

# **Unity 5 Graphics Smörgåsbord**

Aras Pranckevičius  
Rendering Plumber



A talk about all kinds of graphics things that we added in Unity 5.0.

# Overview

- Unity 5 adds a lot of graphics goodies!
- Let's run through them real fast



Unity 5 is out now, and for a full list of new and improved things, see the release notes: <http://unity3d.com/unity/whats-new/unity-5.0>



## Physically Based Shading



One big area that ties into a lot of other systems is physically based shading (PBS).

Our primary goal was to make objects in Unity 5 look much better out of the box and to modernize the built-in shaders we ship with.

Unity's rendering engine is quite flexible , and a lot of these things are possible to implement manually in Unity 4 (see Alloy, Skyshop, ShaderForge etc.), but it requires a lot of graphics knowledge.

In 5, we're making that "what is built-in" much better — but of course it is still possible to customize if you want to, and in some ways the rendering pipeline is even more customizable now (more on that later).

## Motivation for PBS

- Predictable look under different lighting conditions
- Less guesswork, fewer ad-hoc parameters
- More interop with other tools (Toolbag, Substance, ...)
- Allow capture/scanning of real world data.



Physically Based Shading has been a hot topic recently — the movie industry has started using it a number of years ago, and games in the last couple of years have been moving to physically based shading too.

Primary motivation for that is actually saving on content creation time — by better modeling how light actually behaves, it is possible to make the content look good under different lighting conditions. In the past, you often had to tweak diffuse & specular maps if you wanted to place the same object in daylight & night time environments.

Many of PBS approaches settle down on just a handful of material parameters that are fairly intuitive. The same fairly well defined parameters also mean that content can be authored and previewed in more tools and still look sensible in all of them.

PBS also allows to develop hardware, software & workflows that capture and scan real world materials and bring them into your game (Quixel etc.).

# PBS: Energy Conservation

- Don't reflect more light than you receive
- Sharper reflections = stronger; blurrier reflections = dimmer
- Specular takes away from diffuse



In a nutshell, physically based shading often ends up being about several big things:

Energy conservation. This simply means that a surface should not reflect more light than what shines onto it. Some of the light is absorbed by the surface (and often converted into heat), and the rest is reflected - but never more than what was the incoming light. Unless the surface is a light bulb or other light emitter, that is.

For example, for a smooth surface the reflected specular highlight will be small but bright. For a rough surface, about the same amount of light will be reflected, but since it will be spread around much more, it will be dimmer.

Diffuse lighting is light that went a bit into the surface, bounced around and went back out. Specular reflection is light that just reflected straight from the surface. It's logical that if the light portion already reflected from the surface, did not simultaneously went into it. So "energy conservation" in the actual shader math often ends up just adjusting diffuse reflectivity based on how much specular light was reflected at that point.

## PBS: Specular & Fresnel

- Everything has specular
- Everything has Fresnel
  - “Surface becomes more reflective at grazing angles”

## PBS: Image Based Lighting

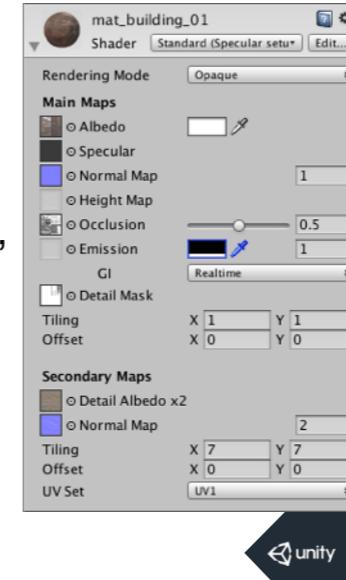
- Not *strictly* a requirement for PBS but...
- It's good for approximating the “entire lighting environment”



“Image Based Lighting” is a technique to approximate entire lighting environment using fairly simple & efficient means. Entire lighting environment is “how much light is coming from each of the possible directions”. Think outdoors — you don’t only get the sunlight; you also get quite a lot of illumination from all directions on the sky. More on image based lighting later, in reflection probes section.

# Standard Shader

- Built-in shader for most everyday materials
- No more hunting for “which shader to use?”
- Features driven by streamlined inspector



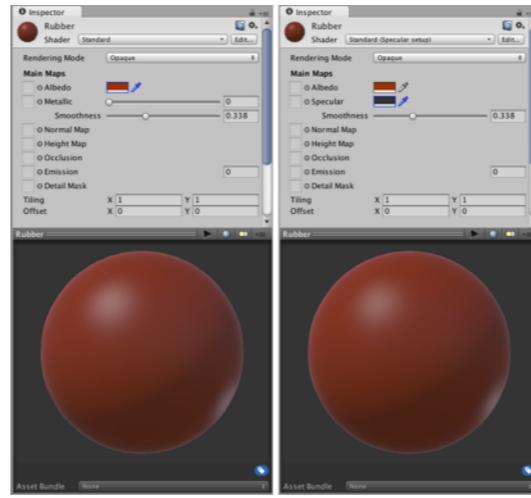
PBS in Unity is mostly achieved via the new built-in Standard shader. One of the things about it is all the physically based stuff, but another thing we wanted to solve is the “so which shader should I use now?” problem. Unity 4 had a lot of built-in shaders that were all very similar, just with slightly different features. So you could have “Transparent Reflective Normalmapped with Specular” shader and so on, and it was hard to find them (and a lot of the variants did not exist).

Now there's one (well ok two — Metallic & Specular workflow) shaders for most everyday needs, and their features are controlled by the inspector. For example, you can switch between different transparency modes with a drop down. Or if you don't use a normal map, then internally a different (and faster) shader variant will be used that skips all the normal mapping calculations.

More in Unity 5 documentation, but here's several several features of Standard shader that I wanted to point out.

# Metallic vs Specular

- Metallic:
  - Albedo (RGB)
  - Metallic (R)  
Smoothness (A)
- 0 metallic:
  - Diffuse=albedo
  - Specular 4% gray
- 1 metallic:
  - Diffuse=black
  - Specular=albedo

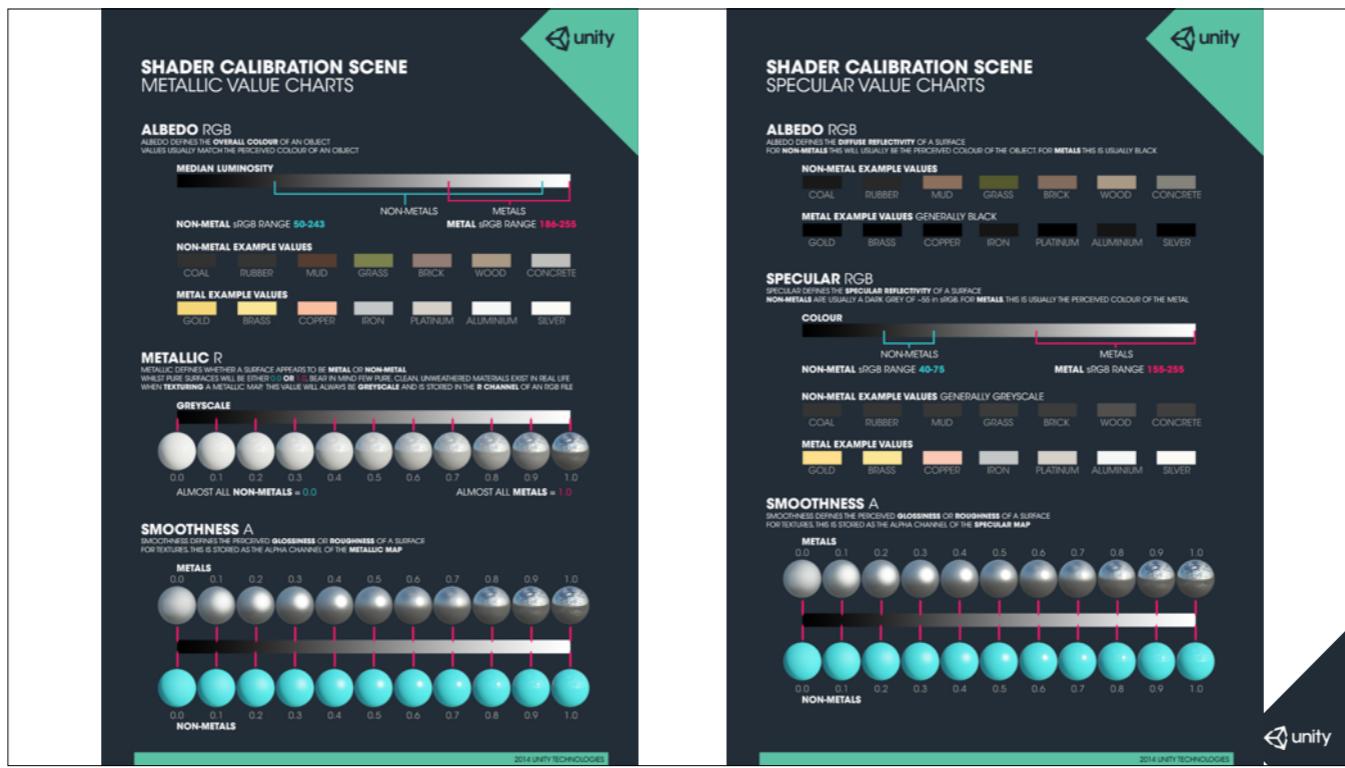


- Specular:
  - Diffuse (RGB)
  - Specular (RGB)  
Smoothness (A)

We do support both so called "Metallic workflow" and "Specular workflow". Their primary differences are in how and which textures you author for your materials — and which one you want to use largely depends on the other tools in your art pipeline and habits.

