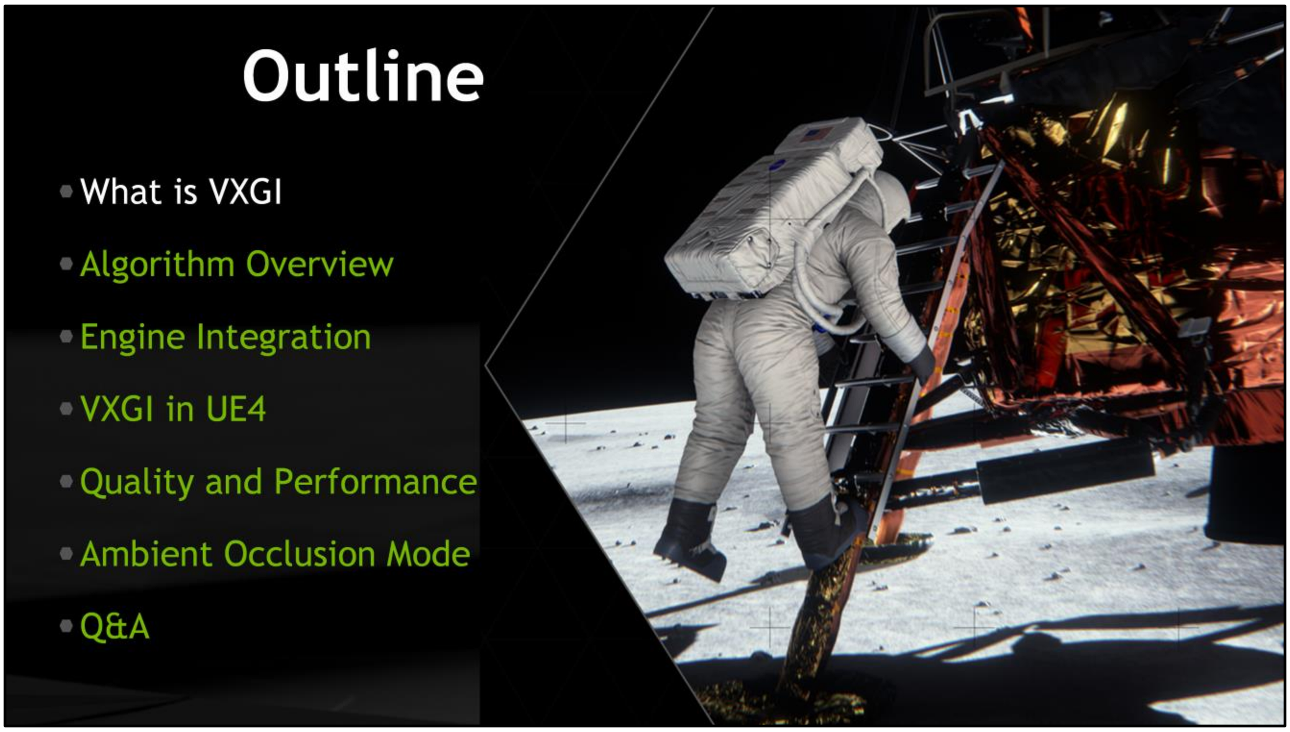
NVIDIA GEFORCE GTX

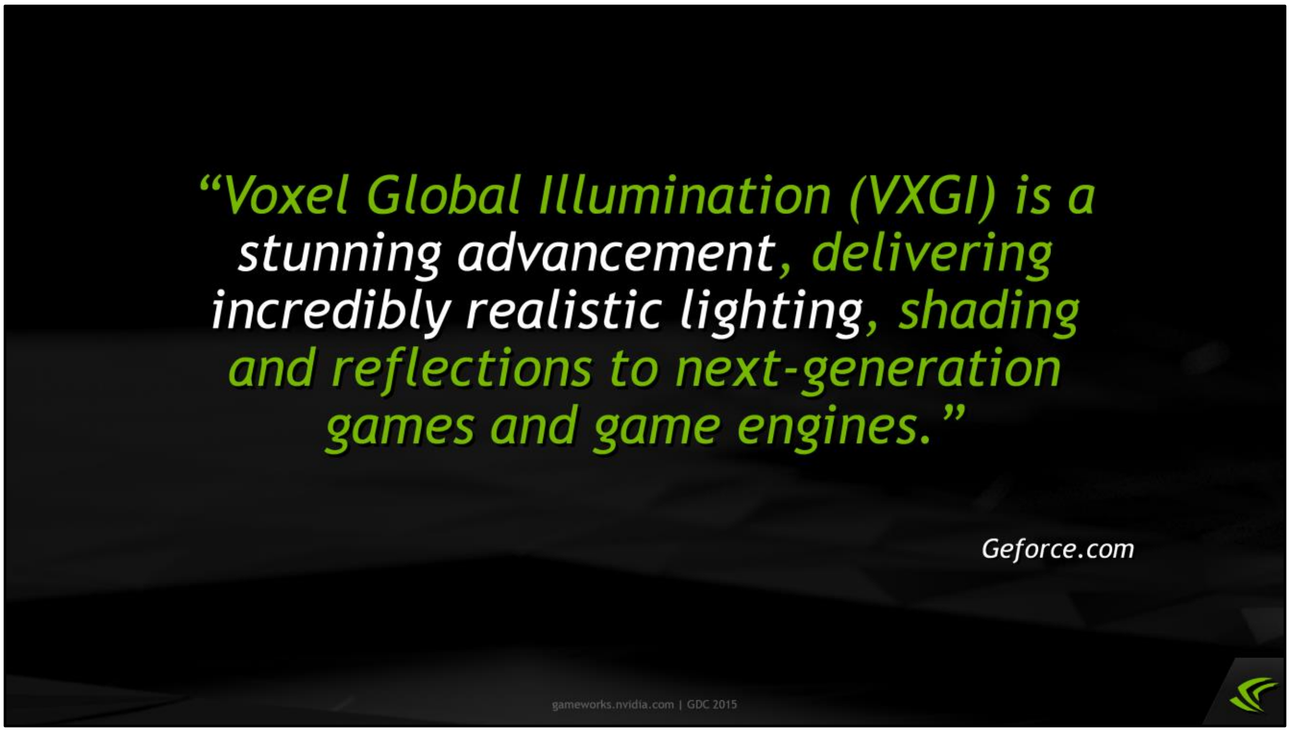
NVIDIA VXGI: Dynamic Global Illumination for Games

Alexey Panteleev
Developer Technology Engineer, NVIDIA



Outline

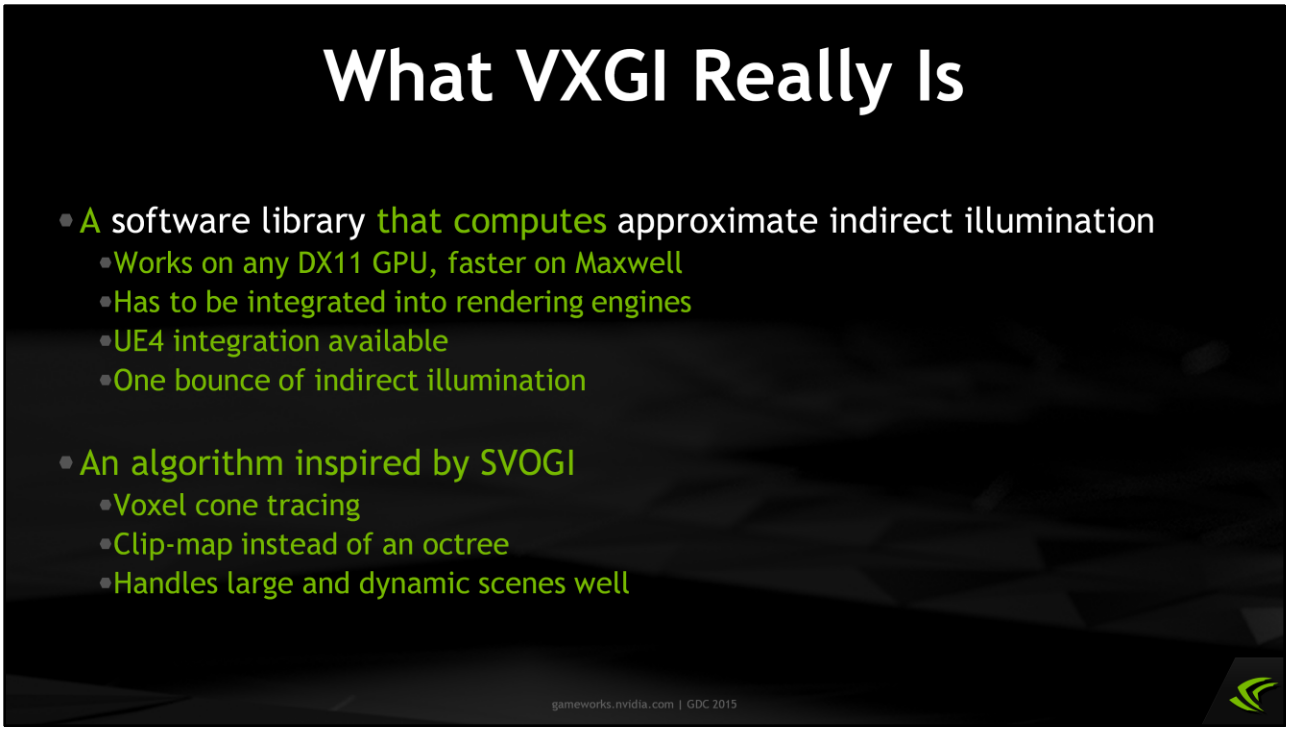
- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A


"Voxel Global Illumination (VXGI) is a stunning advancement, delivering incredibly realistic lighting, shading and reflections to next-generation games and game engines."

Geforce.com

gameworks.nvidia.com | GDC 2015

After the release of Maxwell in September last year, a number of press articles appeared that describe VXGI simply as a technology to improve lighting in games. While that is certainly true, it doesn't tell you anything about what VXGI really is. So I'm going to clarify what it is now.



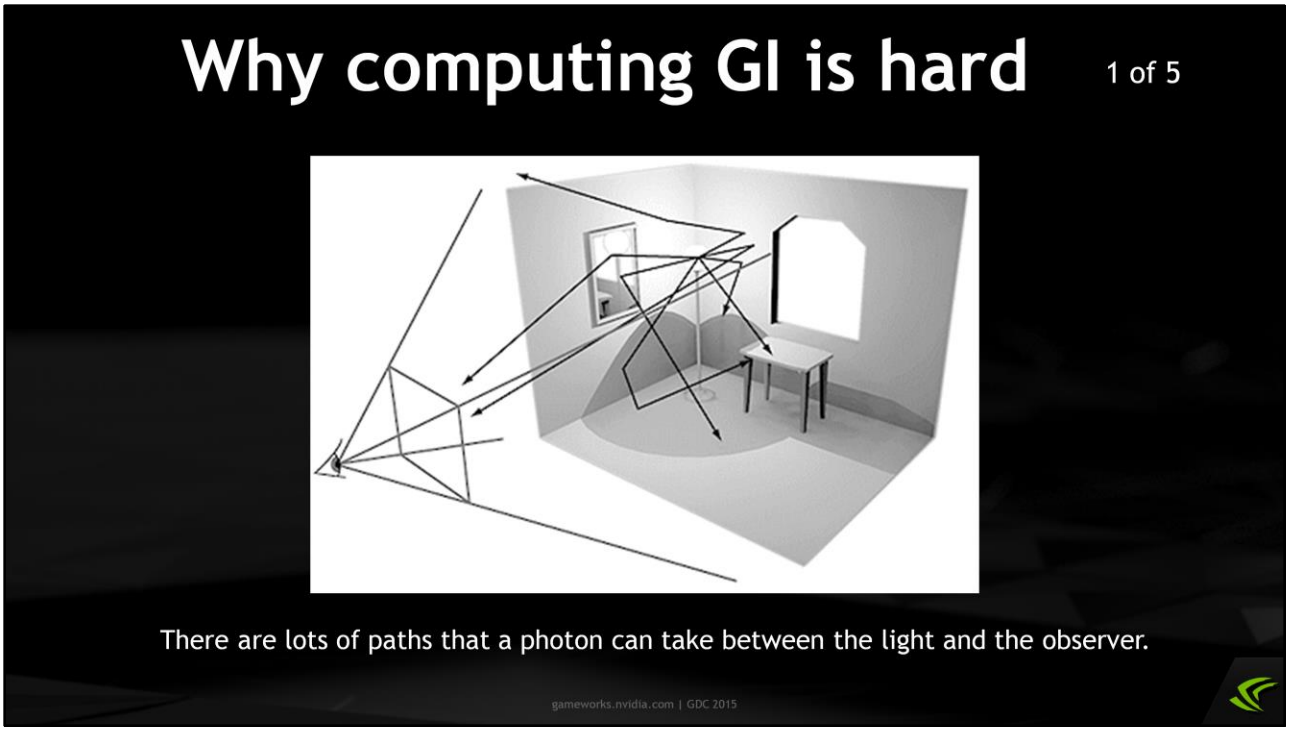
What VXGI Really Is

- A software library that computes approximate indirect illumination
  - Works on any DX11 GPU, faster on Maxwell
  - Has to be integrated into rendering engines
  - UE4 integration available
  - One bounce of indirect illumination
- An algorithm inspired by SVOGI
  - Voxel cone tracing
  - Clip-map instead of an octree
  - Handles large and dynamic scenes well

gameworks.nvidia.com | GDC 2015

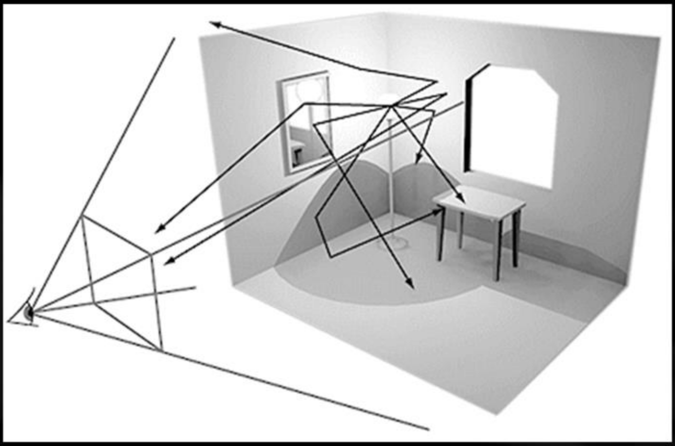
First of all, VXGI is a software library implementing a complicated rendering technique. It is not built into the driver, you cannot use a switch in the NVIDIA control panel to have it automatically applied to any game like you can with FXAA for example. It has to be integrated into the rendering engine, and sometimes the integration can be challenging. So we integrated VXGI into Unreal Engine 4 for you to be able to use it relatively easily.

VXGI takes the geometry in your scene and uses it to compute indirect diffuse and indirect specular illumination. That is done using an algorithm known as Voxel Cone Tracing. It is similar to SVOGI, or Sparse Voxel Octree Global Illumination, with one important difference: instead of an octree, we use a clip-map, which is much faster and which allows us to process large and fully dynamic scenes in real time. No preprocessing of any kind is required.



Why computing GI is hard

1 of 5

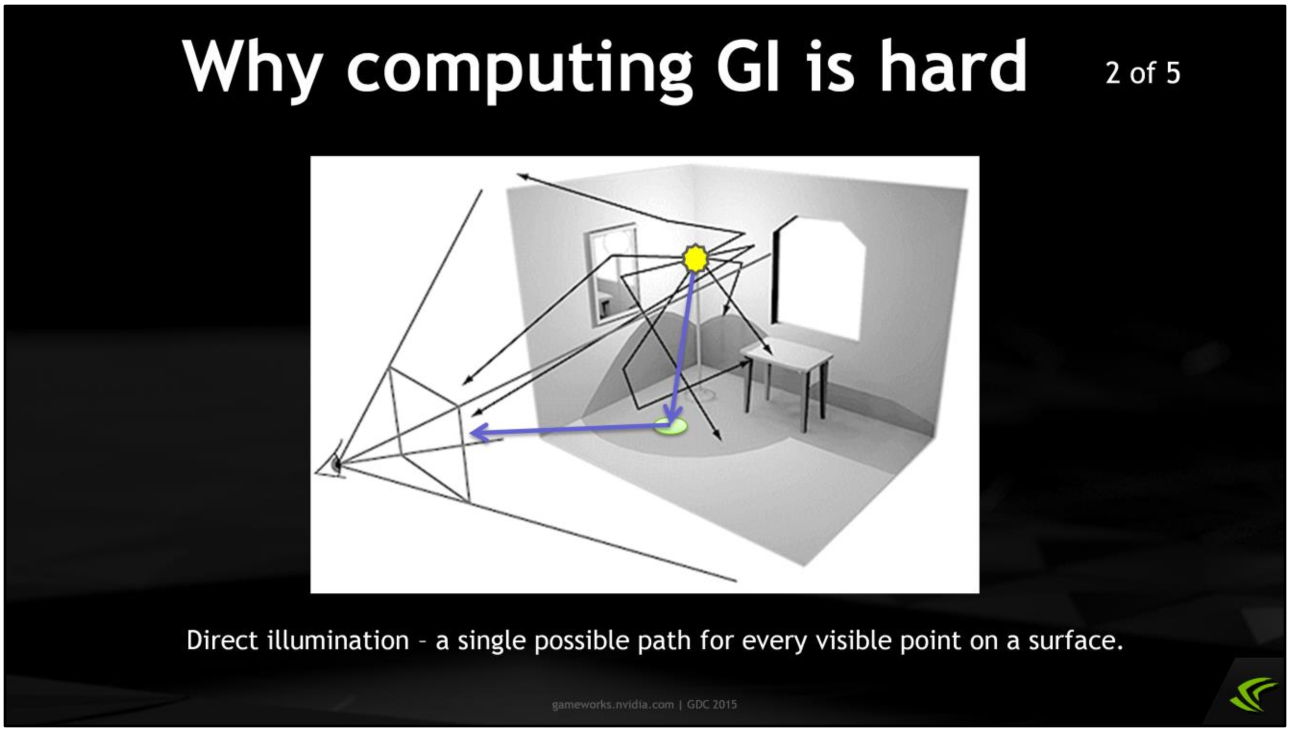


There are lots of paths that a photon can take between the light and the observer.

gameworks.nvidia.com | GDC 2015

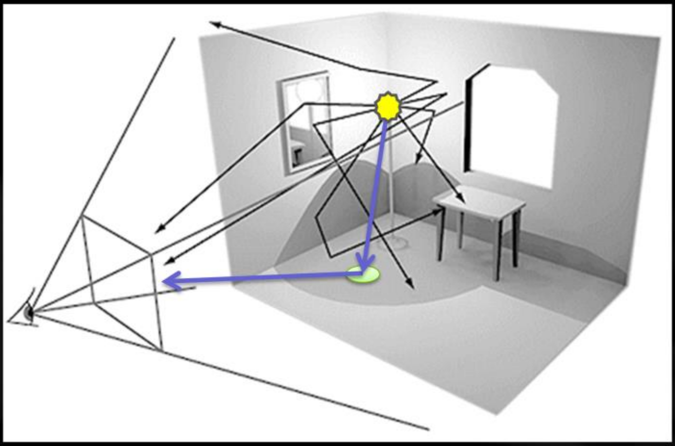
Now let me explain why computing global illumination is so hard, why real-time GI is such a big deal and what prevented it from happening 10 years ago.

The problem is, physical light emitters produce an immense number of photons per second. And the number of different paths that a photon can take from the emitter to the observer is virtually infinite. So we use integral approximations that deal with rays or beams, but even with them the problem is still computationally hard.



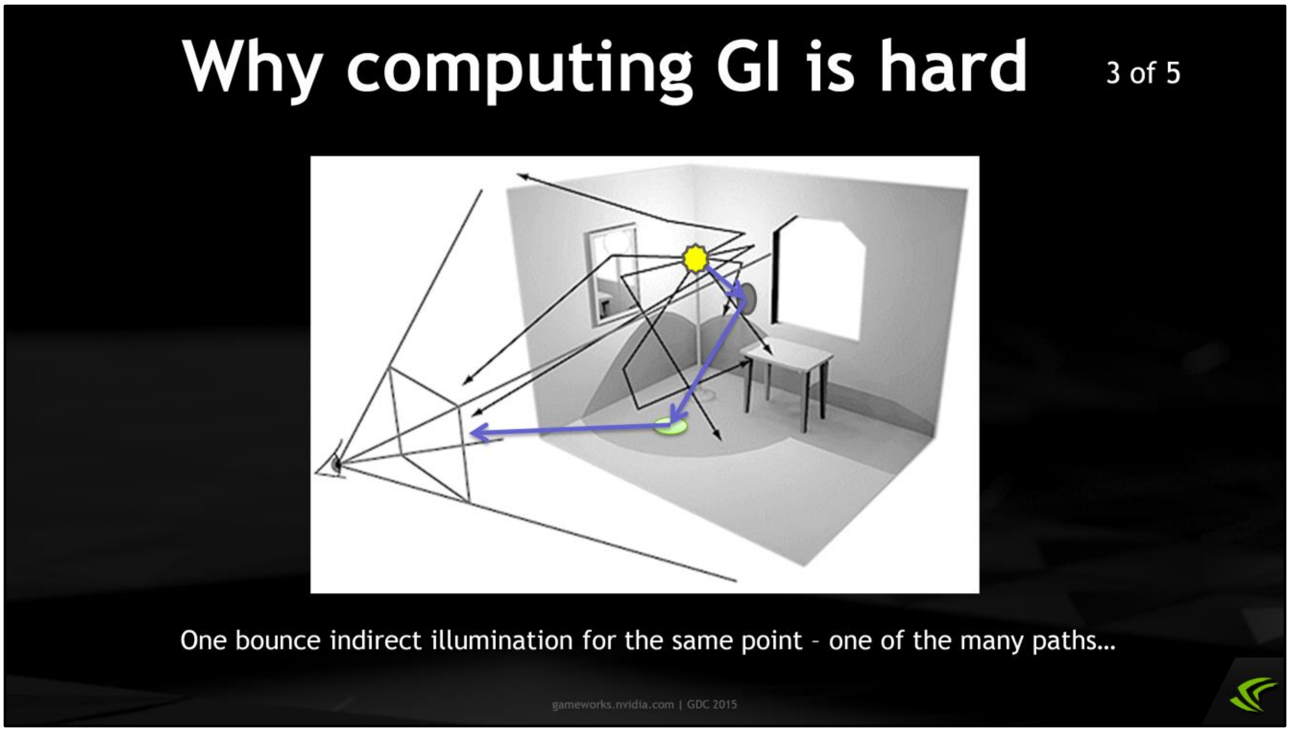
Why computing GI is hard

2 of 5



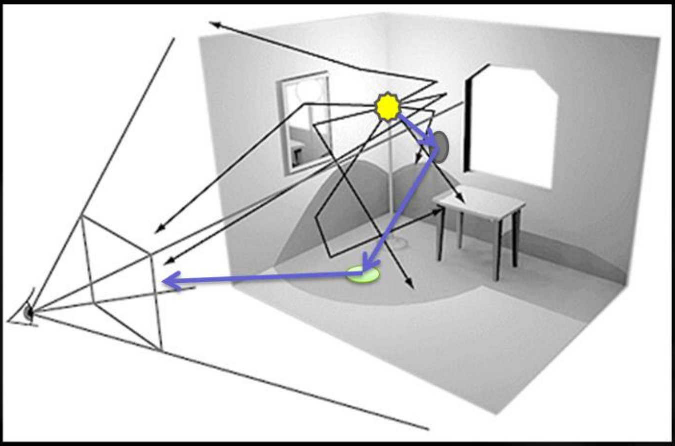
Direct illumination – a single possible path for every visible point on a surface.

