  and then divide by  $2\pi$  to get a value that ranges from 0.0 to 1.0.

# Procedural Gradients



Here's what that looks like as a material network in Unreal. Unreal doesn't expose the `atan2` function, but we can do the calculation in a custom node.



# Procedural Gradients

## Radial Noise

```
float time : TIME; //Get time from the app
time = time * 0.2; //Use a small fraction of it

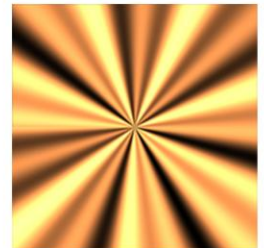
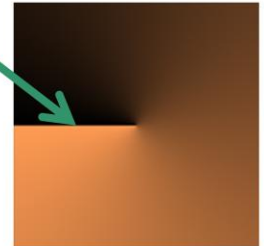
float2 uv = IN.UV.xy - float2(0.5, 0.5);
float blend = (atan2(uv.y, uv.x) + PI) / PI2;

float freq1 = 10.0 * PI2;
float freq2 = 20.0 * PI2;

float wave = sin((blend + time) * freq1 + 1) * 0.5;
wave = wave + sin((blend - time) * freq2 + 1) * 0.5;

float3 color = lerp(start, end, wave);
```

seam

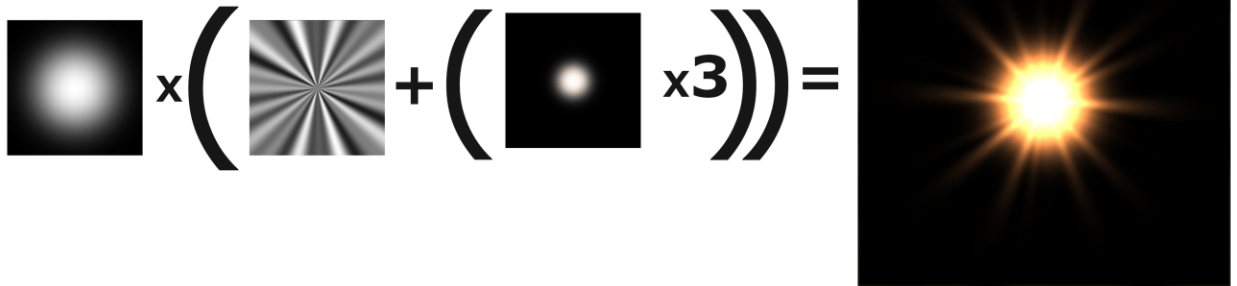


Similar to the way we used the `sin()` function to add waves to our spherical gradient. We can use `sin()` to create radial noise. In this case we'll use two `sin` calculations to create wave interference which will give us some organic breakup to the repeating sine pattern. Our gradient starts and ends at the same vector as it sweeps around the quad. We don't want to see a visible seam where the start and end match up. Recall that `sin()` repeats every  $2\pi$ . So if we multiple our frequency values by  $2\pi$  we can guarantee that end of the `sin` graph will match up with its beginning.

And again here's what that looks like as a material network...

# Procedural Gradients

## Lens Flare



As I mentioned at the beginning you can combine these gradient techniques to create a range of effects. One example is a lens flare. If we combine the spherical gradient, radial and radial noise gradient we can get a high quality animated lens flare effect without using a single texture.

# Bulgy Pipes

- Mesh Pre-processing
- Sine Wave Manipulation
- Vertex Deformation



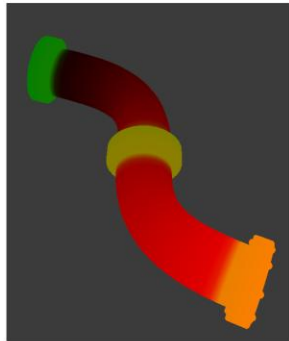
Ambient environment animation is quick to implement in a vertex shader. In this section I'm going to discuss techniques for achieving controllable deformations via the creation and setup of a cartoony bulging pipe example. This example will cover mesh pre-processing, sine wave manipulation and vertex deformation.

# Deformations

- Content Prep



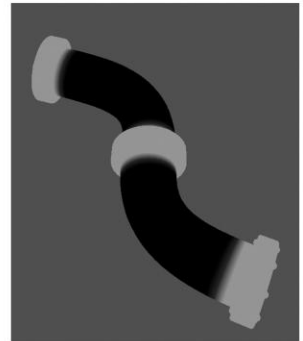
Mesh



Vertex Colors



Red Channel



Green Channel

To begin we need to author our model in a way that will allow us to control the deformation in the shader. After creating the model, we need to parameterize the vertices so that we can control how the deformations are applied. There's two ways to add parameterization. We can use one or more UV sets and/or we can make use of the vertex colors. I'm a fan of using vertex colors for a number of reasons:

- Easy to visualize
- Can be painted
- Can be more easily edited

- 1.) So as a first step I've modeled this bendy piece of pipe with couplings and rivets and setup UV's
- 2.) As a second step I apply a surface shader with a red to black ramp and bake this color into the vertices of the pipe. This coloring represents and 0.0 to 1.0 parameter range across the surface.
- 3.) I then hand apply green to the areas around the couplings.

Later I'll use the green channel of the vertex colors to act as a mask to prevent deformation from occurring in these areas.

The model has now been pre-processed with all the built-in information that we'll need to help control our animation.

## Bulgy Pipes

```
//Constants
float PI = 3.1415926535897932384626433832795;
float PI2 = 6.283185307179586476925286766559;

//Variable parameter declarations

float Flow </*...*/> = 0.50; //Anim control ranges from 0.0-1.0

float BulgeLength </*...*/> = 0.15; //Range in 0-1 parameter space

float BulgeScale </*...*/> = 5.00; //Max displacement of vertices
```

Now we need to setup the vertex shader that will create the deformation effect.

First we'll need to declare some constants to use in the vertex shader. We'll be using both PI and PI2.

Next we define some variables that the user can tweak to control the bulging effect:

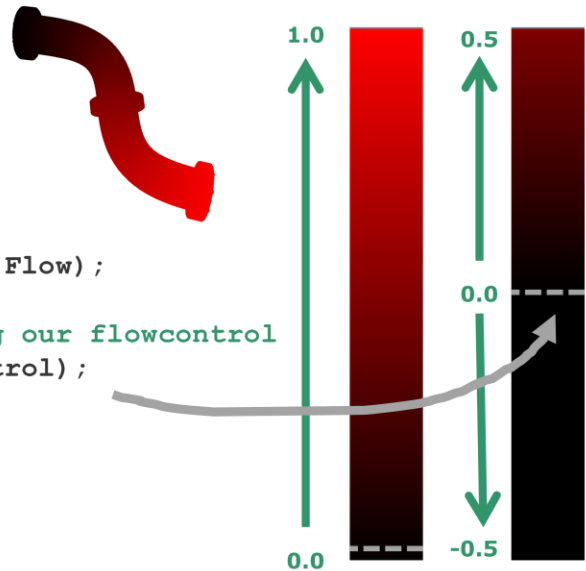
Flow = This value will range from 0.0 to 1.0 and will represent the flow of liquid from the start to the end of the pipe. This is the animation control.

BulgeLength = This value represents the length of the bulging area in relation to the length of the entire pipe.

BulgeScale = This value represents how far the points will be displaced away from their original position as the deformation passes through the pipe.

## Bulgy Pipes

```
//In the vertex shader...  
  
//Remap our animation control  
float flowcontrol = lerp(-0.5,1.5,Flow);  
  
//Shift the parameterization using our flowcontrol  
float center= (IN.Color.r-flowcontrol);
```



In the vertex shader we remap our user-defined Flow parameter to a range that will allow the bulge to travel outside the bounds of the pipe. This allows us to present a more intuitive min/max range in the UI and then adapt here in the vertex shader.

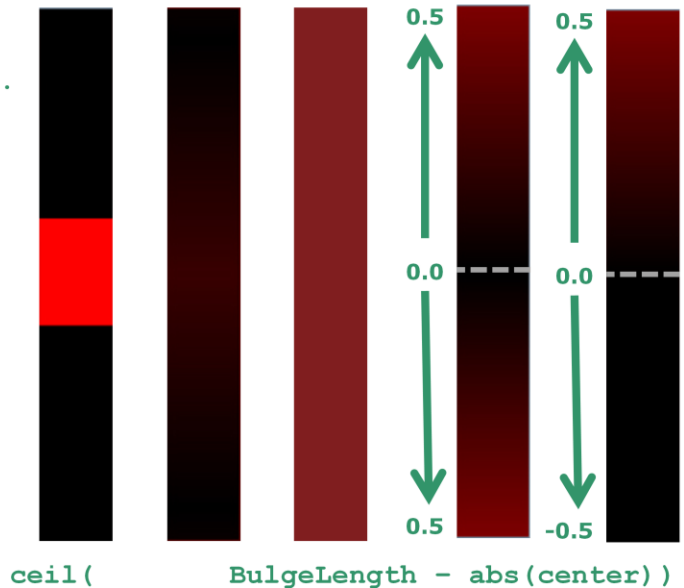
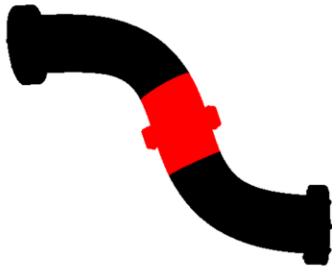
Next we calculate a new center point that will allow us to place the bulge anywhere along the length of the pipe. We do this by subtracting our flowcontrol value from the red vertex channel parameterization we setup on the model. This moves the zero point of the parameterization to where-ever our flowcontrol value is (see image above).



## Bulgy Pipes

//Still in the vertex shader...

```
float mask =  
ceil(BulgeLength-abs(center));
```



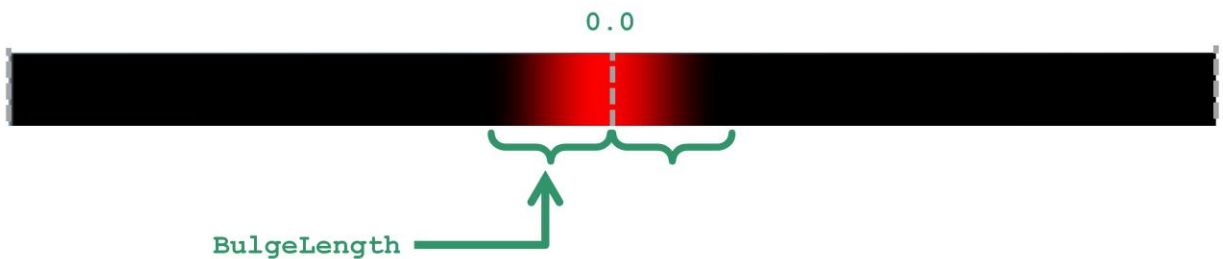
We now need to calculate a mask so that our bulge only effects vertices that are less than BulgeLength distance from the center we calculated. We do this by first getting the absolute value of **center**, then subtracting the absolute value of center from BulgeLength. Finally we use a ceil() function to push any non-zero values to 1.0. This gives us a hard mask that we can use later to isolate our deformation effect.

As an aside this technique is great for doing all sorts of effects in the pixel shader as well!

# Bulgy Pipes

What we need:

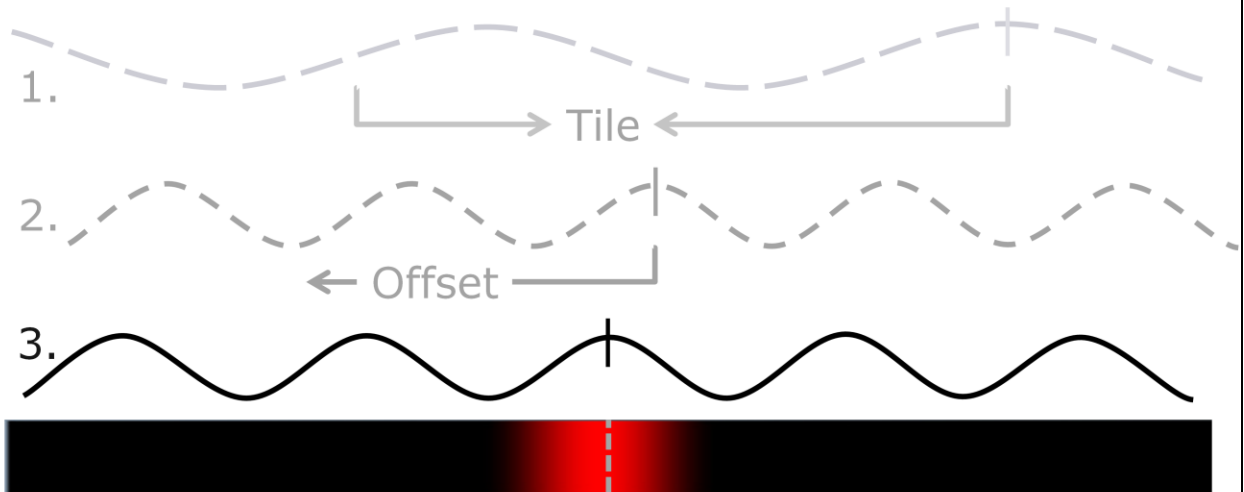
- sine wave aligned to center
- peak to trough = bulge length



Next we need a bulge gradient that has smooth falloff so that we will get a nice center belly to the bulge and a falloff to the non-deforming areas. The sine function will come to our aid (you can also use the cosine function as an alternative).

Sine waves have a smooth ease-in and ease-out at the crest and trough of the wave. We need to calculate a sine wave whose crest is centered over our bulge center and whose distance from crest to trough matches our bulge length. We want to end up with something like what's pictured above.

# Bulgy Pipes

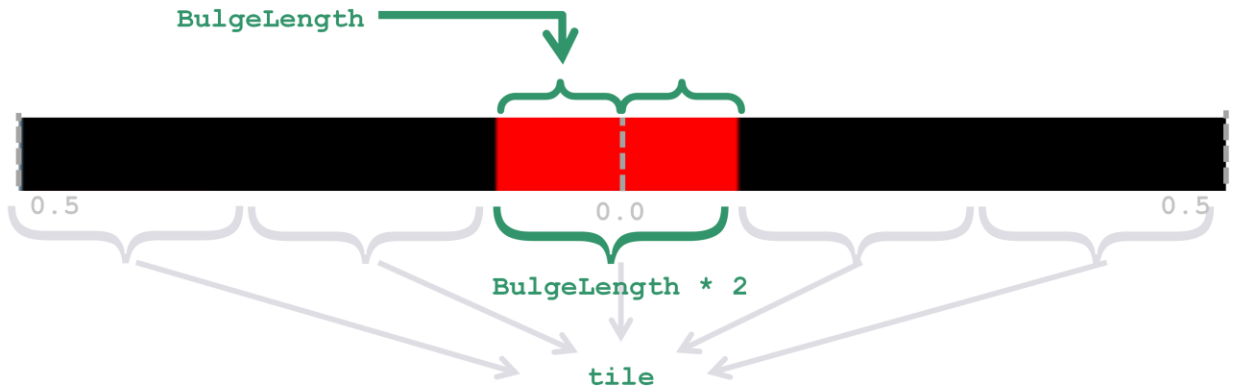


When we feed a gradient value (that is a value that changes over a range of pixels) to the sin function (in this case it's our red vertex color) we get smoothly repeating wave. What we want is to determine the right value to feed the sin function so that this wave pattern repeats at the right frequency and is aligned to our center. In order to do this we are going to need to tile and offset the sine wave to get it positioned correctly.

# Bulgy Pipes

//Still In the vertex shader...

```
float tile = 1.0/(BulgeLength*2);
```



We need to determine the desired distance between repeats in the sine wave. We'll call this variable "tile".

Recall that in order to determine the mask we subtracted the absolute value of the center from our BulgeLength. This means that we were using values on both sides of the 0.0 center line which means that the total length of our bulge is actually  $2.0 \times \text{BulgeLength}$ .

In order to determine the tiling we need to know how many times the  $2.0 \times \text{BulgeLength}$  repeats inside our parameter range which is 0.0 to 1.0. So we divide 1.0 by the  $2 \times \text{BulgeLength}$  to determine the tiling.

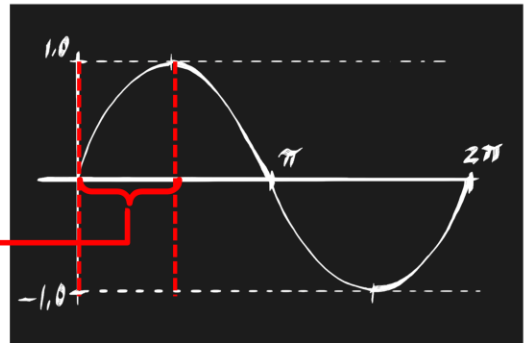
# Bulgy Pipes

//Still In the vertex shader...

```
float halfPI = (PI*0.5);  
float sineInput = (center*tile*PI2) + halfPI;  
float falloff = (sin(sineInput)+1)*0.5; //get sine and normalize it
```



Half PI ←



We want our parameter range to tile within the 0.0 to 1.0 range so we multiply the center variable by our tile variable.

A sine wave repeats every  $2\pi$ , so if we multiple our tiled range by  $2\pi$  we'll see the sine wave repeat that many times within the parameter range (similar to how we made the radial gradient repeat in the lens flare example earlier).

The sine wave doesn't start at the crest or trough of the wave, but one half  $\pi$  to the left. If we want to align the crest to the center then we need to add half  $\pi$  to our sineInput. Now that we have our calculated sineInput we pass it to the `sin()` function.

