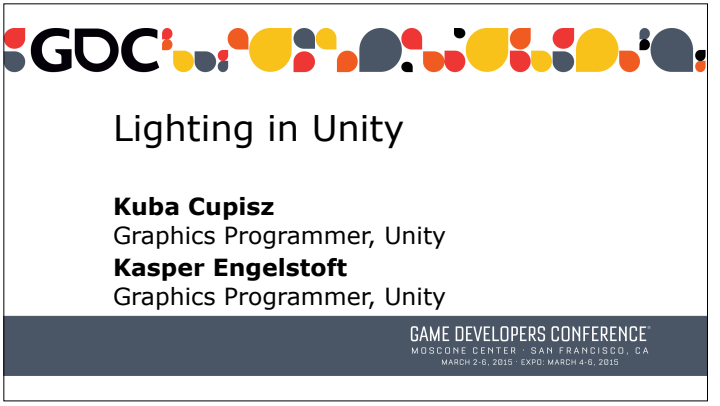
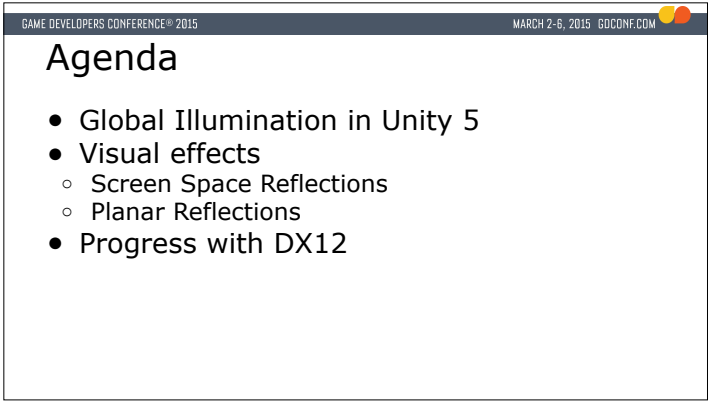


Welcome to the talk about lighting in Unity 5.
I'm Kuba Cupisz and this is Kasper Engelstoft and we're graphics programmers at Unity Technologies.

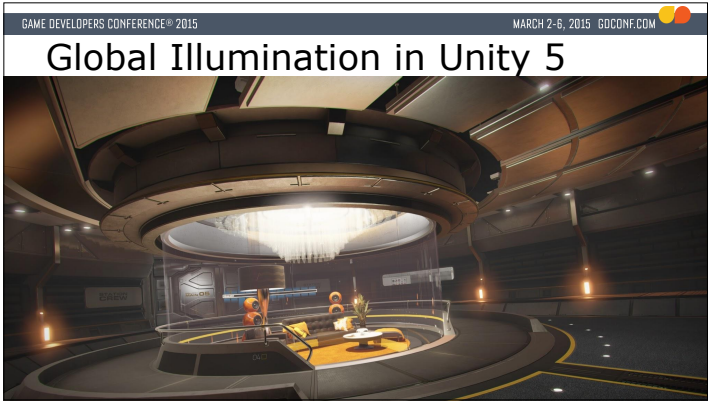


Today I'll cover how global illumination is handled in Unity 5. Then two related techniques will be described: screen-space reflections and planar reflections.

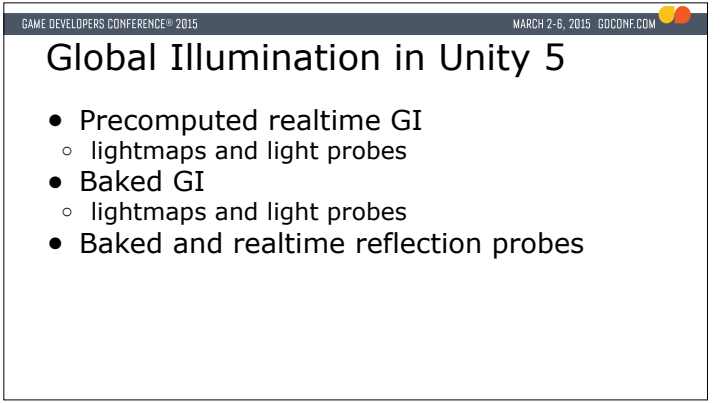
Then Kasper we'll talk about our progress with DX12.

To give you a bit of context for the talk, let’s watch a video from a project that one of our users made in the beta:
http://youtu.be/Fr4H0crCb0E

What is important to keep in mind when listening to this talk is that Unity is used in many many different ways. We aren't writing graphics code for 4 million players. We're writing it for 4 million developers and 600 million more players.



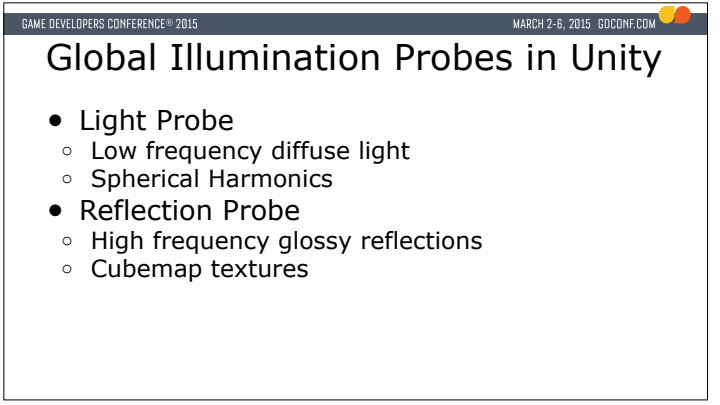
The way we approach some problems and the tradeoffs we need to make might be different than what you’re used to.



Let's start by talking about GI.

Global Illumination in Unity 5 is handled by 3 subsystems:
- precomputed realtime GI that delivers lightmaps and light probes that are updated at runtime
- baked GI that gives lightmaps and light probes that don't change at runtime
- reflection probes that can be baked or rendered at runtime

Those 3 components can freely be combined.



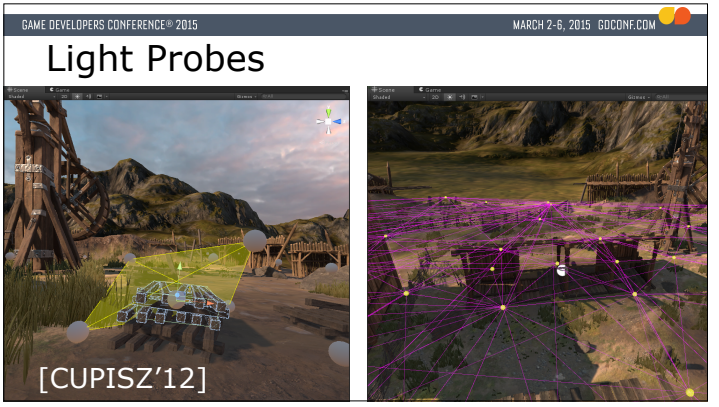
We use the following naming conventions for probes in Unity:
- a light probe is an SH probe that stores low frequency diffuse lighting in spherical harmonics
- a reflection probe is a probe that stores high frequency glossy reflections in cubemap textures

Precomputed Realtime GI

- Lightmaps and light probes updated realtime
  - dynamically change light sources, emissive materials and environment lighting
- Static objects lit by realtime lightmaps
- Dynamic objects lit by realtime light probes
  - 4 SH coefficients per channel
- Direct light computed on the GPU

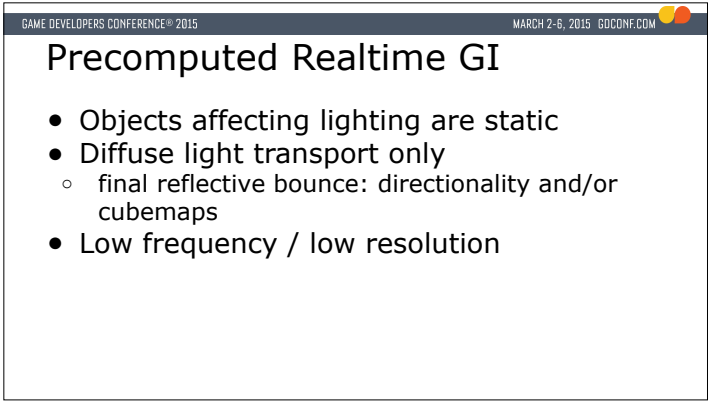
Precomputed Realtime GI uses Enlighten to precalculate light transport data that allows for computing GI at runtime. Think of it as precomputed global visibility which can be resolved in realtime to derive GI for the current light setup. Changes to light sources (so creating, deleting and changing their properties), emissive material changes and changes to environment lighting are the runtime inputs, and outputs are calculated asynchronously.

Static objects are lit by realtime lightmaps whereas dynamic objects are lit by realtime light probes.
Direct lighting is calculated on the GPU the normal way.