Direct illumination is relatively easy to compute because there is only a single path that leads from the emitter to the observer with one reflection.



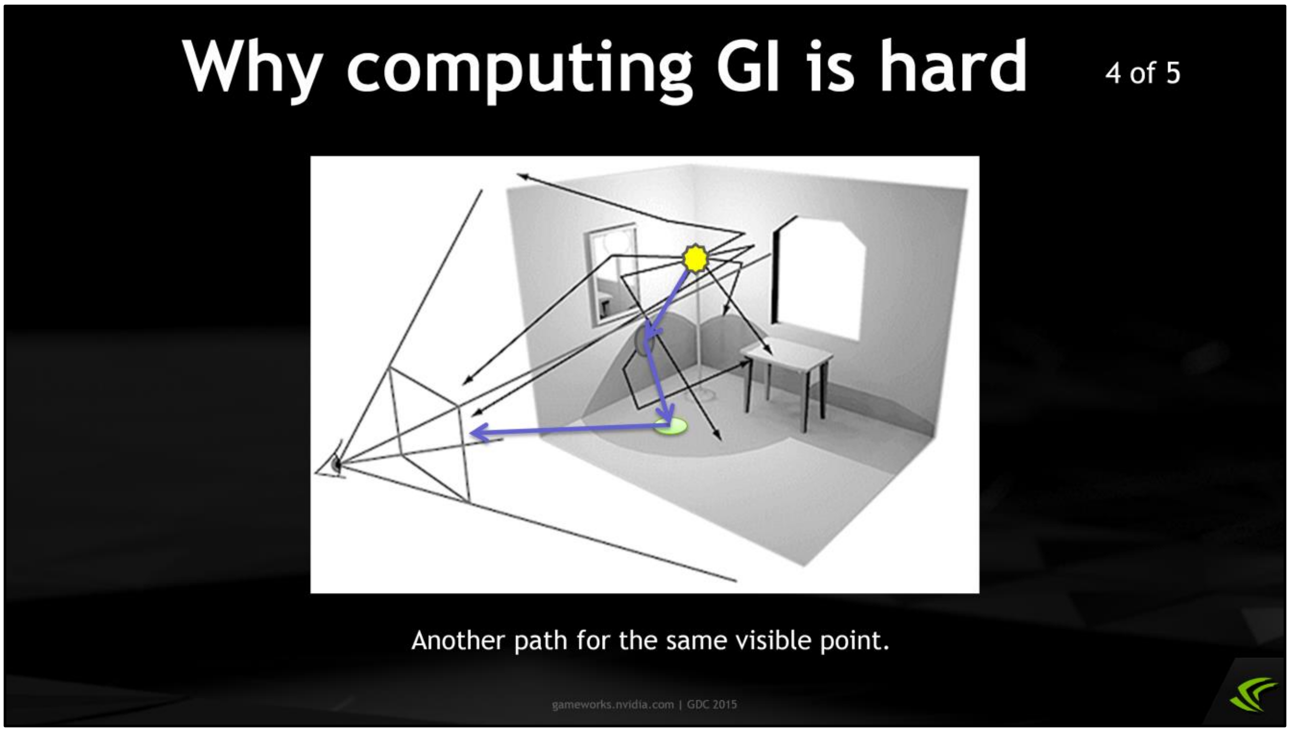
Why computing GI is hard

3 of 5



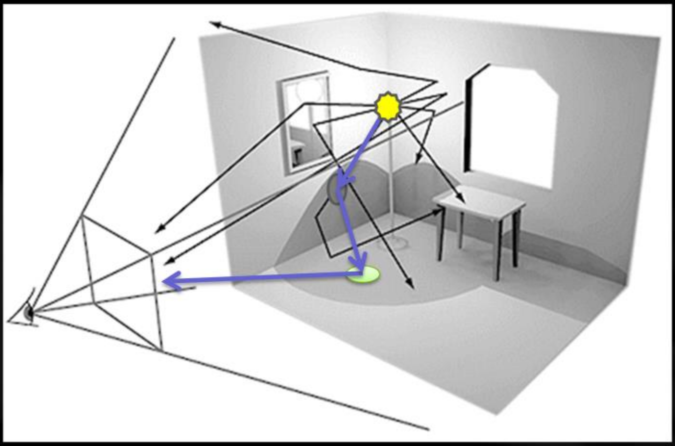
One bounce indirect illumination for the same point - one of the many paths…

One bounce indirect illumination is much harder because there are many possible paths from the emitter to the final surface visible to the observer.



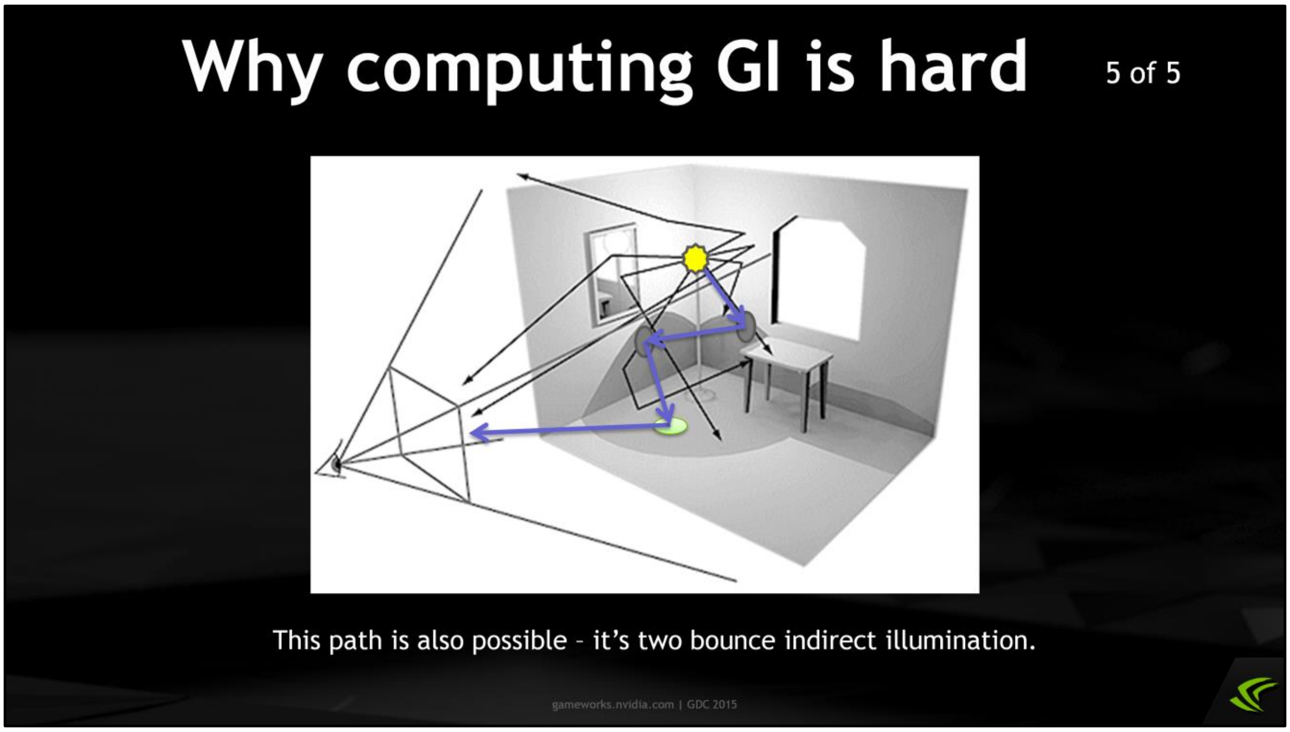
Why computing GI is hard

4 of 5



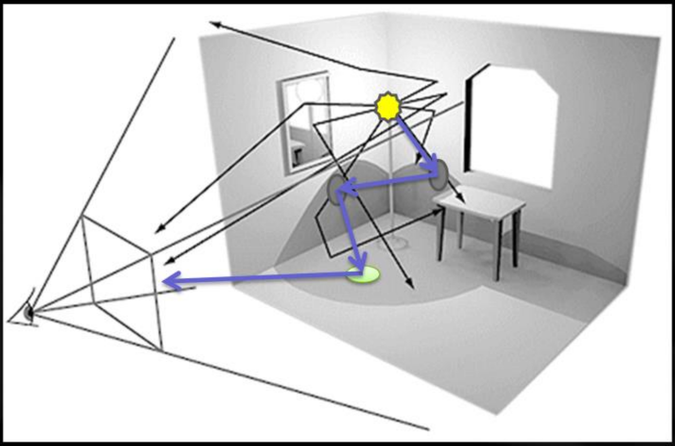
Another path for the same visible point.

This is a different path for the same emitter, observer, and the final visible surface – one of the many.



Why computing GI is hard

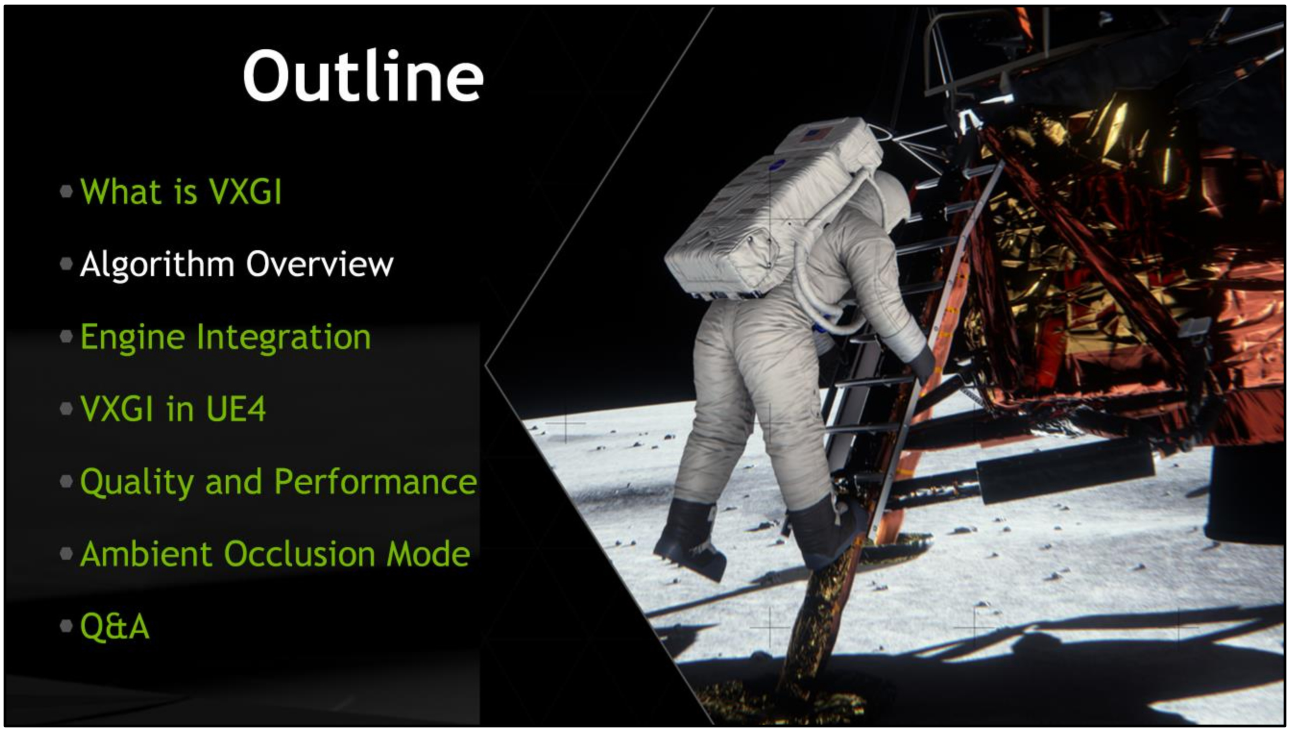
5 of 5



This path is also possible – it's two bounce indirect illumination.

gameworks.nvidia.com | GDC 2015

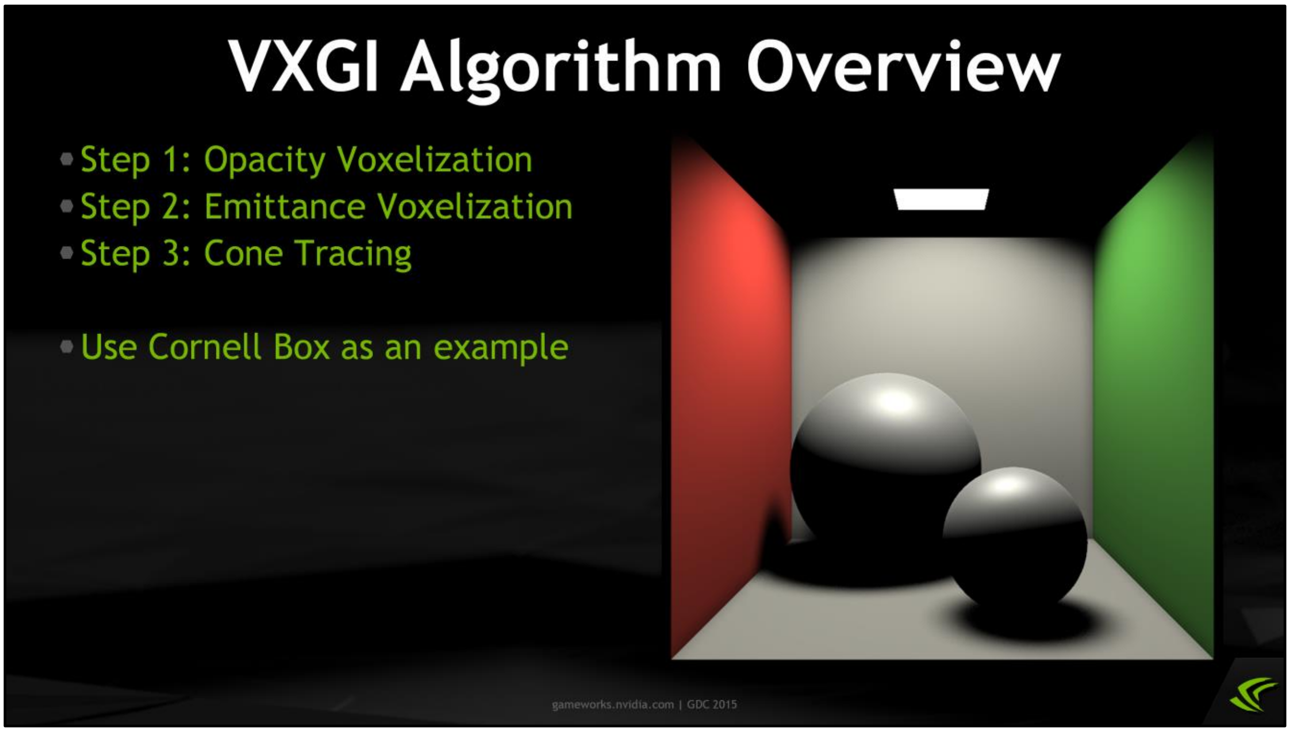
There are paths with more than two reflections, and computing them makes two or N bounce indirect illumination. The exact solution gets exponentially more expensive as the number of steps increase. So we have to use coarse approximations to make GI computation times practical, and even more coarse approximations to make it happen in real time.



Outline

- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A

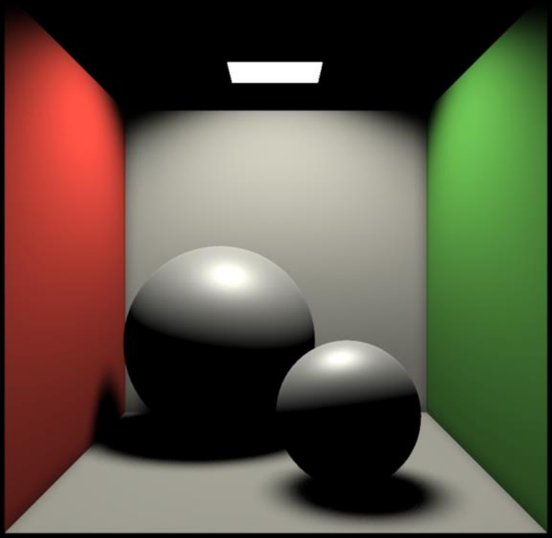
Now let’s talk about the algorithm that is used in VXGI, without getting too deep into the details.



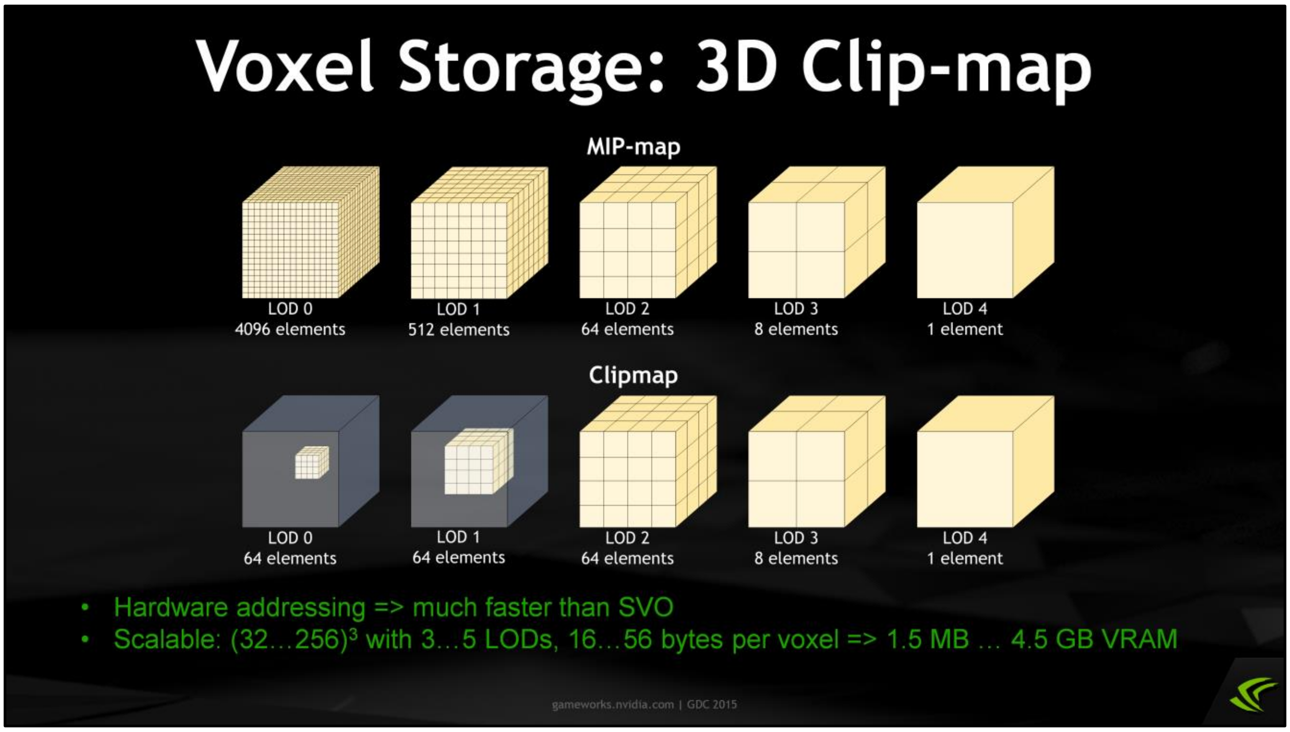
VXGI Algorithm Overview

- Step 1: Opacity Voxelization
- Step 2: Emittance Voxelization
- Step 3: Cone Tracing

- Use Cornell Box as an example



The algorithm consists of three basic steps: opacity voxelization; emittance voxelization; and cone tracing. Let’s take a look at each of the steps using the well known Cornell Box scene as an example.



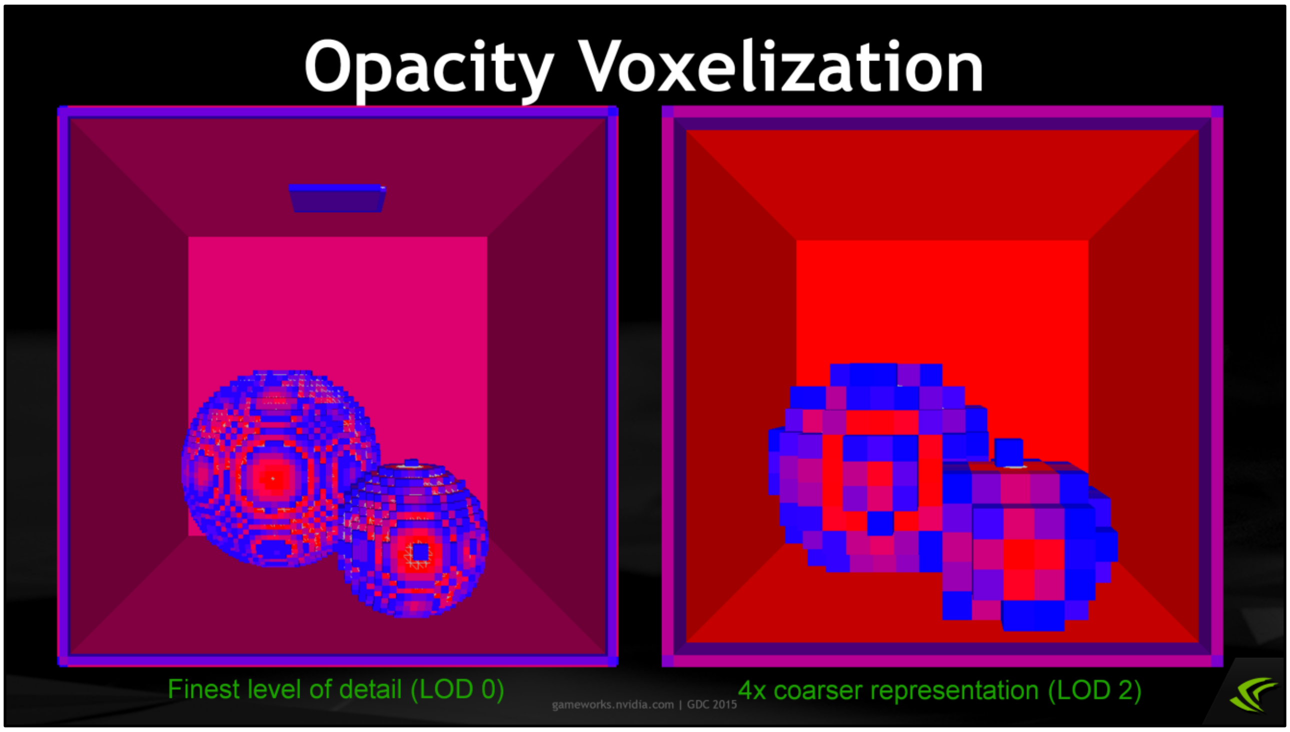
Voxel Storage: 3D Clip-map

- Hardware addressing => much faster than SVO
- Scalable: $(32…256)^3$ with 3…5 LODs, 16…56 bytes per voxel => 1.5 MB … 4.5 GB VRAM

gameworks.nvidia.com | GDC 2015

First of all, the algorithm deals with voxel data. And the structure that we store the voxel data in is very important. Previous implementations of voxel cone tracing used an octree, which is memory efficient but very slow to update. Other implementations used cascaded 3D textures. Our version uses a clip-map, which is similar to a cascaded texture. It differs from a regular mip-map in the finer levels of detail, which are clipped to cover a smaller region of space – hence the name, clip-map. So several finer levels of a clip-map have the same size in voxels, and have the same spatial resolution as finer levels of a mip-map. When such clip-map is centered around the camera, its finer levels of detail cover regions that are close to the camera, and coarser levels of detail cover larger areas that are further away. Due to perspective distortion, we don't really need high resolution data far away from the camera, which maps to clip-map geometry nicely.

