



SIGGRAPH 2013

Destiny: From Mythic Science Fiction to Rendering in Real-Time

Natalya Tatarchuk, Engineering Architect, Bungie

Chris Tchou, Senior Graphics Engineer, Bungie

Joe Venzon, Senior Graphics Engineer, Bungie

Outline

- What is *Destiny*
- Graphics Technology and Features behind *Destiny's* new rendering engine
- Conclusions

What is *Destiny*

Graphics Technology and Features behind *Destiny's* new rendering engine

Deferred rendering in *Destiny*

Our deferred rendering extensions

Transparent lighting

Particle rendering optimizations

Conclusions

Bungie Graphics Team

Hao Chen, Chris Tchou, Joe Venzon, Jason Tranchida,
Brad Loos, Brandon Whitley, Jason Hoerner, Alexis
Haraux

+ Activision Central Tech

Josh Bloomstein, Ryan Sammartino, Charlie Birtwhistle, Paul
Malin, Jan Van Valburg, Chris Cookson

A lot of people worked on our game, and I want to do a shout out to all their hard work!

Pre-alpha state → some of these techniques are still being worked on and may change



The game is in pre-alpha state

Some of these techniques are still being worked on and may change

Can't show full set of assets as some are still being finalized

A dark, atmospheric slide featuring the word "DESTINY" in white capital letters. To the right of the letter "Y" is the iconic "Worm" logo from the game. The background is a dark, textured space scene with a glowing circular element on the left.

DESTINY

Bungie is entering a brave new world of Destiny – our upcoming title revealed earlier this year



A few basic facts about Destiny:

It's a shared-world shooter.

This means a shooter that takes place in a world you share with other players.

A game where you invest in a character that you persists across all game modes.

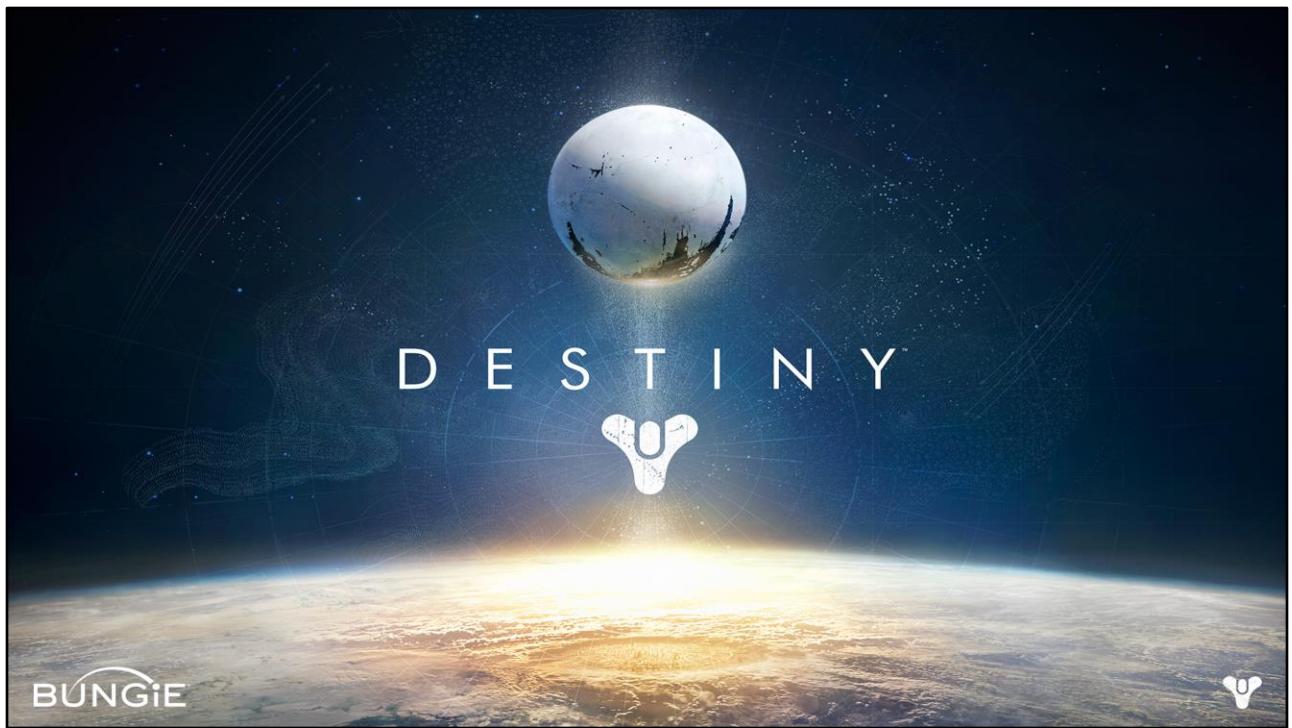
The character you build for the co-operative story content is the same character you'll use in competitive multiplayer.



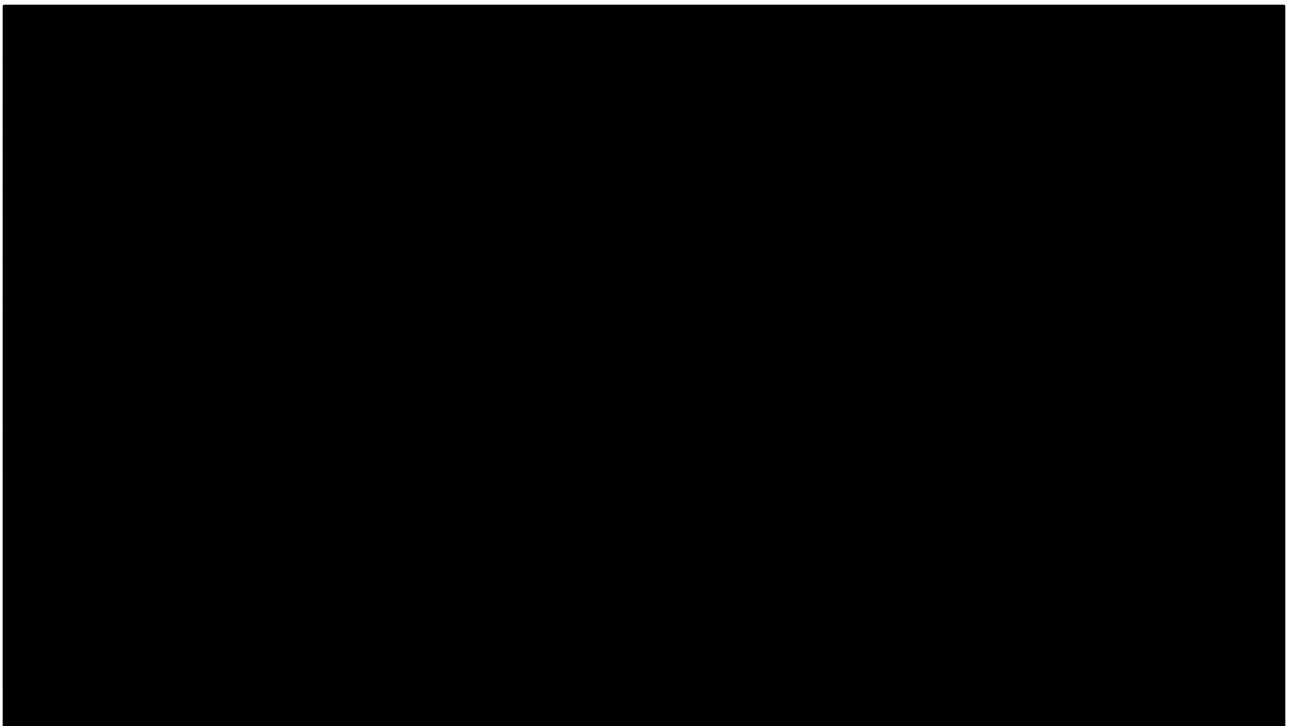
Destiny is an action game through and through, with the same great sandbox and emergent gameplay you'd expect from Bungie.

I'm sure there will be plenty of Bungie folk at next GDC talking about Destiny's game mechanics etc.

Today, I am here to talk about what it takes to bring Destiny to life from the standpoint of real-time rendering.



Which brings us to the recent reveal of Destiny gameplay we showed at E3... so hot off the presses



INTRO video (Destiny E3)



Destiny is filled with mystery and adventure, where each destination telling it's own story.

How do we build that? How do we give the artists and designers what they need to tell these stories?

We asked ourselves that question when we first started to work on Destiny. And unsurprisingly the answer came back... To do this thing right we need to build a whole new engine and import pipeline and set of tools around our new engine. That was terrifying because it meant we were leaving behind the old and comfortable tools that we knew so well But it was also exhilarating because we had an opportunity to build something new and amazing.



Let's take a look at the pillars of Destiny design from the vantage point of a graphics engineer (or a researcher).

Bungie is a design-driven company. Which means that all engineering features first and foremost must serve the game. If you work on the most glorious graphics feature, but it'll never be experienced by the player, or, worse yet, it actually could impede content creation or design, that is not a successful graphics feature in our book. We start from our creative vision for the game, and the work from that to translate it into features that will help actualize and ground this vision.

What does it take? What is the vision of Destiny?

Pillars of Destiny's World

As Translated by a Graphics Engineer

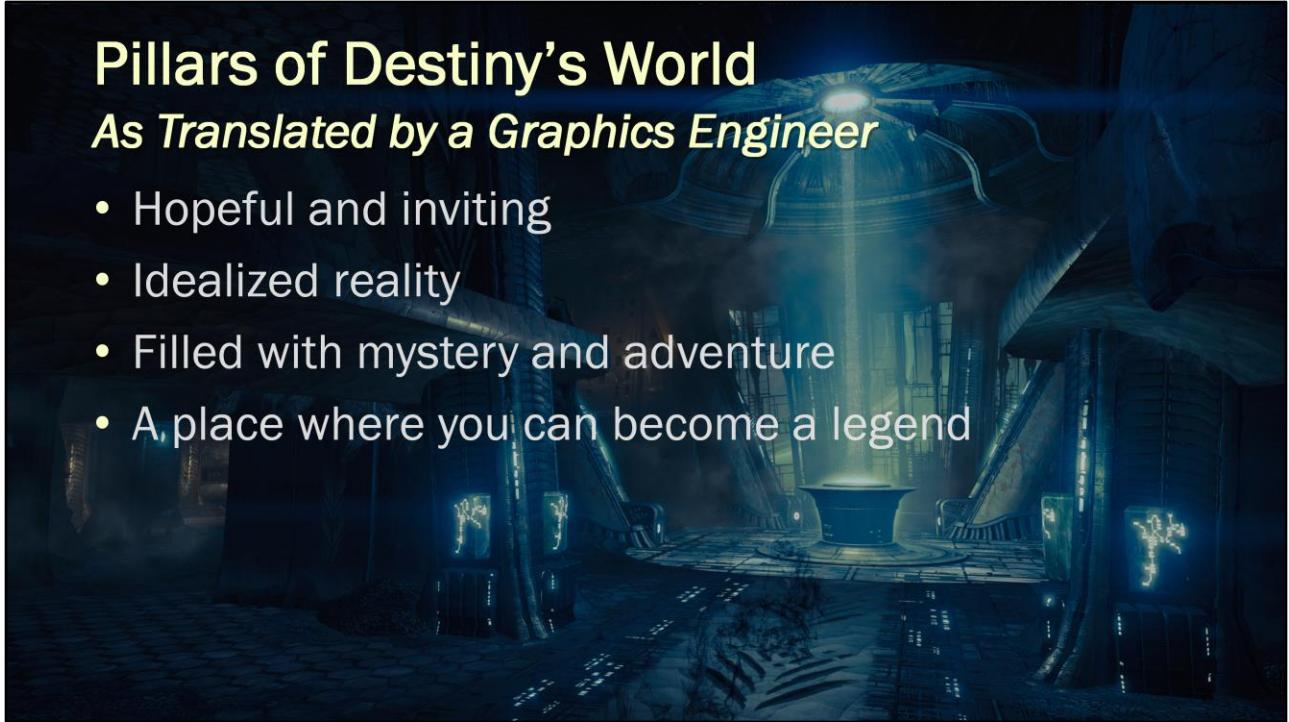


So what are the pillars?

Pillars of Destiny's World

As Translated by a Graphics Engineer

- Hopeful and inviting
- Idealized reality
- Filled with mystery and adventure
- A place where you can become a legend



Here are Destiny design pillars



Hopeful and Inviting

#1: People don't like to spend a lot of time in dismal, hostile places.
Sure, there had to be some dark and dangerous places in Destiny's world.
But overall we wanted it to **be a hopeful and inviting place.**

In graphics features terms this meant

Idealized Reality



Pillar #2: We also wanted a world that was instantly familiar, but left room for the strange.

A world where palette, composition and mood was more important than photo-realism. ***Idealized reality***

In graphics terms, this meant Photo-realistic yet stylized material model
Giving artists controls that were meaningful to them. Physically based up to a point – predictable response with artist-friendly and intuitive for controls for our techniques.
Allow artists to paint BRDFs and atmospheric models to go *beyond physically based* models



Enabling palette and mood also means having high quality lighting – after all, achieving artistic vision means giving them tool to express it.

And no lighting is feasible without a good solution for ambient occlusion (both global illumination and dynamic (HDAO)

Here is a scene from our level ...



And here is the ambient occlusion contribution by itself for the scene in previous slide
We have developed a new pipeline for fast generation of ambient occlusion for our
levels for global illumination in conjunction with NVIDIA research with Peter-Pike
Sloan (which is a topic of a whole separate new talk)



Also what idealized reality can be achieved without *Lens flares or Custom color grading and chromatic aberration effects*

This is a large cave underneath the surface of the moon. Notice the bumpy rocks illuminated by the light pouring in from the cave opening, and the cracks and creases are darkened by its surroundings , and the light beams casted by the bright sun strike through the thin atmosphere.



And, of course, God rays..

Imagine taking your fire-team to Earth and watch the Sunset over the ruins of the Golden Age...

Filled with Mystery and Adventure



Next pillar is “Filled with mystery and adventure”

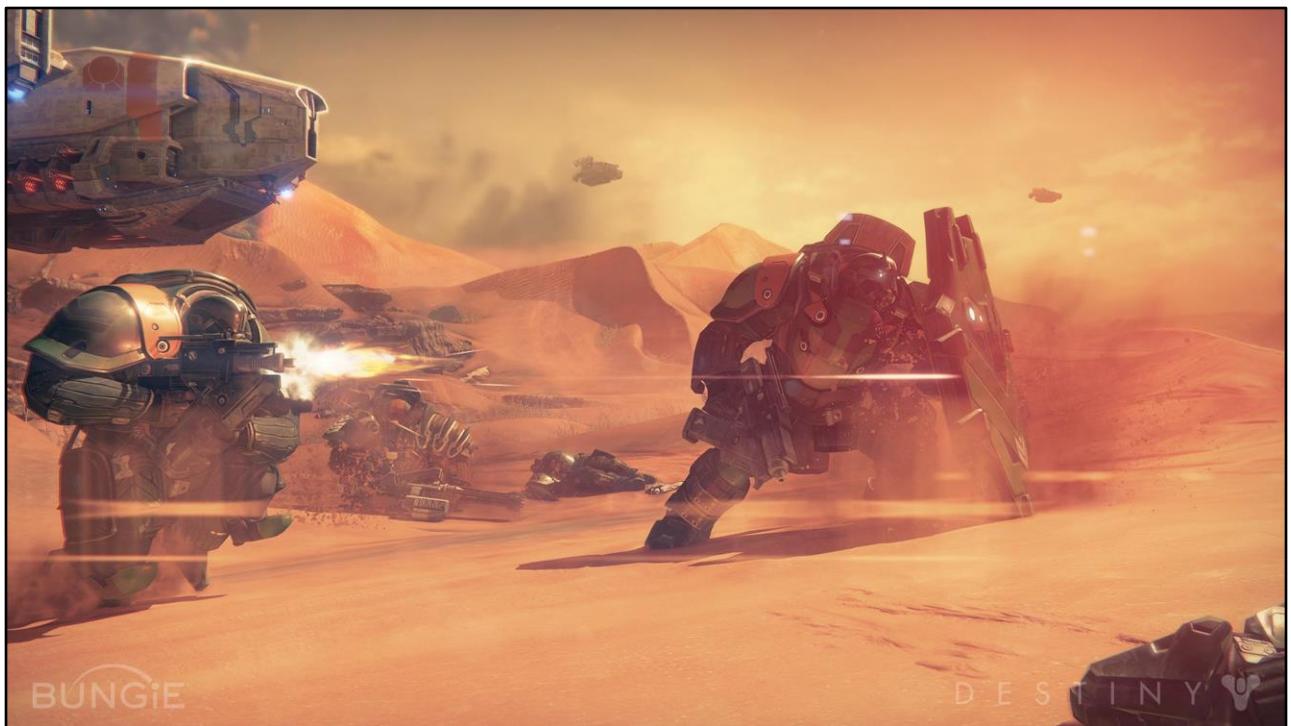
#3: A place that made you curious. A vast frontier of adventure.