"Metallic workflow" is where there's one "base color" (Albedo) texture, and another texture that defined which parts of the object are made of metal (versus which parts are made of dielectric material). Then the shader computes diffuse & specular colors based on that. For metals, diffuse is black and specular is the texture color; whereas for non-metals diffuse is the texture color and specular is 4% gray (which matches most of non-metal materials). This workflow is fairly easy to understand, and can be more intuitive if the textures are painted by hand.

In "Specular workflow" the diffuse & specular colors are specified explicitly in two different textures. This allows slightly more material variation (e.g. having non-metals with different specular color than 4% gray — some gemstones have higher specular color). It also allows making materials that don't exist in real world, but might exist in some parallel universe :) — e.g. non-metals with colored specular or whatever. Separate diffuse & specular color maps are also what's often produced by various material scanning/acquisition tools.



We have nice charts of color values for common materials in the standard shader documentation.

# Transparency

- Physically plausible “Transparent” mode
  - Glass, water, ...
  - Specular & reflections visible, even at alpha=0!
- Gameplay/Effects oriented “Fade” mode
  - For fading out objects
  - Invisible at alpha=0



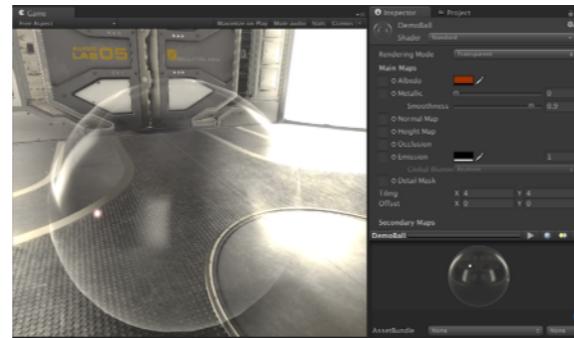
Another thing we have in the Standard shader is several transparency modes.

The “Transparent” mode is meant for physical objects that should look realistic, for example glass or water. Specular highlights and reflections are visible on these objects, even when alpha channel is set to zero — that is, these objects can never be fully transparent.

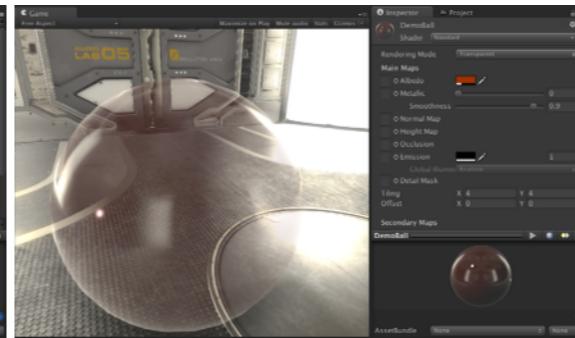
Another mode we have is called “Fade”, and it just fades out objects based on alpha channel. At alpha of zero the object is completely invisible. This does not match how surfaces like glass behave in the real world, but is useful for many gameplay related purposes, for example to fade out objects before making them disappear.

# Transparency: Transparent

Alpha=0



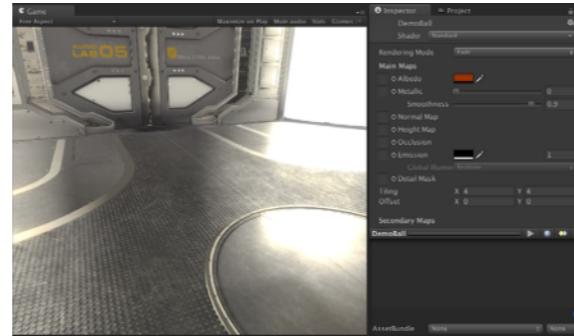
Alpha=0.3



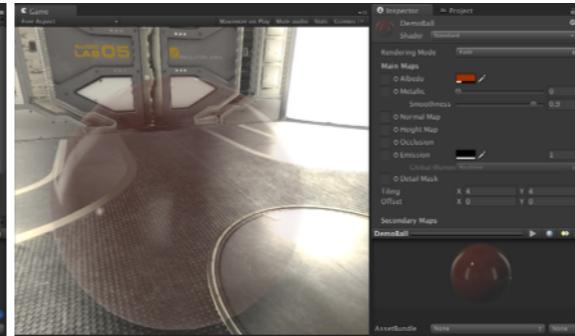
Here's a comparison: the physically plausible Transparent mode with zero and 30% alpha. Notice that reflections are visible even at zero alpha.

# Transparency: Fade

Alpha=0



Alpha=0.3



And here are the same objects using Fade transparency mode. Completely transparent at zero alpha, and proportionally dimmer specular highlights at non-zero alpha too.

# Math Details

- 3 BRDFs for different hardware/performance
- BRDF1: PC/consoles, SM3.0+
  - Derived from Disney; Torrance-Sparrow  $\mu$ facet model
  - Blinn-Phong NDF, Smith V, Schlick F
- BRDF2: Mobile ES3.0+
  - Modified “Minimalist Cook Torrance”
  - Approximate Kelemen & Szirmay-Kalos V, approx F
- BRDF3: Old hardware (DX9 SM2.0, DX11 9.x, ES2.0)
  - Blinn-Phong in RDF form, implicit V, no F
  - Math done via lookup texture



A moment about implementation details of Unity 5 Standard shader. The shader comes with three levels of fidelity for different hardware capability levels. We call them BRDF1, BRDF2 and BRDF3.

The “full fidelity” shader is used on any decent PCs and consoles, and is the primary shading model. Then we have BRDF2, which is a somewhat approximated math model, and is used on mobile platforms. It does less math calculations in the shader to run faster. And finally we have BRDF3, which is used on what you could call “old hardware” — DirectX 9 shader model 2 and OpenGL ES 2 platforms. This is even more approximated, and most of the math calculations in the shader are replaced by a lookup texture.

Everything in `UnityStandardBRDF.cginc` shader include file (check out Unity built-in shader source); but a summary here:

BRDF1: Main Physically Based BRDF. Derived from Disney work and based on Torrance-Sparrow micro-facet model.

$$\text{BRDF} = kD / \pi + kS * (D * V * F) / 4$$

$$I = \text{BRDF} * \text{NdotL}$$

NDF = Normalized BlinnPhong (optional GGX); Smith for Visibility Term; Schlick approximation for Fresnel

BRDF2: Based on “Minimalist Cook Torrance” (<http://www.thetenthplanet.de/archives/255>), slightly modified.

BlinnPhong as NDF

Modified Kelemen and Szirmay-Kalos for Visibility term

Fresnel approximated with  $1/LdotH$

BRDF3: Old school, not microfacet based Modified Normalized Blinn-Phong BRDF. Implementation uses LUT for saving ALUs.

Normalized BlinnPhong in RDF form, Implicit Visibility term, No Fresnel term.

## How much perf does it cost?

- Small scene to test on mobile & PC
- No shadows, no post-fx; just raw cost of shader itself
- Render into 1920x1080 everywhere



All that math stuff sounds complicated; doesn't that make this shader run slow?

I did a small test; render a scene with a bunch of objects using Standard shader and measure the frame time on the GPU. The scene has no shadows, no post processing, no other graphics effects on purpose — so that it only measures the cost of the shader itself. It also renders into a 1080p on all platforms/devices so that we get comparable timings.

## How much perf does it cost?



unity

Here's that small scene. Nothing fancy, just some objects with textures and whatnot, and a single directional light without shadows. Your particular scene might have different complexity, overdraw, polygon density etc. of course; and all that can affect performance!