Finally we use the two step process described earlier to normalize the sine value (add 1.0 and multiply by 0.5)

Alternatively, you can replace the HLSL in this slide with the following:

```
float cosInput = (center*tile*PI2);  
float falloff = (cos(cosInput)+1)*0.5;
```

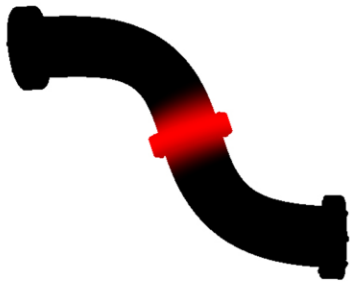
The cosine wave starts with the crest already centered over the center (0) line so we can skip the addition of halfPI

# Bulgy Pipes

```
//Still In the vertex shader...
```

```
float bulge = falloff * mask;
```

```
float offset = bulge * BulgeScale * (1-IN.Color.g);
```



mask



+

falloff



=

bulge

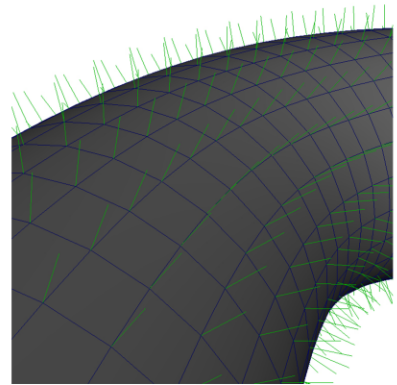


Now that we have our bulge falloff we multiply it by our mask value to get our final bulge. We then multiple the bulge by our BulgeScale to get the finally offset value for the vertices.

# Bulgy Pipes

```
//Still In the vertex shader...
float4 oPos = float4(0,0,0,1);
oPos.xyz =
IN.Position.xyz + (normal * offset.xxx);

float4 wPosition = mul(wvpMatrix, oPos);
float4 wNormal = mul(witMatrix, float4(normal,1));
```



We multiple our offset by the vertex normal and add that to the local position to get the final displaced position of our vertices.

And then finally we transform the vertex into worldviewprojection space.

We can also transform the normals depending on how we are doing our lighting. It's important to note that our normals are not getting deformed by this process, but when this effect is animating it's hard to notice.

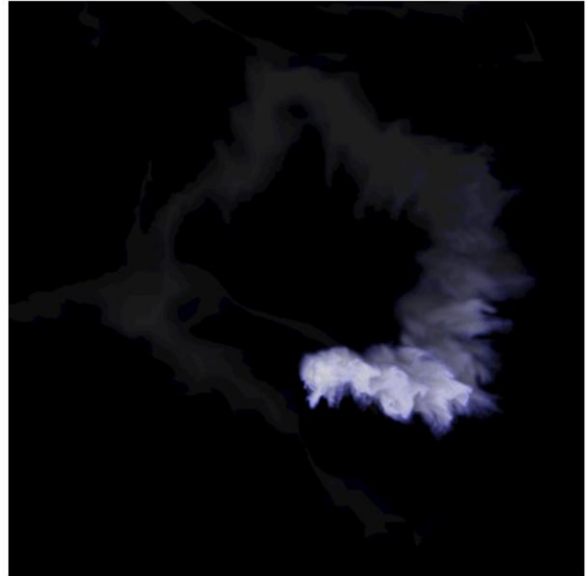


[illegible]

328

# Pseudo Fluid

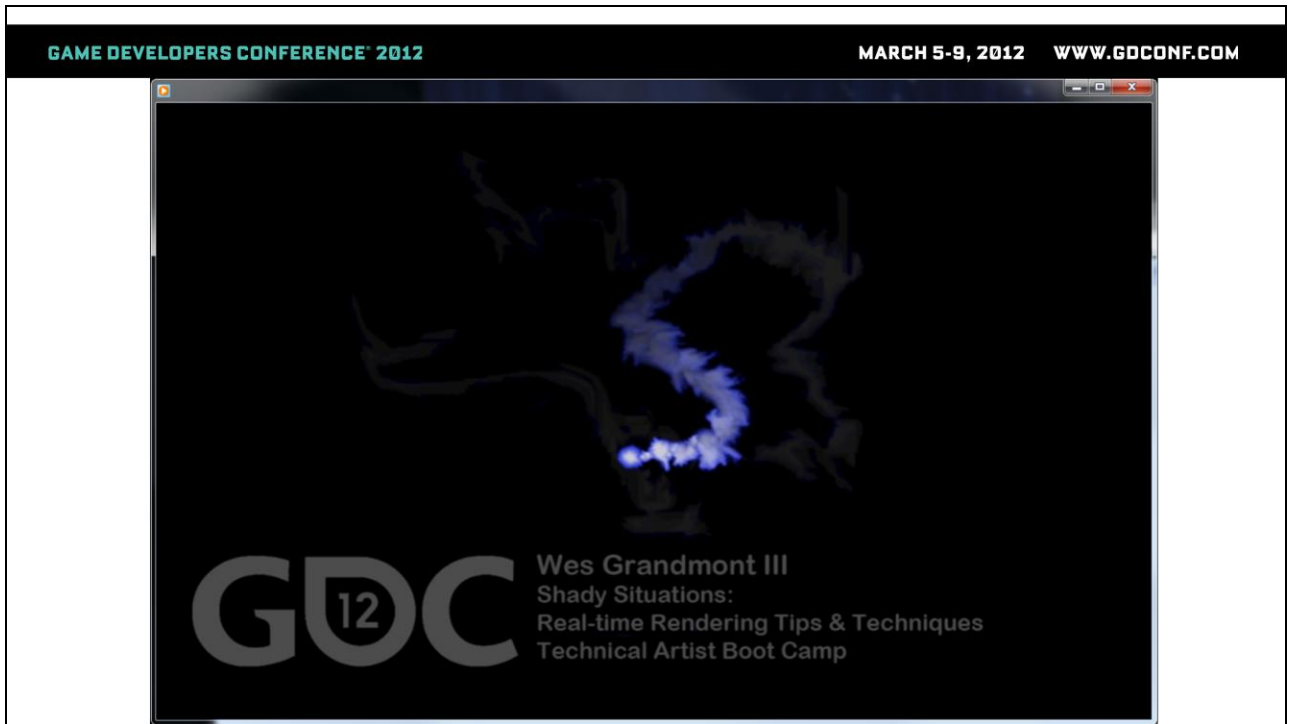
- Use of render targets
- Post-process materials
- Offset mapping



This is a psuedo-liquid technique. I call it a psuedo-fluid because it behaves similar to a more expensive fluid sim, but cheats much of the math. It demonstrates some key concepts including:

- Use of render targets
- Post-process materials
- Offset mapping

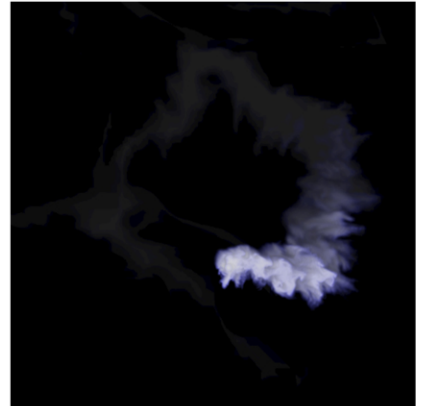
Because Maya's cgfxShader doesn't support rendertargets this demo was setup exclusively in UDK in order to illustrate the concepts.



Demo Movie

## Pseudo Fluid: Ingredients

- Off-screen sprite-based particle system
- PSysRT: Particle System Render Target
- Scene Capture Actor: writes to PSysRT
- AccRT: Accumulation Render Target
- Scene Capture Actor with Custom Post Process Material
- Output Material applied to mesh in view

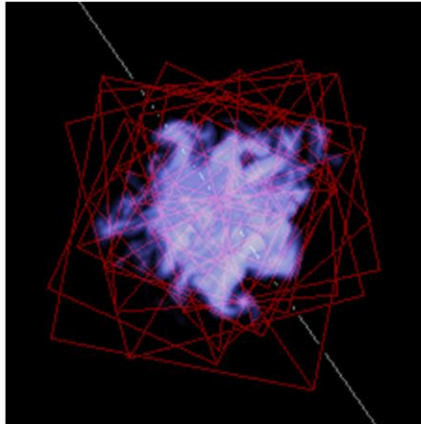


The setup for this pseudo-fluid consists of 6 main ingredients:

- 1.) Off-screen sprite-based particle system
- 2.) PSysRT: Particle System Render Target (1024x1024)
- 3.) Scene Capture Actor that writes to the PSysRT
- 4.) AccRT: Accumulation Render Target (512x512 with Linear Gamma)
- 5.) Scene Capture Actor with Custom Post Process Material
- 6.) Output Material Applies to Mesh in camera view

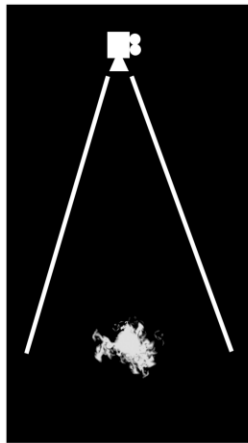
## Pseudo Fluid: Particle system

- Off-screen

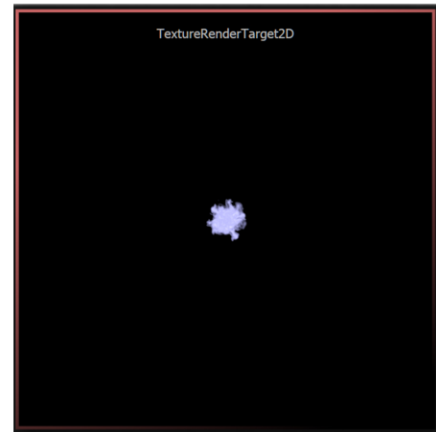
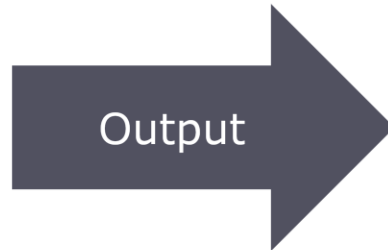


Off-screen sprite-based particle system. This defines the initial behavior of the fluid. The shape of the sprite, size, rotation, rate all effect the final look. This particle system exists out in space away from our main scene so that we can capture it in isolation.

## Pseudo Fluid: Scene Capture Actor



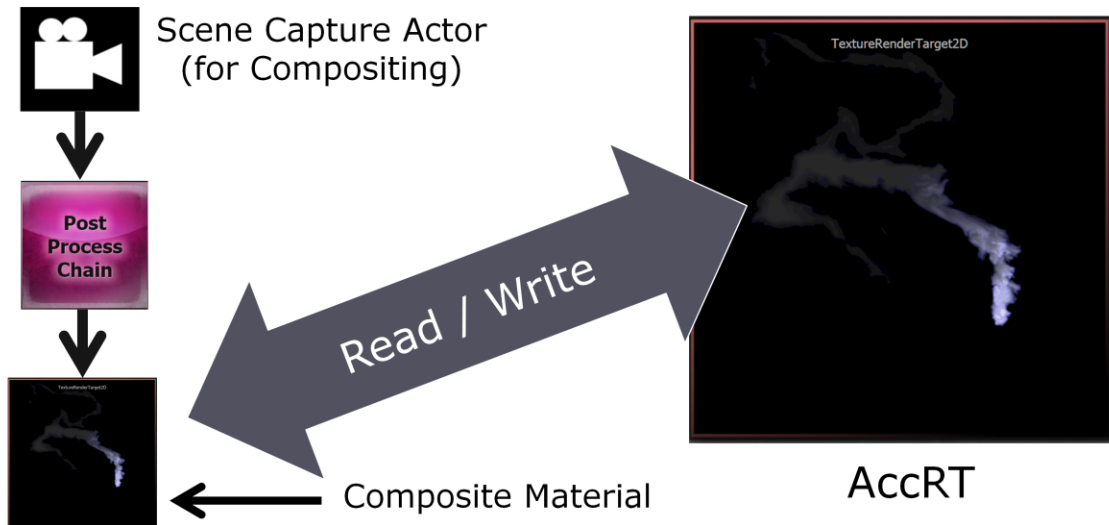
Scene Capture  
Actor



PSysRT

Next we create a 1024x1024 render target. We'll refer to this as the PSysRT (Particle System Render Target). Next we add a Scene Capture Actor to the level that focuses on the particle system. The Scene Capture Actor will take a snapshot of the particle system each frame and write the result into the PSysRT.

## Pseudo Fluid: Scene Capture Actor



We'll create a 512x512 render target with Linear Gamma that we'll call AccRT (Accumulation Render Target). And we'll add a second Scene Capture Actor.

This second Scene Capture Actor is just there to act as a compositor. We'll use its ability to specify a custom post-process chain to allow us to do some texture processing in a special compositing material.

The compositing material will combine the AccRT and the PSysRT and then write the result back out to the AccRT.

## Pseudo Fluid: Composite Material

- **Convection = Advection + Diffusion**
  - **Advection**
    - Shift UV's using an animated 2D Vector field
  - **Diffusion**
    - Read from Higher resolution rendertarget
    - Write to lower resolution rendertarget
    - Bilinear filtering handles the diffusion
- **Dissipation**
  - Accumulation buffer is attenuated each frame

Most of the magic for this effect happens in the post-process compositing material so lets drill down a little deeper there. In this material there are a few different things happening to simulate various fluid properties.

Fluid Convection is the combination of both advection and diffusion.

Advection is the transport of a scalar quantity in a vector field. Basically, how stuff gets pushed around as it moves through a medium. We'll approximate this by combining several texture samples to create a non-periodic animated 2D vector field (just a fancy way of saying we'll make some red and green channels that will act as uv offsets). We'll combine these offsets when reading the AccRT texture so that every frame the AccRT will continually evolve and change.

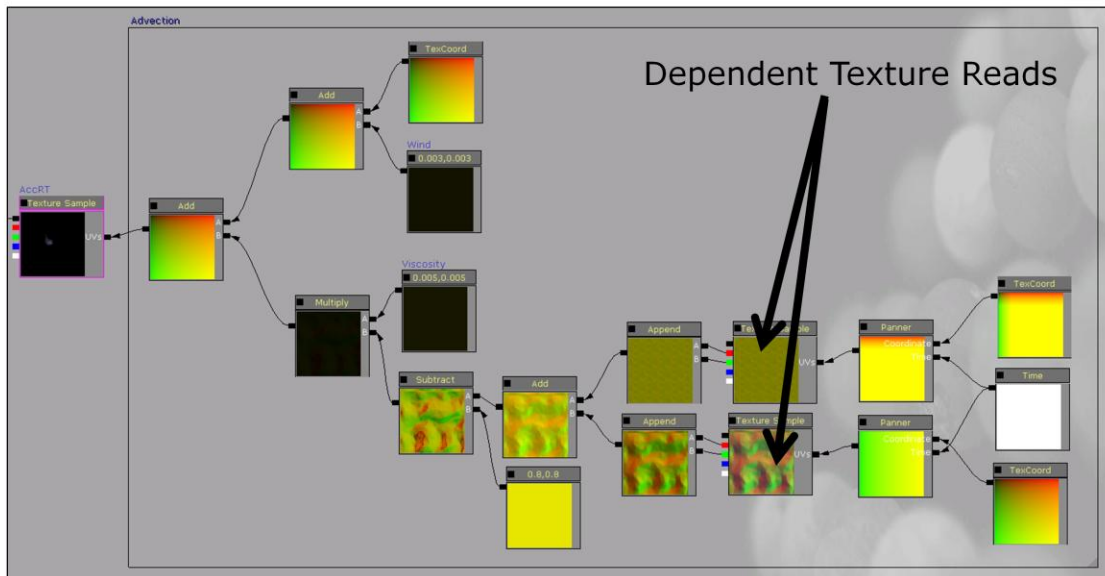
For the diffusion portion of our convection solver we'll read from our higher resolution PSysRt and write to the lower resolution AccRT. Thanks to bilinear filtering we'll get a some



free diffusion as the pixels blur while being downsampled.

Finally we can optionally dissipate our fluid (you might want to keep it around in which case you can skip this step) by slightly darkening the AccRT each frame. This will cause the accumulated samples to fade out over time.

## Pseudo Fluid: Advection



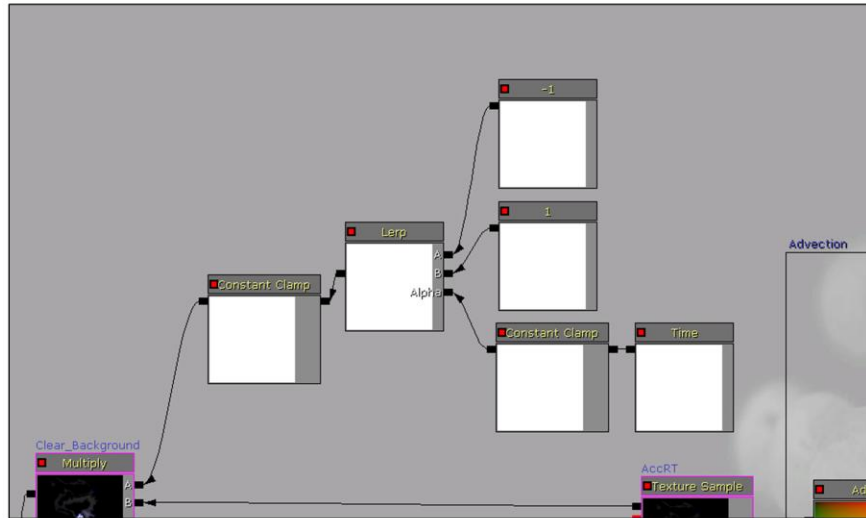
So let's take these concepts and show how they are implemented in the composting material. Here we see the advection implementation. We're sampling two textures. Both textures are tiled and panning at different rates and directions. These are combined in order to create a final float2 value that can be added to the base texture coordinates in order to create a final offset UV that we'll use to sample the AccRT.