Filled with history, waiting to be explored.

Which in graphics feature terms can be translated as ... **Space magic: EFFECTS!**

EFFECTS! EFFECTS!

Heres another example, the whole scene is lit by the **particle beam** in the middle of the room, and there are many small lights that creates interesting pools of light and shadows. Now all this is **real-time** of course, so the lights can animate, and flicker which they do in the real game which makes this static scene comes to life even without the characters living in it. Another by-product of **real-time lighting**, which was our intent all along, since the world of Destiny is persistent, and people from all over the world will join the game from different time zones, is dynamic time of day. Meaning players can experience different time from sun rise to sun set and at night while playing the game. To do dynamic time of day, lighting, shadow, atmosphere, even ambient sound all have to work in coordinated fashion.



“Filled with mystery and adventure”

Plentiful Atmospheric effects and high quality atmosphere rendering allowed us to create many different mysterious places



“Filled with mystery and adventure”

Using Depth of field and motion blur and other screen effects punctuates the sense of dynamism and adventure in our game (and here is example from a cinematic)



A Place Where You Can Become a Legend

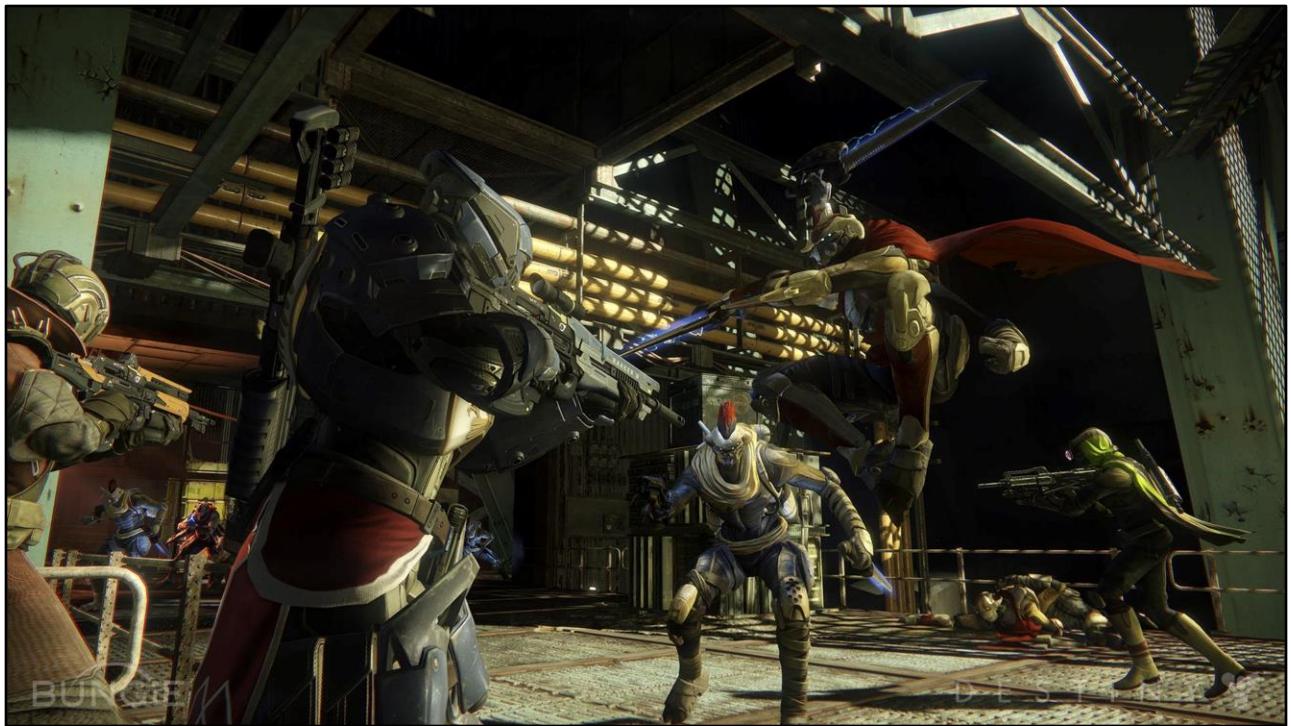
And the last design pillar was “**A place where you can become a legend**”

#4: Where every visit made a difference.

And where every victory increased your legendary status among friends and enemies.



In this shot you can see the **three players with different progression** – the foreground character is the highest level character in this scene, and you can clearly see that in his impressive gear set, but even other characters have very distinctive looks that are instantly identifiable by their silhouettes which players made their own through our customization system.



We needed to find a way to avoid breaking the illusion for our legends and that meant lots and lots of **Simulated cloth**

Notice the cloaks, sashes, capes and other garments that are moving along with the characters in the battle here



And did I mention EFFECTS?!

How could you become a legend in a sci-fi world without special abilities like Nova bomb here which are all done with particle and screen effects in our game

Destiny Graphics Features

- Vector Terrain
- Moving Trees
- Rivers and Lakes
- Customizable Gear
- Real-time Cloth
- Facial Tech
- High quality real-time shadows
- FX pipeline
- Imposter System
- Tessellation
- High quality AA
- Large Scale AO (GI)
- Visibility
- Dynamic time of day

So what does this all mean in terms of the graphics features we had to develop for the new Destiny engine?

We spent 4 years on this engine and we were forced to be brave, and tackle quite a few hard problems. 4 years later we have built a truly state-of the-art engine with many cool features. It supports multiple platforms by design. It has a flexible and efficient multi-threaded architecture that allow us map well to current generation of console as well as whatever comes in the horizon. It has awesome character tech like customizable gear and convincing facial animations, and many others. But our biggest achievement, and one that we worked the hardest, and we are most proud of, is the new art production pipeline that removes many of the shackles from our artists, and let them focus on art creation instead of technical nitty-gritty. At the end of the day, all this cool technology is for nothing if our artists can't use it.

Destiny Graphics Features

- Lens flares
- Atmospheric lighting
- Volumetric Fog
- Dual Quaternion skinning
- Linear blend skinning with non-uniform per-bone scaling
- Custom facial animation system
- Run-time animation retargeting
- Simulated Cloth
- Subsurface scattering for skin, hair, other translucency
- Efficient chunking for rendering with automatic LOD

And more features

Destiny Graphics Features

- Custom character lighting rig
- Object effects galore (disappearance, etc.) and complex and custom effects system
- Automatic runtime plating for character gear
 - Customizable even post release
- Complex and detailed shadows
- Wind impulse system
- Decorators:
 - response to external disturbances and motion under influence of wind.
- HDAO
- High quality cinematic depth of field and motion blur
- Water system with realistic dynamic reflections and response system
- Sky and Clouds:
 - dynamic time of day.
 - 3D clouds and cloud layers.
- Atmosphere and fog:
 - scattering from artificial lights.
 - god rays.

And even more features... and then I got tired of typing them individually

Destiny Engine Goals

- Accommodate a large spectrum of art styles
- Achieve good balance between artistic style choices and realism
- High-quality Visuals
- *Non-goal: be a general-purpose graphics engine*

Accommodate a large spectrum of art styles

Achieve good balance between artistic style choices and realism.

Be modular and extensible where it makes sense.

NON GOALS

Be a general purpose graphics engine.

Be a showcase of anything other than the actual games we are delivering.

Destiny Renderer Key Focus Areas

- Efficient content creation process and pipeline
- Believable and complex characters
- Highly interactive, dynamic worlds
- Ability to handle huge complexity
- Architecting to fully utilize the power of the next-gen hardware
- Scalable for current generation of consoles

KEY AREAS TO FOCUS

Content creation process and pipeline.

Lifelike characters.

Highly interactive, dynamic worlds.

The ability to handle huge complexity.

Architecting to fully utilize the power of the next-gen hardware.

Destiny Engine Technical Goals

- 5 platforms
 - Xbox 360, Xbox ONE, PS3, PS4 and PC (studio-wide development platform)
- Scalable performance and memory footprint
- Low latency input response
 - From button press to display flip

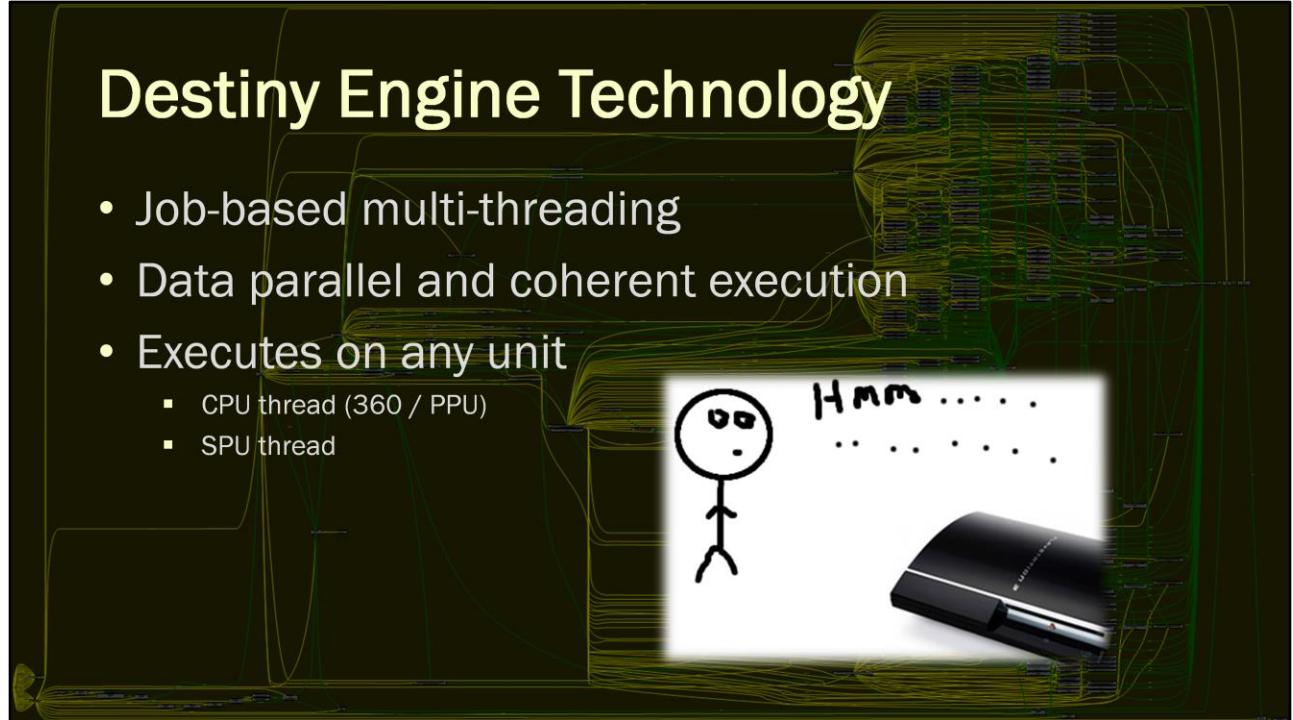
Which of course necessitated that we have a flexible and powerful engine.

The goals for our new Destiny Renderer were quite ambitious:

- Data-driven rendering pipeline
- Well-designed multi-threading architecture
- Support multiple platforms
 - Ability to load balance across heterogeneous compute platforms:
 - Ex: 6 HW threads on 360 versus 2 HW threads + 6 SPUs on PS3 versus next-gen
- Well integrated with Bungie's object system and supports determinism
- And, of course, since we are a high paced shooter game, we need to make sure that performance is king as we need to maintain low latency input response (which, for 30 fps, means somewhere in the 70..90 ms range).

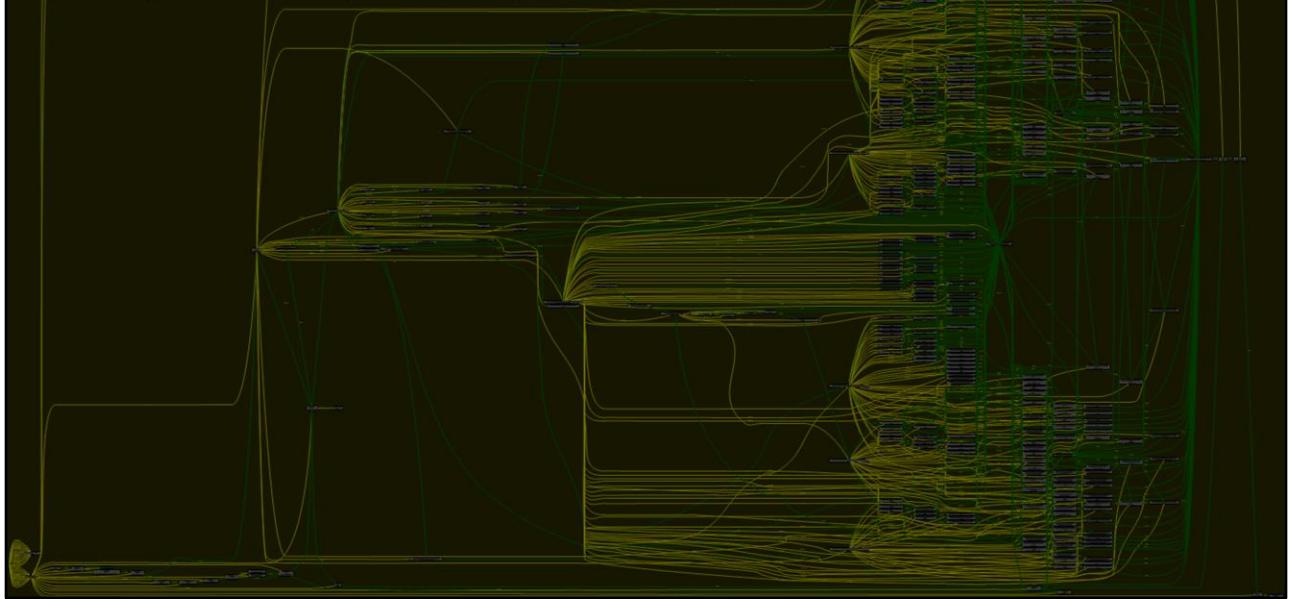
Destiny Engine Technology

- Job-based multi-threading
- Data parallel and coherent execution
- Executes on any unit
 - CPU thread (360 / PPU)
 - SPU thread



Destiny renderer was one of the key clients and driving forces behind our engine's multi-threading and we're currently executing a wide variety of rendering jobs on SPU and all CPU units

So What Makes Games Tick?..



Let's take a deeper look of what it means to make games tick...



