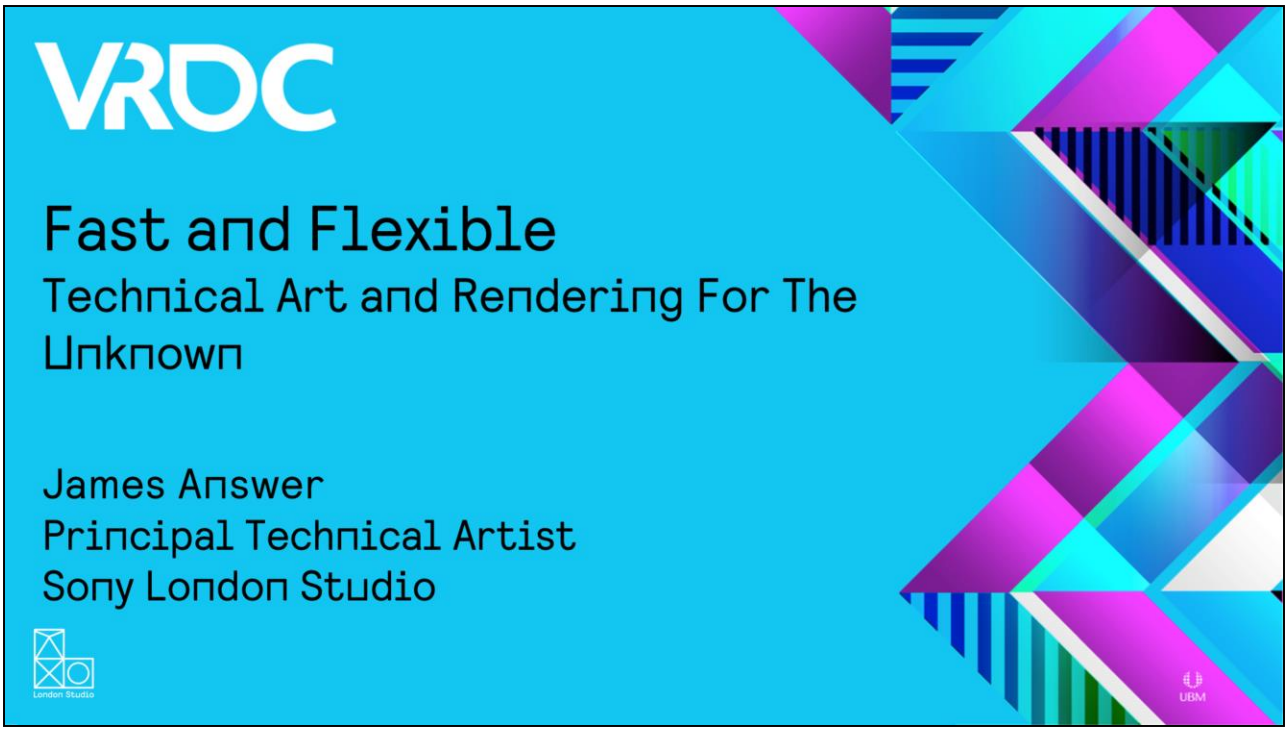
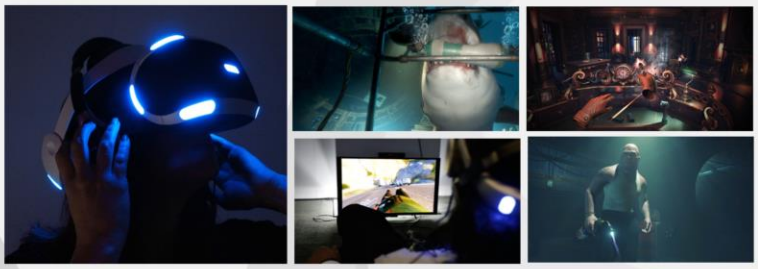

VRDC
Fast and Flexible
Technical Art and Rendering For The Unknown
James Answer
Principal Technical Artist
Sony London Studio
London Studio
UBM

Introduction

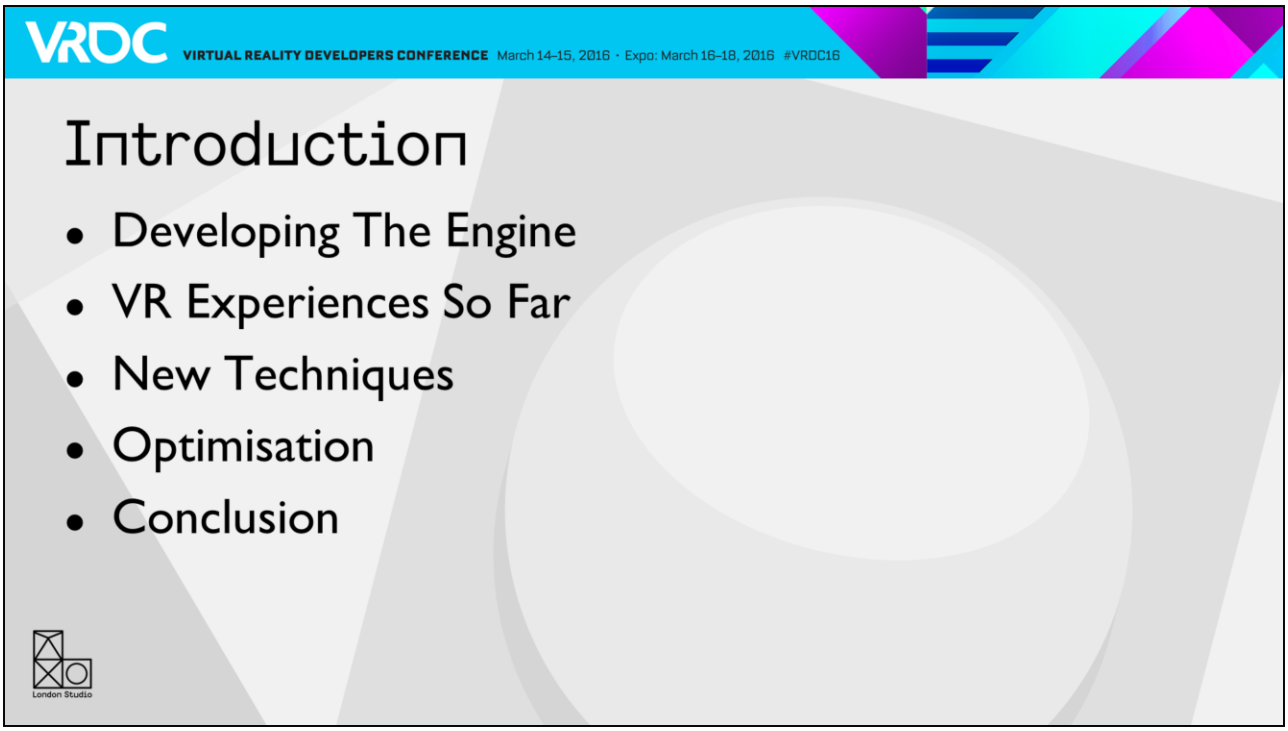


London Studio

London Studio

Hello everyone, and thanks for coming today.

I'm James Answer, and I'm Principal Technical Artist at Sony's London Studio, where I've worked for almost 9 years. The team I am on is currently working on virtual reality for Sony's forthcoming Playstation VR, and I'm going to give you an overview of how we developed our tools and technology around it.



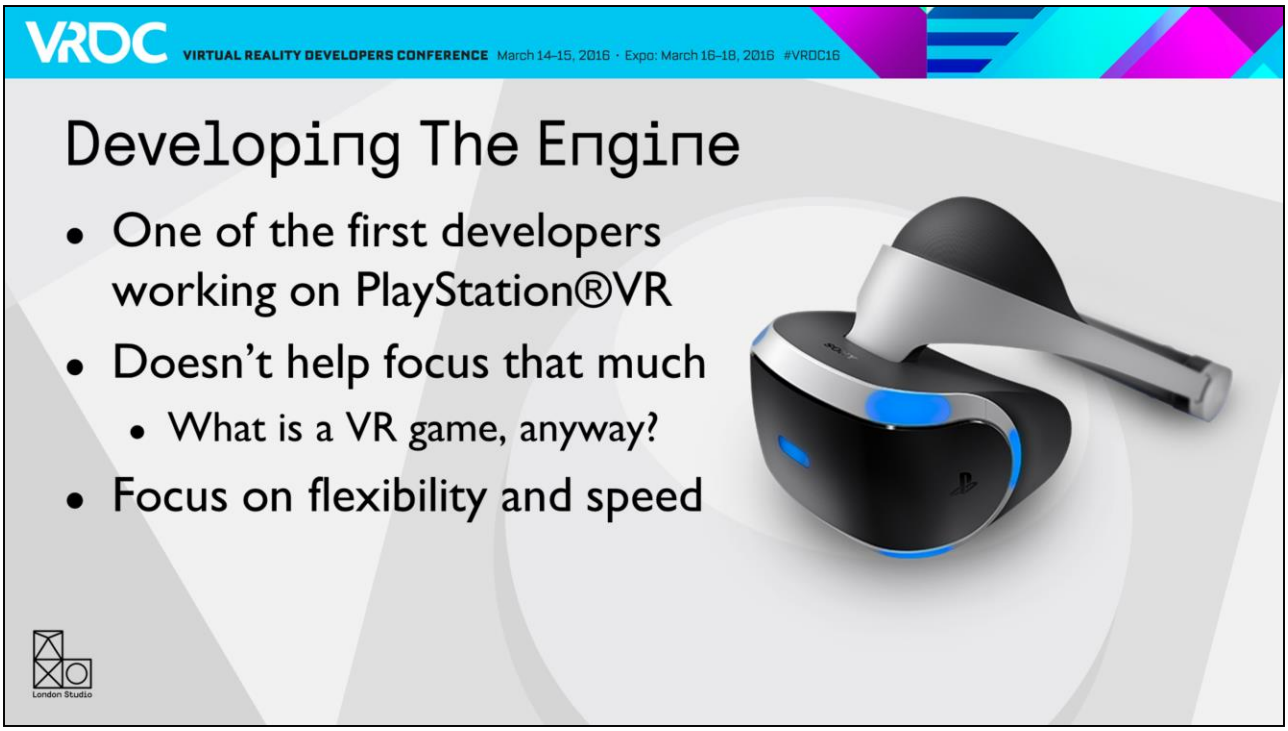
First I'm going to talk about the early work on our new engine that we started preparing for Playstation 4 and VR

Next, we'll have a look at the experiences the studio has shown so far, and how they have shaped our engine

We'll then have a look at the tech that we've developed since last year

And finally, content optimisation for VR

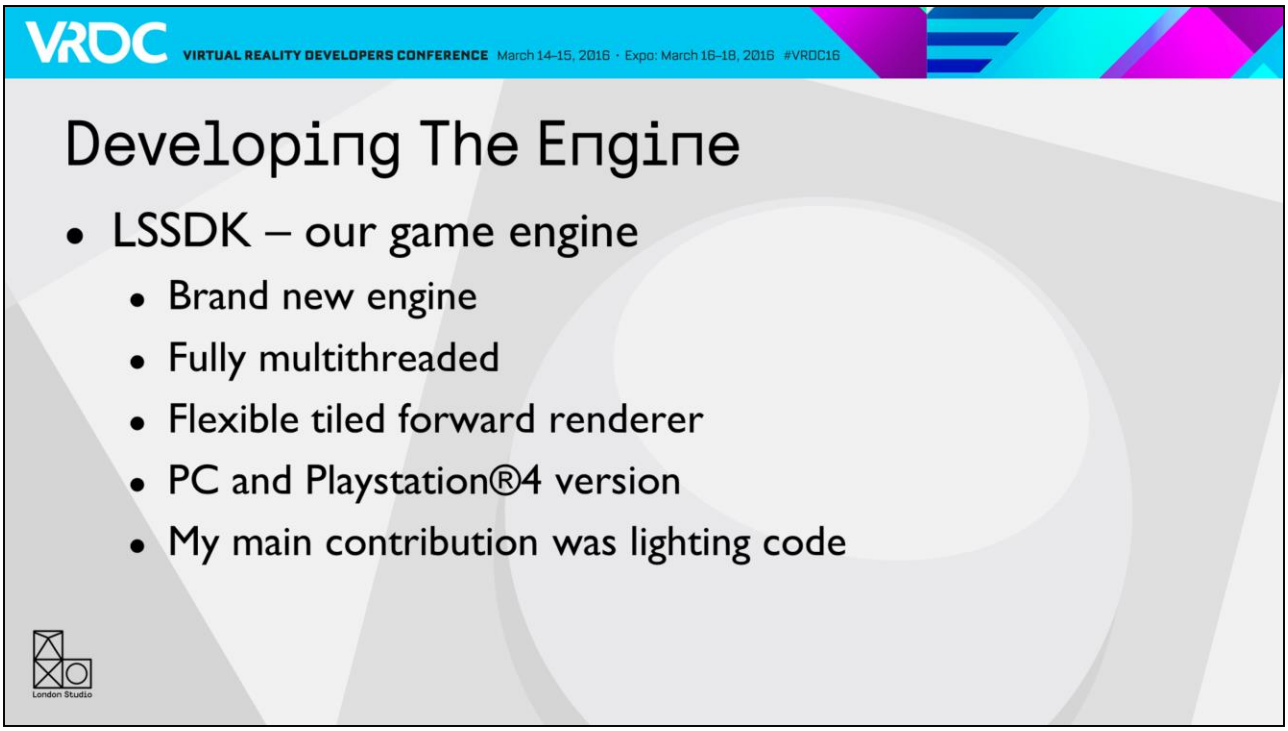
We don't have a lot of time and a lot to cover, but we'll see if we can fit in a couple of questions. Otherwise, I'll be in the wrap up room afterwards.



London Studio has a long history of working with peripherals – we pioneered camera based gaming and augmented reality with EyeToy, EyePet and Wonderbook, as well as working with microphones and companion apps on Singstar. So working on VR is a natural fit for us, as we're used to exploring new technologies and the possibilities they bring. We became of the first developers working with Morpheus, which was the codename for PlayStation VR. And true to it's name, Morpheus became the source of all my dreams and nightmares over the last two years. A dream because VR is incredibly cool to work with, and being able to inhabit the worlds you create is amazing. And a nightmare because there are a lot of challenges in being the first!

So our remit as a first party developer and being VR evangelists is to introduce and showcase VR to the world, which means both the public and other developers.
Still doesn't help us focus that much - what *is* a VR game? It could be any genre or setting. And in the end we decided to work on many different games to best explore what VR can do.

As a result, we decided the main focus on tools and tech should flexibility and speed, so we could be reactive to whatever concepts we thought were going to work best.



We developed a brand new engine called LSSDK

Fully multithreaded, and designed with the PS4 architecture in mind.

To keep things flexible, we used a tiled forward renderer. This allows us to add new shading models easily without having to worry about packing attributes into a g-buffer, and also means transparencies can be handled identically to opaque objects. We also wanted to have the option of trying out

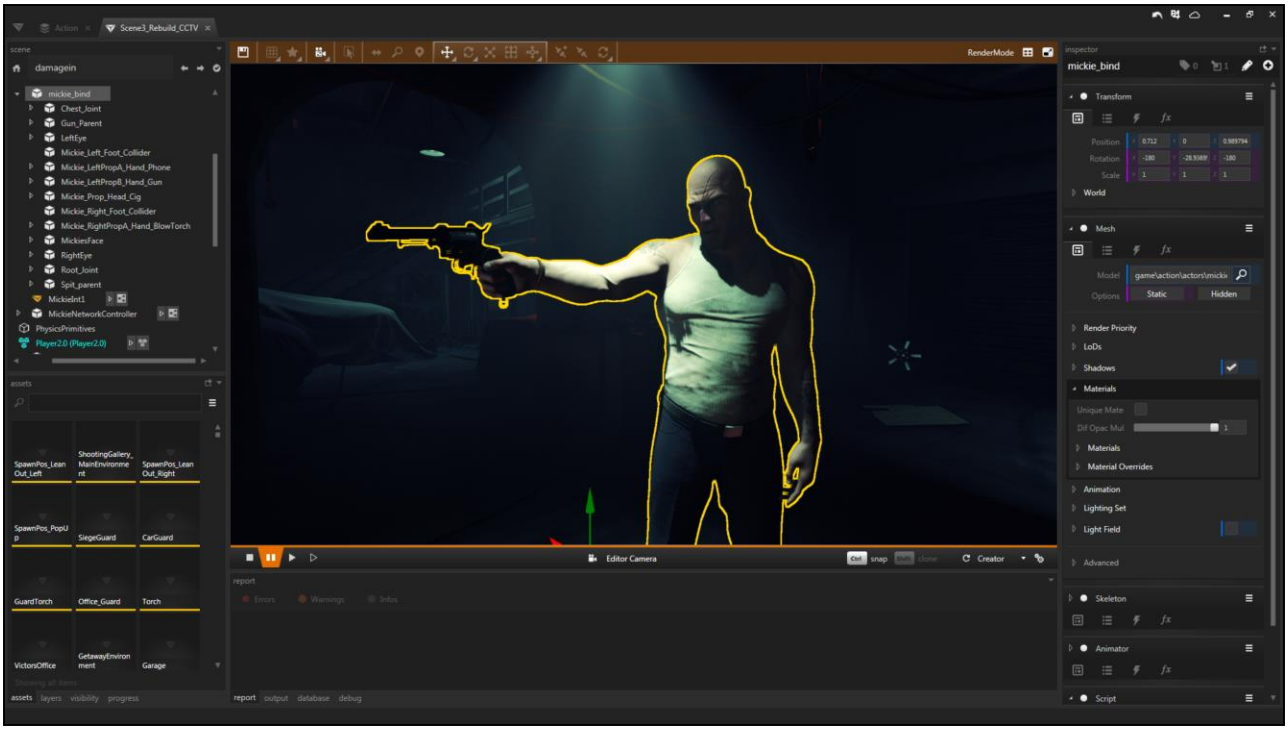
A PC version is also built alongside, which isn't optimised, but allows us to have full WYSIWYG editing on PC.

For my part, I implemented the direct lighting models for the new engine, based on the GGX lighting model, as well as shading models for pre-integrated skin, hair and eyes.

We put a large effort into making a new game editor called Creator. Creator acts as a interface to the game engine, which is running alongside in a separate process. One nice thing about this is if the game crashes for any reason, Creator carries on running, so you can just reboot the game and carry on where you left off.

You can hot reload any resource without restarting the game, so models, textures, animations and scripts can all update on the fly. This gives us a one click export for most of our assets to seeing them in the headset.

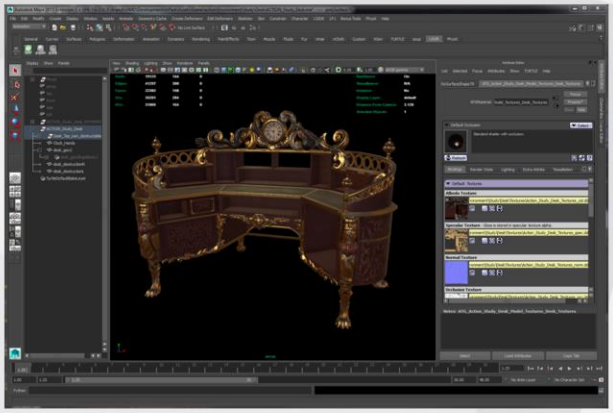
You can also have multiple tabs open, running game sessions on different platforms. This means you can have the game running in VR on PS4 whilst you are editing the game through another camera on PC.



This is Creator. Although it looks like the game is running within it, it's actually a display from another process. We also use the remote play functionality that's used for streaming to Vita to give you a display on PC when you are editing on PS4.

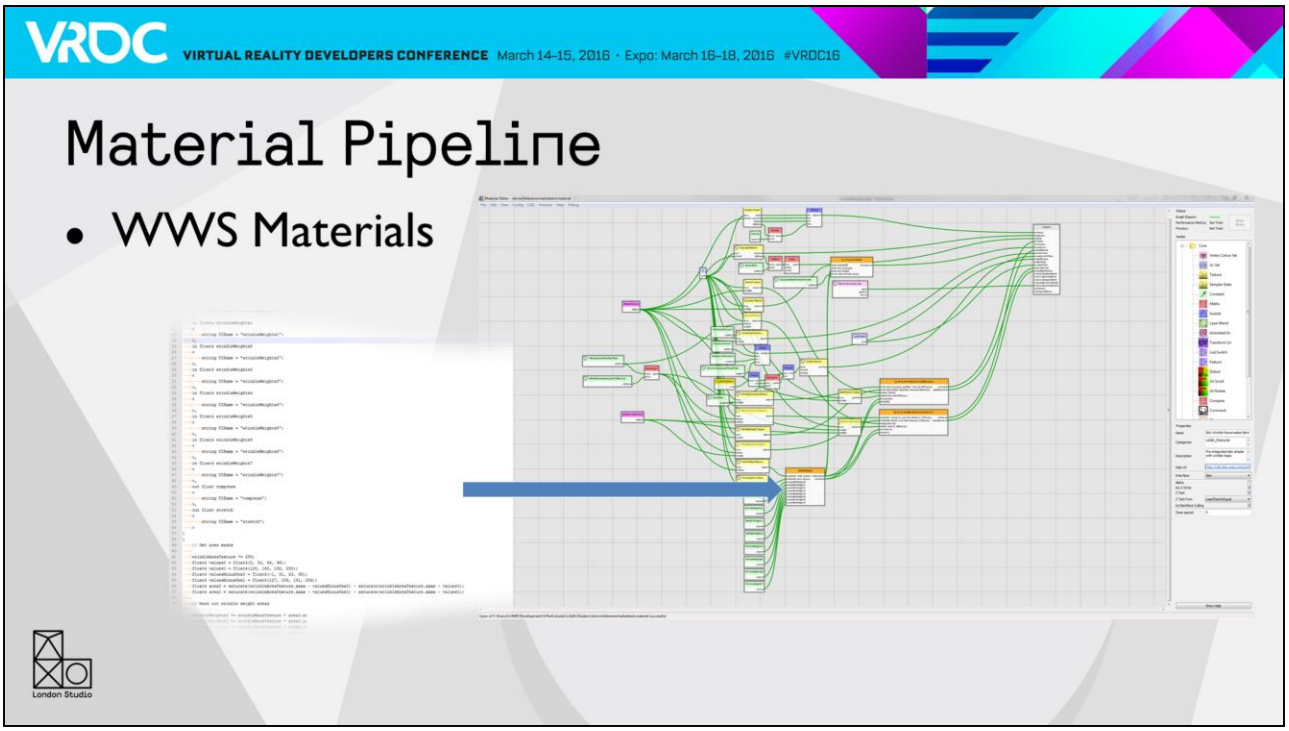
Maya Tools

- Used for content generation – no level design
- Game shaders all work in Maya
  - Material authoring happens here



London Studio

We use Maya as our primary art authoring tool, though this is largely for asset creation, so models, skeletons and animations. The placement and gameplay functionality are done through Creator. We don't run our engine in Maya, but our shader system is available here, which means we can get a reasonably good idea of what an asset will look like in game.