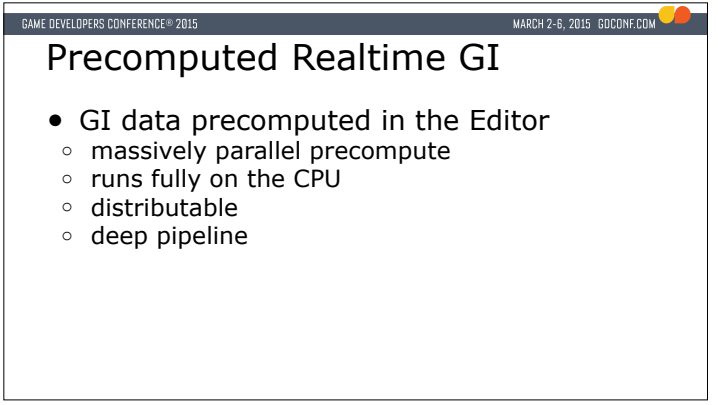
Light probes are just points at which lighting is calculated and encoded into spherical harmonics. This cloud of points is tetrahedralized as that gives us a clean approach to interpolation and extrapolation. It also avoids under- and oversampling. Please check Robert Cupisz's excellent talk on the topic from GDC 2012.

[CUPISZ'12] Robert Cupisz, "Light Probe Interpolation Using Tetrahedral Tessellations", GDC 2012, http://gdcvault.com/play/1015312/Light-Probe-Interpolation-Using-Tetrahedral



The precompute happens in the Editor and all the objects affecting GI and their positions need to be known and cannot change at runtime. Or rather, they can, but the lighting will not update because of that.
Only the diffuse light transport is handled in the simulation, but there is a possibility to have the final reflective bounce. That can be achieved either by using additional directionality information or reflection probes. More on that later.

Indirect lighting is typically low frequency and that allows the realtime lightmaps to use low resolution. To make that possible your typical lightmapping UVs go through a process of simplification (where insignificant charts are projected onto other ones). Also the chart hull is aligned to texel centers, which allows for completely getting rid of bleeding between neighbouring charts while keeping the packing tight.

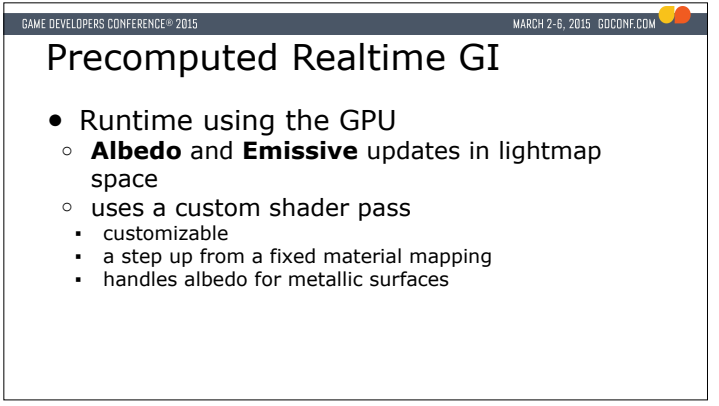


As mentioned earlier the GI data is precomputed in the Editor. The world is split into chunks which allows most stages (especially the slow ones) to go really wide.
It runs fully on the CPU and that means that it's easier to distribute to different machines. Most existing cloud solutions still don't have proper GPUs or proper GPU access, and those that do cost a lot more. Also consistency of results is important.
The pipeline is "deep", so when a local change is made a lot of already calculated data will be reused.

Precomputed Realtime GI

- Runtime running on the CPU
  - games are typically GPU bound
  - low-end GPUs not powerful enough
  - heterogeneous GPU ecosystem
  - portable
  - optimized platform specific solvers

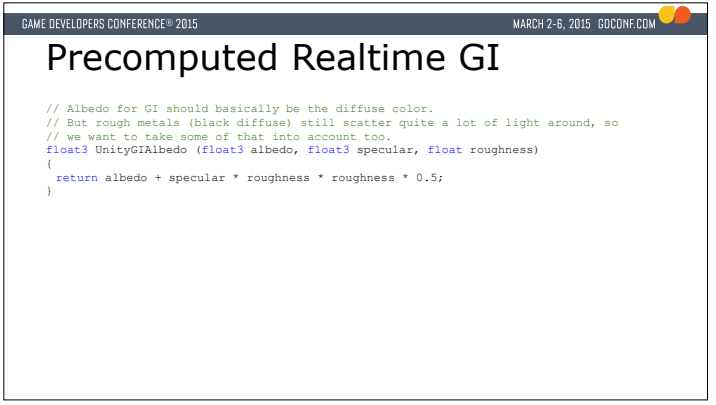
The runtime itself also runs on the CPU. Current games are still typically GPU bound. Low-end GPUs are getting more and more powerful by the minute, but are still not fast enough for what we need.
The GPU ecosystem is still quite… diverse.
The runtime is simple and the code is extremely portable. There are however platform specific solvers that have been hand-optimized: SSE, Neon, etc.



Part of the runtime still runs on the GPU. That is used for albedo and emissive rendering in lightmap space.

The way the object looks is normally defined by its shader. Unity 3 and 4 used a simple mapping from material properties to lightmapper material properties. It worked ok, but was based on certain naming conventions, tags, etc. It couldn't accommodate for any custom operations users might do to calculate surface properties.

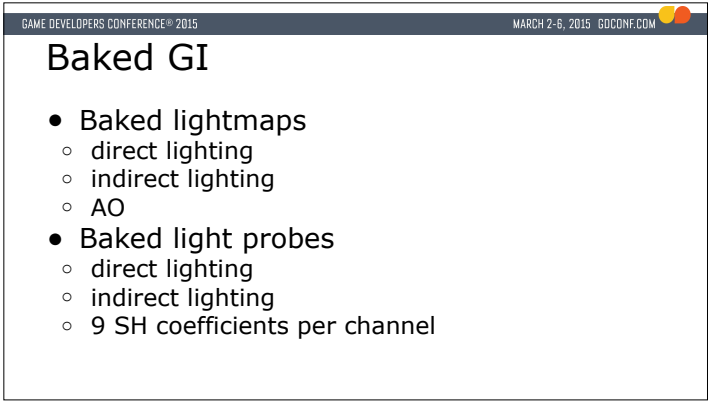
In Unity 5 albedo and emissive is rendered in lightmap space on the GPU and there is a special shader pass that controls the behaviour. That means that the code that defines how the object looks on screen and how it looks to the lightmapper is kept in one place and can easily be customized along with the shader. Also, you can control GI without affecting the shader used for realtime rendering.



Enlighten handles diffuse transport and uses surface albedo on each bounce. Metallic surfaces with black or almost black albedo would not bounce any light. The shader pass that renders albedo biases it towards a brighter color with the hue of the metal.

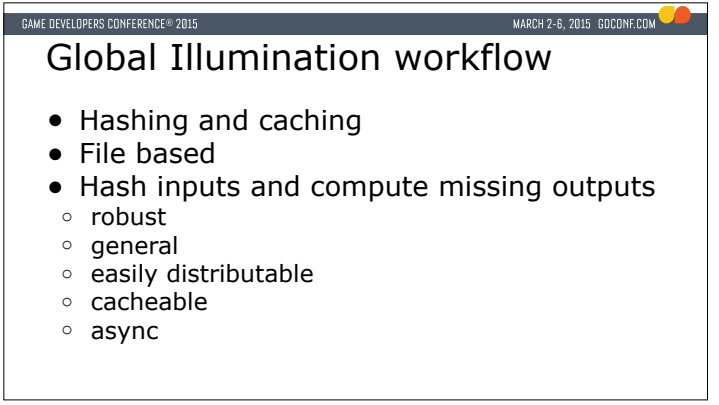
Dielectric materials (wood, plastic, plastic, stone, concrete, leather, skin) have white specular reflectance. Metals have spectral specular reflectance.

This formula was derived based on experiments to generally look good. One can also write their own version of this shader pass.



To keep the runtime cost low and also allow for high resolution lightmaps we still have baked lightmaps and light probes.

Baked lightmaps store direct lighting and indirect optionally multiplied by AO to artificially boost contrast.
Baked light probes store direct and indirect lighting in 9 SH coefficients per channel.

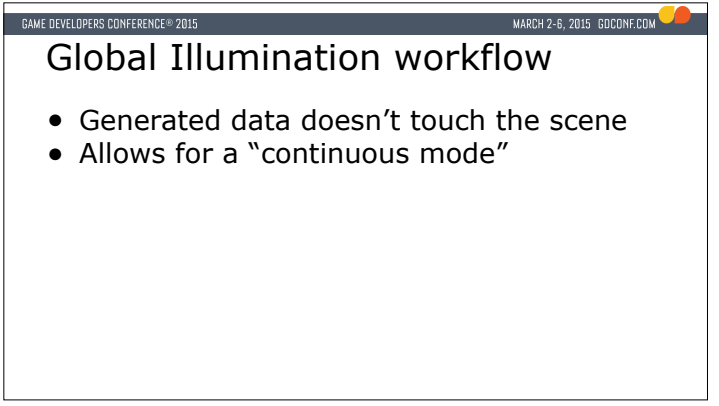


What changed entirely in Unity 5 is the workflow. It's based on hashing and caching.
Inputs to global illumination computations are hashed (geometry, instances, lights, environment lighting, materials, textures, etc.). The pipeline is split into stages and each stage can have multiple jobs running in parallel. Each job has a hash (which is the cumulative hash of all the inputs) and writes the output into a file named with the hash.
The jobs in the following stages can then use that hash as their input and access this file easily.

For each phase a set of inputs affecting the output is hashed.

This approach is very robust - the state is computed in each tick and compared with the already processed state. The Editor and jobs running in external processes can be killed and the system will pick up where it left off.
It's also general - the system is clean and relatively special-case-free. Because of the fact that most stages do in fact schedule many jobs and each job in a stage is completely separate from others -- the jobs can easily be distributed.

