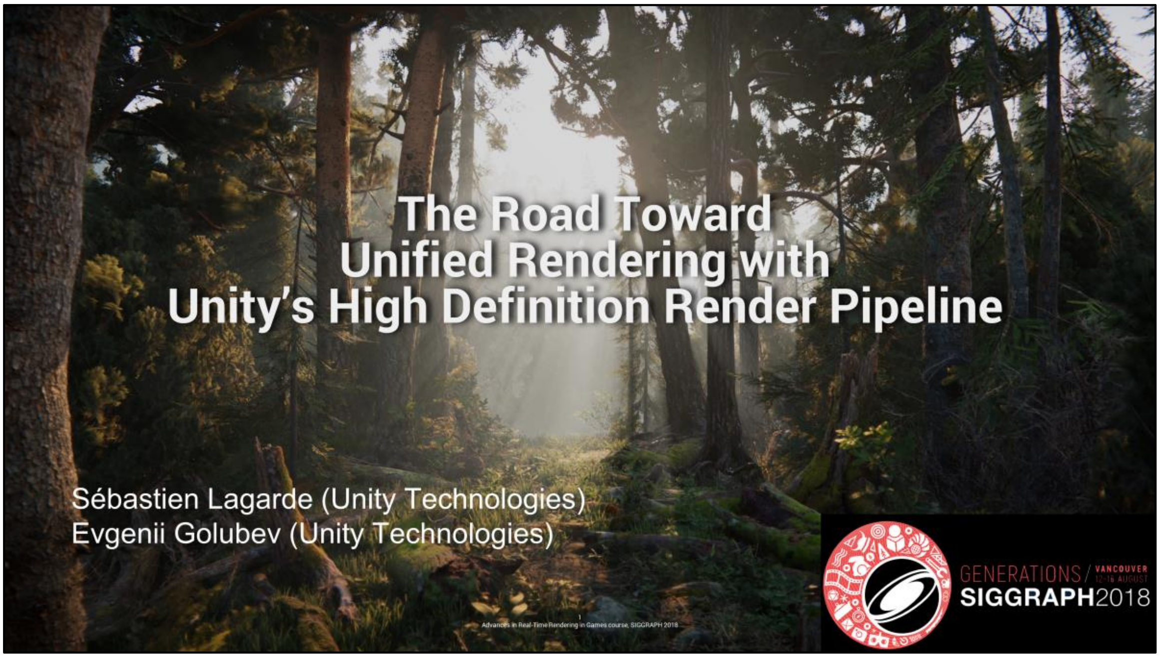
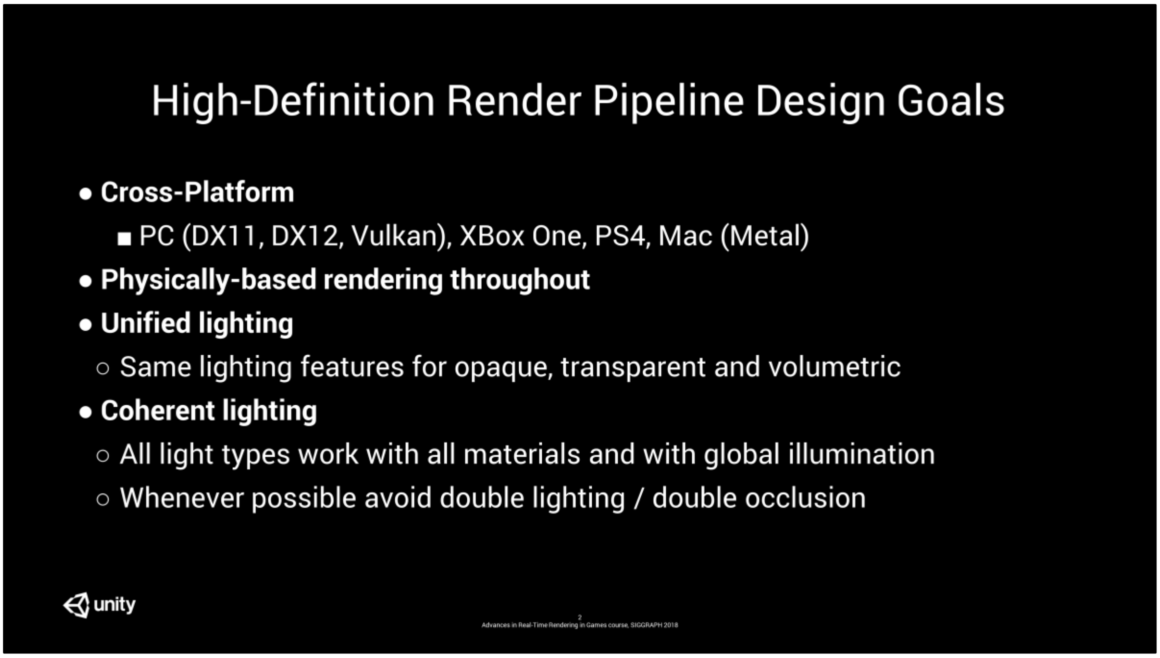
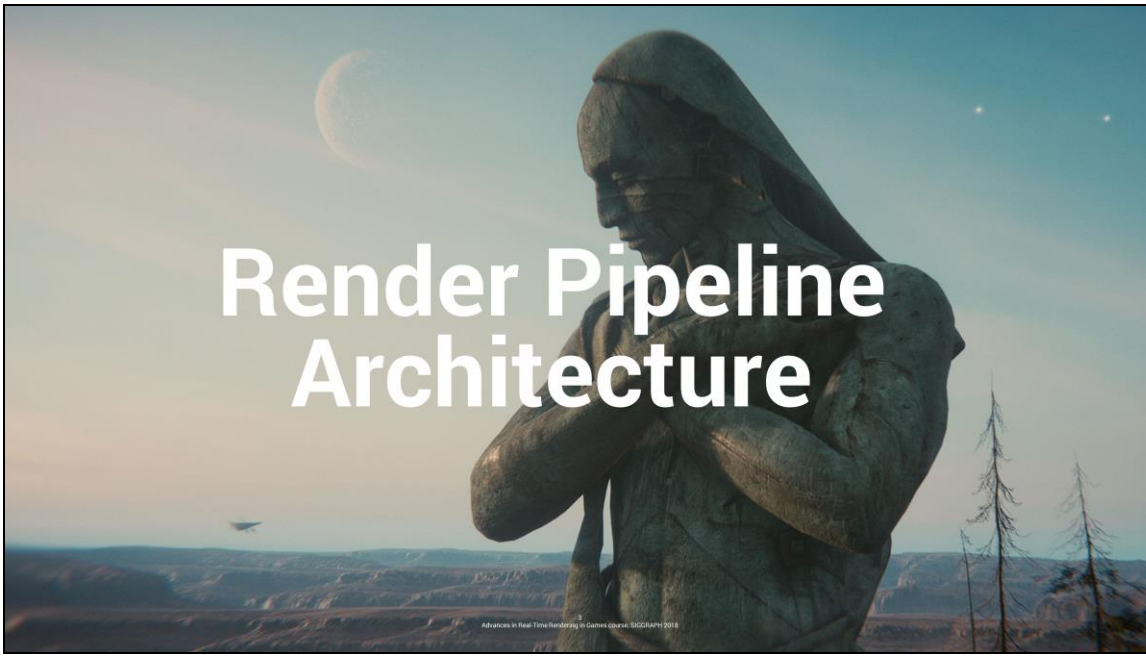


Everything that we're presenting today has been a real team effort. A lot of people contributed to the systems we're describing



PBR: Material, lighting, camera

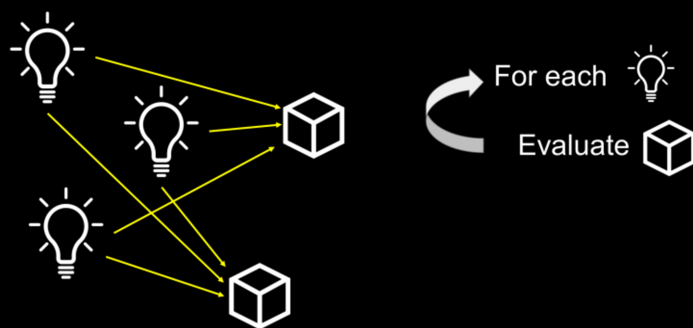

Render Pipeline
Architecture
3
Advances in Real-Time Rendering in Games course, SIGGRAPH 2018

Render Pipeline Architecture

- Key components:
  - Lighting and material architecture
  - GBuffer Design
  - Forward / Deferred path features parity (aka *Features parity*)
  - Decal architecture
- Follow up by
  - Material overview
  - Volumetric lighting

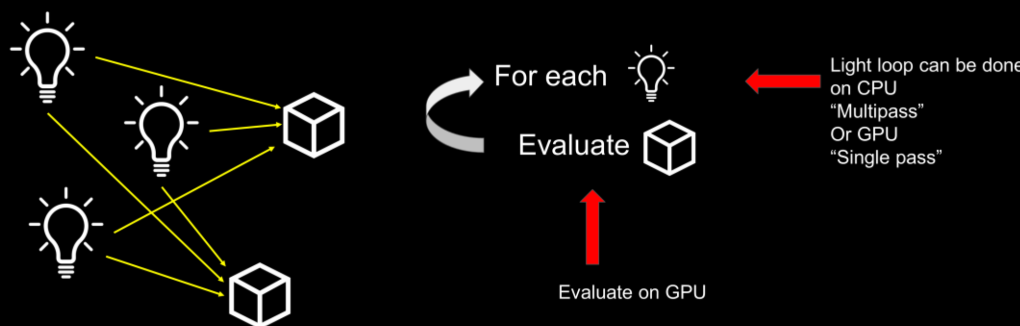
Lighting Architecture

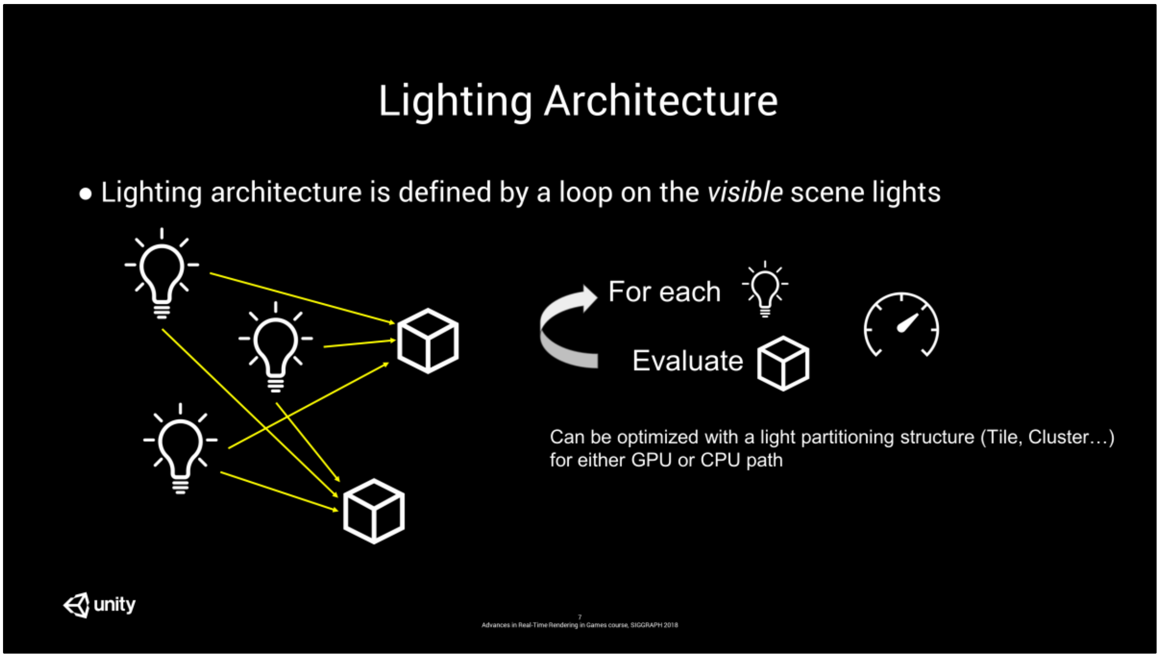
- Lighting architecture is defined by a loop on the *visible* scene lights



