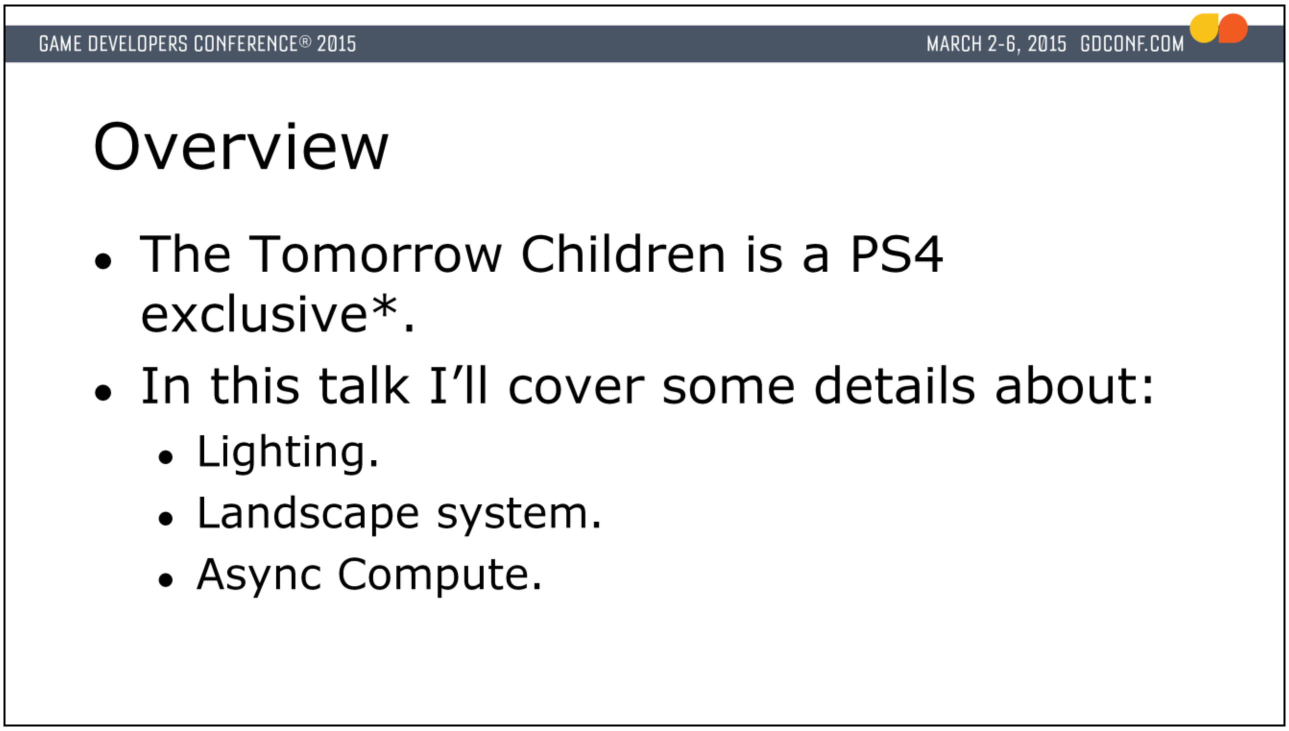
The Technology Of The Tomorrow Children

James McLaren
Director Of Engine Technology.
Q-Games Ltd.

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015



Overview

- The Tomorrow Children is a PS4 exclusive*.
- In this talk I'll cover some details about:
  - Lighting.
  - Landscape system.
  - Async Compute.

*Note: Previously said "Summer". This was from an earlier set of slides.

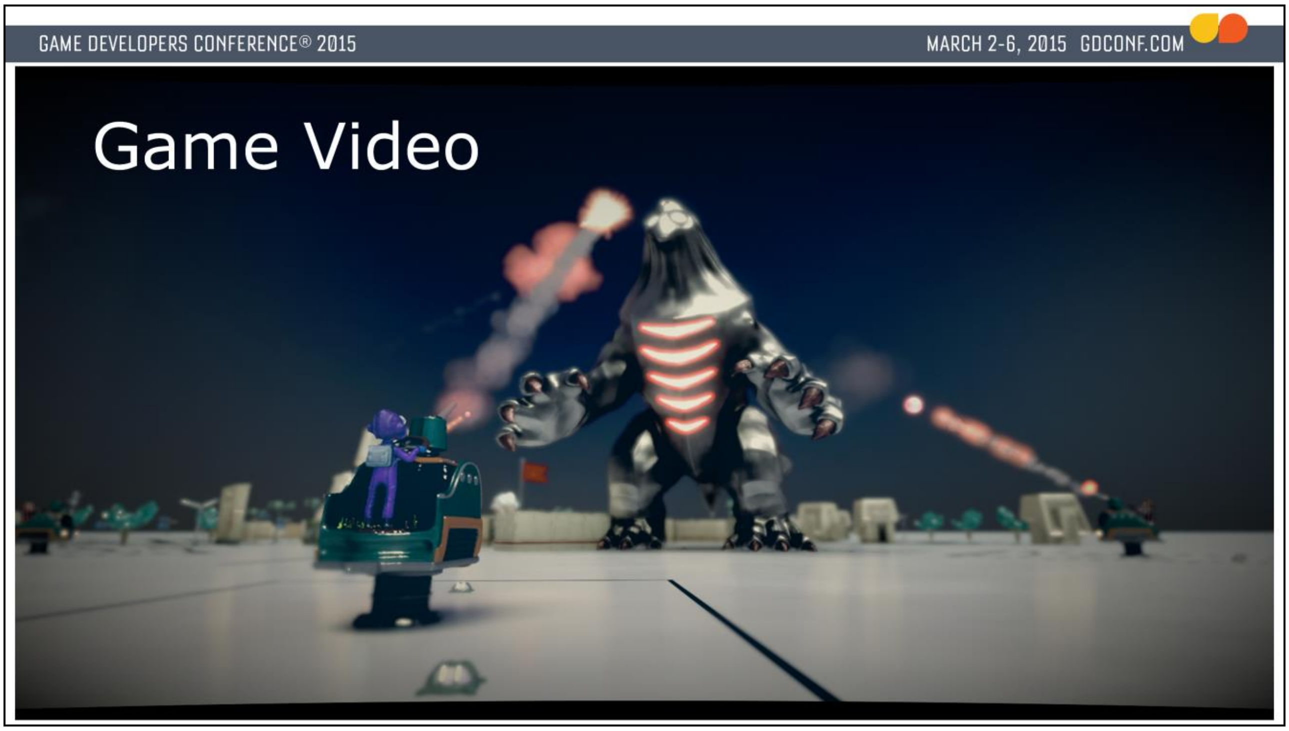
Hi, I'm James McLaren, Director of Engine Technology at Q-Games out in Kyoto, Japan.

So today I'm going to talk a little bit about some of the technology that we've put in our upcoming PS4 game "The Tomorrow Children".

I'm mostly going to be talking about the Lighting system, for which we implemented a form of realtime global illumination.

But I'll also give a few details about our landscape system,

And I'll hopefully have time at the end to talk a little bit about our use of Asyncronous Compute.



Game Video

So I’m just going to run a trailer for the game just in case any of you haven’t seen the madness that we’ve been making....

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Lighting

Attempting to implement Global Illumination seemed like a lofty goal,

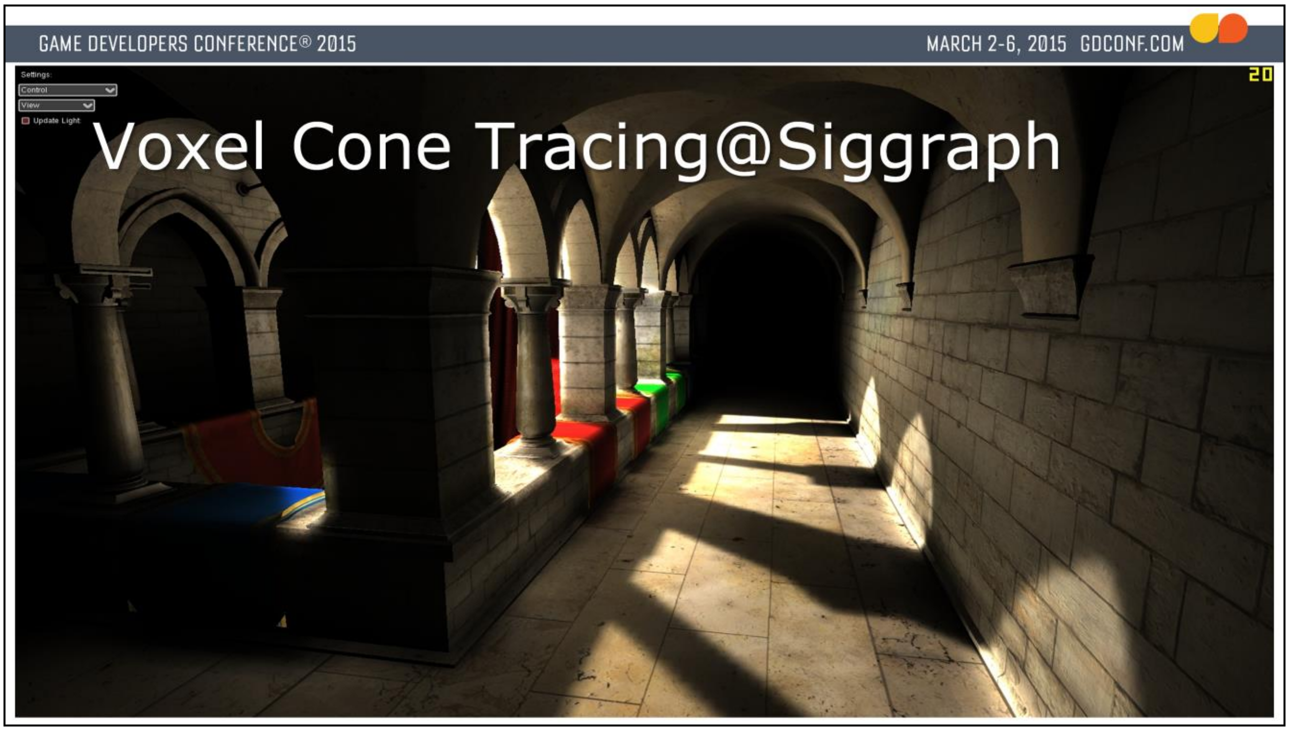
but it was necessary for us because of the unique look that we were aiming for.

We certainly hoped the PS4 would have the power to let us achieve it,

But it wasn't entirely clear at the start what was the best path to get there.

We looked into Light Propagation Volumes, and some Virtual Point Light methods,

But after some research the route that seemed most promising to us was



Voxel Cone Tracing@Siggraph

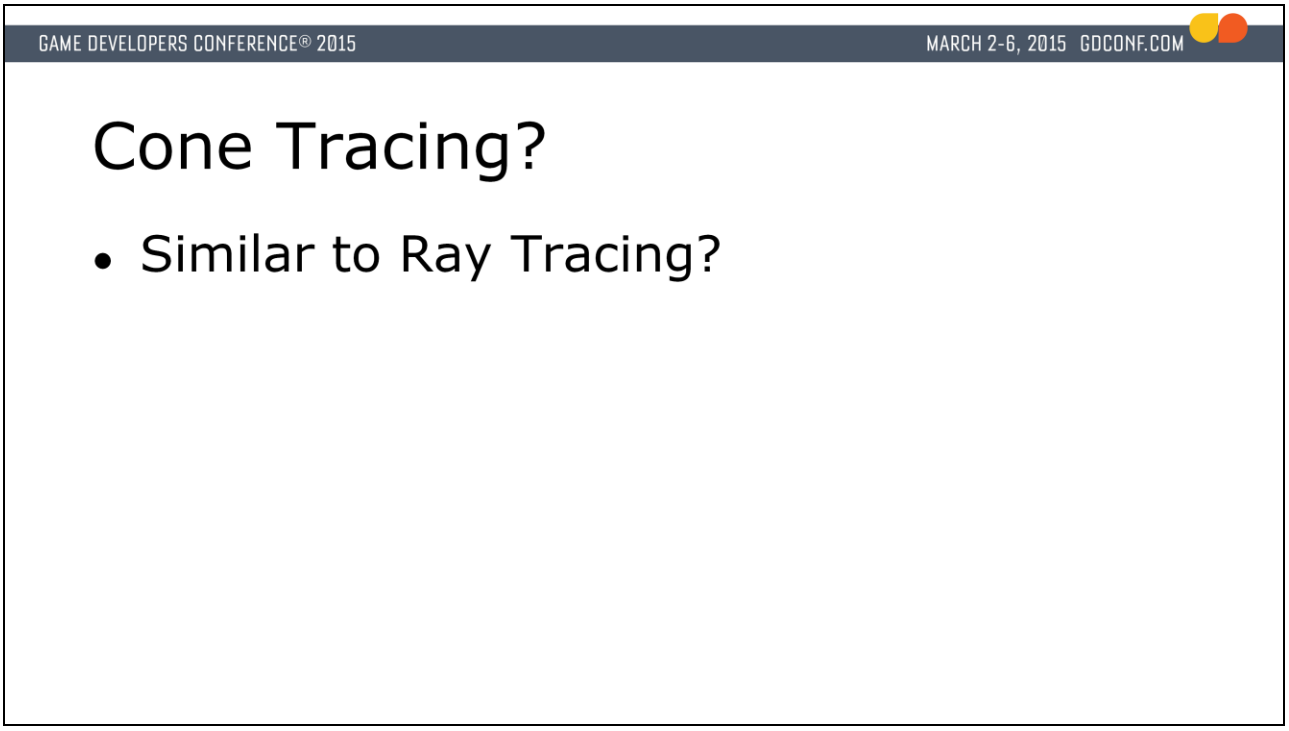
This very interesting talk that was given at Siggraph 2011 by Cyril Crassin.

His work was on a technique he called Voxel Cone Tracing, which was capable of producing Global Illumination effects in real time on a high end GPU.

This worked by voxelizing the scene into what he called a Sparse Voxel Octree, injecting lighting information into this structure,

and then tracing cones through that from the location of a pixel in world space in order to gather the indirect illumination affecting it.

As you can see from the image, this gives a very pleasing result, and it caused quite a stir at Siggraph, as it felt like quite a big step forward for realtime GI.



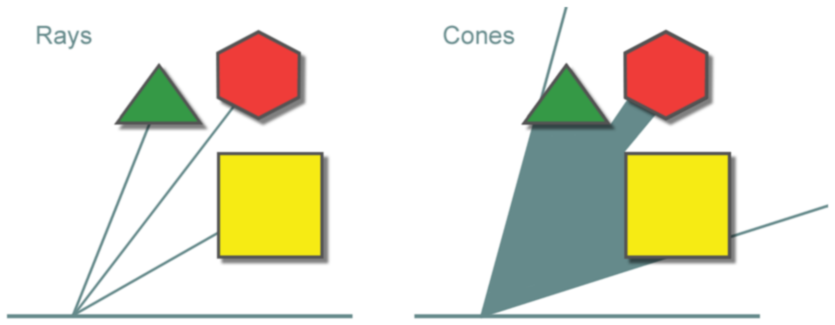
Cone Tracing?

- Similar to Ray Tracing?

So, what is Voxel Cone Tracing?

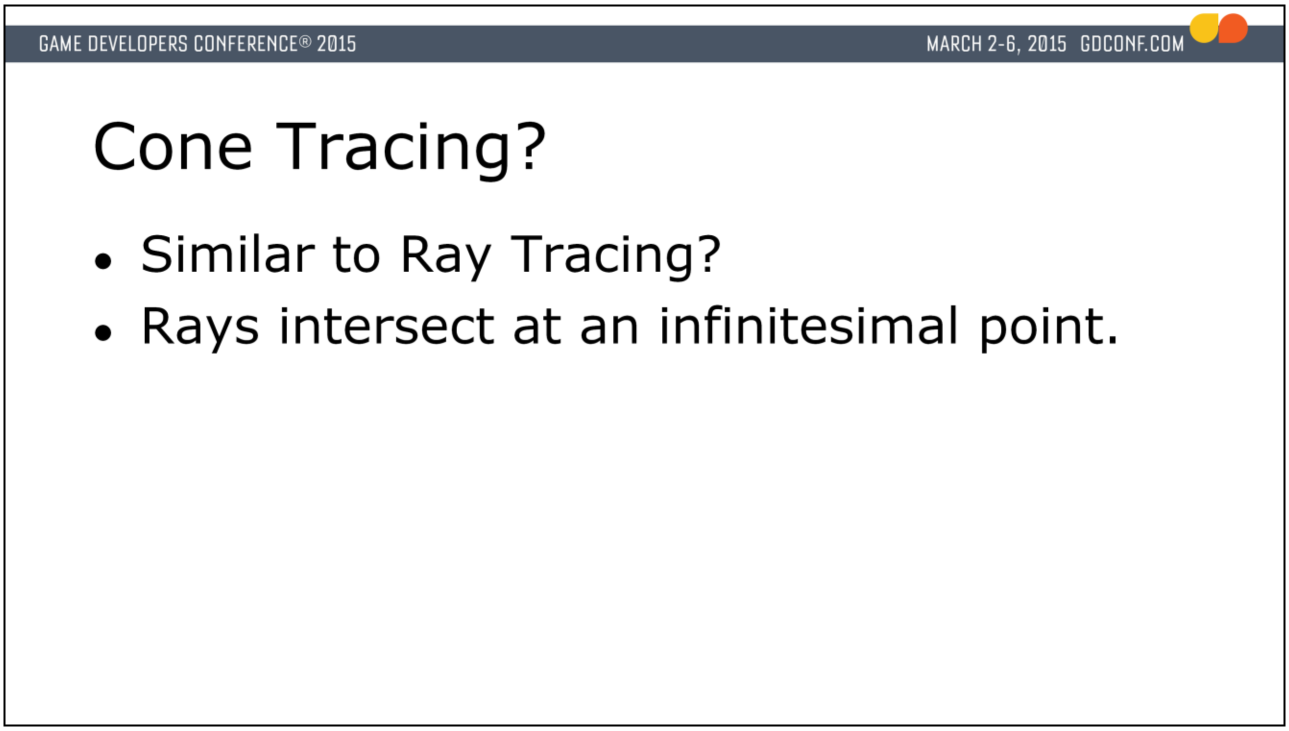
Well, it’s a technique that shares some similarities with ray tracing

Like Ray Tracing?



So, for both techniques we're trying to obtain a number of samples of the incident radiance at a point by shooting out primitives, and intersecting them with the scene.

And if we take enough well distributed samples, then we can combine them together to form an estimate for the incident lighting at our point, which we could then feed through a BRDF that represented the material properties at our point, and calculate the exitant lighting.

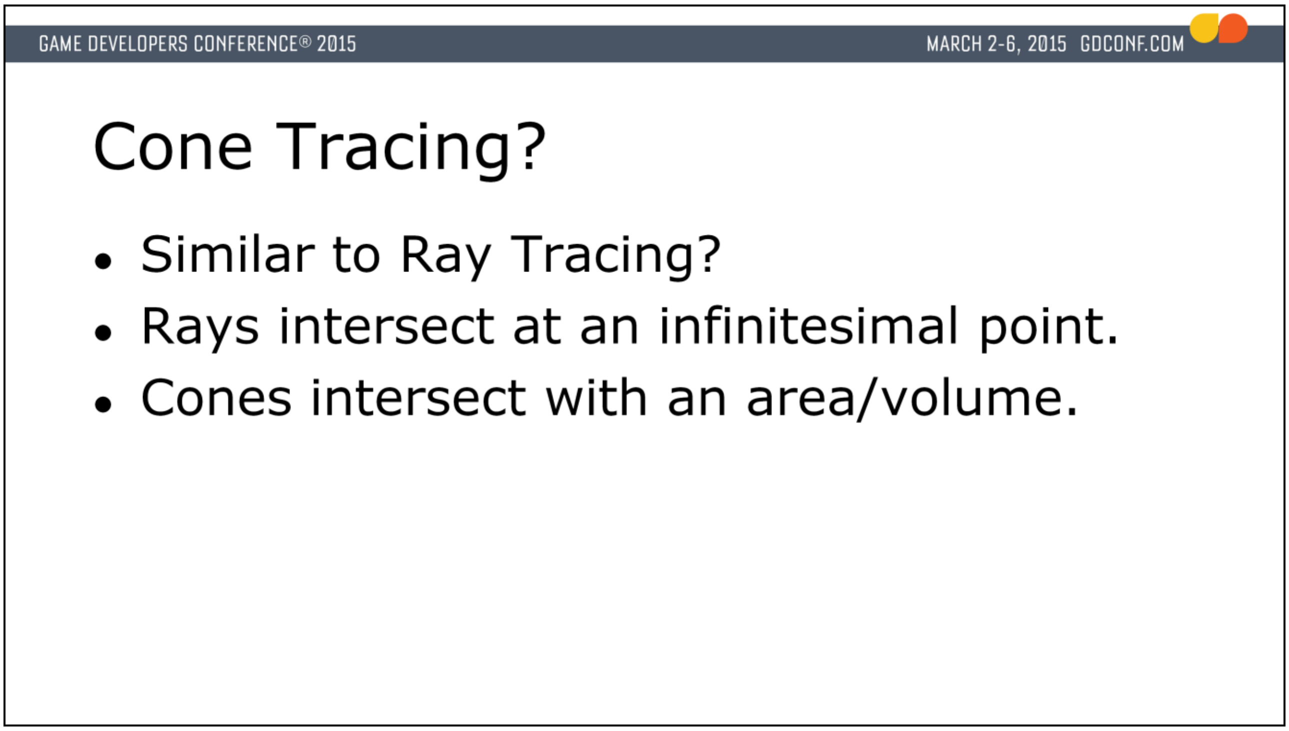

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Cone Tracing?

- Similar to Ray Tracing?
- Rays intersect at an infinitesimal point.

So the key difference between the two approaches is what happens when we evaluate the intersection of our primitives with the scene.

With a ray the intersection is at a point,

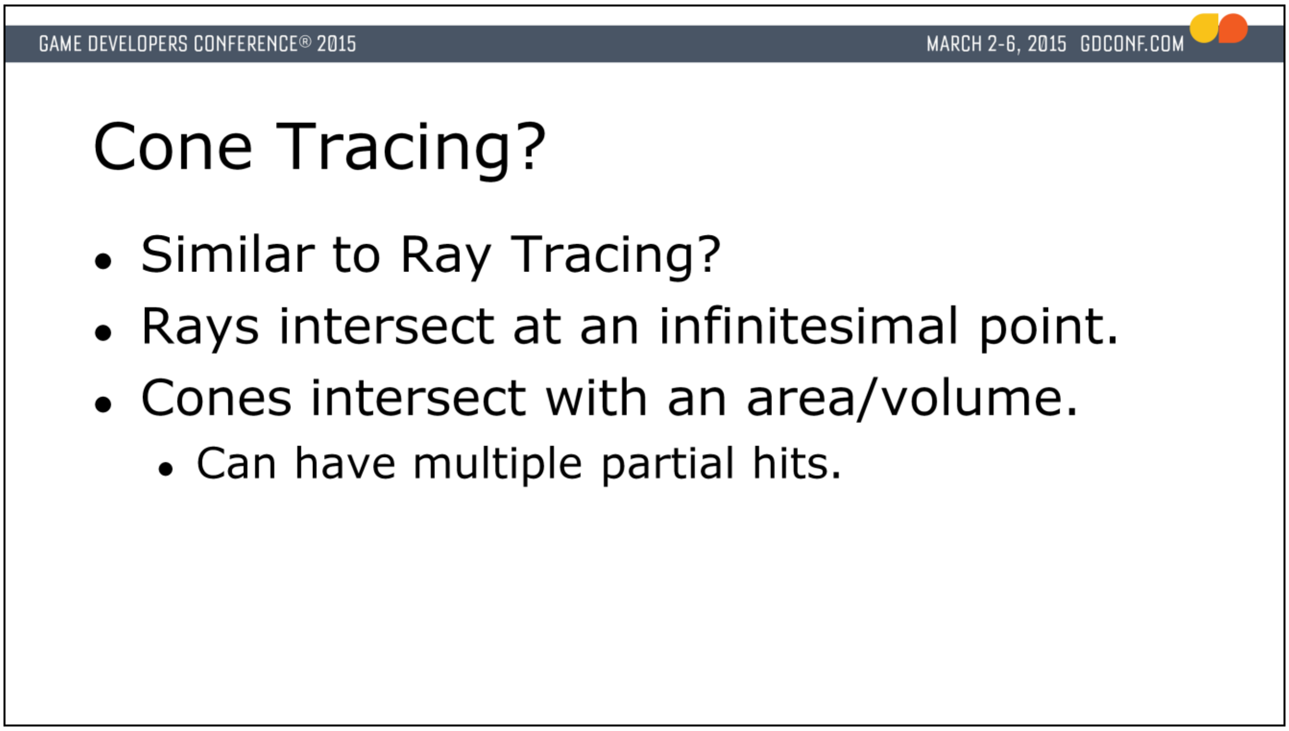

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Cone Tracing?

- Similar to Ray Tracing?
- Rays intersect at an infinitesimal point.
- Cones intersect with an area/volume.

where as with a Cone, it ends up being a an area or perhaps a volume, depending on how you are thinking about it.

The important thing, is that because it's no longer a point, the properties of our estimate change.

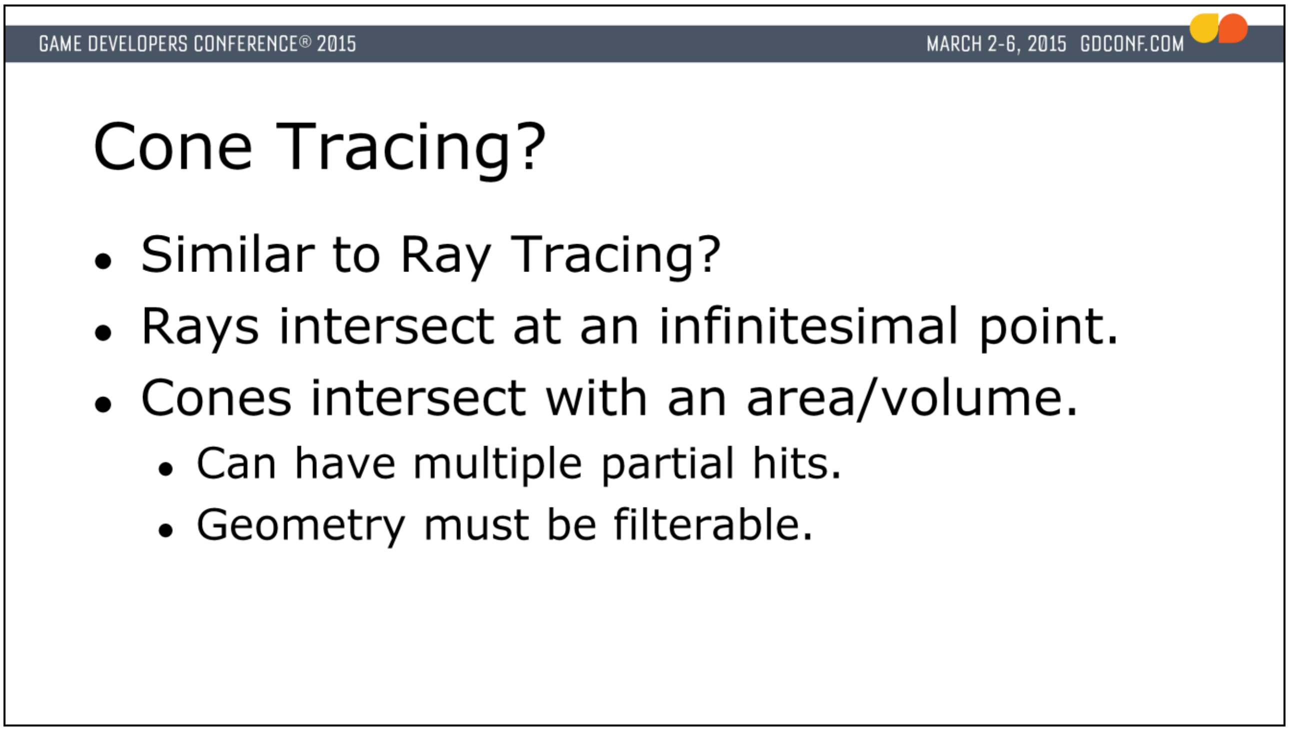

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Cone Tracing?

- Similar to Ray Tracing?
- Rays intersect at an infinitesimal point.
- Cones intersect with an area/volume.
  - Can have multiple partial hits.

Firstly, we aren’t necessarily looking in just one location in the scene for our intersection anymore,

we can have multiple partial hits by our cone.

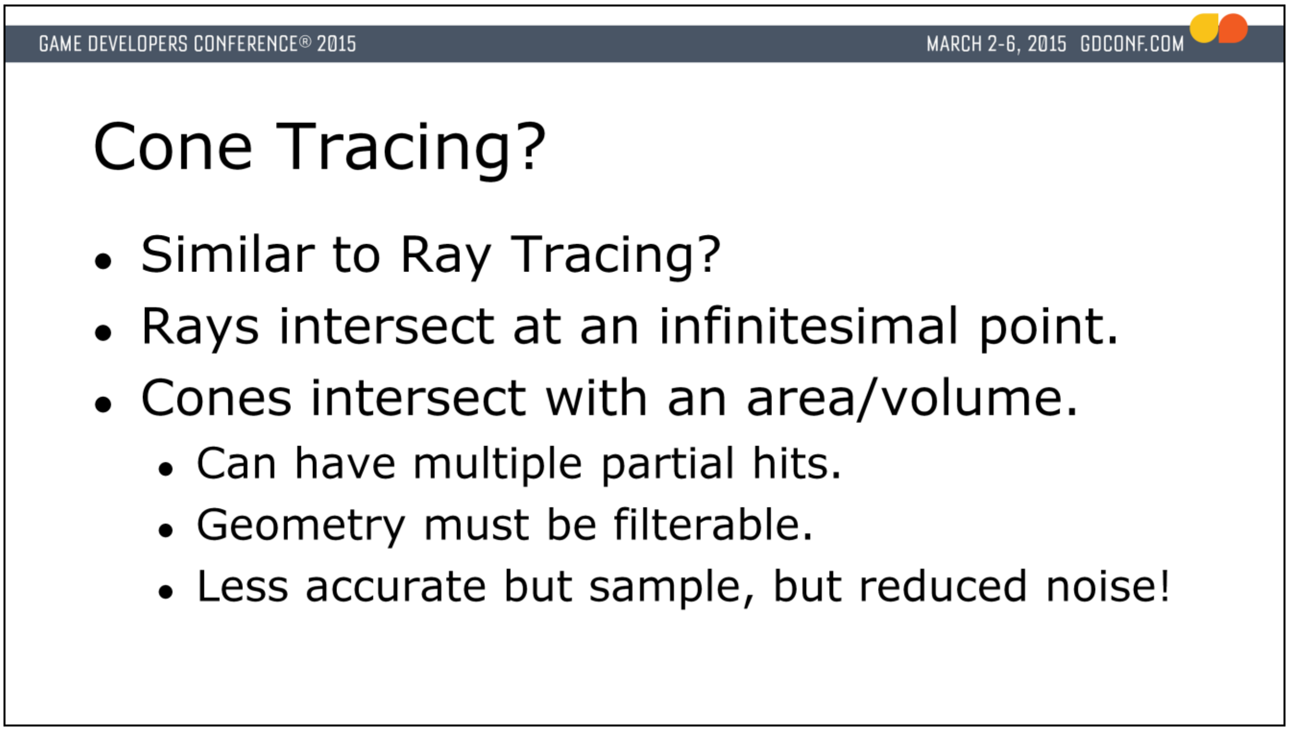

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Cone Tracing?

- Similar to Ray Tracing?
- Rays intersect at an infinitesimal point.
- Cones intersect with an area/volume.
  - Can have multiple partial hits.
  - Geometry must be filterable.

And secondly, because of the need to evaluate the scene over an area, our scene has to be filterable.

Also, because we are filtering we are no longer getting an exact value, we are getting an average, and so the accuracy of our estimate goes down.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

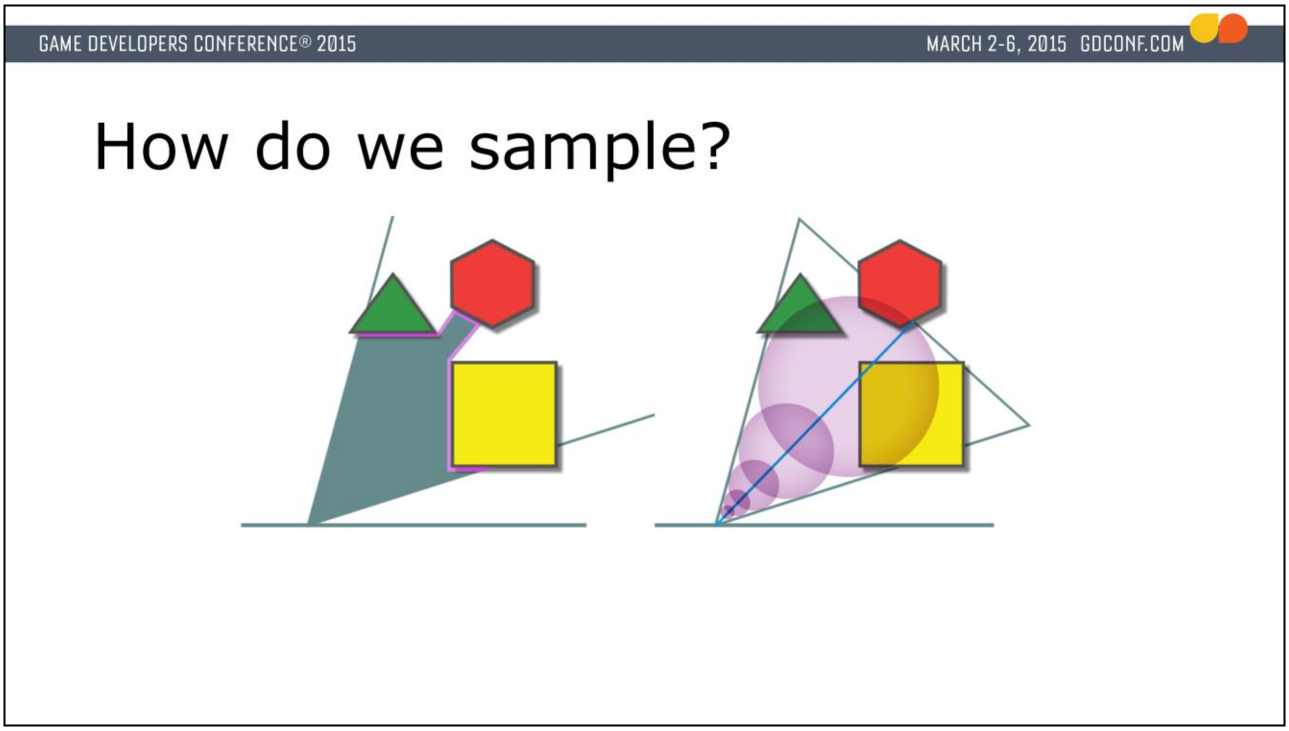
Cone Tracing?

- Similar to Ray Tracing?
- Rays intersect at an infinitesimal point.
- Cones intersect with an area/volume.
  - Can have multiple partial hits.
  - Geometry must be filterable.
  - Less accurate but sample, but reduced noise!

But on the upside, because we are evaluating an average, the noise, that we would typically get from ray tracing, is largely absent.

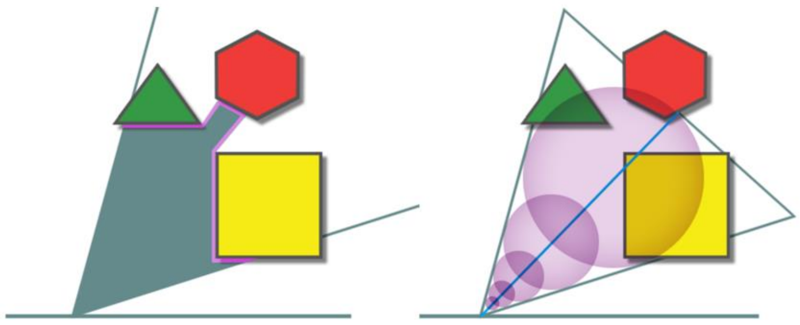
It was this property about cone tracing that really grabbed my attention when I saw Cyril Crassin's presentation.

Suddenly we had a technique where we could get a reasonable estimate of the irradiance at point, with a small number of samples, and because the scene geometry was filtered, we wouldn't have any noise, and it would be fast.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

How do we sample?

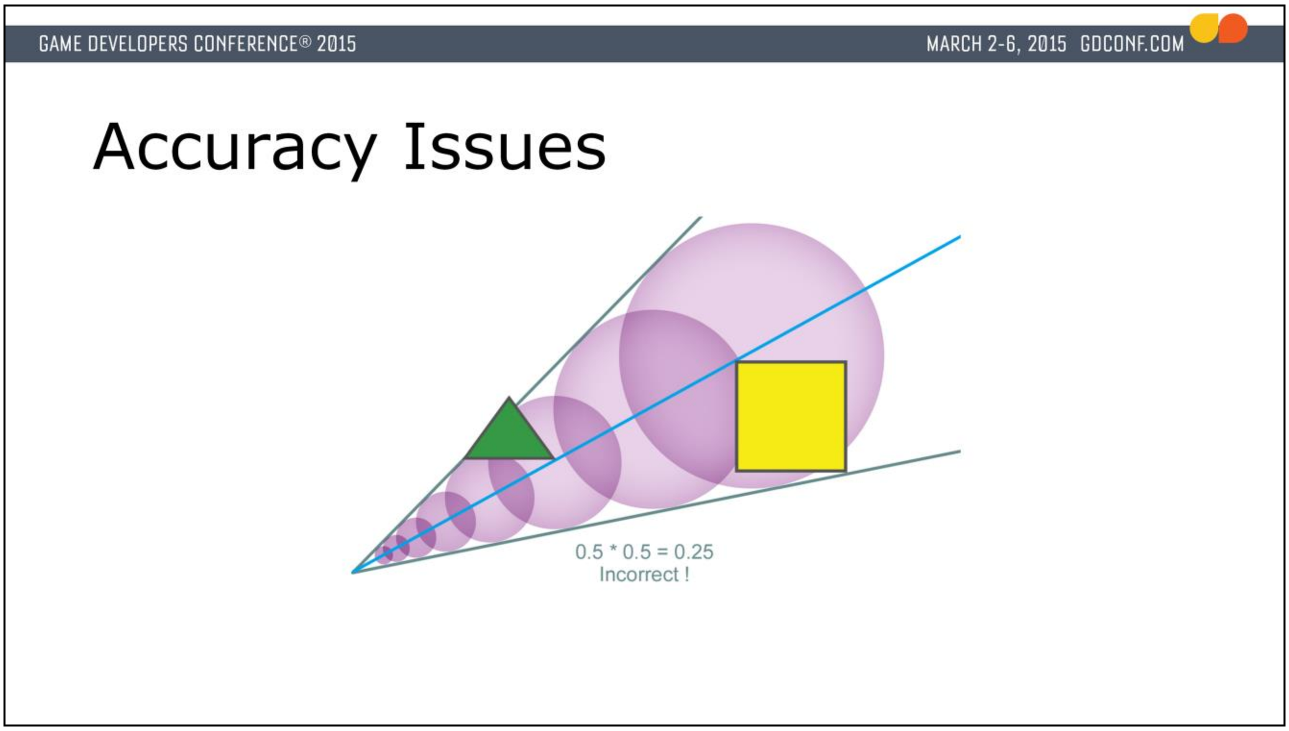


So obviously the challenge is, how do we sample from our cone.

The purple surface area in the picture on the left defining where we intersect is not a very easy thing to evaluate.

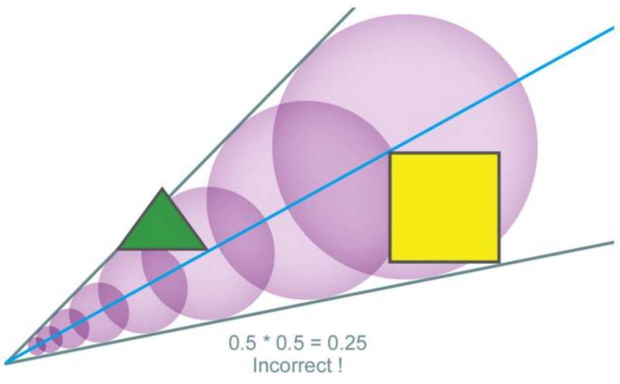
So instead, we take a number of volume samples along the cone, with each sample returning an estimate of light reflected towards to apex of the cone, as well as an estimate of the occlusion in that direction.

It turns out that we can combine these samples with the same basic rules we would use for ray marching through a volume.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Accuracy Issues



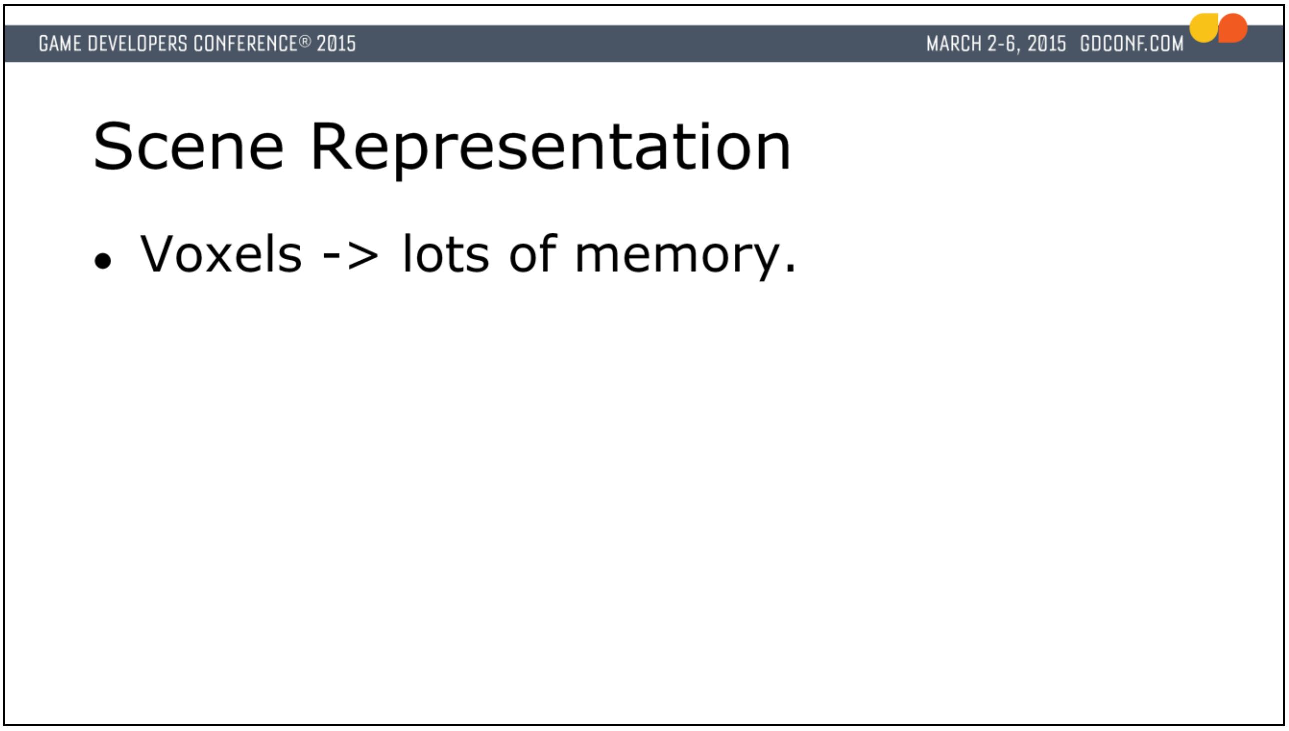
One thing worth noting is that because we are using volume samples, we are potentially going to get inaccurate results in the case where we have a cone that is partially occluded, as we don't carry any information about the shape of the occlusion onto the next sampling step.

So in this example, we have two partial occlusions of our cone, both of them occlude 50% of the light from the sky.

But as you can see in reality, if we were combine these two, we should get 100% occlusion of light.

Where as our cone trace will actually tell us that we can still see 25% of the light, because all we do is just naïvely combine their occlusion in the same way we would with alpha blending.

This doesn't tend to be such a big issue in practice, but it is worth bearing in mind that cone tracing is only a very rough estimate.



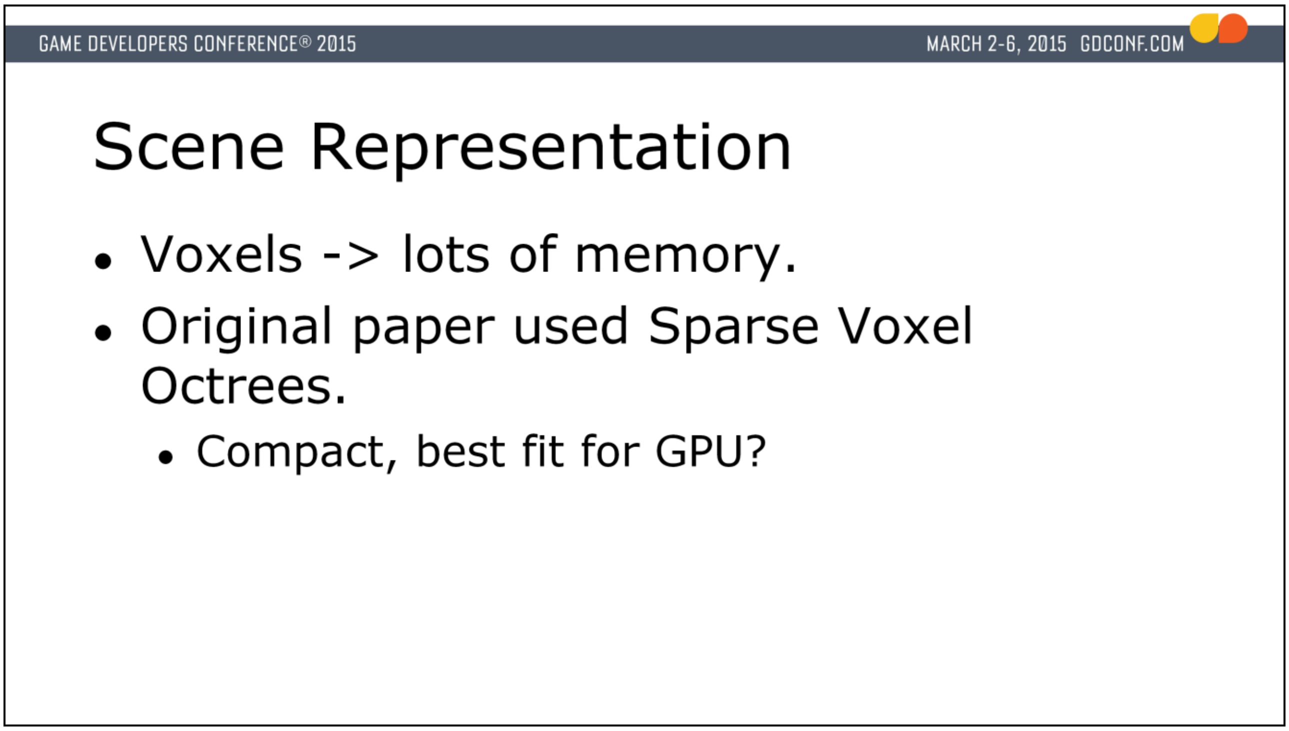
Scene Representation

- Voxels -> lots of memory.

So now we know what we need to do, accumulate our irradiance data as we march along our cone,

so the next big question is how do we store our scene to accommodate that?

Well, Voxels are the obvious answer, but a naïve voxel representation is likely to use a very large amount of memory.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

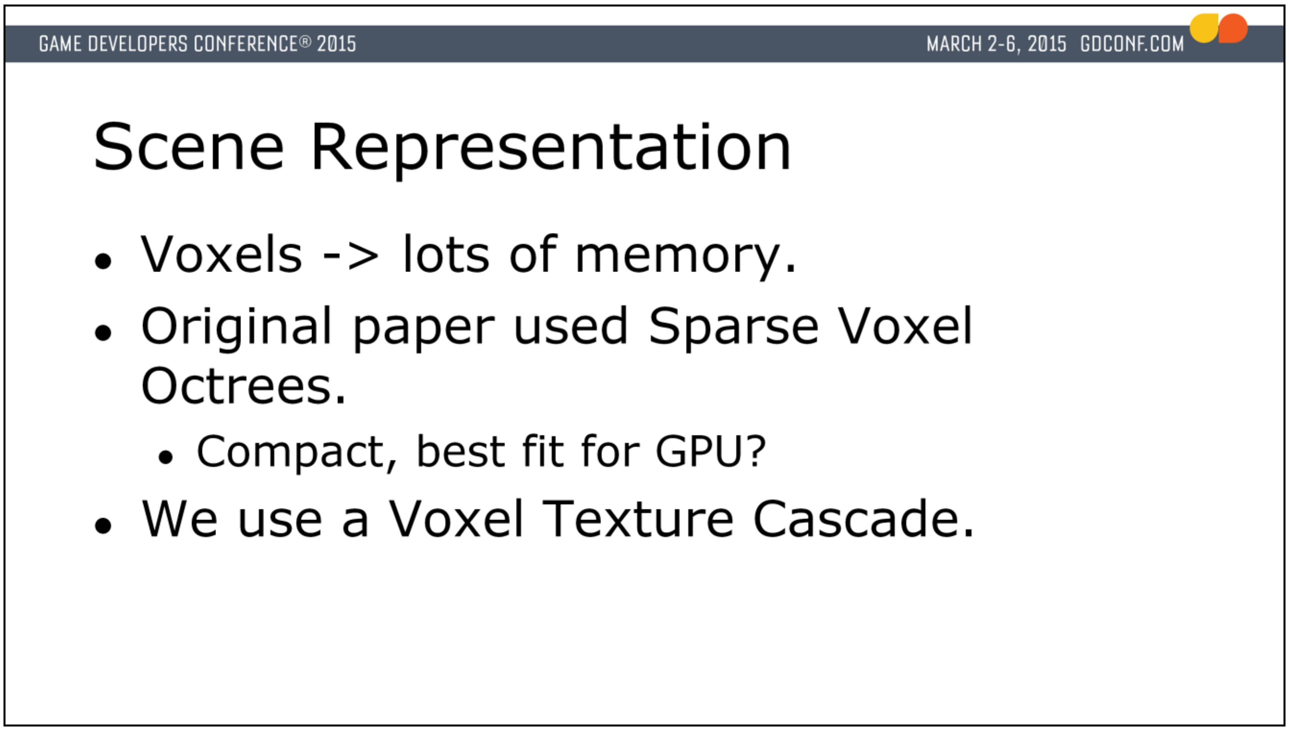
Scene Representation

- Voxels -> lots of memory.
- Original paper used Sparse Voxel Octrees.
  - Compact, best fit for GPU?

Crassin’s original paper solved this problem via the use of a Sparse Voxel Octree, which are certainly very compact,

But we weren’t sure it was the best fit for the GPU. There is still lots of walking and pointer chasing in there.

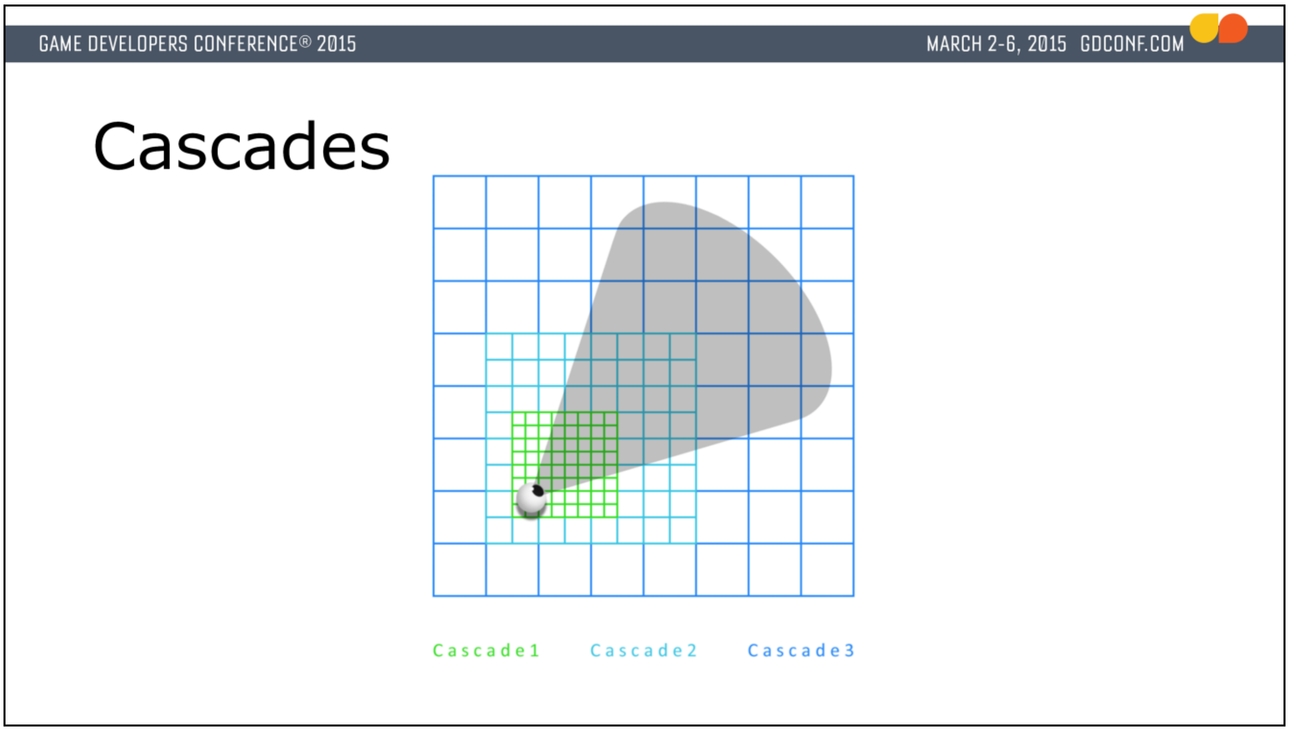
So we weren’t 100% sure it was going to give us the performance we needed.



Scene Representation

- Voxels -> lots of memory.
- Original paper used Sparse Voxel Octrees.
  - Compact, best fit for GPU?
- We use a Voxel Texture Cascade.

So rather than use a Sparse Voxel Octree, we chose instead to use cascades of voxel textures.



Cascades

With texture cascades, we have multiple overlapping textures, with each level being the same resolution,

but with the dimensions of area they cover doubling with each successive level.

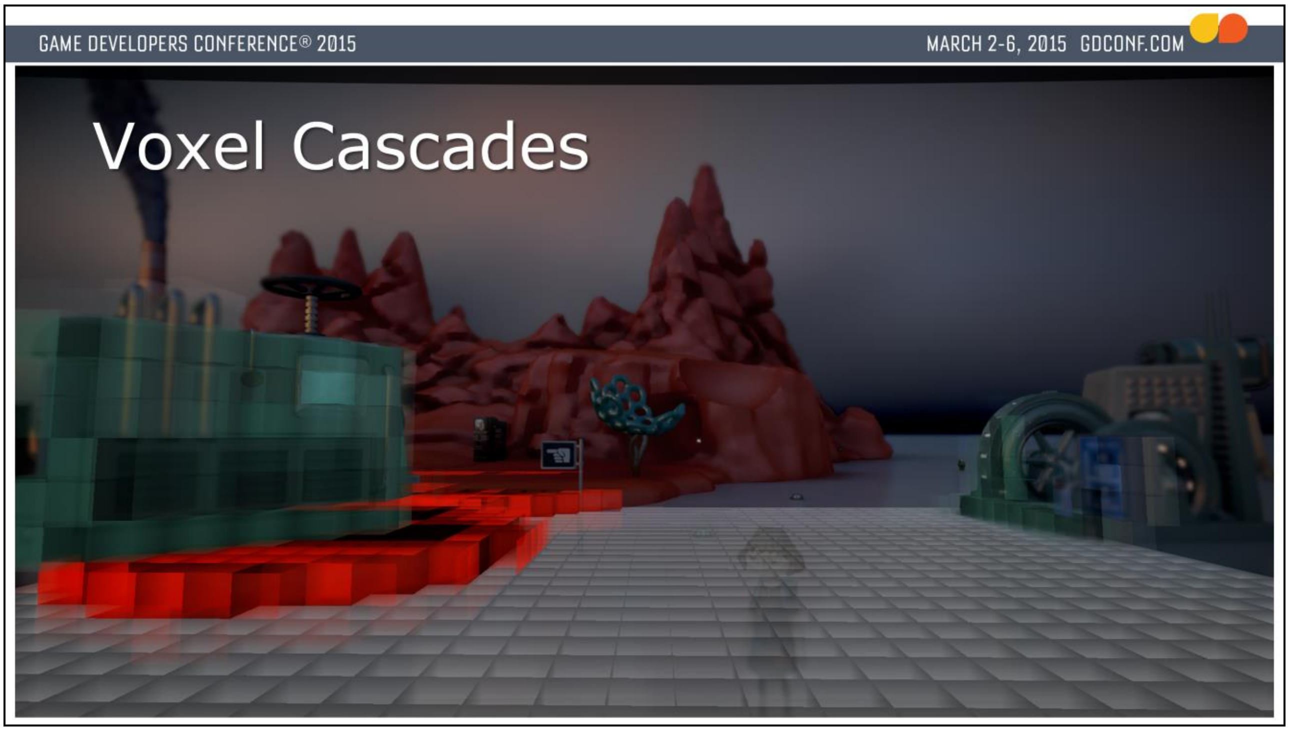
This scheme helps reduce the amount of memory we use, and also provide a natural LODing,

And because everything is just a 3D texture lookup, we felt that that this is also a much more natural fit for the GPU.

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

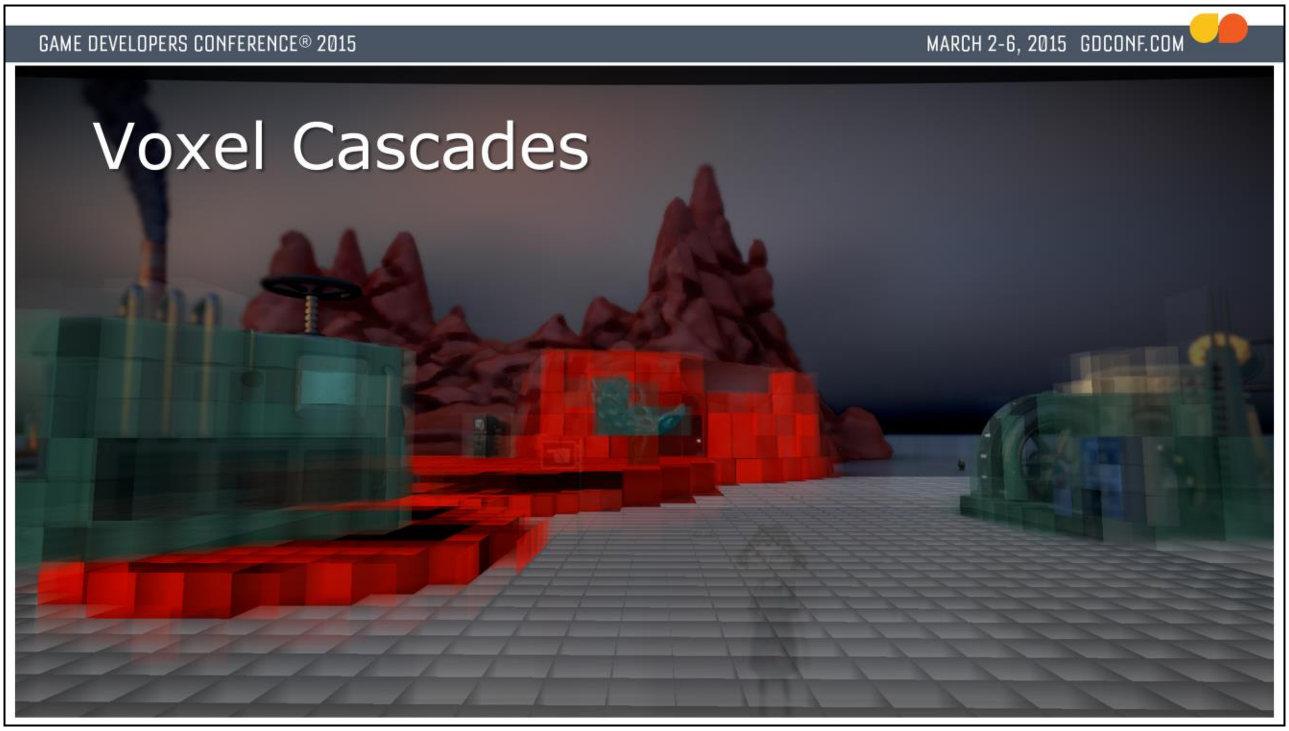
Voxel Cascades

So here's a quick look at what this looks like for us in a debug view on one of our test levels.



Voxel Cascades

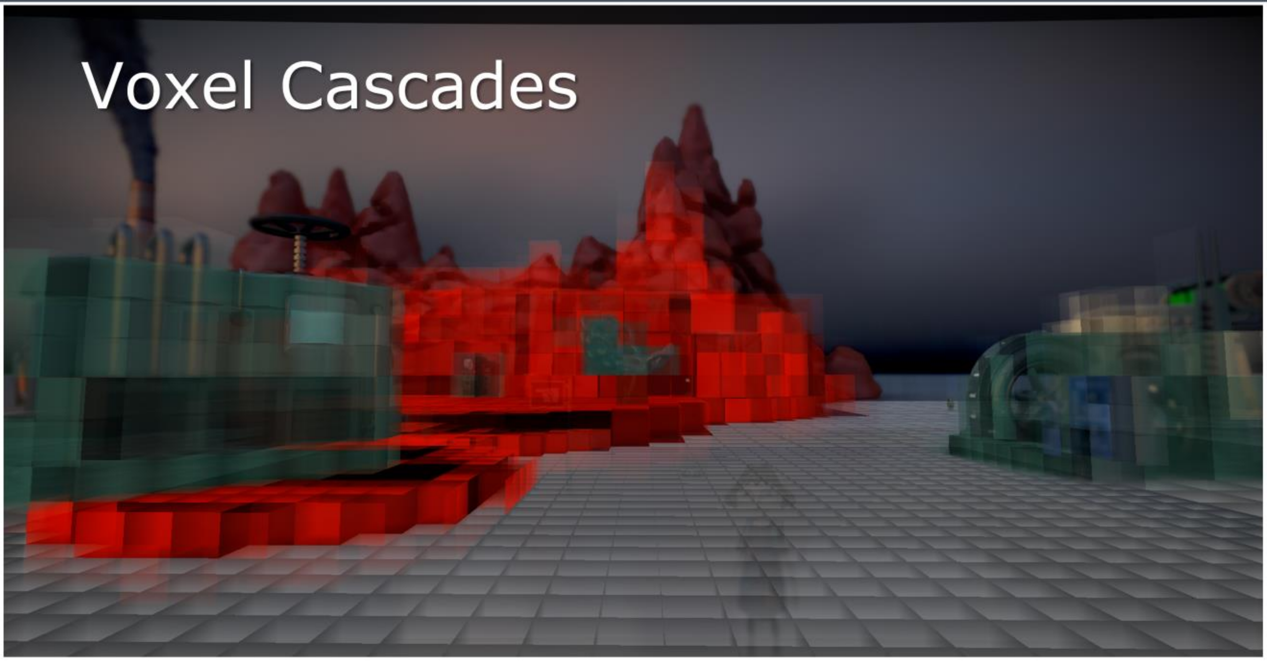
Here's what's represented in the first 32x32x32 cascade level.

