OLIVIER DIONNE: Good morning, everyone. And thanks for attending our talk entitled, *Tips and Tricks to Get the Most Out of Your VR Experiences in Stingray.* My name is Olivier. And I work as a software development manager on Stingray's rendering team.

I also asked Ben and Andrew to join me today in order for them to share some valuable information. So we have a lot of material to cover. And let's get started.

I'll first start off by giving a brief overview of our flexible data-driven renderer. And since VR is all about performance, I'll present the profiling tools shipped with Stingray to debug your scenes, and then switch over to present some of the main VR optimizations that we've put in the engine this past year. And then I'll hand it off to Ben to present how quickly you can build VR experiences with basic flow nodes that are shipped with our templates. And Andrew will follow up with best practices, in terms of content optimization for VR.

So Autodesk Stingray is a modern game engine designed for architectural visualization specialists, and also for pro-indie gamemakers. Based on the core technology from the proven Bitsquid engine, Stingray strives to be an open and flexible interaction platform. And as illustrated by a few of these screenshots I'm going to show, Stingray's render can power a broad variety of games and applications.

So this image in the last kind of showed a first-person point of view with more realistic shading. And then we go from more of a top-down view with toon shading. And again, this runs on multiple platforms.

And here's another game with a similar point of view. And then another one built as like a 2.5D side-scroller with more toon shading and different post effects. But other than entertainment, and I imagine of more interest to you, we've put a lot of effort these past few months to improve our overall rendering quality for architectural scenes. So lots of work went into our lighting solutions and post effects to give a sense of grounding and presence to objects. And since Autodesk Live Viewer is built on top of Stingray, it will pick up these improvements in its next release.

And finally, over the last year, we've improved VR performances in general and really made it a first-class citizen of our renderer, giving you a bit more headroom to push the quality bar even further. All the previous apartment screenshots are from a demo that we've built running

in VR at the Future of Making Things booth. And there's some fun interactions you can play around with. So if you have a bit of time, it's worth checking out.

As we've seen with just a few examples, Stingray is able to cater to very different needs with respect to rendering. We ranged from refresh rates going from 30 hertz to 120 hertz different screen resolutions, depending on the target platform, different artistic styles, post effects, and a lot more. And to be able to handle all these different requirements, you really need a flexible renderer.

And for us, this means exposing the majority of our pipeline through human readable data files. And shaders, resource creation, resource manipulation, and the flow of our rendering pipeline is entirely expressed in data. So the format that singer uses is simplified JSON files. So this allows us to make the renderer hot-reloadable for quick iteration times and really fast experimentations.

The main entry point to Stingray renderer is found in files that have an extension of type render config. So these render configuration files drive the entire renderer by binding all rendering subsystems and dictating the composition flow of a frame. They also define quality settings, device capabilities, and default shader libraries to load.

The three key ingredients that build up these configuration files are: resource sets for GPU memory allocation, deallocations, resource generators for resource updating and manipulation, and layer configurations for general frame scheduling. So more details on this part is covered in the handout if you're interested. But the main takeaway message here is that you can tailor the renderer to your needs by modifying the data files without the need to compile a single line of C++ code or having source access.

And having a flexible renderer was key to getting initial VR support up and running quickly in Stingray. However, it's important not to underestimate the many challenges, in terms of rendering, that VR brings to the table. On mainstream head-mounted displays, such as the HTC Vive and Oculus Rift, the requirements are 90 hertz refresh rate with a frame buffer resolution of 2160 by 1200.

But to reduce aliasing artifacts, the off-screen buffer is in reality supersampled to a scale of 1.5x in each dimension. So the final resolution is really 3240 by 1800. In other words, we only have 11 milliseconds to shade 5.8 million pixels for one stereo frame. To really see the impact,

it helps to look at it in terms of the number of pixels to shade per second. So when we put that in perspective, we observe that VR is actually more demanding than running an application or game on a 4k screen at native resolution.

So our first implementation of year and Stingray took the naive approach of performing sequential stereo rendering. This meant creating two distinct scene cameras to represent both eyes and submitting the entire scene for rendering twice. It was originally the easiest strategy to follow in order to support and understand different hardware and the particularities of tracking systems. And at this point, new VR software development kids were also in beta, so quite in flux. And we would usually see updates every three to four weeks.

Obviously, it wasn't going to be our final iteration on general performance due to draw calls and state changes being doubled. This strategy is clearly inefficient. But although we were not satisfied with our initial version, we still demoed it and powered the VR experience in the Live Design booth at AU Last year. And for this short amount of time we had to develop Vive support, we were quite happy to receive a positive review from CGarchitect.