Our clip-map is scalable: the size and number of clipped levels varies, as does voxel format. Various combinations of these parameters result in memory requirements ranging from megabytes to gigabytes of video memory, where the most detailed and expensive options produce higher quality diffuse and especially specular reflections.

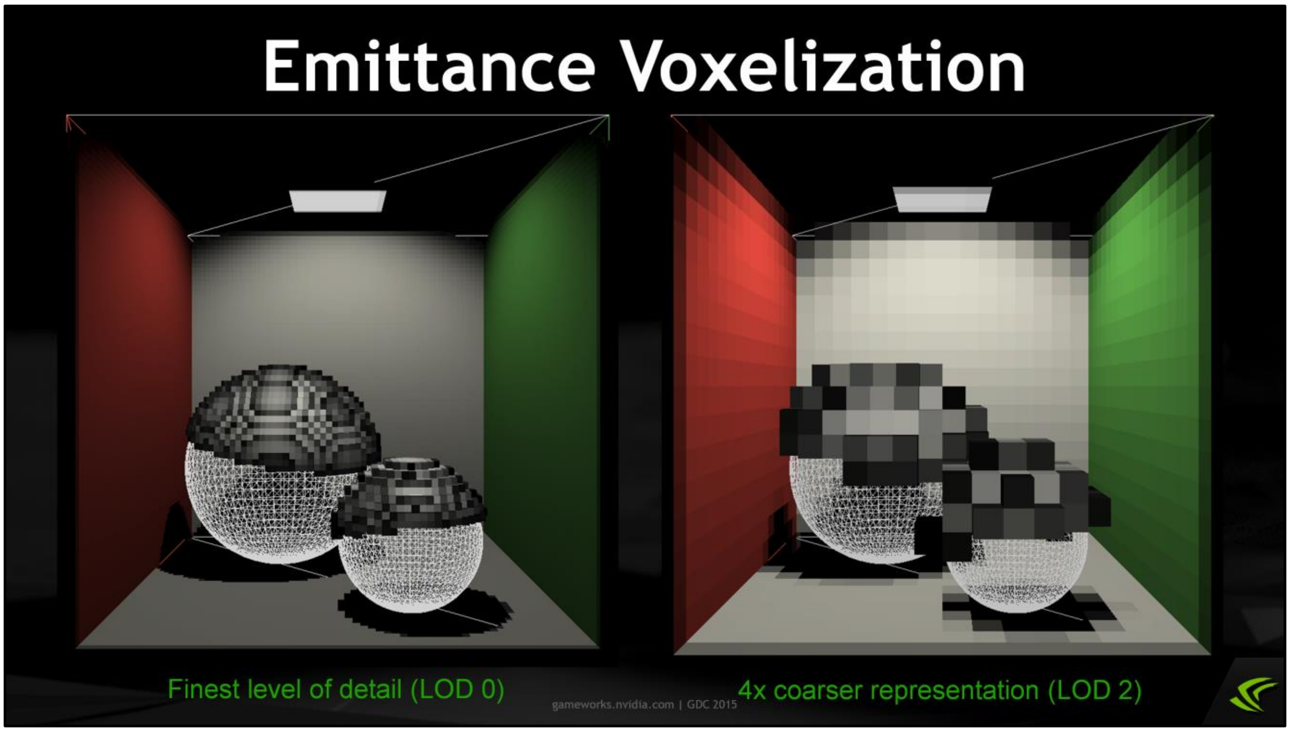


Opacity Voxelization

Finest level of detail (LOD 0)
4x coarser representation (LOD 2)
gameworks.nvidia.com | GDC 2015

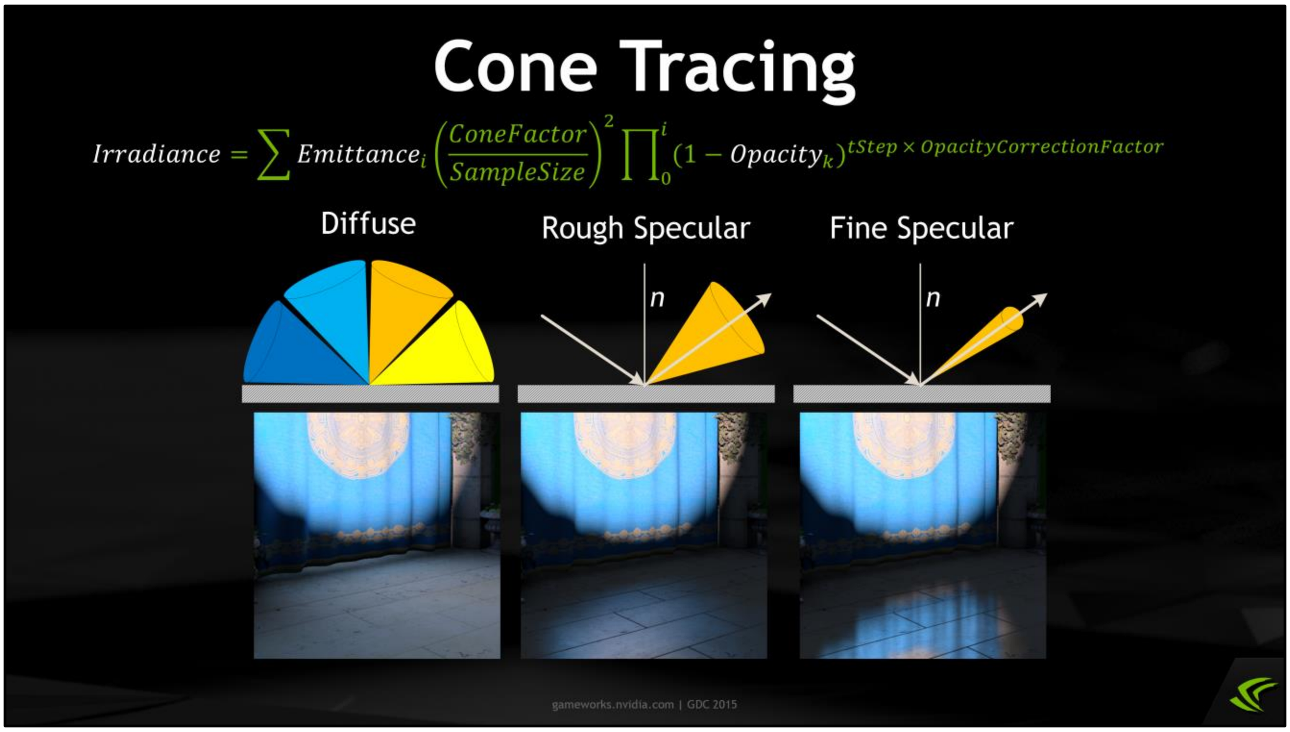
The first step of the VXGI algorithm is opacity voxelization and downsampling. It means that scene geometry is converted to a map which encodes approximate opacity of space, averaged over voxels. So, for example a cube whose size matches one voxel will produce a fully opaque voxel. But after downsampling, in the next coarser LOD, the opacity of the enclosing voxel will be only 25%. One more level up and it's only 6.25%, and so on.

To perform voxelization, the application renders the scene with special matrices, geometry and pixel shaders provided by VXGI, at a resolution equal to the clip-map size. The voxel map is updated by pixel shader threads using atomic operations on a UAV.


Emittance Voxelization
Finest level of detail (LOD 0)
4x coarser representation (LOD 2)
gameworks.nvidia.com | GDC 2015

The next algorithm step is emittance voxelization and downsampling. Emittance voxels basically store the amount of light that geometry in their volume reflects or emits into all directions. For example, if there is a straight wall going through a slab of voxels parallel to one of the major planes, the color of the voxels should match the diffuse color of the wall shaded in screen space. Downsampling does not affect the color of the wall as long as it doesn't become smaller than a voxel.

On these pictures, you can see that the colors on the right is different from the colors on the left. This is not a bug; this is a result of anti-aliased voxelization that distributes reflected color between two adjacent voxels, and you only see the closest voxel on the pictures.



Cone Tracing

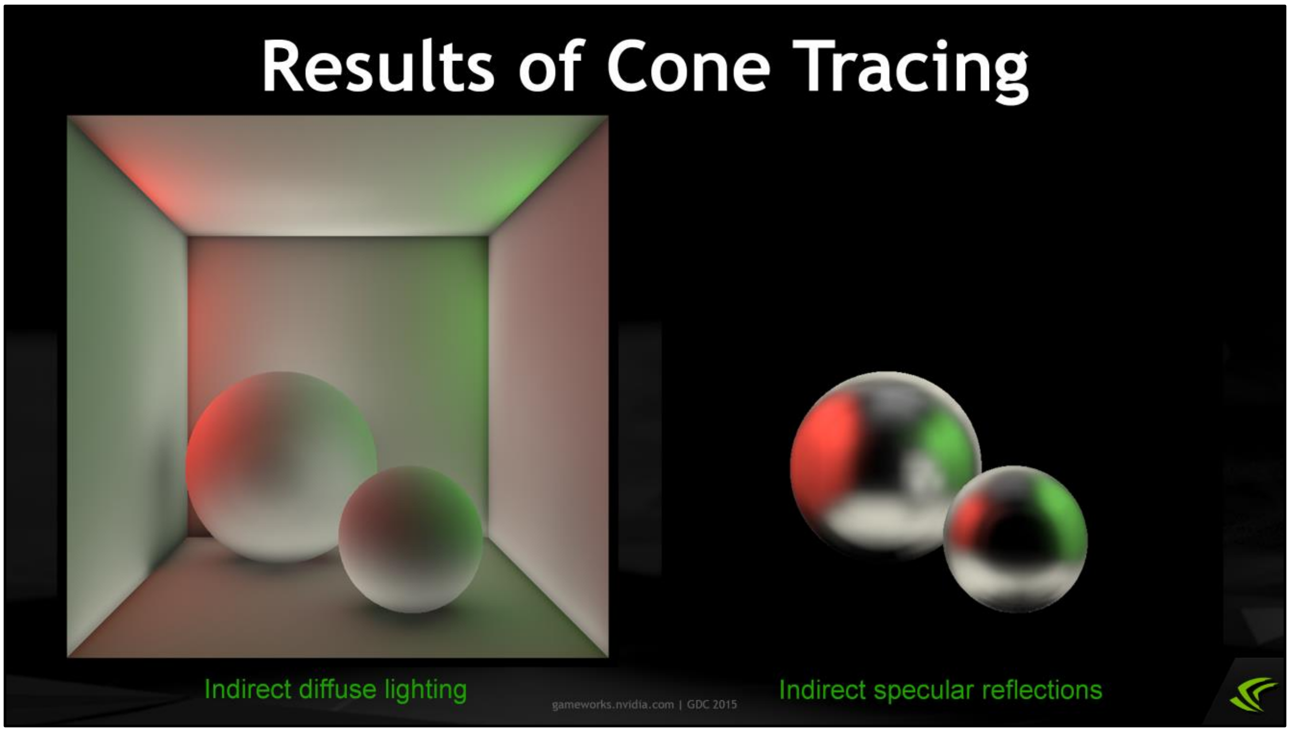
$$Irradiance = \sum Emittance_i \left(\frac{ConeFactor}{SampleSize}\right)^2 \prod_0^i (1 - Opacity_k)^{tStep \times OpacityCorrectionFactor}$$

The final step of the algorithm is cone tracing. Cone tracing is a process that computes approximate surface irradiance coming from a set of directions, given the primary direction and the cone angle. It is roughly described by the equation on the slide. It marches along the given direction, taking samples from the emittance texture that contribute to irradiance. In order to reduce light leaking, occlusion is also calculated as the product of opacities from all the previous samples.

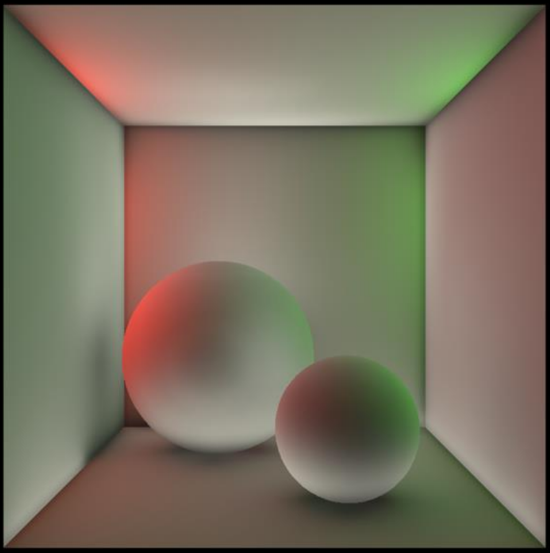
Cone tracing is basically a primitive, a function that lets you calculate irradiance. There are many possible uses for such function. We give you access to this function in your shaders and we also implement two primary use cases: computing diffuse and specular indirect illumination channels for a G-buffer.

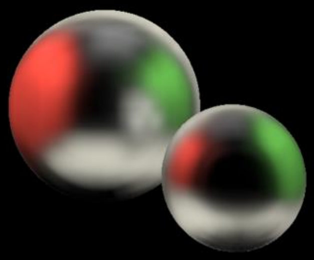
Diffuse tracing shoots several cones from every visible surface, into a set of directions covering the hemisphere.

Specular tracing shoots one cone from every visible specular surface in the reflected view direction. The angle of this cone depends on surface roughness: for rough surfaces, a wide cone is used; for finer surfaces, a narrow cone is used. There is a lower limit on surface roughness, however, because the spatial resolution of our voxel representation is insufficient to compute perfect mirror-like reflections.



Results of Cone Tracing

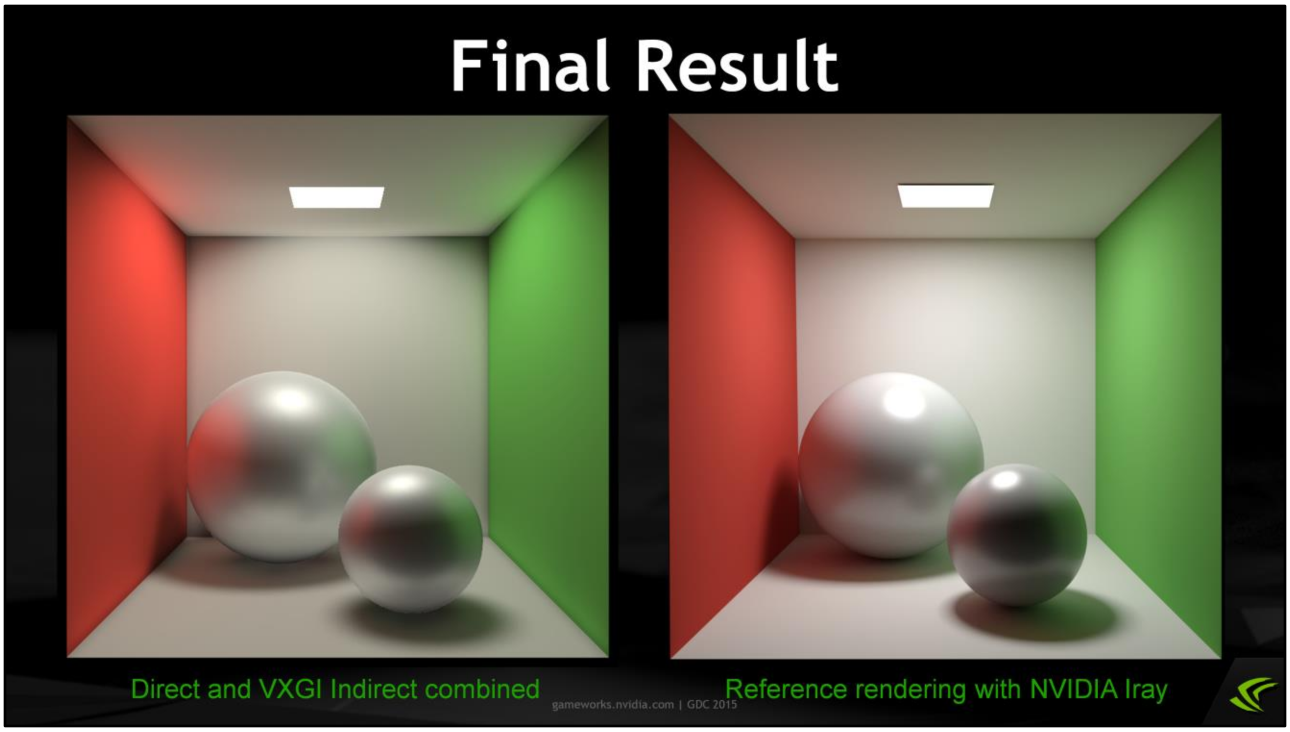




gameworks.nvidia.com | GDC 2015

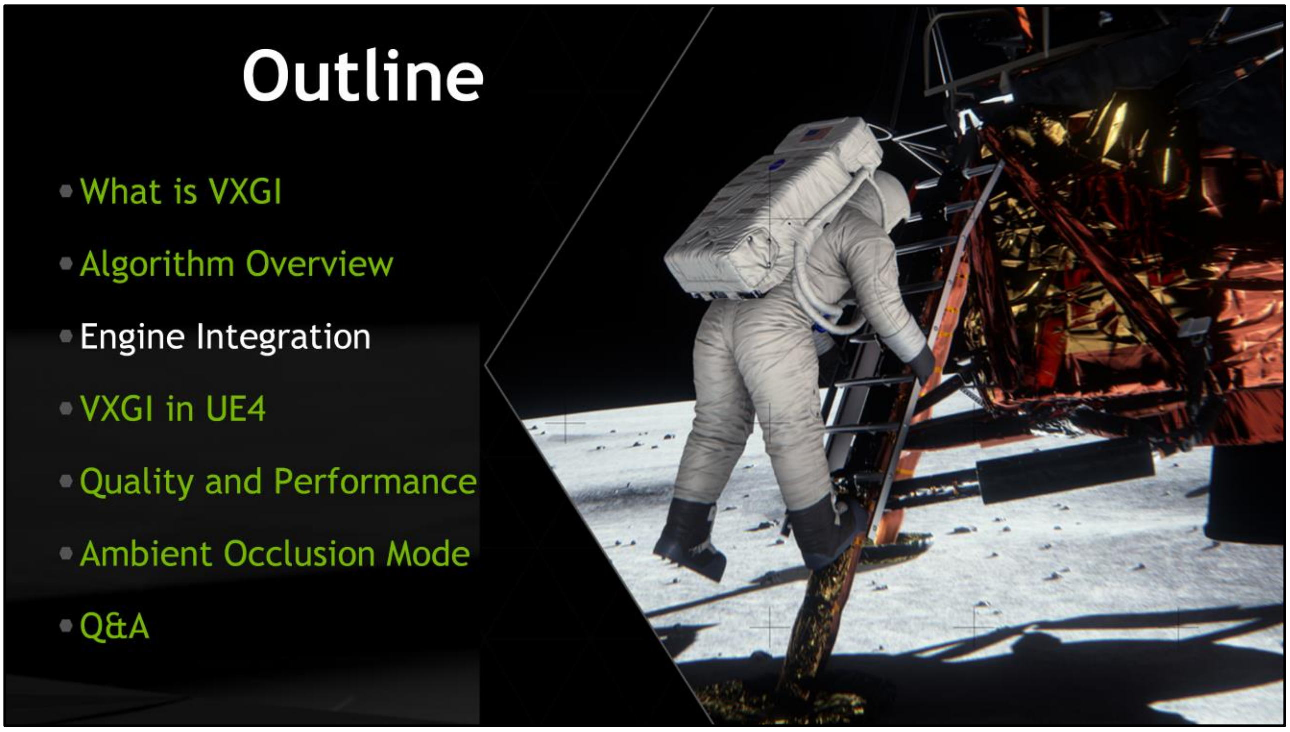
You can see the results of diffuse and specular cone tracing passes on these pictures. Indirect diffuse channel is on the left. There is a red tint on the left sphere, it comes from the red left wall; and there's a green tint on the right sphere, it comes from the green right wall. But the walls themselves are not red or green on this picture because it shows indirect irradiance only, not outgoing radiance, so their albedo does not affect the picture.

Indirect specular channel is on the right. Specular reflections are computed only for specular objects, which are determined by the roughness channel in the G-buffer. You can see a fuzzy reflection of the whole scene in each sphere.


Final Result
Direct and VXGI Indirect combined
Reference rendering with NVIDIA Iray
gameworks.nvidia.com | GDC 2015

Combining the indirect illumination channels with direct illumination and adding tone mapping and a light bloom produces the image on the left. On the right is the reference rendering of the same scene produced by Nvidia Iray, a GPU ray tracer. The images look pretty similar, although there are some minor differences. More importantly, the right image was rendered in about 10 minutes, while the left image is rendered at about 500 frames per second on a modern high-end GPU.