I've noted the dependent texture reads here because these are usually something to be avoided. You can modify this portion of the algorithm to suit your performance needs. The important thing is that you end up with compelling set of offset values since these will make or break the perception of the final effect having fluid-like movement.

Another note about authoring the vector field textures is that the values in the R and G channels range from 0.0 to 1.0. If you want vectors that face in all directions you'll need to subtract 0.5 from the values so that they range from -0.5 to

0.5. It's not necessary to normalize the value because we multiply the final combined texture samples by a viscosity multiplier. The viscosity value is very small because we only want really small offset from the vector field each frame so that the fluid doesn't move too fast.

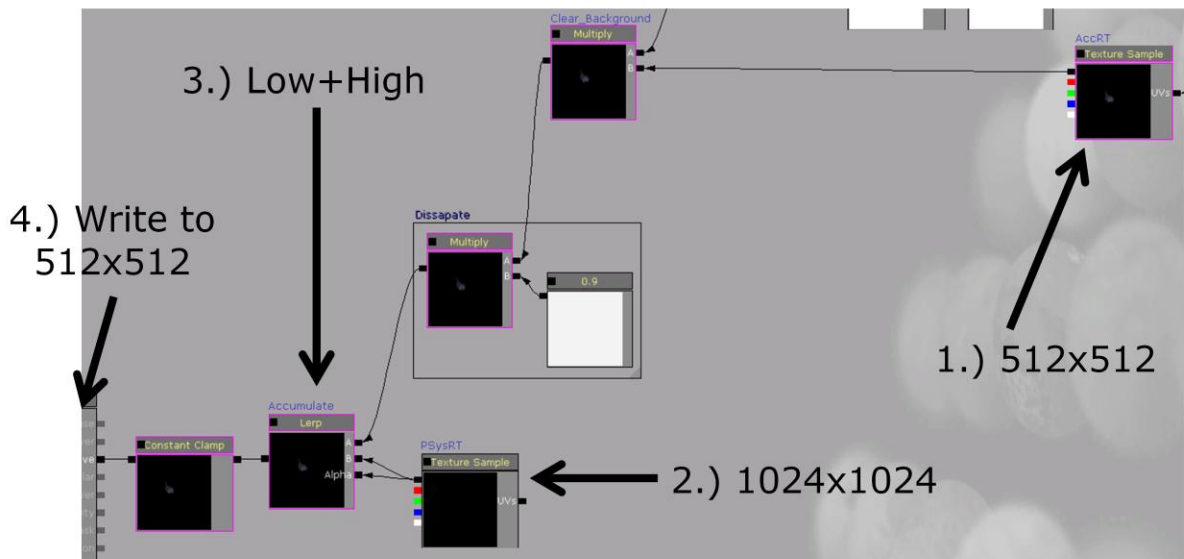
## Pseudo Fluid: Clear Render Target



HACK WORK-AROUND WARNING: Unreal initializes it's render targets with a green background. We want a black background for the render target. We can do this in the composite material by multiplying by 0 for the first few frames of time and then afterword just multiplying by 1.0. This can be done by driving a lerp with a clamped TIME attribute so that the render target gets cleared out when time is 0.0 and then quickly lerps to 1.0.

(It would be better and cleaner to change the initialization color for the rendertargets, but I'm not aware of a way to access this value in UDK).

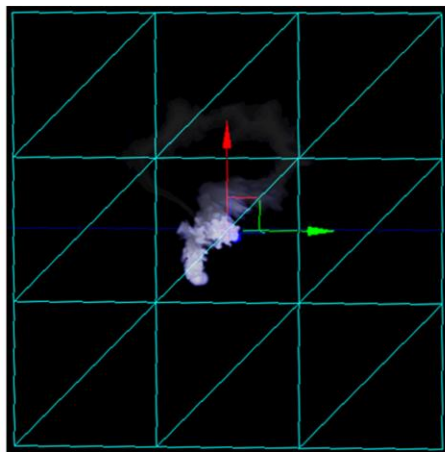
## Pseudo Fluid: Diffusion and Dissipation



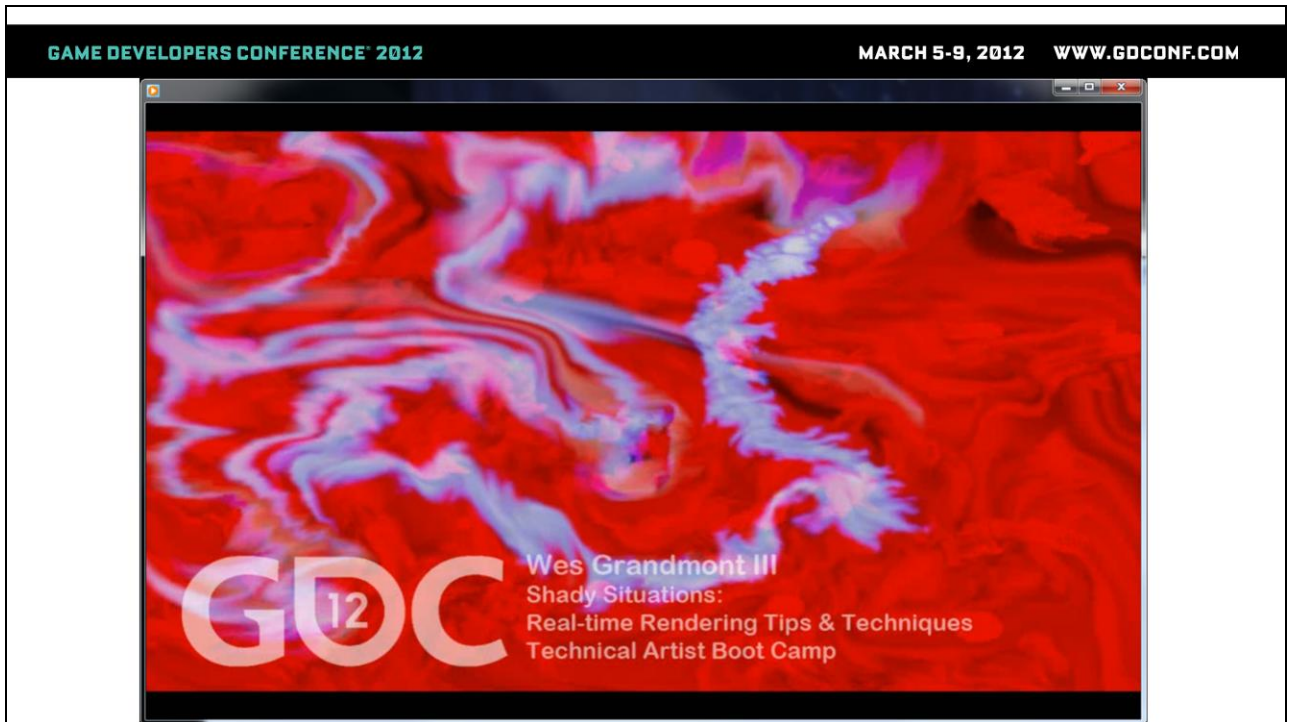
Here's the last part of our compositor, where the "compositing" actually happens. We take our advected AccRT sample, darken it slightly to create the dissipation effect, then use a lerp to alpha blend the higher resolution non-advected PSysRT on top.

The final result gets written back to the AccRT. This feedback loop is possible because we are using this material in a post-process chain that is being referenced by a scene capture actor, that is writing it's results to the AccRT.

## Pseudo Fluid: Mesh with Fluid Material



The final part of this setup is pretty straight forward. We take the AccRT and use it in a material that we apply to a surface in our camera view and we're done!



## Demo Movie

Some cool highlights of this technique are:

- You can mix different colored fluids together
- You can animate the emitter position
- You can combine the animated vector field with a static vector field to have fluids flow around obstacles

# Suggested Reading

- <http://developer.nvidia.com> (**Resources**)
  - **Nvidia's The Cg Tutorial**, by Randima Fernando and Mark J. Kilgard, published by Addison Wesley
  - **Nvidia's GPU Gems Series**, published by Addison Wesley
    - Also available on Nvidia's website
  - **Cg User Manual** (part of Nvidia Cg Toolkit installation)
- **ShaderX Series**, from Thomson Learning, Inc. published by Charles River Media

Before we wrap things up, here are some resources that I've found helpful over the years.



## In Closing...

- Don't forget Performance!
- Being Helpful

Questions?

<http://www.linkedin.com/in/wesgrandmont>

As I mentioned at the beginning of this talk. All of these techniques contain building blocks that can be used as-is or adapted and combined to solve various visual problems. I haven't really talked too much about performance, but it's an important thing to consider when deciding whether or not to use any technique.

We want our games to run at least 30 fps, so we have to make sure everything we need to render each frame can finish in under 33ms. Profiling tools like PIX for the Xbox 360 are essential in this process. In the end, game development is a delicate balancing act so it's important as technical artists that we measure and evaluate the impact of any solution in context with everything else that needs to be rendered before we decide whether it's a good choice for our game.

In closing, I wanted to tie back into the rest of the day's talks and say a brief word about technical art. For me, being a technical artist is all about enabling the team to do their best work. Whether it's tools, pipelines, rigs, shaders, optimization

or just plain firefighting our job is to identify the things that slow the team and the project down and make them better. I've found that if you approach everything with the spirit of a sincere desire to help you can't go wrong.

Thanks for listening and I hope you found something useful to incorporate into your own projects!



Hi I'm Will Smith. I'm a technical artist at Volition, where I've worked on a variety of titles over the past four years. Technical Artists at Volition are usually generalists, so I've had the unique opportunity to get to work on nearly all of Volition's more recent titles, including Saints Row 2, Red Faction, and of course Saints Row the Third. I had the opportunity to work on SR3 for the last 8 or so months of production, and we shipped last year with a huge effort from the entire team both at Volition and THQ. So it's a real pleasure to be here today to present, and I hope to provide an informative lecture to finish off the boot camp – especially after hearing my colleagues present who are in every way extremely tough acts to follow! So without further ado: Unusual UVs – Illuminating Night Windows in Saints Row: The Third.

But first – Tech Art:



I tried to draw a diagram of what the responsibilities of a technical artist at Volition are, but I think this picture sums us up a little better. As we've heard from the preceding presentations today, we know tech art covers a wide range of skill sets and capabilities, all geared towards an interdisciplinary approach to problem solving. Since technical artists help resolve such a wide variety of problems, it becomes difficult to traditionally define what a technical artist actually *does*. Some Volition TAs prefer working on environment art, others cinematics, others shaders or tools programming. Thankfully I don't want to attempt to define what technical art is for this presentation, merely to confirm that its breadth is wide and varied.

What we do know is that tech artists help resolve a variety of problems.

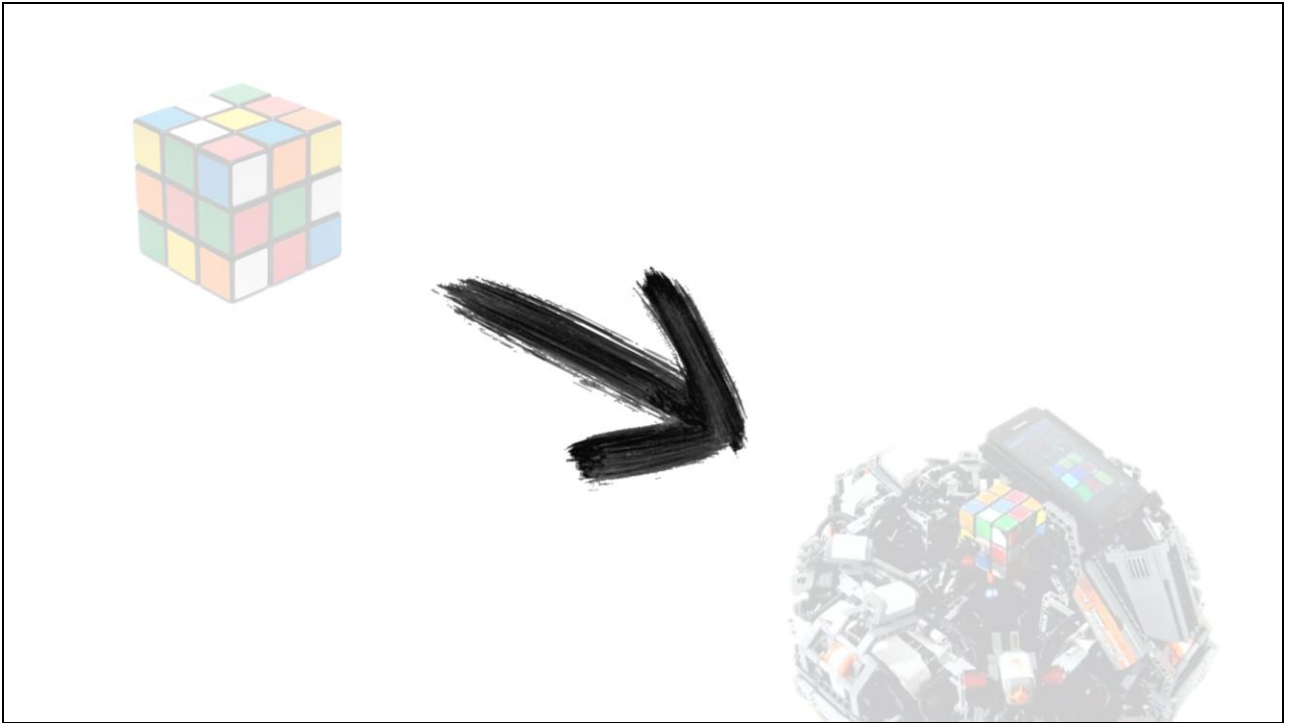


We've seen a range of clever solutions to many common development issues presented today. Still, there's no way to predict the intricacies and caveats of future game development problem cases. It's always going to be difficult to formulate an approach to a new and *unique* problem. It's especially difficult to imagine how a tech artist might go about solving a particular problem without any idea of the eventual solution in mind.

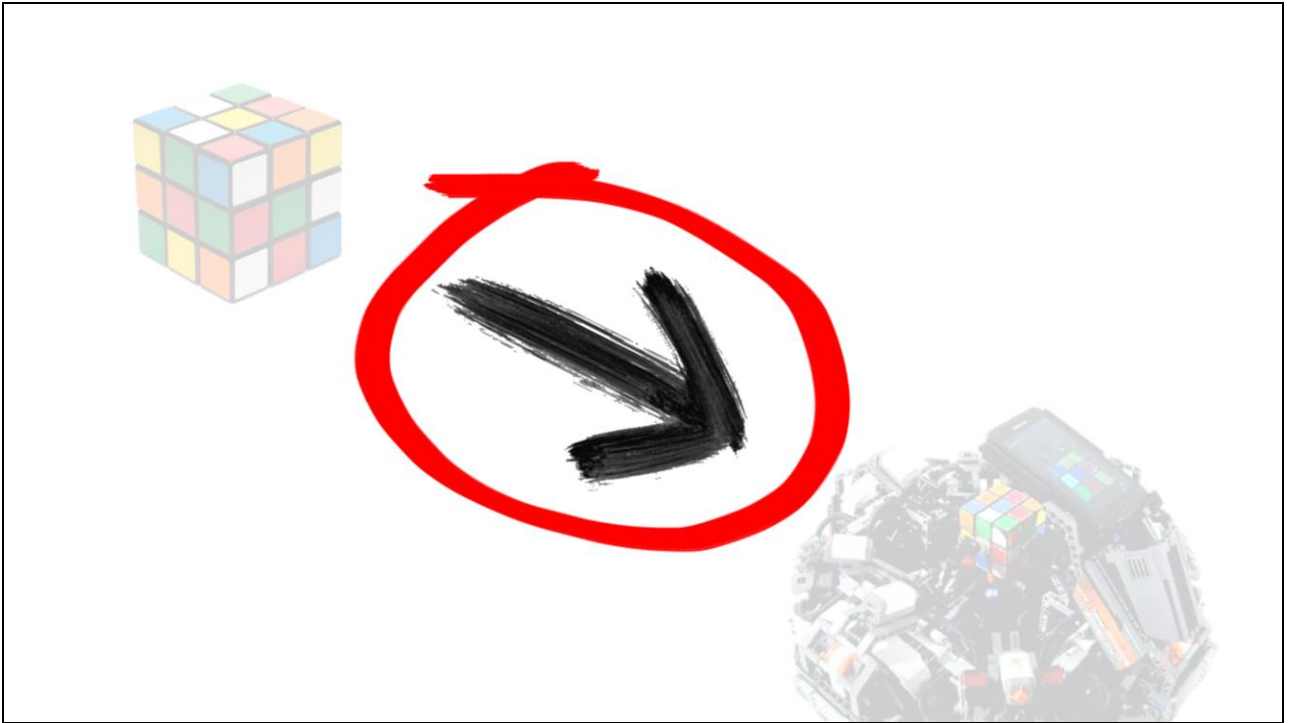
Often a good solution isn't obvious.



Often what's involved in defining the problem can be tricky too. The end product may be the result of intense collaboration, or a single-handed fix. It may be a quick update, or require investigation into entirely new features, tools or pipelines – all of which may force tech art to interleave with associated development disciplines.



So it's the *process* I'm interested in conveying.



Instead of providing a finished product or finished solution, this presentation is going to be a generic overview of a process – the steps *I* took (for better or worse) to resolve a fairly simple example of a real-life game development problem. I don't mean this as a caveat - my hope is that by conveying my thought process and experimentation, I can show a solution within the context of the steps taken to arrive at it – which I hope ends up being even more useful than the solution by itself.