So before we delve into VR optimizations developed over the past year, let's get familiar with the existing performance tools that are available in Stingray today. This is really the first step that you should take in order to build great stutter-free VR experiences. So from the viewport, you can enable the Artist Performance HUD And this is a great way to get a general overview of your performances for the scene you are developing.

You'll see things like the amount of VRAM in use and the number of current visible primitives. Also, statistics on the bottom HUD show how much time is spent in different rendering passes, such as G-buffer creation, shadows, post effects, and more. And this is a great starting point when doing an initial performance pass but we do have more tools.

If you want more details, you can actually launch the external console by pressing Alt-2 from the Editor and access our core profiling tool. So the Profiler window lets you record profiling data that you can share with other developers. It lets you pause and continue the running instance of the engine for trace inspection. The first view you'll initially see is the Frames View. It displays the total time for each rendered frame, letting you easily detect spikes that are worth inspecting.

And if you pause the game and double click on a frame-- so one of those bars-- you'll be brought to the Threads View, which represents all the available cores on your system as black

lines and show the scope time recorded for a specific work on that selected frame. The top three lines are always the GPU thread followed by the Main Game thread, and then the Rendering thread. And below are all the available cores on your system either at work, or available to take incoming jobs spawned from the game or render threads.

The two other views available are the Tree and Aggregate views. So the Tree view gives you the same information as the Thread view, except it's displayed as a tree showing the timing in milliseconds to the right of each individual scope. And the aggregate view is similar to the Tree view, except that it gathers and computes scope timings over several frames using a specific formula.

This last view is useful to find the maximum, minimum, or average times for the scopes shown. So going back to our initial profiling trace when performing sequential serial rendering, this would be our typical frame breakdown. So the amount of work is clearly doubled on both the GPU and render threads for each eye. But obviously, it was a start. And we could definitely improve on our initial take.

So now in terms of optimizations, in Stingray, the camera frustum is used during different stages of our pipeline. So first, we use it to optimize our rendering by performing frustum culling. This step gathers and prepares all objects for rendering that are either partially or totally inside the camera view volume, while others are discarded. So we also use the frustum to compute cascaded shadow maps in order to reduce perspective aliasing of shadows. In terms of work, this means we slice the frustum into four different sections and compute a new shadow map from our directional light for each different section.

And finally, we use the frustum during our clustered deferred shading pass. This step voxelizes the view volume and stores the different local light contributions inside the affected voxel buckets. This data is then passed onto the shaders to compute the final light contribution.

Doing these three different actions twice for each eye is costly and unnecessary. Considering that the general view direction for each eye is equivalent and the distance between both eyes is quite small, it's worth computing a single enclosing frustum to minimize computations on the render and GPU threads. And then just compute the mapping between both eyes.

Another optimization that we use is the hidden area mask to early out on pixels that aren't visible in the final image, as seen through the [? H&D. ?] In other words, the first thing to be

drawn in our depth stencil buffer when in VR mode-- I'm sorry. So in other words, this is the first thing that we draw in our depth stencil buffer in VR mode. So this ensures that we call out geometry that you can't see, and apply as post-processing only to visible pixels.

And finally, another main VR optimization that was to implement instant stereo rendering-- so this means using hardware accelerated geometry instancing to produce both the left and right eye version for the scene in a single rendering past instead of two. So distinct eye transforms are stored in the [? constant ?] buffer. And stereo rendering is handled by the vertex shaders.

So even instances using the left eye matrices, and the clip position is shifted to the left, while odd instances use the right matrices. And they're shifted to the right. And we also adjust the clip plane dynamically to prevent spill over opposite eyes. So this removes the need to double [INAUDIBLE] and state changes.

Now if we compare the optimized results to our previous sequential serial rendering trace, we observe that everything is done in one rendering pass. And profiling traces show about 40 to 50 performance gains on the render thread, while we're more in the range of 20% to 35% faster on the GPU thread. And that's mostly because know shadows being done once, early bails on pixels, and just general shader optimizations.

Notice the timing on our previous strategy, which was about-- if we look at the top thread-- so it was about 7.5 ms versus the new version, which is about 4 ms. So the takeaway message here is that you have more GPU cycles now for increased VR quality.

But that's not all. We still have plenty of work ahead of us, in terms of performance, that we'd like to tackle over the next releases. So we'll be looking into supporting adaptive VR quality. So in other words, if the hardware can't support the scene, and we notice that we're missing frames, then we'll reduce the supersampling scale factor so that VSync is respected.

