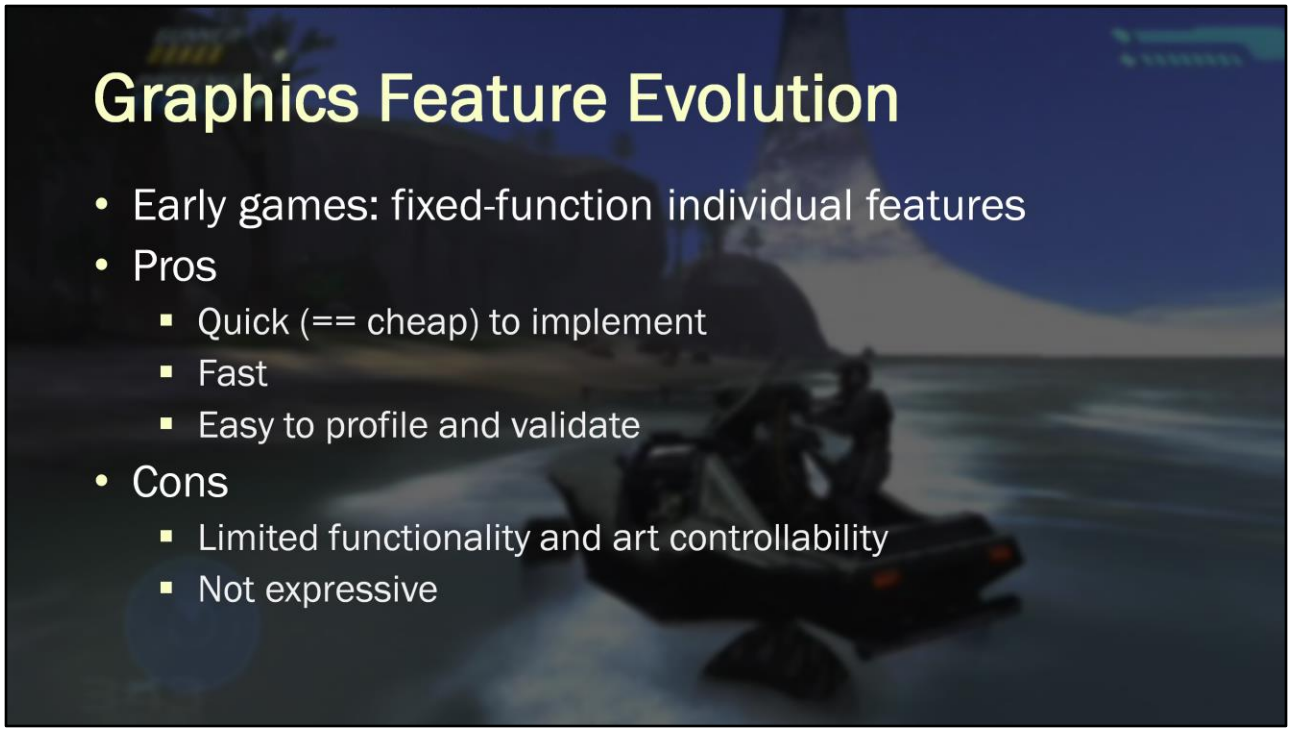
Hello and welcome!
My name is Natalya Tatarchuk, and I am a Graphics Engineering architect at Bungie, a video game company. Today we will be talking about Applied Graphics Research for Video Games: Solving Real-World Problems under Real-World Constraints.

Which brings us to this question.. **What makes video games special?** At least when we're talking about graphics research?
Let's take a look at the current trends in video game development.

If we look back, waaay back, Halo Combat Evolved had actually a good resume of individual graphics features: Terrain system, foliage, water, ice, snow, particle effects, simple flag simulation, vehicles.



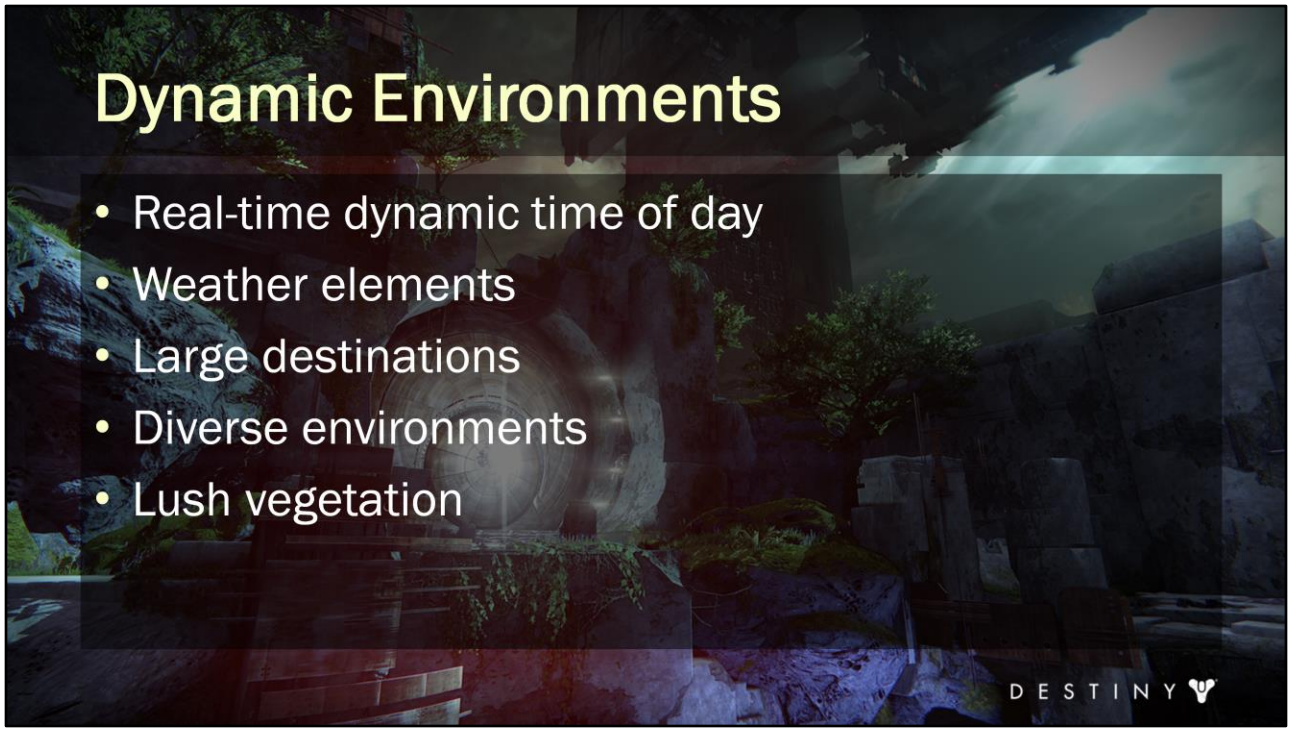
These early game features were fairly fixed-function. That made sense – it's a good route for small project – *don't overgeneralize without need*.
They were cheap to implement, were fast, and relatively easy to test. However, they were not expressive, and had limited functionality and art controllability.
But we've come a long way from there to now…

In the current world of player expectations, for AAA blockbuster games (just like movies), the expectations have risen drastically.
Technology has been progressing by leaps in the last decade or so and consumers expect modern games to showcase their capabilities accordingly.

Players have increased expectation of visual quality, independent of genre. This means high-quality lighting and shadows in real-time, high-quality cinematic postprocessing (blurs, depth of field, etc.).

At the same time, consumers expect a leap in immersion for games, which means increased fidelity. This includes increased resolution, larger destinations, denser and more diverse environments.

Players crave non-linear gameplay. Worlds that evolve and would always give the players new reasons to come back for more gameplay and adventure. What this means is that rather than building a singular experience a player can experience once, game developers wish to build content that players can explore on their own over and over again.

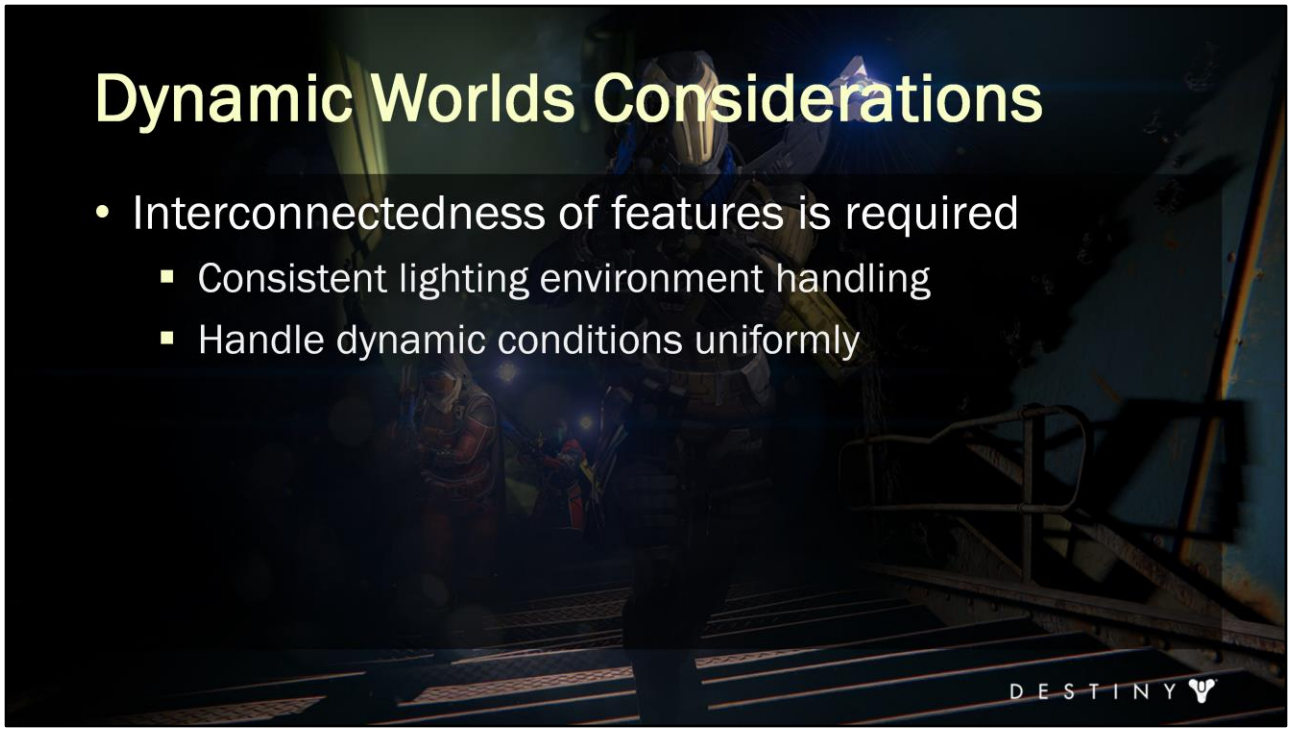


To make these non-linear game worlds work, we need dynamic environments. This means supporting in real-time dynamic time of day, weather elements, such as rain, snow, wind, etc. AAA blockbuster games in the latest years have large destinations, with diverse environments, lush vegetation. This means that the latest video games pack many more features and these features are far more diverse – they are no longer fixed function.

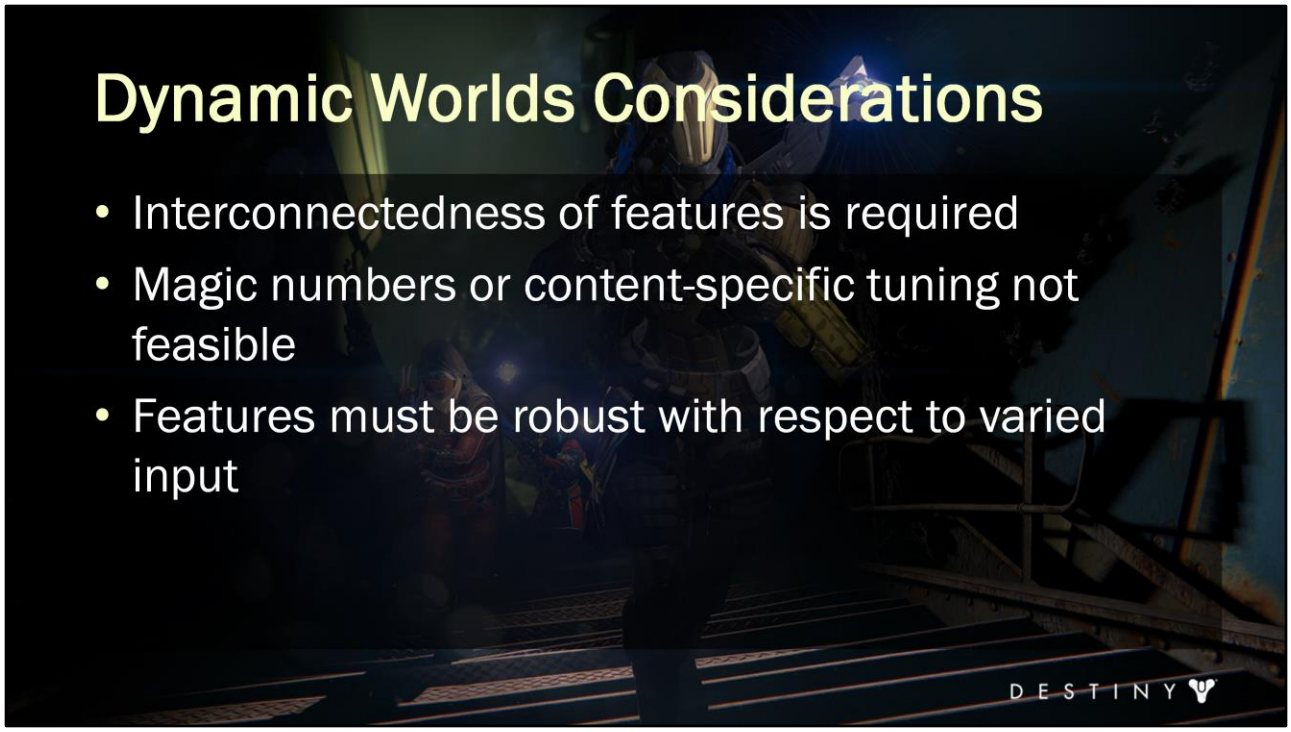
To make all of the above work, it is important that the control of the creation is shifted to the hands of content creators - the artists, designers and animators. Most of the AAA houses have about a third or less of their workforce consist of engineers and the vast majority of the company is made up from artists, animators and designers. This means that the features we provide must provide a great deal of art controllability, have intuitive controls and must be robust and sufficiently express complex phenomena.

To make the game worlds feel immersive, they need varied, consistent and complex simulations across variety of game systems and graphics features. We can't make our world feel truly dynamic if the elements do not convey the response to player actions. But this also means that it's not enough to have just a single elements. You must make all systems simulate dynamically, together.

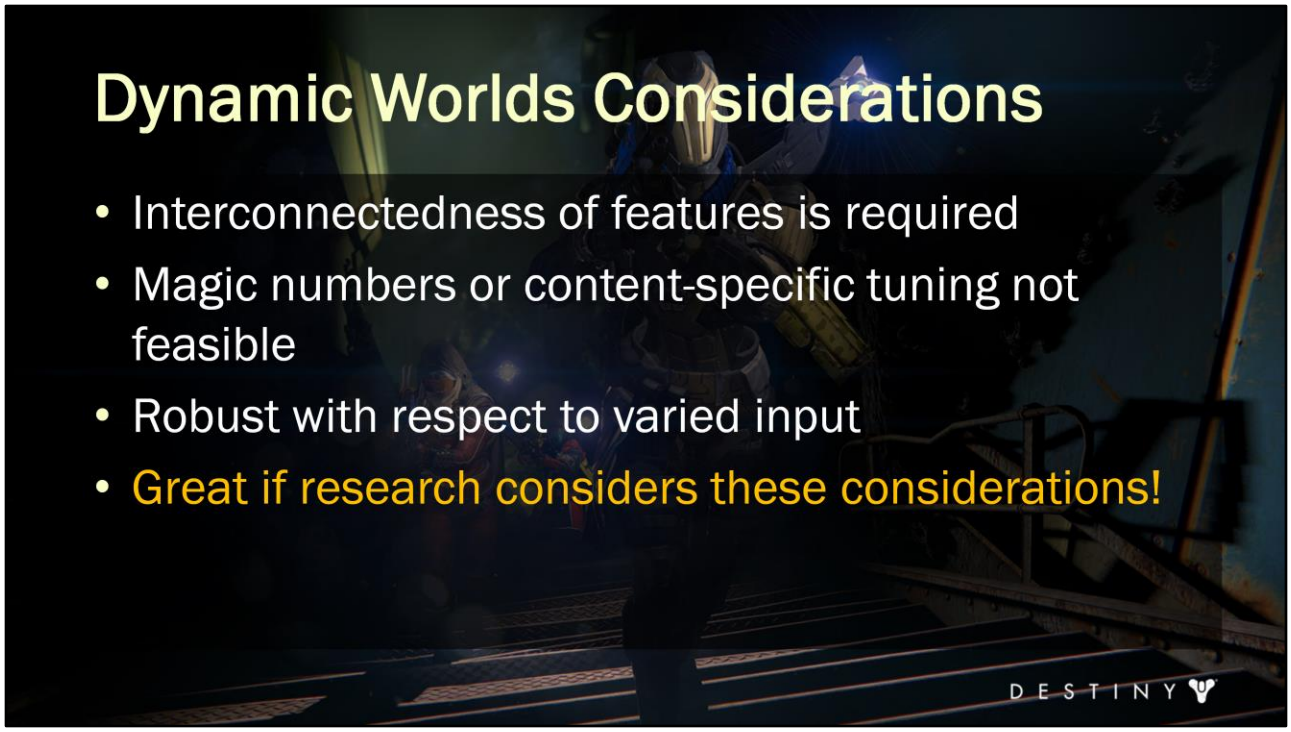
For example, in this scene, the player is moving around, aiming, and performing gameplay actions. We see cloth simulation responding to that, the grass movement responds to the player walking through, water ripples and splashes.