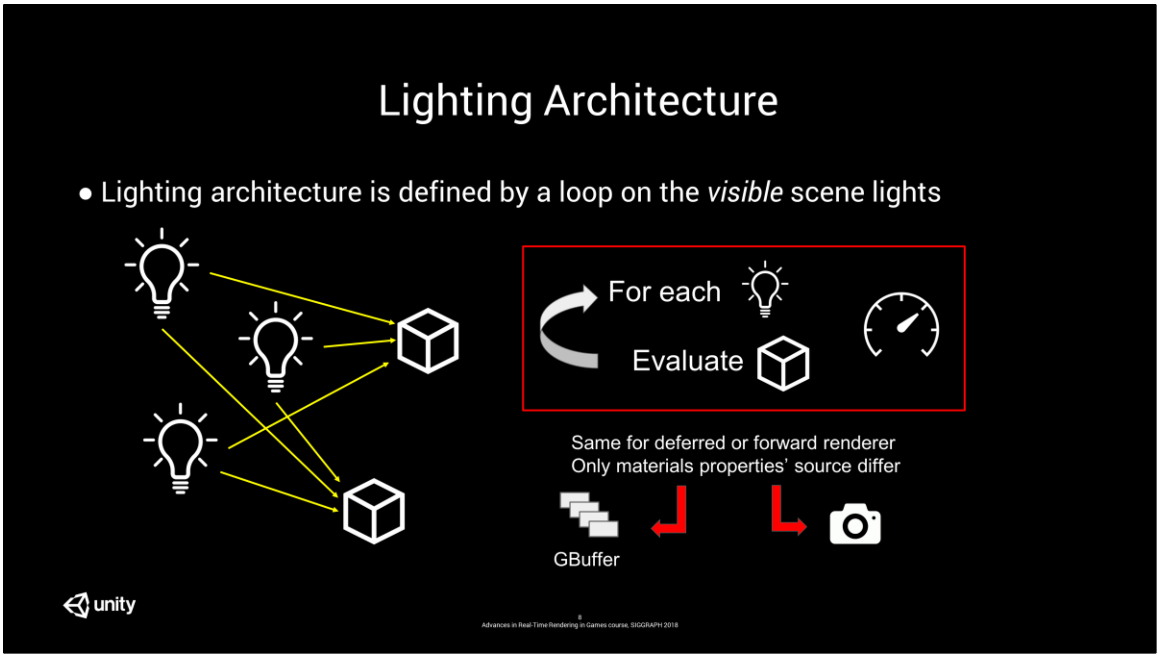
Lighting Architecture

- Lighting architecture is defined by a loop on the *visible* scene lights

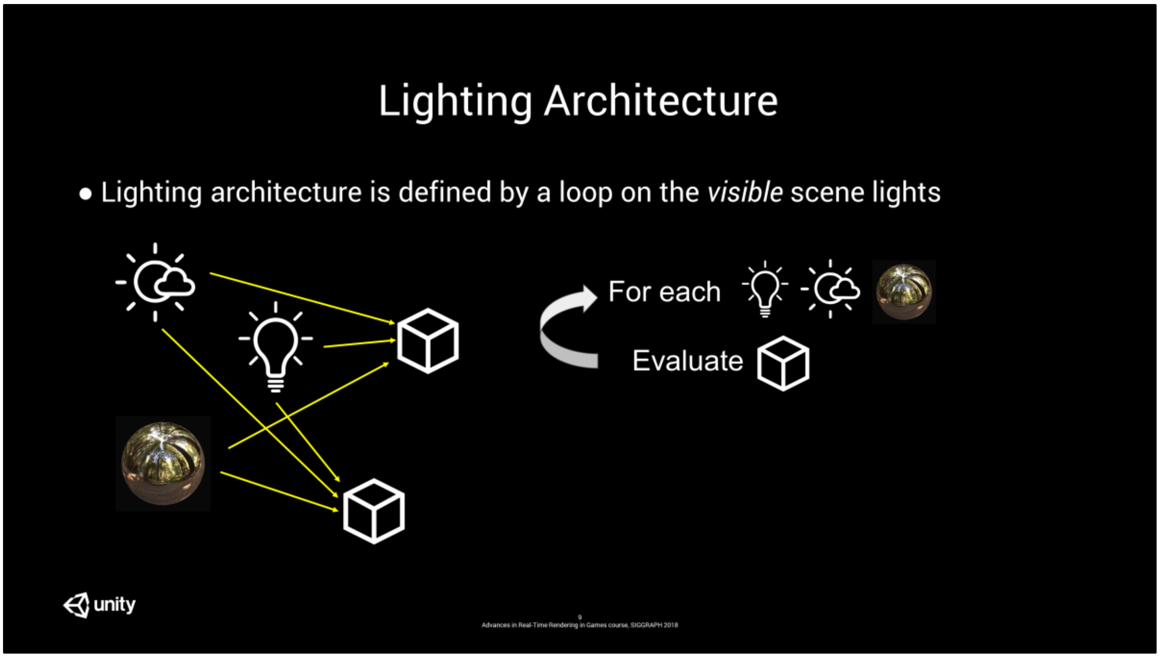




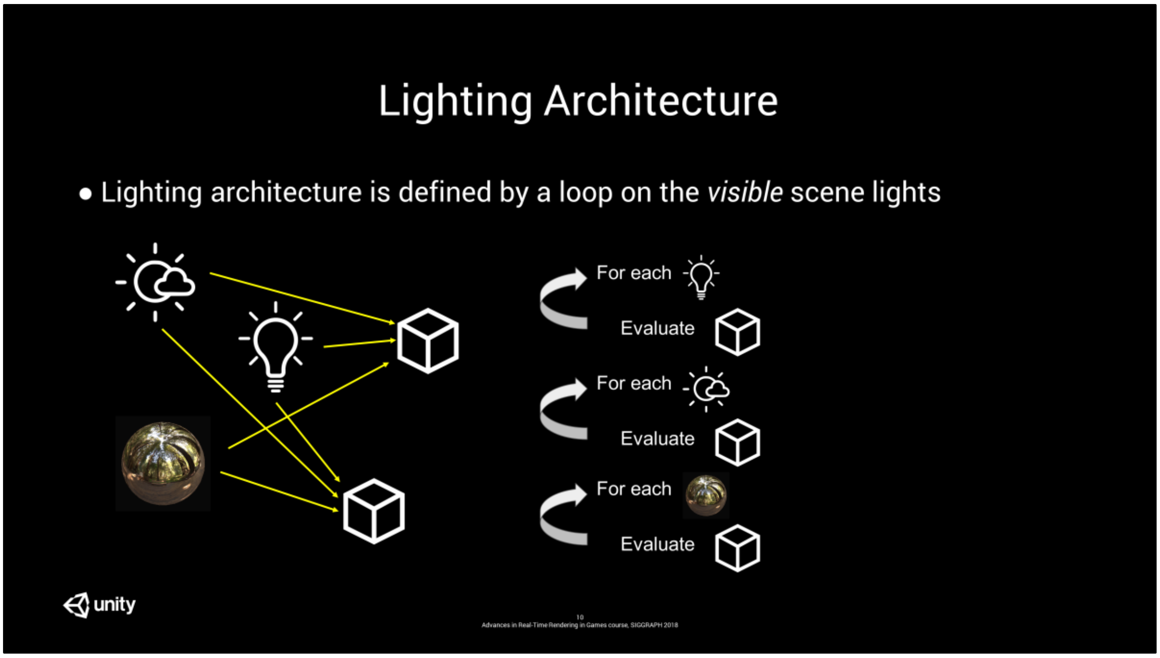
Such a loop can be optimized with CPU or GPU help to remove lights that don’t affect the material.



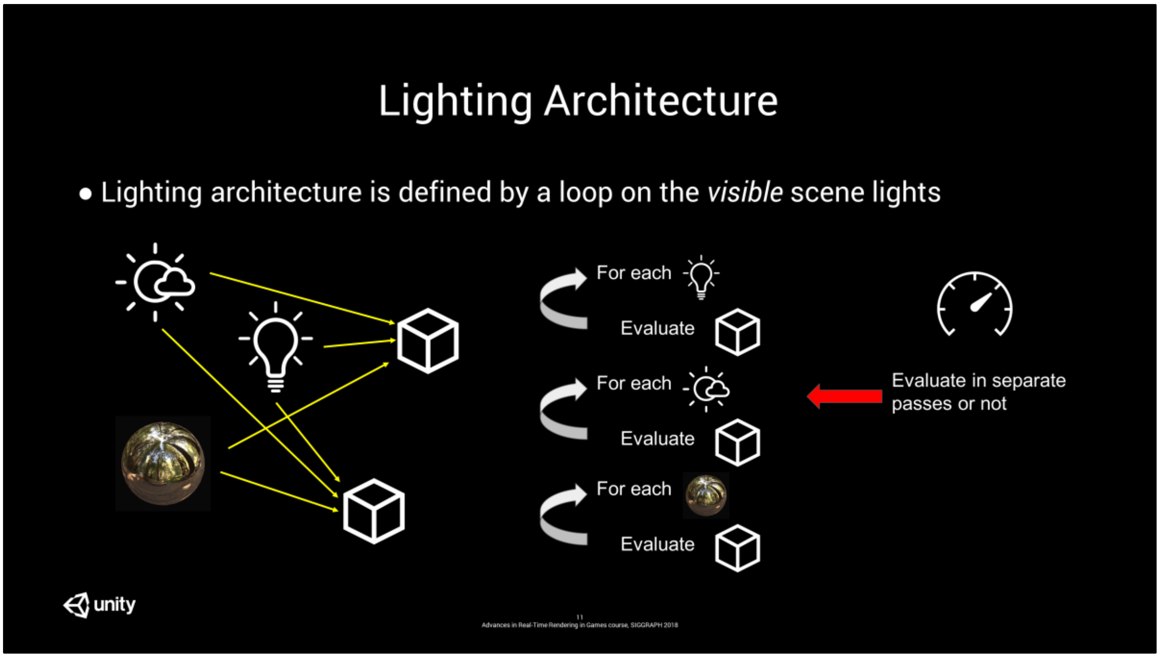
Note that this light loop is conceptually identical in deferred or in forward. Only the source of the properties of the material differ. In deferred it comes from the GBuffer and in Forward it comes from the object uniforms / textures.



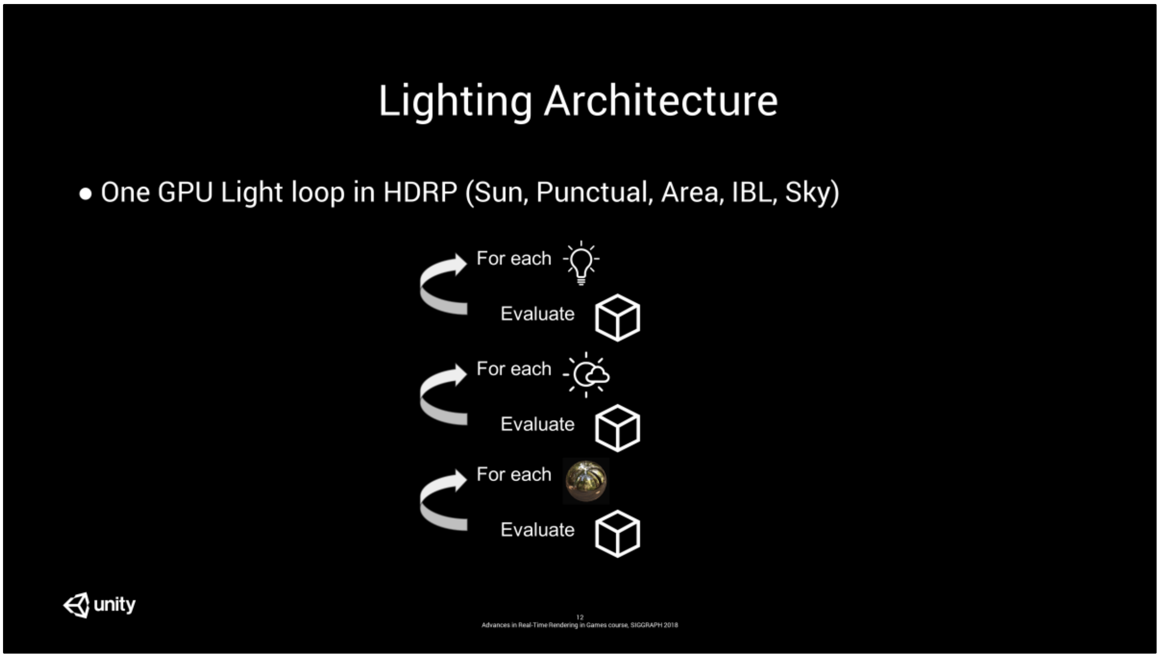
For each light TYPE, evaluate material response



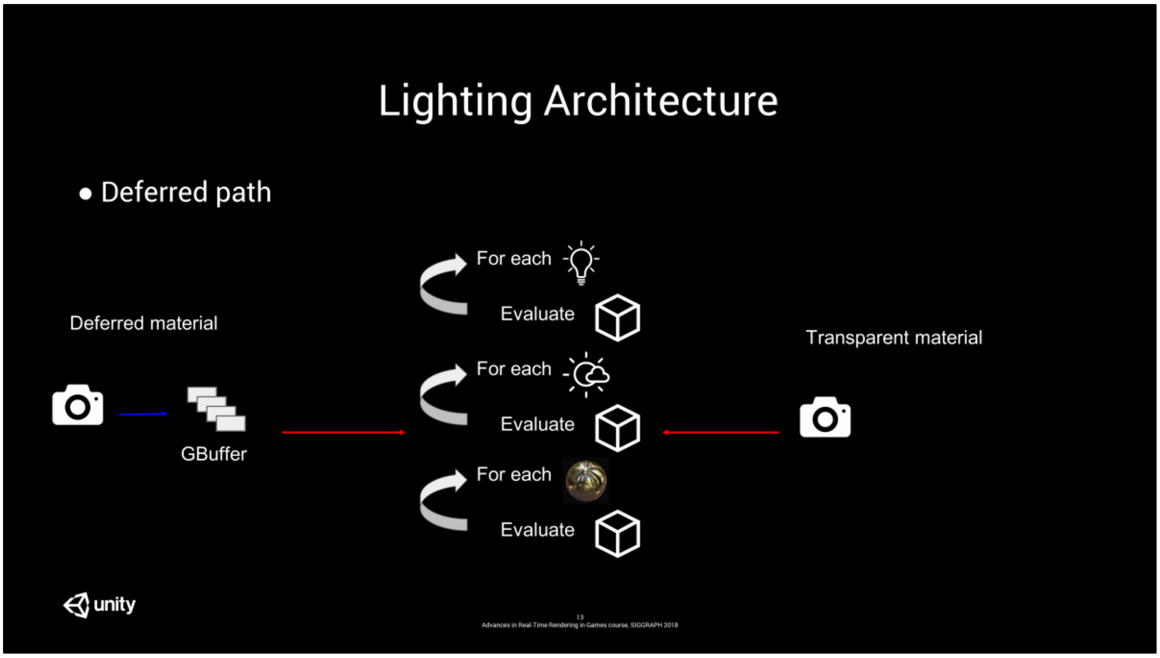
For performance reasons, in game there is often a coupling between a light type and the material evaluation response. Like we pre-integrate IBL by the lighting model of the material.
So we need to do one loop for each light TYPE.