On the flip side, if we have the extra bandwidth, then we can increase that value for better quality. And we'd like to handle this dynamically within our renderer so users don't have to worry about it. We're also interested in foveated rendering in order to add more pixel density for sharper areas of the lenses for increased quality and performance, basically. And then we're looking into multi GPU setups. And we're actively collaborating with NVIDIA on this. And finally, we'd like to bring the majority of these optimizations to the mobile platforms.

So now I'm going to hand it off to Ben so he can show you how to build quick and interesting

VR experiences using flow nodes provided in our templates.

BENJAMIN SLAPCOFF: Hello, everybody. So in this section, I'm going to go over a few examples of VR functionalities built using the flow visual scripting language. The goal here is just to show you how easily you can build your own functionalities for your VR projects using flow.

So before getting into the examples, I'll give a brief overview of the VR templates that ship with Stingray that will help get you started creating your own VR projects. We have templates for both the Oculus and HTC Vive. And we tried keeping them as similar as possible to make it easy to develop for one or the other. Both our Oculus and SteamVR systems have very similar Lua API calls, as well as a similar set of flow nodes.

So just to give you an idea, here are the flow nodes for our VR systems, these ones being the ones SteamVR. We have input nodes that deal with things like button presses, touch input, or haptic feedback, linking nodes, which deal with attaching objects in your scene to the devices that are tracked by your VR system, such as your controllers or your headmount display.

You have device pose nodes that get you information about where your track devices are located and how they're oriented, tracking space nodes, which deal with mapping the real-world tracking space defined by your VR system to the virtual environment of your project. And finally, we have effect nodes for the Vibe that lets you do simple fade-in and fade-out effects.

So similarly, here are the nodes for the Oculus. There may be a few discrepancies between the two. But all the essential functionality remain the same. These nodes, the SteamVR and Oculus nodes, in conjunction with the default flow nodes, cover all of the functionality necessary to implement any behavior you want for either device.

Getting back to the VR templates that ship with Stingray, both templates have their initialization done in either the Oculus or SteamVR Lua files. These files handle things like, setting up the camera to be used by our VR system or calling the set-up API methods or spawning the controller units. As well, each Stingray project also has a settings.ini file. And in the VR templates, it's used for things like telling the renderer we actually want to render for VR or specifying the VR render target scale, which is used for super sampling purposes, or even defines how the mirror window, which is the desktop view of what we see in VR, should behave.

Finally, the VR templates also come with lots of pre-made functionality built in flow. Here are

some graphs for doing things like teleporting, picking up objects, or setting up controllers. As you can see, they're quite complex. But if you break it down, nothing too complicated is happening.

Obviously, we don't have time to go over everything shown here. But I'll give you a few simplified examples. And hopefully, it will give you an idea of how to work with flow.

So we'll start off here blank, in a room with a gift box. And we want to reach the box. But we can't move. So a good way to get there would be to implement a teleportation mechanism.

In VR, users can start feeling nauseous when there's a mismatch motion between what the user sees and their actual movements. Thumbstick input, like a traditional video doesn't necessarily lead to good VR experience. So teleportation is a good way to move around a level without feeling that sense of nausea.

So here's the flow graph for our teleport functionality. Basically, what we want to happen, is we want to teleport to the location we're pointing at with our controller when we pull the trigger. So this is where we start. This is the node that detects when we press a button on our SteamVR controller. We use it here to detect when we pull the trigger.

And when we do pull the trigger, it leads into the raycast node. The raycast node here is what we use to get the location of where we're pointing at. It shoots a ray from a specified position in a specified direction until it hit something or it exceeds the maximum length we set.

In our case, we want to shoot a ray from the controller position in the direction that the controller is facing. And these two nodes get us just that. The SteamVR controller pose has information about both its position and its rotation. So we use the controller position as the start position for the ray. And then we get the forward vector of the controller's rotation to get the ray's direction.

Finally, we can use the raycast his hit position to set the new SteamVR tracking space position. So what is the tracking space? Well, with the Vive, users can find play areas that they can move around in. The Lighthouse sensors that come with the Vive create a bounding volume that the HMD and controllers are tracking in. So when we set the tracking space position, we're setting where this play area maps to maps into our virtual environment.

Let's look at what we have so far with a short little video. You can see that we point to where we want to go. And when we pull the trigger, we end up just there. If you pay attention though,

when we go from one place to another, the transition is really abrupt. And in VR, this can create a really jarring experience for the user.

A simple way to fix that, to ease the transition, would be to fade the screen to black very briefly, move the tracking space to the raycast hit postition, and then revert the fade. And luckily, doing this is as easy as adding the SteamVR fade-in and fade-out node.