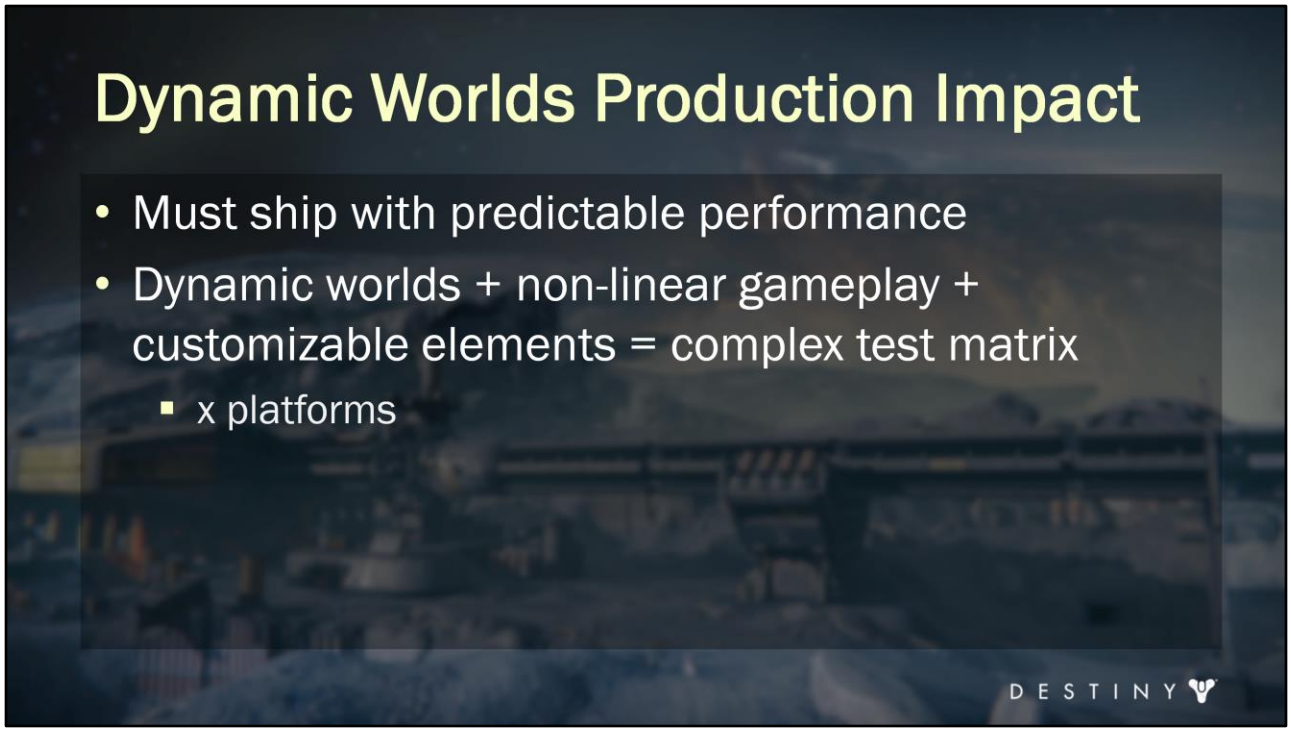
Making all these systems play well together incurs additional tax for each individual feature as they now need to be designed not in isolation, but in correspondence to the other features in our game worlds.
For example, shadows, global illumination, occlusion, translucence, etc. must handle time of day. All elements in the world need to go through the same path (for example, lighting environment) to be consistent in all situations. It's harder to get away with custom hacks (the crutch of our earlier development).



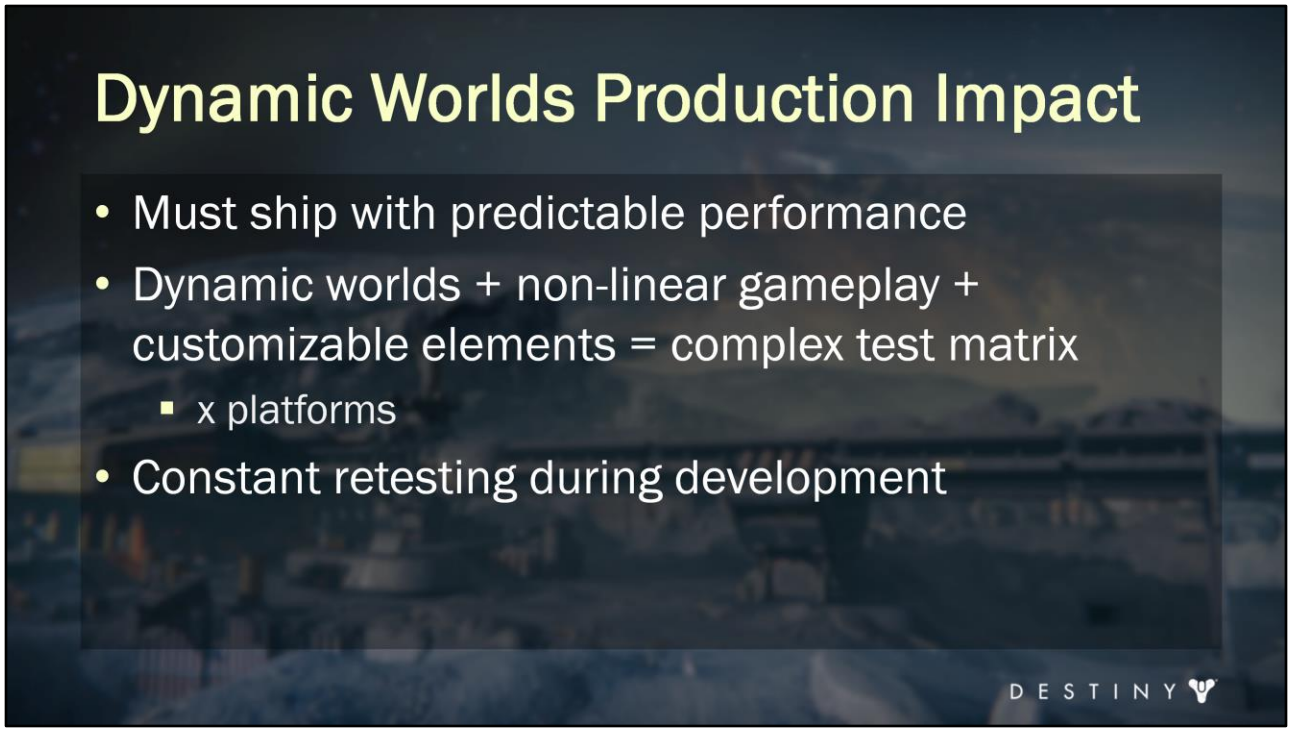
This, combined with the sheer amount of content needed for high-quality experience, means that requiring custom tuning of parameters to make your algorithm work is an utterly unrealistic expectation, and often is simply not doable.
What does it mean to tune cascade shadow split for individual levels when you have dynamic time of day?
Which time of day should we set the split value for? What about tuning shadow distances – accounting for time of day, or accounting for varied viewpoints from the player? What about tuning when you have dynamic weather? This also means that the features themselves must be robust to varied input.



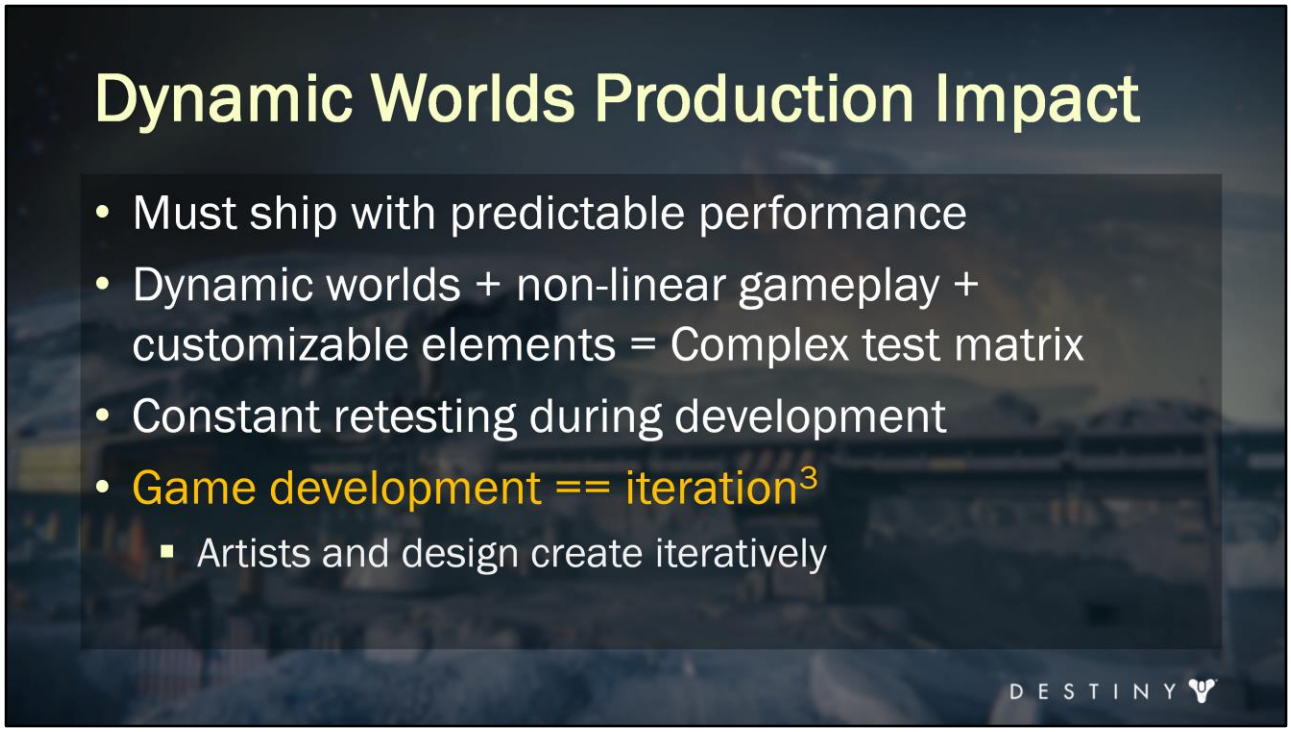
Thus a call to research – consider these requirements! The good news is that they make for more interesting problems! Of course, that this adds complexity for otherwise stand-alone techniques, so this request is by no means required, but considering this requirement only serves to improve the algorithms – and increases the chances of production adopting them.



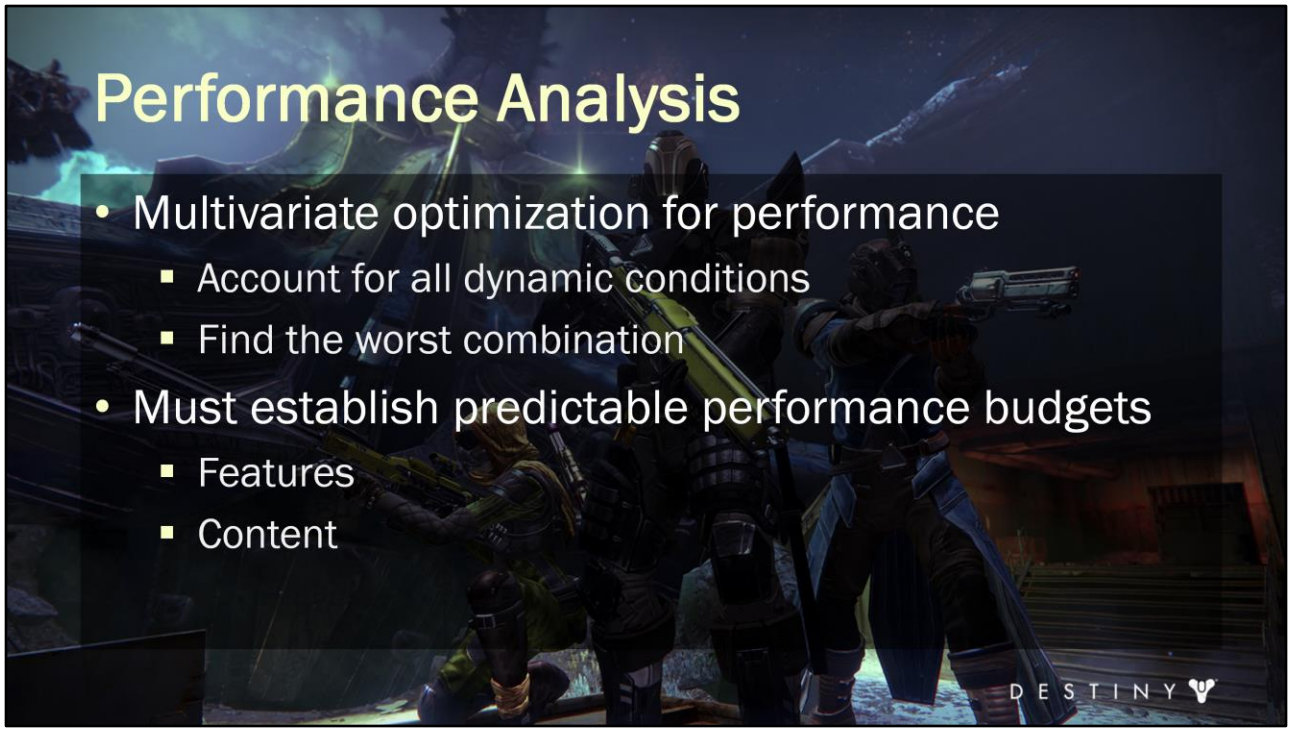
Creating fully dynamic worlds also has an impact on test and performance optimization. Games must ship with predictable performance in order to provide a good experience to the player. Combining dynamic elements with non-linear gameplay and player-driven customization means that the amount of testing increases exponentially. In Destiny, we had to test the entire matrix of our destinations, with all of the activities, with all of the different permutations of gear, on all of our shipping platforms.



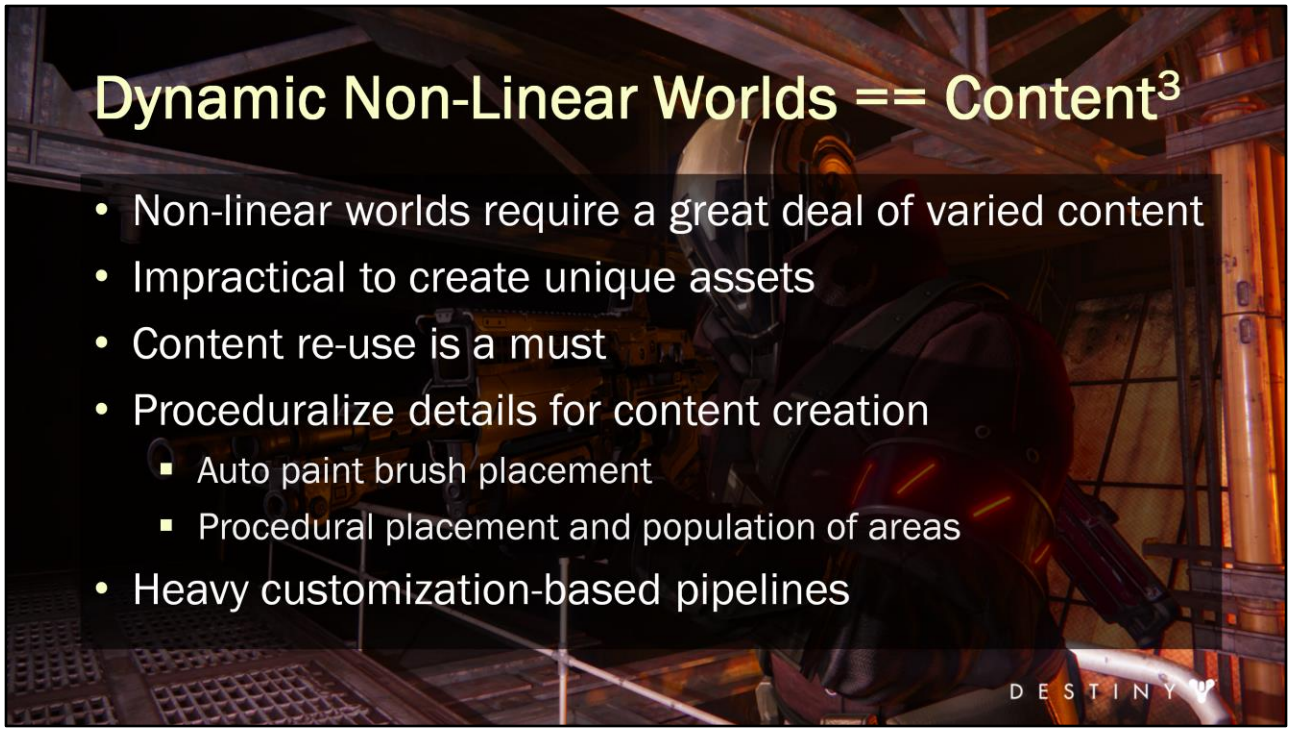
And we had to test that full matrix EVERY time major features were implemented or content re-work completed. Testing this thoroughly may be an overkill to some, but as recent rough launches show, it pays off when your players are having fun playing your game instead of experiencing poor perf or continuous online problems.



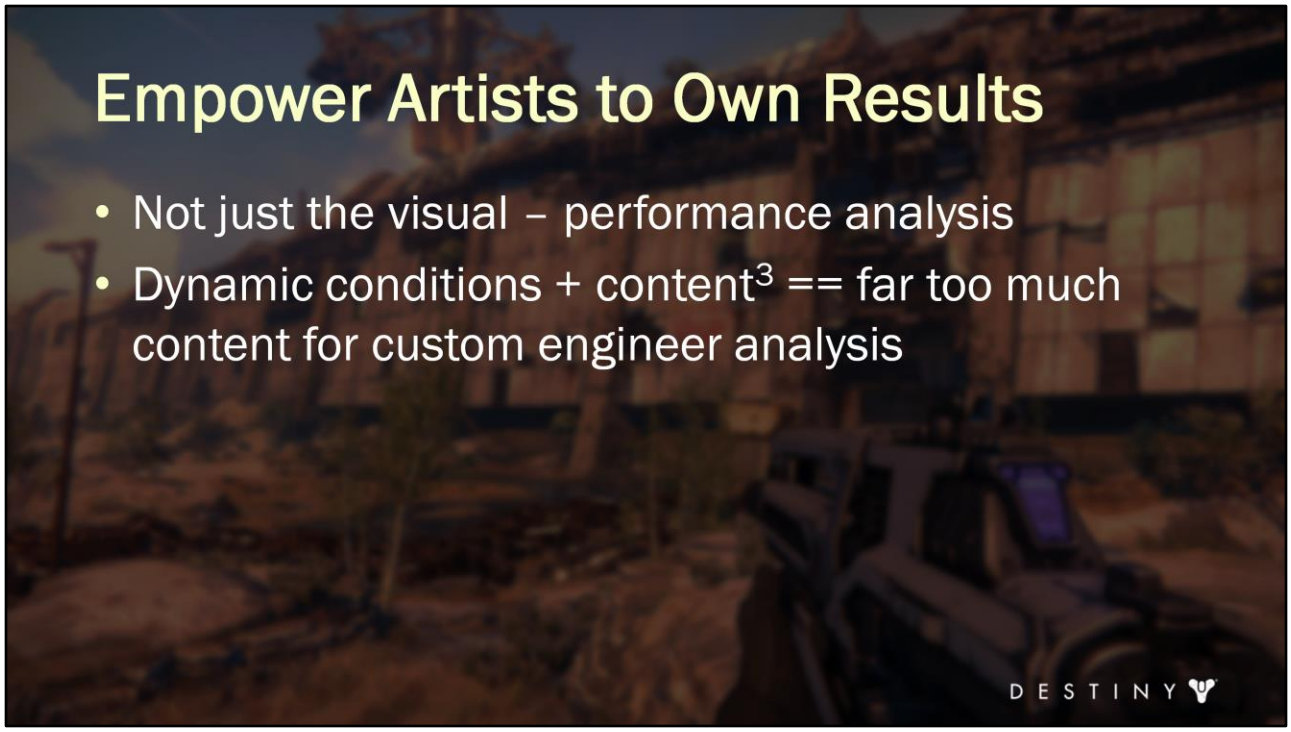
And let's not forget that game development is, first and foremost, all about iteration. We implement features, design levels, concept characters, bring them to life only to realize that something isn't quite right, and then iterate, iterate, iterate! Artists and design create iteratively. This of course means continuous re-evaluation and re-testing. That's another reason why magic numbers are so intractable in real world constraints