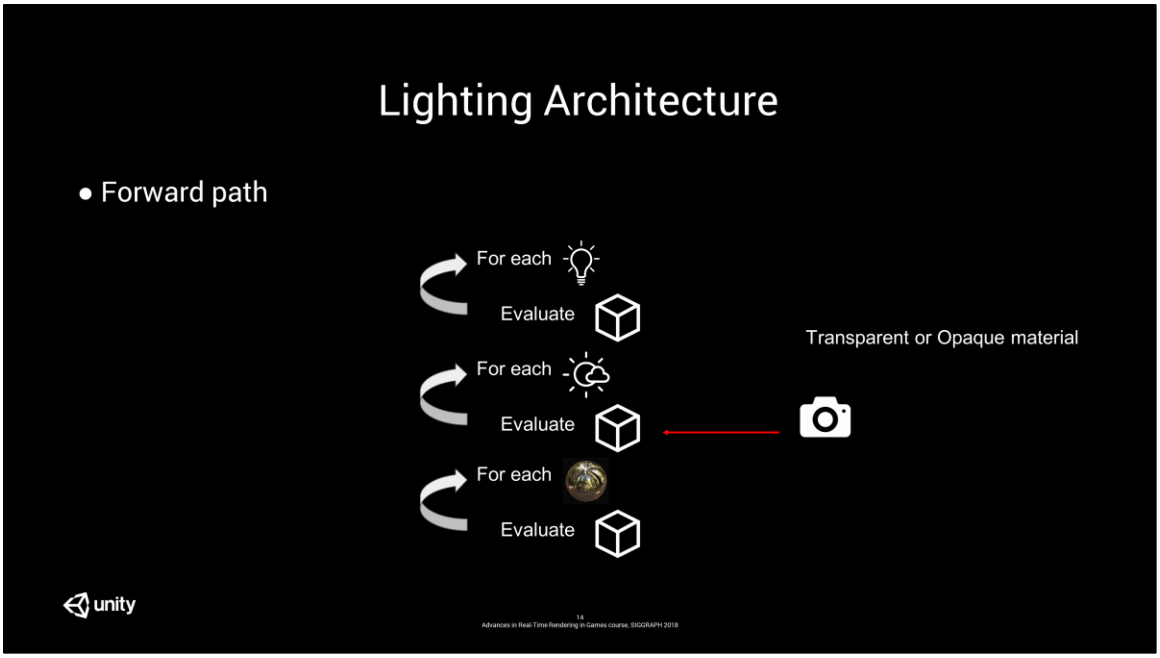
This series of light type loop is sometime split in different call, performance may vary



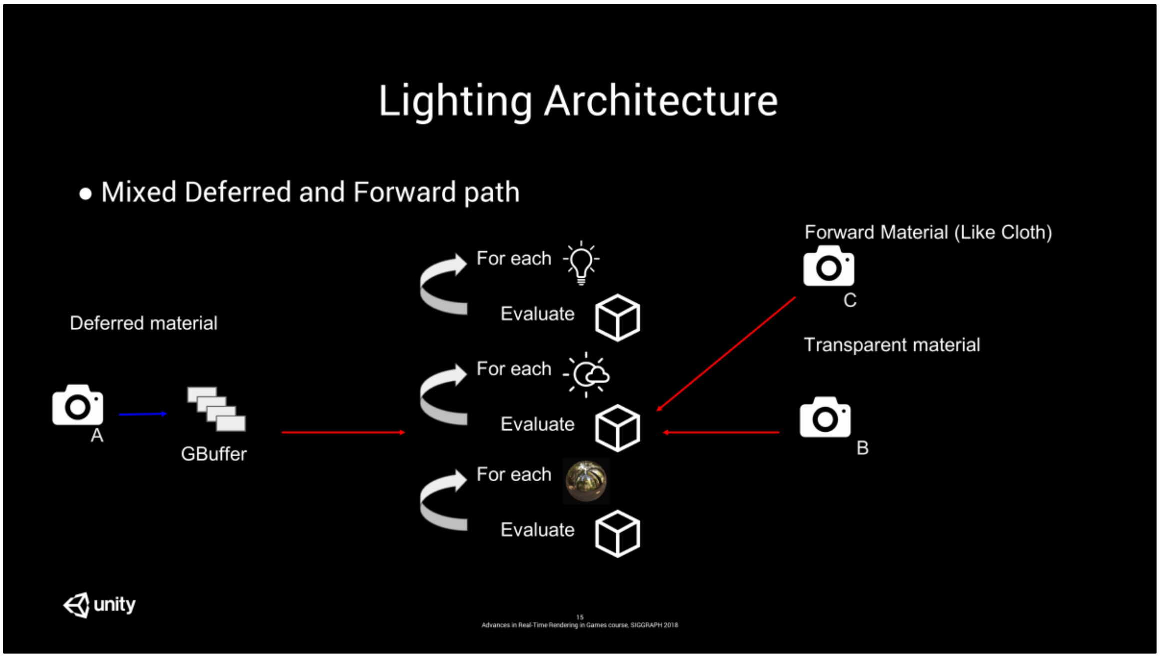
In HDRP we use a single light loop with all the light type. We have Sun, punctual light (spot, point), area light, IBL, sky.



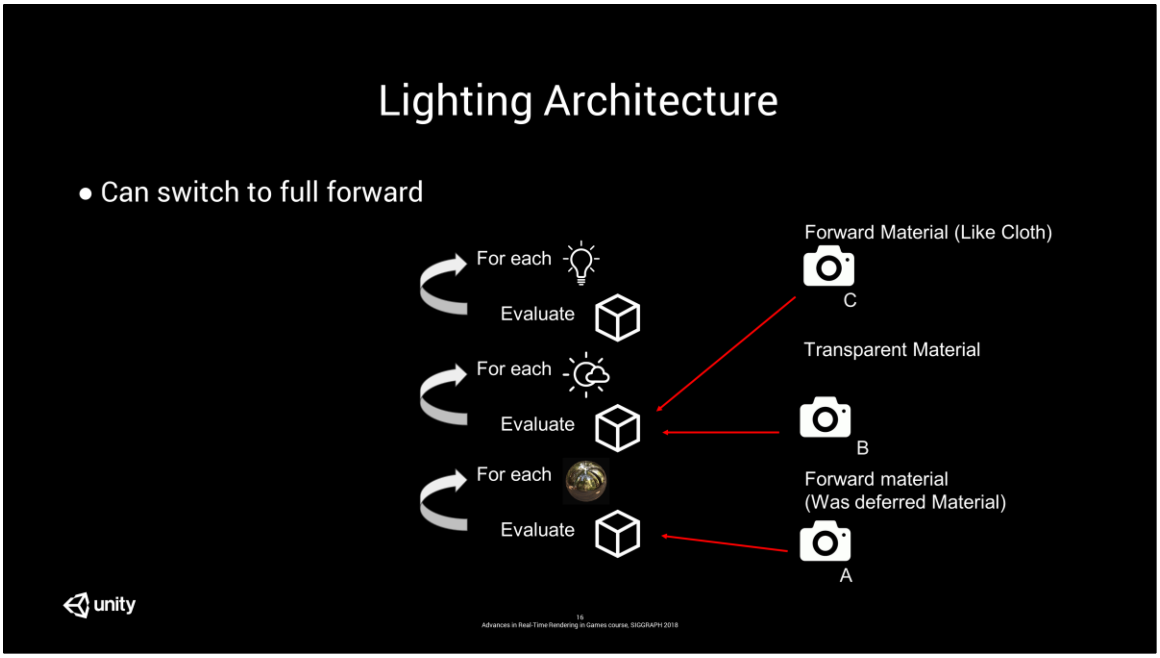
HDRP support both deferred and forward renderer. Let’s take the example of deferred renderer. We have one deferred material that fill a gbuffer and one transparent material that use
The same light loop. Unified lighting.



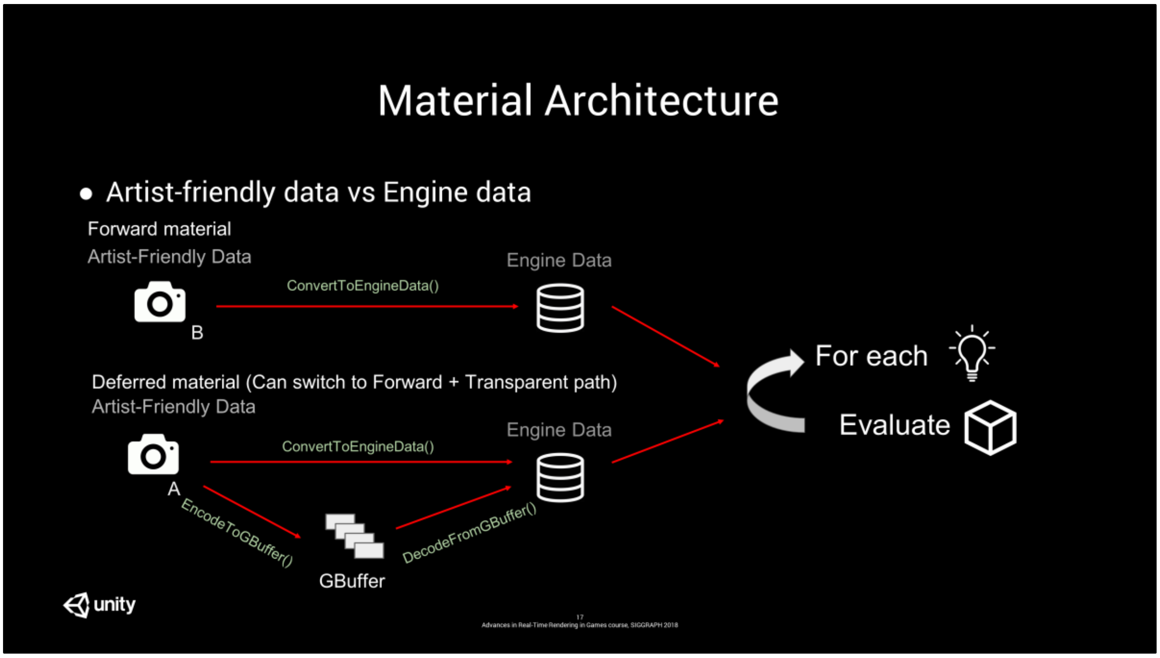
Example of forward renderer. We have forward opaque and transparent material



HDRP also support both forward and deferred at the same time. In this case, in addition to what we have seen for deferred path, we also have forward opaque material.



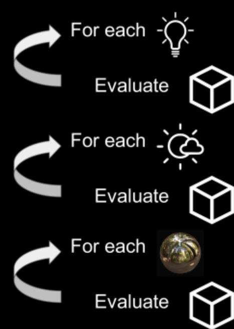
We can switch (dynamically or not) to full forward



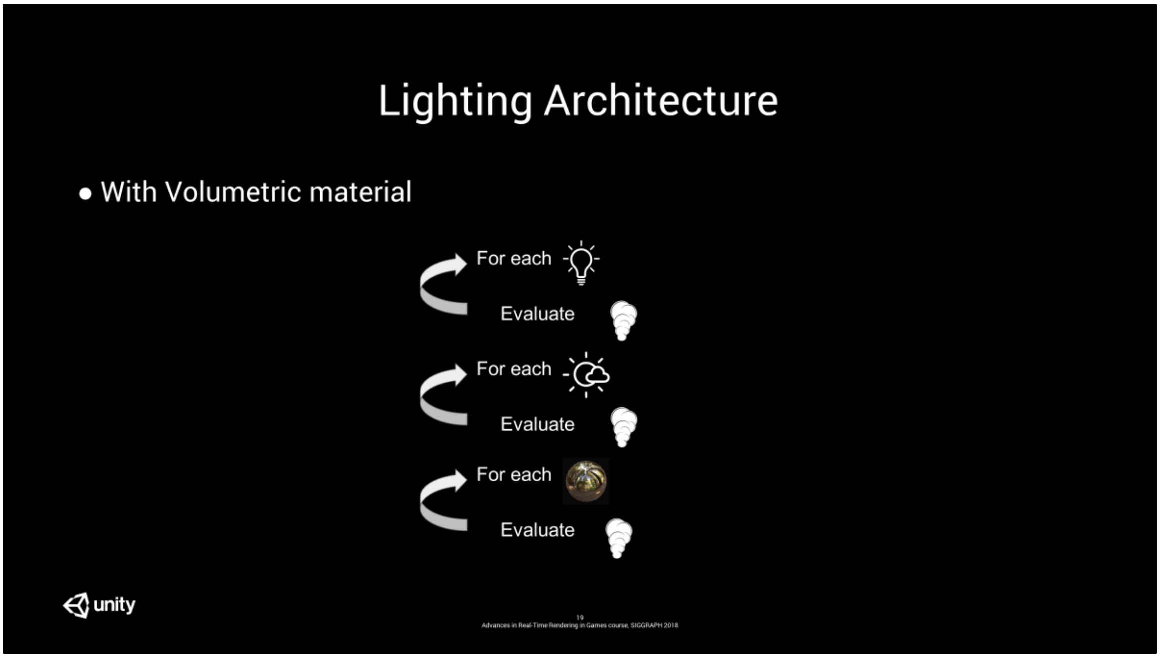
This schema show how we write a material in HDRP to work with our deferred and forward architecture. These are guidelines. And we introduce the concept of artists friendly data and engine friendly data.
Let’s say that the inputs fill by an artists in a UI or shader graph is artists friendly data. Like Smoothess. We add a conversion function to engine data (For example roughness), that the lighting engine is able to used.
The Gbuffer is then just an intermediate storage. It can be compressed.
Material of HDRP need to follow this material guidelines to fit in the lighting architecture.