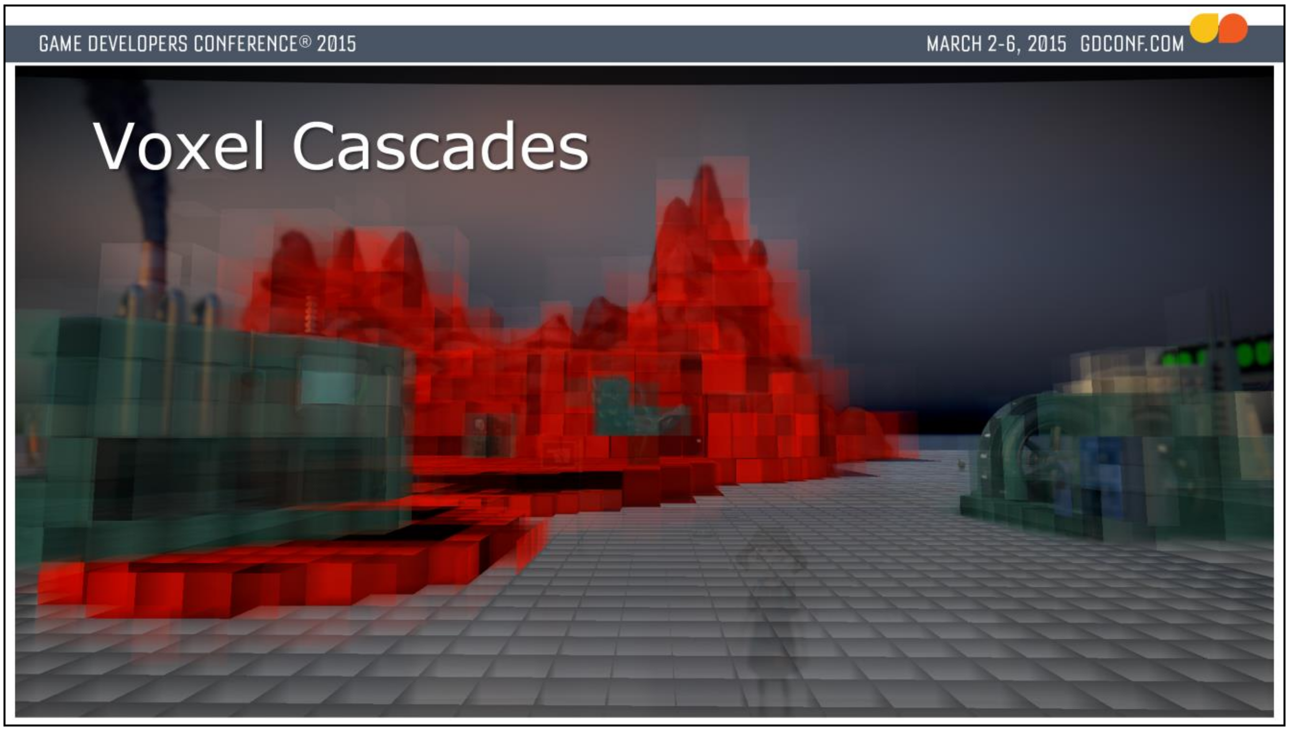


Voxel Cascades

And you can see as I add more cascades

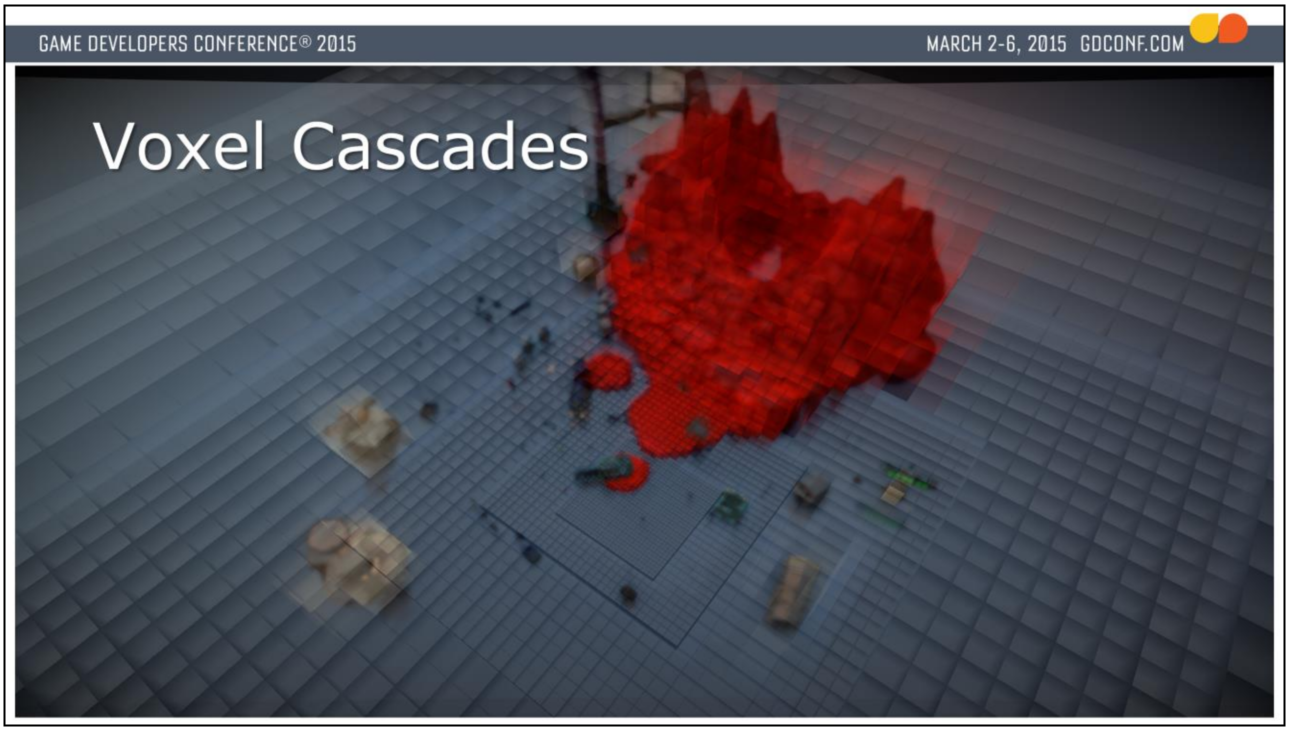
Voxel Cascades




GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Voxel Cascades

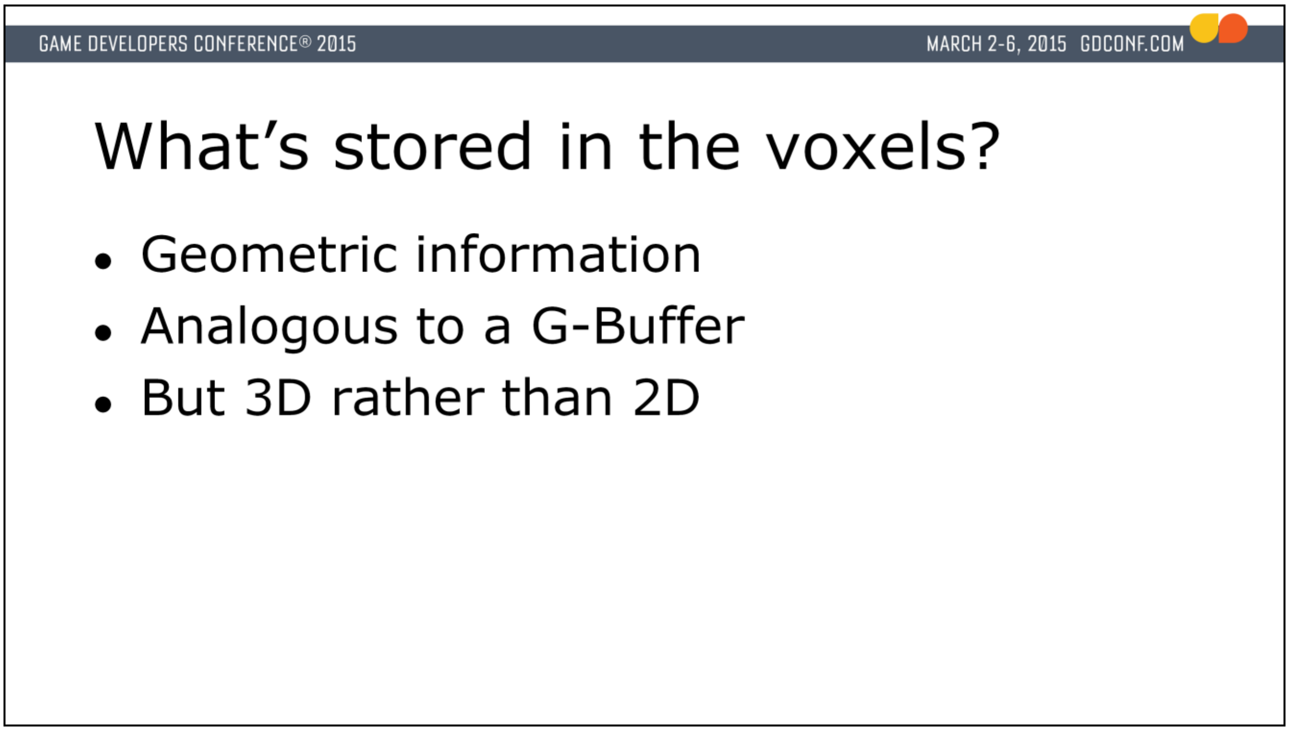
You can see that we get something that sort of approximates the original scene


GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

Voxel Cascades

Looking from above we can see the concentric pattern of the cascades that will be familiar to anyone who's implemented Clip Maps, or Light Propagation Volumes.

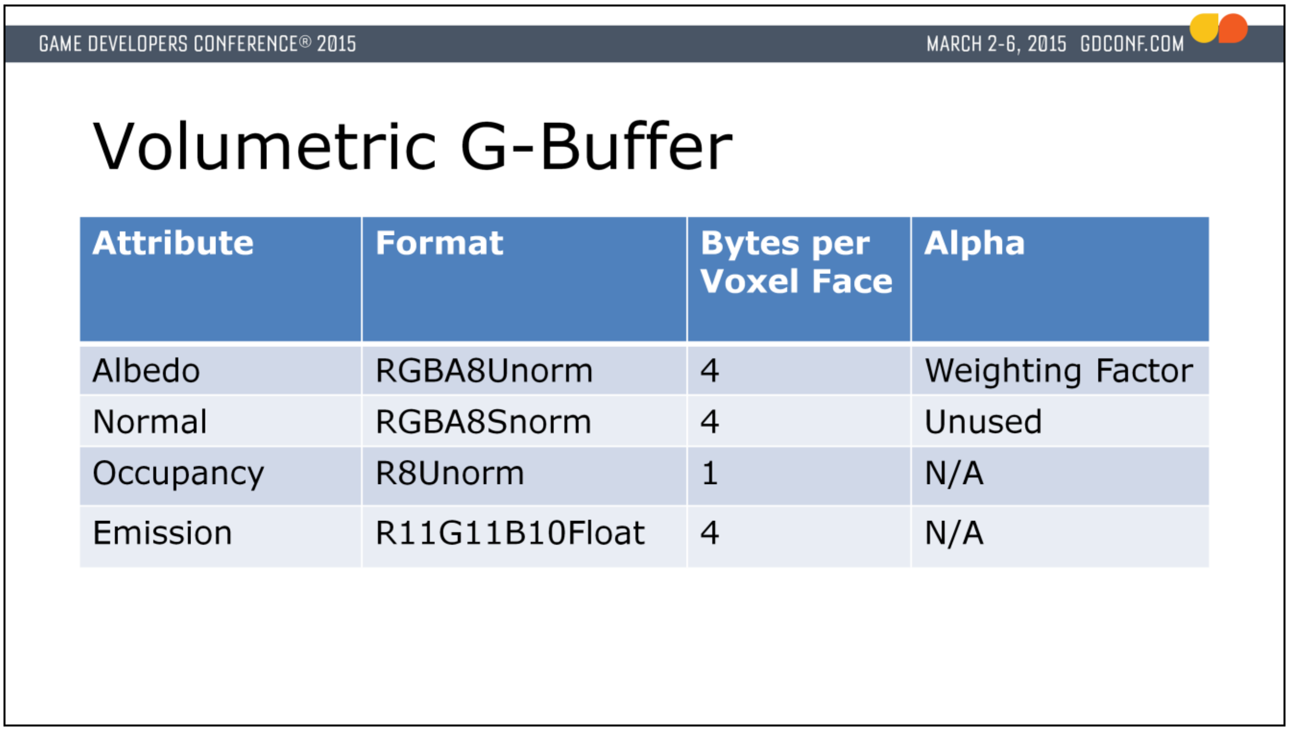


What’s stored in the voxels?

- Geometric information
- Analogous to a G-Buffer
- But 3D rather than 2D

For every voxel in our cascades, we need to store some information about the geometry that’s contained in it.

You can think of this as being analogous to a G-Buffer, but in 3D, rather than in 2D.



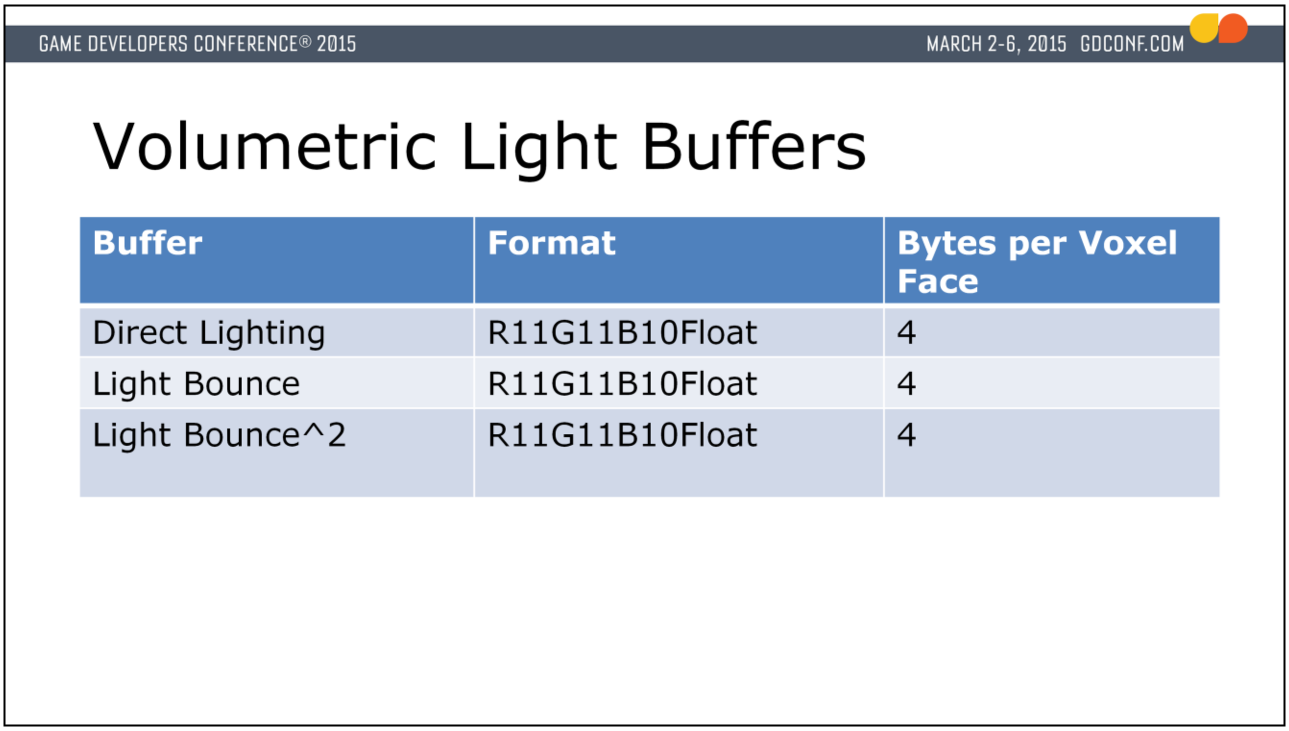
Volumetric G-Buffer

Attribute	Format	Bytes per Voxel Face	Alpha
Albedo	RGBA8Unorm	4	Weighting Factor
Normal	RGBA8Snorm	4	Unused
Occupancy	R8Unorm	1	N/A
Emission	R11G11B10Float	4	N/A

For each attribute that we want to store about our geometry, we have a separate 3D texture.

You can see in the table here, the 4 attributes that we store, which are Albedo, Normal, Occupancy and Emission.

These combine to give us 13 bytes per voxel face. If you're wondering what the "face" part of all this is, don't worry, I'll get to that in just a minute.



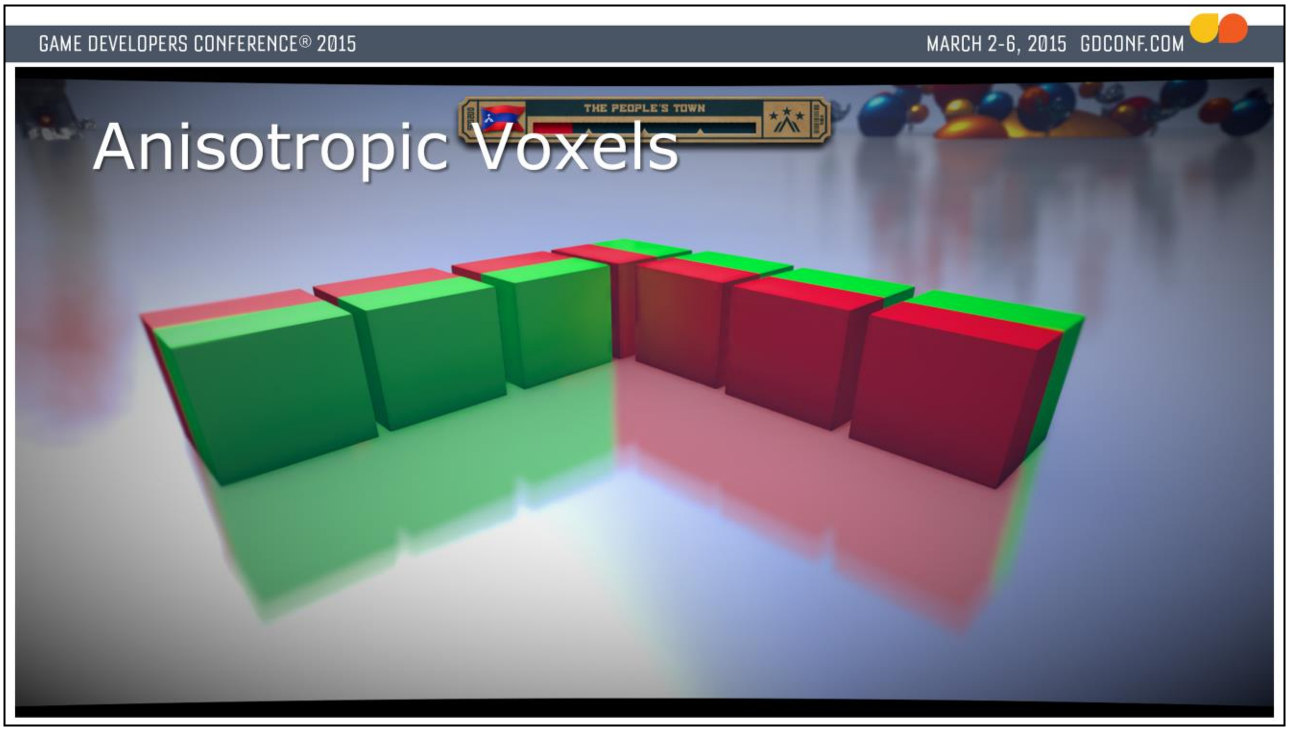
Volumetric Light Buffers

Buffer	Format	Bytes per Voxel Face
Direct Lighting	R11G11B10Float	4
Light Bounce	R11G11B10Float	4
Light Bounce^2	R11G11B10Float	4

Now just Gbuffers aren't enough, we are also going to need space to store the what are the equivalent of our light buffers in a deferred algorithm.

We need a volumetric buffer that stores our direct lighting, our first bounce lighting, and also our second bounce lighting.

Each of these is stored in 11,11,10 floating point format, to give us a good trade off of accuracy vs size.



Anisotropic Voxels

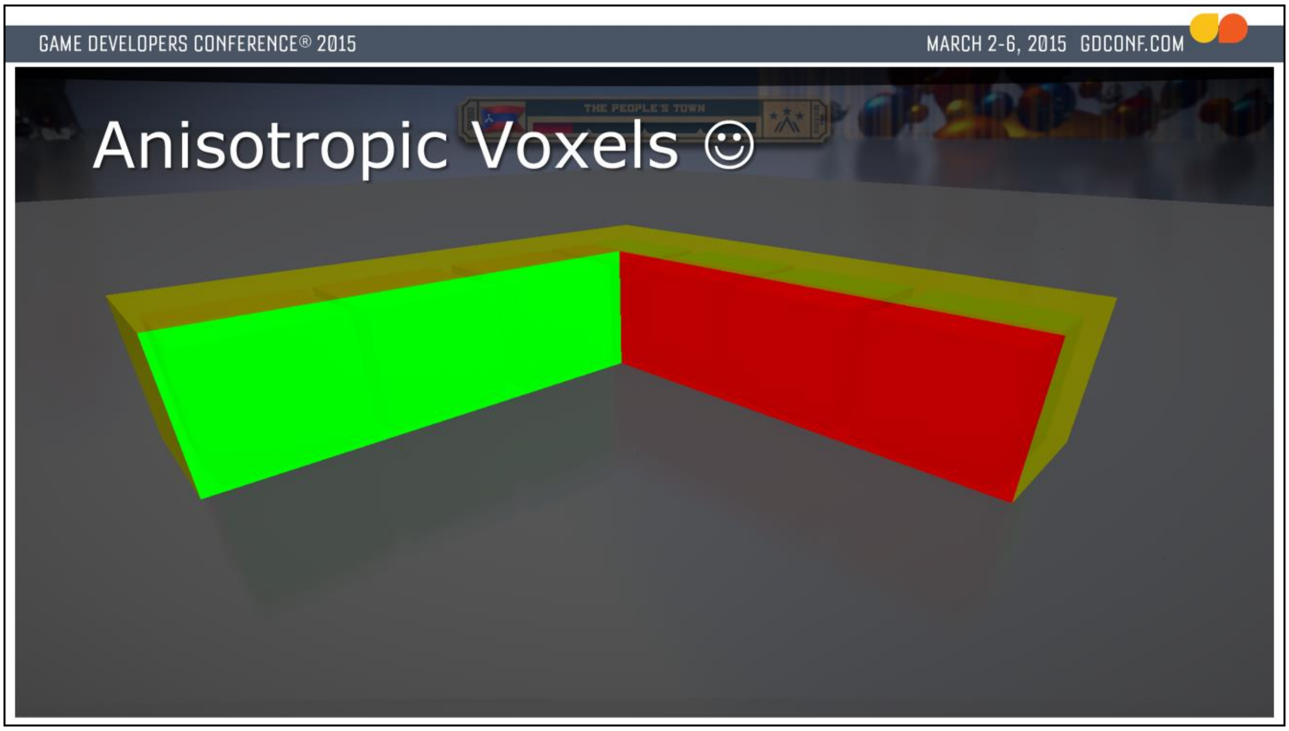
So, it turns out that if you want good quality, then actually you really want to store information not just per voxel but per voxel face.

This means that for each voxel we are actually storing 6 sets of Gbuffer information, and the same for our light buffers.

What this gets us though is quite important, it allows our voxels to be anisotropic.

Take the multicoloured boxes we have here,

which each fit in a single voxel.

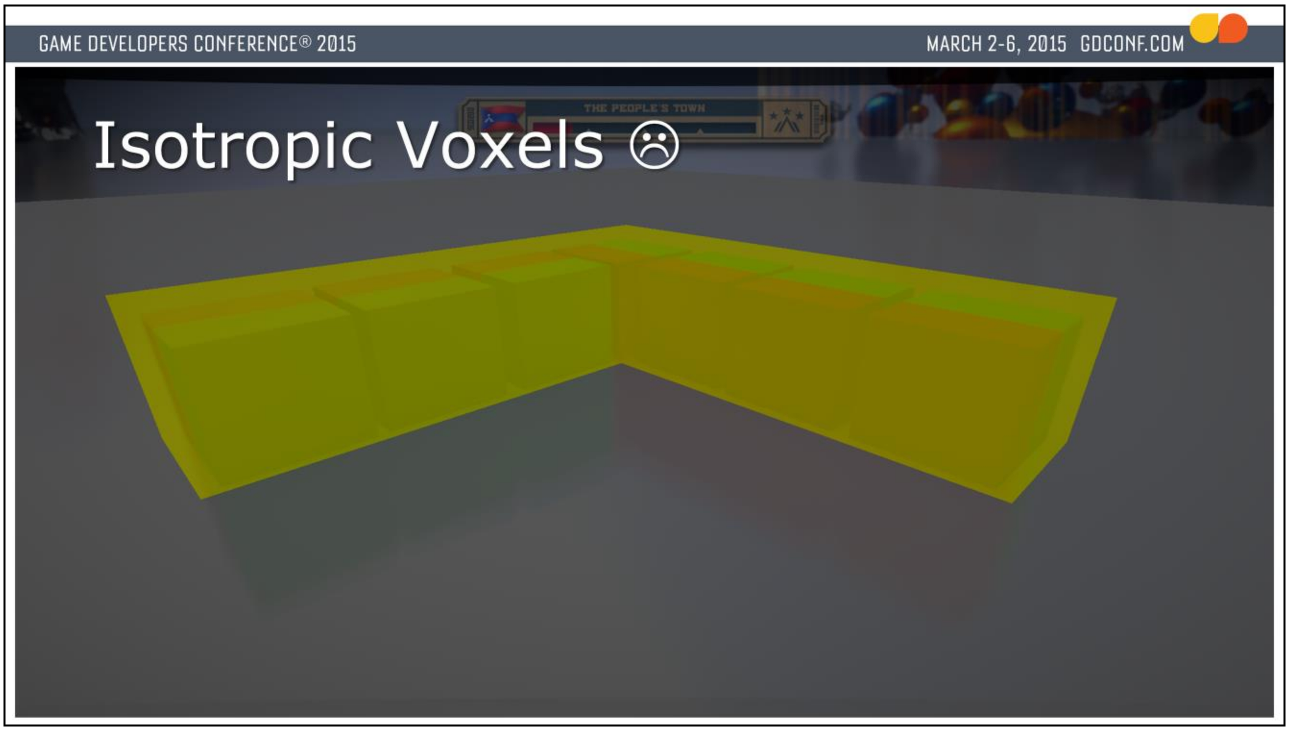

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Anisotropic Voxels ☺

This is how their albedo looks with anisotropic voxels.

If this was to end up being represented as isotropic voxel, i.e with only one albedo, per voxel,

then the best we could do would be to somehow have pick one of the faces for the value of the albedo, or take an average of it, and do the same for our other attributes.



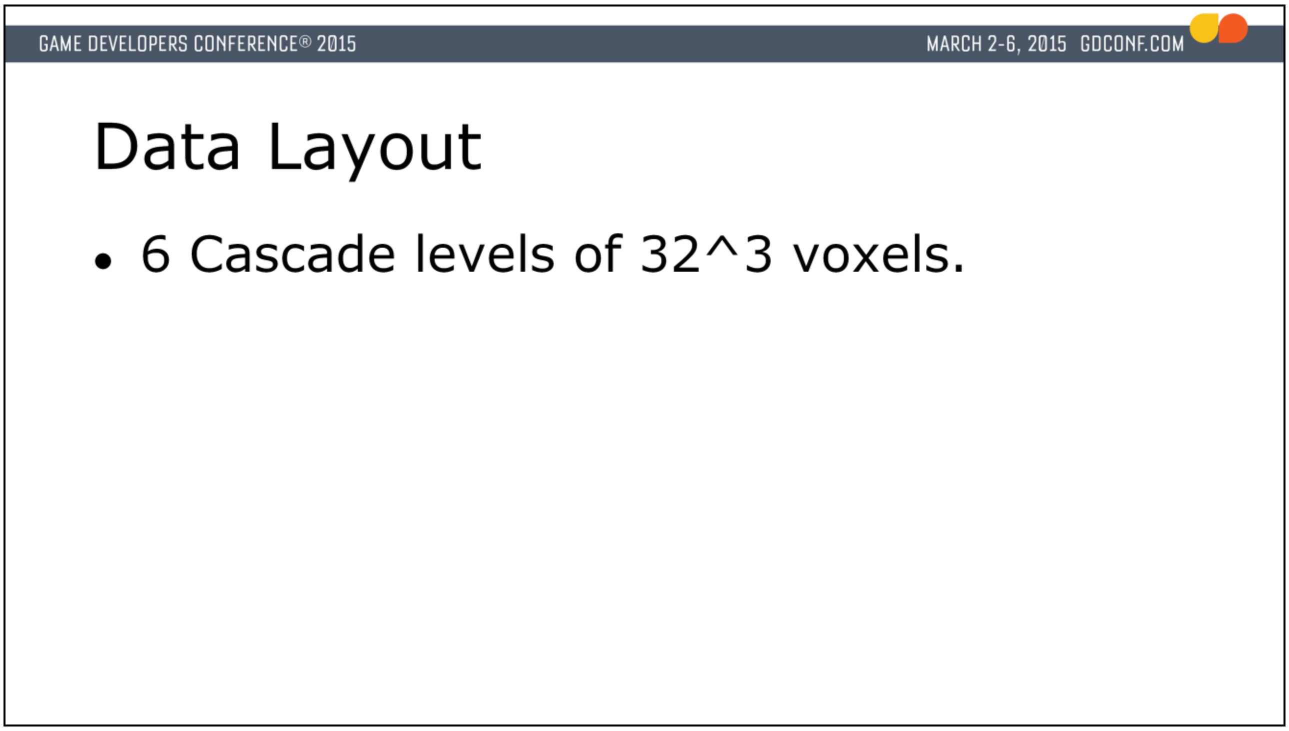
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Isotropic Voxels ☹

So our albedo colour would end up looking like this

Which is obviously not a very accurate representation.

Worse still, if we did this for our light buffers, then if we shone a light on one side of the cube, our buffer for direct lighting would end up telling us that we had some light being emitted from this voxel, even if we were are trying to determine how it looks from the unlit direction.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

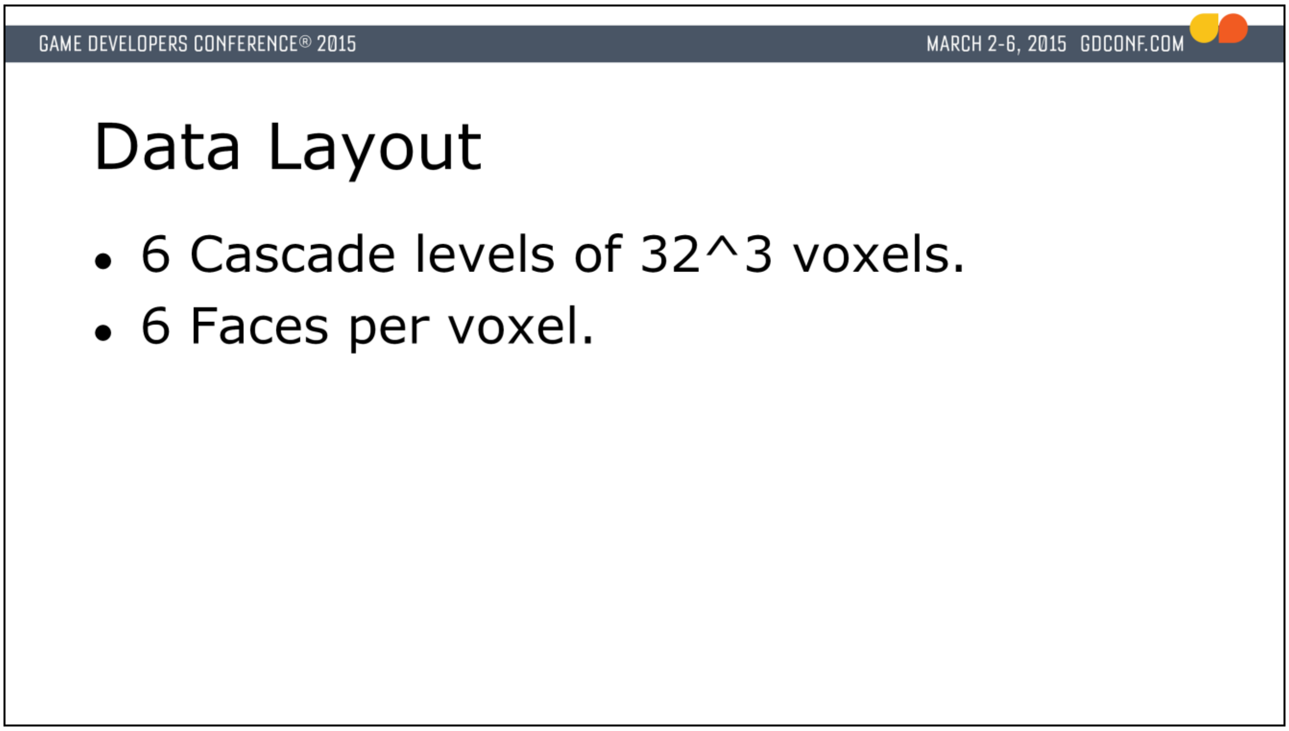
Data Layout

- 6 Cascade levels of 32^3 voxels.

So how do we actually store all of this in memory?

As we saw earlier, each attribute gets it’s own texture, and the same for our light buffers.

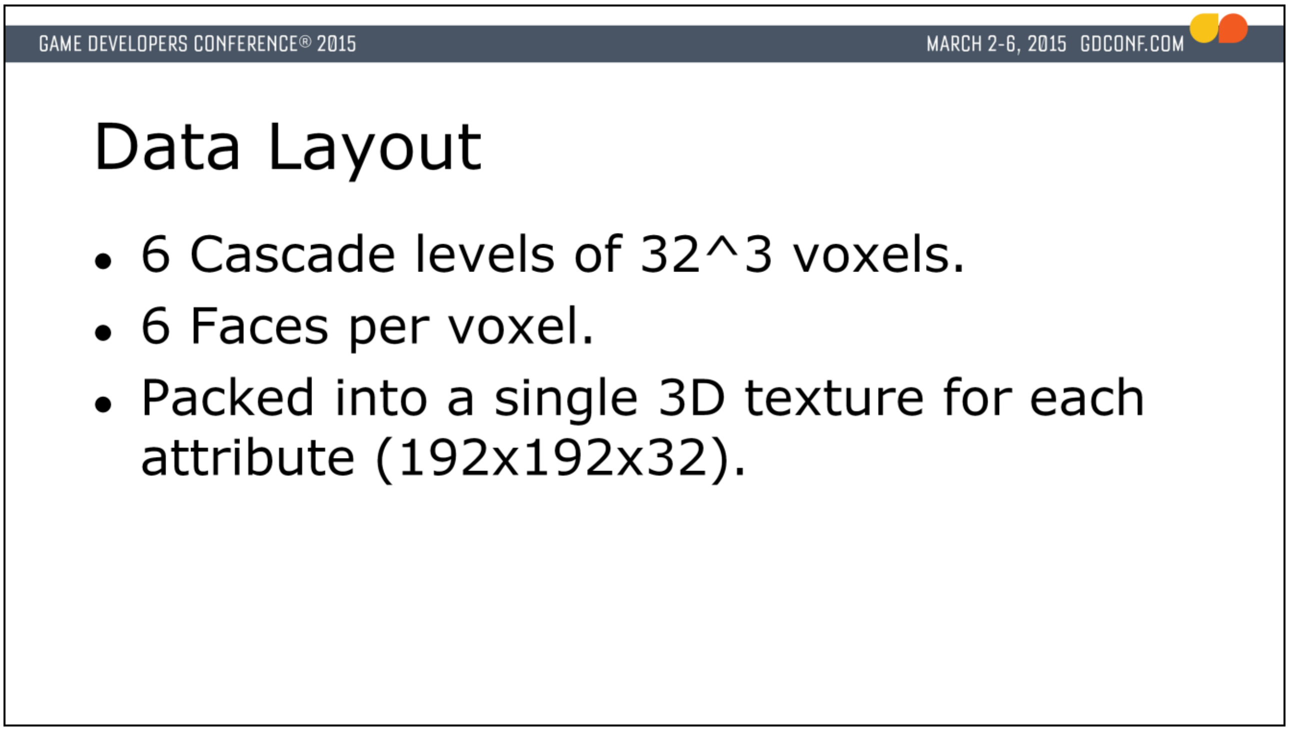
We have 6 cascade levels, each of which is 32^3 voxels, which allows us to cover objects a fair way off into the distance.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Data Layout

- 6 Cascade levels of 32^3 voxels.
- 6 Faces per voxel.

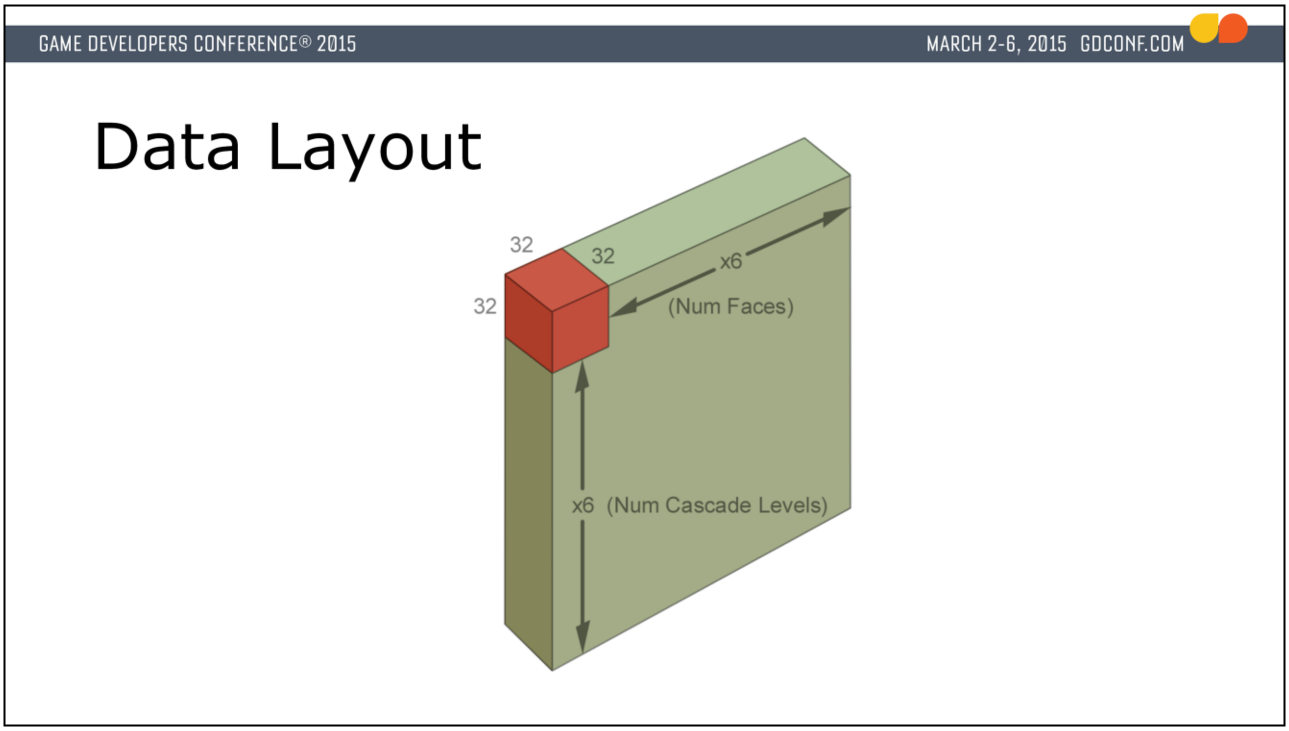
And as I just described, each voxel also needs to store values for each of it's 6 faces.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Data Layout

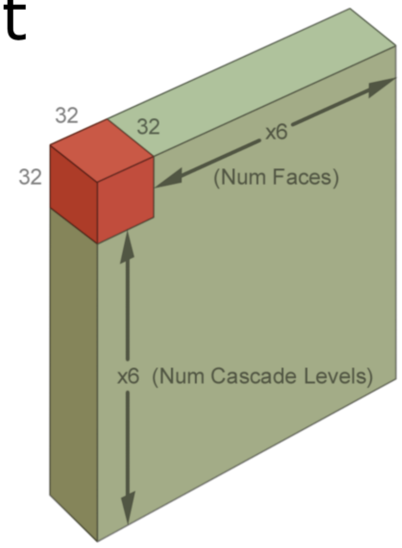
- 6 Cascade levels of 32^3 voxels.
- 6 Faces per voxel.
- Packed into a single 3D texture for each attribute (192x192x32).

And so we end up packing all 6 cube faces for our anisotropic voxels and all 6 cascade levels into a single 3D texture per attribute.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Data Layout



Within each texture we need to tile repeated blocks of 32^3 voxels for each of our 6 faces in X.

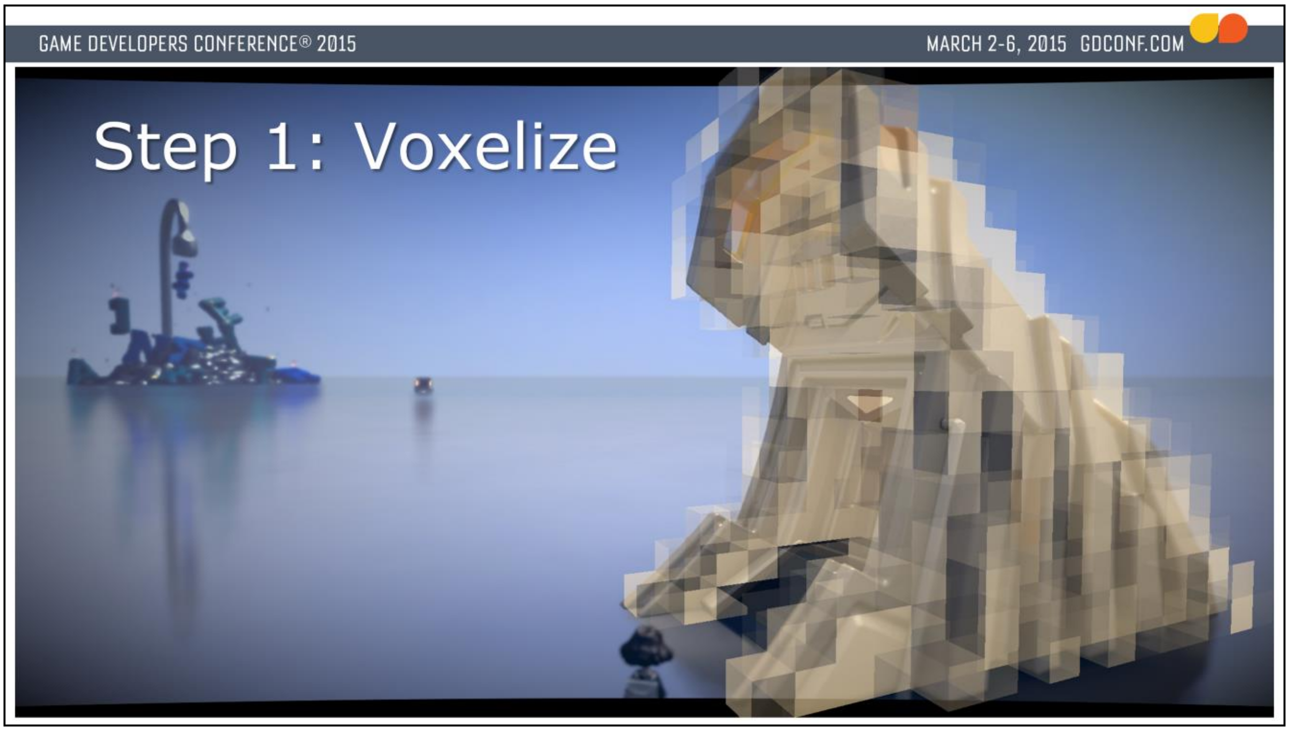
And we also tile our blocks in Y for our cascade levels.

This setup allows us to easily do trilinear interpolation between our voxels,

but we do have to be careful when we sample to always clamp to the edge of our cascades levels to avoid bleeding from other faces or levels..

How do we use this to get GI?

So now we know what cone tracing is,
and how we’re roughly going to store our scene,
lets look at how to use that to get some global illumination



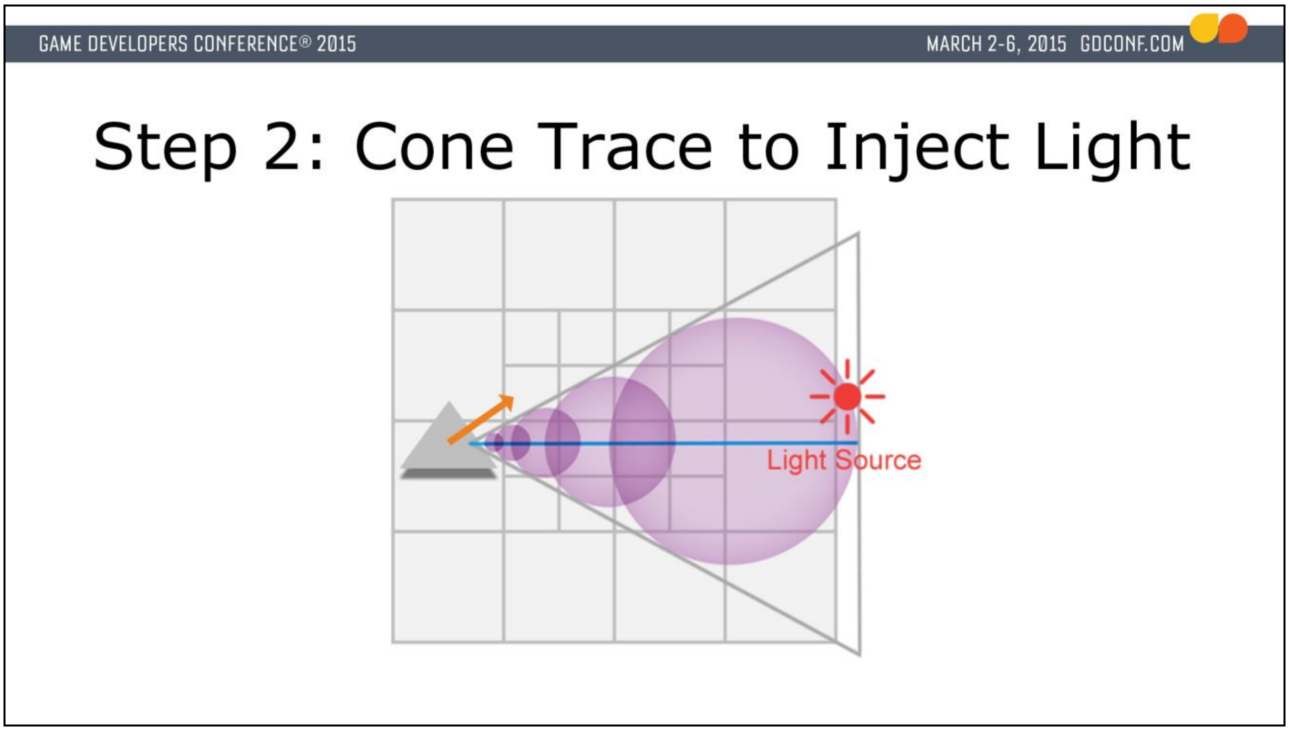
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Step 1: Voxelize

We need to start off by voxelizing our scene into our Voxel Texture Cascade.

This gives us a scene representation that we can begin to work with.

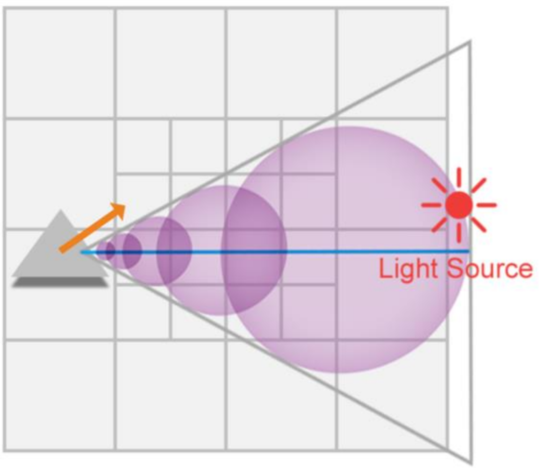
When we voxelize we're essentially building our Volumetric Gbuffer.



GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

Step 2: Cone Trace to Inject Light



Then we need to inject lighting into our volumetric light buffers.

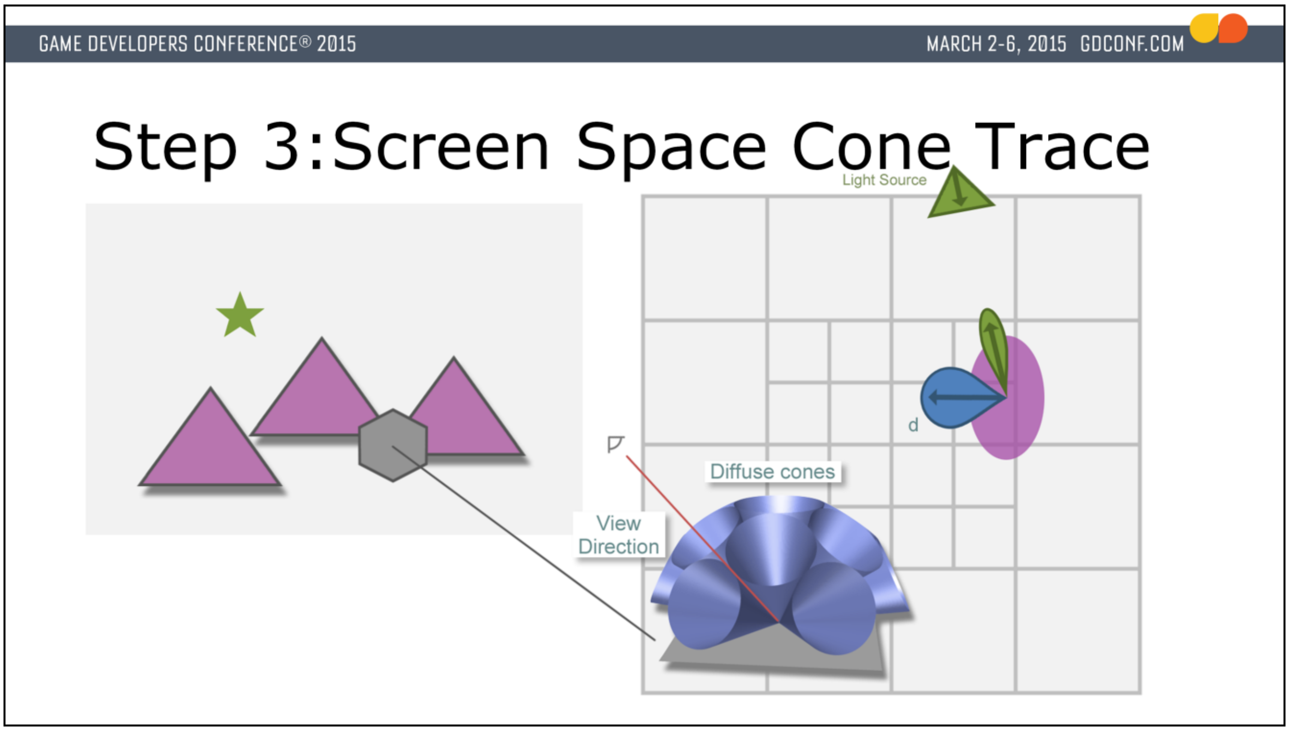
We actually do this by tracing several cones for each voxel.

In the simple case where we are just doing one bounce, and injecting light from the sky,

then you can think of this as just tracing a cone to determine sky occlusion in a certain direction.

As I described before , by looking at the occupancy for each sample as we move along the cone, we can get a fairly good estimate for how much we can see of the sky in that direction.

We will do this for all of the levels of our cascade, although we do stagger this update so that we only update one level per frame.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Step 3:Screen Space Cone Trace

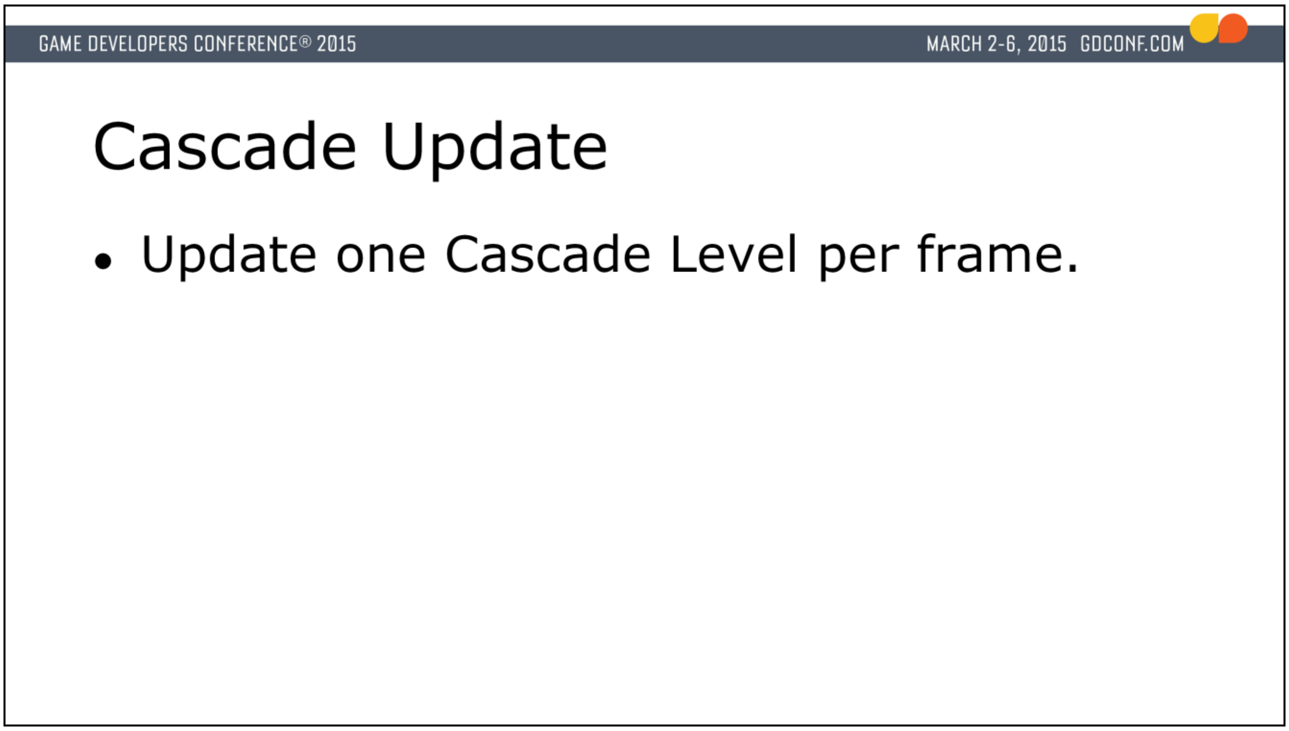
Now that we have filled our direct lighting buffer with illumination information,

we can now trace several cones from the world space position of each pixel to get our indirect lighting.

Generating and Updating Cascades

So now I'll take a closer look at some of the details of how we will go about

Generating and updating the data that will be in the cascades.



Cascade Update

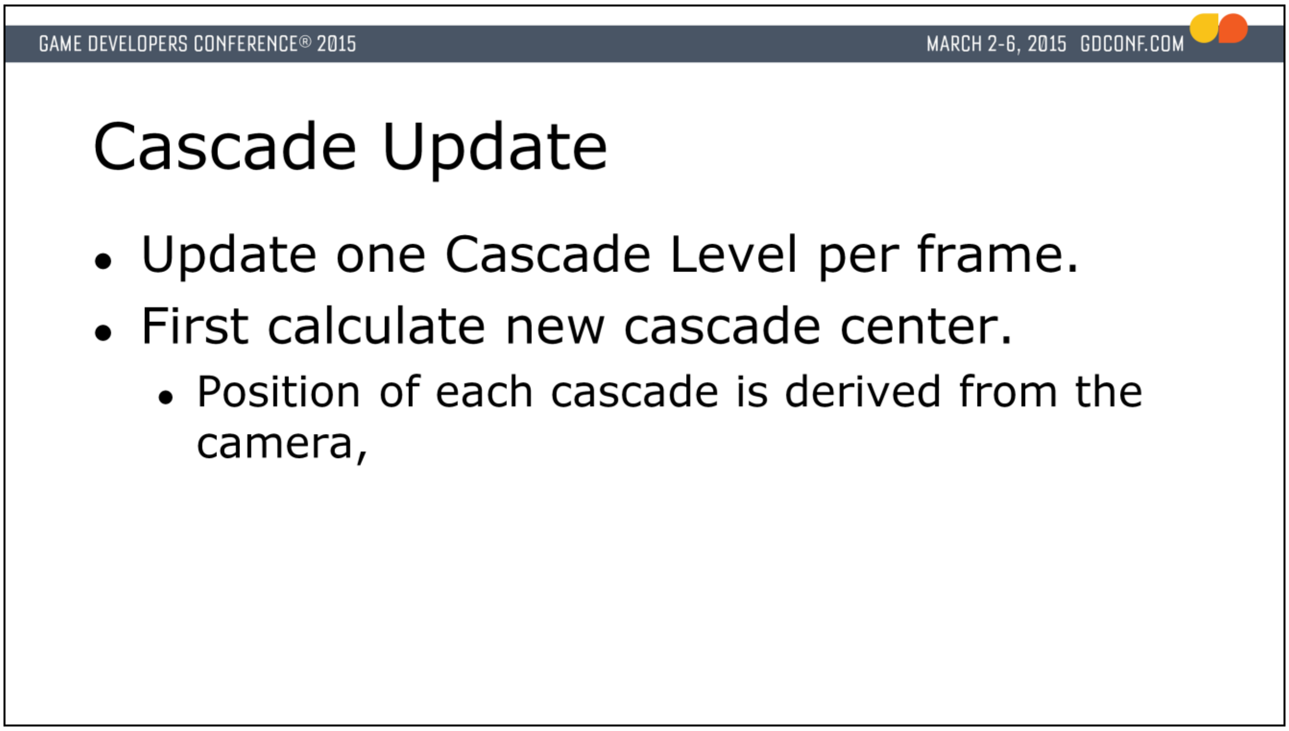
- Update one Cascade Level per frame.

We only update one cascade level per frame for speed,

but we bias which level is chosen to ensure more frequent updates for the higher detail cascade levels nearest to the camera.

So the closest cascade level is updated every 2 frames, the next level every 4 frames, and so on.

This helps to ensure that the lighting closest to the action is updated at a reasonable speed.

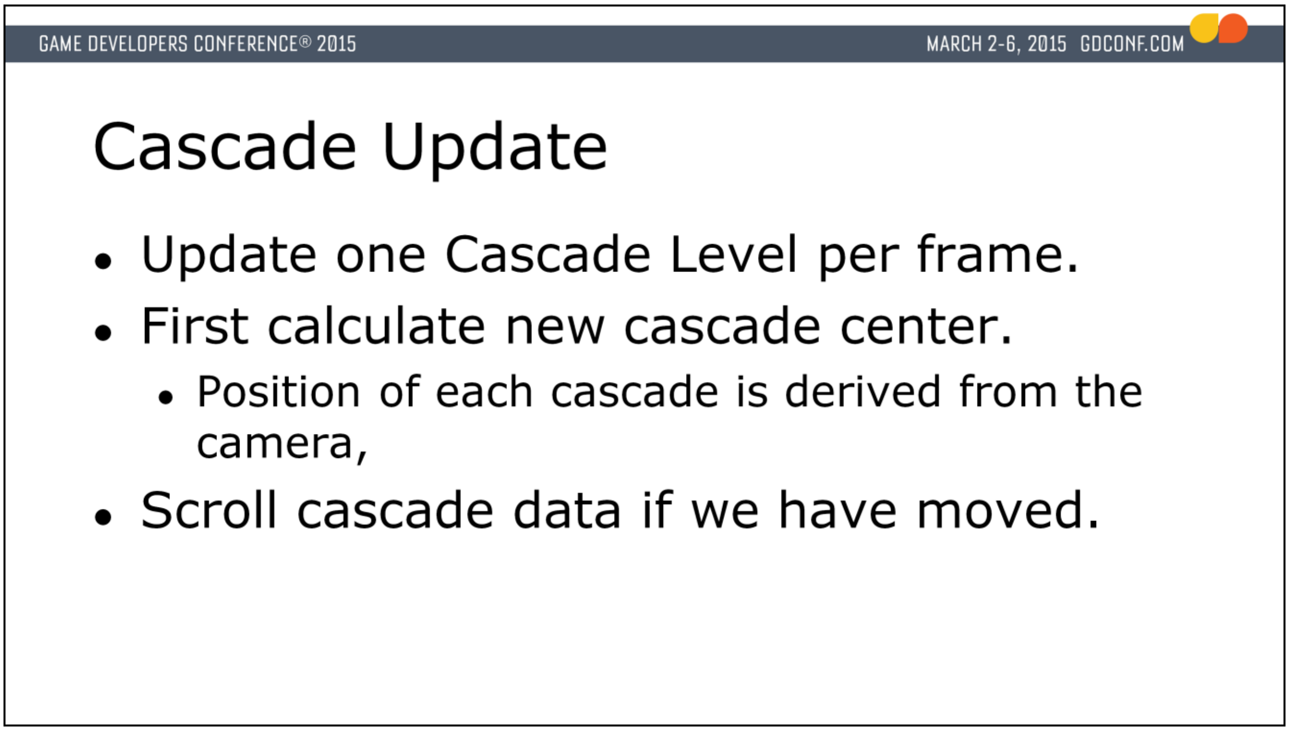


Cascade Update

- Update one Cascade Level per frame.
- First calculate new cascade center.
  - Position of each cascade is derived from the camera,

The first thing we need to do when updating our cascade level for this frame is to calculate it's new center if the viewer has moved..

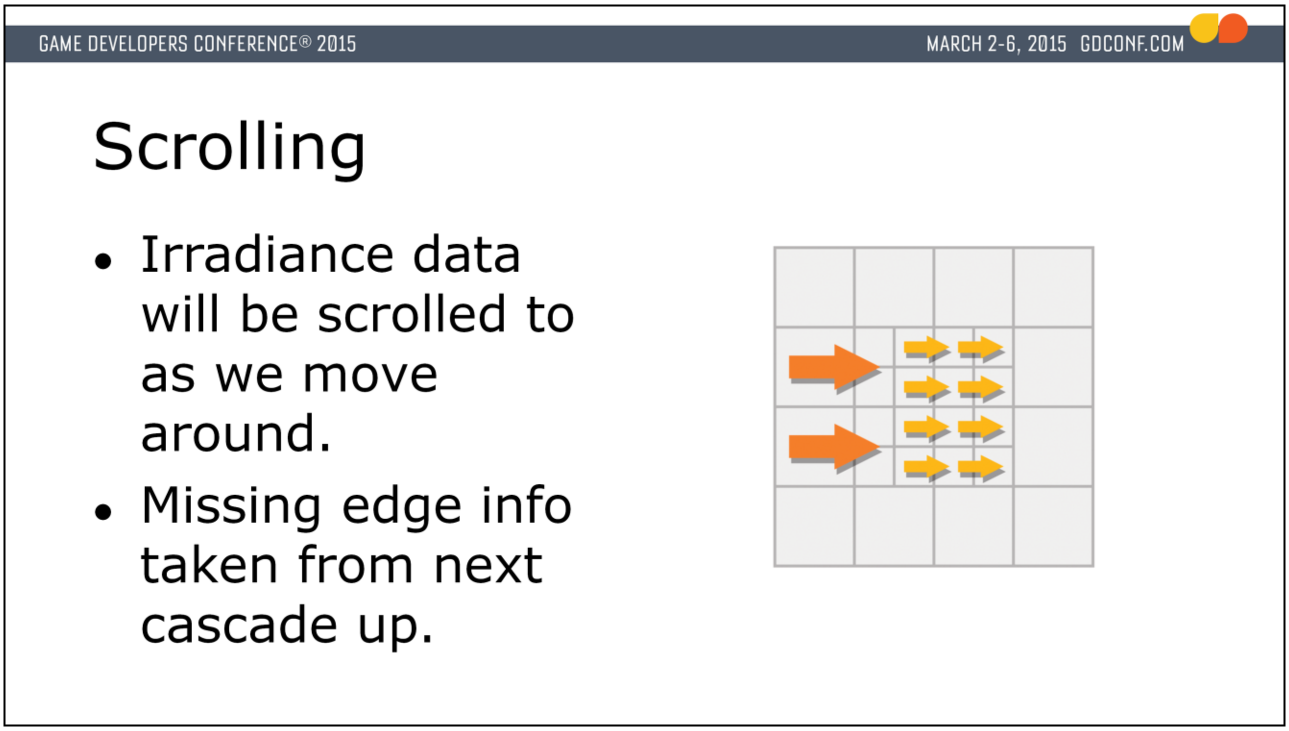
This is of course based on the position of the camera.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Cascade Update

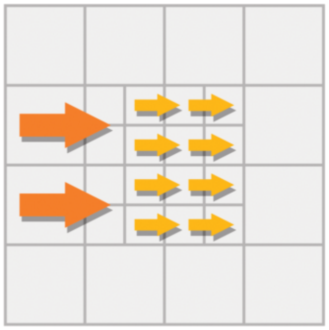
- Update one Cascade Level per frame.
- First calculate new cascade center.
  - Position of each cascade is derived from the camera,
- Scroll cascade data if we have moved.

We must then scroll any data we have in our cascade level if we have moved,
And we have do this for both our volumetric light buffers and gbuffer data.



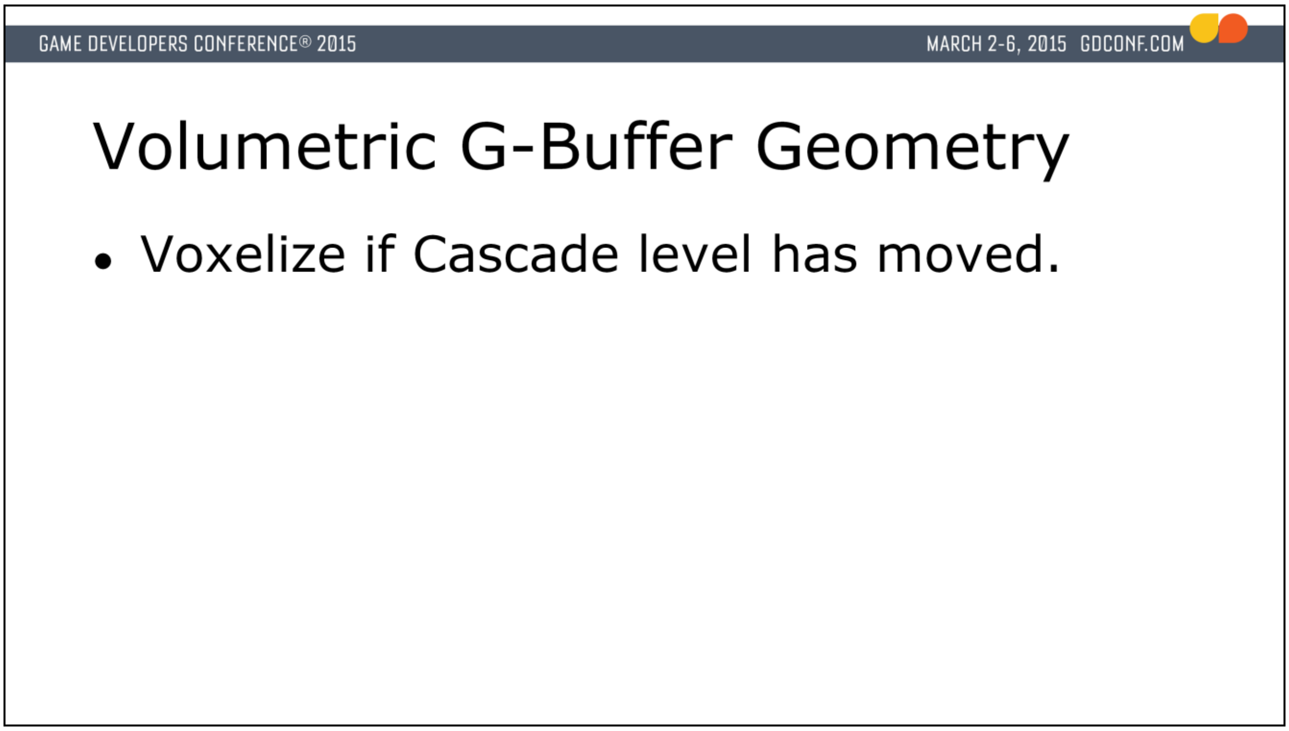
Scrolling

- Irradiance data will be scrolled to as we move around.
- Missing edge info taken from next cascade up.



As a consequence of scrolling our data around we will find that we don't have data for certain edges in our current cascade level.

For our irradiance data, we pull this data from the next cascade level down to give ourselves what is effectively a mipmapped approximation of the lighting on that edge.

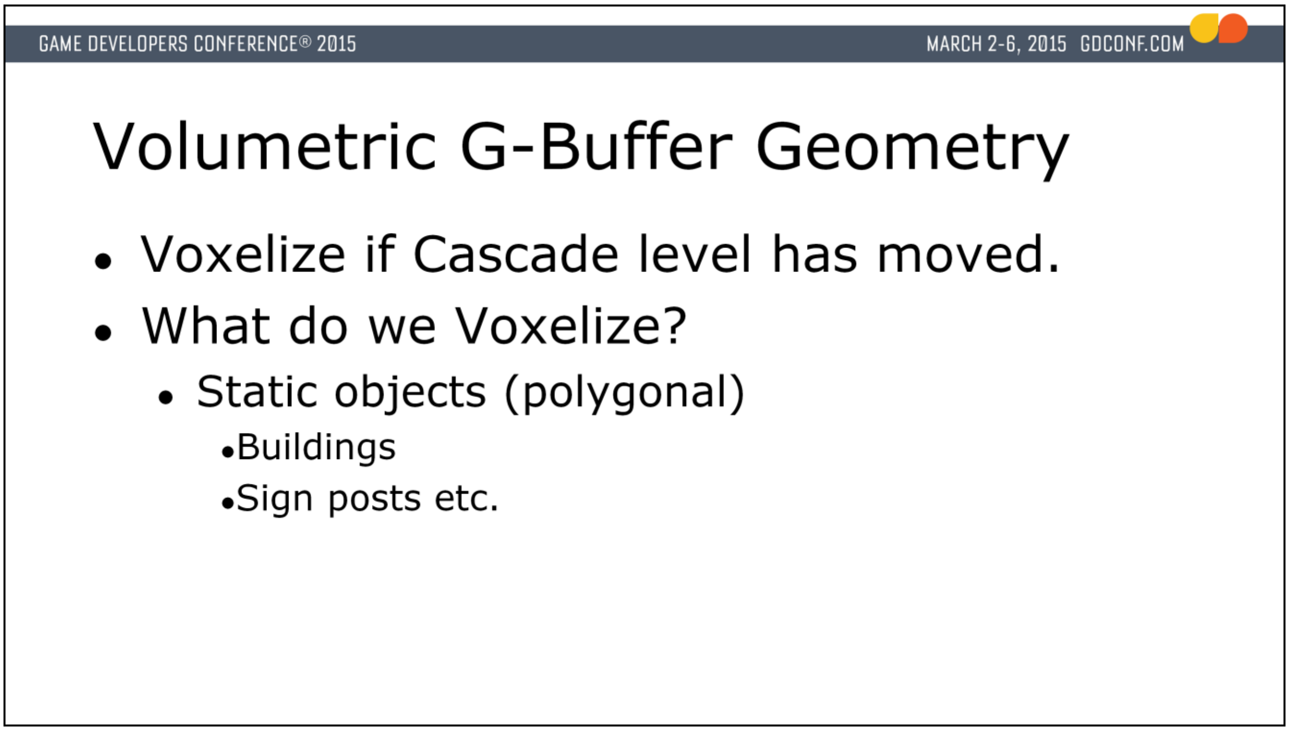


Volumetric G-Buffer Geometry

- Voxelize if Cascade level has moved.