Maybe Tech Art is a little like rock climbing:





...there are infinite routes, and it's not obvious what's best. I've never actually been rock climbing. So I hope that analogy makes sense.

Anyway I guarantee that you'll never find presentation slides that *exactly* solve the problem you've got in front of you. You're going to have to make up some of the rules as you go. You're going to have to fly a little bit by the seat of your pants, no matter how much you plan. And when you've got something working, it's a cruel reality that you'll dream up ten better, more efficient, or more robust ways to do it over again.

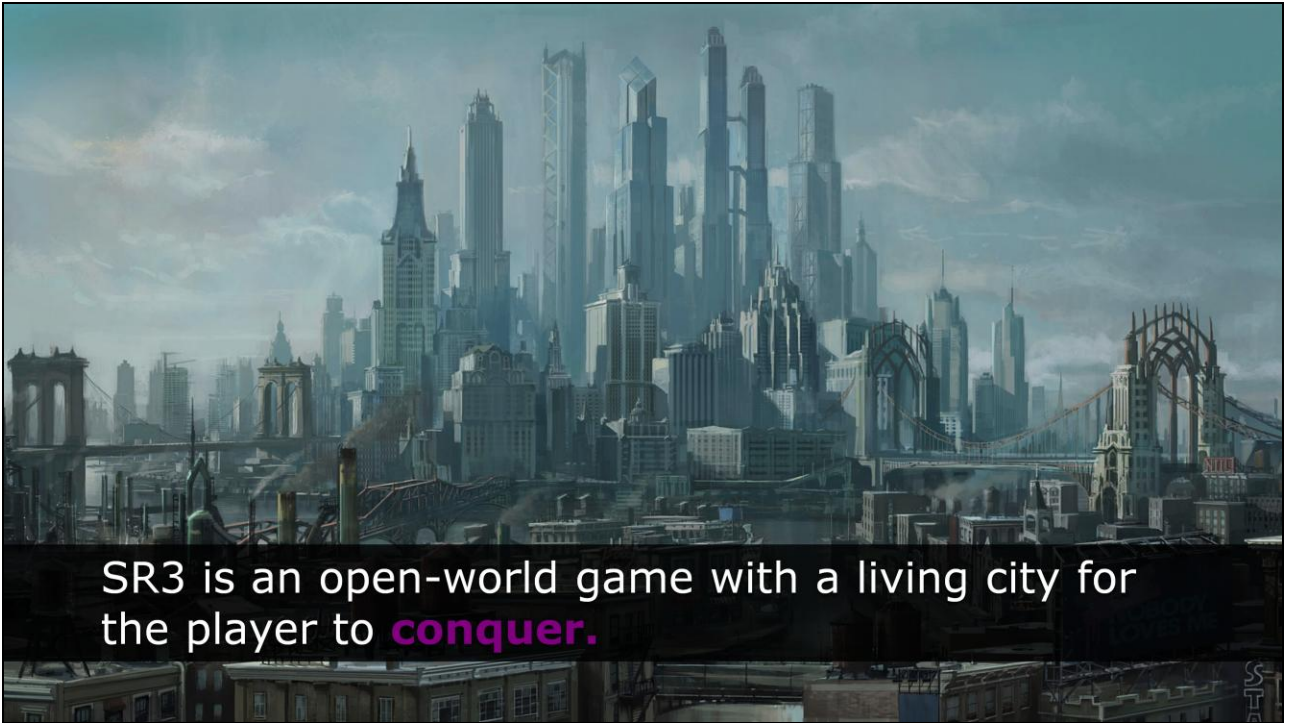


But that's also kinda the point. A Technical Artist's job is essentially to learn, and with more experience comes a greater capability to apply learned lessons to current problems.



Welcome to Steelport, the seedy criminal centerpiece of Saints Row The Third. The brand-new city of Steelport is the sole setting and haven for all the nefarious over-the-top activities that occur in Saints Row The Third. It's like Detroit, if you smeared eight more Detroits on top, then perched it all on a lonely island, then made every inhabitant criminally insane.





SR3 is an open-world game with a living city for the player to **conquer**.

Steelport is an open-world sandbox for the player, a living city that exists solely for the player to explore and conquer.



The city needs to intrinsically convey that importance, it needs to be immediately relevant and central to the player's interest – and it needs to look worthy of that interest; therefore it needs to look populated, alive, and most importantly: it needs to look *worth conquering*.

The trouble is, we're two thirds through production at this point, and at night Steelport looks like this:



It's not a living mega-city yet. Much of the city is still in placeholder form at this point, and the night window system is no exception. This is the reality of commercial game development, and it's often difficult to assess priorities for work on game systems without doing enough investigative work to effectively build or create the systems in question themselves. This is another area of Tech Art specialty.

It may look like the buildings in this image haven't been set up for any kind of night window system at all, but that's not at all the case. While the city appears at first glance to feature stark repeating textures, there are a number of reassuring components at work that we can use to make some preliminary assumptions. It looks like most of our buildings feature explicit UVs, and whatever placeholder night window system is in place is global enough to be apparently affecting the majority of the city at once, both of which are excellent criteria to confirm when we get to the stage of assessing the problem in earnest.

It's a rare case for any discipline, Tech Art included, to build a system of any kind from the ground up. This is a classic example of a partial solution that's been waiting for some Tech Art love and attention – which should probably happen sooner rather than later, in case some important potentially workflow-altering discovery about this system might affect the team as a whole and risk shipping quality.





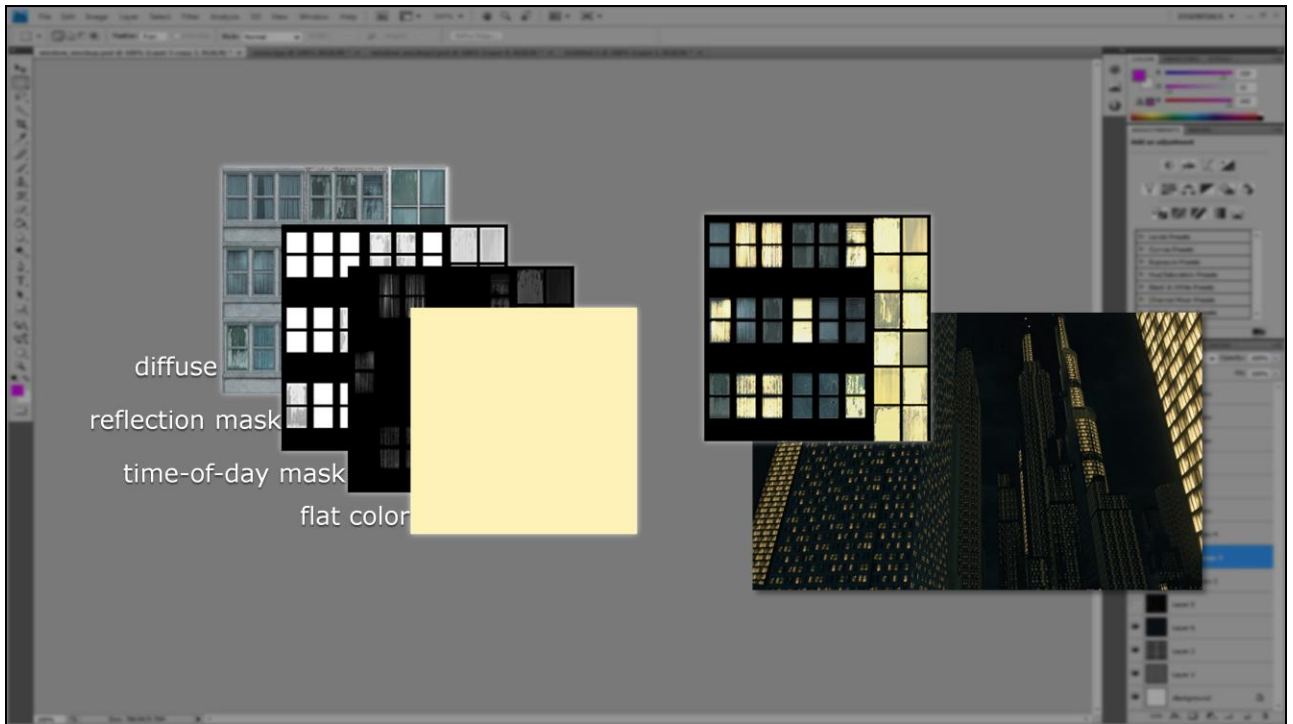
This pre-emptive step is also where Tech Art can make a huge difference. And by that I mean I volunteered for the job. This isn't to say that a "make the windows not horrible" task wasn't somewhere in the production backlog, but I knew I personally had a little time to take a shot at the problem. Sometimes it's scary to take the initiative and crack open an unfamiliar system, especially when schedules are already tight (and they always are), but that's exactly what Tech Art is for. So I said if nobody was already on this – I'd be happy to tackle it. The last thing I wanted was to leave the windows in the state they were up to the last hours of production.

I checked up with my art director and leads, and got all the night-window-related tasks crammed onto my to-do list for the next couple days.

Before I planned or arranged anything, before I even planned to plan, I wanted to understand the status quo beyond the cursory visual evaluation of the existing city. I needed to break apart the placeholder shader and at least understand



how the building window glow effect was currently working.



Here was some great news. The existing shader was arranged like so – a set of masks: A reflection mask called out the areas of the diffuse texture that were specifically meant to be reflective, and a separate time-of-day mask indicated the areas of the map that would glow at night using a night-time color value. I've arranged the component pieces here so you can see what I mean.

This greatly simplified my job. It meant that time of day information was already accessible in the form of an explicit mask applied to each diffuse map. The existing system wasn't optimal, of course, since the time of day mask was obviously tiling in correspondence with the associated diffuse map, and therefore resulted in the repetitive tiling effect we were seeing in-game. But to me it was a huge relief – since there was a precedent for the time of day information to exist separately from the diffuse map, I could imagine a system to affect the time of day mask alone, maybe somehow breaking up that tiling pattern without affecting the diffuse information at all. Maybe that was a good place to start.

I might have gone in, tech art guns blazing at that point – but I understand the risk of underestimating the complexity of even a simple task.

I was getting too technical too fast. It's very tempting just to start fiddling with shaders. I needed to step back again, and re-assess what I was doing from a more informed perspective. I needed to bring a little Art to bear on my Tech. That brought me back to our

SR3 production mantra: "exaggerealism".

## "Exaggerealism"



SR3 is 'exaggereal', in a variety of ways. We intentionally pursued larger-than-life looks and attitudes in our character concepts, and it seemed sensible to me that our city should reflect the same kind of mood or expression. I interpreted the buildings in Steelport as included in the philosophy of exaggerealism, and almost as characters in and of themselves.

Characters in Saints Row 3 are exaggerated; nearly cartoons. I was aware that by building a solution for the night window lighting problem I'd be affecting the 'look' of the city significantly, and I wanted to make sure to adhere to the philosophical and artistic intent of the environment. In this way, Steelport is really a giant, exaggereal character.

## "Exaggerealism"



Steelport is really a giant, exaggereal character

I didn't want to do anything to dampen that idea.



The character of SR3's buildings vary from the mundane to the ridiculous. From tiny trailers to warehouses to what we called "super-scrappers". At this point in production, Steelport's geometry was already larger-than life.

Several of our buildings topped out at more than 200 stories, comfortably as tall as the Burj Khalifa in Dubai. Some were easily taller than the city was wide.

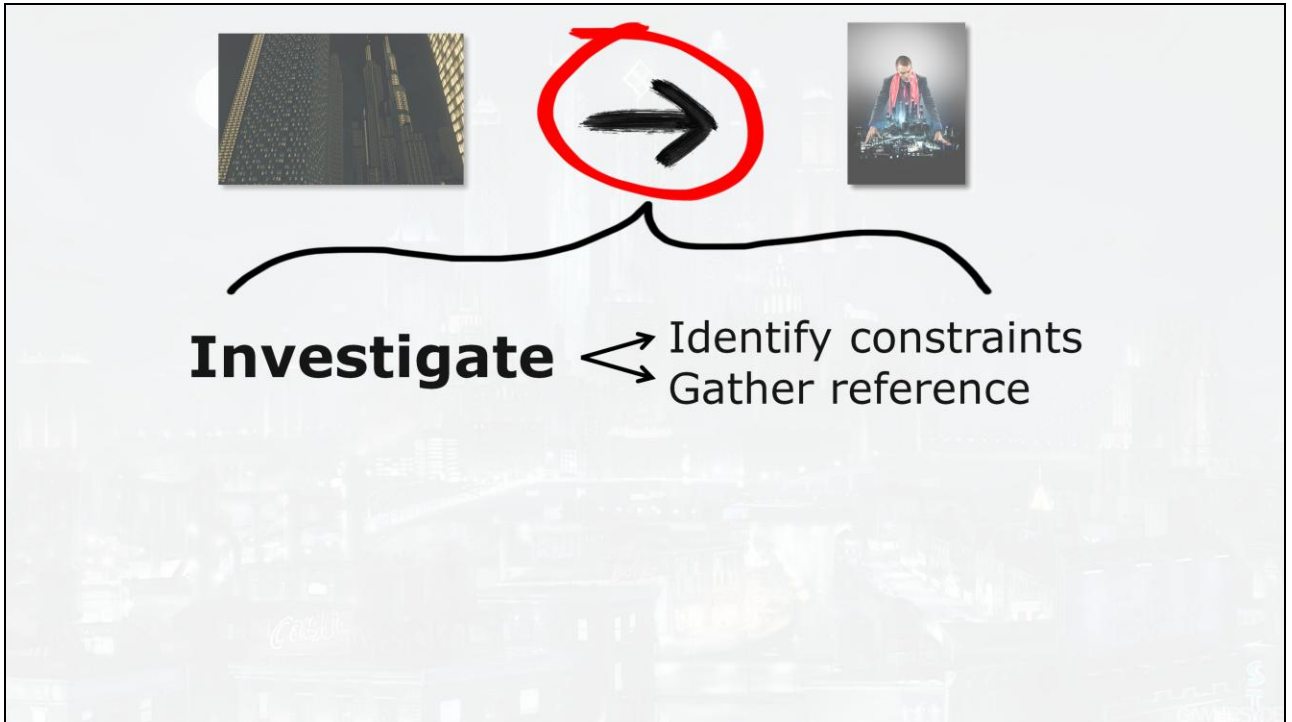
It's not explicitly stated in-game, but each gang owns a distinct tower in the city. While the evil Syndicate gang holds the iconic red tower, both the Deckers and Luchadores have a super-scraper stronghold of their own to dominate the skyline. The Luchadore's green octagonal extruded wrestling-ring evokes the squat stature and volume of the wrestlers, while the Decker's neon blue obelisk has a more technical, corporate feel. Each are obscenely massive, with a frankly stupid number of floors and windows each.

**Goal:** Adapt 'Exaggereal' look to city night windows



With the exaggereal character of the buildings in mind, I have some idea of a goal. It's not just a matter of making the city windows NOT look like corn cobs. The process I have to examine is more accurately defined as adapting the exaggereal look and feel of Saints Row to the buildings via the building night window lights. This may seem like a trivial distinction – but it's a good practice to get into. Defining the goal of a task in context of the artistic requirements of the project is always a good idea. Plus you'll make great friends with your art director and design team. If nothing else, it means that the 'how' and the 'why' of your work become more closely intertwined.





With a wider perspective of my goal, I can focus on the process of achieving it. But in order to plan an attack to accomplish that goal, I still need more information.

More information requires investigation. I knew I needed to identify the constraints I'll be working with. I needed to figure out how best to apply my time and resources to the problem, so I could focus my efforts most effectively. I needed reference material to guide me, and I needed to reconcile that reference material with the existing assets I had to work with. My guess was this would probably mean cracking open buildings in max and seeing how they're constructed in addition to prototyping custom shaders. But for now, that was just a hunch.

Only once I have more information can I even think about





**Investigate**  Identify constraints  
Gather reference

**Implement?**

implementation, let alone



**Investigate**  Identify constraints  
Gather reference

**Implement?**

**Iterate?**

iterating on any hypothetical implementation.



Everyone works under the constraints of time, resources and feasibility. The specific problem of the building night windows potentially affected every building in the game, all of which were authored to a variety of standards, artistic and technical parameters, and of course stylistic differences.

An ideal resolution would be a system that covered the most assets as possible with a unified solution. Since all window materials in the game (super-scrappers and residential houses alike) referenced a small set of placeholder window shaders, a shader-based initial approach seemed like it'd have the most benefit. I should be able modify the shader and a small number of materials and hopefully affect the look of every window in the game.

This would be far preferable to a solution that would require modification of every building asset. There are nearly hundreds of unique buildings in Steelport, and the more general and geometry-agnostic my solution the better.

But to visualize the potential 'look' of the city at night, I needed to gather reference material.



I primarily referenced our SR3 concept art, which had some great views of a stylized Steelport at night, from various perspectives.









Real-life reference helped ground the concepts in my mind,





and I even cursorily examined other games to see how they light their cities.

Even as a gamer I was surprised at the range of looks and moods that different night window systems conveyed that I hadn't really noticed before. The look of Deus Ex's desaturated yellowy dystopia, for example, worked to evoke a far different feeling than Steelport's towering super-scraper concepts – and that uniqueness was something I wanted to maintain.