Lighting Architecture

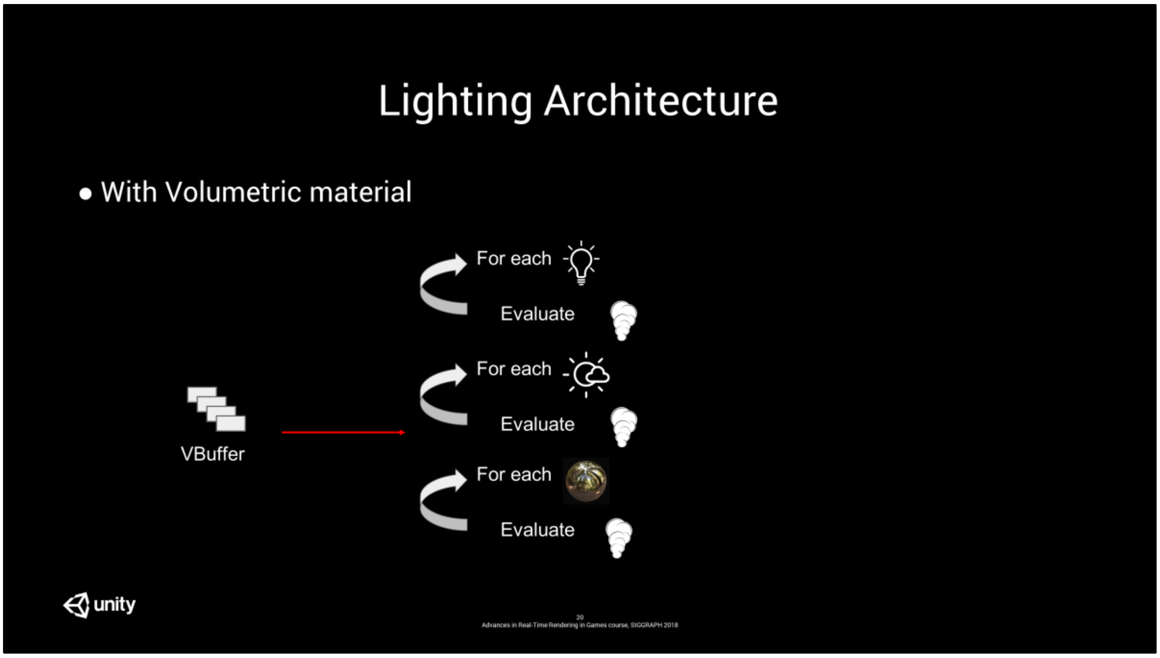
- Remember



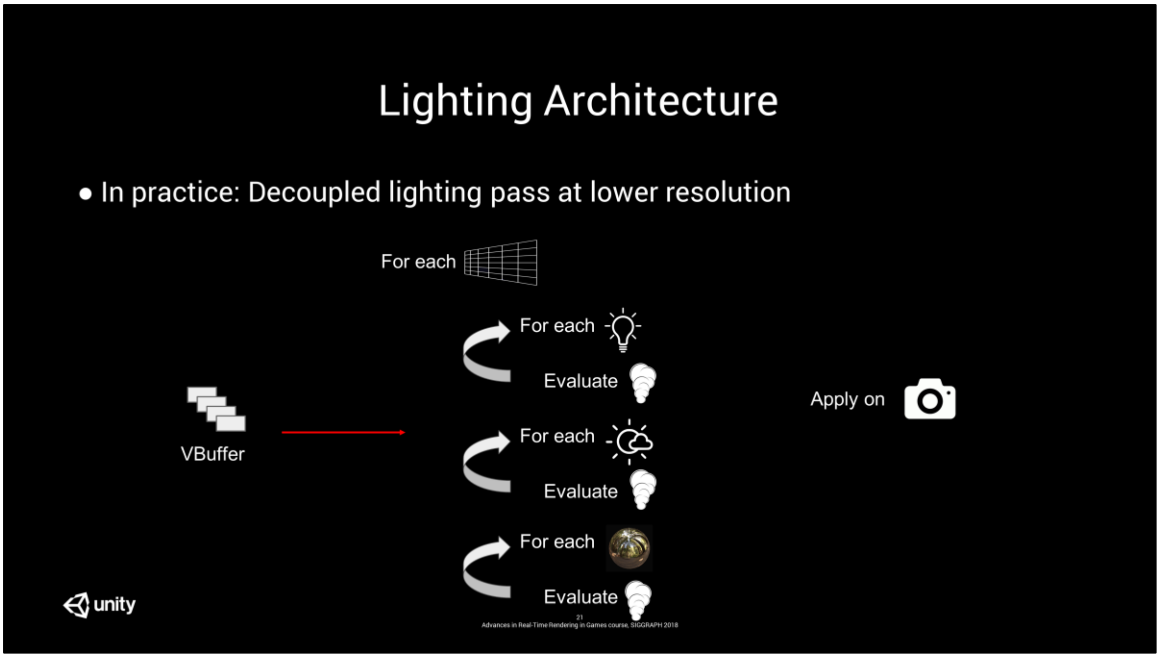
unity



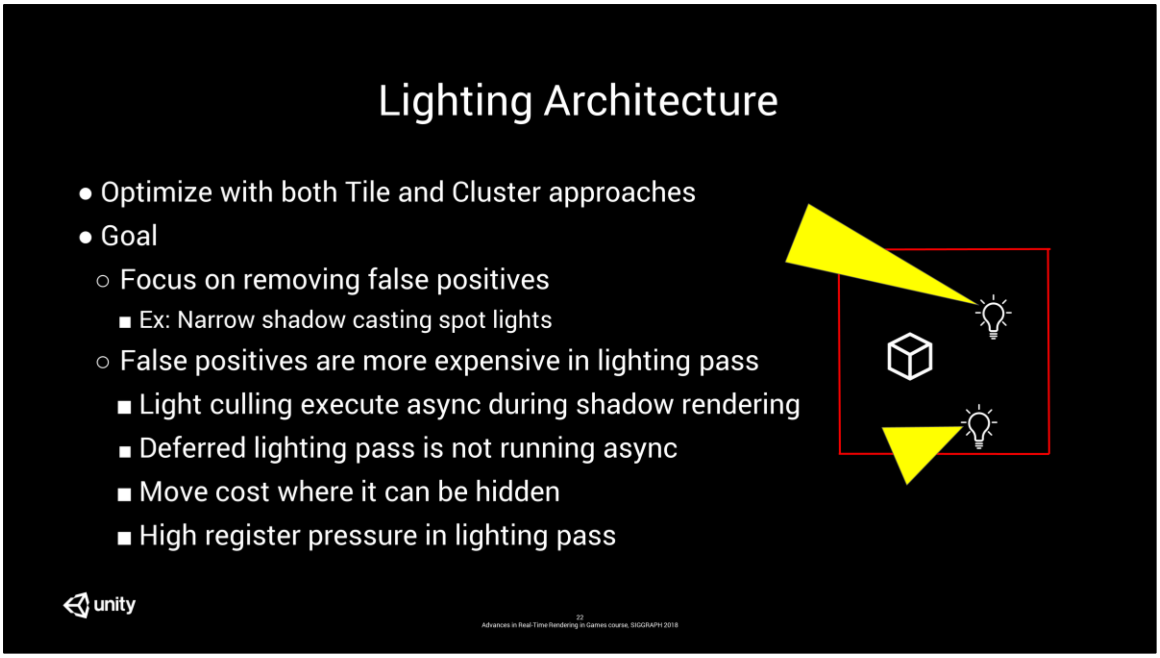
With volumetric the concept is exactly the same, except we used volumetric material instead (absorption, scattering)



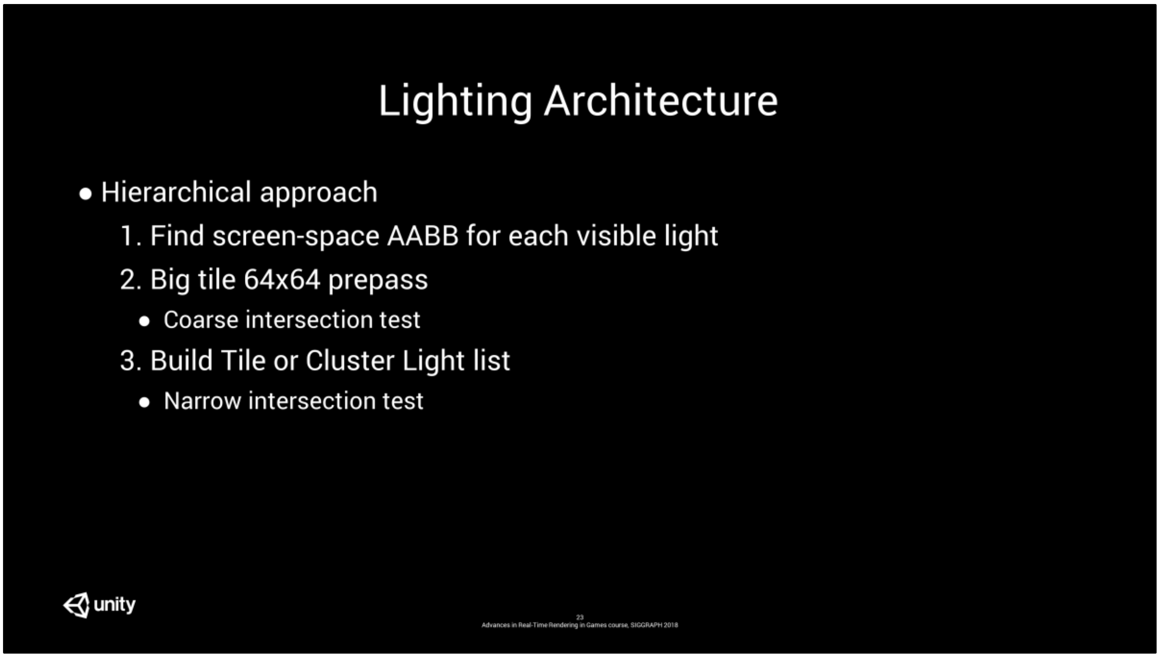
Similar to GBuffer we use VBuffer as input of volumetric material for evaluation.



And in practice we decouple the lighting pass and evaluate for each cell of a froxels. Then apply the result on the opaque and transparent material in a separate pass.



1. Major emphasis on aggressive (but fast) removal of false positives even for spot light with sphere cap.
   - Important since spot lights are often shadow casting and narrow.
   - List building using basic bounding sphere testing is highly insufficient.
2. All list building work is absorbed by leveraging asynchronous compute.
3. False positives are much more expensive to deal with during lighting rather than early on.
   - Final lighting shader has higher loop complexity and greater register pressure.
   - Final lighting shader cannot leverage asynchronous compute during rendering of shadow maps.
4. The lists can be used for either deferred or forward or both.
5. Lists are delivered in order of increasing index to preserve order by type which helps reduce thread divergence during lighting.

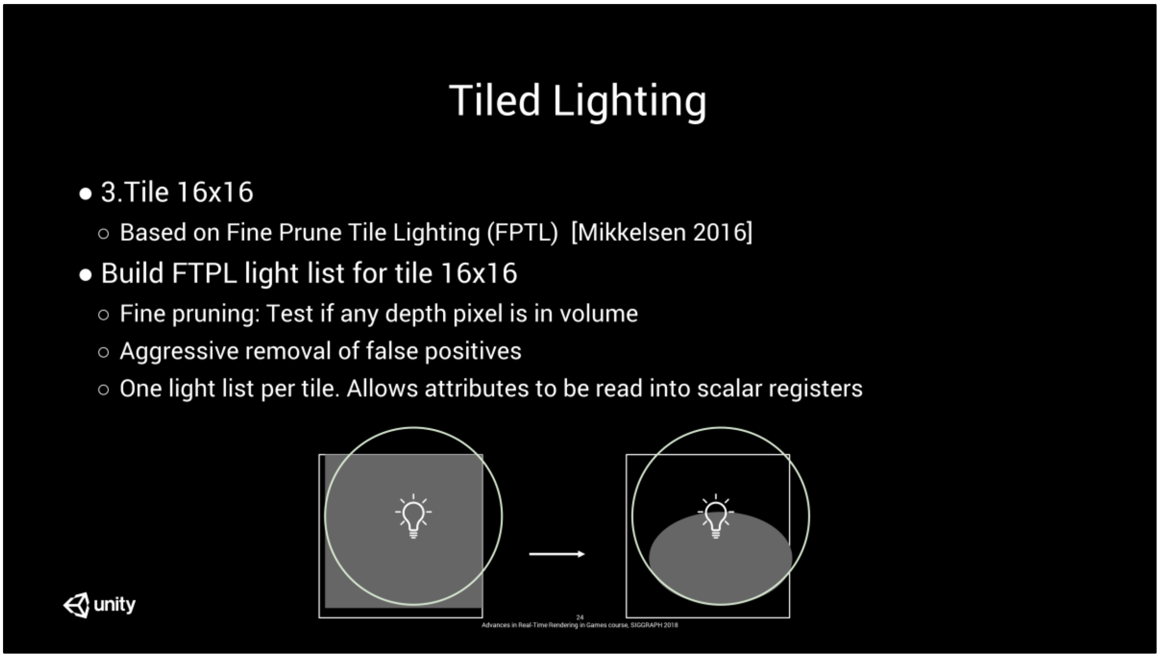


1. Find screen-space AABB for each visible light
2. Big tile 64x64 tile pre-pass. Use AABBs for initial early out (2D no depth).
  - Follow up with exact intersection test between tile and convex hull.
  - Use bounding sphere as an extra testing criteria (helps with point lights and sphere capped spot lights).