If the cascade level has moved then we also have to voxelize any new geometry at the edges

As well as any geometry that has just appeared or been changed.



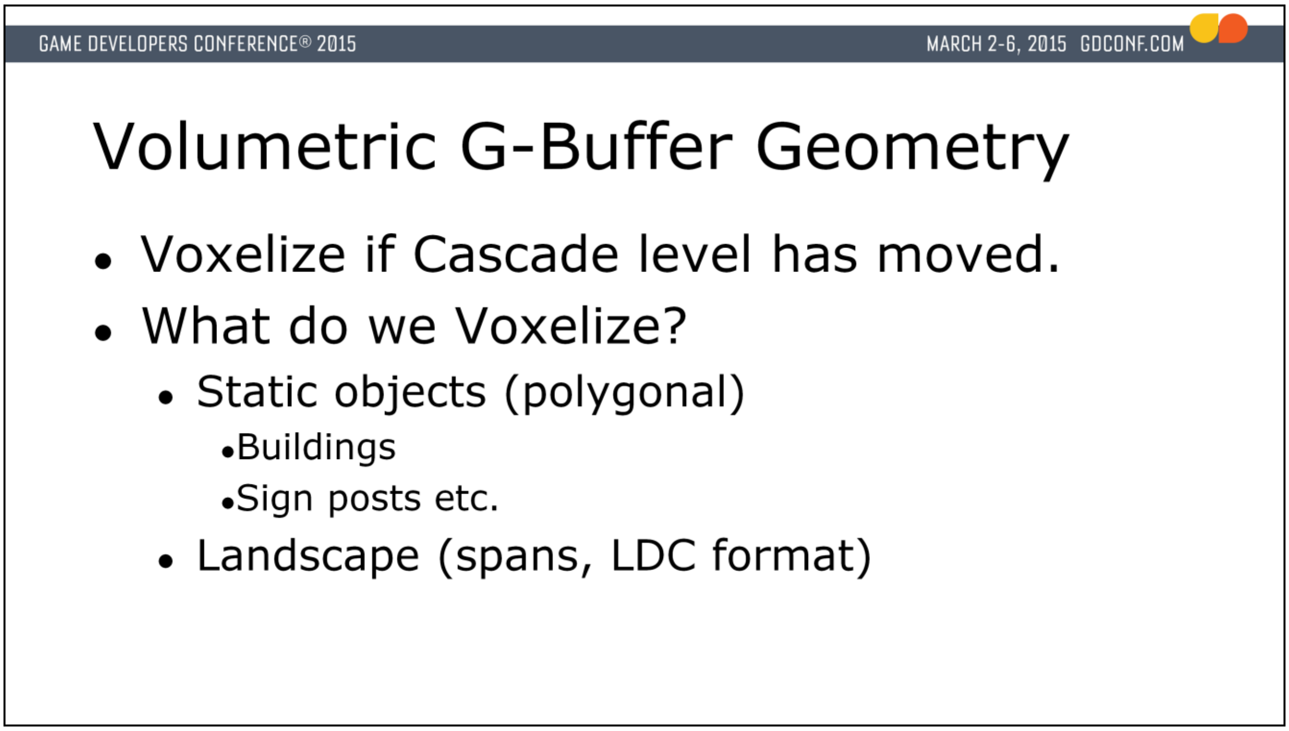
Volumetric G-Buffer Geometry

- Voxelize if Cascade level has moved.
- What do we Voxelize?
  - Static objects (polygonal)
    - Buildings
    - Sign posts etc.

For us, there are two main sources of data for this geometry.

Firstly, we have our static objects, such a buildings, sign posts, that are part of our city that have a traditional polygonal representation,

For these just use a low LOD version of the our normal rendering geometry.

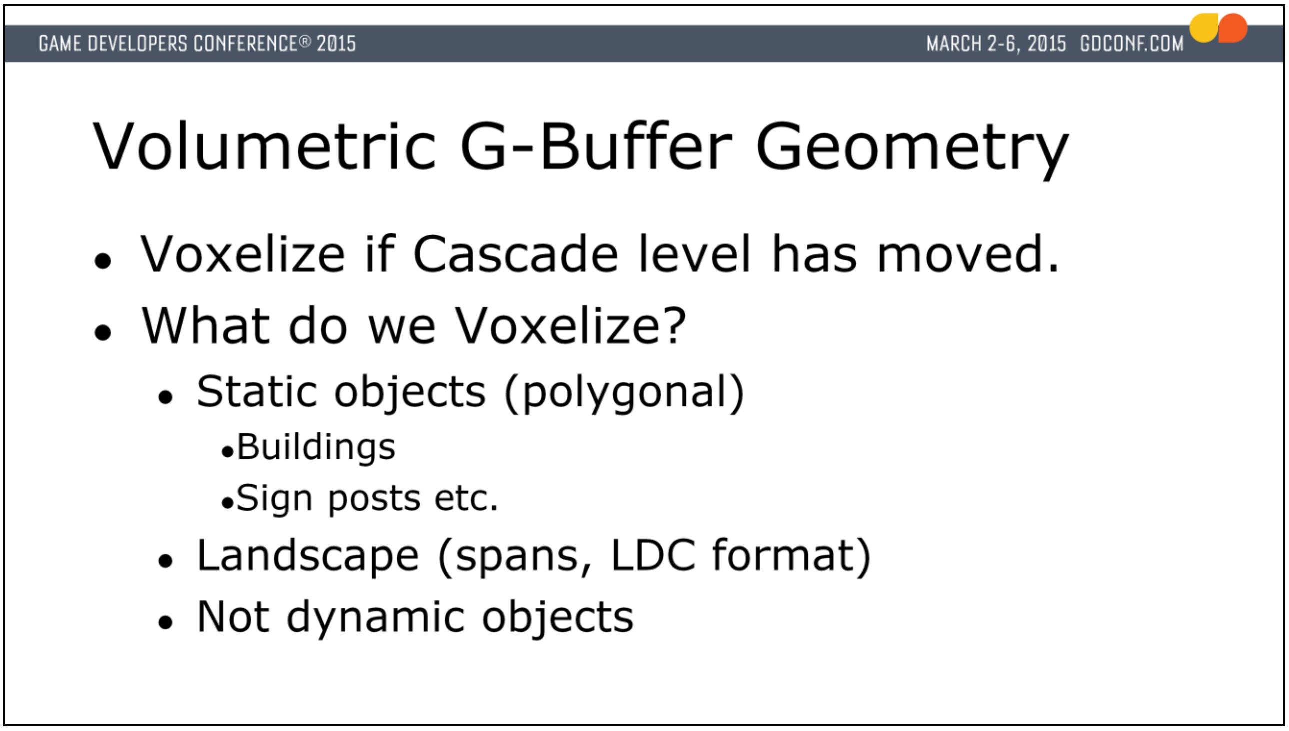

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Volumetric G-Buffer Geometry

- Voxelize if Cascade level has moved.
- What do we Voxelize?
  - Static objects (polygonal)
    - Buildings
    - Sign posts etc.
  - Landscape (spans, LDC format)

Secondly, we also have our landscape, which is user modifable, but still relatively static.

This is stored internally in a sort of span list representation, that I'll talk a little bit about later.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

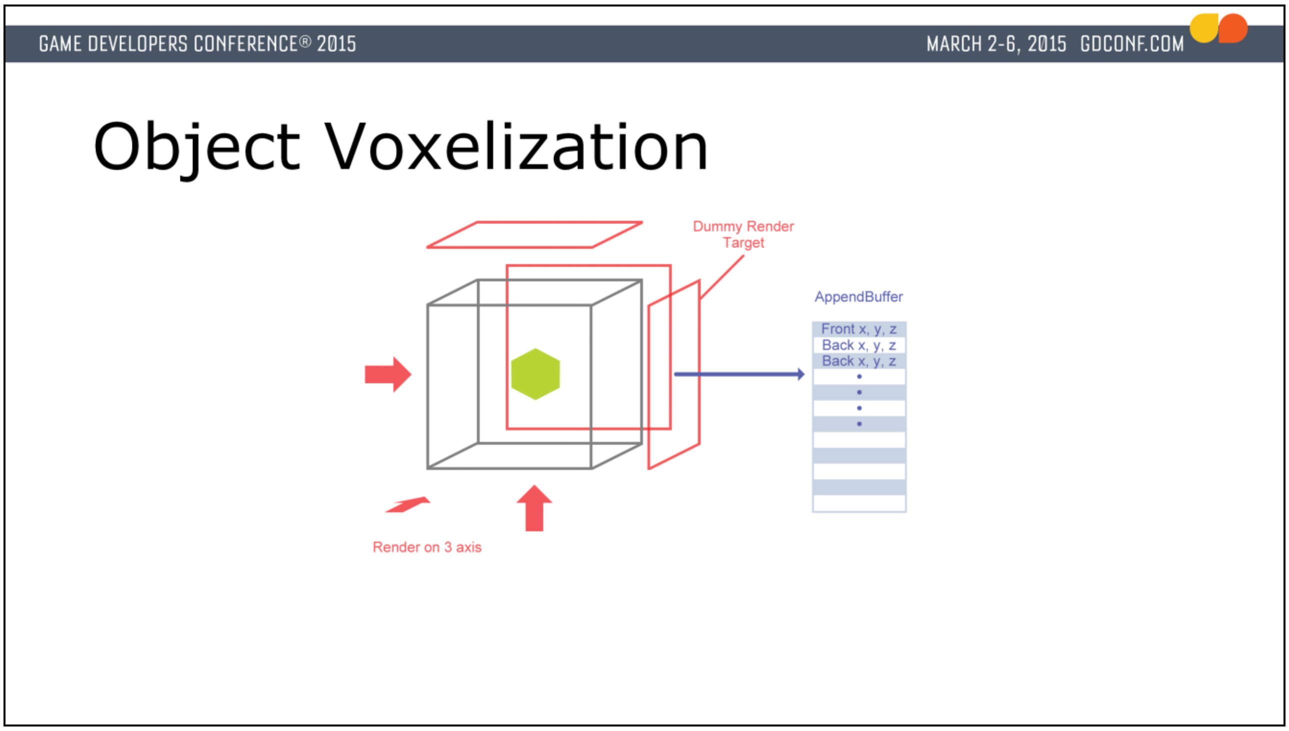
Volumetric G-Buffer Geometry

- Voxelize if Cascade level has moved.
- What do we Voxelize?
  - Static objects (polygonal)
    - Buildings
    - Sign posts etc.
  - Landscape (spans, LDC format)
  - Not dynamic objects

It’s worth noting that we don’t use dynamic objects such as characters and vehicles to populate our Volumetric G-Buffers, as they would force us to do a lot of repeated voxelization as they move around.

Voxelization of Polygonal Objects

So lets say we have an object, how do we go about voxelizing it?

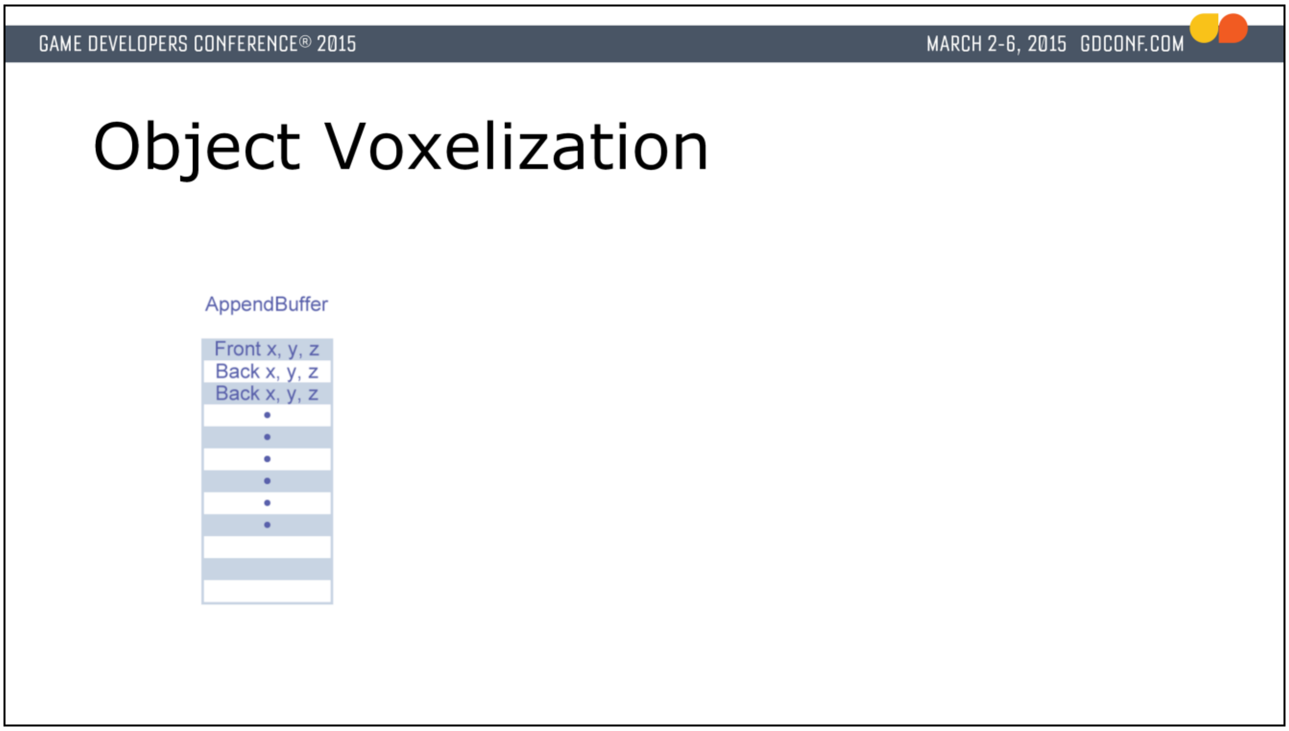


Object Voxelization

Objects are voxelized using the hardware rasterizer by performing a draw call for each axis.

This is not a traditional render however as the z test and back face culling are turned off and we don't actually write any pixels via the ROPS.

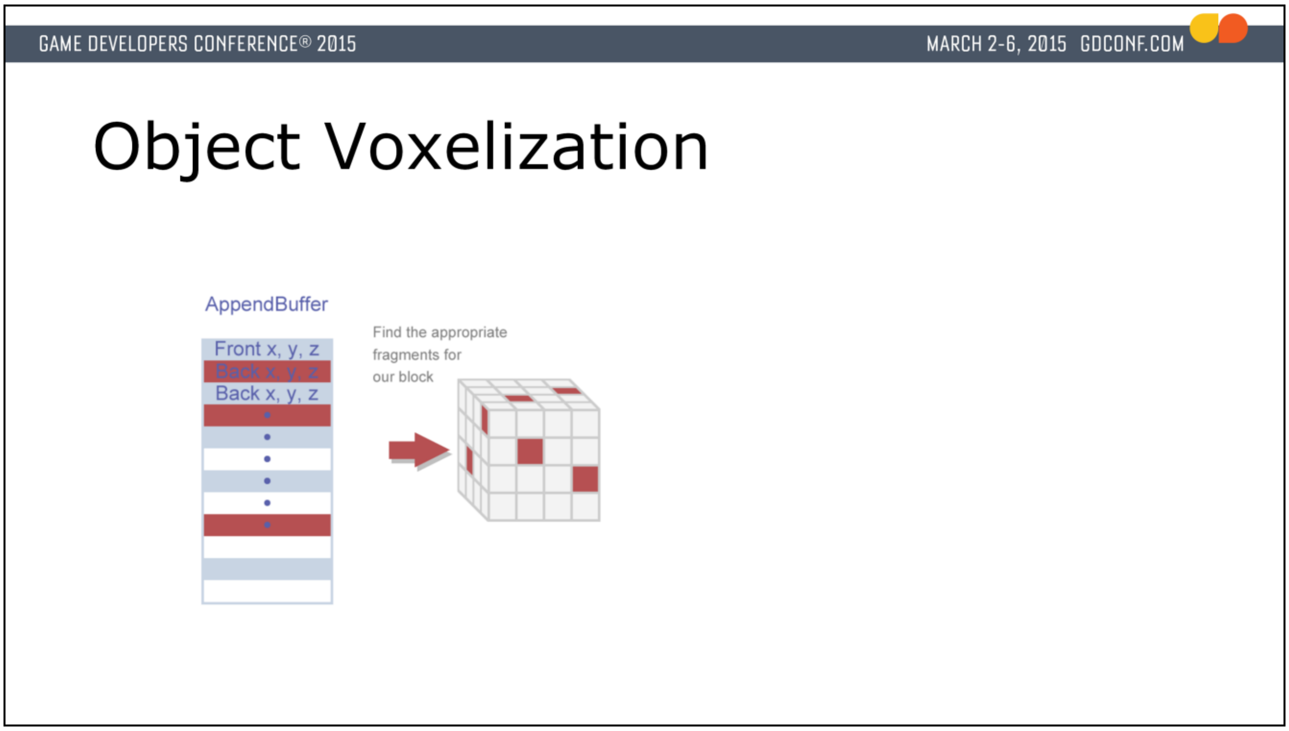
What we do do however is to build up a list of entry and exit points to our geometry, along with their associated depth, pixel coordinates, and material attributes



Object Voxelization

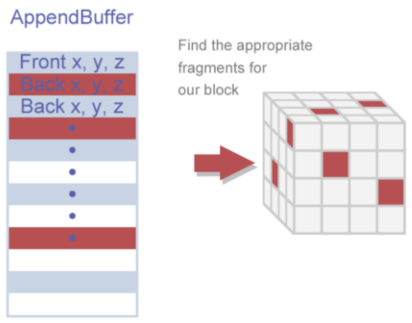
We can then perform some additional processing of this list, to help us determine what should be recorded in each voxel.

.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

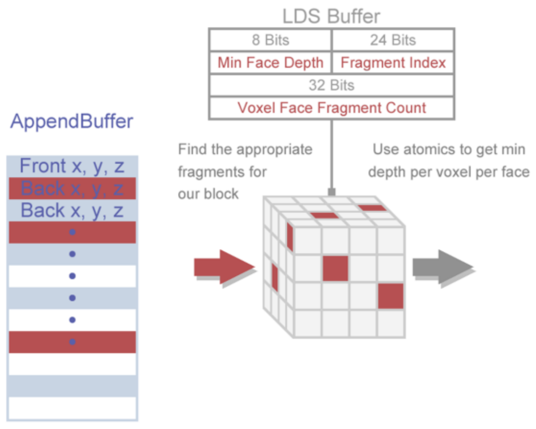
Object Voxelization



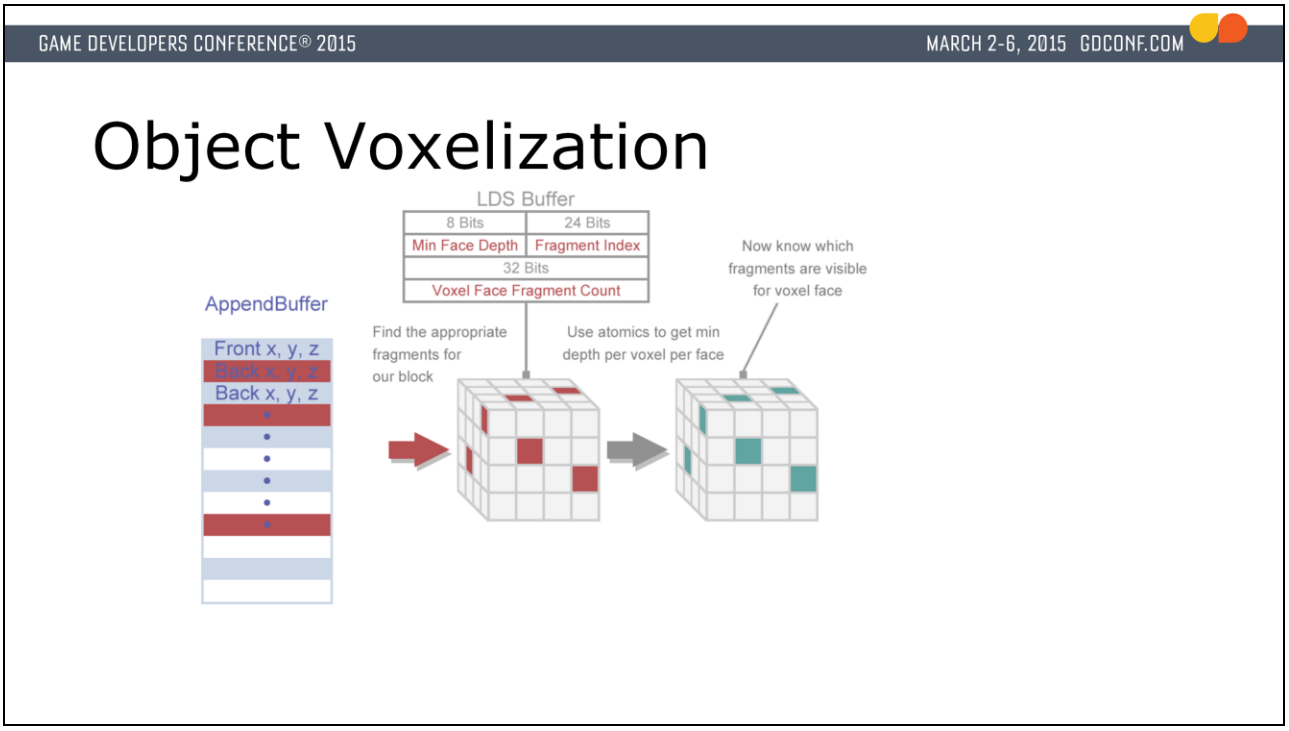
We start by dividing up the space we are voxelizing into what we call "micro blocks" of 4x4x4 voxels,

and in a compute shader, we make a pass through the append buffer in a thread group for each micro block, pulling out the fragments that are in that block

Object Voxelization


LDS Buffer
8 Bits
24 Bits
Min Face Depth
Fragment Index
32 Bits
Voxel Face Fragment Count
AppendBuffer
Front x, y, z
Back x, y, z
Back x, y, z
Find the appropriate fragments for our block
Use atomics to get min depth per voxel per face

For each fragment that falls in the block, we use atomics to write to a buffer in LDS that keeps track of which fragment has the minimum depth on each voxel face, as well as how many fragments fall in that voxel face.

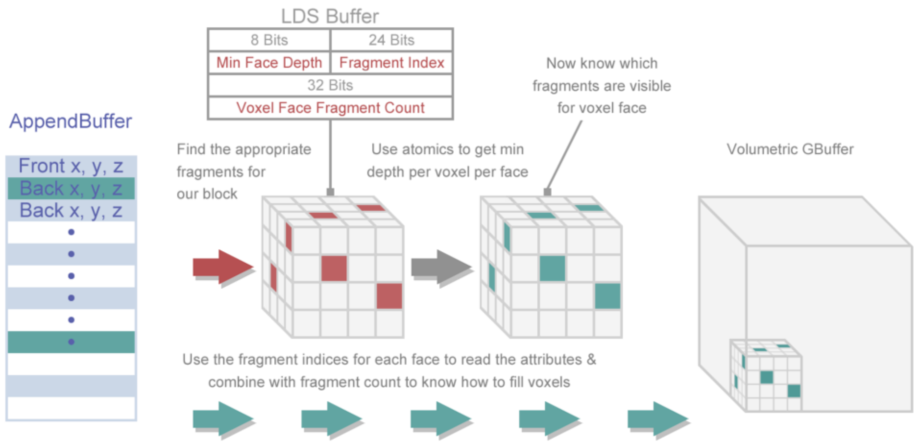

GAME DEVELOPERS CONFERENCE® 2015
MARCH 2-6, 2015 GDCONF.COM

Object Voxelization

LDS Buffer
8 Bits
24 Bits
Min Face Depth
Fragment Index
32 Bits
Voxel Face Fragment Count
AppendBuffer
Front x, y, z
Back x, y, z
Back x, y, z
Find the appropriate fragments for our block
Use atomics to get min depth per voxel per face
Now know which fragments are visible for voxel face

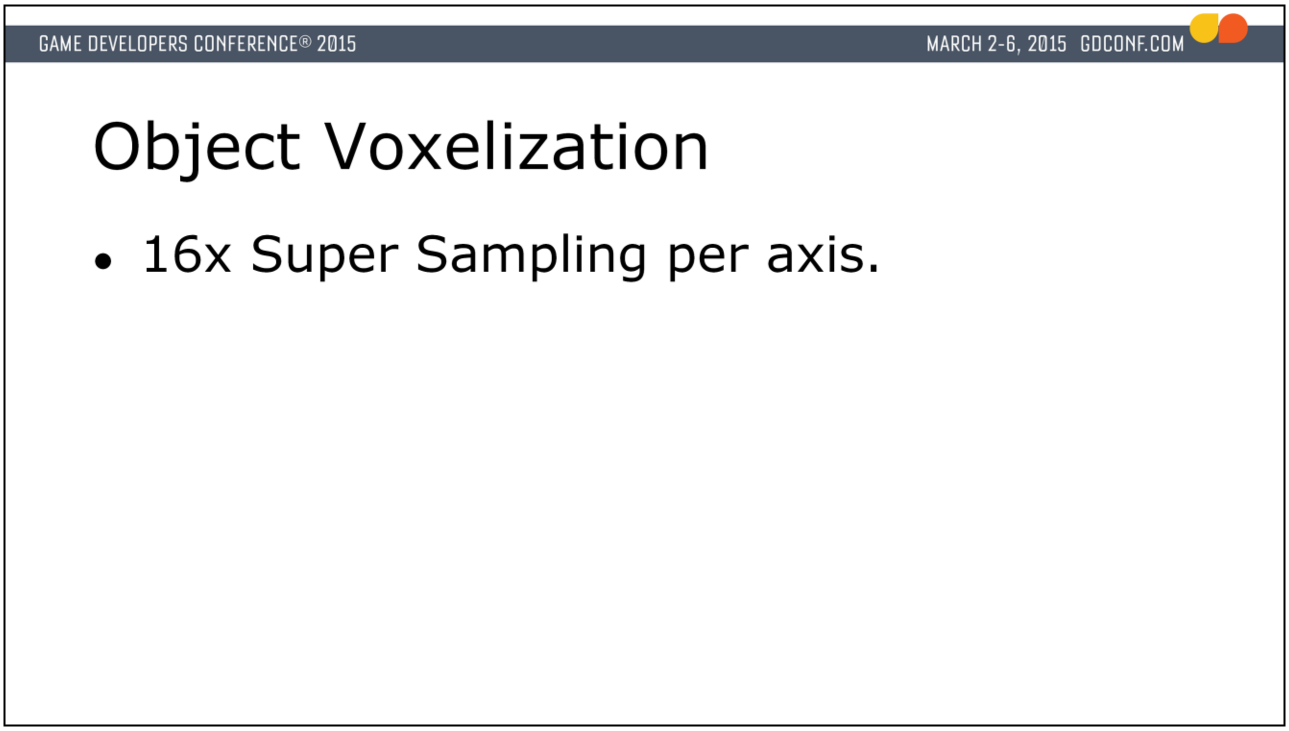
Once we have this information we now know which fragments are the front most for each voxel face.

Object Voxelization


LDS Buffer
8 Bits
24 Bits
Min Face Depth
Fragment Index
32 Bits
Voxel Face Fragment Count
AppendBuffer
Front x, y, z
Back x, y, z
Back x, y, z
Find the appropriate fragments for our block
Use atomics to get min depth per voxel per face
Now know which fragments are visible for voxel face
Volumetric GBuffer
Use the fragment indices for each face to read the attributes & combine with fragment count to know how to fill voxels

And then we can read the material attributes for the front most fragments for each face,

and use that in conjunction with the fragment count to determine what we should write out into our Volumtric Gbuffer for each voxel.

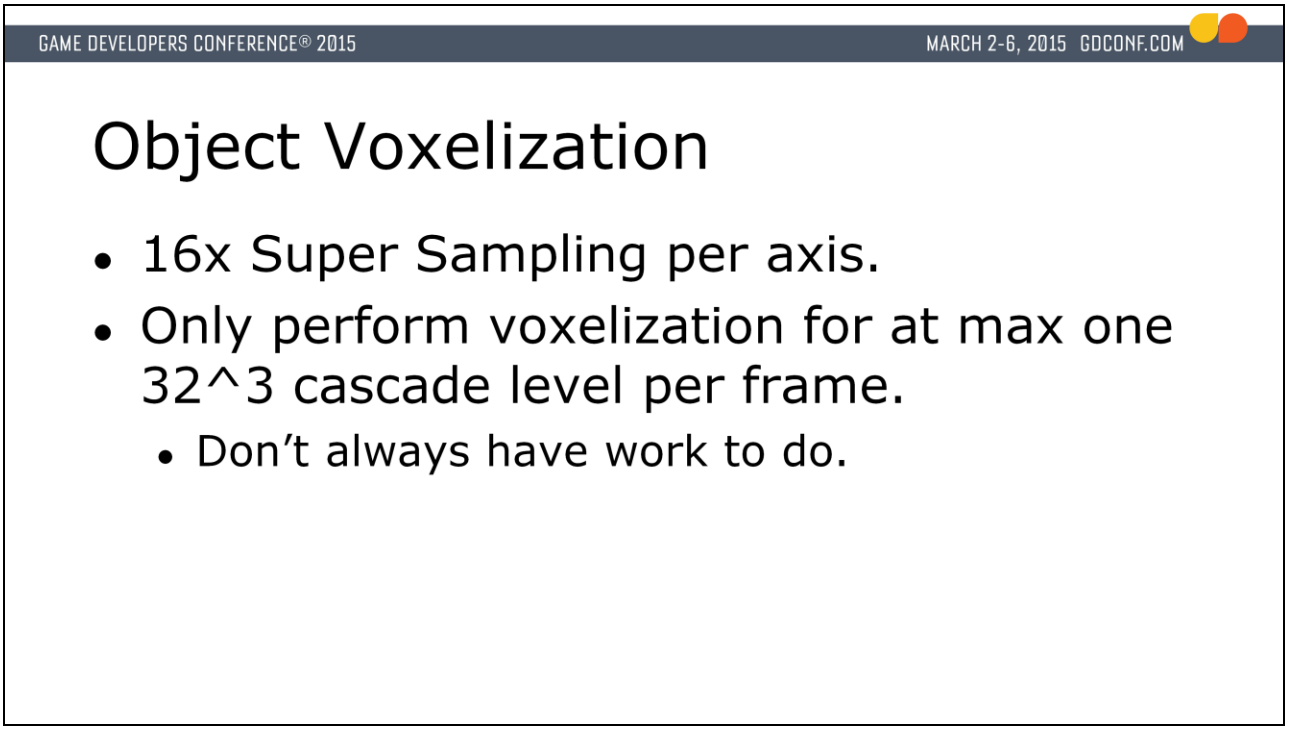

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Object Voxelization

- 16x Super Sampling per axis.

One thing to note is that when we rasterize our objects on each axis, we actually set things up so that the dummy render target that we use has 16x the final resolution that we need.

We use this to perform Super Sampling, which helps us to Anti Alias our resulting voxelized geometry.

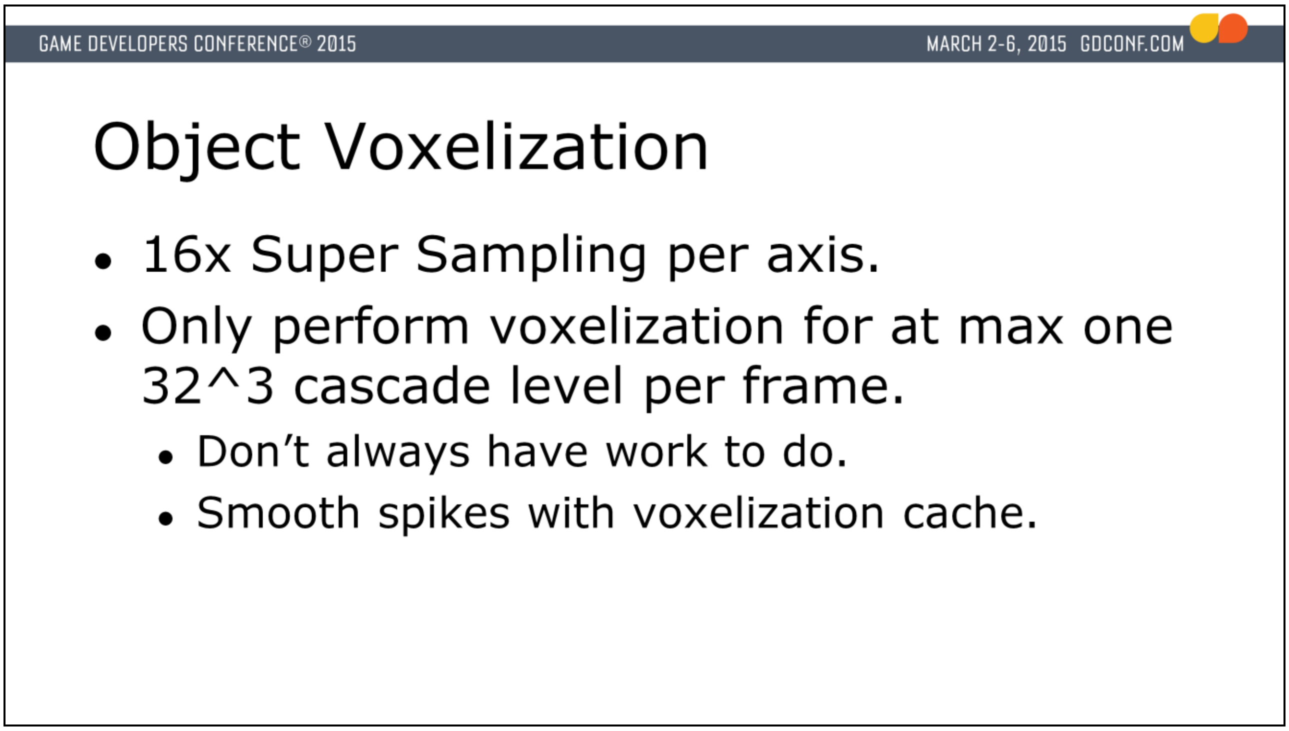

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Object Voxelization

- 16x Super Sampling per axis.
- Only perform voxelization for at max one 32^3 cascade level per frame.
  - Don't always have work to do.

Another detail is that we only ever perform voxelization for the current cascade level that we are updating.

And of course if the camera isn't causing the cascade level to move around, then usually we don't have any work to do.



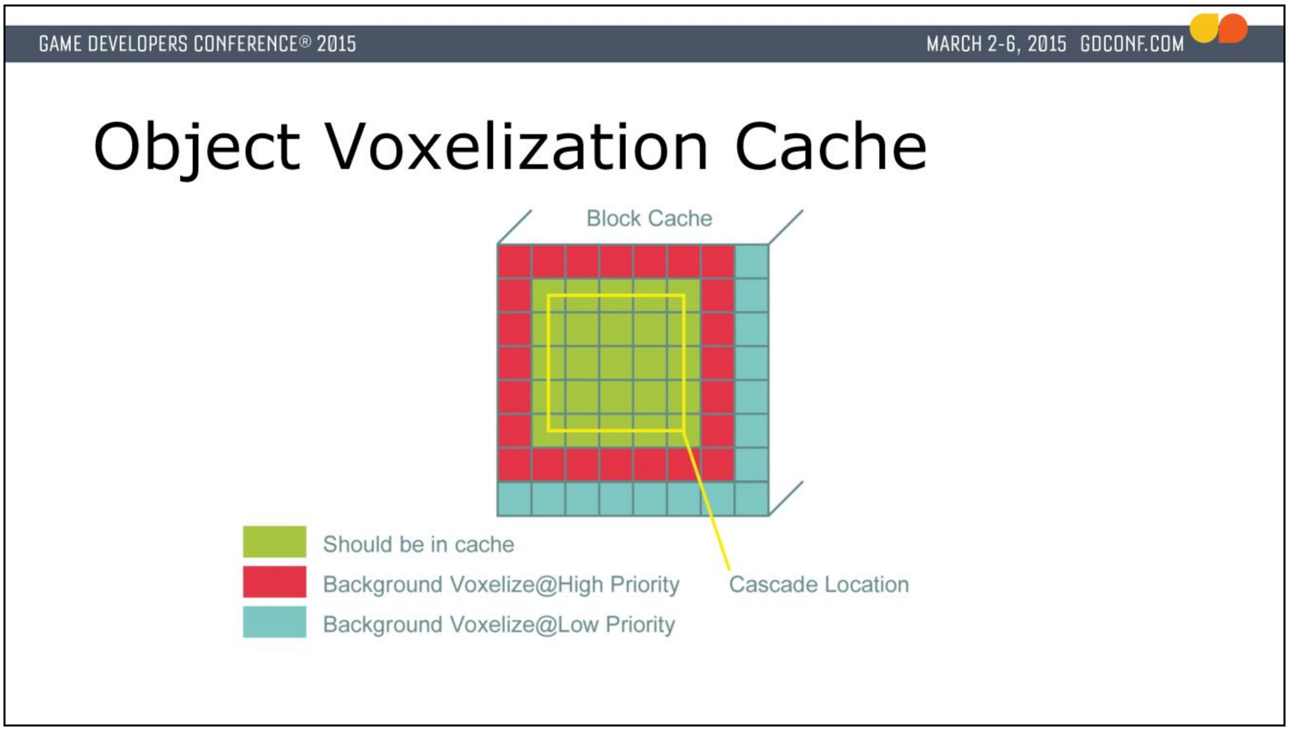
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Object Voxelization

- 16x Super Sampling per axis.
- Only perform voxelization for at max one 32^3 cascade level per frame.
  - Don’t always have work to do.
  - Smooth spikes with voxelization cache.

In order to avoid sudden frame spikes due to voxelization when a cascade moves,

we don’t actually voxelize per object but by region instead.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Object Voxelization Cache

Block Cache
Should be in cache
Background Voxelize@High Priority
Background Voxelize@Low Priority
Cascade Location

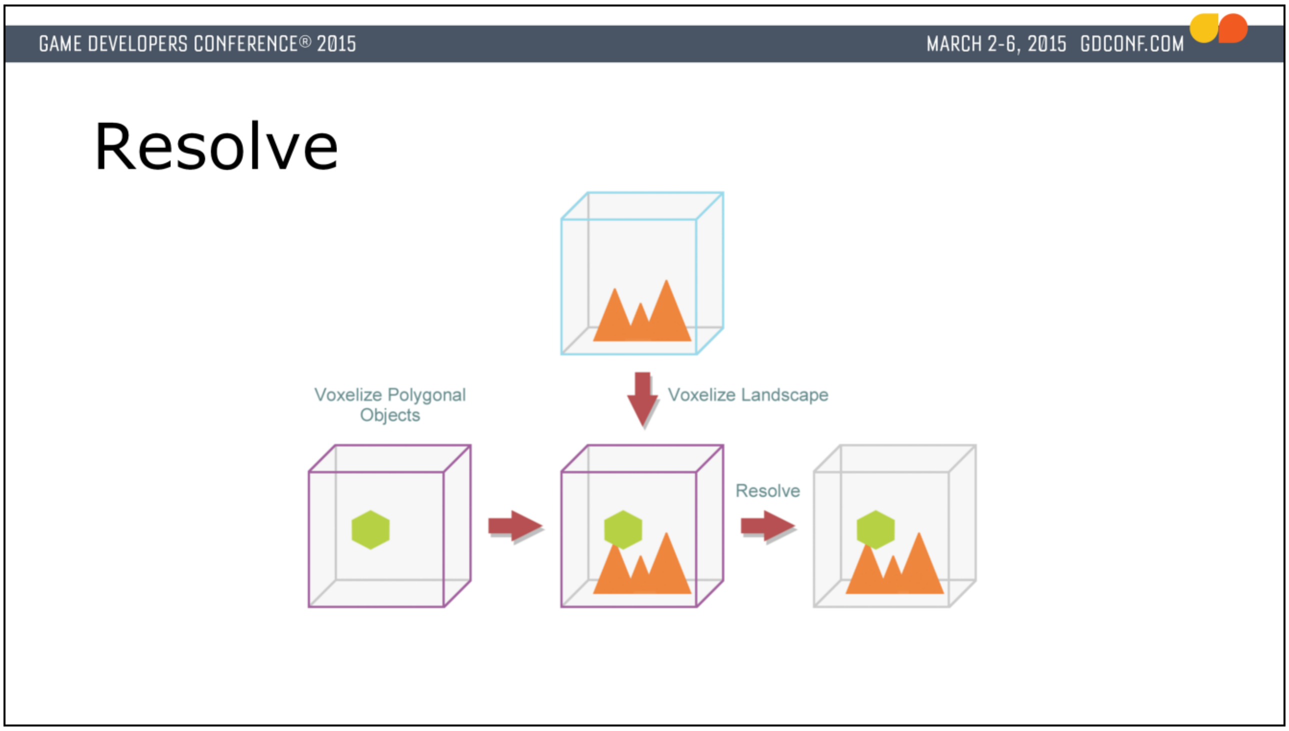
We slowly voxelize blocks of voxels in the background into a cache that covers an area slightly larger than the cascade level.

These blocks, span an area covering several of the “micro blocks” I talked about earlier.

There is some compression magic that goes on here to avoid us using too much memory due to all of this.

But when our cascade moves, it’s actually this cache we use to fill the edges of our cascade level.

As a result we can amortize the cost of voxelization.

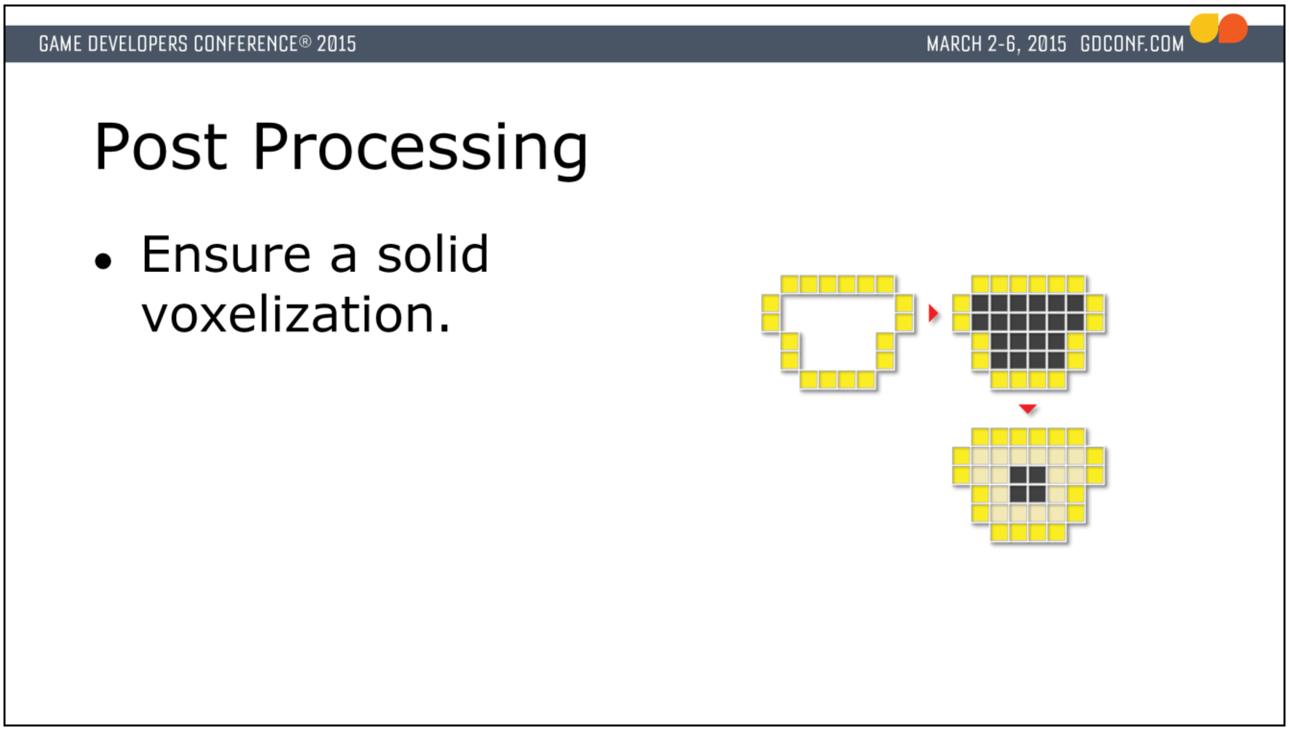


Resolve

Our landscape is stored as a Layered Depth Cube, which is essentially a set of span lists in each axis, and is thus very easy to voxelize

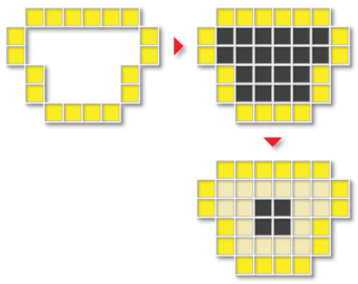
When we move, the landscape must also be voxelized, and combined with the contents of our decompressed object cache.

Once combined, they can be resolved down into our 32^3 cascade level.



Post Processing

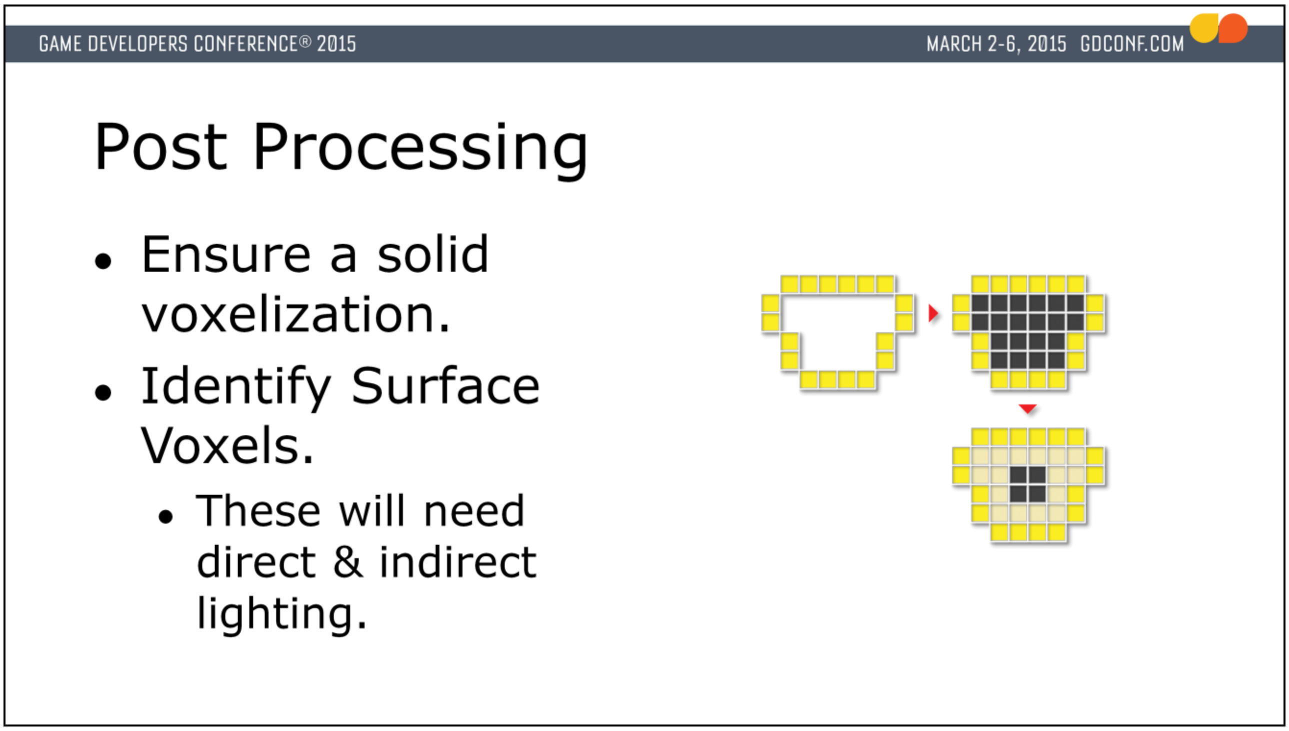
- Ensure a solid voxelization.



One thing that we found important for robustness was to ensure that the voxelization was solid.

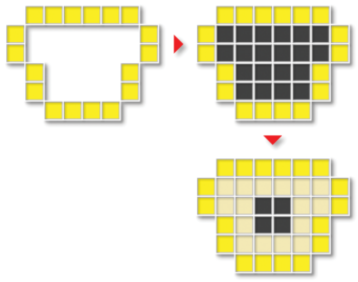
So as a post step after voxelization we fill spans between surface voxels with opaque voxels.

We also propagate our surface attributes inward to the first subsurface layer of voxels to ensure we can’t trace past important information.



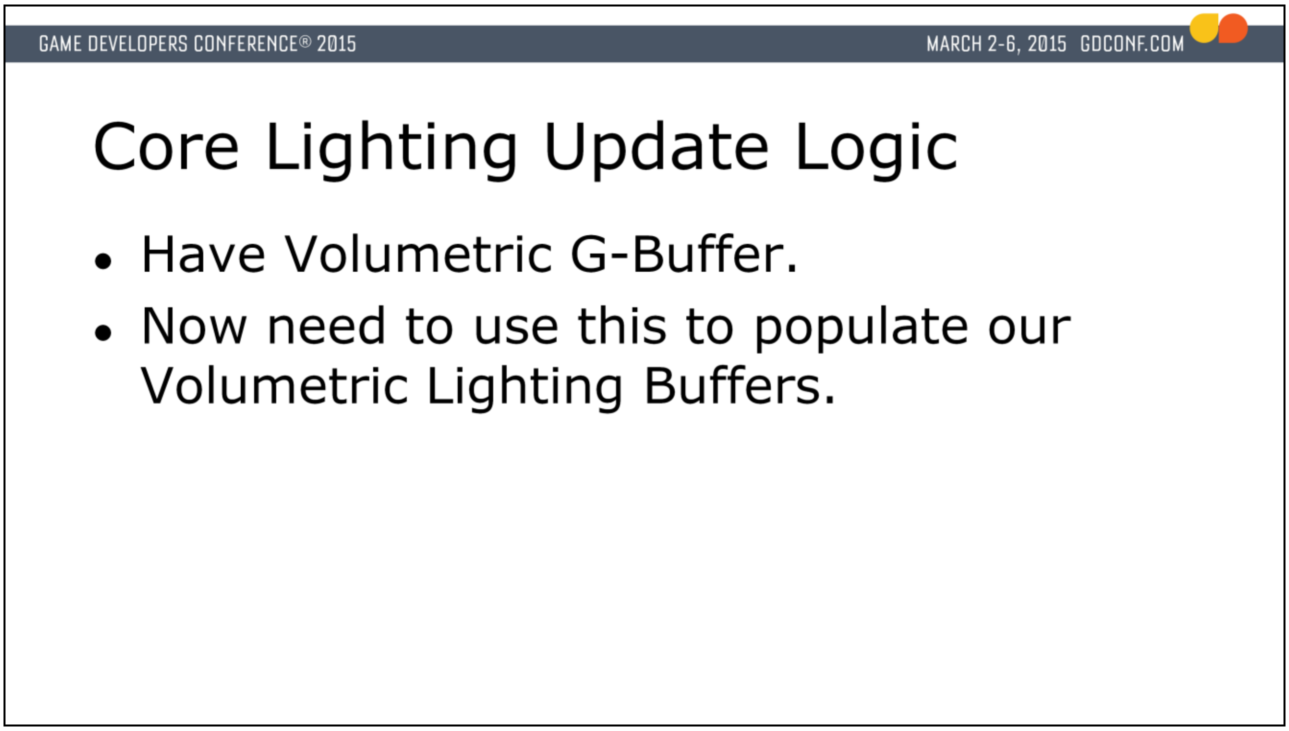
Post Processing

- Ensure a solid voxelization.
- Identify Surface Voxels.
  - These will need direct & indirect lighting.



At the end of our voxelization process we scan for voxels that are on the surface of our geometry, and write these out to an RW_Buffer.

These are the voxels that we will trace from to update the direct and indirect lighting for our cascade.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Core Lighting Update Logic

- Have Volumetric G-Buffer.
- Now need to use this to populate our Volumetric Lighting Buffers.

So now we have filled our volumetric G-Buffers with a representation of the geometry in the scene.

The next step is for us to use this information to fill our volumetric light buffers.

We do this by tracing cones through our scene representation for each surface voxel we have identified for our current cascade level.

GAME DEVELOPERS CONFERENCE® 2015
MARCH 2-6, 2015 GDCONF.COM

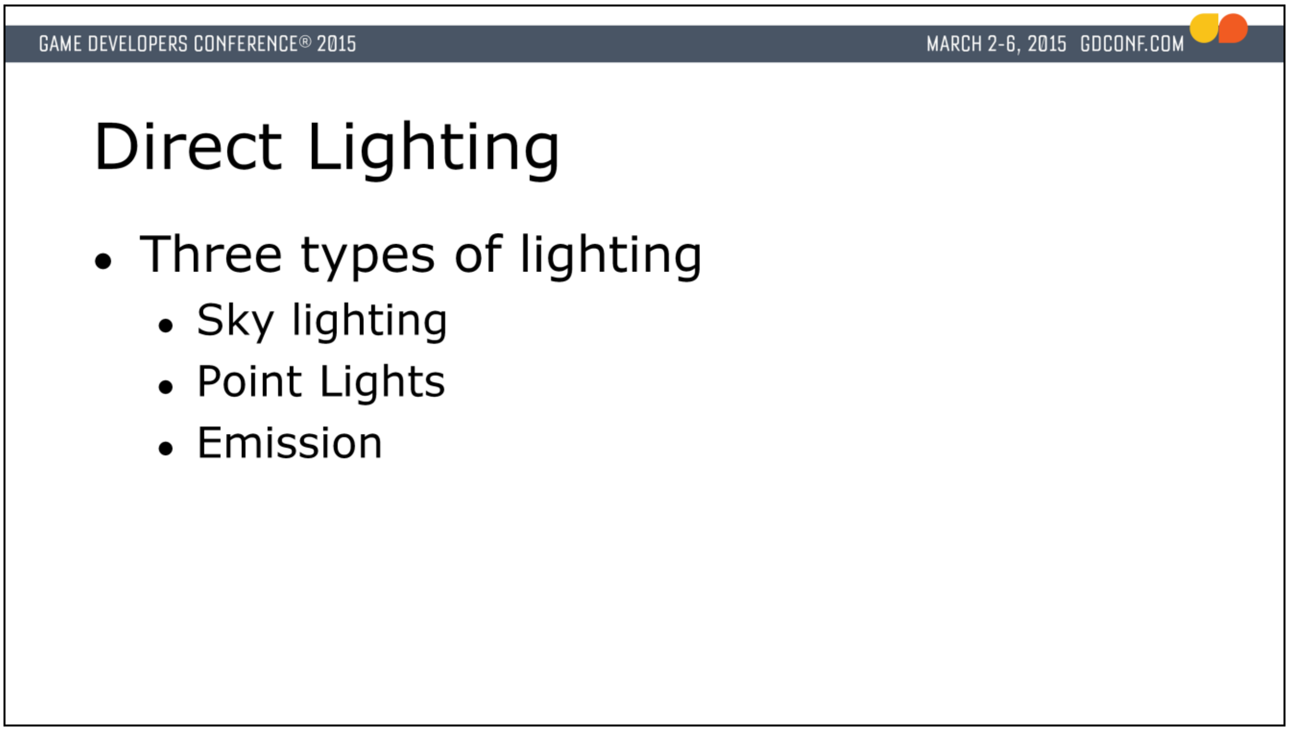
Picking our directions

So we need to pick some directions to trace in when we gather our direct and indirect lighting.

We chose to trace in 16 fixed directions that are well distributed over the unit sphere.

There is actually a whole branch of mathematics behind how these should be chosen, anyone interested should look up Spherical T Designs, but suffice to say, we pick a “good” set of directions that are guaranteed to give a nice integration over the sphere.

We use the same directions all the way through the pipeline, both for cones traced from voxels, and also cones traces for pixels.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Direct Lighting

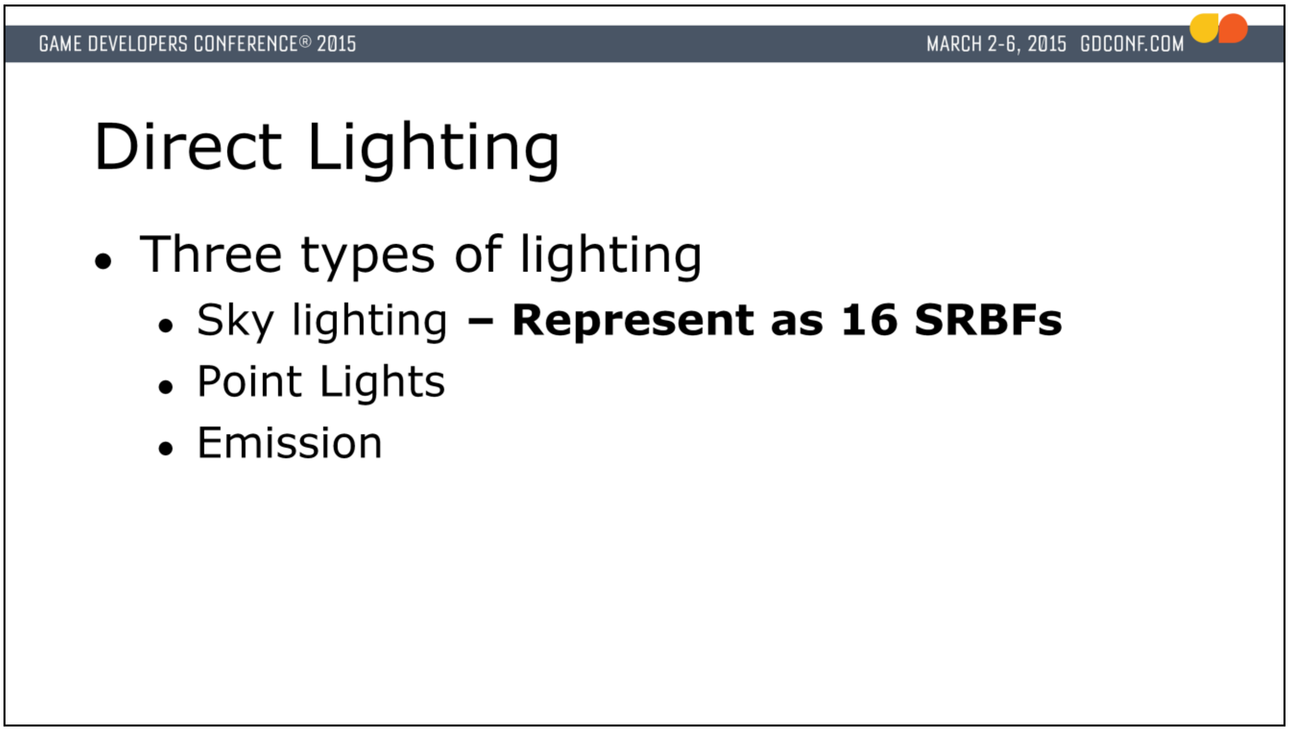
- Three types of lighting
  - Sky lighting
  - Point Lights
  - Emission

So once we have our geometry represented, we need to inject direct lighting into our cascade structure.

We support 3 types of direct lighting.

Light from the sky, point lights, and emissive materials.

For the first two of these, the light is gathered via cone tracing in the 16 directions we just described.

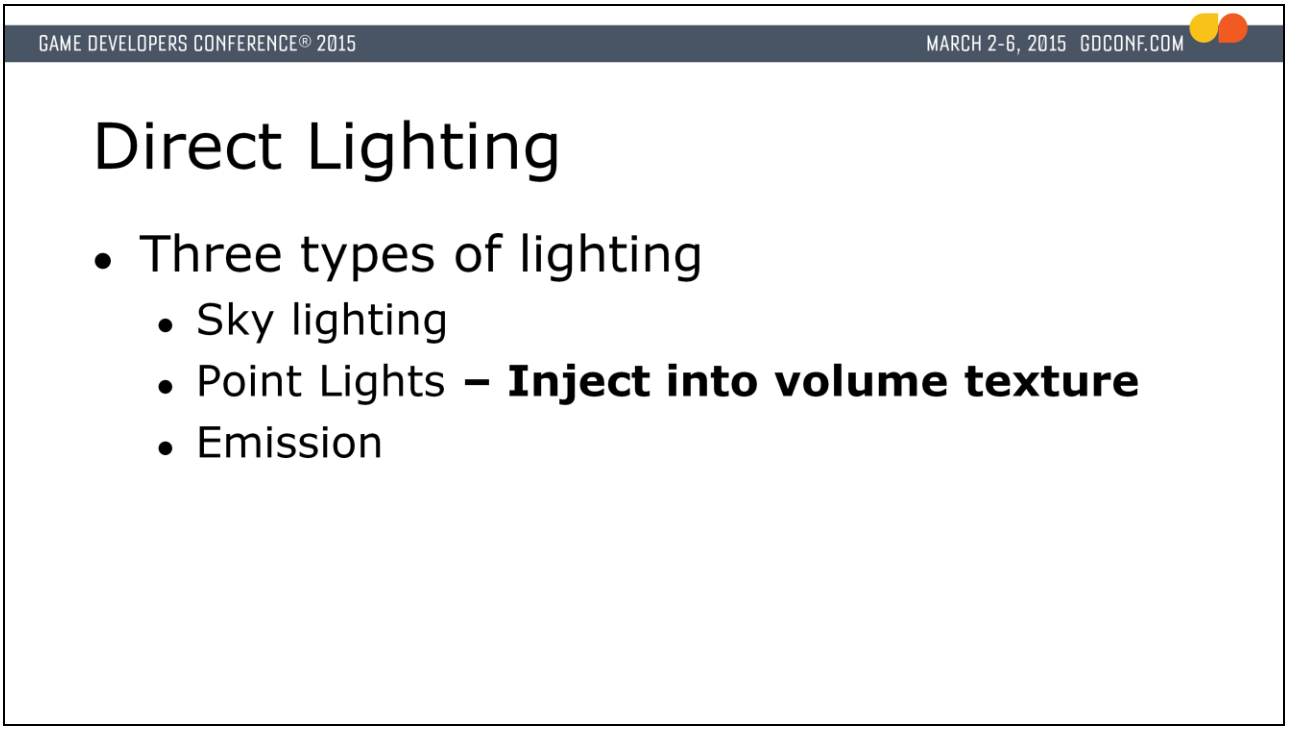


Direct Lighting

- Three types of lighting
  - Sky lighting **– Represent as 16 SRBFs**
  - Point Lights
  - Emission

In order to make injecting lighting from the sky easy, we represent our sky lighting as 16 Spherical Radial Basis Functions, aligned to the 16 directions that we will cone trace in.

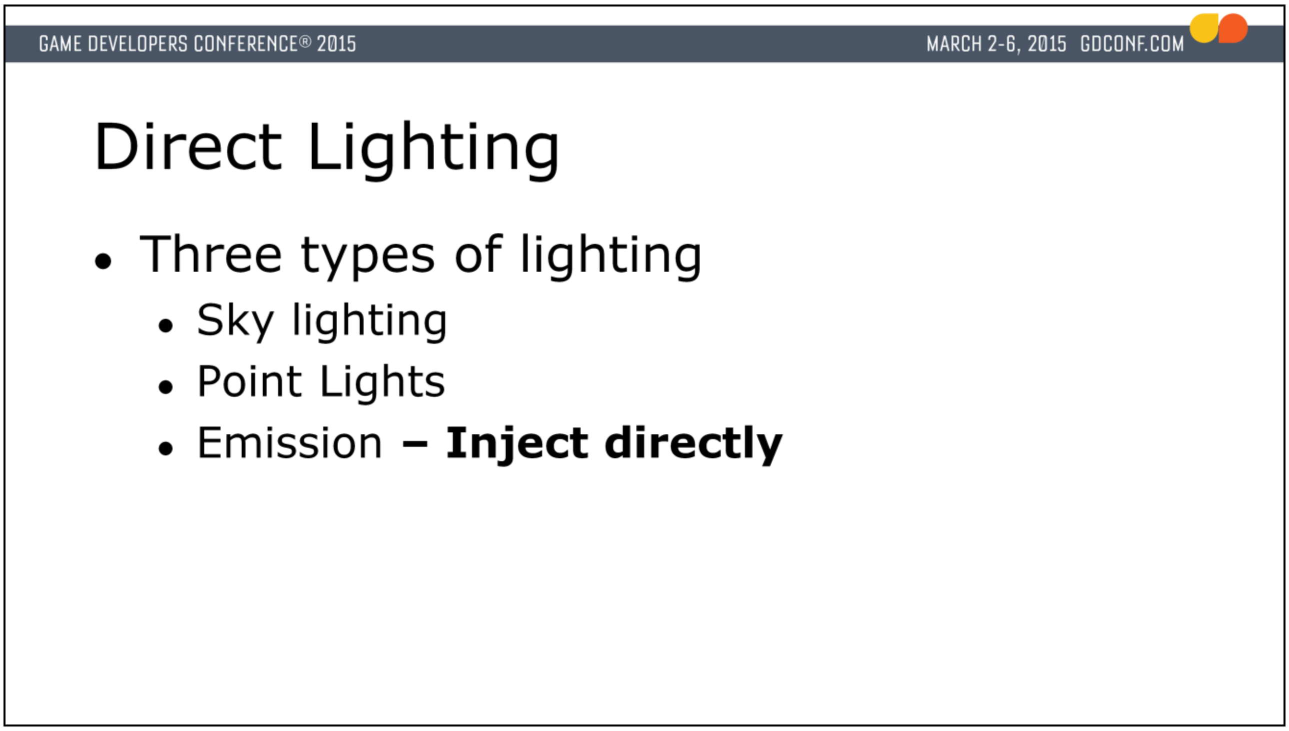
This does have the effect that it makes it difficult for us to represent sharp high frequency sun lighting, and it’s associated shadowing in our game, but as we had specifically made the choice with our games art style to avoid this, it wasn’t a major limitation for us.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Direct Lighting

- Three types of lighting
  - Sky lighting
  - Point Lights **– Inject into volume texture**
  - Emission

For point lights, we use a geometry shader to inject them into a volume texture, which is also cascaded like our other buffers, and we can query from this as we perform a cone trace in our 16 directions, using the occlusion that we accumulate along the way to provide plausible shadowing.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Direct Lighting

- Three types of lighting
  - Sky lighting
  - Point Lights
  - Emission **– Inject directly**

Emission is the simplest of the three, and we simply inject the values directly into our direct lighting buffer at each voxel face.