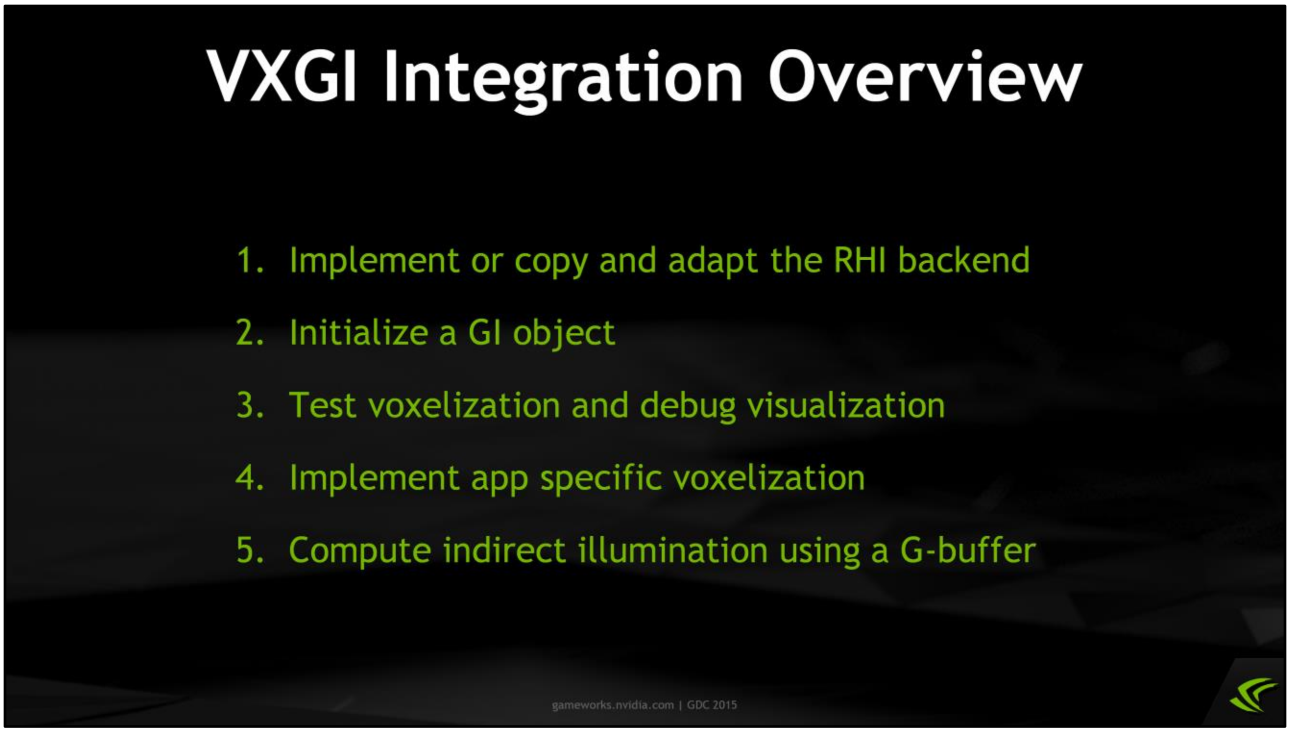
Demo: San Miguel



Outline

- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A

The next section of my talk is more practical, it describes engine integration, that is, what do you have to do in order to enable VXGI in your game or application.

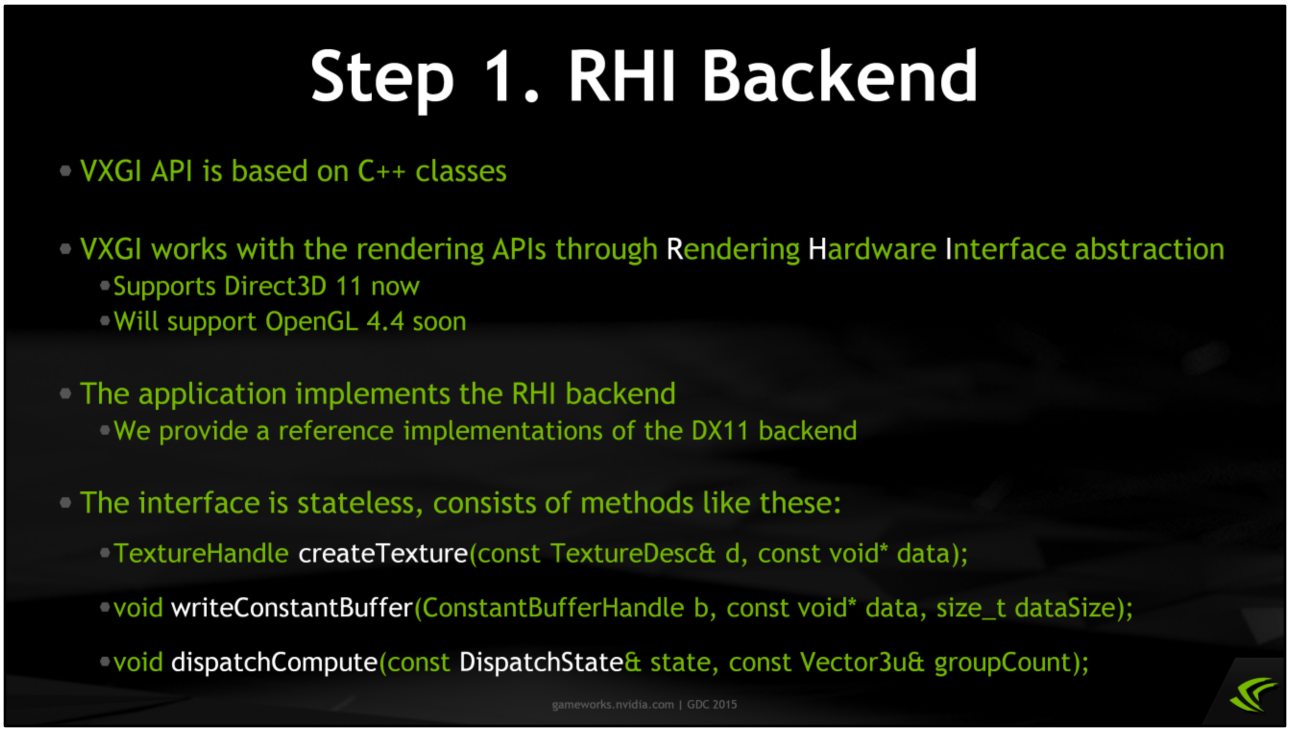


VXGI Integration Overview

1. Implement or copy and adapt the RHI backend
2. Initialize a GI object
3. Test voxelization and debug visualization
4. Implement app specific voxelization
5. Compute indirect illumination using a G-buffer

gameworks.nvidia.com | GDC 2015

The integration process consists of five major steps, and I'll shortly describe all of them in detail. First, you need to connect VXGI with the rendering API using a translation layer, RHI. Then you create a VXGI interface object and test that the rendering API works properly, the matrices are in the correct layout and so on using the built-in voxelization test scene. After that you can move on to implementing custom voxelization for your scene and computing the indirect illumination channels for the G-buffer.

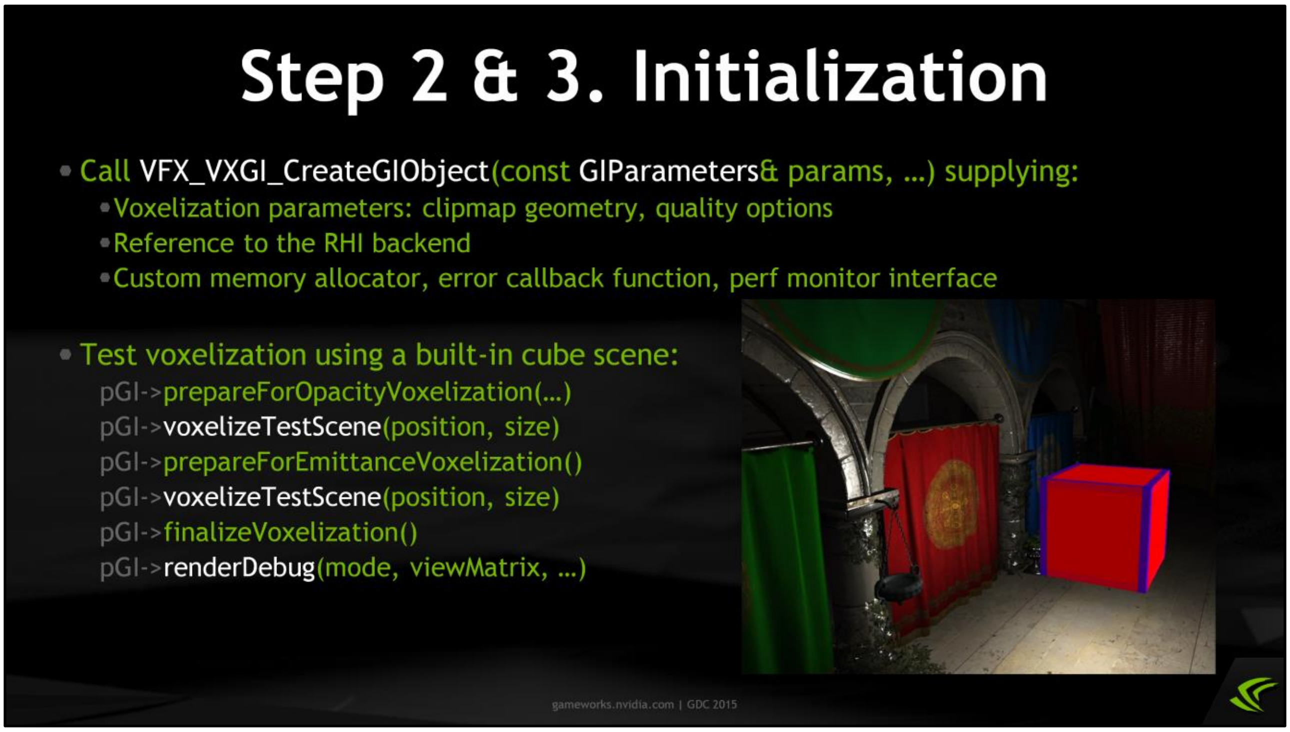


Step 1. RHI Backend

- VXGI API is based on C++ classes
- VXGI works with the rendering APIs through Rendering Hardware Interface abstraction
  - Supports Direct3D 11 now
  - Will support OpenGL 4.4 soon
- The application implements the RHI backend
  - We provide a reference implementations of the DX11 backend
- The interface is stateless, consists of methods like these:
  - TextureHandle createTexture(const TextureDesc& d, const void* data);
  - void writeConstantBuffer(ConstantBufferHandle b, const void* data, size_t dataSize);
  - void dispatchCompute(const DispatchState& state, const Vector3u& groupCount);

The first step, as mentioned before, is connecting VXGI with the rendering API such as Direct3D 11. Actually, that is the only API that is supported by VXGI right now, but we're working on OpenGL 4.4 support as well. So, VXGI uses an abstraction layer over the rendering API, and your application has to implement that layer. We provide a reference implementation of D3D11 RHI for simple engines with our samples, and a different implementation inside the UE4 integration. You can take one of those and use it as is, or you can modify it to fit your needs – it's provided in source code form.

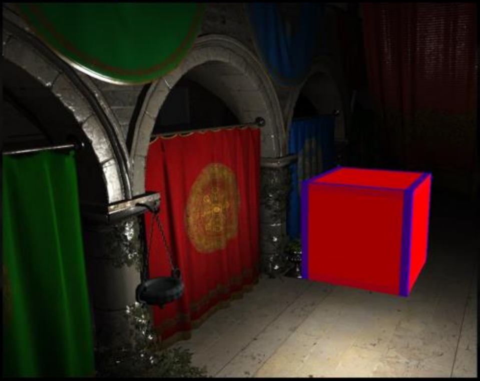
The RHI is a relatively simple interface, and one of its important properties it that it's stateless. VXGI always provides a structure describing the full state, including shaders, resource bindings and so on, necessary for completing each draw or dispatch call.



Step 2 & 3. Initialization

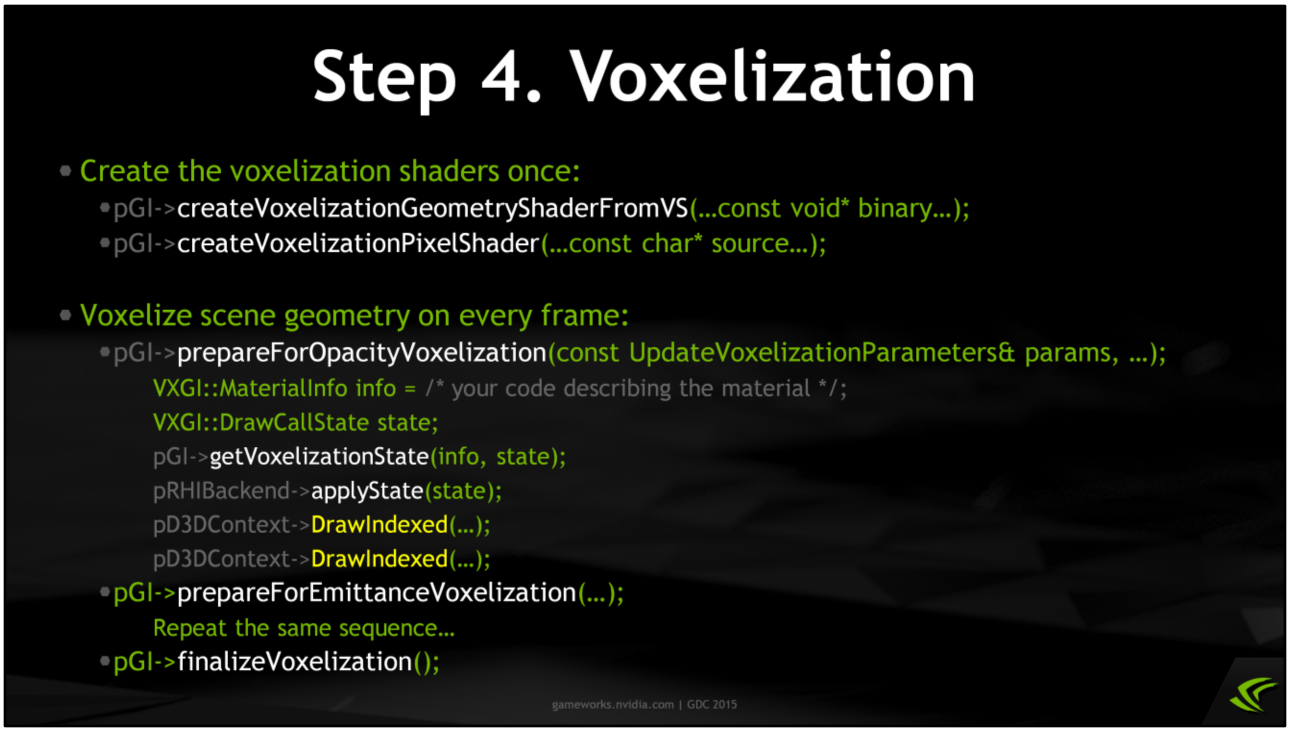
- Call VFX_VXGI_CreateGIObject(const GIParameters& params, ...) supplying:
    - Voxelization parameters: clipmap geometry, quality options
    - Reference to the RHI backend
    - Custom memory allocator, error callback function, perf monitor interface
- Test voxelization using a built-in cube scene:

```
pGI->prepareForOpacityVoxelization(...)
pGI->voxelizeTestScene(position, size)
pGI->prepareForEmittanceVoxelization()
pGI->voxelizeTestScene(position, size)
pGI->finalizeVoxelization()
pGI->renderDebug(mode, viewMatrix, ...)
```



When you have an RHI implementation, you can create the VXGI interface object using one of the few functions exported by its DLL. Along with the backend, the function takes voxelization parameters, such as clip-map resolution and voxel storage format, and some other minor things.

You can test whether the backend works properly by voxelizing a built-in test scene. The call sequence for that is presented on the slide, and it should produce an opaque cube emitting white light, with position and size specified by the caller. The final call, renderDebug, renders a ray-marched visualization of the voxel textures, similar to those screenshots from Cornell Box earlier in this presentation. If you also render the G-buffer and perform diffuse and specular cone tracing using this cube, you should see an image similar to one presented on the slide.



Step 4. Voxelization

- Create the voxelization shaders once:
  - `pGI->createVoxelizationGeometryShaderFromVS(…const void* binary…);`
  - `pGI->createVoxelizationPixelShader(…const char* source…);`
- Voxelize scene geometry on every frame:
  - `pGI->prepareForOpacityVoxelization(const UpdateVoxelizationParameters& params, …);`

```
VXGI::MaterialInfo info = /* your code describing the material */;
VXGI::DrawCallState state;
pGI->getVoxelizationState(info, state);
pRHIBackend->applyState(state);
pD3DContext->DrawIndexed(…);
pD3DContext->DrawIndexed(…);
```

  - `pGI->prepareForEmittanceVoxelization(…);`

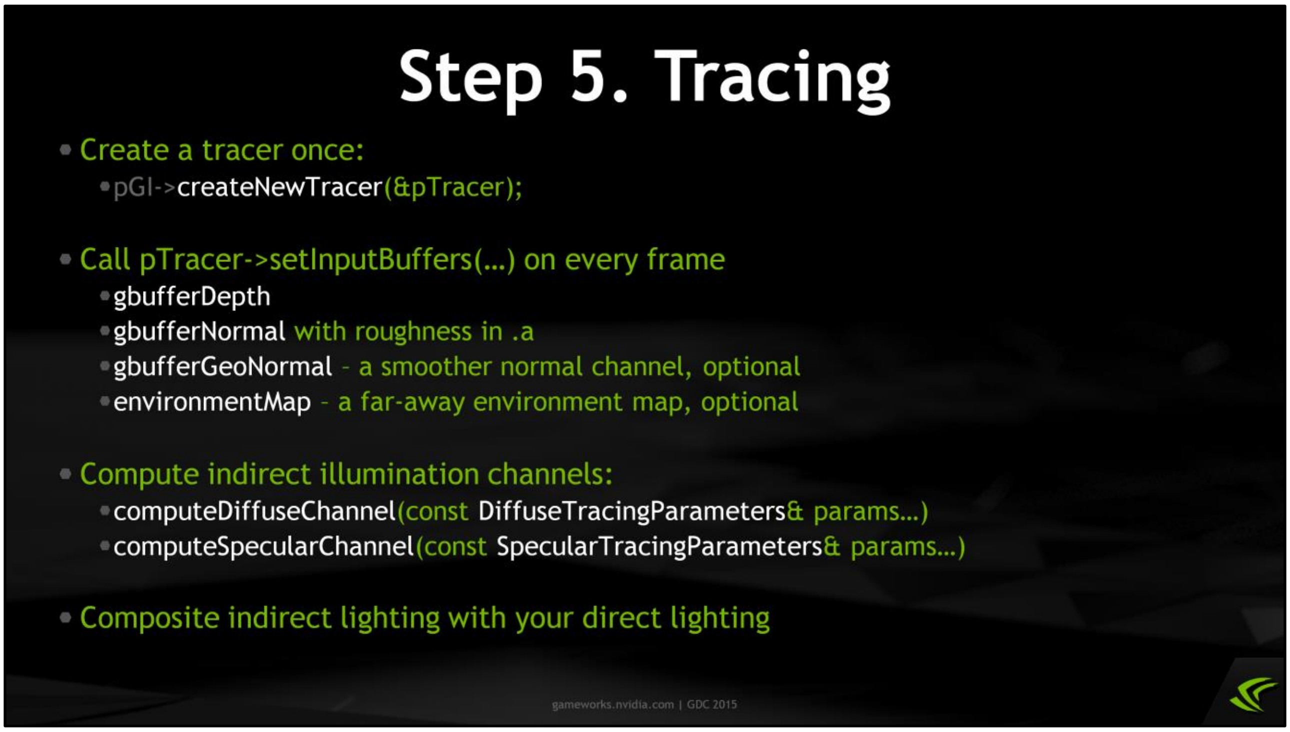
    Repeat the same sequence…
  - `pGI->finalizeVoxelization();`

gameworks.nvidia.com | GDC 2015

The next step in VXGI integration is implementing custom voxelization. To voxelize geometry, you need to create at least two VXGI shader sets: one geometry shader and one pixel shader. Geometry shaders are generated by VXGI from a list of attributes that need to be passed from the vertex shader to the pixel shader. Pixel shaders are combined from your source code that computes material parameters and our code that updates the voxel texture, but for simplicity you can use a built-in default pixel shader that assumes that all materials are opaque and emit white color – just like the test cube.

After creating the voxelization shaders once, you need to voxelize your scene on every frame. It’s not strictly necessary to do it on every frame unless some objects change or the clip-map is moved, but normally some amount of voxelization does happen on every frame.

So, to voxelize the scene, you need to switch VXGI into the right state, then fill out the VXGI material information structure which includes references to voxelization shaders, and convert that structure to drawing state. Apply that state, add your vertex shader and other resources, for example material textures needed for the pixel shader, and draw your geometry. Then switch VXGI into the emittance voxelization state and repeat the voxelization process – you can draw a different set of objects for that, or use different materials.



Step 5. Tracing

- Create a tracer once:
  - pGI->createNewTracer(&pTracer);
- Call pTracer->setInputBuffers(...) on every frame
  - gbufferDepth
  - gbufferNormal with roughness in .a
  - gbufferGeoNormal - a smoother normal channel, optional
  - environmentMap - a far-away environment map, optional
- Compute indirect illumination channels:
  - computeDiffuseChannel(const DiffuseTracingParameters& params...)
  - computeSpecularChannel(const SpecularTracingParameters& params...)
- Composite indirect lighting with your direct lighting

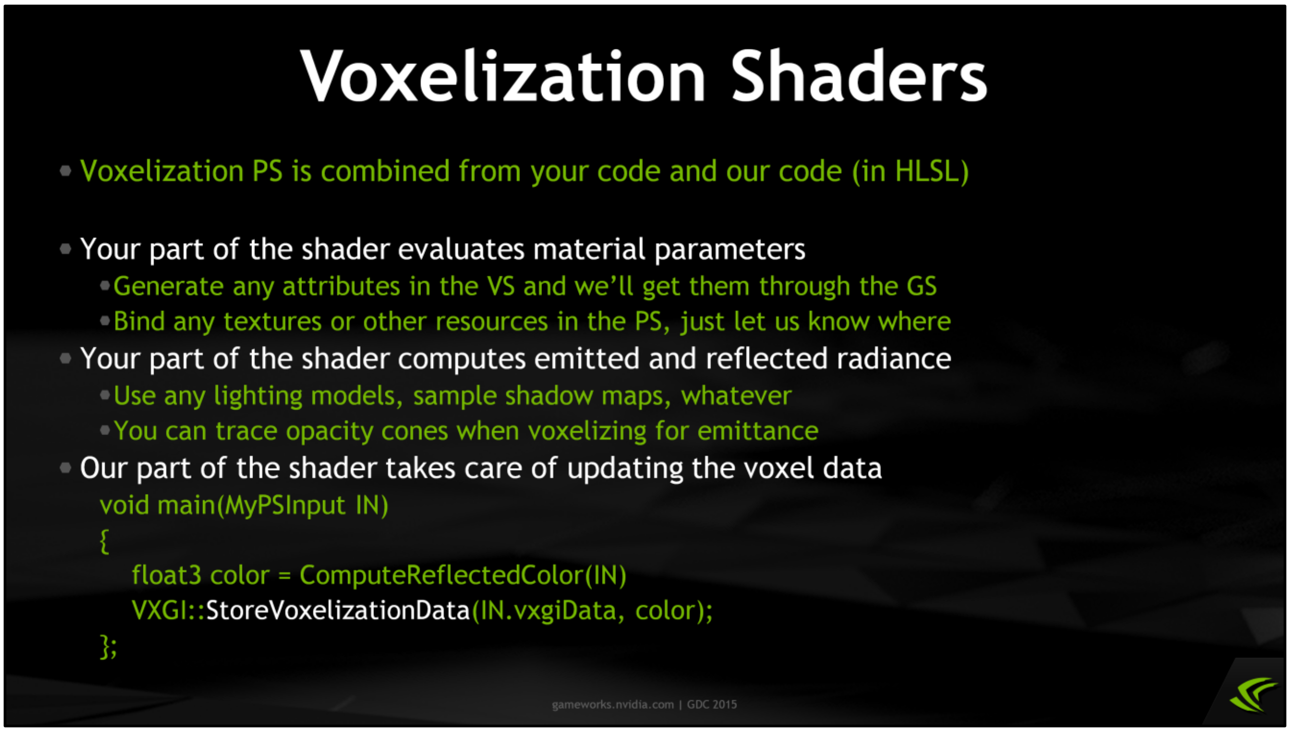
gameworks.nvidia.com | GDC 2015

When voxelization is finished, VXGI is ready to compute indirect lighting. It is computed by objects called tracers, and you can have as many of them as necessary, one per view. For example, you can have two tracers for stereo rendering. Tracers implement temporal filtering, which uses information from previous frames, so don't create new tracers on every frame.