Basically first comes AABB pass, then comes big tile pass which uses what AABB pass produced and then comes FPTL and Clustered list building passes which use both what big tile pass but also what AABB pass produced fptl and clustered both use the list of potential tile overlaps generated in big tile prepass
they both use the AABBs to test the ones left in the list from big tile prepass
the convex hull is the oriented bounding box but with 2 scale values so we can squeeze the top 4 vertices along separate axis X and Y to create either a pyramid or a wedge when the scales are set to 1.0 it's just an obb if they are less than 1.0 they get scaled inward. If they both go all the way in it becomes a pyramid if only one goes in its a wedge and then the bounding sphere helps give us the sphere capped part of the spot light, the obb and the two scales is what we use as the convex hull but we also use the bounding sphere as an extra constraint for rejecting more tiles which is important for both point lights but also for the sphere cap in both FPTL and clustered they loop over what the big tile pass generated per 64x64 tile as a list. They both

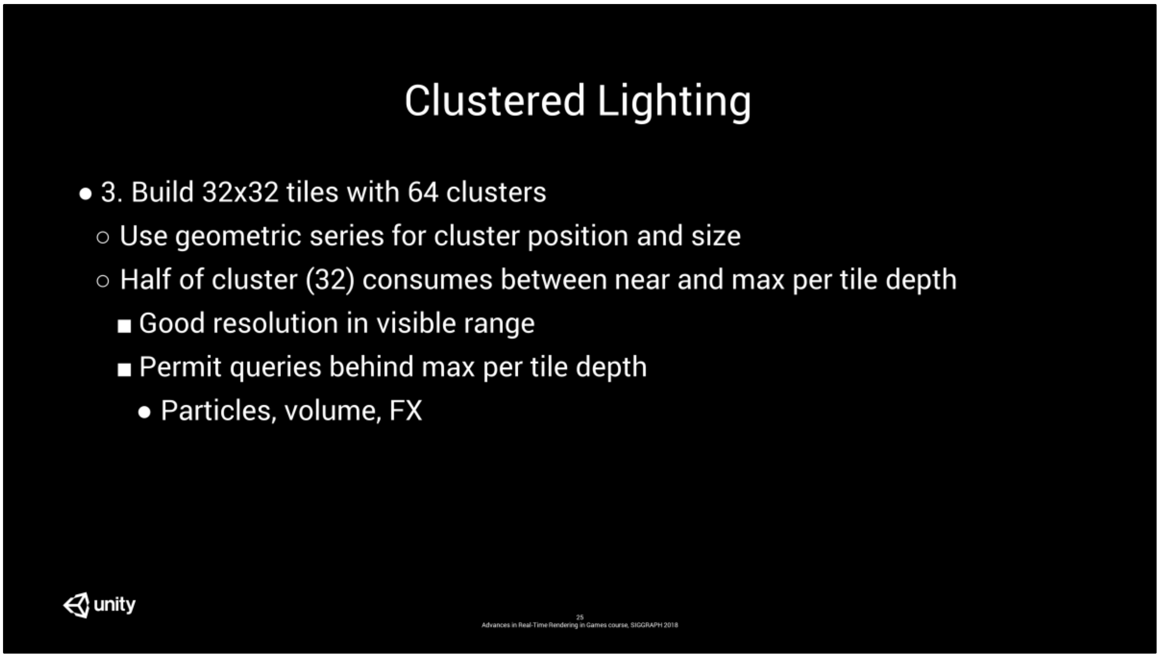
check AABBs first against the list and build up a coarse list in LDS. Then they both follow up with checking if the silhouette of the bounding sphere overlaps the tile for each light in the coarse list. finally fptl does fine pruning and clustered checks clusters against the remaining lights
when bigtile prepass checks AABBs it's 2D and against 64x64 tiles. When fptl does it for instance it's 3D aabb test and against 16x16 tiles

for fptl it's particularly tight since it only needs it to cover opaque pixels so we can reject a lot in that early pass alone by including that min/max depth in the intersection test so big tile only does .xy in the aabb test. fptl does .xyz (edited) the sphere overlap against tile is 2D overlap test in all cases but of course you can remove a little extra because it's a smaller tile so I decided to do the test again because it's pretty cheap for clustered the second AABB test is still 2D but it prunes a little extra since it runs on 32x32 instead of 64x64 and it's a very fast test



FPTL implementation
1. For FPTL we use 16x16 tiles with no clustering.
2. First do trivial AABB test (3D). Then do tile vs. bounding sphere test.
big tile only does .xy in the aabb test. fptl does .xyz
3. Fine pruning removes any light that does not have at least one opaque pixel/point inside its true volume.
   - Aggressive removal of false positives but works for opaque only.
4. Since all false positives are removed and since FPTL is for opaques only we write one list per tile.
   - During deferred this allows light attributes to be read into scalar registers instead of vector registers since all pixels in the tile visit the same list.
   - No thread divergence during deferred since all threads processing the tile read the same list.

What is new compare to the article referenced:
1. Clustered
2. big tile prepass
3. fast silhouette of sphere vs. 2D tile overlap test



Clustered implementation
1. Cluster resolution is 32x32 tiles with 64 clusters.
2. Performs accurate but fast cluster vs. light intersection test - even for sphere capped spot lights.
3. Use geometric series to establish cluster position and size.
4. Common ratio established per tile such that half of the clusters (32) are consumed between near plane and max. opaque depth.
   - Provides highly optimal cluster resolution in the visible area while still permitting queries behind max. opaque depth (for things like particles, volume lights and transmission fx).

(3 + 4) it's choosing the parameter for the geometric series such that it has spent exactly half of the clusters by the time it reaches max opaque depth in the tile

Lighting Architecture

- Performance
  - PS4 @ 1080p

Scene (Time in μs)	Forward Tile	Forward Cluster
Opaque (around 30 punctual lights, 3 area lights, 5 environment lights)	5675.5	7135.9

- Note: MSAA can have non negligible impact (not measure here)

Lighting Architecture

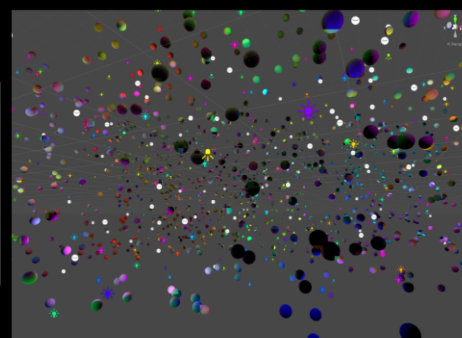
- In HDRP
  - Transparent materials use cluster
  - Deferred materials use FPTL
  - Forward opaque materials can choose between FPTL or cluster

unity

Lighting Architecture

- Tile / Cluster Performance (no MSAA)
  - 1080p PS4: Tile + Cluster list generation

Scene (Time in µs)	Light count	Total	ScreenBounds AABB	Big Tile	Build Tile light list	Build Cluster light list
Punctual lights	10	955.755	167.119	25.788	359.718	403.13
Punctual lights	50	1006.902	169.906	49.681	370.274	417.041
Punctual lights	100	1095.595	170.902	91.869	387.872	444.952
Punctual lights	500	2646.833	170.367	390.881	701.966	1383.619
Mixed lights	10	954.28	168.348	37.324	359.753	388.855
Mixed lights	50	1030.835	166.841	80.266	371.427	412.301
Mixed lights	100	1445.809	175.426	241.569	456.004	572.81
Mixed lights	500	4475.837	179.4	1059.159	1100.698	2136.58

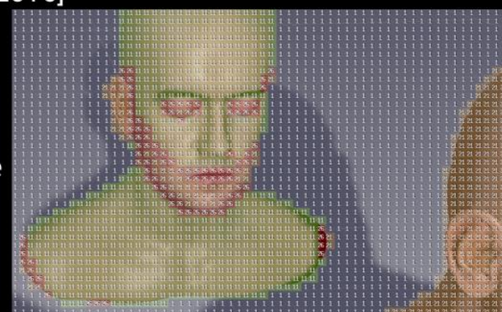


unity

Entry cost is expensive: 1ms, almost same cost for 1 or 10 light, but it scale well.
Scene on the side is display with our debug tile lighting mode

Lighting Architecture

- Want to reduce VGPR pressure
- Deferred renderer [Coffin 2011][Garawany 2016]
  - Material classification
  - Light classification
    - Big win when dealing with area lights
  - Can’t cover all variants - Need worst case
- Forward renderer
  - Implicit material classification
  - No possible light classification

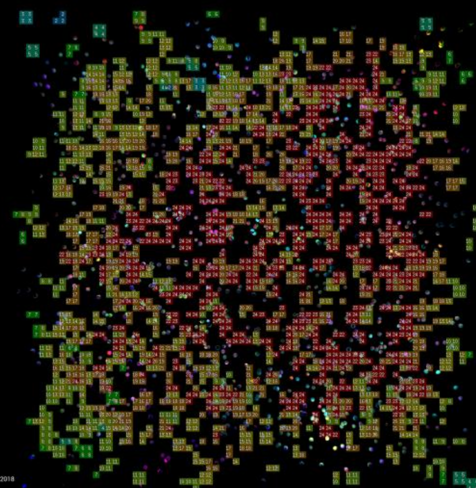


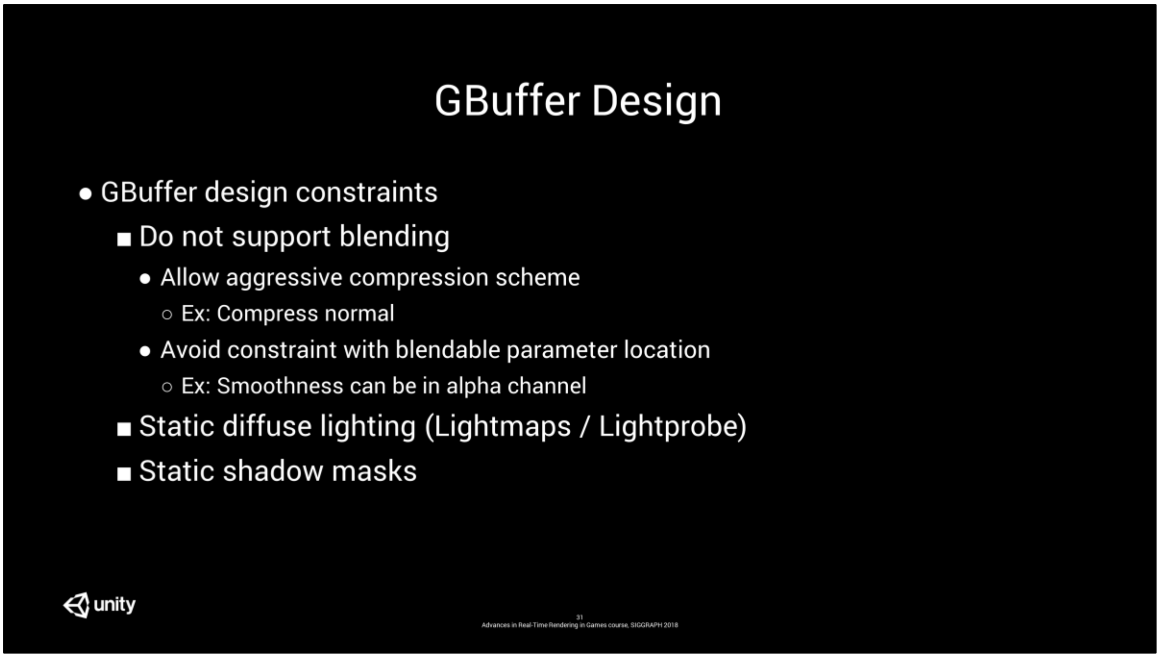
unity

Lighting Architecture

- Classification Performance
  - 1080p PS4 - Deferred Lighting pass

Scene (Time in µs) Simple material	Light count	Deferred Lighting Pass without classification	Deferred Lighting Pass with classification	Gain
Punctual lights	10	664.911	468.459	196.452
Punctual lights	50	1032.637	668.69	363.947
Punctual lights	100	971.958	555.371	416.587
Punctual lights	500	2575.065	1227.77	1347.295
Mixed lights (Punctual, area, environment)	10	758.027	560.903	197.124
Mixed lights	50	935.974	679.763	256.211
Mixed lights	100	1819.571	1302.828	516.743
Mixed lights	500	4973.466	3595.635	1377.831





Static diffuse lighting is sampled during GBuffer pass

GBuffer Design

Standard	R	G	B	A
RT0 RGBA8 sRGB	BaseColor.rgb			Specular Occlusion
RT1 RGBA8	Normal.xy (Octahedral 12/12)			Perceptual Smoothness
RT2 RGBA8	Material Data			FeaturesMask(3) / Material Data
RT3 RGB111110f	Static diffuse lighting			
(Optional) RT4 RGBA8	Extra specular occlusion data		Ambient Occlusion	Light Layering Mask
(Optional) RT5 RGBA8	4 Shadow Masks			

- Ambient occlusion apply on static lighting during GBuffer pass if no RT5
  - This implies double occlusion when combined with SSAO
- Deferred Material classification use RT2 only