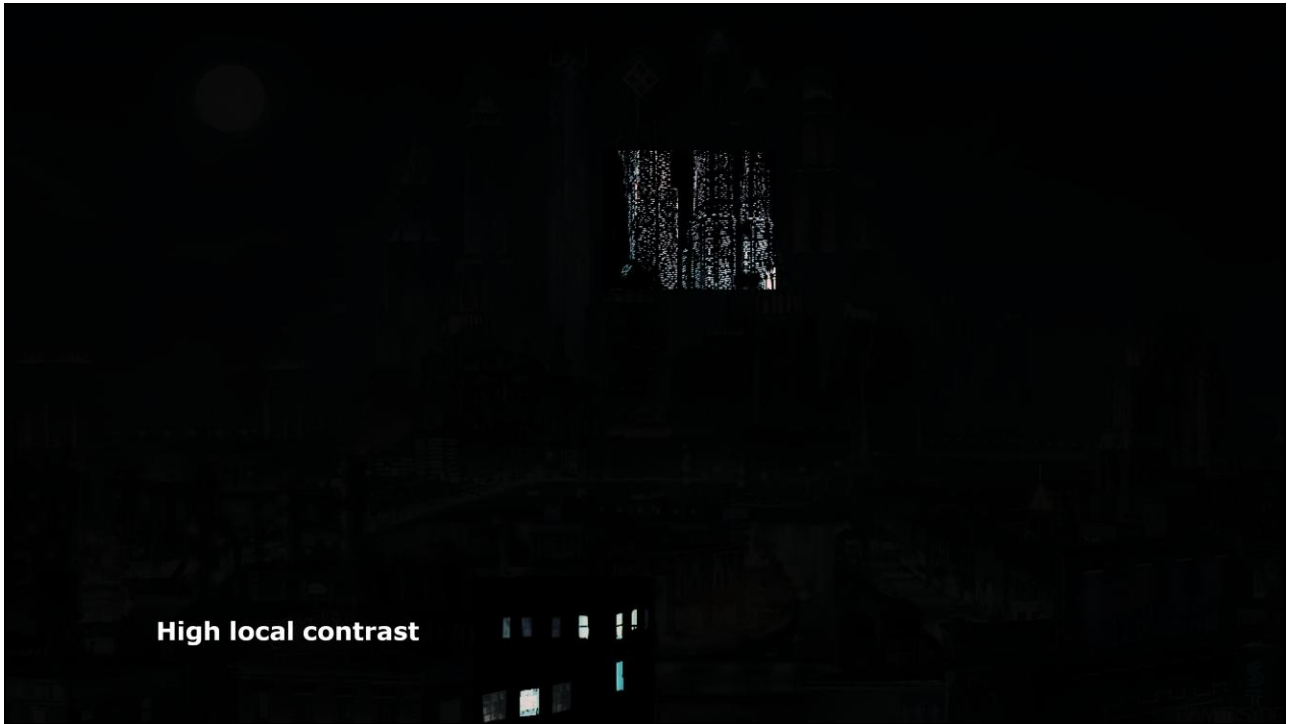
I set myself some criteria to distill the components of an 'exaggereal' look for the night windows:



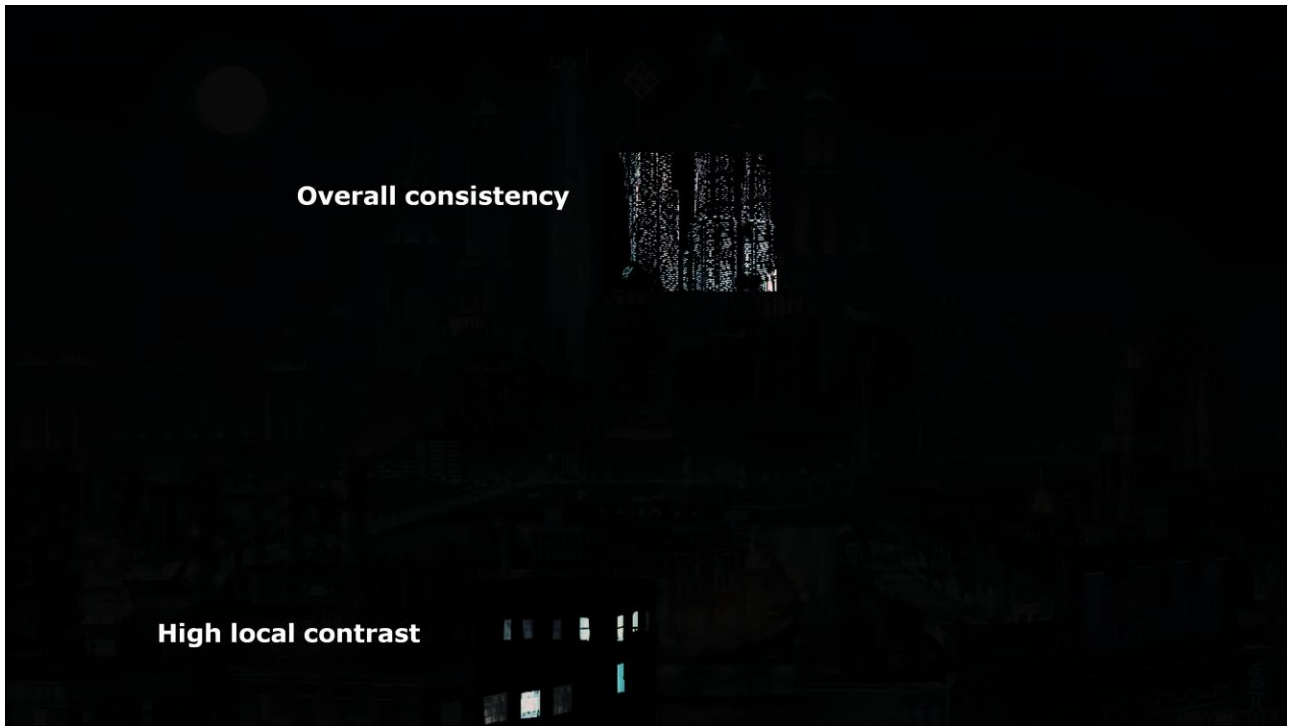
Steelport required (and already featured) lots of dense, brightly-lit windows.



Specifically the concepts called for high local contrast at near distances,



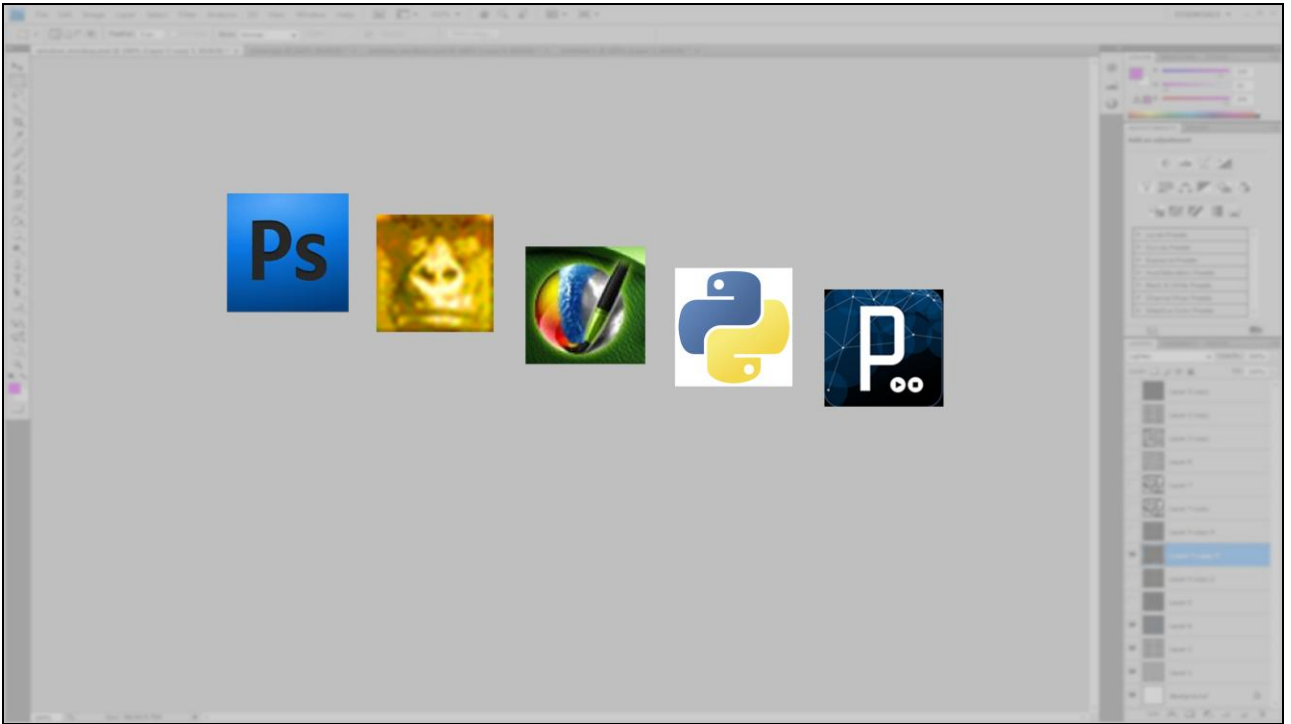
But overall consistency at far distance – still uniformly lit and occupied.



At this point I felt ready to begin an initial implementation. I needed to promote the exaggereal character of the city, preferably through a small set of updates to the window shaders and materials to be most efficient. I also needed to stay true to the concepts in terms of contrast and uniformity.

My working plan was to build a global mask map that would be applied procedurally via the window shaders. I'd tile the map enough to appear consistent at a distance, but it'd contain enough noise to contrast across individual windows when closely inspected.

My next step is to create a visual mock-up.



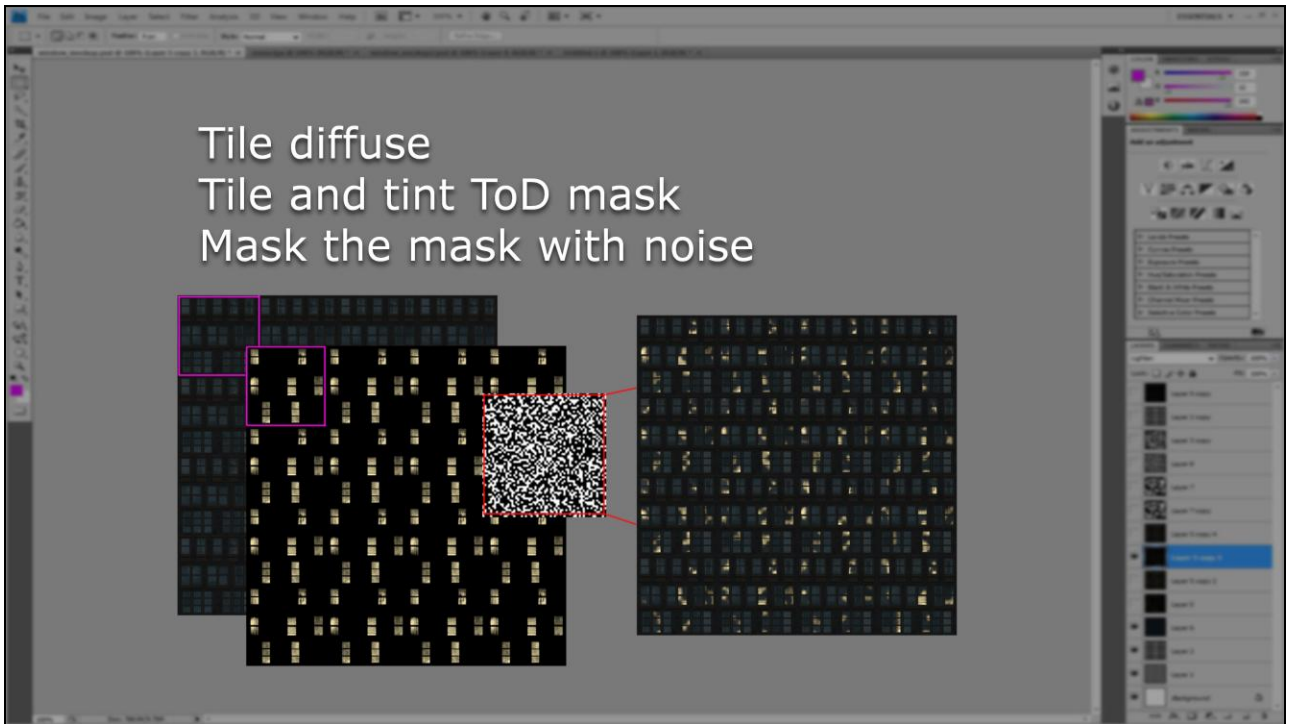
The idea here is to distill a kind of pseudo-algorithm for the effect I'm looking for. I don't want to necessarily write the shader math outright, but just get a basic idea. I'm hoping I can mask the time-of-day mask, for example. I'd like to see what kind of effect that might have – and speed of iteration is key.

Different prototyping tools have varying strengths. Personally I lean pretty heavily on photoshop to flesh out effects I'm looking for, then fall back to python and the python imaging modules for more complex results outside of Photoshop's toolset. This takes a little set-up time, and there are infinite alternate ways to get an idea for what kind of result you're looking for.

FX Composer, and even Rendermonkey might map more directly to an eventual end-result, since you can mock up results directly in HLSL if you want. I find processing is a little cumbersome for the job, but if you're a java whiz then it might work great for you.

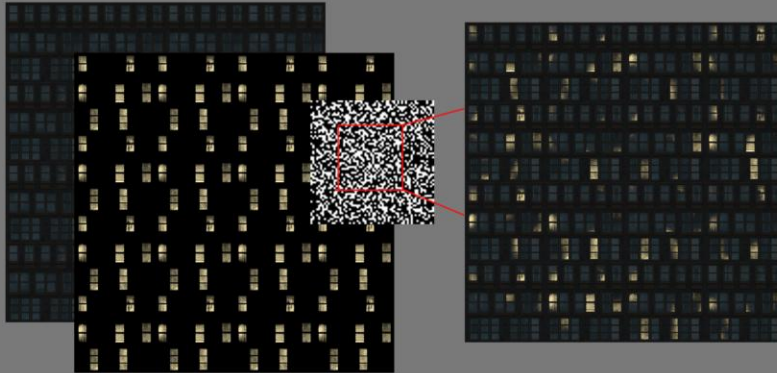
Suites like Houdini are also a possibility, but I haven't yet had an opportunity to play with that toolset myself.



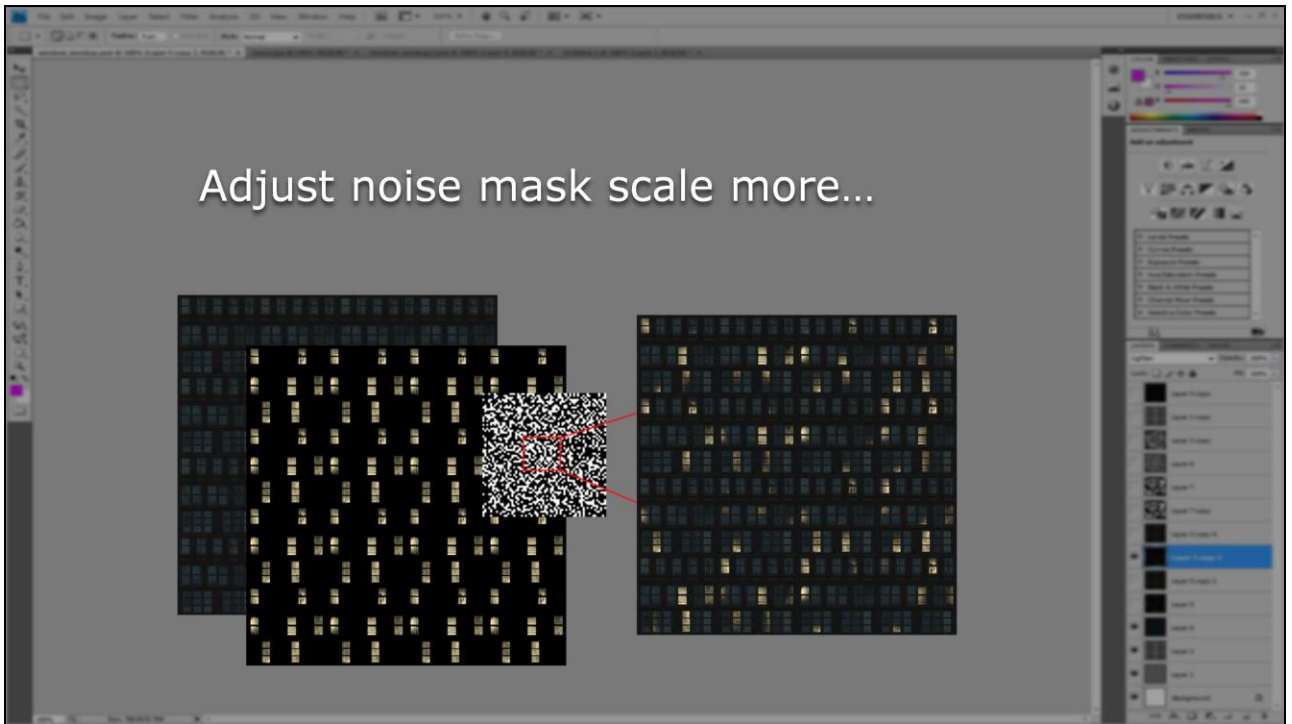


Starting out in photoshop my first step is to tile a portion of an existing diffuse map and align it with a tinted version of the existing time-of-day mask. I'm hoping to retain as much information from these masks as possible. I want to break up the monotony of the time of day mask, so I drive it through an additional noise map. This should break up the glow effect that is applied back onto the original diffuse. Unfortunately, the result is less than perfect. Splotchy yellow shapes don't read as illuminated windows, and I'm not sure this idea is going to work.