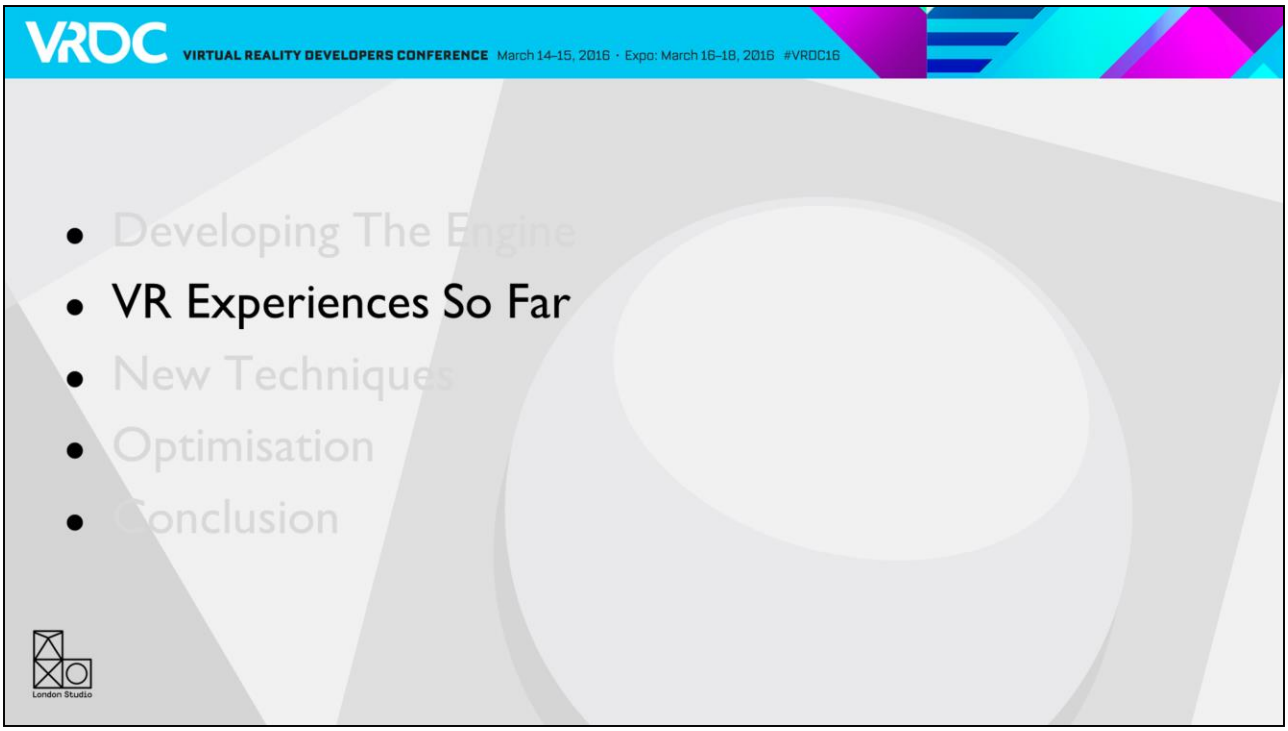
For the new engine we decided that we would have all our shaders go through Materials, which is Sony World Wide Studios shared node based material editor, in part because the backend shader code had become an order of magnitude more complex, making the barrier to entry even higher. In the new engine the lighting code is still handled with hand written code, with surface parameters like albedo, gloss, normals being plugged in to this black box.

Personally, I prefer writing code to node graphs a lot of the time for complex effects. Big node graphs can often end up being "write only", so you know what you are doing when you author it, and it all seems to make sense at the time, until you come back a week later and it just looks like spaghetti. We alleviate this to a degree by creating custom nodes with shader code in.

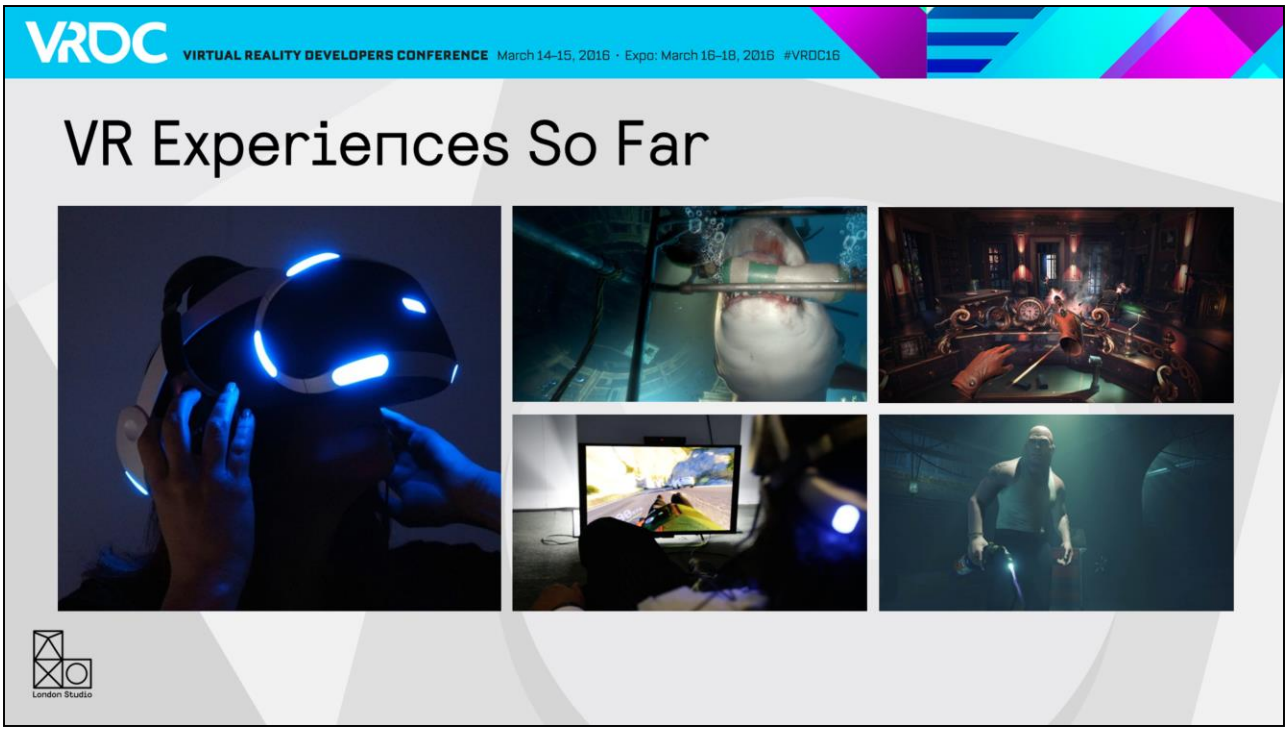
<click>

All the orange nodes in this graph are bespoke code, with multiple inputs and outputs. This really helps us use the best fit of code vs nodes where each is suited.

It takes about a minute to generate a shader for PC and PS4, but we can iterate quicker on them in Maya. As the Maya version of the shader uses a single permutation, it only takes a couple of seconds to recompile.



Now I'm going to talk about how we applied the tools and tech to our experiences.



We've been developing on VR for over 2 years now, working alongside the development of the headset, and showcasing experiences at various shows and events throughout. The engine has also grown alongside in reaction to the team's needs. So I'm going to go through all of our experiences in order to show you how the engine has developed.

First, I'm going to show you some footage of our experience which has the working title "Into The Deep".


London Studio

We experimented with many different concepts, but the one we wanted to show first was Into The Deep. We showed this at GDC 2014 with the first public showing of Morpheus. The Deep is a trip though a coral reef in a diving cage down in to the depths of the ocean, where you encounter an enormous great white shark up close.

And we already some have pretty unique rendering requirements!

Physically based scattering and absorption to give a realistic ocean feel

Caustics which diffuse as you go further into the ocean

We didn't even have a skybox, so have to have a custom underwater surface shader to refract a cubemap above the surface or reflect below the surface.

But more than that, this was really the first real test of our engine and tools

Basically everyone was getting up to speed at this point, and learning how to use physically based materials. We didn't have much time to rest though, as we swiftly moved on to our next game, VR Luge.


DRIVE SAFE
London Studio

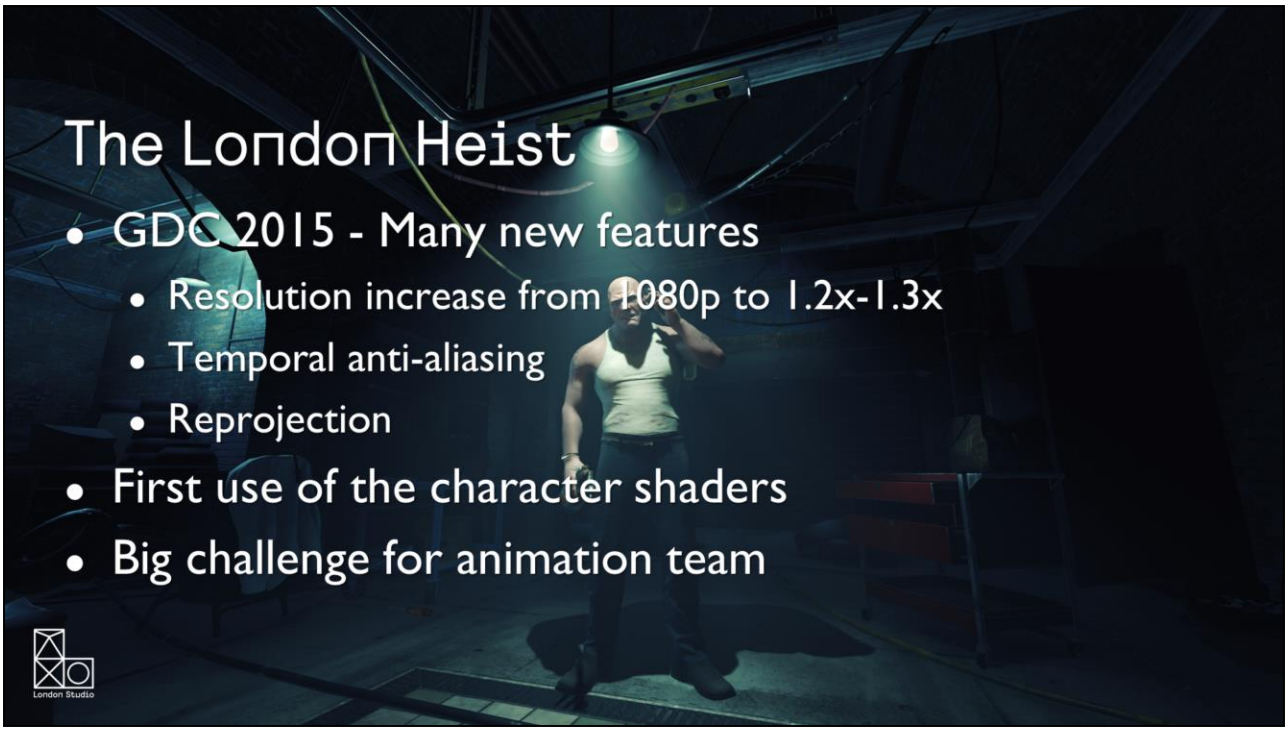
VR Luge is an adrenaline filled rush down a mountainside on a luge, dodging traffic, controlled entirely with the headset through leaning. We showed this for the first time at E3 2014, so pretty soon after the last demo. For the Luge we developed localised cubemaps, as we wanted to have reflections as you go through tunnels.

We also added an atmospheric sky simulation to give a realistic sky colour and scattering on the scenery, which can be augmented with cloud textures

We added decal support – as we are using a forward renderer, decals can be quite expensive, so we try to use stencilled alpha on decals where possible, but we do have the option of blended. Our prepass also outputs normals which allows us normal blending.

The next game was The London Heist.


London Studio



Our first showing of The London Heist was last year at GDC, and was an experience where you are being interrogated by a burly gangster called Mickie, who demands to know what happened during a botched heist. You then have a flashback to the heist itself, where you can interact with objects and have a shootout with dual Move controllers.

We've had a fair amount of time since the last game at this point, so we have many new features

The main one is a resolution increase from targeting 1080p to 1.2-1.3 times that. That doesn't sound like a lot but it's actually 1.7x the number of pixels being displayed.

This was a huge challenge, so a large portion of our time went into delivering engine optimisations, which I won't go into detail on, but big ticket items were things like asynchronous compute light tiling and deferred shadows.

We also added temporal anti-aliasing. I've seen advice previously about not using temporal AA on VR. Now, it is a trade off, in that it does tend to have a blurring effect on the image. However, we've found to create a sense of presence it's more important to remove flickering pixels that make it obvious you are looking at a game than it is to have a sharper image.

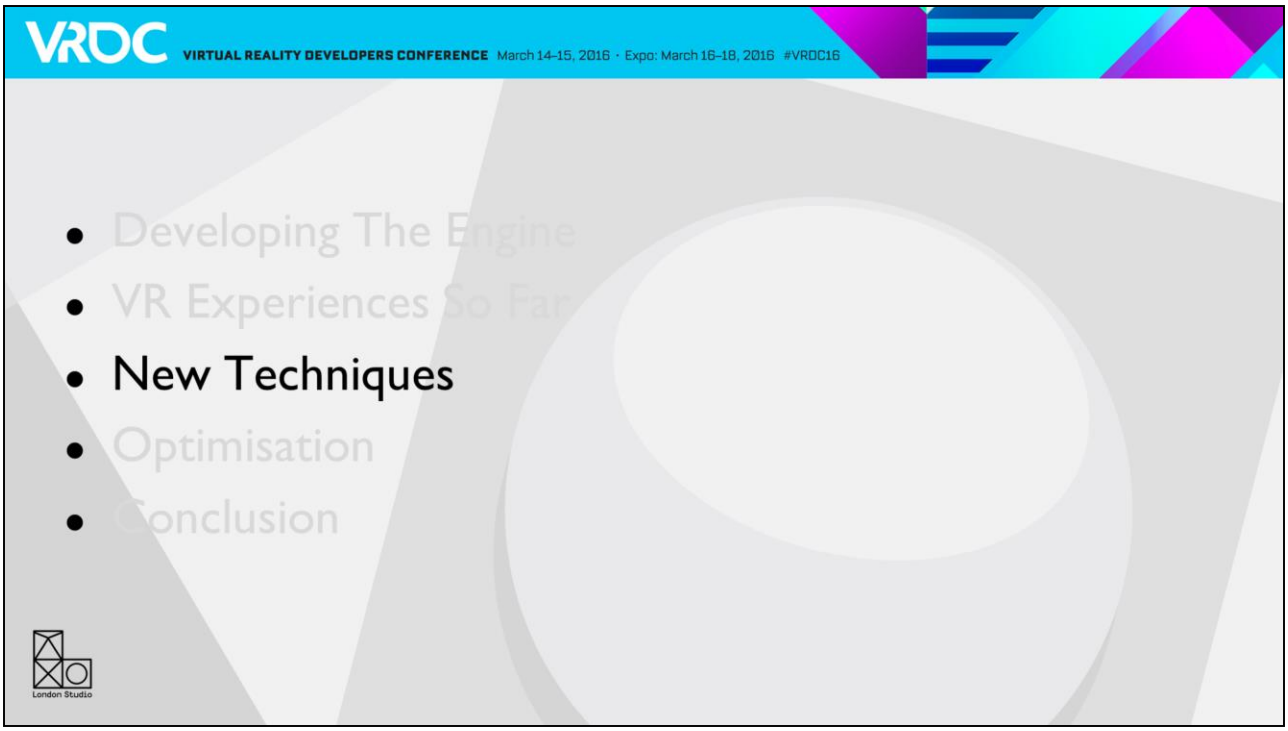
It also takes a chunk out of our frametime compared with something like FXAA, and you have to be mindful of artifacts which are much more visible in VR, but we think overall the results are worth it

Reprojection shown for the first time – at this year's GDC we showed a new version of PlayStation VR with a 120hz refresh rate. Reprojection allows us to continue to render at 60hz, but rotates the rendered image based on the current headset orientation at 120hz. As well as smoothing out the motion, this process happens at the last possible moment before being sent to the display, reducing latency. This was a big leap in presence. Removing the lag makes you think your head is moving within the environment, rather than controlling a camera.

Our first use of the character shaders – this is one area where it shows how unpredictable and varied the development experience was. We actually developed the lighting models for skin, hair and eyes very early on, but they remained unused for a year!

This was also a our first attempt at interacting with a character in VR. We use Morpheme as our animation system, which lets us make Mickie reactive to your position and act differently depending on what you do, rather than being a linear cutscene.

The London Heist went down extremely well, and we showed another segment of it at E3 called The Getaway.



That brings us up to date. So what's new?

New Techniques

- It's been a while since our last demo
- Working on other experiences
- Good core feature set
- More cutting edge features
  - Ambient lighting
  - Reducing pixel workload
  - Character shading improvements

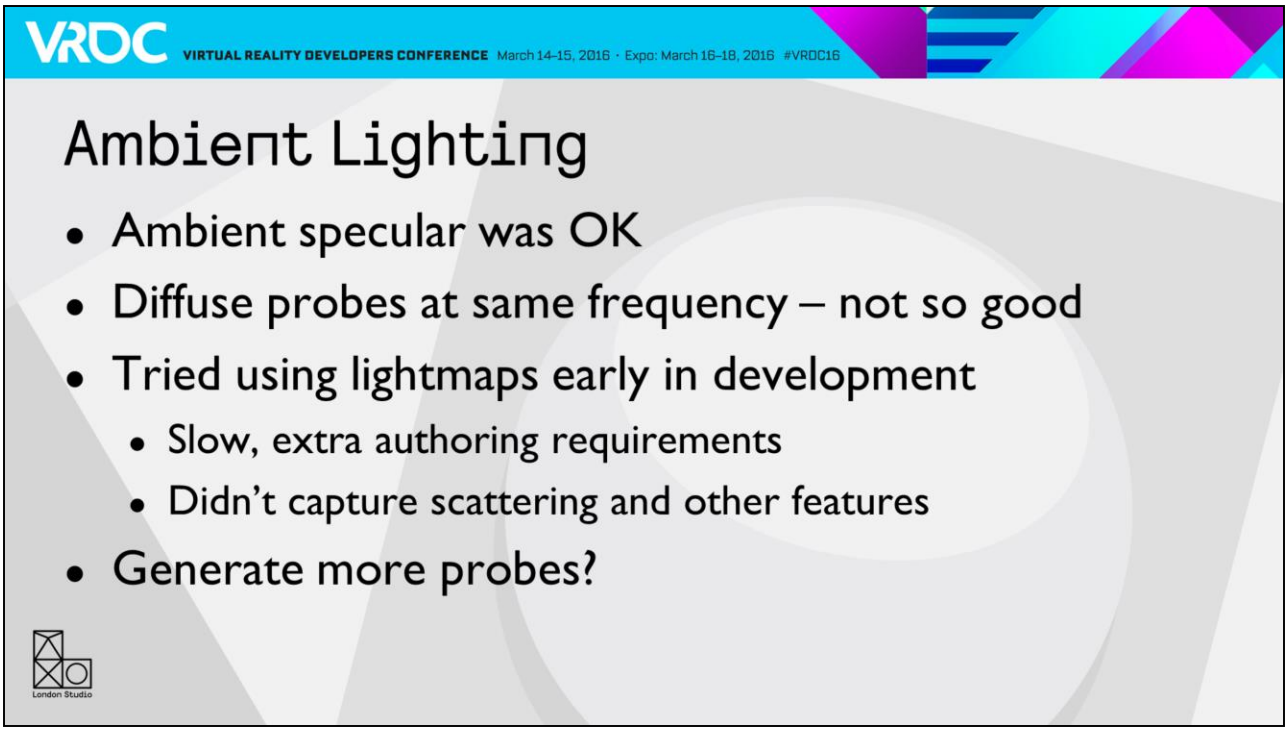
London Studio

The Getaway was shown to the public at E3 last June, so what have we been working on since?

Well, we have been working on other experiences, but I can't talk about them just yet.

After The Getaway we had a pretty good core feature set and performance, so we started to work on some more cutting edge techniques.

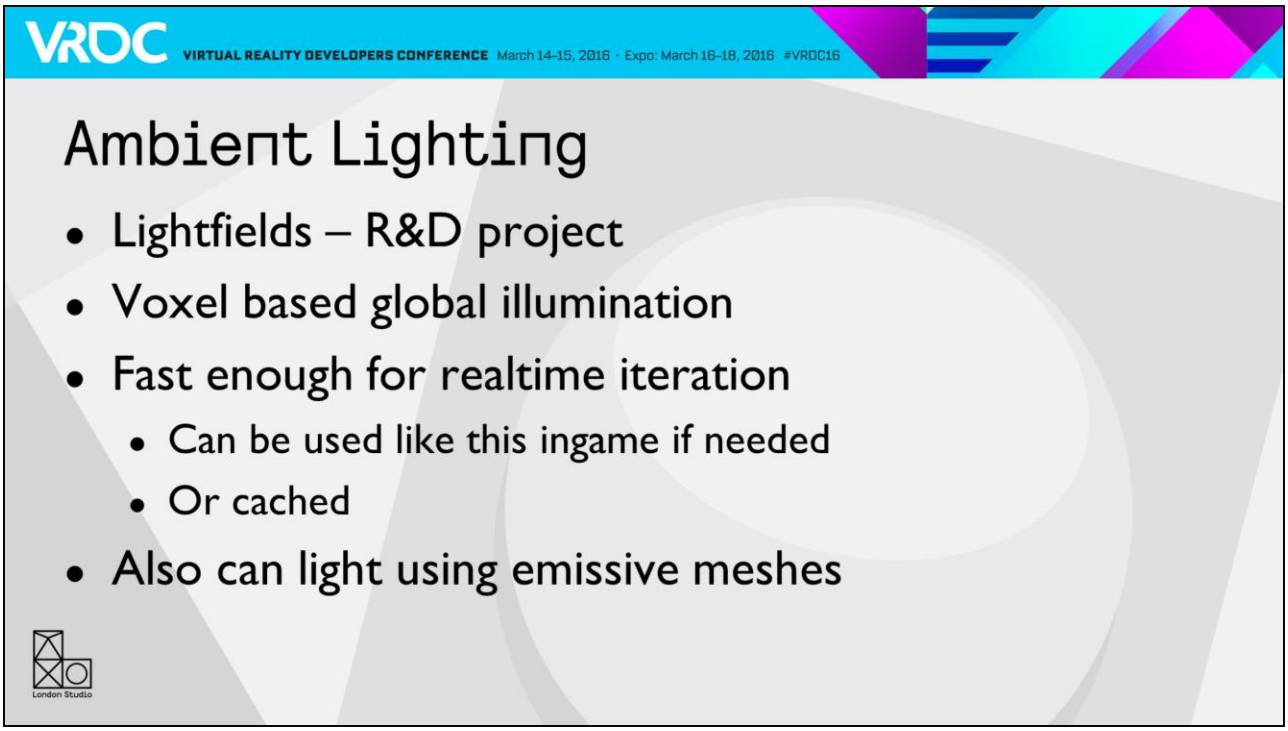
I'm going to talk about our new ambient lighting, some optimisations which allow us to hit higher resolutions, and character shading improvements.