Bounce Light

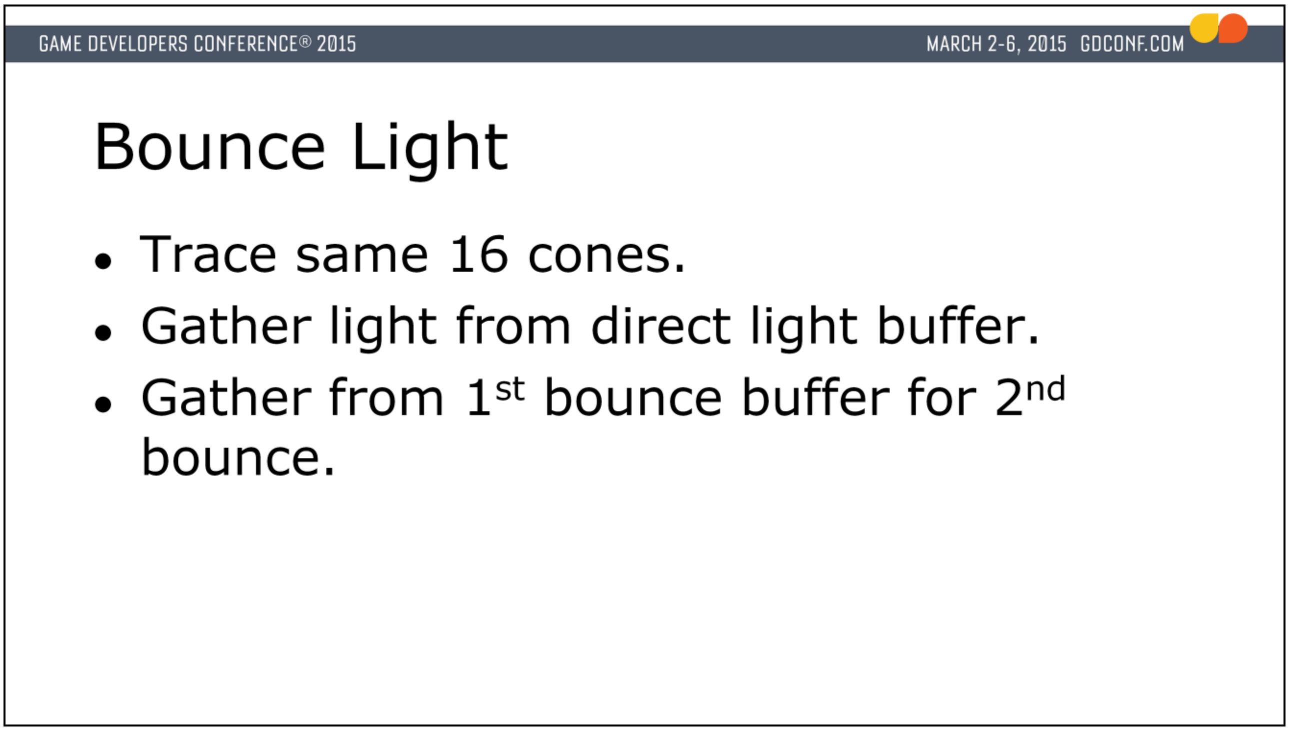
- Trace same 16 cones.
- Gather light from direct light buffer.

So now that we have injected our direct lighting into our volumetric light buffer,

we can go about using that to generate our bounce lighting,

Which we do by again tracing cones in each of our 16 directions,

but this time, we sample from the direct light buffer as we trace along our cones.

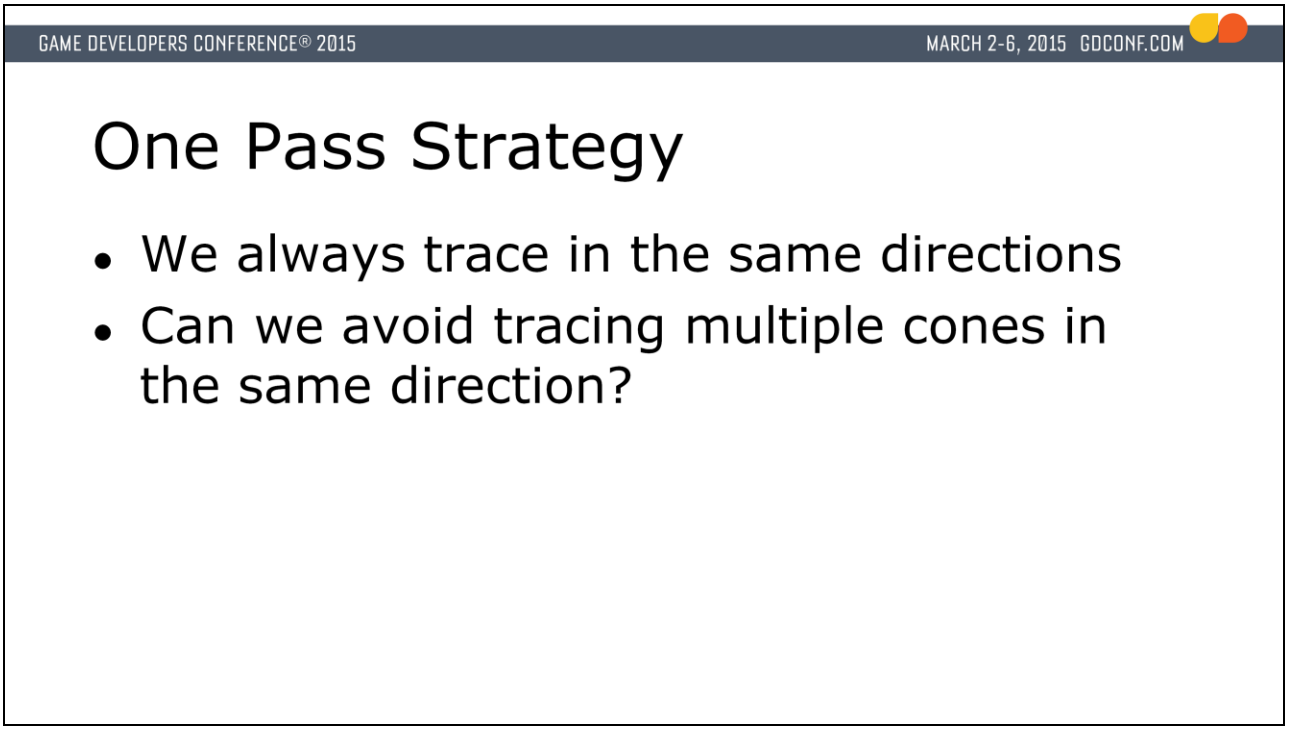


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Bounce Light

- Trace same 16 cones.
- Gather light from direct light buffer.
- Gather from 1st bounce buffer for 2nd bounce.

And of course, once we have the first bounce lighting, we can do the same thing again, and trace cones, sampling from that to get our second bounce.

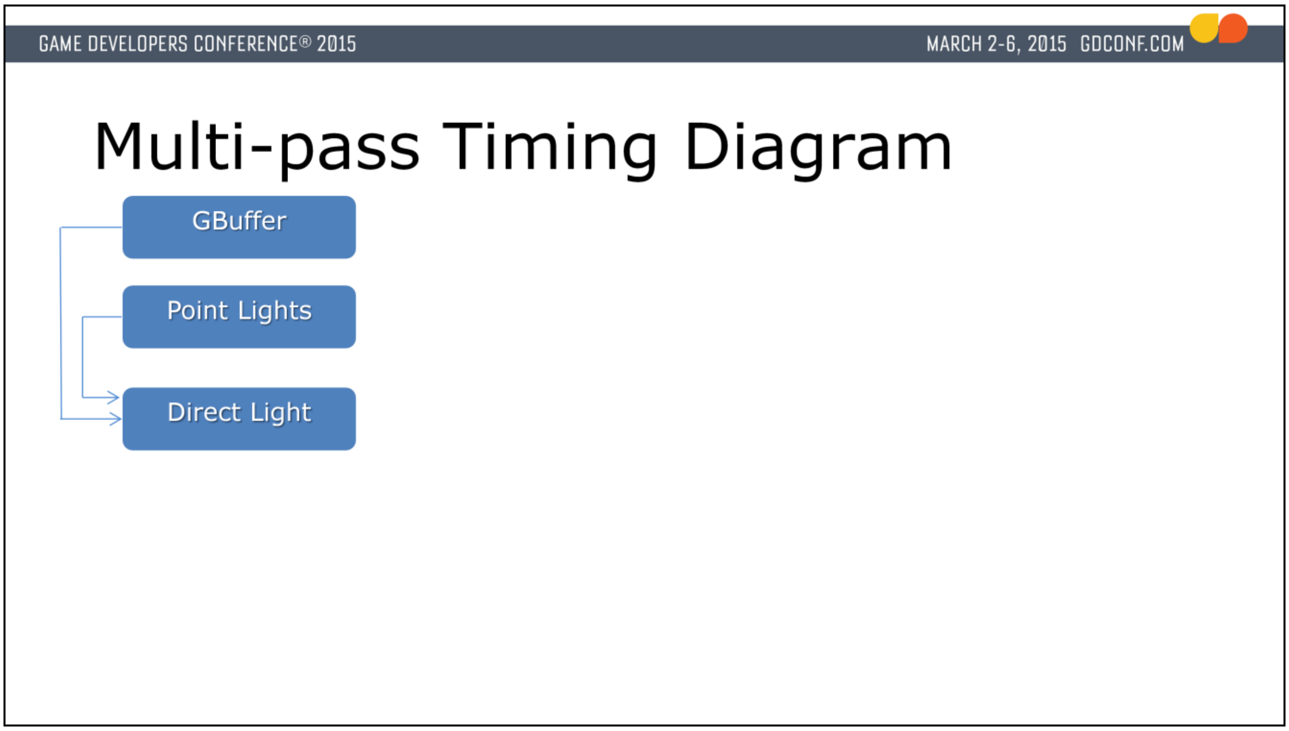

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

One Pass Strategy

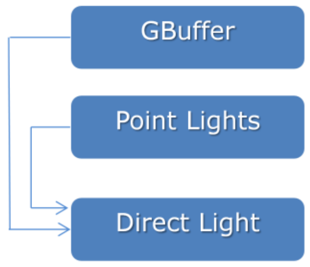
- We always trace in the same directions
- Can we avoid tracing multiple cones in the same direction?

So with the strategy I've just described, we would have to perform 3 sets of cone traces for each surface voxel if we want to get our direct lighting and 2 bounces of indirect lighting stored in our volumetric data structure.

However, it turns out that, because we are always tracing in a fixed set of directions, the cones we trace from a surface voxel for our direct lighting will sample from the exact same voxels as for our indirect lighting, and so we can fold these three cone traces down into just one.

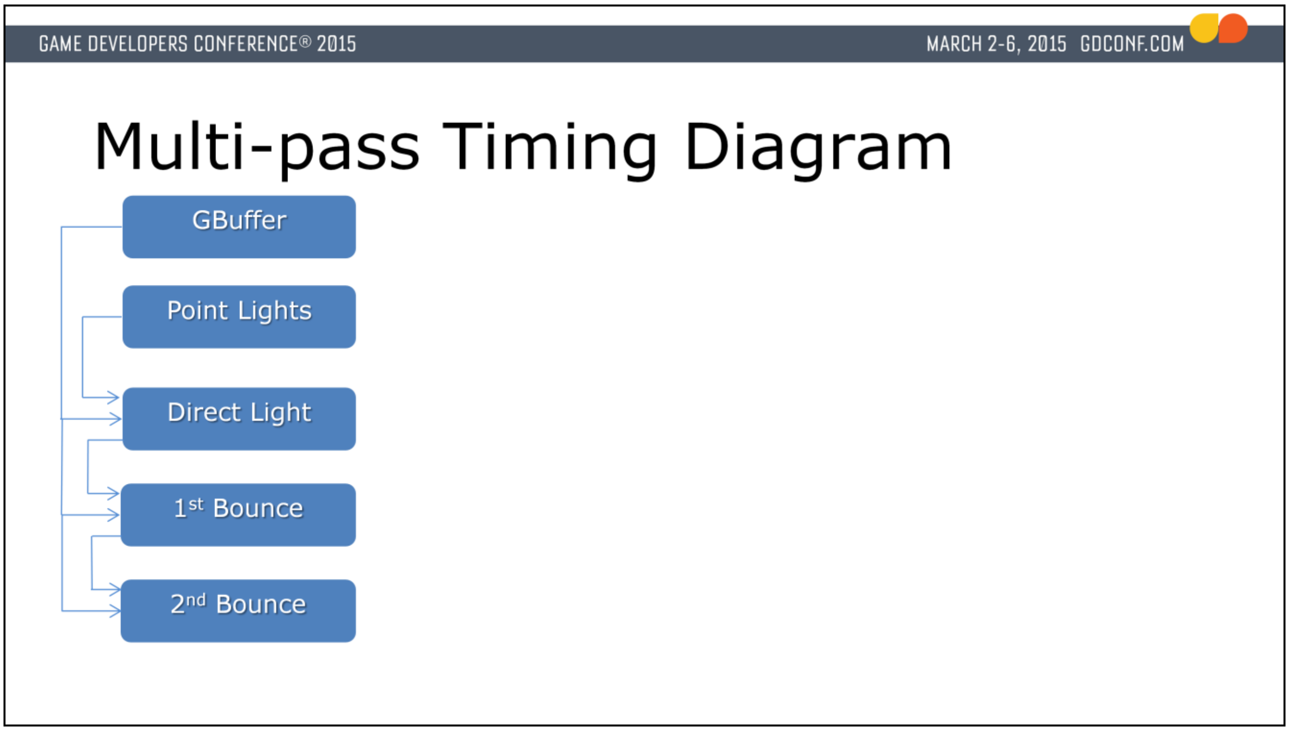


Multi-pass Timing Diagram



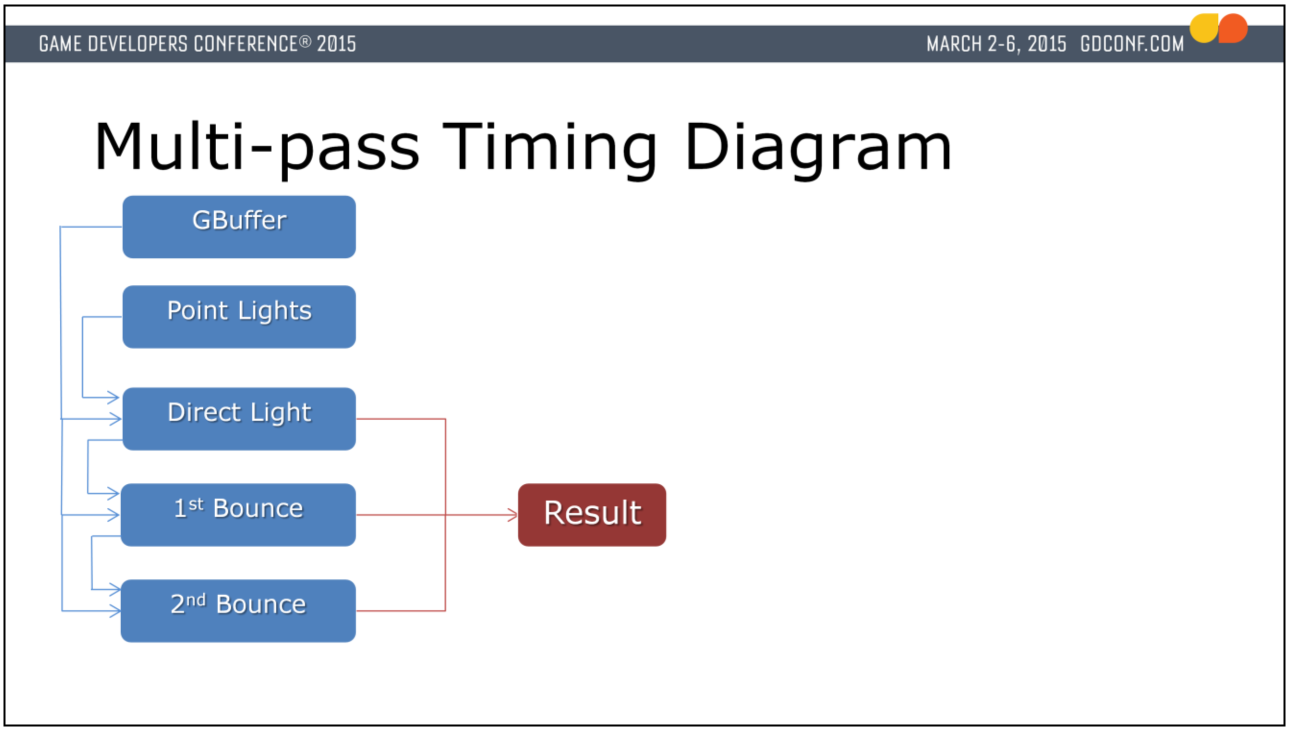
So I'll try to describe that with a diagram

If we imagine our update, then we perform a cone trace with our volumetric gbuffer and our point light texture, to get our direct lighting

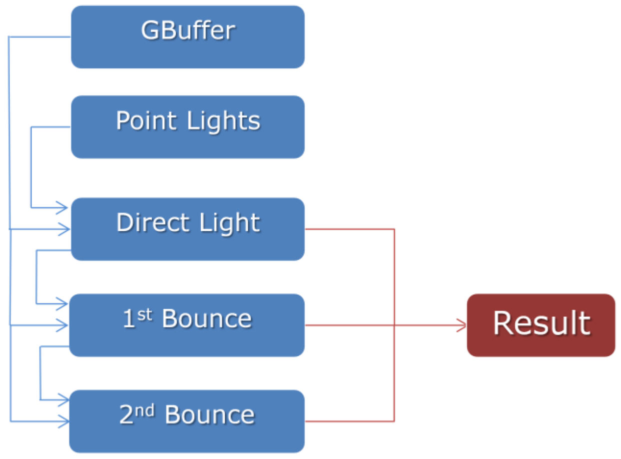


Multi-pass Timing Diagram

And then cone trace with that again to get our 1st bounce, and then second bound lighting

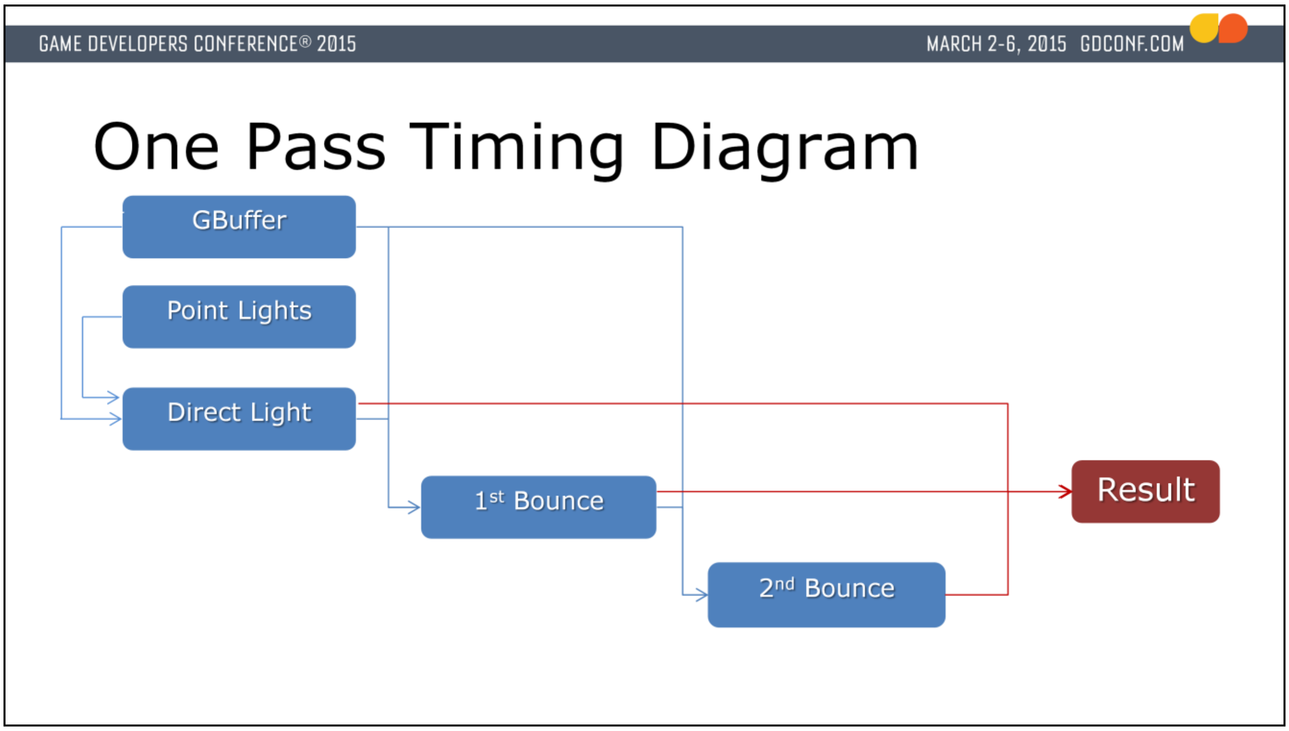

GAME DEVELOPERS CONFERENCE® 2015
MARCH 2-6, 2015 GDCONF.COM

Multi-pass Timing Diagram



Once we have the direct lighting and our bounce lighting results, we can sum them to get our final volumetric lighting.

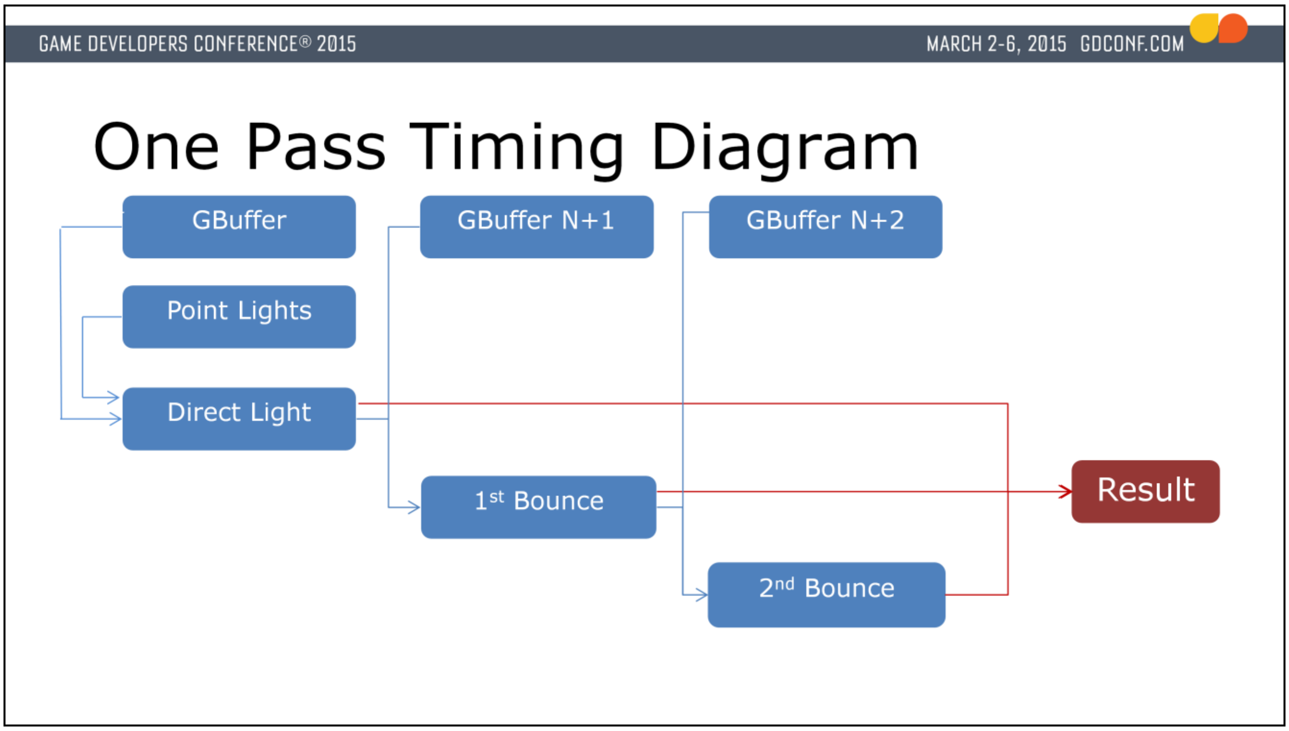
But as I said, this requires three sets of cone traces per frame.



One Pass Timing Diagram

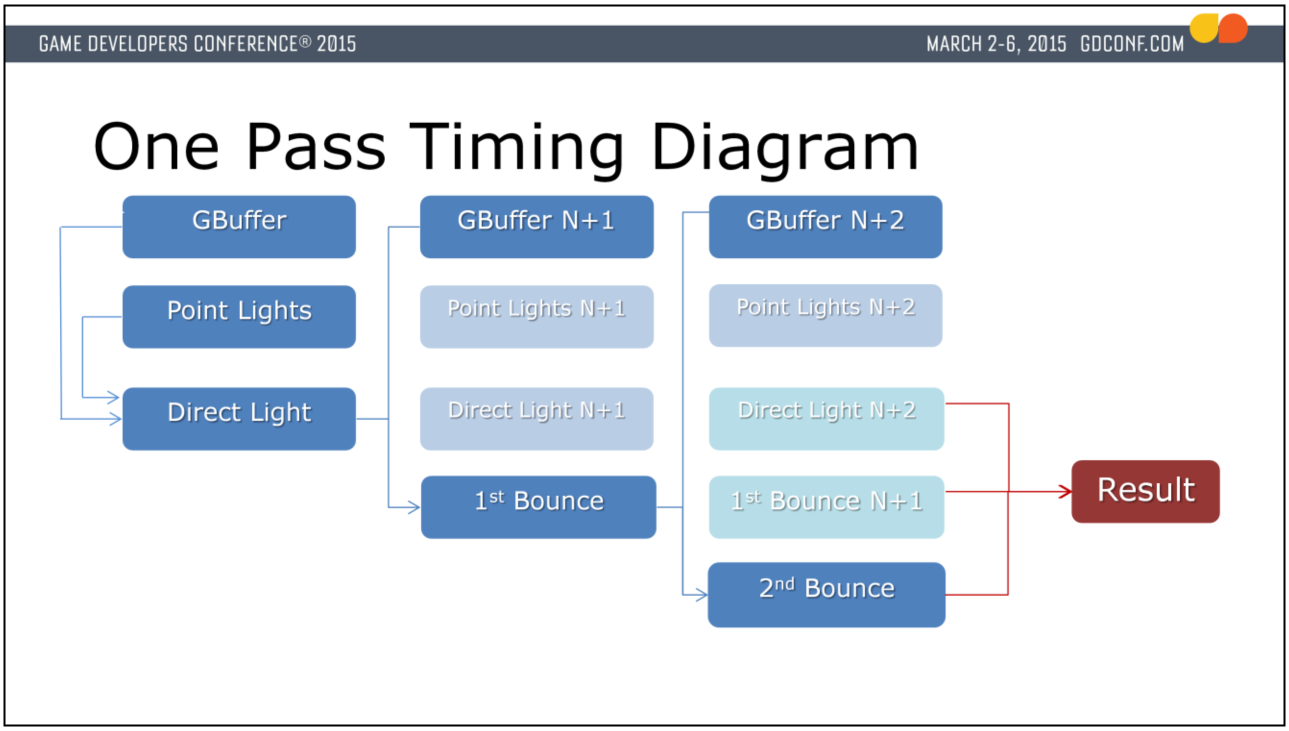
So we can start to transform this into a single pass solution by shifting this so that our calculations are staggered across frames

So we're doing the same as before, but now we have another 2 frames of latency


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

One Pass Timing Diagram

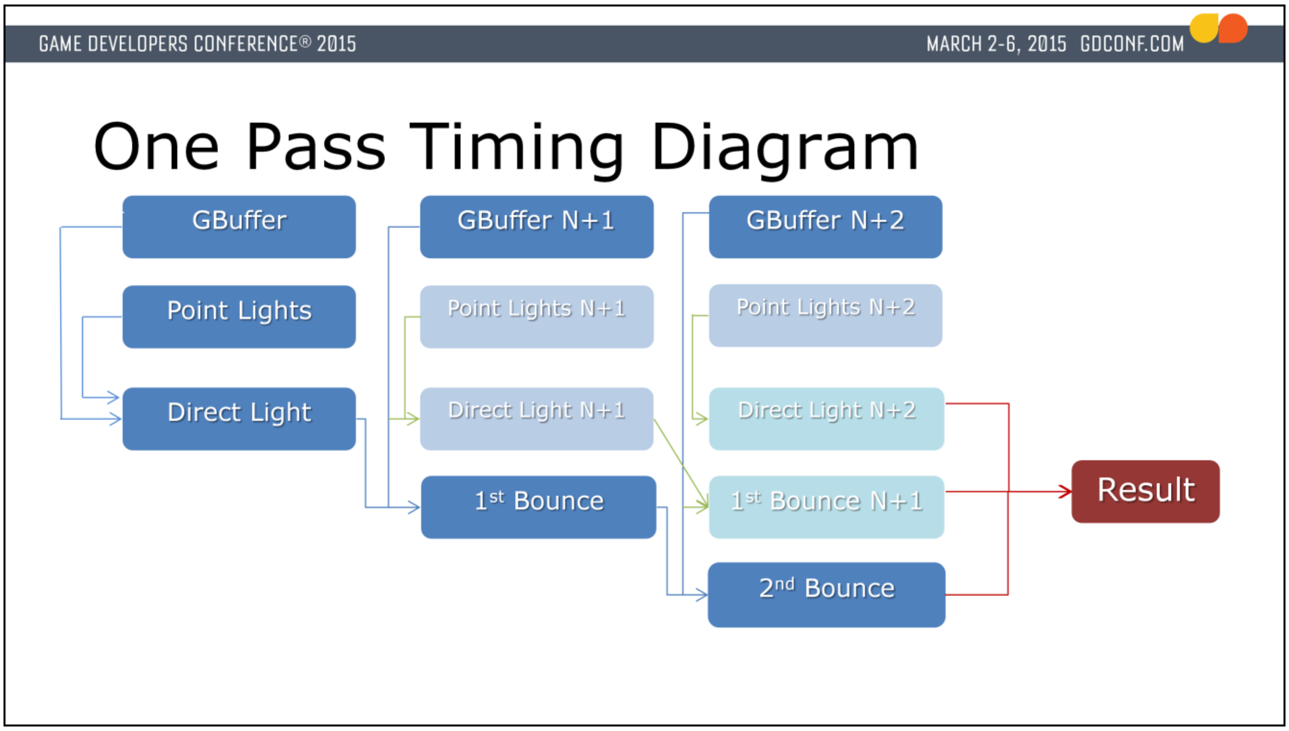
Then we can note, that because our volumetric Gbuffer doesn’t change that quickly, we can use the gbuffer for the current frame when we calculate the 1st Bounce and 2nd Bounce texture and we’ll get almost the same result,



One Pass Timing Diagram

Now because we have interleaved our calculations, we are still always calculating new version of each of the direct lighting, and the 1st and second bounce lighting each frame.

So we can actually get away with just adding up the versions of the each of these values that we are calculating on any given frame, and still get something similar to the original result, and as an added bonus, the 2 frames latency we had on the lighting has now been removed.

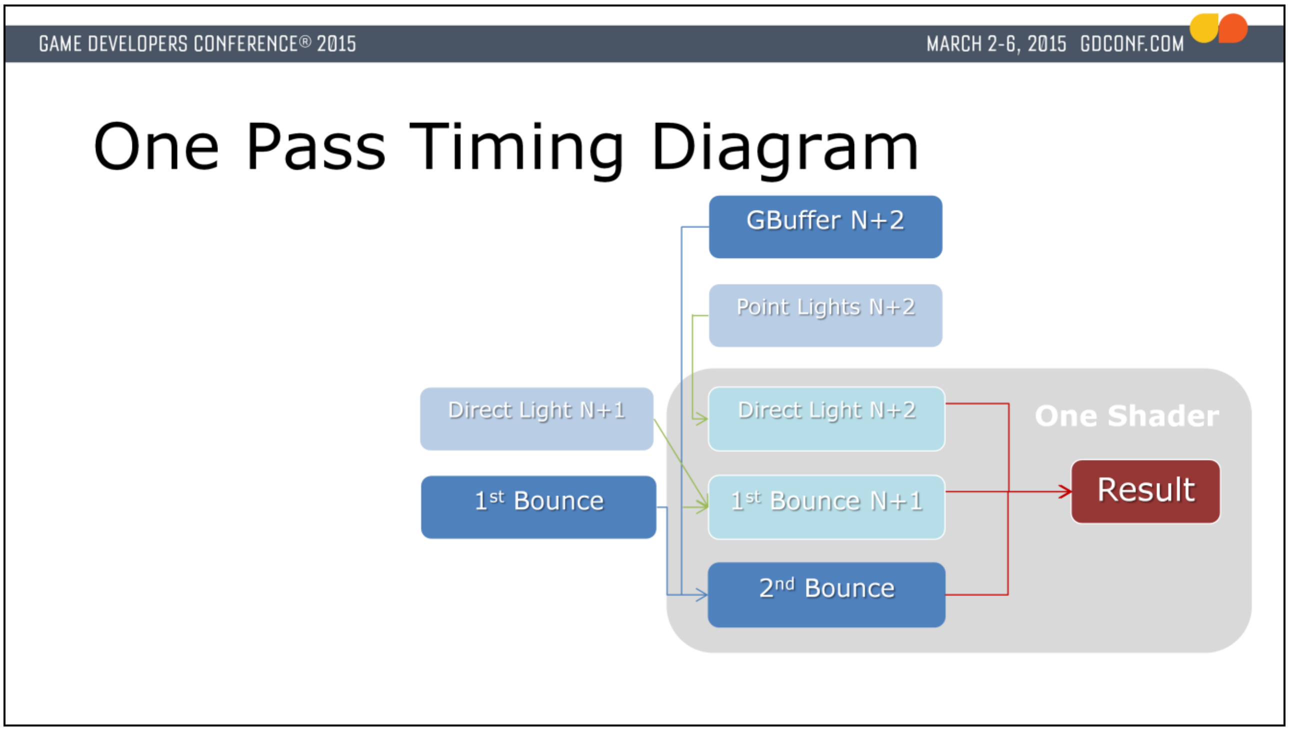


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

One Pass Timing Diagram

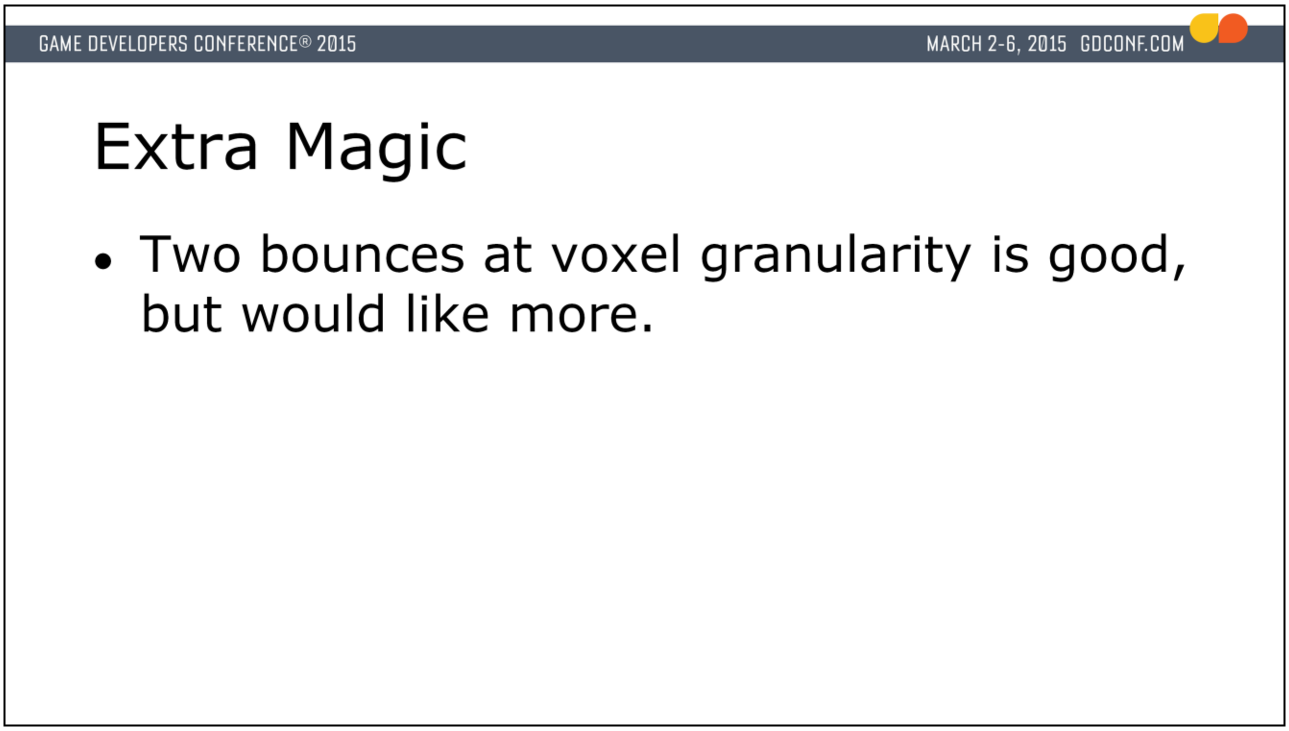
Finally, if we add a few more arrows to show what’s going where, and we look at what we need as input for the last frame in our diagram,

Then you’ll see that everything comes from either our volumetric light buffers from the previous frame, or from our gbuffer and point light texture for the current frame.



One Pass Timing Diagram

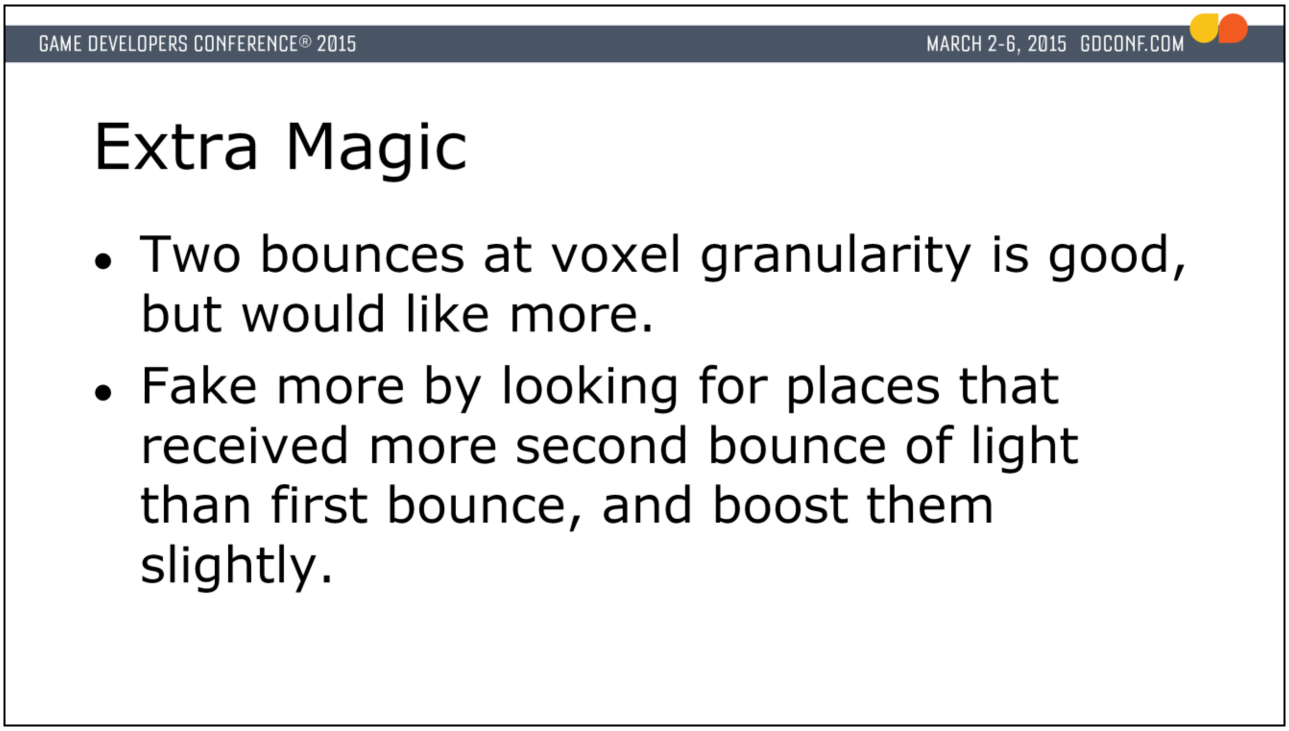
And so in this way we can make a single shader that does one cone trace pass, but that simultaneously reads from and updates our direct lighting, our bounce lighting, and our result texture all in one go.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Extra Magic

- Two bounces at voxel granularity is good, but would like more.

So, with what I've described until now, we have two bounces of light at our voxel granularity, which is nice, but we would like to have even more.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

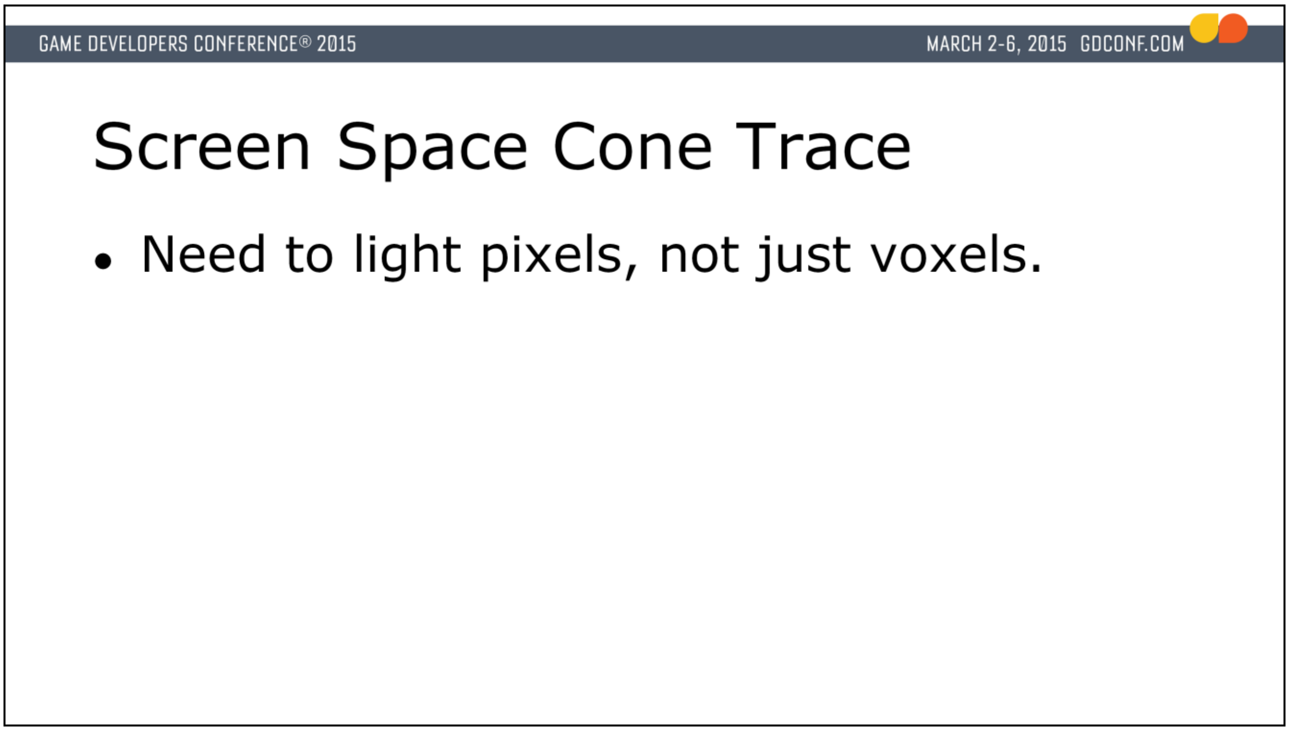
Extra Magic

- Two bounces at voxel granularity is good, but would like more.
- Fake more by looking for places that received more second bounce of light than first bounce, and boost them slightly.

So, we try to fake just a little bit more by looking for places that received more second bounce light than first bounce light,

and surmising that they would probably get more illumination from a third bounce.

We add this extrapolated extra bounce to our final result.

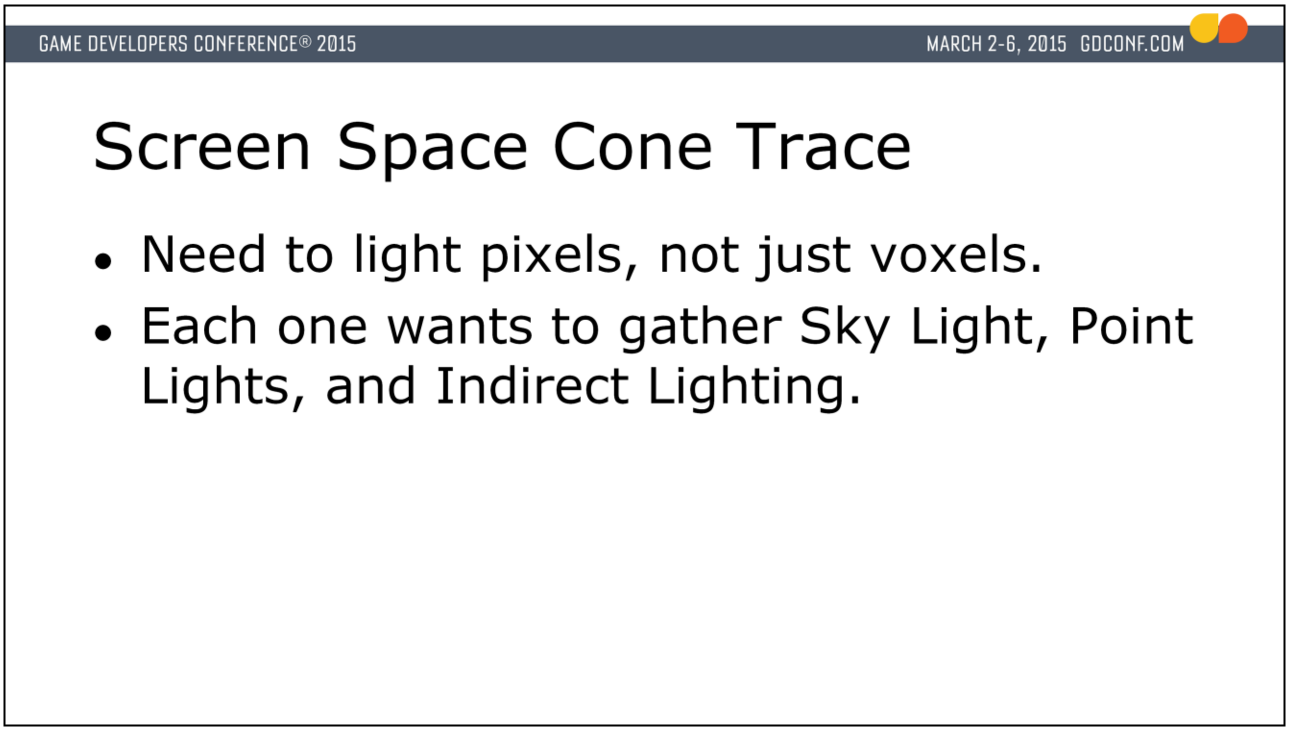


Screen Space Cone Trace

- Need to light pixels, not just voxels.

Now we've managed to fill our volumetric data structure with details about the scenes lighting.

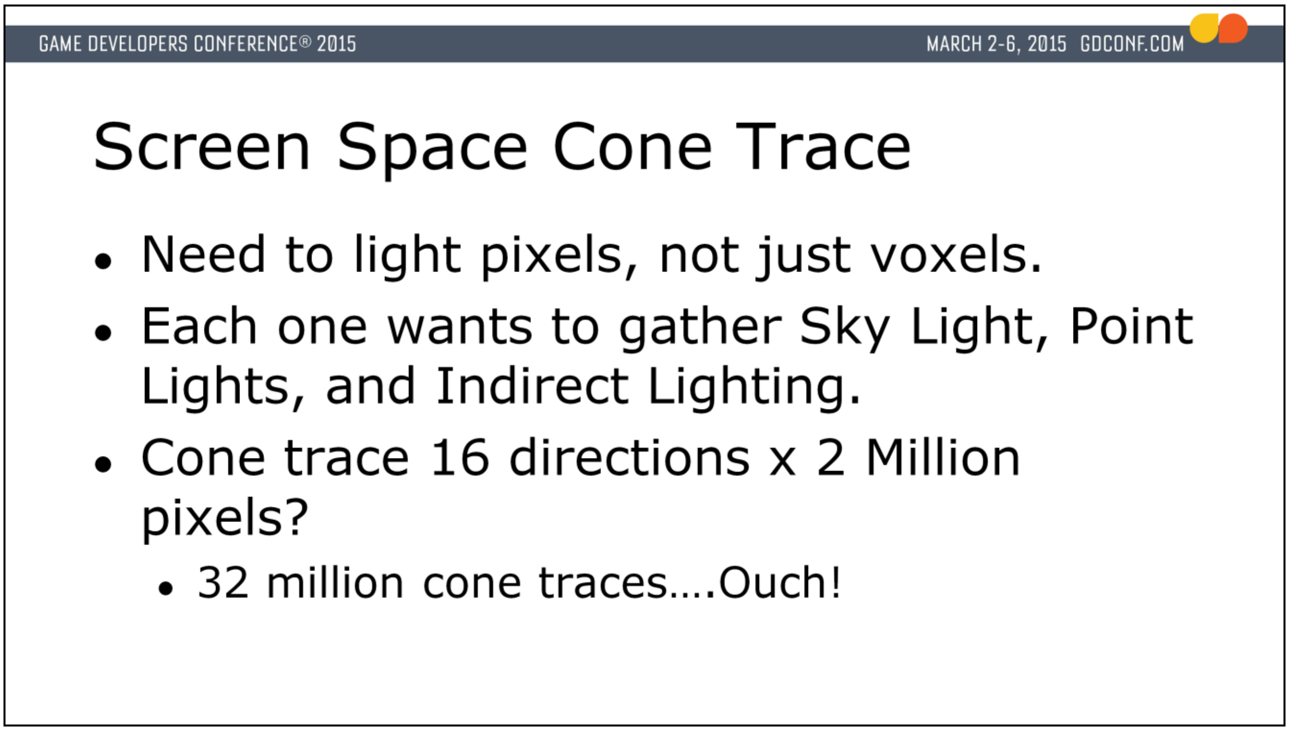
But ultimately we are interested in lighting pixels not voxels, so we need to get our data into screen space.



Screen Space Cone Trace

- Need to light pixels, not just voxels.
- Each one wants to gather Sky Light, Point Lights, and Indirect Lighting.

For each pixel we want to gather lighting information from the sky, from point lights, and also from indirect lighting.

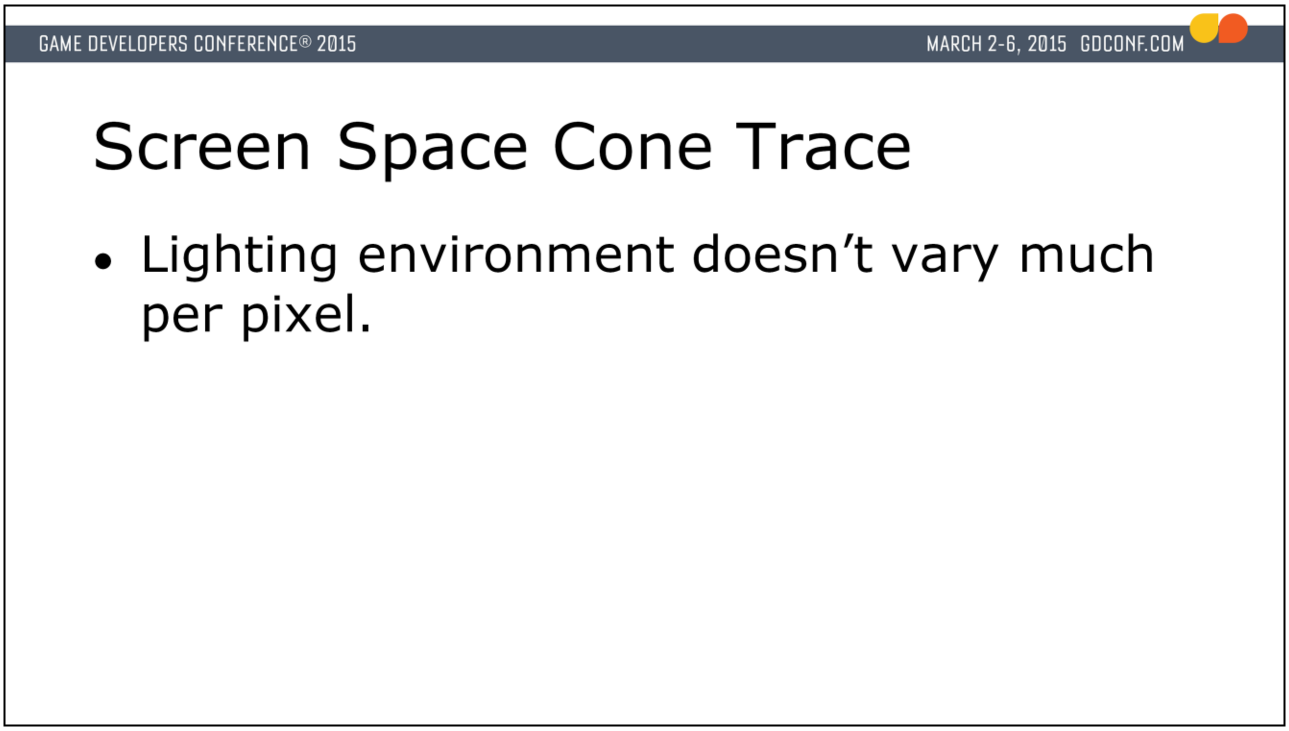


Screen Space Cone Trace

- Need to light pixels, not just voxels.
- Each one wants to gather Sky Light, Point Lights, and Indirect Lighting.
- Cone trace 16 directions x 2 Million pixels?
  - 32 million cone traces….Ouch!

We can achieve this by cone tracing through our volumetric data structure from the world space position of each pixel.

Unfortunately if we just naively do this, then we get 32 million cone traces per frame to light our pixels at 1080p… that's just a few too many!



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

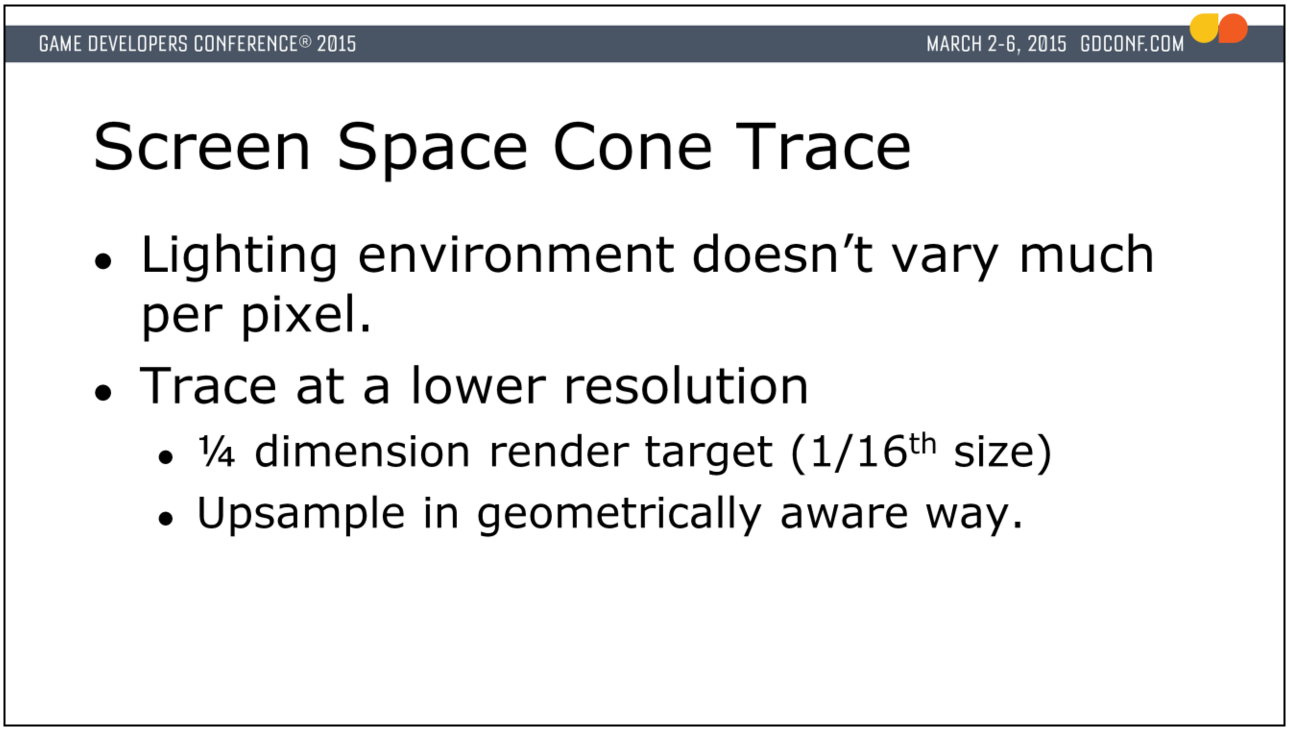
Screen Space Cone Trace

- Lighting environment doesn't vary much per pixel.

Thankfully, whilst the actual lighting between pixels might change considerably, due to normals , materials and the like

The lighting environment generally does not.

The best way to think about this is that what we are actually doing for each pixel when we do all this cone tracing is effectively just build a very low resolution, low frequency environment map, and because we don't have hard shadows, this doesn't change very drastically between pixels, as long as we don't have large depth discontinuities.

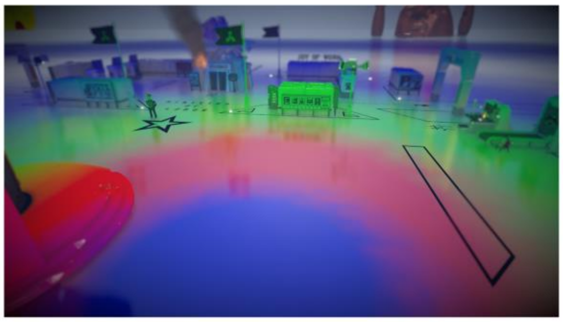


Screen Space Cone Trace

- Lighting environment doesn't vary much per pixel.
- Trace at a lower resolution
  - ¼ dimension render target (1/16th size)
  - Upsample in geometrically aware way.

So this means that we can do all of this work at a much lower resolution, say 16th of the size, and intelligently upscale it in a geometrically aware way to our actual screen resolution.

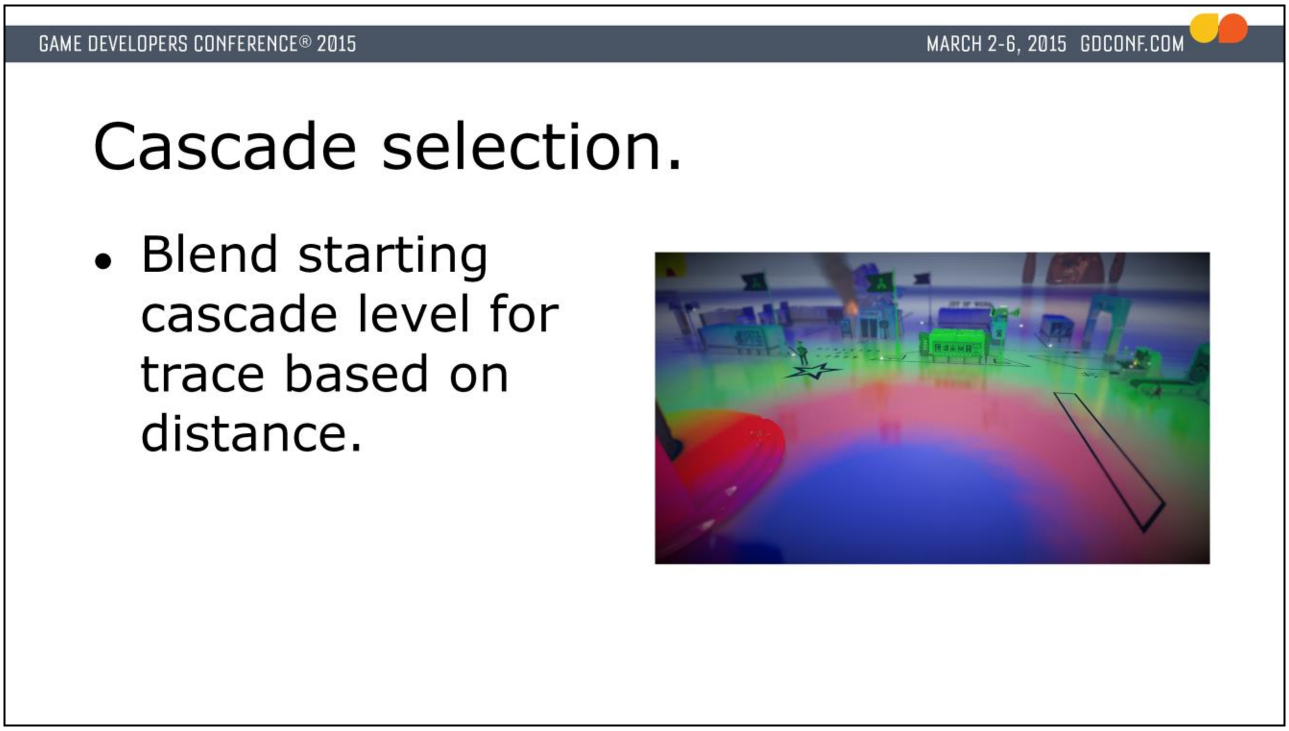
Cascade selection.



When we trace in screen space, we have to figure out which cascade level we should start our cone trace from.

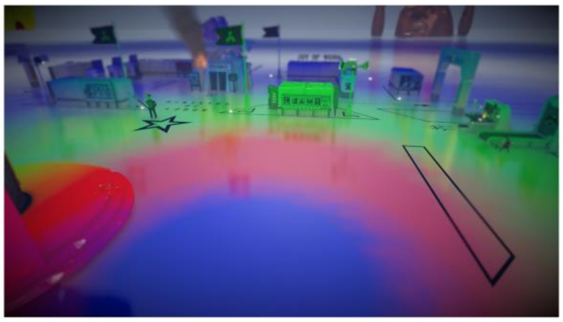
You can see in the picture, in which we have coloured the cascade levels we start from, that we are actually conservative in how we do that.

As we have concentric circles rather than concentric squares.



Cascade selection.

- Blend starting cascade level for trace based on distance.

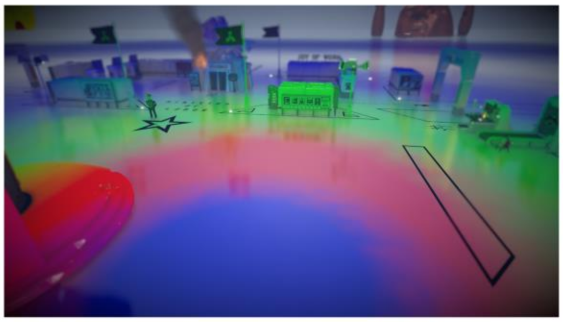


We use a distance based function, rather than a strict determination of the minimum cascade level we could possibly trace from.

This seems like we are wasting data, but we need to do this in order to ensure that we don’t have more detail in some directions than others.

Cascade selection.

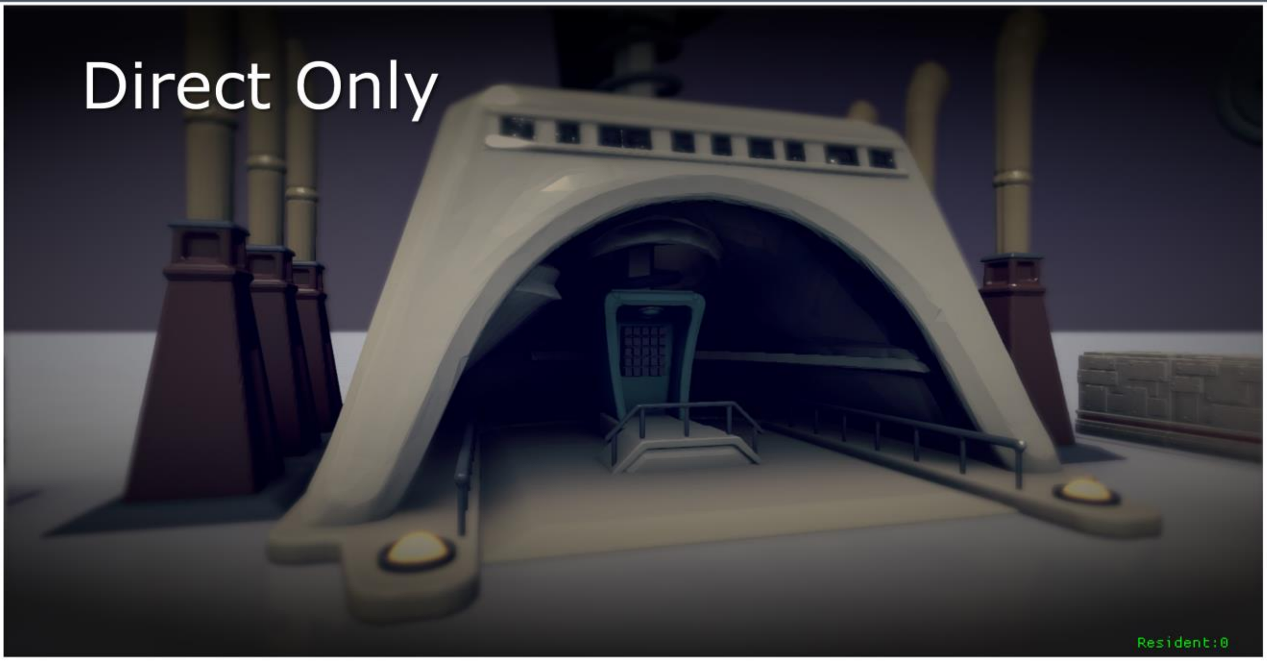
- Blend starting cascade level for trace based on distance.
- 3D noise offset



It’s also worth nothing that we have to smoothly blend between which cascade level we start from,

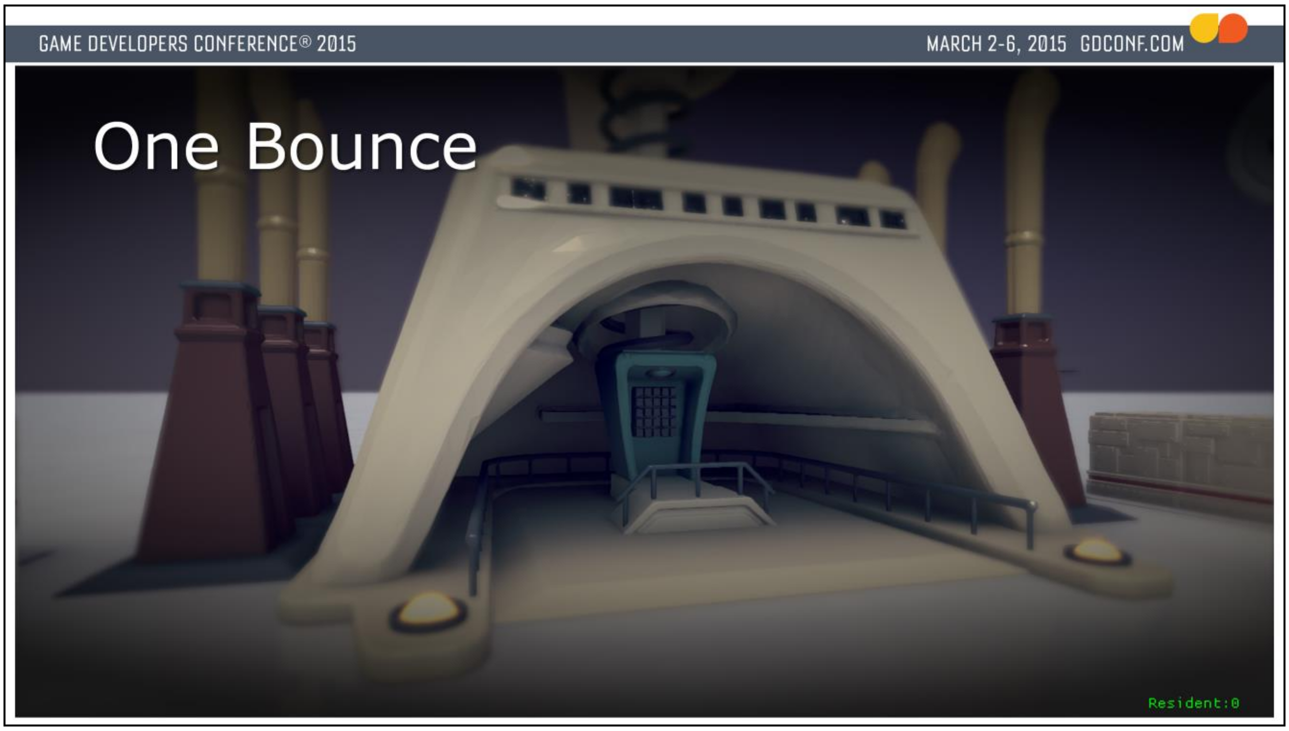
and that it also helps to add a little bit of noise to our blend here, so that its harder for the eye to pick up the transition as the cascades move over the landscape.