To use a tracer, you need to provide it with G-buffer channels, including depth and normal, and the view and projection matrices. Optionally you can use two more channels, one for smoother normals that use materials without normal maps; another for a far-away environment map that can be used for specular reflections if the traced cones do not hit anything.

Computing indirect illumination channels is easy, just call one of these methods – computeDiffuseChannel or computeSpecularChannel. These methods accept structures that contain a lot of tuning parameters that affect quality and performance. These parameters are documented in the VXGI header file.

Compose the indirect channels with your direct lighting, add post-processing effects and that's it!



Voxelization Shaders

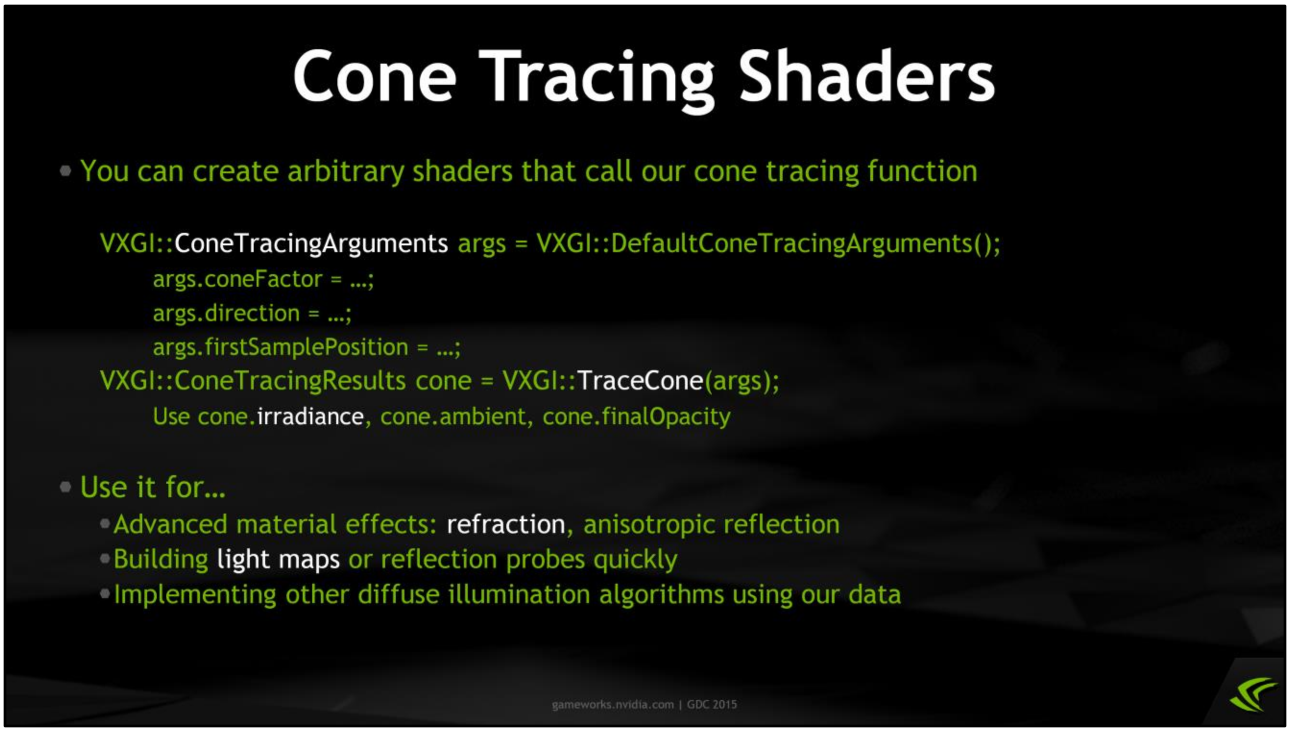
- Voxelization PS is combined from your code and our code (in HLSL)

- Your part of the shader evaluates material parameters
  - Generate any attributes in the VS and we'll get them through the GS
  - Bind any textures or other resources in the PS, just let us know where
- Your part of the shader computes emitted and reflected radiance
  - Use any lighting models, sample shadow maps, whatever
  - You can trace opacity cones when voxelizing for emittance
- Our part of the shader takes care of updating the voxel data

```
void main(MyPSInput IN)
{
    float3 color = ComputeReflectedColor(IN)
    VXGI::StoreVoxelizationData(IN.vxgiData, color);
};
```

gameworks.nvidia.com | GDC 2015

As I mentioned before, voxelization pixel shaders can be generated by combining VXGI code that updates the voxel structure and your code that shades materials. More specifically, your vertex shader generates any attributes required for the materials, and our geometry shader passes them through to the pixel shader. The pixel shader main function with all its inputs and resources is written by you. Use the resources, compute the reflected plus emitted color just like you would for forward screen-space shading, and then instead of outputting the result into SV_Target, call StoreVoxelizationData function, defined externally, to let our code write the result to the voxel clip-map.



Cone Tracing Shaders

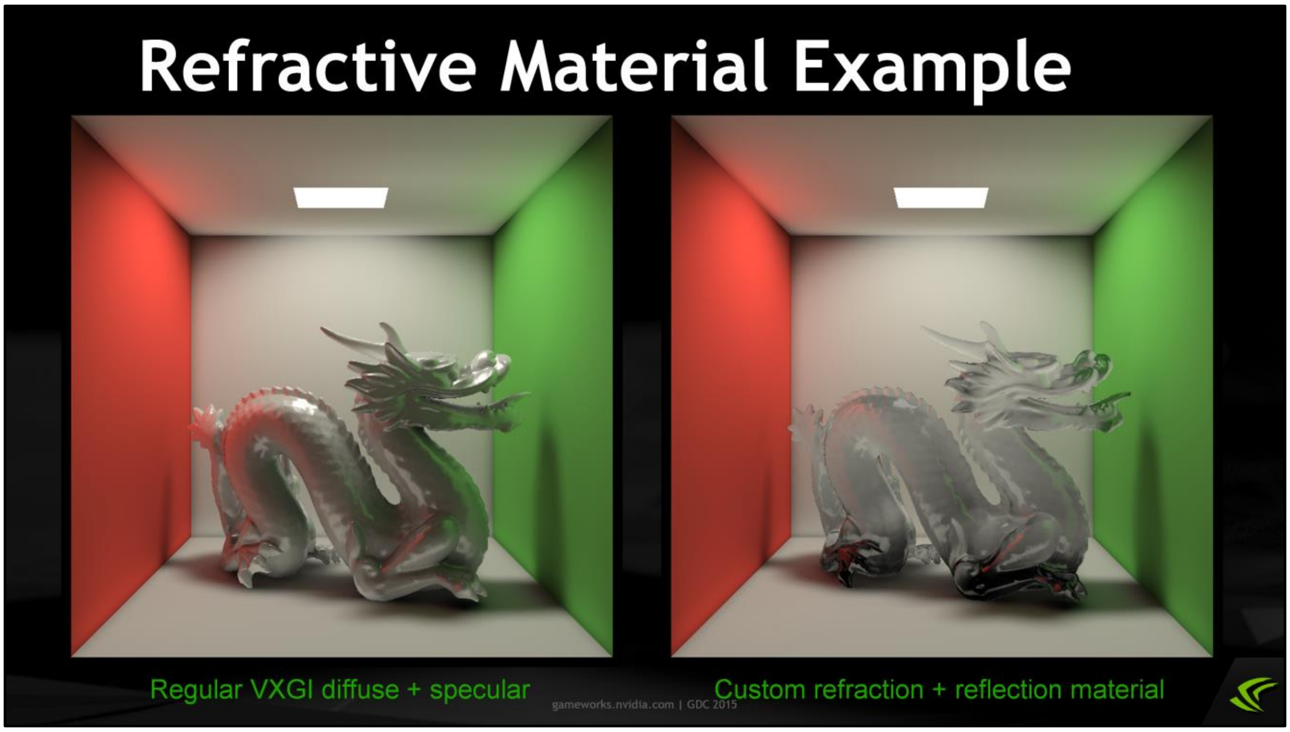
- You can create arbitrary shaders that call our cone tracing function

```
VXGI::ConeTracingArguments args = VXGI::DefaultConeTracingArguments();
    args.coneFactor = …;
    args.direction = …;
    args.firstSamplePosition = …;
VXGI::ConeTracingResults cone = VXGI::TraceCone(args);
    Use cone.irradiance, cone.ambient, cone.finalOpacity
```

- Use it for…
  - Advanced material effects: refraction, anisotropic reflection
  - Building light maps or reflection probes quickly
  - Implementing other diffuse illumination algorithms using our data

Another important type of shader that you can create with VXGI is one that uses the cone tracing primitive. It can be a pixel or compute shader, and it can call the TraceCone function to compute irradiance coming to any point from any direction, any number of times.

There are many possible uses of the cone tracing function, such as advanced world-space material effects like refraction, building dynamic light maps or reflection probes in real time which may be more efficient than doing G-buffer based tracing.



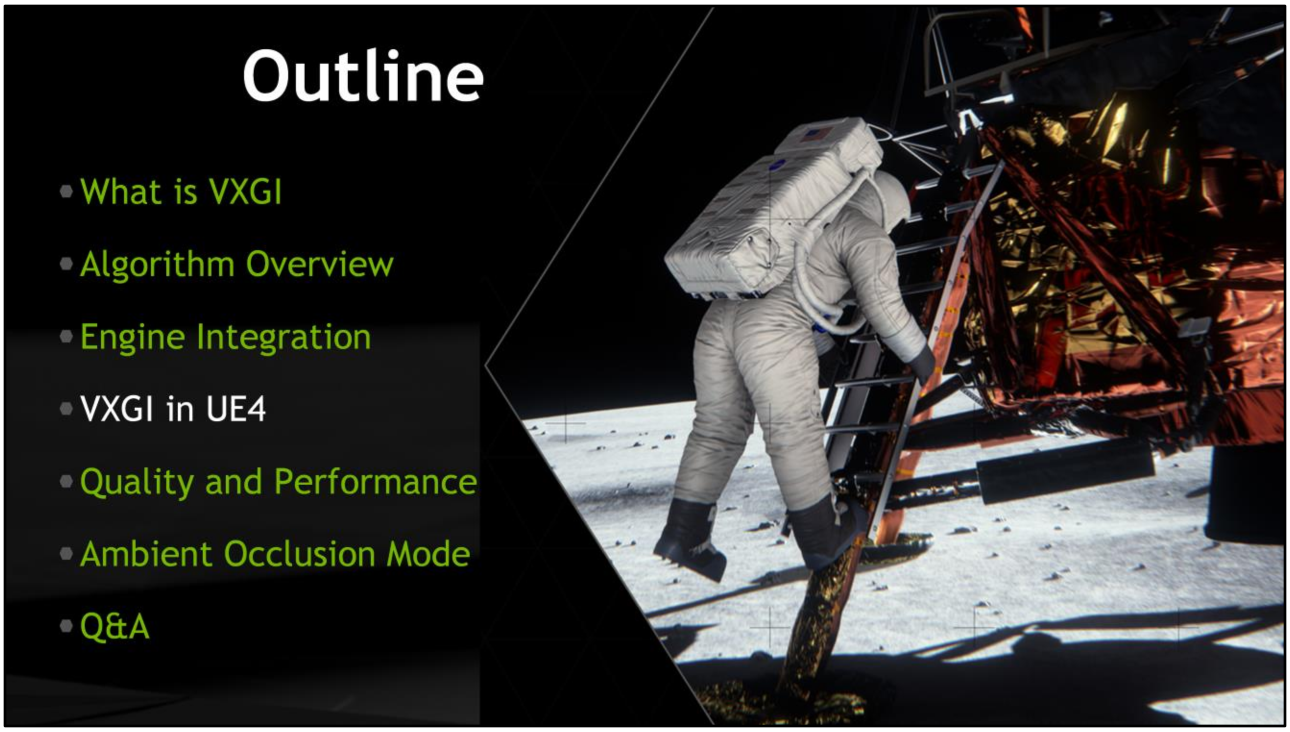
Refractive Material Example

Regular VXGI diffuse + specular

Custom refraction + reflection material

gameworks.nvidia.com | GDC 2015

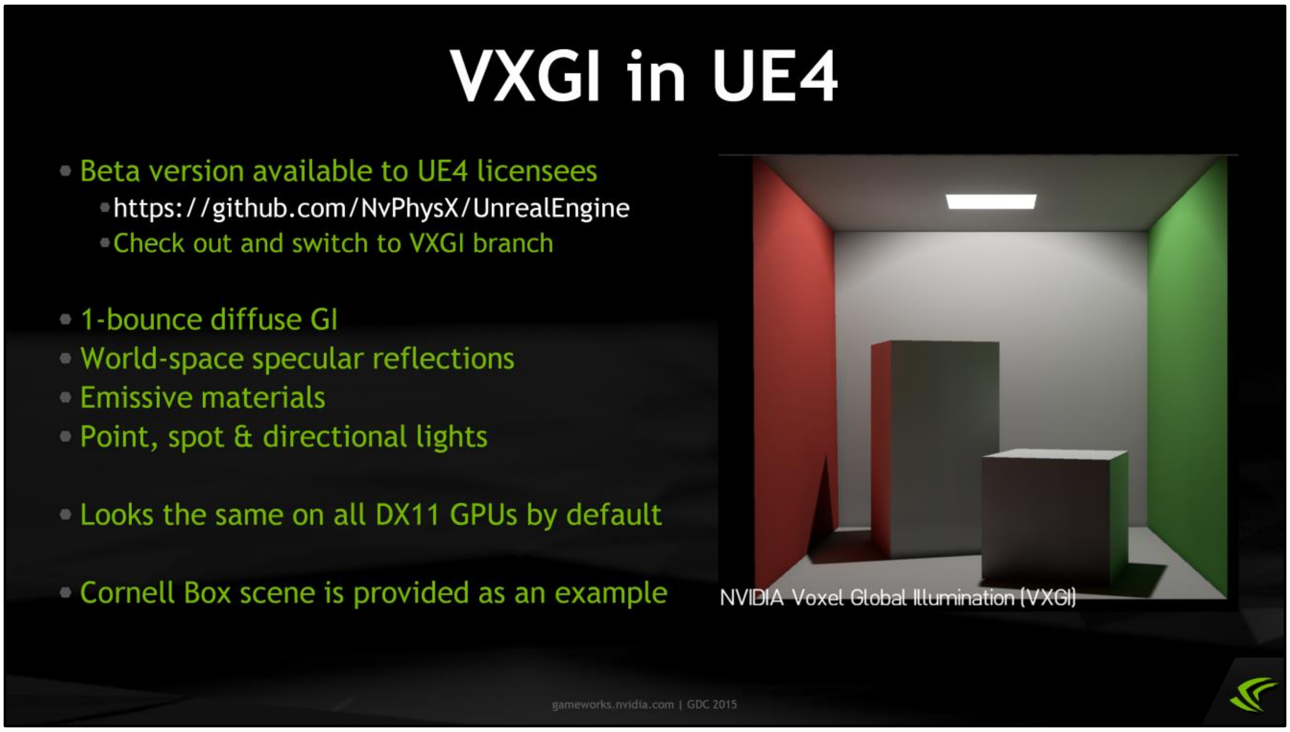
Here is an example of such material using a custom cone tracing shader. On the left you can see a dragon model rendered with standard VXGI diffuse and specular tracing. On the right is the same dragon rendered with a custom reflection and refraction material. It looks really good and works fast, although there is the same problem as with specular reflections: we do not have enough voxel resolution to produce perfect mirror reflections or perfect refractions, only fuzzy ones are possible.



Outline

- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A

The next section of my talk is about the integration of VXGI into Unreal Engine 4.



VXGI in UE4

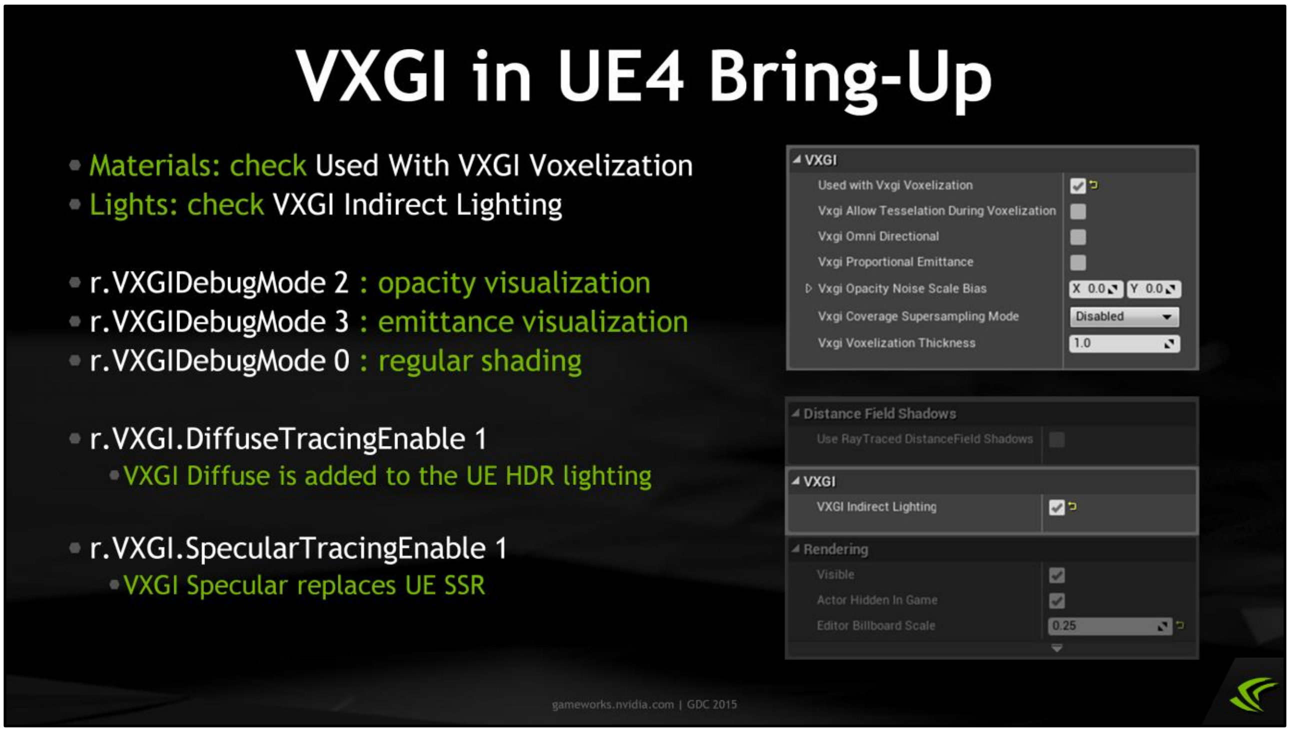
- Beta version available to UE4 licensees
  - https://github.com/NvPhysX/UnrealEngine
  - Check out and switch to VXGI branch
- 1-bounce diffuse GI
- World-space specular reflections
- Emissive materials
- Point, spot & directional lights
- Looks the same on all DX11 GPUs by default
- Cornell Box scene is provided as an example

NVIDIA Voxel Global Illumination (VXGI)

gameworks.nvidia.com | GDC 2015

For your convenience, we have integrated VXGI into Unreal Engine 4, and it is already available for free on Github to everyone who has their Github account associated with their Unreal Engine account. Just check out the repository from the URL on the slide, switch to the VXGI branch and build it.

It supports almost all features VXGI itself does, such as indirect diffuse and specular illumination in a fully dynamic environment, all kinds of lights and emissive materials. Certain features such as temporal filtering of cone tracing results have not been integrated yet, but we're working on it.