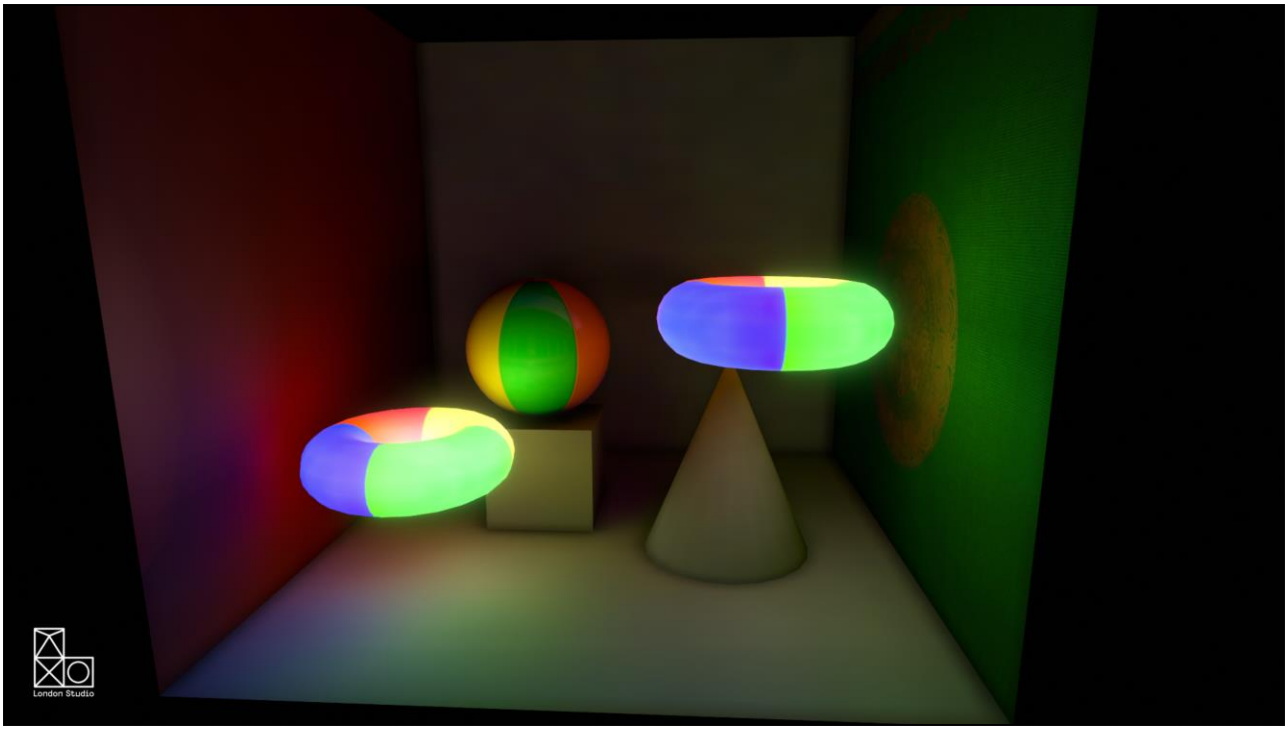
We wanted to make some improvements to our ambient lighting system. The specular was OK, and fairly industry standard using parallax corrected cubemaps, but our diffuse ambient was captured at the same frequency, so we had a bounding box covering large areas. This gave a very flat look to the environments, and didn't capture much variation in the bounce lighting.

Early on in development we tried using lightmaps, but abandoned them for a few reasons. We disliked the slow turnaround time of bakes, and also having to layout lightmap uvs. In addition to that, the first experience we made was the Deep, and our lightmap solution didn't know anything about the scattering and absorption that are key to the look. Lightmaps also only work for static geometry, so we would still need another solution for dynamic objects.

Our first thought was to simply generate more probes, by flooding the scene based on a voxel representation of the level, but this would still take a reasonable amount of time to update.



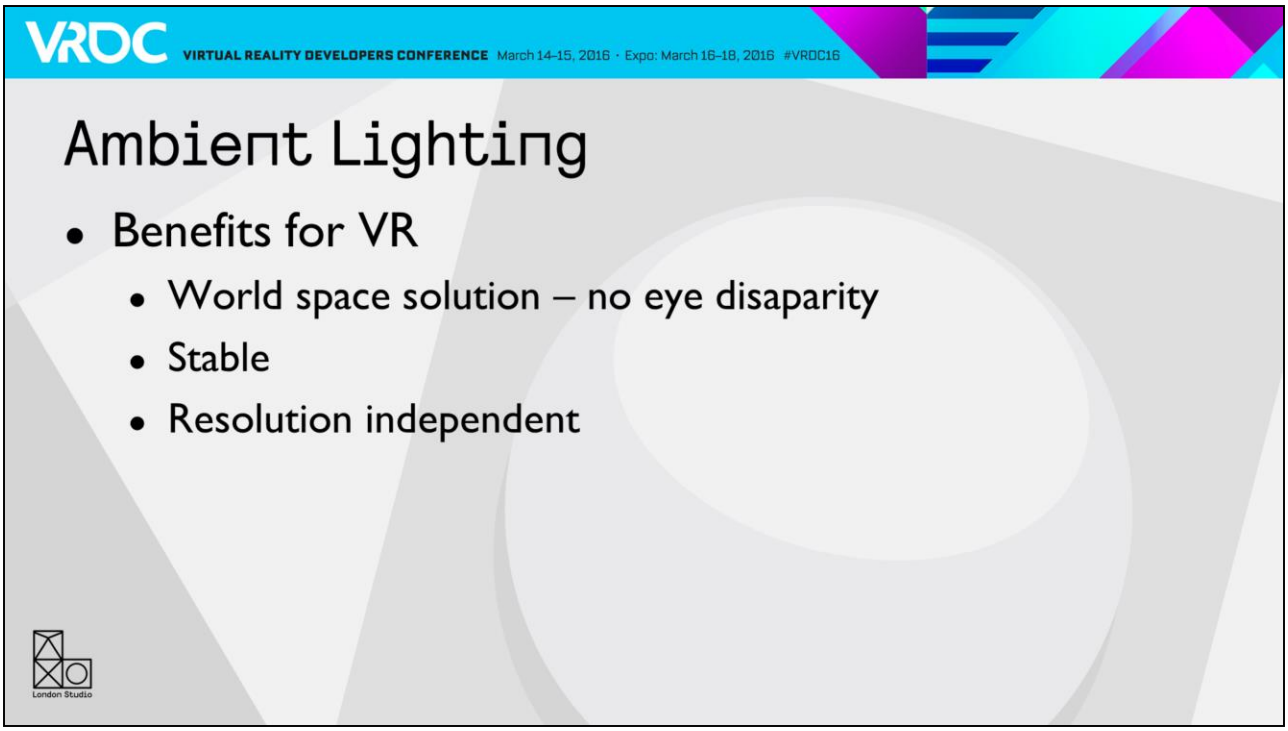
Bruno, one of our graphics programmers, had another idea though, and developed a solution called lightfields. This is a voxel based global illumination solution that's fast enough to run in realtime for VR. As well as giving us the flexibility to have dynamically updating ambient light, we also have the option of caching off the results. The benefit of this is instantaneous feedback whilst lighting. The solution also supports lighting from emissive meshes, which gives us some cheap area lighting where we need it.



Here’s a Cornell box being lit entirely by emissive meshes.

This is the Heist scene with our old lighting setup, realtime lights for the wall lights, and a couple of ambient probes for diffuse and specular.

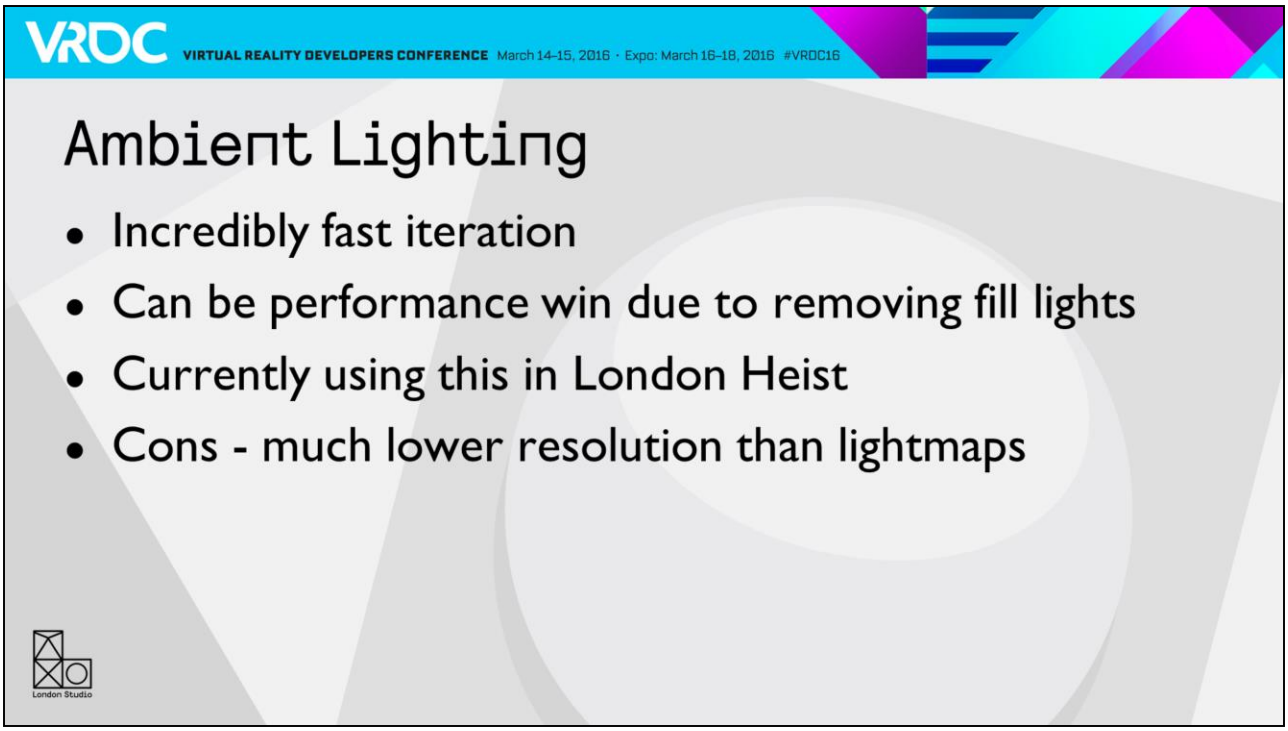
And this is the lightfields version. You can immediately see a lot more variation and depth to the ambient lighting, and more correct wide scale occlusion.



Lightfields is a good fit for VR. As well as the speed, it's a world space solution, so there's no difference between the eyes. It's stable, so there's no flicker or shimmering, and it's resolution independent, in that most of the work is done off screen.

On the left there’s a shot with lightfields, and on the right is a shot without. You probably can’t really notice much of an improvement, and that’s deliberate as this was a test to show how we can reduce our light complexity.

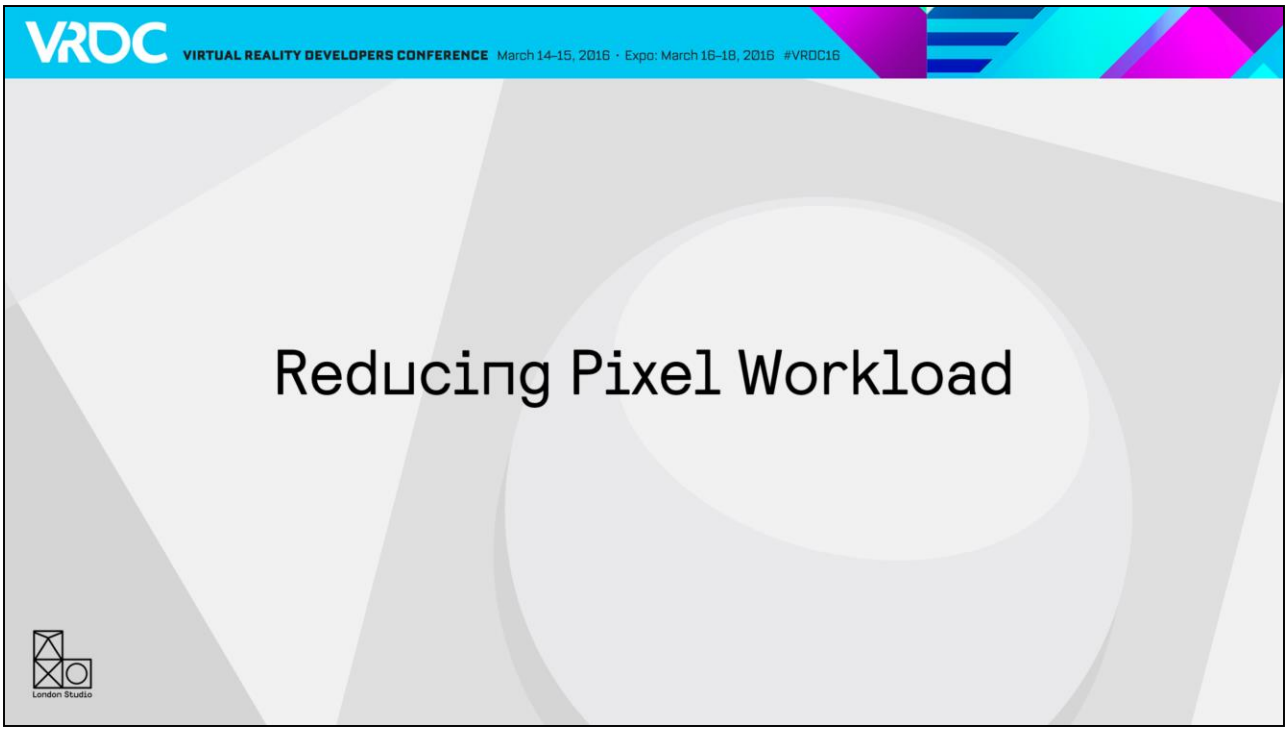
The original version used a lot of manually placed fill lights to create interest in the ambient. On the right you can see the amount of complex lighting we have in the scene to simulate the bounce light. With lightfields we could massively simplify the lighting to a few key lights whilst maintaining the look. You do lose some artistic control, but interestingly the original version done by eye was remarkably similar to the lightfields version. You do still have the option of placing fill lights into the scene as well, which can be set to only affect the lightfield, making them essentially free.



The big win with lightfields has been the speed of iteration, and also giving us much more interesting lighting results without having to manually place fill lights.

We are currently using the static version of lightfields in the London Heist, we need to work on a larger scale version for VR Luge and Into the Deep

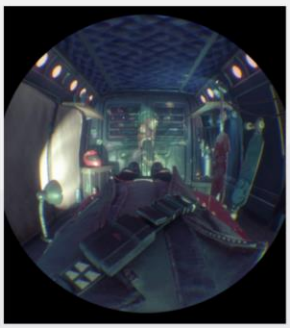
The main disadvantage compared to lightmaps is that it's much lower resolution. This means you can only really use it for ambient or fill rather than baking spotlights onto the environment, for instance.



Next I'm going to talk about how we managed the resolution requirements of VR

Reducing Pixel Workload

- We noticed not all the rendered image was used
- Created an "Inverse Distort" mask
- ~20% saving



London Studio

The left image shows what we render before distortion for the lenses. The red area isn't needed, so we mask it out. This saves us about 20% frame time for no visual loss. I think this is covered in all the VR SDKs at this point.

Reducing Pixel Workload

- Lens distortion means we render at high res in centre
  - Ideally 1.4x to get pixel matching

The lens distortion applied for viewing in the headset means we need to render at a high resolution in centre of the image

Ideally this should be 1.4x1080p to get 1:1 pixel matching

I'm going to show you a little video to explain

London Studio

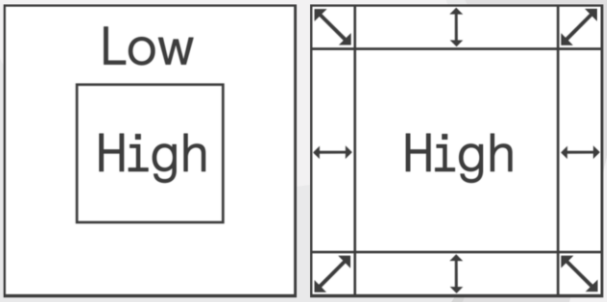
Here you can see the final image we need for the lens in the headset.

We have to render a larger image, then essentially shrink wrap it onto the final image

Notice how the centre remains at the same size.

Reducing Pixel Workload

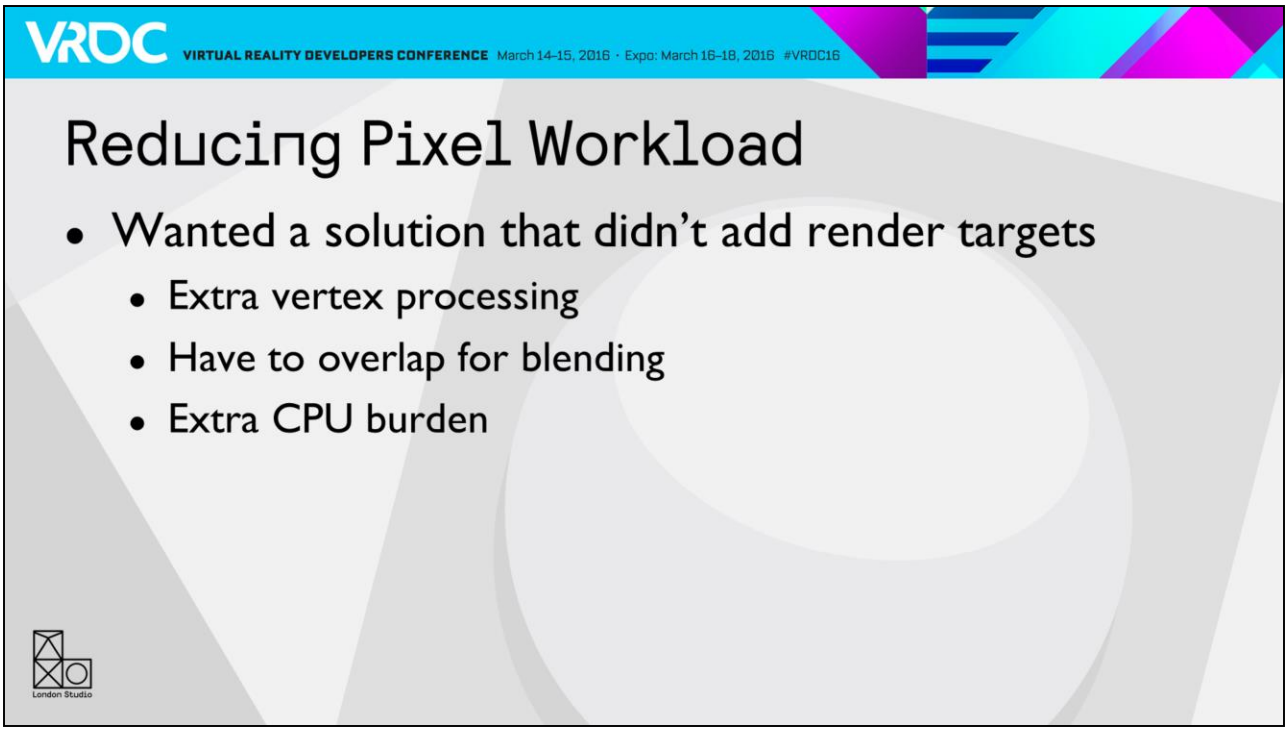
- Wasted work on outside of image
- Other solutions proposed
  - Render high res insert
  - Targets stitched together



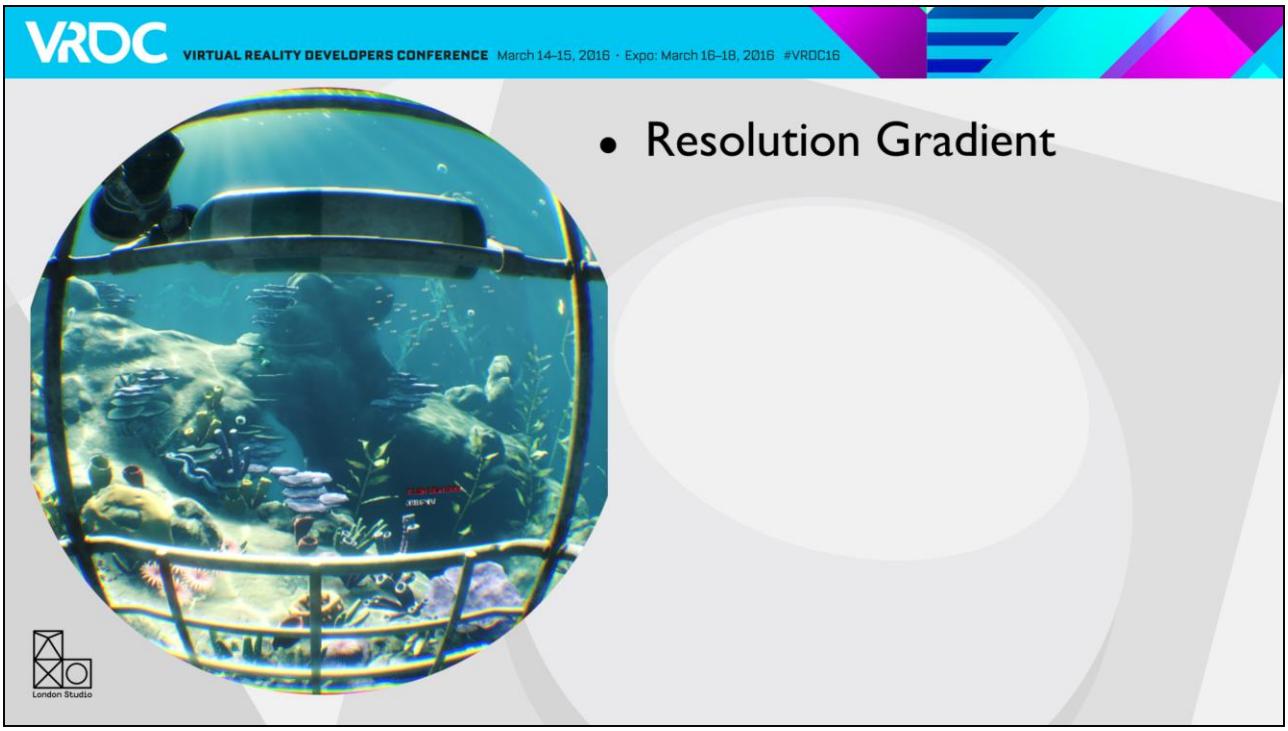
London Studio

But as you can see, there's a lot of resolution used on the outside of the image. Not only is it oversampled compared to the middle, but generally speaking you don't care about it as much.