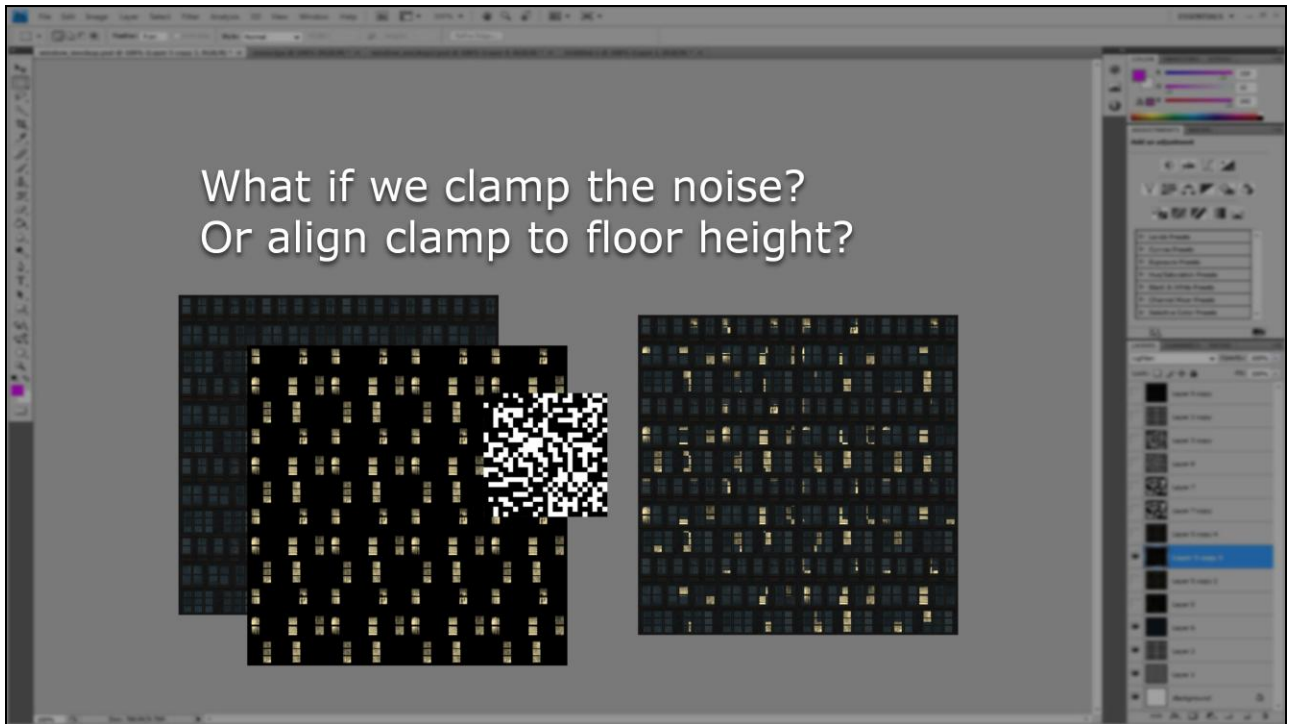
What if we adjust the noise mask scale



Adjusting the noise mask scale seems to help. More windows are hit fully by the noise map, and the monotony of the time-of-day mask is still not as noticeable. Some windows are still very oddly partially lit though.



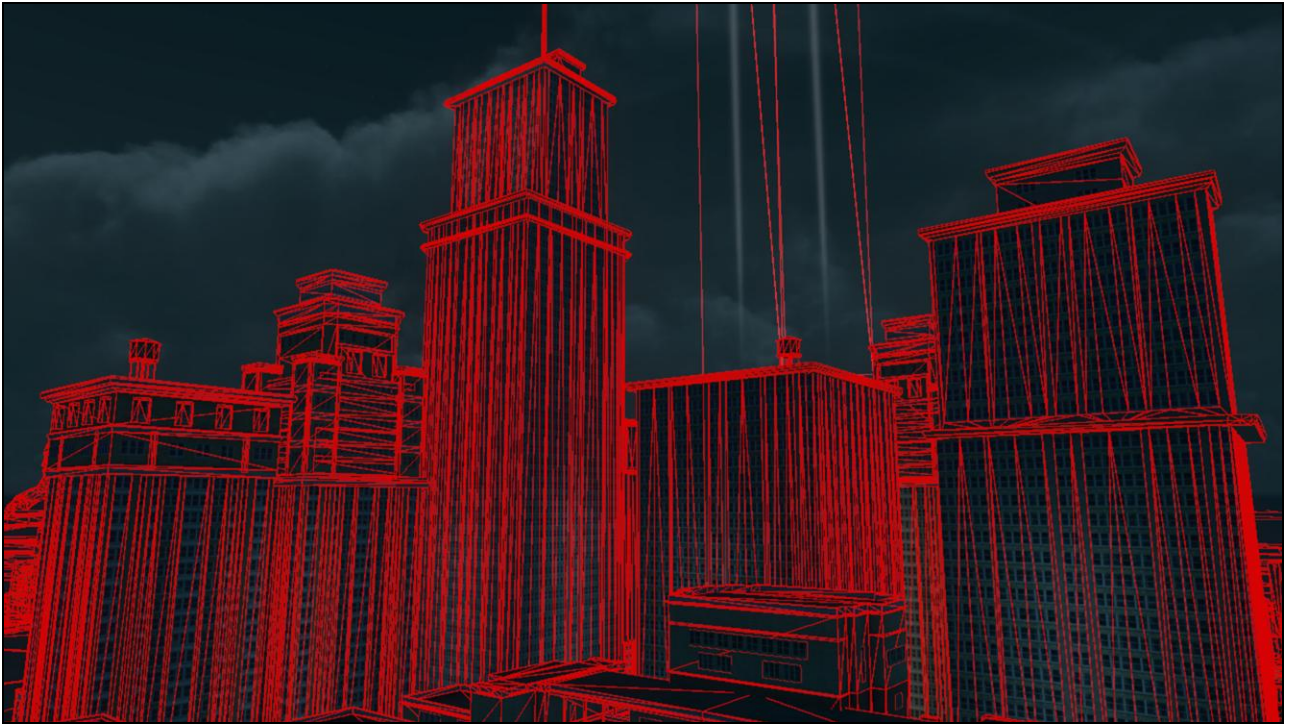
Adjusting the noise mask scale even more seems to help more. Windows are more likely to hit a full noise mask pixel, so they look well illuminated, but most windows are still only partially lit.



It'd be cool if I could clamp the noise map to align with the windows themselves, that way there wouldn't even be any partially lit windows. Unfortunately I can't just arbitrarily clamp the noise map, otherwise I'll get the visible pixilated effect you see here. To line up windows with the noise map I'd need to make some assumption about either the layout of existing building UVs or the overall world-space height of window heights in the world, and I'm not sure either of those are constant across our assets.



I notice something unusual as soon as I start looking into how many of the buildings in the world are constructed.

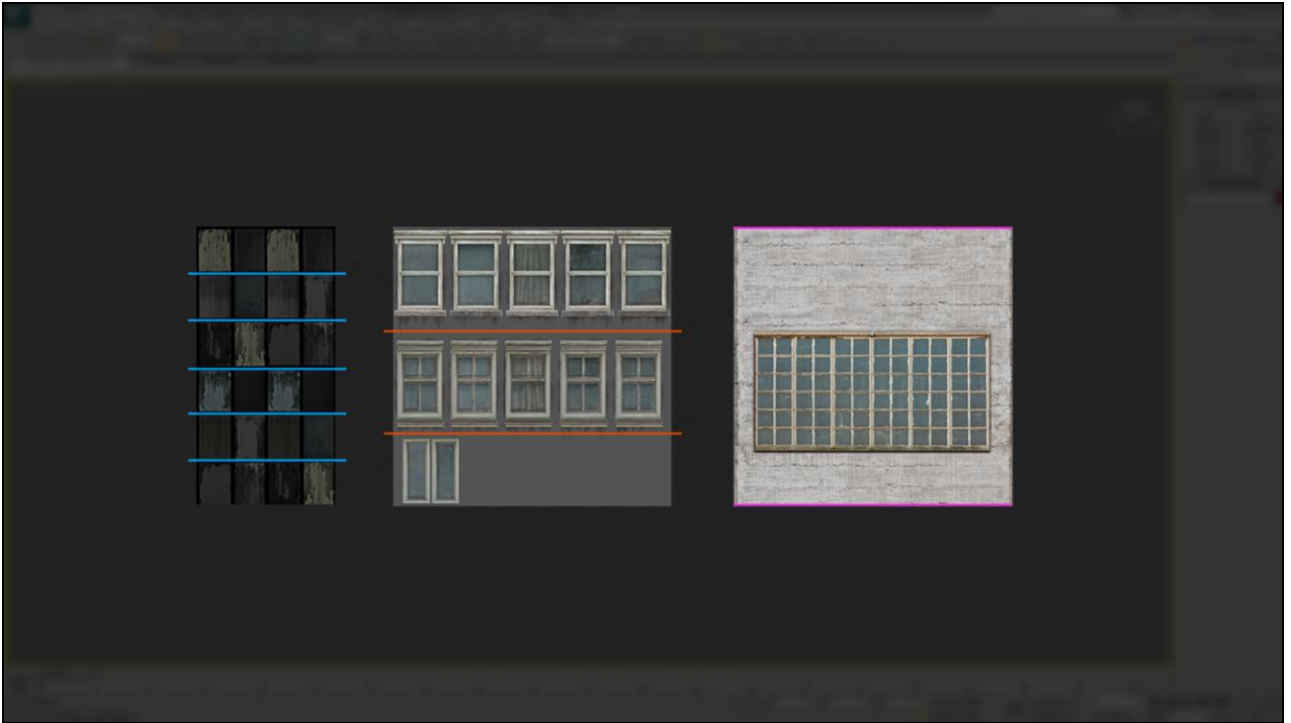


In particular, building wireframe geometry seems ...unusual for the silhouette shapes the buildings are ultimately conveying. Since I'm interested in potentially taking advantage of standardized UV mapping to align a glow map to the windows, I need to find out exactly why the buildings are constructed the way they are.



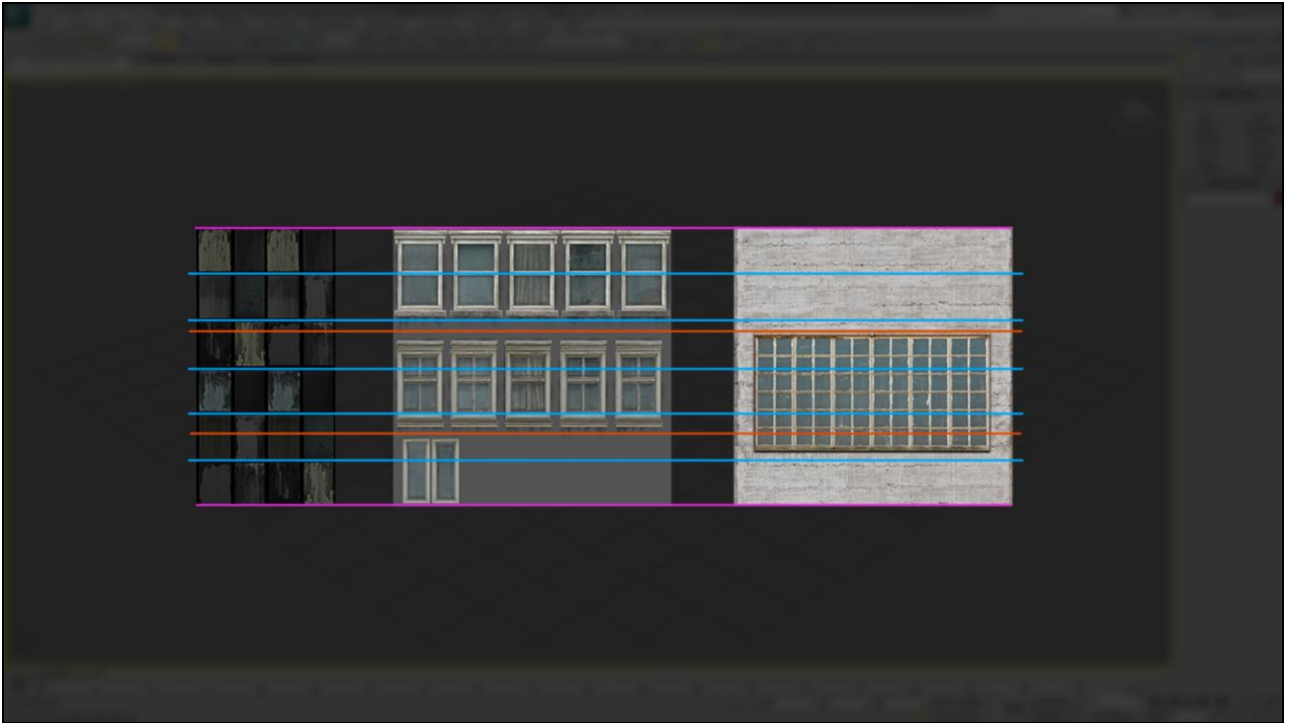


Here's an example window texture, one of a huge library of textures used on SR3. It's painted in modular strips, so different kinds of windows are contained on one map. Unfortunately, this means that in order to use one section of the map but not others, buildings require geometry cuts to support Uving that texture information appropriately. This makes it difficult for me to align a potential window mask horizontally in world-space, not to mention raises efficiency issues for general world asset construction I didn't know about.



Here are three more random SR3 window textures. I've highlighted the vertical breaks where windows are defined on these textures. In keeping with general SR3 building construction, they are typically explicitly UV'd by artists onto varying building geometry to use some, all or a portion of their texture, depending on what the artist and asset require.

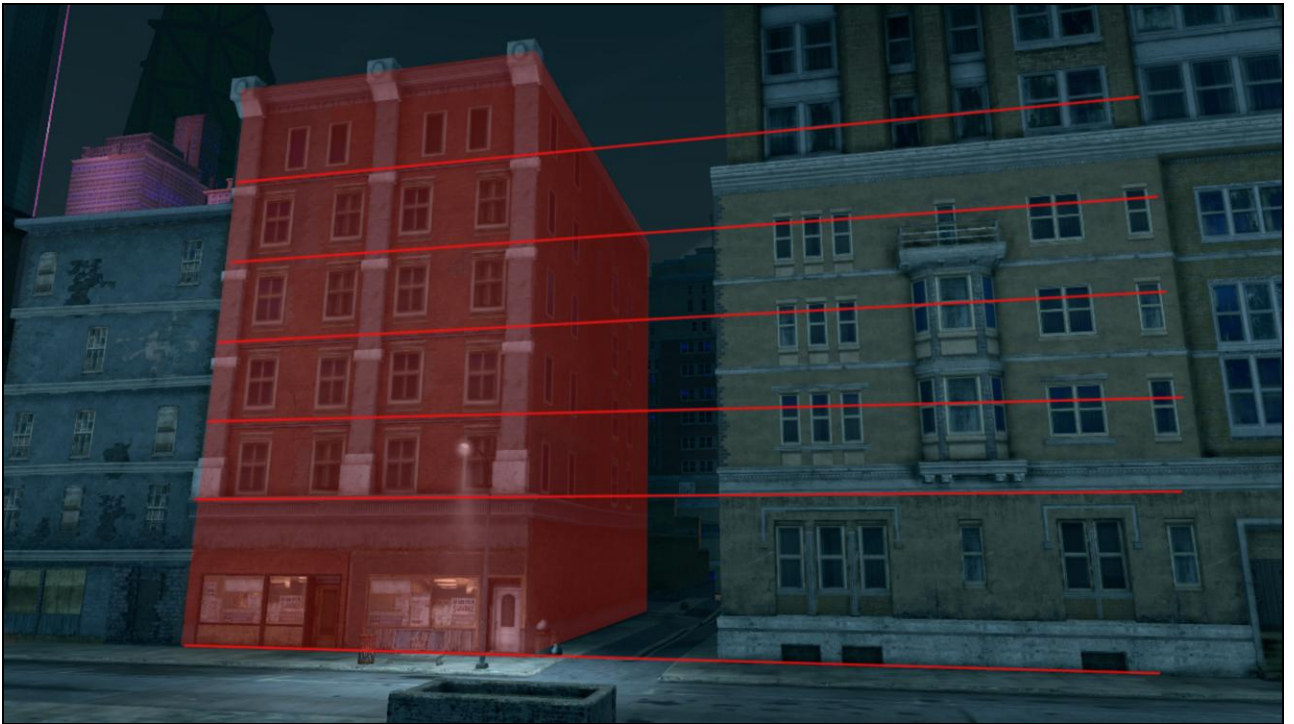




Here I've extended the lines that define the windows on these textures. Unfortunately, these textures don't share any horizontal or vertical consistency for a potential masking rule. Which means I won't be able to rely on hard-coded guides for my clamped mask in UV-space either.