BIG BANG THEORY VID



More on Destiny Renderer & Visibility System Architecture

Making Game Worlds from Polygon Soup: Visibility, Spatial hierarchy and Rendering Challenges (SIGGRAPH 2011)

Hao Chen and Natalya Tatarchuk (Bungie), Ari Silvennoinen (Umbra)

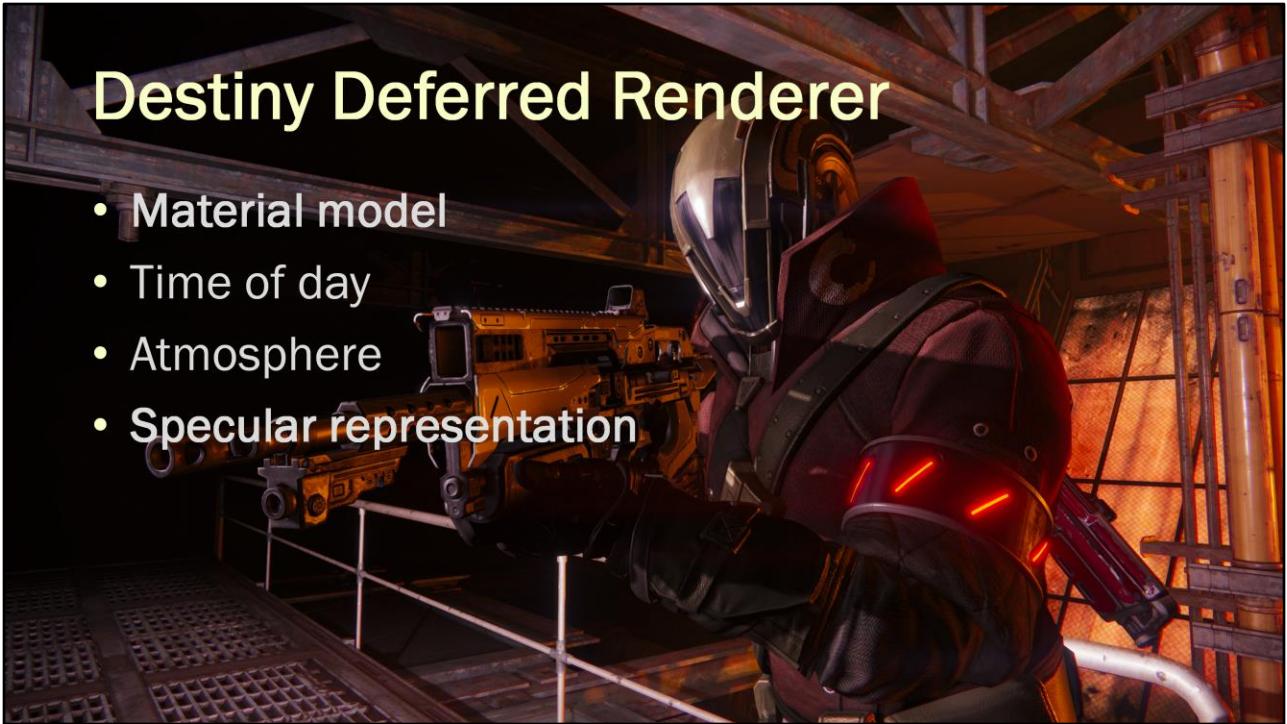
<http://advances.realtimerendering.com/s2011/index.html>

Powering up *Destiny's* Level Creation and Rendering with Umbra 3 (GDC 2013)

Hao Chen (Bungie) & Otso Makinen (Umbra)

Destiny Deferred Renderer

- Material model
- Time of day
- Atmosphere
- Specular representation



Rendering

Our deferred renderer had to support a flexible material model, dynamic time of day, full-scale atmospheric effects and a powerful physically based specular representation

Because of the scope of this presentation, I won't get into our implementation for time of day and atmosphere (We actually covered some of those topics in our 2009 presentation in this very course)

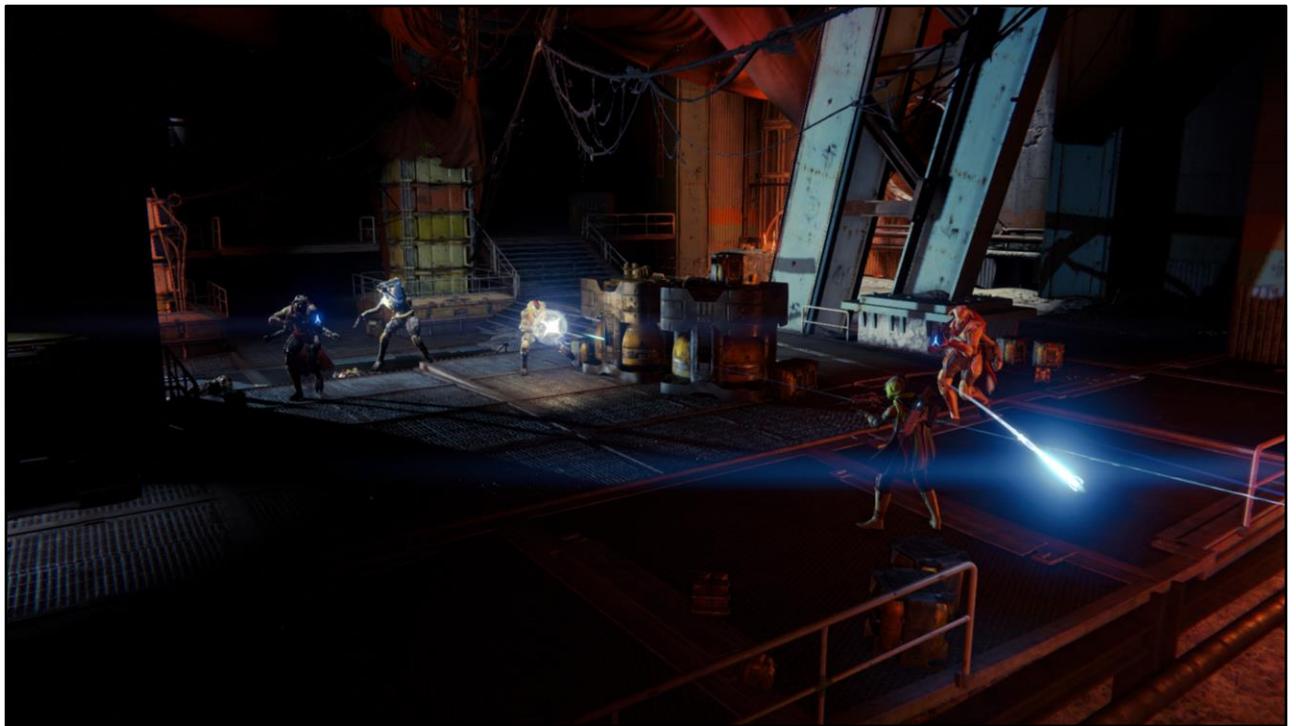
However, I will describe our material model and specular representation

Go faster, look better!

Our goals for the new rendering method are, simply, to go faster and look better!

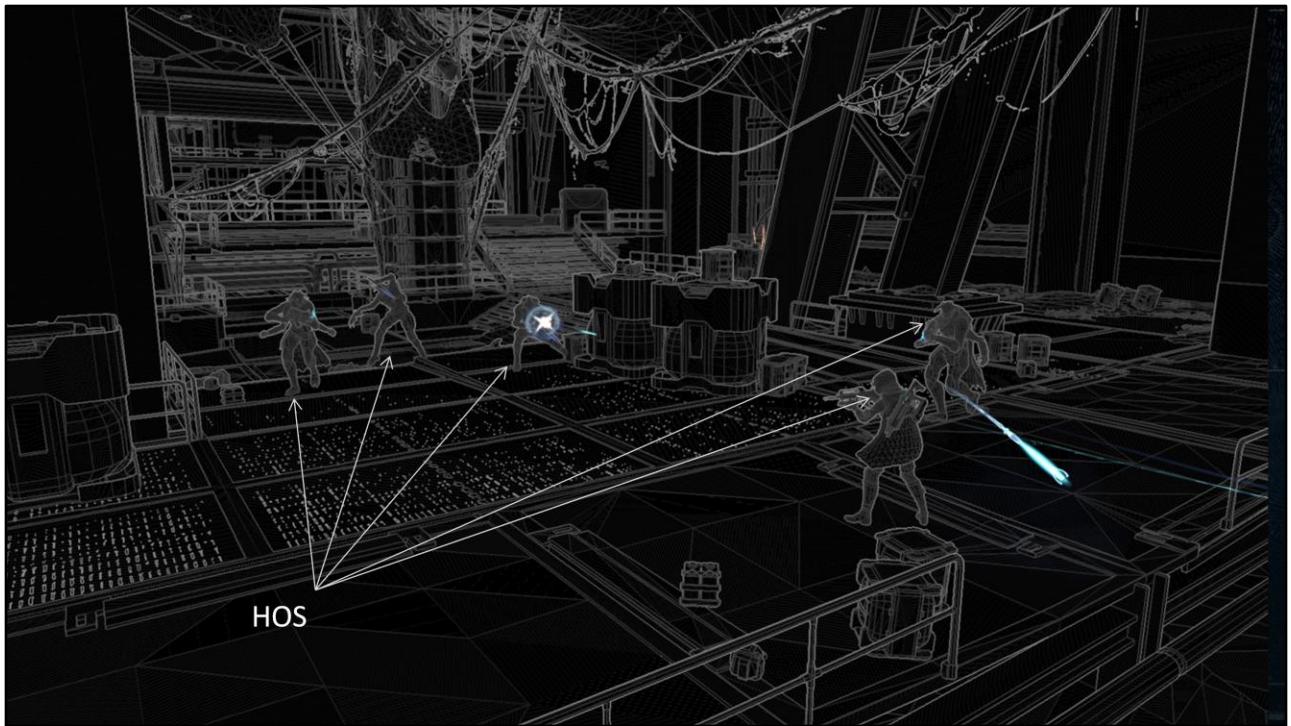
How do we do that? Well, first observation is: we want to **reduce the number of passes** over object and level geometry.

Geometry is expensive (and not just in terms of memory).



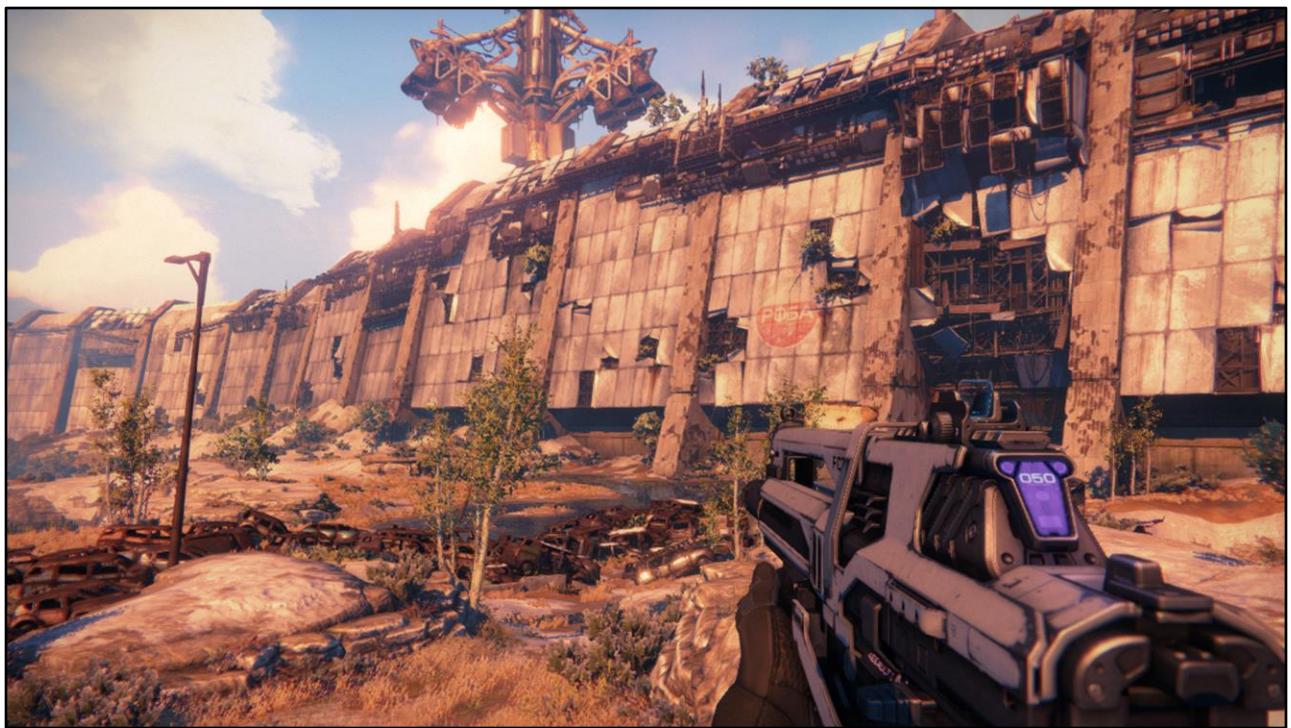
Each geometry pass takes a significant amount of time.

Even if we look at a scene from one of our levels in Destiny, and then break down the geometry density present here...



We'll see that there is quite a bit of complexity in the scene, including **higher order tessellated objects (the characters)**.

Sending this dense geometry several times through the pipeline is very detrimental.



Or another example from our Old Russia level in the game



As you can see we have quite a bit of geometric density in our levels, including very dense foliage and decorators.

At the minimum, your CPU has to marshal the objects, submit to the GPU command buffer, and the GPU has to transform the vertices and fill the pixels.



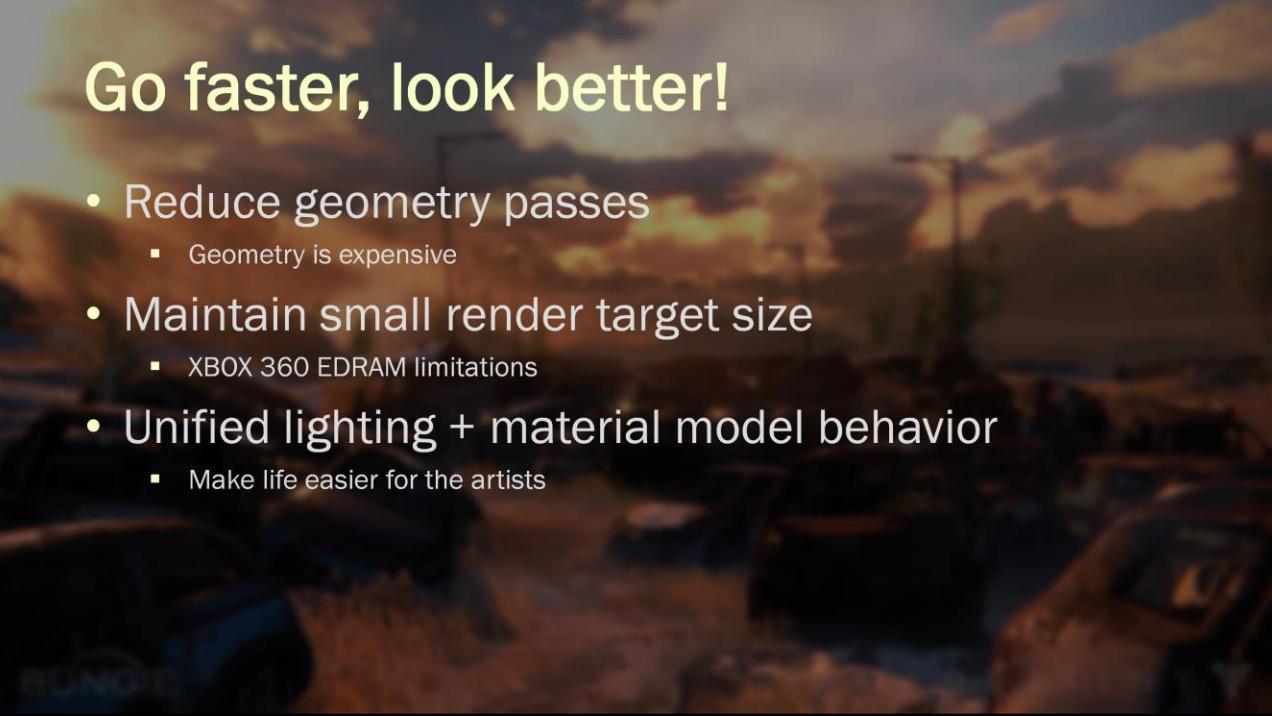
Go faster, look better!