Direct Only



So know we know all the details, of how to calculate everything, how does this all look?

Here is our scene with just direct cone traced illumination.

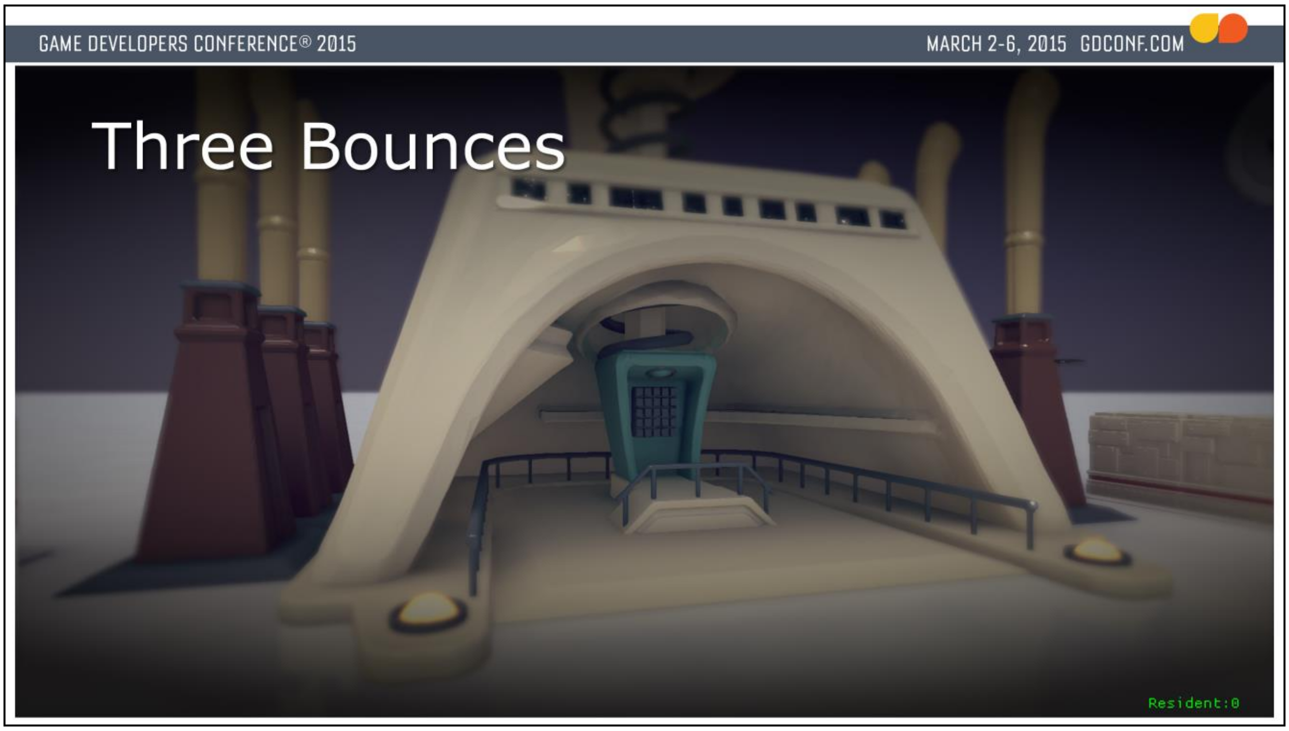

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

One Bounce

And you can see what starts happening as we add in the indirect lighting. Here we have just one bounce.

Two Bounces

And now two.

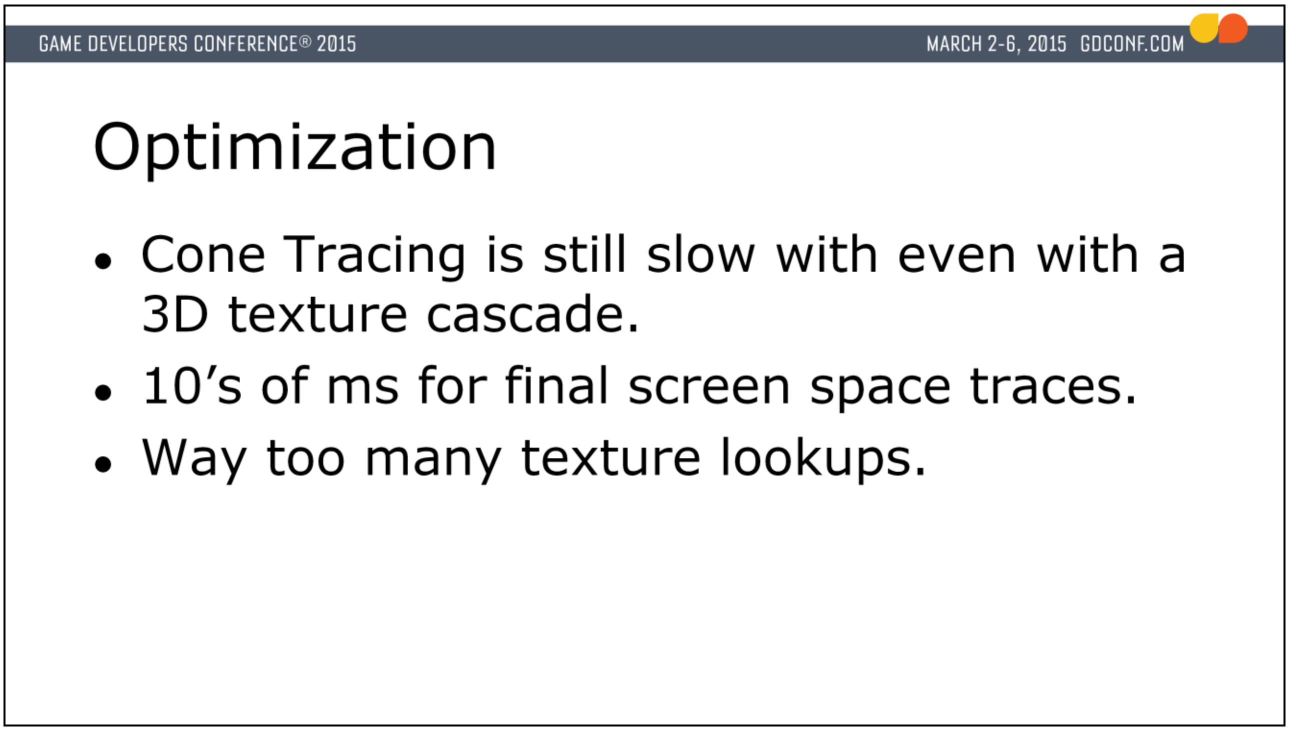

GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

Three Bounces

And three.

The third bounce BTW comes from our cone trace from our pixels.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Optimization

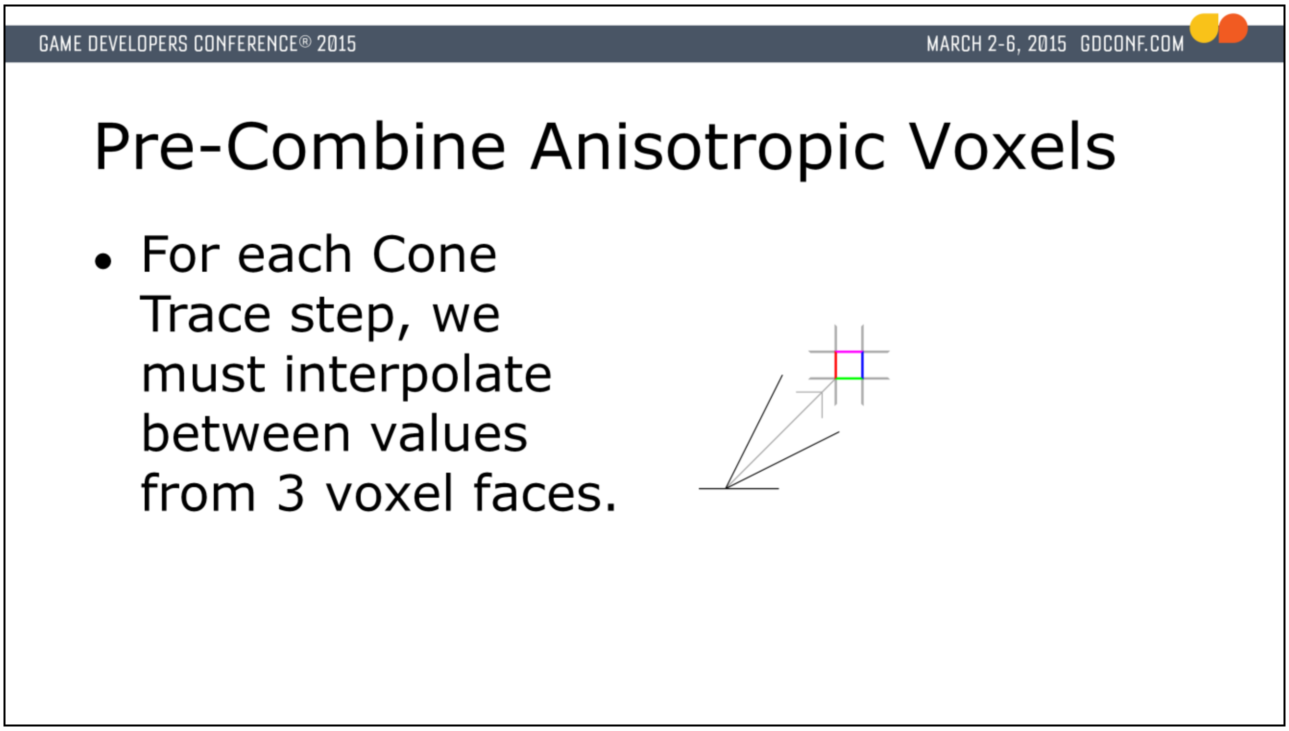
- Cone Tracing is still slow with even with a 3D texture cascade.
- 10’s of ms for final screen space traces.
- Way too many texture lookups.

So, what I’ve describes up to now all of this works,

but it still doesn’t run at speed that is practical on the PS4,

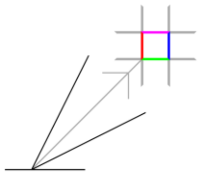
certainly not if we want to fit in 30ms and have time left over for anything else.

So we have had to cut a few more corner to get to a workable speed.



Pre-Combine Anisotropic Voxels

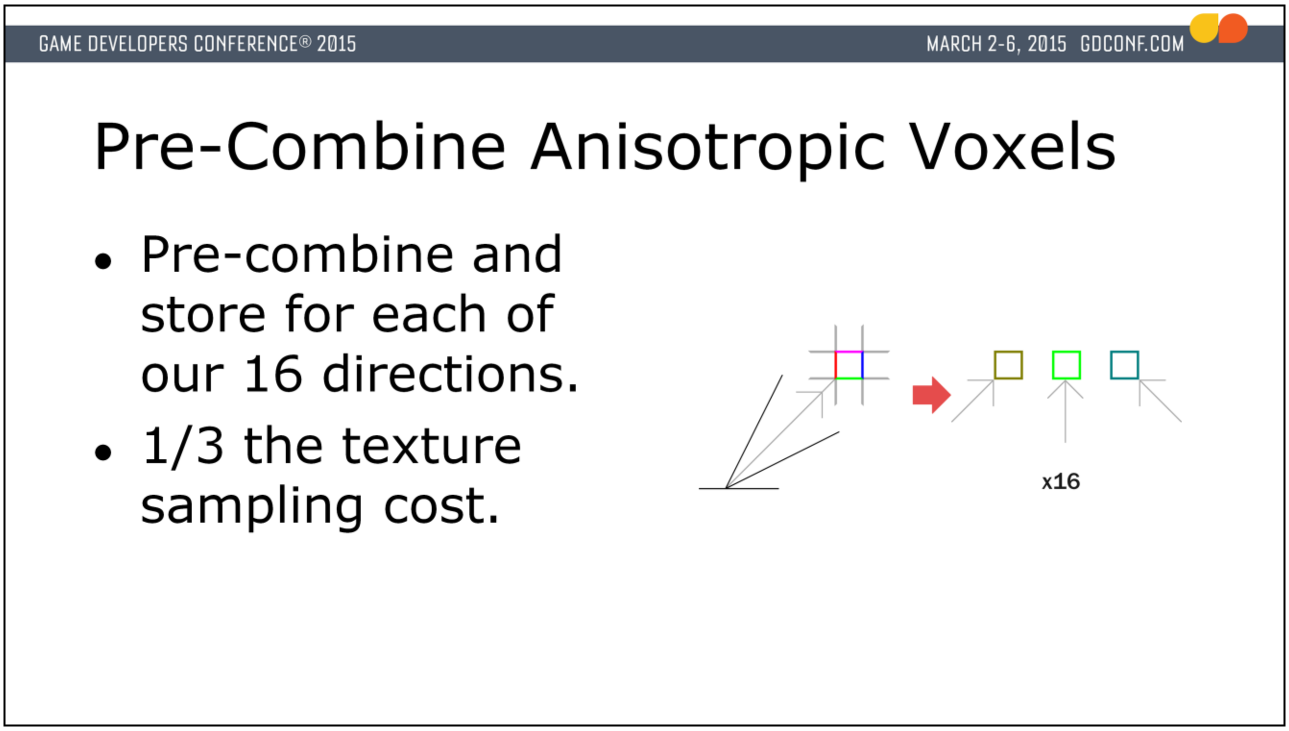
- For each Cone Trace step, we must interpolate between values from 3 voxel faces.



The first thing we did is note that because our voxels are anisotropic,

as we trace, we are constantly having to do an interpolation between the values of 3 face voxels.

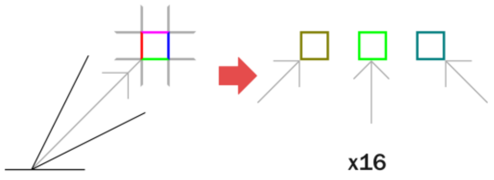
Which is quite costly in terms of the number of texture lookups.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Pre-Combine Anisotropic Voxels

- Pre-combine and store for each of our 16 directions.
- 1/3 the texture sampling cost.

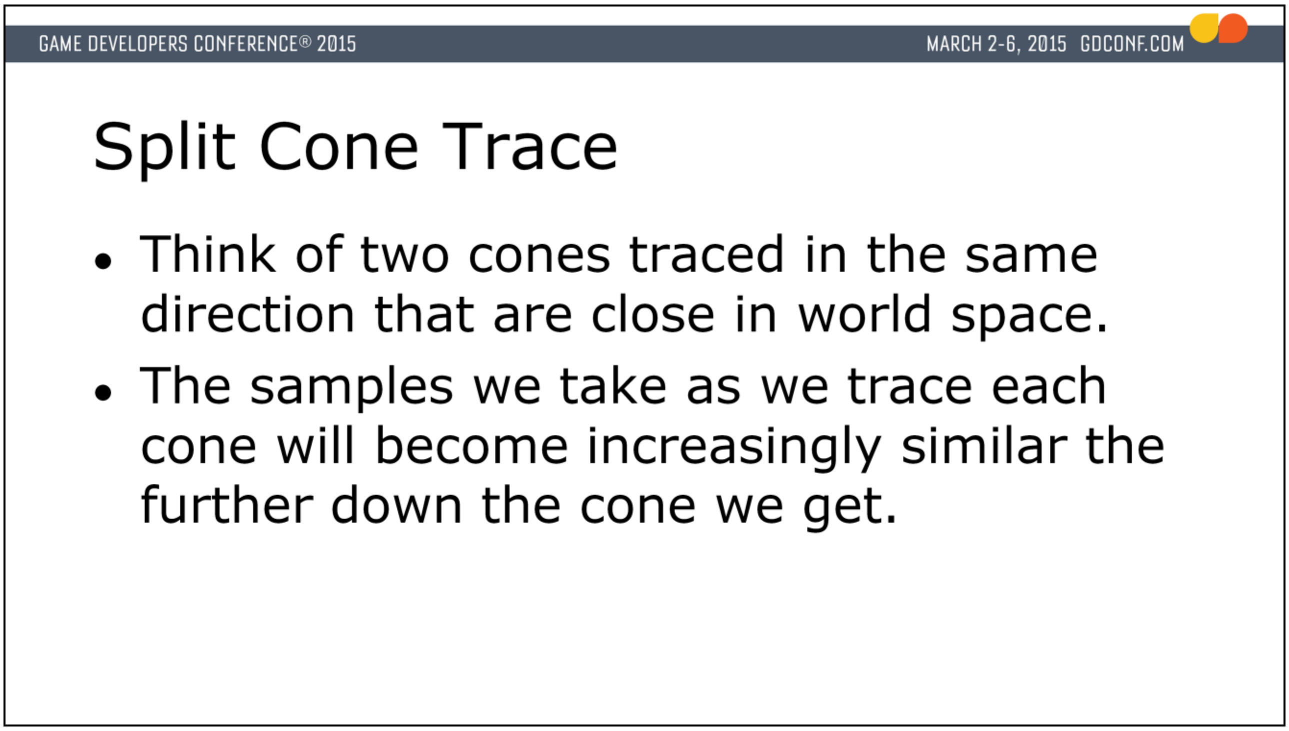


But because we have fixed the directions we will trace, we can actually just pre-combine these values for each of our 16 directions.

And store the whole thing in another texture.

There is a slight overhead for doing this, but it is relatively cheap, and it reduces the number of texture lookups we need for the

Cone tracing steps by a factor of 3.



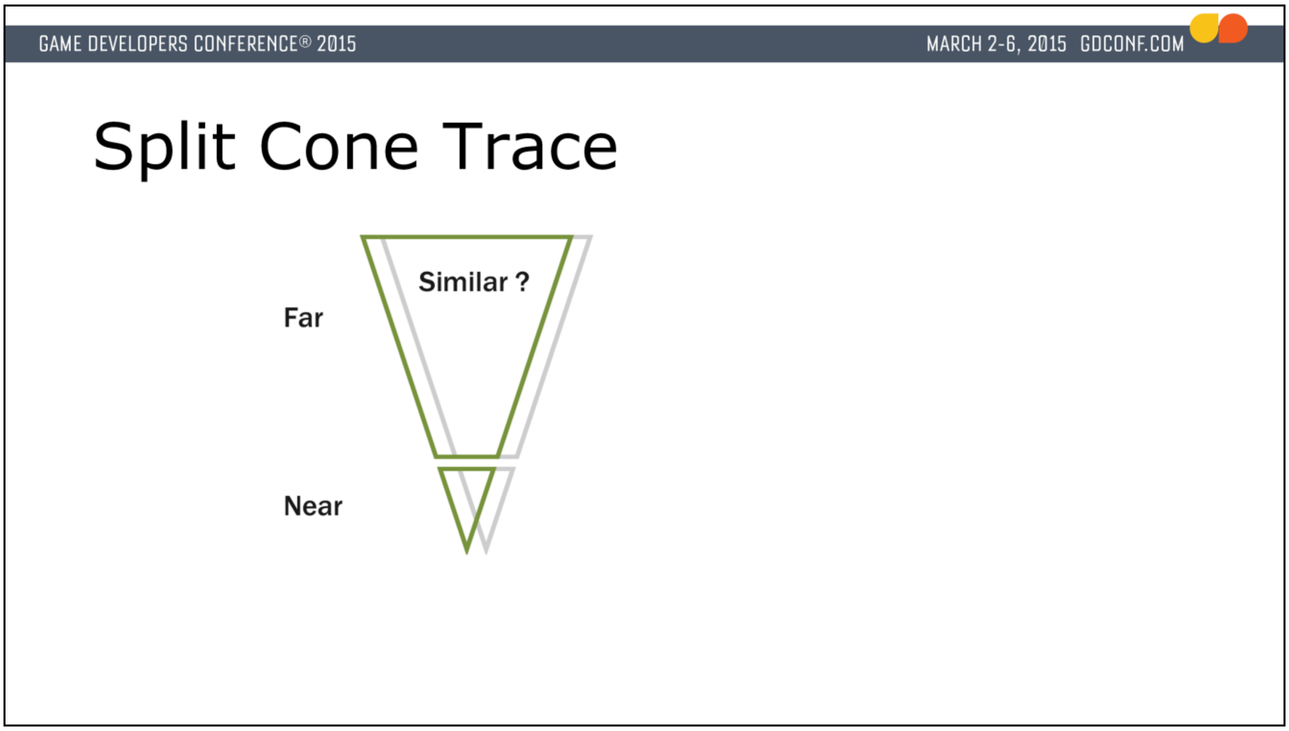
Split Cone Trace

- Think of two cones traced in the same direction that are close in world space.
- The samples we take as we trace each cone will become increasingly similar the further down the cone we get.

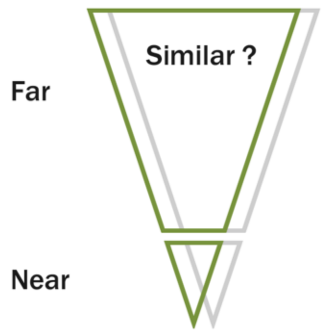
The second big optimization we make is simply to take advantage of parallax.

If we trace two cones in the same direction from a similar point in space,

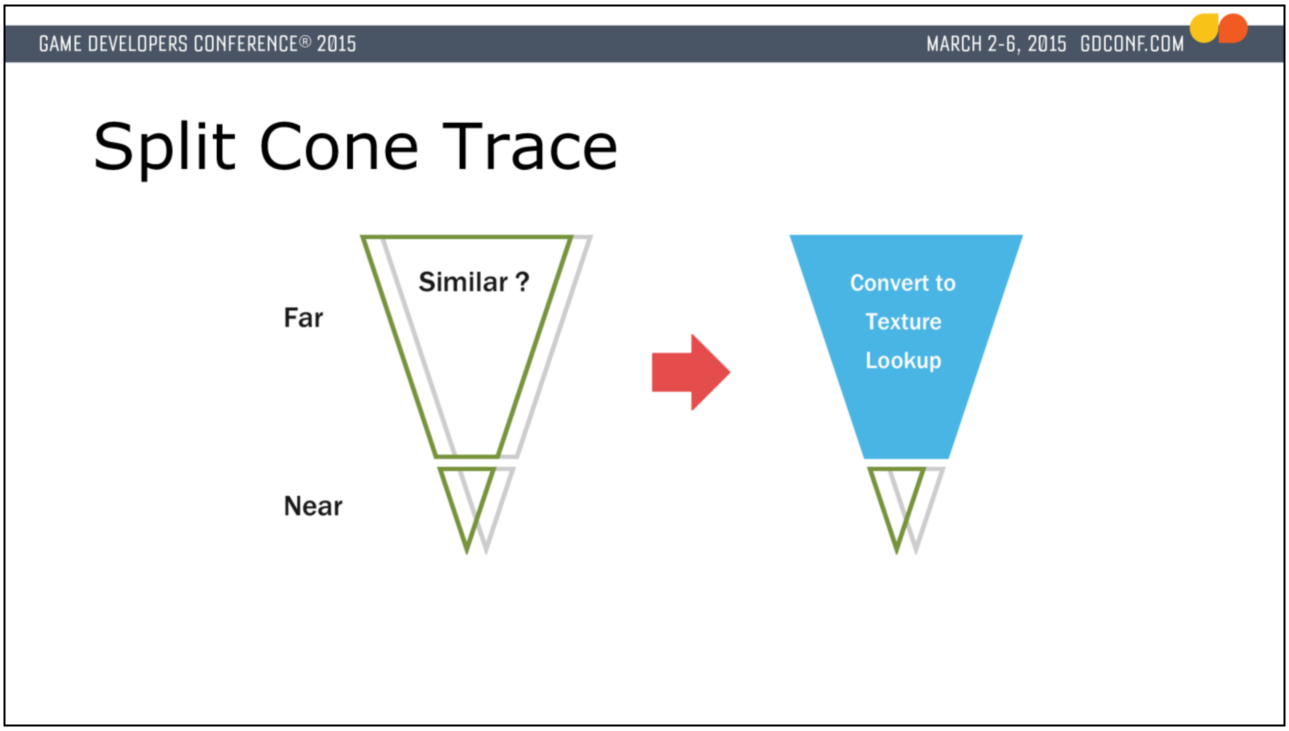
the voxel data we access becomes increasingly similar as we move towards the far end of the cones.



Split Cone Trace



So instead of tracing this data repeatedly, we trace this “far” cone data once, from the center of each voxel our cascades,


GAME DEVELOPERS CONFERENCE® 2015
MARCH 2-6, 2015 GDCONF.COM

Split Cone Trace

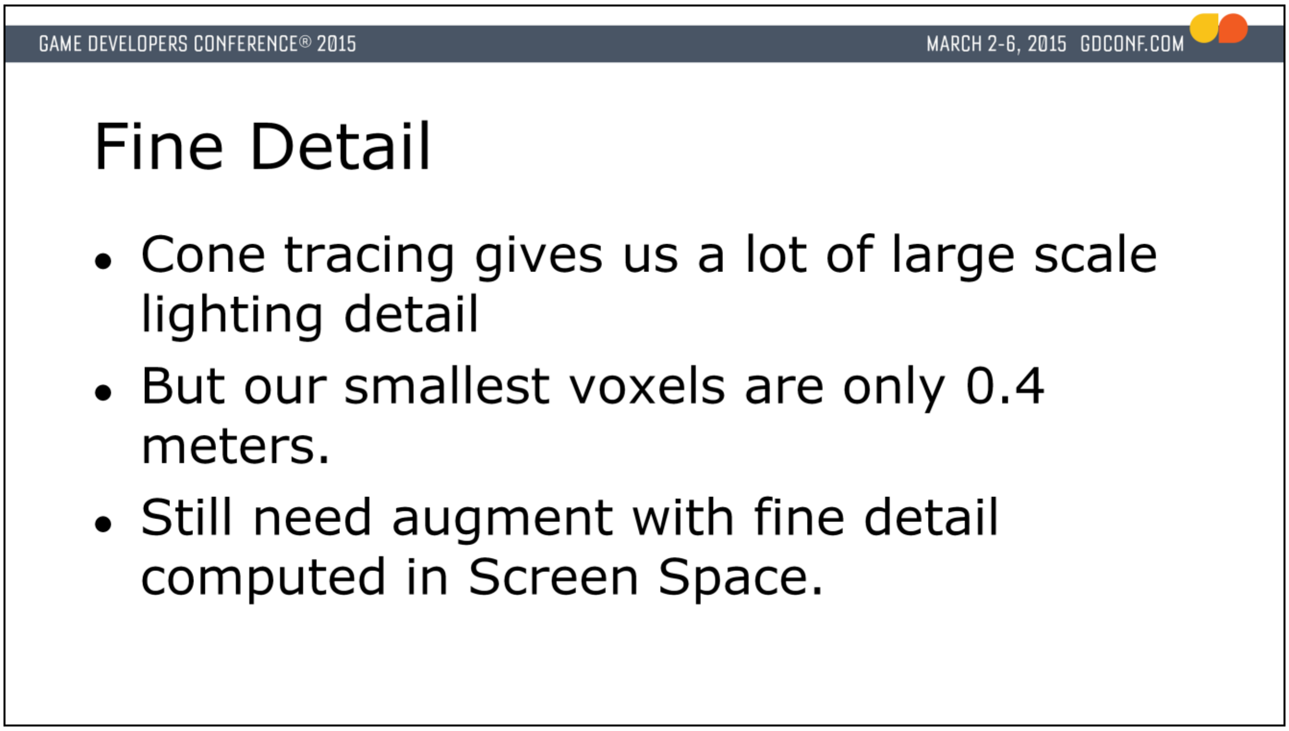
Far
Near
Similar ?
Convert to Texture Lookup

And store this in another texture cascade, which we can then trilinearly interpolate from in the future, to reconstruct this data.

Then we only need to trace the "near" part of the cone, and use our texture for the "far" part.

And of course we can tune where the "far" cone trace starts for the best balance between quality and speed.

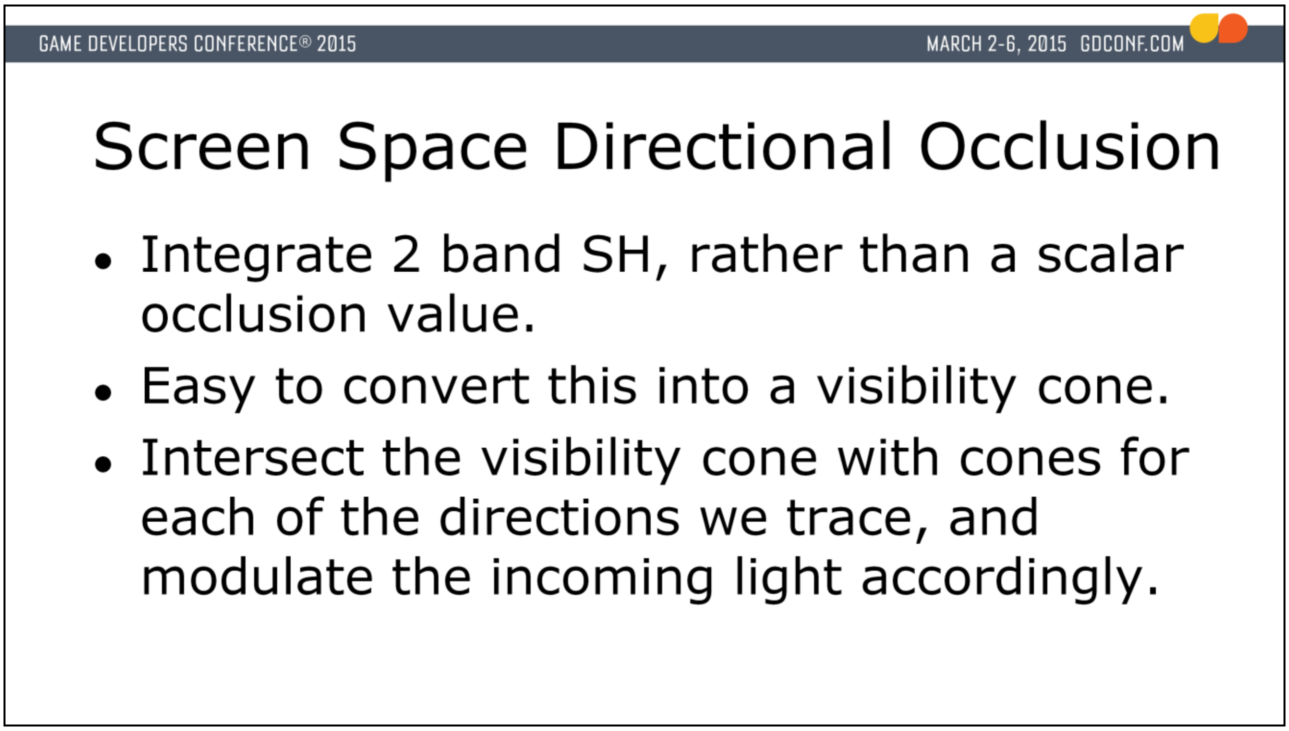
For us this is set to about a meter or so away for our closest cascade level.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Fine Detail

- Cone tracing gives us a lot of large scale lighting detail
- But our smallest voxels are only 0.4 meters.
- Still need augment with fine detail computed in Screen Space.

Cone tracing is great, but our implementations smallest voxel size is only 0.4m, so we need to augment this with extra detail computed in screen space.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Screen Space Directional Occlusion

- Integrate 2 band SH, rather than a scalar occlusion value.
- Easy to convert this into a visibility cone.
- Intersect the visibility cone with cones for each of the directions we trace, and modulate the incoming light accordingly.

So for this we need some screen space occlusion.

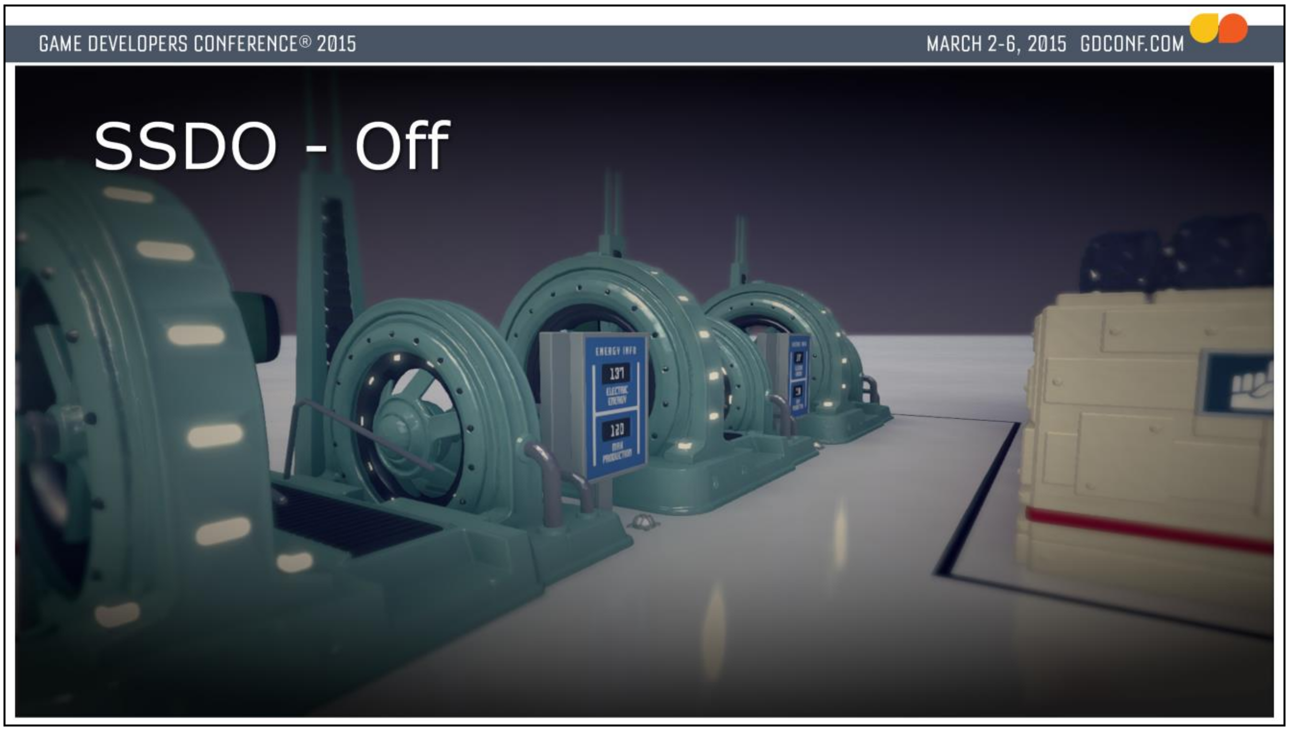
Note, that I left out the word ambient there,

We do something similar to Screen-Space Bent cones from GPU Pro 3

And integrate 2 band SH rather than a single scalar occlusion value.

This can be easily converted into a visibility cone at each pixel, that we can then intersect with the cones from our 16 trace directions

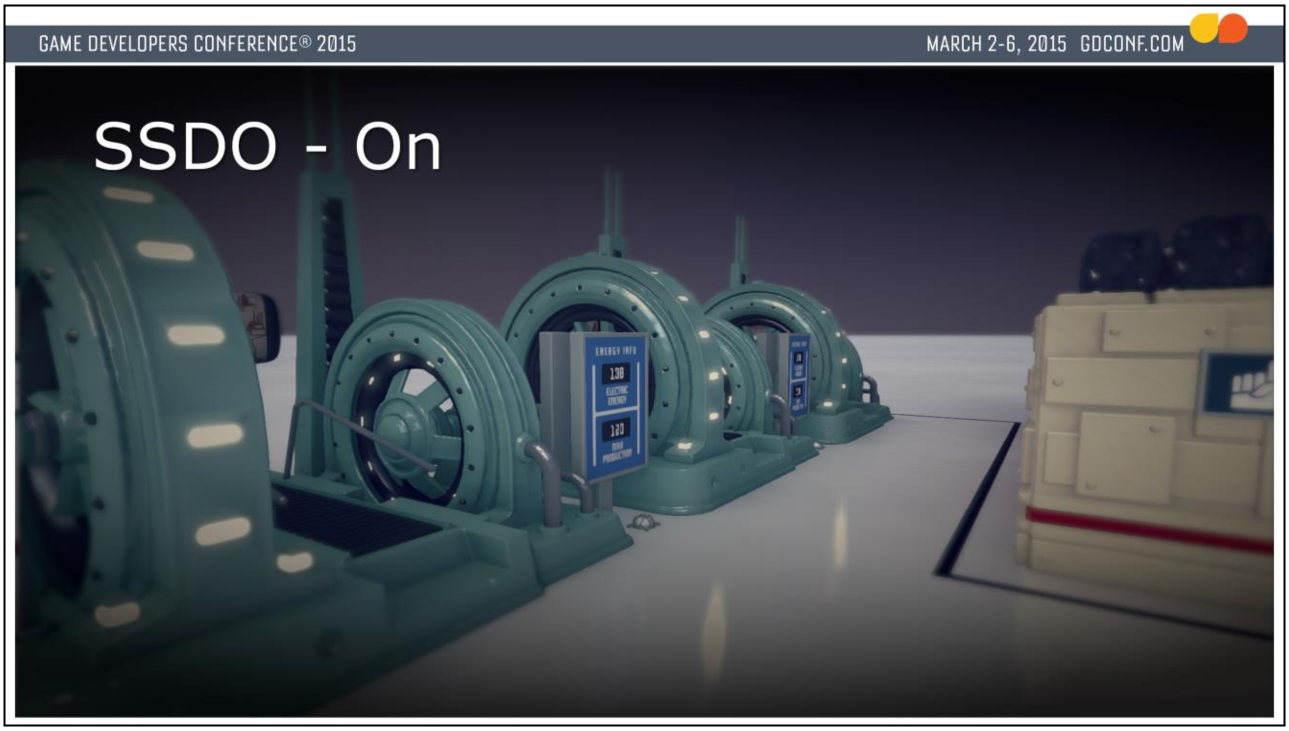
Keeping this occlusion directional rather than just a single scalar value really helps, especially when we deal with specular.


GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

SSDO - Off

So, here is an example scene without screen space occlusion



GAME DEVELOPERS CONFERENCE® 2015

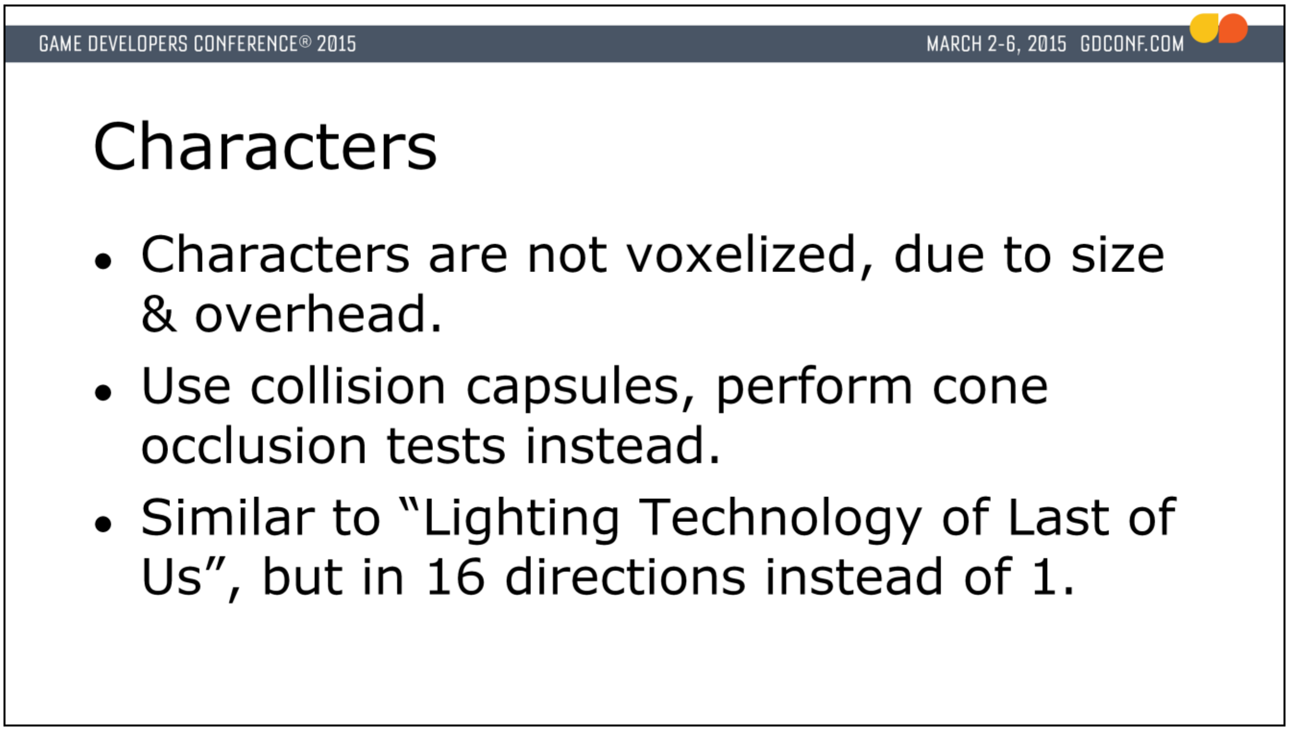
MARCH 2-6, 2015 GDCONF.COM

SSDO - On

And now with, you can see what a big difference that makes.

Characters

As I said earlier Characters are not voxelized due to the size of our voxels.
And the extra overhead it would cause
But we still want nice soft shadows from them


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

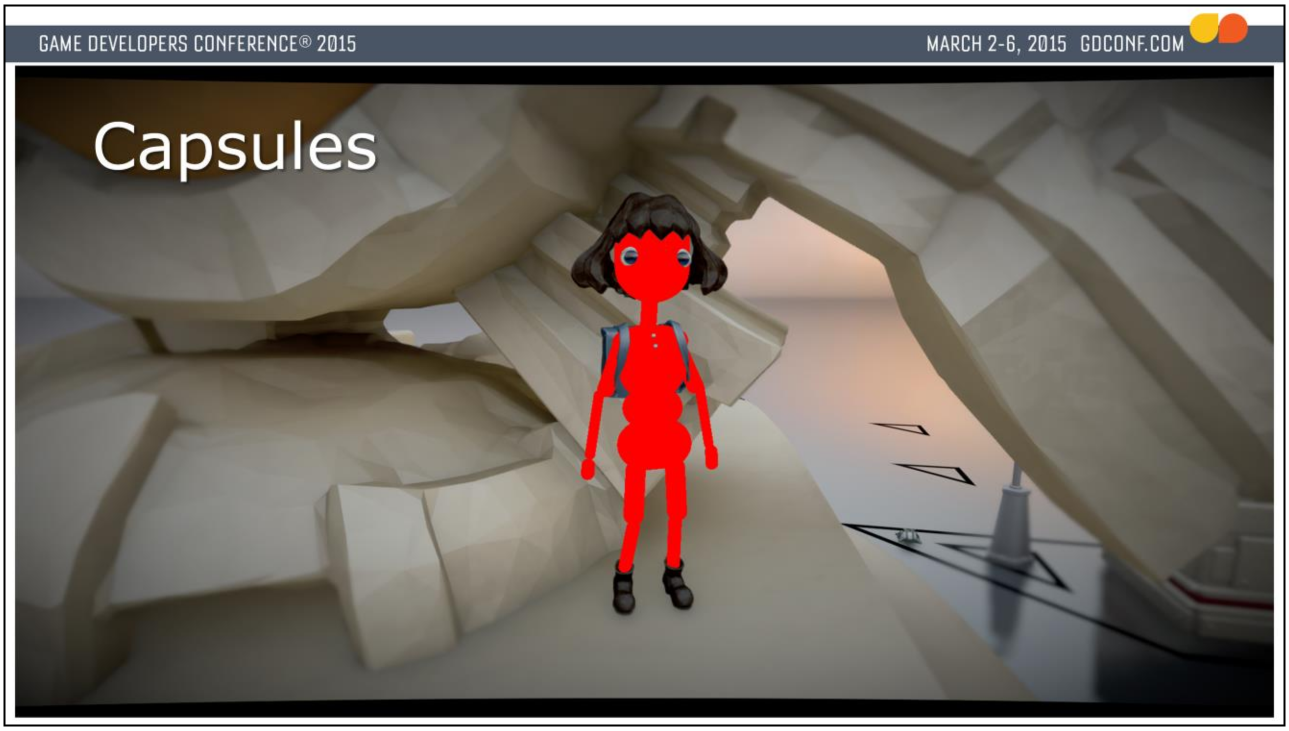
Characters

- Characters are not voxelized, due to size & overhead.
- Use collision capsules, perform cone occlusion tests instead.
- Similar to "Lighting Technology of Last of Us", but in 16 directions instead of 1.

So we use the characters collision volumes, to generate occlusion in a very similar way to what was done in The Last of Us.

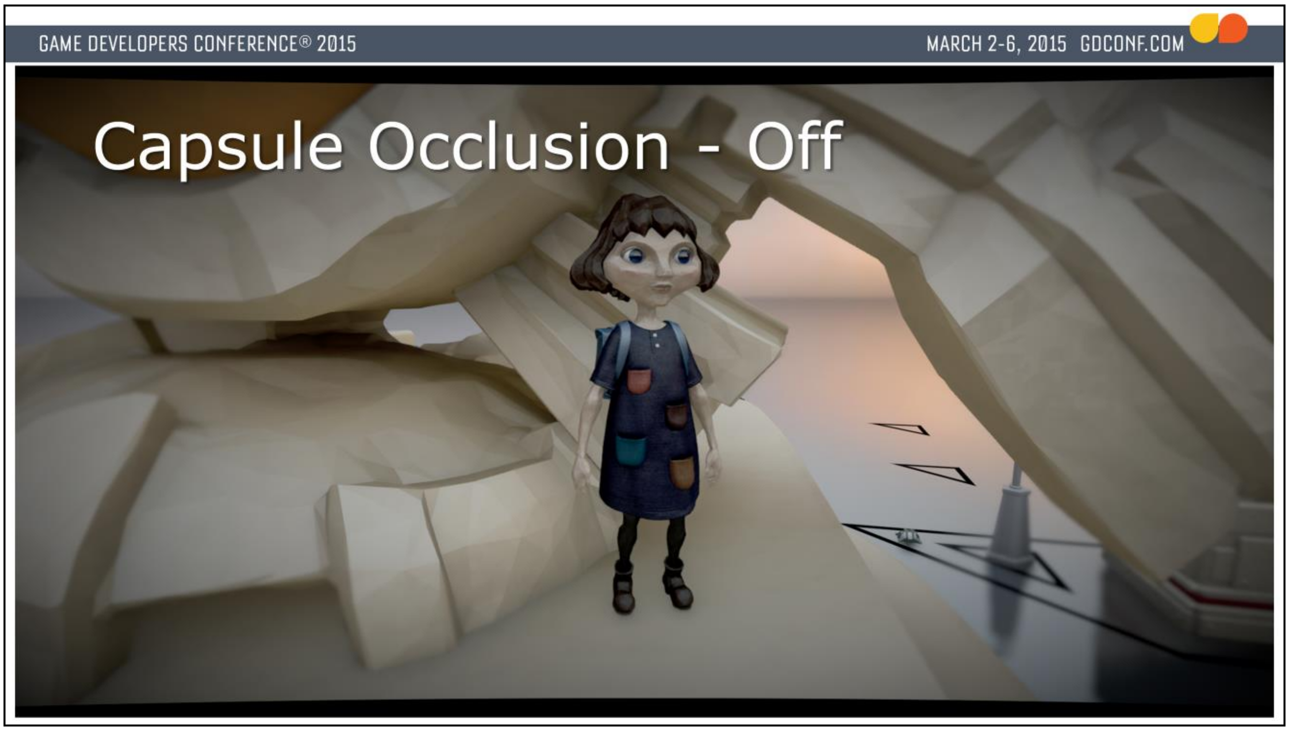
Except that we have to do a cone overlap test in each of our 16 cone tracing directions, rather than just a single primary direction.

The result of this is a 16 layer deep screen space texture array which we will use in our final combine step to modulate our lighting results.



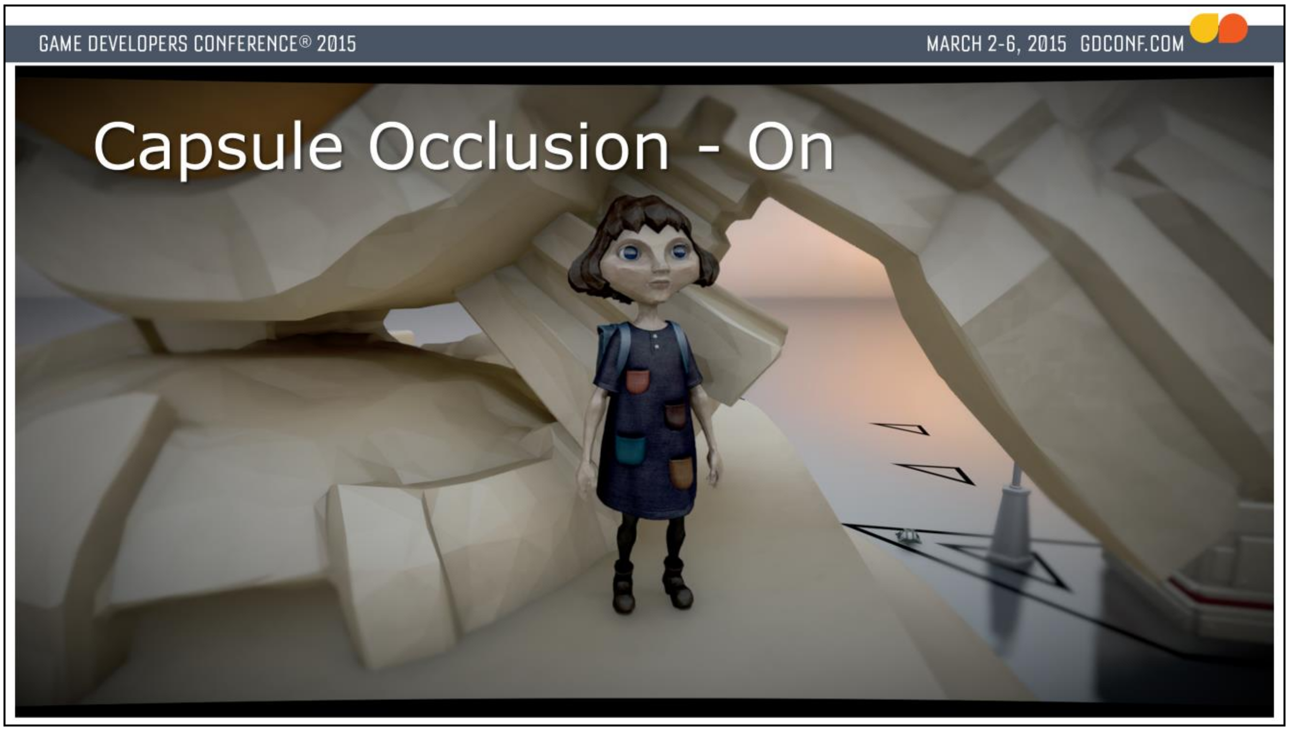
Capsules

Here you can see the volumes we are using for the main character.



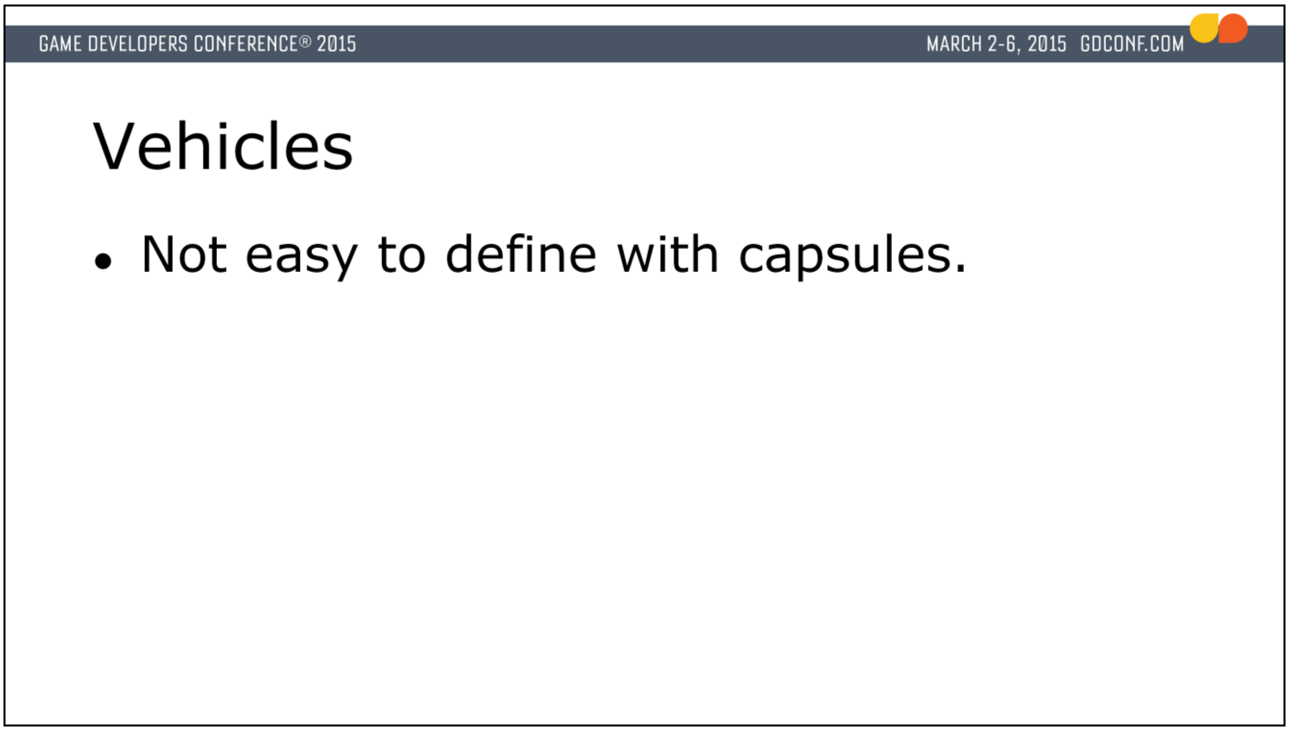
Capsule Occlusion - Off

As you can see that if we use these for occlusion,



Capsule Occlusion - On

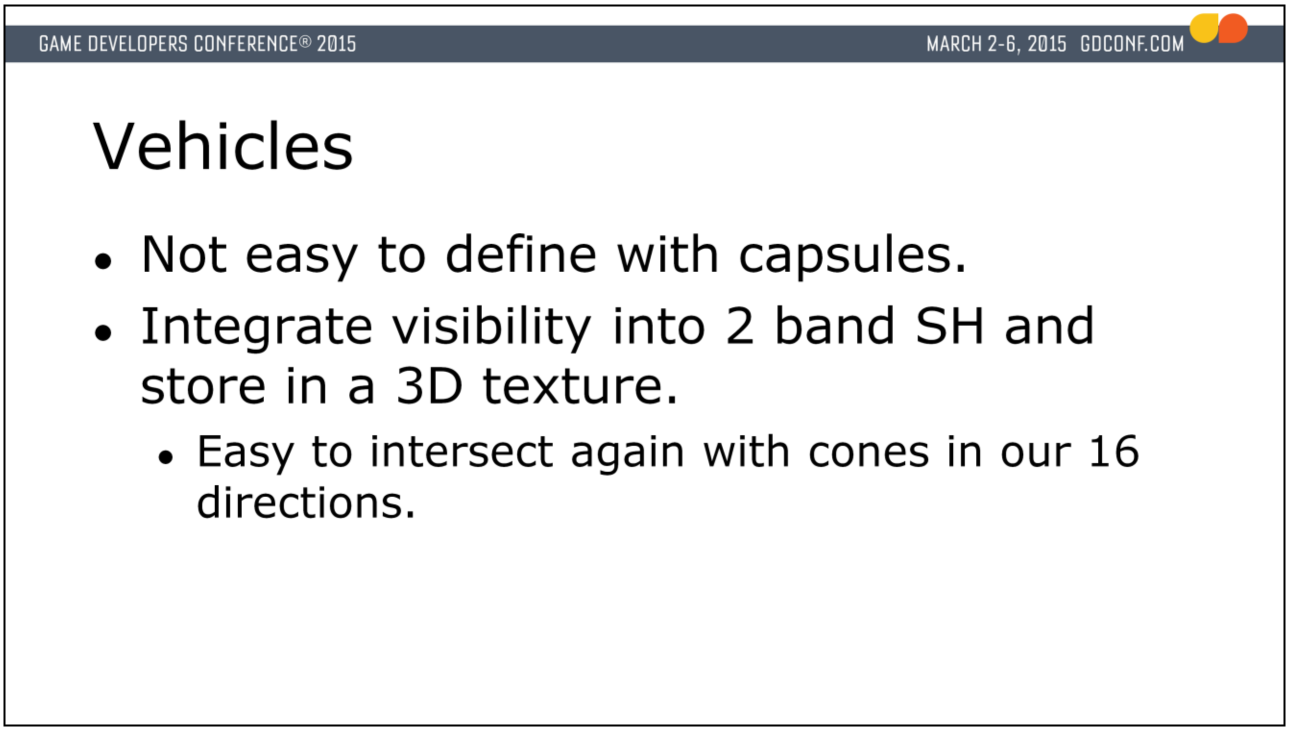
Then we get something that helps the character feel much more rooted in the world


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Vehicles

- Not easy to define with capsules.

So that takes care of characters, but capsules are not a very good fit for some of the other dynamic things we have in the game, namely vehicles


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Vehicles

- Not easy to define with capsules.
- Integrate visibility into 2 band SH and store in a 3D texture.
  - Easy to intersect again with cones in our 16 directions.

For these we instead build a 3D texture containing 2 bands of SH coefficients that describe a visibility fn which we can then use to intersect with the cones from our 16 directions.

Again, this outputs to the same 16 layer deep texture array as the characters.

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

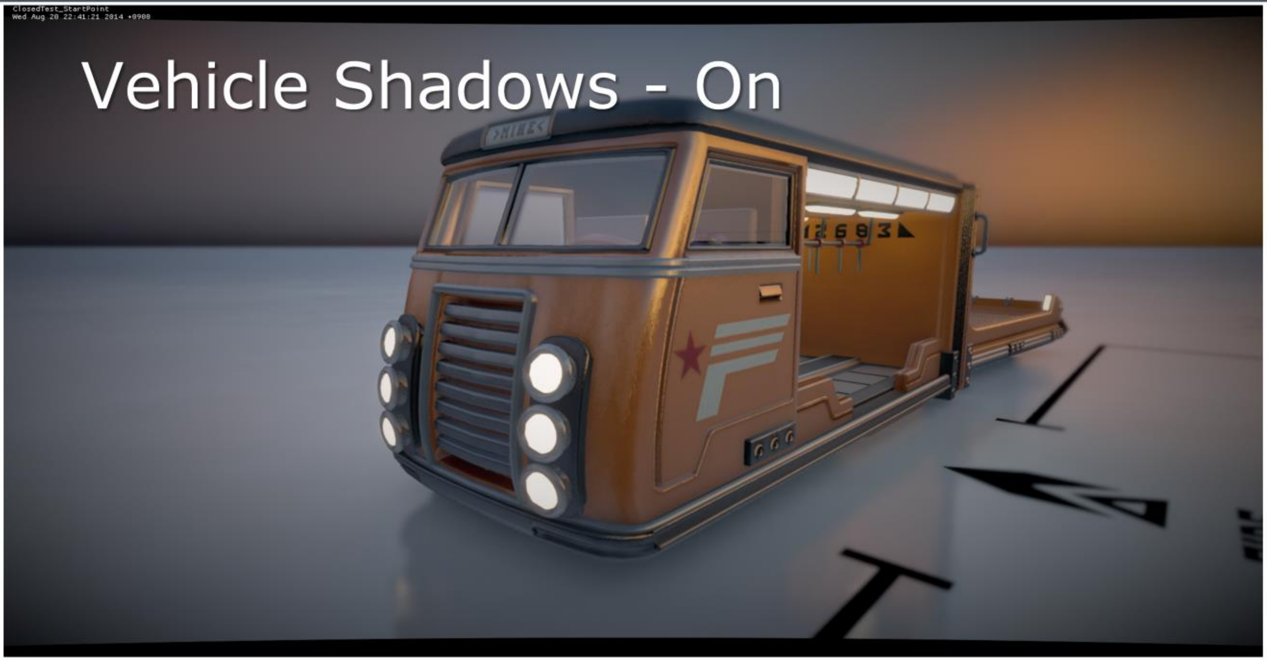
Vehicle Shadows - Off

ClosedTest_StartPoint
Wed Aug 20 22:41:21 2014 +0900

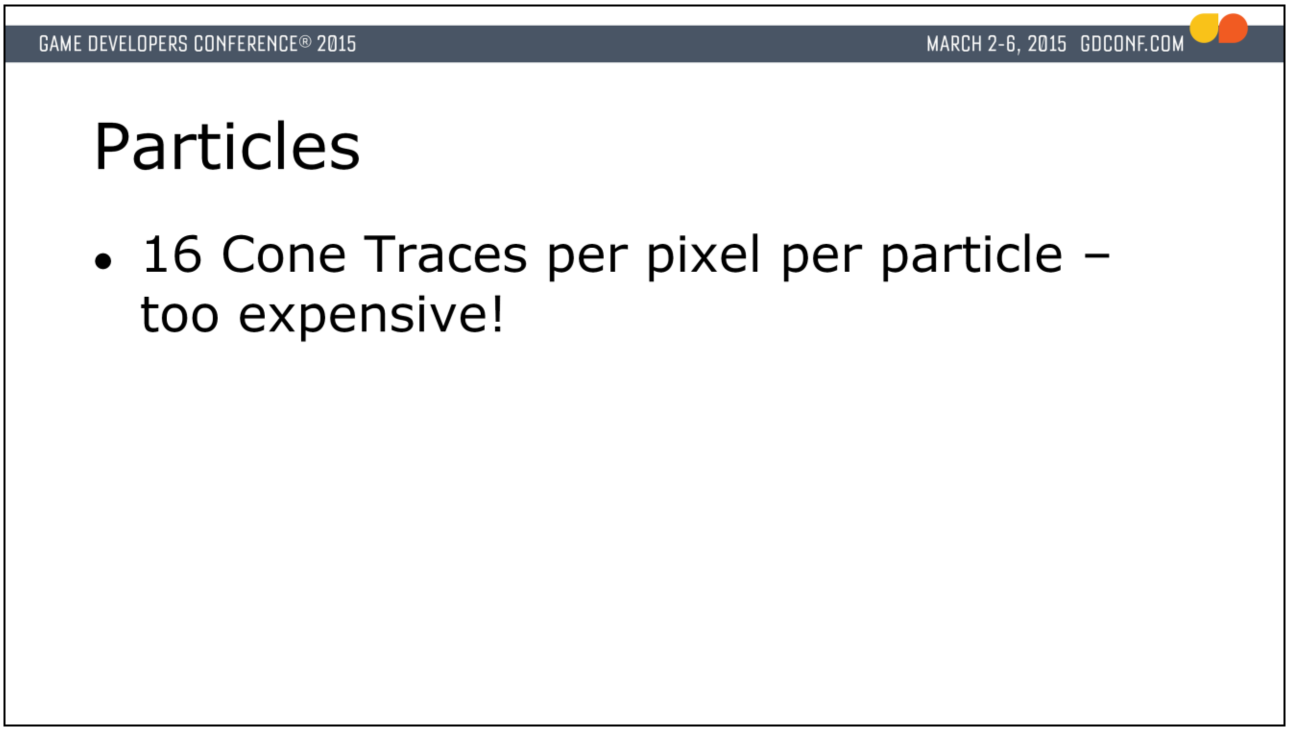
So here you can see our bus, without any occlusions

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Vehicle Shadows - On



And now with.