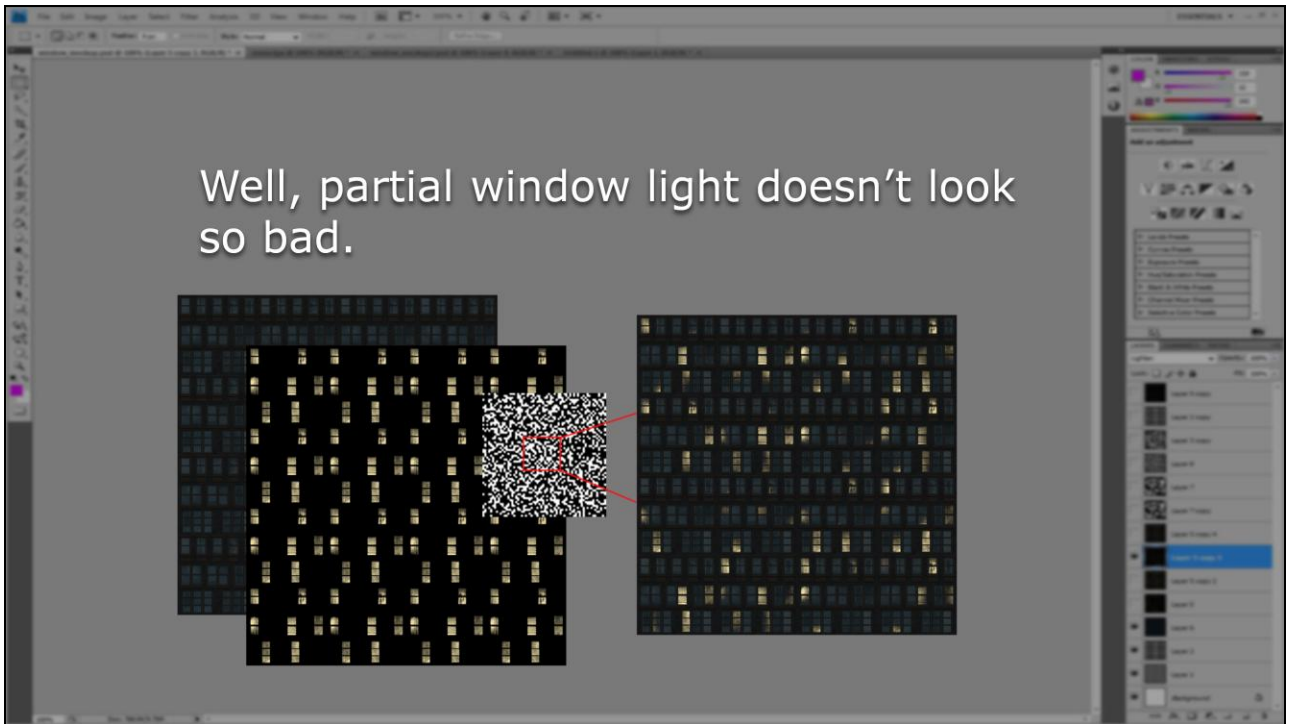
Unfortunately the same problem exists in world-space.



Building floors don't necessarily correspond to their neighbors, so the 'floors' of this building highlighted in red...

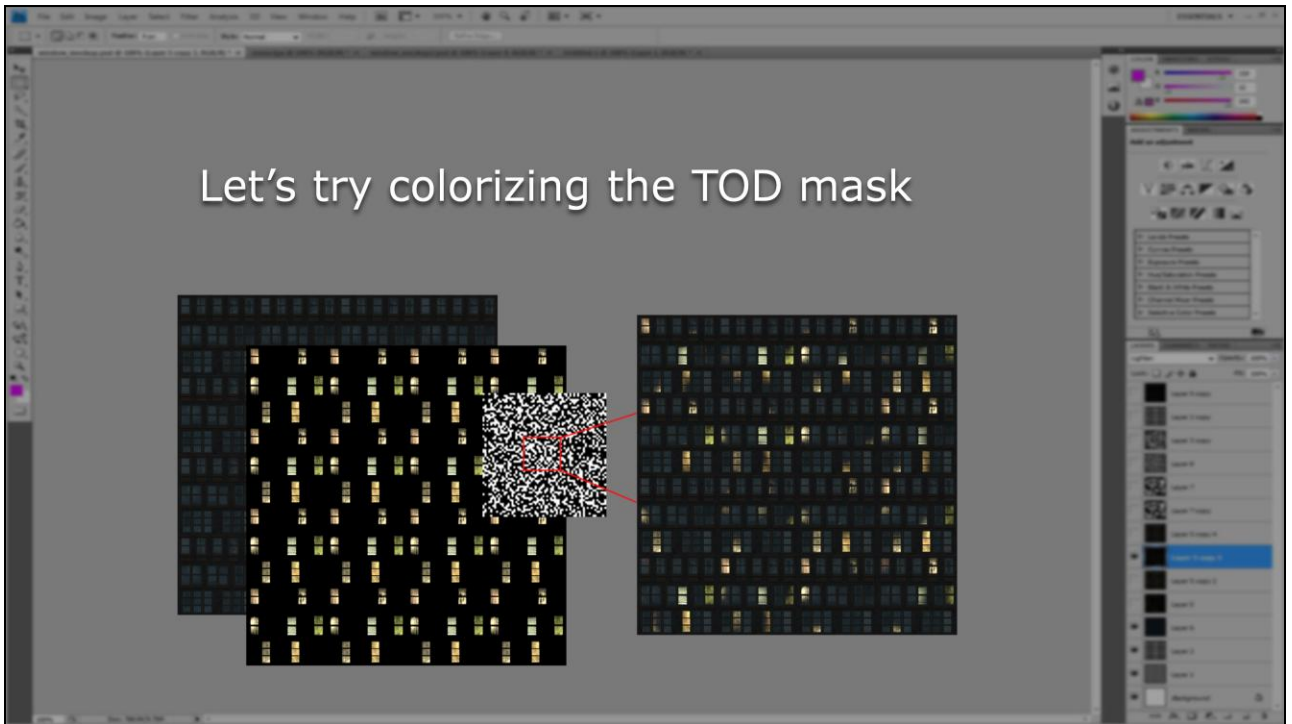


Don't align with the floors of it's neighbor, highlighted here in green. There's also no rule defining the height of individual floors within a single building. This makes sense – these buildings are of two distinct architectural styles. So it's looking pretty tricky to align a window glow mask in pure world-space. First and second floors in particular appear to vary greatly in relative height. At this point I'm not so sure if I can easily clamp the noise to make some windows fully illuminated and others dark without directly modifying asset geometry.

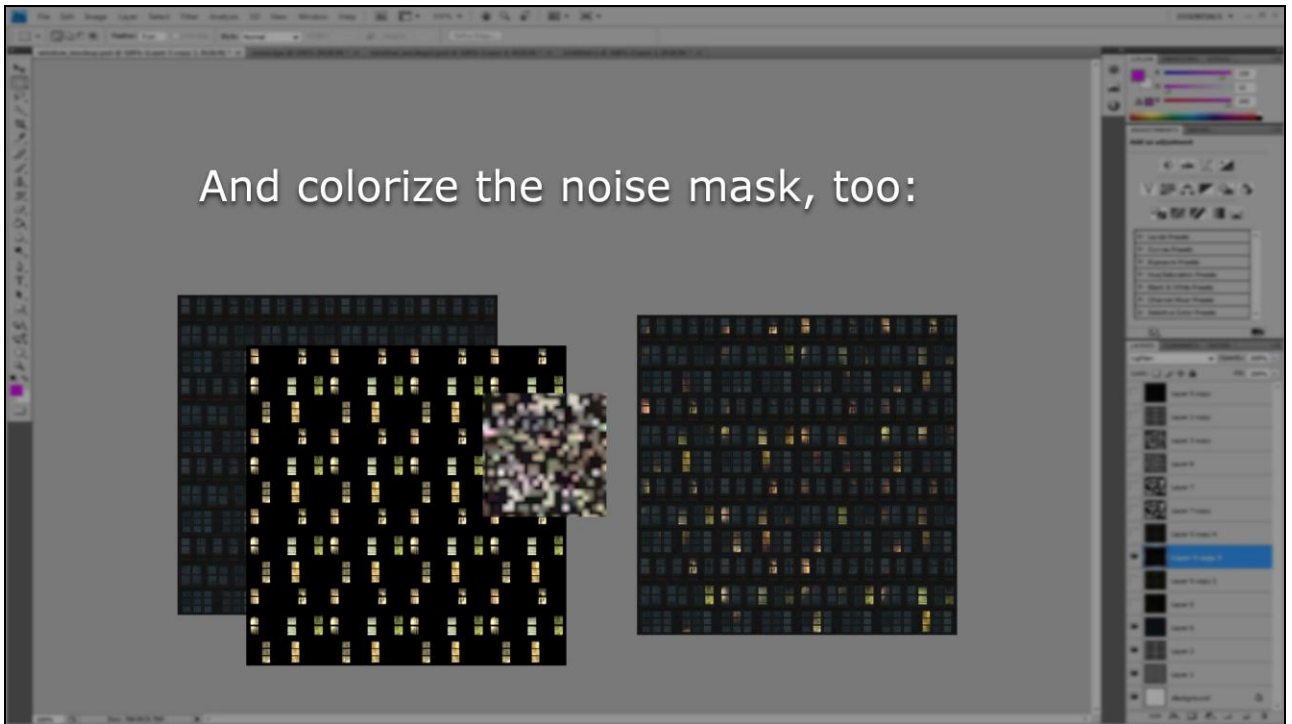


I might return to clamping the TOD mask map, but for the time being I'm going to switch gears and move my mock-up in a different direction. This isn't something I anticipated encountering, and it's raised further concerns about asset construction that I wasn't even aware of before. I haven't even begun to apply a solution for window glow maps and I'm already wondering if our structure construction methods don't need an invasive overhaul. At this point I've convinced myself that partial window lights don't really look that bad after all, and I should try colorizing the different masks instead.





Getting back to the mock-up. The next step I try is to colorize the time-of-day mask, and visualize that effect on the potential output. This provides some much-needed variation to the windows themselves, though it's still potentially a source of visible tiling. The time-of-day mask tiles along with the diffuse map; I'm assuming they will share the same UV information after all. So I think I need to break up the TOD color information using the noise map, just like I'm breaking up the luminance values of the time-of-day mask using noise.



It's starting to look pretty colorful – maybe too colorful. At least it's not visibly tiling, and that's a plus. At this point I think I have a viable idea to translate into HLSL and attempt to hook up in the game. I'm going to try and map the noise map on building geometry so that it covers the TOD mask in much the same way as the photoshop mock-up. At least, that's the plan.

A dark, blurry screenshot of a code editor. A semi-transparent black rectangular box is overlaid on the right side of the image. Inside this box, the text "Translate the mockup into an HLSL prototype" is written in white. The background shows faint outlines of code lines and a sidebar on the left.

Translate the mockup into an HLSL prototype

Here's some ultra-basic HLSL that should push our world-space coordinates out into color information.



Translate the mockup into an HLSL prototype

Simple world-space tiling:

`world_pos` is vertex position value + local offset

The idea is just to plug the X and Y world-space coordinate into the R and G output color, and we should see vertical gradient lines all over the world.

Translate the mockup into an HLSL prototype

Simple world-space tiling:

`world_pos` is vertex position value + local offset

Just so we can see the values:

```
float3 out_color = float3 (
    fmod( world_pos.x, 1.0 ),
    fmod( world_pos.z, 1.0 ),
    0.0 );
```

In specific HLSL syntax, this just means we construct a new float3 color using world position X and Z. Since values in world-space aren't within the zero to one range,

Translate the mockup into an HLSL prototype

Simple world-space tiling:

`world_pos` is vertex position value + local offset

Just so we can see the values:

```
float3 out_color = float3 (  
    fmod( world_pos.x, 1.0 ),  
    fmod( world_pos.z, 1.0 ),  
    0.0 );
```

I modulo the values against 1.0, so I should see the vertical lines I'm looking for.



And that's pretty much what we get! This is good – these might be good values to use as UV coordinates to map a mask texture onto all these windows.

Just plug it right into the sampler for world-space glow:

```
tex2D( glowmask, world_pos.xy);
```

So let's plug it in and try. Let's stick some world positions into our texture sampler and see what comes out.



Of course, just passing two values of a position into a UV look-up naturally doesn't just work. The effect is a tiled noise map that's effectively extruded through the world on one axis. On one axis buildings look fairly decent, but the mask streaks horribly along the other. This makes sense, I'm throwing away the world-space z position, after all. Faces along the z axis need that value to be mapped appropriately.

It occurs to me that most building faces in the world are aligned generally towards North and South, or East and West. My idea is to pick the closest axis to the normal direction of each face and then use the appropriate world-space component for the horizontal value of the UV lookup. So depending I should be able to map the mask onto any face in either the North-South or East-West directions, automatically.

Get NS-EW direction bias:

```
float bias =  
step( abs( dot( vert_normal, float3(1,0,0) ) ), 0.5 );
```

*North*

Here's some HLSL I used to do it. I extract a bias from the vertex normal. Dot the vert normal against North,

Get NS-EW direction bias:

```
float bias =  
step( abs( dot( vert_normal, float3(1,0,0) ) ), 0.5 );
```

Take the absolute value,



Get NS-EW direction bias:

```
float bias =  
step( abs( dot( vert_normal, float3(1,0,0) ) ) , 0.5 );
```

And step the result against 0.5.

Get NS-EW direction bias:

```
float bias =  
step( abs( dot( vert_normal, float3(1,0,0) ) ) , 0.5 );  
  
new_uv = float2(  
    lerp( world_pos.z, world_pos.x, bias ),  
    world_pos.y  
);
```

Then lerp between x and z of the world pos. This way I can pull the more relevant of the two horizontal world position values, X or Z,

Get NS-EW direction bias:

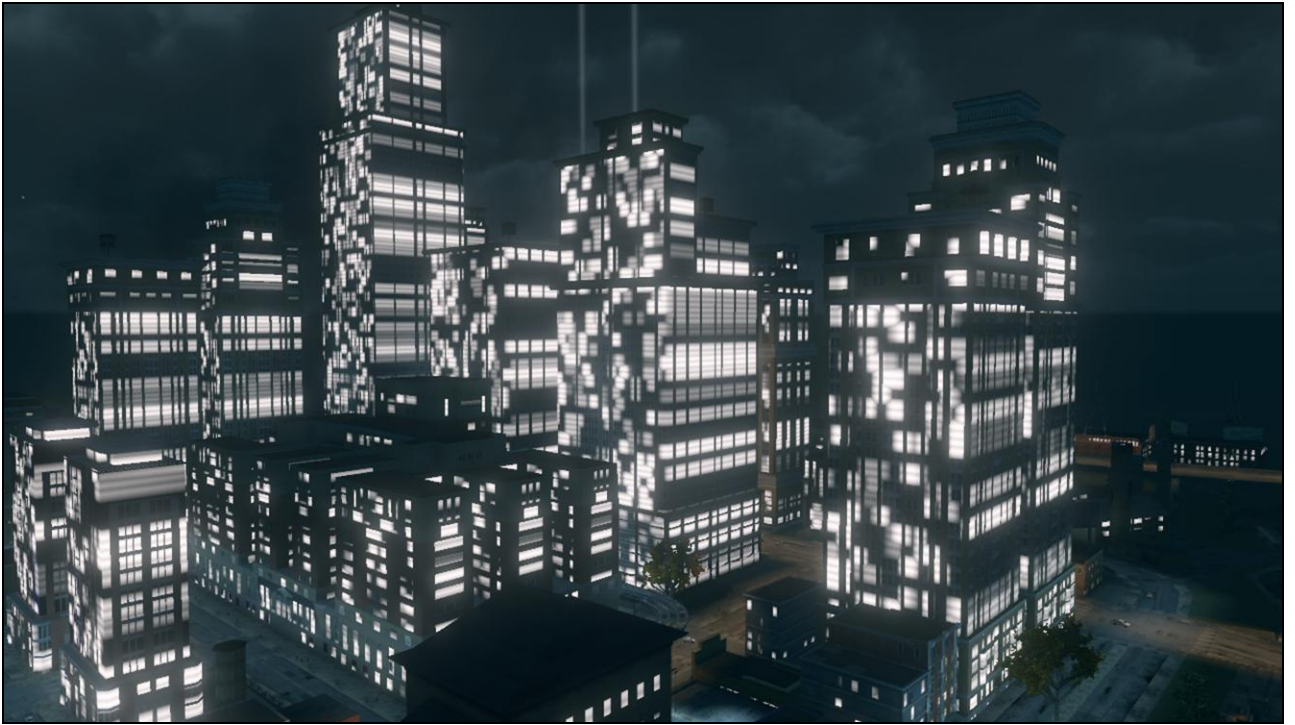
```
float bias =  
step( abs( dot( vert_normal, float3(1,0,0) ) ) , 0.5 );  
  
new_uv = float2(  
    lerp( world_pos.z, world_pos.x, bias ),  
    world_pos.y  
);
```

into the single horizontal value of the two-dimensional UV  
lookup,

Get NS-EW direction bias:

```
float bias =  
step( abs( dot( vert_normal, float3(1,0,0) ) ) , 0.5 );  
  
new_uv = float2(  
    lerp( world_pos.z, world_pos.x, bias ),  
    world_pos.y  
);  
  
tex2D( glowmask, new_uv );
```

And use that as the 'UV' coordinate to sample the noise texture.



Here's what we had before. If we apply that adjustment...



...with the north-south / east-west bias in place, the mask map maps mostly correctly onto the faces of the buildings using world-space values. It's still being effectively extruded through the world, but the angle of each face now dictates which 'extrusion' so to speak, it takes part in. This is the 'unusual' part of the UVs I'm talking about. Unlike traditional UV coordinates, there's nothing on the buildings that explicitly dictates the alignment of the mask – the UV information is derived purely from the vertex position of the geometry in the world.

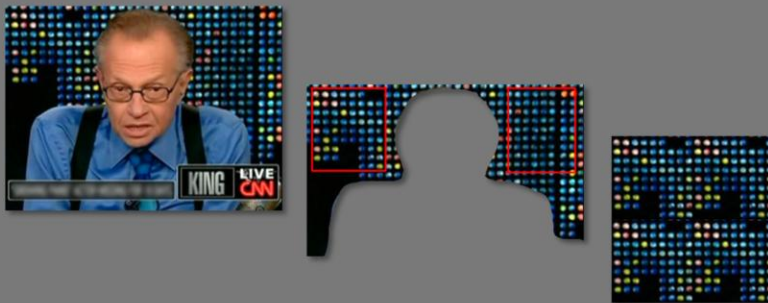


ToD mask applied

The Time of Day mask dictates what windows are illuminated at night – it tiles, so some windows will never illuminate. However this tiling effect is reduced when pushed through the noise of the global mask map, since a pseudorandom pattern of additional windows also won't illuminate when multiplied through that mask. Applying the ToD mask reduces the effect of the glow significantly, since now you can only see the glow on areas of the building textures that are supposed to be windows – this is what we want. But we're not done yet.