- Reduce geometry passes
 - Geometry is expensive
- Maintain small render target size
 - XBOX 360 EDRAM limitations

Additionally ,since we are shipping on current generation consoles, we also want to keep a small render target size, because the XBOX 360 has very severe EDRAM limitations (10 MB)

This limits how big the buffers you can be rendering to.

When you go over those limits, you have to tile, which increases your latency, and also indirectly increases the number of GPU geometry passes, since you will have to render objects in both tiles.



Go faster, look better!

- Reduce geometry passes
 - Geometry is expensive
- Maintain small render target size
 - XBOX 360 EDRAM limitations
- Unified lighting + material model behavior
 - Make life easier for the artists

We also want to simplify the behavior of the lights + materials.

Halo Reach had 5 different light types, 3 geometry types, and 3 material models, and many combinations did not work well, or at all.

For example, cheap lights on single-pass opaque geometry would use a dark gray diffuse color and had no specular.

On the other hand, re-render lights, widely used in cinematics, did not pick up cubemaps.



Go faster, look better!

- Reduce geometry passes
 - Geometry is expensive
- Maintain small render target size
 - XBOX 360 EDRAM limitations
- Unified lighting + material model behavior
 - Make life easier for the artists
- Simplify shaders
 - Easier to debug + optimize
 - Uses less shader memory

Finally, we want to simplify the shaders, because simple shaders are inherently easier to optimize + debug, and they take up less memory.

So let's go through the various options that we have for rendering approaches

Forward Rendering

Opaque Geometry
Materials, Lights + Shading

Lit Result

The first is standard ‘forward’ rendering.

This uses a single pass over the geometry, in which the shader calculates all of the materials, lighting and shading.

It's about as simple as it gets, conceptually.

Forward Rendering



The shaders on the geometry are basically directly spitting out the final lit result.

This uses the minimum number of geometry passes (1).

But, as soon as a small light gets close to a piece of geometry, you can end up wasting a lot of time calculating it over the entire surface – so there wasted time calculating lights.

Forward Rendering



Lit Result

Additionally, because the single shader does all of the work, you can end up paying for the full render, light + shade cost of a pixel many times over, (because of overdraw during rendering).

That single shader can also get pretty complicated when you start to have a large number of light types and shading options.

Forward Rendering



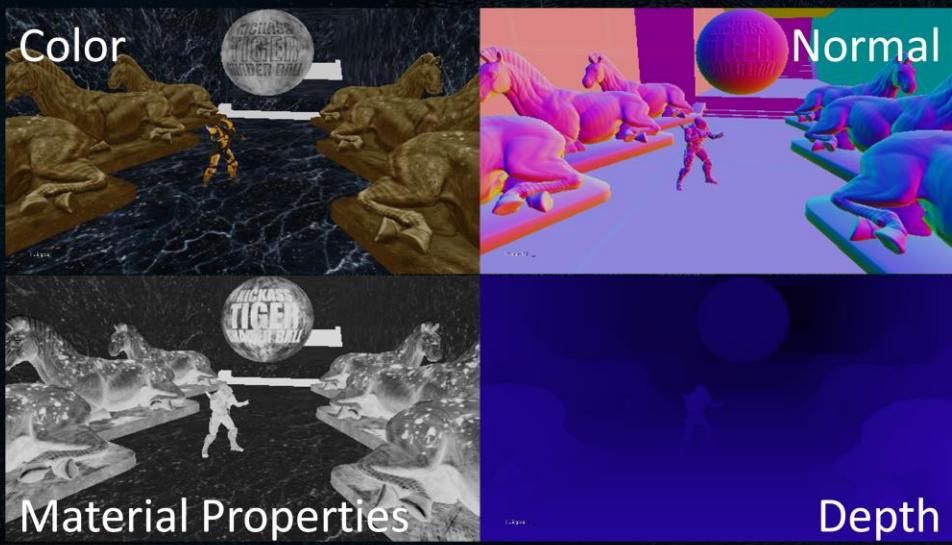
Lit Result

Most engines that use this technique actually create many versions of each shader, to optimize for different numbers and types of lights.

And, inevitably, they have to put restrictions on these numbers to keep the number of shader permutations to a reasonable level.

As a result this approach fell out of style for many recent game engines

Deferred Rendering



The main alternative to forward rendering, is a family of techniques known as deferred rendering.

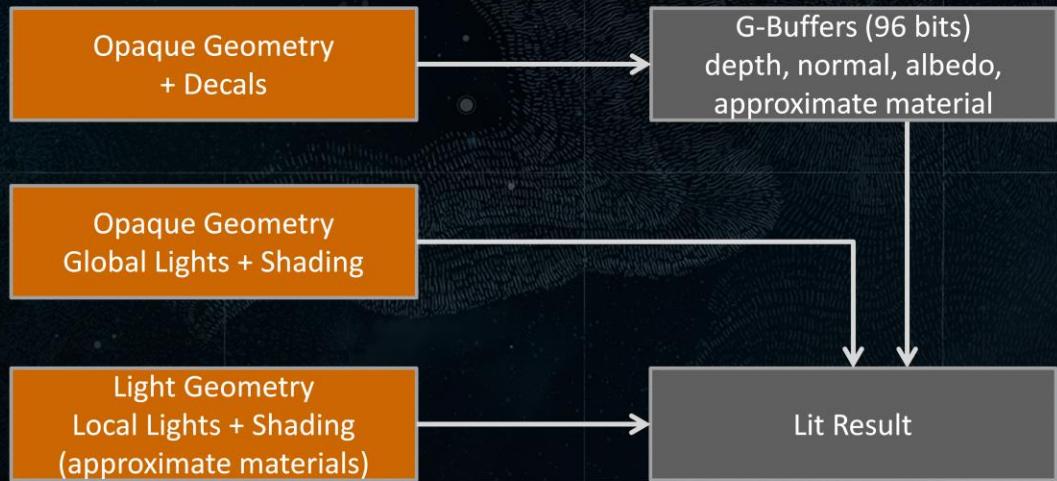
It's deferred because your geometry shaders don't directly spit out the final lit result.

Instead they spit out a description of the surface: for example: depth, color, normal, and material properties.

Then later passes use these buffers to apply the lighting and shading.

Hybrid Deferred Rendering

Halo : Reach

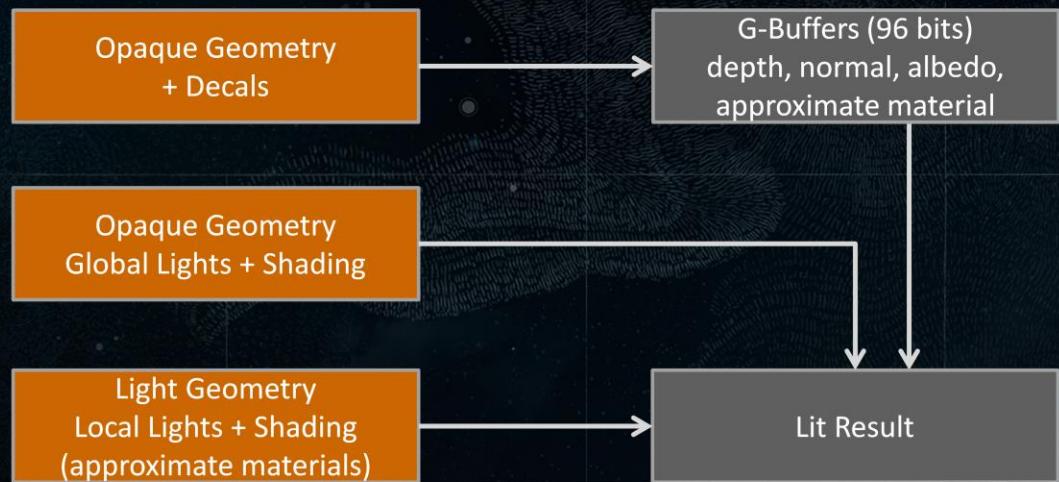


In Halo : Reach we actually used a hybrid deferred rendering technique.

When it was possible, we rendered using the forward technique in the previous slide, especially for vertex-heavy geometry such as decorators and foliage, because we only wanted to run through those vertices once per frame.

Hybrid Deferred Rendering

Halo : Reach



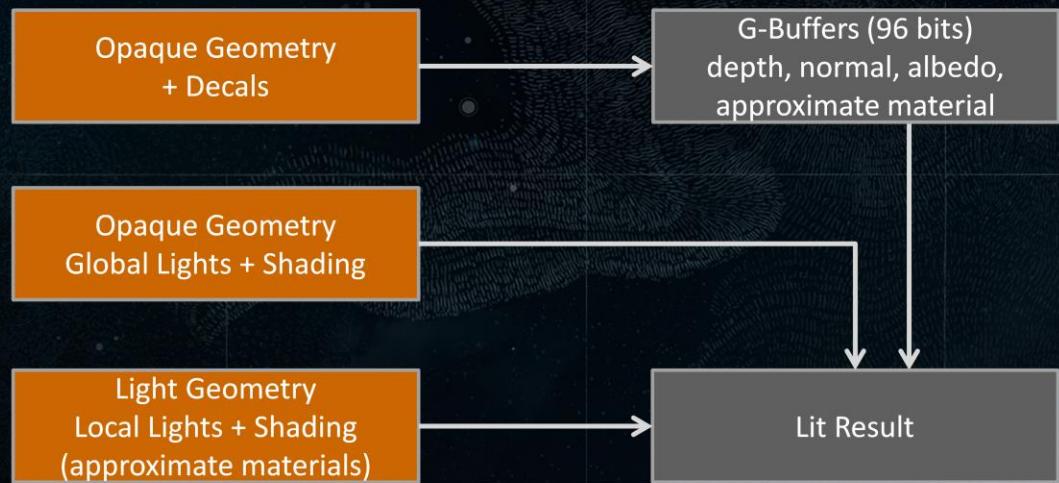
However the majority of the geometry used a deferred rendering approach with two geometry passes.

The first pass over the geometry spit out depth, normals, albedo color and a (very) approximate material model into the g-buffers. And by very approximate, I mean it only used 10 bits total... tiny

We then rendered global lighting, from lightmaps + inline lights, and applied shading in a second pass over the geometry.

Hybrid Deferred Rendering

Halo : Reach



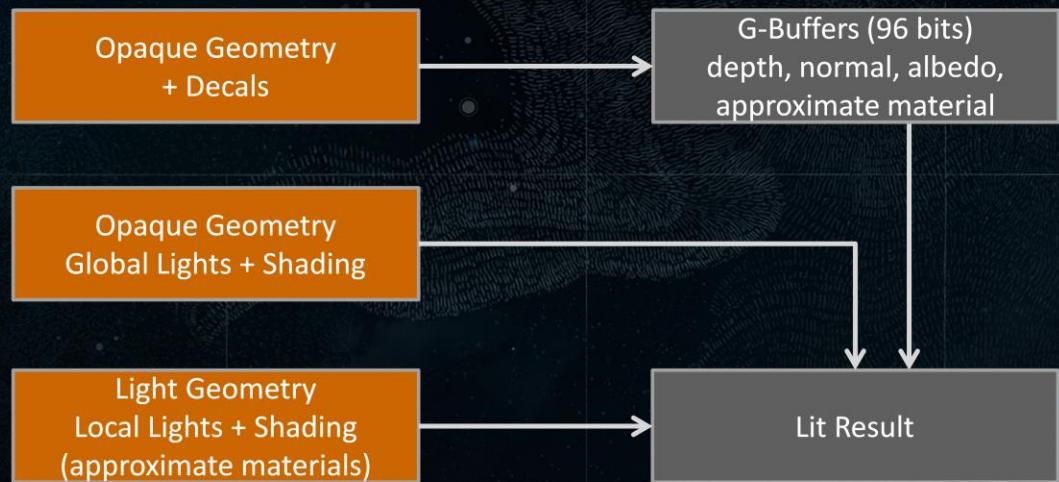
On top of that we added cheap deferred lights that made use of the approximate material model.

The hybrid approach lets us switch between forward and semi-deferred rendering, whichever was most suited or required for a piece of geometry.

The cheap deferred lights let us create hundreds of small point lights, for effects + level highlights.

Hybrid Deferred Rendering

Halo : Reach

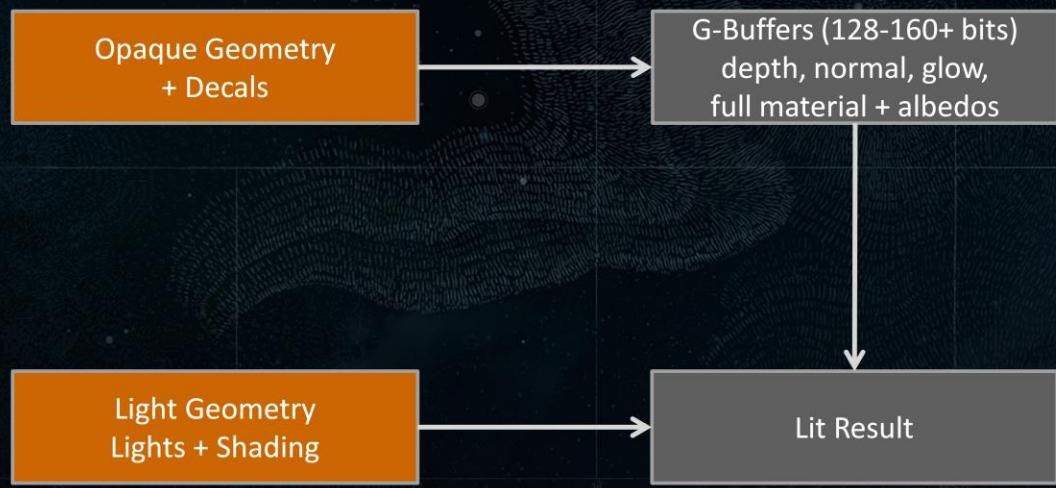


However, as I noted before, the number of permutations of light + geometry types resulted in a massive headache for both the programmers and the artists.

And, on top of that, our deferred method required two passes over geometry, which cost us quite a bit.

The main alternative that other games use, is:

Standard Deferred Rendering

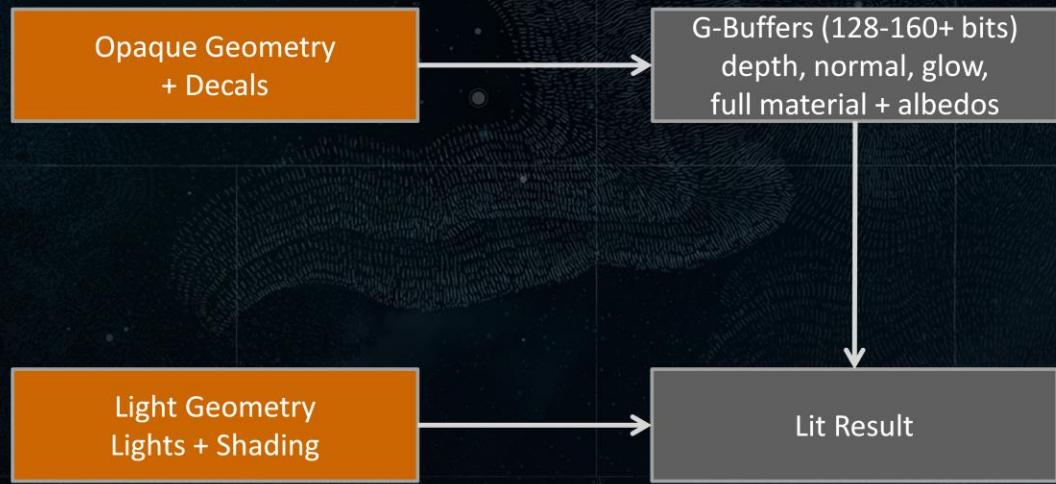


What I'm calling 'standard' deferred rendering.