Before setting our new tracking space position to the recast hit location, we fade the screen to black. And then once we're at the new location, we fade back in. Another really simple improvement we can add is just showing a marker at our teleportation location, a teleport location, so we can see exactly where we're going.

Doing this is also really easy. Just spot a marker unit. And then when you're holding down the trigger, set the marker's position to the raycast position. And you'll see where you're going to end up teleporting to.

There is also one last more subtle improvement that we can add that will improve the user experience. So what we're doing here, is we're getting the HMD's, the head-mounted f display's, local position, and subtracting it from our teleport destination. So it may not be exactly clear why we're doing this. So here's a little diagram explaining why.

Basically, when we select a position to teleport to, that's where we want to end up. To do this, we move the tracking space. However, the player can potentially move within the tracking space.

So in the first example, we have our teleport destination, the red X. And we set our tracking space to that position. Since your position is relative to the tracking space, you don't end up exactly where we're pointing to. So you can see the green circle, which is you, didn't end up on the red X.

In the second example, we shift the position we set the tracking space to by the HMD's local position within that tracking space so that our teleport destination is exactly where the player ends up. This is what the player expects, and leads to an overall better experience.

So this is our final flow graph with all of the components put together. As you can see, we didn't do anything too complicated. And we end up with a quite robust teleportation mechanism. Here's a video demonstrating what we have so far. Can finally get to the gift box

and see exactly where we're going and have a nice transition to get there. Now we want to see what's actually inside the box. So we have to be able to pick up objects

So for our second example, we'll look at how we can pick up and interact with dynamic objects in our scene. To start off, we'll need a way to know when the controller is close enough to an object to pick it up. This is the controller unit open in the Unit Editor. This is how the user will see the controller in-game.

However, I've added a sphere mesh to the controller and made it invisible. This is how it would look if we had it visible. The sphere has a physics actor associated to it, which we'll use as a physics trigger.

Whenever an object enters the sphere, it will trigger an event inflow, at which point you can store necessary information about-- this nearby objects will be able to pick it up upon button press. So here's the flow for that. The physics trigger node is triggered when an object enters your sphere, at which point you can set flags specifying that we're close enough to an object to be able to pick it up, and as well as keep track of the nearby object.

So that's just what we're doing here. Whenever we're near an object, we set a flag. And we keep track of the unit that we want to pick up.

Next, we actually have to detect when we press a button to pick up that object. So it might seem like a lot of things are going on here. But most of it is handling inputs, making sure that the flag that defines if we're touching something is set, and getting the actual object that were touching. The important parts of this graph that do most of the heavy lifting are the link and unlink nodes.

These nodes can be used to link units to the track VR devices. It's important to use these flow nodes for picking up objects, and not do it by simply setting the position of that object to the controller. The reason for this stems from the fact that Stingray game engine has all of its game logic and calculations done on one thread, and all the rendering done on another thread. In order to ensure that we never starve the GPU and that we always have resources ready for it to consume, the renderer thread actually deals with data that's a frame behind the game thread, and then offloads it to the GPU thread.

If the render thread dealt with data that was processed for the same frame as the game thread, we run the risk of the GPU waiting for the CPU to finish off its calculations. In our VR

systems, the track devices are updated on the render thread to make sure that we see tracked objects exactly at the position that the Vive or Oculus systems have been tracked at, including any position-predicting things that those systems do.

So if you set an object's position in Flow or Lua to the controller's position, that's happening on the game thread. And object will therefore lag a frame behind. Using the link node to tracker nodes specifies that the object you're linking to a track device should have its position updated on the render thread.

Here's a slowed-down example of two different ways of linking an object to a tracker. The blue ball is using the link node to tracker node, whereas the red ball is getting its position updated in Lua. You can see that the red ball is dragging behind.

So with all of that done, we can finally teleport to our box and see what's in the box. We can teleport to the box. And we're able to pick up the cover. And then if we look inside, we'll see that there's a bunch of flow nodes, the greatest gift of all.

With flow, you can create in-depth functionality without ever needing to touch code. The only thing you need to know to get started with flow is understanding with the different nodes do. And the online resources and documentation can help you get on your way in no time.

There's documentation for the Lua API and all the different flow nodes, as well as various different tutorials to help you get started. Once you have all the functionality you want in your project, you still have to make sure it runs well. Andrew will talking next about content optimization for VR.

ANDREW GRANT: Hello. So, optimizing content for VR. As content creators, it's very important for you to make sure your content runs performant in VR, because as you know, if you drop frames and you lose performance, you can make your users sick.

My experience with working with VR, I've worked a lot on taking under-performing Stingray projects and finding reasons why they're underperforming and optimizing them. For instance, I worked last year on the AU Vive demo that you may have seen. I've worked on a number of projects after that working on projects that have come from offline rendering resources.