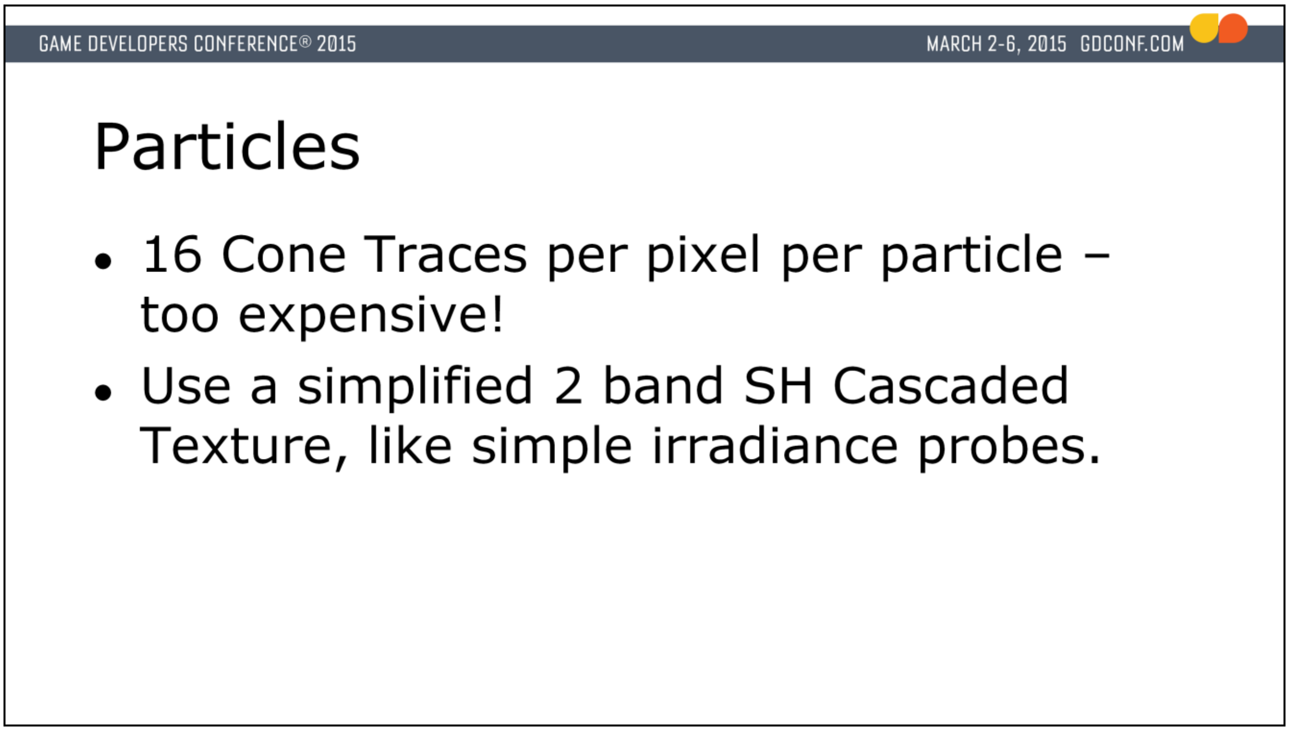


Particles

- 16 Cone Traces per pixel per particle – too expensive!

So, we also have to deal with lighting transparent objects such as particles.

But the potential cost of doing 16 cone traces per pixel per particle, are just too expensive to really be viable.



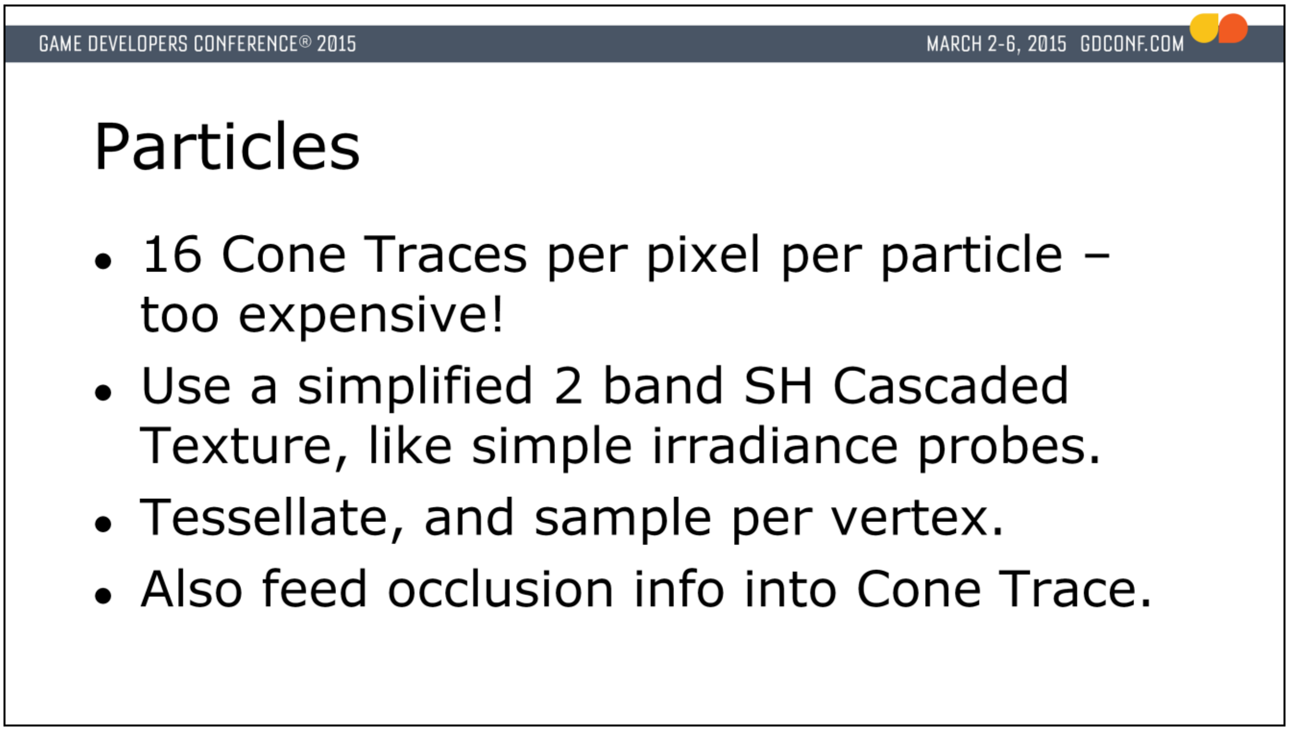
Particles

- 16 Cone Traces per pixel per particle – too expensive!
- Use a simplified 2 band SH Cascaded Texture, like simple irradiance probes.

What we do instead (and this might be getting familiar to you now) is build another texture!

We can cone trace from the center of each voxel in our cascades, and get ourselves a texture where for any point in space we can query and get a rough approximation of the incoming light in all 16 directions.

But 16 texture lookups is still way too expensive, so we then encode this texture as 2 band SH, which gets us down to just 3 texture lookups.

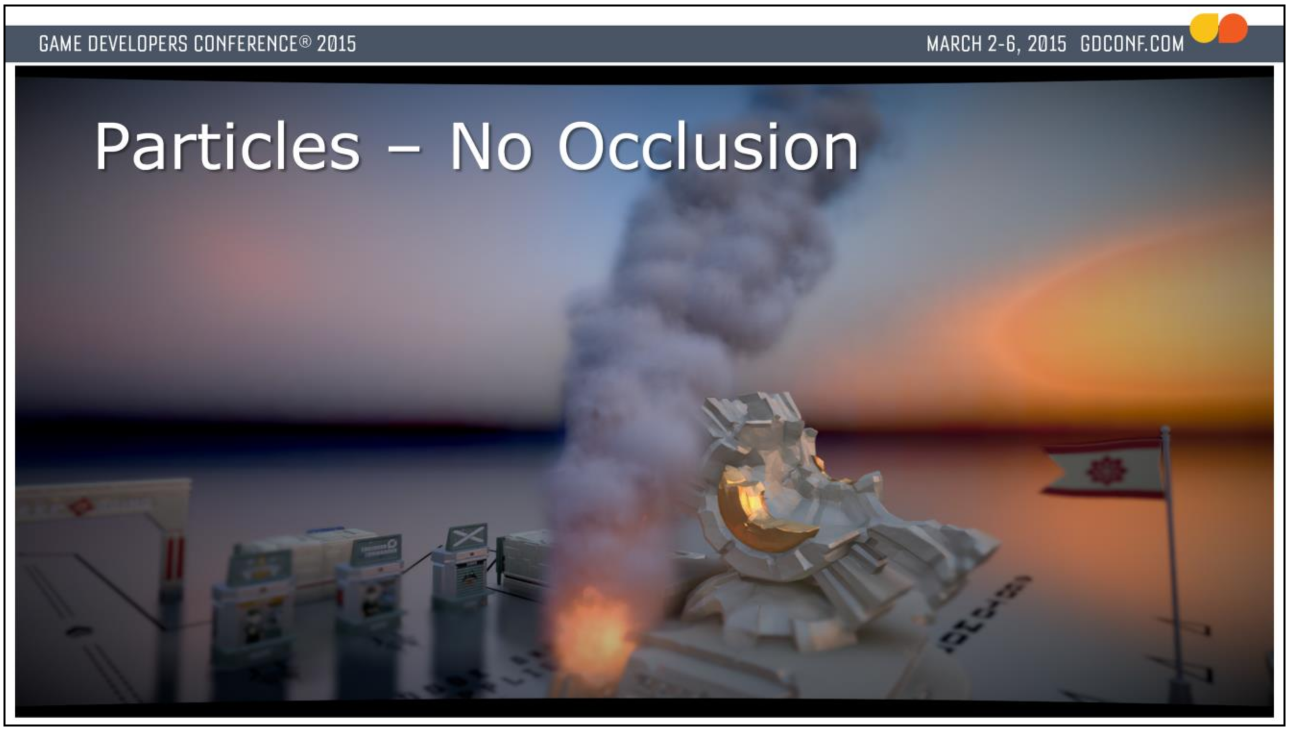

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Particles

- 16 Cone Traces per pixel per particle – too expensive!
- Use a simplified 2 band SH Cascaded Texture, like simple irradiance probes.
- Tessellate, and sample per vertex.
- Also feed occlusion info into Cone Trace.

We then tessellate our particles, and sample from this texture per vertex.

We also have the particles fill a dynamic occlusion texture, which is fed back into the cone tracing,



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Particles – No Occlusion

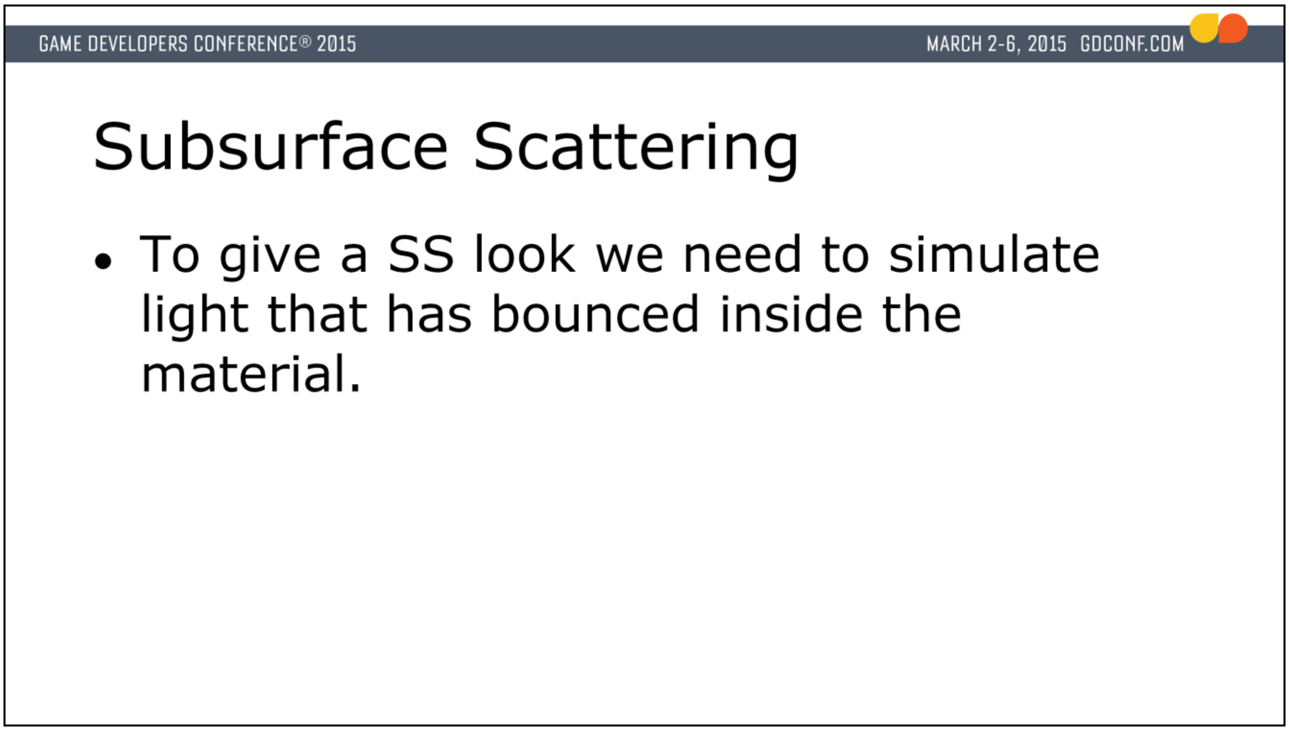
So here you can see some smoke without occlusion

GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

Particles with Dynamic Occlusion

And you can see how if I turn on the occluson
it allows the smoke to have a much more of a volumetric feel
And a sense of presence.



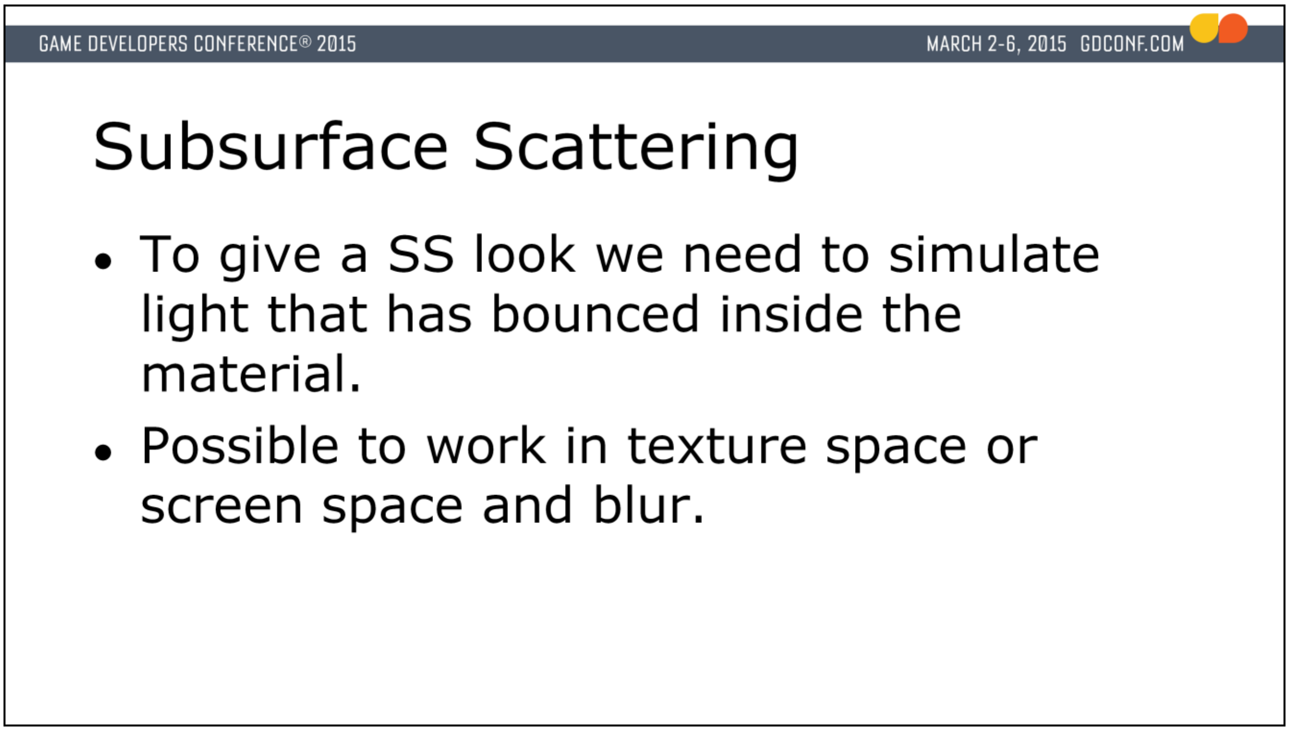
Subsurface Scattering

- To give a SS look we need to simulate light that has bounced inside the material.

The nice thing about having this SH texture is that we can use it for other things.

Typically Subsurface scattering is a difficult thing to simulate in realtime.

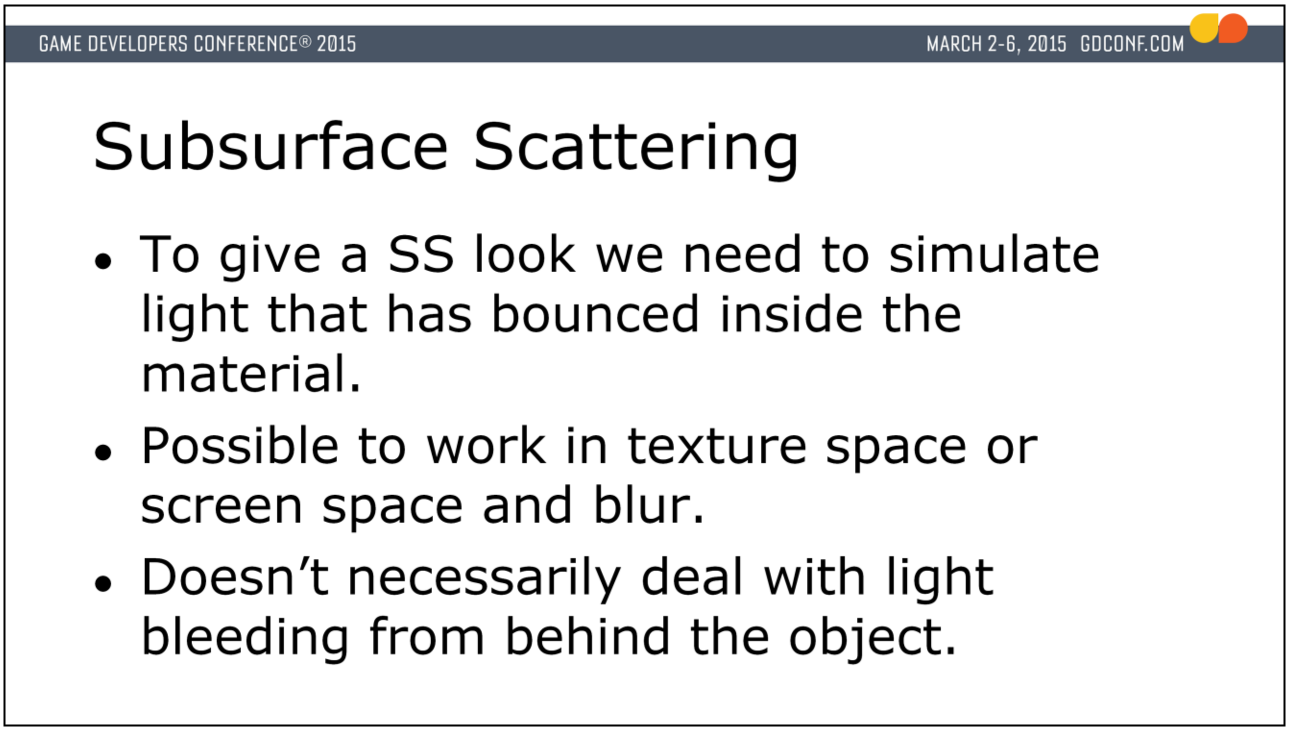
We need to simulate light entering an object at multiple different points, and bouncing around inside the material before exiting at the point seen by the viewer.



Subsurface Scattering

- To give a SS look we need to simulate light that has bounced inside the material.
- Possible to work in texture space or screen space and blur.

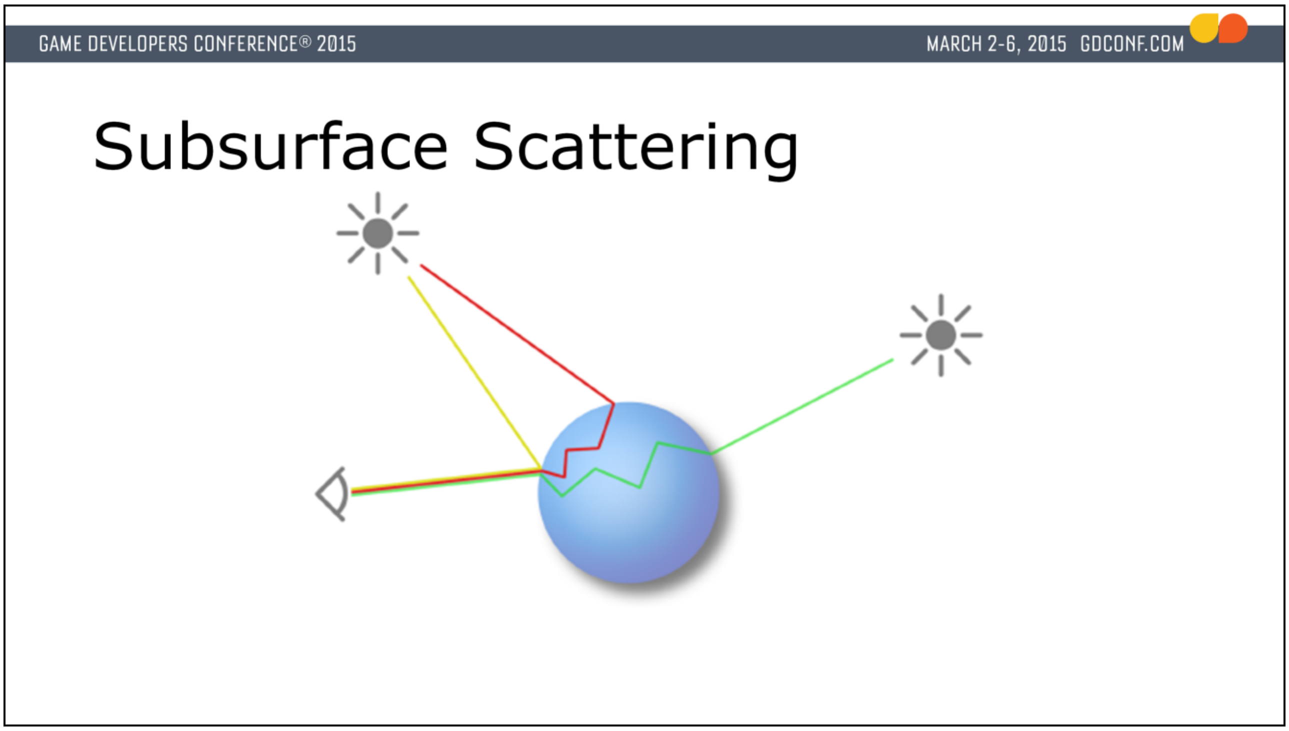
One possible approach that has been tried is to blur the lighting information in either texture space, or in a geometry aware way in screen space.



Subsurface Scattering

- To give a SS look we need to simulate light that has bounced inside the material.
- Possible to work in texture space or screen space and blur.
- Doesn’t necessarily deal with light bleeding from behind the object.

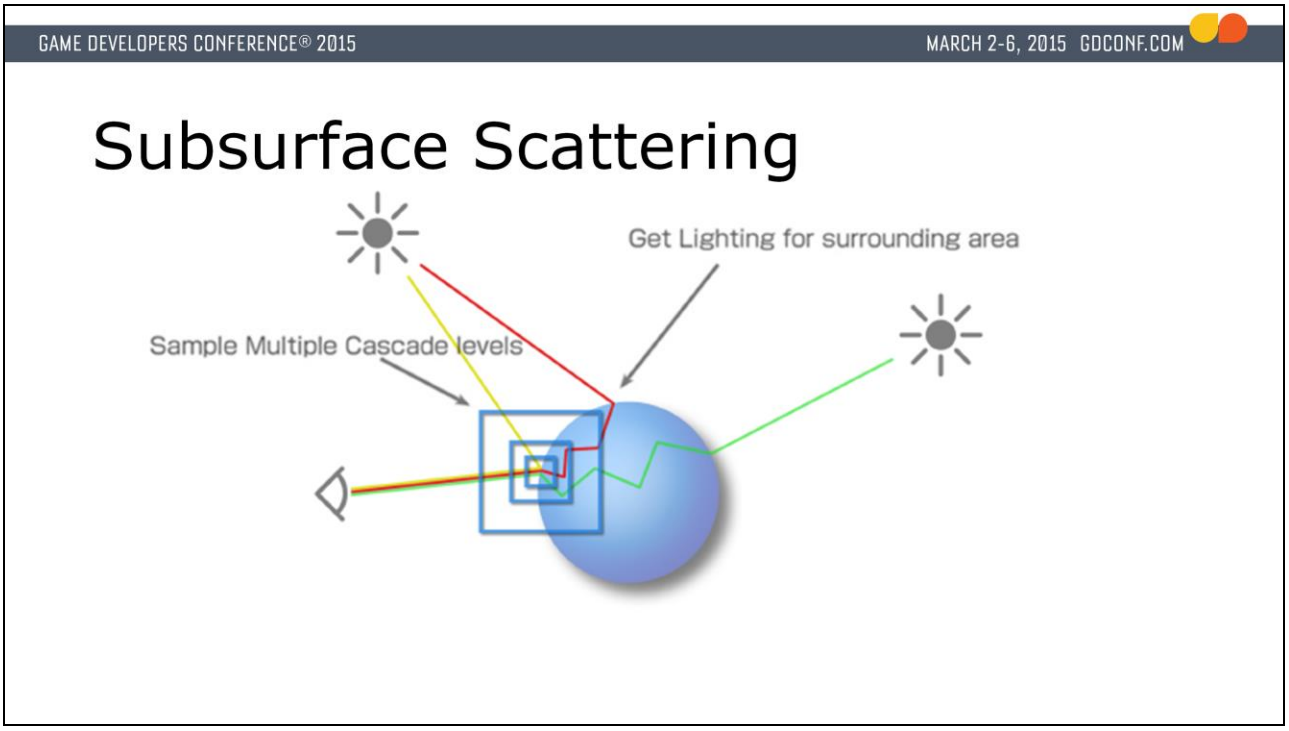
This works to some degree, but doesn’t help us that much with lights that are behind the object.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Subsurface Scattering

So we have some chance of getting some good results for the red ray in our diagram, but not the green one.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Subsurface Scattering

Get Lighting for surrounding area
Sample Multiple Cascade levels

Thankfully the SH texture we built for particles gives us another way to tackle this problem.

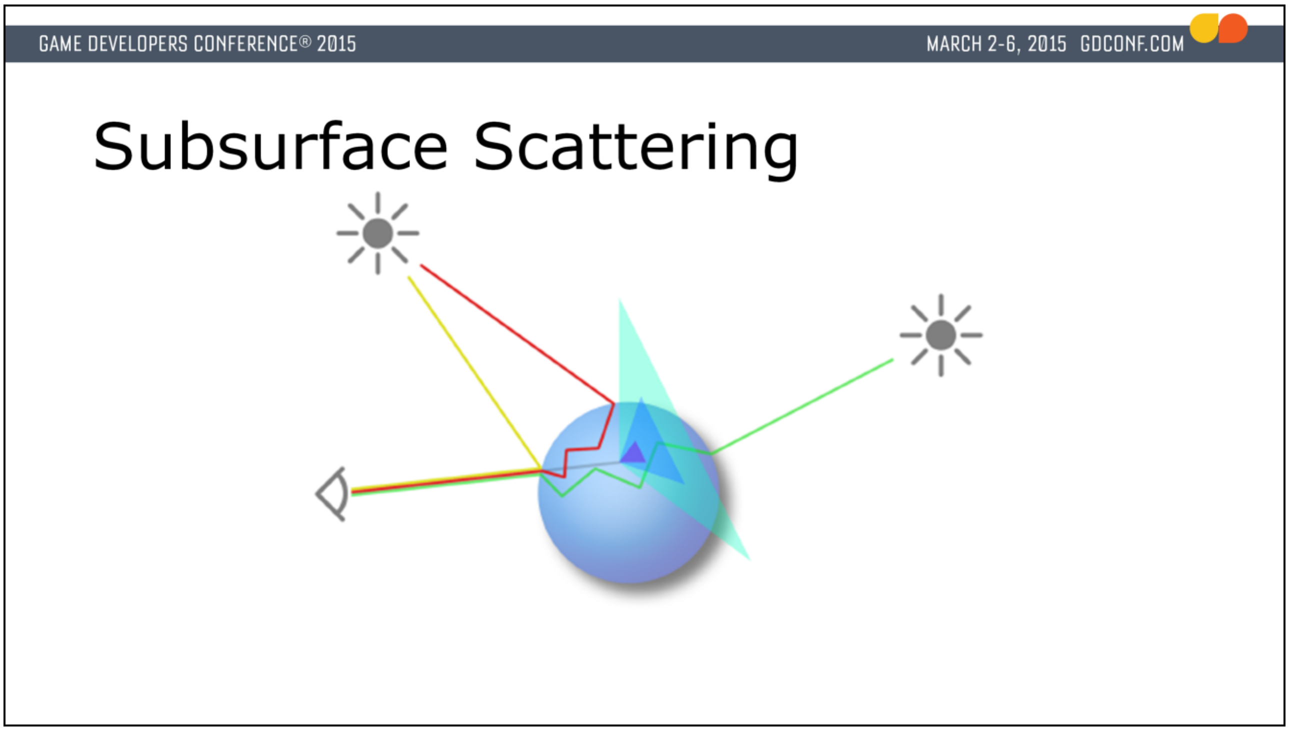
The texture we have build is effectively a light field for the scene that we can sample at any point in space,

And can quickly give us the irradiance at any point that we're interested in.

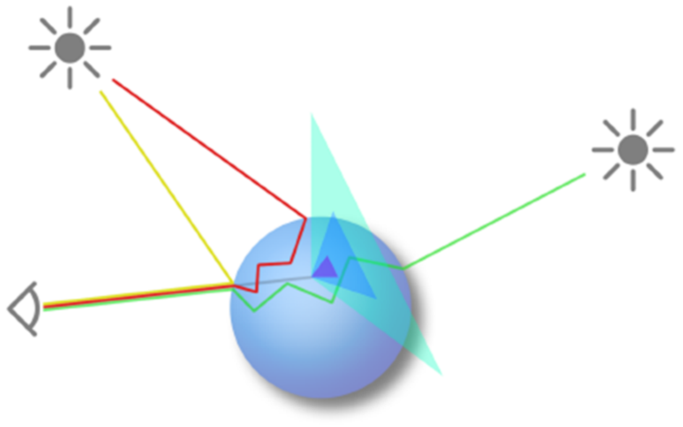
Also, going up the cascade levels gives us the information about the incident lighting over a wider and wider region of space.

So we can simply take a weighted average of the irradiance from different cascade levels, and use that to gather light that doesn't directly hit our viewpoint.

In our implementation we sample over the whole sphere, effectively just taking the 1st SH band, to accumulate this lighting, which we call the "static SS" shading.



Subsurface Scattering



We can also give a directional effect by ray marching through the object away from the viewer, sampling from different cascade levels as we go, gathering light using a projected SH cone in the direction of the ray march.

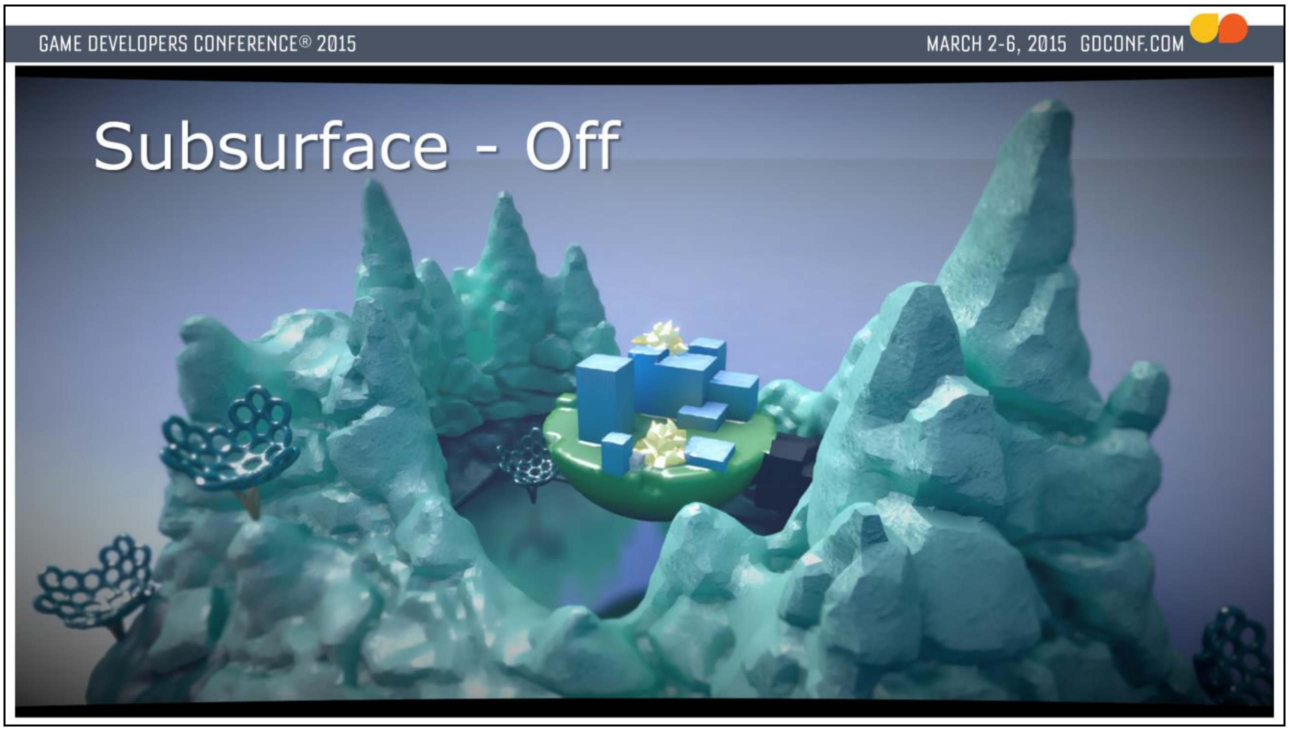
This we call our "directional SS" term.

To get a rich range of material looks, we interpolate between the normal diffuse lighting we get from our cone tracing, and these two sub surface lighting results.

We also add a parameter we call "frosting", which uses a screen space local thickness parameter we calculate along with our SSDO to modulate the albedo of the material to help accentuate the look.

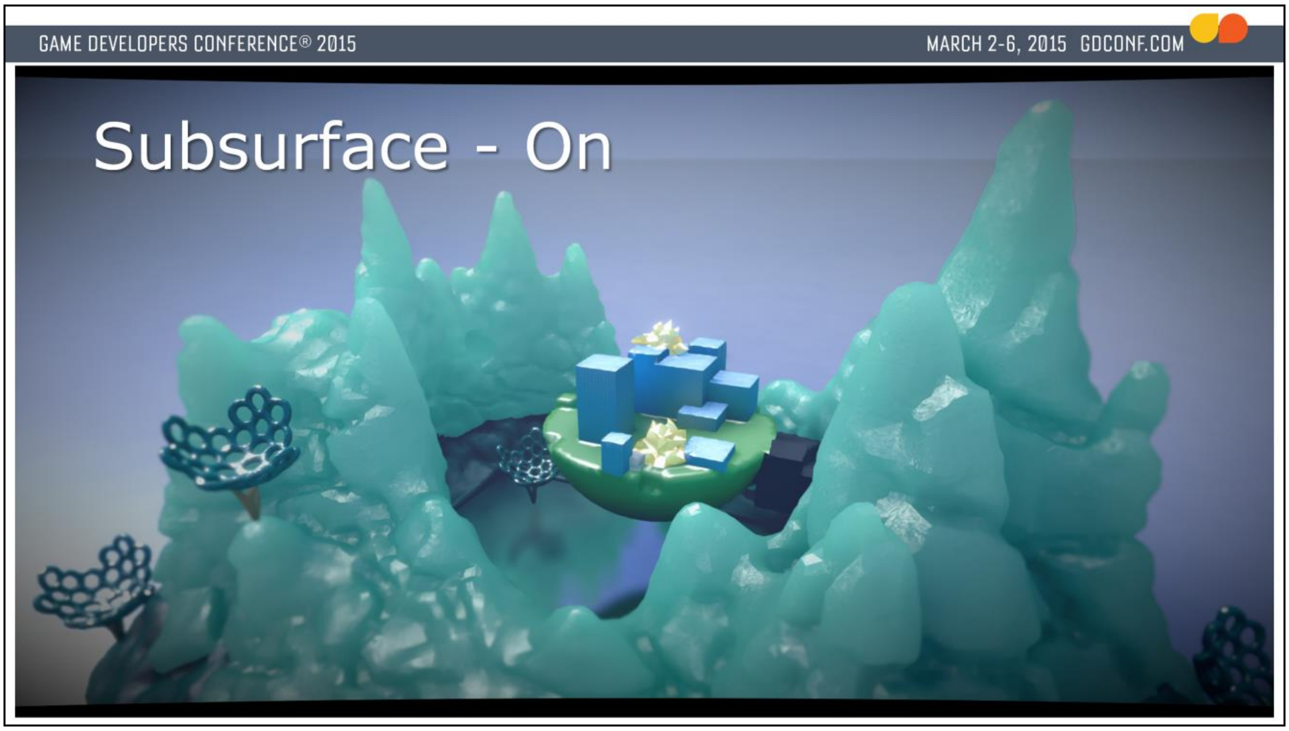
//But in order to take as few samples as possible we only take one raymarch step, and take multiple samples from our cascade at this position,

//Using progressively wider SH cones as we ascend our cascade levels..



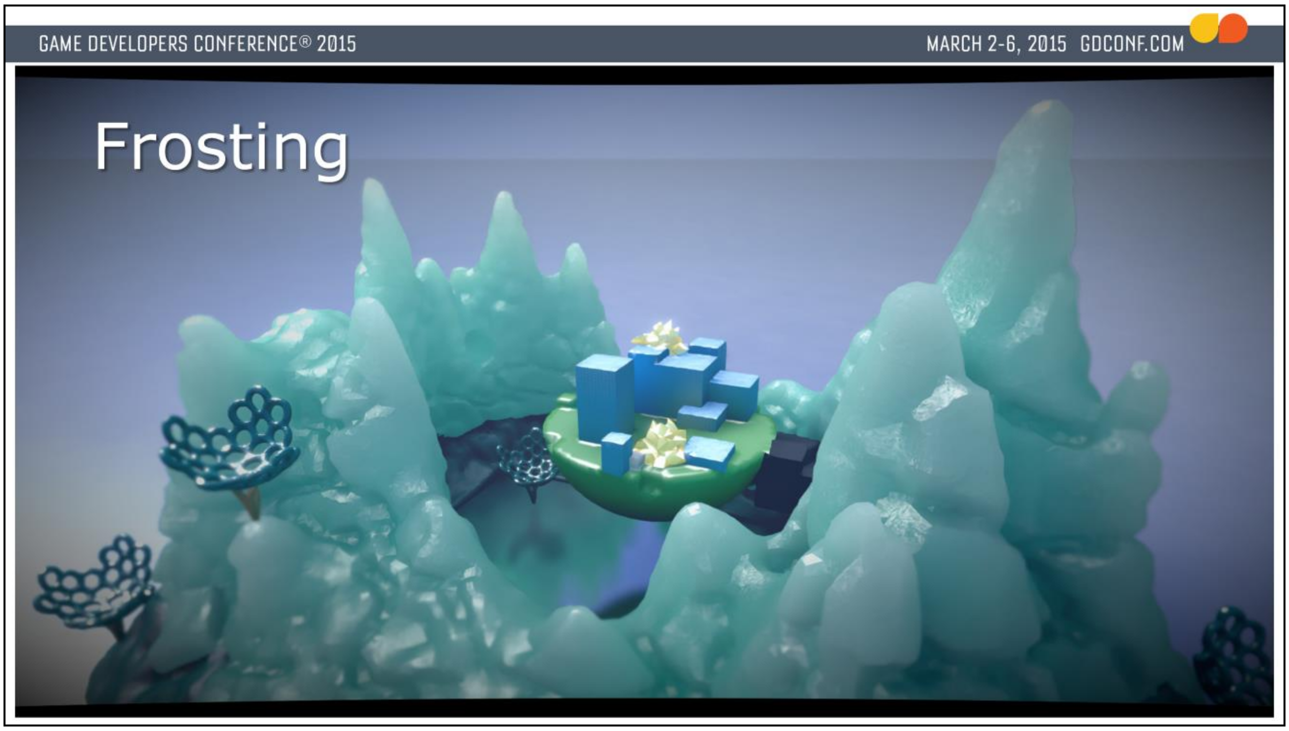
Subsurface - Off

So here you can see some mountains that are light with our vanilla cone trace lighting.



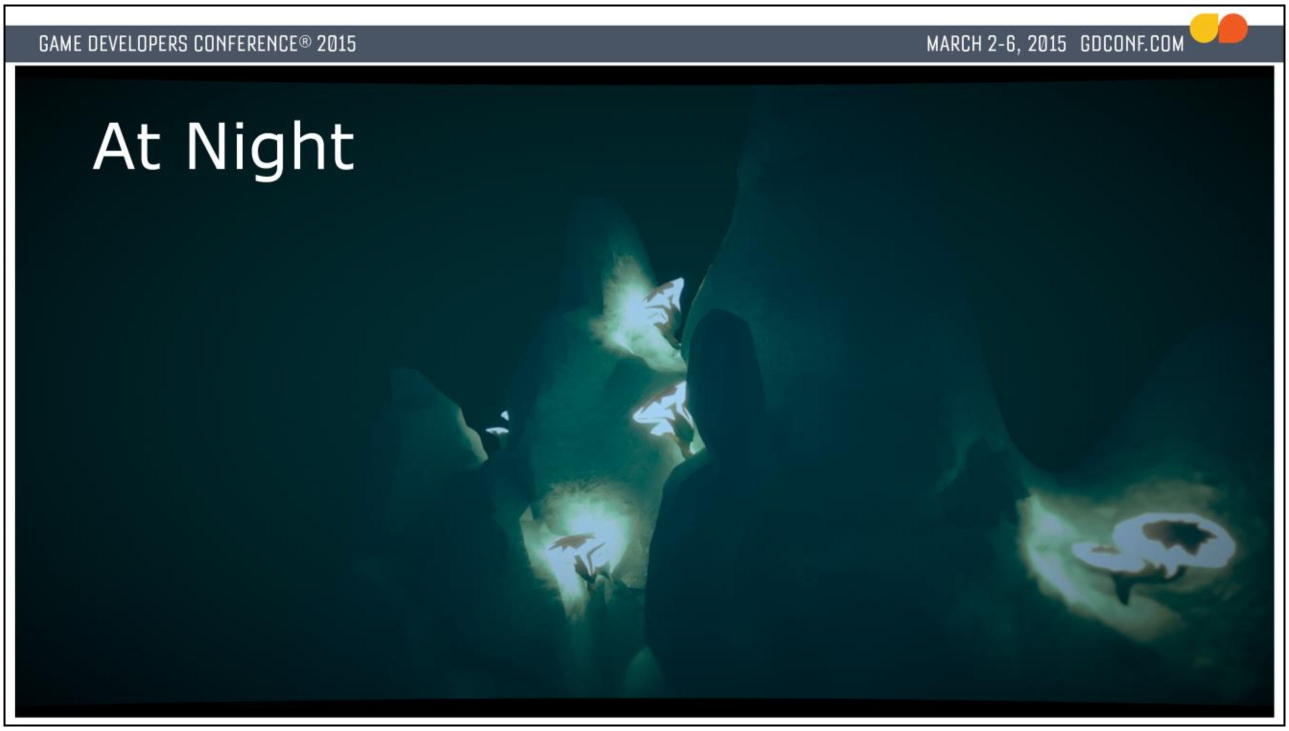
Subsurface - On

And you can see how we can turn on sub surface scattering and give these mountains a nice semi translucent waxy appearance.



Frosting

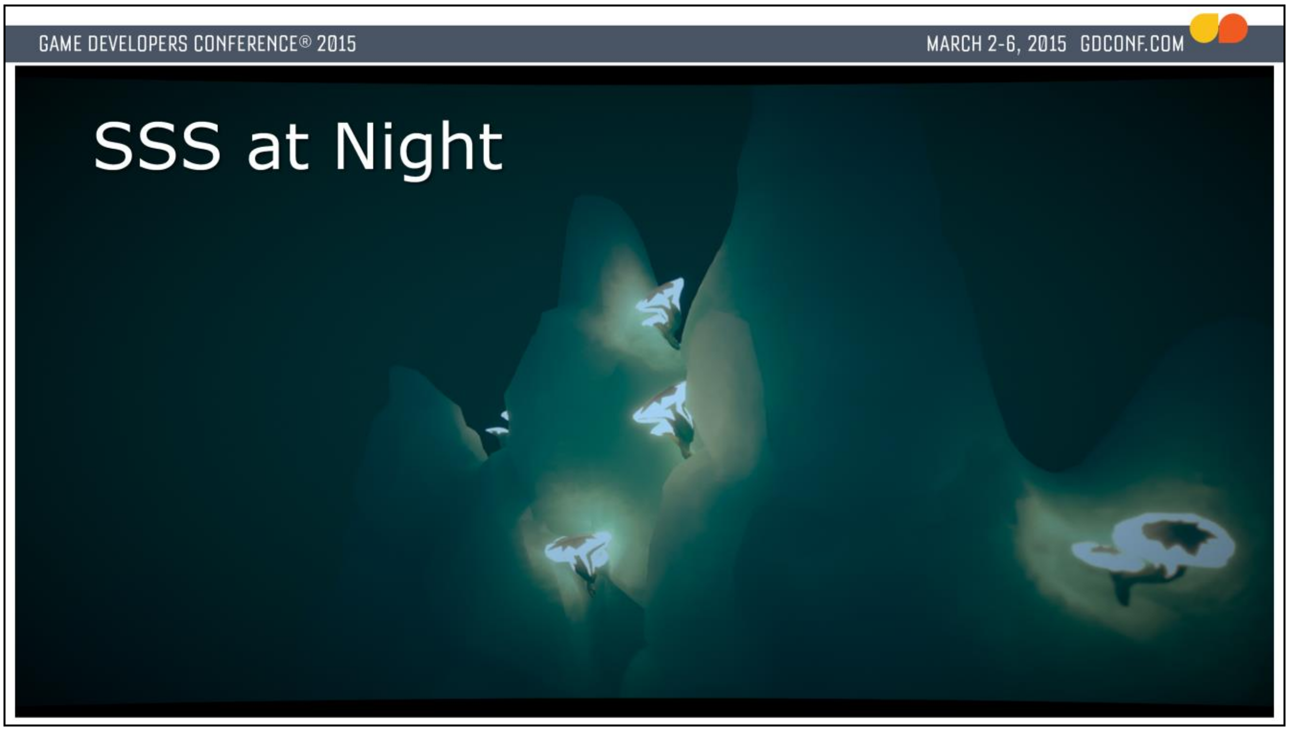
And then we can add the frosting effect to accentuate the thin regions, and we start to have something that looks quite believable.


GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

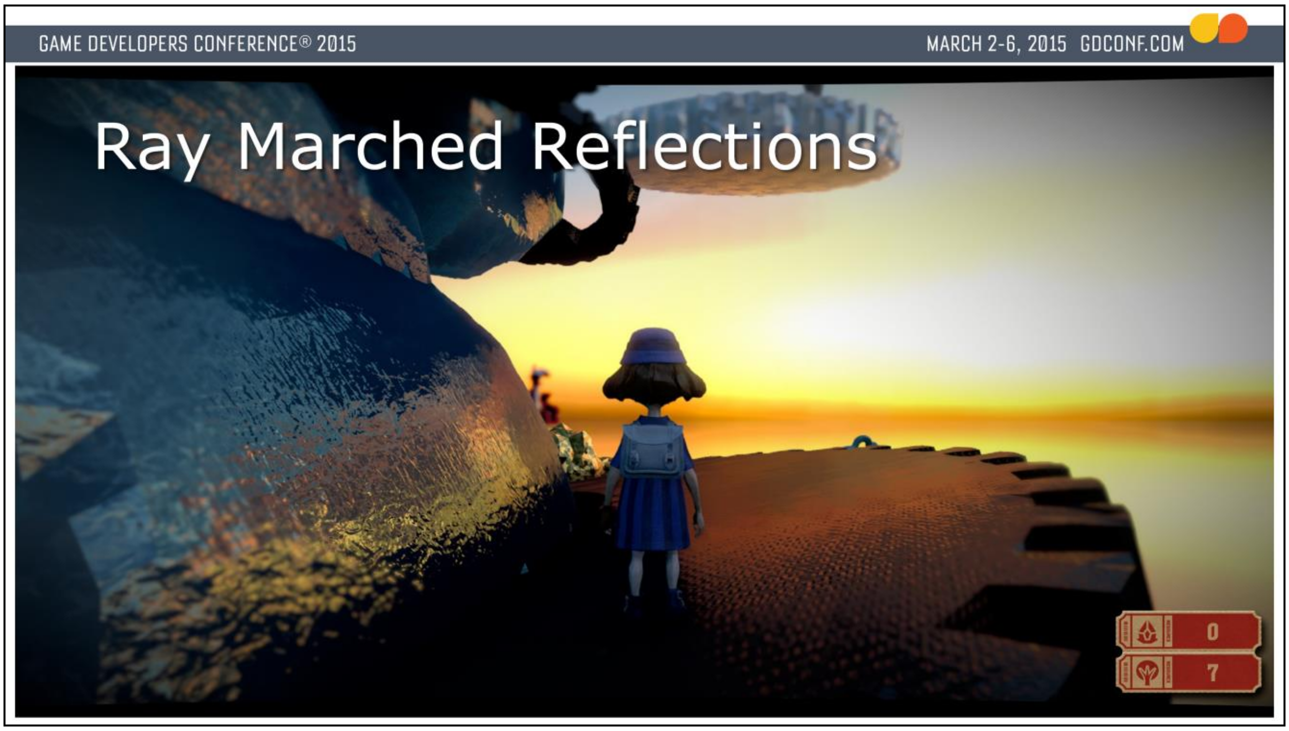
At Night

If we take a look at this same material at night



SSS at Night

We can see more clearly how this sub surface sampling scheme allows light to bleed through the landscape



GAME DEVELOPERS CONFERENCE® 2015

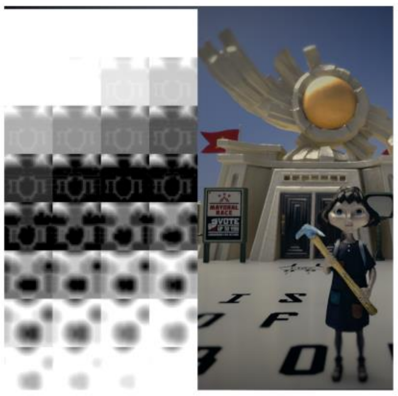
MARCH 2-6, 2015 GDCONF.COM

Ray Marched Reflections

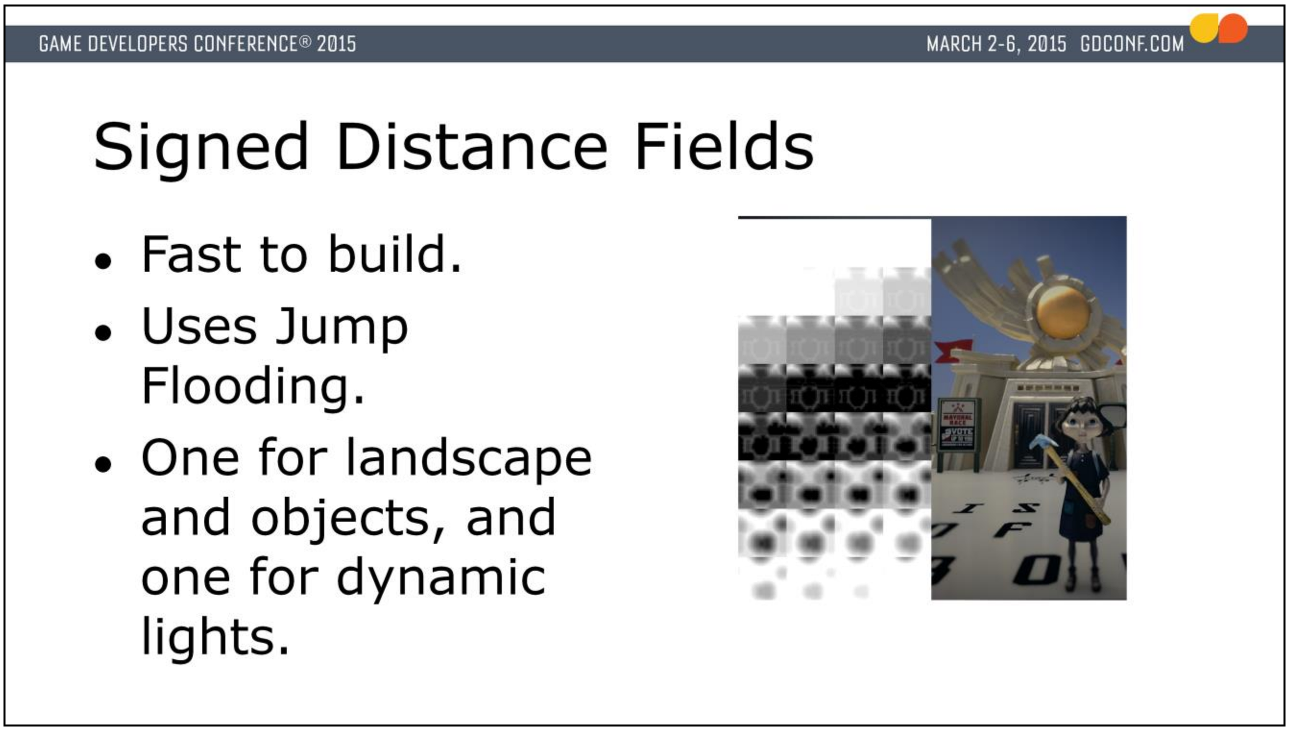
The simplified SH texture, is also very useful for other effects, like reflections

Signed Distance Fields

- Fast to build.
- Uses Jump Flooding.

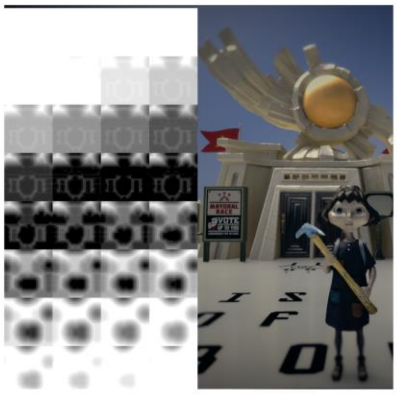


It's very fast to build a Signed Distance Field for our cascades with Jump Flooding, even though they are 3D.



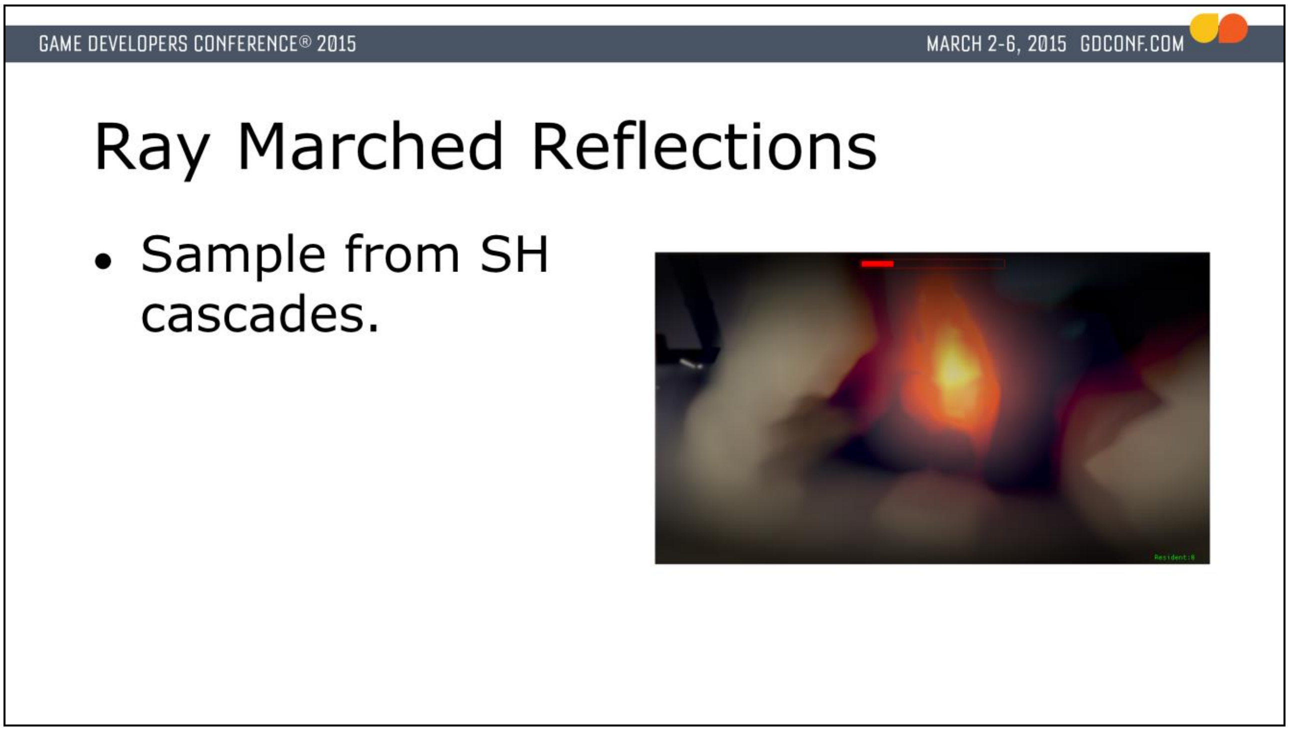
Signed Distance Fields

- Fast to build.
- Uses Jump Flooding.
- One for landscape and objects, and one for dynamic lights.



We generate one for our landscape and objects, and one for our lights.

Once we have these they can be used to accelerate a ray march, through our voxel data.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Ray Marched Reflections

- Sample from SH cascades.

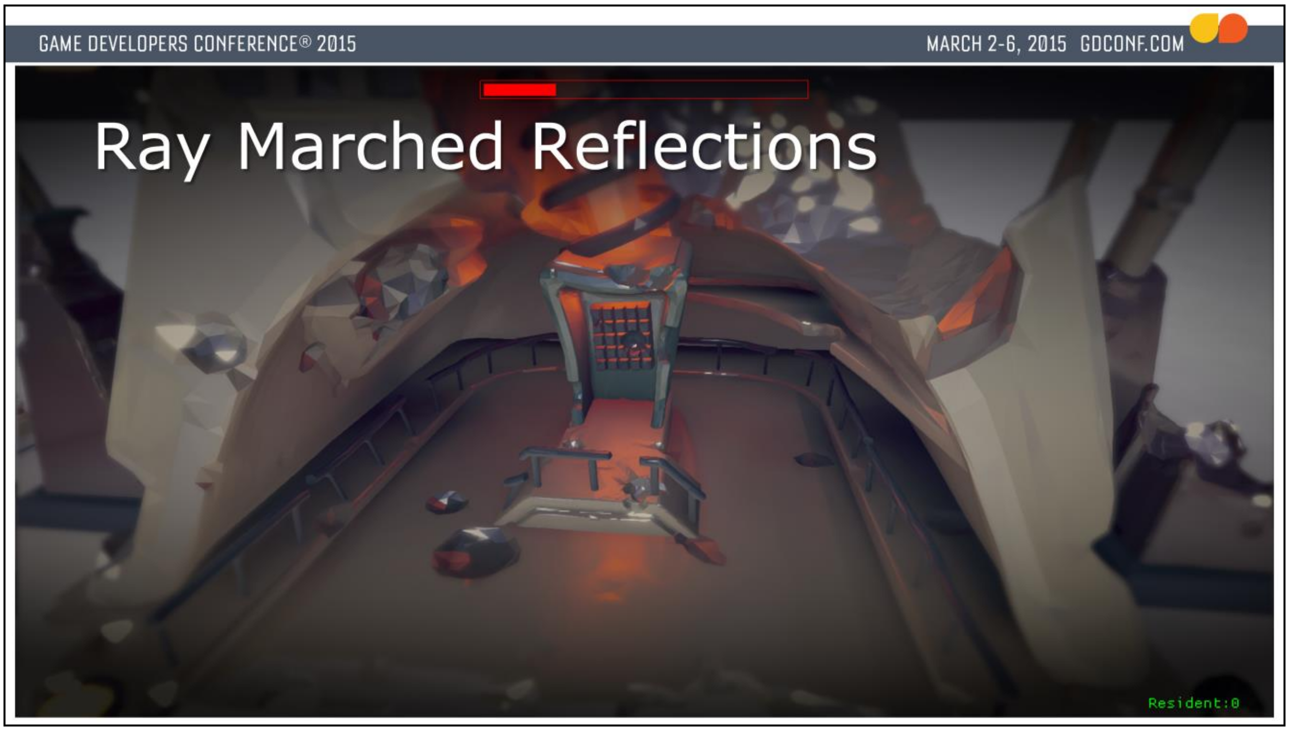
We can then sample from our SH cascade texture when we get close to a surface or a light, and accumulate the results.

Ray Marched Reflections

- Sample from SH cascades.
- Very rough.
- But not dependent on screen space.

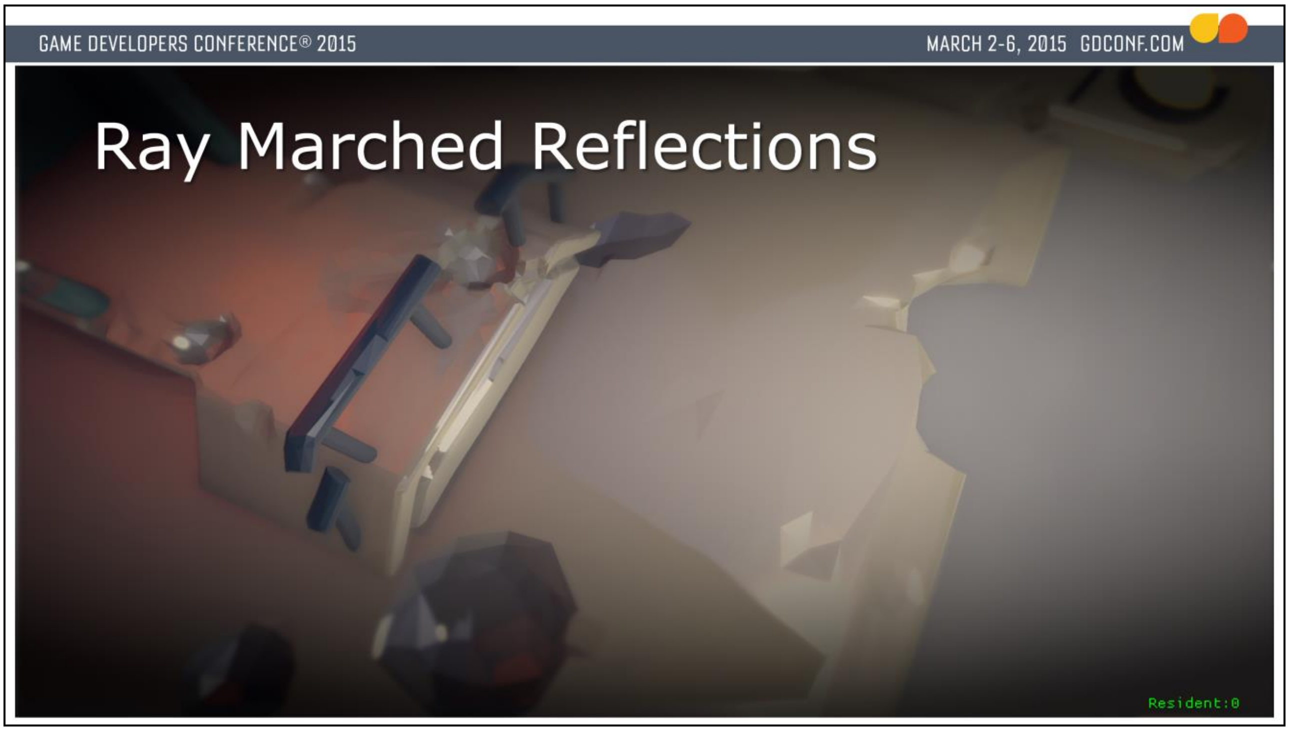
You can see from my picture here that this gives us a very course view of the world, but it’s good enough for glossy specular reflections

And has the advantage that we can reflect objects even when they are off screen.



