GEOMETRY AND MATERIAL REPRESENTATION IN GAMES

LESSONS LEARNED AND CHALLENGES REMAINING

Wade Brainerd: ACTIVISION
Natalya Tatarchuk: BUNGIE

Render the Possibilities
SIGGRAPH2016

Structure

- Geometry representation principles
- Destiny character creation pipeline
- Artist feedback about workflows
- Deep manually-driven pipelines are error prone
- New representations using hardware tessellation
- Call to action: Make a better tessellator
- Call to action: A separable pipeline

BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

The purpose of this presentation is to establish basic principles of geometry representation in games, and to set goals for academia to develop new representations and processes that improve our ability to generate content for games.

For our research process, we conducted a survey of artists at our own studios as well as representatives from other AAA game and film studios. We have summarized the responses and will present that data here.

We will describe elements of our pipelines that we feel have potential for improvement, and post-mortem attempts to use new content representations in our games.

We will propose a hardware change that could improve our ability to utilize hardware tessellation to enable new content representations in our games.

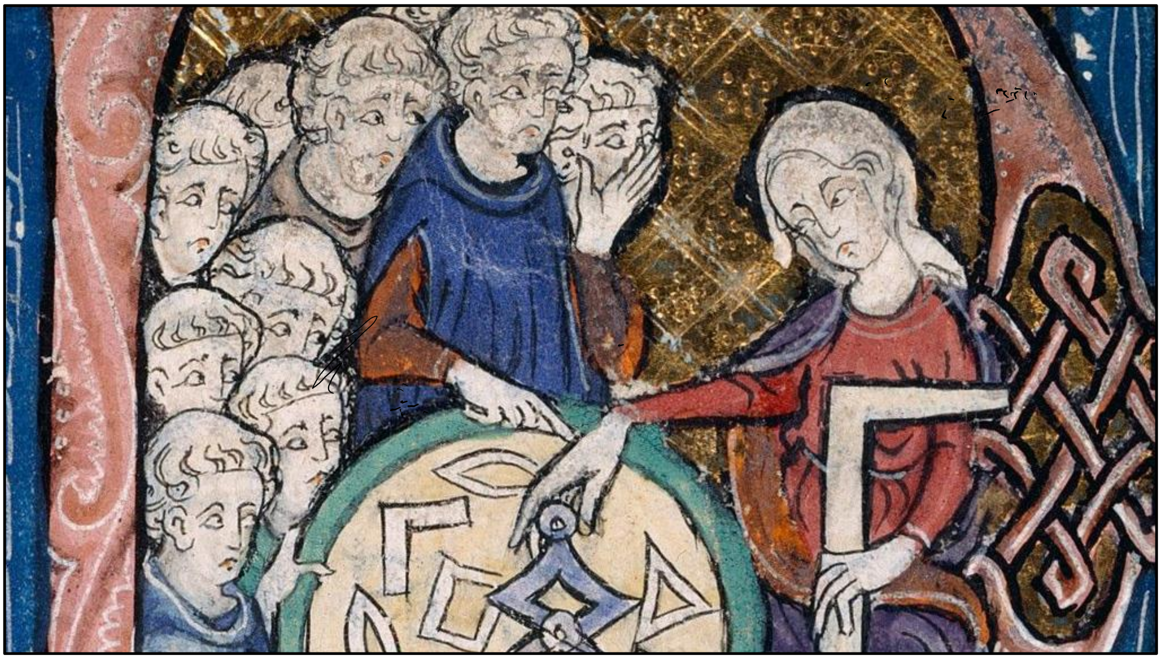
Finally, we will propose a new high-level model for 3D content generation where the high-resolution asset is developed entirely separately from the real-time representation.

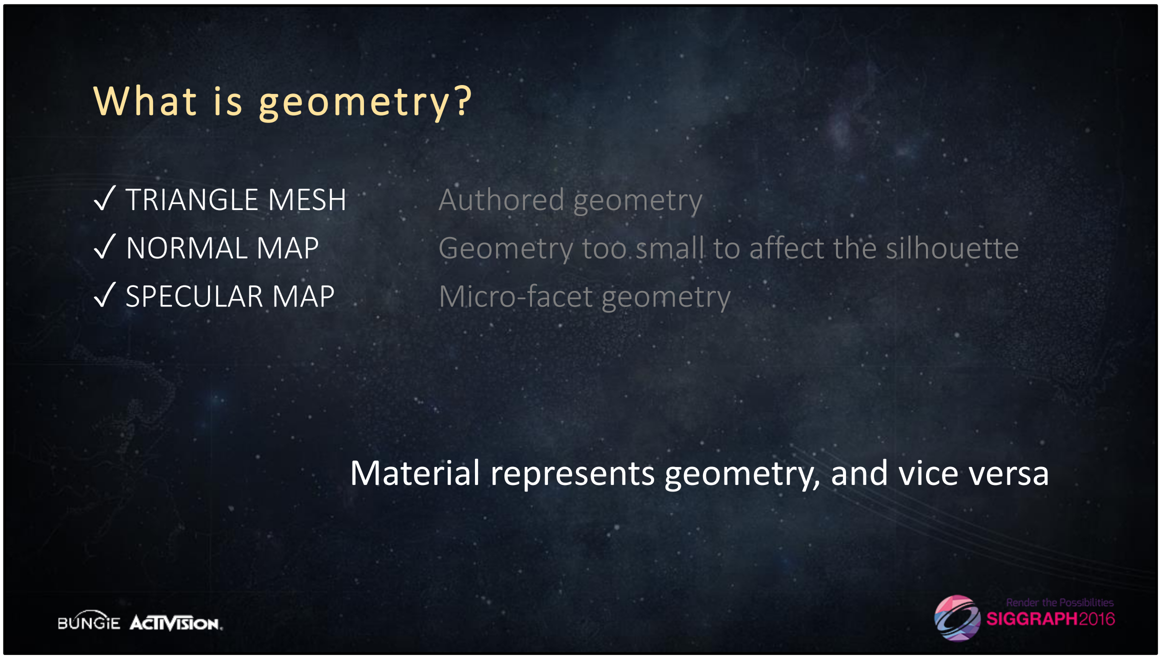
DESTINY

CALL OF DUTY

Given we both work at AAA game studios developing first person action games, we’re going to focus on that perspective, but expect that that our conclusions transfer to other situations.


Geometry: Principles
BUNGIE
ACTIVISION
Render the Possibilities
SIGGRAPH2016





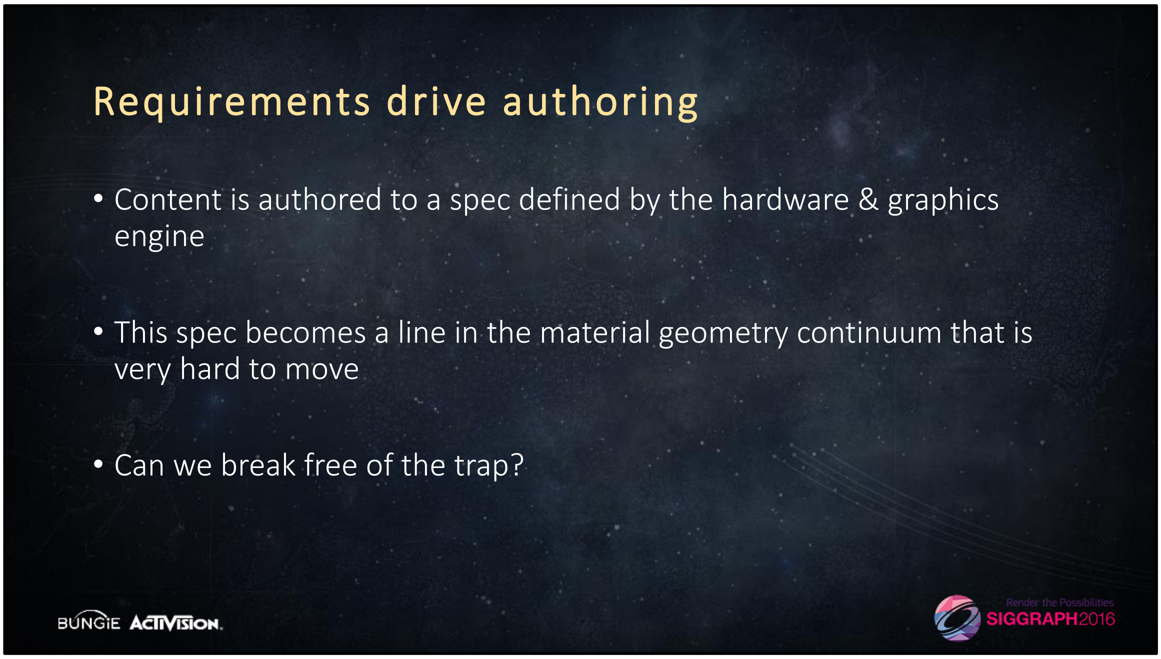
We want to begin by establishing what 3D geometry is, in principle, from the perspective of game content creation.

Authored triangle meshes are the most obvious geometry representation, but aspects of material authoring are also geometry. For example, normal maps represent geometry that is too small to author as triangles. Specular maps also represent a statistical distribution of geometry at an even smaller scale.

The geometry of an object is actually a representation of the materials it is made of. Likewise, a material is actually a representation of small geometric features.

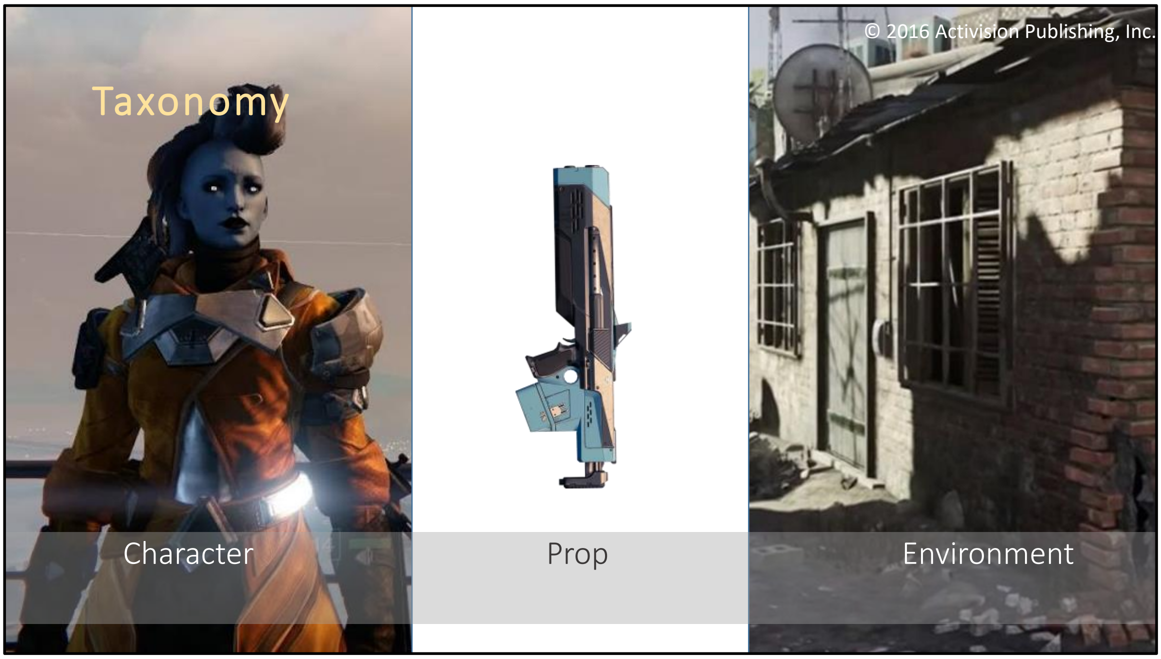
In 3D content, geometry and material are a continuum, representing the physical properties of an object at different scales.

When designing a 3D renderer, we draw a line in the sand - through the material / geometry continuum. It’s chosen given the capabilities of the hardware and software, and the desired level of detail.



This line dictates the requirements of the content creation pipeline. If content needs to be scalable, multiple representations may need to be created. But once the content is created, it is impossible to move the line.

Efficiency might improve if we could make the best choice for construction, and derive the runtime representation(s).



For the purpose of discussion, we divide geometry and material production into three categories: characters, props and environment.

Characters are produced on the order of tens to hundreds unique assets per title, are rigged and animated extensively, and utilize discrete LOD.

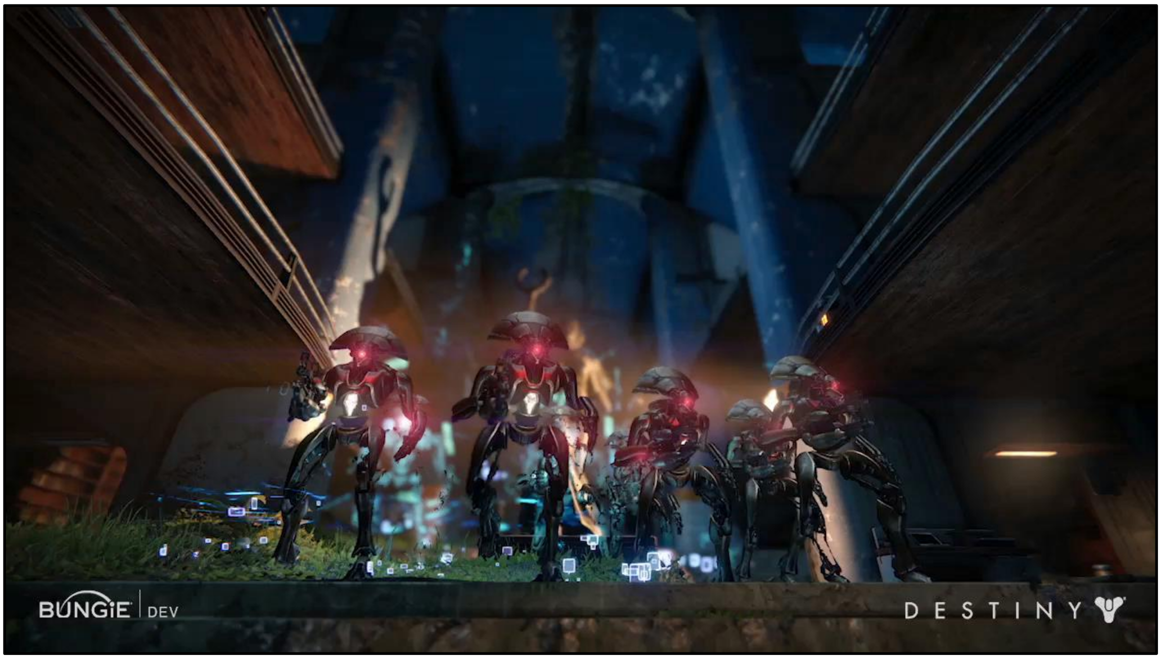
Props are produced of thousands unique assets per title, sometimes have moving parts, and also utilize discrete LOD.

The environment is a single continuous asset per scene, only animates in superficial or highly scripted ways, but importantly requires continuous LOD. Typically there are tens of levels or destinations in a game.

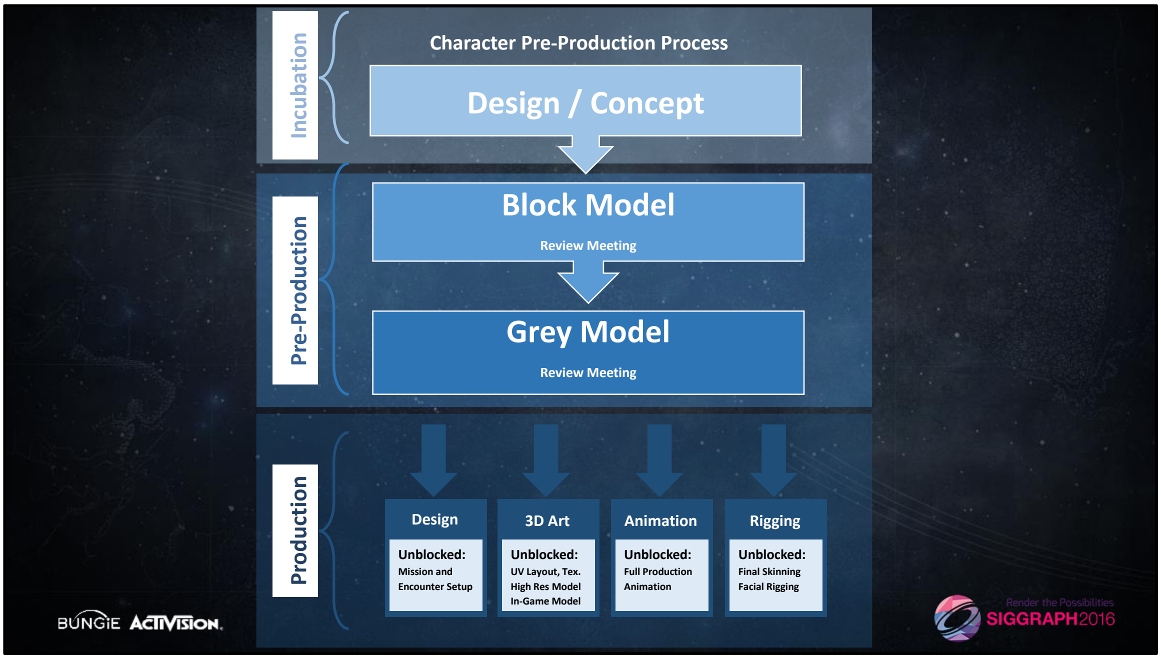
We chose to focus our attention on challenges in character and prop production- environment challenges tend to be application specific, and already been given lots of research attention.

Let’s take a look at the development process for Destiny characters

Let's look at an example of our character creation pipeline to create these friendly fellas, the Vex Goblins, an alien race in Destiny (here seen rendered in game).



Here they are in action, stomping around their native Venus and shining their robot eyes at us.



At a high level character design process roughly fits three stages shown here – incubation, preproduction, and production.

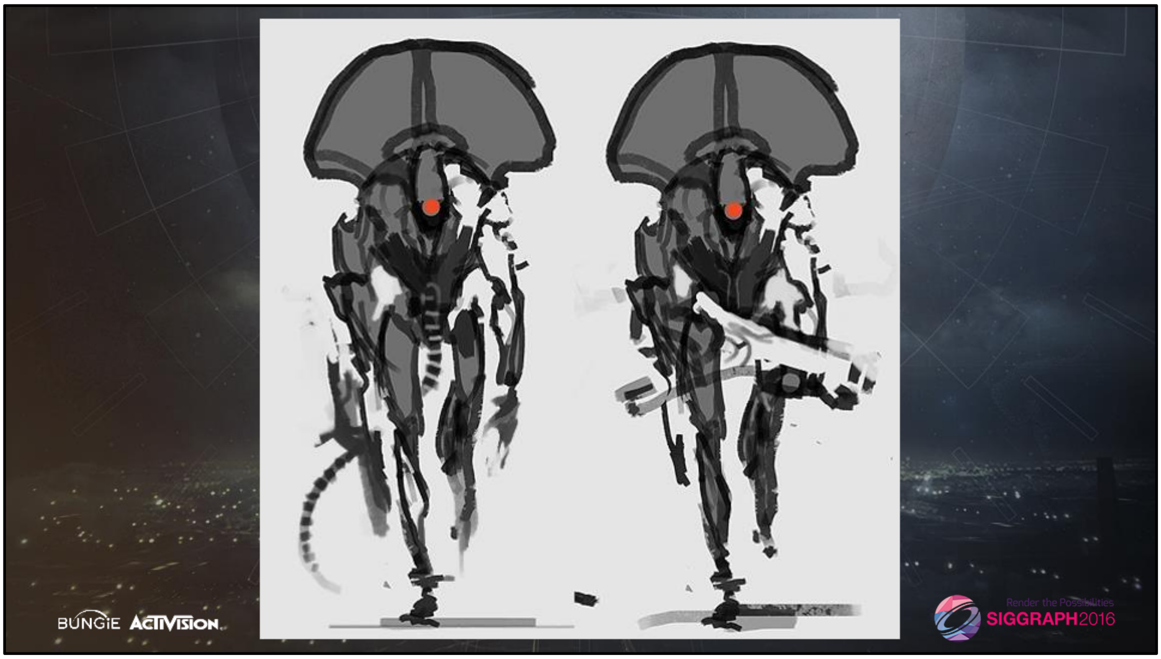
They break down into **design/concept**, **block model, grey model** explorations and final **production** phases.

Each stage roughly corresponds to a set of people that are involved in making the assets.

During the first phase, *design / concept*, the designers, creative direction, art direction, concept art and fiction representatives are all exploring the *concept* of a particular character.

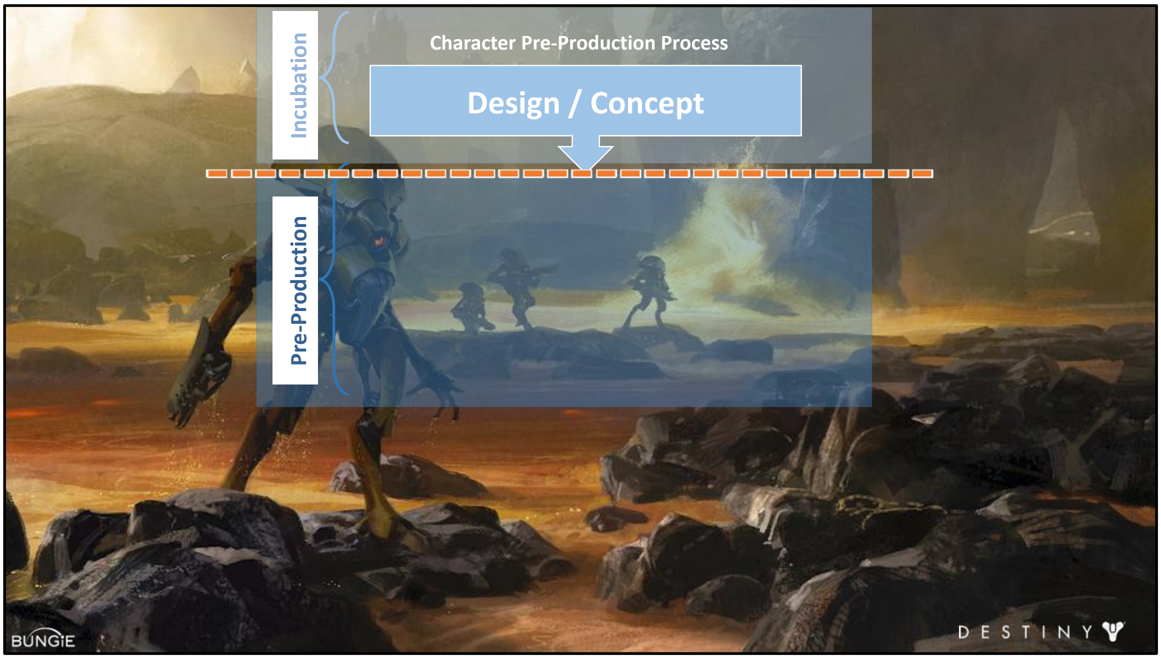
As the character goes through this phase, each review for its design at Bungie includes stakeholders from concepts, 3D art (modeling, shading and texturing), Animation and cinematics representatives, rigging, FX, and the corresponding engineering owners) in order to understand the implication behind feasibility of certain concepts.

Note that no actual work has been done on the character's in-game assets at this point: no modeling, texturing, etc. All of the discussion is done entirely with concept art.