VXGI in UE4 Bring-Up

- Materials: check Used With VXGI Voxelization
- Lights: check VXGI Indirect Lighting

- r.VXGIDebugMode 2 : opacity visualization
- r.VXGIDebugMode 3 : emittance visualization
- r.VXGIDebugMode 0 : regular shading

- r.VXGI.DiffuseTracingEnable 1
  - VXGI Diffuse is added to the UE HDR lighting

- r.VXGI.SpecularTracingEnable 1
  - VXGI Specular replaces UE SSR

VXGI

Used with Vxgi Voxelization	✓
Vxgi Allow Tesselation During Voxelization	
Vxgi Omni Directional	
Vxgi Proportional Emittance	
Vxgi Opacity Noise Scale Bias	X 0.0 Y 0.0
Vxgi Coverage Supersampling Mode	Disabled
Vxgi Voxelization Thickness	1.0

Distance Field Shadows

Use RayTraced DistanceField Shadows	

VXGI

VXGI Indirect Lighting	✓

Rendering

Visible	✓
Actor Hidden In Game	✓
Editor Billboard Scale	0.25

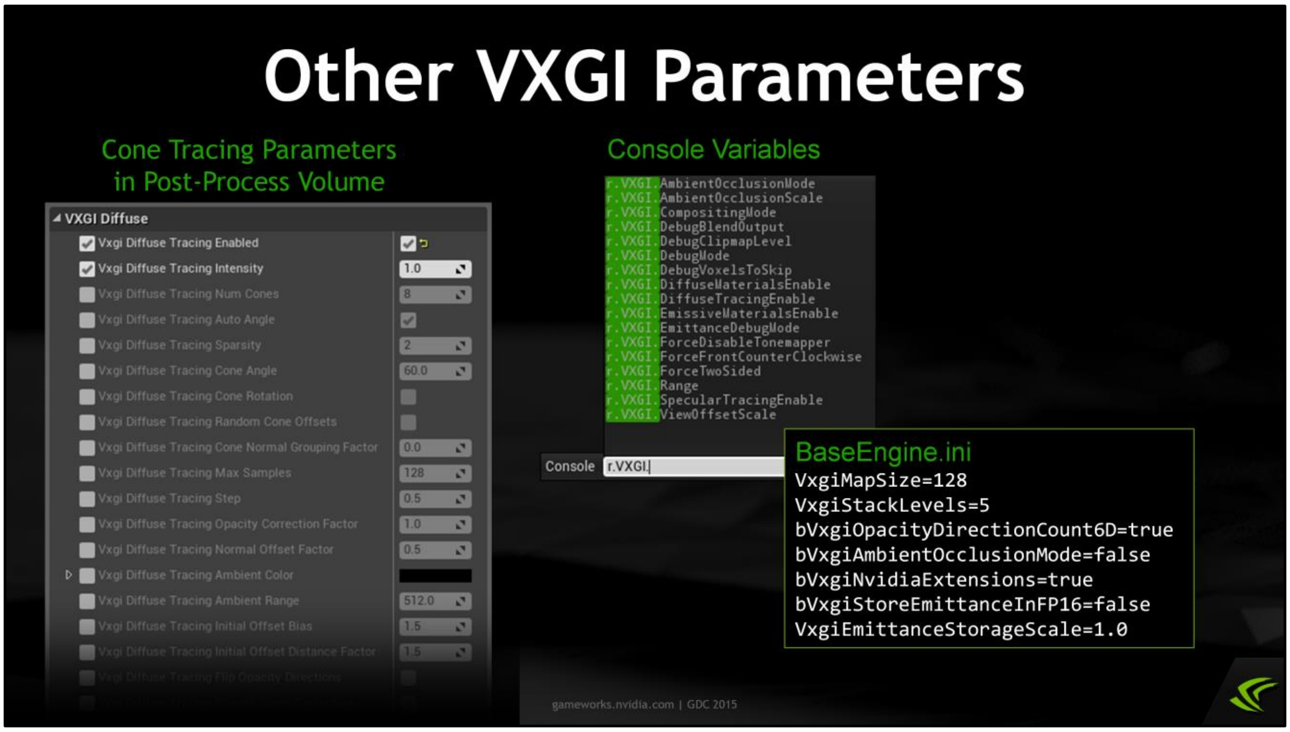
gameworks.nvidia.com | GDC 2015

We provide the Cornell Box scene that is already set up with VXGI as an example. To enable VXGI in your maps, follow these steps.

First, you need to make sure that solid objects are voxelized by checking the "Used with VXGI voxelization" box in all of their materials.

Second, you need to enable voxelization of the lights that you want to contribute to indirect illumination – and there is a check box "VXGI indirect lighting" in light parameters for that.

Then you can use one of the debug modes to see whether voxelization works properly, switch back to normal rendering and enable VXGI diffuse and specular using these console variables. And that's it – you should be getting indirect illumination at this point.



Other VXGI Parameters

Cone Tracing Parameters in Post-Process Volume

VXGI Diffuse	
✓ Vxgi Diffuse Tracing Enabled	✓
✓ Vxgi Diffuse Tracing Intensity	1.0
Vxgi Diffuse Tracing Num Cones	8
Vxgi Diffuse Tracing Auto Angle	✓
Vxgi Diffuse Tracing Sparsity	2
Vxgi Diffuse Tracing Cone Angle	60.0
Vxgi Diffuse Tracing Cone Rotation	
Vxgi Diffuse Tracing Random Cone Offsets	
Vxgi Diffuse Tracing Cone Normal Grouping Factor	0.0
Vxgi Diffuse Tracing Max Samples	128
Vxgi Diffuse Tracing Step	0.5
Vxgi Diffuse Tracing Opacity Correction Factor	1.0
Vxgi Diffuse Tracing Normal Offset Factor	0.5
Vxgi Diffuse Tracing Ambient Color	■
Vxgi Diffuse Tracing Ambient Range	512.0
Vxgi Diffuse Tracing Initial Offset Bias	1.5
Vxgi Diffuse Tracing Initial Offset Distance Factor	1.5
Vxgi Diffuse Tracing Flip Opacity Directions	

Console Variables

```
r.VXGI.AmbientOcclusionMode
r.VXGI.AmbientOcclusionScale
r.VXGI.CompositingMode
r.VXGI.DebugBlendOutput
r.VXGI.DebugClipmapLevel
r.VXGI.DebugMode
r.VXGI.DebugVoxelsToSkip
r.VXGI.DiffuseMaterialsEnable
r.VXGI.DiffuseTracingEnable
r.VXGI.EmissiveMaterialsEnable
r.VXGI.EmittanceDebugMode
r.VXGI.ForceDisableTonemapper
r.VXGI.ForceFrontCounterClockwise
r.VXGI.ForceTwoSided
r.VXGI.Range
r.VXGI.SpecularTracingEnable
r.VXGI.ViewOffsetScale
```

Console `r.VXGI.`

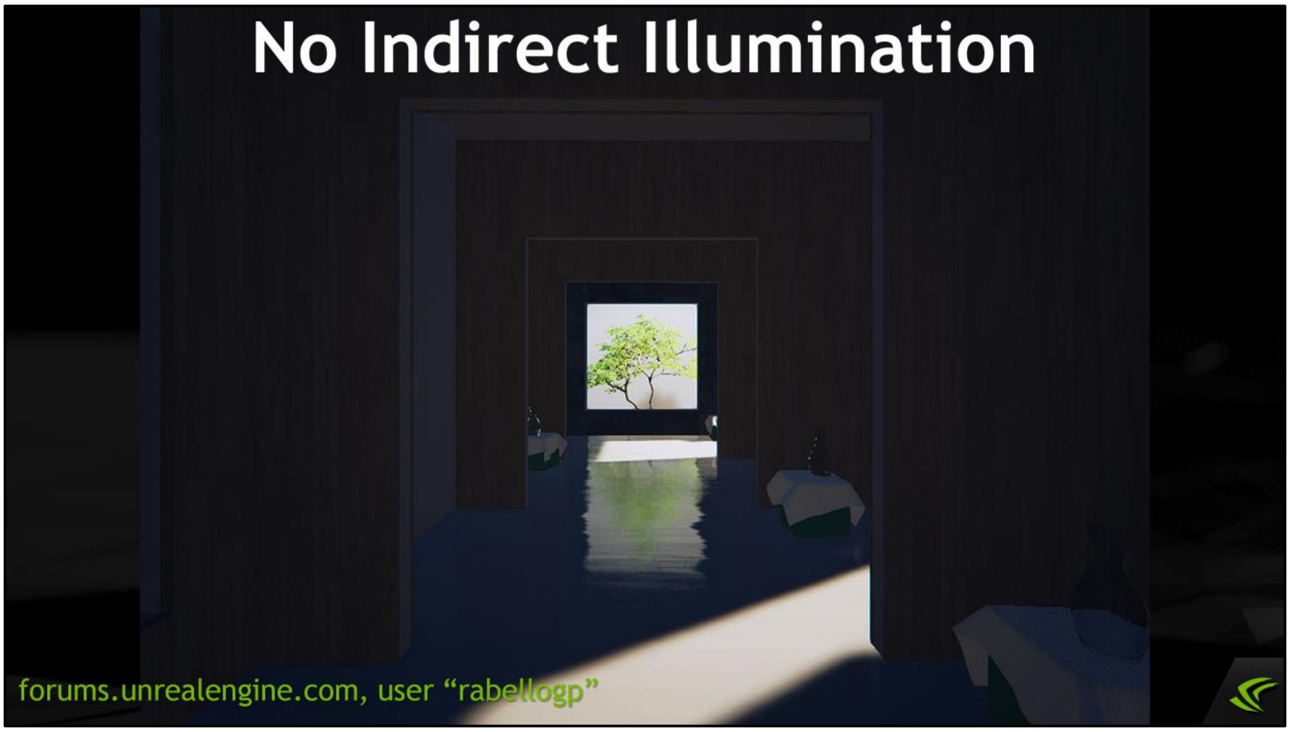
BaseEngine.ini

```
VxgiMapSize=128
VxgiStackLevels=5
bVxgiOpacityDirectionCount6D=true
bVxgiAmbientOcclusionMode=false
bVxgiNvidiaExtensions=true
bVxgiStoreEmittanceInFP16=false
VxgiEmittanceStorageScale=1.0
```

There are many other parameters that control VXGI in Unreal Engine, including: a lot of diffuse and specular tracing parameters in the post-process volume; some console variables that control debug visualizations, mostly; and several global parameters in the BaseEngine.ini file that control voxelization quality.

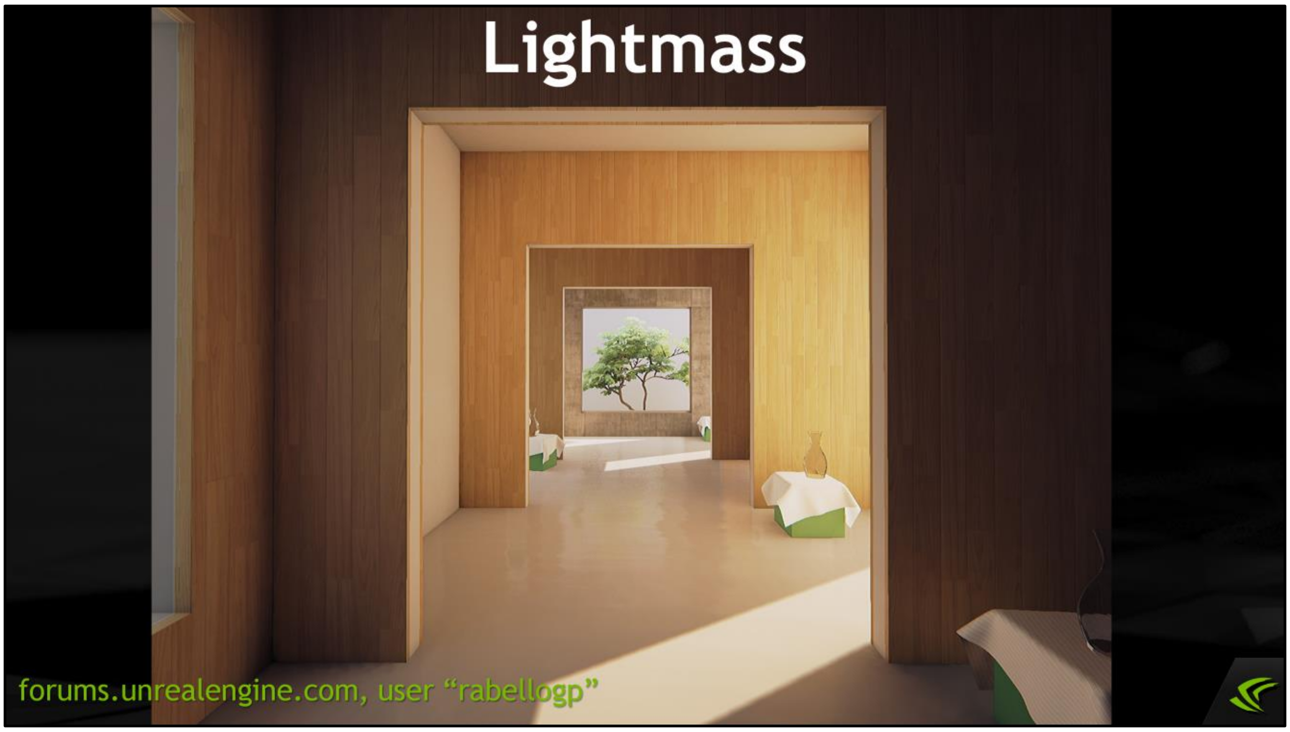
Demo: UE4 Editor

No Indirect Illumination



Since the release of VXGI within Unreal Engine about a month ago, a number of users started experimenting with it and have generated some good-looking content, which I would like to share with you.

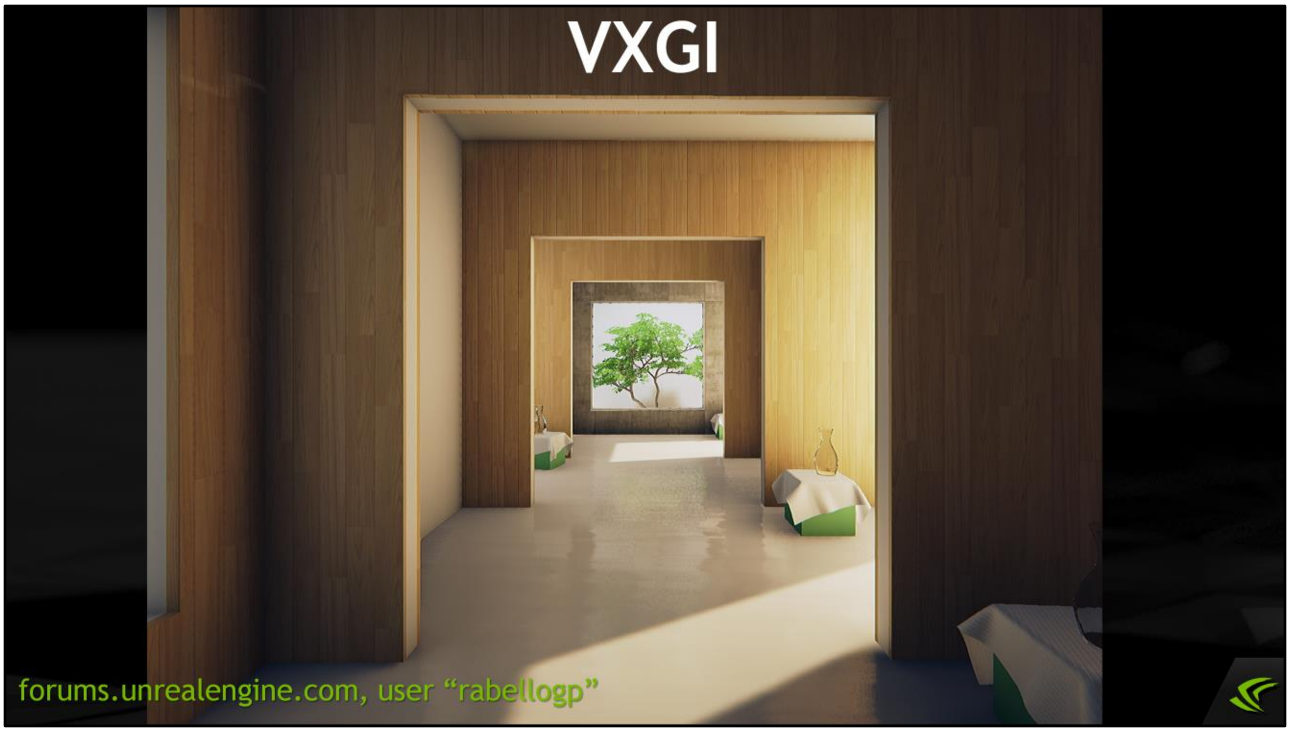
Here is an interior scene with no indirect illumination, just direct lighting and some ambient.



Lightmass

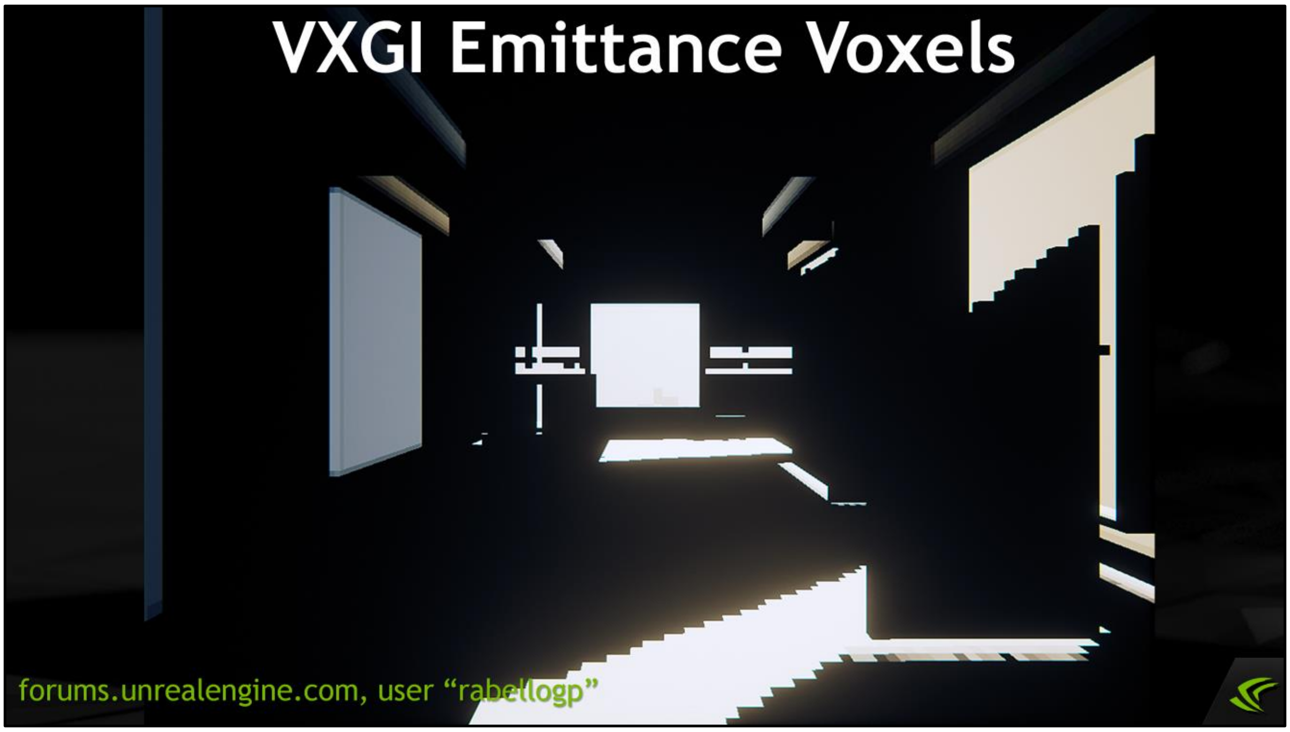
forums.unrealengine.com, user “rabellogp”

This is the same scene with light maps generated by Lightmass – it looks much better.



And the same scene with VXGI instead of Lightmass – the result is very similar, and in some places it looks even better, but computed in real time.

VXGI Emittance Voxels



This is the emittance voxel visualization for that scene, and it includes some directly lit voxels on the floor and emissive boxes for windows simulating lighting from objects outside – this is a useful technique called portal lights, and it works well with VXGI.

Elemental With VXGI

forums.unrealengine.com, user “ryanjon2040”

This is the Elemental demo with VXGI.

Effects Cave

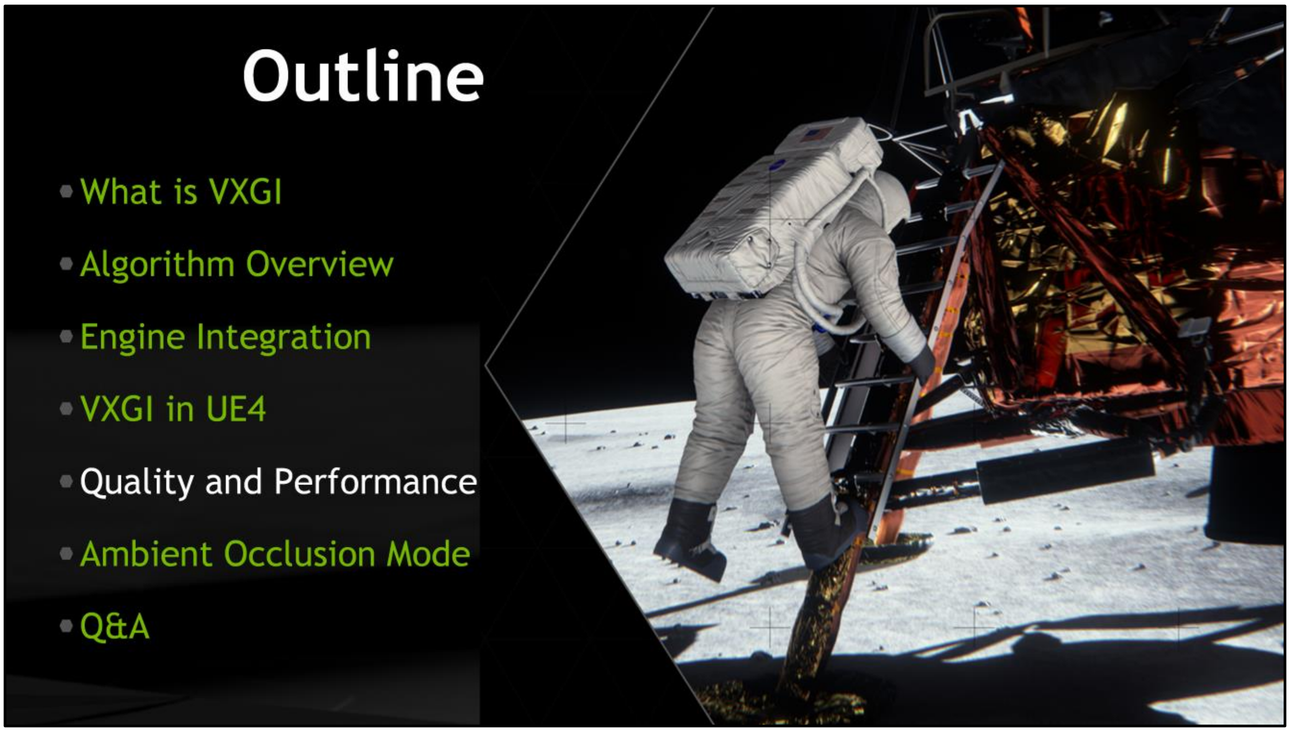
forums.unrealengine.com, user “Ad3ViLl”

And this is the Effects Cave demo with regular lighting – it looks good already, but we can add something to it…

Effects Cave With VXGI

forums.unrealengine.com, user “Ad3ViLl”

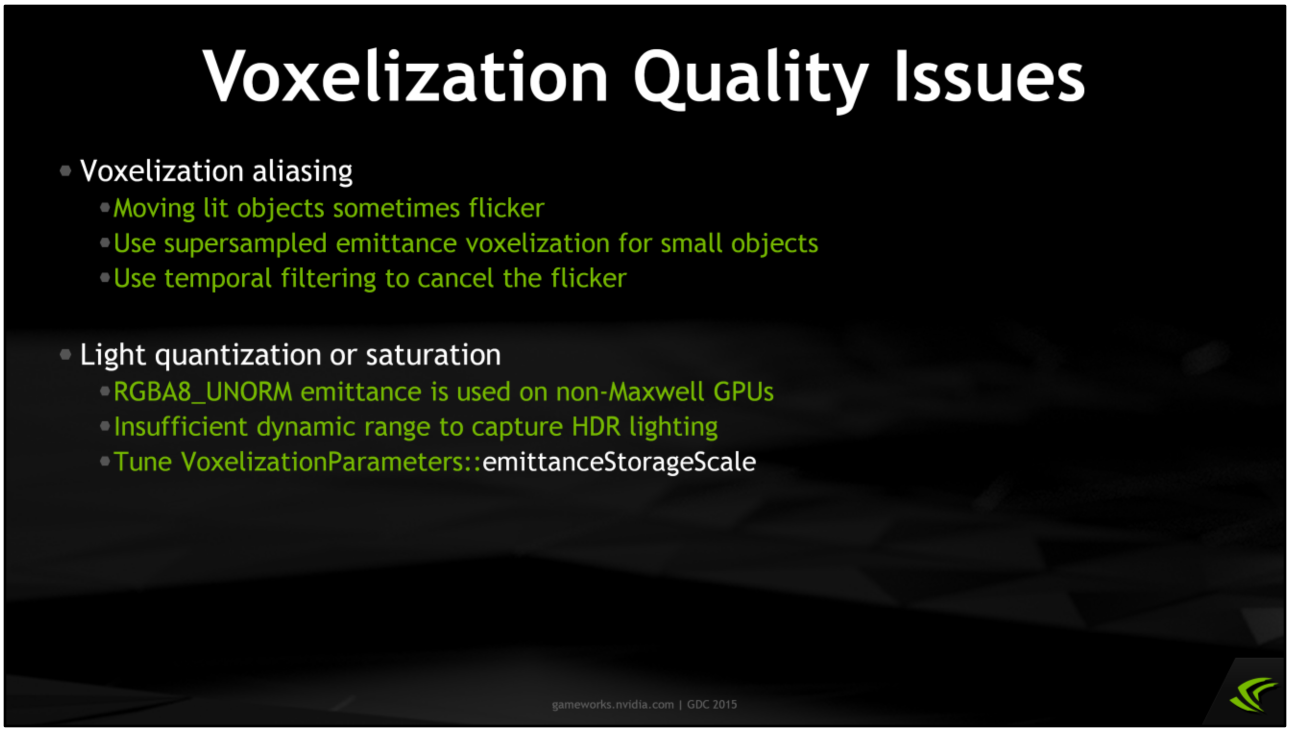
VXGI makes it look more interesting, if not more realistic. This scene has a lot of local lights simulating indirect illumination already, so it’s not really designed for GI.