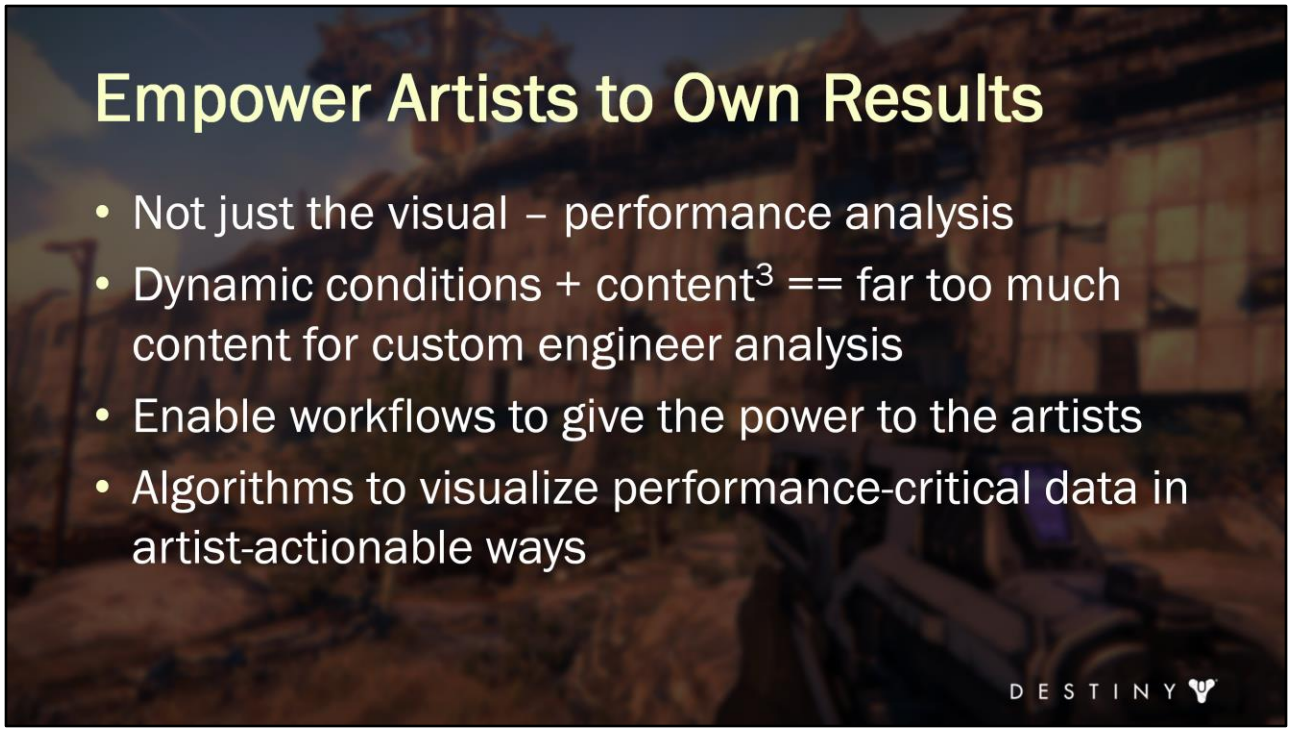
Having many dynamic elements (TOD, weather, destruction, etc.) also puts a tax on performance optimization as the amount of variables explodes. We need to account for dynamic conditions and find the worst combinations to know that the performance will be predictably bounded in a safe bracket suitable for the gameplay input latency we have set out to achieve.
To help this, we must establish predictable performance budgets for our features and content. This means that we need to know the behavior of algorithms across varied conditions. Is there a worst-case scenario that absolutely tanks performance? If yes, we might not be able to use it.



As I mentioned earlier, we need a great deal more content to have dynamic worlds with non-linear gameplay. When we look at the fidelity of the destinations AAA titles are striving to ship now, creating unique assets for all of the gameplay goals is impractical, even for game companies with large production teams. We have to re-use content effectively. This means that game developers invest in algorithms and tools pipelines that help proceduralize content creation, for example, automatic population of decorator locations based on painted importance information, etc. This is another reason why many have turned toward heavy customization pipelines which allow create vast variety of content based on componentized building blocks.



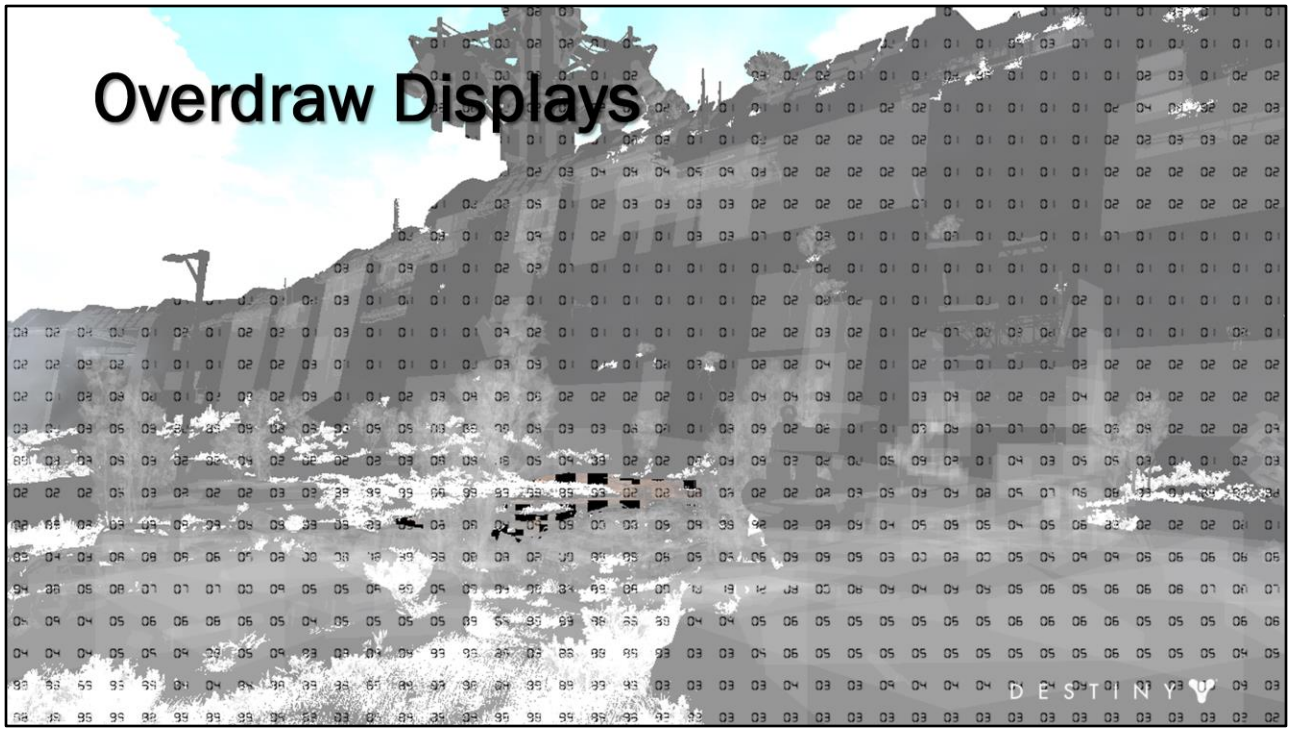
When we look at the sheer amount of content AAA games need to create, it becomes very clear that we must empower the artists to own end-to-end results for their work. We need to empower the authoring improvements where we can (artists would love it if they could truly rid of Uvs forever for real-time). But we also have to consider that the content has to be created with specific platforms in mind. And this is true for analysis of content for performance behavior.



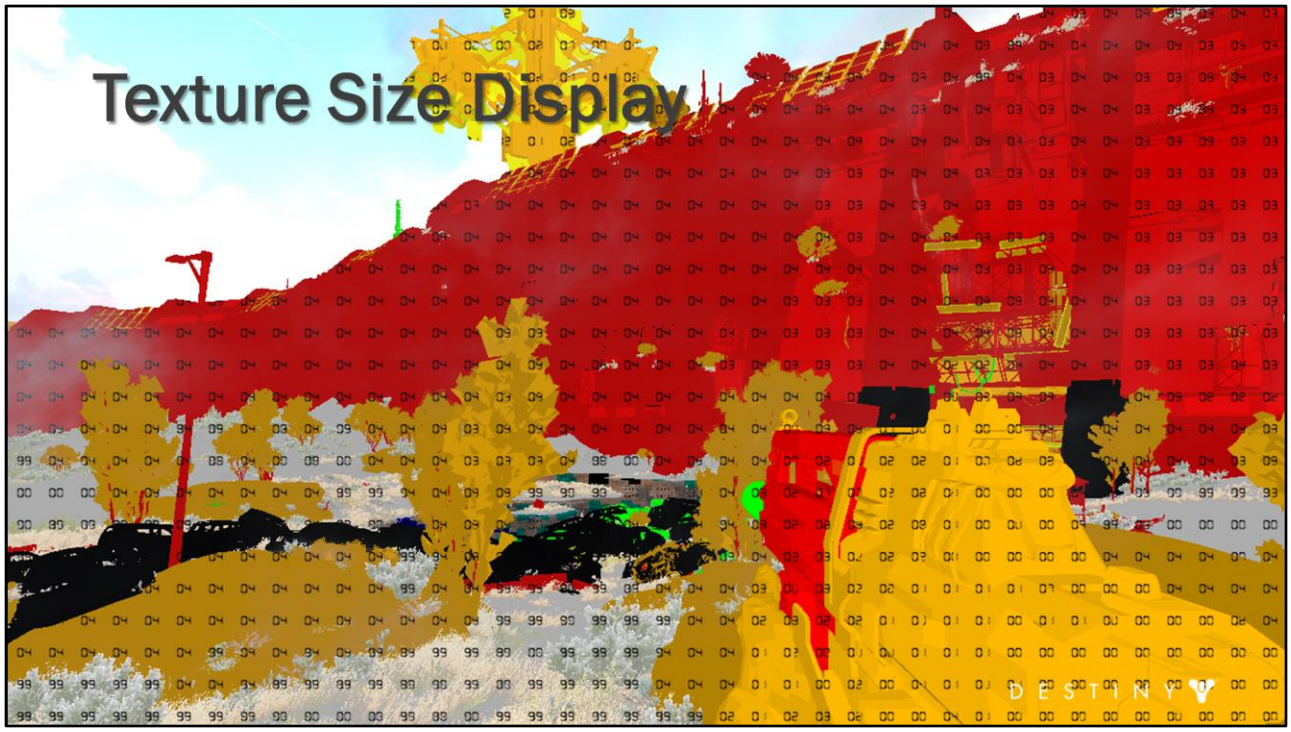
We firmly believe that empowering artists to be able to change their assets as quickly as possible without having to involve an engineer makes development loop far faster overall. By the time we the graphics engineers would load a level into a GPU profiler tool and take a look to see which feature “cost” too much, several hours may have gone by. But if an artist could just take a look at some visualization mode and understand the cause of the problematic performance issue, they can simply change the content on the spot. Thus we have invested and will continue to invest into algorithms to visualize performance-critical data in artist-actionable ways

Let’s take a quick look of an example of how this worked in our game. Here is the default view from one of our levels in the game

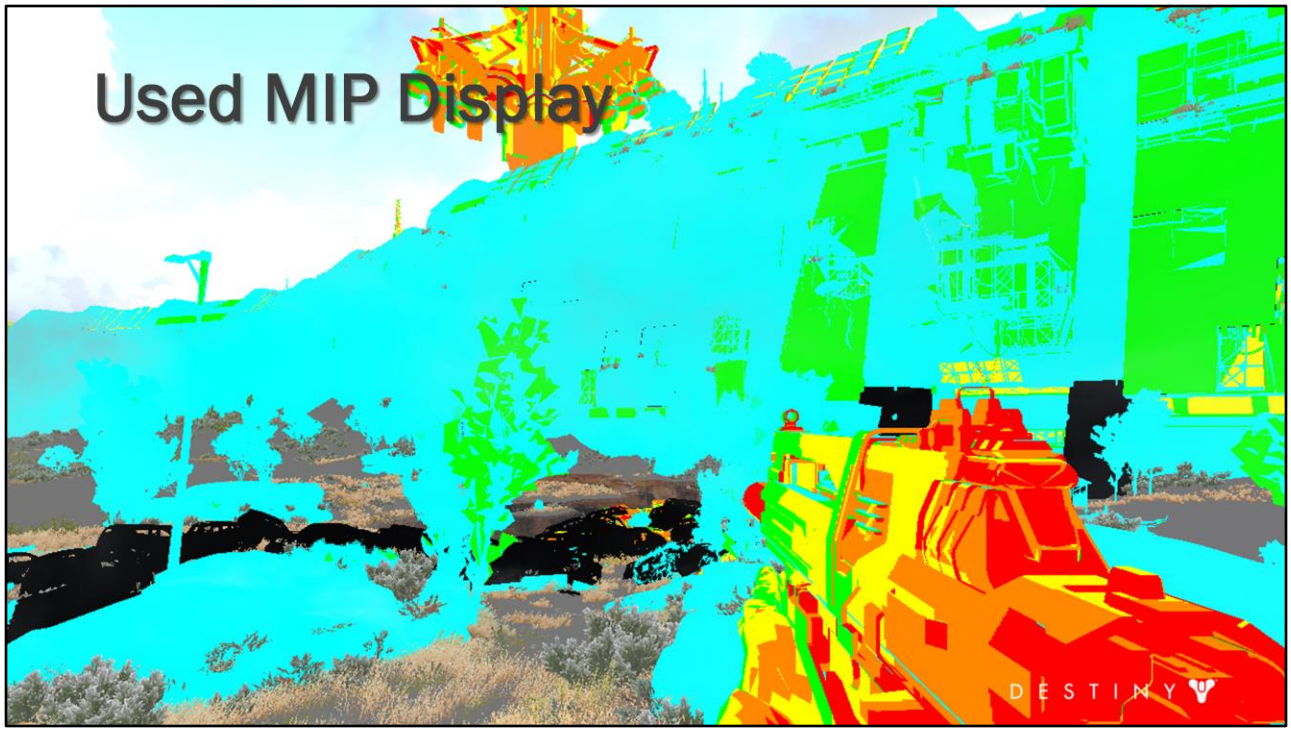
We can start from the obvious choices such as on-screen geometry budgets to show the number of vertices, drawcalls, etc. (actual numbers are hard to see but you get the idea). These budgets are developed with specific platforms in mind.



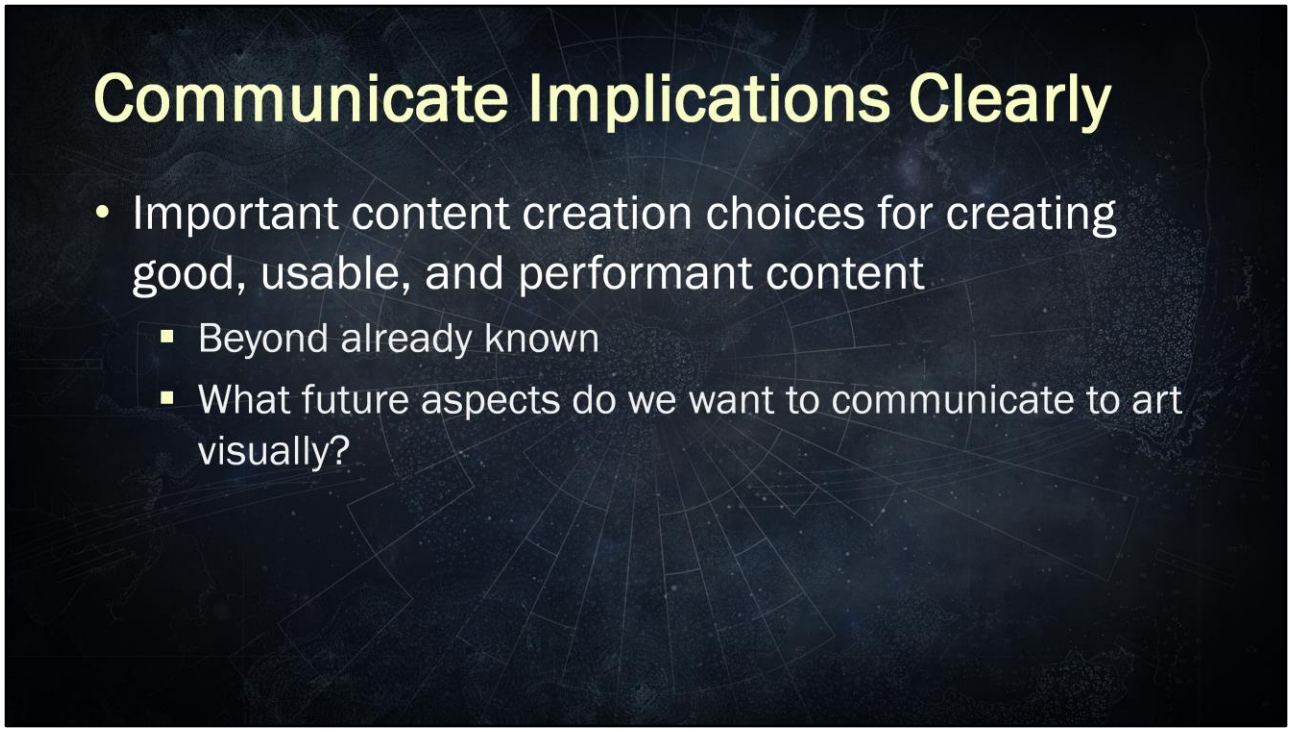
And add in-game visualization modes for other properties – for example, the overdraw mode helps artists control transparent elements performance by limiting overdraw for foliage and other transparents (directly correllating to performance of those elements in our pipeline). We have a similar mode for visualizing lights overdraw mode which helps lighting artists evaluate performance cost for their lighting setup.



Some of the other displays we added were texture size display which allowed the artists to see if they're killing texture cache unnecessarily (for example, maybe your object didn't need a 1K texture because you never see it very closely)



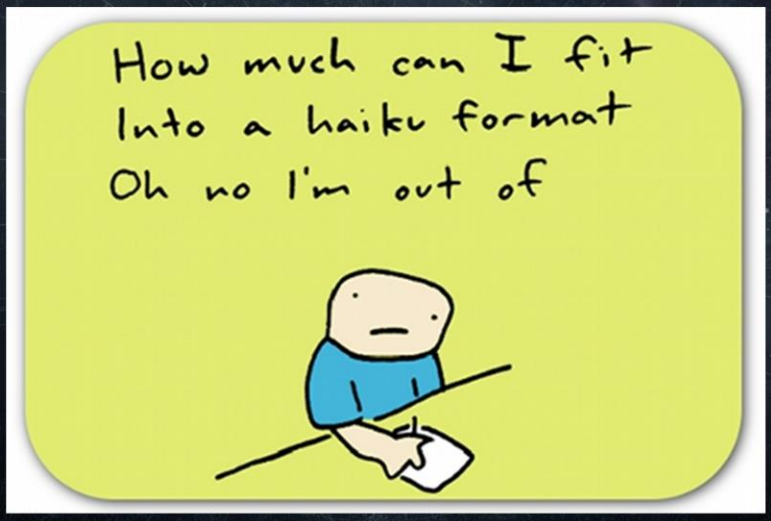
Or even a mip used display, so that you can see what MIP levels were actually perceivable in game (and maybe cut down on memory requirements by reducing textures)
And many more (runtime pixel cost, triange / drawcall visualization, etc.)



So when developing graphics techniques we always must think – how will we communicate the implications of the algorithms for content authoring choices to help content creators generate good, usable and performant content. We should always think – what aspects of our algorithms do we want to expose to communicate to art visually the important constraints that they need to incorporate into their production pipeline?