This method requires only one pass over the geometry to generate g-buffers that describe the **entire** material model.

Then the lighting and shading is done in a separate stage, using the light geometry.

Standard Deferred Rendering

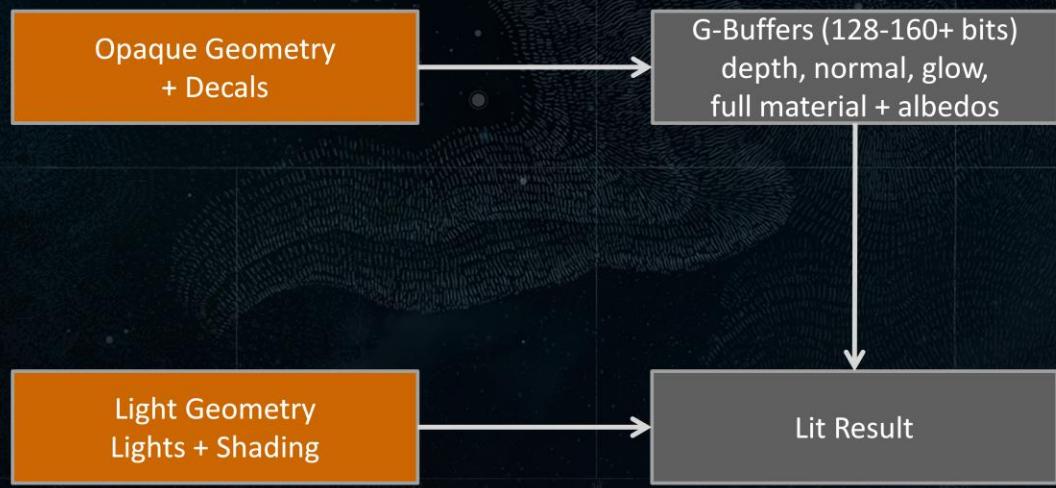


By separating the lighting and shading out of the geometry pass, this approach gains some advantages over forward rendering.

As each shader contains only part of the rendering equation, they are individually simpler, and easier to optimize.

And although you are paying additional pixel fill cost for the multiple passes, that is not often the limiting factor in forward rendering.

Standard Deferred Rendering

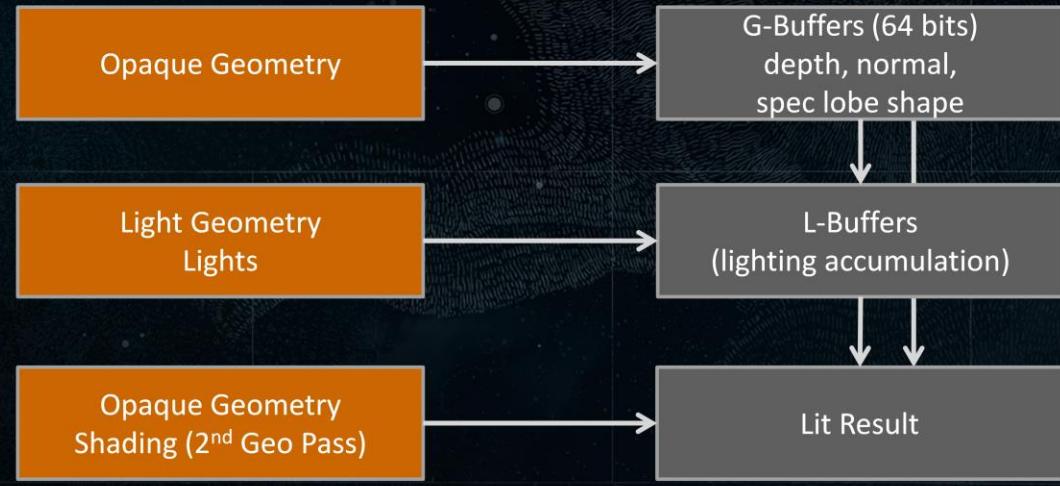


And, most importantly, the cost of a light scales with the number of pixels they touch, not with the number of objects nearby.

The main drawback with standard deferred rendering, however, is that it typically uses extremely wide g-buffers, of up to 160 bits or more.

And these will require multiple tiles on the XBOX 360, with all of the problems that entails.

Pre-pass Deferred Rendering

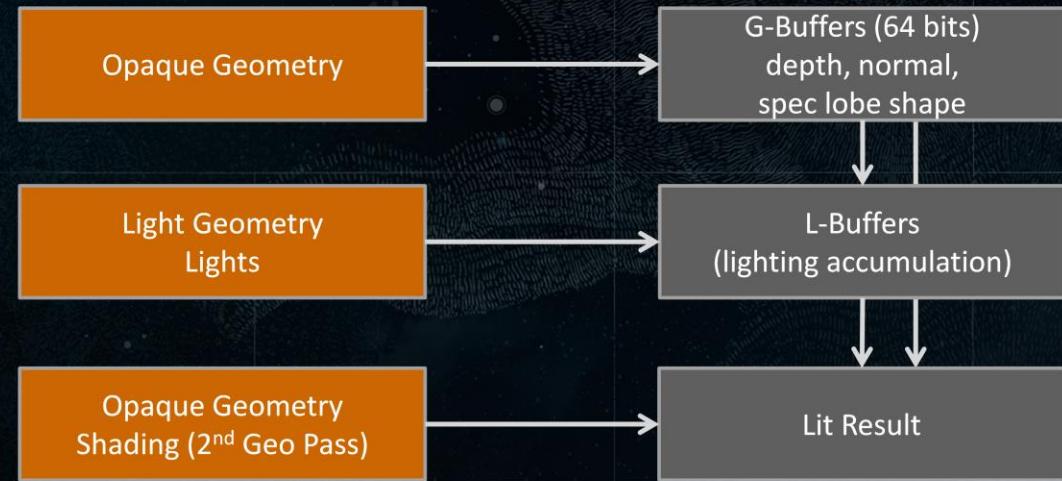


To address the wide buffers, some engines started to use a method called ‘pre-pass’ deferred rendering.

They essentially record a very limited set of data in the first geometry pass – just what is required to do lighting accumulation in the second stage.

The first-stage shaders typically calculate only the normal and specular lobe shape, and the resulting G-buffer is a lean 64 bits, as it doesn’t have to store any color information.

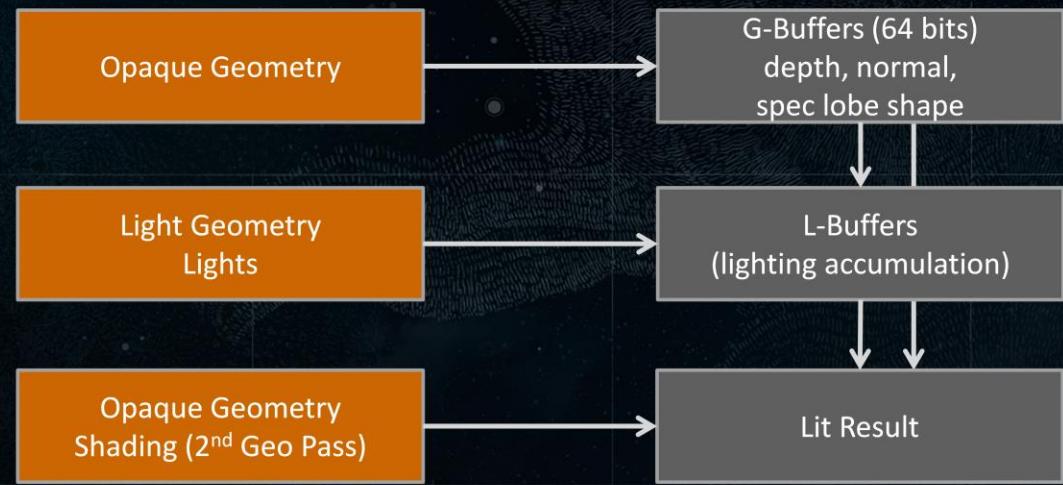
Pre-pass Deferred Rendering



The second stage then accumulates lighting, but it doesn't apply the shading yet.

The shading is applied in the final geometry pass, using the results of the first two passes.

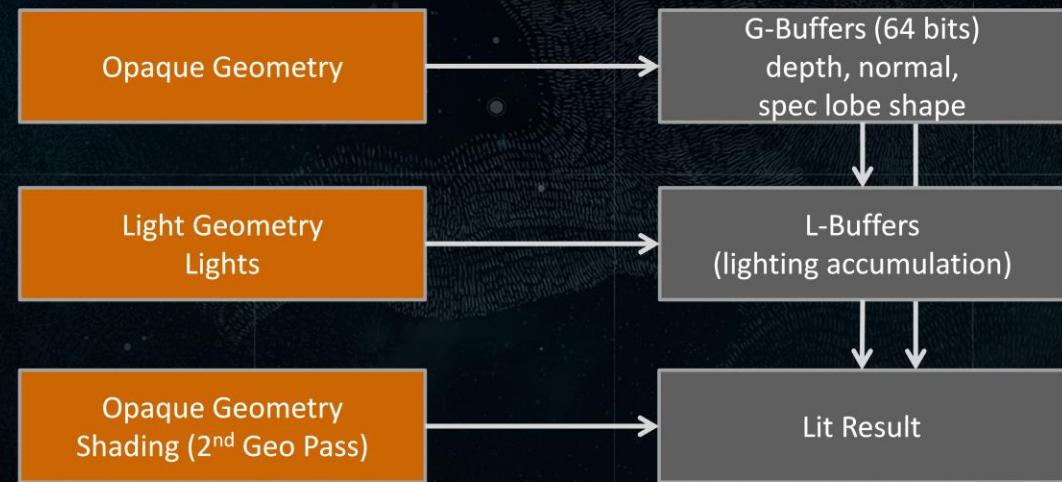
Pre-pass Deferred Rendering



This final geometry pass is nice in that it allows custom shading per object. On the downside, the cost is a **complete second pass** over all geometry in the scene (and there are no exceptions to this, because we need some kind of surface color information!).

(Also, because you no longer have an albedo buffer anywhere, you lose the ability to have cheap decals. They can't just modify that buffer. Essentially they have to do their own lighting and blend on top of the Lit Result, like transparent geometry.)

Pre-pass Deferred Rendering



Overall, we like this approach. But, we really don't want to pay for the full second pass over all the geometry.

Destiny Engine Deferred Rendering



So on to the Destiny solution!

The Destiny deferred rendering method uses a G-buffer of 96-bits per pixel, 50% bigger than pre-pass, but the same size as Halo Reach.

Destiny Engine Deferred Rendering



We stuff all of the surface and material properties we need into these 96 bits. So in this sense it is similar to a highly-compressed standard deferred renderer.

However, we keep the separated lighting and shading passes of the pre-pass deferred renderer.

Destiny Engine Deferred Rendering



The lighting is done exactly the same as in pre-pass deferred rendering, but the shading is done as a full screen pass – every pixel on the screen uses exactly the same material model and shading method.

From the standpoint of CPU setup cost and GPU vertex cost, full-screen shading is a clear win over a second geometry pass.

And, if you really want the custom shading on specific pieces of geometry, like you got with pre-pass deferred rendering, you can still choose to do so, by using a geometry shading pass only on those objects.

Destiny Engine Deferred Rendering



However, with a sufficiently good material model, the vast majority of the geometry should use the default full-screen shading.

(Also, because we have color buffers, we can support cheap decals.)

Destiny G-Buffers

8	8	8	8
Albedo Color RGB			Ambient Occlusion
Normal XYZ * (Biased Specular Smoothness)			Material ID
Depth			Stencil

- **96 bits/pixel**
 - Up to 1200x720 resolution on XBOX 360, no EDRAM tiling

The key to the Destiny approach is the highly compressed and flexible G-Buffer representation.

At 96 bits per pixel, this means we can use a resolution up to 1200 x 720 on the XBOX 360, without any tiling.

Destiny G-Buffers

8	8	8	8
Albedo Color RGB			Ambient Occlusion
Normal XYZ * (Biased Specular Smoothness)			Material ID
Depth			Stencil

- **96 bits/pixel**
 - Up to 1200x720 resolution on XBOX 360, no EDRAM tiling

Here, the albedo color is just a standard 24 bit gamma-encoded color. The normal is encoded using a 2D unwrapping of the sphere, biased to provide more precision towards the camera.

Destiny G-Buffers

8	8	8	8
Albedo Color RGB			Ambient Occlusion
Normal XYZ * (Biased Specular Smoothness)			Material ID
Depth			Stencil

- **96 bits/pixel**
 - Up to 1200x720 resolution on XBOX 360, no EDRAM tiling

Whereas standard deferred rendering may spend between 32 and 64 bits on the material model, here the complete set of material model parameters totals 8 bits, stored in the alpha channel of the normal. Note that we are also able to pack an 8 bit representation for our precomputed ambient occlusion factor, which we apply to our lighting result.

We can get away with such a small representation by using

Material Library

- Material model parameters stored in tables
 - In the G-Buffers we only store a single index into the tables
- Expressive + Customizable
 - Table is stored as a texture
 - Specified using authored curves or painted textures

A material library.

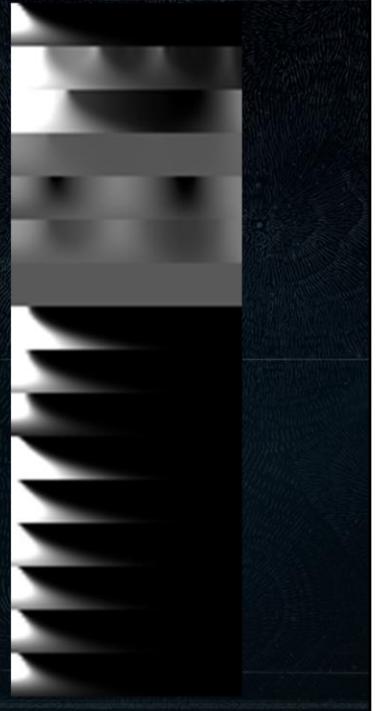
The material library stores material parameters in a table
Then the G-buffers only have to store indices into this table.

This is very expressive in practice..

Because the table is stored as a texture, the material parameters can be specified by (technical) artists using authored curves or painted textures.