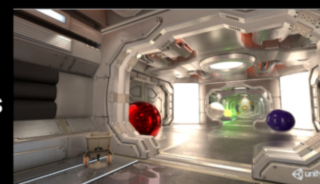
unity

32

Default to 4 RT for XboneOne
Light layering is light linking, mean linking a light to a set of objects, so it only affect those objects.
RT4 can be dynamically allocated. For example it could be enabled only when doing in game cinematics but not during regular gameplay

Features Parity

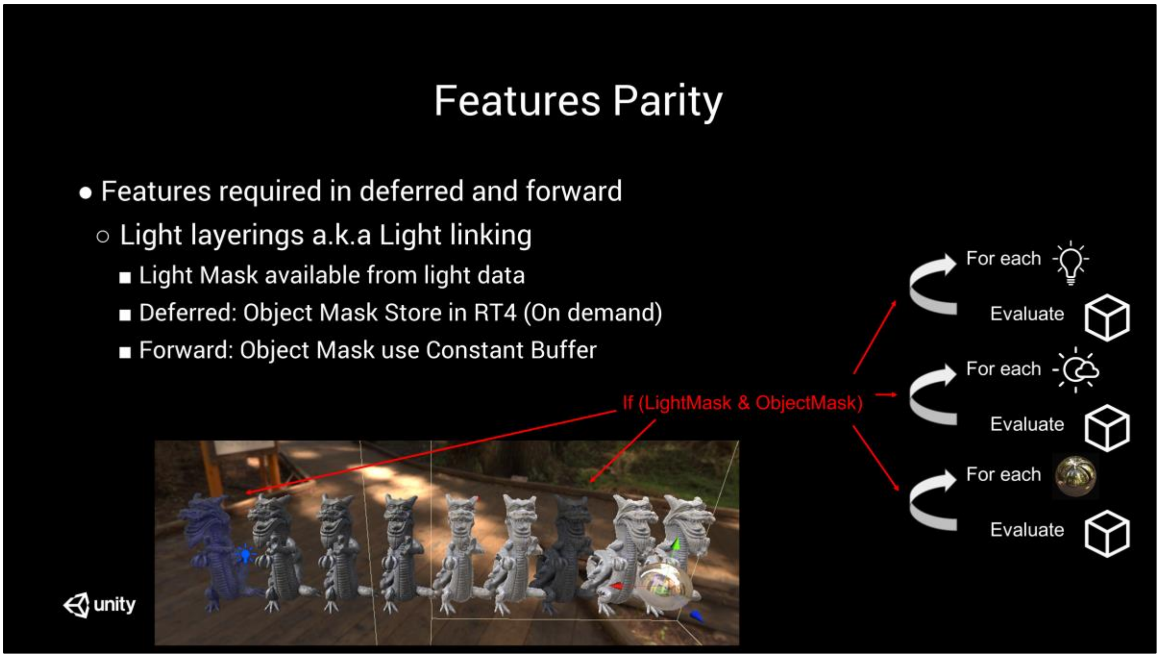
- Engine features often vary with selected rendering path
  - Want SSAO? SSR? - Use deferred path
- HDRP supports a mix of Deferred and Forward Materials
  - Need the same features to be supported
- HDRP is designed to have features parity from the start
  - Allows users to select based on their performance need



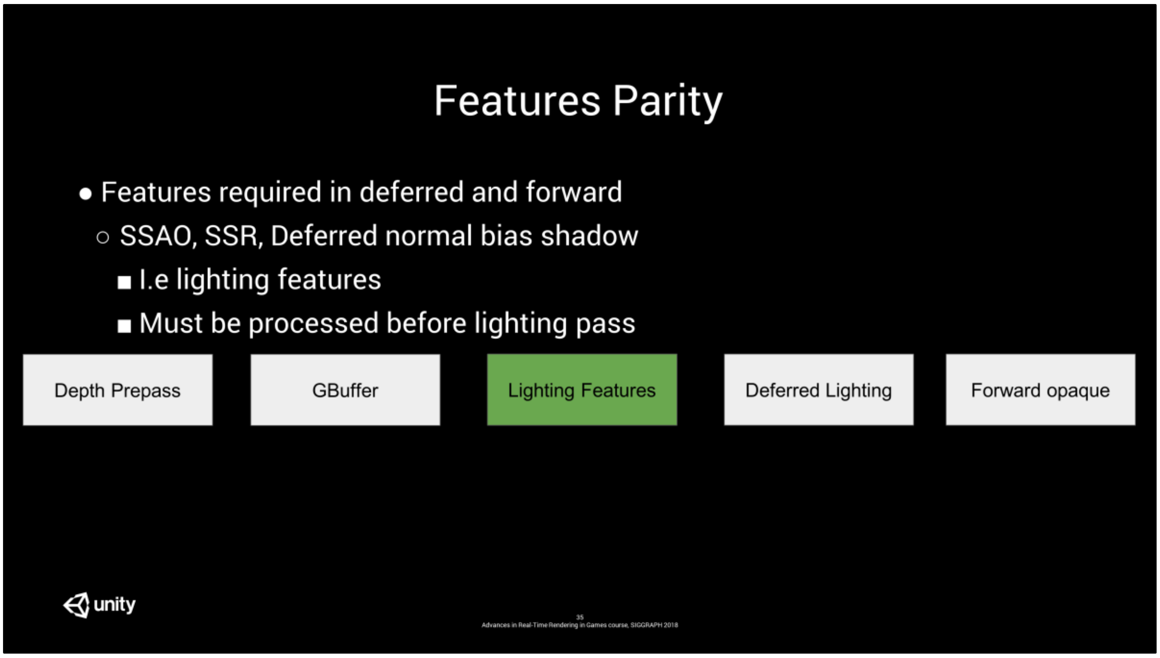
SSR

SSAO

SSR: Screen space reflection
SSAO: Screen space ambient occlusion



Light layering can be enabled per camera, meaning that we allocated an extra RT only when required. Typically for in game cinematic
The blue dragon is affected only by blue light and the grey in the middle of the white is not affected by reflection probe

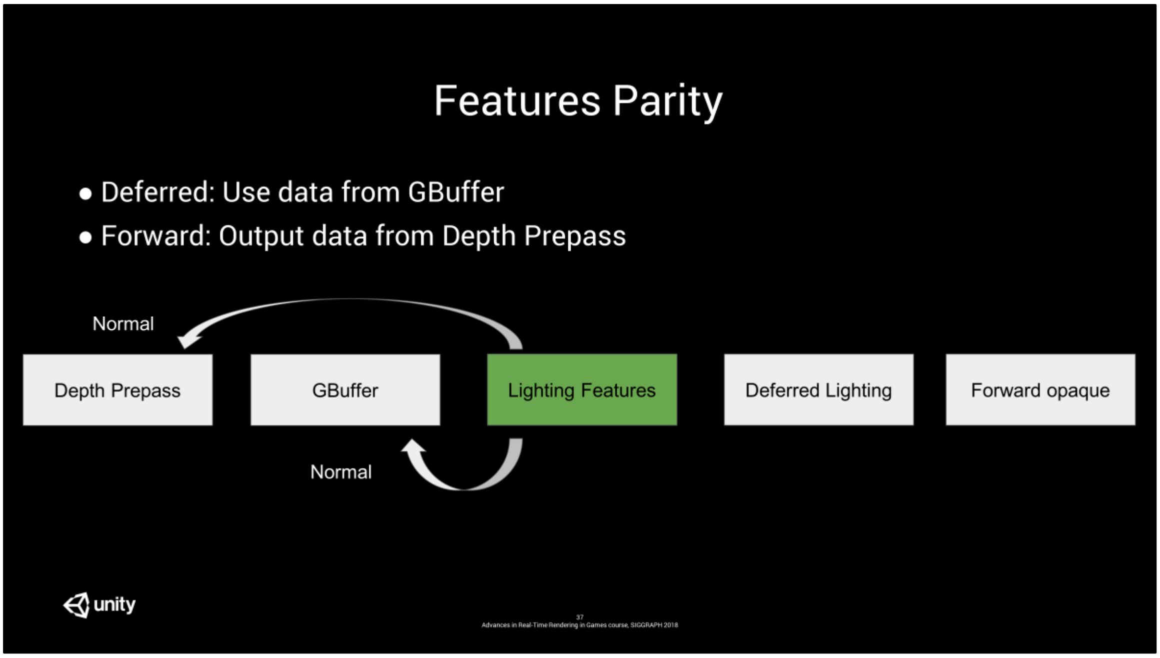


SSR: Screen space reflection
SSAO: Screen space refraction

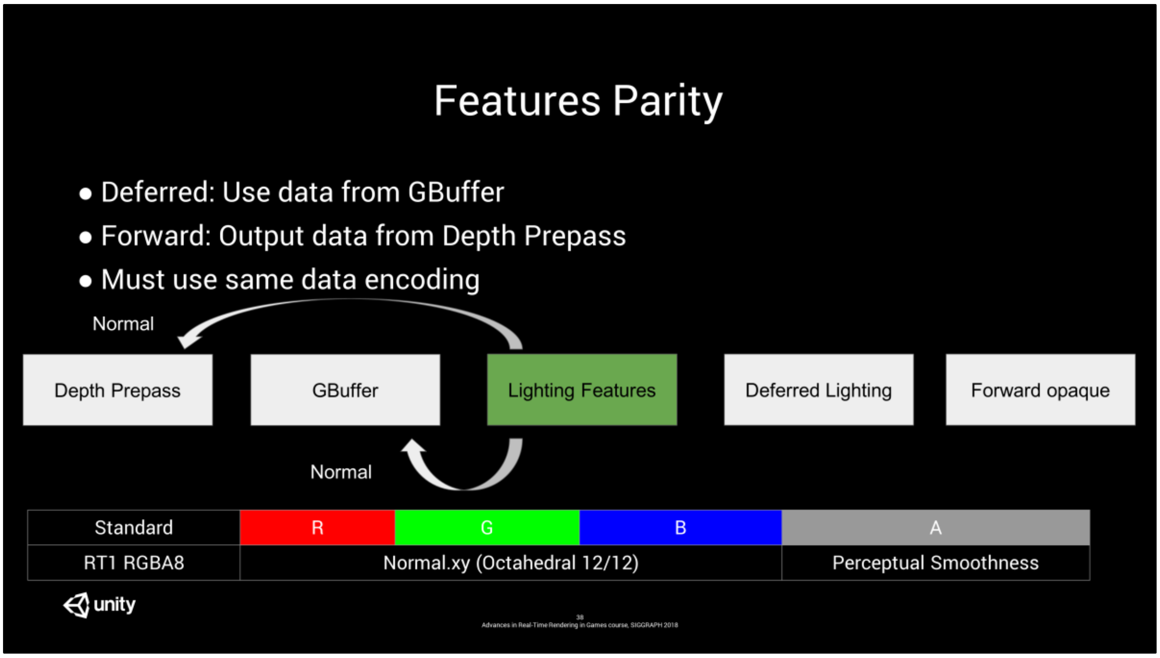
Features Parity

- Deferred: Use data from GBuffer

unity



For forward path we require to output data during depth prepass. Note: for opaque forward material we always perform a depth prepass in HDRP


Features Parity
Deferred: Use data from GBuffer
Forward: Output data from Depth Prepass
Must use same data encoding
Normal
Depth Prepass
GBuffer
Lighting Features
Deferred Lighting
Forward opaque
Normal
Standard
R
G
B
A
RT1 RGBA8
Normal.xy (Octahedral 12/12)
Perceptual Smoothness
unity
38
Advances in Real-Time Rendering in Games course, SIGGRAPH 2018

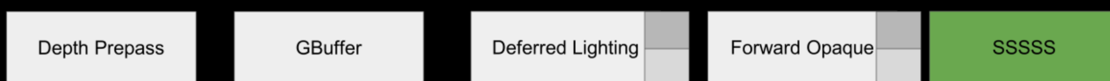
Features Parity

- Features required in deferred and forward
  - Screen-space subsurface scattering (SSSSS)
    - Must be processed after the lighting pass