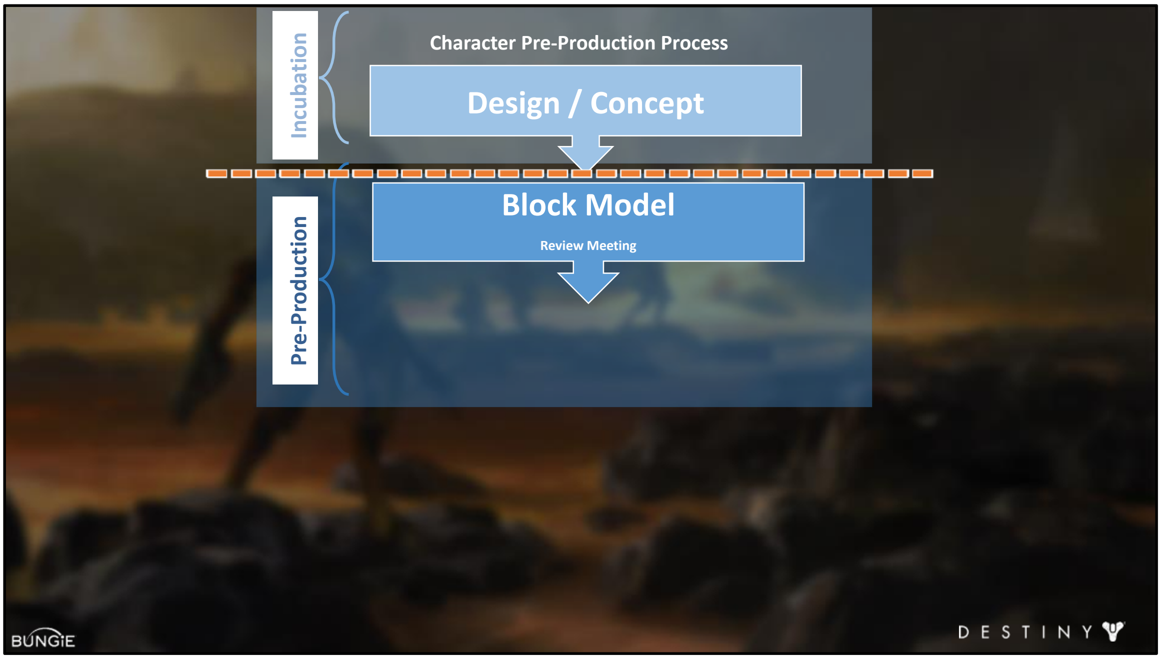
So here is an example of such concept work for this character design. Here's an early Vex sketch to explore this character's silhouette, significant visual features (the eye, the shape of the head, for example), and early proportions.

We start getting a sense for the proportions, materials, and rough idea of the concept for their movement at this phase. We also get a feeling for the gameplay design for these characters.

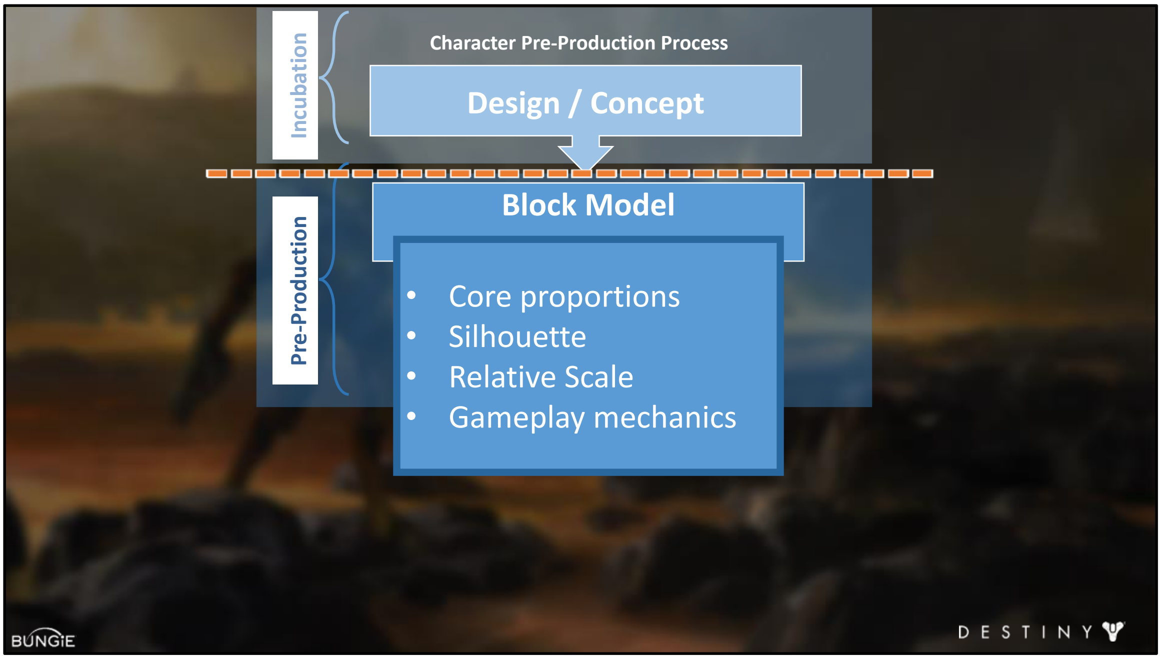


Once the concept phase completed, the concept art department moves on to a different character, and the next phase starts. The concepts for this character are locked.

Then preproduction phase begins.



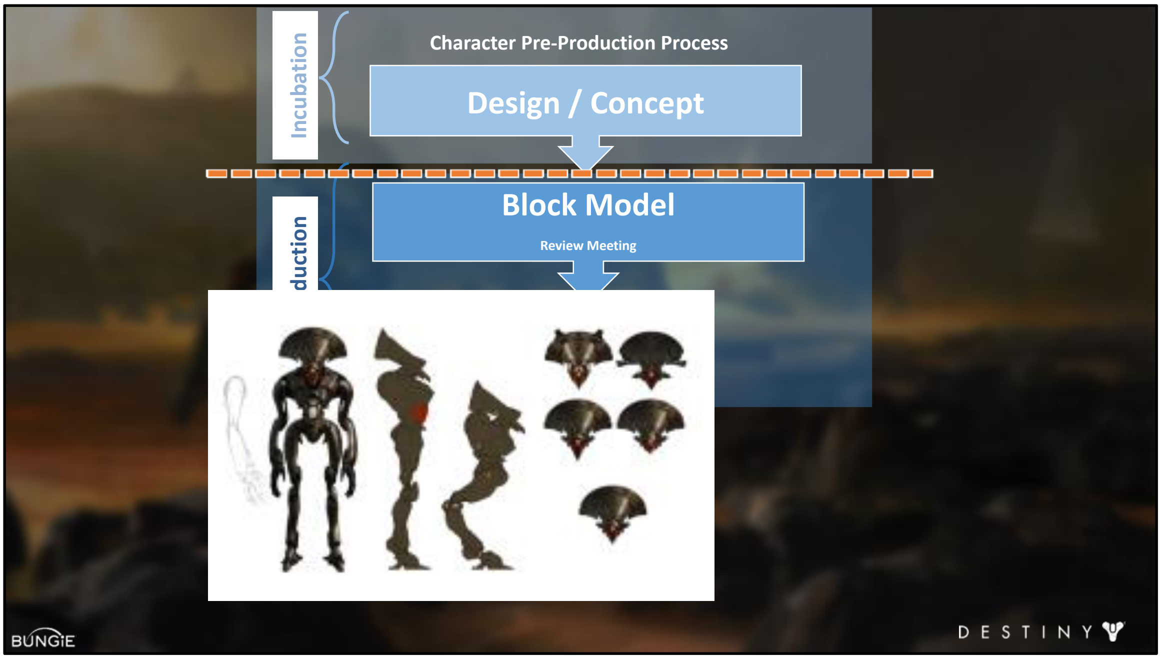
During the first pre-production phase for an asset, the **block model** phase, a further exploration **loop** begins for character design. This loop involves design representative(s), 3D art (primarily the modeler), animation / cinematics and rigging artists.



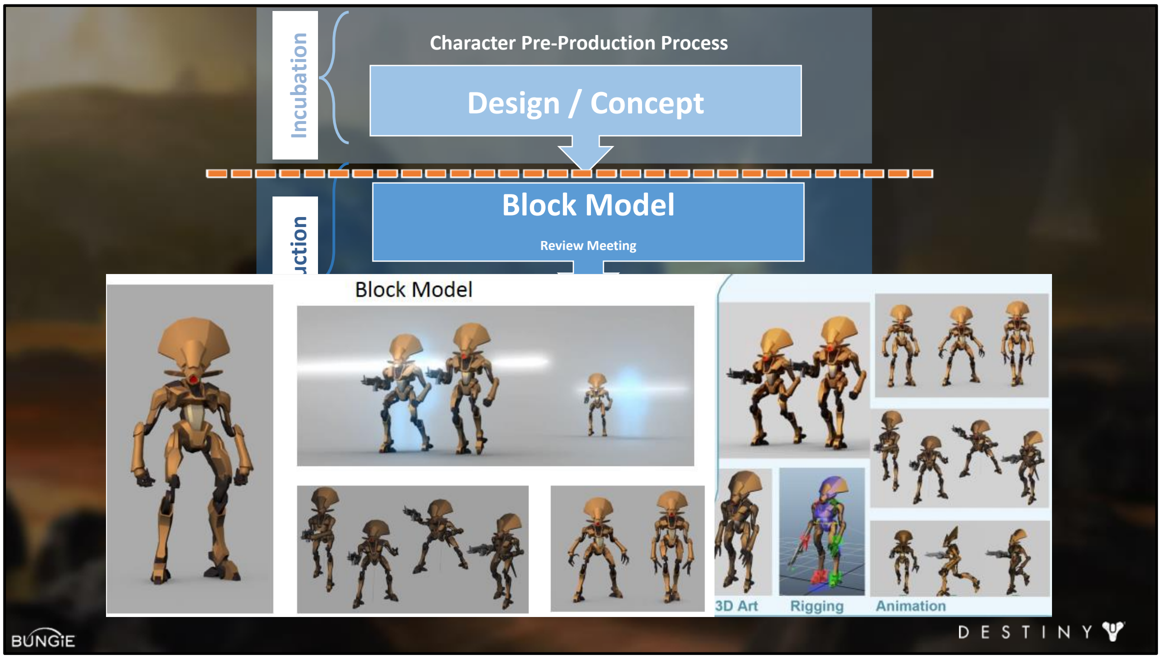
The goal of the **block model phase** is to identify *the core proportions of the character*, *define its distinguishing silhouette*, its *relative scale in game*, and, of course, *the gameplay mechanics.*

A note about animation and rigging – why are we talking about it in the geometry and material representation challenges? Although these elements are not important for static environment or props, they are **paramount for character design**

- **Modelers often iterate with riggers** to adjust the model for the desired range of animations
- Character design incorporates visual concept as well as movement concept
- At Bungie this is done during the grey model iteration loop before any expensive work has been done on high resolution assets. The reason for that is if, based on animation and rigging review, you realize that the characters' proportions need to change, or its geometry needs to differ, these changes are cheaper to implement (you don't need to change the high resolution Z brush model, rebake all the textures, or adjust all existing production animation sequences at this point, none of that has been created).



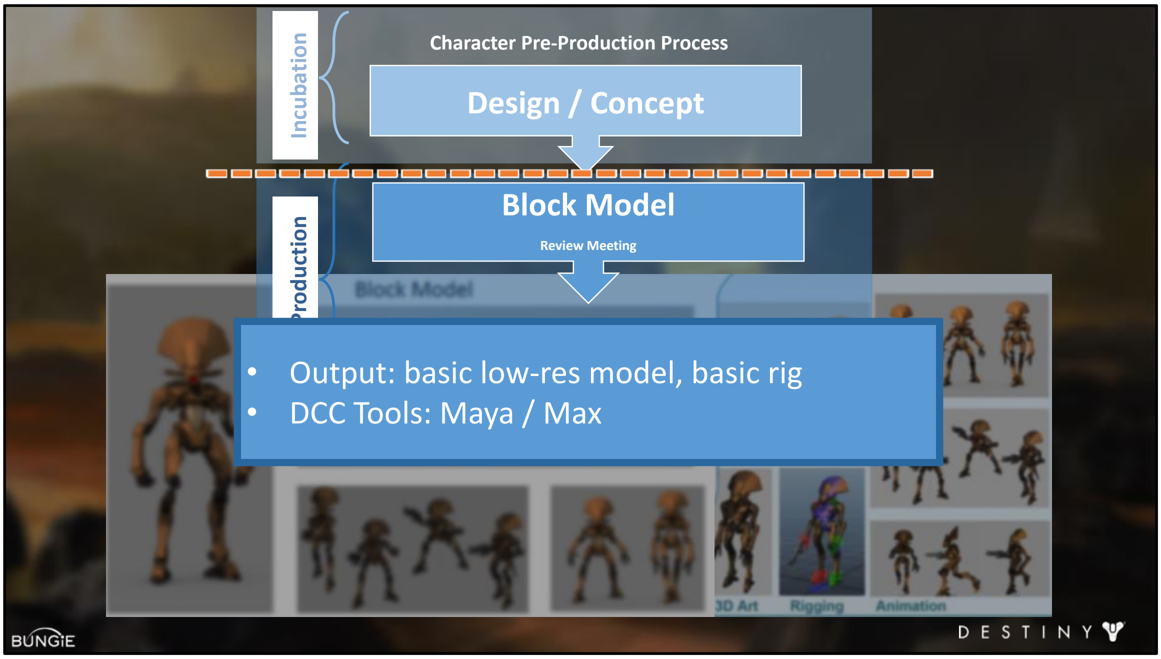
The Vex Goblin, the little fellow we're exploring here, went through several iterations of the block model phases for Destiny with the help of concept art doing paint overs the existing low res model to **refine the overall silhouette shape**. We also investigated a knee shape that could bend backwards during the crouching defensive state.



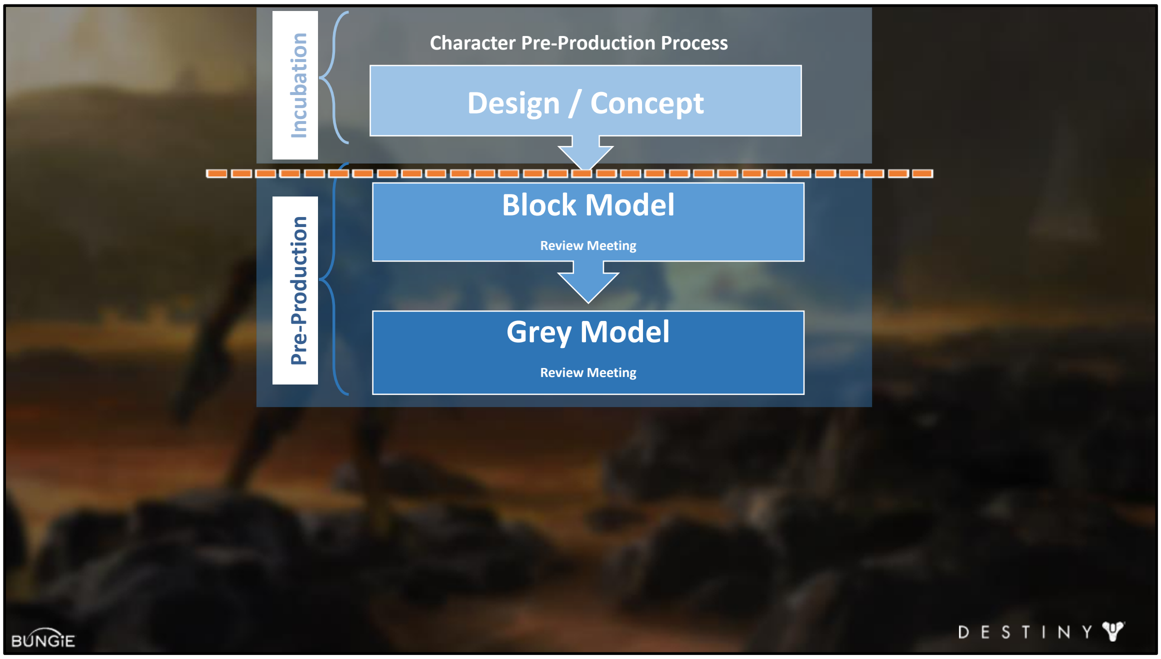
The actual block model looked like this in game. During the block model phase we focus on:

- Defining the skeleton / bones for this character
- Creating low resolution polygon objects to represent character volumes for each bone segment
- Testing that the silhouette is well defined and readable
- Using pose testing to identify range of motion

The most important thing at this stage is to define and lock down the core joint positions and rough geometry mass-out for the character.

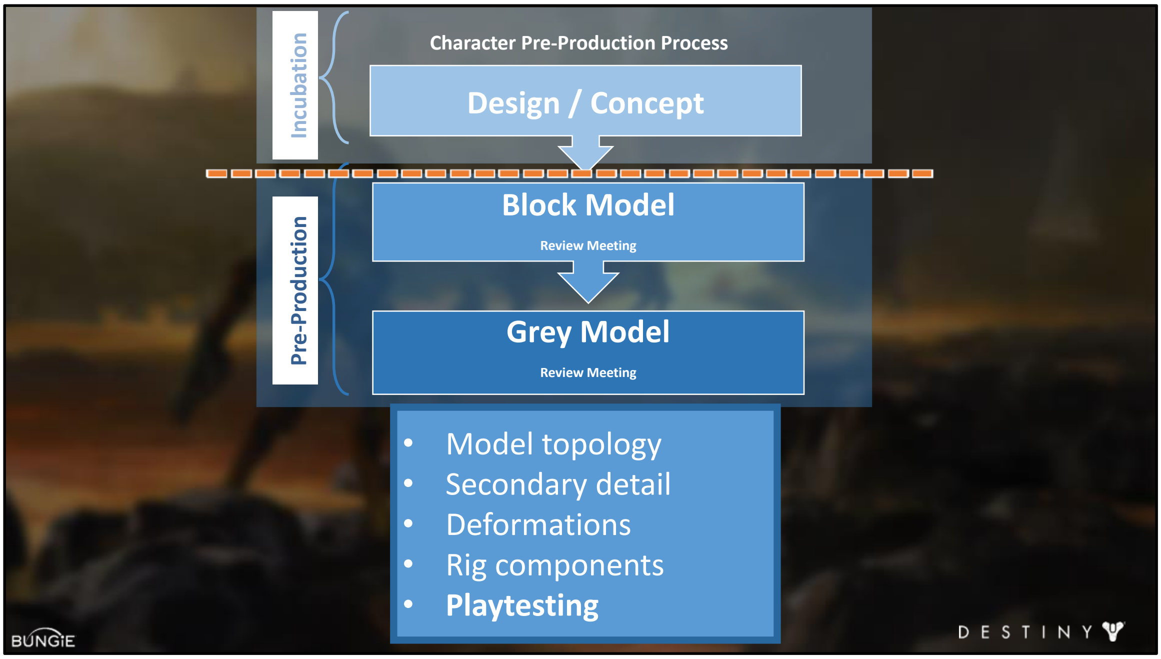


At this point, art works primarily in Maya for characters and Max for hard surface objects, and the goal is to create a very simple, roughed out – block – model. There is no texturing or any other expensive operations in place until this phase is complete. Riggers do a simple rig on the object to explore the movement, but it's all super basic at this point. This is a relative inexpensive phase still, in terms of production cost.

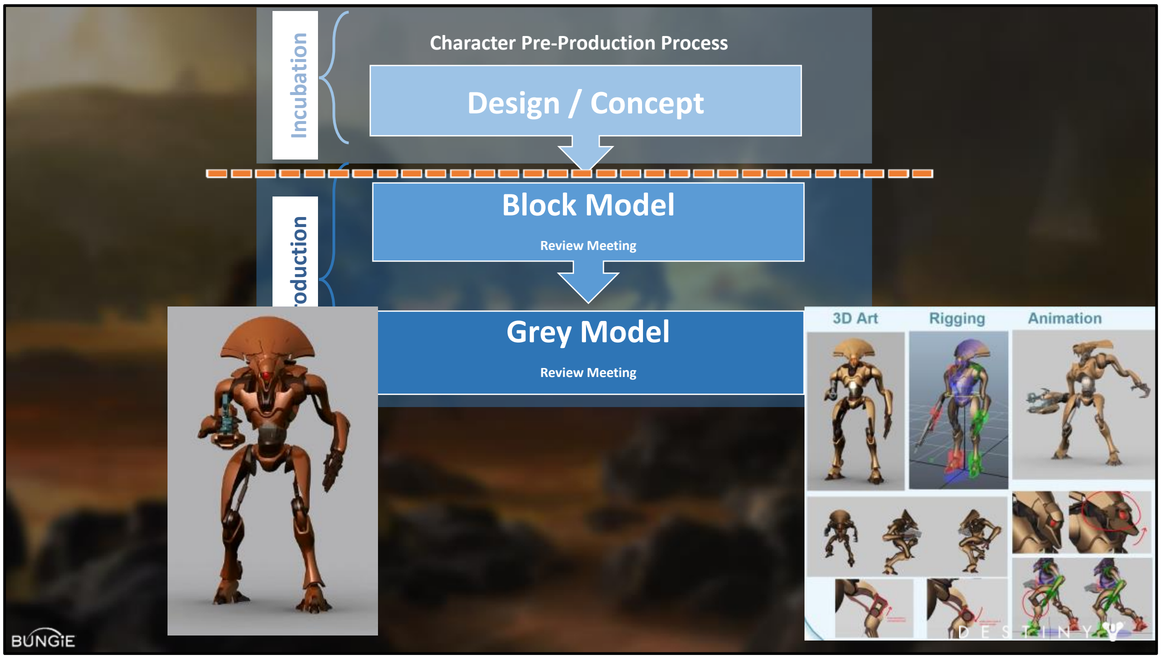


Next we enter the **grey model phase**

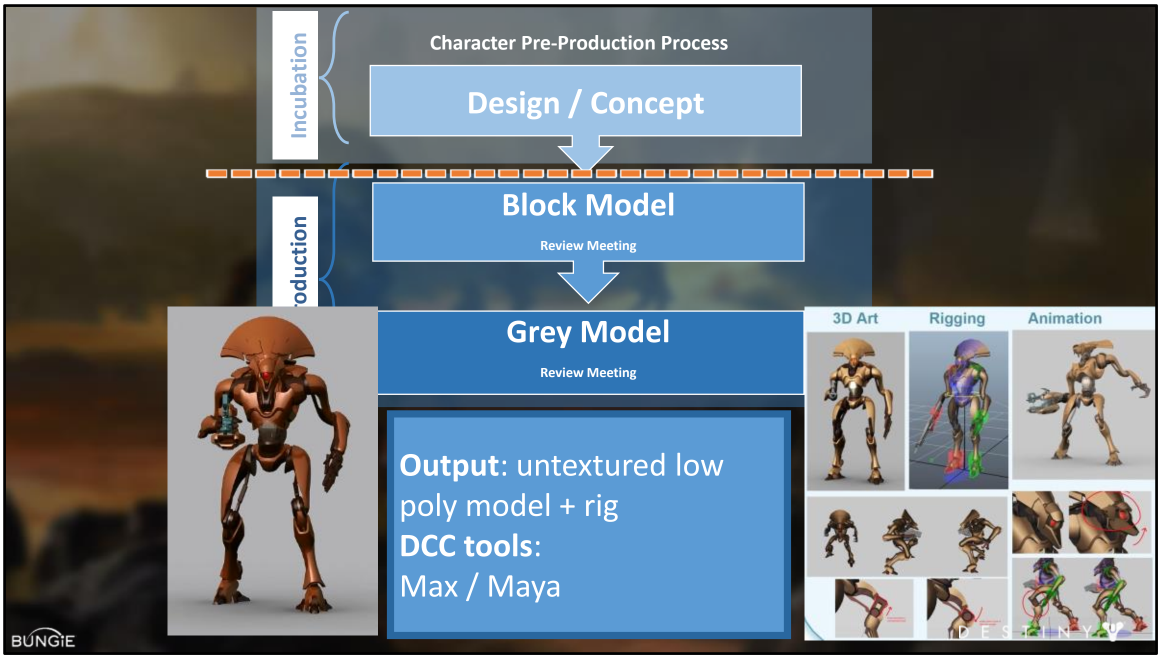
The goal of the grey model phase is to define the **Model topology**, identify secondary details, deformations, the major rig components. This is also where initial playtesting occur. How does this model feel in game? The goal of this phase is to identify all important changes to the character design before all the expensive ***production*** work began on the model (the high resolution modeling, texturing, etc.). This **iteration loop** is still relatively cheap and allows us to understand important decisions that can impact production of high quality assets.