Specular Lobe ID (10 bits)

- Controls the shape of the specular highlight
- 4 bits → Shape
 - Painted by artist
- 6 bits → Roughness
 - Calculated by an offline preprocessing task



The specular lobe index (10 bits stored next to the normal)

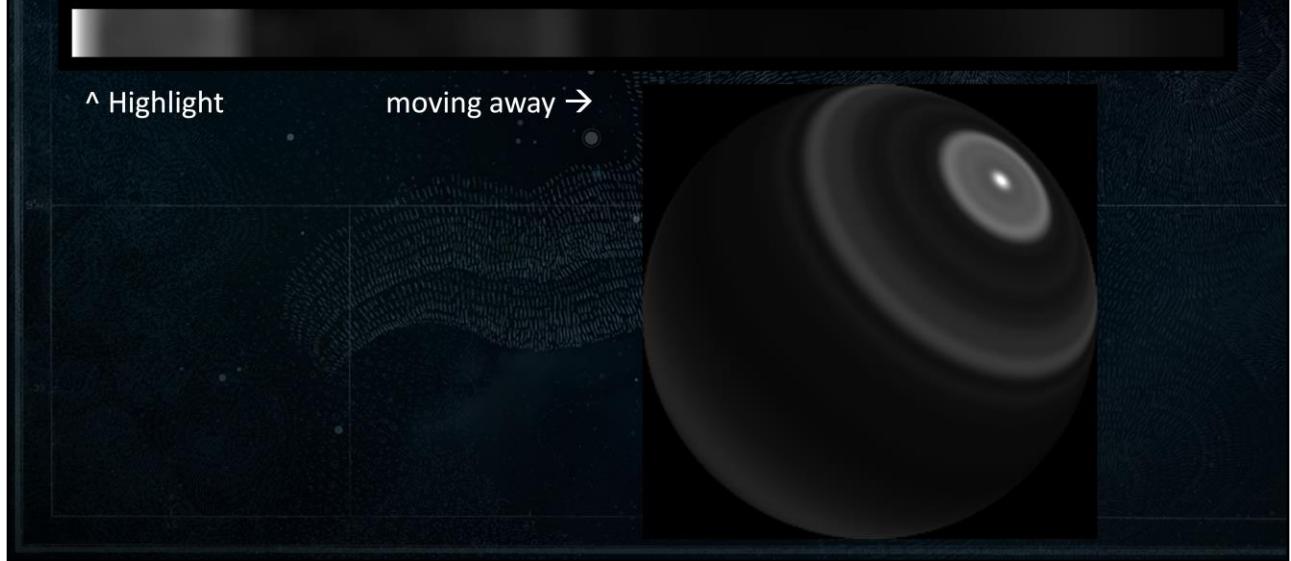
Controls the shape of the specular highlight.

4 bits specify the lobe shape, which the artists paint.

And 6 bits specify the roughness variations, which are automatically calculated during import.

So our final lookup table texture looks like this.

Lobe ID controls ‘Shape’

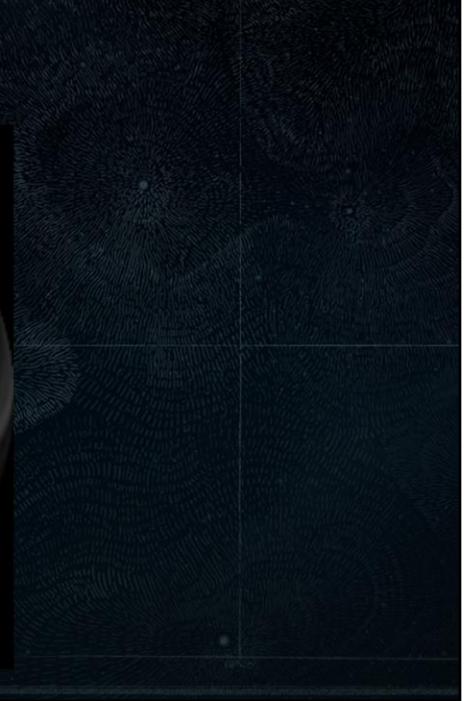
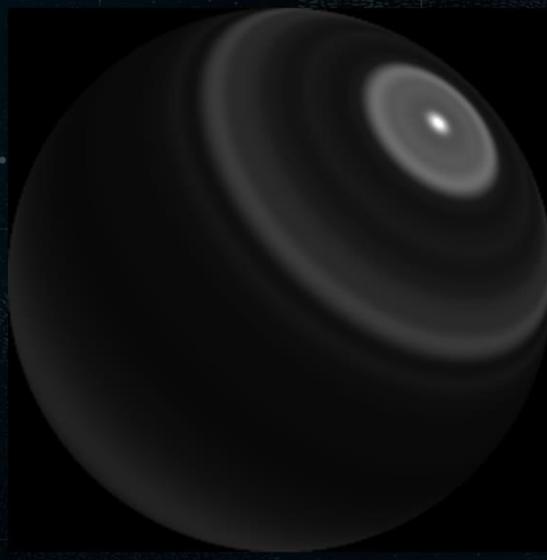


Here's how a single line of that texture is interpreted.

The left side represents the center of the specular highlight.

And as we move to the right, it describes the falloff away from the specular highlight.

Roughness Variations



The automatic roughness variations are calculated to preserve total energy over the sphere. This is the computation that is done during our content's offline preprocessing (aka 'import' step as we call it)

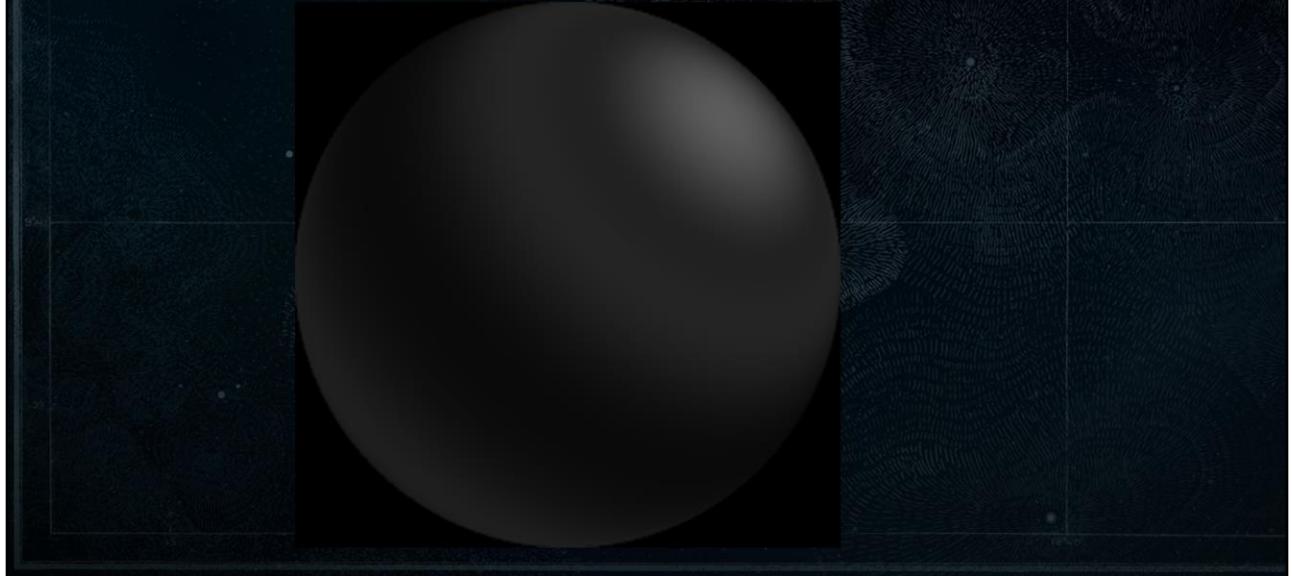
Roughness Variations



Roughness Variations



Roughness Variations



Roughness Variations



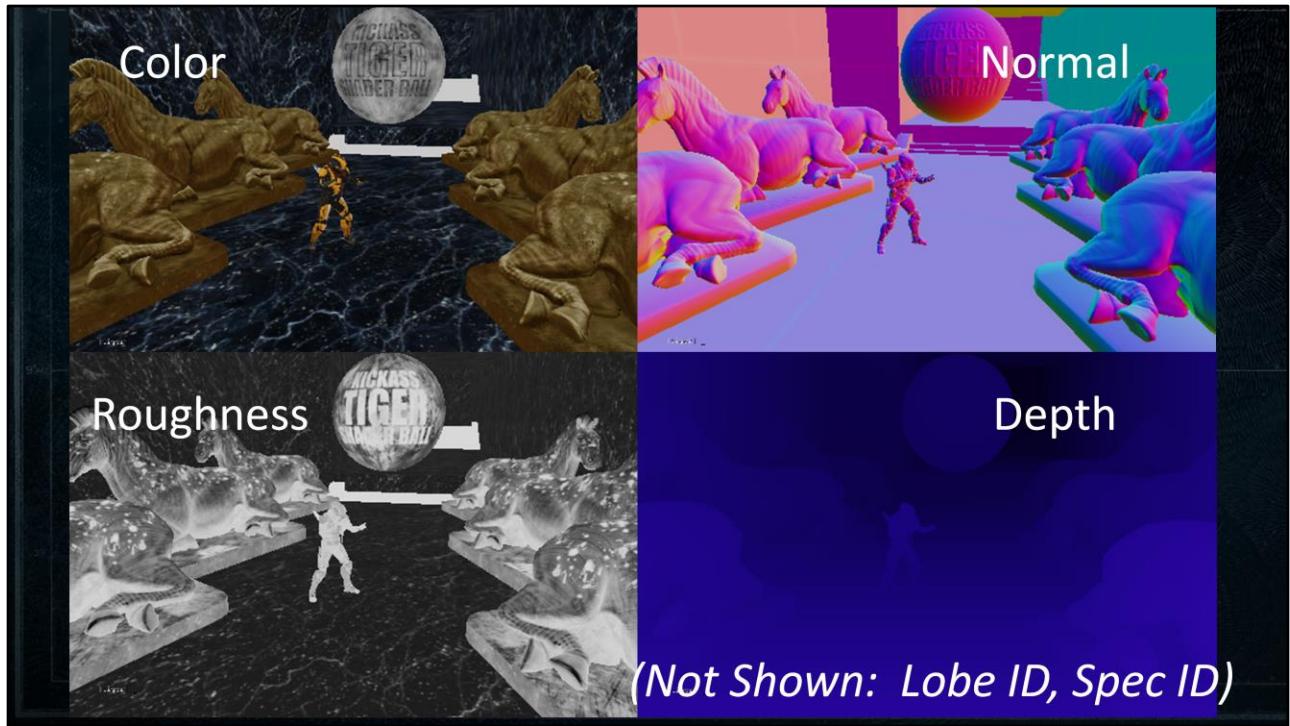
Specular Tint ID (8 bits)

- Controls the color of the specular highlight
- $\text{RGBA} = \text{Texture Lookup}$
 - $X = \text{dot}(N, V)$
 - $Y = \text{spec_tint_ID}$
- Specular color is a transform of albedo
 - Currently: $\text{albedo} * A + \text{RGB}$

The specular tint ID controls specular color.

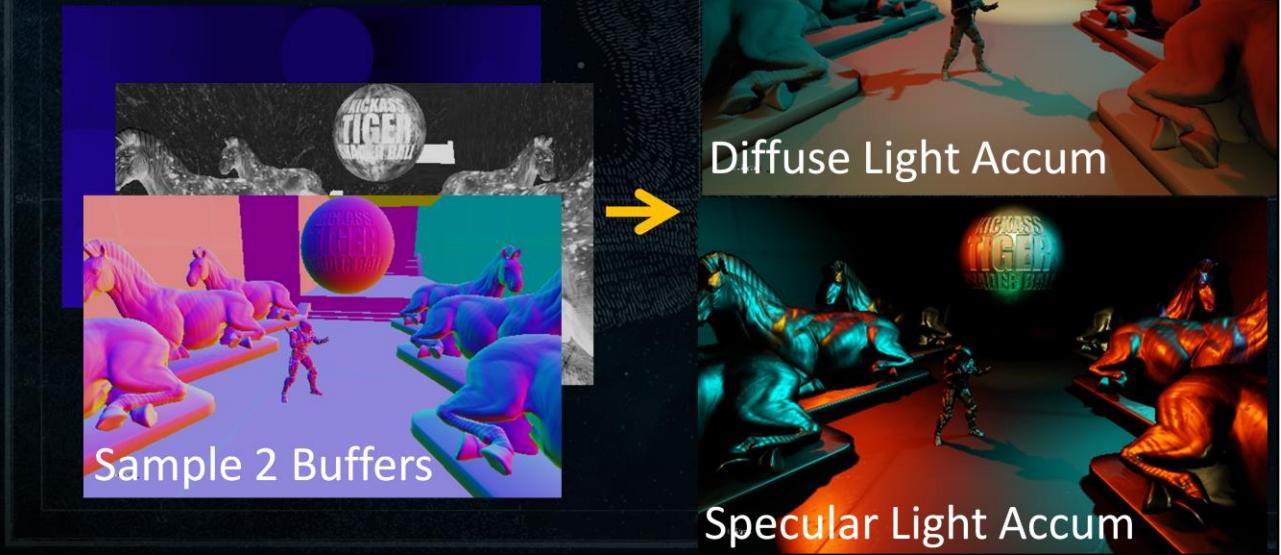
It is essentially a transform on the albedo color.

Although separately these parameters are totally 18 bits, we remap them using a look up table to squeeze them into an 8 bit representation



So, to tell the story in pictures, we start with our G-buffers, rendered using a geometry pass.

Lighting



The we calculate lighting by rendering light geometry, and accumulating into diffuse and specular light accumulation buffers.

Note that because of the way we packed the g-buffers, we only have to sample two of the g-buffer textures to get all the parameters needed for lighting:

(depth, normal + spec lobe shape)

Diffuse Light Accum



Diffuse Color

Shading

Diffuse Lighting



Finally we apply shading, which is essentially calculated like this:

Specular Light Accum



*



Specular Color

Shading

Specular Lighting



Diffuse Lighting



+



Specular Lighting

Shading

Lit Result



And the finally the diffuse and specular lighting are added to get the final result.

Overview

	Geom. Passes	Bits/Pixel	Unified Model	Simple Shader
Forward	1	32	Y	----
Hybrid	1-2	96	N	---
Standard	1	128-160+	Y	+
Pre-pass	2	64	Y	++
Destiny	1	96	Y	++

So, going back to the original list of goals, here's how each of the approaches stacks up:

With our approach,

Memory footprint fits in EDRAM (96 bpp)

Single pass over geometry (especially important for decorators / foliage)

Halo)

Allows cheap deferred decals

Complex material appearance

Separate lighting / shading / geometry shaders simplifies shaders

Deferred... Tiled, Clustered, etc.?

- Note that this talk is *only about the shading part* of our deferred pipeline
- The lighting is part of a separate talk, and outside of scope of this equation
 - But don't assume vanilla deferred lighting application

Extending Deferred

- Many shading models require information about lights *while computing shading*
 - Anisotropic, subsurface scattering, etc.
- Integrating those to deferred renderer can be a challenge
 - Since lighting information may not be known unless going to über-shader

Of course, even with this flexible model, we still found the need to extend it to support additional shading models as many of those required additional information about lights during their shading phase.

But remember the limitations of the forward-rendered I mentioned earlier? Forward-rendered objects would need to have separate lighting environment setup, and they typically do not receive all regular ‘deferred’ lights. This causes forward-rendered objects to not ‘sit well’ in the regular environment - *NOT DESIRED AT ALL*

And, of course, it increases shader permutation matrix - “special” shaders complicate art workflows, pain to maintain for us, and frustrating to optimize

Extensions to *Destiny*'s Deferred Model

- Specular response is affected by diffuse (albedo) color
- Cubemaps can modify diffuse color and specular roughness
 - Cheap evaluation during G-buffer pass
 - Evaluated for full lighting environment
 - Shadows properly affect all cubemaps

In our material model, specular response can be affected by diffuse (albedo) color – this is controlled by setting specular tint to be driven from albedo color as a material choice. This simple decision allows us to extend *Destiny*'s deferred material model to include either additional shading models or make some of the operations cheaper to execute.

One nice example of that is that cubemaps can modify diffuse color and specular roughness *before we output those to g-buffer*

Cheap evaluation during G-buffer pass – causes cubemaps to be lit by full lighting environment

Incorporating Subsurface Scattering



Another example was integrating subsurface scattering in our engine. For implementing subsurface scattering, we modify the diffuse lighting buffer directly in screenspace (as a form of post-processing).