Only the relatively cheap code hashing things runs on the main thread and everything else is completely asynchronous.



Since all of this data is now cached/stored outside of the scene the scene is not made dirty when a bake finishes.

Unity 5 has two modes. The continuous mode that automatically schedules jobs to keep the system up to date with current scene contents. And a non-continuous mode where the bake is triggered by the user and the end results are not only cached, but also stored in the Assets folder, so that they can be versioned.

Lightmapping modes

- 3 modes for baked & realtime lightmaps
- Quality vs performance

Lightmapping is a tricky business and unfortunately forced us to provide options that allow to pick the quality vs performance ratio.
Options are rarely good as the user needs to discover them, understand them and pick the right one.
There are 3 modes available for baked and realtime.

Lightmapping modes

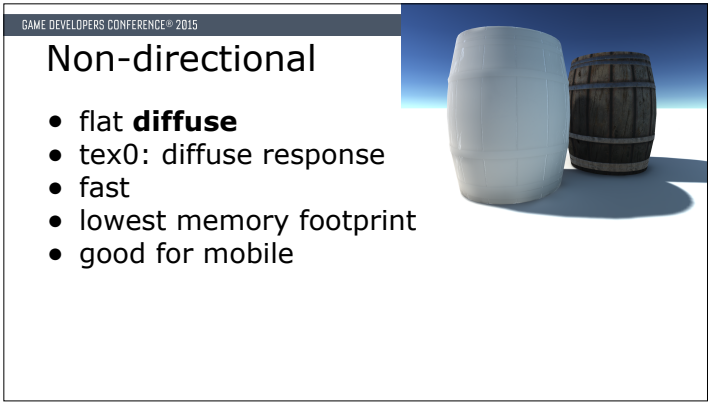
- Quality
  - normal mapping
  - view dependent lighting

By quality I don’t mean the resolution, but mostly the “effects” that are typically not present on lightmapped surfaces: normal mapping and view dependent lighting.

Lightmapping modes

- Performance
  - shading performance
  - memory footprint

Performance on the other hand is defined by the shading complexity and the memory footprint.



The non-directional mode uses just a single lightmap, storing information about how much light leaves the surface, assuming it's purely diffuse. Objects lit this way will appear flat (normalmaps won't be used) and diffuse (even if the material is specular), but otherwise will be lit correctly.

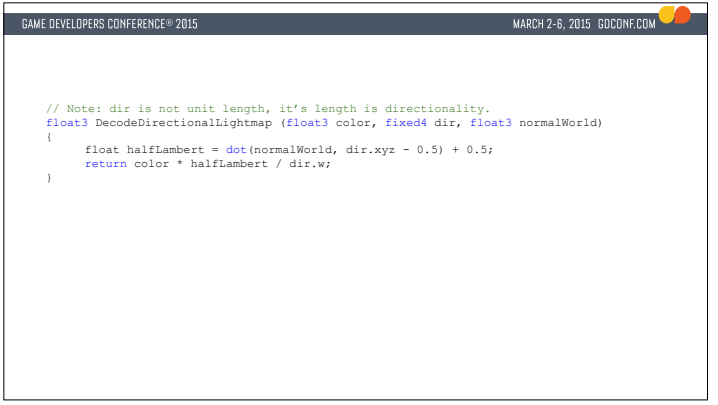
Note: realtime lighting is not subject to those limitations and, if used, will just be added on top.

Directional mode

- normalmapped **diffuse**
- tex0: diffuse response
- tex1: directionality (fixed4)
  - dominant light direction, length=directionality
  - rebalancing coefficient
- combined directionality for direct and indirect
- relatively fast, twice the memory

The directional mode adds a secondary lightmap, which stores the incoming dominant light direction and a factor proportional to how much light in the first lightmap is the result of light coming in along the dominant direction. The rest is then assumed to come uniformly from the entire hemisphere. That information allows the material to be normalmapped, but it will still appear purely diffuse.

Note: combined directional information may cause issues where the dominant light direction for direct and indirect diverges a lot.



In addition to the scaled direction vector, the w-component of the directional texture contains an important rescaling value. This factor handles rescaling due to the variable length of the direction vector, and the effect of the half-Lambertian falloff. It ensures that the additional directional information does not have any impact on lighting surfaces which do not need it (that is, surfaces that have the same normal as the one used during the radiosity computation).

Directional specular

- **diffuse** and **specular**
  - both normalmapped
- tex0: intensity
- tex1: directionality (fixed3)
  - dominant light direction, length=directionality

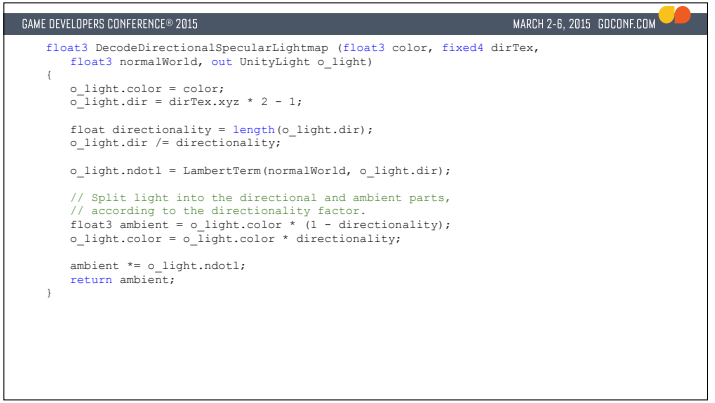
The directional specular mode allows for full shading. Like the previous mode, this one uses two lightmaps. Unlike the two other modes, light is stored as incoming intensity. That extra information allows the shader to run the same BRDF that's usually reserved for realtime lights, resulting in a full-featured material appearance - now also interacting with indirect light.

Directional specular mode

- separate directionality for direct and **indirect**
- less fast, quadrupled memory footprint
- 2 textures, 4 texture samples

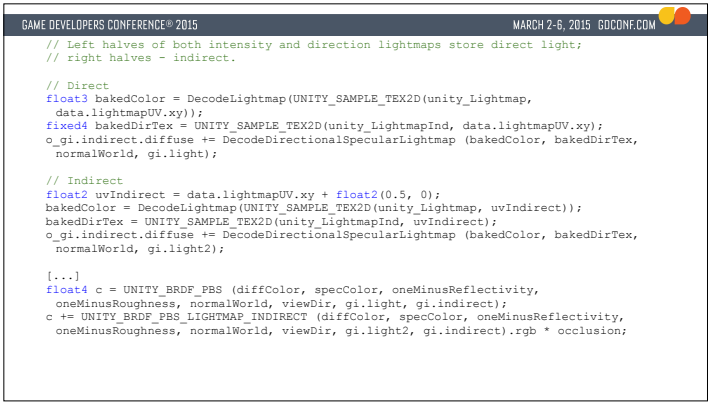
Those two lightmap sets are split in halves, left side stores direct light, right - indirect.

This mode is more expensive to decode and uses 4 times the amount of memory of the non-directional mode.



The length of the direction vector is the light's "directionality", i.e. 1 for all light coming from this direction, lower values for more spread out, ambient light.
The incoming lighting is split into two parts: one treated as the directional lighting (for which the BRDF is run) and the other which is just the ambient.

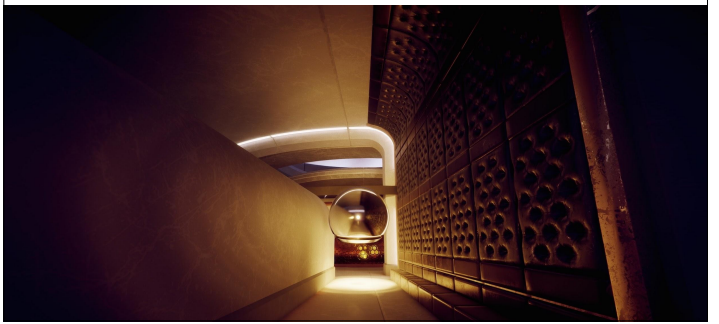
The last multiplication by ndotl is technically incorrect, but helps hide jagged light edge at object silhouettes and makes normalmaps show up.



```
// Left halves of both intensity and direction lightmaps store direct light;
// right halves - indirect.

// Direct
float3 bakedColor = DecodeLightmap(UNITY_SAMPLE_TEX2D(unity_Lightmap,
 data.lightmapUV.xy));
fixed4 bakedDirTex = UNITY_SAMPLE_TEX2D(unity_LightmapInd, data.lightmapUV.xy);
o_gi.indirect.diffuse += DecodeDirectionalSpecularLightmap (bakedColor, bakedDirTex,
 normalWorld, gi.light);

// Indirect
float2 uvIndirect = data.lightmapUV.xy + float2(0.5, 0);
bakedColor = DecodeLightmap(UNITY_SAMPLE_TEX2D(unity_Lightmap, uvIndirect));
bakedDirTex = UNITY_SAMPLE_TEX2D(unity_LightmapInd, uvIndirect);
o_gi.indirect.diffuse += DecodeDirectionalSpecularLightmap (bakedColor, bakedDirTex,
 normalWorld, gi.light2);

[...]
float4 c = UNITY_BRDF_PBS (diffColor, specColor, oneMinusReflectivity,
 oneMinusRoughness, normalWorld, viewDir, gi.light, gi.indirect);
c += UNITY_BRDF_PBS_LIGHTMAP_INDIRECT (diffColor, specColor, oneMinusReflectivity,
 oneMinusRoughness, normalWorld, viewDir, gi.light2, gi.indirect).rgb * occlusion;
```

Direct and indirect are decoded in the same way. The difference is that a cheaper version of the BRDF is run on the extracted indirect lighting information and the result is multiplied by a high res occlusion map.