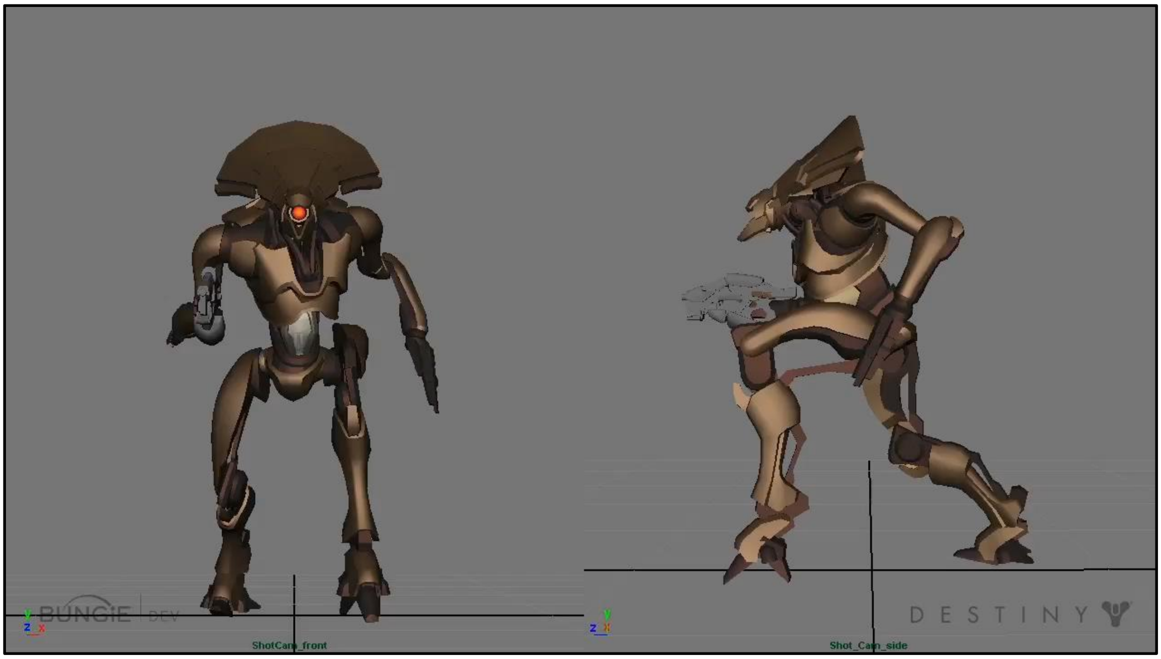
The goal of the grey model phase is to define the **Model topology**, identify secondary details, deformations, the major rig components. This is also where initial playtesting occur. How does this model feel in game? The goal of this phase is to identify all important changes to the character design before all the expensive *production* work began on the model (the high resolution modeling, texturing, etc.). This iteration loop is still relatively cheap and allows us to understand important decisions that can impact production of high quality assets.



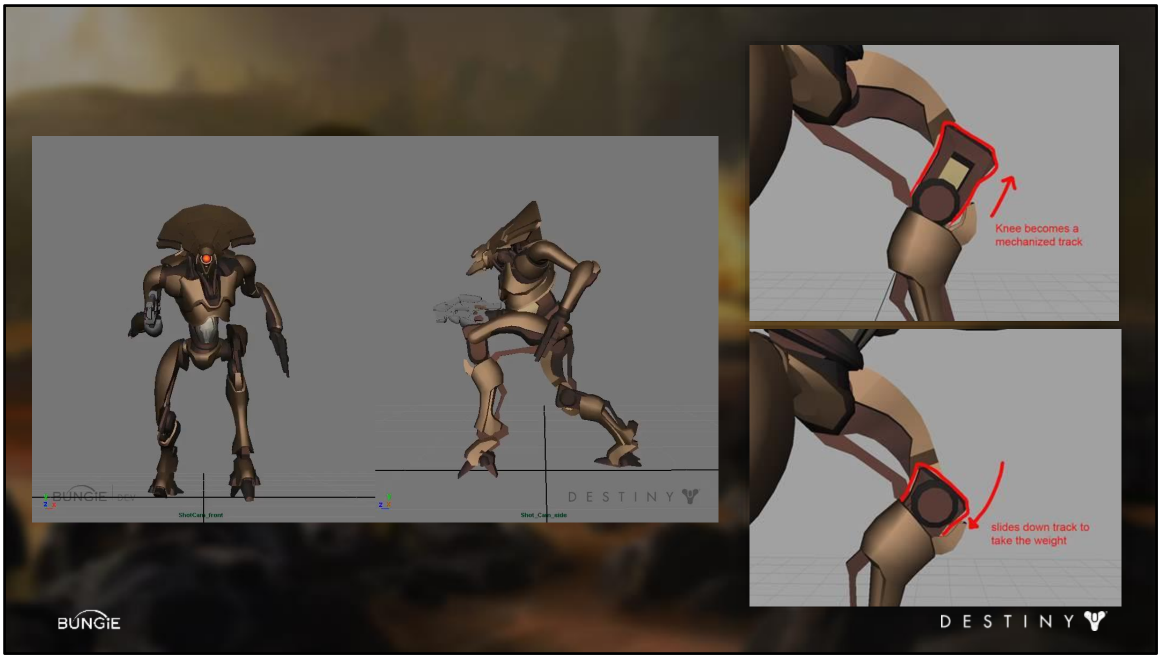
In this phase, we combine the block models to make a single manifold model. Note that we do not create UV layout or textures yet. We identify the bind pose that will be best for skin deformations. On the model, keep evenly spaced quads at medium resolution suitable for taking into hi res modeling. Additionally, indicate flexible and rigid surfaces on the model. Include block models to represent all permutation add-ons such as packs and armor.



At this point, the asset is still a low-poly cage authored in DCC tools without any material representations. Vertex weights also haven't been finessed.

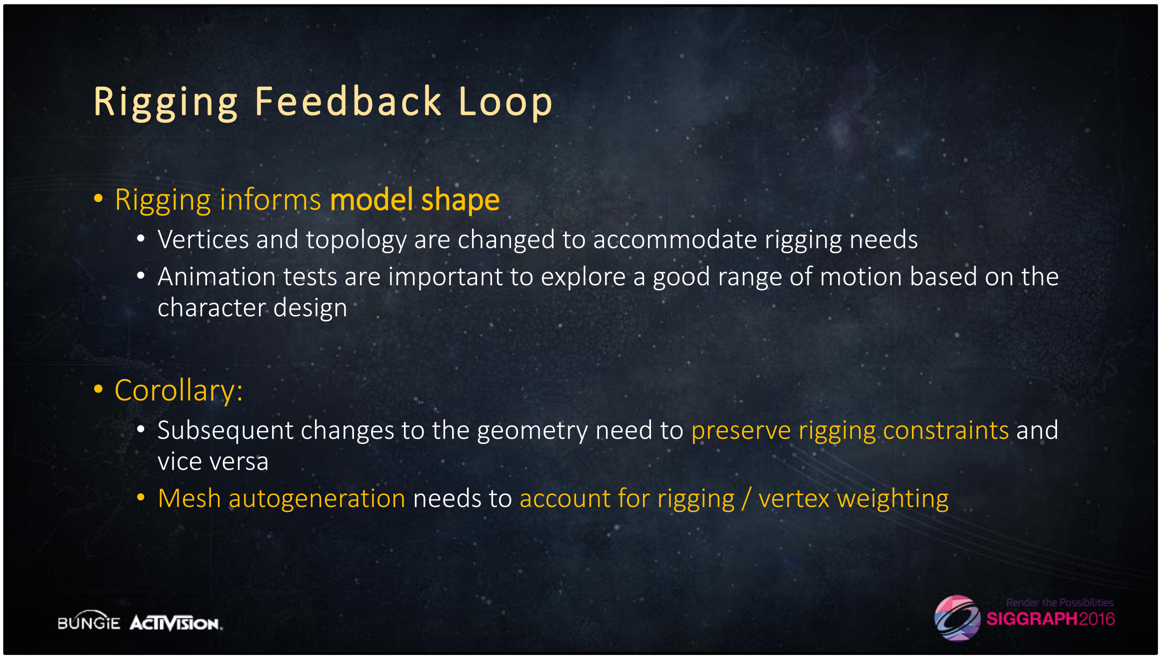


Then as you saw in our pipeline diagram, a block model was constructed and moved into grey model stage for animation exploration.
Here is one of the rigging exploration test for the Vex Goblin that you saw in the diagrams earlier – this is one of his friendly stomp walk cycles



During this animation exploration we see supporting mechanical animated leg geometry. Knee pieces becomes animatable (on a “track”)
Previously unseen knee geometry will now be visible

Previously unseen knee geometry will now be visible and the geometry will need to be adjusted



During the grey model phase, rigging informs **model shape** (i.e. geometry directly)

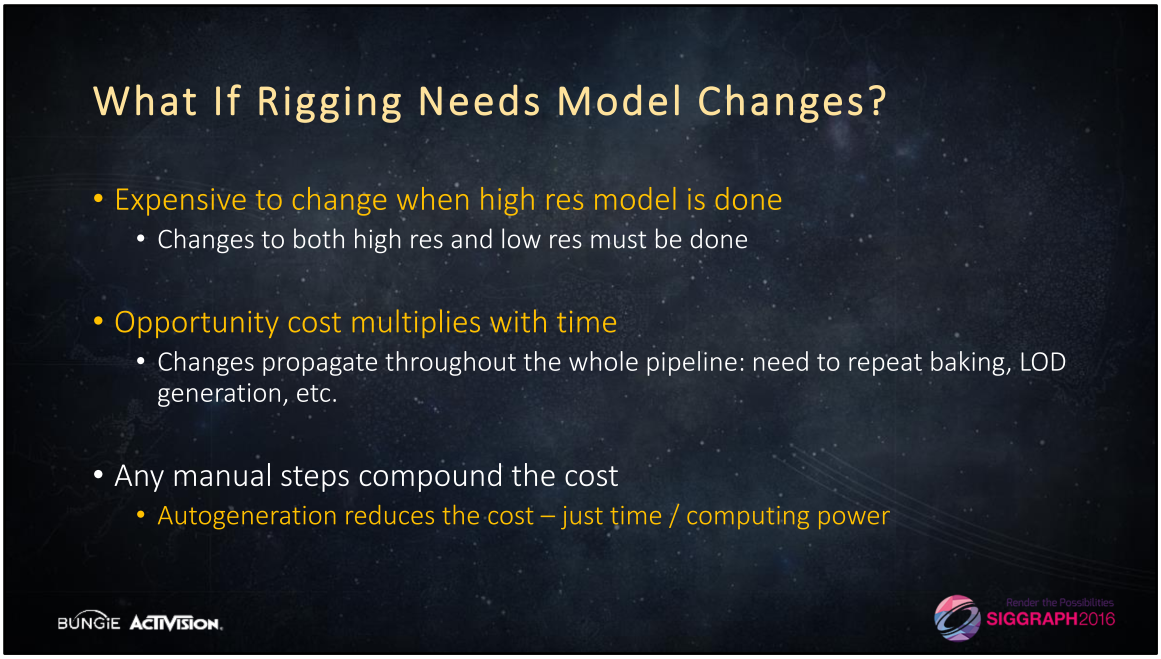
Vertices and topology are changed to accommodate rigging needs

Animation tests are important to explore a good range of motion based on the character design

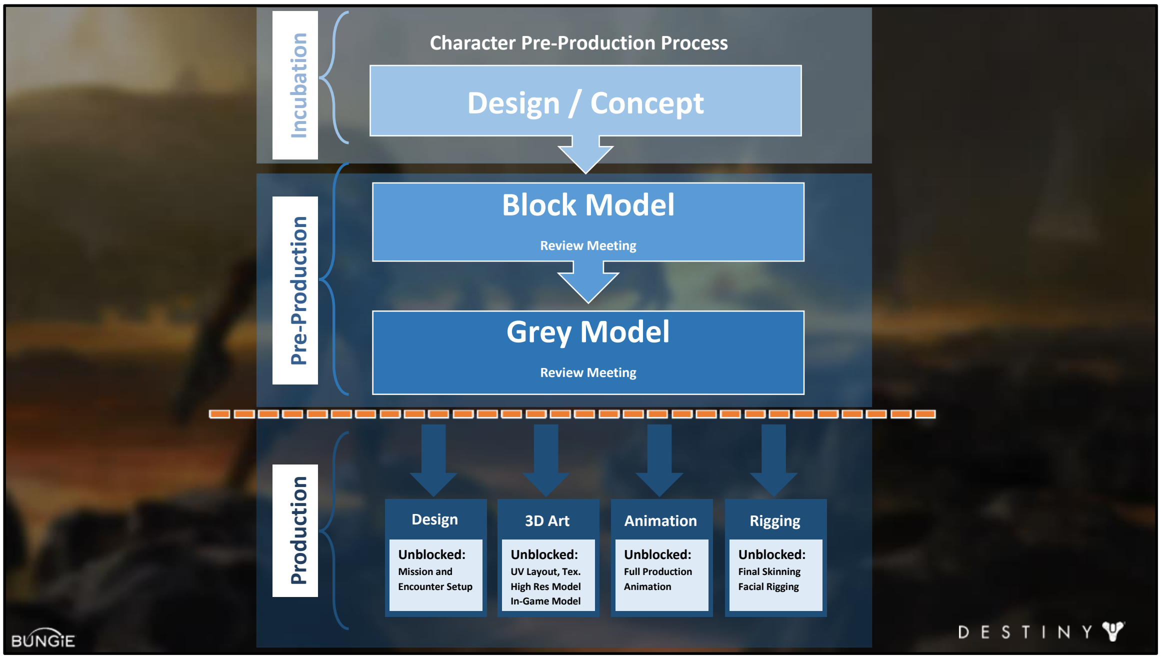
Corollary:

Subsequent changes to the geometry need to preserve rigging constraints (including vertex weighting)

This is important for any autosimplification or auto-generation stack that will be used to generate geometry post grey-model phase.

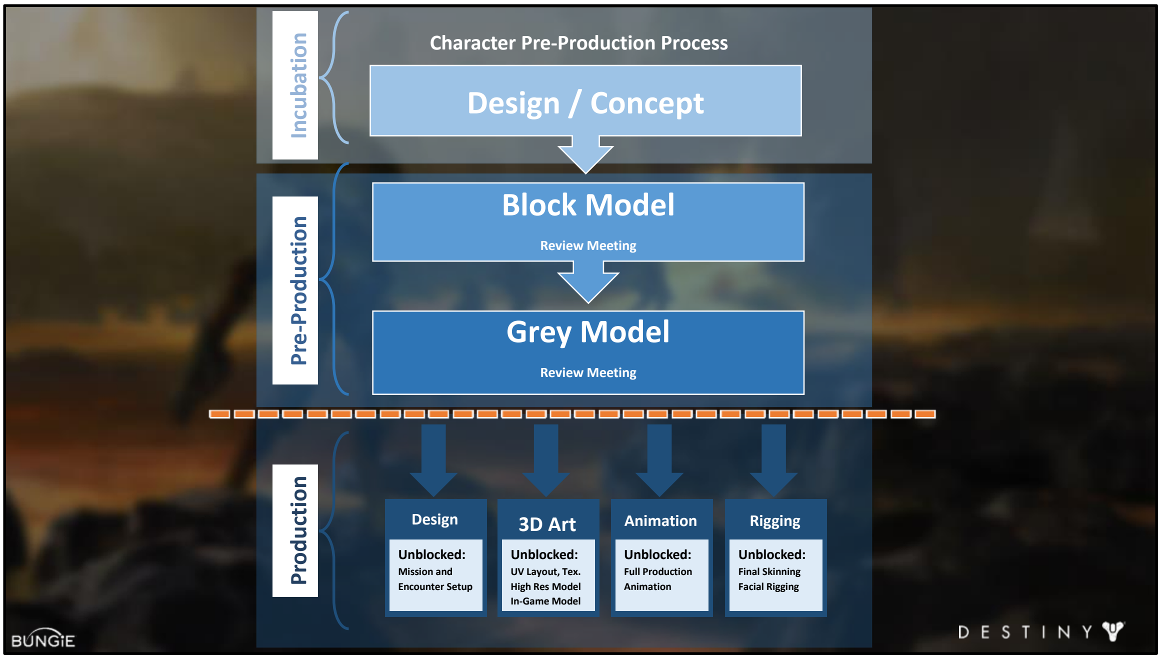


Once we have generated the high resolution assets (Zbrush model, etc.) changes that are necessary to accommodate rigging needs can be much more expensive to implement (and may ripple through many stages, like UV layout, texture baking, etc.). Any manual steps in the process compound the cost of changes. Of course, that's the main motivation to introducing auto-generation into the pipeline – to drastically reduce the cost of any changes (just recompute!)



Once we are done with the grey model phase, we enter the final stage - **the production phase** for this asset.

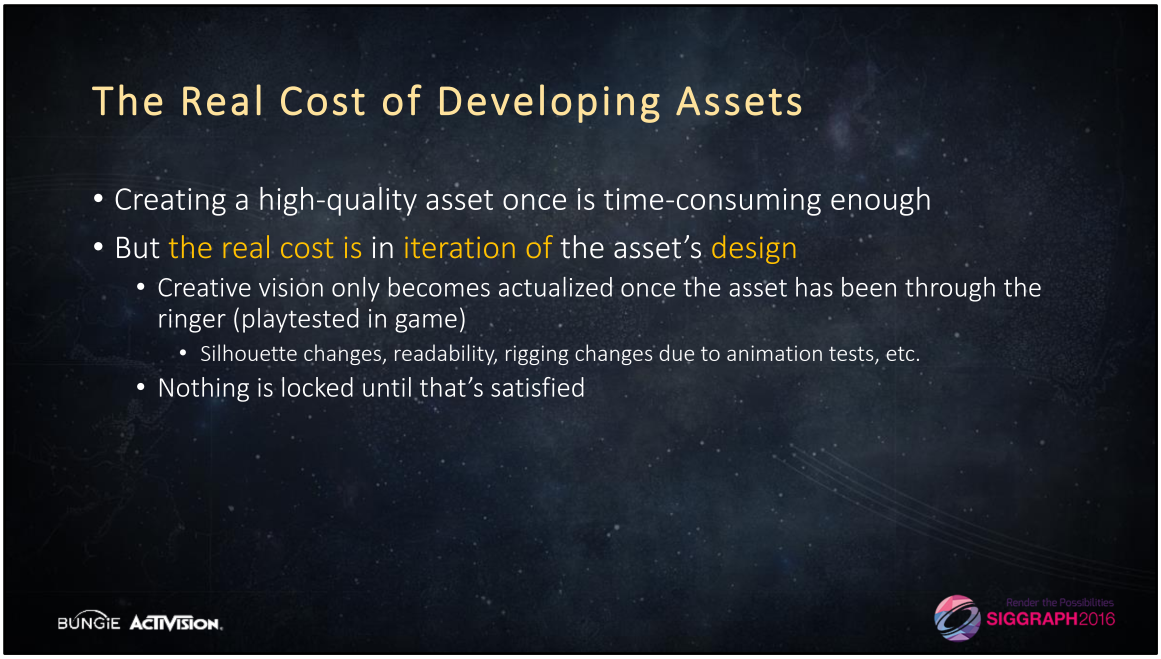
At this point, design team starts working on mission and encounter setup with this character. Animation starts doing full production animation cycles, rigging does the final skinning and facial rigging for the character (if the latter is desired), and, …



of course, 3D art gets busy doing the expensive work on the character. UV unwrapping starts, texturing; modelers build high resolution models in the appropriate tool (typically Max for hard surfaces, Maya for inorganic moving objects, and Zbrush for organic surfaces). This is also where baking of material representation occurs to enable the final shading / lighting passes start occurring in game on the material representation for this object.

And at the end of this phase we have our assets in game with the full material stack, lit and rendered in engine

And at the end of this phase we have our assets

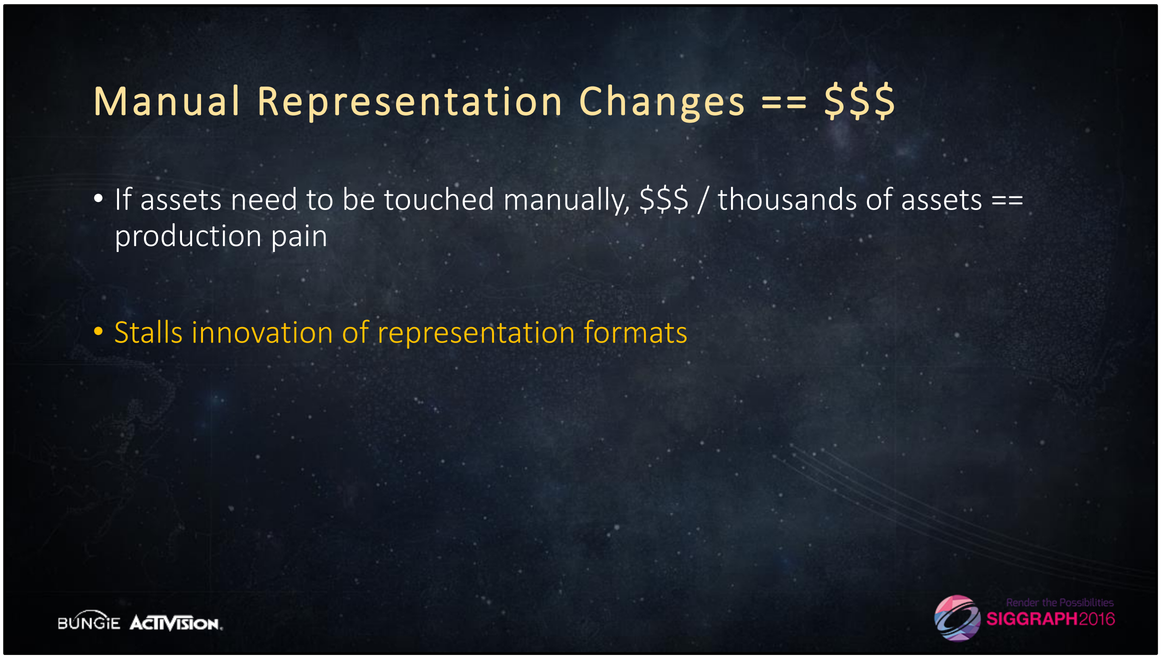


It’s definitely true that for high-fidelity graphics, there is a very high cost of creating the high-quality asset. Once! But once we start thinking about all the iterations the assets go through during their lifespan (for design reviews, animations, etc.) the cost of iterating on an asset becomes the dominant factor. In games, the reality is that the creative vision (and thus the purpose for this asset) is only actualized once the asset is in the game, and players (and designers) can experience this asset in the actual context, examining its readability in game, its behavior for animation and the purpose in game, the silhouette readability, and so forth. This means that the actual content may not be locked for a number of iterations.

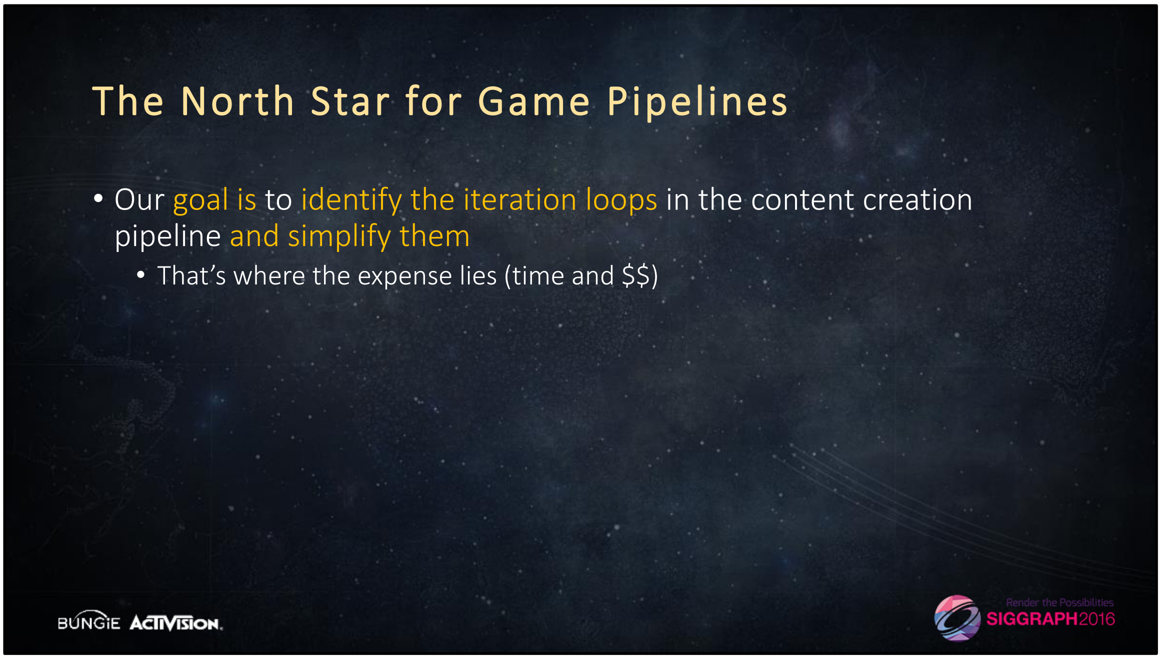
The important corollary for this is that changes to any of the asset elements have a deep propagation stacks. This is particularly true for any elements that require non-iteration friendly rebake – or worse – re-author elements. If you needed to change UVs for any reason, it may now mean you need to re-rip all the maps (ex: normal, displacement). You changed the compression for texture adding support for BC7, let's say – another rebake. Of course, some of these elements are automatable, but, however, even with that, it's still a time-consuming process.