So dealing with problems of taking content that was made for offline rendering and making it work in real-time is part of what this talk is about. So I've kind of I've come up with this checklist to go through when you are starting out, to tackle the problem. I'll go through each

one of these in detail.

To begin with, what I usually like to do is to try to eliminate any unnecessary costs in my project to begin with. You want to start from a base level and get all the essential things you need. And then at the end, you build back up and bring up the quality.

So to begin with, I reduce overdraw. All of our templates inserting race ship with overdraw set to 1.6. I like to just reduce them to 1.4, or even lower if I need to.

You could even disable TAA, which is our anti-aliasing solution, and switch it to FXAA. But a lot of our rendering quality really depends on the TAA. So this is kind of an extreme step. But sometimes, you have to start from the lowest common denominator.

The next thing I would do is to go through and disable all unnecessary post-processes. As all post-processes come at a cost. The only ones I really like to keep, at least from a beginning standpoint, is bloom and auto-exposure. Things like screen space, ambient inclusion, and screen space reflections are quite costly. So getting rid of those right away to get to a constant frame rate is a good place to start.

The next thing I'll talk about is polygon reduction. I kind of think of this as a red herring because Stingray can really handle a lot of polygons. Its more about how you use them. So so just doing polygon reduction is not in not your only solution, and usually is not even the first thing I start with.

And if you're using Live Design, for example, they will automatically optimize polygons for you. But if you need to do it yourself, there's a couple of tools you can use, like 3DS Max has Optimize and ProOptimizer nodes. Maya has PolyReduce.

If you need a more robust solution, Simplygon is a good choice. And there's some open-source solutions that are really promising, like the instant field aligned meshes that is available on GitHub.

The main thing that I usually run into first is texture blowout. What this means is, when you're rendering in real time, you're really dependent on your GPU. And if you are running out of GPU texture memory, you will absolutely hit performance problems in VR.

And by default, Stingray does not set compression on your textures when you import them, when you drag and drop a texture into it. So that means they come in as an uncompressed

DES. So even if you have a JPEG image that seems like it's small, if it's a high-resolution image, it can become it can be very large as an uncompressed TDS.

The first thing I do for textures, is I just go through and make sure they are all reasonably sized. The tool I like to use to make sure that I am in fact running out of memory is Process Explorer. Process Explorer is available on Microsoft TechNet. Just search for Process Explorer on Google.

It has a GPU tab. And on that tab is a bar. If that bar is maxed out, then you're running out of texture memory on your GPU. And you need to reduce your texture sizes.

To do that I use a tool called XnView. It's a free software that you can download. The reason why I like to use XnView is because you can go to your project and show all the files in your project recursively, all your images, and then sort them by properties, like their size. And then once you sort them, you can just go through and batch process those images and resize them to a reasonable size.

And when I say "reasonable size", I mean most of the time, you're going to be somewhere around 1024 by 1024 for most of your textures. If you're 2048 or 4096, you should have really good reasons for that, like that's very high resolution. And the reason why we're usually using powers of 2 is because of compression algorithms.

Some compression algorithms require these dimensions. Some require a perfect square. And some require multiples of 4. Generally speaking, we use power of 2 in real-time applications.

If your image, when you import it, does not fulfill those requirements of the compression algorithm, we will resize them in Stingray so we don't error out. But that means we're taking your smaller image that doesn't conform to a larger image. And we're not adding quality; we're just creating more waste.

But we're not we're breaking. So it's just wasteful. So you'll want to use the power of 2 for your textures.

When you do set compression on your textures-- that's the next step, is you need to set your pressure and your textures-- depending on the type of texture that it's for, you'll want to set specific compression types.

So for your color textures with no alpha, you'll want to use DXT1. For color with alpha, DXT5,

normal BC5, RMA textures-- I'll speak a little bit more about it later-- you use DXT1. And single-channel linear textures, such as bump maps, and ambient-inclusion maps, roughness maps, etc., that are gray scale we would recommend BC4.

To enable texture compression, you will use the Texture Manager. To get the Texture Manager, just double click on any texture in Asset Browser in Stingray. And then you'll see in the output format there on the right, that's a dropdown where you can select your compression settings.

Here you can discard the largest MIP maps So if you don't want to destructively resize your textures on [? disk, ?] you can do it there. And you'll drop the highest MIPS in your textures. And you and you'll keep the large-- that happens on compile time. So your original textures are untouched. So

After you've addressed all your texture problems, the first thing you'll probably want to look at a shadow casting. To evaluate whether or not you are actually having a shadow-casting problem, you'll go back to your Artist Performance HUD that we saw earlier. The first line here is shadow casters. So if those shadow casters are red, then you have a shadow-casting problem. And you should address them.