Ray Marched Reflections

So in this scene with our burning town hall. We can see the reflection of the fire on the floor, and if we look down…

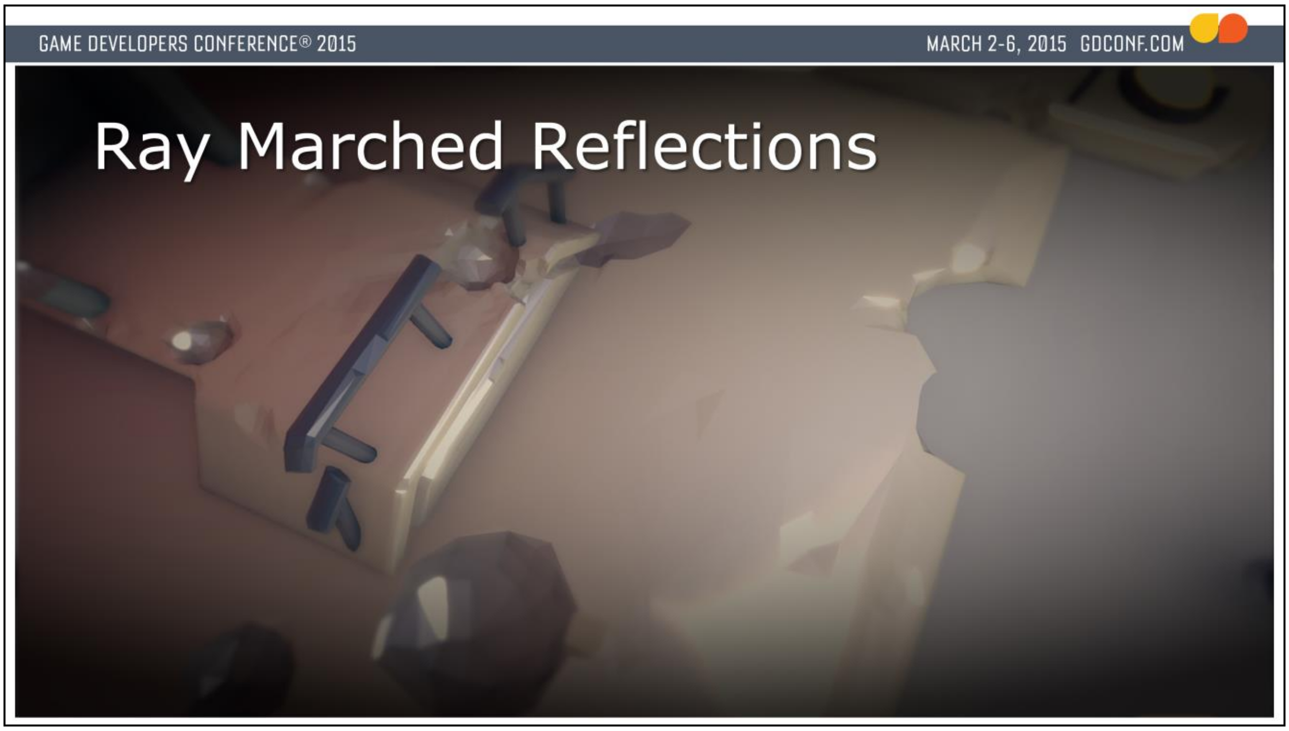


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Ray Marched Reflections

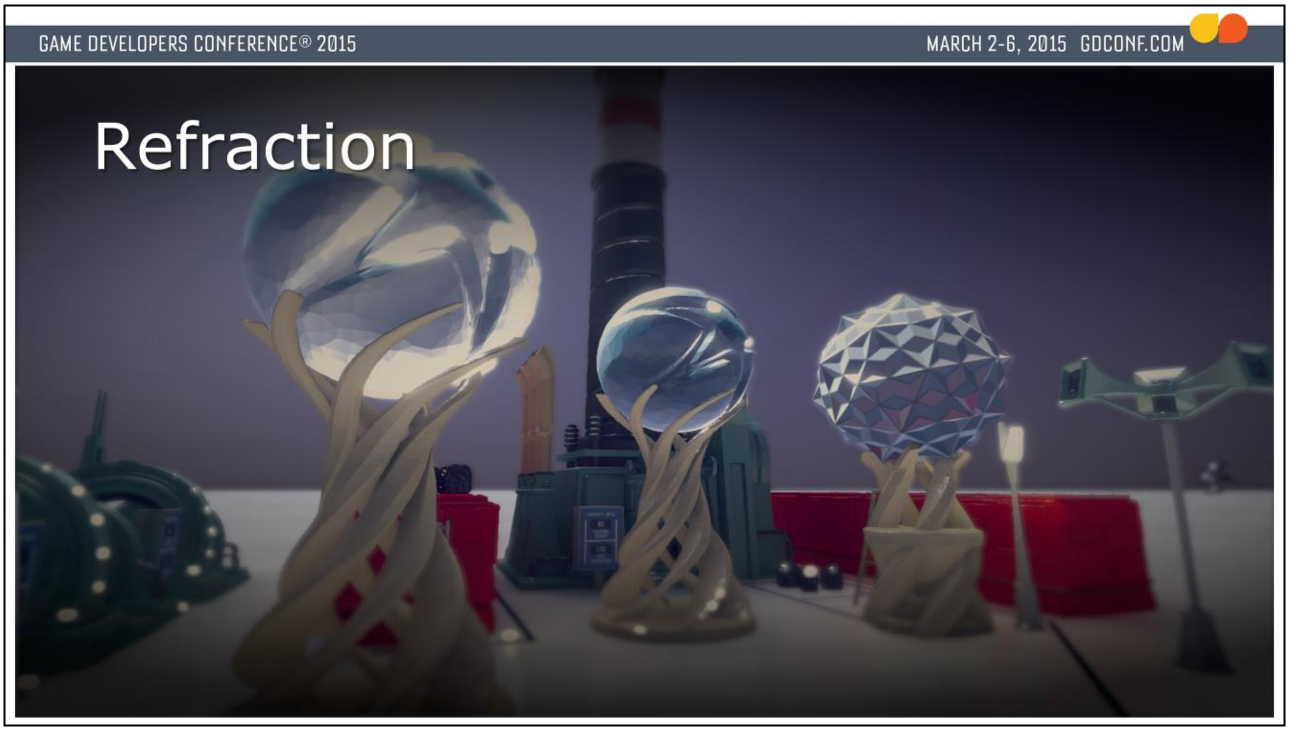
You can see that this still give us a nice, relatively sharp image of our surroundings,

Notice how we can see the fire and the hole in the roof,



Ray Marched Reflections

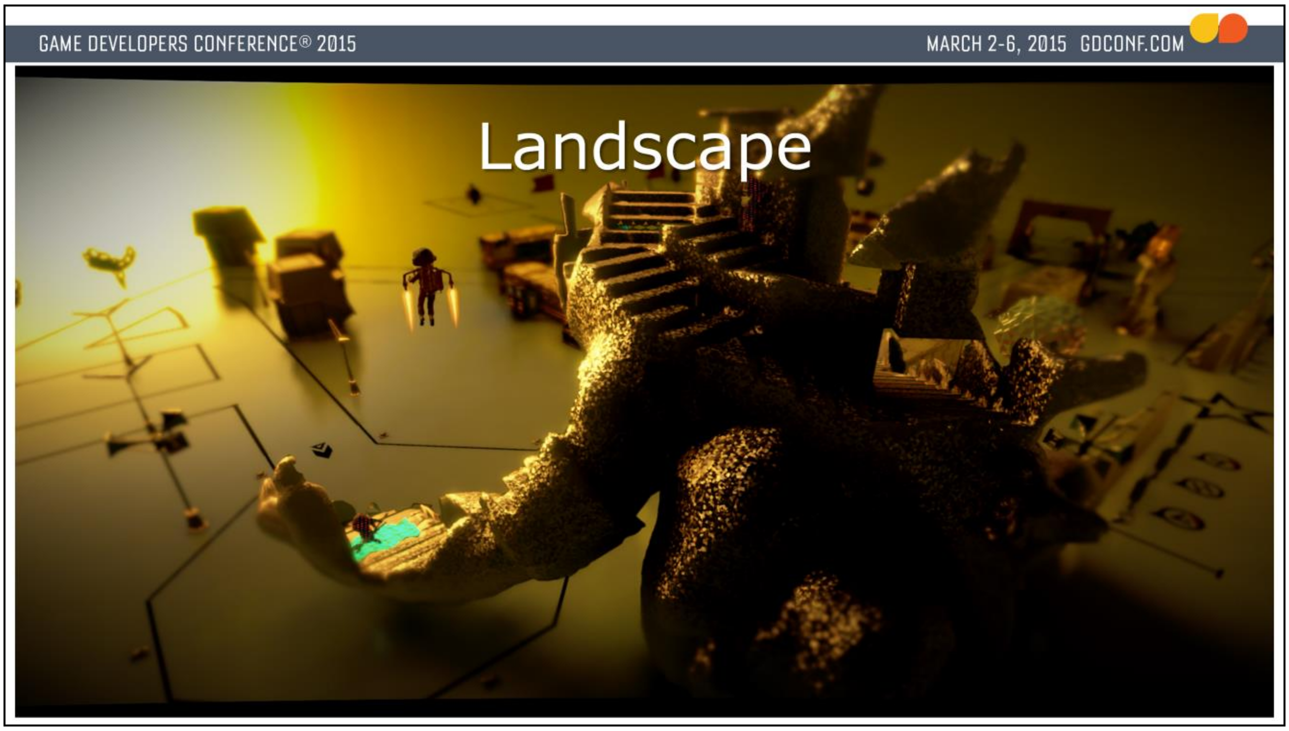
And you can see all the detail we lose if I turn the effect off and use something simpler



Refraction

And of course, we can also extend this approach to allow us to have objects that appear to exhibit glossy refraction.

Like these monuments.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Landscape

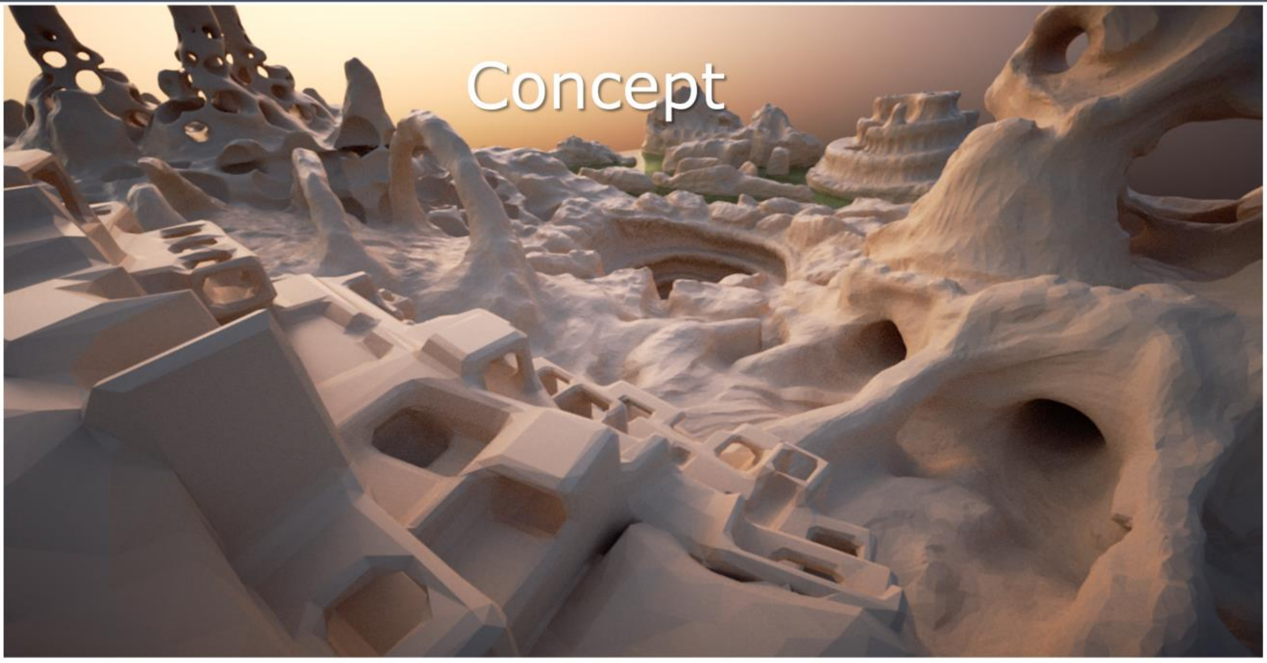
So that pretty much wraps things up in terms of what I'm going to describe about our games lighting.

But of course that's only half of the story. Our game is also heavily reliant on the flexible landscape system that we've built,

And so I'll go briefly into a few details of that, so that you can get an idea of how we were able to achieve it.

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Concept

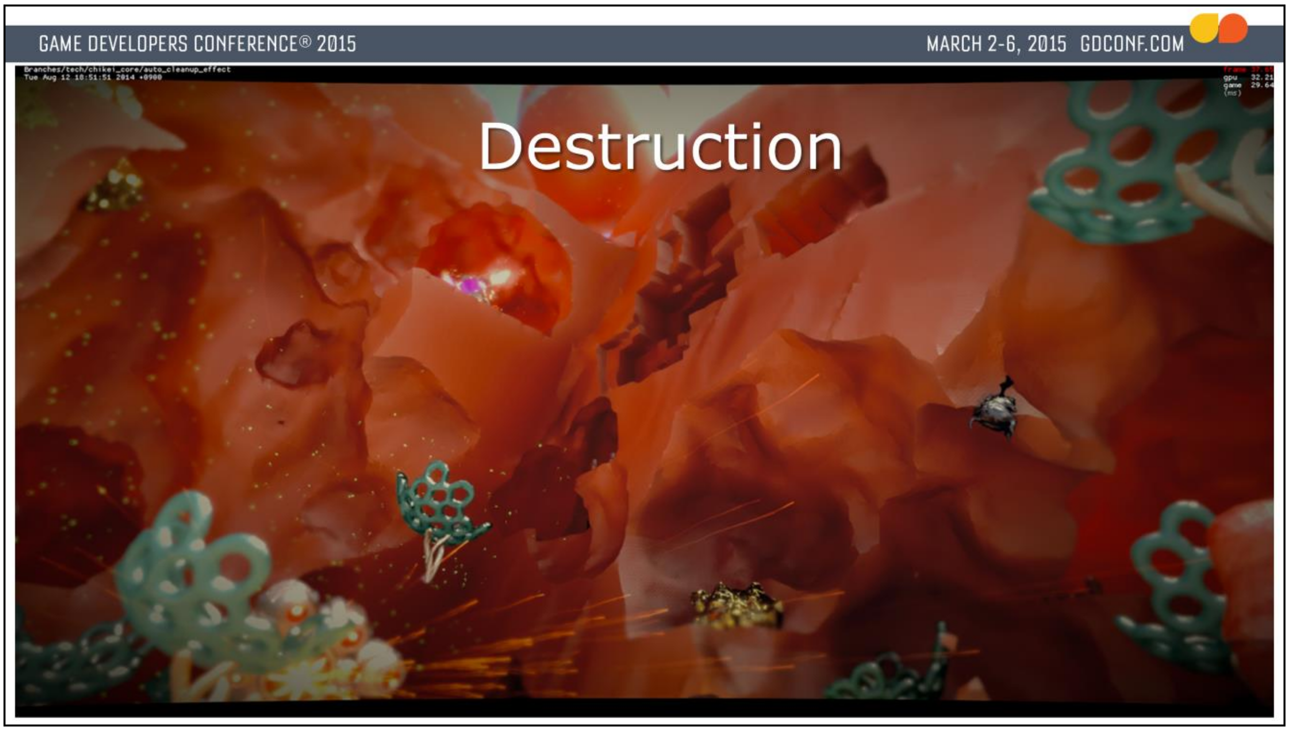


So, when we started out we had some pretty crazy concept renders to work from.
This is what we got when we asked our artists to show us the terrain features they would like to support.
They made all of this in zbrush.
And you can see from all the tunnels and holes that we had no real hope of being able to achieve this with height fields and displacement.

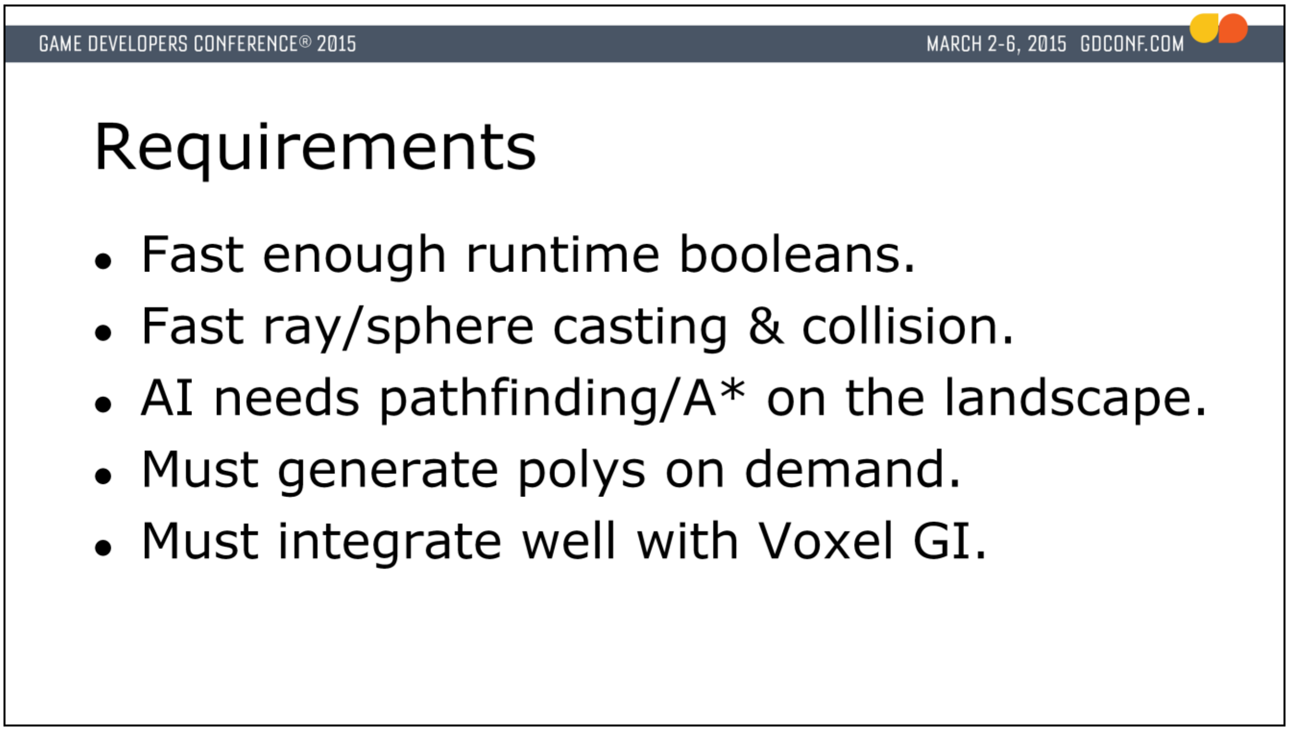
GAME DEVELOPERS CONFERENCE® 2015
MARCH 2-6, 2015 GDCONF.COM

Creation

On top of this, it soon became clear that in order to support various gameplay features, the landscape would need to be dynamic, supporting both creation.



And also destruction, via some form of Boolean CSG operations.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Requirements

- Fast enough runtime booleans.
- Fast ray/sphere casting & collision.
- AI needs pathfinding/A* on the landscape.
- Must generate polys on demand.
- Must integrate well with Voxel GI.

So we found ourselves looking around for a data structure that would be appropriate for our needs.

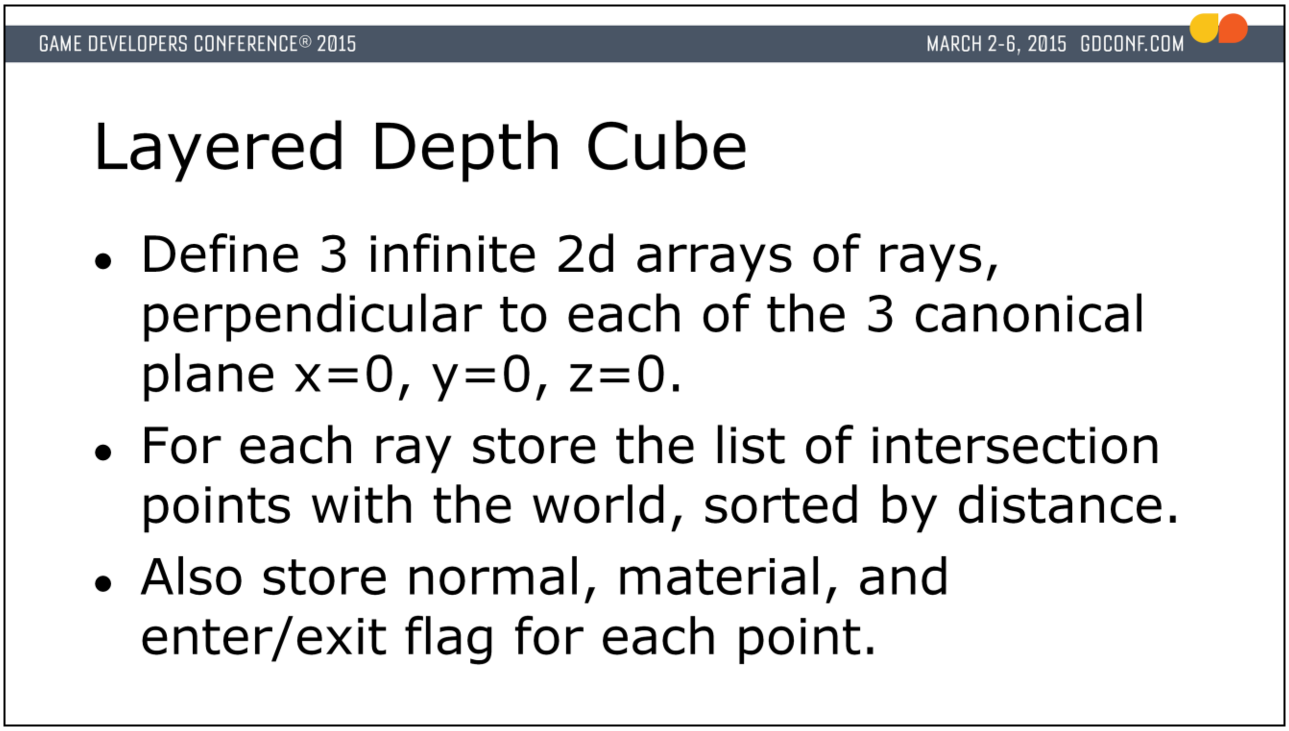
It would need to support booleans, allow us to perform ray casts and handle collisions.

It also had to allow us to run pathfinding queries, as well as easily generate polygons from it on demand.

On top of this, it also had to be integrate well with the Global Illumination system that we were planning, and be queriable from the GPU.

We evaluated several options for how would could do all this, including a brief flirtation with distance fields, which have the potential to be very flexible for this kind of thing

But what we eventually settled on was what’s called a Layered Depth Cube

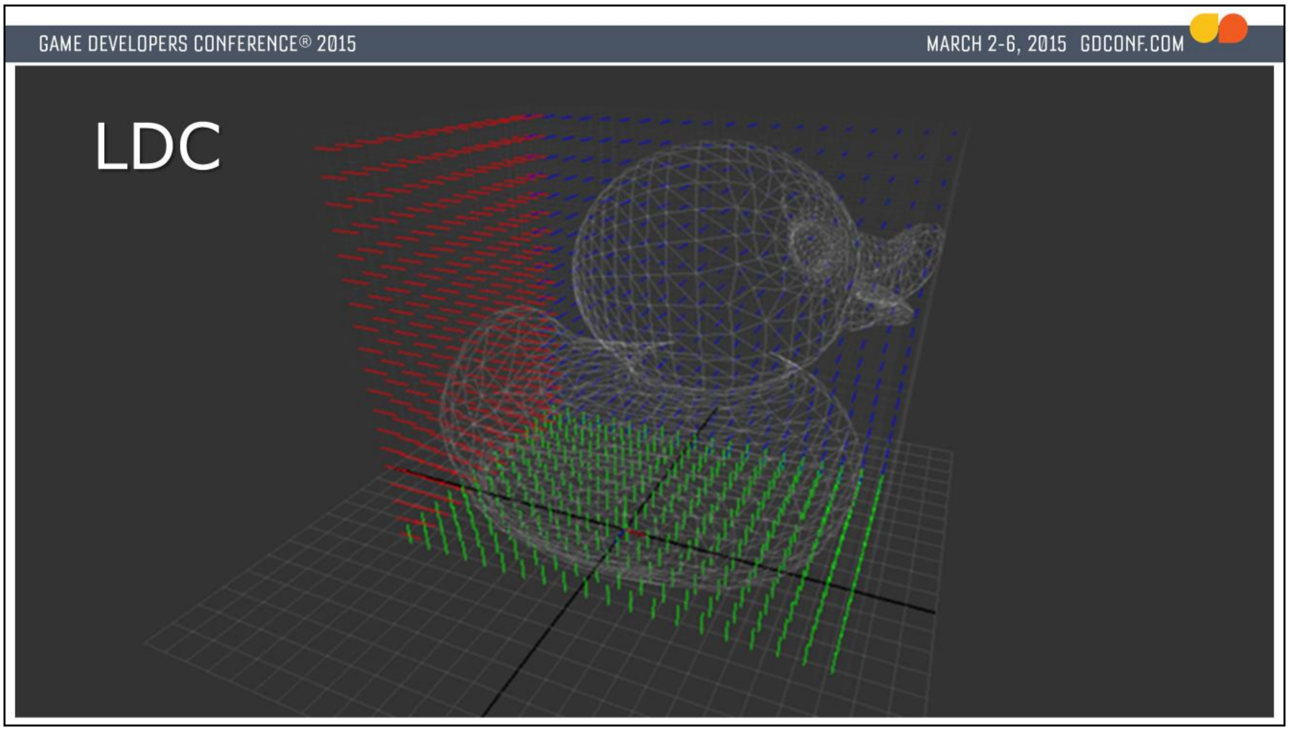


Layered Depth Cube

- Define 3 infinite 2d arrays of rays, perpendicular to each of the 3 canonical plane x=0, y=0, z=0.
- For each ray store the list of intersection points with the world, sorted by distance.
- Also store normal, material, and enter/exit flag for each point.

So in a Layered Depth Cube, we shoot rays down the center of the cells of 3 grids, one aligned to each axis,

and store a sorted list of all the points of intersection along with their attributes and whether we are entering or exiting our geometry.



LDC

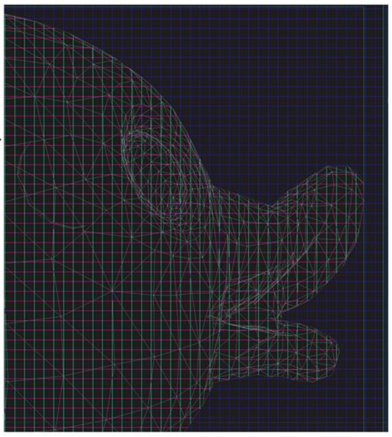
So you have something like this , for this simple duck scene.

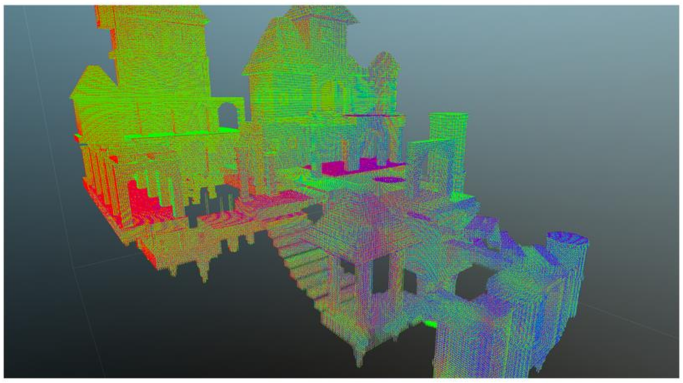
Here you can see the starting points of the rays that we will trace.

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

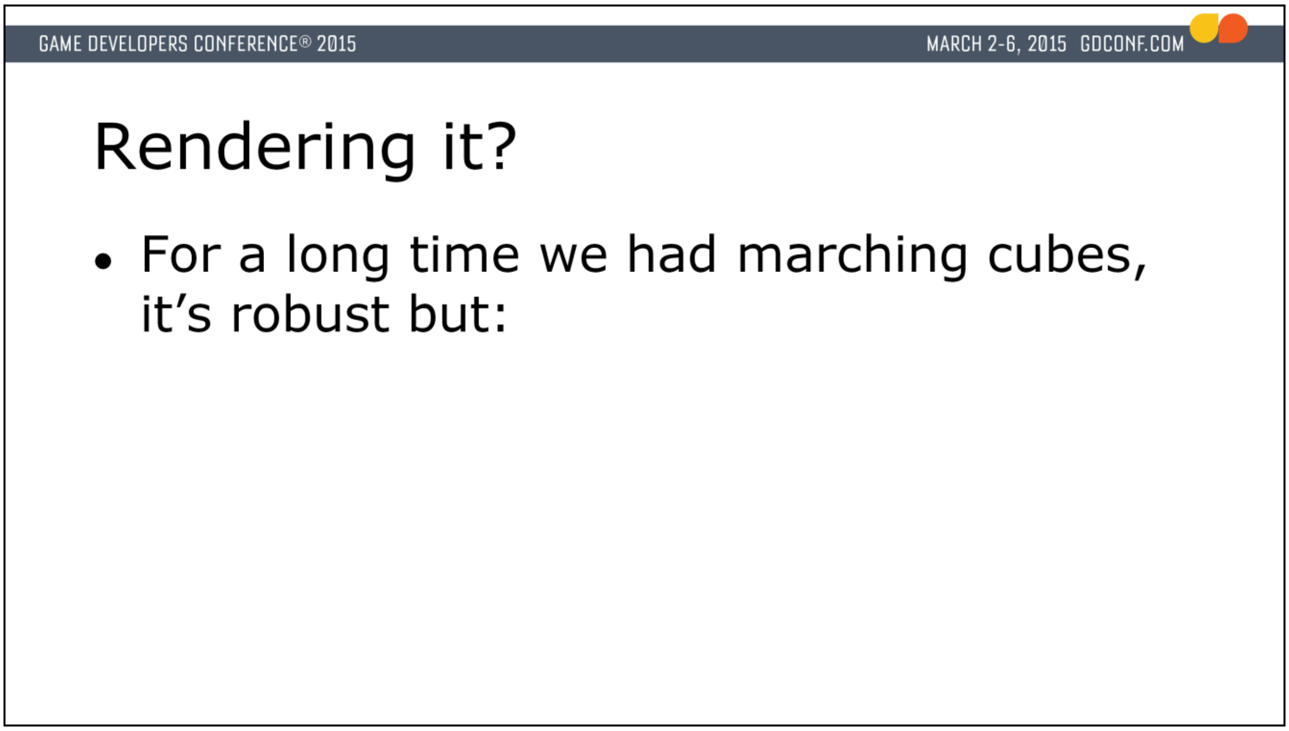
LDC / Overview

- Slice showing x and y rays/lines of matter →
- ↓ Similar debug draw in 3D





And here we see the spans that we extract from the duck on the along the x and y axis,
And the intersection point on each axis for this more complex scene below.
As the structure is so simple and regular, it very easy to use to perform CSG operations on it.
We do have some extra levels of hierarchy to our data structure that we use to help keep things performant,
But I unfortunately I don’t have the time to go into those today.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

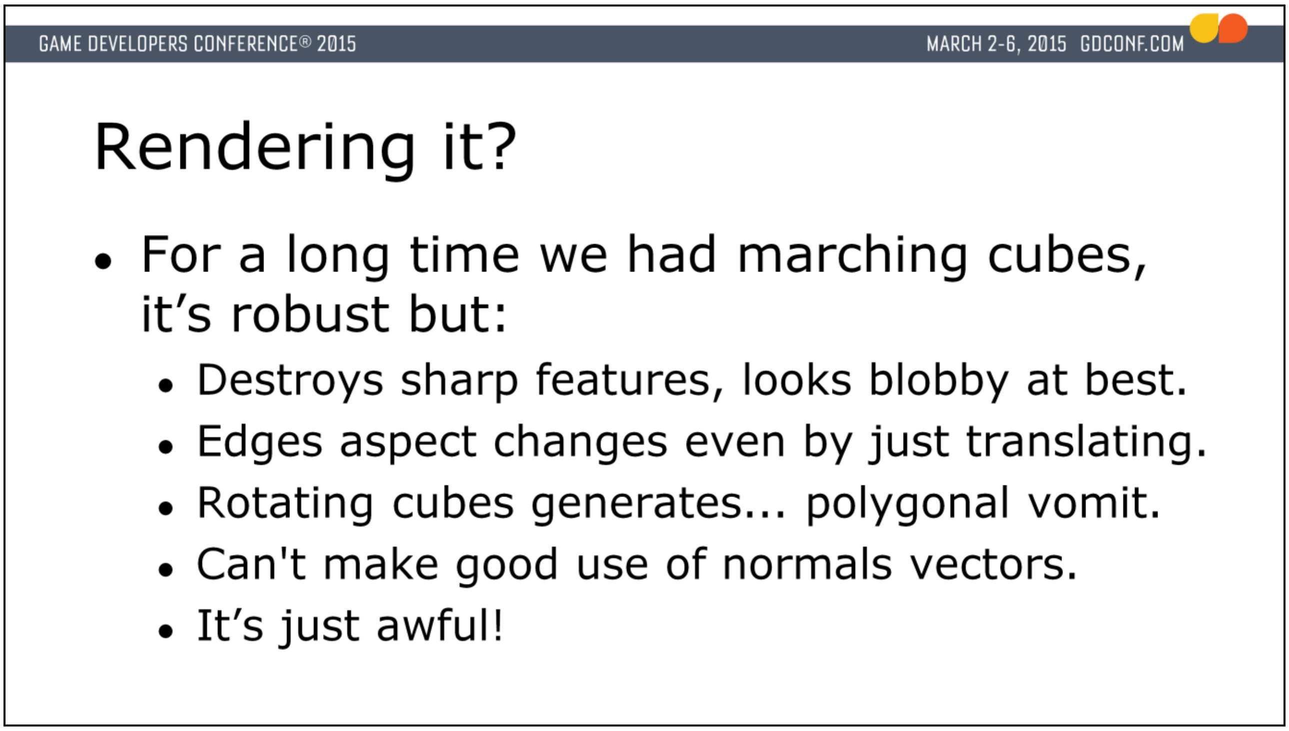
Rendering it?

- For a long time we had marching cubes, it's robust but:

So how do we take the LDC and turn it into something renderable?

We did look at various splatting based techniques, but didn't get very far with them,

and for most of the project we had settled, somewhat uneasily on marching cubes.



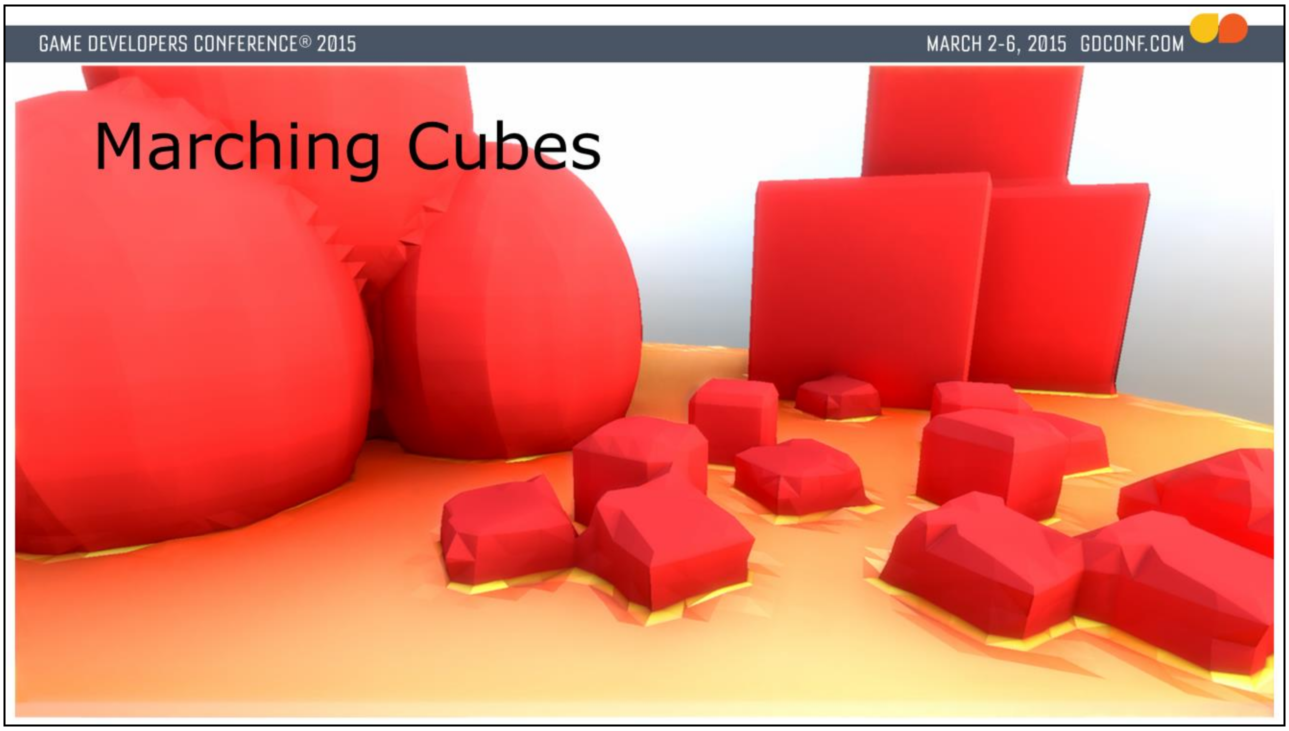
Rendering it?

- For a long time we had marching cubes, it's robust but:
  - Destroys sharp features, looks blobby at best.
  - Edges aspect changes even by just translating.
  - Rotating cubes generates... polygonal vomit.
  - Can't make good use of normals vectors.
  - It's just awful!

But there were some major issues with the output.

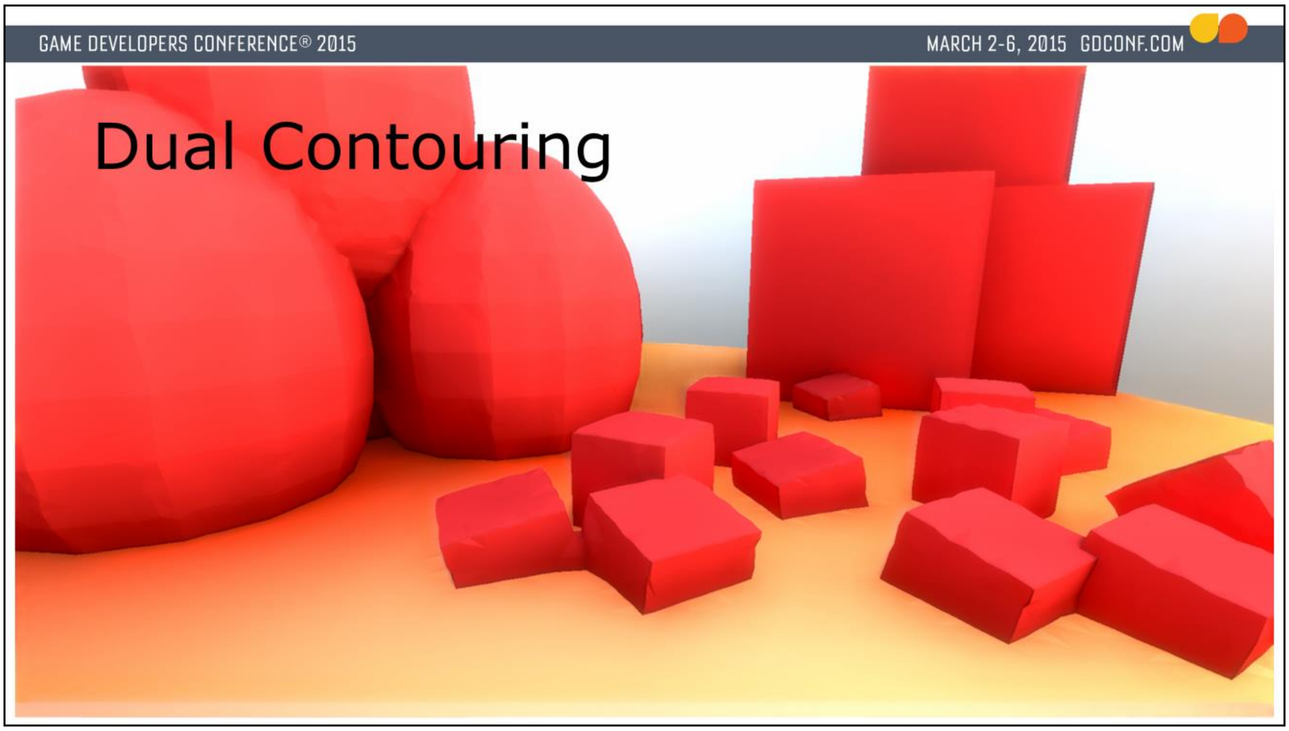
It destroyed sharp features, created results that seemed to vary wildly with the orientation of the input mesh,

We also have a nagging feeling that we should be able to make some use of the normal vectors that we had stored at each LDC point, which marching cubes largely threw away.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Marching Cubes

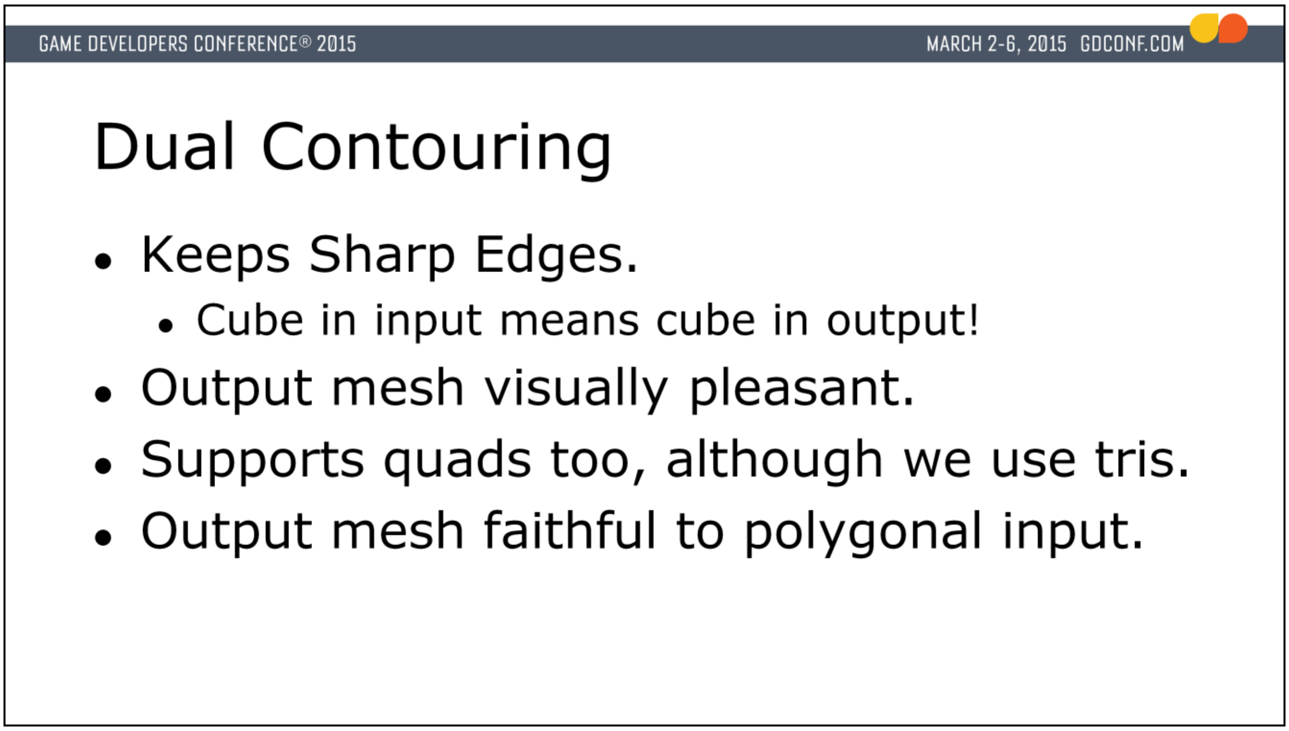
So here you can see how our output looked with Marching Cubes.
The blobs at the front are cubes added via booleans.
There was a tentative form of material blending going on, which is the yellow.
Naturally, we didn't really feel that this was good enough.
And so we decided to experiment with a technique called Dual Contouring.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Dual Contouring

Which as you can see gives a much better result and takes good advantage of point and normal data present in the LDC.

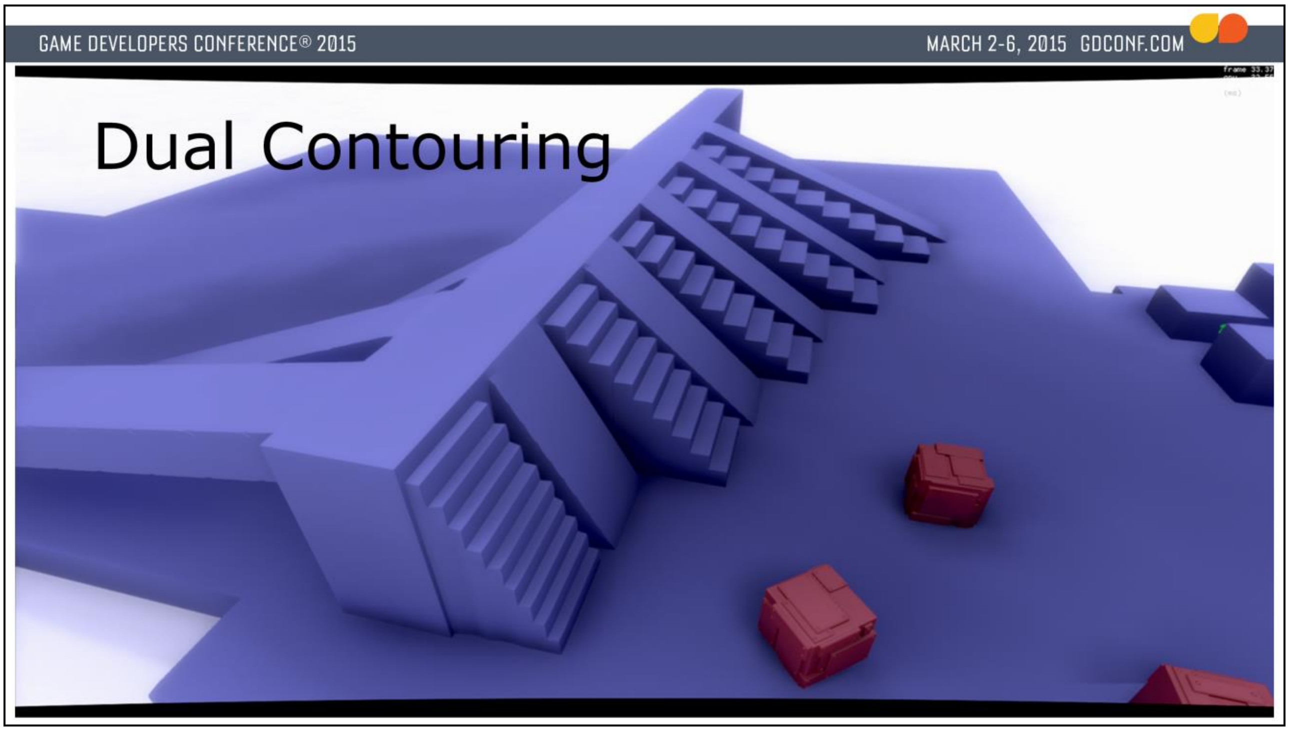

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Dual Contouring

- Keeps Sharp Edges.
  - Cube in input means cube in output!
- Output mesh visually pleasant.
- Supports quads too, although we use tris.
- Output mesh faithful to polygonal input.

So with this means of polygonization, Edges stay sharp, and cubes stay mostly as cubes.

In general, as long as you have enough resolution, this does a much better job of producing meshes that are faithful to the original polygonal input.

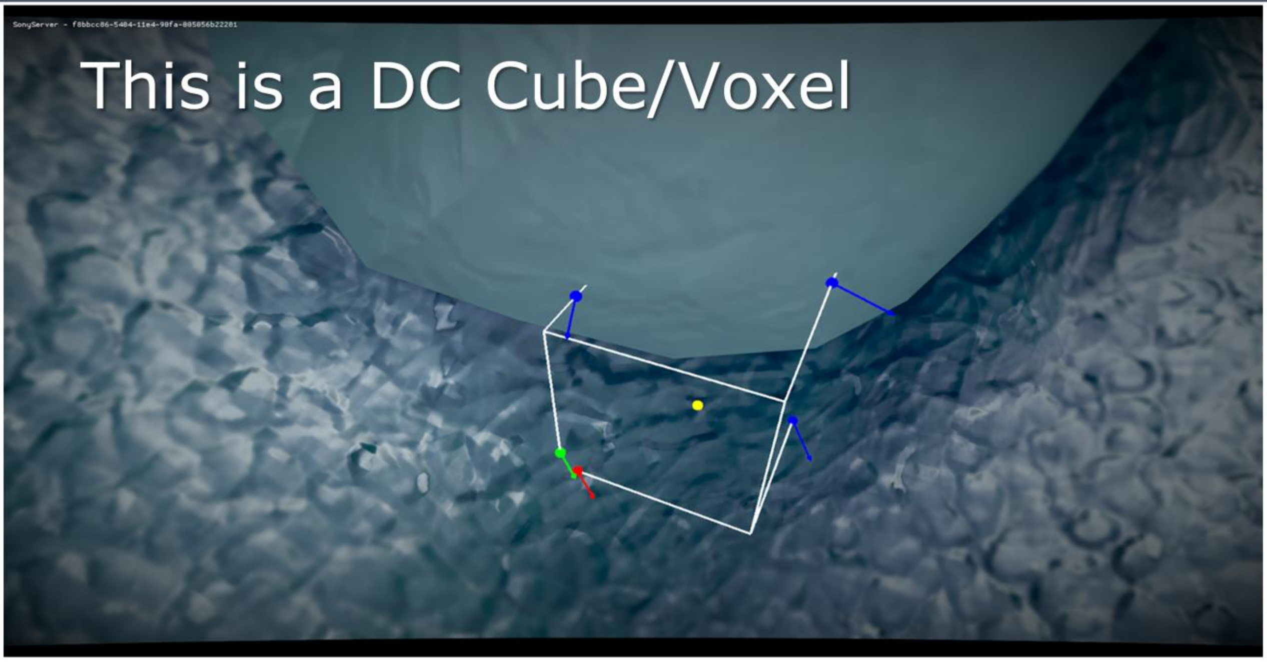


Dual Contouring

And it enables geometrically complex scenes like this one..

This is test scene that was used to debug the character controller.

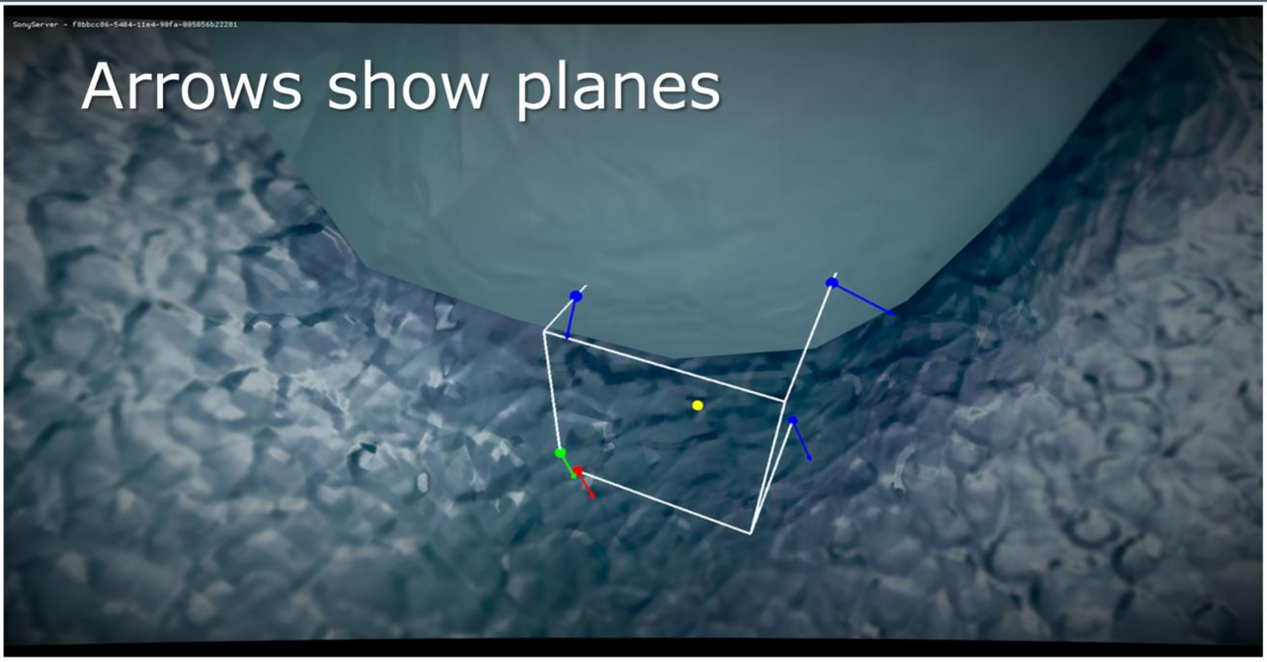
This is a DC Cube/Voxel



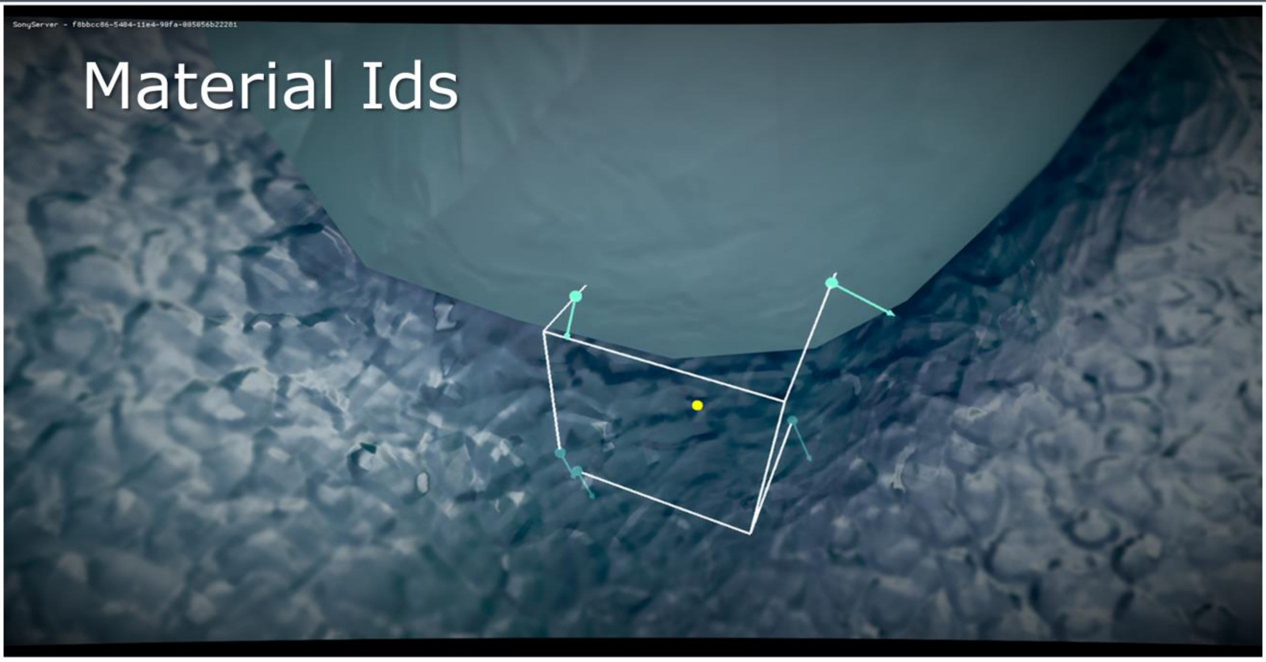
So I'll try to give a brief description of how dual contouring works.

In the picture we have a our landscape, and a cube or voxel within it that we wish to polygonize.

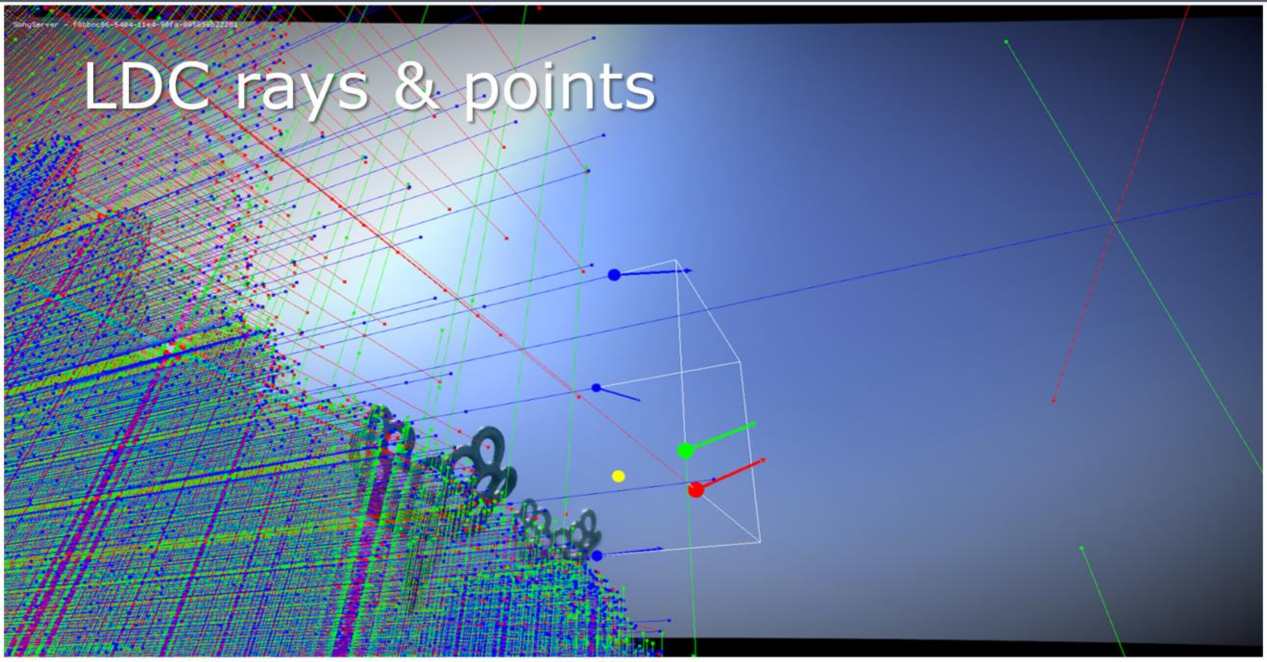
Arrows show planes



Material Ids



LDC rays & points

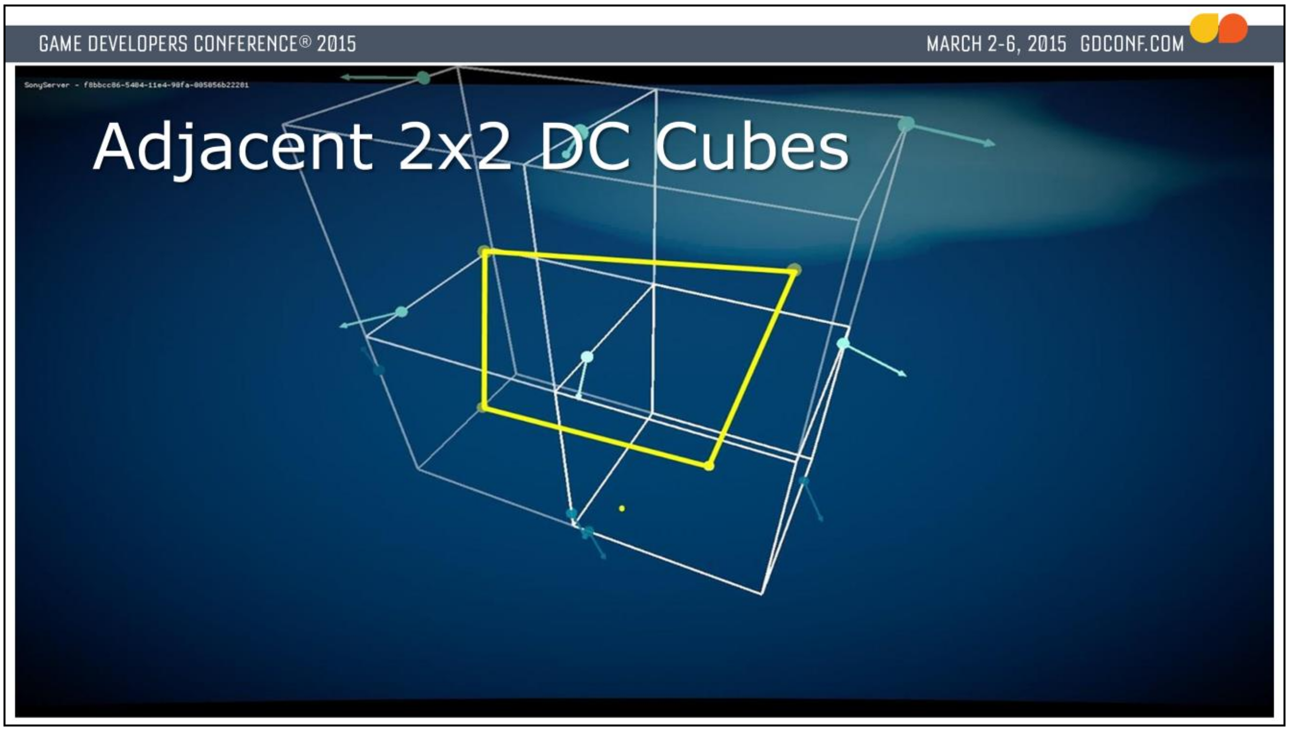


So the Edges of the DC cube are actually aligned with the rays and point intersection lists that we generate in the LDC,

Because of this we can easily walk this structure and pull out intersection point that our inside the DC cube we are interested in,

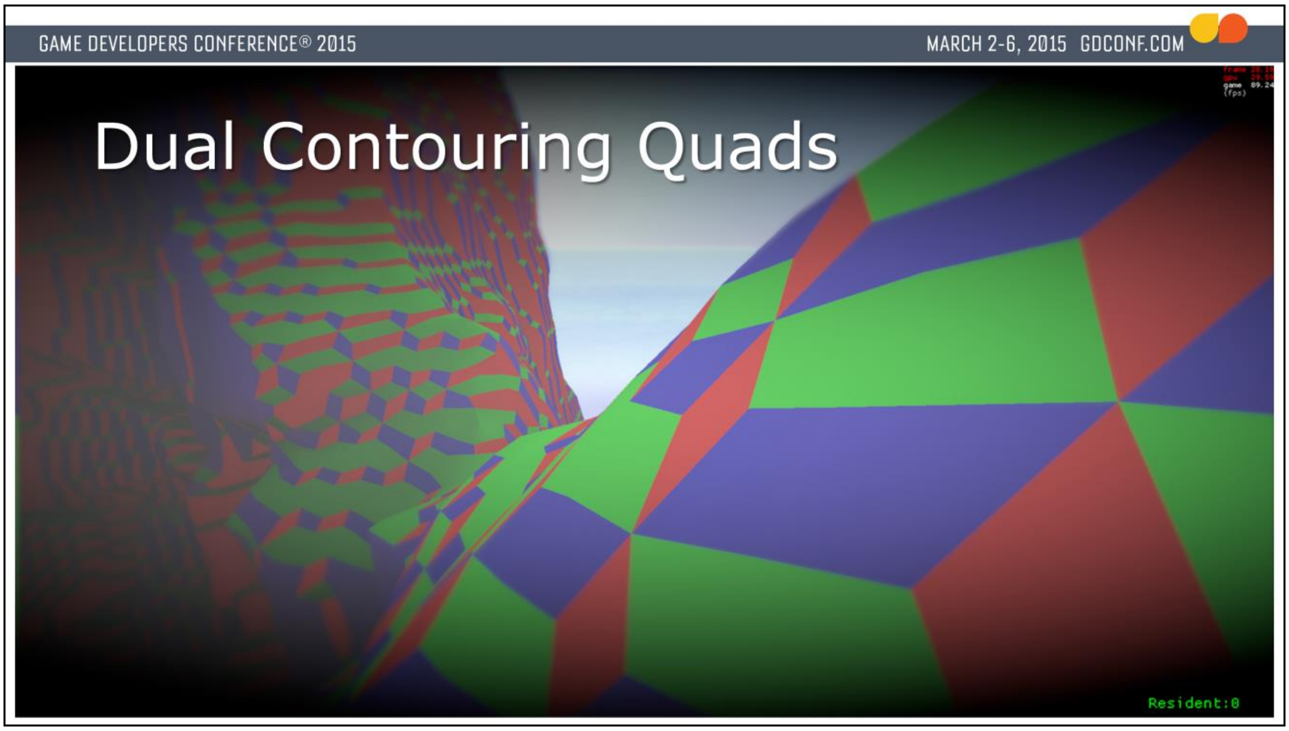
And use them to generate our input planes.

hopefully be able to make this out in the diagram above.



Adjacent 2x2 DC Cubes

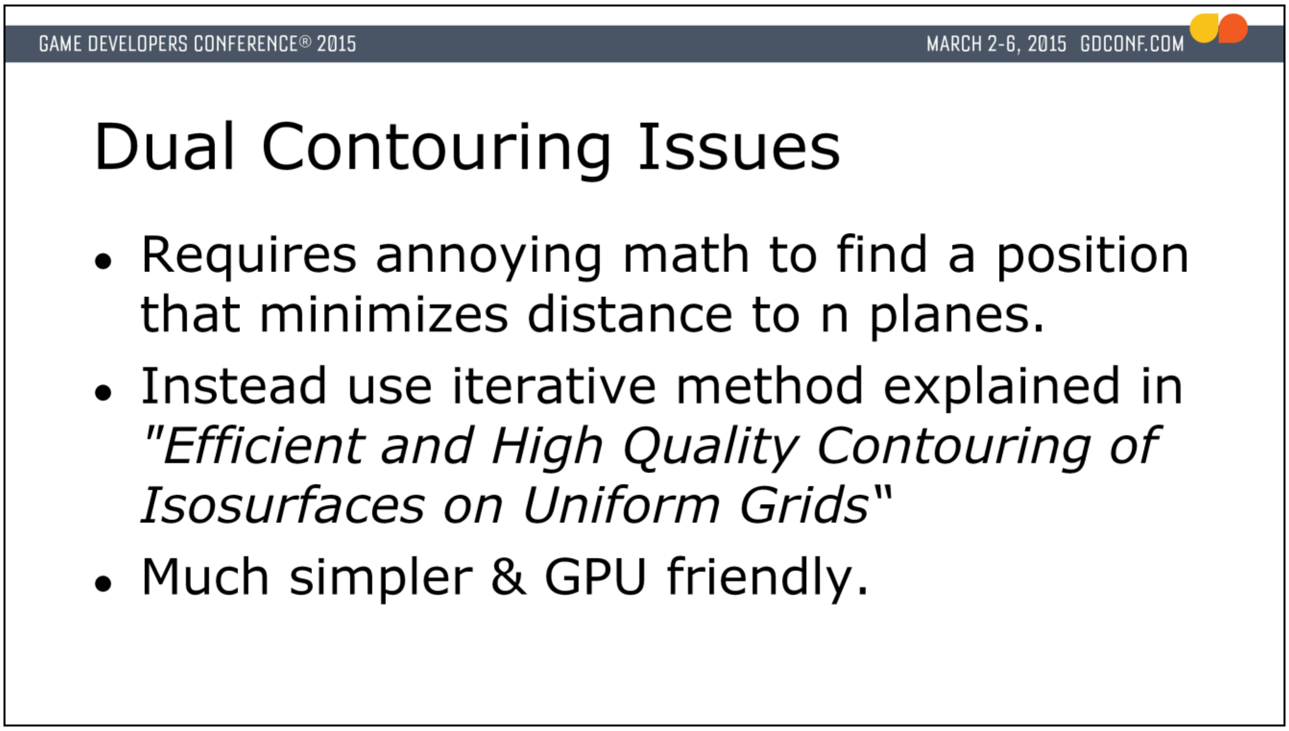
Now, unlike marching cube that polygonizes all voxels independently,
dual contouring considers voxels in groups of 2x2, for each axis.
And creates a quad connecting the four minimized points from each of the 4 cubes, assuming that they exist.