To deal with this, game production pipelines developed sign-off processes – essentially allowing a time frame where there is tolerance to changes to a particular element of the asset, but sign-offs phases exist because the representation is rigid. For example, no changes to silhouette in production phase – redoing Zbrush *and* low res *and* animation / rigging → $$$ production-wise.

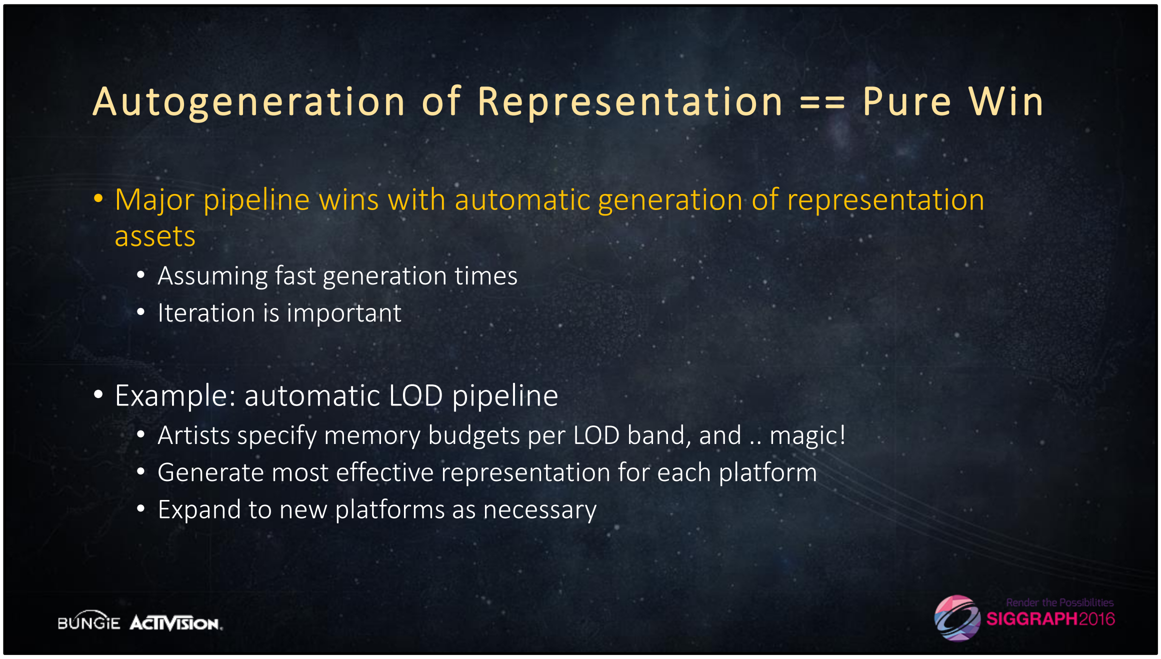
Ideally, of course, we develop content creation pipelines that *minimize* any rigid phases, allowing as flexible and iterative process for asset creation. That allows creators to focus more on iterating the vision, and less worry is spent on the costly sign-offs.



Of course, even worse than re-bakes, if the assets need to be touched manually, this equates to significant costs in artist times. If you consider that need for thousands of assets that might be present in any given game level, this can significantly increase production pain (and costs). This also can signify stalling injection of new representation if affects production time. A good examples of that have been the switches to physically-based rendering approaches (which required major changes to the textures assets) and adding tessellation (which required changes to the low-resolution polygonal cages and displacement maps).



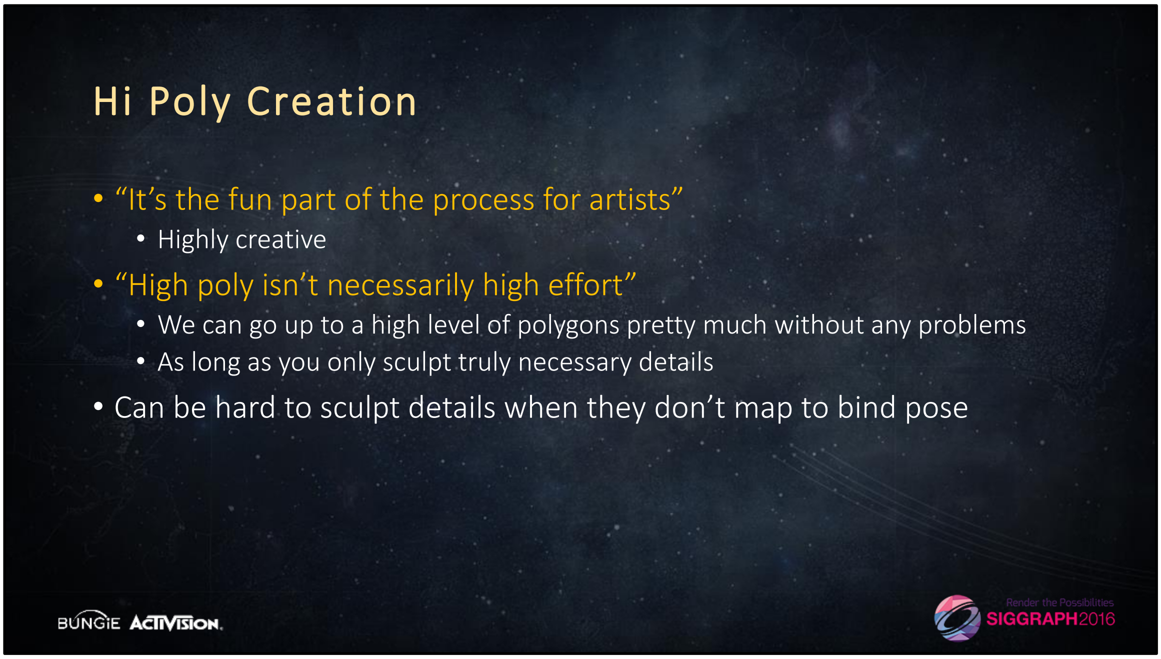
So what we’re aiming to do is to determine what iteration loops exist currently in the game production pipelines for asset creation and simplify them, replacing manual steps with automation and reducing complexity of bake processes. This is ultimately what will drive the production costs down for game creation.



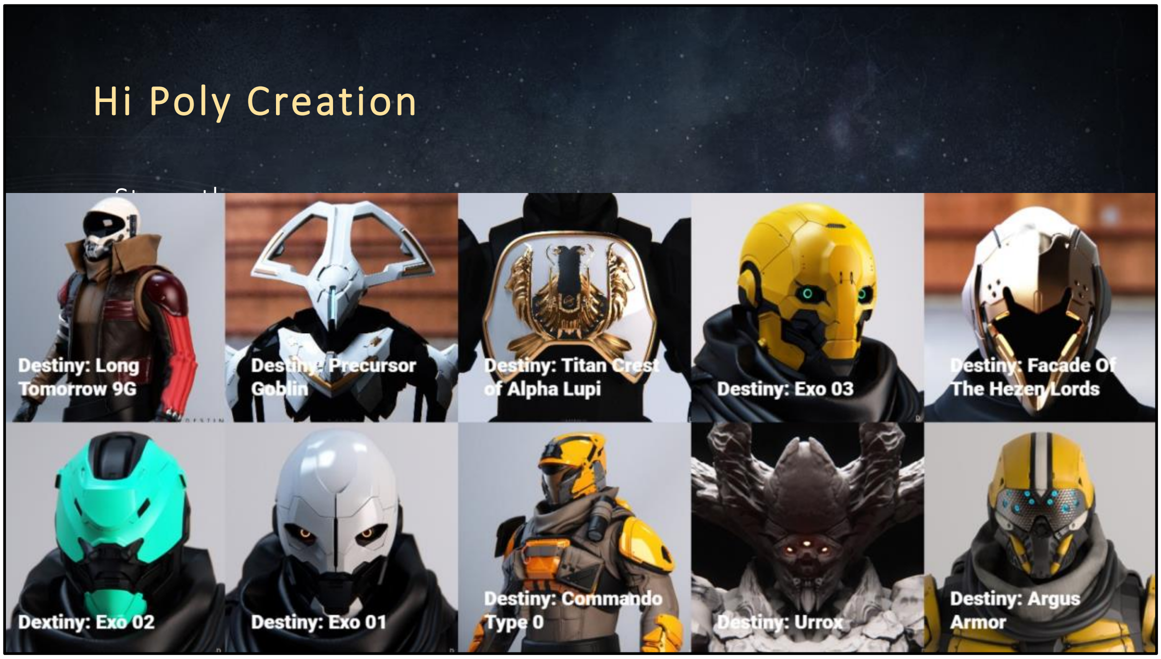
If we can add automatic generation of representation assets (textures, meshes, etc.) this equates to significant production gains. The important to consideration to keep in mind is that we're doing this to *allow iteration*, thus we shouldn't add automatic generation of various representations (LODs, etc.) if they stall iteration.
A good example of where that occurred is automatic LOD generation – originally LODs had to be generated by hand manually, and that was a painful, time-consuming, and fragile process. Implementing automatic simplification elements of the pipeline allowed us to remove that requirement, allowing artists to focus only on base mesh generation. Of course it is still important to add controllability by providing, for example, heuristics for controlling the reduction amount or memory footprint per LOD band. It also enabled us to seamlessly add new LODs for new platforms, without having to manually re-author thousands of assets.

We also wanted to take a pulse from the artist community (both games and film) to understand what their impressions and pain points were in this domain.

This is the list of studios who responded.

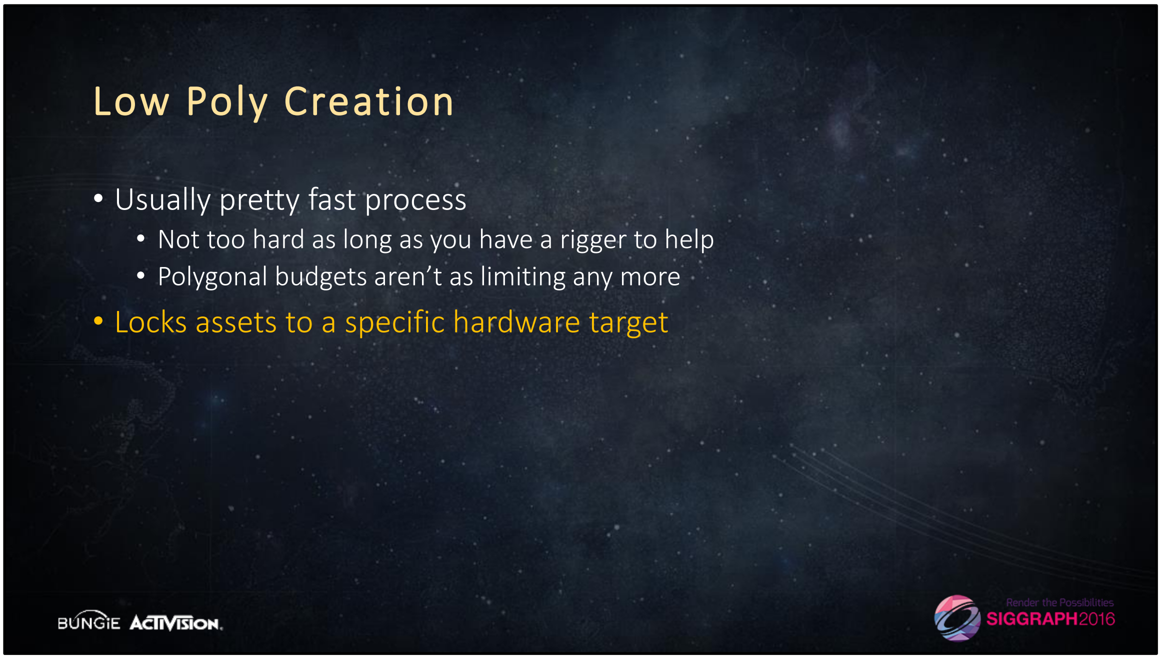


Consistently, the artists relayed that high polygon creation is where they want to spend their time. With the advent of the latest high-poly pipelines (Zbrush, mudbox, etc.), it's quite easy to add details going up in the extremely high level of polygons without any problems or long efforts. The major difficulties the artists encounter in that area is sculpting details which may not map to bind pose (example: armpits). But other than that, this is a very creative phase of content creation.

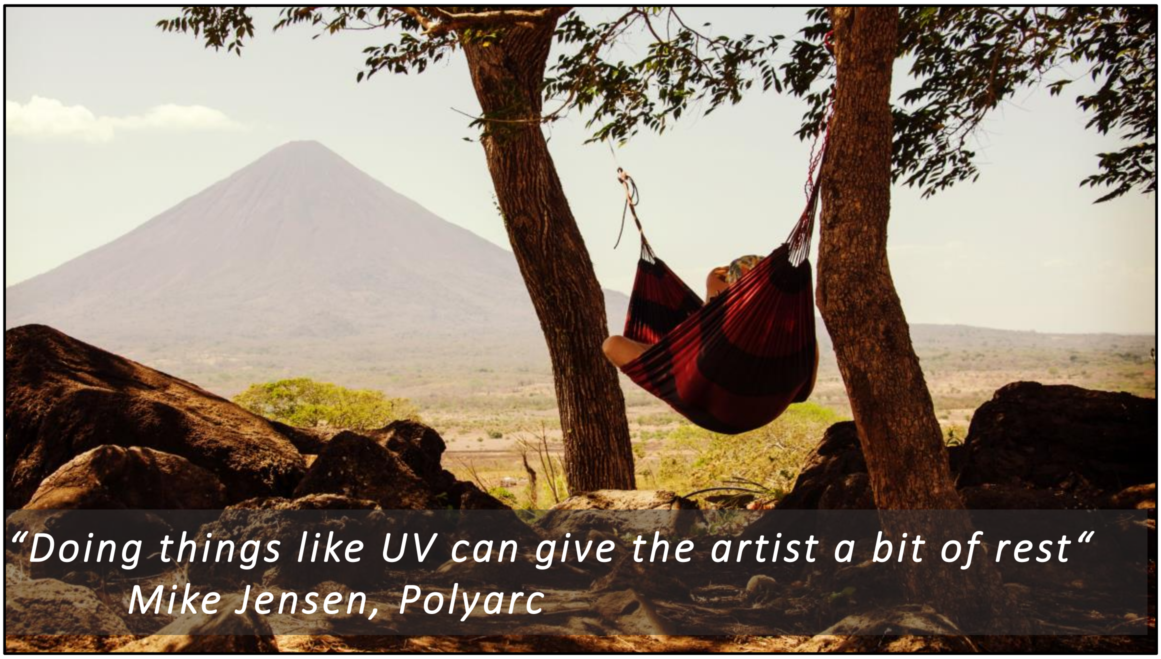


This is one of the most fun parts of the process for artists

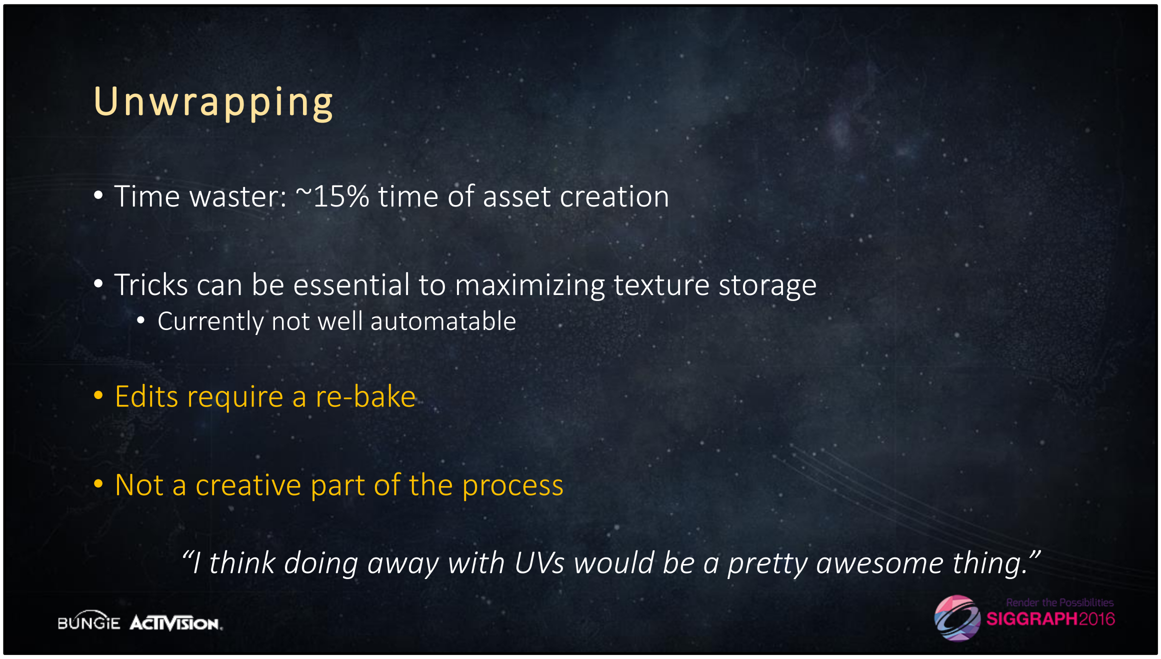
They get to go wild scupting beautiful things in Zbrush / Maya / Max. Here are some examples of Destiny gear that were created by an amazing artist Mike Jensen



Next, they can generate low-poly in DCC tools as well – the most complexity there is with the help of a rigger, ensuring that all rigging requirements are satisfied. However, creating specific low polygonal assets does lock the actual asset to a specific hardware target, typically through pre-defined polygonal budgets. Changing this budget means re-authoring, and that’s where you start seeing the cost pile-up.



The irony: “Doing things like UV gives an artist a bit of a rest for their brain”



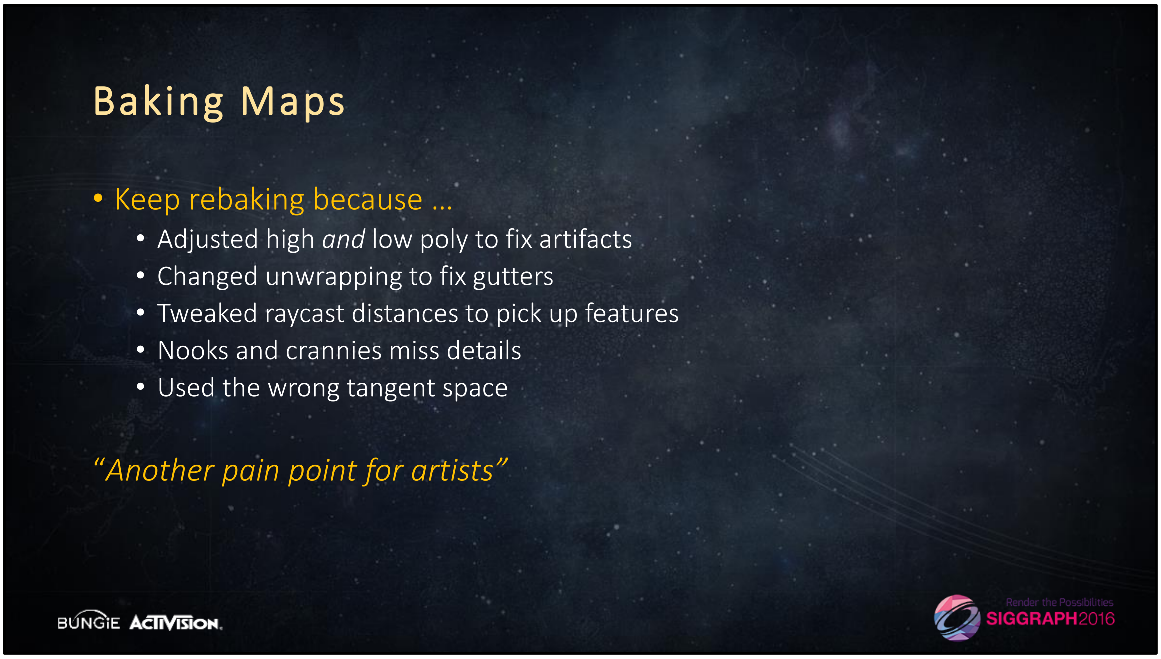
Although uv unwrapping is pretty fast with good tools, it's not fire and forget – artists often still have to do a lot of tricks to maximize texture storage (UV island overlap, scaling UV islands, etc.)

Usually pretty fast process
Flexible toolchain
You can choose from any bakes you like!.. Whatever floats your boat.. Or your non-Atkins diet anyway..
But you get tired from so many decisions..

A myriad of tools – to each their own
3dsmax, Xnormal, or handplane. xNormal, CrazyBump to Maya. Baking from highres (whether it's zbrushed, photogrammetry, or high res model) for most things. For small mechanical details artist can use Ndo. Substance Painter to create normal maps

 3DSMax

 baking from Zbrush / Photogrammetry sources, Ndo for small details…



Across thousands of assets
$$ COSTLY $$

Baking normal map is actually another pain point for artists- to be more precise, baking repeatedly while adjusting the low poly to minimize artifacts.
Weaknesses:

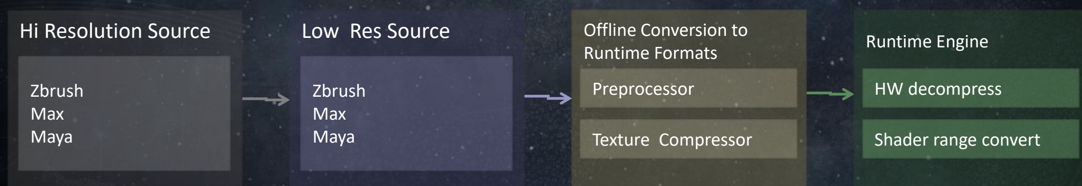
- Nooks and Crannies again an issue. Shouldn't we optimize the UV space automatically based on feature frequency so that artists don't need to?
- Inconsistent tangent spaces between engine and DCC tools are problematic
- Once the decisions are done, the asset is baked. But at that point, it's incredibly hard to identify any bugs which brings us to the next point..

Deep Pipelines Debugging is Fun

BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

A Deep Source to Runtime Pipeline



- Errors can happen at any stage
- Hard to diagnose – where do you hunt this bug?

BUNGIE ACTIVISION

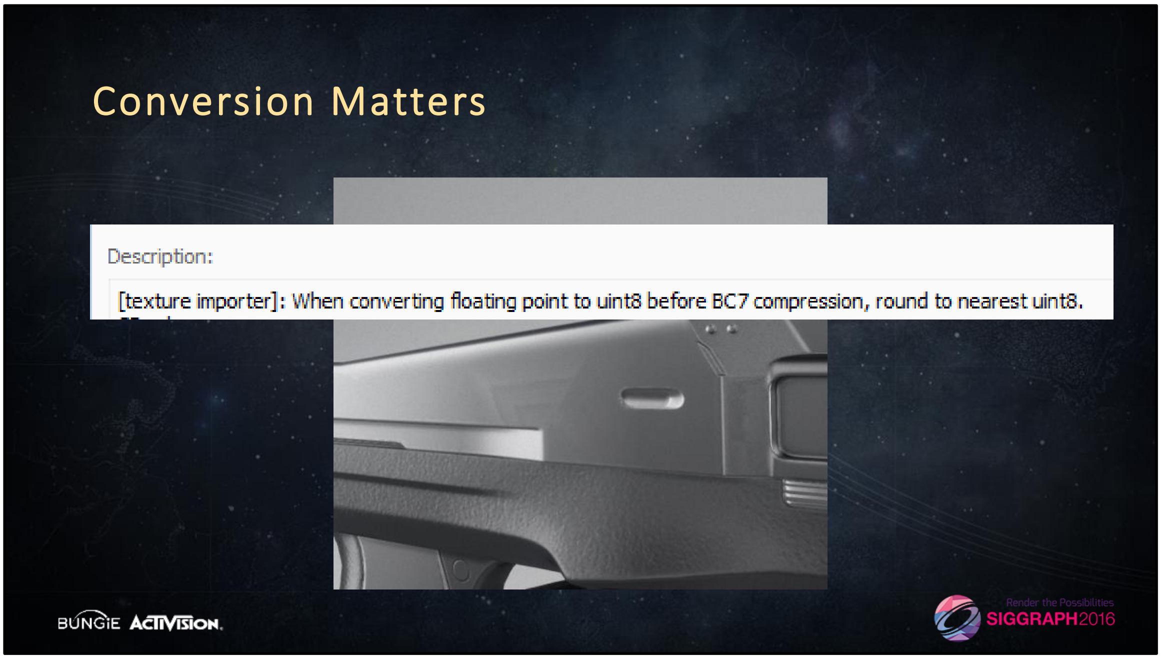
Render the Possibilities SIGGRAPH2016

Here is a pic of a gun in game.