Here is a different shot of the same character in a simplified lighting environment and no anisotropic for hair (just to break down the subsurface)

In this shot, we don't apply any subsurface scattering on the lighting and **you can see the harsh shadows on the character's face**



Here's the diffuse lighting buffer for that shot



Next, we perform a custom subsurface scattering in a screenspace pass on the diffuse lighting accumulation buffer to compute skin diffusion.

To optimize rendering in this case, we render a skin pre-pass, which generates a stencil mask for pixels that need to have subsurface diffusion pass enabled, as well as output diffusion parameters

The skin prepass' stencil ensures that we can engage Hi Stencil reject for the diffusion pass, only causing it to work on pixels that need it.

Then we run a screenspace pass on that buffer, sampling the diffuse lighting accumulation buffer, and blurring it using a custom subsurface diffusion blur. The results are written out to the output diffuse lighting buffer (as a new target) and then sent through our regular shading pipeline (including shadows application for all shadows).



Which results in a softened lighting for this character. We had to come up with a subsurface scattering model that would support alien characters such as the crow with custom diffusion profiles that are artist-authored with variety of subsurface control knobs.

We actually started by performing the diffusion pass for both diffuse lobe and specular lobe accumulation buffers but we found that doing that destroyed the specular response for the surface (we use the Kelemen specular model for these) as it overblurred the energy response. So instead, we performed diffusion step only on diffuse lobe accumulation buffer and kept specular response unperturbed, which is actually more physically accurate, since the oily specular layer doesn't get strongly affected by subsurface response.



And here's the comparison again without subsurface
And ..



With subsurface in the final shot

Performing our subsurface scattering on the diffuse lobe accumulation buffers ensures that all subsurface scattered objects are not forward-lit with special lighting, but, rather have full integrated lighting environment.

We actually enable subsurface scattering on variety of materials, including skin (as shown in this image), but also hair (to get blond highlights showing through), and other materials. You'll see more examples of those effects in the near months as we release more of our character material into the wild.

But What About Anisotropy?

Another extension that we pursued was adding anisotropy support.

Anisotropy and Deferred Rendering

- Anisotropy requires light source(s) information during shading
- Thus typically rendered as forward-lit (with the usual pain)
- But without anisotropy, we can't have good hair!
 - And many other interesting materials

Traditionally, Anisotropy and Deferred Rendering Were Poor Friends

Extending deferred renderer to support anisotropic models can be a challenge – since you need information about the actual light source at the point of shading which is typically not available, most anisotropic effects become forward rendered.

Additionally, typical deferred paths don't typically support anisotropic lighting model in their uber-shader pass.

However, for our game, having good anisotropic materials that were integrated with our lighting model for all characters in our sandbox game (which can be quite a large amount of the screen) was very important, especially because we had characters with hair that, though stylized, needed a good anisotropic response to produce believable material

Destiny Deferred ♥ Anisotropy



To sort out this issue, we built upon the previous observation that we've been running with of “specular color is derived from albedo color”, and integrated anisotropic highlight computation into G-buffer pass

Anisotropic direction was driven by the dominant light, but the overall lighting for the anisotropic material is affected by the full lighting environment.

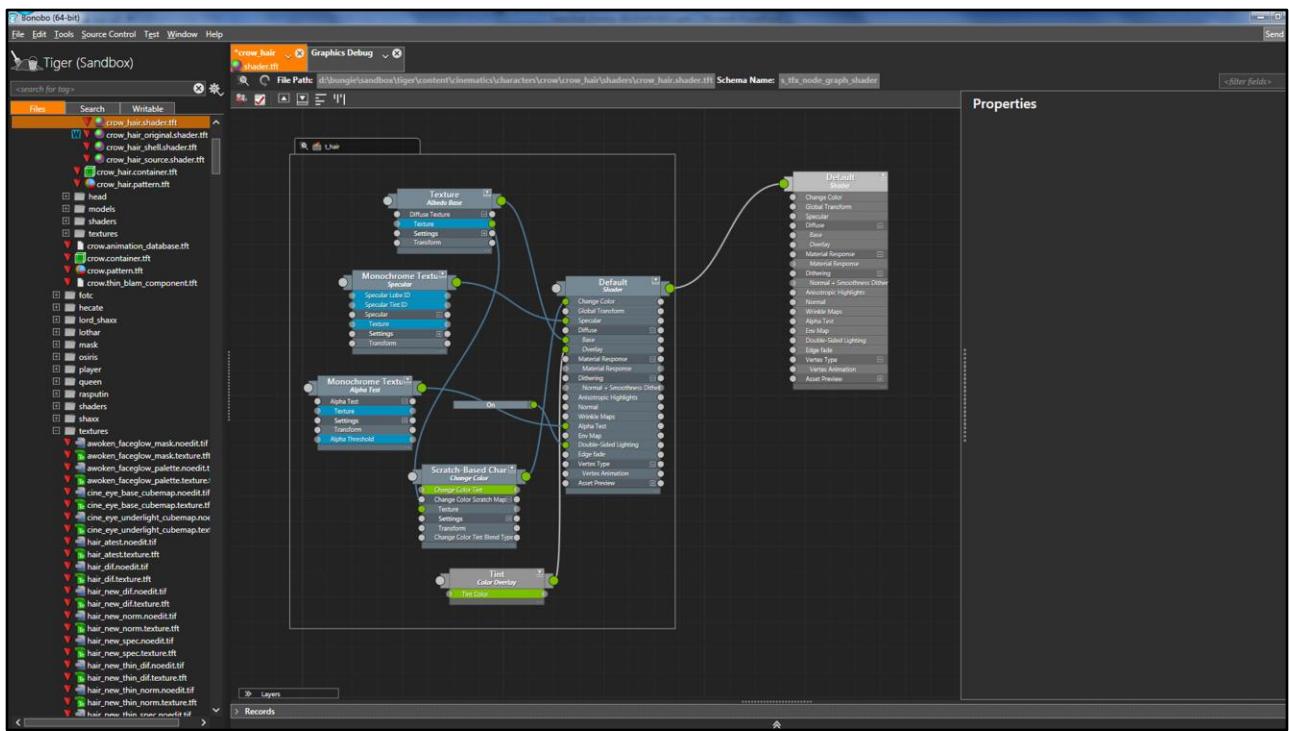
Destiny Deferred ♥ Anisotropy



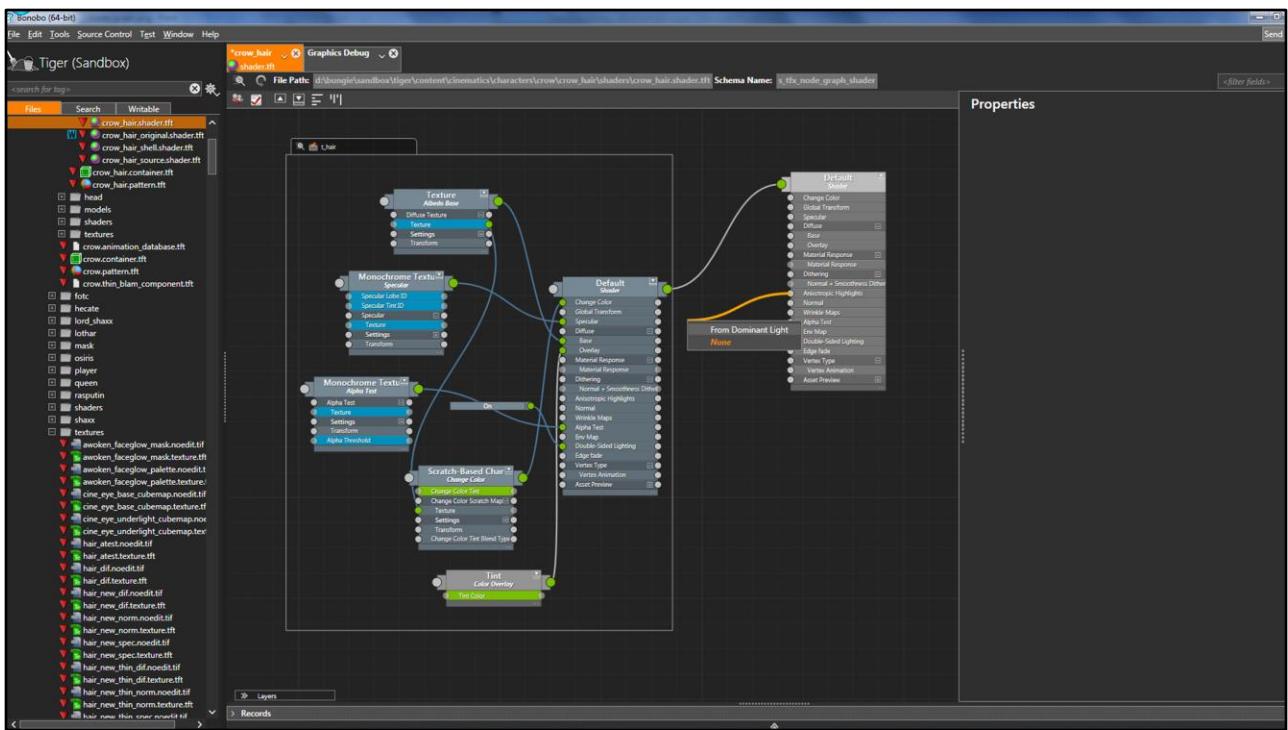
In this context, a *dominant* light source is either the sun or a custom dominant light source auto-determined per character or object with anisotropic shader (such as indoors or in cinematics). For example, for the latter case, we export a channel from a shader (which is a data driven input for our shaders) and during cinematics Maya export, we run a script which, based on light bounds, intensity and other parameters, determines which light is going to be dominant for each individual object. This parameter is then animated along with the rest of properties and exported out with our animated data for each object.

Note that we can smoothly switch that parameter from one light to the next across shot boundaries in all scenarios, as well as blend between different types of dominant light sources.

So an artist just selects an anisotropic component in the shader graph (selectable for any regular shader) – in Destiny, we support node-based shader graph editing, which is built on custom shader component blocks, and then compiled to HLSL.



Here we have an example of our internal node-graph-based shader editor tool and one of the shaders for character's hair



To add anisotropic, the artist would simply drag out the ‘anisotropic highlight’ component and setup its parameters – other than that, this is a our regular deferred shader

This is great since it means that:

No nasty shader permutations

And with this ability, the artists are also often combining anisotropic highlight component that with subsurface scattering and other effects – all rendered within the deferred renderer’s main pipeline

Destiny Deferred ♥ Anisotropy



We can compute the full anisotropic model during our g-buffer pass (when that particular material is known). Then we simply modify albedo color and specular roughness that we output for each pixel to the g-buffer to include anisotropic highlight.

For example, for hair we used a custom-modified simplified Marshner-based model. For anisotropic we drive our specular tint color from the diffuse color, and thus generate an anisotropic highlight in the resulted shaded result. What we have found is that since specular color is driven from albedo, we do not add this response to the specular lobe accumulation in order to preserve the overall energy of the response (and not cause blow outs).



And here's the example from the slide before – here we used anisotropy on our character's, Crow, hair. What you can see here is that the result is well-integrated into all lighting types and full shadows [Note that I punched up the brightness and contrast to make this visible on the projector]

And most importantly results in plausible anisotropic specular response

An interesting observation that drove this integration: even though we computed anisotropy response only for the dominant light, even in scenes when there were multiple light sources, the result “read” correctly. Turns out people distinguish the presence of anisotropy, but not necessarily correctly determine direction of anisotropy for all light sources present. (Of course, we graphics folks will sense that something isn't quite right, but it works in this case). SHIP IT.

Transparents and Deferred



In Destiny, we have a great deal of transparent objects being rendered. Traditionally, Bungie games have always had very strong effects rendering (see Chris Tchou's Halo Reach Effects Tech presentation from GDC 2011) for a lot of the details about our Halo engine's effects pipeline). However, one of the biggest challenges with transparent objects in a deferred renderer is getting them to be lit consistently with the rest of the environment.

Destiny Transparents Lighting Goals

- Consistency of lighting with opaque
- Consistency of shadowing with opaque
- Atmosphere application



For Destiny, we made our explicit goal to make transparents appear to sit in the same lighting environment as the opaque objects

This meant:

*All shadowing lights cast
shadows onto transparents*

Atmosphere is applied

*Full Lighting environment is
applied in a consistent manner
with the opaque objects*

Destiny Transparent Lighting Approach

- Dynamically place light probes where transparents are
- Each frame build a low-order spherical harmonic for that probe
 - Take lights and shadows into account

High level summary of how we approach transparent lighting in Destiny: we light transparents by placing light probes where the transparents are and, each frame, building a low-order spherical harmonic that takes into account lights and shadows.

Hm... That Sounds Familiar...

- Irradiance Volumes have been doing this for ages... on CPU
 - Requires a lot of caching and time slicing to improve performance for current gen
- We are computing on GPU with a compute-friendly approach
 - Current and next-gen console oriented

If this sounds familiar, you may have read an old presentation I've done at GDC Europe 2005

(http://developer.amd.com/wordpress/media/2012/10/Tatarchuk_Irradiance_Volumes.pdf) on Irradiance Volumes.

However,

What's different than previous approaches we've used before is that instead of doing this on the CPU, which is slow and requires a lot of caching/time-slicing to improve performance, we do it on the GPU with a compute-like approach that's friendly to present-gen consoles.

Per Frame: Build Probe List

- When processing visible transparent objects, CPU builds a list of light probe points
- Points are written to a fast thread-safe lock-free buffer
 - We build object lists in jobs, so this is important
- XYZW where each component is a 32-bit float
 - Light probe radius is stored in W

Each frame:

during multithreaded processing of visible transparent render entities, the CPU builds a list of light probe points

points are written to a fast threadsafe lock-free buffer

XYZW where each component is a 32-bit float; light probe radius is stored in W

Per Frame: Submit Light Probe Buffer to GPU

- Limit number of probes to 1024
- Encoded as 64 x 16 RGBA32F texture
- Double-buffered to prevent stalls

CPU sends the light probe point buffer to the GPU
we limit the number of points to 1024
this is encoded as a 64x16 rgba32f texture
double buffered to prevent stalls

Light Probe GPU Generation

- Setup MRT with SH light environment surfaces
 - 3 64 x 16 RGBA16F render targets
 - Each render target encodes 4 SH coefficients for a color channel
- Render a quad to the SH surfaces for sunlight
 - Map directional light to SH coefficients

Setup MRT with spherical harmonic light environment surfaces
three 64x16 rgba16f render targets
each render target encodes 4 SH coefficients for a color channel
render a quad to the SH surfaces for sunlight
map directional light to SH coefficients

Light Probe GPU Generation: Shadows

- Sample cascade shadow maps to determine shadowing per probe
 - PCF at the light point buffer position
 - Use a sample radius based on light probe radius

sample shadow cascades to determine shadowing

sample cascades with PCF at the light probe point buffer position using a sample radius determined by the light probe's radius

Light Probe GPU Generation: Light Environment

- Lights can be tagged as affecting transparents by artists
 - Our lights are just a set of shader components selectable by artists
- For each transparent-affecting light
 - Render a quad to the SH surfaces
 - Map the light's parameters to SH coefficients given the light probe point buffer position

for each light in the main view that is tagged as affecting transparents, render a quad to the SH surfaces

map the light's parameters to SH coefficients given the light probe point buffer position

Rendering Transparents

- Transparents are rendered post main deferred passes
 - After deferred lighting and shadowing has been applied

Rendering Transparents

- When rendering a transparent object
 - Sample the SH surfaces
 - Evaluate the SH model given per pixel normal
 - Apply the lighting to the pixel color
 - Victory!
- Our ambient lighting model is cheap and gets computed afterward

when rendering the transparent

sample the SH surfaces

evaluate the SH model given the per pixel normal

apply the lighting to the pixel color

our ambient lighting model is cheap and gets computed afterward

Performance

- Apply all visible transparent-affecting lights to all light probes every frame (for simplicity)
 - Seems wasteful, but GPUs are great at exactly this type of parallelism
- Xbox 360 Performance:
 - Applying sunlight to all light probes: 3 μ s per cascade
 - Applying each light: ~ under 3 μ s
 - Total cost for a scene with 1024 light probes and 50 lights: ~0.15 ms

Perf:

This approach applies all visible transparent-affecting lights to all light probes each frame. That seems wasteful, but GPUs are really good at this kind of parallelism.