So with all these expectations for video games we have to remember that video games ship to run in real-time (game real-time, 30 or 60 fps on constrained platforms). When people hear the word 'constraint', a typical response is a slight desire to run. "Why, darn it, why?" But what about the beauty that constraints bring? Self-imposed constraints, when you have none, can only improve your work. After all, setting the right set of constraints is how you deliver products…

Know and ♥ Your Constraints

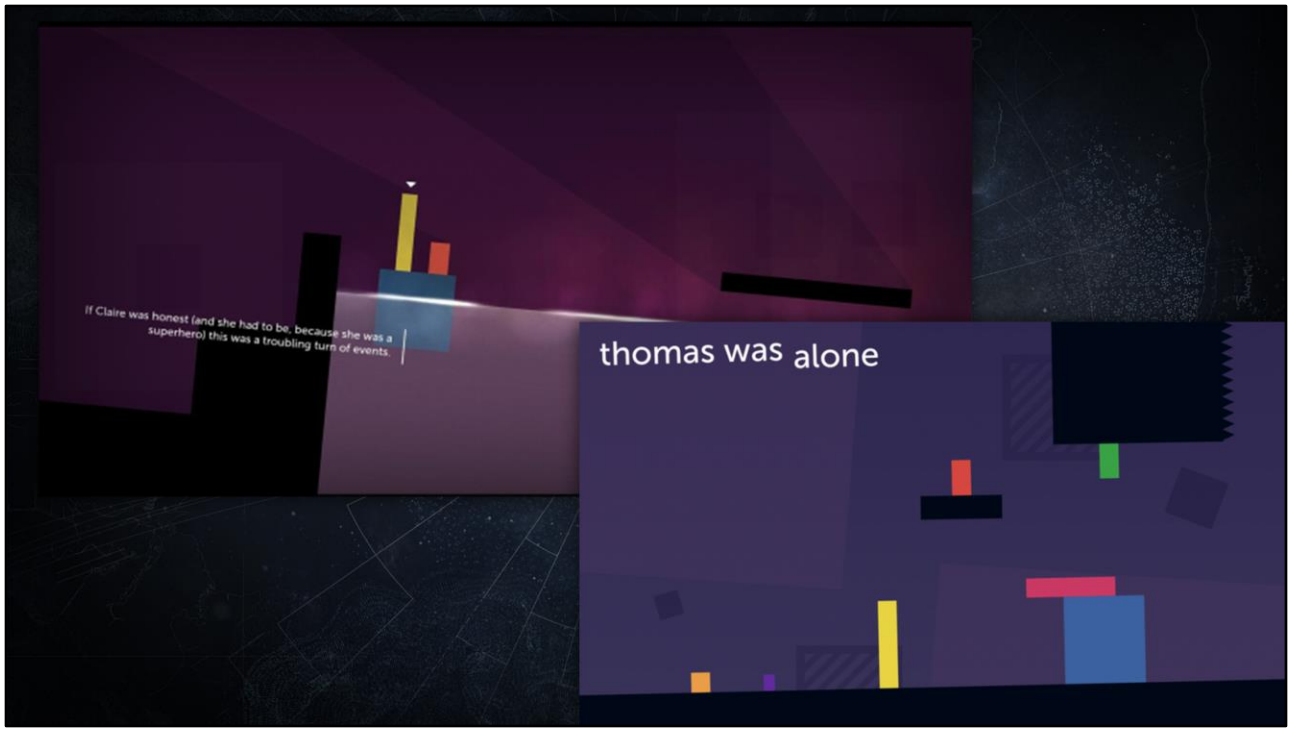


We know that having concrete obstacles may prompt you to step back and start looking at the big picture. SPU was a very good such obstacle, prompting game developers to completely reformulate their solutions to generate data- or task-parallel designs. What doesn't kill you, after all, well, makes you curse stronger... But it also makes you come up with really interesting solutions.

We can use constraints to guide us to new, utterly unexpected solutions. And of course, never take constraints for granted. After all, sometimes we get so focused at a wall in front of us, we may not realize that the solution takes us into an entirely different land. And in the research field, you can smartly use known constraints to look at the big picture to look further into the future, changing hardware pipelines, changing rendering paradigms.

In order to solve any problems, we have to correctly identify the problem statement.
And because problem statements consist not only of the end goal, but the constraints that define the possible space,
We must correctly identify both the goals AND constraints before we set out to solve problems.

So naturally while developing video games we encounter a few real-world constraints.. Where ‘real-world’ in this case is defined by ‘AAA blockbuster game production needs’ since that is the world that I’m deeply ingrained in.



But many of these constraints apply, in a scaled down fashion – and perhaps even more extreme – to the world of indie games as well

There are a number of real constraints that video games creation has to be aware of (and we already touched on some of these). I won't cover all of these in depth but it's enough to give you a taste.
On the high level we can list them as these buckets:
- <Artistic> vision / game play design constraints
- <Production> constraints: resources (man power, farm generation, timeline)
- <Platforms> - Hardware and System Constraints (Performance, Memory)
- <Risk> Management
- <Content> Production
- <Scalability>: Cross-Platform and Cross-Generation
- <Legal/Intellectual> Property

In video games, as in many other creative pursuits, everything begins with an idea. It all begins with the design and artistic vision, the creative concept. After all, the technology must first and foremost serve the aesthetic purpose of *gameplay*, the experience we want the player to have

Here, for example, we have a concept image from the early days of Destiny, for one of our destinations, Mars.

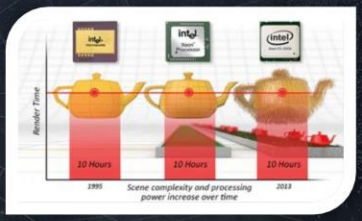
And here is an image from the resulting in-game rendering of that level.

All of the technology implemented for video games must take the artist and gameplay vision and **communicate** it to the player. This means that the graphics algorithms and features we implement must make sense within the aesthetic that the video game wants to achieve.

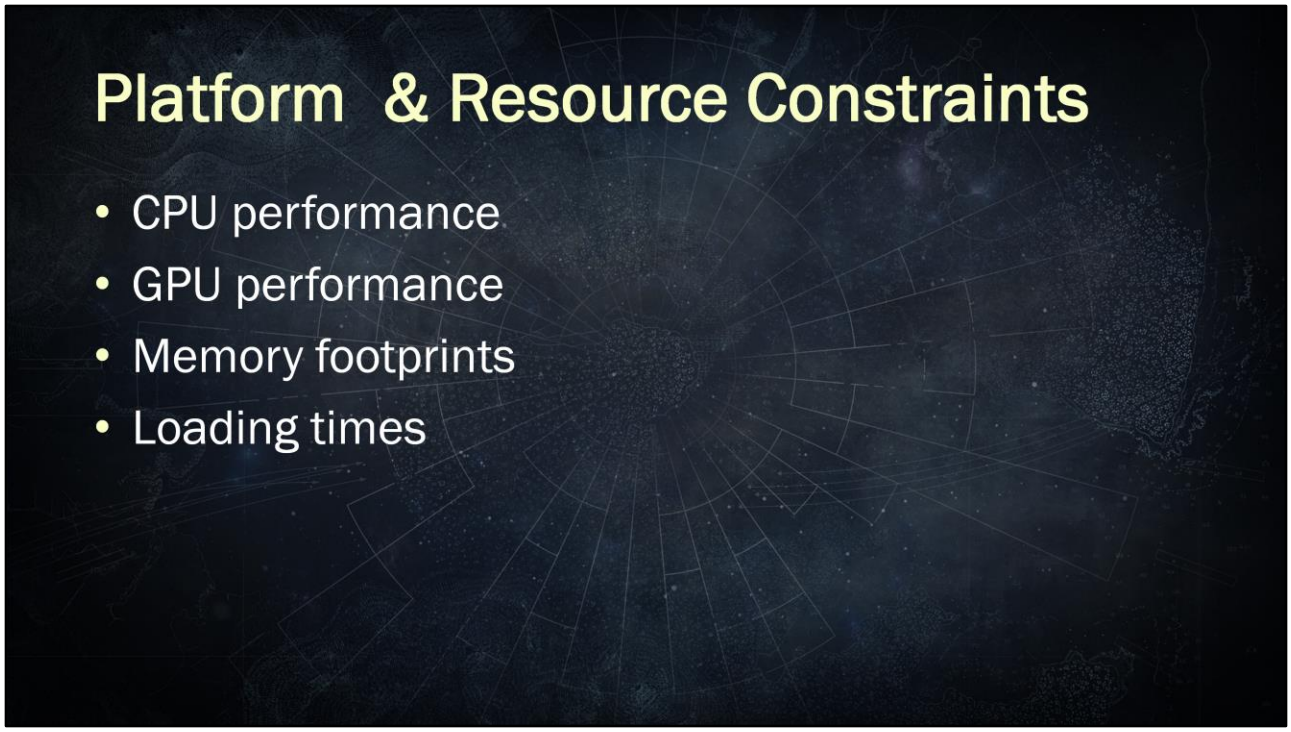
Enabling palette and mood also means having high quality lighting – after all, achieving artistic vision means giving the artists tools to express it.

What's the Challenge?

- *As technology advances, rendering time remains constant.*
  —Blinn's Law
- Holds true for games as well
  - Time to preprocess a level
  - Time to render a frame stays constant (33 or 16 ms)
- Yet the communicate the design and art goals

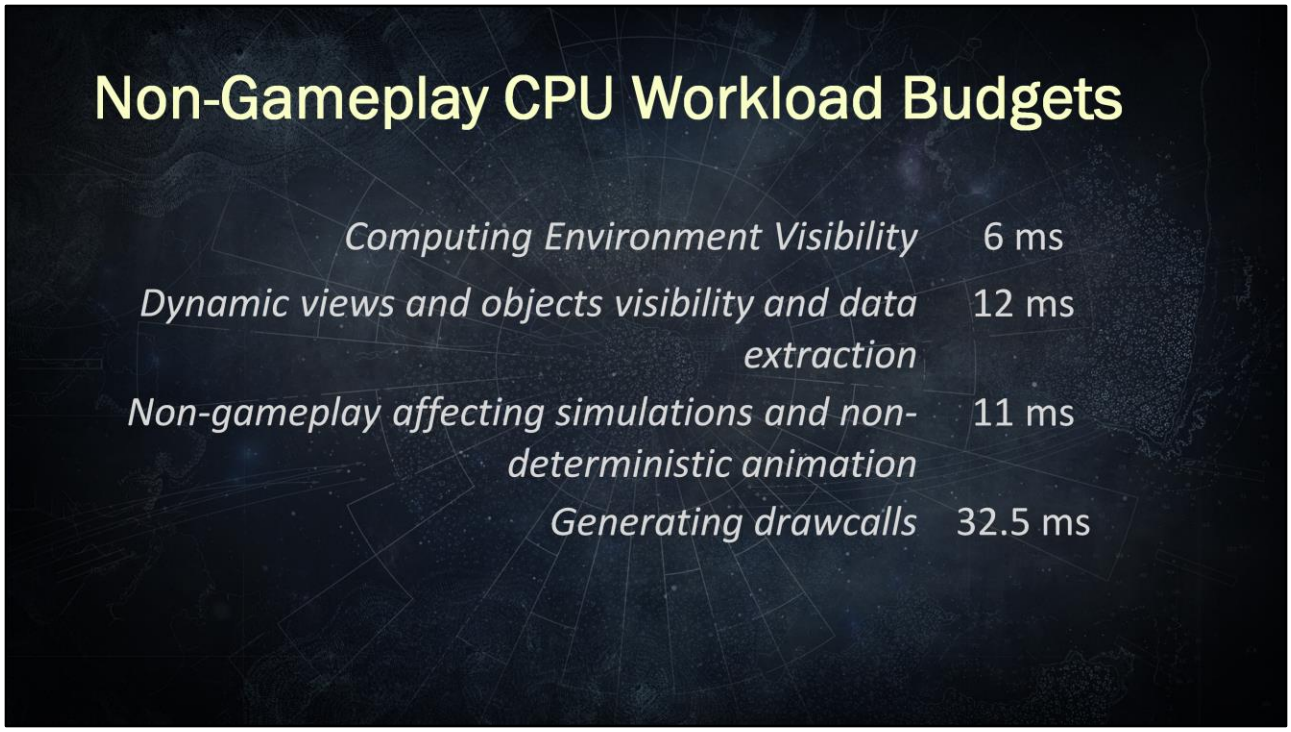


But to communicate the artistic vision may mean completely impractical requests. Blinn's law which essentially states that the time to render a frame remains constant, stays relevant for all media. While a game development frame does not take 10 hours to render, this law is still accurate for both time it takes to preprocess a level and for the time it takes to render a game frame. We got our 33 or 16 ms to fill – and must communicate design or artistic vision in a practical way.

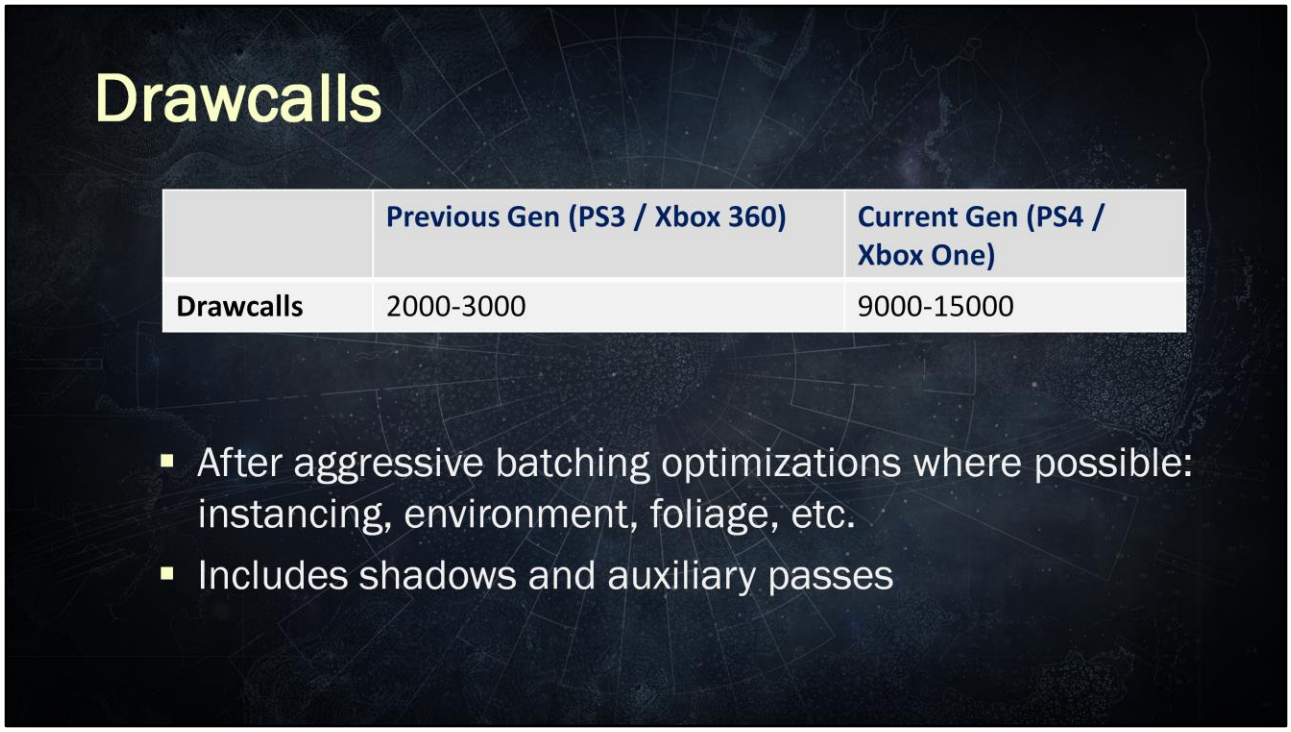


Here are the platform specific constraints that we have to mind while trying to bring the artistic vision to life:

- CPU and GPU performance
- Memory footprint
- Loading times

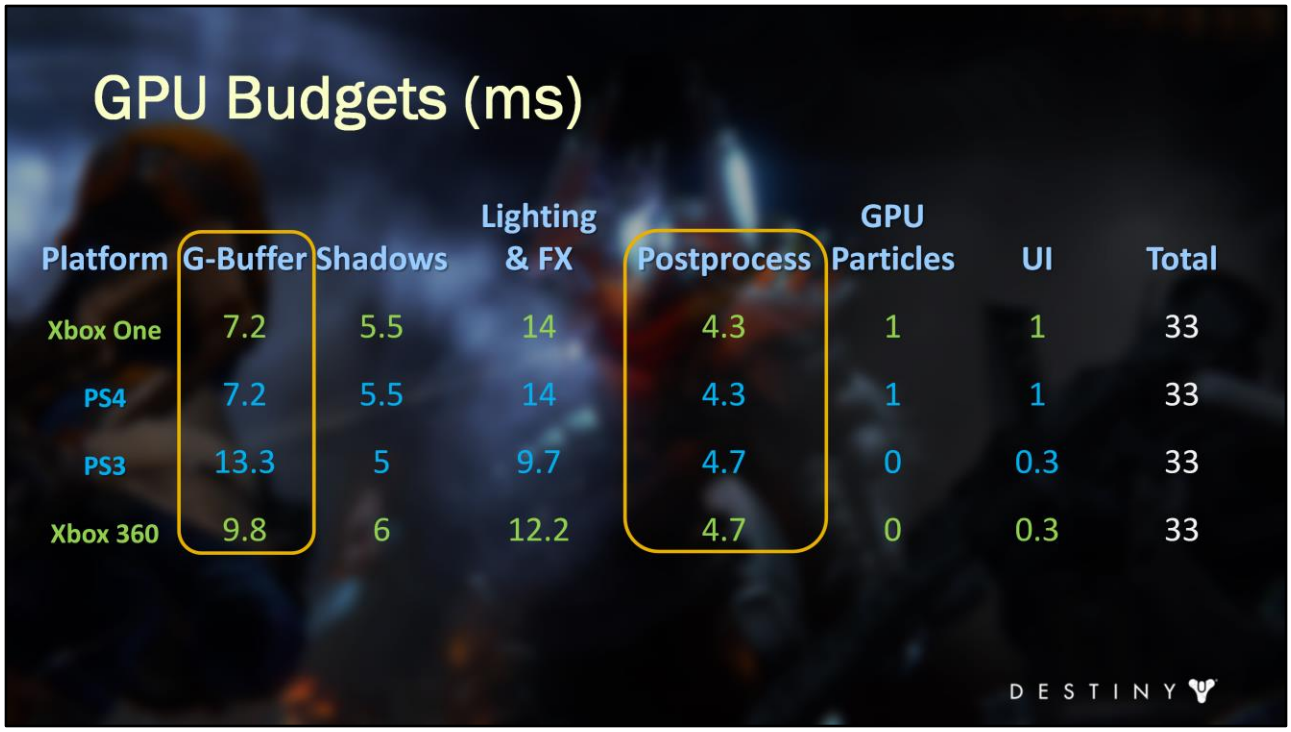


Here are the CPU workload budgets that we used to ship Destiny (which ran at 30 fps) for rendering workloads. I excluded gameplay affecting budgets though they obviously are also very important. Note that these budgets held for *all* of our shipping platforms – they didn't magically increase on more capable PS4 or Xbox One. But at the same time we were trying to render far more on those platforms. You will also notice that these did not add up to 33 ms – that is because we overlapped some of the computations (for example, submit to the GPU for previous frames overlaps with simulation for current frame).