Dual Contouring Quads

So here you can see the results of all this.
This debug view shows the dual contouring quad colored by the axis they “come from”,

The surface here is smooth although this debug view does make it look like otherwise.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

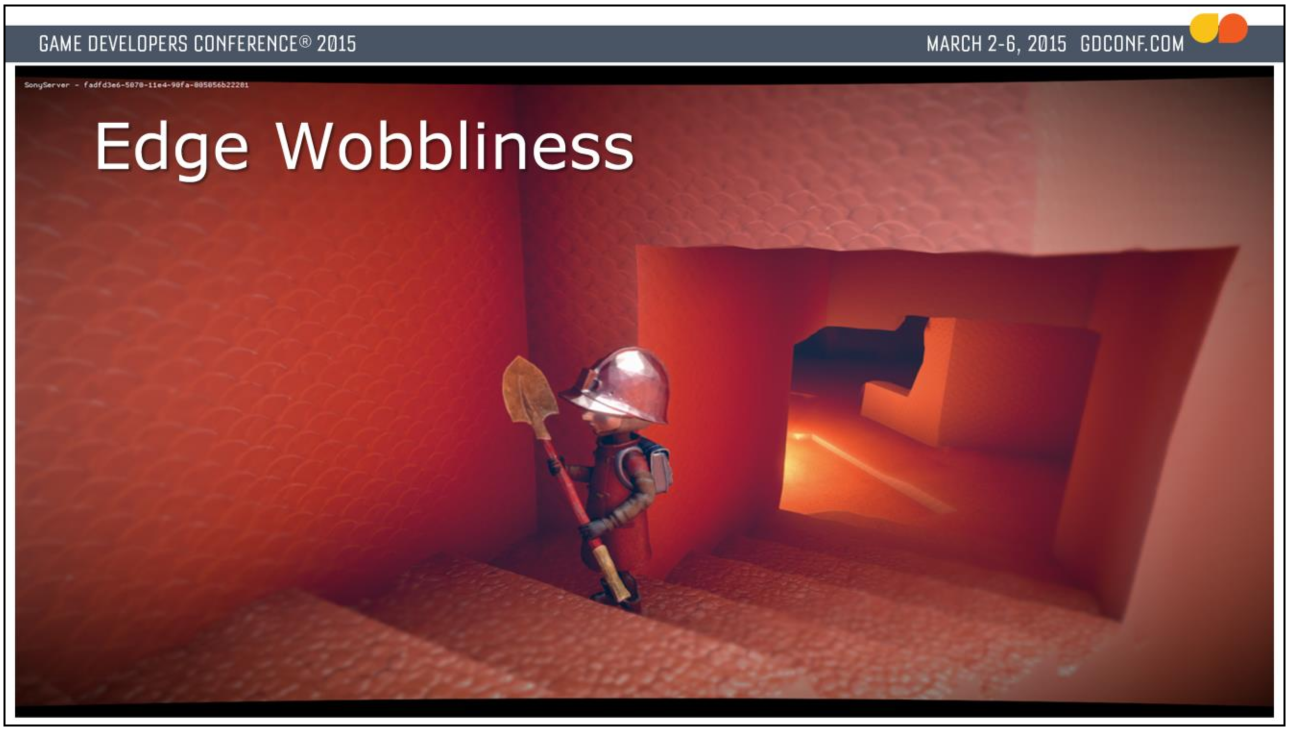
Dual Contouring Issues

- Requires annoying math to find a position that minimizes distance to n planes.
- Instead use iterative method explained in *"Efficient and High Quality Contouring of Isosurfaces on Uniform Grids"*
- Much simpler & GPU friendly.

So Dual Contouring is much better than marching cubes, but it's not without it's issues.
The main problem is that at the core of dual contouring lies an annoying optimization problem that is not very GPU friendly at all.

We use the more intuitive iterative method described in the paper "*Efficient and High Quality Contouring of Isosurfaces on Uniform Grids*" which is trivial to implement.

The only draw back is that it seems to make some sharp edges a little bit wobbly,



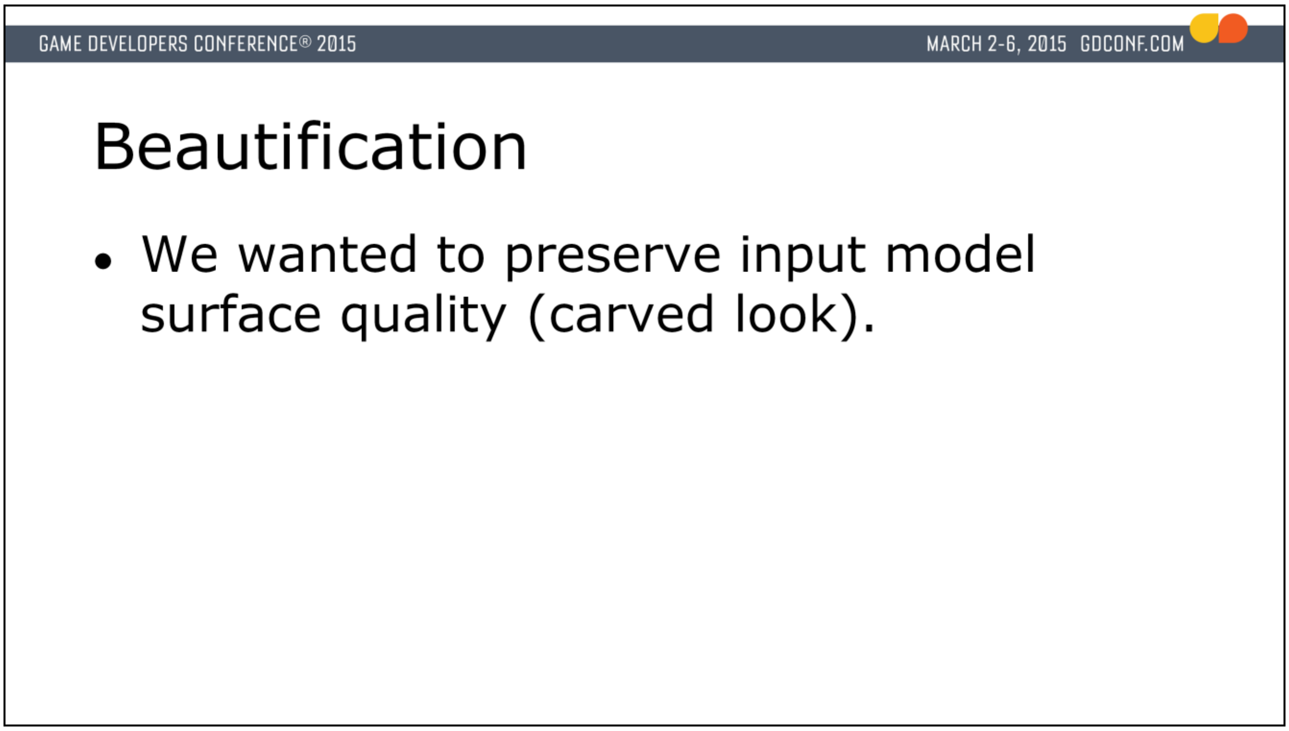
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

SonyServer - fadfd3e6-5070-11e4-90fa-005056b22201

Edge Wobbliness

Hopefully you can see the issue in this picture.

It’s not perfect, but in practice this has not been a big deal for us, and we’ve been able to minimize it’s impact by tweaking the number of iterations and forces used in the iteration process.



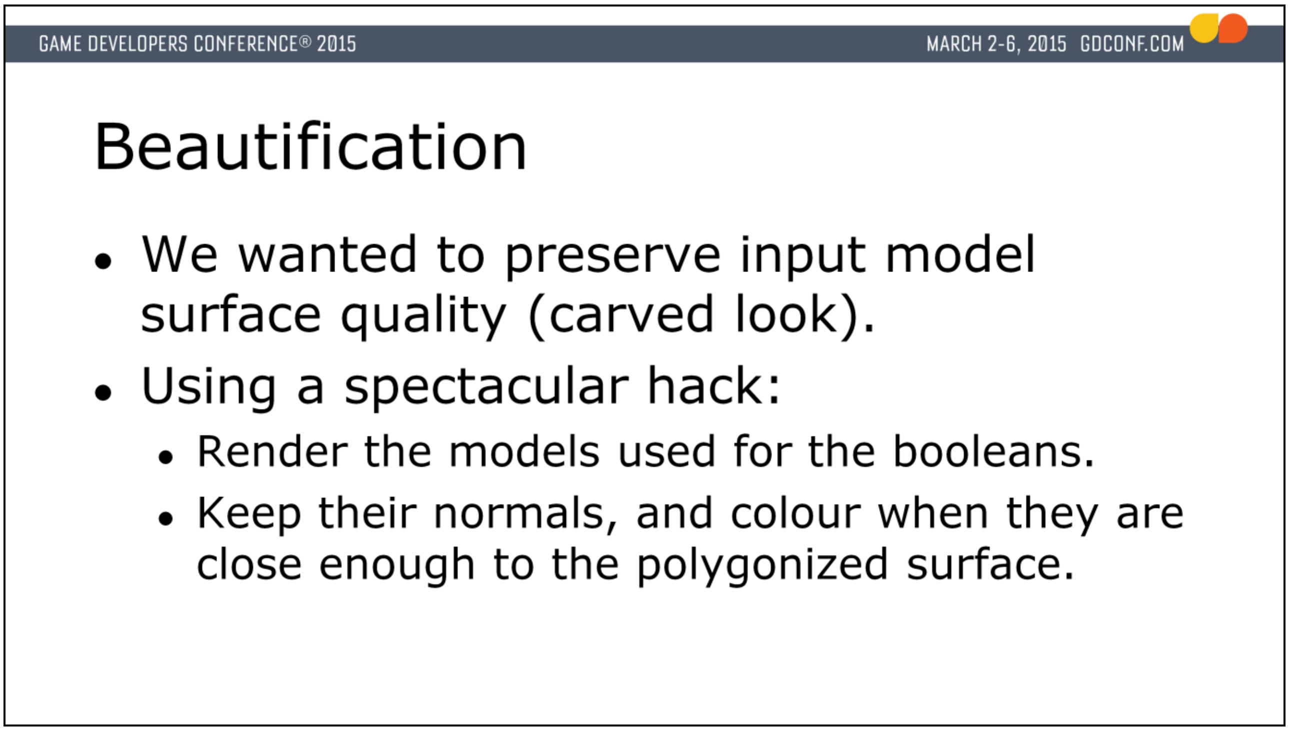
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Beautification

- We wanted to preserve input model surface quality (carved look).

So Dual Contouring was definitely a step up,
but converting geometry to the LDC loses us too much resolution in some instances where we want fine features.
So we've been experimenting with what we call a "beautifier" pass

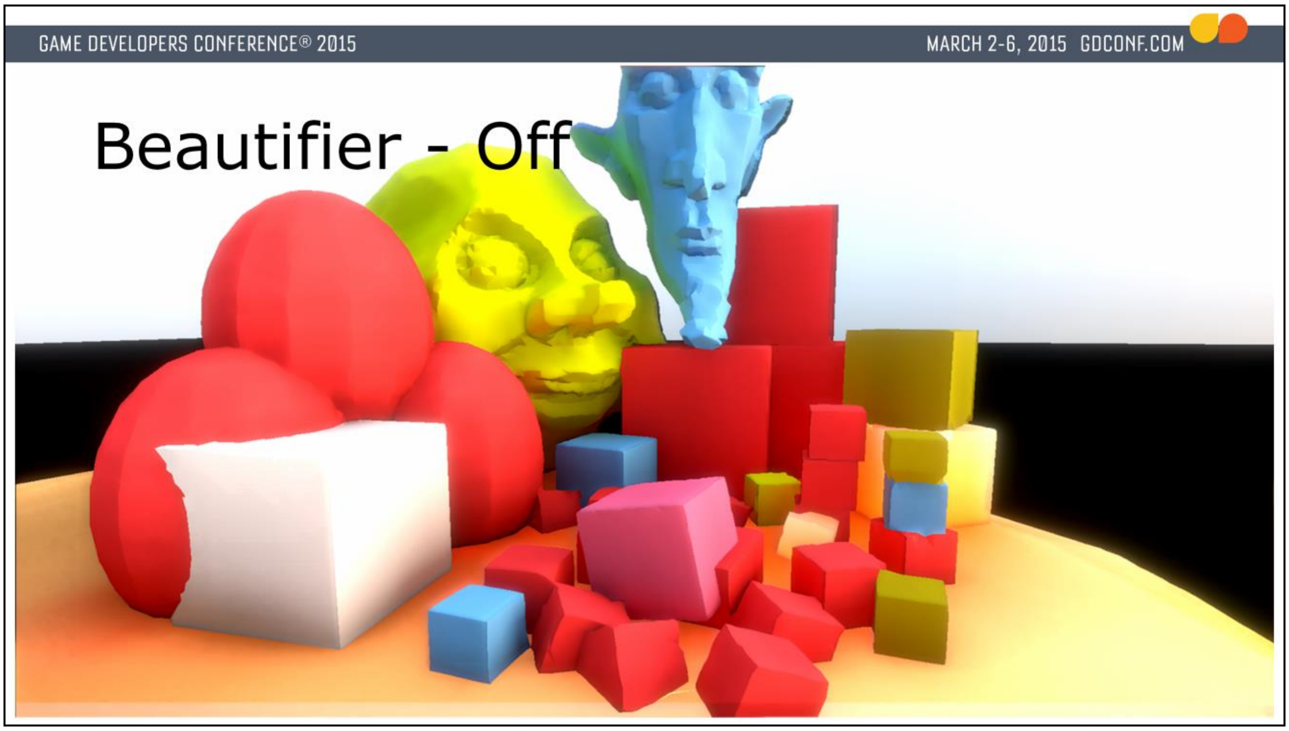
This beautifier hack was inspired by a voronoi rasterization paper "Voronoi Rasterization of Sparse Point Sets"



Beautification

- We wanted to preserve input model surface quality (carved look).
- Using a spectacular hack:
  - Render the models used for the booleans.
  - Keep their normals, and colour when they are close enough to the polygonized surface.

The method described in that paper didn't quite cut it for us, but inspired us to simply redraw the original polygonal parts on top and select source or destination pixel based on a depth threshold so that we could get greater detail in our normals and colours.
This turned out to be relatively fast, and even works when we dig.
It does however suffer from a few ghosting artifacts if you recreate different geometry that is close to what was originally there.

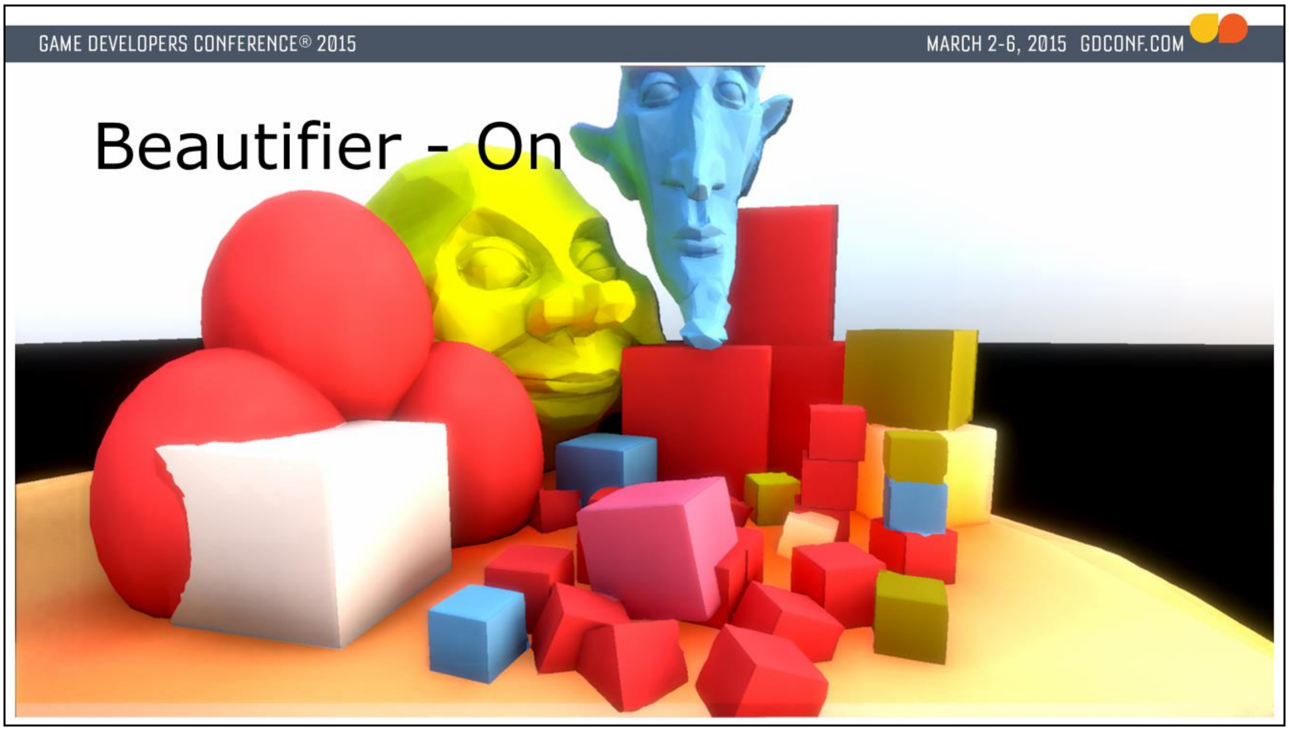


GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

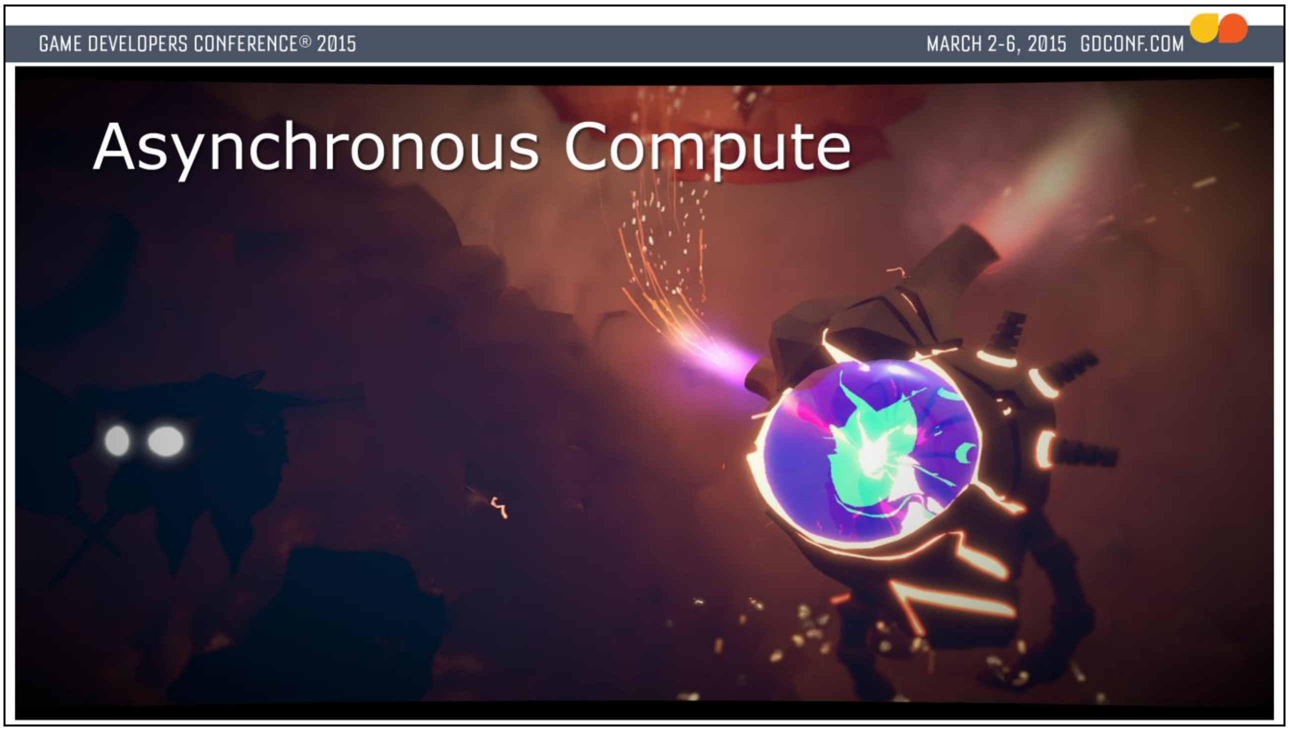
Beautifier - Off

So here you can see a scene with some of the problems that the beautifier is trying to solve.
Hopefully you can see that some edges don’t come out so well on the woman’s face.



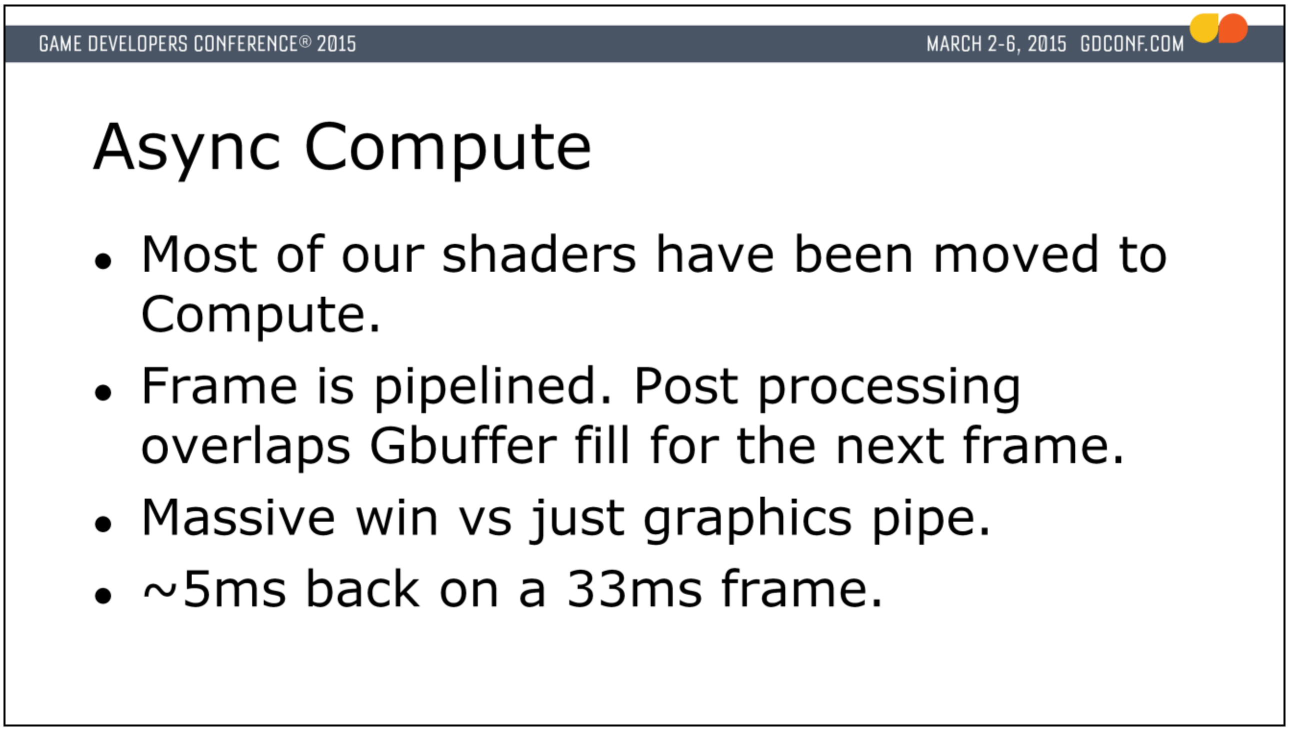
Beautifier - On

And now if we turn the beautifier on then you can see how much cleaner things get.



Asynchronous Compute

So, one thing I would like to mention before I finish is about our use of Async Compute.



Async Compute

- Most of our shaders have been moved to Compute.
- Frame is pipelined. Post processing overlaps Gbuffer fill for the next frame.
- Massive win vs just graphics pipe.
- ~5ms back on a 33ms frame.

We've used it very heavily throughout the project, and most of our Screen Space (or Voxel Space) work is in compute shaders,

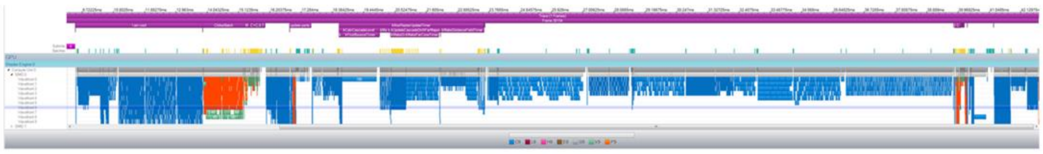
with large amounts of that running on 3 async compute queues that we have set up in addition to our graphics context.

On a heavy scene we get back around 5ms on a 33ms frame from using Async Compute.

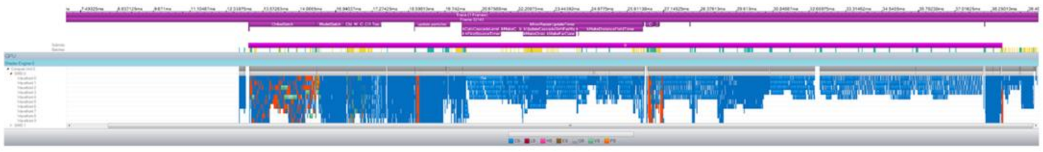
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Async Compute

Graphics Pipe: ~ 33ms



Async Compute: ~ 27ms



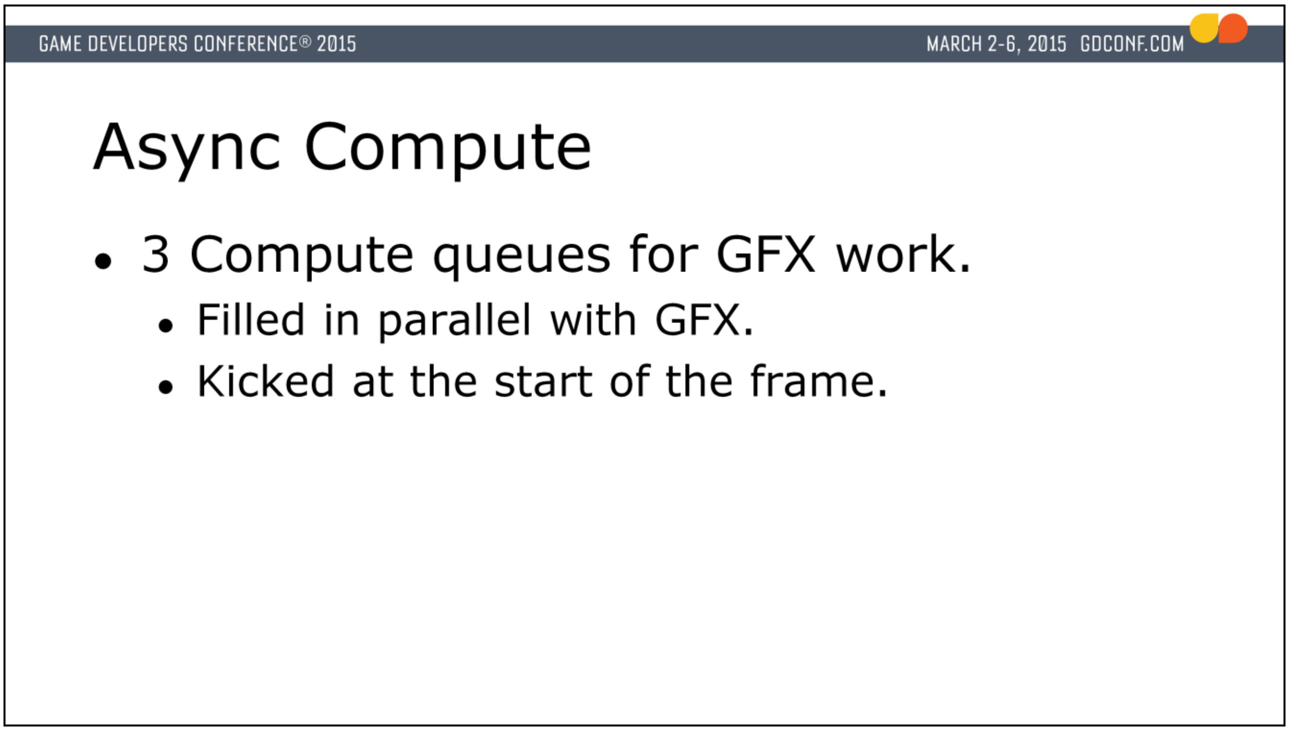
here is a RTTV capture of the same, fairly heavy scene.

On the top we're using just the graphics pipe.

On the bottom we're using Async Compute.

As you can see on the bottom, everything is a lot more overlapped, and we take about 5 or 6ms less.

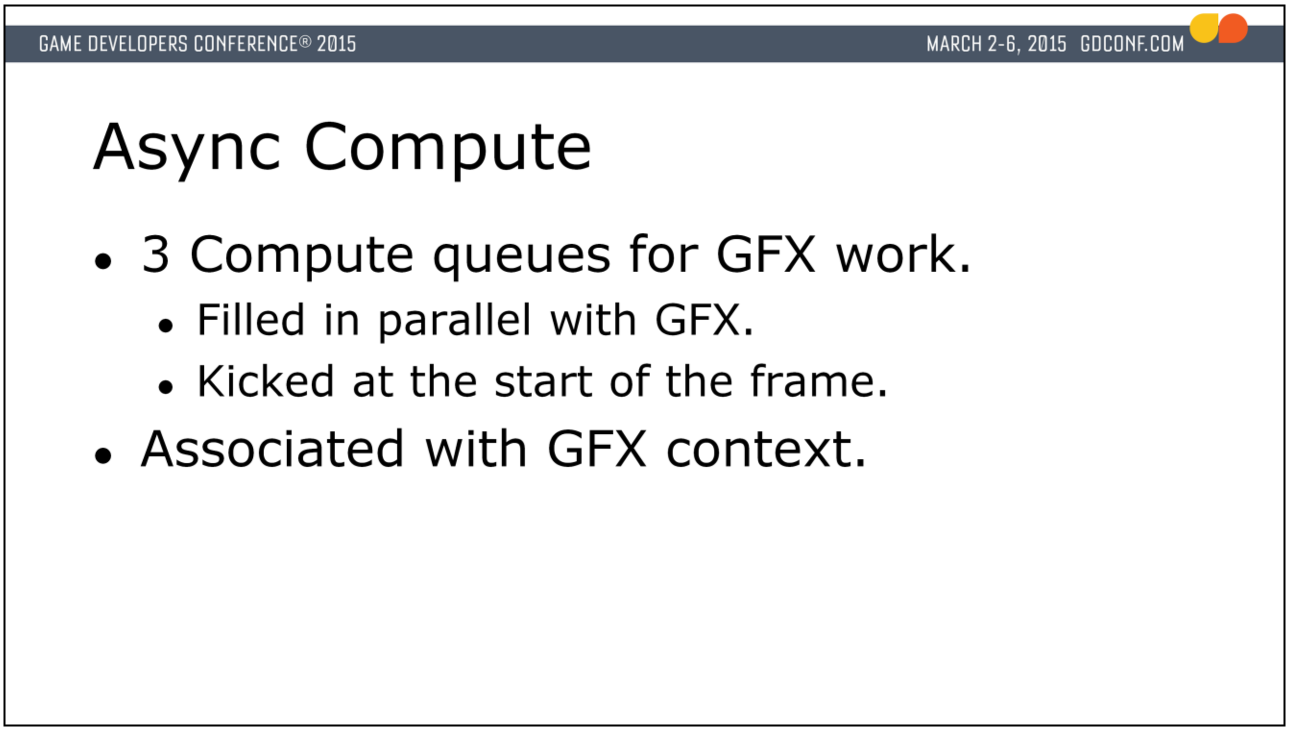
This is with exactly the same shaders, doing exactly the same work.



Async Compute

- 3 Compute queues for GFX work.
  - Filled in parallel with GFX.
  - Kicked at the start of the frame.

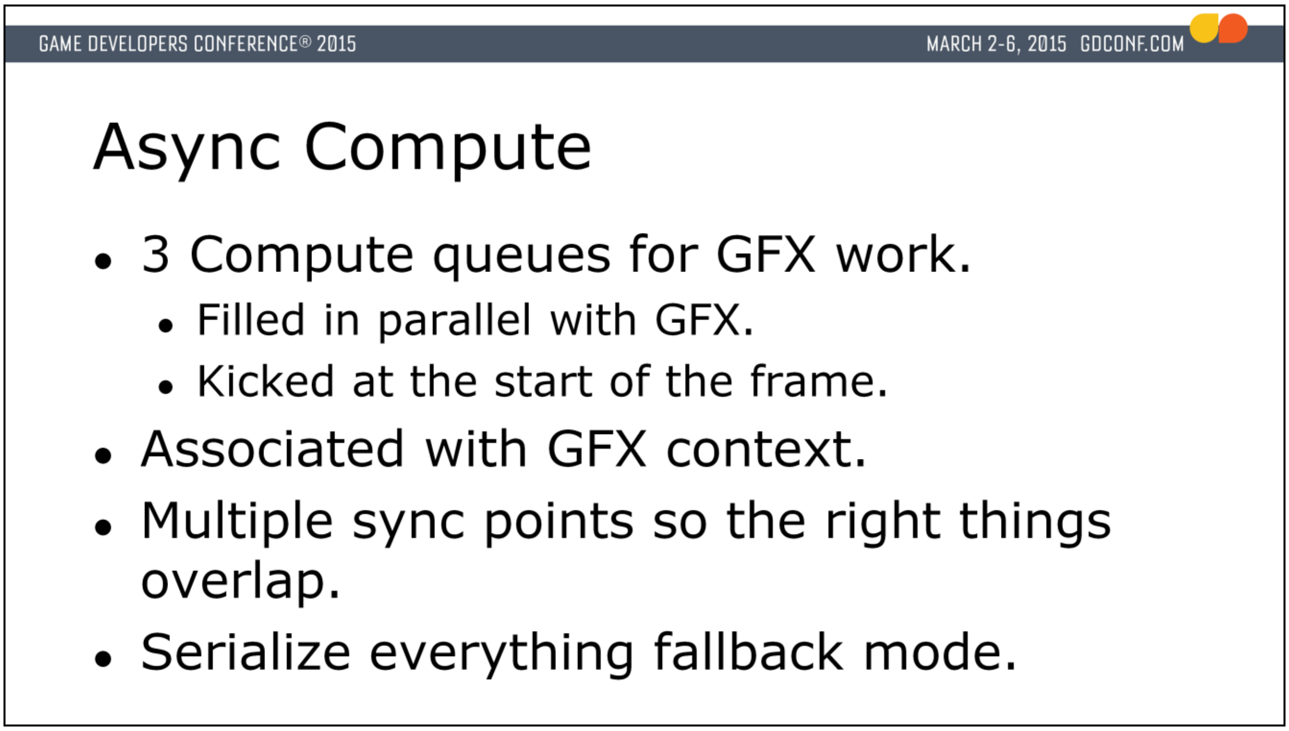
These 3 compute queues are filled at the same time we fill our graphics context , and kicked along with that work at the start of the frame.



Async Compute

- 3 Compute queues for GFX work.
  - Filled in parallel with GFX.
  - Kicked at the start of the frame.
- Associated with GFX context.

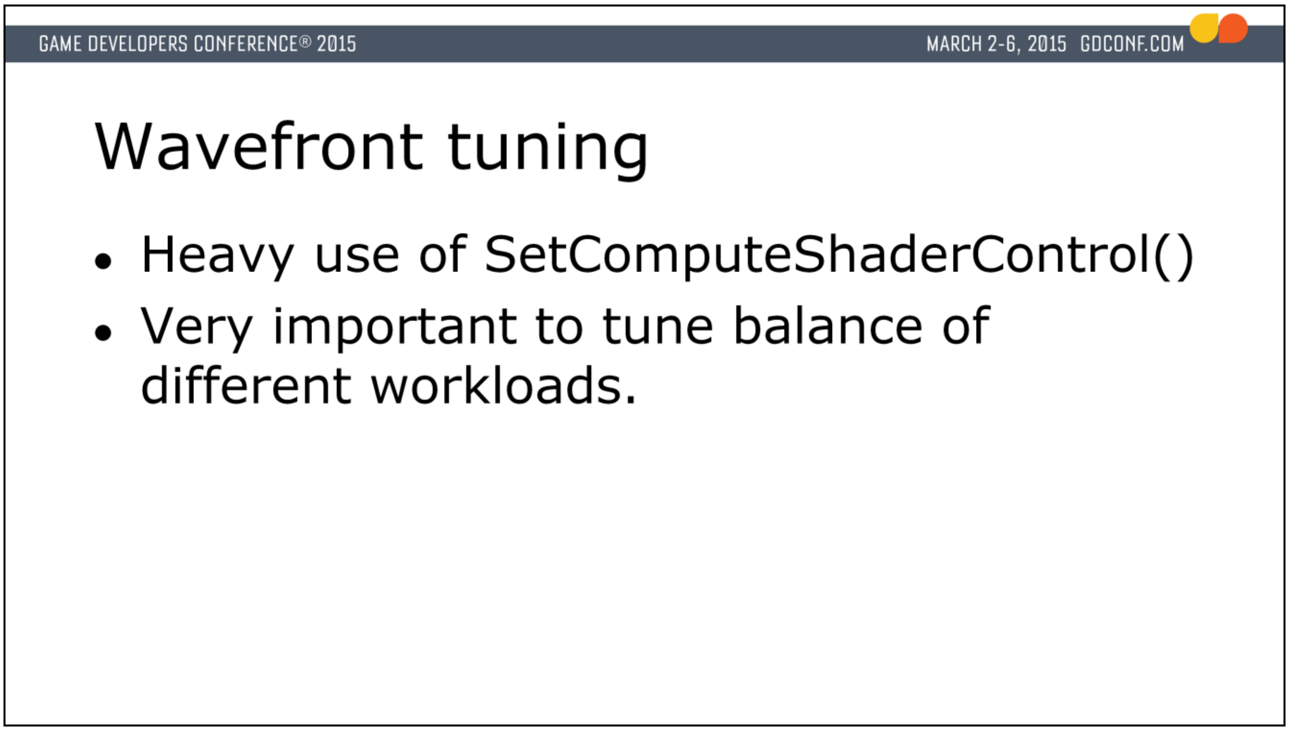
We have an interface setup where our graphics context also keeps track of the contexts for the compute queues, so our code, frequently asks for one of the 3 compute contexts, and fills commands in there instead of the graphics context.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Async Compute

- 3 Compute queues for GFX work.
  - Filled in parallel with GFX.
  - Kicked at the start of the frame.
- Associated with GFX context.
- Multiple sync points so the right things overlap.
- Serialize everything fallback mode.

We have some utility functions to enable easy synchronization between the queues and the main context, as well as a fallback mode that forces everything to run in serial, which is particularly helpful when you are tying to look at the fundamental performance of a particular shader.

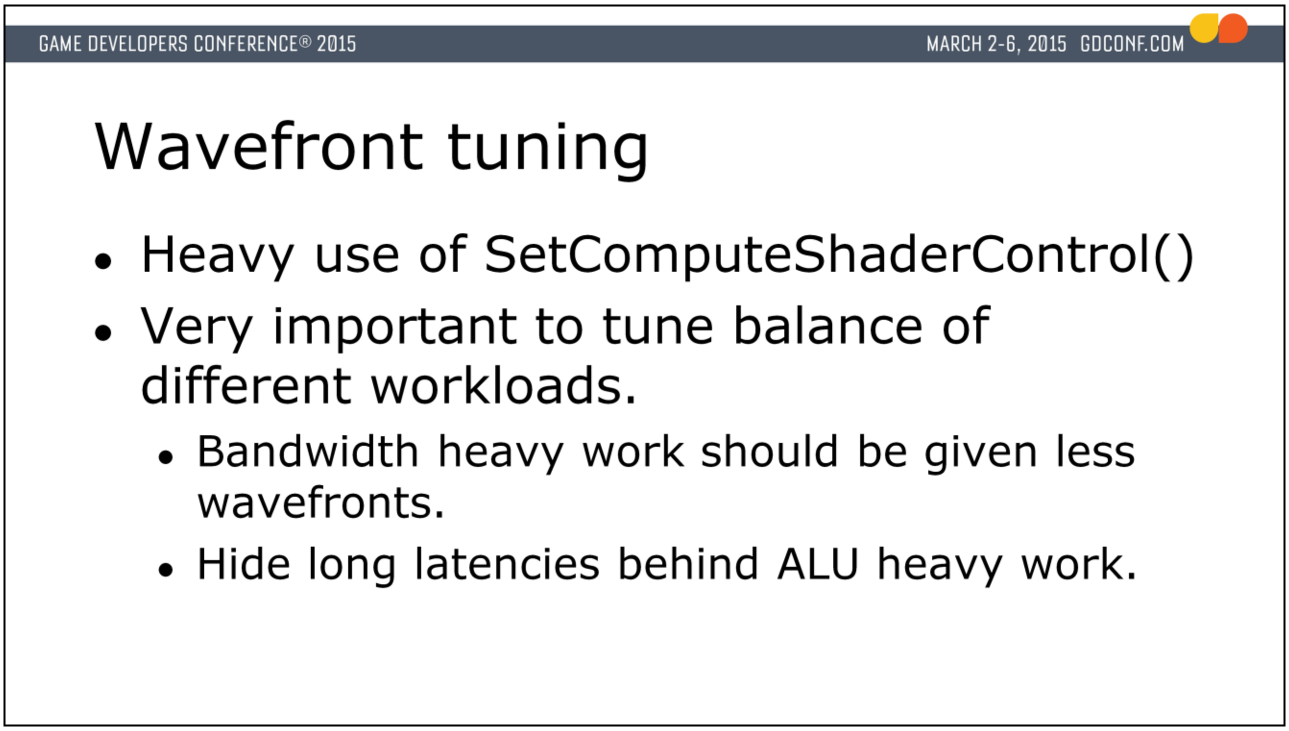

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Wavefront tuning

- Heavy use of SetComputeShaderControl()
- Very important to tune balance of different workloads.

One thing we discovered during the course of all this was that it's very important to tune the number of wavefronts that each queue, as well as the graphics context, can have in flight,

so that you get a nice mix of work on the GPU.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Wavefront tuning

- Heavy use of SetComputeShaderControl()
- Very important to tune balance of different workloads.
  - Bandwidth heavy work should be given less wavefronts.
  - Hide long latencies behind ALU heavy work.

In general we've found that we got the best balance if we toned down the number of wavefronts assigned to bandwidth heavy work, so the latency of texture fetches could be hidden behind useful ALU ops from a shader running on another pipe.

Anyway, I'd really encourage anyone working on the PS4 to take a look at async compute, if you haven't done so already

It's a very big win.

GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

Thankyou

james@q-games.com

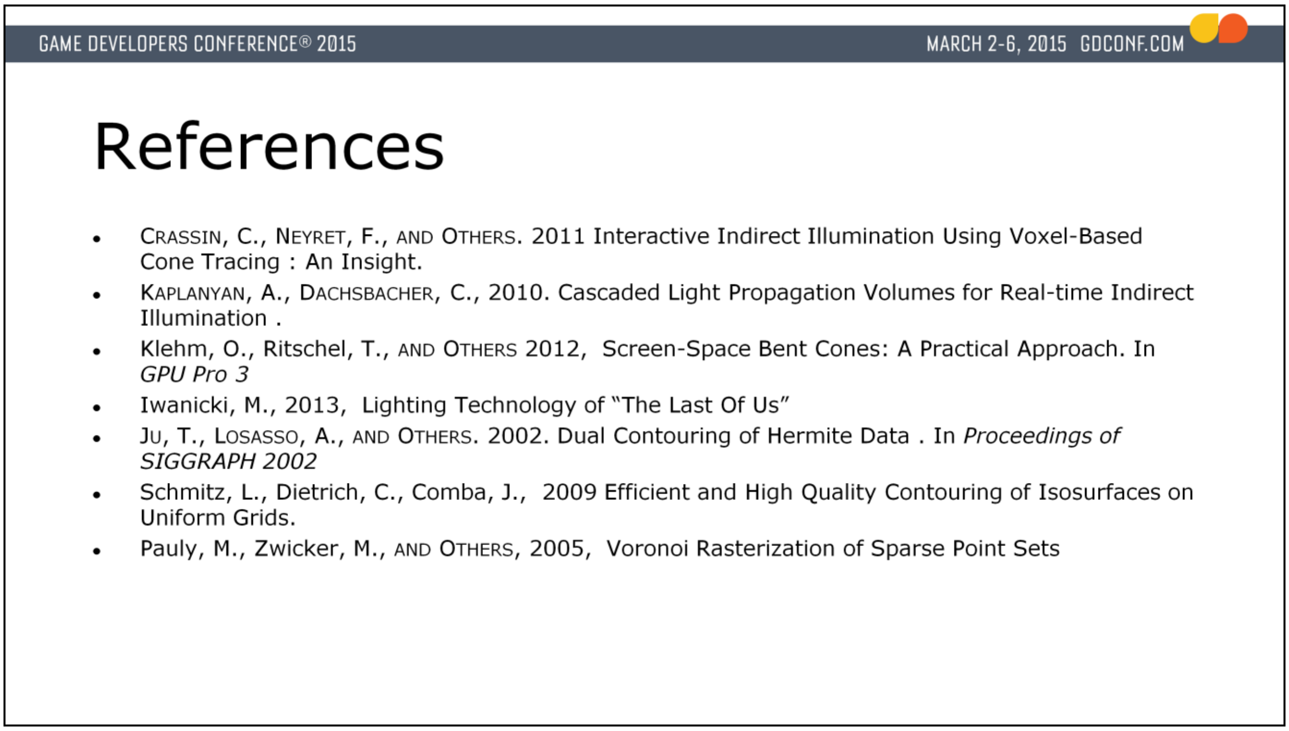
That’s all I have. Thankyou very much for listening, and I’ve hopefully got some time to take any questions that you have. I’ll bring up my good friend Jaymin Kessler to help out with the questions in case there is anything that he can field better than me.

If you can please speak into the mikes when asking the question and state your name and afflitiation when you do so.

And also don’t forget to fill in the online survery about this session that should have been sent to you while you were here.

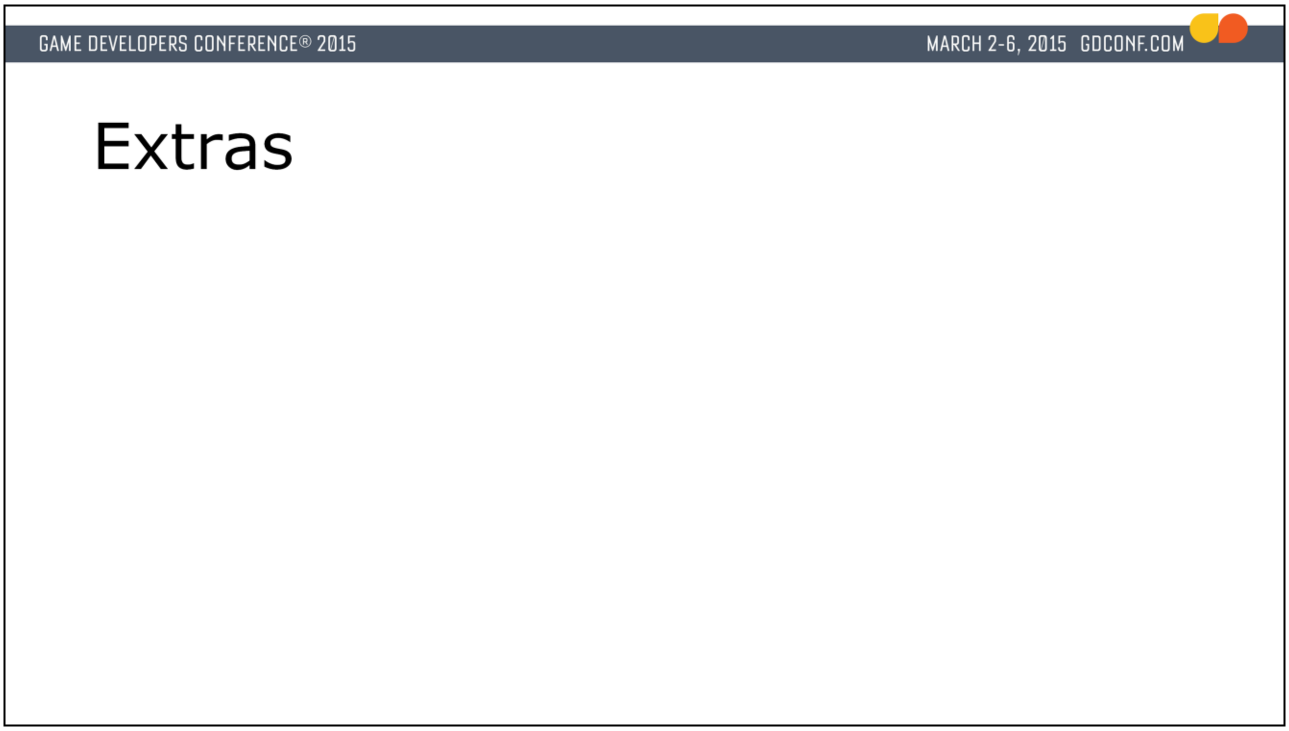
We're Hiring

recruit@q-games.com

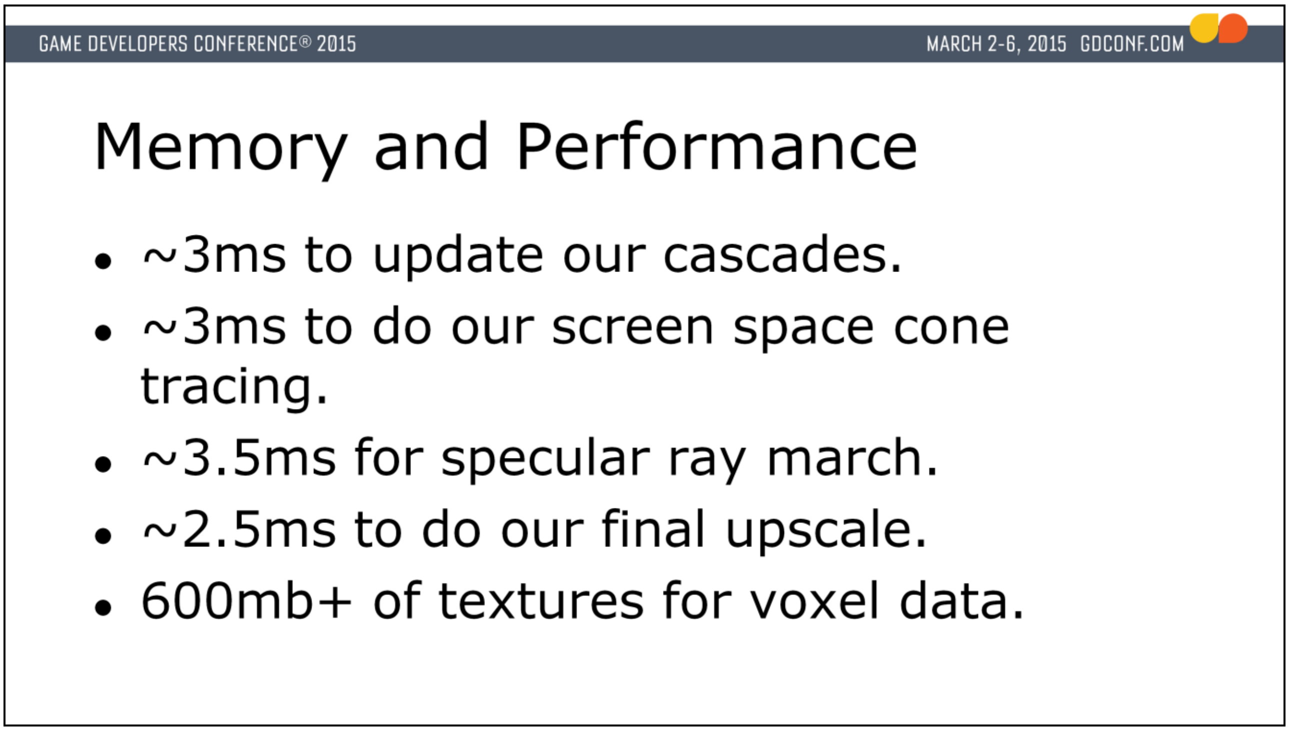


References

- CRASSIN, C., NEYRET, F., AND OTHERS. 2011 Interactive Indirect Illumination Using Voxel-Based Cone Tracing : An Insight.
- KAPLANYAN, A., DACHSBACHER, C., 2010. Cascaded Light Propagation Volumes for Real-time Indirect Illumination .
- Klehm, O., Ritschel, T., AND OTHERS 2012, Screen-Space Bent Cones: A Practical Approach. In *GPU Pro 3*
- Iwanicki, M., 2013, Lighting Technology of "The Last Of Us"
- JU, T., LOSASSO, A., AND OTHERS. 2002. Dual Contouring of Hermite Data . In *Proceedings of SIGGRAPH 2002*
- Schmitz, L., Dietrich, C., Comba, J., 2009 Efficient and High Quality Contouring of Isosurfaces on Uniform Grids.
- Pauly, M., Zwicker, M., AND OTHERS, 2005, Voronoi Rasterization of Sparse Point Sets



Extras


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Memory and Performance

- ~3ms to update our cascades.
- ~3ms to do our screen space cone tracing.
- ~3.5ms for specular ray march.
- ~2.5ms to do our final upscale.
- 600mb+ of textures for voxel data.

Just a few quick words before I wrap up about how expensive this all is.

It’s generally taking us on the order of about 3ms a frame to update our cascades, slightly more if we have to voxelize.

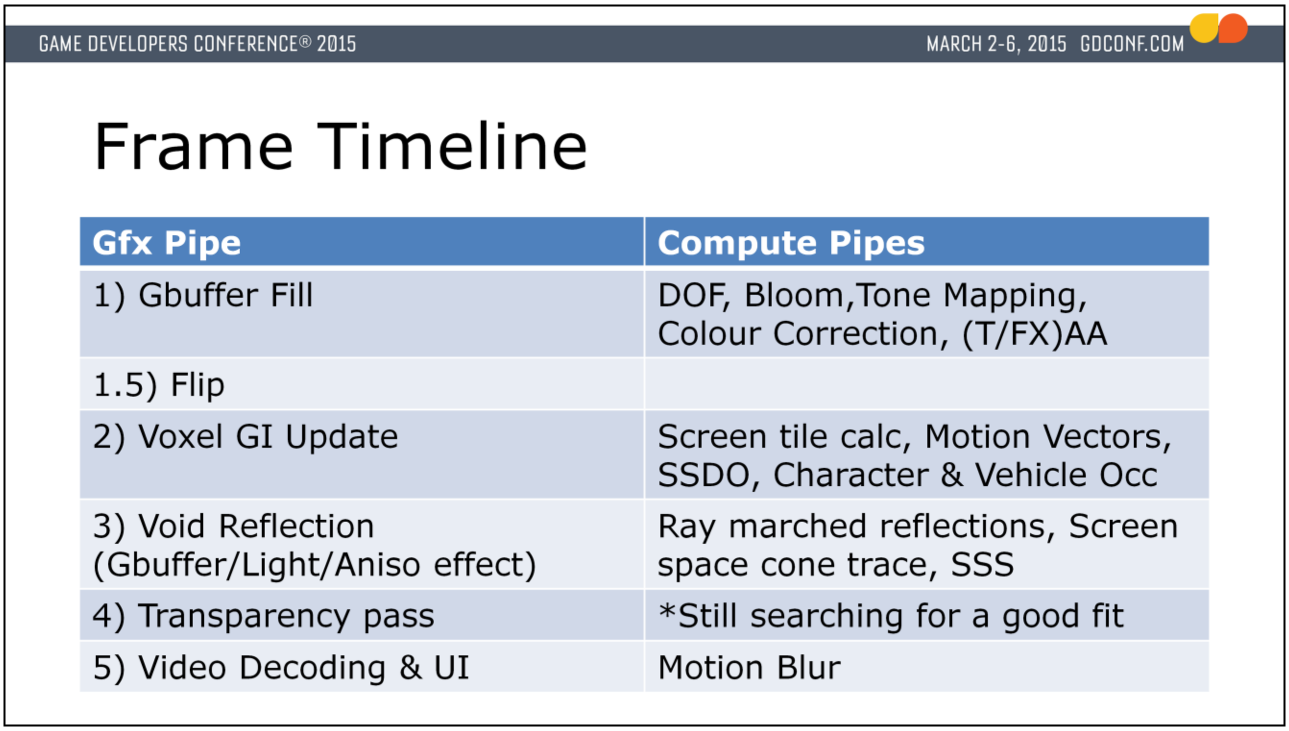
Our screen space cone tracing takes somewhere on the order of 3ms.

3.5 ms for the specular ray march,

and 2.5ms to do our final upscale and combine pass, that takes all the various elements, including SSDO and occlusion, and spits out a shaded pixel.

And of course, as you could probably guess from my repeated mention of the word “texture” we use a rather large amount of memory for textures,

currently somewhere north of 600mb.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Frame Timeline

Gfx Pipe	Compute Pipes
1) Gbuffer Fill	DOF, Bloom,Tone Mapping, Colour Correction, (T/FX)AA
1.5) Flip	
2) Voxel GI Update	Screen tile calc, Motion Vectors, SSDO, Character & Vehicle Occ
3) Void Reflection (Gbuffer/Light/Aniso effect)	Ray marched reflections, Screen space cone trace, SSS
4) Transparency pass	*Still searching for a good fit
5) Video Decoding & UI	Motion Blur

So just to give you an idea of how pipelined everything has become for us. Here’s a quick overview of what happens when in our frame.

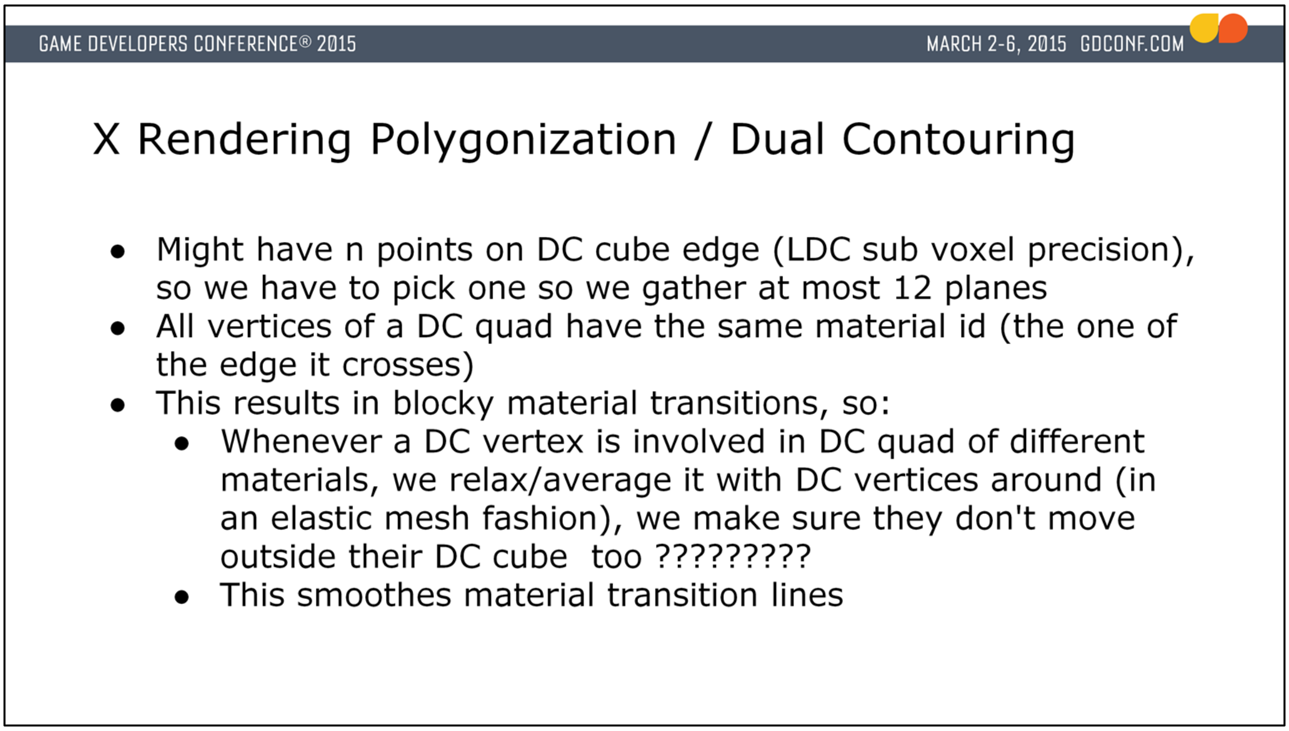
As you can see we have lots of post effect work overlapped with our Gbuffer fill pass. This really helps us to mop up all the holes that we usually have when vertex shader work dominates, and there wouldn’t usually be enough wavefronts to fill the GPU.

We flip immediately after this, and then we move on to updating the Voxel GI. This has been left on the graphics pipe because we need fragment and vertex shaders for voxelization.

In parallel with this, we calculate some tile lists for from the info we have in our Gbuffers, as well as doing motion blur and our occlusion calculations.

We have a second Gbuffer pass that we use for our anisotropic reflections, and we do this in parallel with most of our backend Screen space cone tracing and reflection work.

And then we finish off with the transparency pass, which we currently don’t have that much that we can overlap with, and our video decoding and UI rendering, which we manage to overlap with some more motion blur work.



X Rendering Polygonization / Dual Contouring

- Might have n points on DC cube edge (LDC sub voxel precision), so we have to pick one so we gather at most 12 planes
- All vertices of a DC quad have the same material id (the one of the edge it crosses)
- This results in blocky material transitions, so:
    - Whenever a DC vertex is involved in DC quad of different materials, we relax/average it with DC vertices around (in an elastic mesh fashion), we make sure they don't move outside their DC cube too ?????????
    - This smoothes material transition lines

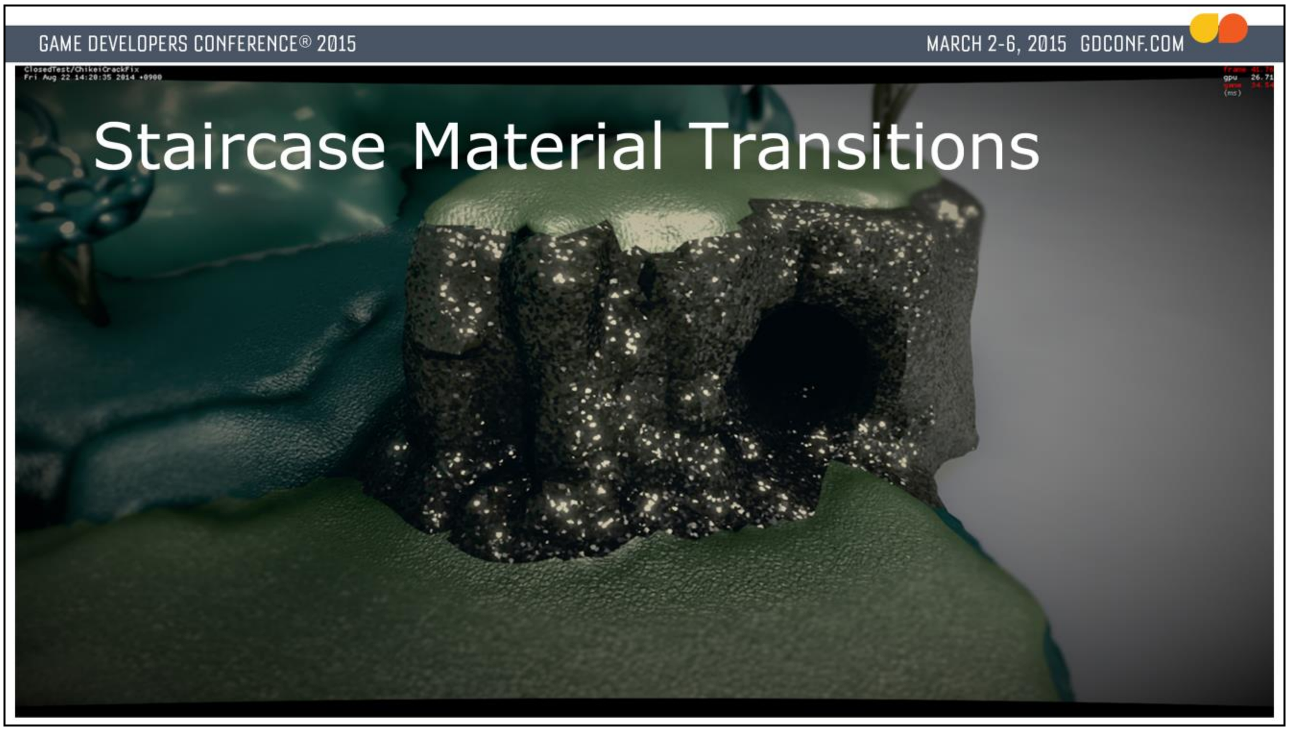
There are a few issues when using dual contouring in our fixed 0.2 “voxel size” setup.

First is that LDC might enable an arbitrary amount of surface planes along DC cube edges.

We have to pick at most one per edge so we have at most 12 surface planes per DC cube (don’t really want to do anything more complicated than that!)

Secondly those planes might have different material IDs and that results in very staircase like transition.

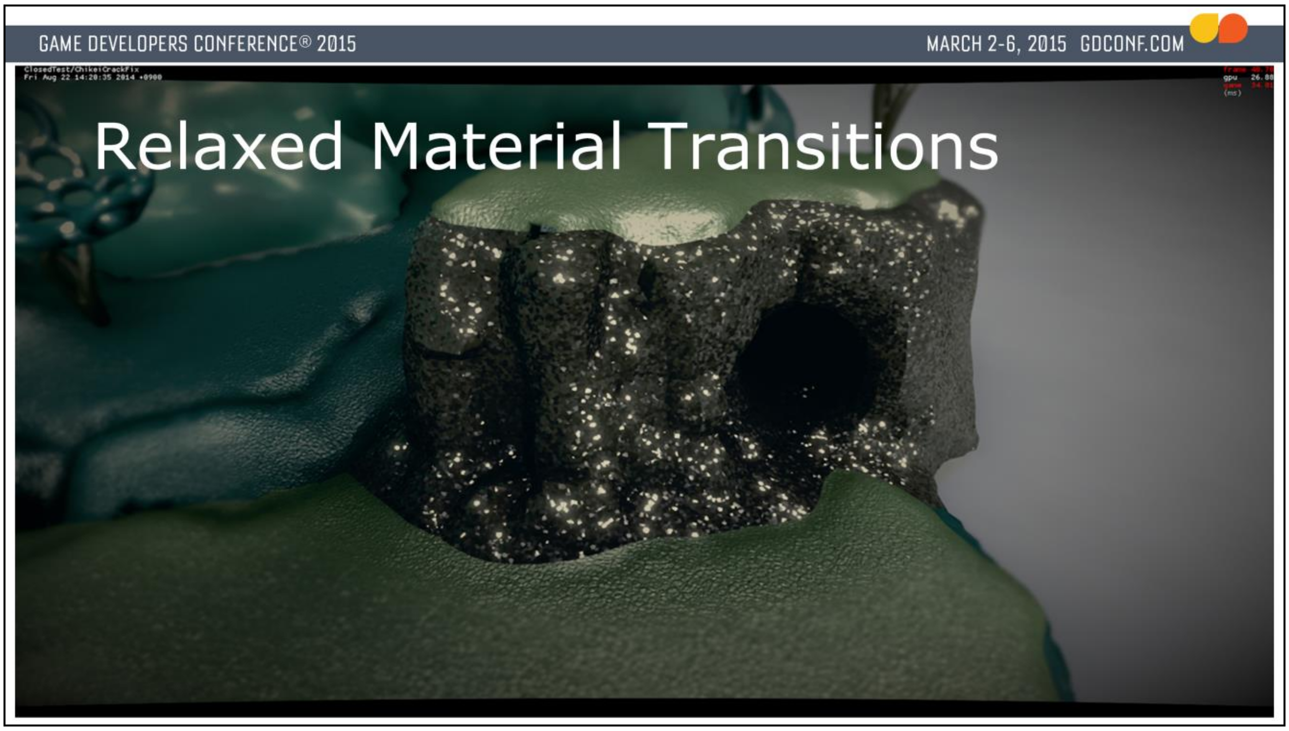
For DC vertex that were calculated from set of surface planes with different materials, we “relax” them in an elastic net fashion to make the material transition boundary more pleasing to the eye.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Staircase Material Transitions

So here you can see how our material transitions look if we don't do this relaxation step.

As you can see the grass/rock transition look kind of horrible and blocky



Relaxed Material Transitions

But after elastic net relaxations of DC vertices, the material transition look a lot easier on the eye.