Depth Prepass | GBuffer | Deferred Lighting | Forward Opaque | SSSSS

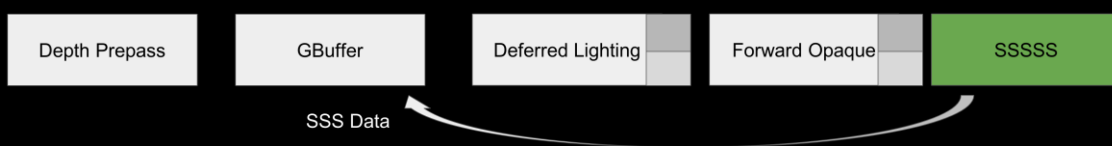
Features Parity

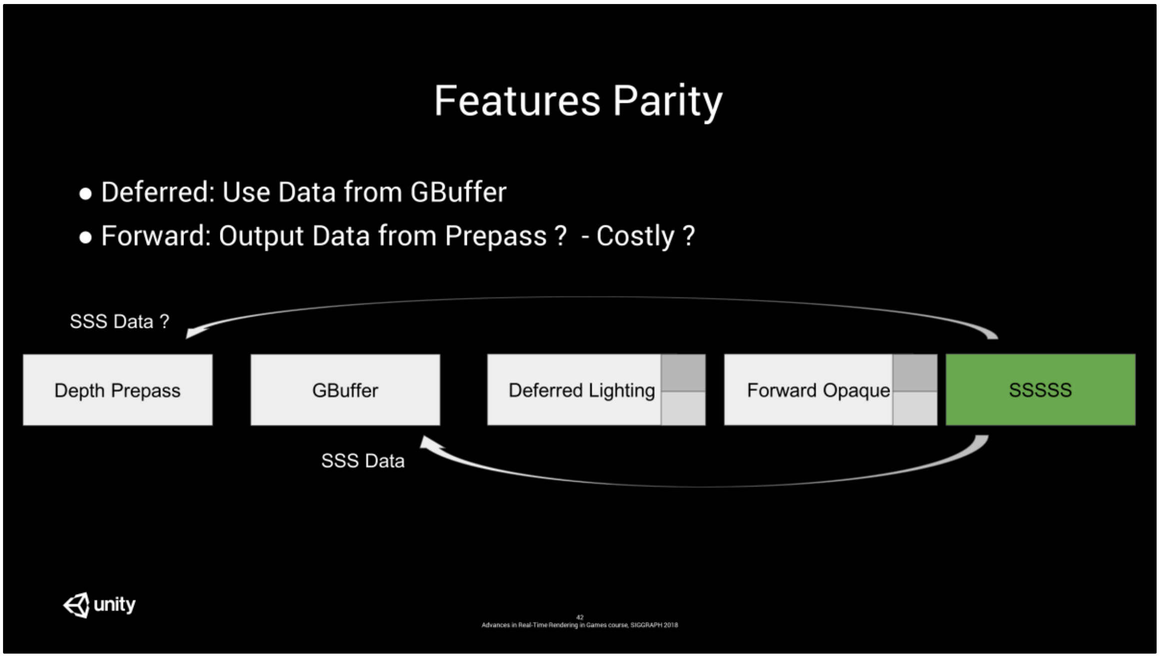
- Features required in deferred and forward
  - Screen-space subsurface scattering (SSSSS)
    - Must be processed after the lighting pass
    - Separate diffuse (RGB111110f) and specular lighting (RGBA16f)



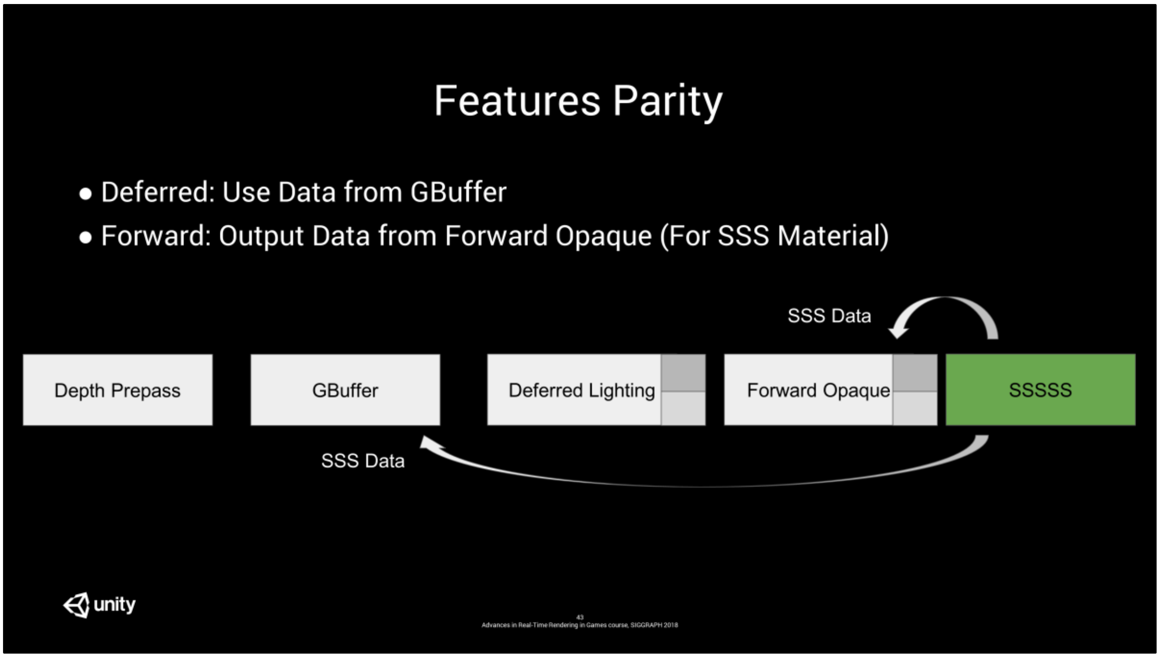
Features Parity

- Deferred: Use Data from GBuffer





Outputing SSS data during prepass will make prepass expensive. We prefer to avoid it.

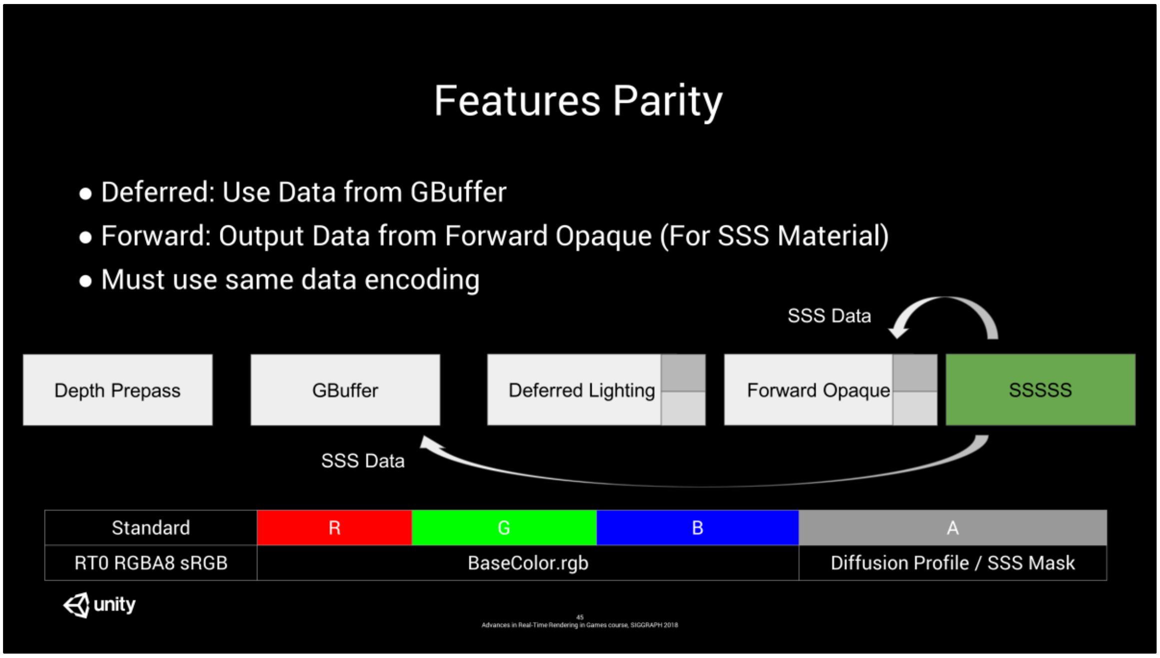


Note: For XBoneOne we aim at keep 4 RT 32 bit. This is what we get. 1RT diffuse, 2 RT specular , 1 RT sss data

Features Parity

- Deferred: Use Data from GBuffer
- Forward: Output Data from Forward Opaque (For SSS Material)
- Must use same data encoding

Standard	R	G	B	A
RT0 RGBA8 sRGB	BaseColor.rgb			Specular Occlusion



Note: This is discuss later but in case of SSS material, we store diffusion profile and SSS Mask and not specular occlusion in RT0, specular occlusion is store in RT2 for SSS material.

Opaque Material Render Pass

- Stencil usage
  - Stencil cleared to 0 at beginning of the frame
  - Deferred Materials tag stencil

Depth Prepass → GBuffer (Regular, SplitLighting) → Deferred Lighting → Forward Opaque → SSSSS

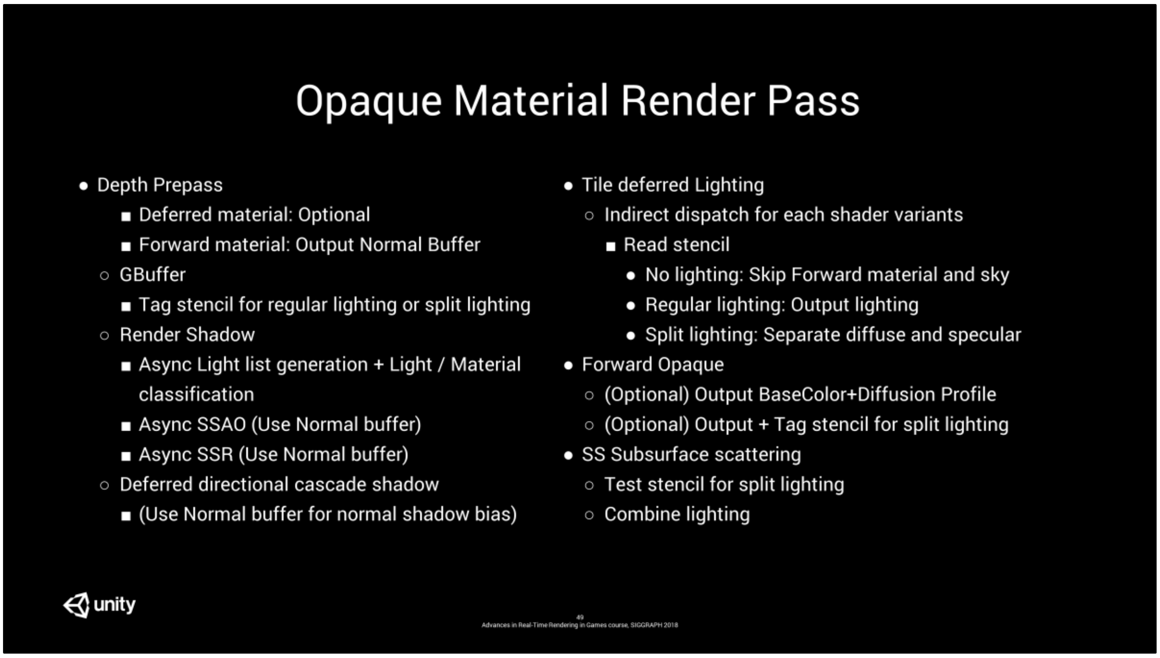
unity

Opaque Material Render Pass

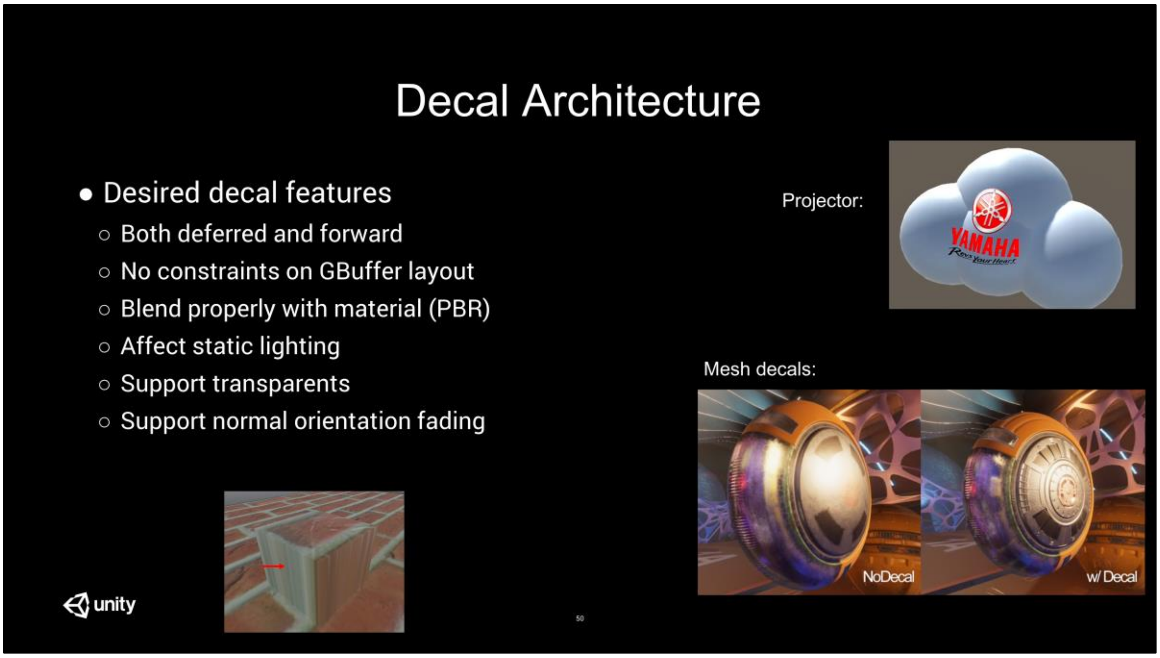
- Stencil usage
  - Deferred Lighting pass
    - Don’t do lighting on Forward Material and Sky
    - SplitLighting done per tile

Opaque Material Render Pass

- Stencil usage
  - Forward Opaque tag stencil for SplitLighting
  - Process SSSSS for SplitLighting tag



Here are all the pass we have speak about, I wont go into the detail of them, if you are interested, slides will be availables after the conferences.



Normal orientation fading is desired by the artists to avoid artifacts when projecting decals along edge that become stretched. Goald is to smoothly out the decal in this case but this require the underlying normal

Decal Architecture

- 3 possible approaches
  - ‘Classic’ deferred decals
    - Blend attributes within GBuffer directly
  - DBuffer (Decal Buffer)
    - Blend attributes into a separate DBuffer
    - Apply DBuffer before lighting in regular pass
  - Cluster decals [Sousa 2016]
    - Decals are like clustered lights
    - Apply before lighting in regular pass

DBuffer is the decal buffer approach use in Unreal engine 4 (There is no presentation about it that I am aware). It is similar to GBuffer but for decals
[Sousa 2016] Tiago Sousa and Jean Geffroy. The devil is in the details: idTech 666.