The first thing to do is to just eliminate shadows where they're not necessary. So any lights that are casting shadows that are set to cast shadows that don't actually need to cast shadows, turn them off. Objects that are in shadow don't need to cast shadow. You don't get darker shadows inside shadows.

If you're in a room, for example, if your VR experience was in this room alone, the floor does not need to cast shadows. The light fixtures inside the ceiling don't need to cast shadows. And sometimes, things like light fixtures can be pretty expensive if they're detailed. So those are the sort of things you can look for to reduce shadow casting.

Another thing to point out is that point lights are six times the cost of spotlights, in terms of shadow casting. I'll go over a little bit more why that is later. But if you can convert your point lights to spotlights, do that. That would be a good benefit.

And then in general, bake lighting-- you only need live shadow casting for things that are basically moving. Or if you really need sharp shadows, the cascaded shadow maps after much sharper in real time than if you were to just completely bake everything. However, if you need

to, you can bake all the ligthing so you have all bake lighting and no live shadow casting. And that can get great performance. And a lot of projects to that.

Another thing you can do is do create shadow casting proxies. you can use this with the Unit Editor in your DCC. In order to do shadow casting proxies, you would take a high-res mesh or not even a high-res, it's not necessarily polygon count that you need to reduce on a mesh for shadow casting, but it's complexity.

So say if this table were a highly detailed photogrammetry-generated table, you could make a low-res version of that mesh to cast the shadows. And it could be as simple mesh. You don't need as much detail on the shadow casting. And in the end Unit Editor all you would do is you would make the shadow casting proxy invisible with shadow casting on. And visible mesh, you would disable shadow casting.

The problem of shadow casting is mostly a problem of batching. So when we try to render a surface to the screen, we dispatch a batch, or a draw call, consisting of data, like vertex buffer, shader code, etc. To the GPU. And then GPU you drivers then render that surface to the screen. And so to do this, there is some amount of overhead in the dispatching process.

When try to render too many batches, that overhead will start to build up, lowering your performance. So like this guy's bicycle, he can deliver a lot of boxes at once. But the amount of time taking to load up that bicycle may be the thing that's really slowing him down. So sure, the GPU can handle a lot of data. But getting it to the GPU can, a lot of times, be your bottleneck.

To evaluate your batch counts, we'll go back to the performance HUD, the Artist Performance HUD. On the left, you have the overall batch counts, batch merging, that is available, and other overuse of complexity of your scene. And in the middle of the main part is just your batches separated by draw calls, and the amount of primitives.

We're going to focus on just shadowcasting batches and G-buffer for this portion. But know that each of those has a batching cost. So what is a batch?

You're going to get one batch, per material, per mesh, per shadow-casting observer, per camera that you're drawing your rendering from. So for example, in a very simple situation, where you have one cube on a plane with a spotlight, you're going to get to shadow casting batches, because for one for the cube, one for the plane from the spotlight, and two G-buffer

batches for the materials.

If we switch that spotlight to a sunlight, we jump to five shadow casting batches. The reason why we're getting five, in this situation, the plane is very large. And the sunlight is what is a directional light, which is what is driving our cascaded shadow maps.

There are four cascaded shadow maps. And so that means there are-- is it four? So there are shadow-casting observers from that direction of light. So that means the plane generates four batches, because it's in four of those observers. And the cube is in one. One thing to note, if you're unlucky, and that cube happens to be within two of your cascades, then this could be six, depending on where that cube is.

And if we go to an omni-light, now we've got six shadow casting batches. Why is that? The reason for that is, if you imagine the omnidirectional light as a cube with a shadow-casting observer going each direction of the cube, that's what's happening. So if we go back to this scene, this cube and the plane are being hit by three of the shadow-casting observers of that omni-light So you end up with six shadow-casting batches in omni-light situation.

So in order to reduce batching, the first thing you usually want to do is to merge meshes. So if you have a mesh with 10,000 different individual meshes or nodes within it, if you merge that into one mesh, you'll get a big performance boost there. And in order to do that, all you need to do is reimport that mesh. Right click, and reimport and turn on merge meshes.

The other thing to do is to reduce the amount of materials. So sometimes a chair may be two different materials. But all that's different is their texture inputs. So something you can do is create atlases where you bake those individual materials into one texture map. So you don't have more batches for that object.