## How much perf does it cost?

- Added ms/frame on GPU compared to Unity 4 BumpSpecular:
  - Fast PC (GeForce GTX 680): **+0.1ms**
  - Medium PC (Radeon HD 7770): **+0.4ms**
  - Laptop (GeForce GT 750M): **+1.4ms**
  - Laptop (Intel Iris Pro): **+3.4ms**
  - iPhone 6 (A8): **+8.3ms**
  - iPadAir (A7): **+12.3ms**
- Not a fair comparison! Now it samples more textures (occlusion, reflection etc.)
  - Also you probably don't want to render 1080p with PBS on older mobiles



Compared to Unity 4 “Bumped Specular” shader, additional GPU cost to render with a physically based shader is as follows. So the takeaway is that indeed, physically based shading does have an additional cost — there are more calculations done in the shaders, and often more textures being used. On PC, it does not really cost much.

On mobile, PBS has a non-negligible GPU cost (we'll keep doing performance improvements over Unity 5.x). So if you're targeting older mobile hardware and want to use physically based shader, you might want to render at lower resolution than what the device actually supports. There's nothing that's forcing you to use physically based shading though — you still have all the Unity 4 legacy shaders, as well as their approximated (but faster) mobile variants. Try it and see if the added graphical fidelity is worth the performance cost in your particular project.

# Surface Shaders

- Can use Standard BRDF in surface shaders too
  - `#pragma surface surf Standard`
  - `void surf (Input IN, inout SurfaceOutputStandard o) { ... }`
  - Similar for specular workflow
- Your own PBS lighting functions
  - “GI” part (lightmaps, ambient, indirect, reflections) is part of lighting function
  - Was hardcoded & non PBS in Unity 4



You can use the Standard shader lighting function in your own custom surface shaders. In fact if you just do a “Create new shader” in Unity 5, you will get a simple surface shader with a physically based lighting function there. It will support all the fancy Unity 5 stuff like realtime global illumination, directional lightmaps and reflection probes.

You can also write your own “physically based” lighting functions for surface shaders. The way “global illumination” data like ambient, lightmaps and reflections are applied is also much more flexible than Unity 4 now.



## Global Illumination & Lightmapping



Next topic: global illumination & light mapping.

# Global Illumination in Unity 5

- Precomputed realtime GI
  - Lightmaps + Light Probes
- Baked Static GI
  - Lightmaps + Light Probes
- Reflection Probes
  - Baked & Realtime



GI is covered by three big pieces in Unity 5.0: "realtime GI" (with precompute step), traditional "baked GI" (baked lightmaps), and reflection probes. Unity 4 only had the middle item, i.e. lightmaps.

# Precomputed Realtime GI

- Lightmaps and probes updated realtime
  - Dynamically change light sources, emissive materials & environment lighting
  - Static objects lit by realtime lightmaps
  - Dynamic objects lit by realtime light probes
    - 2nd order spherical harmonics



Realtime GI is powered by Geomerics Enlighten technology, and is meant for static scenes that can have dynamic lighting (moving lights; lights being created & removed at runtime etc.). Another powerful feature that is supported is emissive materials; the emissive parts can also be animated or colors/intensity changed at runtime and the lighting will update.

Enlighten does a precompute step for the geometry (only static geometry affects GI), and then at runtime updates "dynamic lightmaps" (that contain bounced lighting) and dynamic light probes.

## Precomputed Realtime GI

- Objects that affect lighting are static
- Moving objects pick up GI, but do not affect it
- Diffuse light transport only
  - Final reflective bounce: directionality and/or cubemaps
- Low frequency / low resolution



## Baked Static GI

- Ye olde lightmaps + light probes (3rd order SH)
- Baked direct + indirect + AO



Baked & static GI is just traditional lightmaps. In Unity 5 they are baked by Geomerics Enlighten too (Unity 4 was using Autodesk Beast for this step).

# Directional Lightmaps



Lightmaps can have three modes:

- Simple lightmaps which are just a texture.
- Directional lightmaps that allow to capture normal map details in them, but otherwise still only contain diffuse illumination.
- Directional specular lightmaps that also allow computing view-dependent specular highlights.

Each successive mode looks better, but is more costly in terms of memory usage (more lightmap data) and shader calculations that are done.

# Lightmapping Workflow

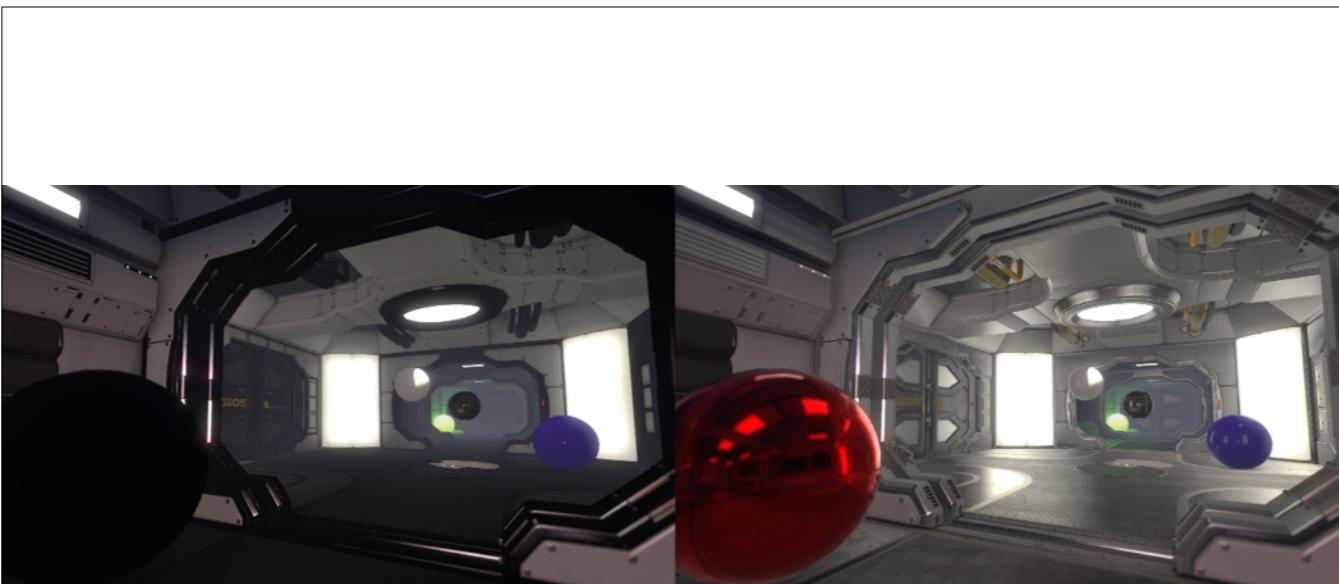
- Hash & cache
  - All inputs (meshes, transforms, lights, materials) hashed
  - Results of jobs cached
- Actual work done in async & parallel jobs
- Continuous baking mode
  - Monitors changes & automatically does stuff
- On-demand baking
  - Press a button
  - Lightmap snapshot saved & can be versioned



Internally, lightmapping computations use a hashing & cashing system. Chunks of world data and all settings are hashed, and results fetched from on-disk cache if they already exist there. If they do not, then background jobs & processes are spawned to compute them, and then they are cached to disk.

This allows to have “continuous baking” mode, where all the lightmapping & reflection probe jobs are computed automatically, whenever anything related in the scene changes.

When continuous baking is turned off, then workflow is more traditional — press a button to bake scene. Results of the bake are saved to an asset which can be versioned.



## Reflection Probes



Lightmaps mostly deal with diffuse light transport, but for reflective surfaces like metals, and especially smooth surfaces, it's important to capture highly detailed reflections. In Unity 5.0 this is achieved by "reflection probes".