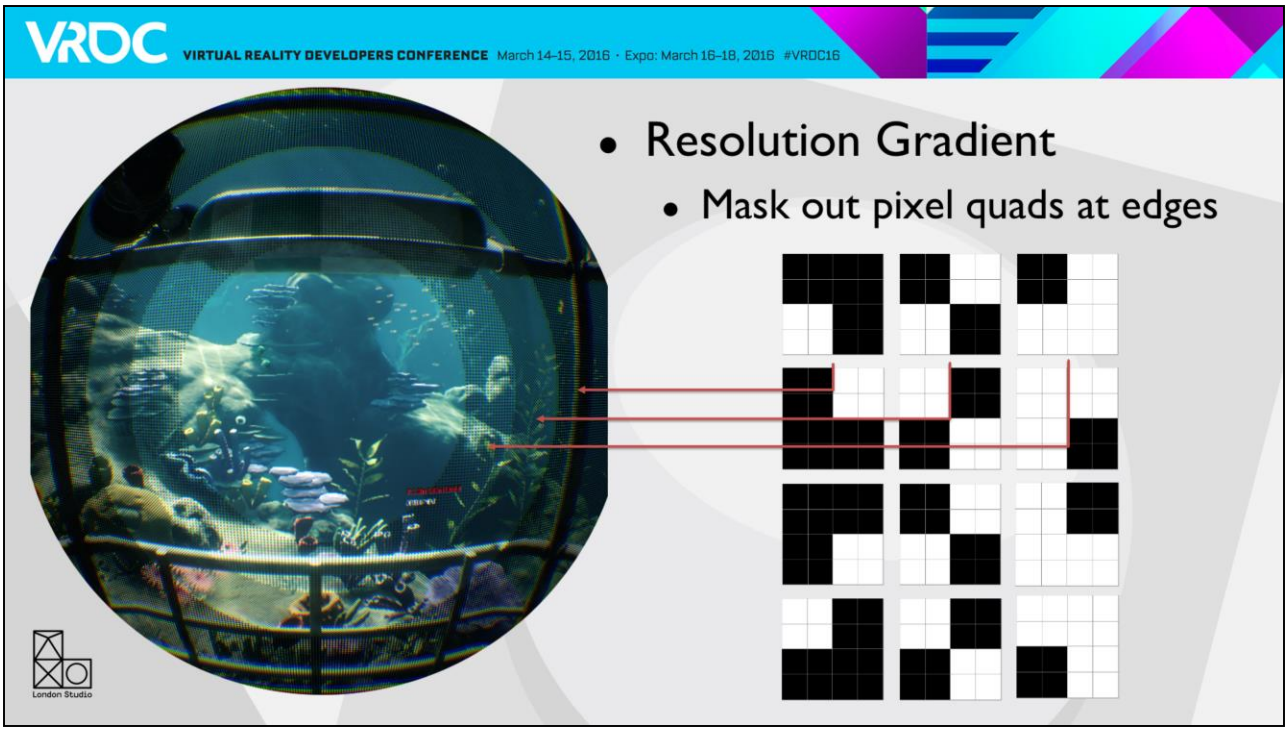
Other solutions that have been proposed in the past include rendering a high resolution insert in the centre, or Nvidia's solution which is to have 9 render targets stitched together.



But we wanted a solution that didn't increase the number of targets doing so has a number of disadvantages like extra vertex processing, having to render overlaps to have a smooth transition between the different resolutions, and the extra CPU burden of rendering the objects multiple times.



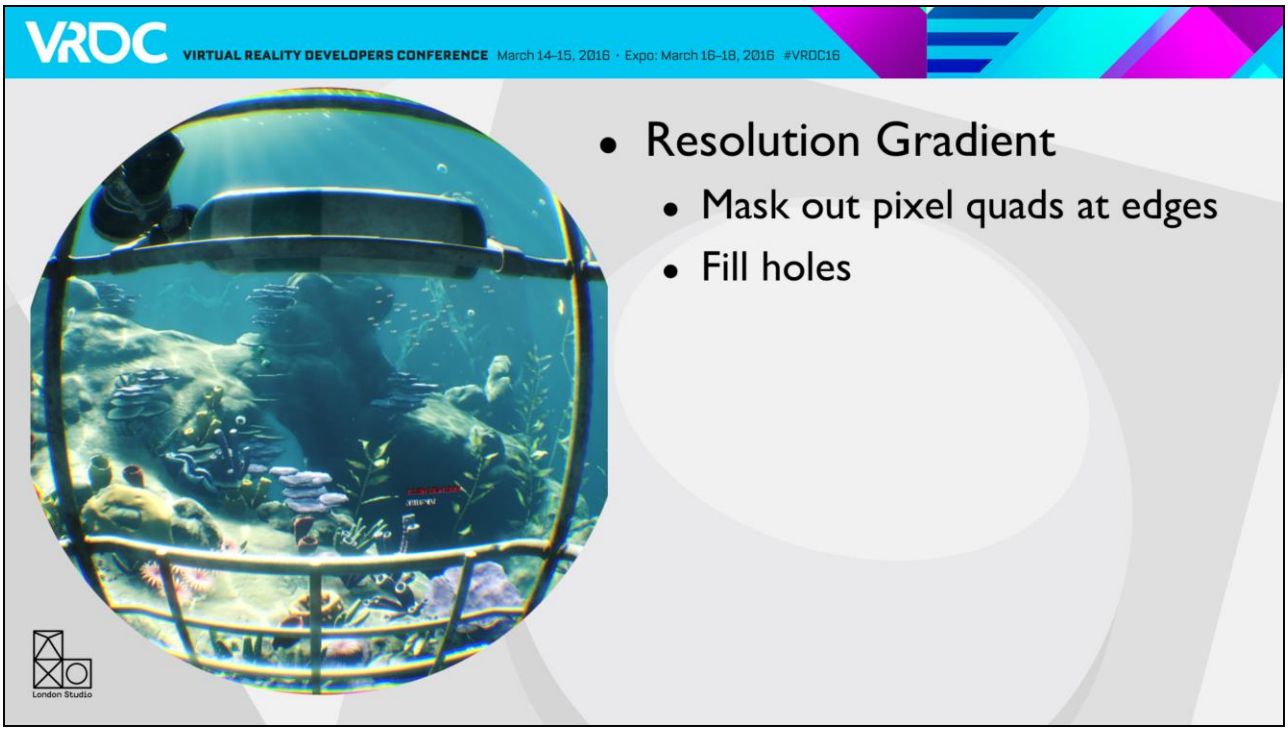
We came up with a technique which we call a resolution gradient. Here's a shot of Into The Deep



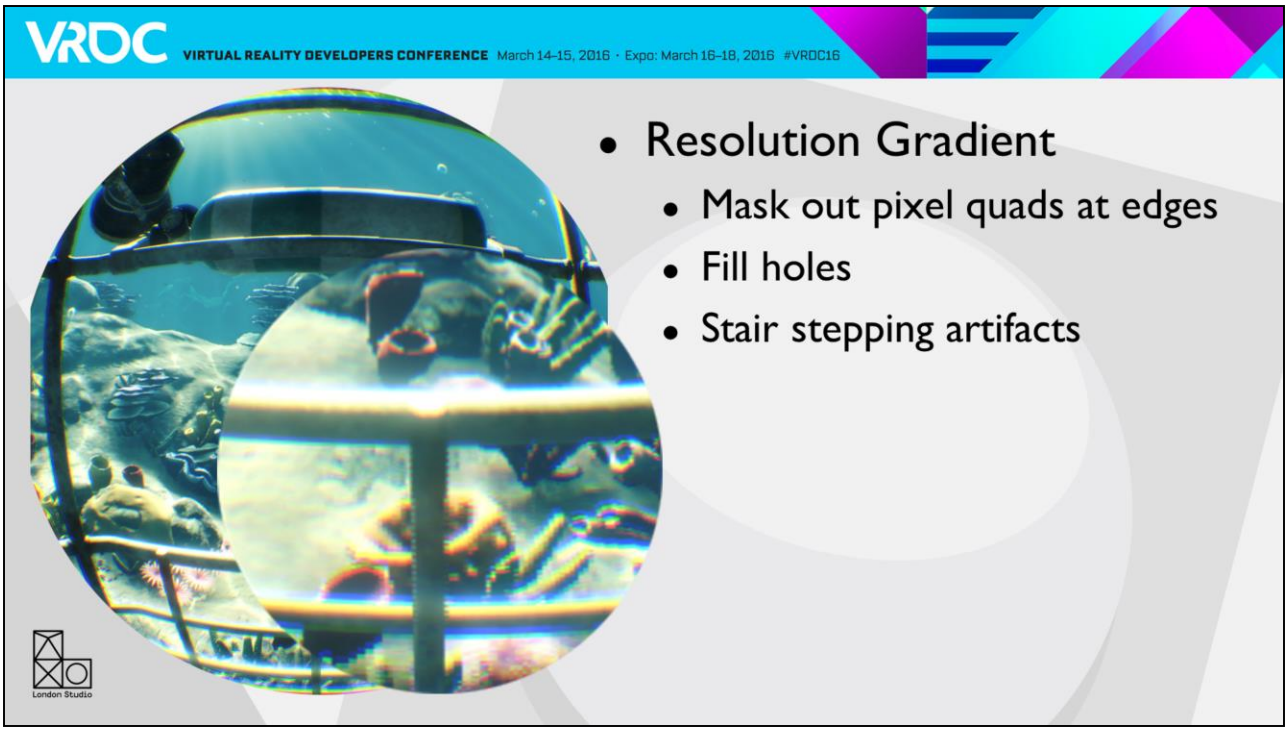
We mask out a dither pattern in increasing amounts, so 25%, 50% and then 75% of the pixels aren't rendered at the edges. We currently mask out 2x2 pixel quads, as masking out single pixels isn't GPU friendly, but I'll come on to this later.

<click>

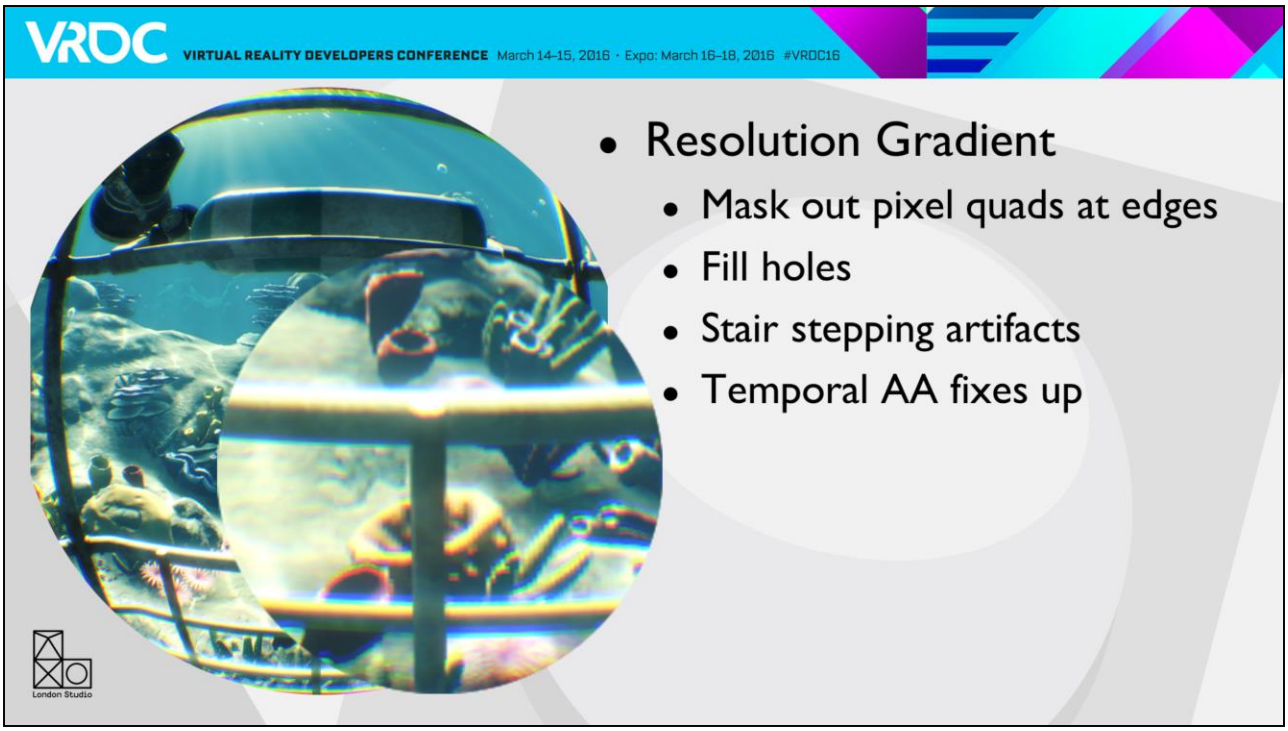
We change these dither patterns every frame.



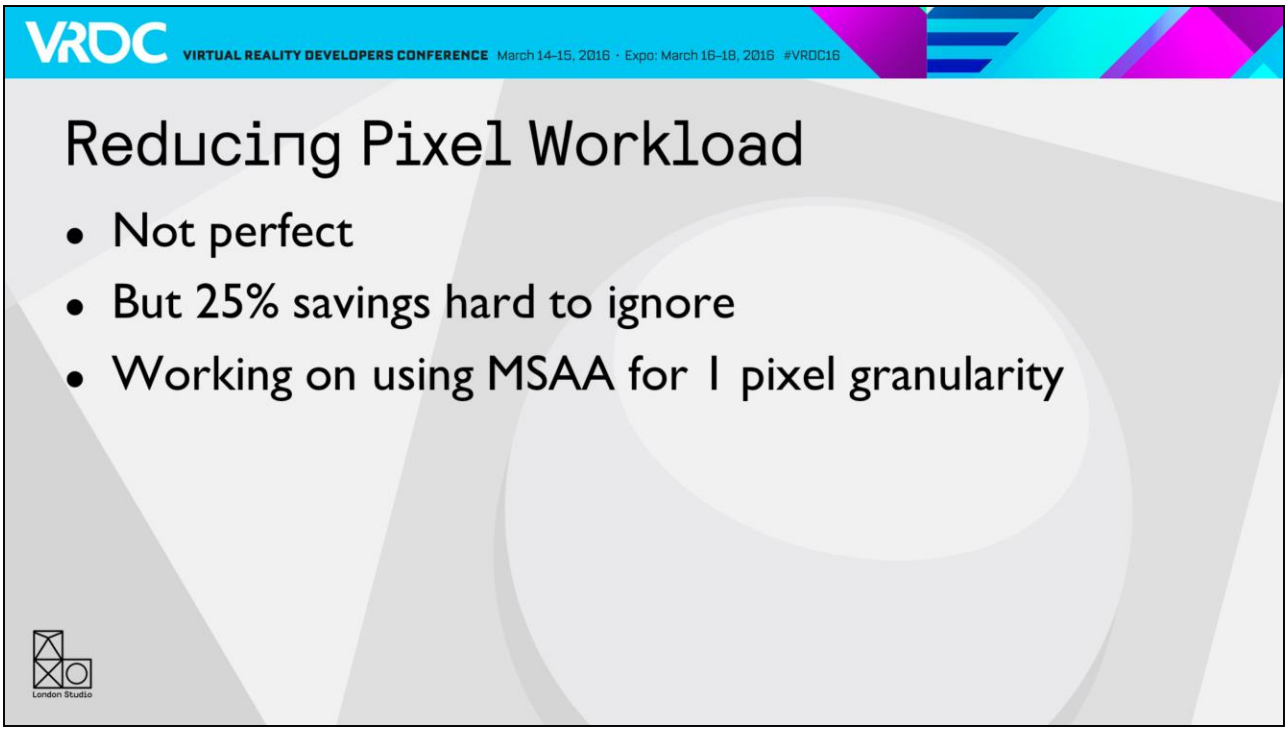
We then fill those holes by taking neighbouring values through a dilation filter



This gives us stair stepping artifacts

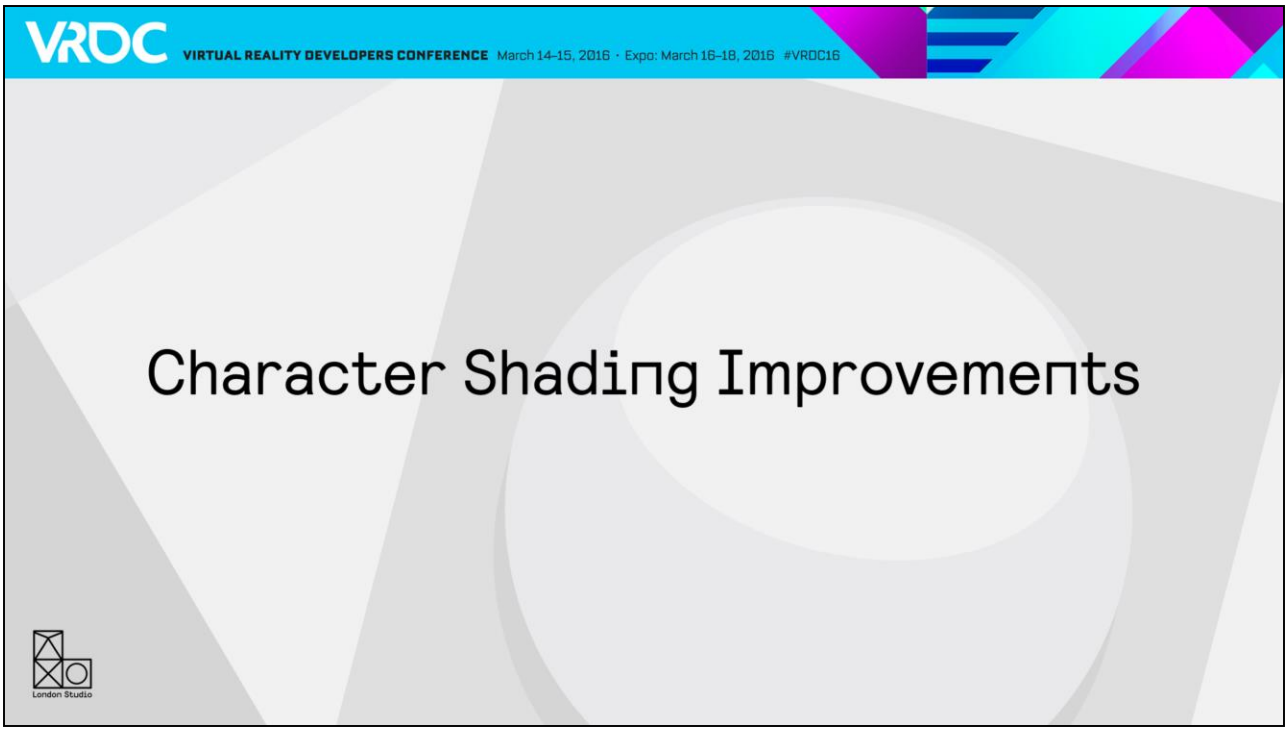


But our temporal anti-aliasing can clean most of this up as we are constantly feeding in new information due to the varying dither pattern.



This technique isn't perfect, you do get some quality loss, but you are generally quite forgiving if it isn't in the centre of your field of view. And we save about 25% GPU time, so it's a good tradeoff.

We are working on a technique using hardware antialiasing to only mask out single pixels rather than 2x2 quads. This should improve the quality to the point where we hope you won't be able to notice it.



The last batch of new techniques I want to talk about are for character rendering.

Character Shading Improvements

- Character performance important for London Heist
- Correct shading gives many cues for character expression

Characters are very important for the London Heist. They are very close, and have your full attention. It's important we get the shading right so you can read the facial expressions clearly.