Reflection Probes



Reflection Probes

- Provide approximated specular reflection
- Component on a GameObject
- A box defines:
  - the influence region
  - the box projection [BEHC'10]

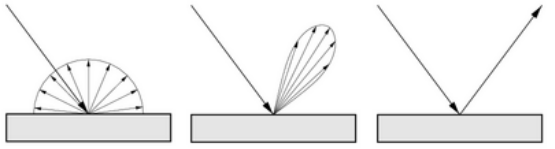
Reflection probes provide approximated specular reflections. They're just cubemaps you're all too familiar with.
They're added as a component on a game object, which allows them to naturally be placed, manipulated and referenced in the world.

Their influence region is defined by a box and all the objects intersecting it will potentially be affected by it. Currently the same box defines the cuboid for the box projection.

[BEHC'10] behc, "Box Projected Cubemap Environment Mapping", http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/

Reflection Probe Convolution

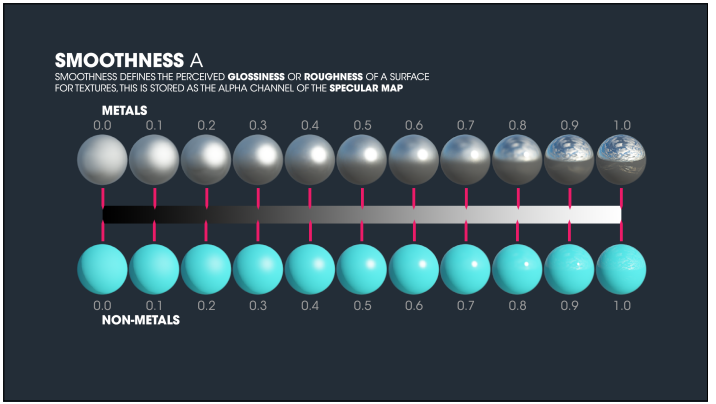
- Needed for rough surfaces



[LAMRUG]

If the object is perfectly specular (a mirror, diagram on the right), a single texel of the cubemap is required to light a point on the surface. However for rough surfaces (diagram in the middle) many more texels are required and sampling those texels at runtime is a computationally expensive process.

[LAMRUG] Image from http://www.lamrug.org/resources/indirectips.html



So instead the cubemap can be preprocessed for a number of specular lobe's (having a different spread) corresponding to different smoothness values.

The specular lobe is a cosine lobe as a typical BRDF has a cosine and an exponent. When the exponent grows the lobe gets narrower and narrower and in the limit gives just a ray.

Ideally one would use the same BRDF for processing the cubemap as for computing analytical lighting in the shader. Different objects can use different BRDFs though, so for processing cubemaps we just need to pick something that is close enough and cheap enough.

Conveniently the less smooth the surface is the lower frequency the specular reflections are. That means that for lower smoothness values one can pack the results of the preprocessing into smaller mip levels of a mip chain without a noticeable loss. Just make sure to use trilinear filtering.

Reflection Probe Convolution

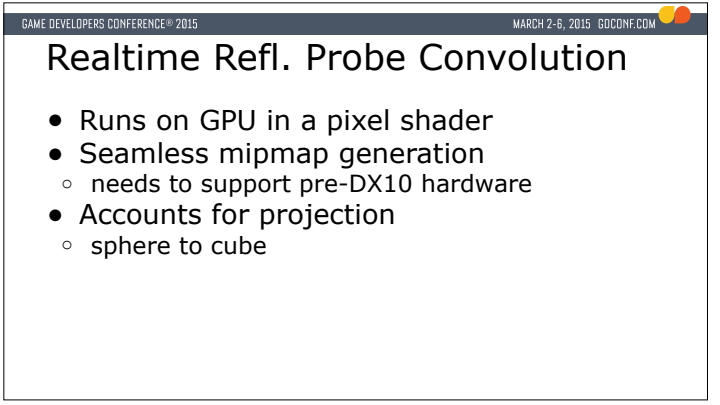
- What is convolution ($f*g$)?
  - integral of the product of the two functions after one is reversed and shifted
  - f - environment lighting signal
  - g - specular part of BRDF
  - in image processing - a filter running over the image

The preprocessing that was just mentioned is called convolution.

Convolution is a mathematical operation on two functions f and g, producing a third function. It's the integral of the product of the two functions after one is reversed and shifted.

In the context of reflection probes function f is the environment lighting signal. Function g is the BRDF. Since reflection probes are only used for the specular part of BRDF, function g is representing just that.

In image processing that's equivalent to just running a filter with a certain kernel size over each position in the image.



Since we already have realtime lightmaps and realtime light probes, reflection probes (with convolution) also needed to be realtime to make the system complete.

Naively computing mipmaps for each face doesn't work for two reasons: texels for contiguous faces are not taken into consideration (resulting in seams) and texels near borders and corners need a wider filter to accommodate the fact that they are further from the center of the cube than the center texels (a cubemap is in fact a function on a sphere projected onto a cube).

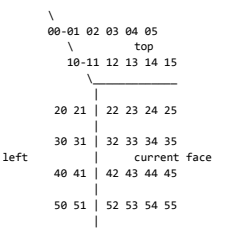
Convolution - folding

- For each texel
  - check the coordinate
  - ‘fold’ it on the neighbouring face if needed

Sampling is done against texel centers. For edges and corners, parts of the 6x6 kernel will need texels from neighbouring faces. For each texel fetch the shader checks the coordinates and ‘folds’ them on the neighbouring face when needed. This means that the uv coordinates will be beyond the [-1, 1] range. The amount that exceeds this range is subtracted from face’s own axis.

Convolution - folding

- One uv coord outside: OK
- Both outside
  - No in between face
  - Fetch any texel along the edge
  - Branch free shader



Folding is well defined if one uv coordinate is outside the range. If both of them exceed the range then they point to an “in between” face that doesn’t exist. Ignoring those samples would add branches to the shader, so instead we fetch any texel along the edge.

For edges and corners, parts of the 6x6 kernel will need texels from neighbouring faces. This means that the uv coordinates will be beyond the [-1, 1] range. The amount that exceeds this range is subtracted from the face’s own axis.

In the example diagram on the right the 6x6 kernel sticks out by 2 rows and 2 columns of texels from the current face. Samples 02 to 15 fetch from the top face. Samples 20 to 51 fetch from the left face. Samples 00, 01, 10 and 11 can sample from texels on either side of the edge.

Convolution - projection

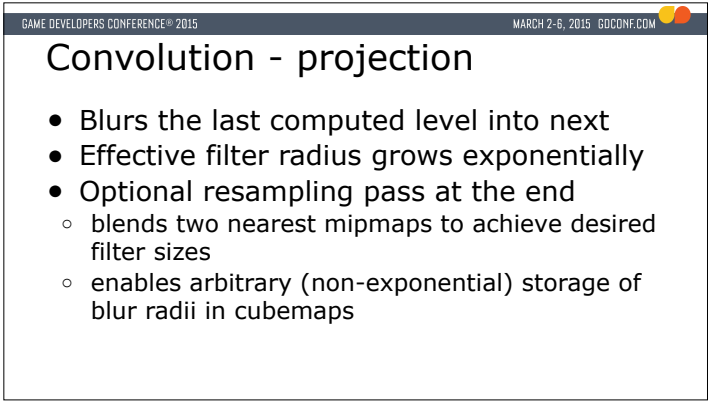
- 6x6 kernel with varying coefficients
  - derivative of uv coords projected on a sphere

$$sqrt(1 + dot(uv, uv))^3, uv \in [-1, 1]$$

- Equiv. to 2x2 filter at the center
- Widens towards the edges

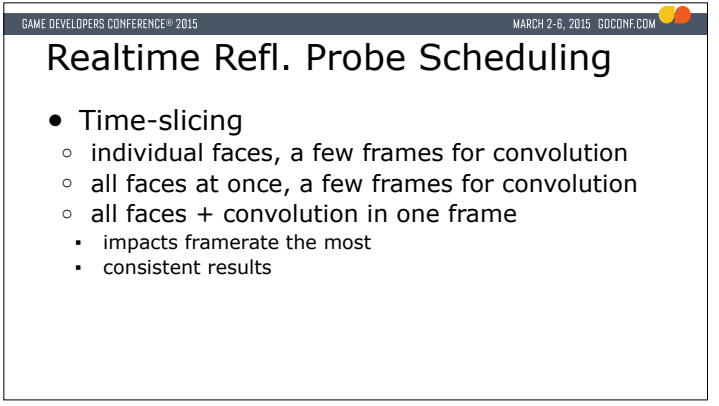
The filter is implemented as a fixed 6x6 kernel with coefficients varying across the face based on the derivative of the uvw coordinate with respect to the spherical coordinate.That gives the scaling factor of sqrt(1 + dot(uv, uv))^3, with uvs in the [-1,1] range.

At the center of the face, the filter is equivalent to a 2x2 box filter (with the remaining coefficients close to zero), giving the same result as a standard 2D mipmap filter. As we move towards edges and corners, the filter widens and more texels are blurred together.