Outline

- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A

Let’s talk about the quality issues that may appear, how to solve these issues, and how to make sure that VXGI works as fast as it can.



Voxelization Quality Issues

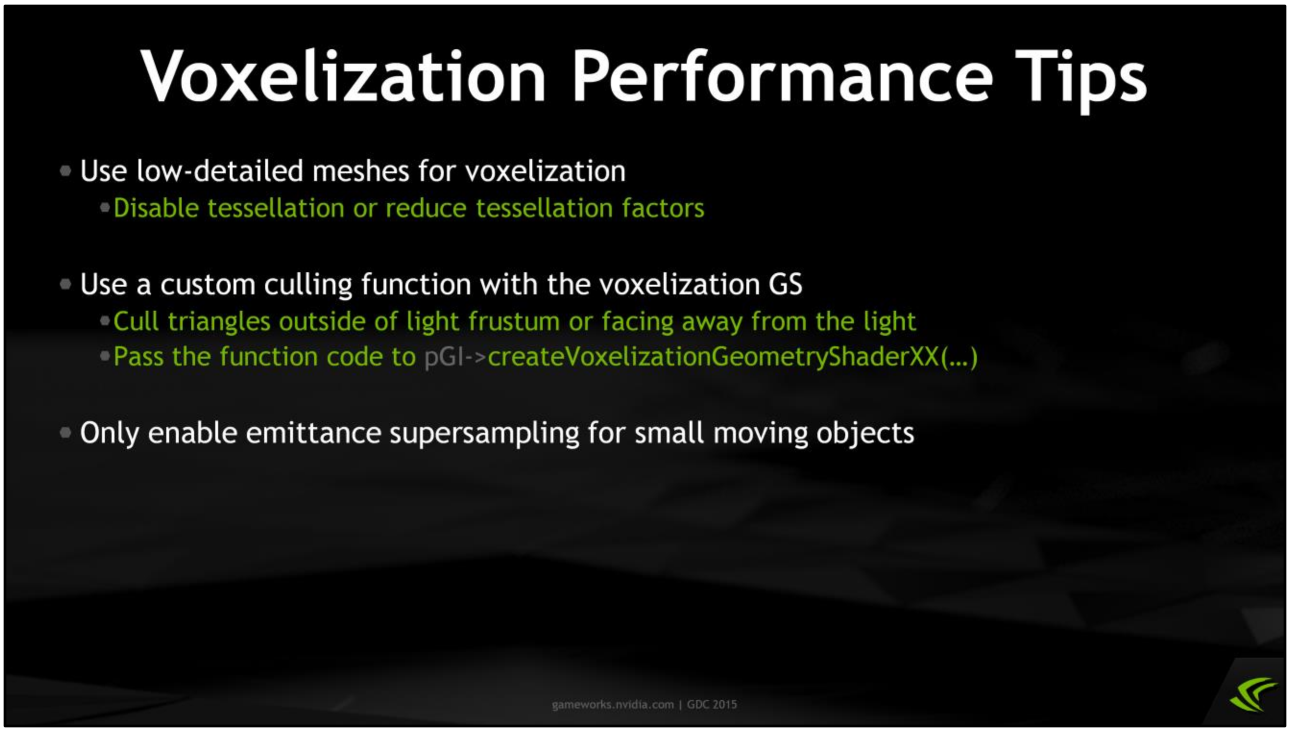
- Voxelization aliasing
  - Moving lit objects sometimes flicker
  - Use supersampled emittance voxelization for small objects
  - Use temporal filtering to cancel the flicker
- Light quantization or saturation
  - RGBA8_UNORM emittance is used on non-Maxwell GPUs
  - Insufficient dynamic range to capture HDR lighting
  - Tune VoxelizationParameters::emittanceStorageScale

gameworks.nvidia.com | GDC 2015

The first step of VXGI algorithm is voxelization, so let's start with that. There are two major issues that happen during voxelization, and the first of them is aliasing.

Since voxelization is performed by rasterizing geometry at a very low resolution, aliasing is really hard to get rid of. You won't see aliasing on static objects because any voxelization of those is plausible, but when objects move, especially slowly, aliasing manifests itself as flickering in indirect illumination or occlusion. There are two things you can do about it. First, if flickering appears on small objects, try enabling supersampled emittance voxelization for those objects or their materials. This mode uses conservative rasterization and uses an optimized sampling pattern in the pixel shader, so it makes sure that small objects are not lost at any resolution, but it is really expensive. Another advice is to use temporal filtering which may cancel the flicker completely.

The second issue with voxelization is that unless FP16 emittance is used, which is possible only on Maxwell, light information can be quantized if it's really dim, or saturated if it's really bright. Try adjusting the emittanceStorageScale voxelization parameter until indirect lighting looks good. We are also considering adding a mode that uses FP32 to store emittance to enable high quality results on non-Maxwell GPUs.



Voxelization Performance Tips

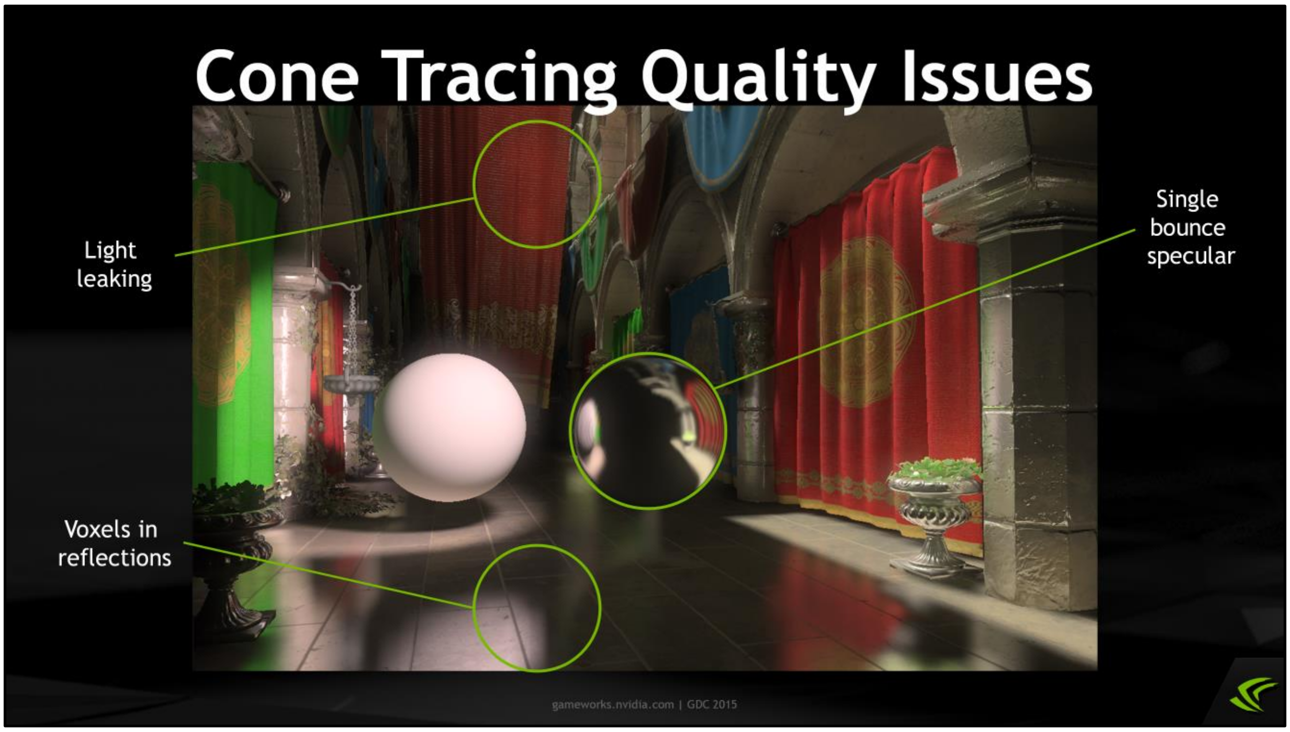
- Use low-detailed meshes for voxelization
  - Disable tessellation or reduce tessellation factors
- Use a custom culling function with the voxelization GS
  - Cull triangles outside of light frustum or facing away from the light
  - Pass the function code to pGI->createVoxelizationGeometryShaderXX(…)
- Only enable emittance supersampling for small moving objects

gameworks.nvidia.com | GDC 2015

Speaking of voxelization performance, there are a few things you can do to improve it. First of all, do not use highly detailed meshes for voxelization. If possible, have separate low-detailed models that have the same general shape as the originals, and that should be enough. And don't use highly detailed tessellation, too, unless it really changes the object shape.

Voxelization of light can be optimized by using a culling function. When creating the voxelization GS, you can pass the code of the function that can discard triangles based on their vertex coordinates. For example, it doesn't make sense to rasterize a triangle that is facing away from the light or that is outside of the light frustum. Also you have the choice of voxelizing all lit geometry once and processing lights in a loop in the shader versus voxelizing lit geometry for each light separately, or any option in between. Most likely, processing several closely located lights at a time will be the fastest option.

And I already mentioned that while supersampled emittance voxelization helps with quality, it's very expensive, so don't enable it on all objects. Especially on non-Maxwell GPUs because Maxwell at least has hardware support for conservative rasterization.

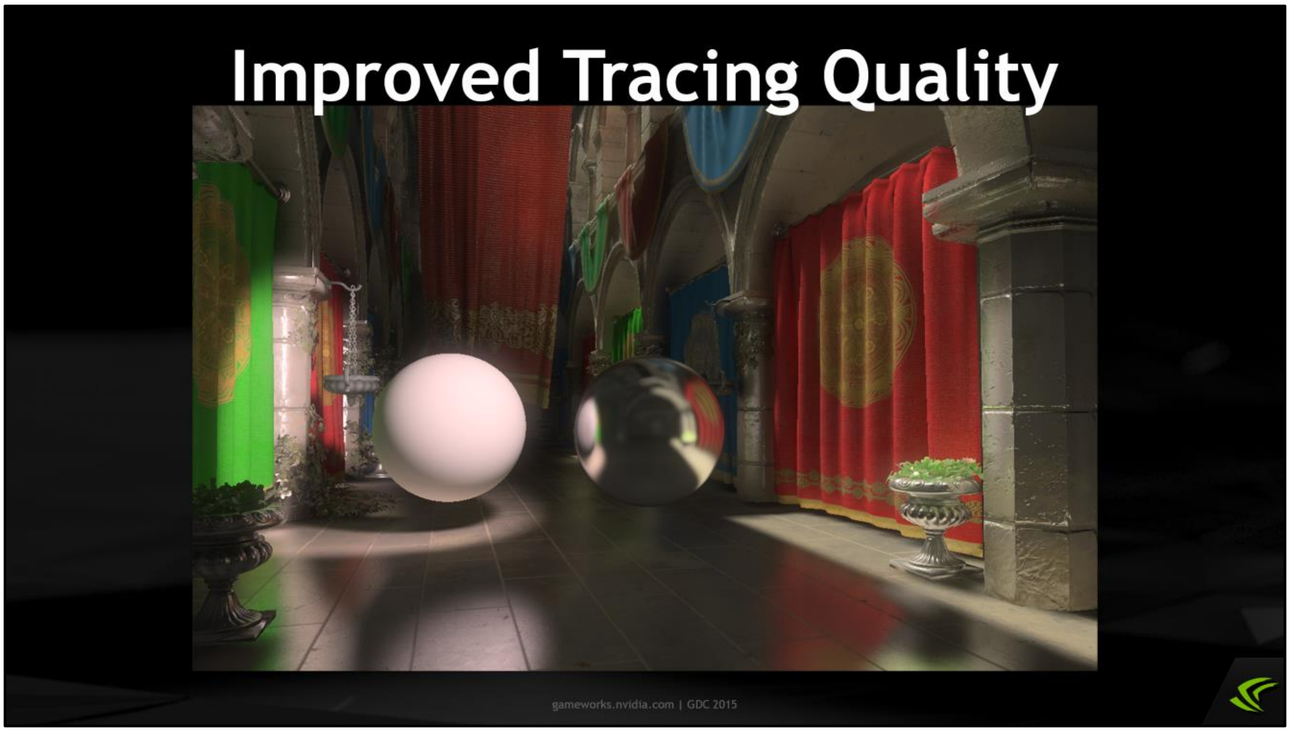


Cone Tracing Quality Issues

After voxelization comes cone tracing, and it also has some issues. One of the most important issues is light leaking, which comes from the fact that we average lighting information over relatively large regions of space. Sometimes light just comes through walls or ceilings, or objects look like they receive subsurface scattering from the other side. To mitigate light leaking, you can try to modify the content and make objects or walls thicker. Also for example if light leaks from the outside of the building into the inside, you can skip the light voxelization on the outside surfaces.

Other two issues are mostly visible in specular reflections. First, if you have a flat reflective surface, like the floor here, you can see that the objects in the reflections are built of voxels, basically the trilinear interpolation artifacts. To hide these artifacts, just make the reflective objects bumpy. Or use another technique like screen space reflections on them. Also there is an experimental feature that's called "tangent jitter" that adds a lot of noise but essentially blurs the reflections with a circular kernel, on average. Temporal filtering can efficiently reduce that noise.

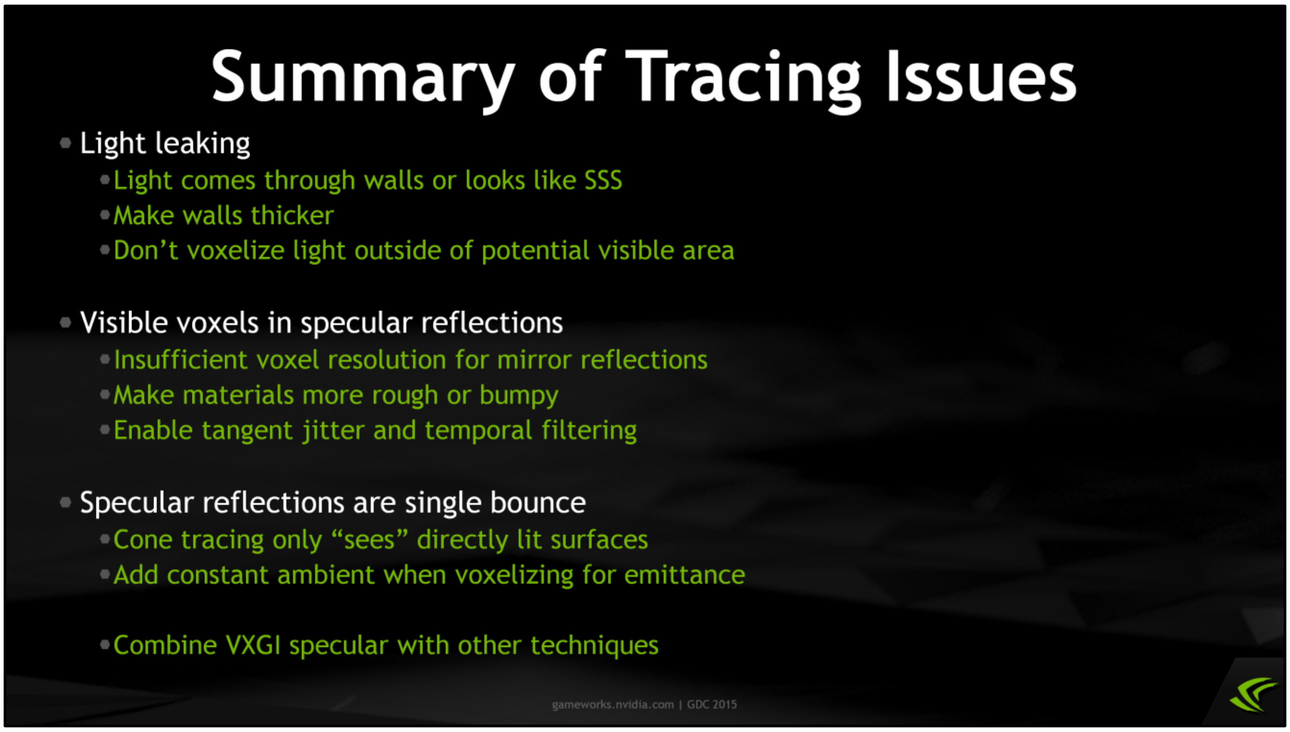
And finally, our specular reflections are only single bounce so far. This means that you only see directly lit surfaces in the reflections. To make other objects appear, just add some amount of ambient light when voxelizing them. In the meantime, we're working on second bounce support, but no promises yet.



Improved Tracing Quality

gameworks.nvidia.com | GDC 2015

This picture shows how adding a little ambient color to voxelization and using tangent jitter improves specular tracing quality. The difference is better visible when the pictures are flipped back and forth.



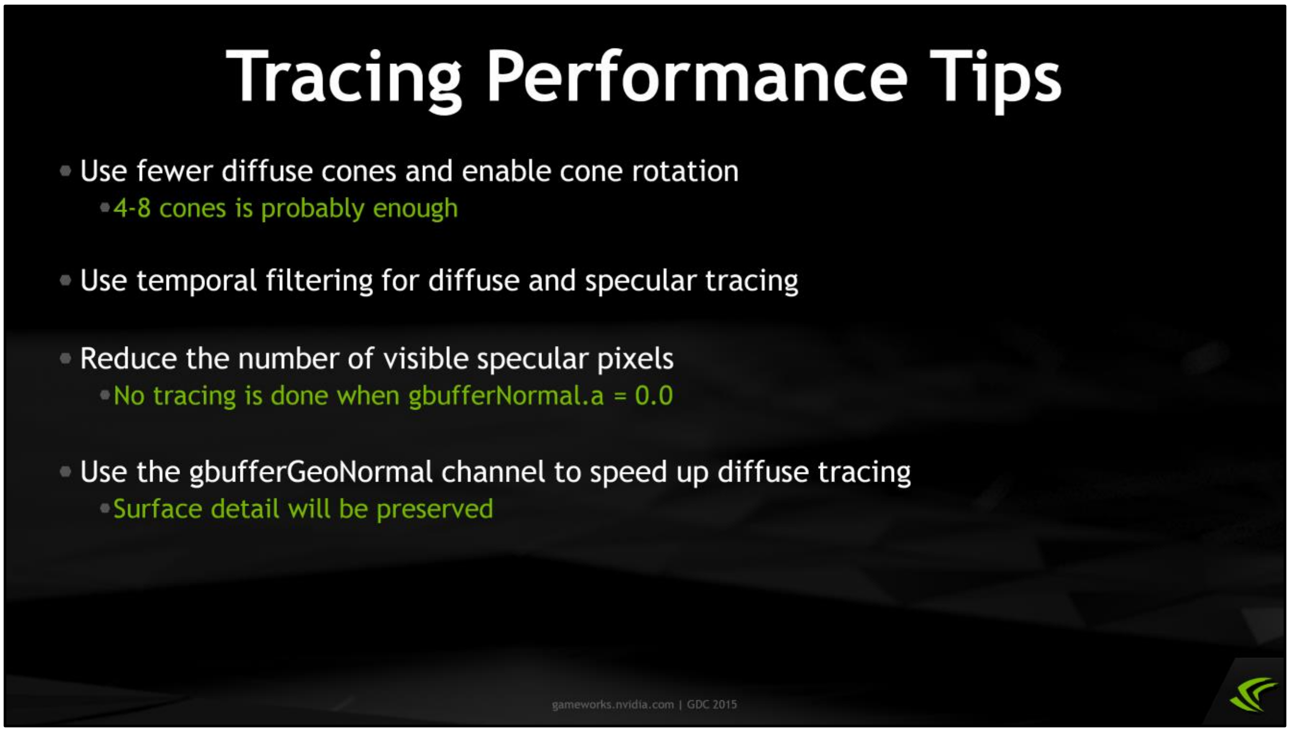
Summary of Tracing Issues

- Light leaking
  - Light comes through walls or looks like SSS
  - Make walls thicker
  - Don’t voxelize light outside of potential visible area
- Visible voxels in specular reflections
  - Insufficient voxel resolution for mirror reflections
  - Make materials more rough or bumpy
  - Enable tangent jitter and temporal filtering
- Specular reflections are single bounce
  - Cone tracing only “sees” directly lit surfaces
  - Add constant ambient when voxelizing for emittance

  - Combine VXGI specular with other techniques

gameworks.nvidia.com | GDC 2015

This is just a summary of the tracing issues and solutions described before.