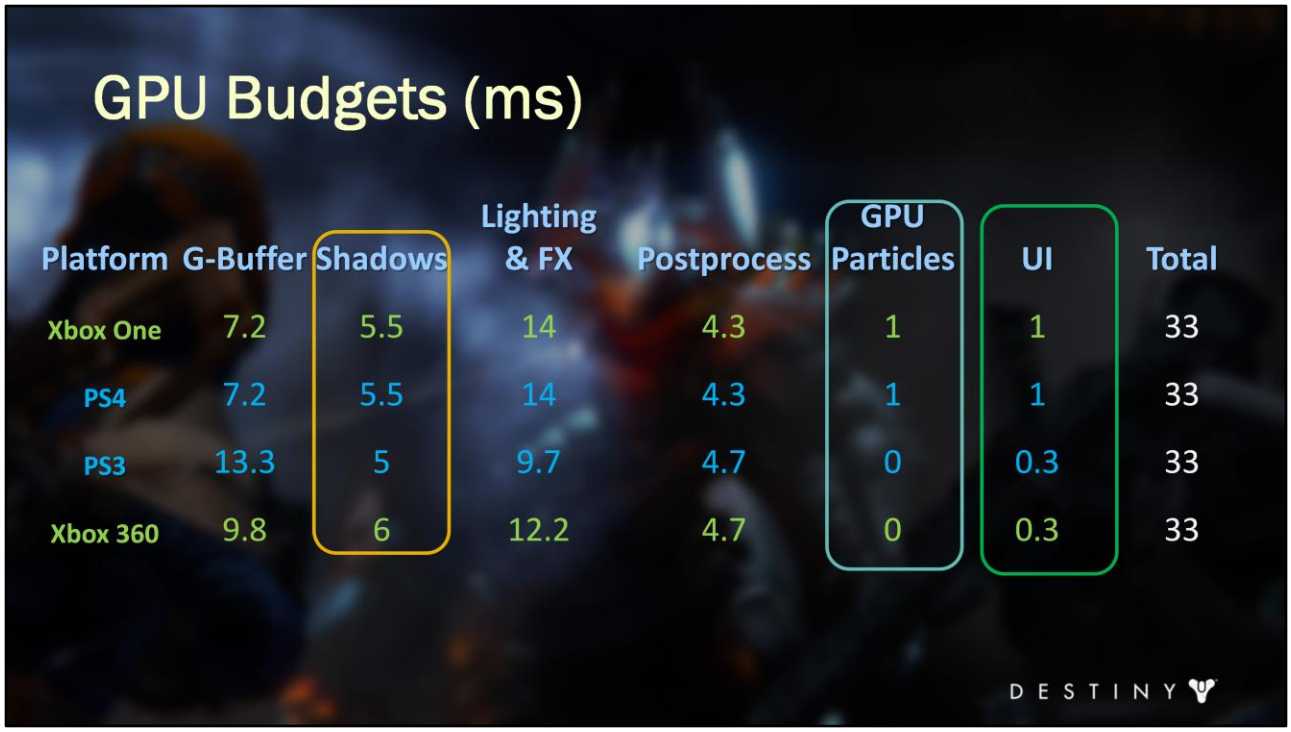


And at the same time while the CPU budgets did not increase, the number of elements we were drawing increased. Here is the example of drawcalls that we were pushing through a single frame of Destiny across two generations of consoles. You notice that we increased nearly an order of magnitude in one generation – without increasing our CPU budgets for submit. However, the CPU power has not increased in the same proportion– so we had to learn to become more efficient on those platforms.

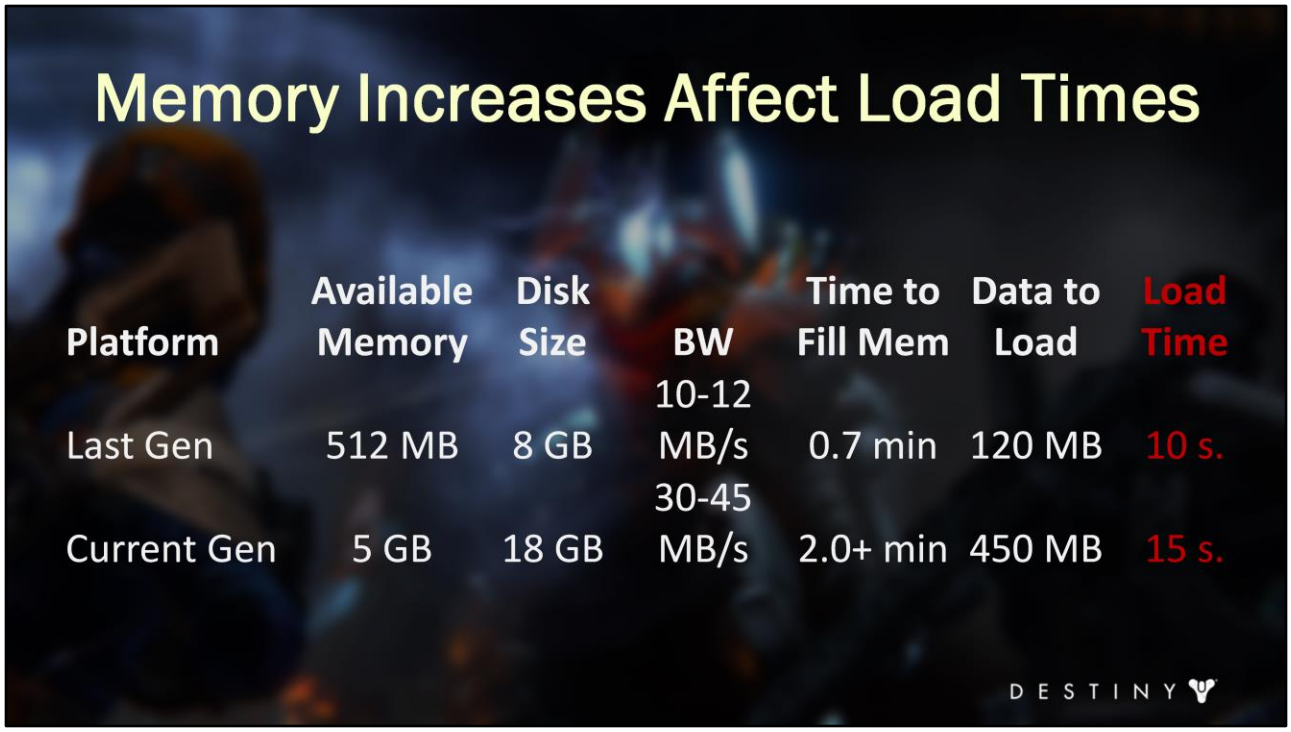
Here are the GPU budgets for roughly-broken down features across different platforms. We see a similar trend.



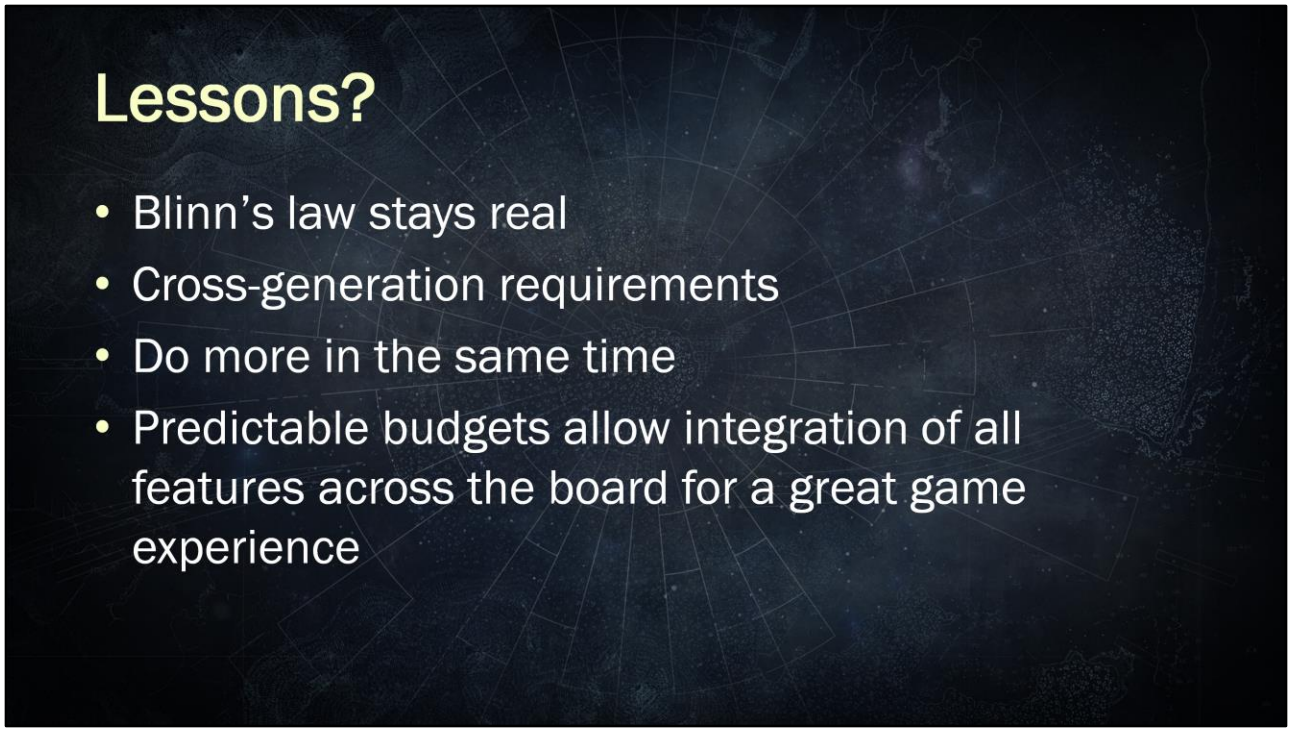
Note that although some of our budgets *decreased* with the higher power platforms that can do more in less time



Many stayed <the same>. While the GPU throughput is higher on the latest generation consoles – we are rendering *more elements* in a given frame (higher quality shadows, more elements in Gbuffer, more FX, etc.). And we also <added> graphics features on new consoles as well as <accounted> for increased resolution costs (for example, running at 1080p on the latest generations).



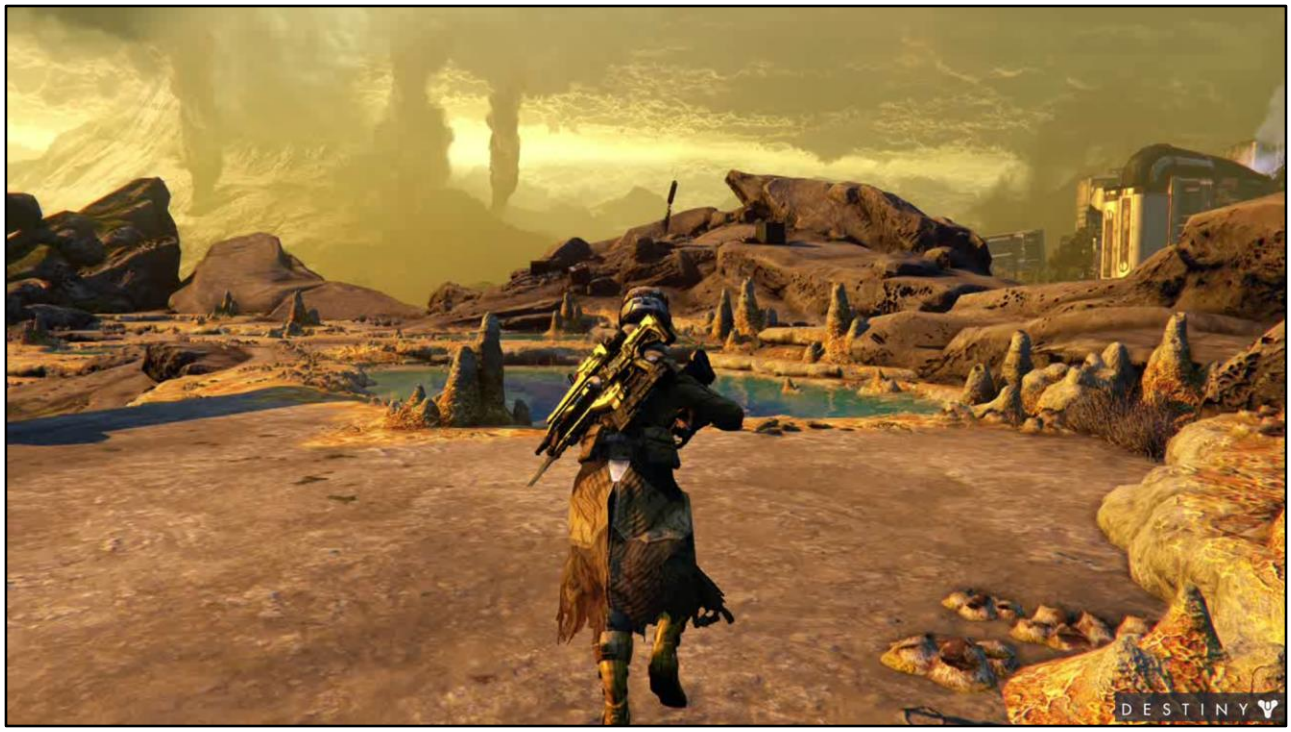
With respect to memory, although the latest generation of consoles is far less memory constrained than previous, there is an axis that has in fact not improved with more memory thrown at the problem. Here is an example of how the loading times changed across the two console generation. Although we have access to far more memory on the current generation on consoles, the loading times *increased* as the I/O bandwidth simply did not grow in the same order as the demands on memory.



So to put it simply, we continue to have to do more in the same time. As Blinn's law states, we have to
do more in the same time. And yet, we have to consider the impact of all the features across several generations – with the same budgets. And of course, having predictable budgets is what's required for creating a robust and reliable game experience.

So with all these constraints in mind, let's take a look at a couple of R&D integration examples from Destiny to explore what lessons we gleamed for R&D process.

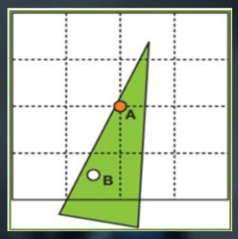
The first case study is our work on developing a technique for low resolution transparency.



Effects are a key component of Destiny's visuals and the sandbox gameplay. We have loads of them. And they all use alpha blending. Because of that, transparents rendering are excessively dominated by the fill cost, thus reducing resolution was crucial.

Key Idea

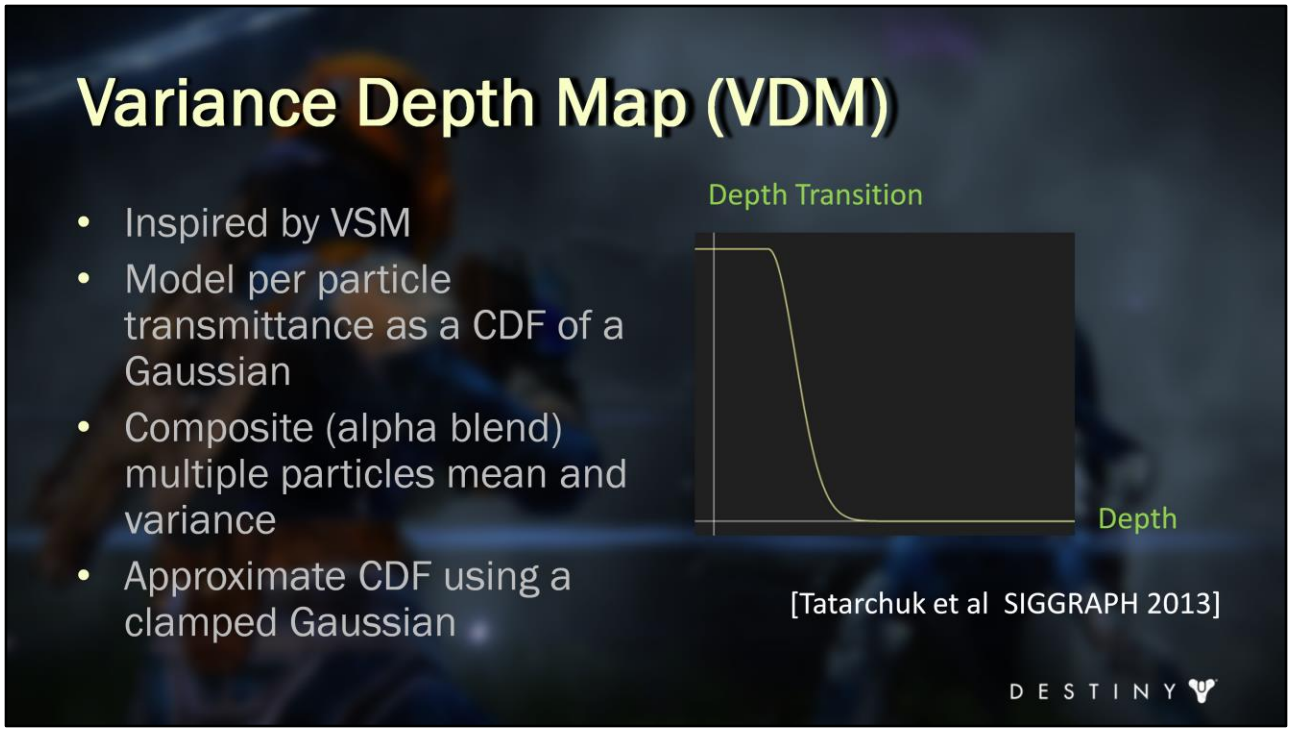
- Render transparent FX to low res buffer
- Use down sampled depth to Z-cull
- Up-sample low res buffer and blend
- Solve/manage edge artifacts



http://http.developer.nvidia.com/GPUGems3/gpugems3_ch23.html

The key idea we started exploring is to render transparents FX to a low resolution buffer. <Then> use down-sampled depth to Z-cull and <upsample> the low resolution buffer to blend with the high res frame. <The key> challenge is to reduce edge artifacts during this blend operation.

Here is an example of the artifacts with straightforward upsample blend: in this case, we rendered particles into a ¼ res target (¼ w x ¼ h) and used regular bilinear upsample to composite. <In this case>we notice the artifacts across depth discontinuities which <when> you look closer are very apparent



We introduced a new technique inspired by <variance> shadow mapping in the Destiny talk in the Advances course at SIGGRAPH 2013. In this technique, we <model> per-particle transmittance as a CDF (Cumulative distribution function) of a Gaussian. We <composite> using alpha-blending mean and variance for multiple particles per pixel. Thus we <approximate> the CDF using a clamped Gaussian.