Has a weird mark.

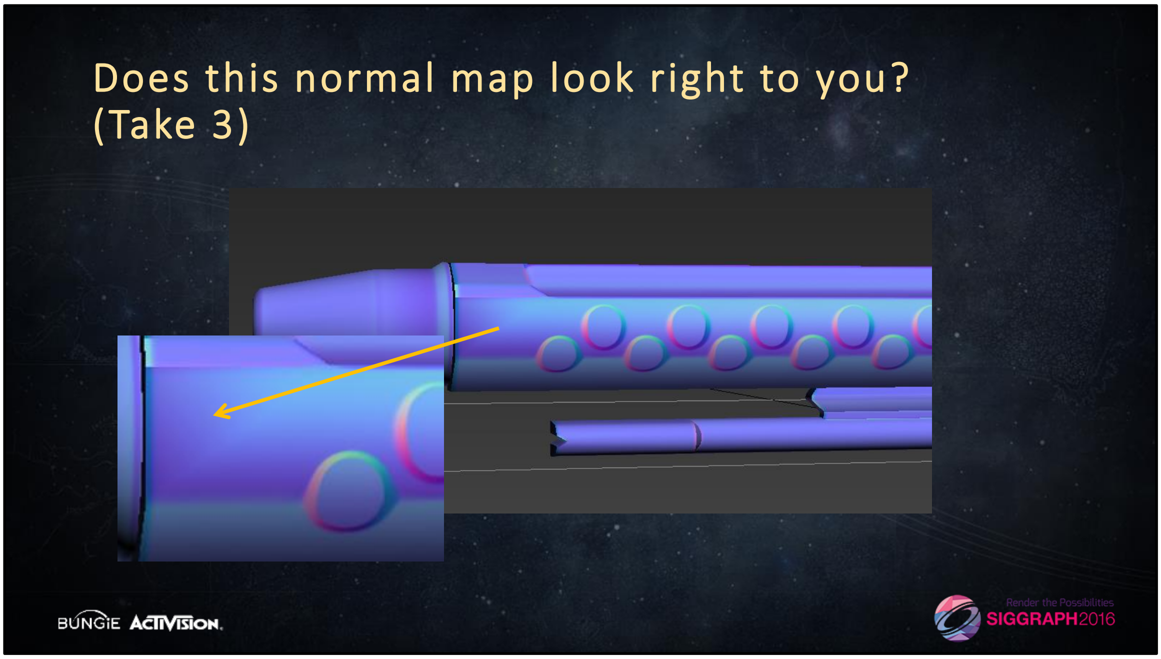
But the normal map itself is perfectly smooth there. So any ideas? What's happening here?

Switching to “NormalEdges” mode I see this. In our engine we have gradients display (we call it “Edges”). It shows us something interesting.



Yep. After a bit of a complex investigation, it turned out the error was in our **texture compressor**.

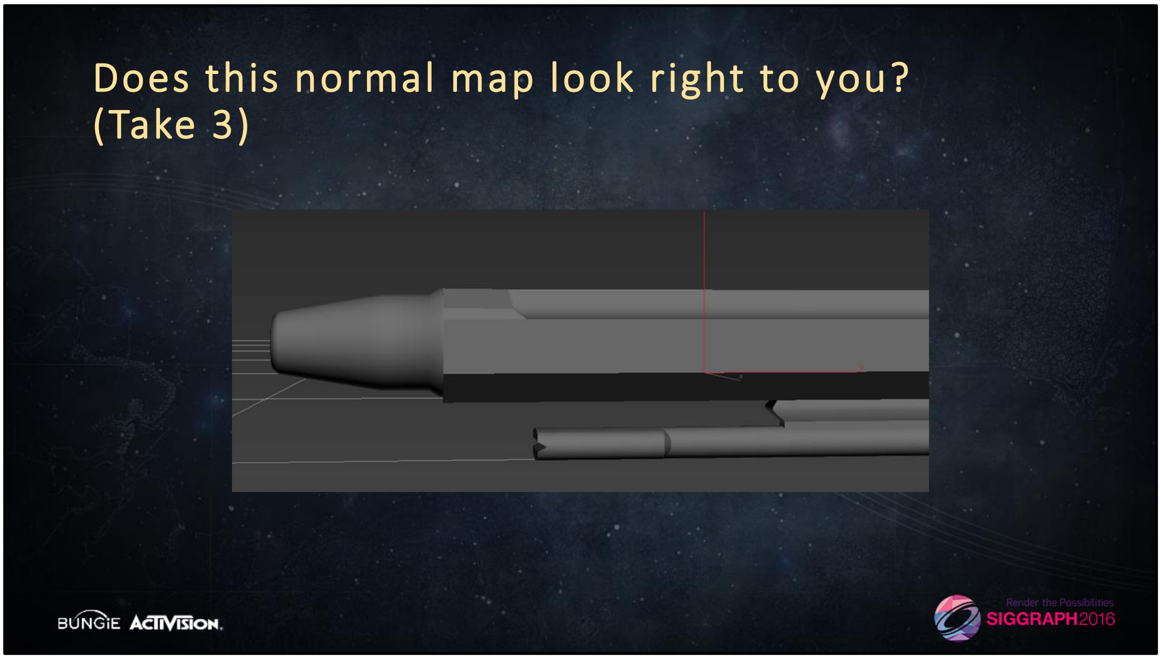
This is a pretty common issue to variety of engines. Here you have a normal map – looks kind of ok.



In this case you can see the “bruising” that happens on the left hand side of the model/normal map.

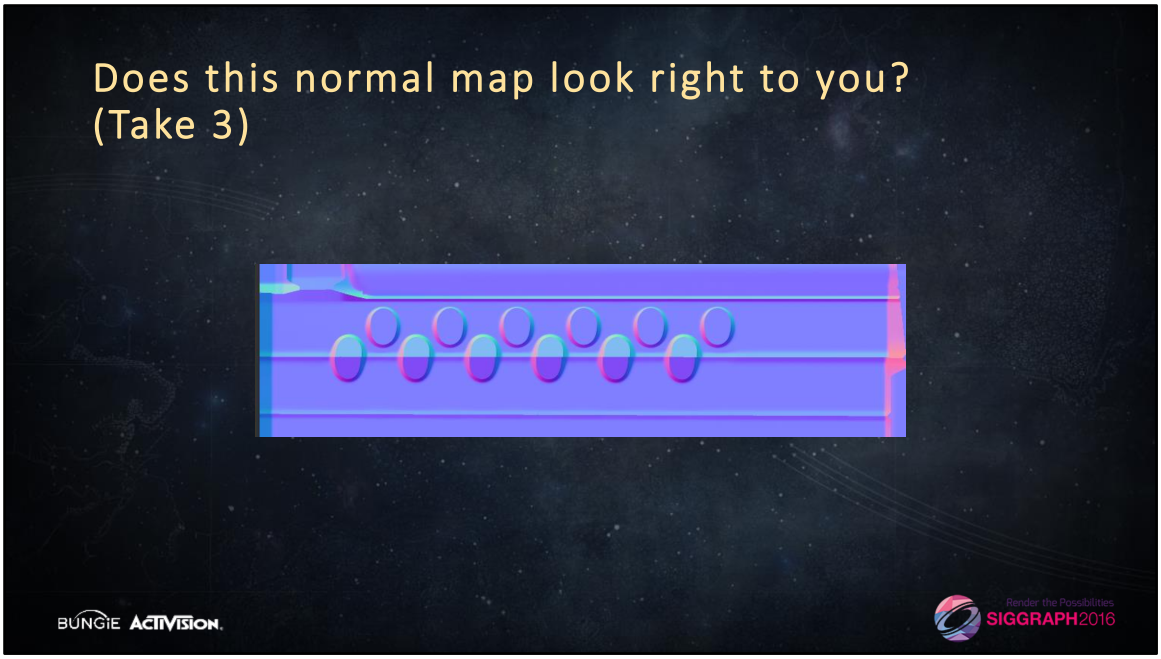
The artist forgot to break the smoothing group on the center face. OR he could have added some very very tight chamfers/control edges to simulate the same fixup.

You can see the bruising somewhat here, on the left side again, from adding 1 smoothing group, wrapping around the angled surface.

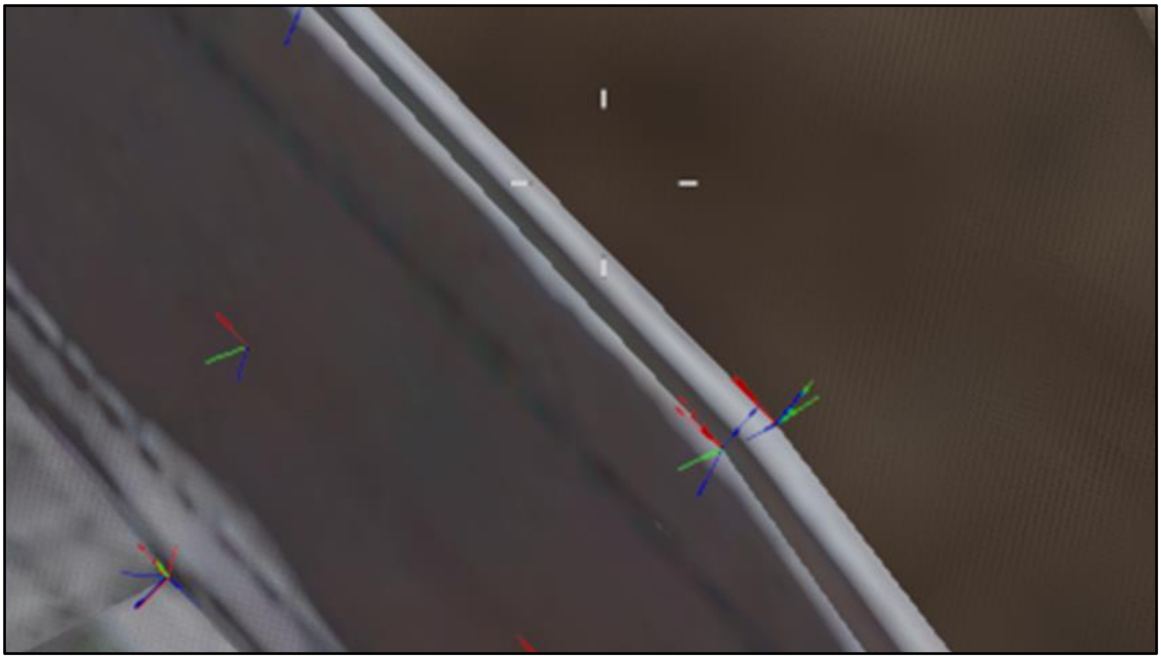


Now I broke the smoothing groups here, to get a hard edge/clean normals.

This will show up even more so, when higher spec/cubemap is applied/visible in-engine.   Even worse/more noticeable if is In ADS/Iron sight on a scope.

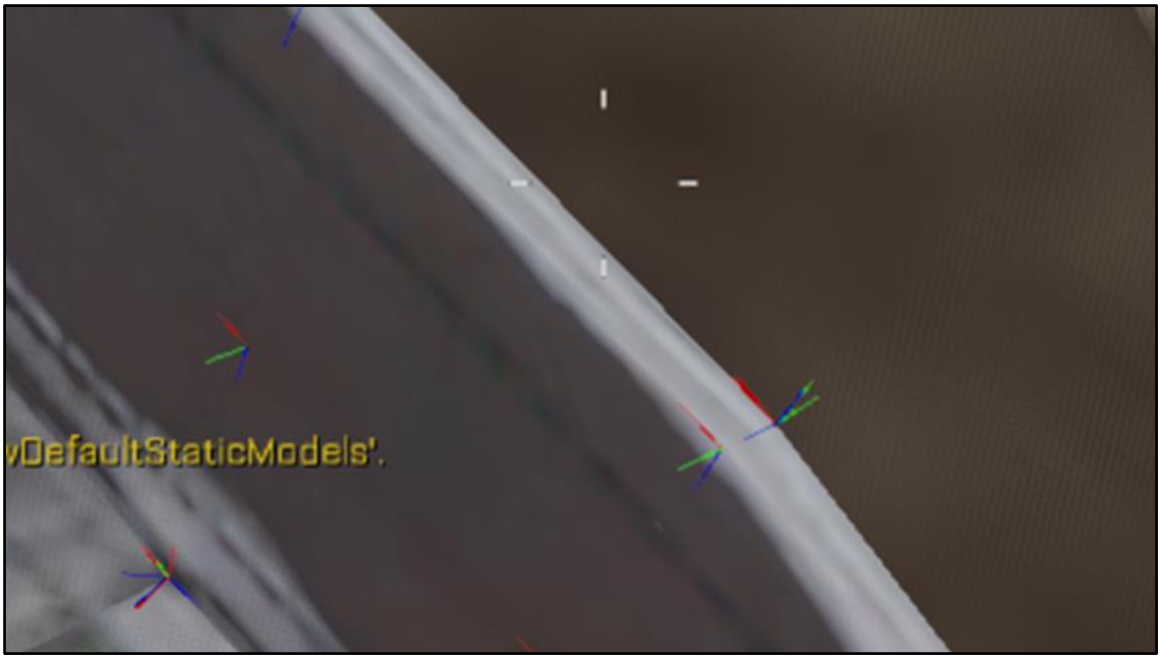


Broken smoothing (or chamfers) now result in a much cleaner normal map, but adds to the geo cost/budget. So that's problematic to address and to diagnose, requires us looking in the full set of tools to understand the relationship between geometry and material representation



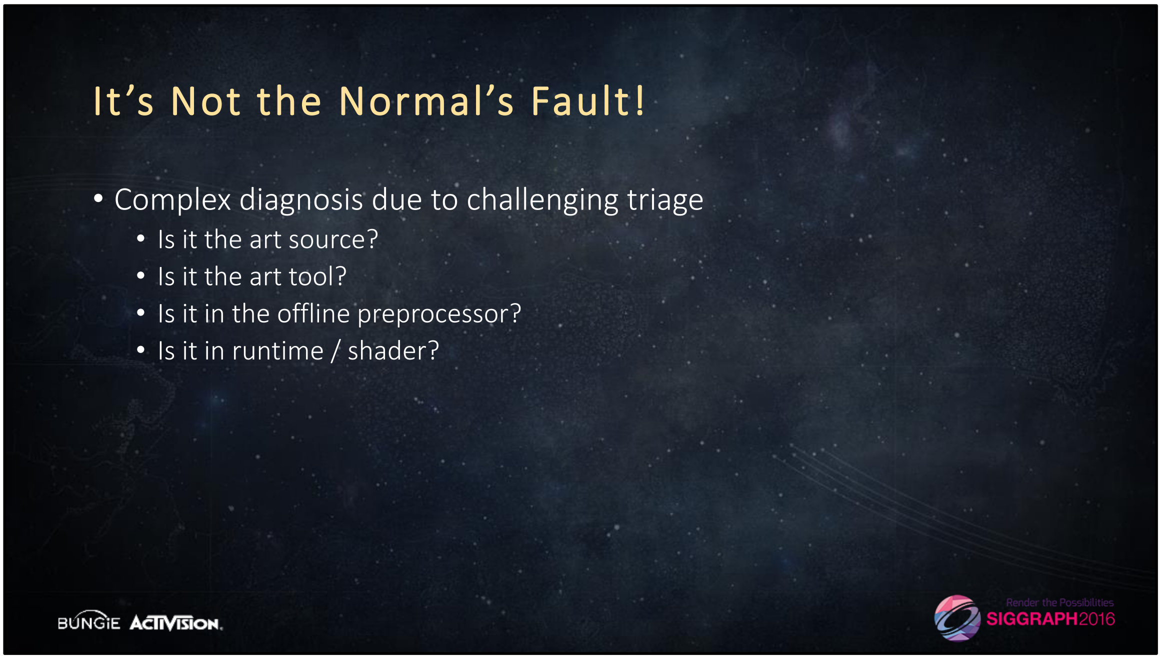
We have the same kinds of problems in COD.

This image shows a dark stripe along the edge of a character’s vest; the character happened to be acted by Kevin Spacey.
The colored lines represent the tangent frame.



This image shows the result when Maya’s tangent space is exported along with the mesh, instead of the game engine calculating its own.

Unfortunately, because all of the game’s assets had been exported from Maya without tangents at this point in the project, we were only able to fix this issue in spots where we noticed the artifact and re-exported.



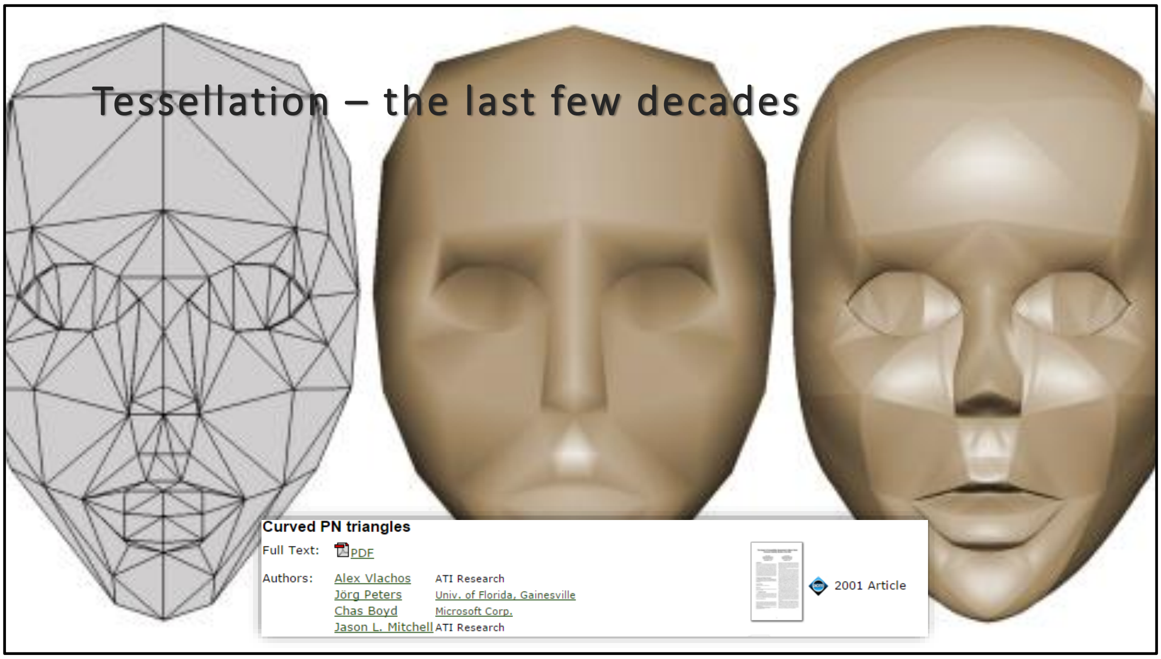
Deep tools to runtime pipeline necessitates time consuming investigations to understand the source of error

Baked normal maps have lots of problems that are hard to diagnose

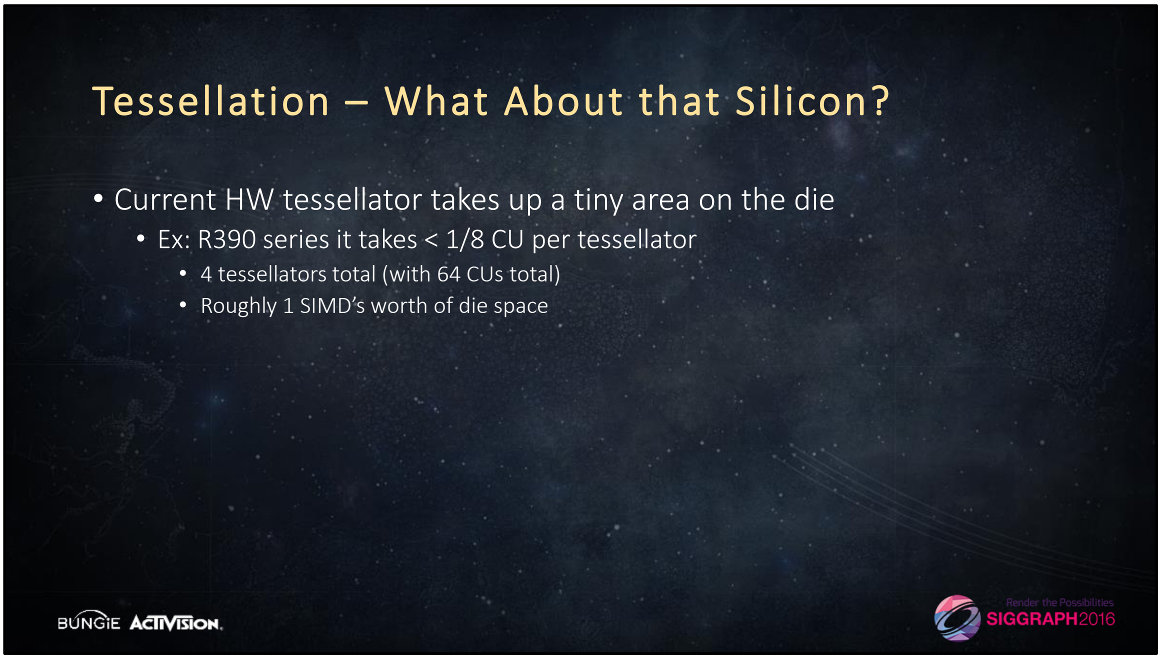
When they are detected our choices are limited

Re-baking is so expensive (especially late in the project)

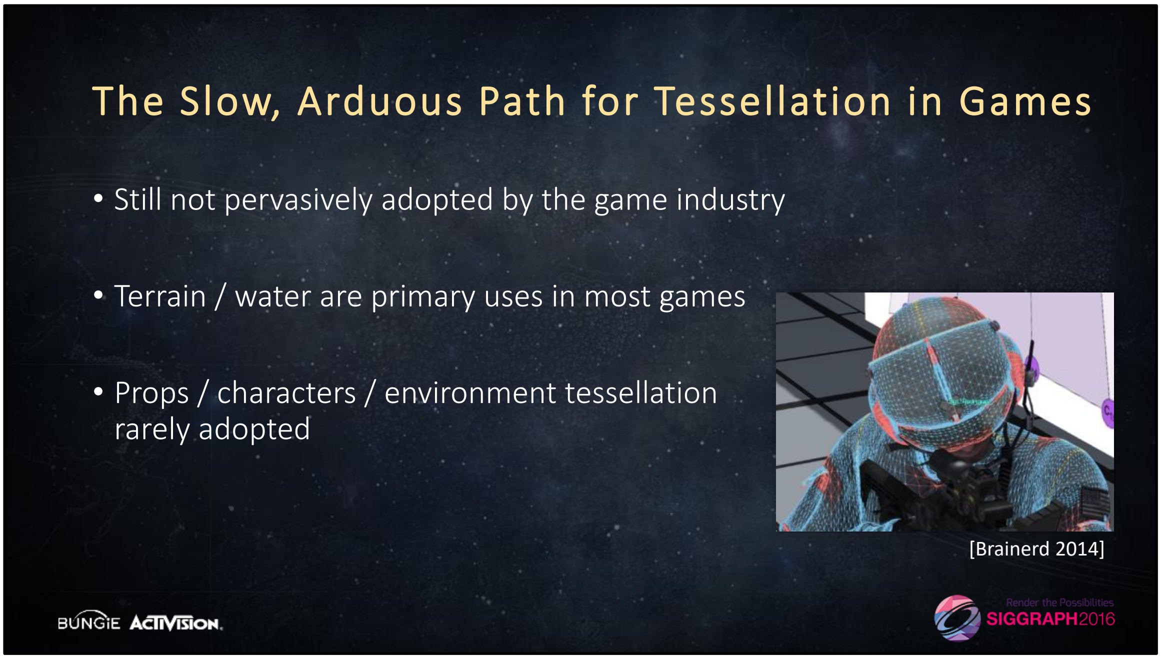

Tessellation: What About That Silicon?
BUNGIE
ACTIVISION
Render the Possibilities
SIGGRAPH2016



Hardware tessellation has been around for a while
PN-Triangles are over a decade old



AMD, dropping tessellation hardware would allow adding <1/8 CU in a high end R390 series (with 4x tessellators, the max). These GPUs have ~64 CUs, adding single CUs not possible anyway.