Tracing Performance Tips

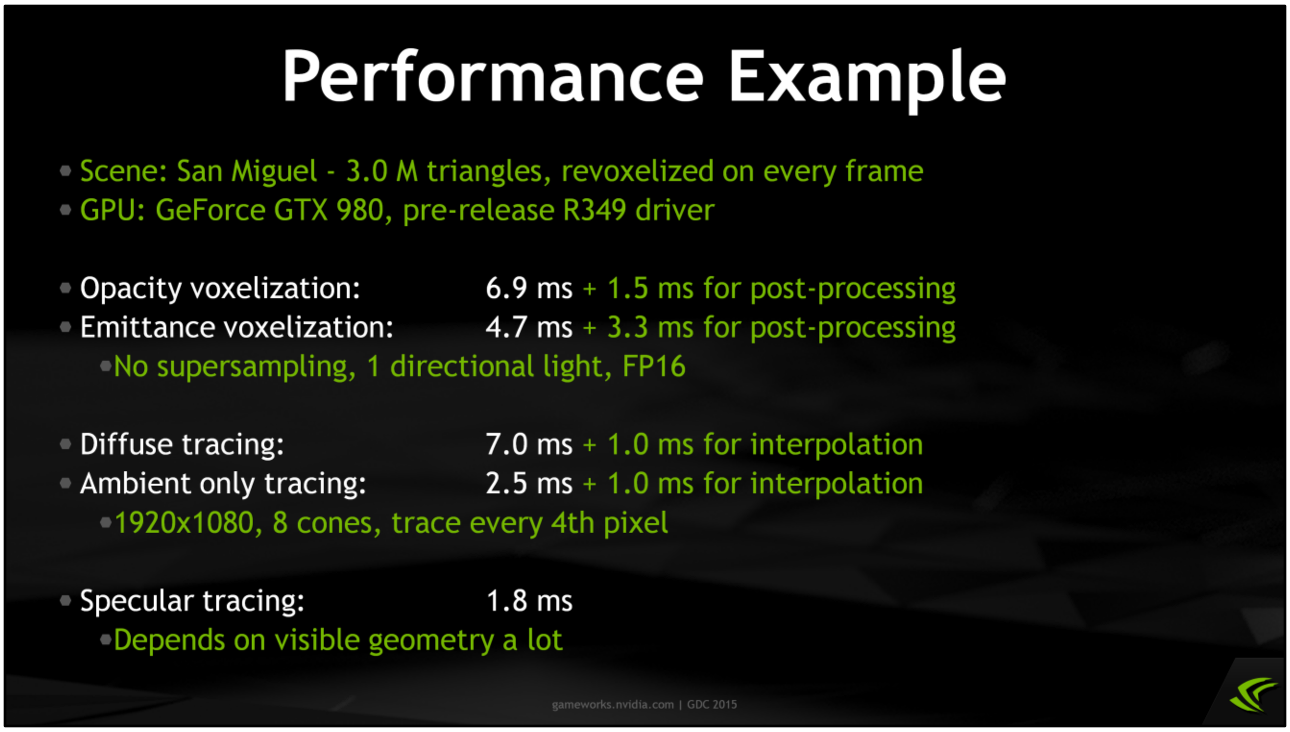
- Use fewer diffuse cones and enable cone rotation
  - 4-8 cones is probably enough
- Use temporal filtering for diffuse and specular tracing
- Reduce the number of visible specular pixels
  - No tracing is done when gbufferNormal.a = 0.0
- Use the gbufferGeoNormal channel to speed up diffuse tracing
  - Surface detail will be preserved

gameworks.nvidia.com | GDC 2015

To improve tracing performance, follow these advices. First, do not use many diffuse cones. 4 or 8 cones are most likely enough. Maybe 16. Sparse diffuse tracing also helps to improve performance a lot, but doesn't work well for scenes with high geometric complexity. But temporal filtering can make it work for such scenes, and also it can reduce flickering on moving objects, and the noise coming from tangent jitter on specular cones.

Reducing the number of specular surfaces is kind of obvious because tracing is performed for every visible specular pixel, so if the surface is not specular, set its roughness to 0.

And finally, the optional G-buffer smooth normal channel, which doesn't include normal maps, helps improve diffuse tracing performance at no image quality co



Performance Example

- Scene: San Miguel - 3.0 M triangles, revoxelized on every frame
- GPU: GeForce GTX 980, pre-release R349 driver

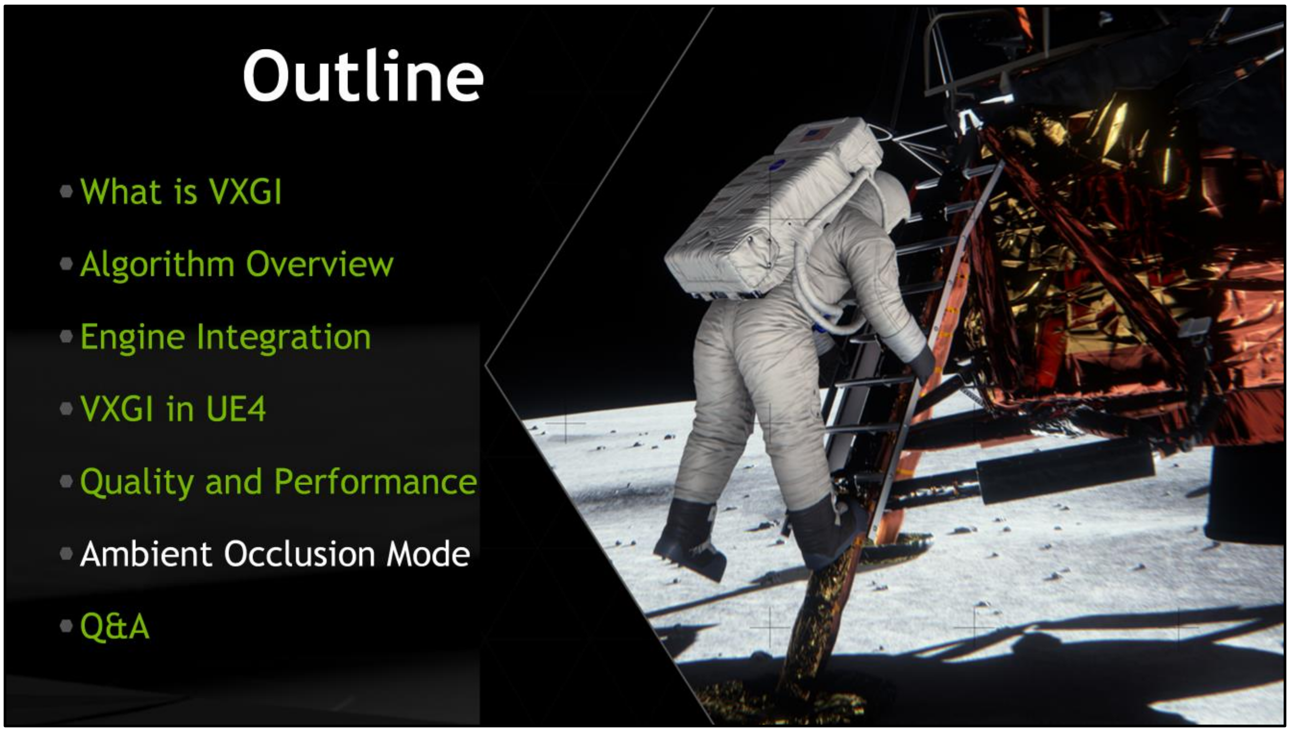
- Opacity voxelization: 6.9 ms + 1.5 ms for post-processing
- Emittance voxelization: 4.7 ms + 3.3 ms for post-processing
  - No supersampling, 1 directional light, FP16

- Diffuse tracing: 7.0 ms + 1.0 ms for interpolation
- Ambient only tracing: 2.5 ms + 1.0 ms for interpolation
  - 1920x1080, 8 cones, trace every 4th pixel

- Specular tracing: 1.8 ms
  - Depends on visible geometry a lot

gameworks.nvidia.com | GDC 2015

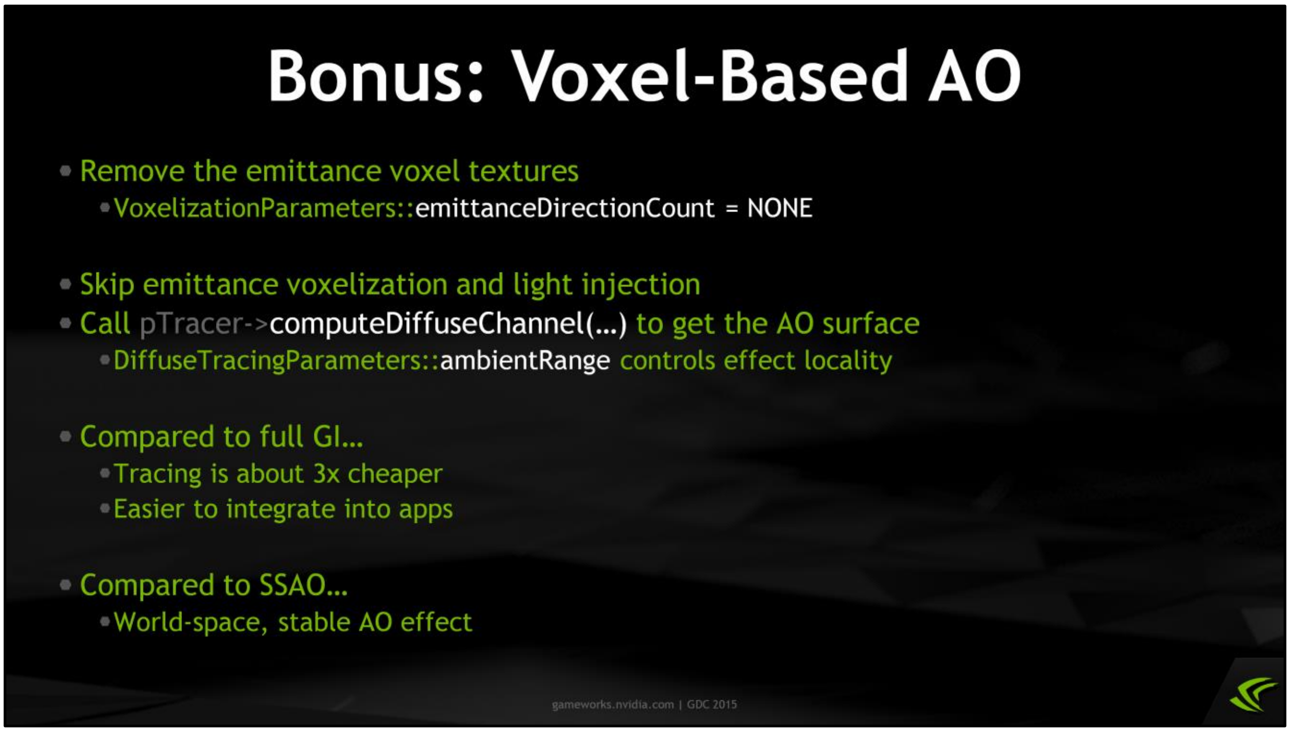
In general, VXGI performance depends on the scene and the configuration a lot. To give you some idea about real numbers, let's consider the San Miguel scene which has 3 million triangles, and that scene is revoxelized completely on every frame – which is optional. So, on a GTX 980 with a pre-release driver that includes some optimizations for the voxelization passes, opacity voxelization of those 3 million triangles takes about 7 milliseconds. And emittance voxelization of the whole scene with one shadow mapped light takes 4.7 milliseconds. Diffuse tracing at full HD resolution with relatively high quality settings takes another 7 milliseconds, and specular tracing at the same resolution takes just under 2 milliseconds – but that is because there are not that many specular surfaces in the scene. This may seem like a lot, but more game-like scenes which have an order of magnitude simpler geometry have better performance.



Outline

- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A

The final part of my presentation is about ambient occlusion that you can also get from VXGI.



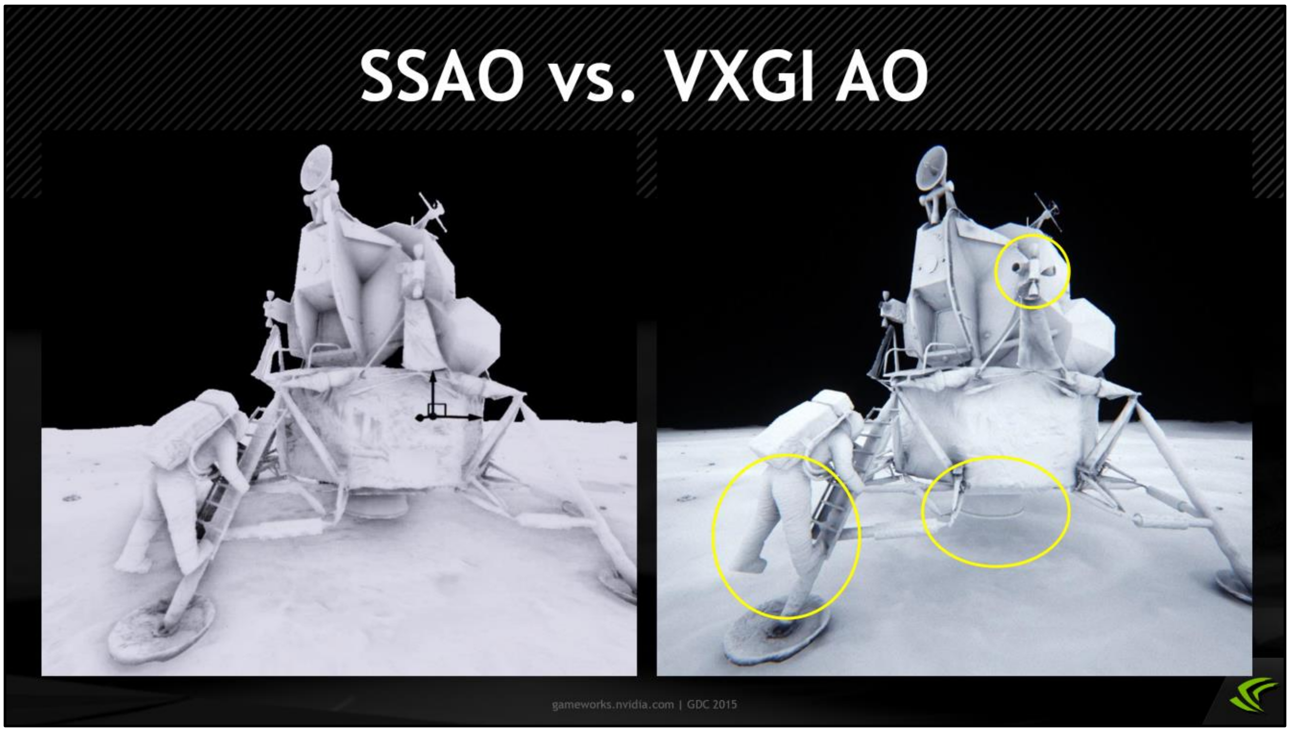
Bonus: Voxel-Based AO

- Remove the emittance voxel textures
  - VoxelizationParameters::emittanceDirectionCount = NONE
- Skip emittance voxelization and light injection
- Call pTracer->computeDiffuseChannel(…) to get the AO surface
  - DiffuseTracingParameters::ambientRange controls effect locality
- Compared to full GI…
  - Tracing is about 3x cheaper
  - Easier to integrate into apps
- Compared to SSAO…
  - World-space, stable AO effect

gameworks.nvidia.com | GDC 2015

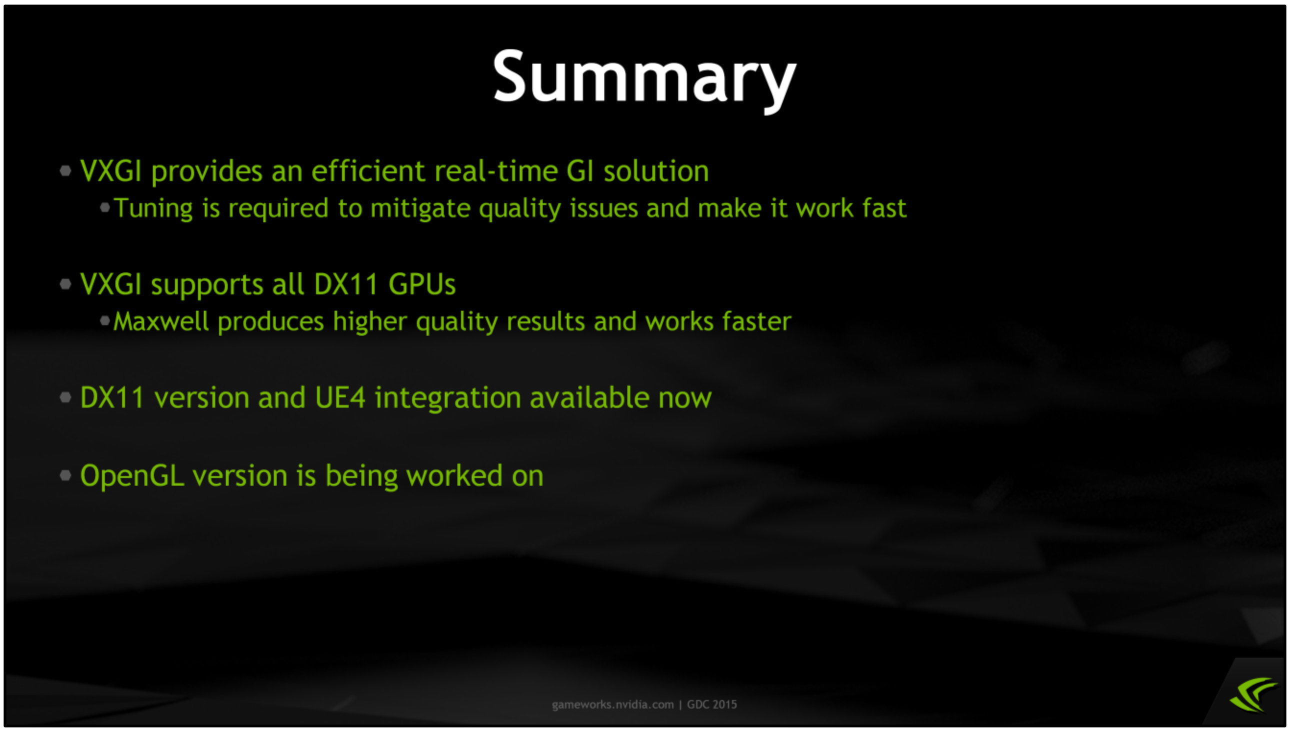
Like I mentioned earlier when talking about cone tracing, we compute occlusion when marching through the cone and accumulating emittance. So, if we drop the whole emittance part, we can get approximate occlusion for every visible point. And that is actually much cheaper than computing global illumination because there is no emittance voxelization, and tracing doesn't have to sample the emittance textures. That makes tracing something like 3 times cheaper, as you may have noticed on the previous slide talking about performance.

Modern game engines compute ambient occlusion based on screen space information, using various algorithms commonly known as screen-space ambient occlusion. The results of SSAO are not always correct, for example when geometry comes near the screen edge or when a far object is close to a near object in screen space. Voxel based AO has no such problems.



SSAO vs. VXGI AO

On the left picture there is a rendering of the Lunar lander with screen-space ambient occlusion, and on the right is the same lander with VXGI ambient occlusion. The moon surface behind the astronaut and some parts of the lander should not be occluded, which is captured correctly by VXGI but not by SSAO.



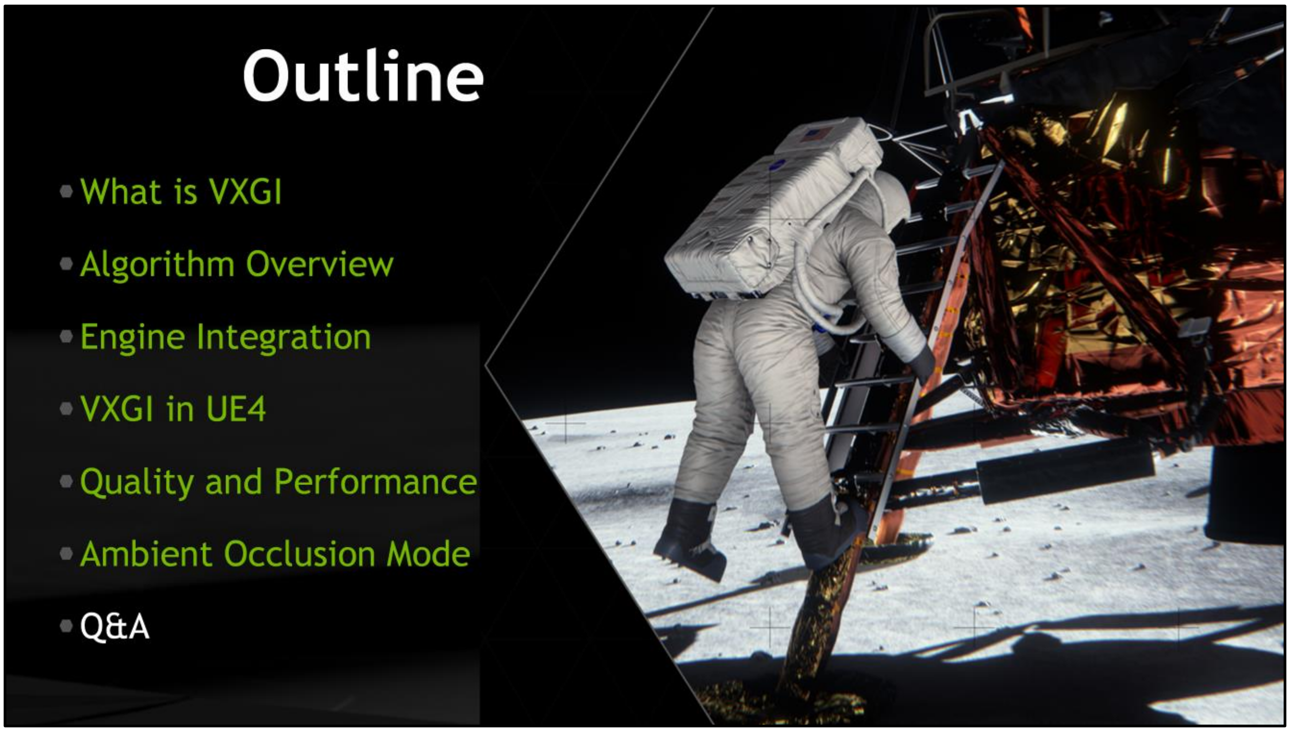
Summary

- VXGI provides an efficient real-time GI solution
  - Tuning is required to mitigate quality issues and make it work fast
- VXGI supports all DX11 GPUs
  - Maxwell produces higher quality results and works faster
- DX11 version and UE4 integration available now
- OpenGL version is being worked on

gameworks.nvidia.com | GDC 2015

To summarize, VXGI is a library that provides an efficient real-time global illumination and ambient occlusion solutions. But in order to make it work fast and hide some quality issues, you may have to tune it or modify your content a little. It works on all DirectX 11 class GPUs, but it was optimized specifically for Maxwell, whose new hardware features allow us to produce higher quality results faster.

You can download and try VXGI as a part of Unreal Engine 4 now; a standalone package with DirectX samples should be available on the Nvidia developer website soon, and the OpenGL version is being worked on.



Outline

- What is VXGI
- Algorithm Overview
- Engine Integration
- VXGI in UE4
- Quality and Performance
- Ambient Occlusion Mode
- Q&A

GameWorks

- Get the latest information for developers from NVIDIA and continue the discussion
- gameworks.nvidia.com