The filtering proceeds by blurring the last computed level into the next one. The results end up in a mip chain, with the effective filter radius growing exponentially across levels.

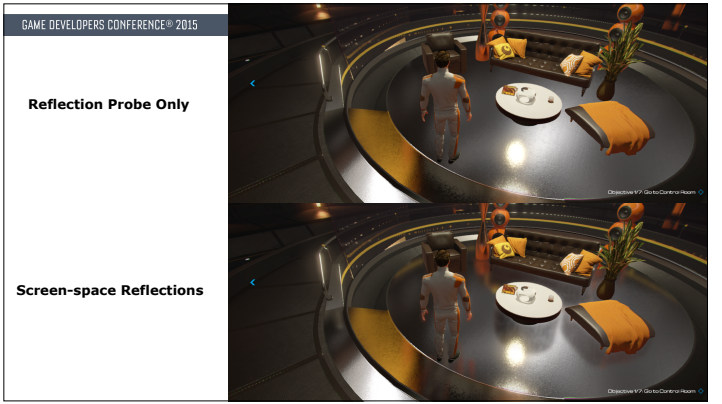
The current reflection probe implementation in Unity is using sub-exponential filter radii distribution which assigns more mipmaps to large blurs - favouring rough surfaces in PBS.



Reflection probe rendering and convolution needs to be scheduled sensibly. The user has control over that via scripting.

For each probe he can schedule its update and has a choice of rendering individual faces or all faces at once, and in both cases spreading convolution over a few more frames.
Alternatively the probe can be rendered and convolved in one frame, giving consistent results, but impacting the frame time the most.

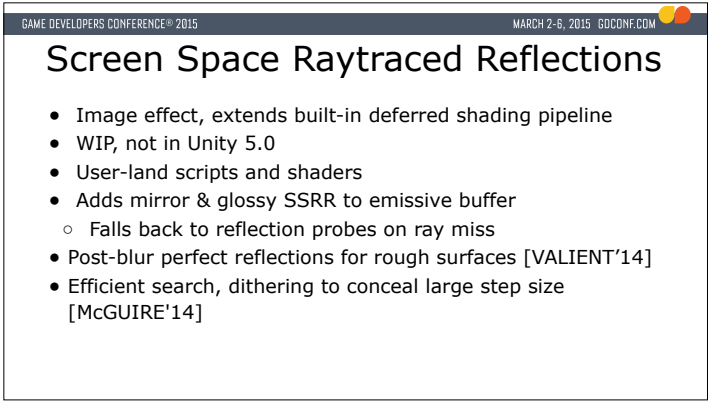
Continuing the topic of reflections, let’s talk about screen space reflections.



The problem with reflection probes is that the environment is captured at very few points in space and they generally assume that what they capture is either infinitely far away or at the faces of a box defined by an artist. Everything that doesn't fall into one of those two categories will show some serious issues.

The most visible issues show up in places where the object showing the reflection intersects with objects it's reflecting.

Note: this scene is a modified version of the "Listening Room" from our Unity Labs demo.



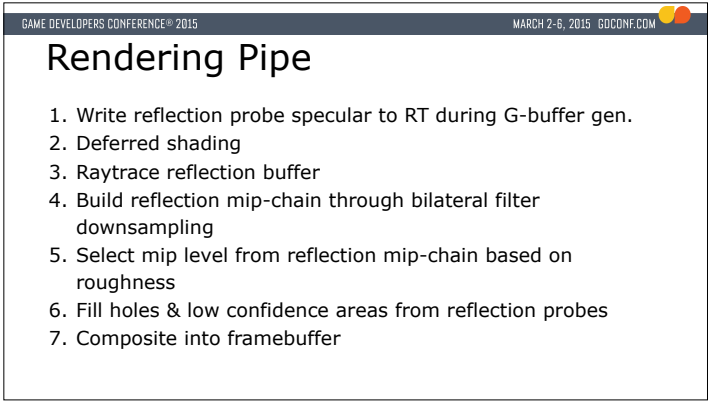
To alleviate the issues with reflection probes screen space reflections can be used. Our implementation is an image effect that extends the built-in deferred shading pipeline.

Please note that this is not available in Unity 5.0, but will however be shipped as soon as possible.

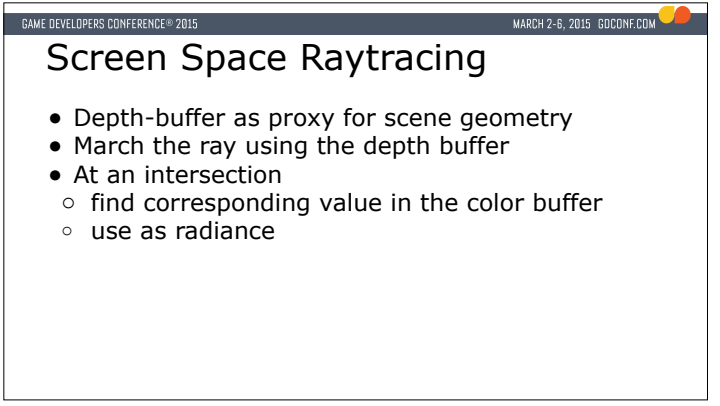
It's done in user-land as scripts and shaders. It allows for mirror and glossy SSRR and adds them to an emissive buffer.

Note: this is not slated for a particular release yet.

[VALIENT'14] Michal Valient, "Reflections and Volumetrics of Killzone Shadow Fall", SIGGRAPH 2014
[McGUIRE'14] Morgan McGuire, Michael Mara, "Efficient GPU Screen-Space Ray Tracing", JCGT 2014

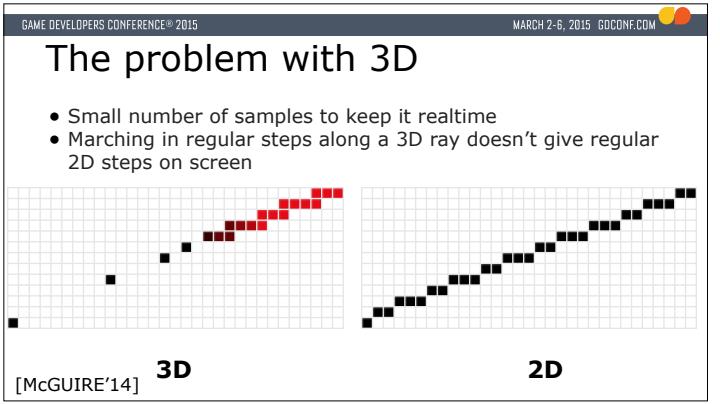


The rendering pipeline for this technique works as follows.



The depth texture is used as a proxy for scene geometry. Raytracing is performed by marching the ray using the depth buffer.

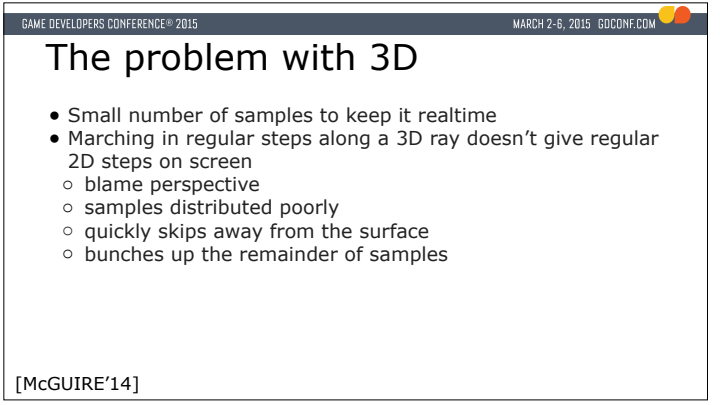
At an intersection a corresponding value in the color buffer is fetched and is used as the radiance in the glossy GI calculation.



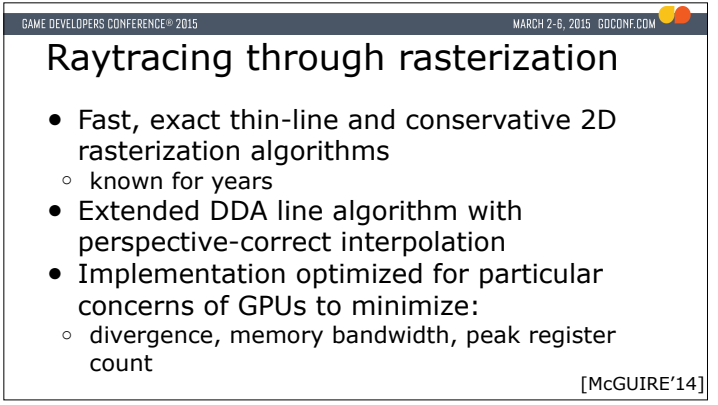
Ray marching is usually limited to a small number of iterations to ensure real-time performance.

Due to perspective, marching in regular steps along a 3D ray doesn't generally give regular 2D steps across the screen.

Image from [McGUIRE'14].



This means that the limited number of samples is distributed poorly. Often the ray will skip away from the surface too quickly, missing nearby objects. The remainder of samples bunches up on few pixels, limiting how far away reflecting objects can be and wasting computations on sampling same pixels repeatedly.