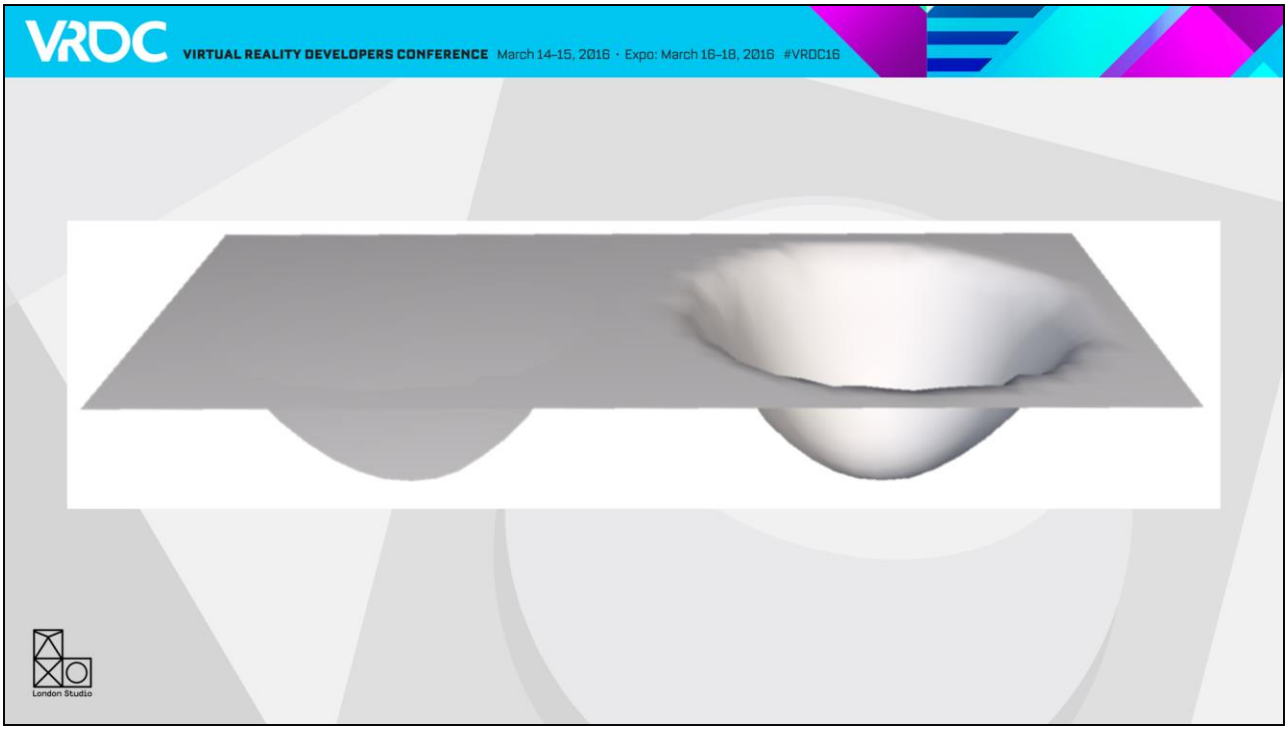
Character Shading Improvements

- Vertex renormalisation
  - In geometry shader
  - First did this on EyePet
  - Vital to getting correct facial expressions if using joints
  - Maya does this automatically, so we need to match

London Studio

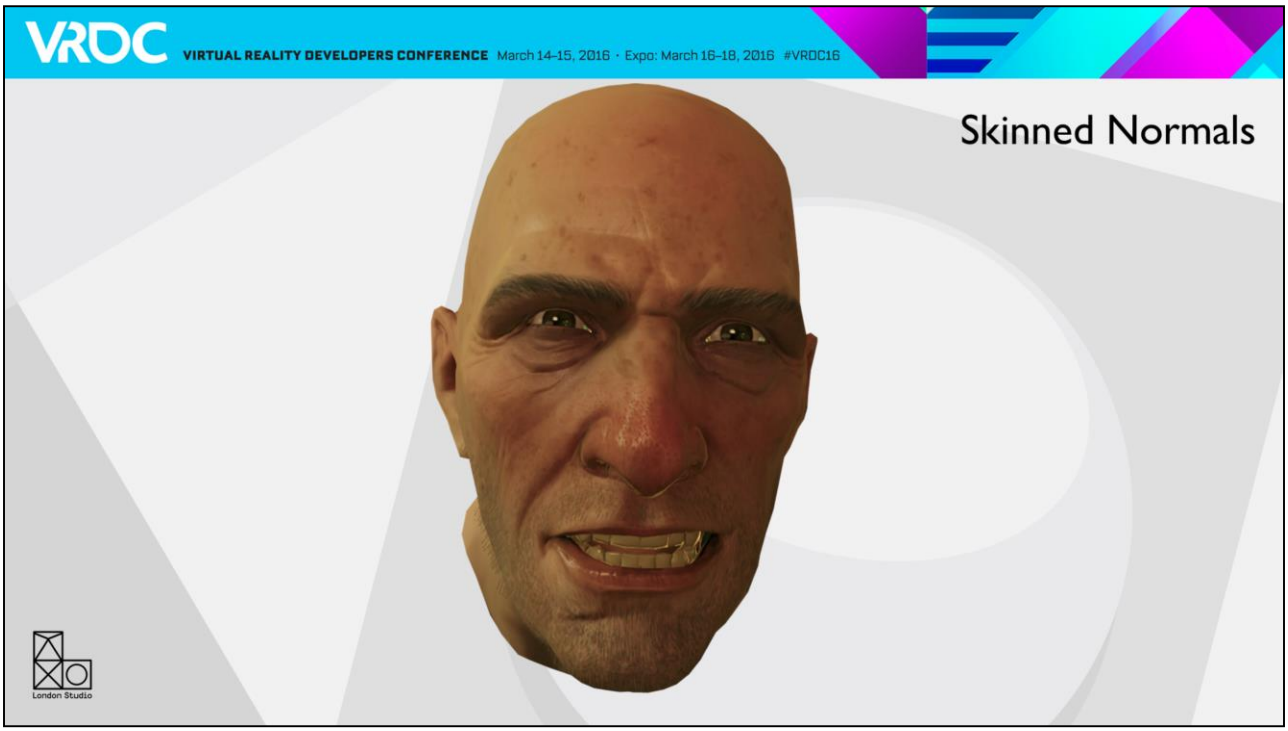
The first thing we added was vertex renormalisation. This is a process we do in a geometry shader, and it calculates new face normals based on the deformed mesh. We first identified this as an issue on EyePet., and had an SPU process to recalculate the normals dynamically.

Maya always processes meshes in this way, so animators need this to work the same in game if they want their facial expressions to match.

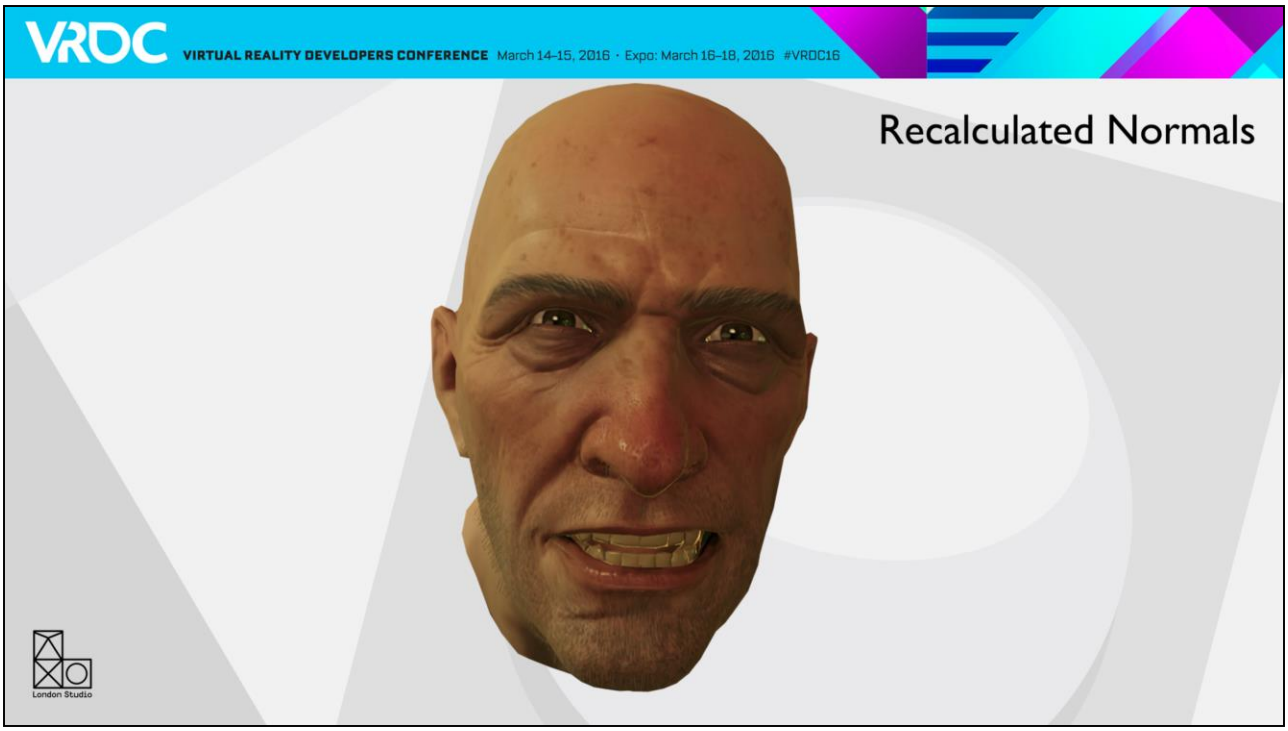


If you rotate joints, the normals work acceptably well – they just rotate with the joint based on the weighting. But facial animation largely uses translations rather than rotations, and that doesn't update the normal at all.

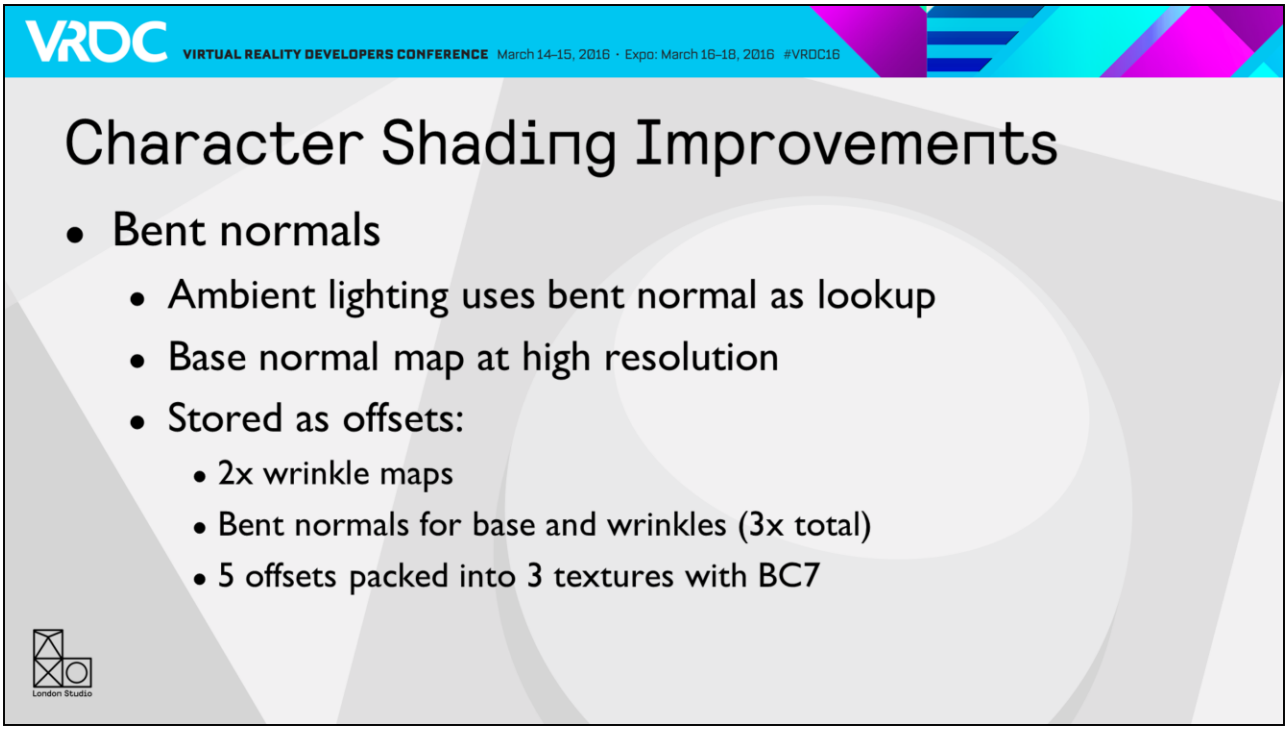
The left section of this image shows what happens if you translate a joint on a skinned mesh using a standard game pipeline. The normals all continue pointing upwards, so the lighting is flat. The right hand section is what you get when you generate new face normals – everything looks like you would expect.



Here’s a standard game skinning pipeline



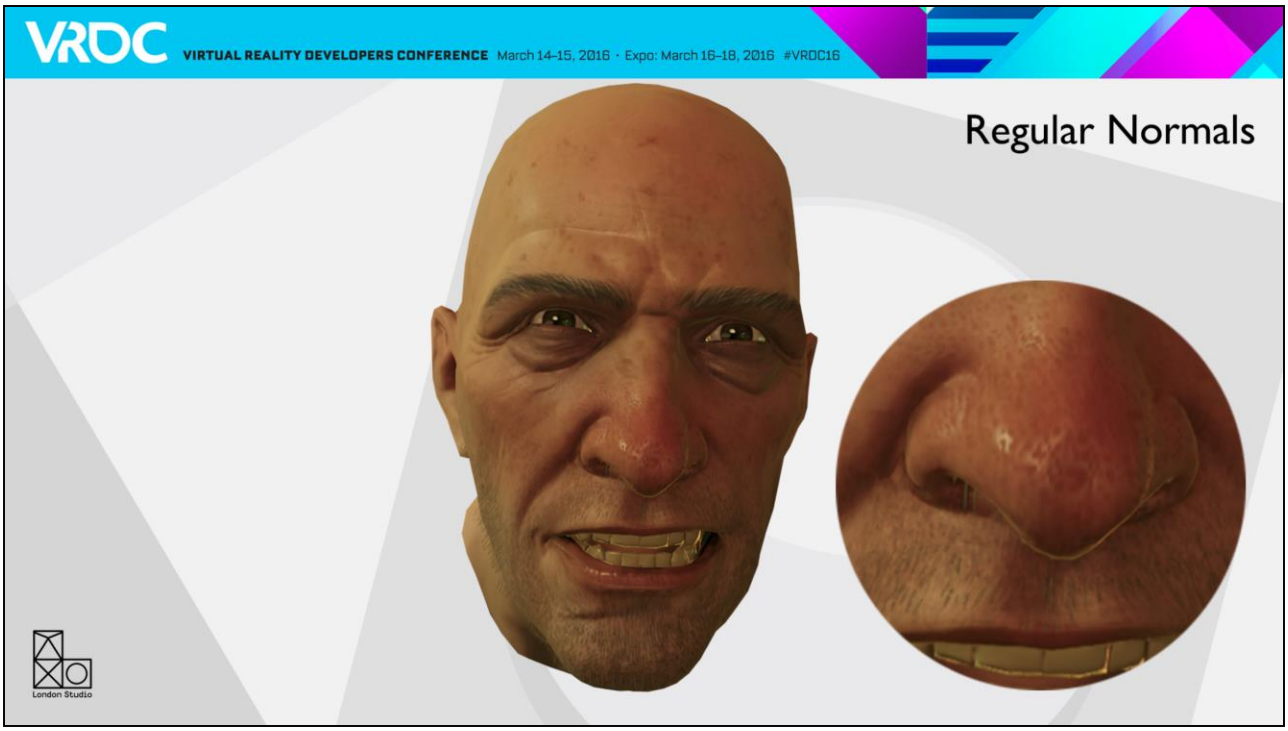
And here's the same mesh with recalculated normals. It's difficult to say that one looks more correct than the other from a still, but an important thing is that the normals are constantly changing on the recalculated version, making the face feel more alive.



The next thing we added was support on the skin shader for bent normals

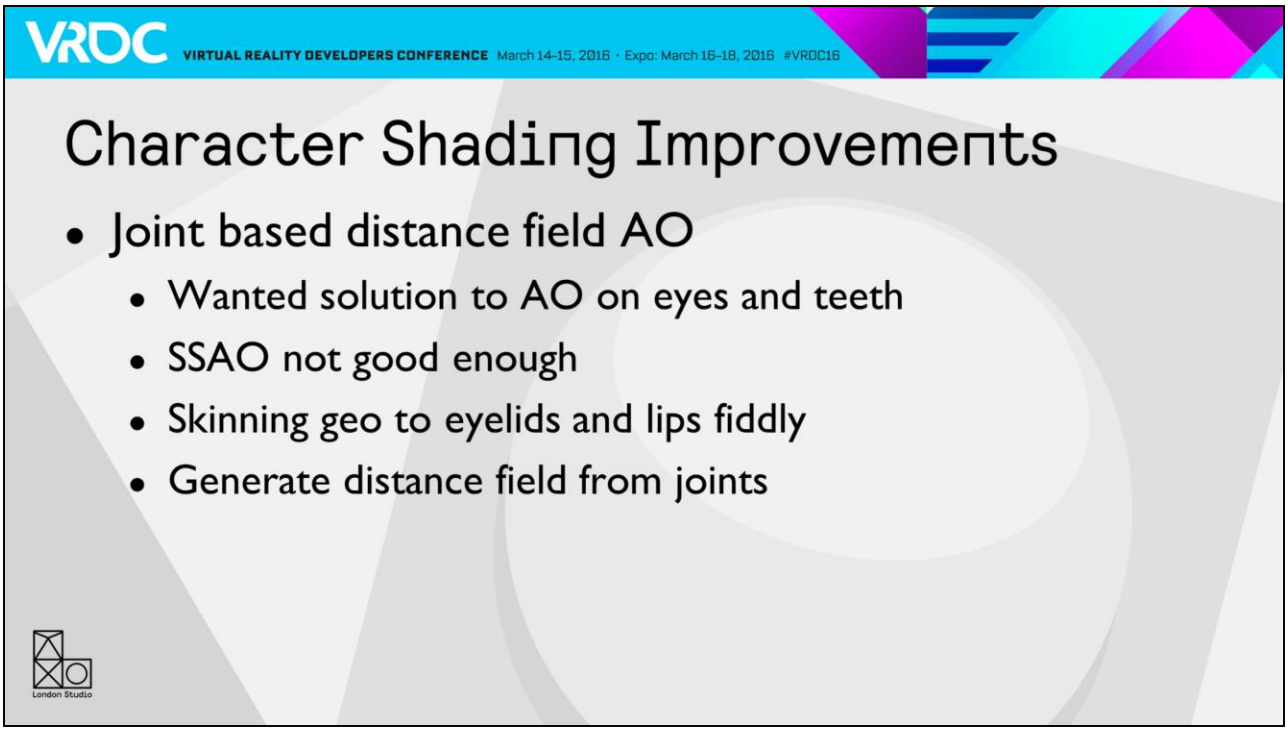
Bent normals are a normal map that store the average unoccluded direction of a point on the mesh. When you apply them to your ambient lighting lookups, you can reduce the amount of light leaking coming through areas that should be occluded.

With storing bent normals and wrinkle maps, we end up having quite a lot of textures. To reduce the memory and bandwidth requirements, we store the wrinkle maps and bent normals as offsets from the high resolution base normal map. We only need the offset maps to reflect coarse lighting changes on the face, with high frequency detail like pores and smaller wrinkles remaining on the base map. We pack the 5 offsets into 3 textures using BC7.



This is standard normal mapping

And bent normals. You can hopefully see the reduced light leaking, particularly on the bottom of the nose.



Lastly, I want to talk about our ambient occlusion on characters

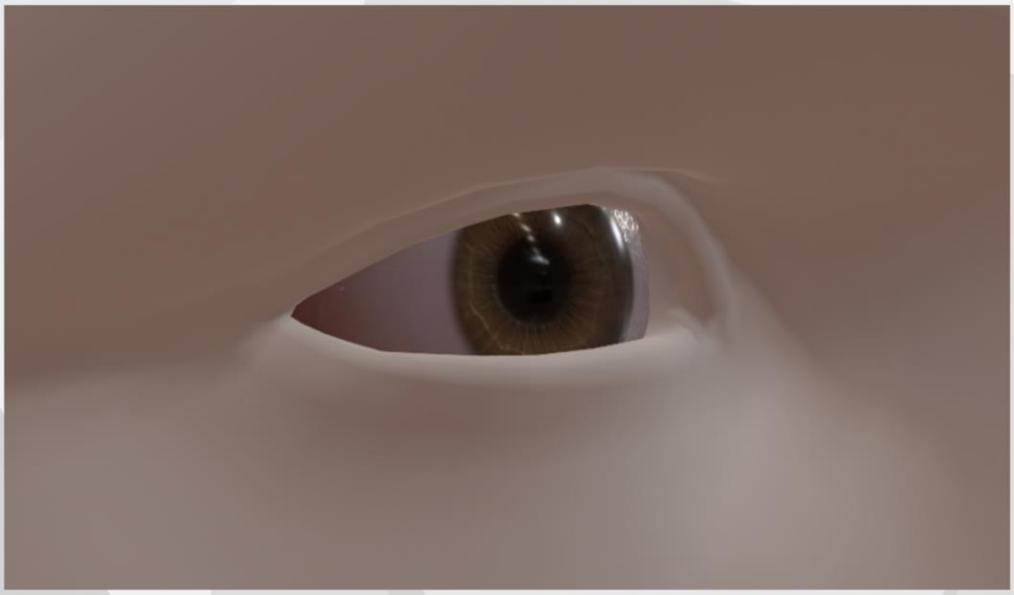
We have a pretty good refractive eye shader, but the most common complaint about it is that it didn't seem bedded in enough. We had similar complaints about teeth being too bright.

This really comes down to having a good AO solution, and in this case screen space ambient occlusion didn't really give us the resolution or control we need.

Previous solutions to this problem have included skinning geometry with an AO gradient on to the eyelids and lips. This is quite a fiddly solution though, as it's difficult to manage interpenetration from the meshes with all poses.

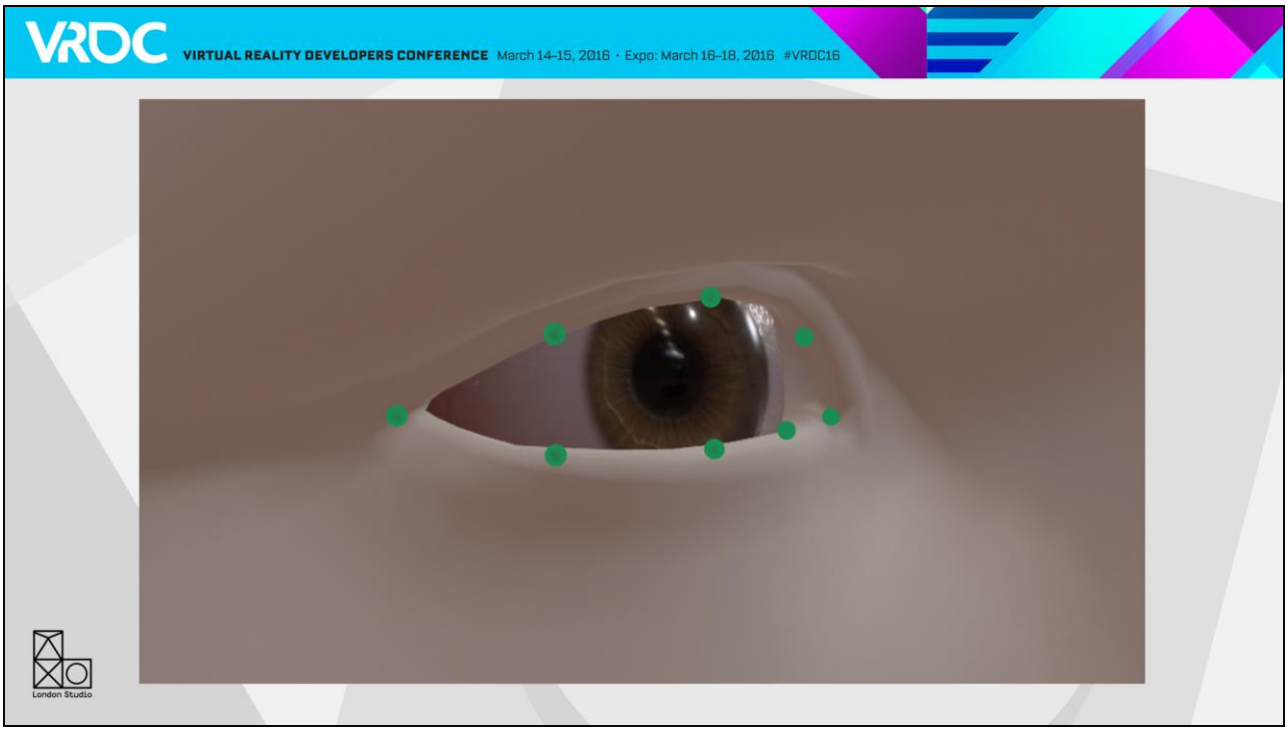
I had the idea to generate a distance field representing the eyelids and mouth. Looking at the skeleton, we already had joints fairly well placed in these areas, so I create a ring of line segments using these positions and calculate the distance to the nearest one.