And then you can create LODs. LODs are similar to your shadow-casting proxy. It's not necessarily just reducing polygon count, although that helps, but but reducing complexity. So in the instance of a chair where you make an atlased version of the materials, the one that's near you can be complex, where the one that's further away can be much more simple, because you're not going to need that much detail in the chair far away, in terms of materials. So overall complexity can be reduced using LODs.

The other thing to do to reduce batches is to use occluders. Occluders are boxes you can place in your scene. Any mesh can be in an occluder.

But in the end, it's the bounds of that mesh that are used. So it's always a box. But we have an occluder box that's included in version 1.6 that you can drop into any scene.

The thing to remember, occluders occlude from all observers. So if you use them naively, you can get some strange results that are unexpected. So for example, if I built an occluder for the ceiling, and we had windows casting shadows on this table, but if I made that occluder for the ceiling too large, and it was bigger than the actual visual mesh of the ceiling, then the the shadow-casting observer of the sun will occlude this table from the sun. So from my perspective, the shadows will disappear.

So you have to be careful with your occluders and make sure you're not bigger than the actual visual meshes that you're occluding. And occluders themselves come at some cost. That's why they're cubes. So you can't do a million of them. But use them wisely.

And the next thing to try to achieve is some batch merging. Batch merging is when you take-- if each of these chairs is an instance, and they're all using the exact same materials, and those materials are supporting instancing, then all of them can be merged into one batch when they're sent to the GPU. The kind of rule for that is to have the same geometry, so instances, same materials. And the materials have to have instancing turned on.

Our standard material when you import meshes does not have instancing turned on. And the thing to know about it is that it's optimized for compatibility. It's not optimized for performance.

So we want stuff that you bring into Stingray to work most of the time. And so it has a lot of switches. It has a lot of texture samplers. And each of those come at some cost. So it's not instaceable because there are all these switches.

If you want to, you can optimize your materials by creating simple materials. And the way I do this is, I start with our standard material. And depending on your data set, some data-- if you're working with a data set that has no ambient occlusion texture, maps then you can just remove that from your standard material.

Our material system has hierarchicals. So you can create parent materials that the child materials will derive from. So I will take the standard material, remove everything that I don't need and I'm not using this project data, and create simple materials.

And then you can save that material as a parent material. And for all your materials in your

project that work with that, you just point to that parent material to get that optimization. And if it's simple enough, you can enable instancing on it.

And then, in addition, we ship with standard RMA material with Stingray. RMA stands for roughness, metalness, and ambient occlusion. And what we're doing there is, you're taking one texture map.

And you're stuffing the roughness, the metalness, and the ambient occlusion into one texture, using the different channels. And it's a very simple material. And it doesn't have any switches and would support instancing.

So in closing, you know working in VR, you're always dealing with tradeoffs. You can't get everything. You can't turn everything on and expect to get good performance. You you always have to make some decisions. But with enough work and effort, you can make anything work.

And know that game artists can help you. If you're coming from an offline rendering world or architectural visizualization, and you haven't had any experience in real time, there's great communities of game developers out there. Those are the sort of people who have been working with real-time technology forever and know all the problems and how to solve them. So look for game artists where you can.

We'd also like to have some acknowledgements from the people on our team who have put in a lot of effort, and thank them for letting us present some of the work at AU. And with that, we've got some time for questions. Yes, sir.

AUDIENCE: I have three questions. Do you guys support multi-tiled UV sets yet?

ANDREW GRANT: The question is, do we support multi-tiled UV sets?

AUDIENCE: Correct.

ANDREW GRANT: No, not yet, I don't think. Like D--

AUDIENCE: [? DEI ?] support.

ANDREW GRANT: Not yet, no.

AUDIENCE: Number one, you were talking about making shadows. Do we have the ability to send that to the render farm to get those [INAUDIBLE] calculated?

ANDREW GRANT: The question is whether or not we can send to render farm for bake shadows, which I think you have an update on, right?

OLIVIER DIONNE: Yeah, that's the idea. It's in the works, basically. But that you can actually-- we have a GPU baker, but we are creating a stand alone version of it. And eventually, we'll get to the cloud.

AUDIENCE: We have to use Backburner [INAUDIBLE] do something like [INAUDIBLE].

ANDREW GRANT: Sir, can you repeat that?

AUDIENCE: Do we have to use Backburner [INAUDIBLE] be able to [INAUDIBLE]

ANDREW GRANT: The question is whether we'd have to use Backburner. I don't think we have--

OLIVIER DIONNE: No. I think you were using Turtle initially.

ANDREW GRANT: Yeah.

OLIVIER DIONNE: But the GPU baker solution that in Stingray works fairly well. So that's what we've been using.

ANDREW GRANT: And we have Beast.