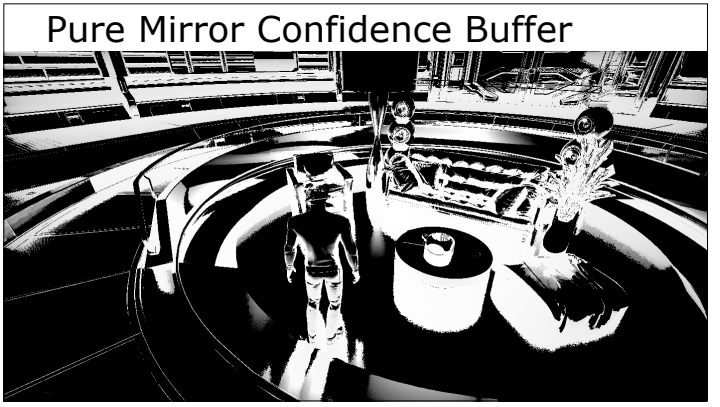


Thin-line 2D rasterization algorithms have been known for years. Morgan McGuire and Michael Mara extended the DDA (digital differential analyzer) line algorithm with perspective-correct interpolation.

The implementation is optimized for particular concerns of GPUs to minimize divergence, memory bandwidth and the peak register count.

Pure Mirror Reflection Buffer

Here is a pure mirror reflection buffer for the scene visible on the bottom left. For performance, since none of the objects in the scene are perfect mirrors, we step 3 pixels at a time in our ray trace.



A non-zero confidence value is assigned at every pixel where our screen-space raytrace hit something before termination.

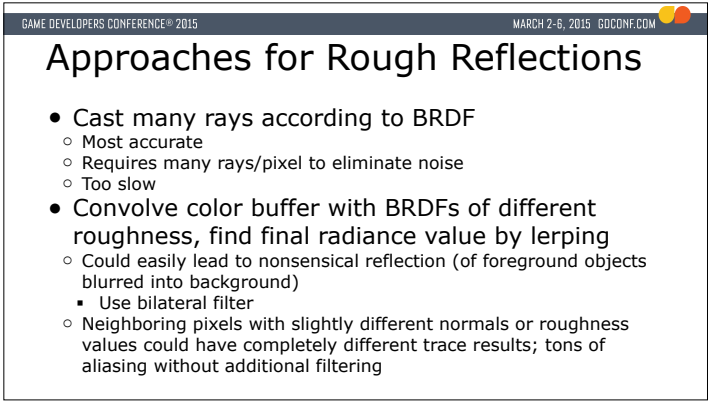
We lower the confidence of rays based on how many steps it took before intersection, as these intersections are prone to change rapidly with small camera movements and could lead to temporal flickering. This falloff was tuned to plausibly resemble the fresnel falloff.

We also lower the confidence of rays that terminate near the sides of the screen to prevent the reflections from popping in as objects enter the view frustum.

Rough Reflection

- Covered mirror reflections
- Most surfaces are not mirrors

So far we've covered mirror reflections, but most surfaces are not mirrors.



There are a few approaches for handling reflections on rough surfaces.

One can cast many rays according to BRDF. That is the most accurate, but requires many rays/pixel to eliminate noise and is too slow on all but the beefiest of machines.

One could also convolve color buffer with BRDFs of different roughness and find final radiance value by lerping.
It could easily lead to nonsensical reflection (of foreground objects blurred into background). This could be mitigated by a bilateral filter.
Neighboring pixels with slightly different normals or roughness values could have completely different trace results though -- tons of aliasing without additional filtering

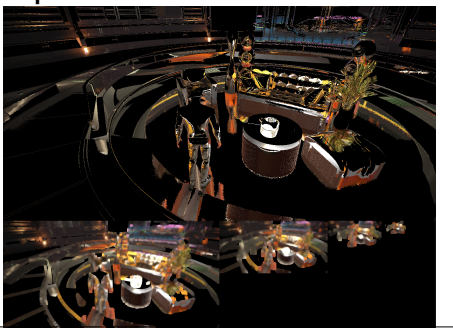
Chosen approach for Rough Reflections

- Adaptation of [Valient'14] (Killzone Shadow Fall)
- Store pure mirror reflection values in a buffer
- Build a mip-chain from this buffer
  - use a filter to approximate convolution with BRDF
- Composite
  - choose mip-level based on roughness

The chosen approach for rough relfections is an adaptation of Michal Valient's work on Killzone Shadow Fall.
First pure mirror reflection values are stored in a buffer.
Then a mip-chain is built from this buffer. A filter is used to approximate convolution with BRDF of increasing roughness as we construct smaller mip levels.
The proper mip-level is chosen based on roughness in the final composite pass.

[VALIENT'14] Michal Valient, "Reflections and Volumetrics of Killzone Shadow Fall", SIGGRAPH 2014

Mipmap Generation



The mip-chain is generated using a 7-tap separable bilateral gaussian blur during downsampling. Bilateral weights based on roughness (could also use normals/depth if artifacts arise). Smaller mip levels correspond to less smoothness (higher roughness).

Gaussian was selected as it approximates the BRDF best.

7-tap to match the reflection probes convolution (for a given roughness value sample the same mip level as in reflection probes), empirically tuned.

Confidence Values

A confidence value is stored per-pixel, based on whether the ray hit a surface in the depth buffer and how far it traveled before hitting anything. These are scalar values, and they are stored in the alpha channel of the reflection buffer. It gets mipped with the color values.

In a compositing pass the mip level to pick from is selected per-pixel based on roughness. This allows for supporting arbitrary roughness variations. This pass is a single trilinear texture read per pixel.

The confidence values stored in the alpha channel are also composited based on roughness. We use the confidence value to lerp between the fallback solution (reflection probes) and the screen-space raytracing.

No reflection probe fallback

Result with SSRR, but without using the fallback reflection probe buffer.

The glossy GI contribution from reflection probes.

Final image without SSRR.

Final Results

Bringing it all together. It grounds objects (including dynamic objects like our scientist) quite effectively and makes the scene more plausible.

Quarter-Resolution SSRR

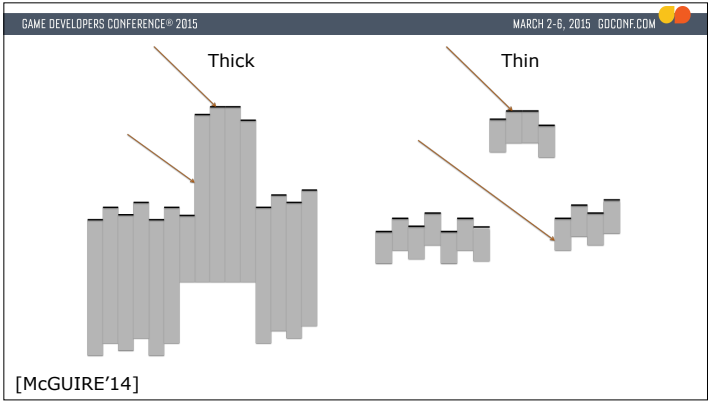
SSRR calculated at quarter resolution gives a proportionate speedup. It scales down to low-powered platforms with a disproportionately small loss in quality.

Thickness of the Depth Buffer

- All the geometry info from depth buffer
- Each texel of the depth buffer
  - infinitely thin sheet?
  - infinitely thick block?
  - something in-between?

All the information on scene geometry comes from the depth buffer.

Do we treat each texel of the depth buffer as:
An infinitely thin sheet?
An infinitely thick block?
Something in-between?



Many previous SSRR techniques treat the depth buffer as infinitely thick.

Infinitely Thick Depth Buffer

Results obtained when treating the depth buffer as infinitely thick.

Infinitely Thick Depth Buffer

This leads to very visible artifacts when there is any significant depth complexity in the scene.
Notice the long orange colored reflections along character’s body. Those are his infinitely deep hands.

Infinitely Thin Depth Buffer

Treating it as infinitely thin removes the “extraneous reflection” artifacts from the infinite thickness approach.

Infinitely Thin Depth Buffer

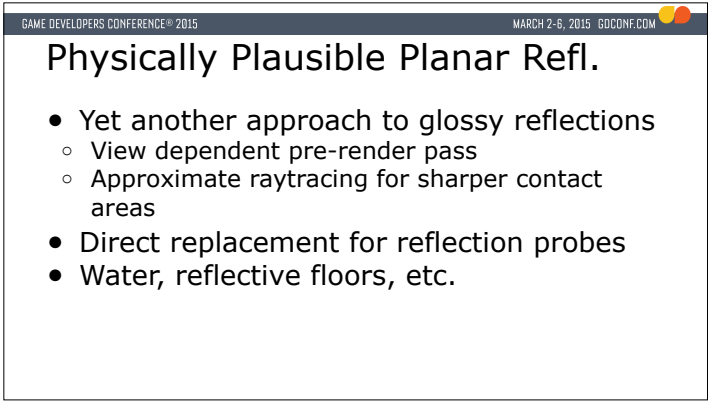
And replaces it with missing reflections, as one can see in the (lack of) reflection of the couch.

0.5m Thick Depth Buffer

Setting the thickness of the depth buffer to the approximate thickness of objects in the scene leads to more plausible results, with fewer artifacts. It is still imperfect, since the only information we really have about objects is in the depth buffer. Having multiple layers of information could improve quality, at an obvious jump in cost.

Physically Plausible Planar Refl.

The special case of planar reflective surfaces can be handled separately.

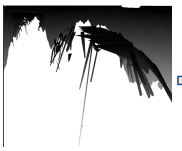


This approach uses a view dependent pre-render pass and approximates raytracing for sharper contact areas.