We can then use this to generate AO and cavity values

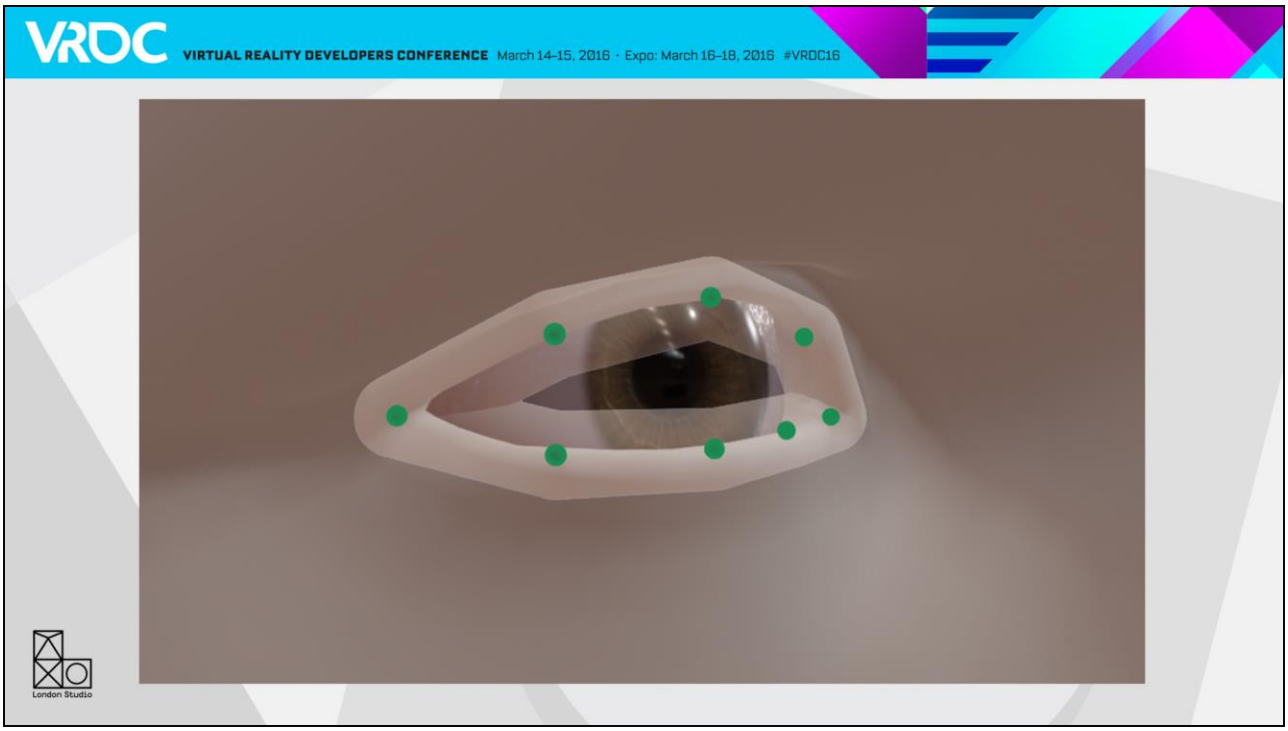


London Studio

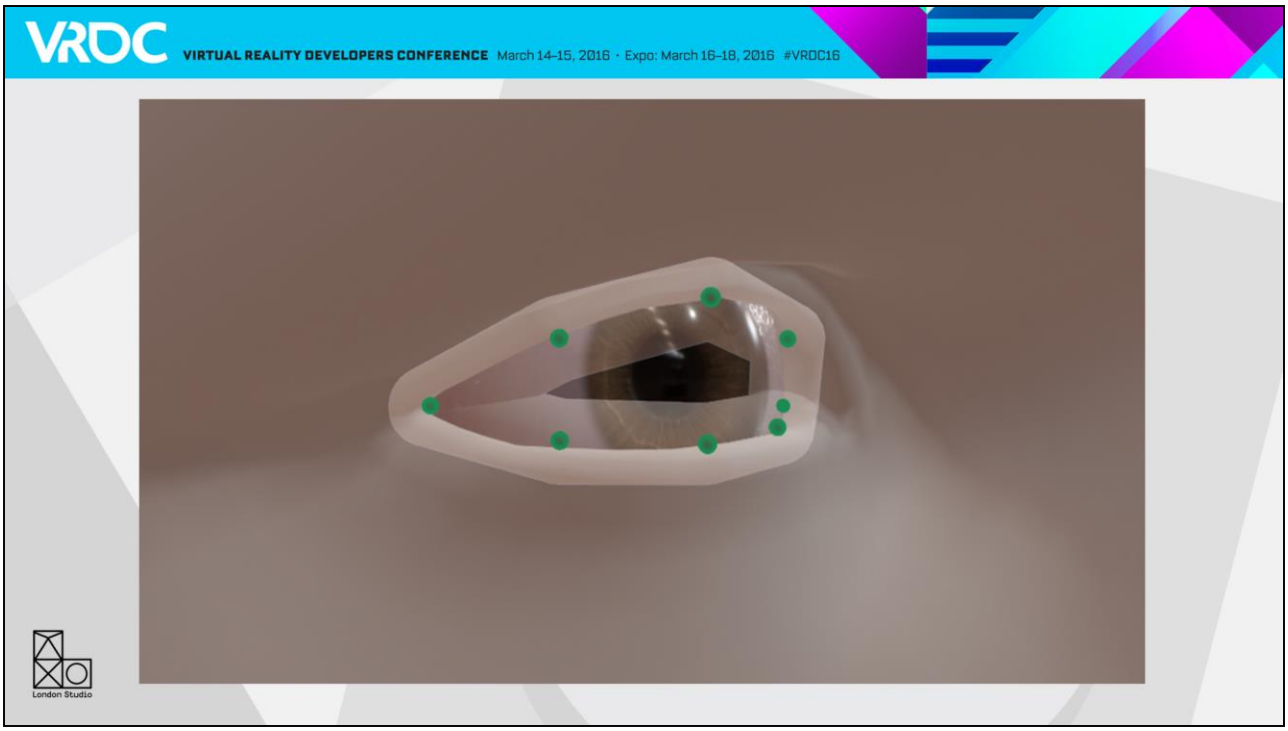
Here’s our eye shader.



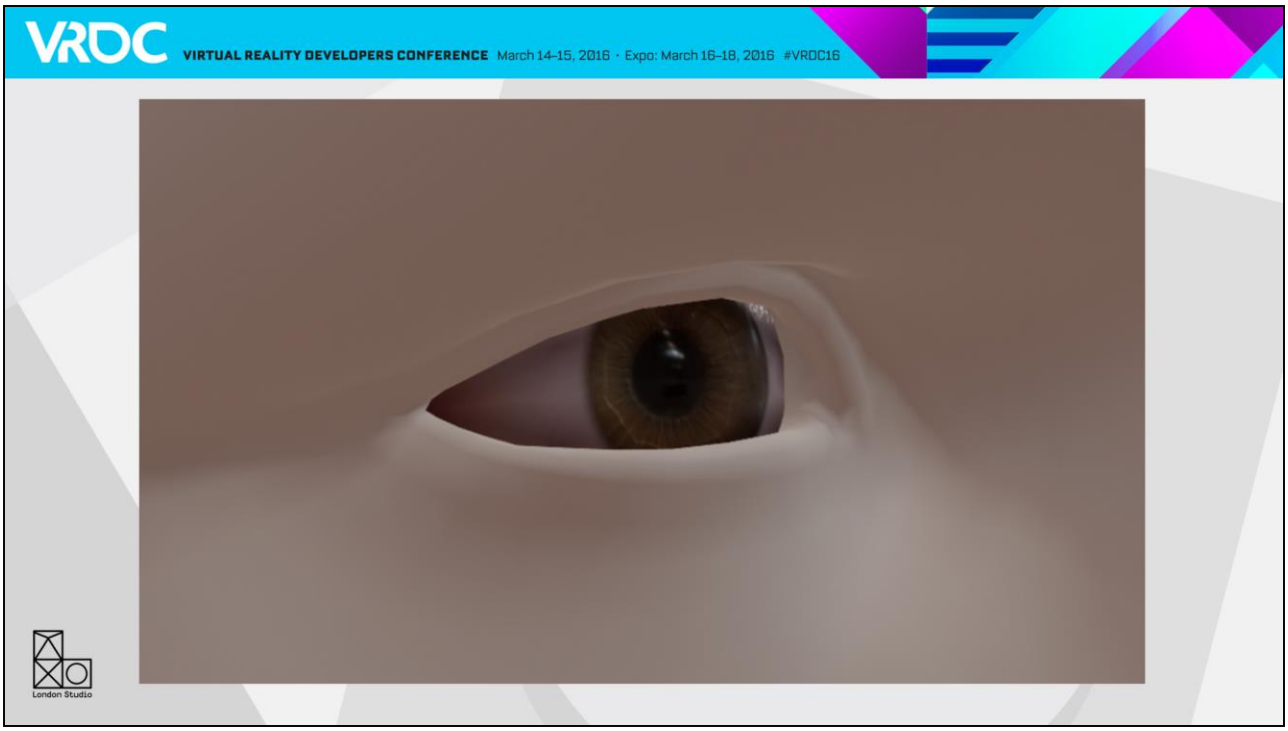
Here are the eyelid joints



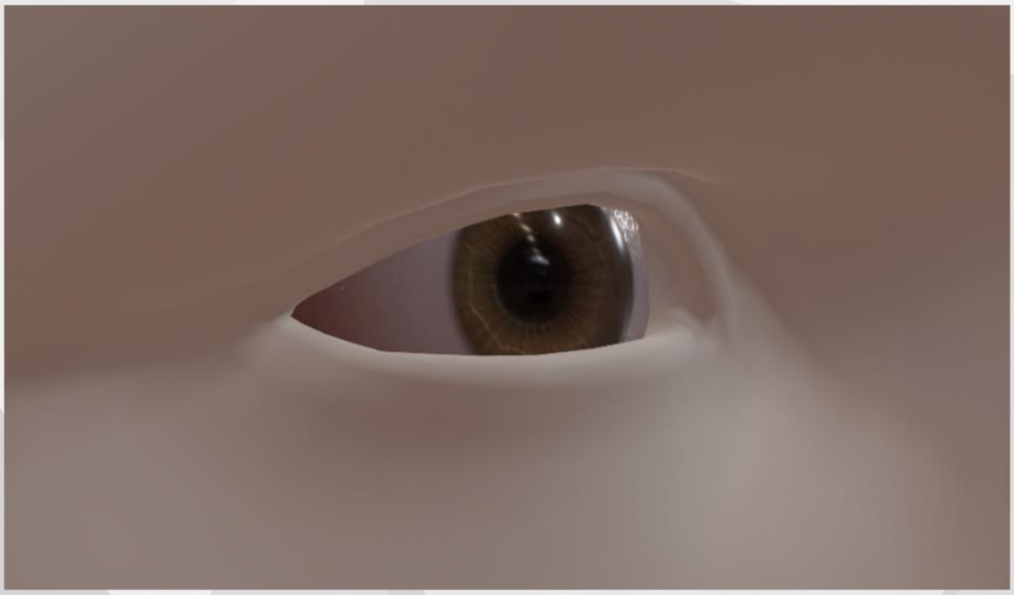
This shape represents a distance field from line segments joined between the joints. We can get this in the shader.



Better still, as the eye is roughly spherical we can project the joint positions onto the eyeball surface reasonably easily to get a better result

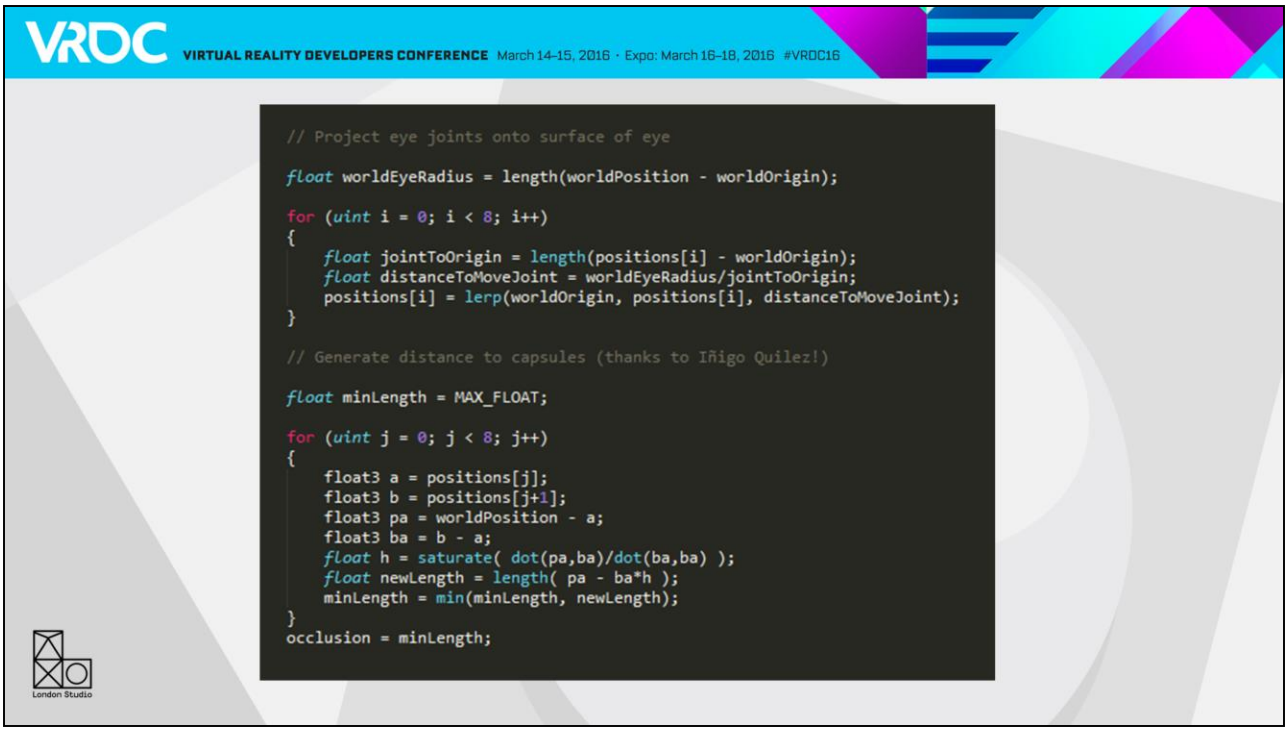


And this is the field applied to the eye, exaggerated a bit for clarity. There are artist controlled values to adjust the amount and distance of AO and cavity values independently.



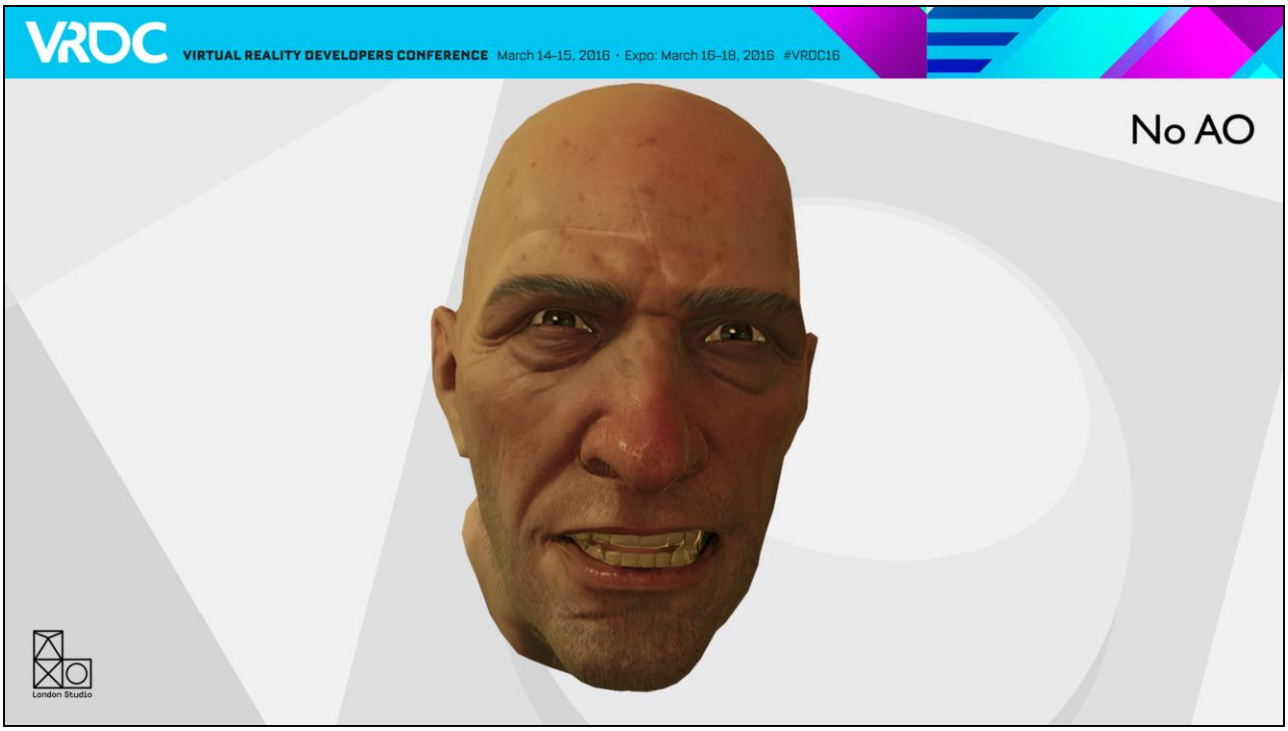
London Studio

As you can see, it’s a large difference.

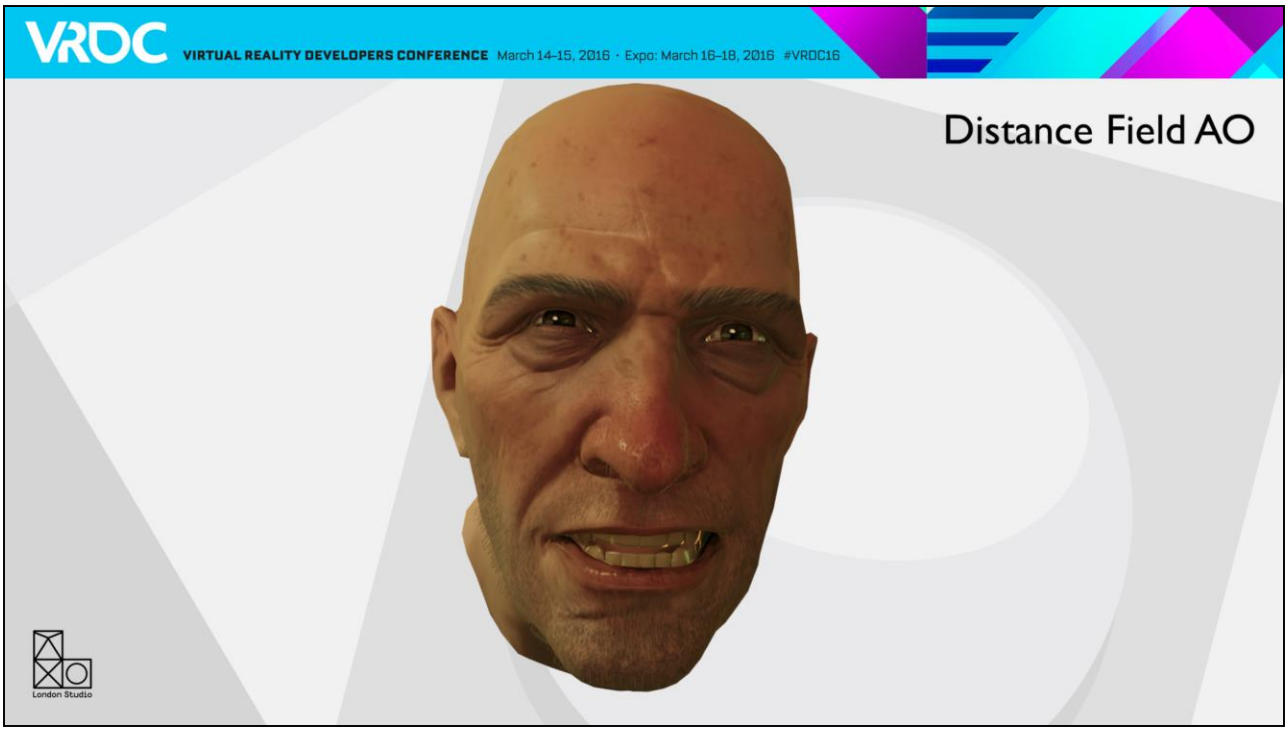



```
// Project eye joints onto surface of eye

float worldEyeRadius = length(worldPosition - worldOrigin);

for (uint i = 0; i < 8; i++)
{
    float jointToOrigin = length(positions[i] - worldOrigin);
    float distanceToMoveJoint = worldEyeRadius/jointToOrigin;
    positions[i] = lerp(worldOrigin, positions[i], distanceToMoveJoint);
}

// Generate distance to capsules (thanks to Iñigo Quilez!)

float minLength = MAX_FLOAT;

for (uint j = 0; j < 8; j++)
{
    float3 a = positions[j];
    float3 b = positions[j+1];
    float3 pa = worldPosition - a;
    float3 ba = b - a;
    float h = saturate( dot(pa,ba)/dot(ba,ba) );
    float newLength = length( pa - ba*h );
    minLength = min(minLength, newLength);
}
occlusion = minLength;
```