Variance Depth Map (VDM)

Output:
Color: RGB * (1.0 - alpha), alpha
Depth: depth * (1.0 - alpha), depth*depth * (1.0 - alpha), 0.0, alpha

Two blend operations:
(S * A0 + RGB0) * A1 + RGB1
S * A1 * A0 + RGB0 * A1 + RGB1
S * (A1 * A0) + (RGB0 * A1 + RGB1)

$$G = k * \exp(-\frac{(\mathrm{x} - mean)^2}{2 * var})$$

When rendering to low-rez buffer:

$$T = saturate\left(\frac{exp2(-\max(x - mean), 0.0)^2}{2 * var * \ln(2)}\right) - e)$$

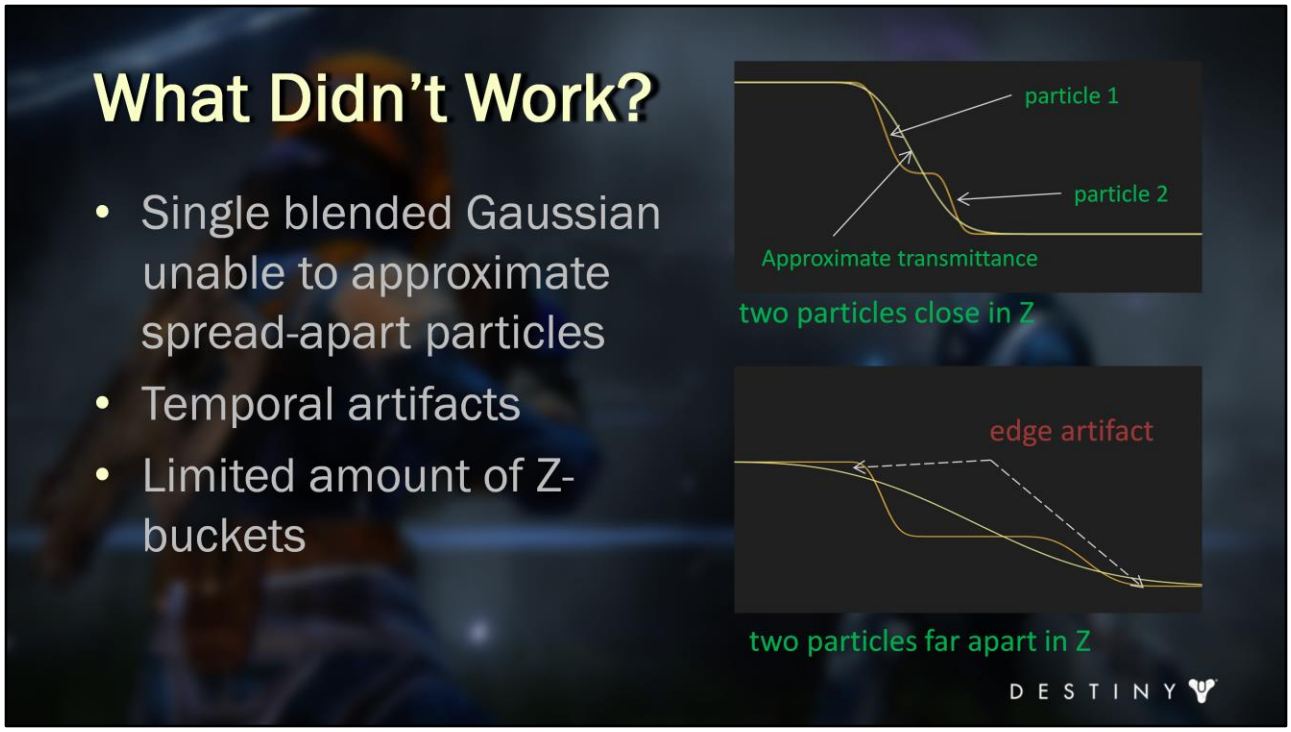
color	RGB blend: D.RGB * S.A + S.RGB A blend: D.A * S.A
depth	RG blend: D.RG * S.A + S.RG * (1.0 - S.A) -- or --- for d 0..1, d^2 < d

DESTINY

Here are the technical details (but you can find them in the slides later or in the 2013 talk's slides).

We used the same ¼ 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. The <artifacts> were reduced and we were excited about this technique. <Looks> great, ship it, right?

However, when we integrated this into the shipping game and started looking at the results on our real destinations, the picture changed. Notice the <artifacts> in the zoomed picture. That was not acceptable as these artifacts were very noticeable when moving and were quite distracting.



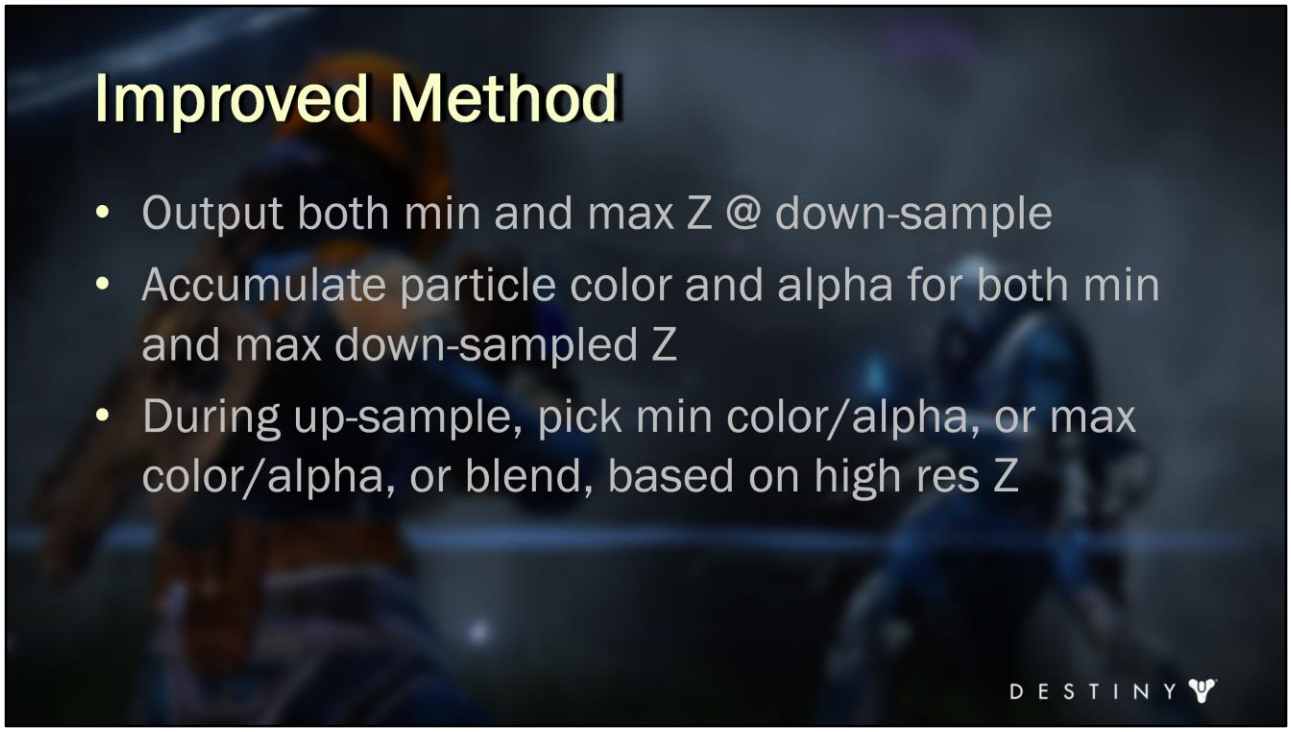
We then analyzed our technique to understand the reason behind these artifacts.
What didn't work?

<First>, Single blended Gaussian does not approximate well when particles are far apart

<Artifacts> are very noticeable in temporal domain

<For> performance reasons, we render particles in Z buckets, and we only have a small number of buckets (to keep performance cost down). This exacerbated the artifacts when particles were spread apart.

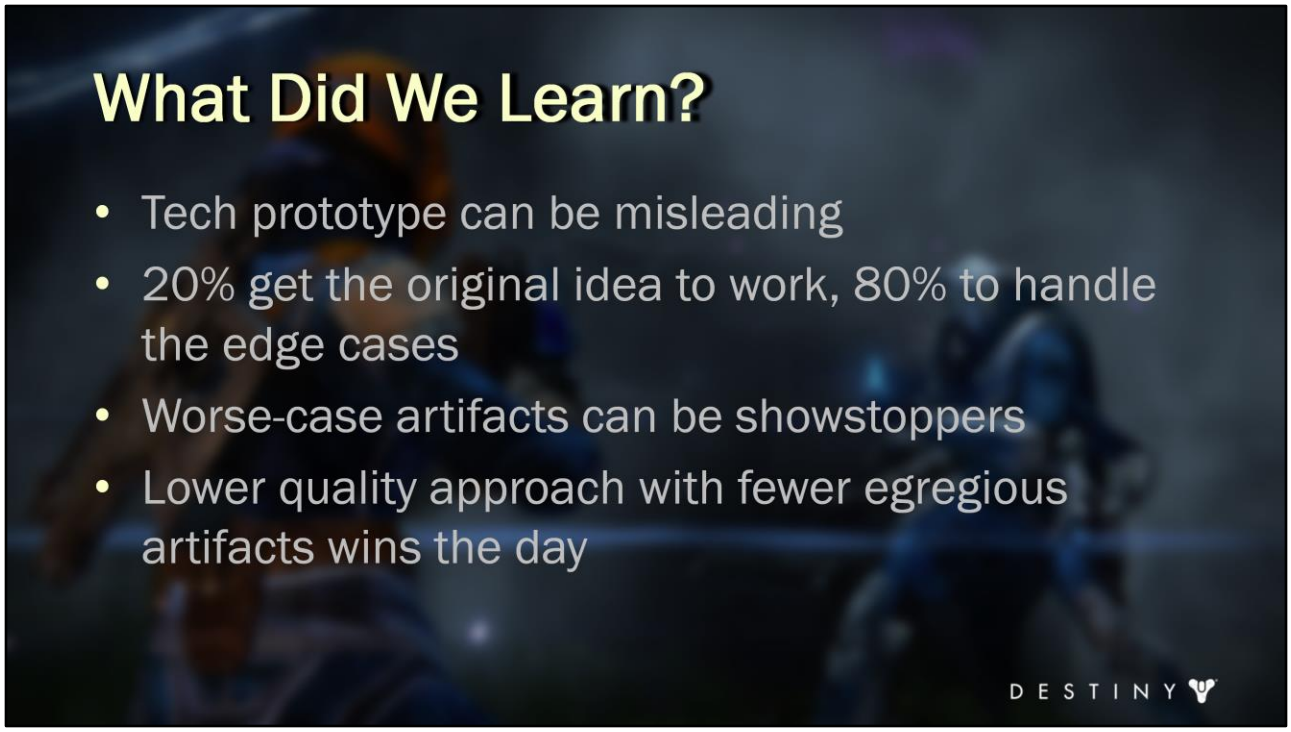
We had to go back to the drawing board.



The improved method we shipped in Destiny does the following:

- Output both min and max Z @ down-sample
- Accumulate particle color and alpha for both min and max down-sampled Z
- During upsample, pick min color/alpha, or max color/alpha, or blend, based on high res Z

Now we an look in the same scene and the artifacts were <not> present. Ready to ship!.

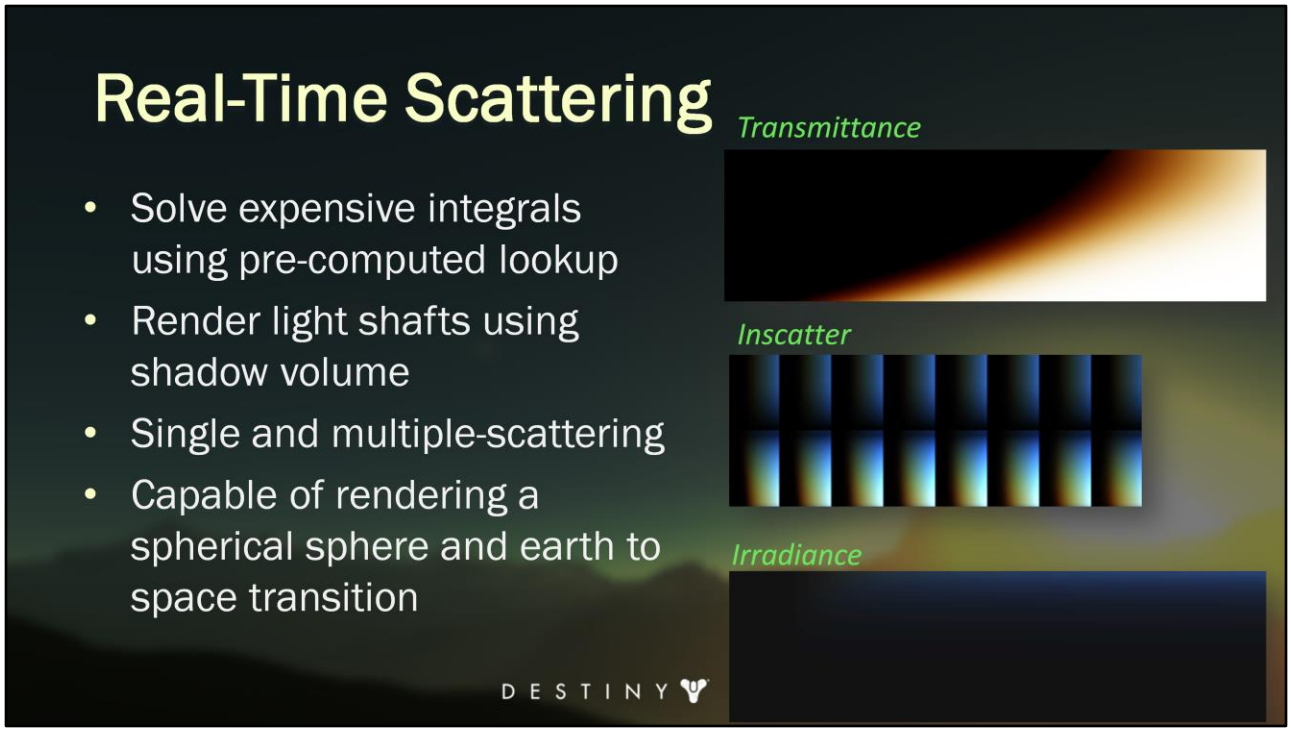


What Did We Learn?

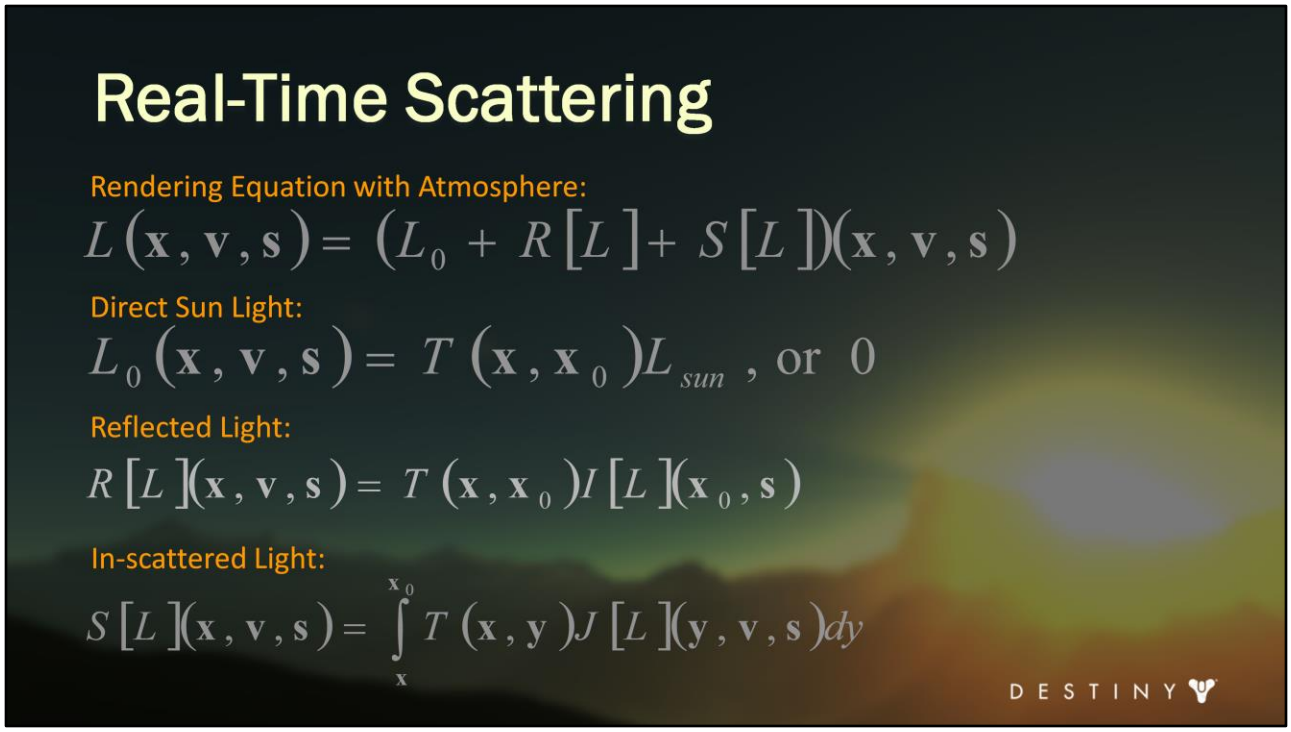
- <An initial> – even successful - tech prototype can be misleading. Even when early results are promising, it is important to explore variety of content to understand the edge cases for the algorithm
- <It> takes about 20% of the total time to get the idea to work, but the remaining 80% will be needed to handle the edge cases
- <Worse-case> artifacts can be prohibitive and cause you to need to significantly reformulate the entire technique
- <In the end>, we found that we prefer a slightly lower quality approach that has fewer egregious artifacts

Another case study from Destiny is our work on atmosphere rendering

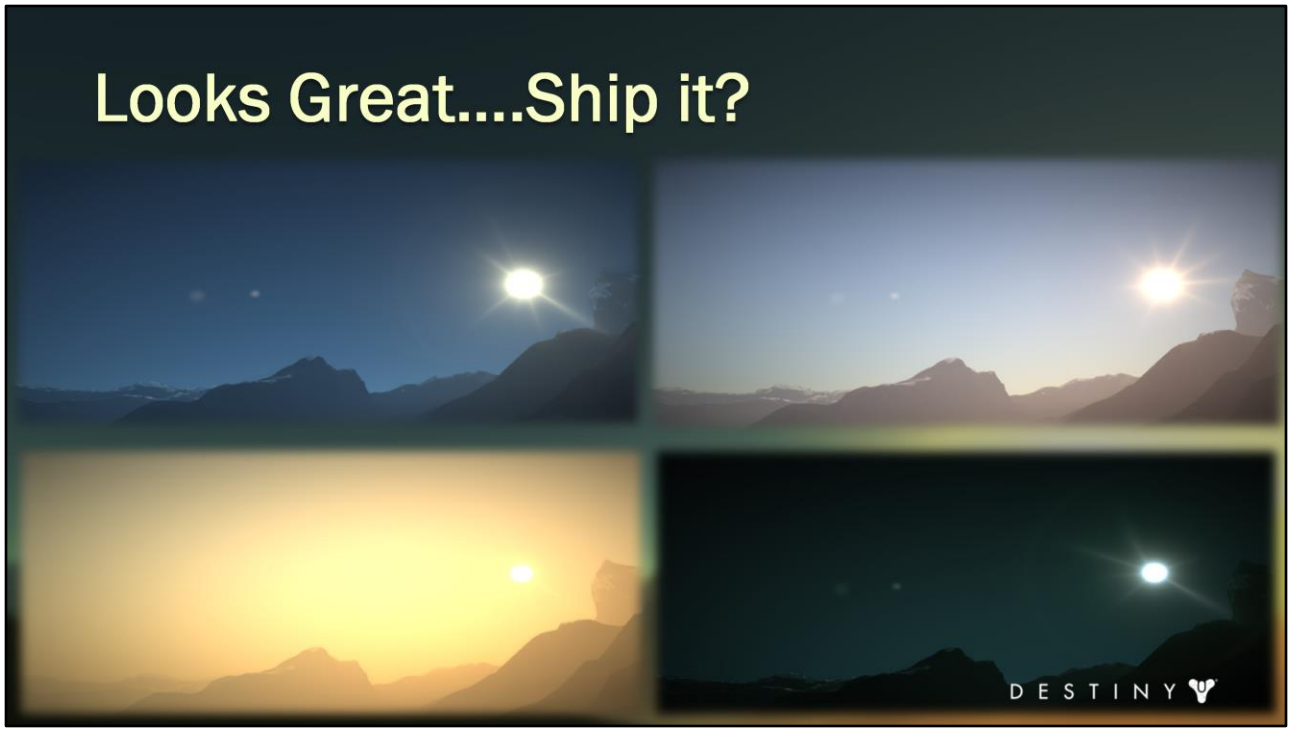
We started from implementing a technique from BrunetonNeyret 2008 in real-time. We covered the details of our implementation in the 2009 talk in Advances at SIGGRAPH 2009.