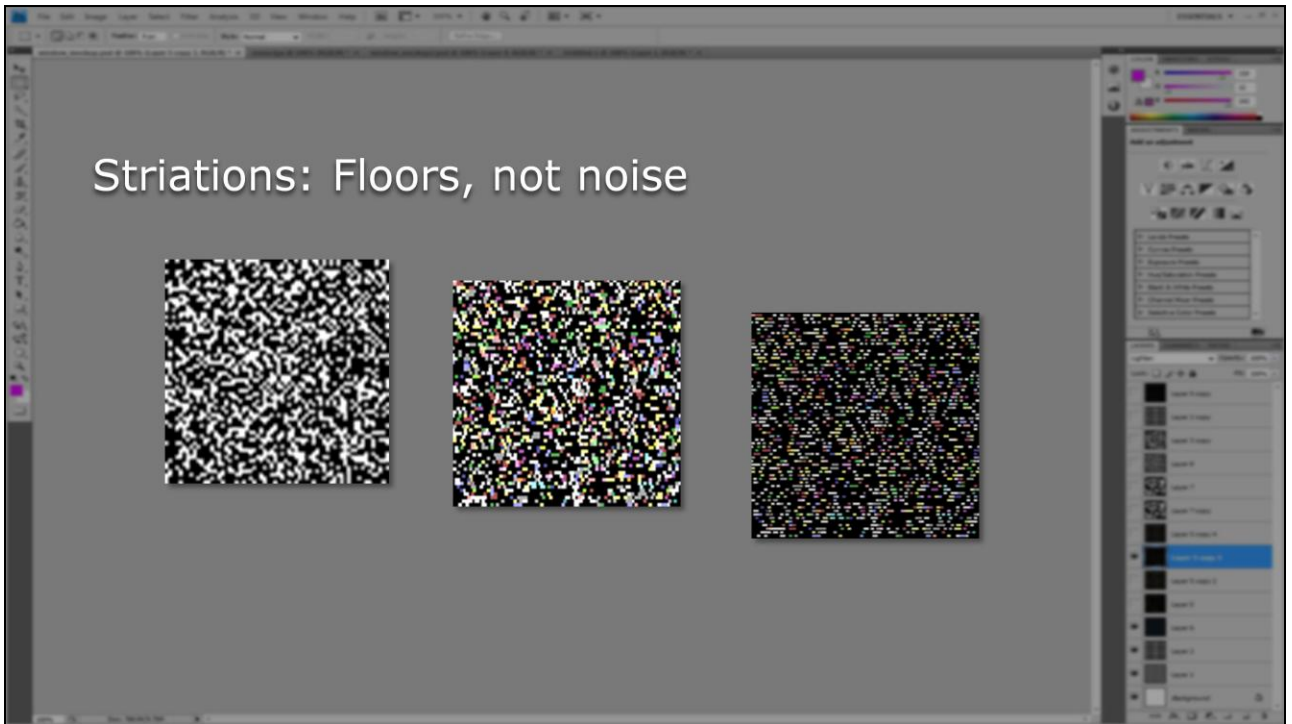
## Authoring a generic window glow noise map



Yeah, kinda like that!

At this point the best place to push more variation is the tiling mask map. I've never actually seen Larry King – but for some reason I remembered the backdrop in that show, and when I dredged my brain for what a global night window illumination noise map might look like, this was I thought of. You get the idea!





## Striations: Floors, not noise

Starting with the generic noise texture from early experimentation, the mask is colored with a combination of the SR3 art direction palette and eyedroppered colors directly from concept art, then evenly distributed. Scott Kircher, one of our rendering programmers, made the observation that it was far more likely to see horizontal rather than vertical patterns of illumination in building windows. Rooms may occupy multiple windows, but it was less likely for a room to feature more than one window vertically. To push a more striated floor appearance, I simply etched dark lines through the mask map. With a bit of luck and some global tweaking, I could align these striations to floor heights in the city.

I plotted this mask right in photoshop, but in retrospect it would have been cool to author a randomly seeded tool to generate a huge pile of them, and pick the best. I'll come back to this idea later. For now though, I'm curious to see what effect colorizing the mask map has on the city.



Before...



...After.

Colorizing the mask map definitely gets me the effect I'm looking for.

Since the global mask map drove the overall appearance of the window lighting so predominantly, it was easy to iterate and adjust the mood of the night-time world by altering the mask map. Along with variation in time-of-day, fog, and glow multiplier, I could emulate some of the looks and moods I had examined while collecting reference.



Here's a Deus-Ex style Steelport, with overbright halogen windows from a denser, yellowy mask. SR3's time of day system was linked to our full-screen look-up-table color correction system, so we could finish what the window glow map started with post-processing over the full frame.





Here's Steelport with a little more Liberty City vibe, driven by a desaturated, whiter, more elongated mask.



Once the content of the mask map was dialed in, I could further multiply the glow map UV lookup by a constant to affect the stretching, tiling, and aspect ratio of the mask over the entirety of the world. These final tweaks were mostly trial-and-error, just trying to balance colorization with getting the striation artifacts to line up best with the floors on most of the buildings.

Playing with the hue, saturation, value, scale and aspect ratio of the mask map allowed me to quickly adjust the look of the city globally to match art direction quickly and easily.

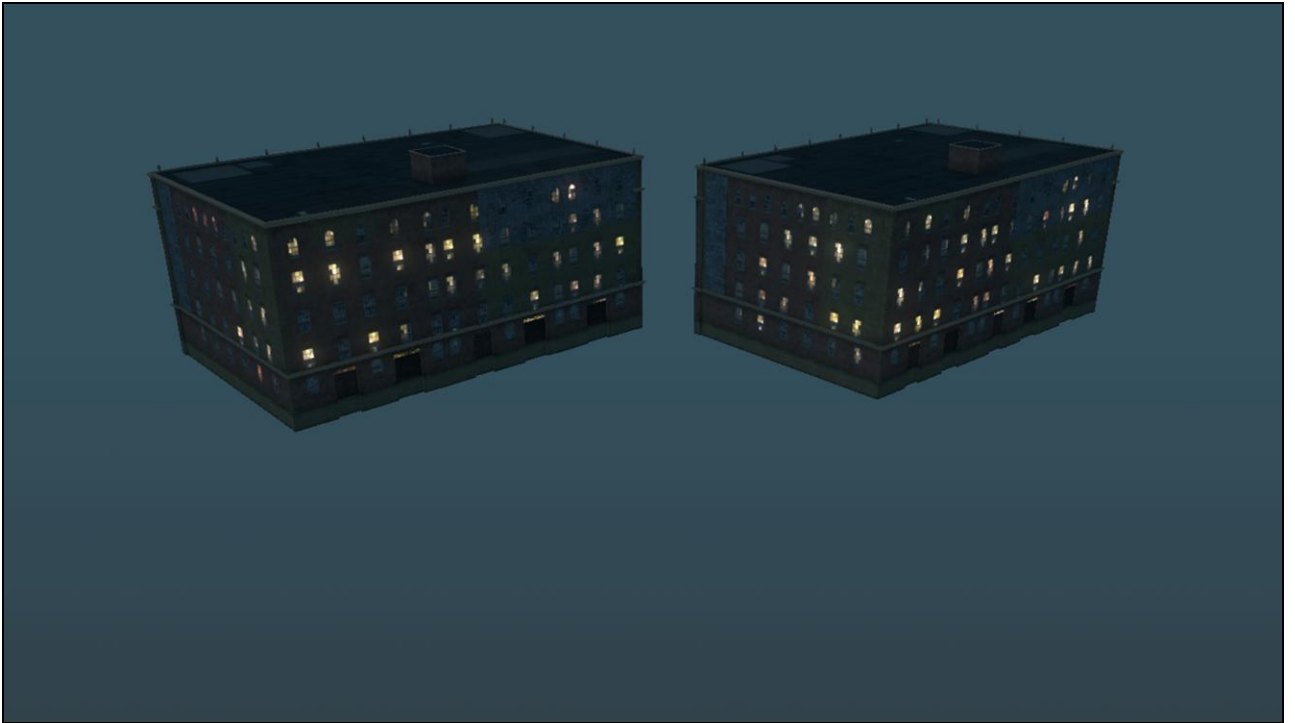
At this point I discovered some awesome ancillary benefits of world-space UV mapping the glow. Happy Accidents.

# Super ethical happy accidents



My original justification for mapping the glow mask in world space was simply so I didn't have to affect every building, I just had to assemble the global shader and then tweak the glow mask map. But this also meant that buildings would look up into different positions in the glow mask if they were positioned differently in the world, and if two buildings were at the same location (meaning right on top of each other) they'd look up into the mask map at nearly the exact same place. I was superficially aware this was the case, but I was very happy about two extremely important and helpful side-effects.

Firstly, by mapping window glow using world-space position that wasn't applied in any way to buildings,



two identical buildings at slightly different positions in the world would get two clearly different window patterns. Check out these two identical warehouses that are mere meters apart...

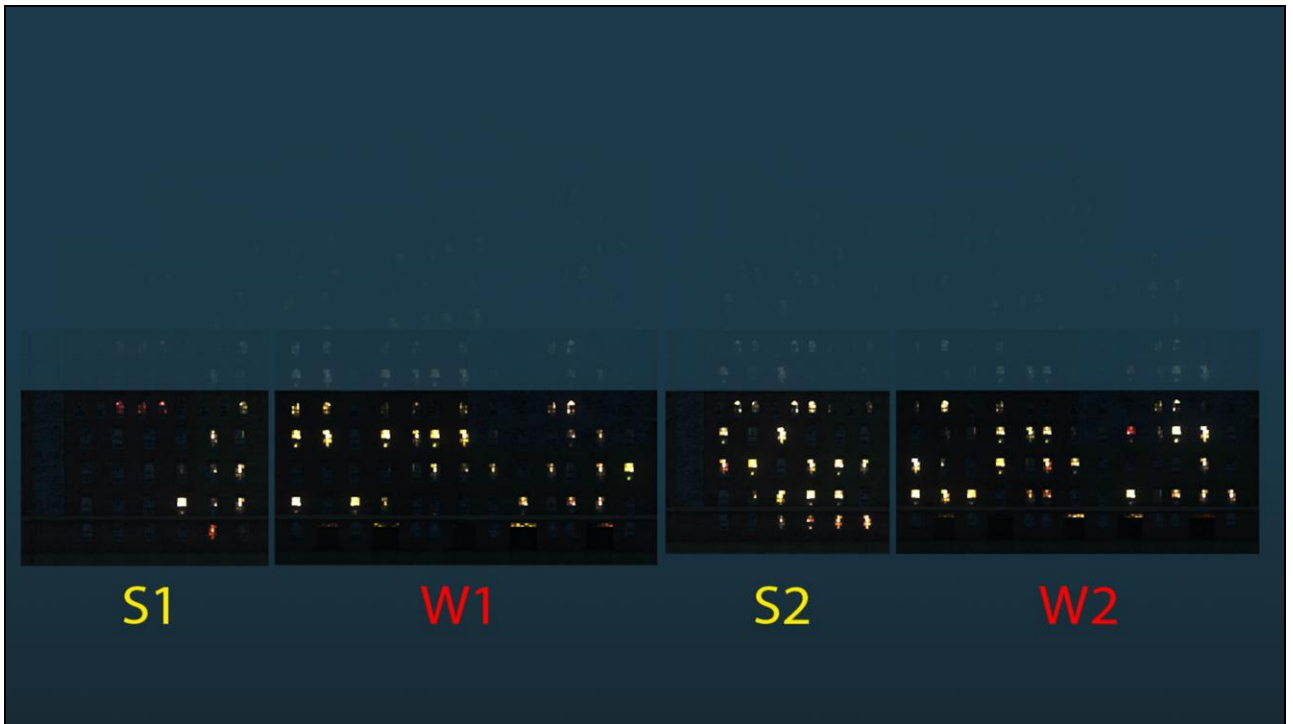




Just for comparison, I'll pull the sides off the buildings



and flatten them out next to each other, so you can see...



The corresponding sides, south to south, west to west, have clearly different glow patterns, even though this is the exact same building. I've boosted the glow in this slide to call it out. This is awesome – it means that duplicated buildings in the world will appear varied (at least at night), and without any input from an artist or world builder. The very act of placing the same building in a new location makes the window glow pattern change, automagically.

Interestingly, this does mean that the window glow pattern visibly 'slides' across the buildings when the artists are moving the structures in our level editor. But since buildings never move once they're in the game, this is just a curiosity, and not a big deal.

The second benefit is that buildings at identical locations will share the exact same glow pattern. At first that seems like an odd thing to be excited about, why would you ever need to put two different buildings at the exact same position? Well that's exactly what our level of detail system effectively is.



As buildings become more or less distant, they swap to a combination of procedurally generated and hand-authored models at varying levels of detail. Distant buildings have fewer details...



And closer buildings have more detail.



As far as the game engine is concerned, LOD models are effectively different buildings altogether. And in the interest of optimization, sometimes they have radically different polygon flows or UV layouts to make them look better or more run more efficiently at a distance. As we get closer to this building, its LOD model is changing.





What's cool about world-space glow mapping is that each successive LOD model will have a pixel-perfect glow map pattern, exactly the same pattern as its higher and lower detail siblings – making the transition between levels of detail almost completely invisible when looking at night-time windows.

How cool is that?!

Now look at that orange/red window there in the center...



And follow it back...









I know you probably can't see on this slide, but that red window is still there – and that continuity over distance helps a ton to sell the physical reality of our game world.



And distant twinkly lights on the xbox are crisp and clear on ridiculous PC setups.

Spot-  
fixer?



Now for some spot fixes. There are outliers to any generalized solution, and window glows, while simple, was no different. I knew there would be some spot-fixing required throughout the city due to the sheer scale and variation of all our building assets.

Thankfully for each of these I only had to subtly modify the shader to account for these differences, just making the global solution work better overall.





Windows on the ceiling! Why are there windows on the ceiling – who would do this – why, oh jeez. No I get it, it's like a hangar with skylights, that's cool. The world-space glow clearly isn't helping these out, though. I've already got the face normal in the shader though to extract the north-or-east bias though, so the easiest fix is to dot that against an up-vector, and bash the glow value as window geometry approaches horizontal.



Ta daa! No problem. And it doesn't seem to hurt any other building in the game world.



What's going on here? There's like a stretched skew squeezed thing going on. Gross.

Turns out this makes perfect sense, and comes back to my assumption that most building faces are either mostly north-south facing or east-west facing. This semi-circular building just has to be different, and normally even that would be ok, since each particular face would choose only one or other axis of mask-mapping. The trouble in this case is however that an erroneous smoothing group is running across these faces, and averaging the vertex normal between the two.





The averaged vertex normal means that the mask map UVs get warped – and the north-or-east bias is freaking out.

To fix this I dive into 3ds max, and pay two vertices to put a hard edge between these faces...



– that separates the vertex normals on this face and lets each face look up into the mask map correctly.



Limited scope of production meant there were some areas for improvement or spot fixes that I didn't have time or resources to attack – and that's another reality of game development.

For example, a better way to align the glow map in world space may have been to construct a matrix using the face normal, tangent and binormal of each face, instead of finding the closest bias to the north-south or east-west directions. Combined with a world-space offset, this might have aligned the mask map more appropriately to faces in the world. Cargo ships have illuminated windows, and while most of them are ok...



The window glows don't tilt to match the list of this partially-sunken ship. Though to be fair it's odd this boat is still powered up – we probably should have just applied a non-illuminated window material to this guy. The engine room is probably well underwater at this point, after all.





Still, I'm fairly pleased with the result of this fix. The goal of the process was to apply the exaggereal look, feel, and mood of Saints Row the Third to the building windows – both in terms of up-close contrast and variation...



...and distant consistency.

The process is ongoing, and I've identified several areas for improvement. I've learned a great deal about how assets are currently constructed in the city, and I can apply that knowledge towards a wide variety of future issues, should they arise.

I've also identified several advantages and potential caveats and restrictions of unusual UV mapping, specifically applying a global mask in world-space. Some of the problems I didn't have time or answers for during production, such as uniformity of texture layouts or raw triangulation of our buildings need to be investigated further.

Most of all I've gained more insight into thinking about game resources in terms of massive, generic, global re-use. Technical standardization is incredibly important, such as texture and UV layout standards – to artistic and stylistic consistency, such as floor heights and intra-building construction rules. The lack or presence of each of these concepts greatly influenced the way I attacked, adapted, and

worked around the problem at hand.



Thinking about a variety of ways to approach any given problem often inspires more ideas and potential solutions for the future. Game development solutions are always on-going, and dreaming up additional creative features is part of the fun.

After the basic night-window implementation, I'm already thinking about how we can make it better. Maybe we can animate or cycle the glow to adjust the seeming 'population' of the city, or maybe we can update the ToD mask map for each mission to affect the visual 'mood' – what would this look like? What per-mission moods might we want? This would be something cool to run by design – I bet they'd be totally on board.

The completion of this cycle is for solutions to open new avenues, ideas, questions and possibilities for design and art direction, by raising ideas like these. Ensuring that efficient technical fixes are grounded in an understanding of the game's artistic and thematic requirements is vital in order for



this to happen.



Thank you!

So I want to thank everyone very much for listening, and I might have time for a couple questions. Thank you very much!

# Technical Artist Boot Camp



## Group Q&A

**GAME DEVELOPERS CONFERENCE**  
SAN FRANCISCO, CA  
MARCH 5-8, 2012  
EXPO DATES: MARCH 7-9

**2012**