## Reflection Probes

- Boxes placed in scene
- Each box captures a reflection cubemap
- Shader samples closest probe(s) for reflection
  - Mip levels for varying smoothness (convolved with Phong lobe)
  - Box projection
  - Blending between probes

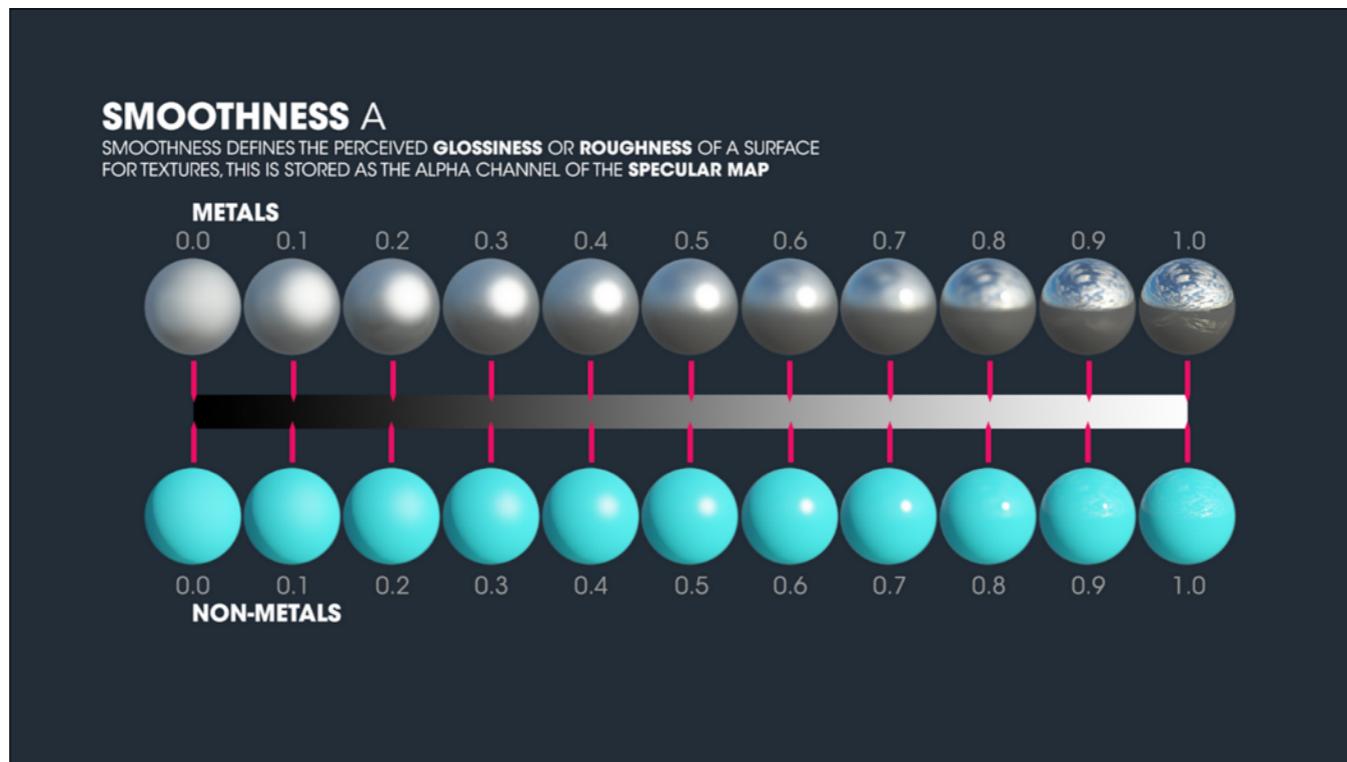


Reflection probes are just boxes placed in the scene; and each of them stores a cubemap of how scene looks like from the probe position.

Objects figure out which probe they fall into, and sample the probe's cubemap to compute reflection. Cubemap has specular convolution applied to the mip levels, so that surfaces of different smoothness can produce proper looking reflections.

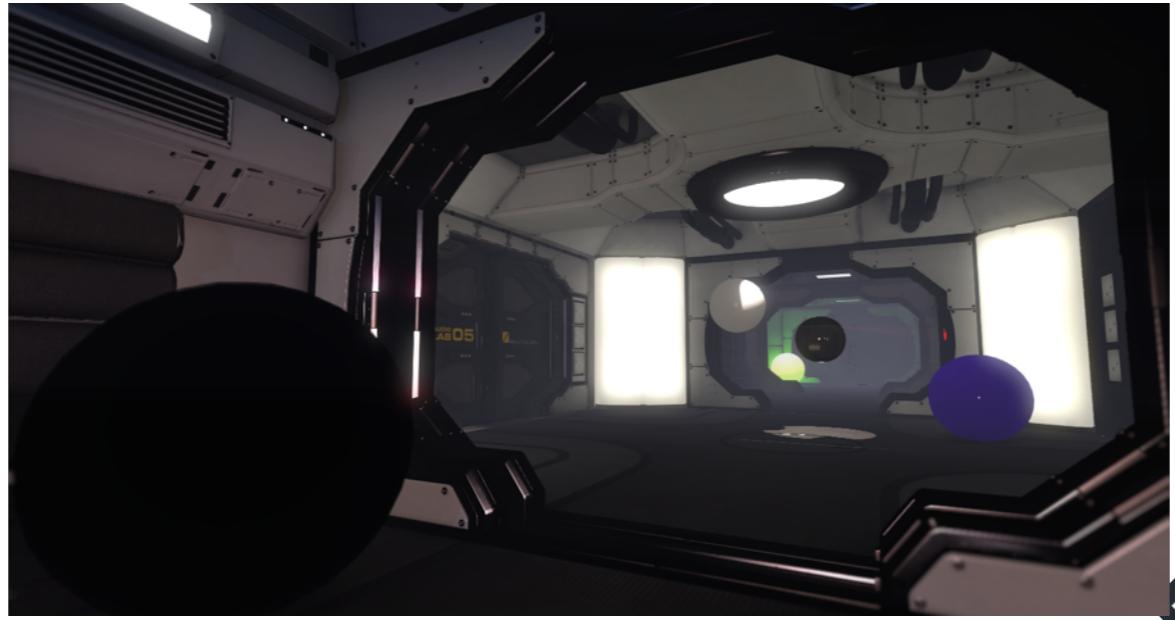
"Box projection" is a trick to adjust direction used to lookup into the cubemap based on where in the probe the object is located; this grounds the reflections to more proper positions.

Blending between two closest probes can also be done, so that while transitioning between probes there's no visual reflection pop.

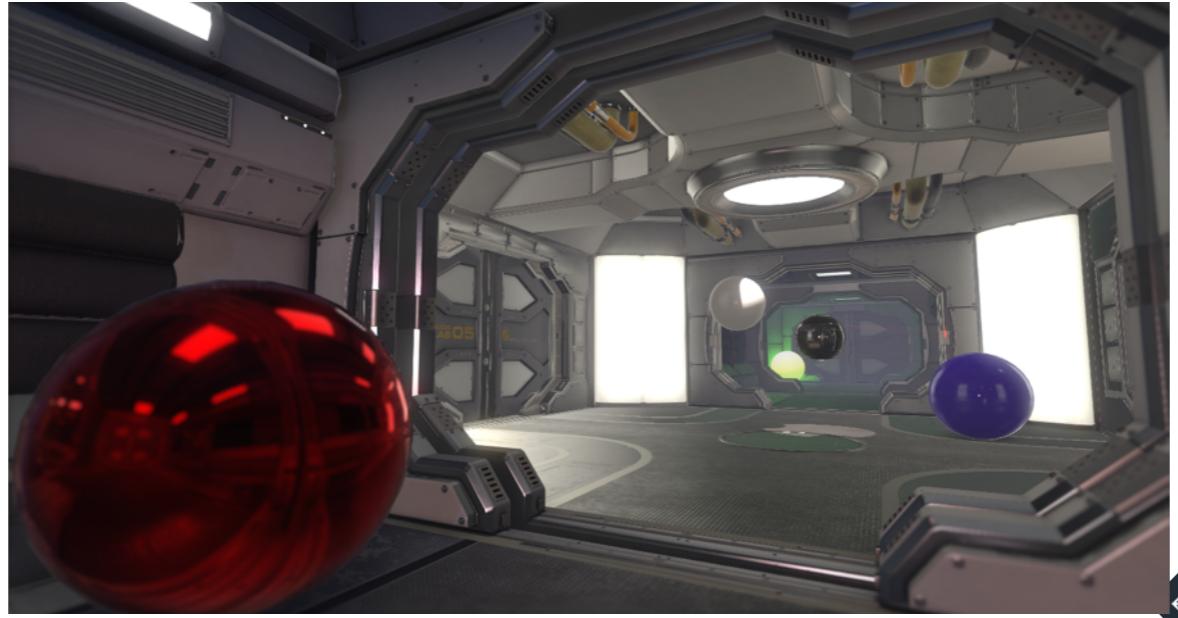


An illustration how surface smoothness affects reflections.