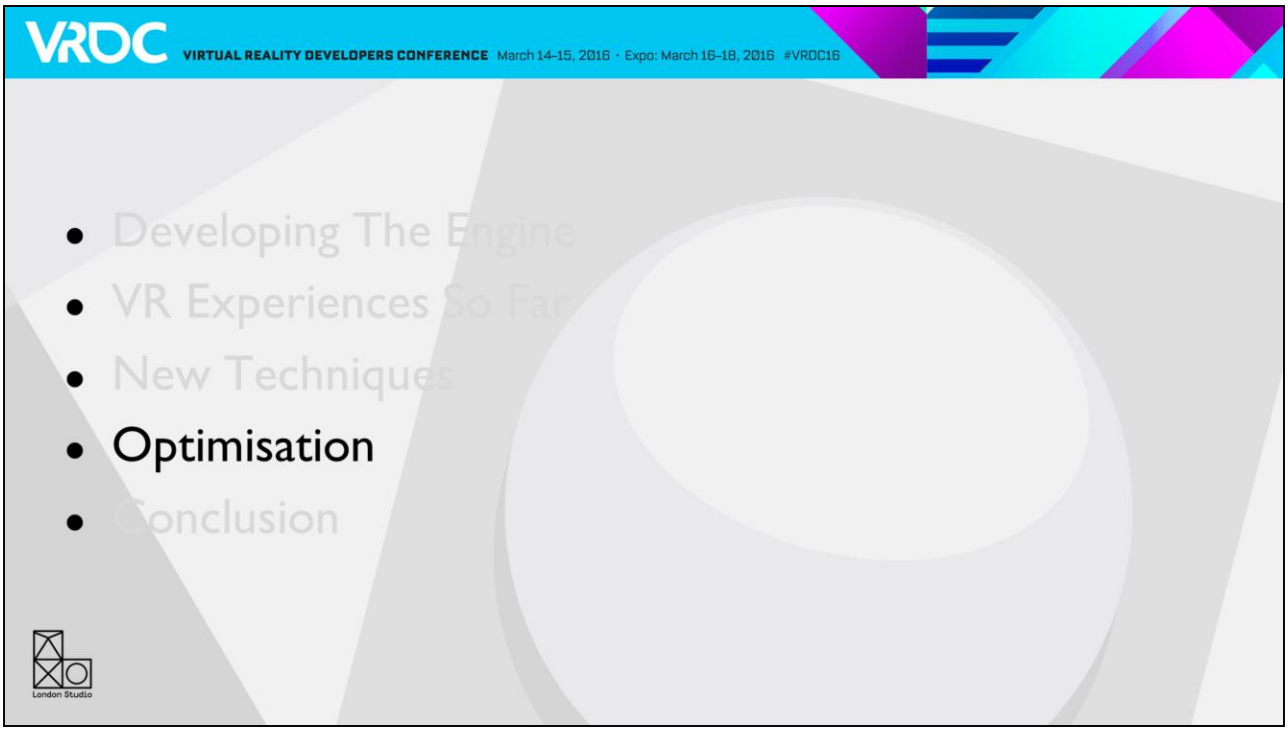
This is the shader code. Thanks to Iñigo Quilez for distance function. It's a fair amount of maths, but the pixel coverage is generally low. This is an example of one of our bespoke shading nodes I mentioned earlier – I think it's much more readable in code.



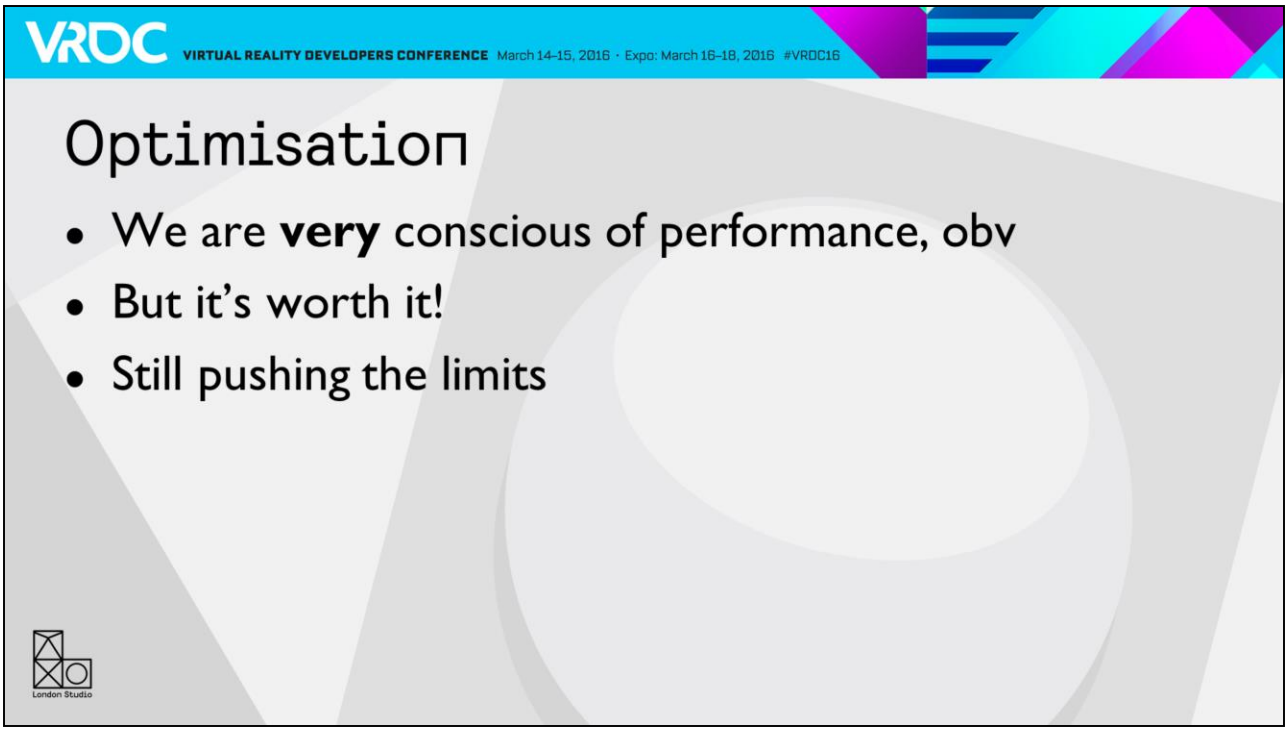
Here’s the face without distance field AO



And with. The facial expression here reads quite differently with the AO applied. We also apply it to the teeth in a similar fashion but without the spherical projection.



My last section today is on optimisation.



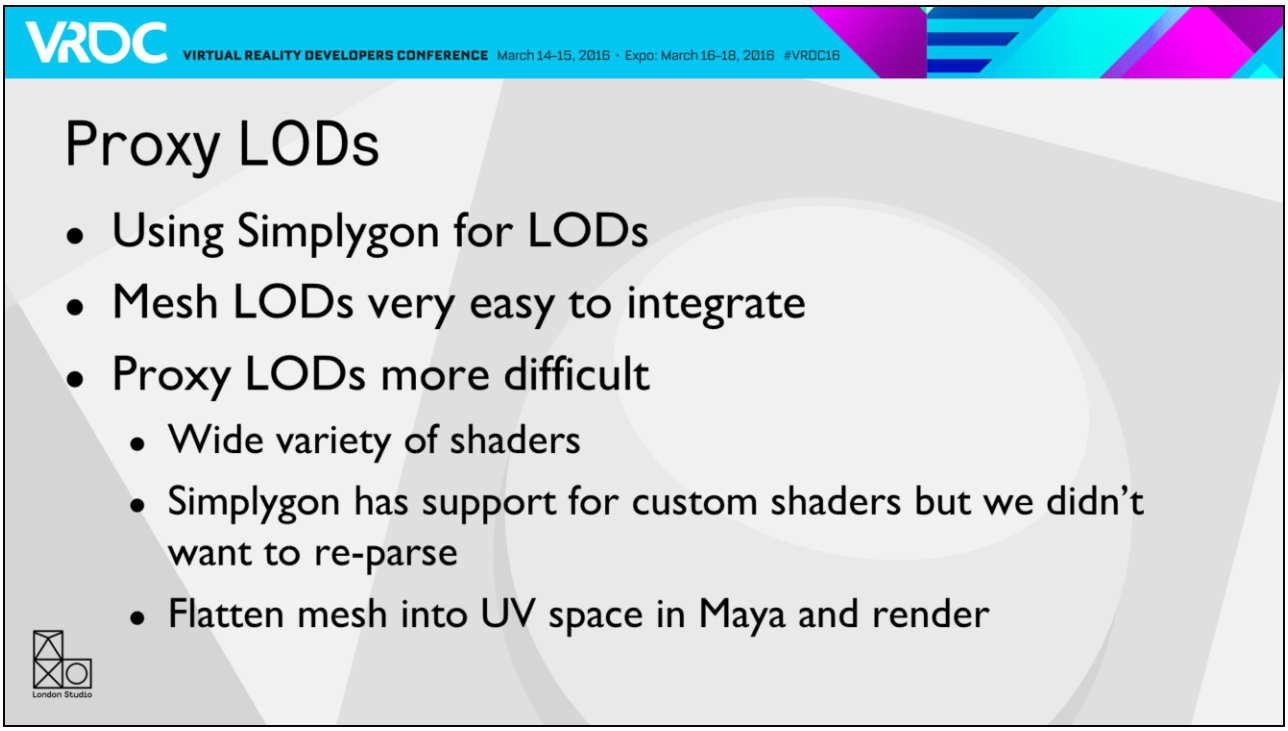
We are very conscious of performance. 60fps in a 2D game is difficult. 60fps with no frame drops ever, at greater than 1080p resolutions is even more difficult. On top of that, after tracking and reprojection are taken off we have about 15ms budget on GPU.

It's important to keep a constant 60 fps though, as immersion is greatly improved, and low frame rates can cause nausea.

It's worth the effort, too. VR may mean you have to make assets that don't look at refined as a 2D game, but the experience is transformative.

Still, as game developers and artists we always want to push things as far as we can

There's nothing really special about optimisation for VR other than that things often need to have more detail if you can get close to them or pick them up, but for performance need to drop to appropriate LODs quickly. As a result we've spent a lot of time on our LOD tools.

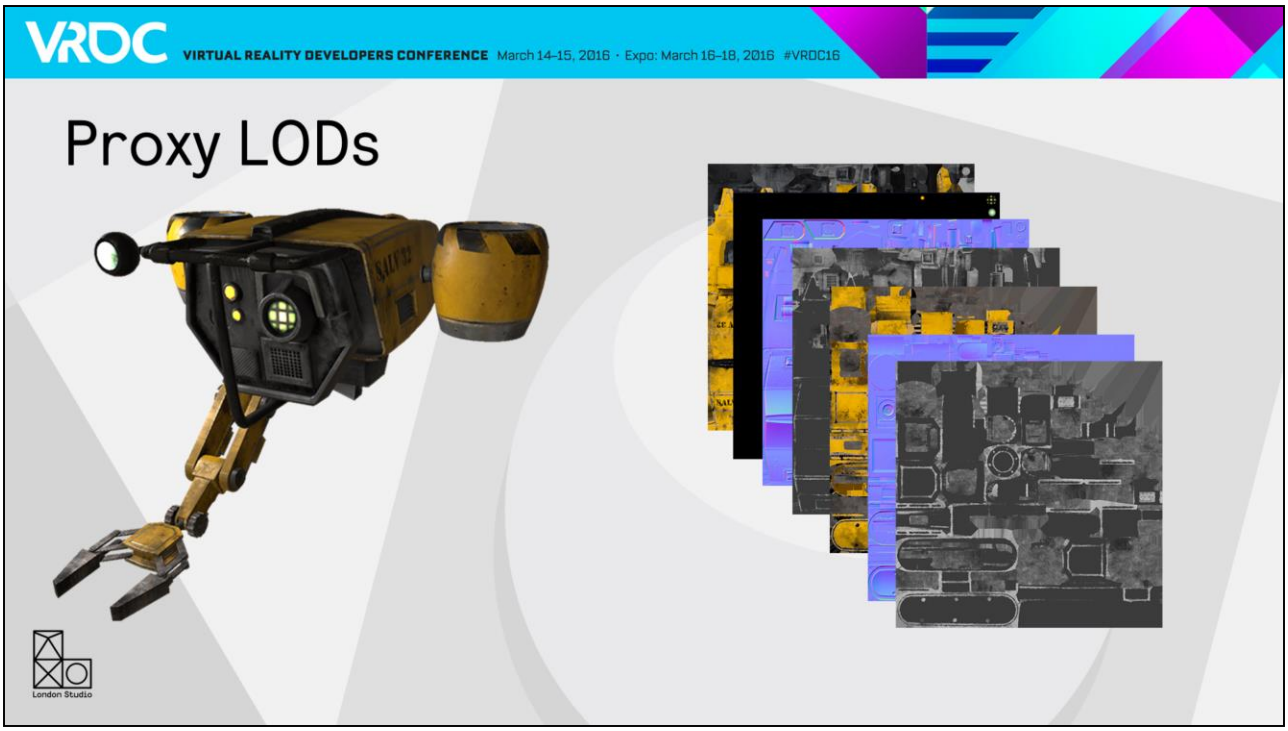


We use Simplygon as our primary tool for generating them. We found that adding the functionality to generate mesh lods, which are basically a poly reduced model using the same materials, was very easy to get in.

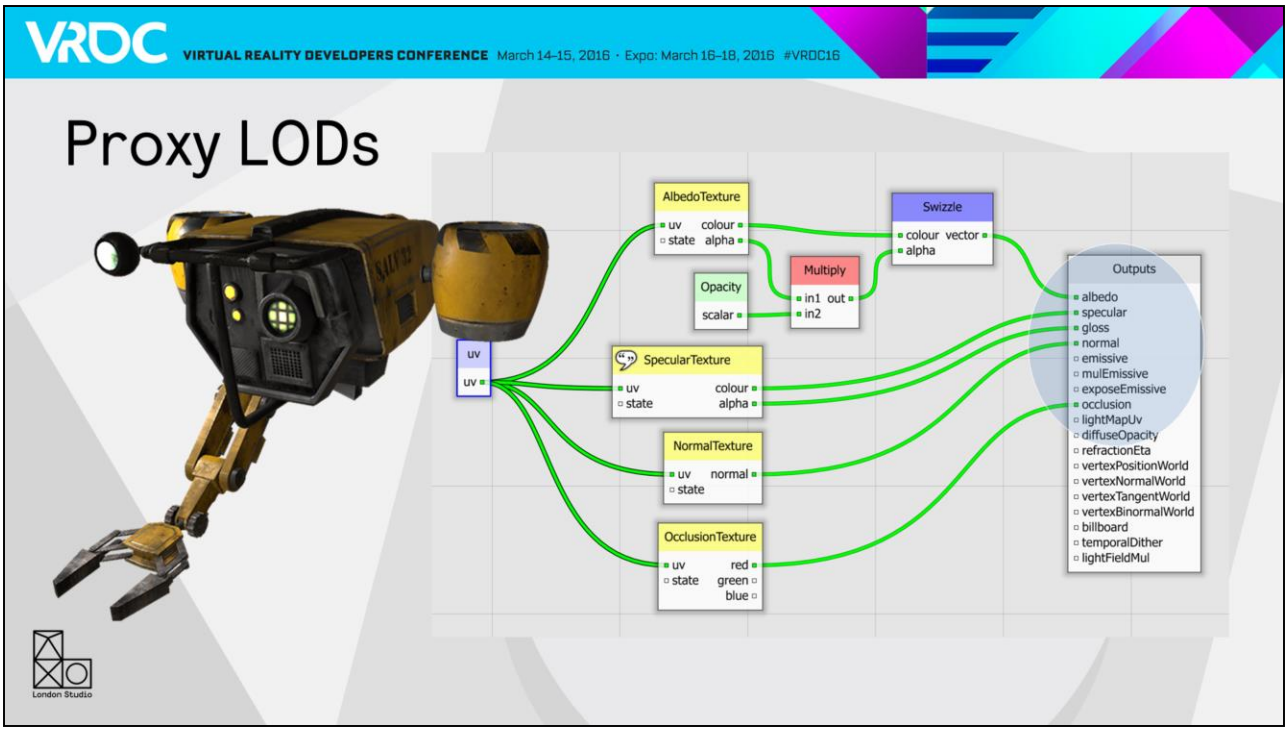
Proxy LODs were more difficult though. Proxy LODs can bake a mesh or number of meshes with many different materials into one entirely new mesh and textures through a voxelisation process. The mesh can in many cases be more efficient, particularly for the lower lods, as it doesn't have to respect the existing uv layout. Because it bakes a new normal map the mesh often lights more correctly, also. The main disadvantage is that you generate new textures for each LOD, so your texture memory requirements go up, but we have some strategies for dealing with this, as I'll go into shortly.

The problem with implementing Proxy LODs is that Simplygon needs to understand how your shaders work. We have a wide variety of shaders that we want to bake down, and although Simplygon has functionality for this, we didn't want to build a system to translate our shaders over.

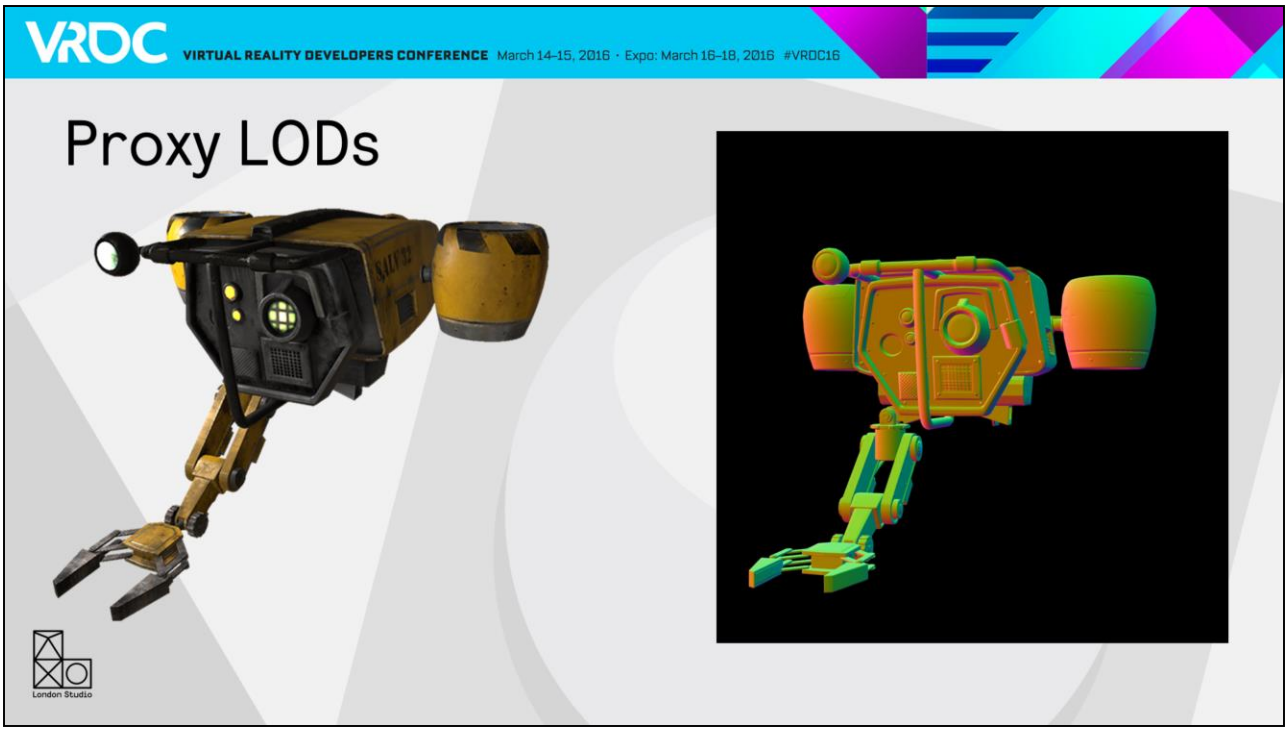
Andrew Maximov from Naughty Dog had a cool suggestion though – if you can already render your materials in Maya (and we can), flatten them into UV space and do some renders! So that's what we did.



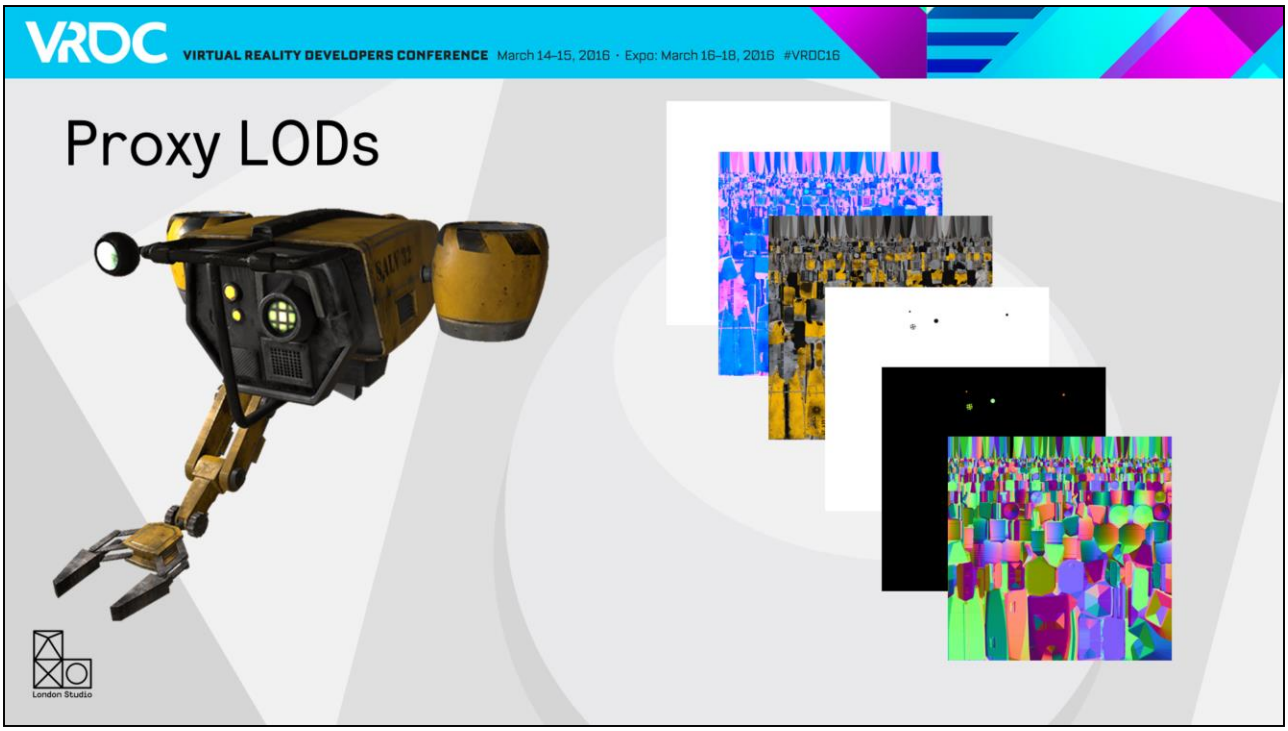
Here we have a drone from The Deep. It's a relatively simple shading setup, 2 materials and 7 textures. Obviously with layered shaders you can have a lot more inputs. The input mesh here is about 14k polys.