It’s a direct replacement for reflection probes in our physically based Standard shader.
It’s well suited for water, reflective floors, etc.

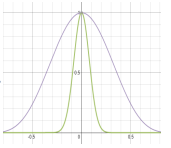
Reflection Convolution

Depth

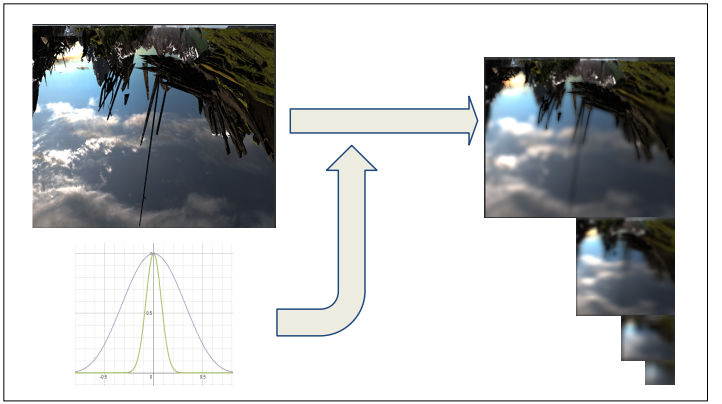


Contact / Falloff

Tweaked kernel



The depth buffer is used during convolution to preserve contact areas.



Similarly to previous techniques a mip-chain is created.

Regular Render with Local Reflection Probe

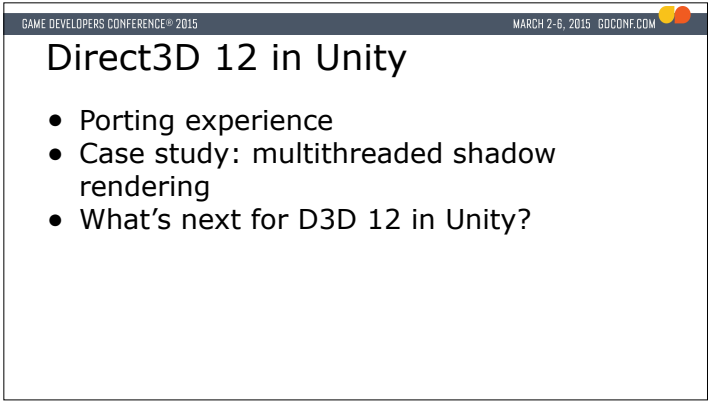
This screenshot shows water rendered using a reflection probe. The resolution is not so great. The reflection is uniform no matter the distance from the water to the object being reflected.

Regular Render with Planar Reflection

Here the water is rendered using planar reflections. The reflection gets sharper where the distance from the water to objects being reflected is low and the reflections actually align nicely. Success!

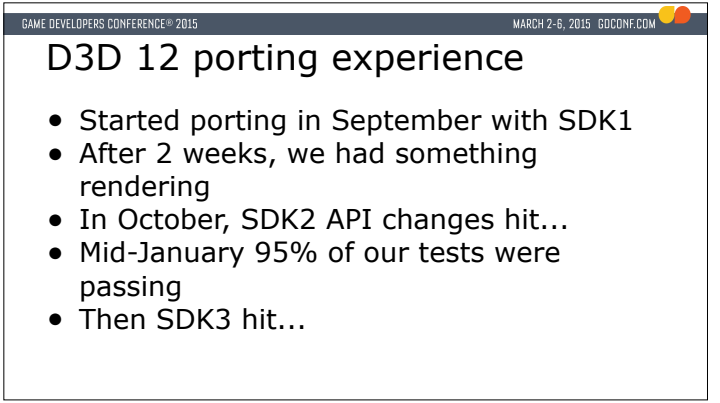
Profit

- Pros
  - Low complexity
  - Easy to setup and integrate
  - Fast convolution
    - 1024^2 <1ms GT750M
    - 512^2 <0.3ms GT750M
- Cons
  - Only works for planar-ish receivers
  - Rendering cost grows with complexity of scene



[Kasper takes over]

Now it is time to talk about Direct3D 12 in Unity.
We have worked closely with Microsoft to bring D3D12 support to Unity. We started porting in September and today I'll first talk about our progress.
Then I will cover the gains we have gotten by moving our shadow map rendering away from the main thread and onto worker threads.
Finally, I will look a bit ahead and talk about our future plans for D3D12.



We started in September 2014. With a Windows10 build with SDK 1 from March 2014 and early IHV drivers.
It took 2 weeks to learn the API and get something rendering in Unity. Luckily our entire shader pipeline just worked with shader model 5.0, which saved us a lot of time, because we were generating working bytecode already.
In October, SDK version 2 hit, it took two weeks to fix the fallout and refactor code.
From October to December we spent the time getting the tests in our internal graphics tests framework green. The majority of time spent here was because of driver issues.
We would probably have moved faster if we had a functional graphics debugger available.
By mid-January - 95% of all of tests worked and then SDK3 hit… the new heaps concept destroyed performance, because every resource would create a heap!
Our dynamic VBO implementation completely rewritten for performance. We now allocate memory in 1 mb chunks. Dynamic VBO grabs from that buffer using a view.
This is where we are today.

D3D 12 optimization case study

- Multi-threading shadow map rendering
- Move work away from main thread
- Generate d3d cmd lists for each of the shadow maps on their own worker threads
- Cmd lists executed in parallel with the main scene cmd list building

We chose the multithreading of shadow maps as our first case study, because this is something that can be applied to your existing engine and you can get a nice performance boost from multithreading and using D3D12.
The main goal is to allow the rendering command lists be created on multiple threads to move work away from the main thread.
D3D12 is forcing us to do changes that will give us nice performance boosts later. Common (Unity code) has to generate intermediate commands that will be later executed by a worker device. In D3D12 however, we can skip the intermediate command list and generate D3D12 command lists directly, thereby removing an intermediate step.

Note: the main thread still builds the cmd lists to render the main scene and executes them all at flip time. A better way to think about it is: the commands are built asynchronously, but they have to be executed in order. So they're queued and then executed like this: foreach(asynccmd) { execute; } maincmd->execute();

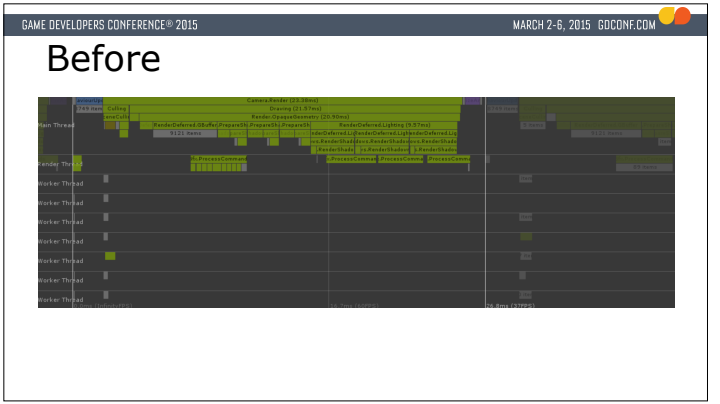
Why shadow maps?

- Rendered before the main scene
- Simple render loop
- Extracting receivers & casters is quite CPU intensive
- The shadow jobs don't require waiting until ID3D12CommandList needs to be executed

Shadow maps are an ideal candidate when it comes to multithreading the rendering pipeline:

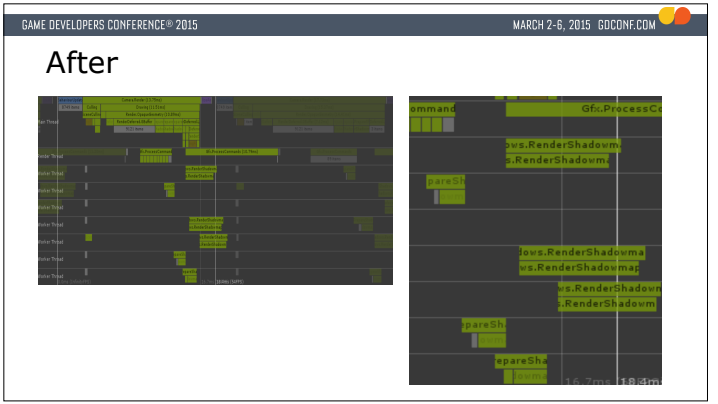
1) They are rendering before the main scene, which means commands can **be generated in parallel with the main scene.**
2) Rendering shadow maps is pretty much *a) SetRenderTarget(shadowMap); b) render everything;* with fewer state changes than normal scene rendering.
3) The extraction of casters & receivers is a very good thing to *not* do on the main thread.
4) That means that it can run in parallel with the whole rendering, and only needing the wait at the end of the frame.

On present, do shadow cmd chains, then main scene...



This test scene consists of 7000 objects that are lit with 3 shadowcasting directional lights. As you can see from the profiler capture, the RenderShadowMaps samples are inlined on the main thread with the rest of the scene. The CPU time spent rendering one frame is 23.38ms (Camera.Render sample)

Before we only had one function: RenderShadowMaps. In that function however there are a few things that can run only on the main thread and those have to be kept on the main thread. So we had to first split the code into what can be moved out of the main thread and what needed to stay on the main thread.



By moving the each of the shadow maps to their own thread the total frame time has dropped to 13.75ms which is a win of 10ms.
To achieve this, the RenderShadowMaps function has been split into PrepareShadowMaps and RenderShadowMaps.