No Reflection Probes

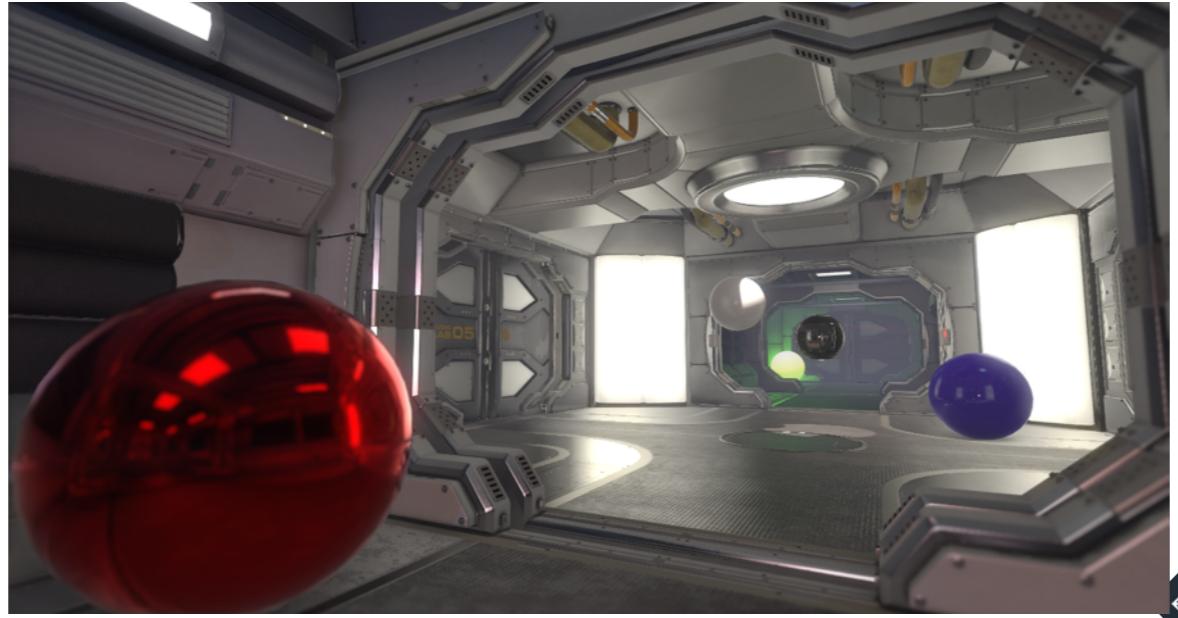


Reflection Probes (no box projection)



Reflection probes store the cubemap from the probe position, so the reflections are only really “correct” when objects are quite far away. When that is not the case, naive use of reflections can result in them looking out of place.

Reflection Probes with box projection



Box projection allows reflections to be in more proper place.

# Baked & Realtime Probes

- Baked
  - Static
  - High quality
- Realtime
  - Useful for dynamic scenes / hero chars
  - Render scene into cubemap at runtime
    - Can be spread over frames
  - Lower quality mip convolution (on the GPU)
    - Can be spread over frames



Baked reflection probes are for static use cases; they are baked during lightmap computations and the specular convolution done on them is high quality.

Realtime probes are rendered & updated in realtime, and are mostly useful for procedurally generated scenes, or when there is a need for moving objects to be reflected. They do have a runtime cost (scene needs to be rendered at runtime), and specular convolution done to their mip levels is quite approximate.

## Future: SSRR

- Screen Space Raytraced Reflections (not in 5.0)
- Current plan:
  - Deferred Shading only (need G-buffer info)
  - Split reflection probes buffer from emission/lightmaps buffer
  - Render probes similar to deferred lights
  - SSRR, fallback to probes where no information



A feature that did not make it to Unity 5.0 but we're working on right now is screen space raytraced reflections. It computes reflections by raytracing against frame's depth texture, and is completely dynamic (no baking step; can reflect everything). Very good at contact reflections. Also it has almost no CPU cost, however quite expensive on the GPU.

Current plan is to make it be deferred shading only (since it needs G-buffer data), and fallback to reflection probes in places of the image where SSRR did not produce valid/correct results (it's based on screenspace information only, so can't handle all cases).

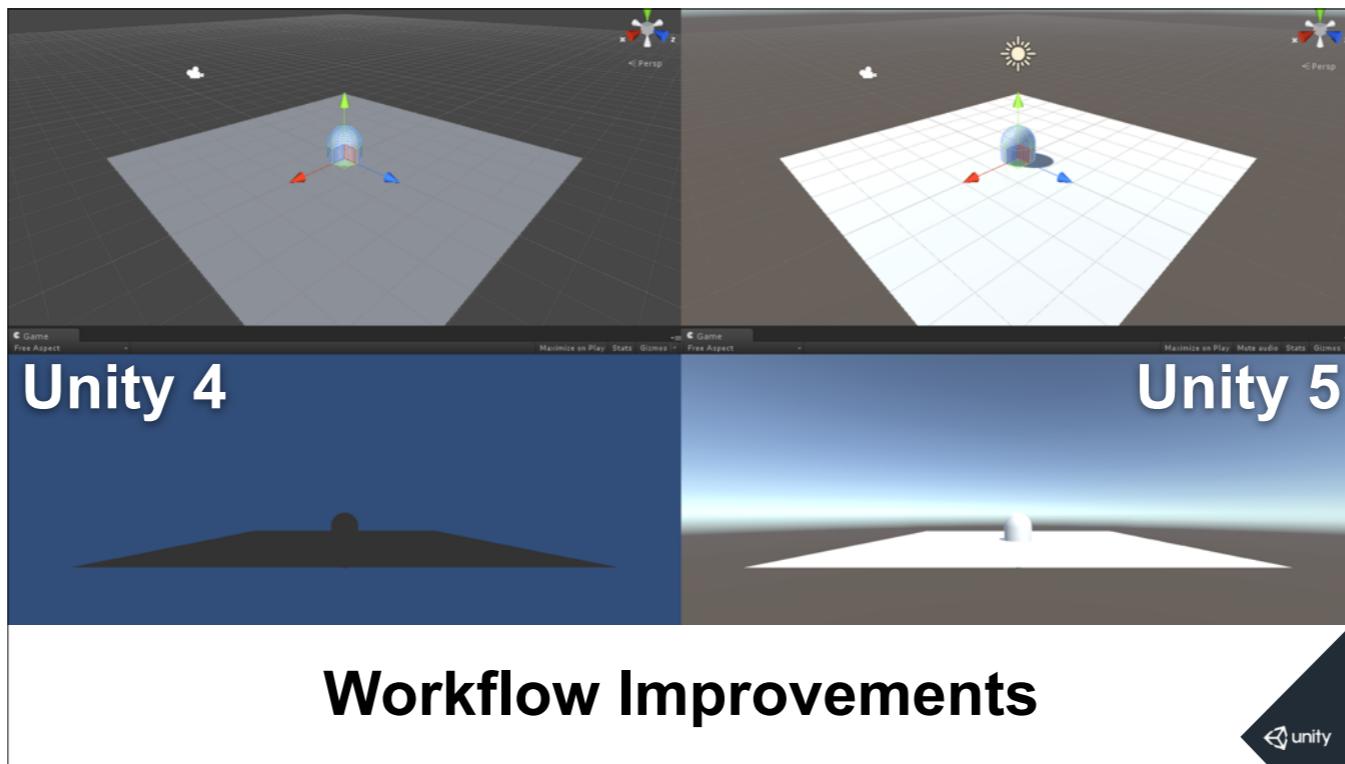
SSRR implementation is largely based on Morgan McGuire & Michael Mara "Efficient GPU Screen-Space Ray Tracing", <http://jcgtr.org/published/0003/04/04/>

No screen-space reflections



With screen-space reflections



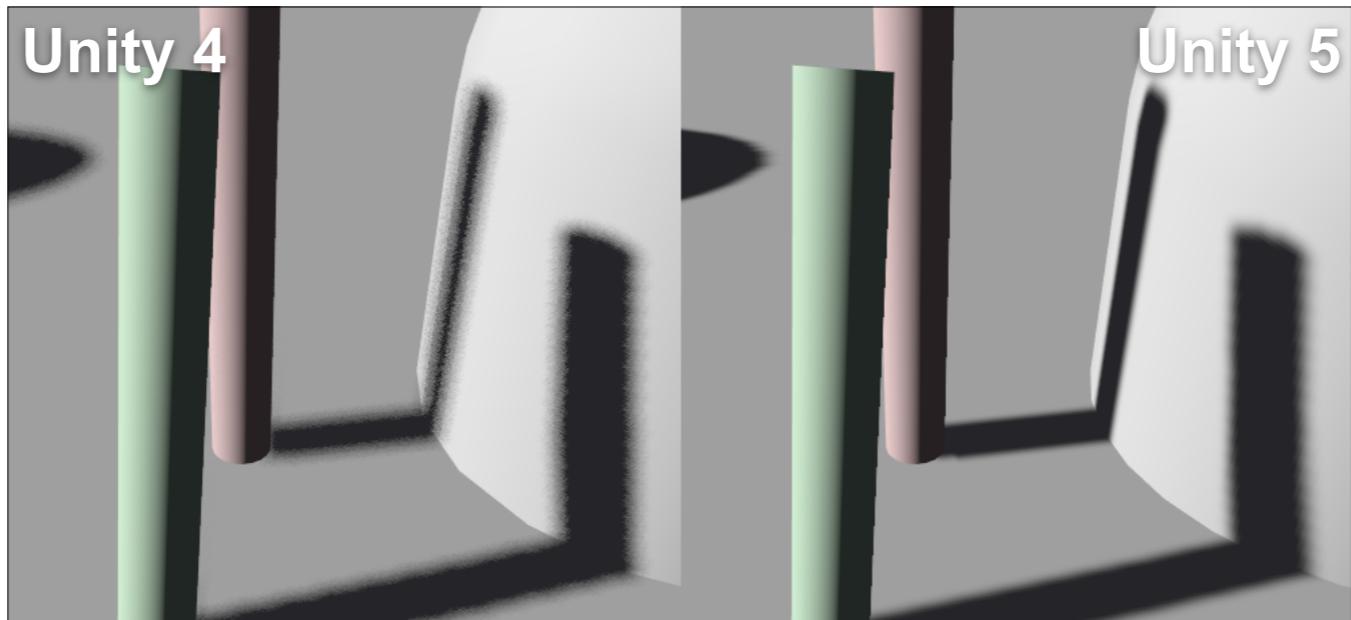


Besides headline graphics features, we did some smaller workflow improvements. For example, if you created a new scene with a plane & capsule in it, it was looking like the image on the left. In Unity 5.0, it looks like the image on the right.