The first thing we do it to bake all the material outputs from the shader. These correspond with the values that we pass from our material editor into our lighting models. This is the shader that's used on the drone. As you can see, it's pretty simple, it doesn't really do much other than read a few textures and pass the values along.



We run the model through Maya's automatic UV mapping to generate a new set of unique UVs, and use the vertex shader to flatten the model into UV space. Here's the process with world space normals.



We go through this process for 6 maps, which are

- Base colour
- World space normals
- Metallic, gloss and occlusion packed into one texture
- Emissive is an HDR value, so we pack this in RGBD format over two textures
- Opacity

We also pad the edges to remove any seams. This all gets sent into Simplygon and we end up with…

Our LOD1 model, which is about 3800 tris. The textures you can see here don't make a lot of visual sense, as they have been swizzled. We convert the base colour from RGB into YCoCg luma and chroma format, and store textures at different resolutions.

The largest texture contains:

- Normal X and Y
- Gloss
- Base colour luminance

These are the most important components

The next texture contains

- The base colour's two chroma channels
- Metallic
- Occlusion

This can not only be stored at a lower resolution to save memory, but we also bias the texture sampling to always sample 1 mip lower than usual to

reduce texture bandwidth.

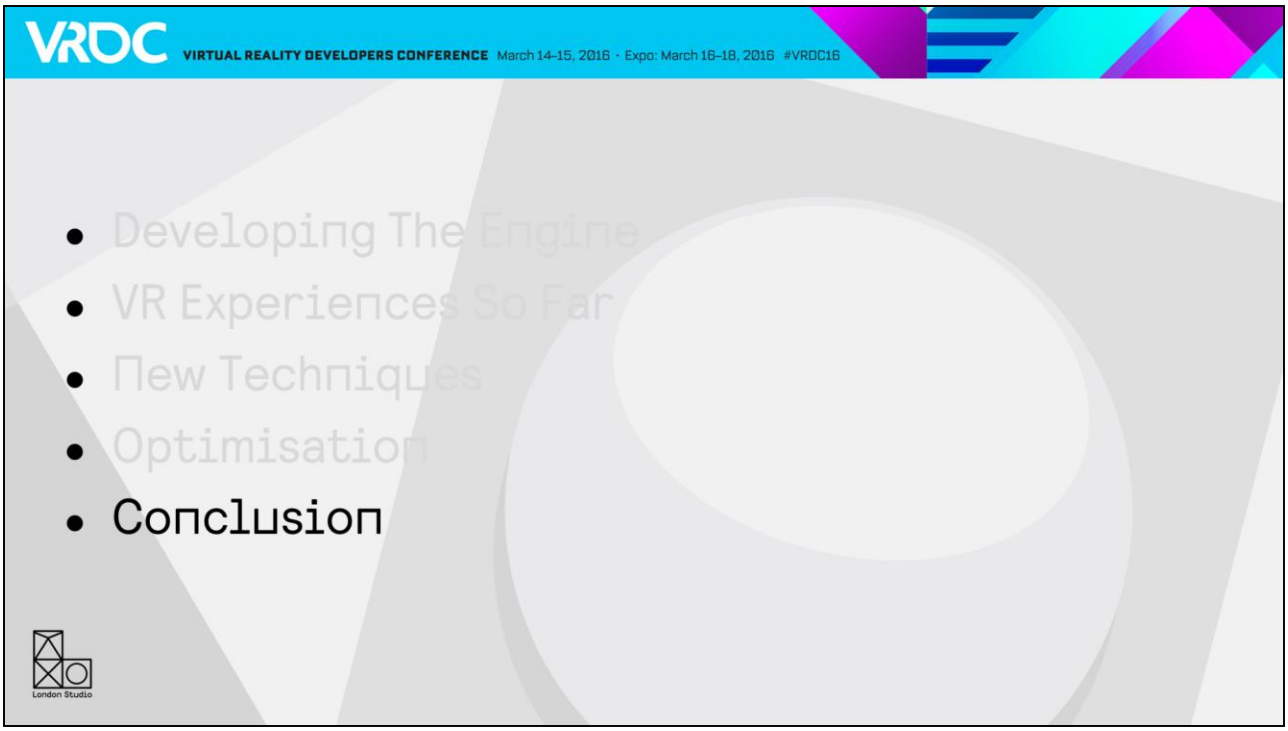
The last texture contains emissive stored in RGBD packed format. For this model, we need emissive, but obviously not all models do. So emissive and opacity textures are only generated and used on models that need them.

All the textures are compressed as BC7.

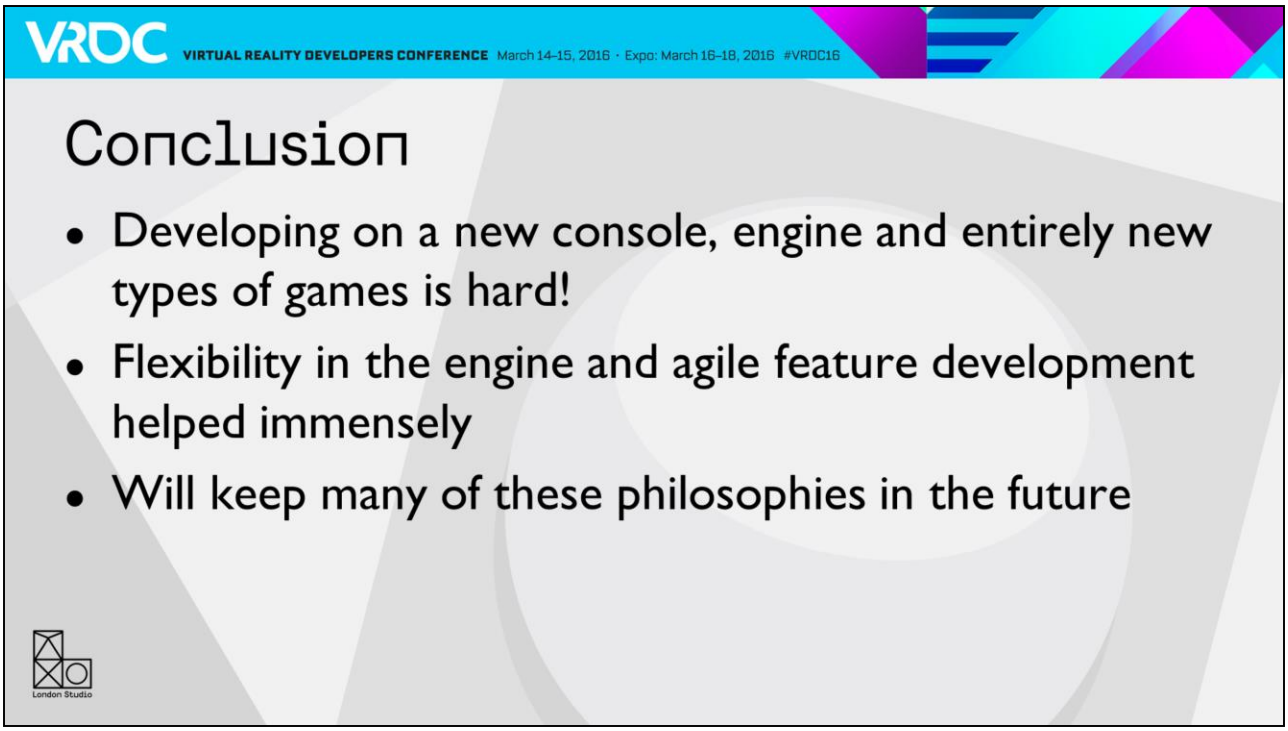
So we already store the second texture at a lower resolution, but in many cases even that can be improved upon.

Here we have replaced the second LOD texture with vertex colours. There is a noticeable difference on this model, particularly on the bottom of the turbines, but many models have very little variance in their chroma or metallic parameters, and you can only spot a difference in the occlusion. The result is that many of our proxy LODs can represent a full physically based material using only 1 texture read.

I hope on our next project to get some of these packing techniques onto our main models as well as the LODs to reduce texture bandwidth.



I think that's about all I have time for today!



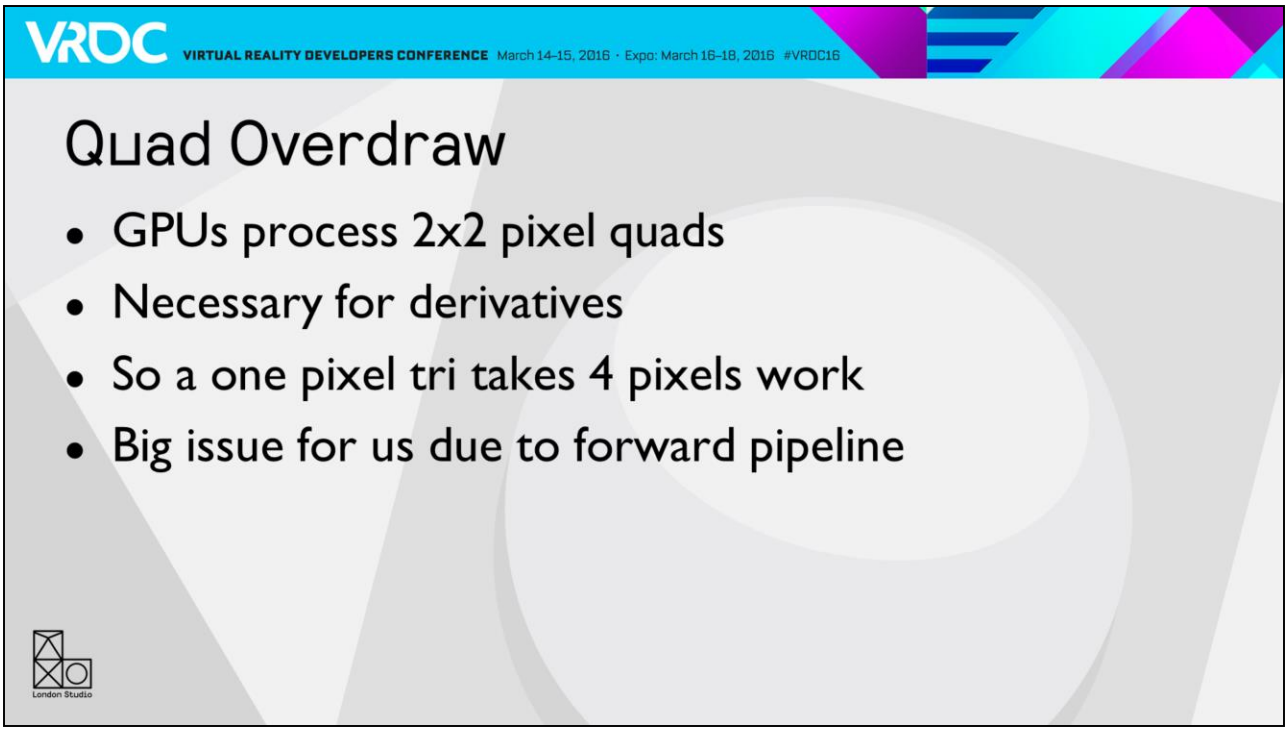
We had a lot to do at the same time. The arrival of PS4, developing a new engine, and also writing the rulebook for VR as we went along was hard

Our approach to making a flexible pipeline made this a lot easier. If in the future we develop a title which has more known content, it's likely that we will shift our focus away from the pure flexibility we have now, to solutions that are a bit more bespoke for the game. However, we are likely to keep the many of the same philosophies in the future, particularly around keeping the iteration times down to as short as possible.

Thank you for coming!

You can email me any (further) questions at this address, and also I'll be doing a wrap up session in Overlook 2022 if you want to chat about anything.

Bonus Slides



The first thing I want to talk about is quad overdraw.

So what is it? Well, GPUs process pixel shaders in 2x2 pixel quads. They need to do this so they can calculate derivatives, which are the rate of change in a parameter between pixels. The most common use for this is for uvs, as the rate of change of the UVs determine what mip-map to use.

Unfortunately, this means if you have multiple triangles in a quad, it costs you. A quad with the maximum of 4 triangles in costs 4x as much to render on the pixel shader.

This affects our renderer more than a deferred renderer. In a deferred pipeline, you get quad overdraw when you render your g-buffer, but as the lighting is all done on a single tri the complicated lighting shaders have no quad overdraw.

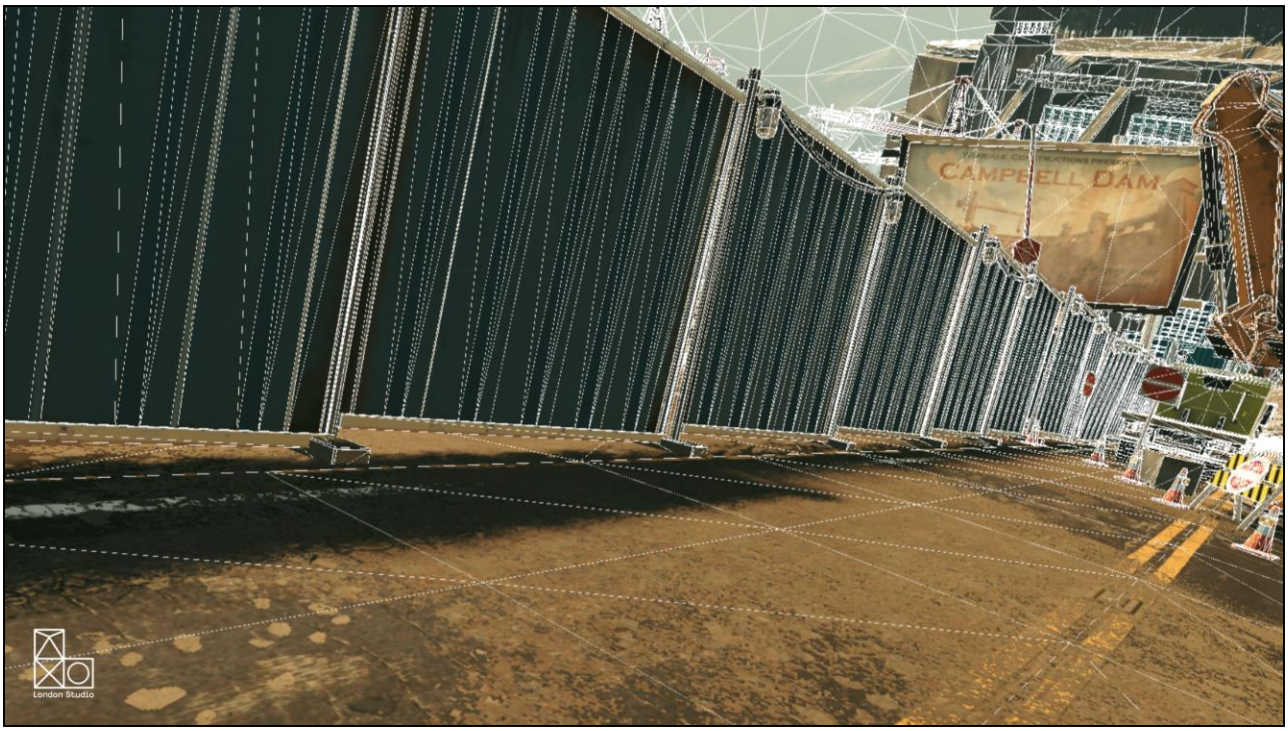
On a forward renderer like ours, you pay the quad overdraw cost on both the prepass and the main pass which does all the lighting, so it hits us worse.

This is a shot from our VR Luge game. We have a fairly standard debug view of wireframe on shaded <click>

This gives us a good idea of the geometry in the scene, and we can see in this case that these corrugated panels have a fair few polys. But although we know it's going to cost the GPU to transform the triangles on the vertex shader, we can get a better idea of how the pixel shaders are affected with the quad overdraw view

This view shows how much extra work we are doing for quad overdraw. Black is fully efficient – the quad only contains 1 triangle, no extra work is done. Blue means the quad contains two triangles, and you always get this on tri edges. Green and red are 3 and 4 respectively. Obviously we want to avoid this if possible.

You can see this section is pretty bad – we would want to replace the corrugated section with a plane with normal map as soon as possible. On the bottom here you can also see the effect of bevelling geometry. Although it's easier to do than adding it into a normal map, and it allows you to tile textures more easily, bevels can hurt performance if not used carefully.

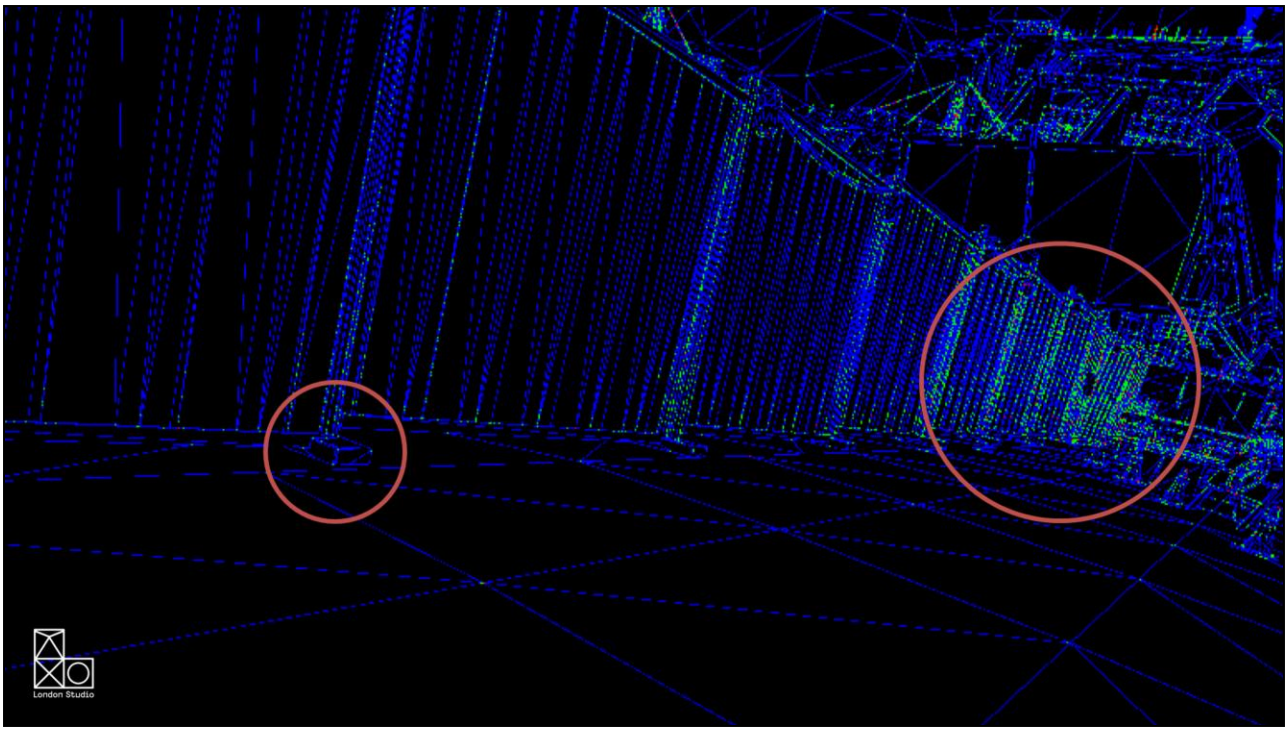


This is a shot from our Street Luge game. We have a fairly standard debug view of wireframe on shaded

This gives us a good idea of the geometry in the scene, and we can see in this case that these corrugated panels have a fair few polys. But although we know it's going to cost the GPU to transform the triangles on the vertex shader, we can get a better idea of how the pixel shaders are affected with the quad overdraw view

This view shows how much extra work we are doing for quad overdraw. Black is fully efficient – the quad only contains 1 triangle, no extra work is done. Blue means the quad contains two triangles, and you always get this on tri edges. Green and red are 3 and 4 respectively. Obviously we want to avoid this if possible.

You can see this section is pretty bad – we would want to replace the corrugated section with a plane with normal map as soon as possible. On the bottom here you can also see the effect of bevelling geometry. Although it's easier to do than adding it into a normal map, and it allows you to tile textures more easily, bevels can hurt performance if not used carefully.



This is a shot from our Street Luge game. We have a fairly standard debug view of wireframe on shaded

This gives us a good idea of the geometry in the scene, and we can see in this case that these corrugated panels have a fair few polys. But although we know it's going to cost the GPU to transform the triangles on the vertex shader, we can get a better idea of how the pixel shaders are affected with the quad overdraw view

<click>

This view shows how much extra work we are doing for quad overdraw. Black is fully efficient – the quad only contains 1 triangle, no extra work is done. Blue means the quad contains two triangles, and you always get this on tri edges. Green and red are 3 and 4 respectively. Obviously we want to avoid this if possible.

You can see this section is pretty bad – we would want to replace the corrugated section with a plane with normal map as soon as possible. On the bottom here you can also see the effect of bevelling geometry. Although it's easier to do than adding it into a normal map, and it allows you to tile textures more easily, bevels can hurt performance if not used carefully.

Quad Overdraw

- No silver bullets
- Educate artists
- http://blog.selfshadow.com/publications/overdraw-in-overdrive/
- LODs help here

What can we do about it?

Well, there aren't any easy wins. The best method to combat it is to avoid using geometry where you can and use normal maps instead, and LOD effectively so that the triangle density is relatively consistent throughout the scene. For further reading and on how to implement the debug view I recommend Stephen Hill's excellent post on the subject.