Still not pervasively adopted by the game industry
Terrain / water are primary uses in most games
    Relatively straightforward use cases
    Interpolative tessellation has cheaper runtime cost
    Displacement maps are relatively easy to author
Props / characters / environment tessellation hasn't been widely adopted in games
    With a few notable exceptions: *Call of Duty: Ghosts, Lost Planet 2, etc.*

Why Is It So Hard?

- Content source representation changes
- Art tools pipeline
- Runtime performance
- Hardware behavior

BUNGIE ACTIVISION

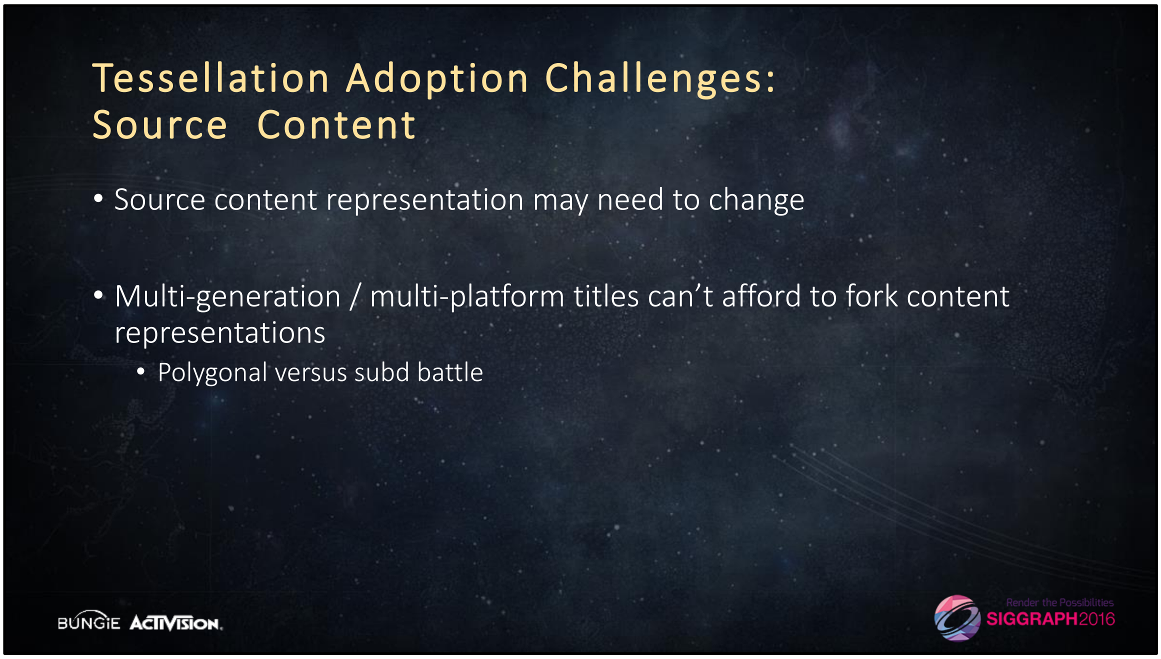
Render the Possibilities SIGGRAPH2016

Why Is It So Hard?

- Content source representation changes
- Art tools pipeline
- Runtime performance
- Hardware behavior

BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

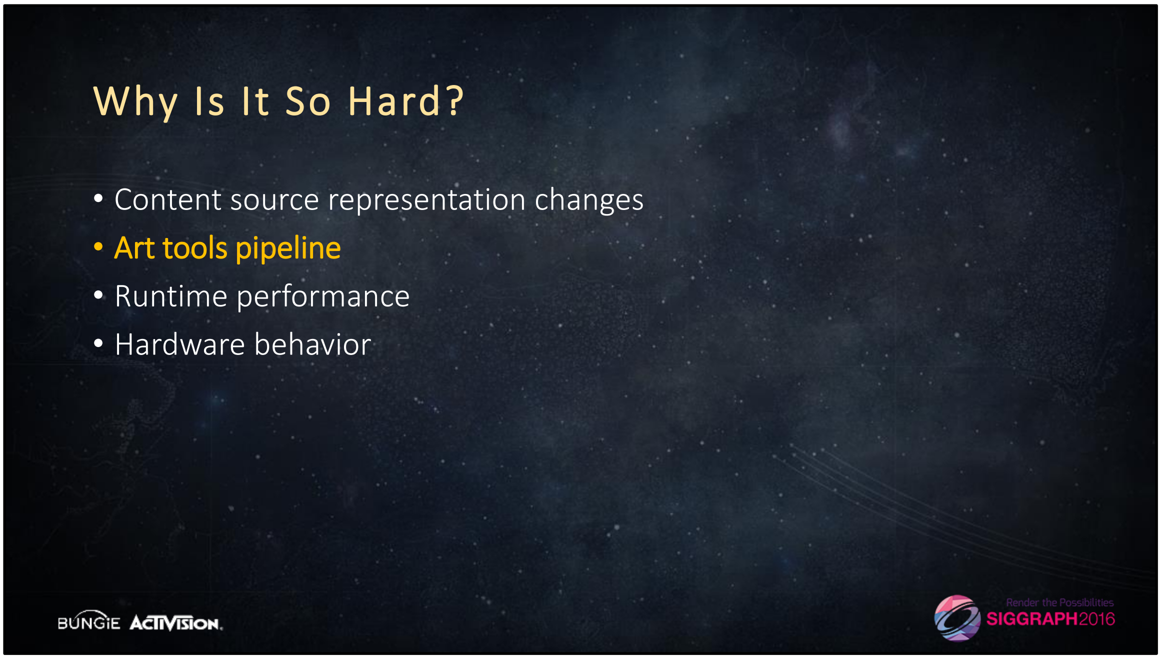


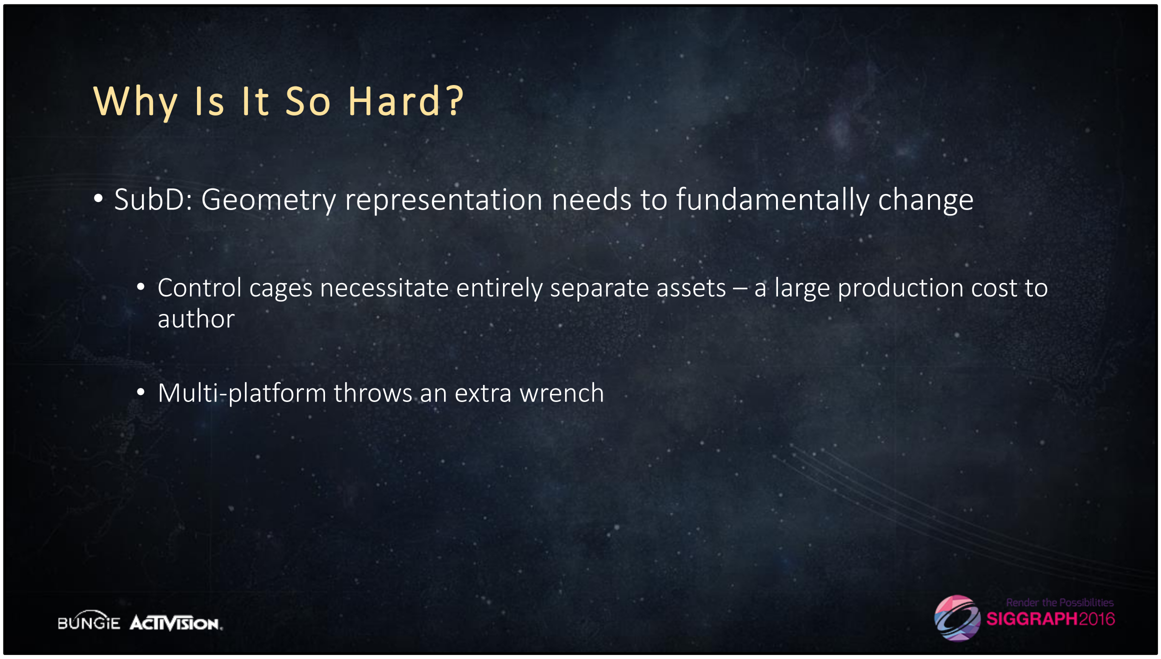
Depending on tessellation method, source content needs to change

- PN triangles need crease edge management specified in content
- Displacement map for most cases need to be generated
- SubD needs control cages
  - This changes the entire authoring pipeline: mesh generation, texture baking and rigging
  - Large associated production cost

Multi-generation / multi-platform titles can't afford to fork content representations

- Content creation is where the vast majority of game costs lies

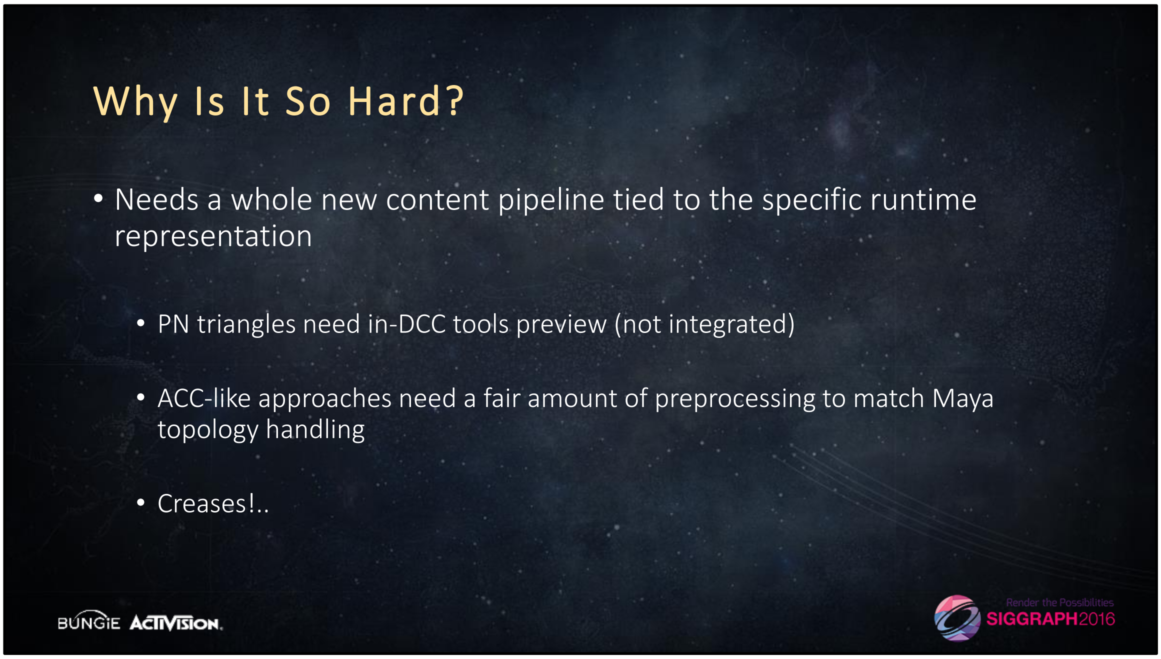

Why Is It So Hard?
• Content source representation changes
• Art tools pipeline
• Runtime performance
• Hardware behavior
BUNGIE ACTIVISION
Render the Possibilities
SIGGRAPH2016



SubD: Geometry representation needs to fundamentally change

Control cages necessitate entirely separate assets – a large production cost to author

Multi-platform throws an extra wrench

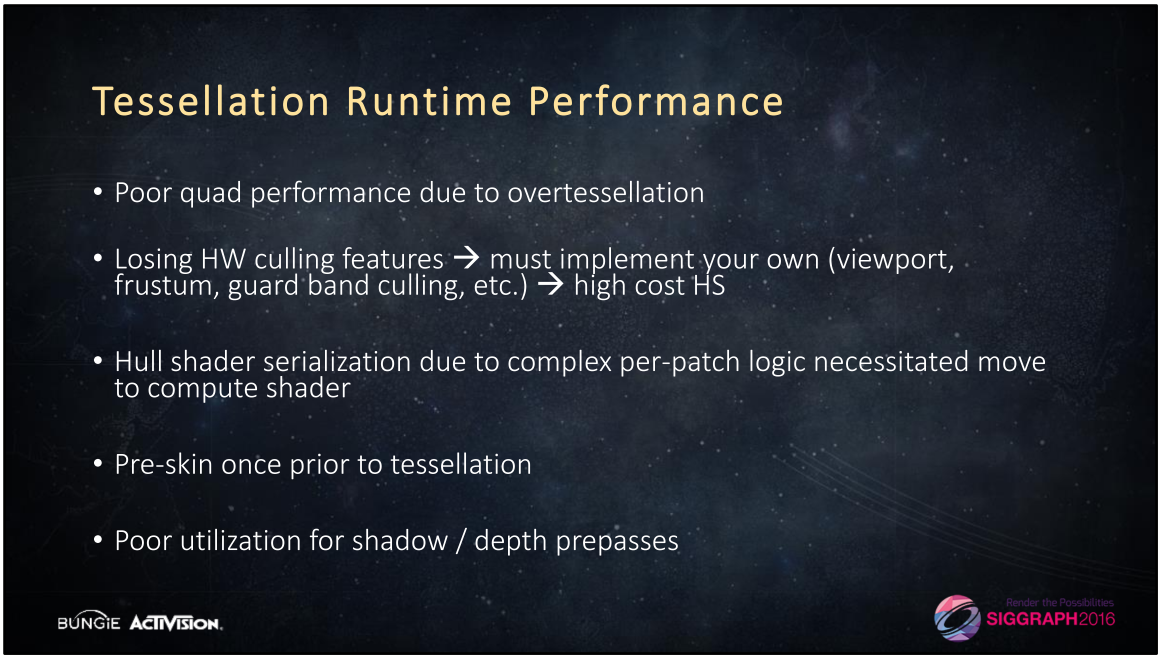


Needs a whole new content pipeline tied to the specific runtime representation

- PN triangles need in-DCC tools preview (not integrated)
- ACC-like approaches need a fair amount of preprocessing to match Maya topology handling
- Creases!..

Why Is It So Hard?

- Content source representation changes
- Art tools pipeline
- Runtime performance
- Hardware behavior



Shadows and depth prepass have poor HW utilization in that case

Option 1: Tight optimization (ex: Brainerd 2014)

Option 2: Drop the depth prepass (you aren't saving much)

Option 3: Compute a conservative interior shell for the tessellated object and draw that instead.

Ala conservative depth in DX11. Relatively complex to get right though. Depends on displacement behavior.

Tessellation: It's All About Commitment

- Once you tessellate, you always tessellate…
- But you only render your object once per frame, right?...?
  - Hm…. Sometimes.?

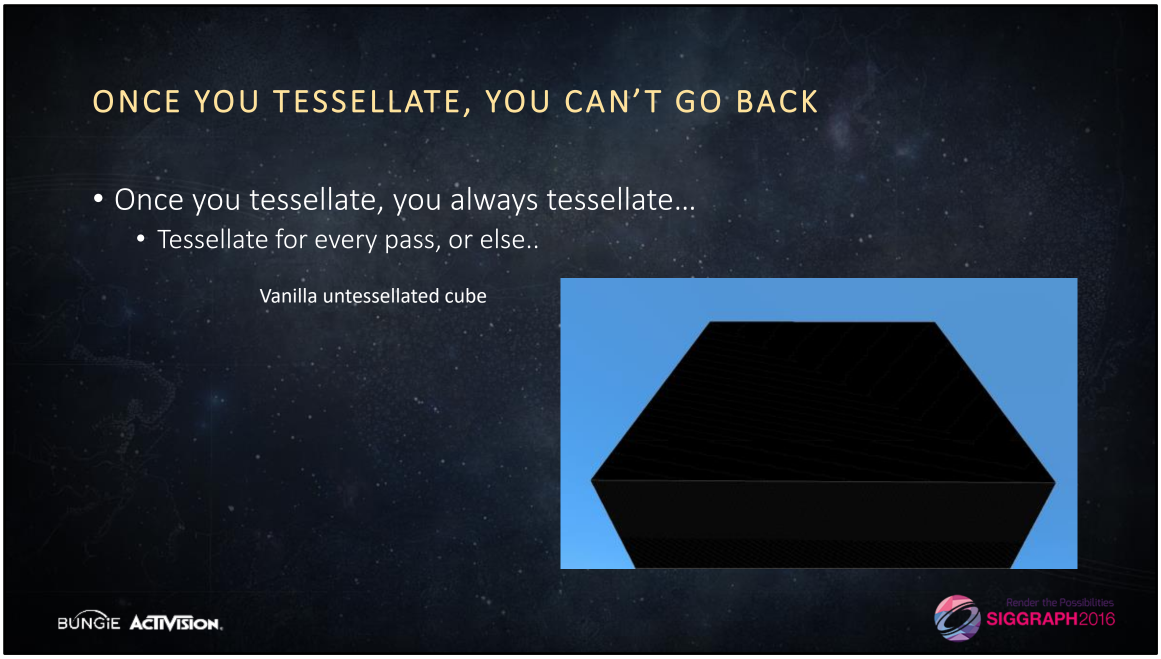
BUNGIE ACTIVISION.

Render the Possibilities SIGGRAPH2016

And other times not so much. For example what if you are a game about space magic? In our game we have dynamically instanced ‘magic’ or, as we refer to the technical feature: “object effects”. Although we do a bunch of optimizations of drawcalls during our offline preprocessing (to merge by shaders, geometry properties, etc.) we also have a bunch of dynamically instanced decals (extruded geometry), etc

So if we look at an average
Indeed! An average player object can be rendered about 5 times a frame.

Here we have a geared character (a Warlock in this case), rendering its gear as opaque geometry, we see some decals on his weapon and armor, he is also rendering to shadow map passes (we use four cascade shadow maps, but also can have local light shadow maps as well), and then finally, when he uses his super – a special power ability – you see that glowy effect on him? That's the power of transparent passes.
In other cases you may want to render your object to depth prepass to avoid shading the pixels behind it.

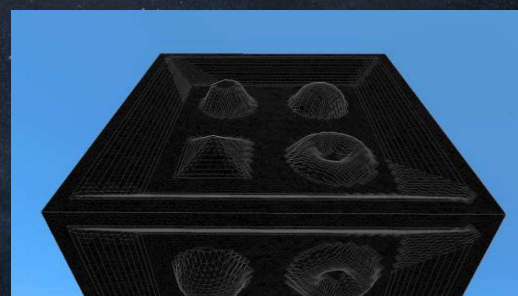


Let's take a look at what happens if you don't tessellate every pass that happens to that object…

ONCE YOU TESSELLATE, YOU CAN'T GO BACK

- Once you tessellate, you always tessellate…
  - Tessellate for every pass, or else..

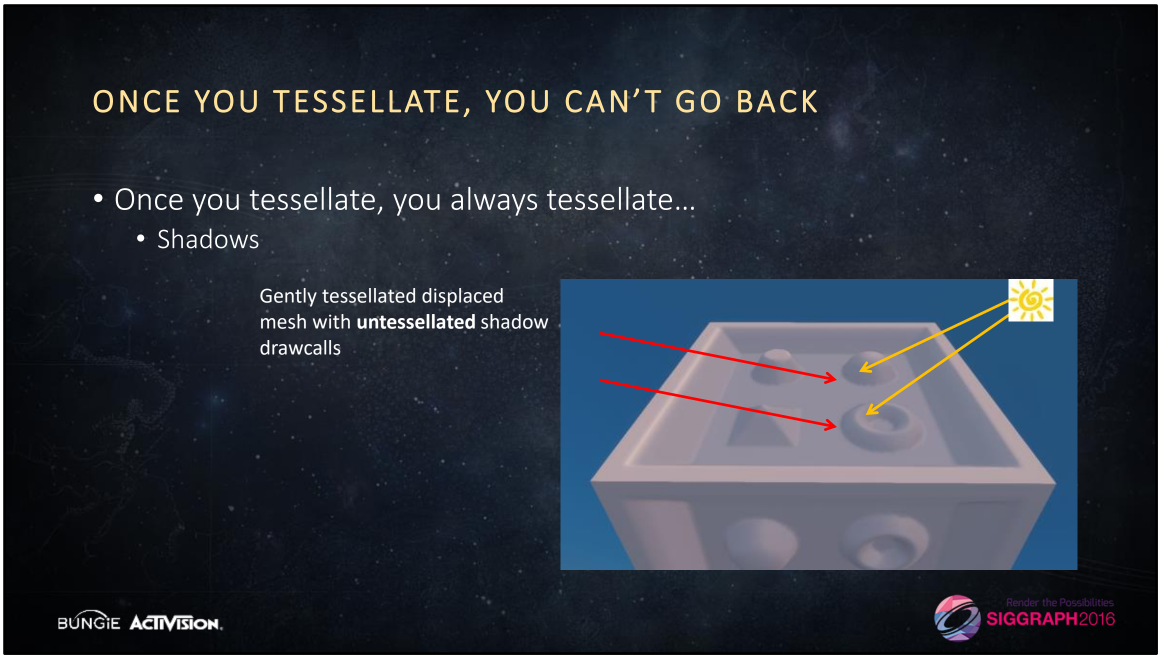
Gently tessellated cube



BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

Let's take a look at what happens if you don't tessellate every pass that happens to that object…

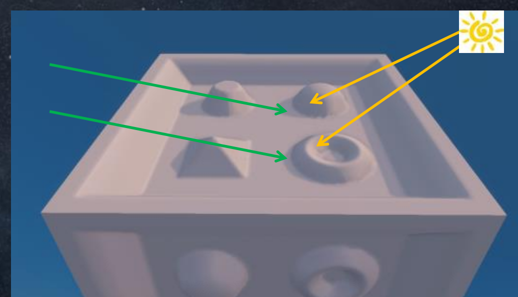


Let's take a look at what happens if you don't tessellate every pass that happens to that object…

ONCE YOU TESSELLATE, YOU CAN'T GO BACK

- Once you tessellate, you always tessellate…
  - Shadows

Gently tessellated displaced mesh with ***tessellated*** shadow drawcalls



BUNGIE ACTIVISION

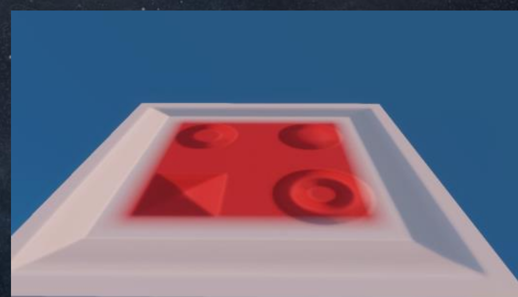
Render the Possibilities SIGGRAPH2016

Let's take a look at what happens if you don't tessellate every pass that happens to that object…

ONCE YOU TESSELLATE, YOU CAN'T GO BACK

- Once you tessellate, you always tessellate…
  - Decals

Untessellated mesh with
untessellated deferred decal



BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

Let's take a look at what happens if you don't tessellate every pass that happens to that object…

Let's take a look at what happens if you don't tessellate every pass that happens to that object…

Why Is It So Hard?