Numbers on xbox360:

applying sunlight to all light probes: about 3 μ s per cascade

applying each light: a little under 3 μ s

total cost for a scene with 1024 light probes and 50 lights: ~0.15 ms

Results

- Transparents get lit by directional sunlight
 - Including specular
- Transparents get shadowed
 - CAVEAT: A single shadow factor for the whole transparent object, *not per-pixel shadows*
- Transparents get lit by static and dynamic point lights

Limitations

- SH is not a perfect match for opaque lighting
 - But plausible enough
 - When the object is transparent, most artifacts are not noticeable

Issues:

SH basis is not a perfect match for opaque lighting

on the plus side, it's plausible enough that you don't notice it when the objects is transparent anyway

Off-screen Light Probes and On-Screen Objects

- Need to inflate light bounds
 - Not just the strictly visible set
- Careful sampling of cascade shadow maps
 - Since those are tightly bound to the view frustum

what to do when the light probe position is off-screen but the object is partially on-screen?

we have to include more lights than just the strictly visible set (inflate light bounds)

we have to be careful about sampling from cascade shadow maps that are tightly bound to the view frustum

issue gets worse the larger the object is due to our approximation of the objects as a point—so we only light small transparent objects, or use separate light probes for different parts of large objects

Limitations

- SH is not a perfect match for opaque lighting
 - But plausible enough
 - When the object is transparent, most artifacts are not noticeable
- Need many samples to get a smooth shadow response
 - Due to per object shadow factor
 - On the plus side: Operating on a 64 x 16 buffer is fast

since the shadow factor is per object we must take a ton of samples to get a smooth response

on the plus side, we operate on a 64x16 buffer so even with a ton of samples, it's not too much time

what to do when the light probe position is off-screen but the object is partially on-screen?

we have to include more lights than just the strictly visible set (inflate light bounds)

we have to be careful about sampling from cascade shadow maps that are tightly bound to the view frustum

issue gets worse the larger the object is due to our approximation of the objects as a point– so we only light small transparent objects, or use separate light probes for different parts of large objects

Performance

- All features must exist within the context of performance
 - CPU / GPU time → direct impact on input latency
- No matter how stellar a technique may be, it must fit into its allotted budget
 - Individual budgets can be 'stretched' in a given frame, but not all budgets at the same time
- After all the game is dynamic, and we can't guarantee that you won't have a giant battle in a scene with many buildings and many trees and many effects and ...

First and foremost we are shipping a game. And that means that

All features must exist within the context of performance

In our case, we are an FPS game, which means that input latency (time from player moving a stick on the controller to them seeing the results on the screen) has to be very low for us to be a responsive game).

And speaking about performance...

- Brings us to the subject of particles

And speaking about performance...that neatly brings us to the subject of particles



Destiny is an effects-heavy game

Particles rendering take up a significant chunk of our GPU budget

Lots of overdraw

Thus particle rendering remained one of the top performance-critical part of our rendering for the game

Cheaper Particles Rendering

- Added low-resolution transparents for *Halo: Reach*
 - $\frac{1}{2} \times \frac{1}{2}$ size
- Artists tagged transparents as low-res
 - For smooth visual effects like smoke, fog, etc.

For *Halo: Reach*, which was also extremely effect-heavy and rich, we added a solution for using $\frac{1}{2}$ res transparents rendering for better performance
Artists could tag specific particle systems or other transparent objects as low resolution – typically used on low frequency effects like smoke, fog, and so forth

Cheaper Particles Rendering

- At runtime, we dynamically bucketed transparent objects into high and low res buckets
 - High res were individual objects
 - Low res buckets contained low resolution transparents bucketed by depth
- Sort the buckets at runtime to minimize the state cost of switching (blend and upres) per frame

At runtime, we would dynamically bucket transparents either as individual objects for high resolution transparents, or into a small set of depth-based buckets for low resolution transparents. We would then sort the low resolution buckets as individual objects along with the high res transparents (we used constant sort using a min max heap). This allowed us to minimize the cost of Hi / Low blend state on Xbox 360 as well as the costly upres.



For *Destiny*, even that wasn't enough... We needed a more performant solution so that we can get all the particles that our effect artists wanted to put, along with the corresponding overdraw, and get them to render fast on current gen platforms

Über-Low Resolution VDM Particles

- Particles that render at $\frac{1}{4} \times \frac{1}{4}$ resolution
- Use a special extra *VDM* (*variance depth map*)
 - Enables them to composite seamlessly with higher resolution depth buffers

Our solution was to create a new method which we titled.... Über-Low Res Particles with variance-depth compositing!

Uber-low rez particles are particles that render at $1/4 \times 1/4$ resolution, with a special extra variance depth map that enables them to composite seamlessly with higher resolution depth buffers.

Compute Über-Low-Res Depth Buffer

- Compute a $\frac{1}{4} \times \frac{1}{4}$ resolution conservative depth buffer
- Each pixel records the farthest depth from the camera
 - For the corresponding 4×4 block of original pixels
- Use this depth buffer for Z culling – *and more importantly, early-Z culling*

We first compute a $1/4 \times 1/4$ resolution conservative depth buffer (each pixel records the farthest depth from the camera for the corresponding 4×4 block of original pixels). We can use this as our depth buffer for Z-culling (and more importantly, early-Z culling).

Rendering Über-Low-Res Particles

- Setup 2 render targets: color and depth transition buffers
 - Assuming a 1280 x 720 buffer

	w	h	format	initial
color (pre-multiplied)	320	180	a8:r8:g8:b8	rgb:0.0 a:1.0
depth transition	320	180	r16:g16	rg:0.0

Then we set up two render targets: (assuming 1280x720 front buffer)

The **color buffer** records the appearance of the pixel (the composite operation to apply it to the high res frame) at the corresponding conservative depth.

Not all of the high resolution pixels are at the conservative depth, however... so we also record how that pixels appearance transitions towards 100% transparency as the depth moves towards the camera.

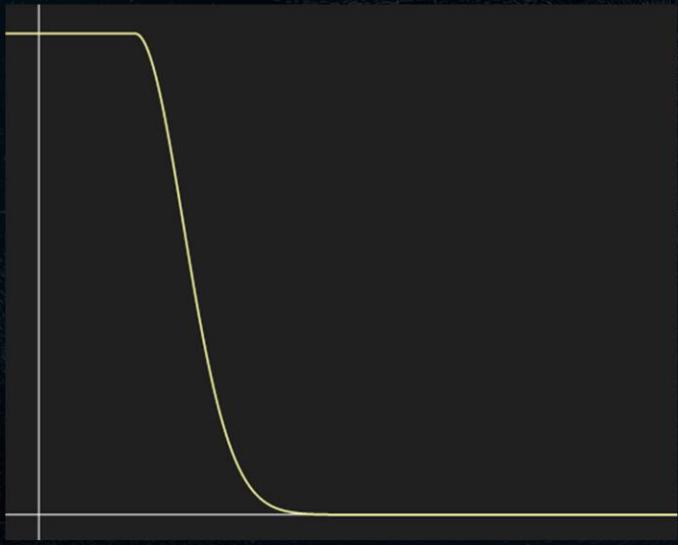
Rendering Über-Low-Res Particles

- Setup 2 render targets: color and depth transition buffers
 - Assuming a 1280 x 720 buffer

	w	h	format	initial
color (pre-multiplied)	320	180	a8:r8:g8:b8	rgb:0.0 a:1.0
depth transition	320	180	r16:g16	rg:0.0

The **depth transition buffer** records how that pixel transitions from the appearance described by the color buffer to 100% transparency as the depth moves towards the camera. We use this transition to properly apply the high resolution depth occlusion when compositing.

Depth Transition Function



The transition function looks something like this:

Where *the Horizontal Axis is depth* (camera on the left, increasing depth on the right)
And the *Vertical Axis is fully transparent* (1.0 at the top), and fully applied color buffer
(0.0 at the bottom)

Note that the range where it is transitioning corresponds to the depth range that the transparencies at this pixel occupy.

For EXTRA CREDIT: we can use this depth description to apply true volumetric lighting to the low rez transparencies! And conveniently those are our low frequency volumetric fog and other effects that benefit the most from that lighting

Particles and Variance-Based Depth

- Trying to approximate per-pixel function:
 $\text{transmission}(\text{depth})$
- Approximate $\text{transmission}(\text{depth})$ as the CDF of a Gaussian, where Gaussian is described by variance of depth

Really, we're trying to approximate the per pixel function: $\text{transmission}(\text{depth})$

The idea is to approximate $\text{transmission}(\text{depth})$ as the CDF of a Gaussian, where the Gaussian is described by the variance depth map.

Variance and Compositing

- VDM rules do not approximate the transmission compositing functions well
 - Especially when combining vastly different variances
- Limit the variance range for each particle
- Sort particles by their depth variance instead of pure depth
 - Results in minimal artifacts

However, VDM rules do not approximate the transmission compositing functions well, especially when we combine vastly different variances.

However, if we restrict the variance range of each particle, and sort particles by their depth-variance instead of pure depth, then the artifacts are fairly minimal.

Variance and Compositing

- VDM rules do not approximate the transmission compositing functions well
 - Especially when combining vastly different variances
- Limit the variance range for each particle
- Sort particles by their depth variance instead of pure depth
 - Results in minimal artifacts

We use the standard way to compute mean and variance from depth + depth², as this method has some good blending properties (blending two different depths with small variances results in a large variance). If we stored mean and variance individually we would not get this. However, the mean in our modified equation corresponds to the 'front' of the particle (essentially the depth at which it begins applying), not the actual mean depth. In our standard depth-fade particles, this, handily enough, corresponds to actual particle depth, as we can only fade behind the particle.

The variance controls how quickly the particle 'fades in' over distance -- a true Gaussian only approaches 100% at infinity, but we can clamp this behavior by subtracting epsilon and saturating, to force it to hit 100% at a specific depth.

Variance and Compositing

- Thus, color buffer records the transparent appearance at the conservative depth estimate

Thus, color buffer records the transparent appearance at the conservative depth estimate

Comparison: High Resolution Render



And here are some comparison slides.

In this screenshot we forced all particles in this scene render with high resolution setting.

Comparison: High Resolution Render



Hi res rendering: Notice the nice composite with the rest of the scene as highlighted in this region here...

Comparison: $\frac{1}{4}$ Res with Bilinear UpSample



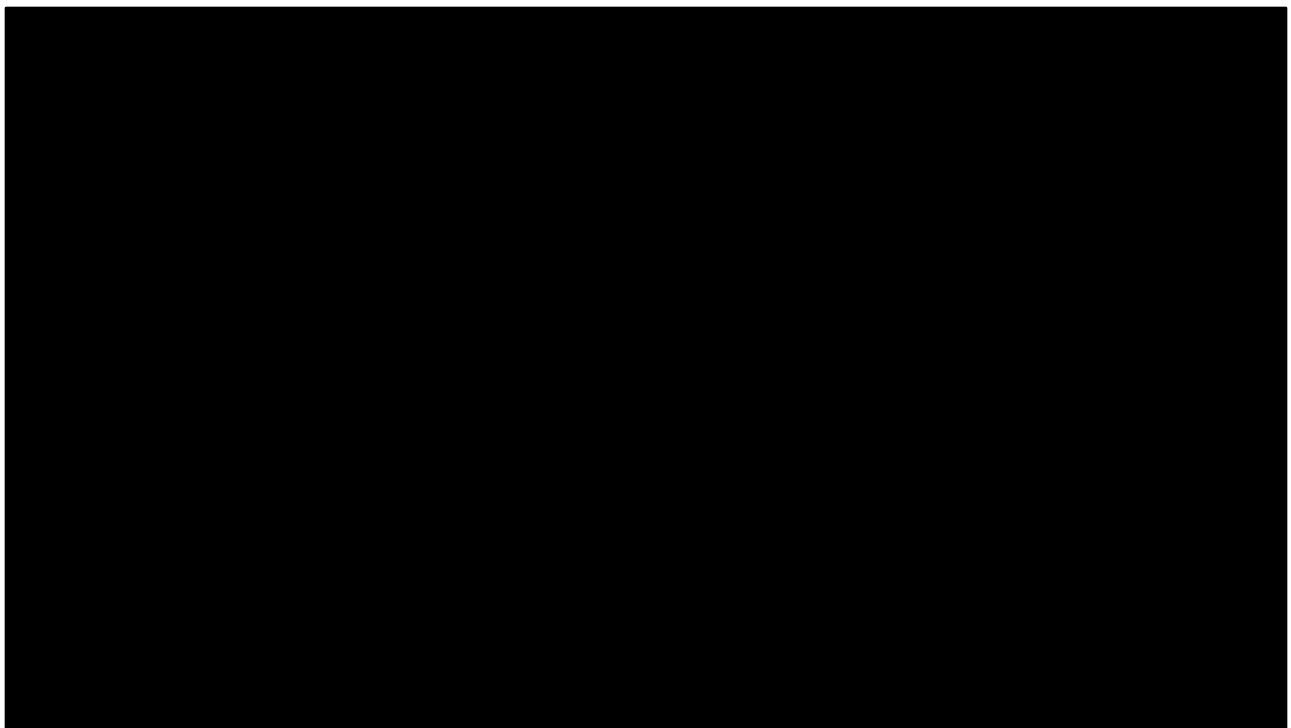
Here is another example: in this case, we rendered particles into a $\frac{1}{4}$ res target ($\frac{1}{4} w \times \frac{1}{4} h$) and used regular bilinear upsample to composite. In this case we notice the artifacts across depth discontinuities..

Comparison: 1/4 Res with VDM



And lastly, the same $\frac{1}{4}$ res rendering but this time using our new VDM technique for compositing.. Notice that we're able to maintain natural transitions along depth discontinuities while still keeping uber-fast rendering cost.

On that note...



OUTRO video (DESTINY E3)

Conclusions

- Developing a true AAA game title means many advanced graphics features
- Must empower artists to create worlds by creating flexible features
- New research is needed to improve art pipelines, new directions for game graphics going forward

Developing a true AAA game title is much joy, but also a bit blood, sweat and tears.. and much joy!

Staying next-gen current means many advanced graphics features

Thanks!

- **Bungie Graphics Team**

- Hao Chen, Chris Tchou, Joe Venzon, Jason Tranchida, Brad Loos, Brandon Whitley, Jason Hoerner, Alexis Haraux

- **Bungie Art Team**

- And in particular: Shi Kai Wang, Scott Shepherd, Ryan Ellis, Tom Doyle, Tom Burlington, Tom Sanocki, Cameron Pinard



A little postcard from the Studio... (in-development shot of a character in one of our levels made by our character artist, Scott Shepherd)

Bungie was founded in 1991 with two goals:

- * Combine brilliant technology and beautiful art with captivating gameplay
- * Achieve World Domination

We're Hiring. Apply Now.

www.bungie.net/careers

[careers@bungie.com](mailto:ccareers@bungie.com)



Questions?

natalya.tatarchuk@yahoo.com



Questions?

Questions?

natalya.tatarchuk@yahoo.com

Slides will be posted on

<http://advances.realtimerendering.com>

Bungie was founded in 1991 with two goals:

- * Combine brilliant technology and beautiful art with captivating gameplay
- * Achieve World Domination

We're Hiring. Apply Now.

www.bungie.net/careers

[careers@bungie.com](mailto:ccareers@bungie.com)





DESTINY

Q&A