OLIVIER DIONNE: And we have Beast.

ANDREW GRANT: --which has a distributed solution.

AUDIENCE: You mentioned resources online. What is your pertinent source for game developers?

ANDREW GRANT: For game developers? So the question is, what online resources for game developers? Places I would go to, Polycount is really good. So if you're dealing with game art, Polycount is really good, the forums for any game engine. We have forums on [INAUDIBLE] good forums where you can ask questions for those particular engines, whatever those game developer magazine's website-- is it-- Gamasutra, for example.

AUDIENCE: I got one. Scripting, the visual scripting, can that be converted into actual code for further development later? Or is it just an add-in?

ANDREW GRANT: The question is whether you can convert virtual scripting to code.

BENJAMIN SLAPCOFF: Yeah, so basically, the flow nodes that we showed you for the SteamVR Oculus projects, those are all defined within the VR templates. And at their core, there are Lua. And you can see

exactly what they're doing in Lua. And anything that you can do with flow, you can also just do straight-up with the Lua API. So you're definitely not locked in to using the flow visual scripting language if you don't prefer.

AUDIENCE: Yeah, start there. [INAUDIBLE]

BENJAMIN SLAPCOFF: Yeah, exactly. It's really good for prototyping, getting things up and running really quickly, It can do that. But if you're more comfortable with code, then it's--

AUDIENCE: [INAUDIBLE] I want to make sure that I can recycle [INAUDIBLE]

BENJAMIN SLAPCOFF: Yeah, definitely.

OLIVIER DIONNE: So basically, you can use Lua afterwards, instead of using the flow nodes. But you can also do C++ plugins. You can actually develop your game in C++ if you like, so to get that increased performance.

BENJAMIN SLAPCOFF: And you can make your own flow nodes as well, with Lua.

ANDREW GRANT: Yeah that's one of the really powerful things about Stingray, I feel, is that you can really quickly and easily just create Lua flow nodes. So like anything that's missing, or there's a lot of things that don't make sense in a visual scripting language to do. So you can make individual nodes to do those in Lua and expose them very quickly and easily. Another question?

AUDIENCE: [INAUDIBLE] draw call?

ANDREW GRANT: Yes, one draw call for when you have [INAUDIBLE]

AUDIENCE: I'm curious about the standardization of [INAUDIBLE] technologies coming out. What kind of pain is that going to be from Stingray's [INAUDIBLE]

ANDREW GRANT: The question is about all the different hardware that is coming out and how much pain it is to standardized for that.

OLIVIER DIONNE: It's a pain. It does take us a lot of cycles. Different hardware has different requirements. We try to target the mainstream hardware. But everything is kind of exposed.

So we're going to be actually providing our plugins are providing our source. So you can see

how everything is built. And you know hopefully the community can pick that up and actually provide, or help us out, to build for different platforms.

AUDIENCE: Are you guys already working with Intel making a prototype at all? [INAUDIBLE] more, because I [INAUDIBLE] about that one. [INAUDIBLE]

ANDREW GRANT: The question is if we're working with Intel on their hardware.

OLIVIER DIONNE: Currently, we're not. We are looking at supporting the HoloLens and things like that. But no, we haven't discussed with Intel.

AUDIENCE: How's that going, the HoloLens [INAUDIBLE]

OLIVIER DIONNE: So the question is, how's the HoloLens going? It's a different device. It's fairly limited in terms of power. But essentially, what's interesting, is we can reuse a good portion of our pipeline for VR, so like, stereo rendering and instancing, all that. So that's fairly interesting. But it's progressing.

AUDIENCE: [INAUDIBLE] you think [INAUDIBLE] upgrade [INAUDIBLE] that would be [INAUDIBLE]

OLIVIER DIONNE: So the question is to bring in content that you've been developing for the Vive to HoloLens I think to answer answer that, I think the content has to be quite different, because virtual reality, you're kind of just immersing that person in 3D, while the HoloLens is more like an AR device. So you will shade a certain amount of pixels. But the rest, you want that to be still transparent, and you can see reality.

And in terms of doing full-screen passes on the HoloLens that's fairly costly. You actually want to reduce the amount of pixels you're going to shade. So the experiences, I think, need to be built a bit differently.

ANDREW GRANT: If anyone is interested, I'll be at the Future of Making Things booth for most of the day. We're doing VR demos there. You have to sign up at the front-- there's a desk at the front of those booths. So feel free to come sign up and experience Stingray VR. Thanks fpr coming

OLIVIER DIONNE: It's worth checking out the zero-gravity engine.

ANDREW GRANT: Exactly. Thank you very much.

OLIVIER DIONNE: Thank you.