# Little Things

- Procedural skybox shader
  - Skyboxes can be HDR too
  - HDR (.exr/.hdr) texture importing too
- New scene in 3D game has skybox & light
  - Ambient & reflection matching skybox
  - Scene view can be HDR too
- Cubemap workflow
  - Automatic detection of cross/latlong/strip/spheremap layouts
  - Texture compression





## Shadows



# Directional Light Shadows

- PCF 5x5 instead of screenspace blur
  - Looks better & runs faster
  - Normal offset shadows
- Cascade split ratios & visualization
- No more shadow collector pass
  - Shadow mask computed from depth texture



Directional light shadows got a significant upgrade in Unity 5. Most obvious one is that soft shadow filtering was changed; it was screenspace blur before and now we're using 5x5 PCF filtering (optimized implementation with just 9 shadowmap samples). In our tests it runs faster too (on both high-end and low-end GPUs).

Shadowmap cascade split ratios can be customized (in Quality Settings UI or from script), and scene view got a shadow cascade visualization mode.

"Shadow collector" shader pass is gone now; screenspace shadow mask is computed from the depth texture (which was what deferred lighting was using Unity 4 too; in Unity 5 both deferred & forward use the same approach).



Shadowmap cascade visualization mode in action. Purple, green, yellow, red correspond to 1st, 2nd, 3rd, 4th cascades.

## Other Shadow Stuffs

- Shadow casting modes
  - Off, On, Two Sided, Shadows Only
- Point/spot shadows no longer rendered 2x
  - If you have both deferred & forward objects
- Better bias for spot lights
- Better precision for point lights (RFloat format)
- Better point/spot filtering not in 5.0 yet, but high on the list





## Deferred Shading

Unity 5 also introduces deferred shading.

# Deferred Shading

- Unity 4 Deferred Lighting can't really do PBS
  - Not enough space in G-buffer to store information
  - Two geometry passes aren't nice either
- Unity 5 adds deferred shading
  - Multiple Render Targets to store G-buffer
  - One geometry pass



Primary motivation was that Unity 4 deferred lighting (a.k.a. "light pre-pass") can not do physically based shading. There's not enough information in the tiny G-buffer to do that (deferred lighting g-buffer only stores depth, normals & specular exponent). Two geometry rendering passes aren't nice either.

Unity 5 deferred shading is classic "multiple render targets G-buffer" approach.

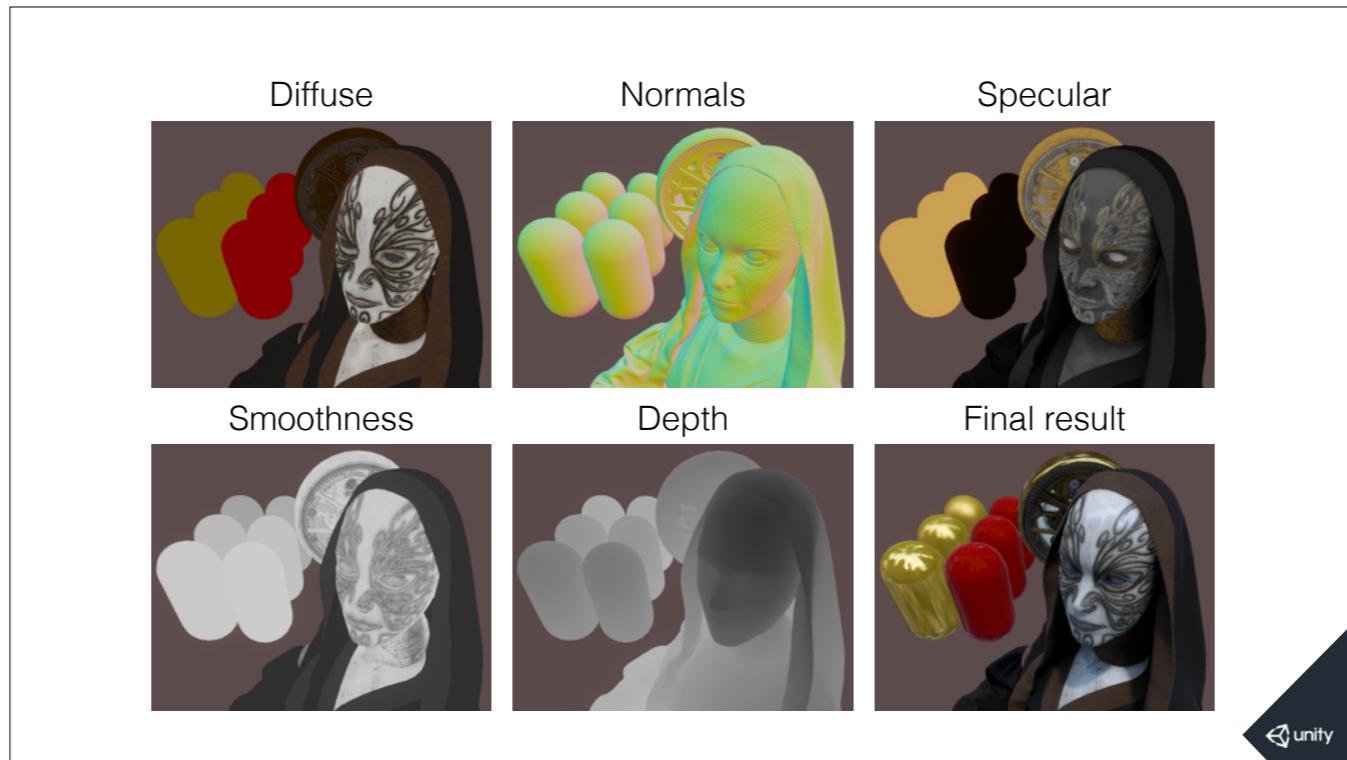
## Deferred G-buffer

- Four render targets, 160bpp (LDR) or 192bpp (HDR)
  - RT0: diffuse color (rgb), occlusion (a)
  - RT1: specular color (rgb), smoothness (a)
  - RT2: world normal (rgb, 10 bit/channel)
  - RT3: emission/light buffer; FP16 when HDR
  - Z-buffer: depth & stencil



Current G-buffer layout is as follows. All render targets 32 bit, except emission/light buffer which can be 64 bit (FP16) when rendering with HDR.

Ability to customize G-buffer layout is not in Unity 5.0, but on our TODO list.



## More Consistency

- Linear & Gamma color spaces more similar now
  - Lightmaps almost the same
  - More similar light intensities & specular highlights
- Removed hidden “\* 2” on light intensity in shaders
  - Custom shaders might need an update in Unity 5
  - Also code or animations that work with light color/intensity



Back to “smaller things”: Unity 5 has more consistency in lighting, especially between Linear & Gamma color spaces. We went over a lot of places where colors or intensities were stored/computed, and made sure to match them more properly based on color space.

Unity 4 shaders used to have an implicit “times two” in light intensity computations; that is gone in Unity 5. Built-in shaders got that removed; and when upgrading an existing project light intensities will be adjusted to match. However, if you are creating lights from code or animating their colors/intensities, you’ll have to use 2x brighter intensity to match Unity 4 behavior.

# Meshes

- Can haz 4 UV coordinates
  - Up from 2 UVs in Unity 4
  - Vertex colors can be floats (Mesh.colors vs colors32)
- No CPU/memory cost for non-uniform scale
  - Shaders do transform / normalization
- Per-Renderer instance mesh data
  - e.g. painting vertex colors in the world
  - MeshRenderer.additionalVertexStreams



Improvements to meshes & geometry: more UV coordinates; ability to use floating point vertex colors.