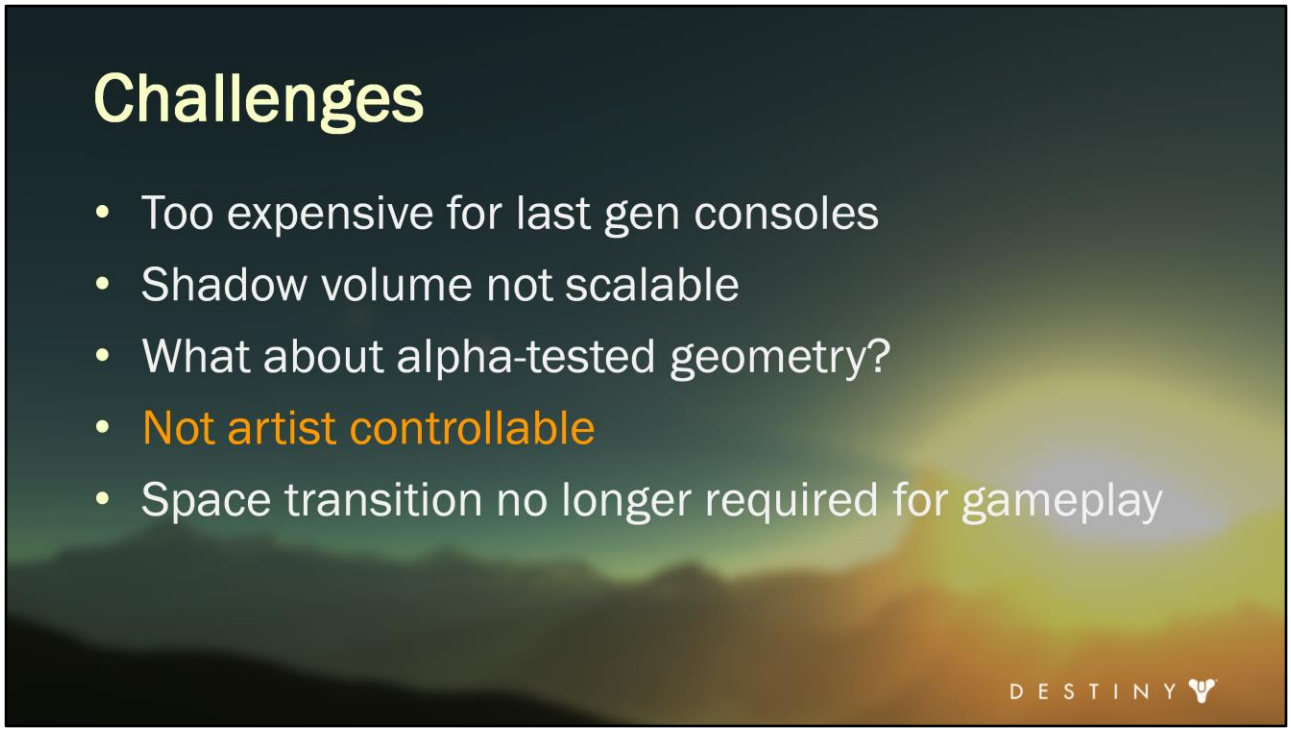
This included single- and multiple scattering effects. In our implementation, pre-computation could happen on the GPU for dynamic time of day support. This method was attractive to us as it included support for atmosphere viewable from space and for light shafts. At the time both were highly desired features for our game design and visuals.



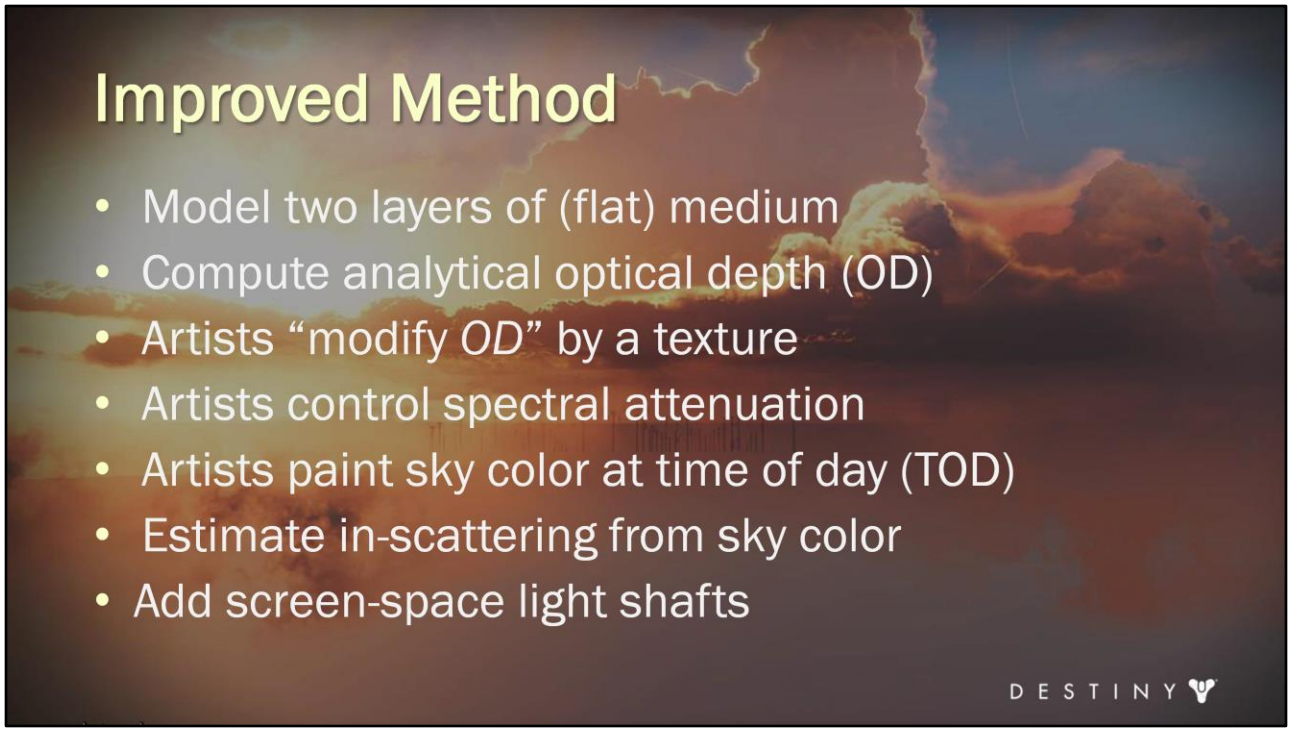
Here is the math for the computation of the atmosphere scattering that we used.



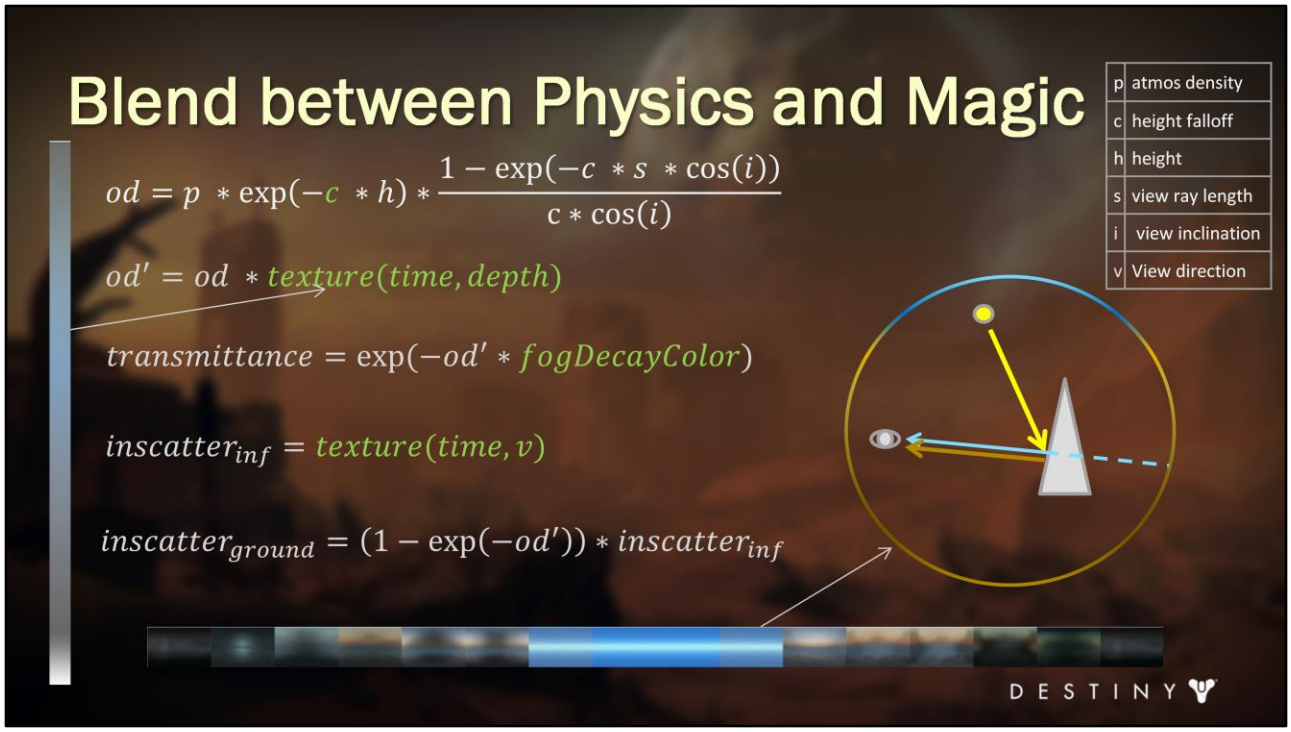
Our implementation allowed us to create realistic atmospheric effects with dynamic time of day, and both sun and moonlight conditions. We were excited. <Ship it?>



However, once we integrated this method into the shipping game we encountered several key challenges.
<First>, the technique was too expensive to compute for last-generation consoles. It had <problems> with the shadow volume scalability and alpha-tested geometry. But <far more> importantly, it did not offer intuitive artist-friendly controls – and Bungie has amazing skybox artists. We could not limit their creativity. And we also <learned> that space transition was no longer required for game play.
This shifted our perspective.

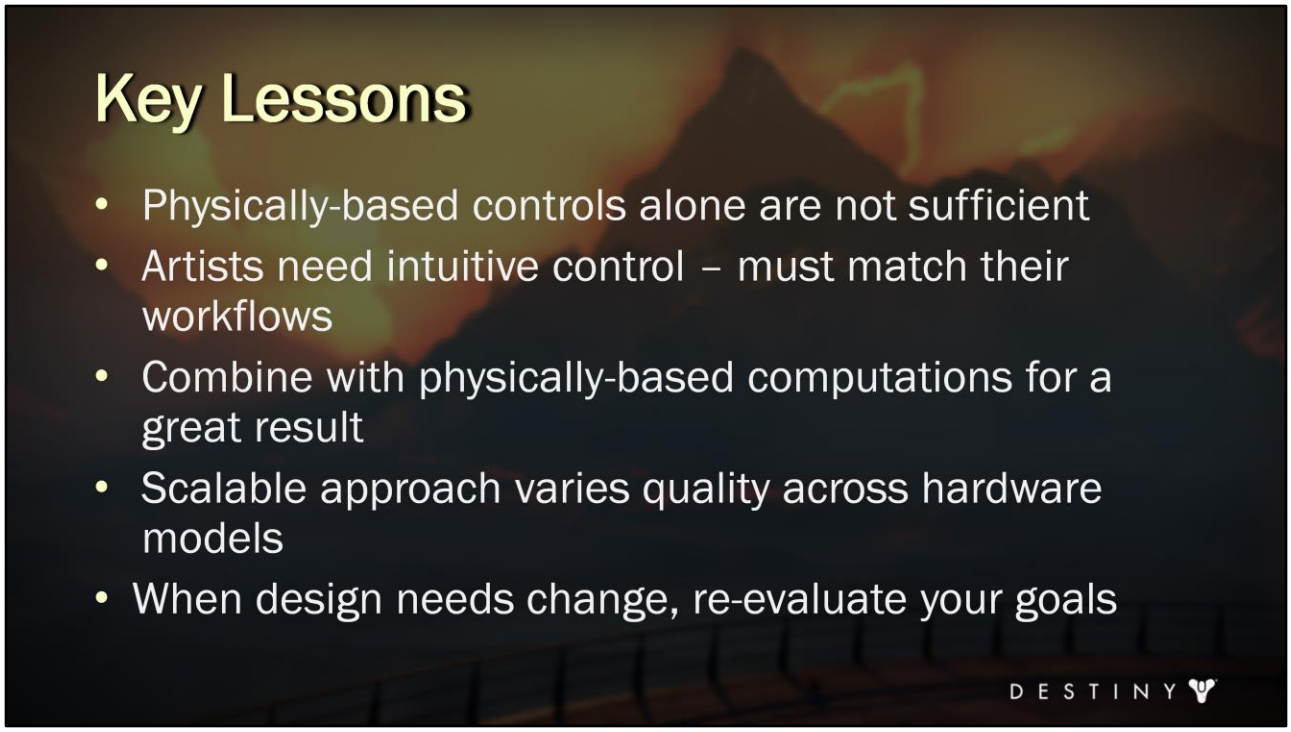


Our improved method accomplished similar goals to our original implementation in the following manner:
<Model> two layers of (flat) medium
<Compute> analytical optical depth (OD)
<Artists> “modify *the optical depth*” by a texture
<Artists> control spectral attenuation
<Artists> paint sky color at time of day (TOD)
<Estimate> in-scattering from sky color
<Add> screen-space light shafts



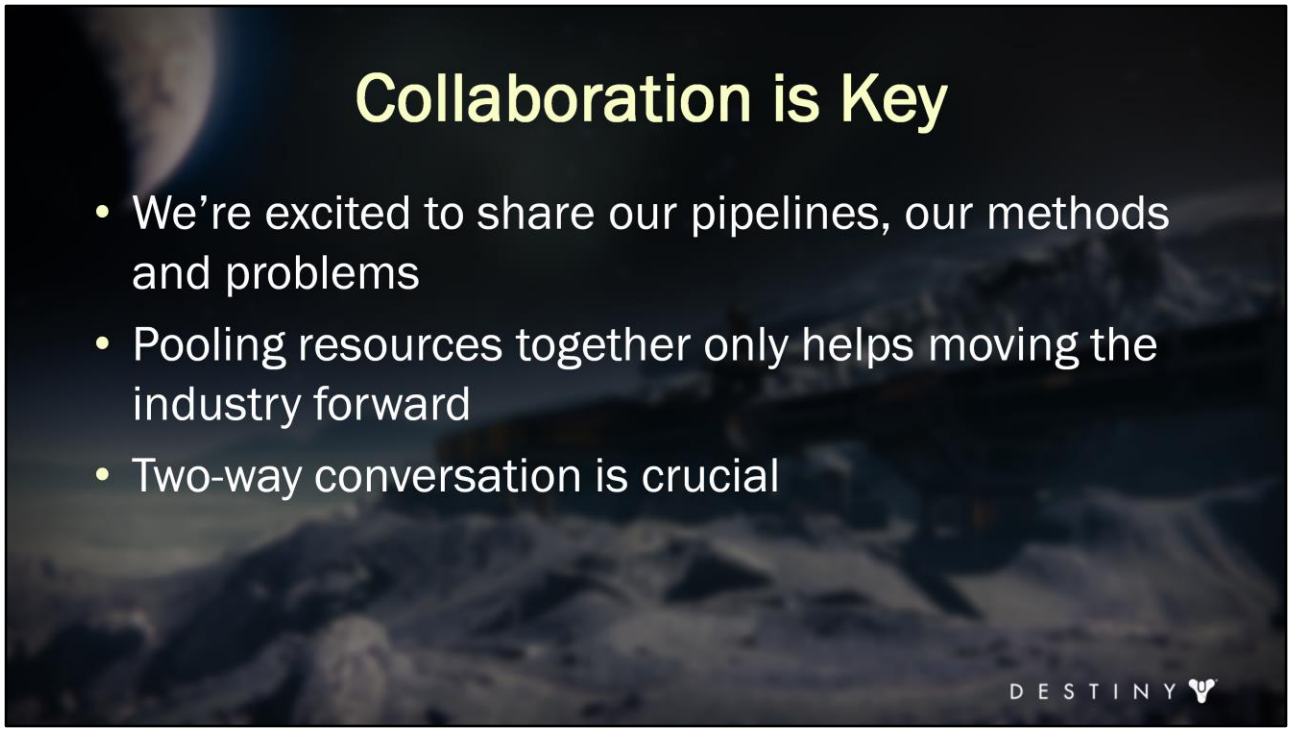
This of course necessitated that we blend between physically based approximation and a bit of magic. Here are the equations we used (will be distributed in the slides) and the textures painted by the artists.

This method was able to achieve great performance on lower end consoles and allowed us to achieve great visuals we were striving for.