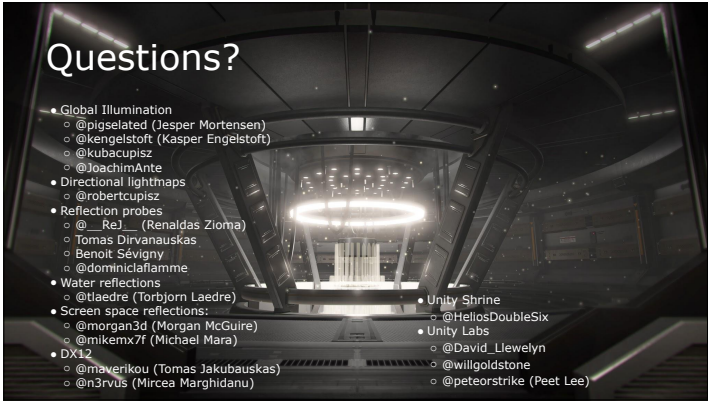
- PrepareShadowMaps (runs on the main thread): Responsible for running the main thread part of the shadow map rendering. It performs the following tasks:
    - Calculates memory requirements for the shadow map temp render buffer.
    - Spawns a job for calculating bounds and shadow cascade parameters. It also performs culling against the cascades.
    - Finally returns a partially preheated header (filled with the shadow matrices and cascade info) that will be used later on by the RenderShadowMaps job.
- RenderShadowMaps (runs on main thread)
    - Creates the shadow map render texture.
    - Launches the ShadowMapJob async job. This will create a native D3D12 ID3D12CommandList which will get executed at PresentFrame time.

The ShadowMapJobs are waited for (if needed) and executed at PresentFrame time.

Future D3D 12 work

- Prerecorded command bundles
  - One bundle per material pass
  - Bundles for standard operations
    - mipmap generation
- Use shader model 5.1 features

Moving on from where we are now, first we aim to be feature complete, then start optimizing.
Instead of executing all the code for setting state every frame, state changes can be prerecorded into a bundle. Drawcalls are very cheap when reissuing command bundles, so we should be able to issue many more. We can generate these bundles for standard operations.
We will also investigate deferred rendering with shader model 5.1. It can be improved because of the new resource binding concept where we can bind 1000s of textures and the shader can select which ones to use.


Questions?
• Global Illumination
○ @pigselated (Jesper Mortensen)
○ @kengelstoft (Kasper Engelstoft)
○ @kubacupisz
○ @JoachimAnte
• Directional lightmaps
○ @robertcupisz
• Reflection probes
○ @__ReJ__ (Renaldas Zioma)
○ Tomas Dirvanauskas
○ Benoit Sévigny
○ @dominiclaflamme
• Water reflections
○ @tlaedre (Torbjorn Laedre)
• Screen space reflections:
○ @morgan3d (Morgan McGuire)
○ @mikemx7f (Michael Mara)
• DX12
○ @maverikou (Tomas Jakubauskas)
○ @n3rvus (Mircea Marghidanu)
• Unity Shrine
○ @HeliosDoubleSix
• Unity Labs
○ @David_Llewelyn
○ @willgoldstone
○ @peteorstrike (Peet Lee)

References

[CUPISZ'12] Robert Cupisz, "Light Probe Interpolation Using Tetrahedral Tessellations", GDC 2012

[McGUIRE'14] Morgan McGuire, Michael Mara, "Efficient GPU Screen-Space Ray Tracing", JCGT 2014

[VALIENT'14] Michal Valient, "Reflections and Volumetrics of Killzone Shadow Fall", SIGGRAPH 2014

[BEHC'10] behc, "Box Projected Cubemap Environment Mapping"

[CUPISZ'12] http://gdcvault.com/play/1015312/Light-Probe-Interpolation-Using-Tetrahedral
[McGUIRE'14] http://jcgt.org/published/0003/04/04/, optimized GLSL implementation available
[BEHC'10] http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping

Appendix

Convolution - folding

```
// Example: for the face pointing in the positive z direction:
// uvw of desired texel: [1.1, 1, 1]
// excess of uv: [.1, 0]
// folded uvw: [1, 1, .9]

void fold (inout float3 c, float3 face)
{
   float3 a = max(c - one, zero);
   float3 b = max(-one - c, zero);

   c = min(max(c, -one), one);

   float m = dot(a + b, one);

   c -= m*face;
}
```

Physically Plausible Planar Refl.

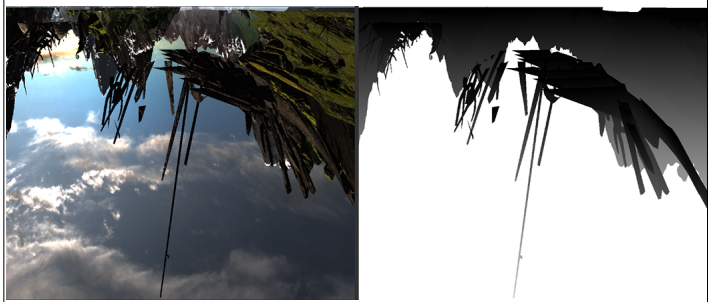
Rendering Recipe

1. Render perfect-mirror plane reflections
2. Realtime convolution to mip-mapped texture
3. Per-frame setup to targeted materials
4. Regular scene rendering
5. Profit

Reflection Rendering

- Same old same old (code in Standard Assets' Water)
  - Choose a reflection plane
  - Reflect camera about plane
  - Calculate oblique frustum
  - Invert back-face culling
  - Render to mirror texture
- Also render depth texture
- Conservative culling masks and far planes to manage reflection rendering cost

Reflection Rendering



Reflection Convolution

- Standard shader expects to map surface roughness to texture mip-levels
- Need to build mip-chain with BRDF matching increasing roughness
- Pixel shader using previous mip-level and generated depth texture as input

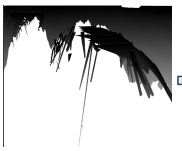
Reflection Convolution

- Convolve to a quarter size texture
  - Usually no need for perfect mirror reflections (yes, even for water that's often fine)
- Convolution kernel has some "special sauce"
  - Separable blur with cosine power lobe and mip drop
  - Uses reflection depth distance to approximate tracing reflection rays
  - Convolution kernel's width tweaked based on depth contact factor

(Cosine power lobe and mip power drop roughly similar to PMREM)
Convolution is fishy in the land of energy conservation, but turns out visually pleasing and reasonably close to reflection probe results.

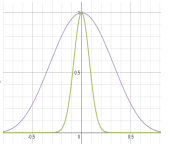
Reflection Convolution

Depth

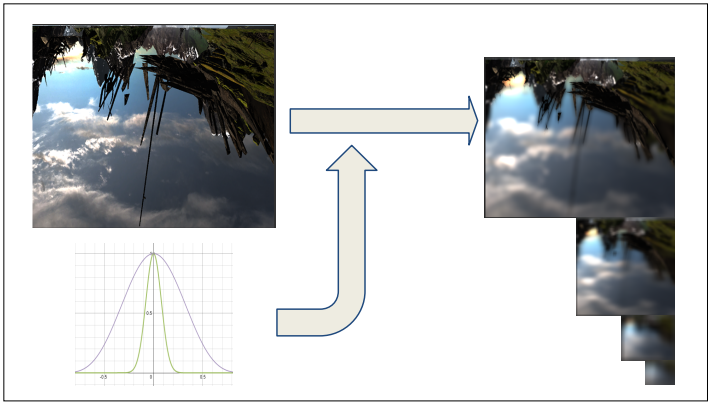


Contact / Falloff

Tweaked kernel



The depth buffer is used during convolution to preserve contact areas.



Note: this image should really be the tweaked kernel being used in each subsequent mip generation step, but Google Arrows weren't cooperating.

Reflection Convolution

- Reflection probes vs planar reflections
  - BRDFs match up reasonably well
  - Very smooth surfaces more blurry than probes as we discard perfect reflection texture

Smoothness 0.40, 0.60 , 0.80, 0.90 and 100.s

Per-frame Setup

- Choice of granularity
  - Scene Global (Shader.EnableKeyword / Set*())
  - Per Material (mat.EnableKeyword / Set*()
- Requires shader support
  - Extremely easy to extend Standard shader in Forward Rendering path
  - Slightly more involved in Deferred, but still classified as easy

Standard Shader Integration

```
// PR_StandardSpecular.shader:
#pragma multi_compile _ PLANE_REFLECTION
#include "PR_UnityStandardCore.cginc"

// PR_UnityStandardCore.cginc:
uniform float   _PlaneReflectionLodSteps, _PlaneReflectionBumpScale, _PlaneReflectionBumpClamp;
uniform sampler2D   _PlaneReflection;

#ifdef PLANE_REFLECTION
        float mip = pow(1.f - s.oneMinusRoughness, 3.f/4.f) * _PlaneReflectionLodSteps;
        float4 uv = float4(i.pos.xy * _ScreenParamsRcp.xy, 0.f, mip);

        // Optional small offset based on bump scale since we only have a plane to sample as opposed to a cube.
        uv.xy += clamp(s.normalWorld.xz * _PlaneReflectionBumpScale, -_PlaneReflectionBumpClamp,
          _PlaneReflectionBumpClamp);

        gi.indirect.specular = tex2Dlod(_PlaneReflection, uv) * atten;
#endif
```