No CPU or memory cost for non-uniformly scaled objects. Unity 4 used to pre-scale meshes on the CPU for that case, which was a problem if you were animating scale; and in general a memory waste to store that pre-scaled model. Now shaders just receive a full transformation matrix and do proper normal/tangent normalization. Shader lighting computations are generally moved to happen in world space now, instead of a mix of object/world/tangent spaces like it was before.

Added ability to setup per-instance mesh data via MeshRenderer.additionalVertexStreams. For example, if doing a world vertex color painting tool; there's no longer a need to create duplicated mesh instances that only differ in their vertex colors; can instead just store a "vertex-color-only mesh" for each instance in the scene.

## Shaders: Stripping

- Build-time stripping of unused shader variants
- `#pragma shader_feature`
- Unused fog / lightmap modes
- Standard shader can have 35k variants in total
  - 375MB if compiled for DX11 only (26MB zipped)
- Most content ends up using several hundred
  - Several megabytes



Shaders got build-time removal of unused shader variants. Standard shader with all its possible options expands to 35 thousand possible variants; and that is 375MB of data just for DX11 only. But majority of scenes never use all these variants; and only need a few megabytes of data. We now analyze the variants used by materials during game build and only include the needed ones.

Downside is if you want to switch to shader variants at runtime that were not used during game build step — they will not be in the build. A workaround for now is to make sure they are somehow included into the build (easiest is putting materials with these variants into Resources folder; that way they are always in the build).

# Shaders: Load Performance

- Shader loading performance
  - Generally loads less variants than Unity 4
- `ShaderVariantCollection` to control preloading
  - List of shaders + their variants
  - Can be recorded from editor
  - Replaces the very blunt `WarmupAllShaders`



Unity 4 generally loaded full shader with all its variants at load time. Unity 5 by default only loads the variants when they are actually needed. In many cases this improves load time & decreases memory usage; however due to on-demand loading can result in small hiccups at runtime.

We've added `ShaderVariantCollection` asset & scripting class to help with this. It's a list of shaders and their variants, and can make sure these variants are fully loaded (by doing both a Unity-side loading, and issuing a dummy draw call to the graphics API to make sure the driver fully compiled the shader).

## Shaders: Fog

- Fog done differently
- Works on all platforms now! (WP8 & consoles)
- Shader macros to do fog
  - Instead of Unity 4 runtime shader patching



Fog is done differently in Unity 5.0; with the benefit that it works on WP8 and consoles.

Now fog computation macros are just part of the shader, instead of runtime shader patching that was in Unity 4. Surface shaders automatically generate fog code (with options to turn it off if not needed); manually written vertex+fragment shaders will need to be updated to handle fog.

## Other Misc Stuff

- More Performance
  - Light culling in forward rendering
  - Shadow caster culling
  - Less SetPass calls, especially with light probes
- More shader keywords
  - 128, up from 64



Light culling (assignment of lights to objects in forward rendering) and shadow caster culling was multithreaded & optimized. Amount of material SetPass calls done is in generally much lower, especially when only changing small amounts of shader data between different objects (e.g. light probe information).

Number of shader variant keywords available is increased to 128 (up from 64 in Unity 4).

## Extensibility: Command Buffers

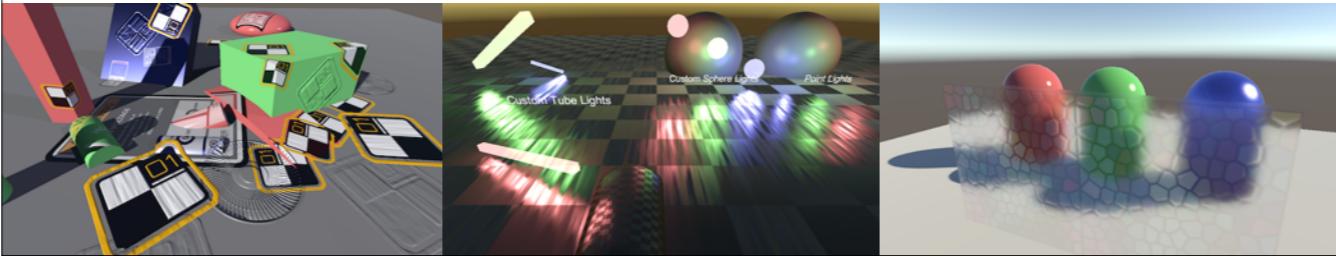
- Create “lists of things to do”
  - Draw mesh, set render target, blit with shader, ...
- Tell cameras to execute them at various points
  - After deferred G-buffer
  - After deferred light pass
  - Before all transparencies
  - ...



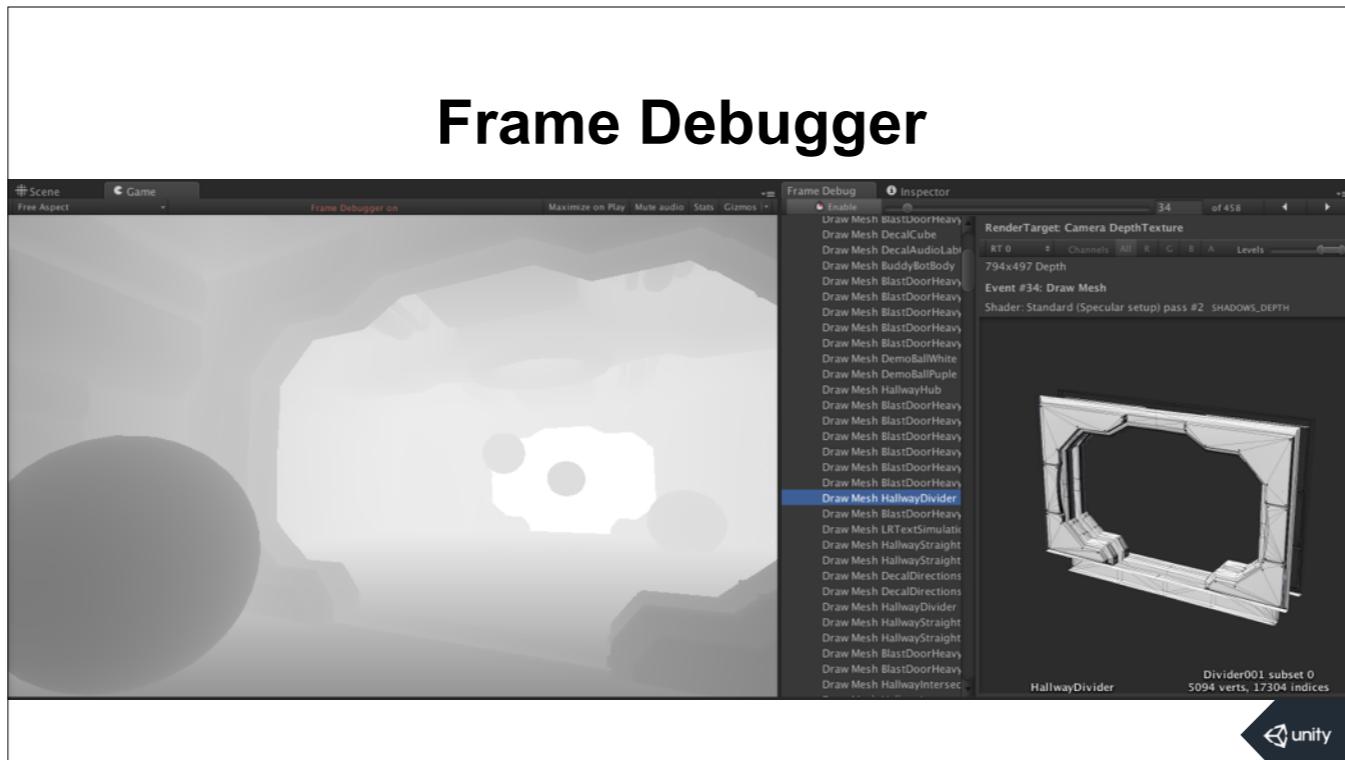
Unity 5 also made the rendering pipeline more extensible — at various points during camera rendering, it is possible to specify “list of things to be done” (we call that a “command buffer”). The commands are fairly high-level, e.g. “draw mesh with this material; set render target; blit from one render target to another using a material etc. Cameras can execute these command buffers at a bunch of places during frame — after deferred G-buffer, before transparencies, after everything is rendered etc.

## Things to do with command buffers

- Deferred decals
- Custom lights in deferred shading
- GrabPass on steroids
- ...



We've already seen people use command buffers to implement custom effects for deferred shading (deferred decals, custom lights, custom fog volumes). There's an example project in documentation showing several simple use cases.