- Content source representation changes
- Art tools pipeline
- Runtime performance
- Hardware behavior

BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

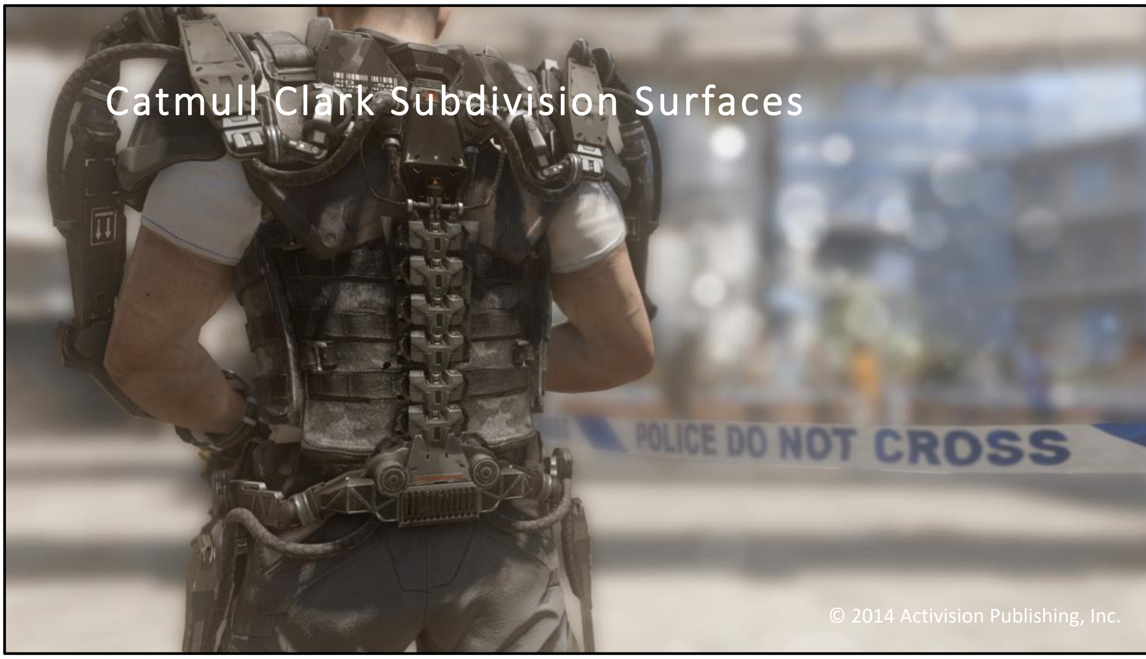
Tessellation Adoption Challenges: Hardware Behavior

- Fine-grain displacement features are often missed due to fixed tessellator pattern
  - Post domain-shader vertices can be at wrong locations, missing displacement features present in the texture
  - Displacement map is typically parameterized by the main UVs which are not tuned to maximize displacement feature density

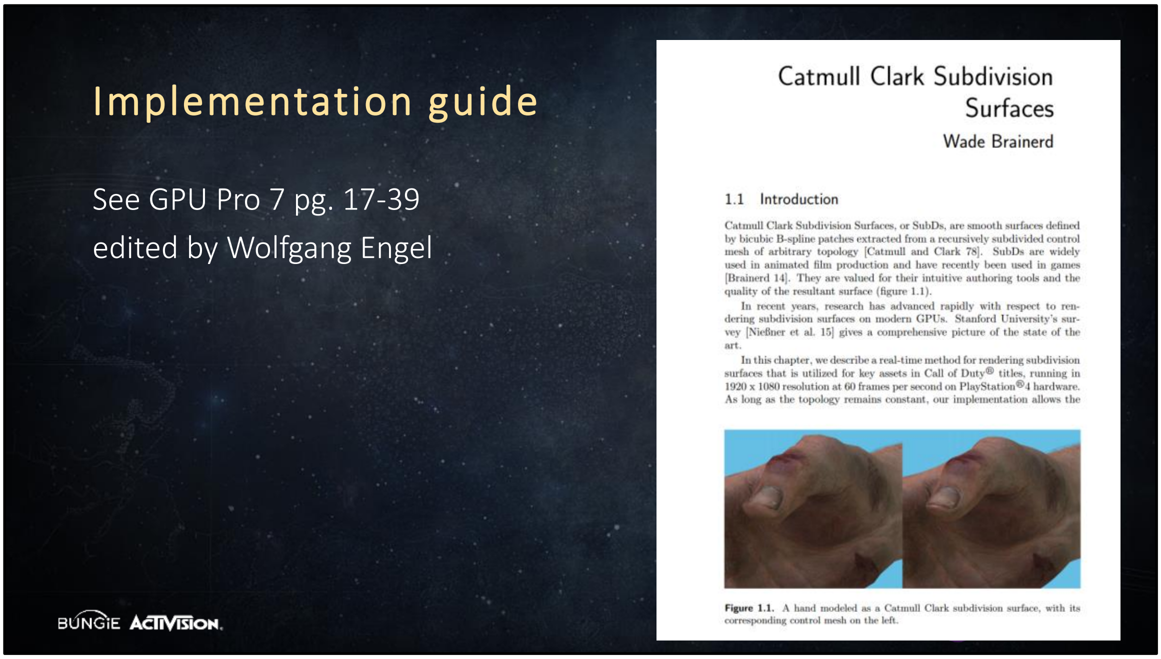
BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016

Now that we've highlighted some issues in our existing content creation pipeline, I'm going to post mortem our efforts to introduce new content representations into the Call of Duty asset pipeline.



In 2013 with Ghosts, we added a real-time subdivision surfaces renderer based on Feature Adaptive Subdivision.



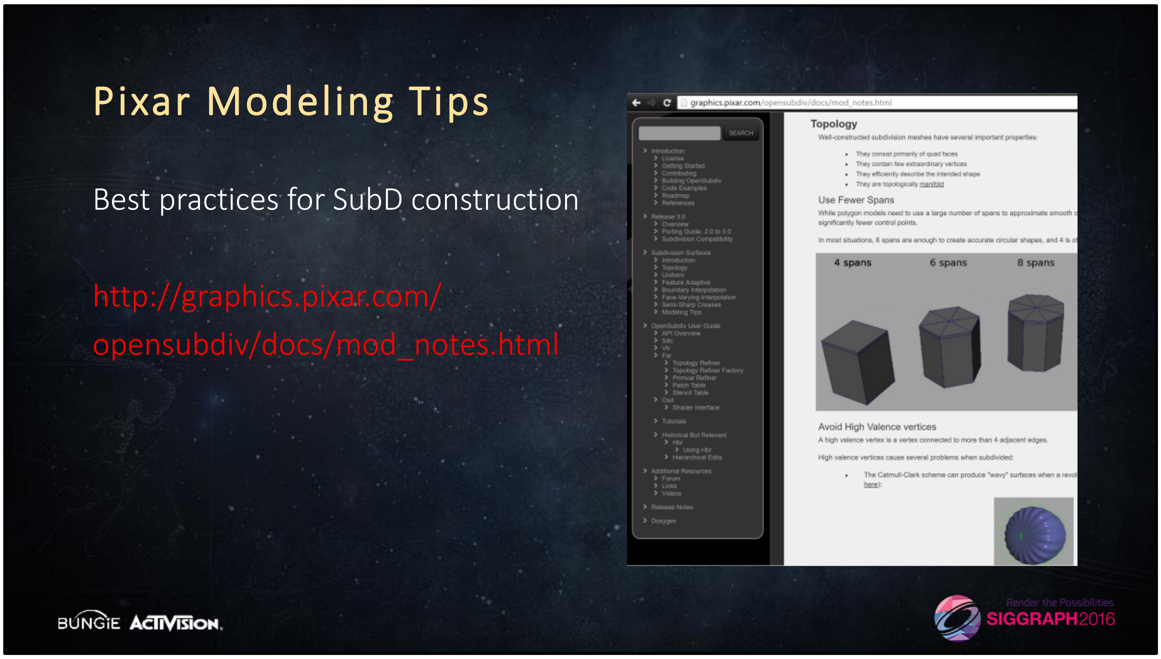
A pretty thorough description of our implementation can be found in the latest book in the GPU Pro series.

The pipeline was initiated at the request of Infinity Ward's lead artist, who felt it would be a great way to establish the model LOD for the new PlayStation 4 and Xbox One consoles.

However, other artists found it was more difficult to build these assets than they expected. They had to learn new modeling rules to create good geometry, and this cost more time than they were saving from modeling the simpler primitives.

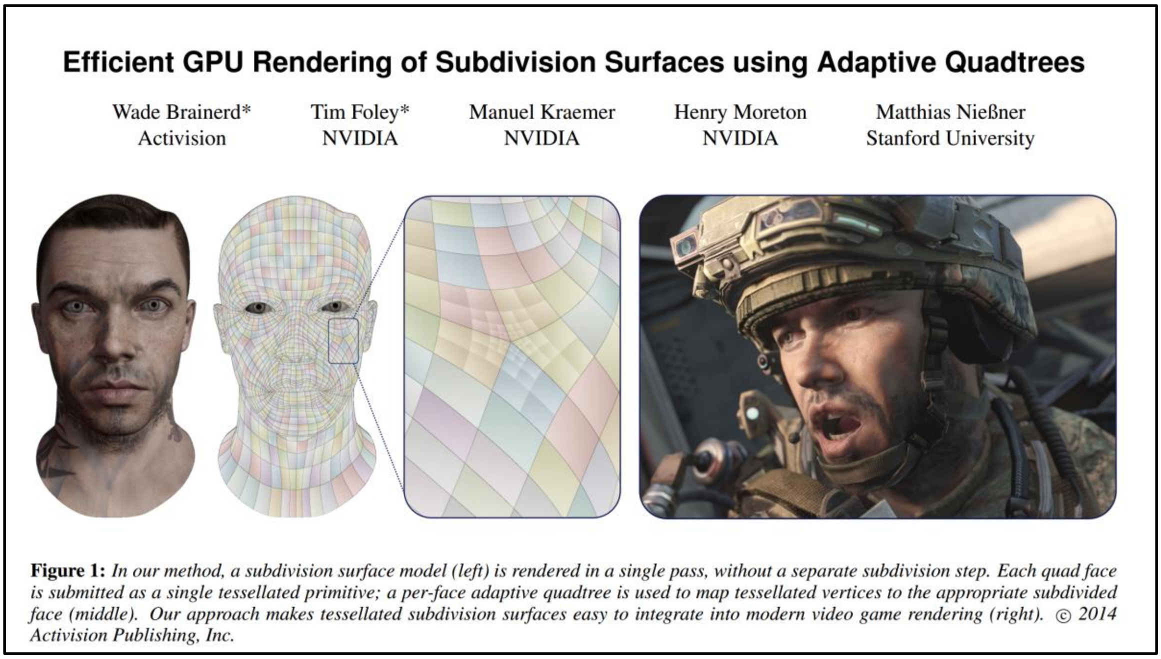
Additionally, normal map baking inconsistencies between the SubD representation and the triangle representation lead to lighting inconsistencies in the LOD chain.

There is a real difference between modeling subDs for baking and modeling subDs for real-time. For baking, you can just press the 'smooth' button and your mesh is generated. But if you do that for real-time, you will end up overtessellating and get garbage for your results.



For a primer on effective SubD modeling, see Pixar's guidelines that are published as part of the OpenSubdiv documentation.

The runtime performance of the pipeline was acceptable but not compelling compared with modeling high triangle counts, and the tables and other supplementary data required by Feature Adaptive Subdivision added memory costs which neutralized savings from using a simpler representation.

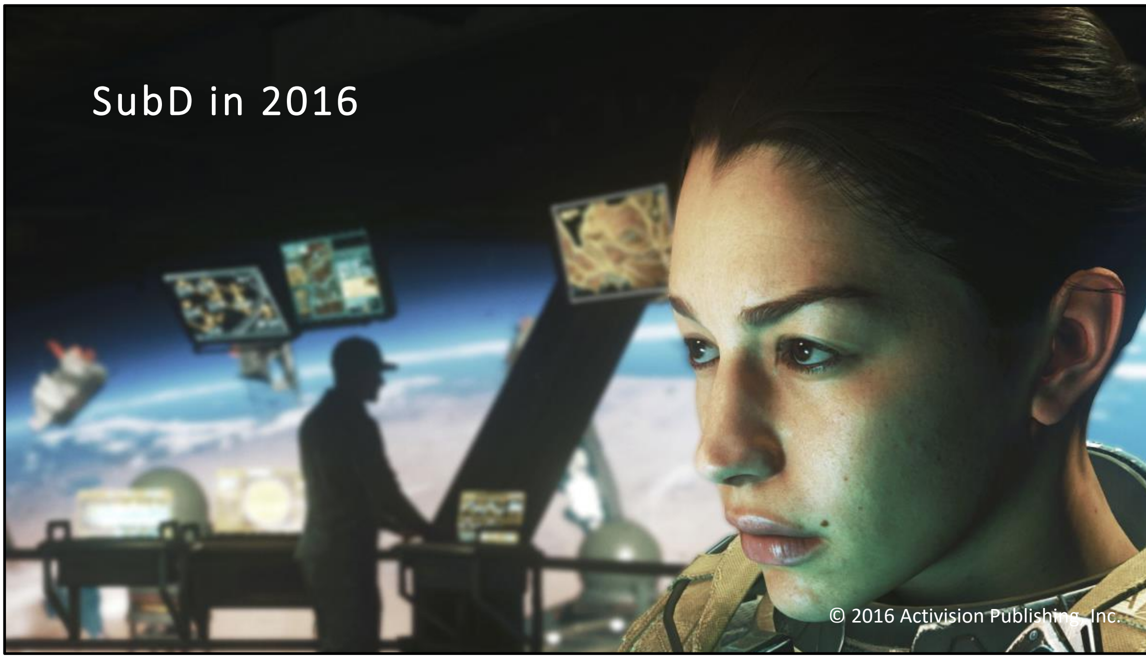

Efficient GPU Rendering of Subdivision Surfaces using Adaptive Quadtrees

Wade Brainerd* Activision
Tim Foley* NVIDIA
Manuel Kraemer NVIDIA
Henry Moreton NVIDIA
Matthias Nießner Stanford University

Figure 1: *In our method, a subdivision surface model (left) is rendered in a single pass, without a separate subdivision step. Each quad face is submitted as a single tessellated primitive; a per-face adaptive quadtree is used to map tessellated vertices to the appropriate subdivided face (middle). Our approach makes tessellated subdivision surfaces easy to integrate into modern video game rendering (right). © 2014 Activision Publishing, Inc.*

In upcoming titles we hope to mitigate some these issues using Adaptive Quad-Tree subdivision (see Tim's AQT talk), which is up to 3x faster and has significantly smaller memory requirements.

This is a great example of a coordination between industry & academia delivering a practical solution that can enable new content representations in games.



However, usage within the titles is ultimately be driven by the artists preferences and we have seen a mixture of use.

This is one of the biggest challenges with introducing a new content representation: the artists have to be behind it all the way, and the technology has to become 100% solid very quickly, or else people will fall back to what they know.

In 2016, use of the SubD pipeline is mixed.

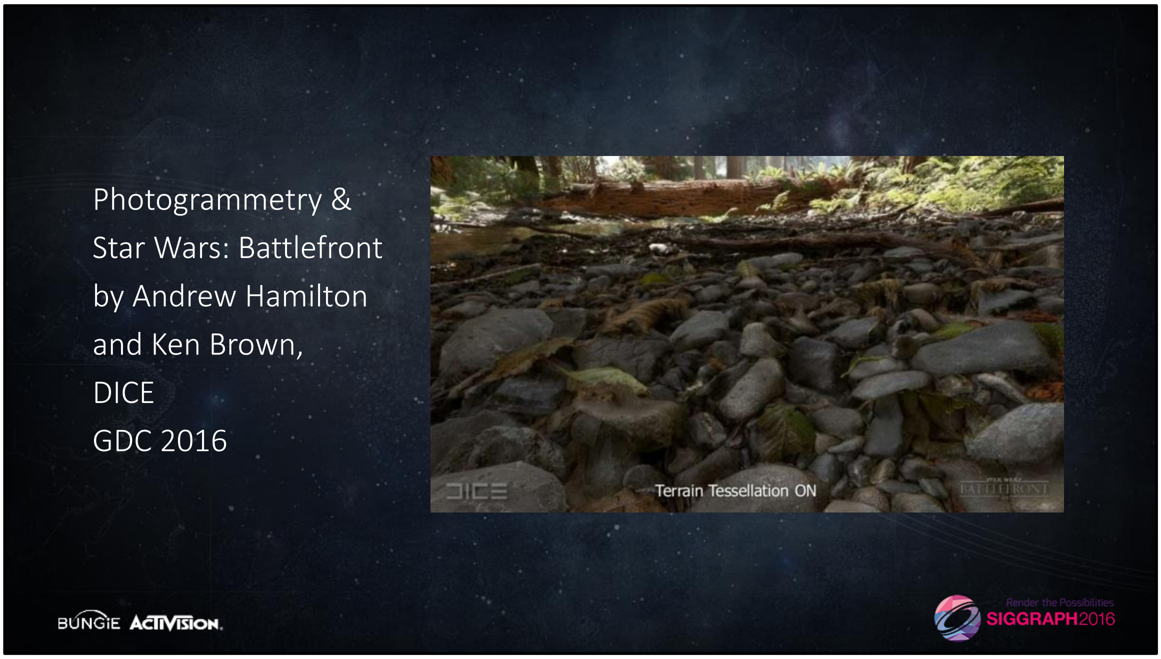
It's still used for many "hero" assets such as principal characters' faces, but the dream of every high LOD prop and vehicle being a subdivision surface has yet to be realized.

I was recently told that the first level of Infinite Warfare SP features several hundred tessellated croissants, so there is that.

Our second attempt to introduce a new content representation was displacement mapped terrain.

This was a programmer-driven initiative but senior artists were on-board from the beginning.

We were able to achieve excellent visual results but at a performance cost, and this again led to underwhelming usage of the new pipeline, despite last minute optimizations that improved performance significantly.



Interestingly, after I described the technology at SIGGRAPH two years ago and another major developer implemented and shipped something similar, internal interest was re-kindled.


Sharing works!
Star Wars Battlefront
By DICE (Electronic Arts)

We’re now using that pipeline more than ever, and results from our survey indicate that interest in displacement mapping is increasing industry-wide.

Now that 4K is becoming more prevalent, the need for geometry amplification is growing and so the need for displacing to obtain fine-grain details in runtime.

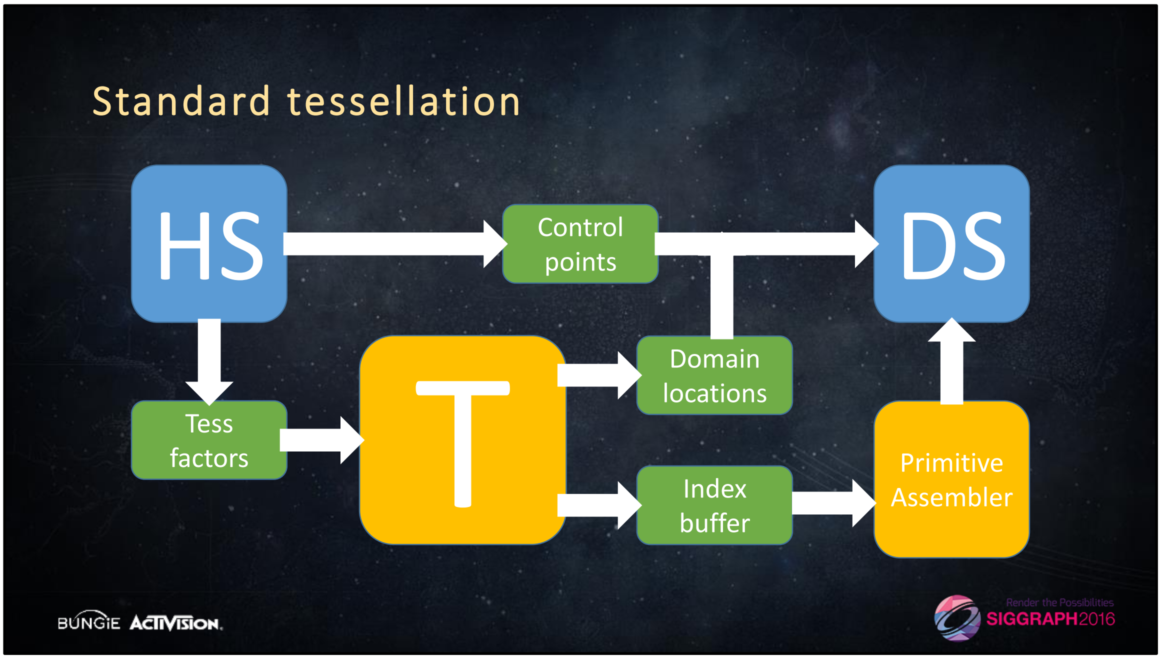
Given that we've been discussing problems with tessellation, we'd like to offer an example of a new direction that might mitigate the issues we have encountered. To be clear, I am not a hardware designer, so proposing hardware changes here is a bit like an artist telling me how to design a renderer. This isn't intended to be a concrete proposal, just as an inspiration to think about tessellation in new ways.

The following proposal is an attempt to extend the underutilized tessellation hardware present in modern DX11 GPUs. It aims to resolve issues with suboptimal tessellation patterns, and to enable the pipeline's reuse as a means for high performance GPU-driven indirect scene submission.

The standard tessellation pipeline consists of: vertex shader, hull shader, tessellator block, domain shader, and pixel shader.

For the purposes of the proposal we'll treat the Vertex shader as part of the Hull shader, since all information from the vertex shader is consumed by the Hull shader. In Direct3D the Hull shader is divided into two parts, but for simplicity I will use the OpenGL model of a single shader.

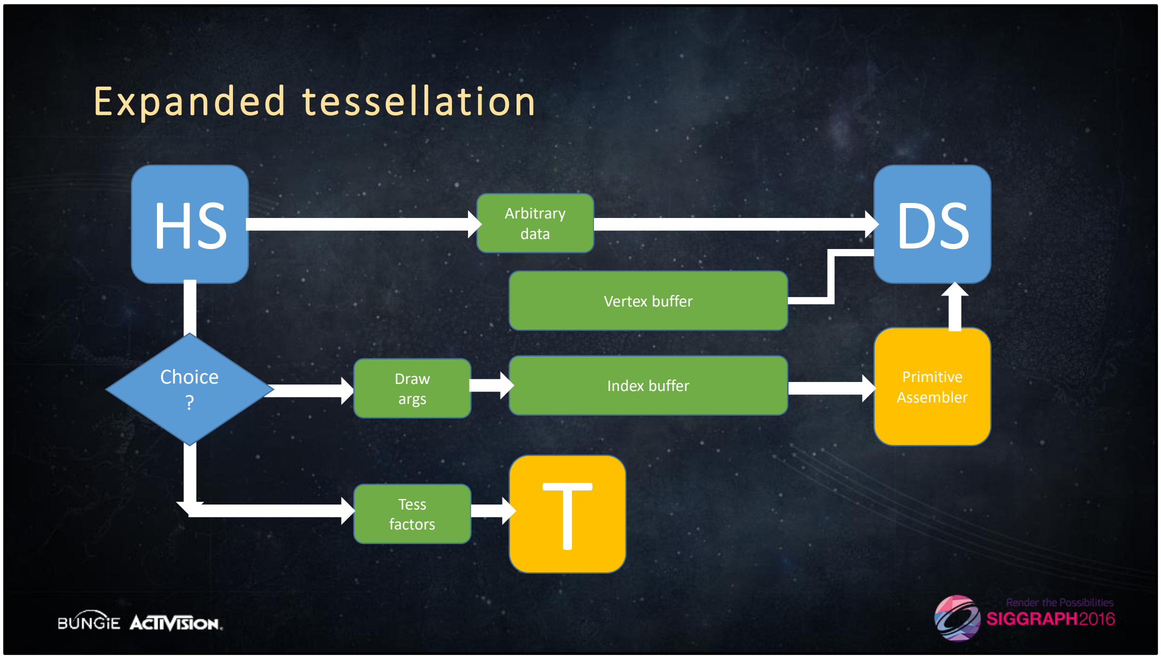
The Pixel shader stage is ignored because its usage is unaffected by tessellation



1. The Hull shader selects tessellation factors (the exact number and their behavior depends on the primitive type) and optionally passes an array of control points to the domain shader.
The Hull shader can cull patches at this stage by returning 0 as the tessellation factor.