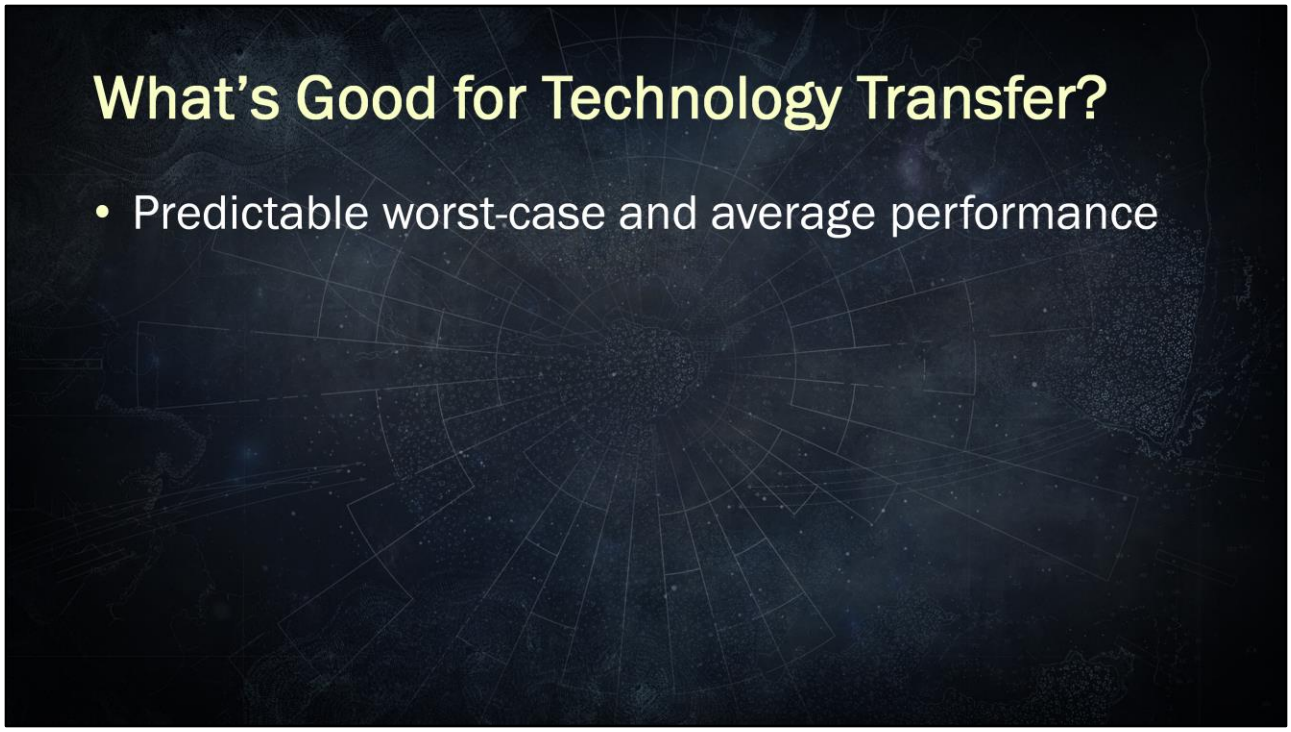


The key lessons we extracted from this foray were that:
<Physically-based> controls alone are not enough
<Artists> need intuitive control – must match their workflows
<We> must combine physically-based computations with art-intuitive techniques for a great overall result
<Having> methods that scale in performance *and* quality across hardware generations is crucial for a shipping engine. We're willing to give up some quality for faster performance on lower-end platforms.
<When> design goals change, it's a time to re-evaluate your goals

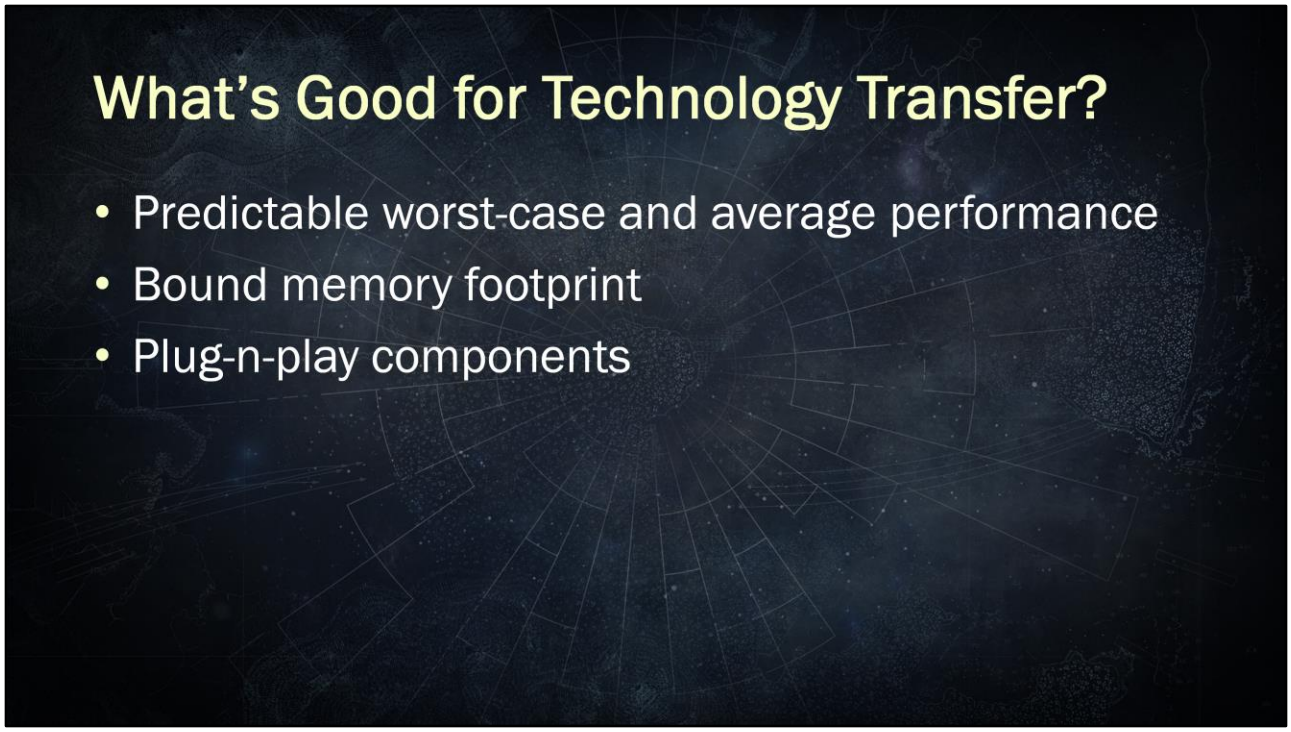
As you can see from the vast list of features, growing production costs and shortening timelines, game developers can use your help with research!



We need to collaborate more effectively with research. We're excited to share our pipelines, our methods and problems to educate the research community about the needs of the game developers.
Pooling resources together only helps moving the industry forward.
Two-way conversation is crucial – by having real problems to solve, research can stay relevant and game technology will move forward by leaps.

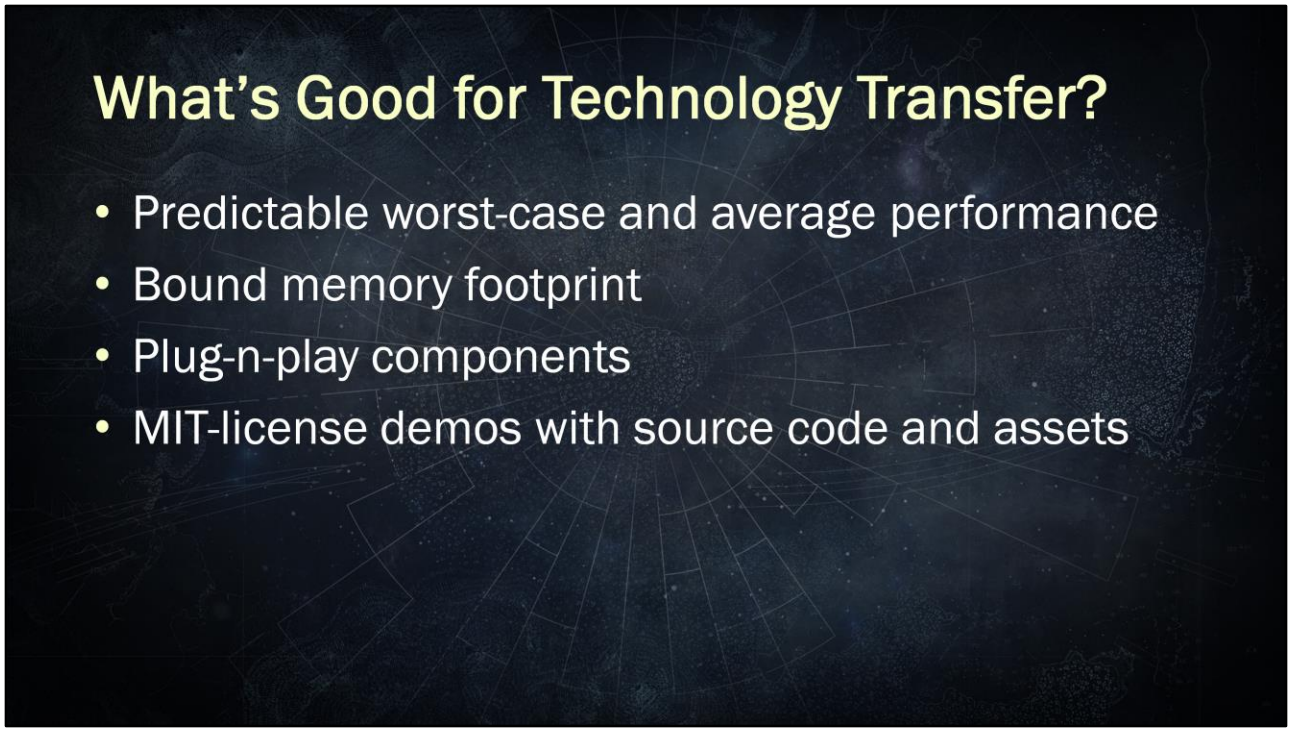


Aside from direct collaboration, what are the properties that make a research technique well-positioned for successful technology transfer to video games? <Having> predictable average and worse-case performance is very important. As you saw, we have explicit budgets for rendering, and knowing the cost of a particular feature can help the decision of whether it makes sense to integrate or not.

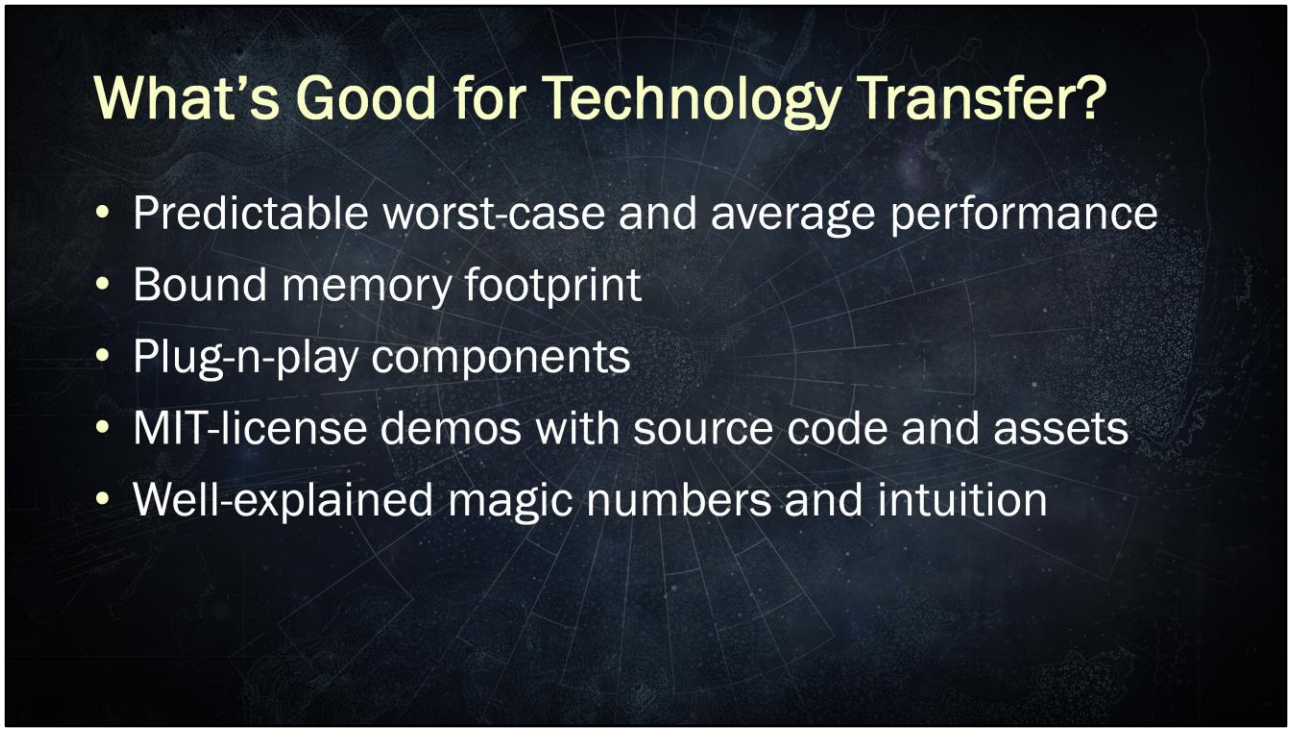


Having bound memory footprint is another key component.

<Game developers> are also always looking for plug-n-play stand-alone examples and systems. Great examples include demos with shader or mesh techniques (for example, the recent HDAO, FXAA examples) as these are easy to consume quickly and validate in the shipping game content. Having a working source code with example assets is a key component for practical research and adoption.

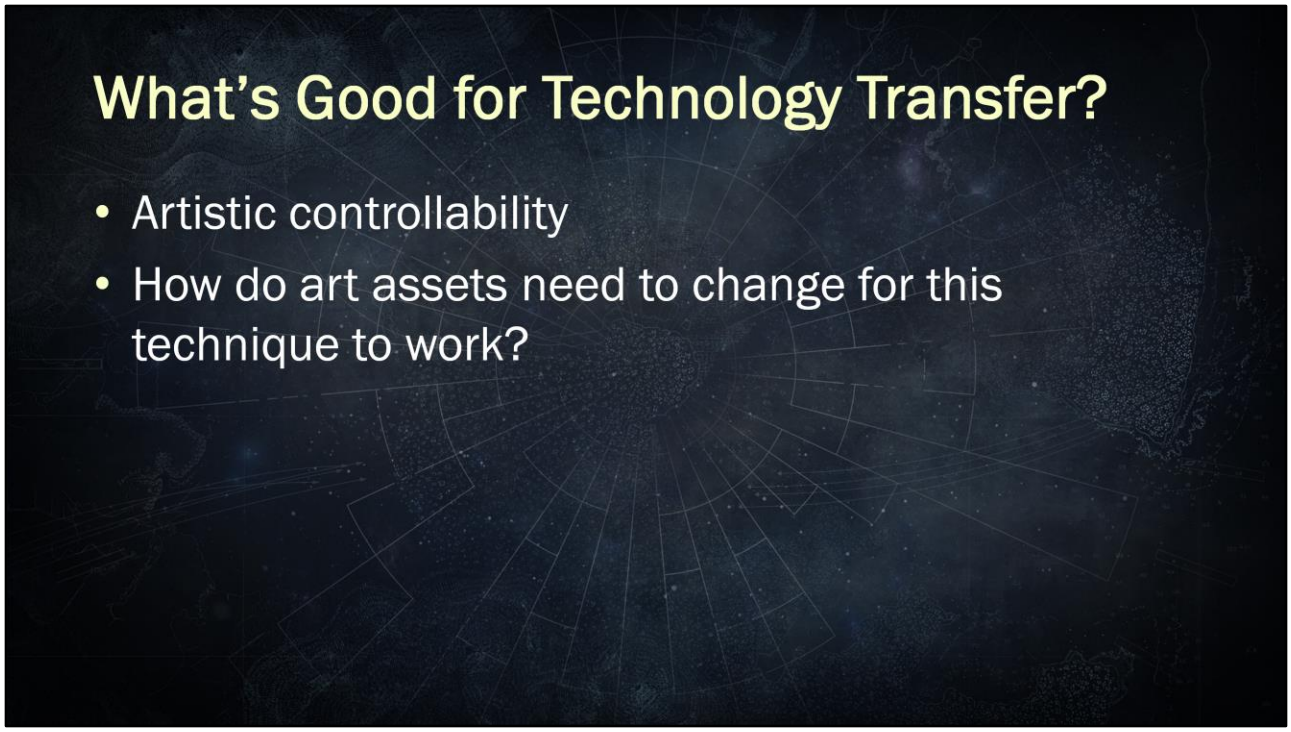


Distributing your demos with MIT license, having well-explained source code and sharing the assets you used would also aid game developers being able to explore your techniques. While I was working in applied R&D world, it was not unreasonable for me to spend 3-6 months or even a year on a single technique. In my game developer days, I usually have anywhere from 2 days to 2 weeks to integrate a feature. And this includes doing that for all of our shipping platforms.



And lastly, though this may be obvious, but it's fantastic when magic numbers used for a particular technique are explained well. This is what's usually covered by the 'is this paper reproducible' on the reviews, but often is ignored for demos.

- An in-depth blog post explaining the intuition behind the technique is fantastic to include
- We often want to understand the cost of iteration on a particular technique

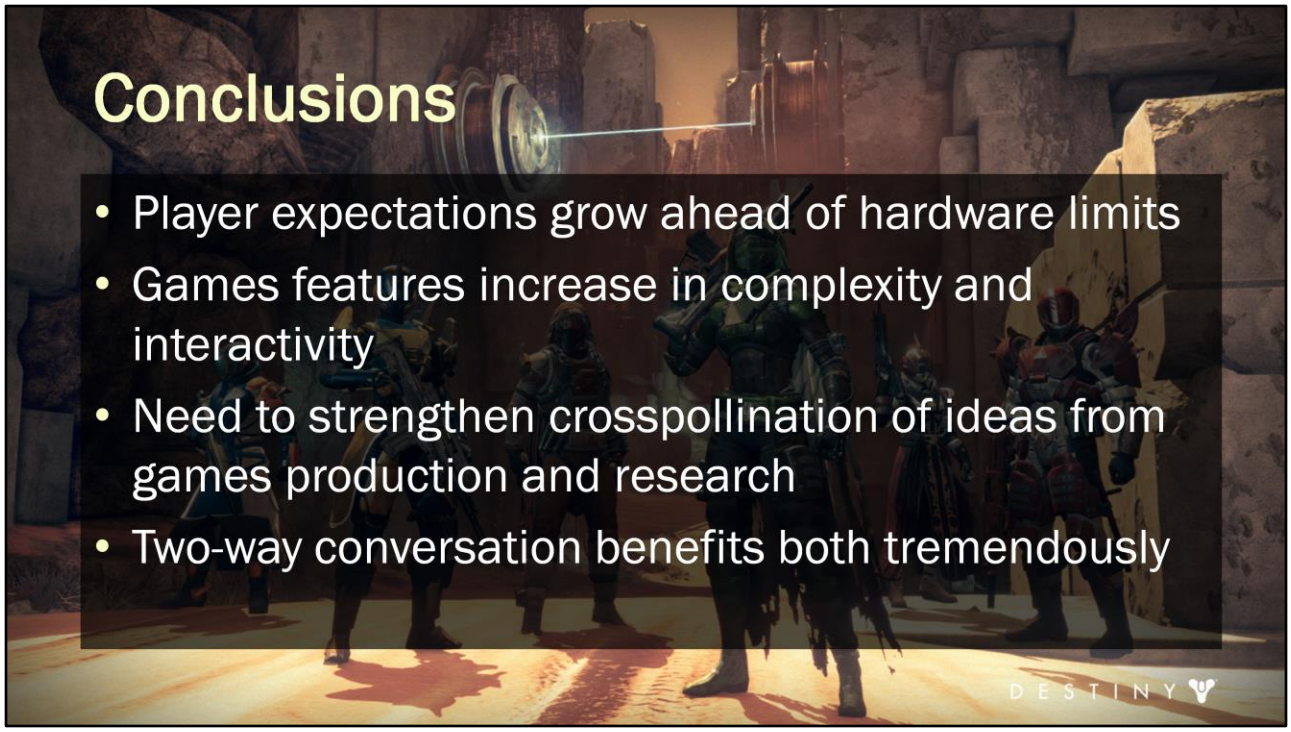


In order to know how well a particular technique fits into games pipelines, it's always helpful to understand how it needs to be integrated. <What> artistic controllability does an algorithm offer? How do existing assets need to change in order to support this method? Do new assets need to be created?

What's Good for Technology Transfer?

- Artistic controllability
- How do art assets need to change for this technique to work?
- Does it work with animated assets?
- What are the limitations? What are the edge cases? Where does the technique breakdown?

Understanding other constraints and limitations is helpful. <How> does the technique work with animated meshes? <Are> there artifacts? Even if there are, it's perfectly fine to distribute the technique – some of the artifacts are easily worked around in practice, whereas others are more difficult to avoid.



In conclusion, the game industry is not slowing down – players expect more immersion and fidelity in the coming years!
Games algorithms increase in complexity and feature richness require more collaboration with research
Two-way conversation benefits both tremendously. Talk to game developers and let's engage more closely to benefit both industry and research!


WE'RE HIRING
7
WWW.BUNGIE.NET/CAREERS
CAREERS@BUNGIE.COM


Questions?
DESTINY

Slides

- http://advances.realtimerendering.com
- Twitter: @mirror2mask