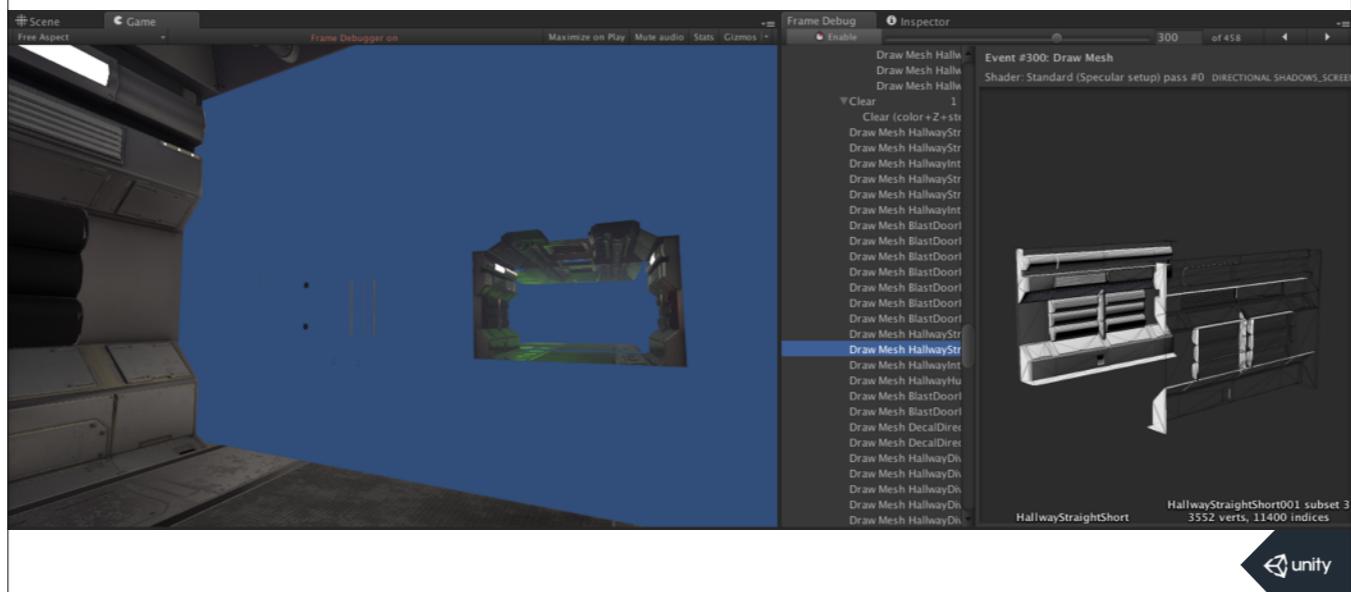


Unity 5 also adds a built-in frame rendering debugger (Window -> Frame Debugger). It lets you scrub through the draw calls made during the frame and see how exactly the frame was rendered (including current render target contents; shader being used etc.).

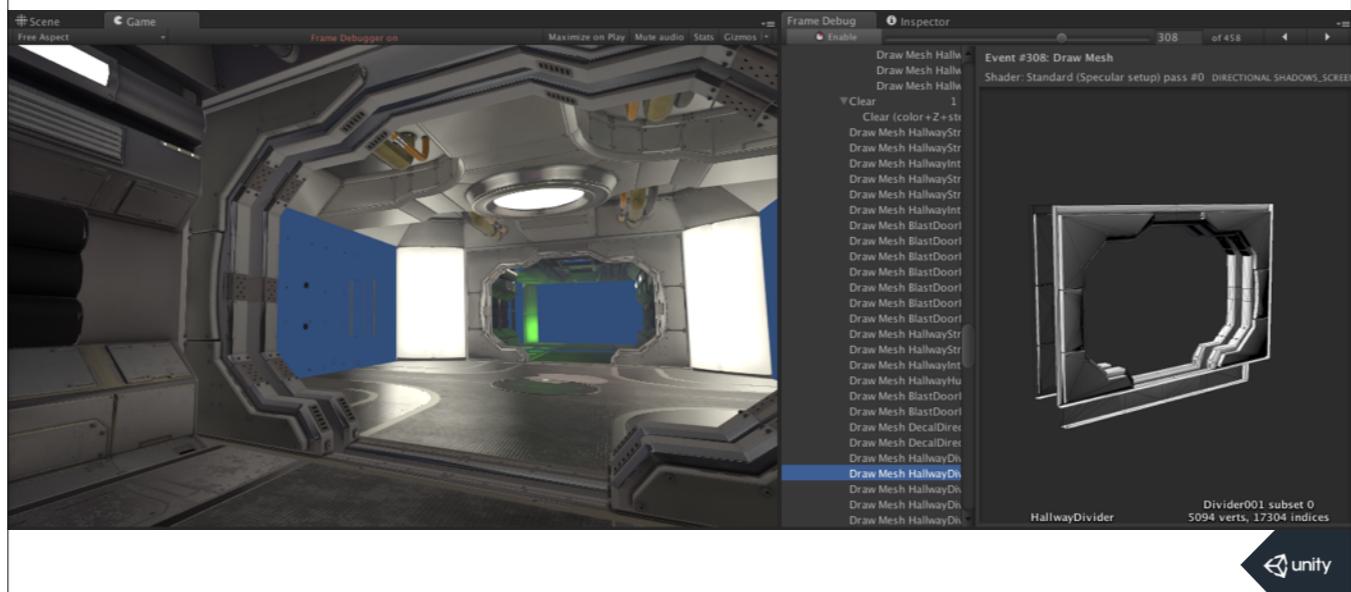
It's a good way to quickly investigate rendering issues, and to learn how the rendering pipeline works in general.

Currently frame debugger only works in the editor (i.e. you can't use it to debug a frame on a mobile device). For more advanced and/or on-device frame debugging, using external 3rd party tools (RenderDoc, Visual Studio graphics analyzer, Intel GPA, NVIDIA NSight, ...) is of course still an option.

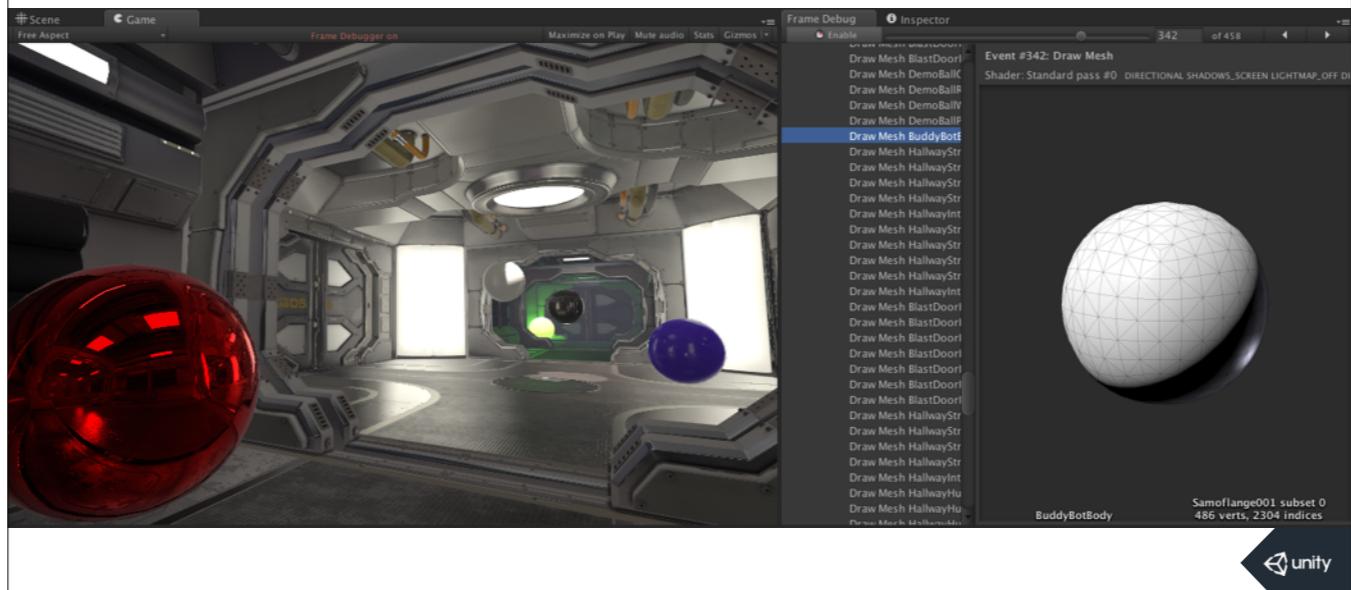
# Frame Debugger



# Frame Debugger



# Frame Debugger





**Questions?**