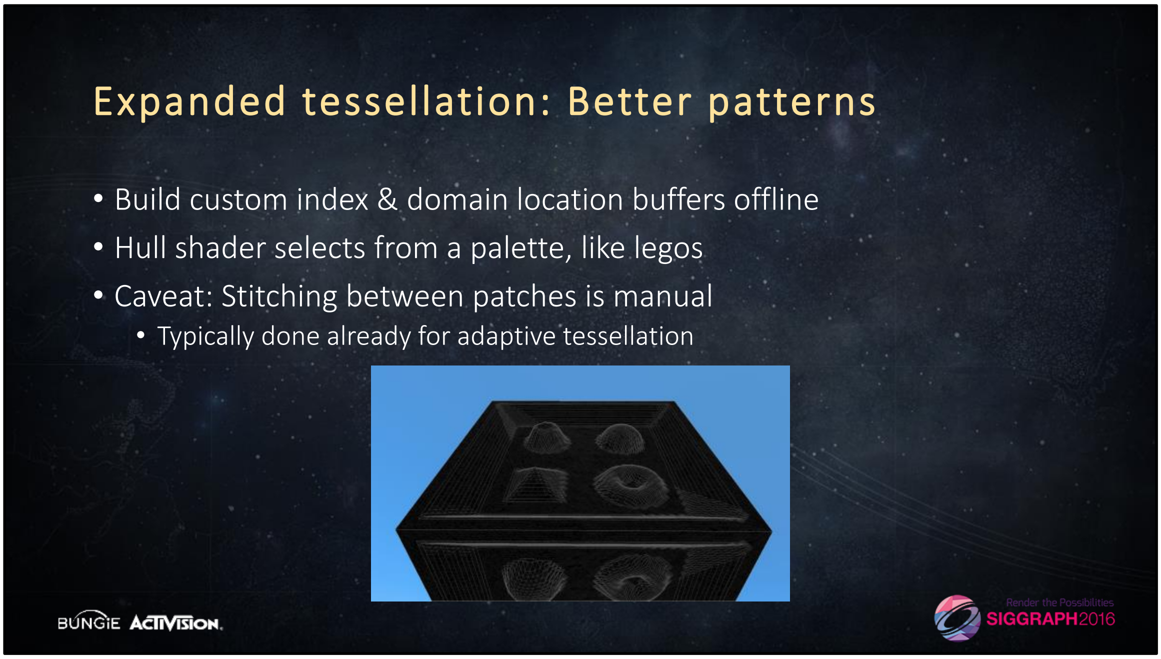
2. The fixed function tessellator block consumes the tessellation factors and effectively synthesizes an index buffer and an array of floating point domain locations across the patch. Modern hardware typically stores the indices and domain locations in a memory-backed ring buffer.

3. The synthesized index buffer and domain locations are fed into the Domain shader, whose output causes triangles to be rasterized.



In this proposal, the Hull shader can choose to return a primitive count and an offset into an index buffer, in addition to a fixed amount of arbitrary parameter data (which utilizes the storage typically reserved for control points).

The choice between tessellation factors and indices is made on a patch-by-patch basis. When indices are chosen, the fixed-function tessellator block is bypassed and the domain shader and rasterizer are driven by the index buffer as in standard triangle rendering.

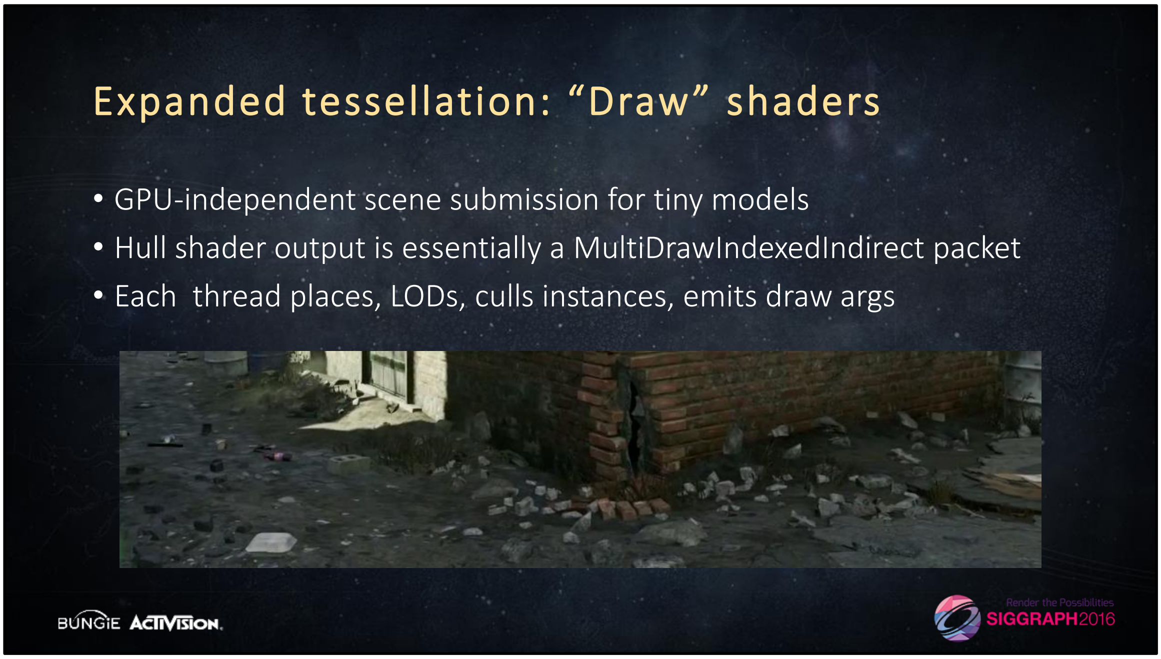


For improved tessellation patterns, programmers construct custom tessellation patterns as a preprocessing step.

The hull shader selects the pattern appropriate to the patch and returns its indices and domain locations from a pre-built buffer. Note that rather than being given a fixed function domain location, the domain shader loads one from the buffer according to its vertex index.

An important caveat is that seamless stitching between patches must be managed by the programmer, utilizing adjacency information in the Hull and/or Domain shader.

While not a perfect solution to the issues with the hardware tessellation pattern, explicit index buffer and domain location control at least make it possible for programmers to utilize the hardware to get good visual results without reverting all the way to compute- or geometry shader-driven geometry synthesis. Because the mode can be selected on a patch-by-patch basis, problematic patches can be treated specially while simple patches continue to use the fixed function hardware.



The other use of explicit index buffers is to treat the Hull shader as a form of “Draw” shader. In the expanded pipeline, the Hull shader output corresponds to the arguments to an API-side Draw call. We can therefore recast the tessellation pipeline as a generator of tiny draws that are treated together as one large draw.

In typical usage, multiple LOD and/or aesthetic variants of models such as debris or foliage can be packed into a shared vertex buffer, and many instances can be drawn by submitting a single “tessellated” DrawPrimitive where each patch corresponds to one model instance.

The Hull shader culls the models against the view frustum, occlusion data structures, etc. and discards invisible instances by setting tessfactor 0. For visible instances, the Hull shader performs a LOD selection calculation and returns the appropriate index buffer offset and primitive count. By passing transforms through parameter data, the Hull shader can even randomize the placement of instances according to a stateless function. Note that instead of interpolating control points using a domain location, the Domain shader simply loads vertex data according to its vertex index.

An alternative way to look at Draw shaders is in the context of Multi Draw Indexed

Indirect Instance. Consider the Hull shader stage a GPU-side on-demand generator of MDII packets, without the need for a Compute shader prepass to generate them. This enables a fully GPU-driven stateless scene submission without CPU interaction.

To conclude, we’ve established that artists are spending effort in a manual and error-prone process to produce a run-time content representation that is not scalable and prevents us from innovating. What can we do about it?

Separation of representations

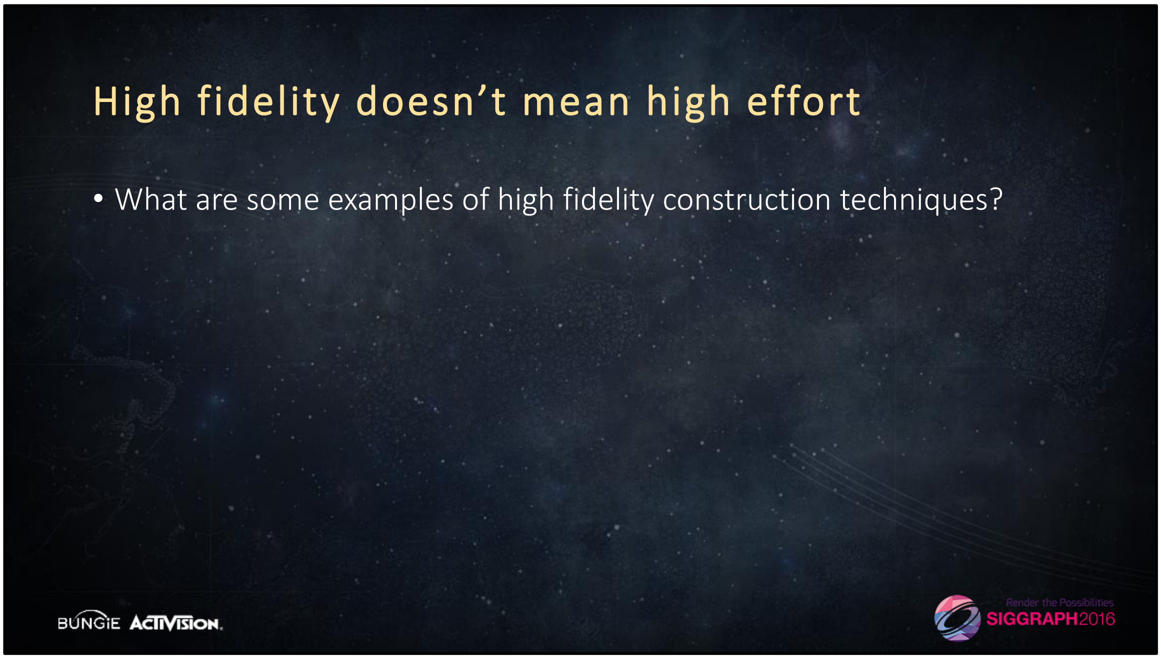
- We have to split the content authoring data representation from the real-time data representation
- This requires that we take control of the entire baking pipeline
- But what representations should we use?

Our challenge to the research community is to develop separable authoring and run-time content representations for games. What we mean by this is that the artists author to the specifications of a high fidelity authoring representation, and an automated pipeline transforms the content into an efficient realtime representation.

The authoring representation should preserve existing workflows for creating high poly models as much as possible, and should require minimal hinting to drive the downressing pipeline.

The transformation pipeline to real-time must be capable of producing triangle mesh LOD stacks and baked textures as required by a modern graphics engine, with maximum efficiency and fidelity to the source assets. It must support animation, rigging, metadata association as required by typical game engines. It must be fast enough to use interactively while authoring to preview results live in the game engine.

Having this pipeline will free artists from spending manual effort to target hardware capabilities directly with their authored representations, saving time and reducing the cost of content changes.



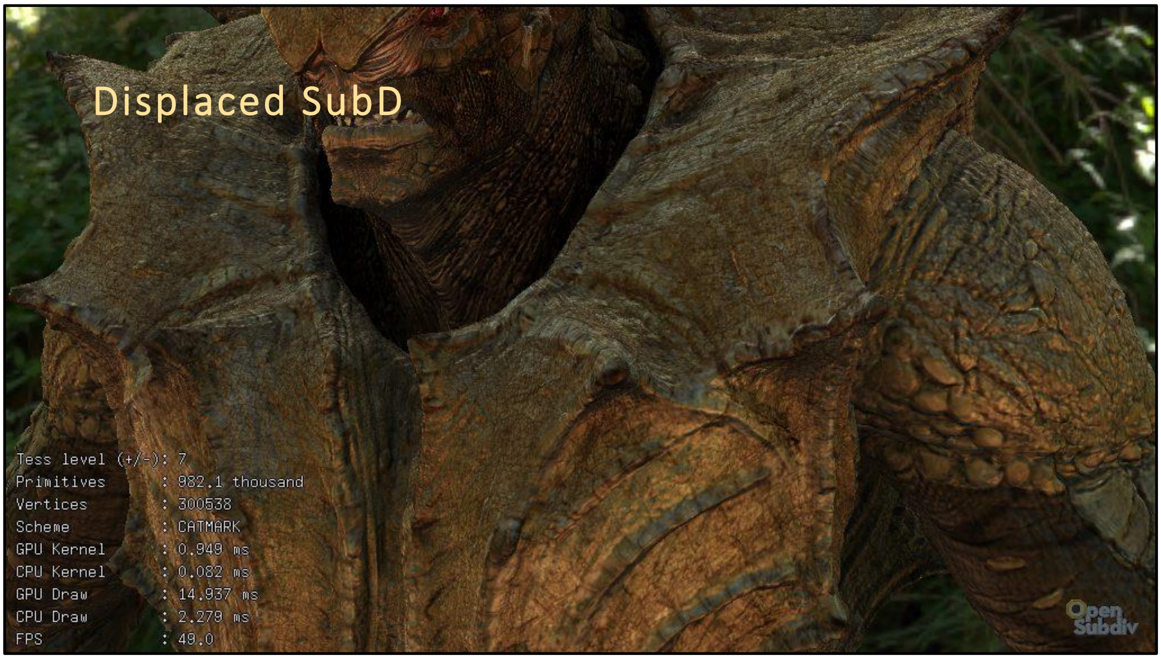
So what are the challenges of creating this kind of automated pipeline? And what kind of high fidelity authored representation should we target?

Here is an example of an asset created for Destiny by Mike Jensen

Make it any way you want, flexible, this could be a fundamental representation
Paint color directly on the high res mesh → vertex colors
Or map it, whatever

How do you rig it though?
How about faces?

Here is an example of an asset created for Destiny by Mike Jensen
Zbrush sculpt of an inorganic asset

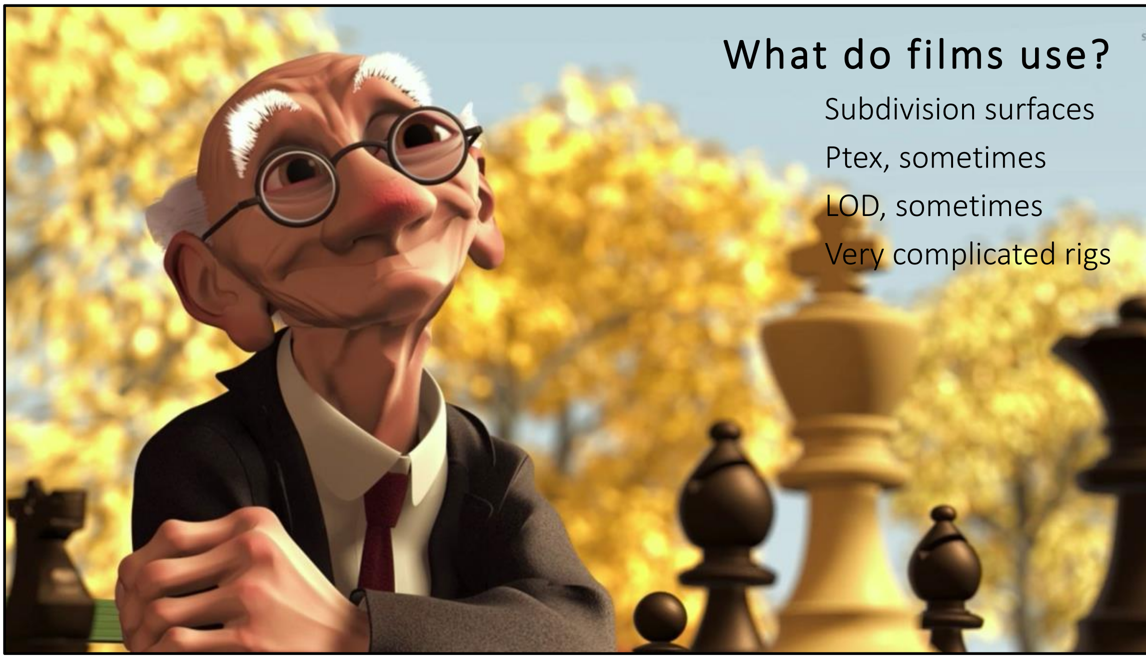


Artists already model SubDs when they use Smooth in Maya
But it would still need simplification, 60k patches is mot feasible for real time.

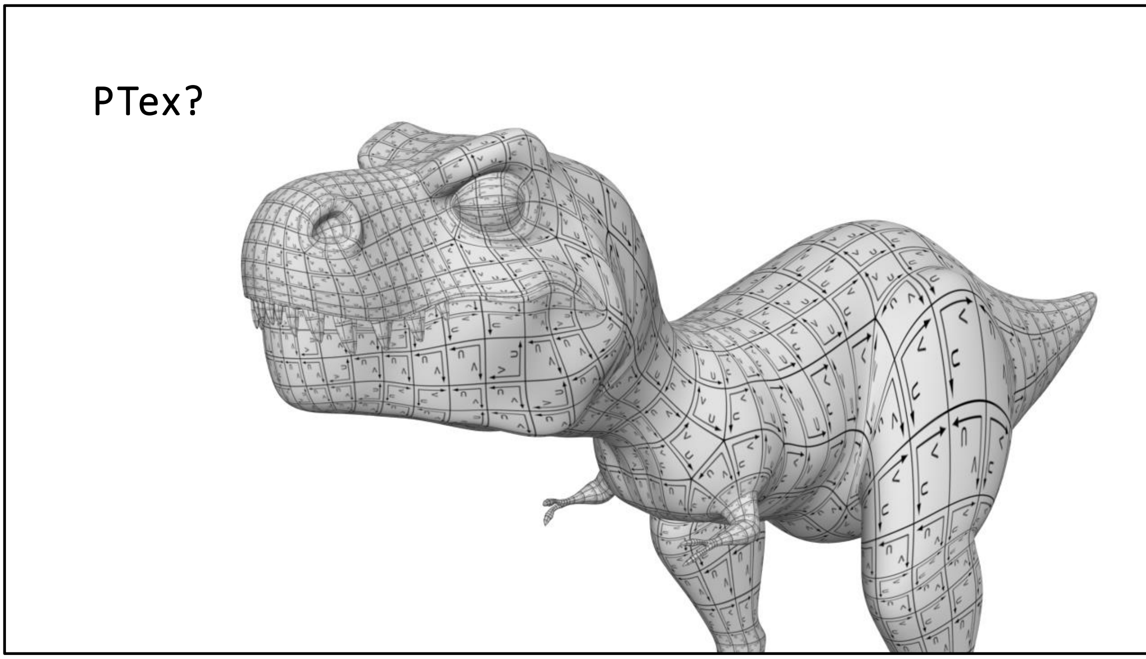
The kitbashing representation is only useful if we automatically generate efficient runtime representation without pushing the burden onto artists
To make this real, we had to convert the individual kitbash models into a coherent runtime representation. This meant

- Merging drawcalls in a smart manner, sorting shaders, atlasing textures and autogenerating LODs from the messily zbuffered kitbashed meshes
- But that was largely transparent to artists which was highly welcome and really opened up their creativity

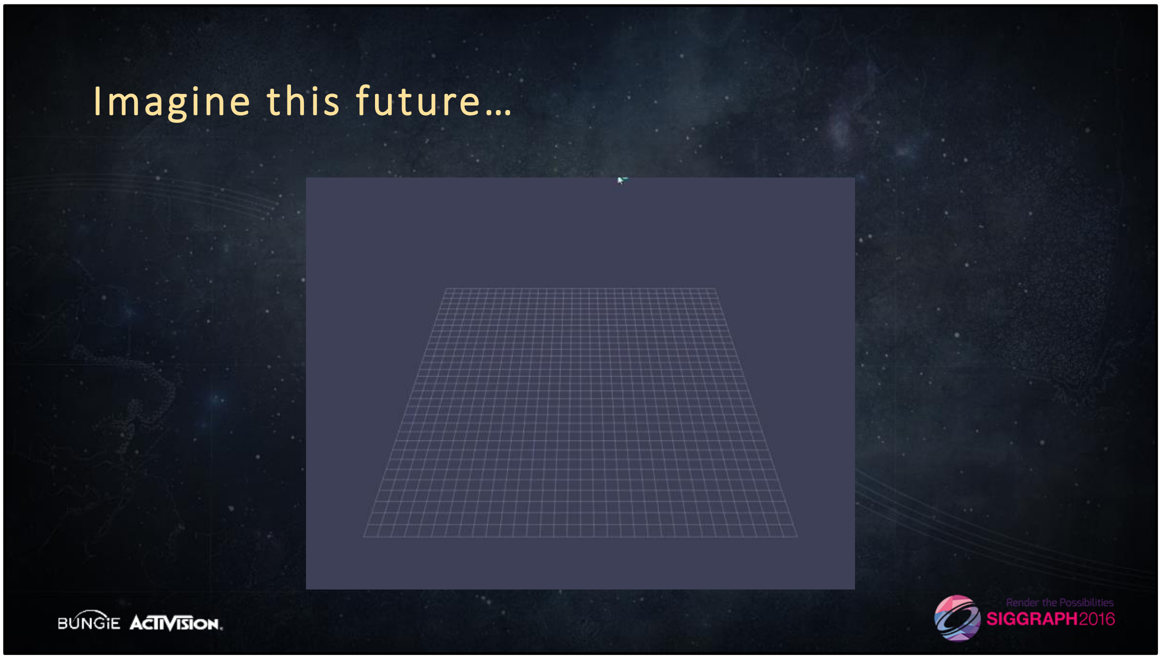
SDFs are an entirely new representation.
How do you rig and animate these?



What representations work in film and why.
Are they appropriate for realtime?



Some film studios love it, some hate it
Again we are talking about separability so real time use is not necessarily important



This video shows a prototype relaxation-based simplification tool, targeting vector displacement maps.

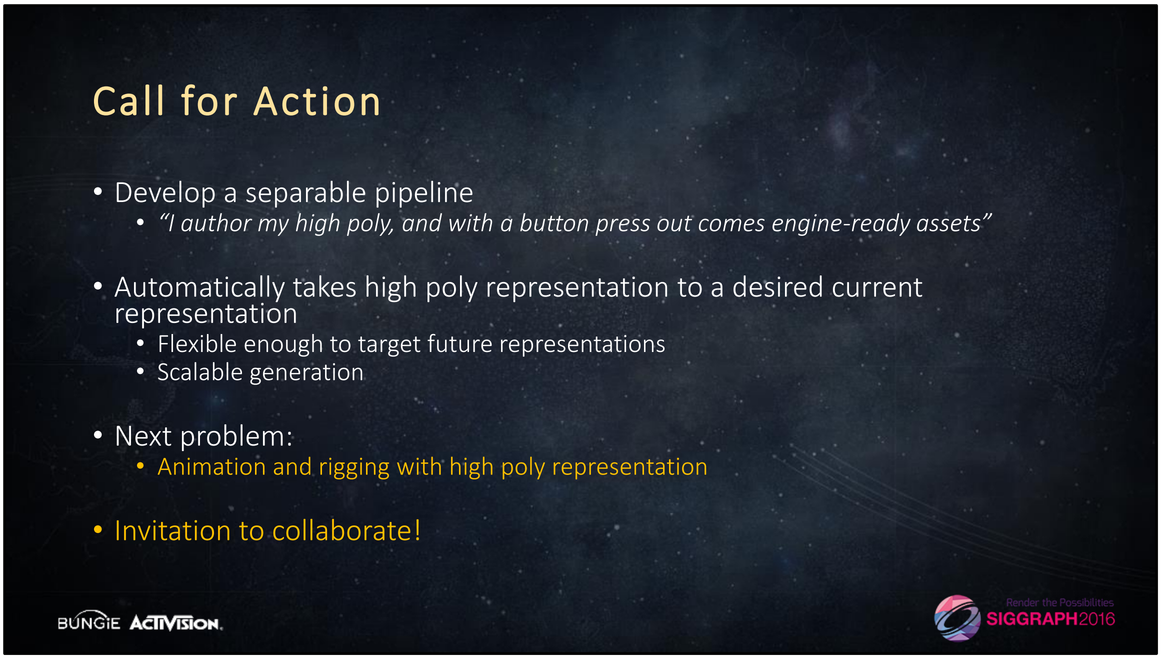
As input it takes a 3d model that is homotopically equivalent to a disk (e.g. a closed mesh with exactly one hole).

The mesh is relaxed onto a square, and a vector displacement map, normal map & color map are generated.

Re-rendering the maps onto a grid using hardware tessellation gives a close approximation of the original mesh, but can be simplified by reducing the tessellation factor.

Up until 2:20 the tool is used to reproduce a simple head model.
Afterwards it is used to transform a million-polygon brick wall model into a tile-able displacement map.

Both models use the same pipeline, demonstrating that very complex inputs can be baked to the same scalable runtime representation.



Develop a separable pipeline – call for action for both development community as much as the research community

Thanks

- Survey artists
- Aaron Lefohn
- Brett Miller
- Rajesh Sharma
- Inigo Quilez
- Graham Wihlidal
- Peter-Pike Sloan
- Danny Chan

BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016


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

We’re hiring too!

ACTIVISION®

https://www.activision.com/careers

BUNGIE ACTIVISION.

Render the Possibilities SIGGRAPH2016

Questions?

BUNGIE ACTIVISION

Render the Possibilities SIGGRAPH2016