Features:	Deferred Decal	DBuffer	Cluster Decal
Arbitrary Gbuffer Layout	No	Yes	Yes
Blending Mode	Many (But variant hell)	Lerp	Lerp
Affect static lighting	No	Yes	Yes
Support deferred and forward	No	Yes	Yes
Support Transparent	No	No	Yes
Support Decal Mesh	Yes	Yes	No
Support Normal fading	Yes	No	Yes

No silver bullet!
Note: We output normal buffer for forward material during prepass, so we can’t do normal fading with DBuffer for forward path

Decal Architecture

- HDRP use DBuffer for Opaque Material
  - Requires full depth prepass
  - Render Projector and Mesh Decals before GBuffer

Depth Prepass | DBuffer | GBuffer | Deferred Lighting | Forward opaque

unity

DBuffer Design

- DBuffer approach uses Decal alpha compositing
  - Same as half resolution particles compositing [Cantlay 2007]
- Supports separate attribute blending
  - Opacity per attribute
  - Be careful packing for AO and Metal - Optional support

	R	G	B	A
RT0 RGBA8 sRGB	DiffuseColor.rgb			DiffuseOpacity
RT1 RGBA8	Normal.rgb			NormalOpacity
RT2 RGBA8	Metallic	AO	Smoothness	SmoothnessOpacity
RT3 RG8 (Optional)	MetallicOpacity			AOOpacity

unity

To support separate attribute blending, each attributes need to have an opacity. The DBuffer layout here show how we have packed the attributes and the opacity.
Note: Multiply blend mode is not supported. We use lerp only.

Decal Architecture

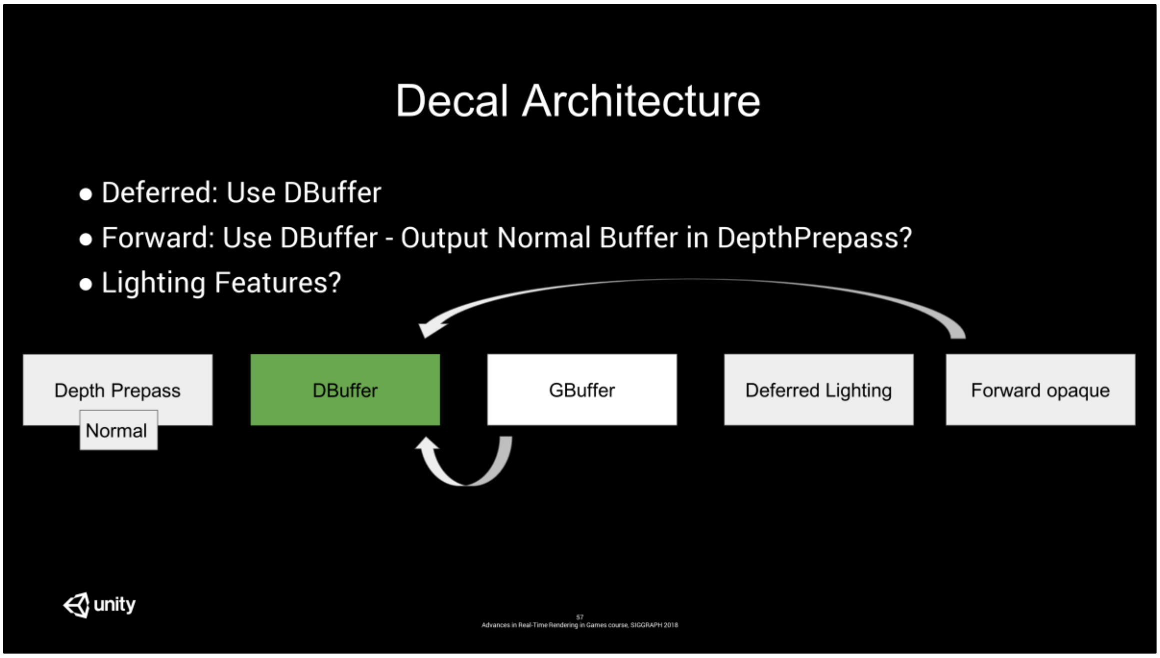
- Deferred: Use DBuffer

Depth Prepass → DBuffer → GBuffer → Deferred Lighting → Forward opaque

Decal Architecture

- Deferred: Use DBuffer
- Forward: Use DBuffer

unity



Remember that for lighting features we use normal buffer during prepass. But in this case the DBuffer don't affect the normal for the lighting features effect

Decal Architecture

- Deferred: Use DBuffer
- Forward: Use DBuffer - Output Normal Buffer in DepthPrepass
- Patch Normal Buffer after DBuffer

Decal Architecture

- Deferred: Use DBuffer
- Forward: Use DBuffer - Output Normal Buffer in DepthPrepass
- Patch Normal Buffer after DBuffer
- Stencil use to optimize patch Normal buffer

Depth Prepass
Tag DecalNormal

DBuffer
Tag Decal

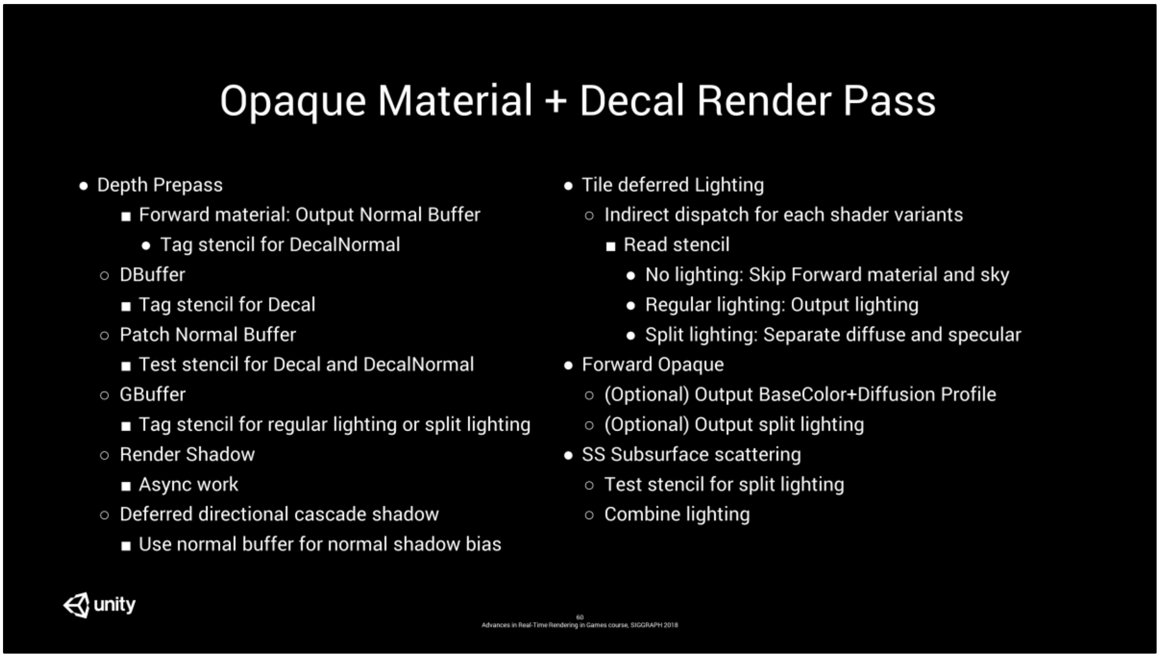
Patch Normal Buffer
Test Decal and DecalNormal

GBuffer

Deferred Lighting

Forward opaque

unity



Here are all the pass we have speak about, I wont go into the detail of them, if you are interested, slides will be availables after the conferences.
Note: now depth prepass is mandatory.
Reduce all aysnc work (SSR, SSAO), in label async work.

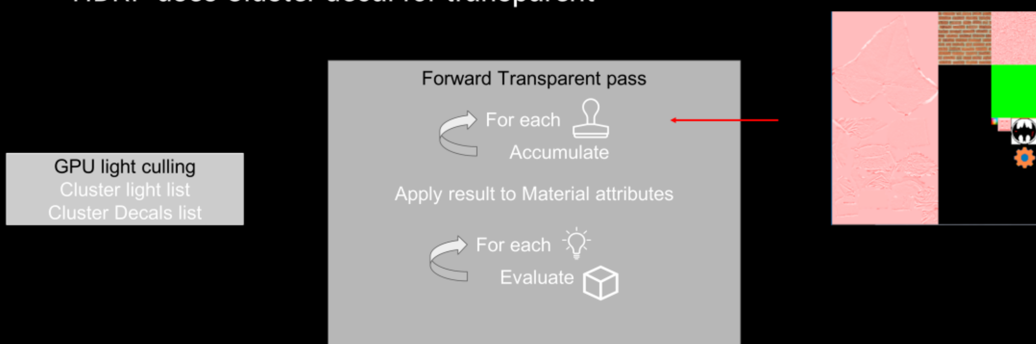
This is our Render Frame allowing features parity betweend deferred and forward.

Decal Architecture

- HDRP uses Cluster decal for transparent
  - Projector only
  - Optional
  - Supports separate attribute blending
  - Cluster Decal list prepared like Cluster Light list
  - Gather textures in one Atlas

Decal Architecture

- HDRP uses Cluster decal for transparent



Decal Architecture

- HDRP uses Cluster decal for transparent
  - Accumulate Decals modification in a loop
    - Can’t use regular UVs for mip
    - Use world space position derivative instead

```
// get world space ddx/ddy for adjacent pixels to be used later in mipmap lod calculation
float3 positionRWSDdx = ddx(positionRWS);
float3 positionRWSDdy = ddy(positionRWS);

for (uint i = 0; i < decalCount; i++)
{
    DecalData decalData = FetchDecal(decalStart, i);

    // need to compute the mipmap LOD manually because we are sampling inside a loop
    float3 positionDSDdx = mul(worldToDecal, float4(positionRWSDdx, 0.0)).xyz;
    float3 positionDSDdy = mul(worldToDecal, float4(positionRWSDdy, 0.0)).xyz;

    float2 sampleDiffuseDdx = positionDSDdx.xz * decalData.diffuseScaleBias.xy; // factor in the atlas scale
    float2 sampleDiffuseDdy = positionDSDdy.xz * decalData.diffuseScaleBias.xy;
    float lodDiffuse = ComputeTextureLOD(sampleDiffuseDdx, sampleDiffuseDdy, _DecalAtlasResolution);
```

unity

Note that to sample the correct mip with cluster decal it is not trivial, we can used the derivative of the position convert to decal space. Take care of the atlas coordinate.

Decal Architecture

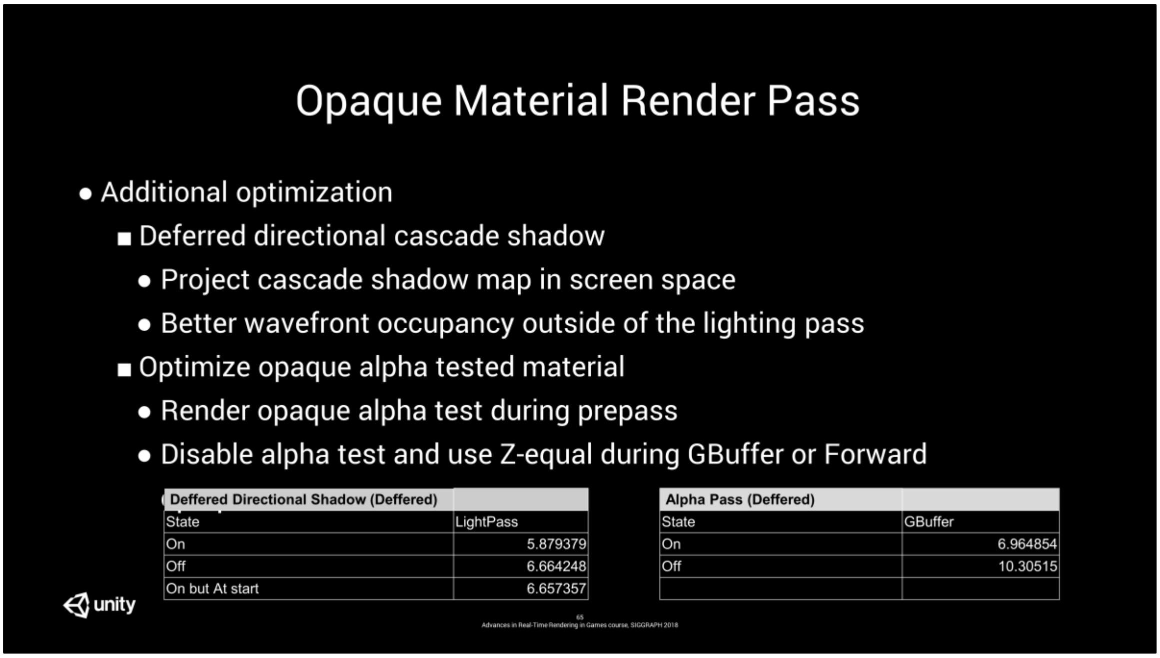
- GPU decals Performance number (Simple scene to highlight entry cost)

Number of Decals	DBuffer pass	GBuffer pass	Forward Transparent pass using Atlas
0 (Decals off)	0	0.557769	2.87549
0	0.018079	0.91711	3.217145
10	0.156309	0.930636	3.267441
60	0.31802	0.93493	3.488841
100	0.450224	0.935481	3.601721
200	0.774024	0.930832	4.030897
400	1.40057	0.914649	4.767138
800	2.675754	0.905647	X (not supported)

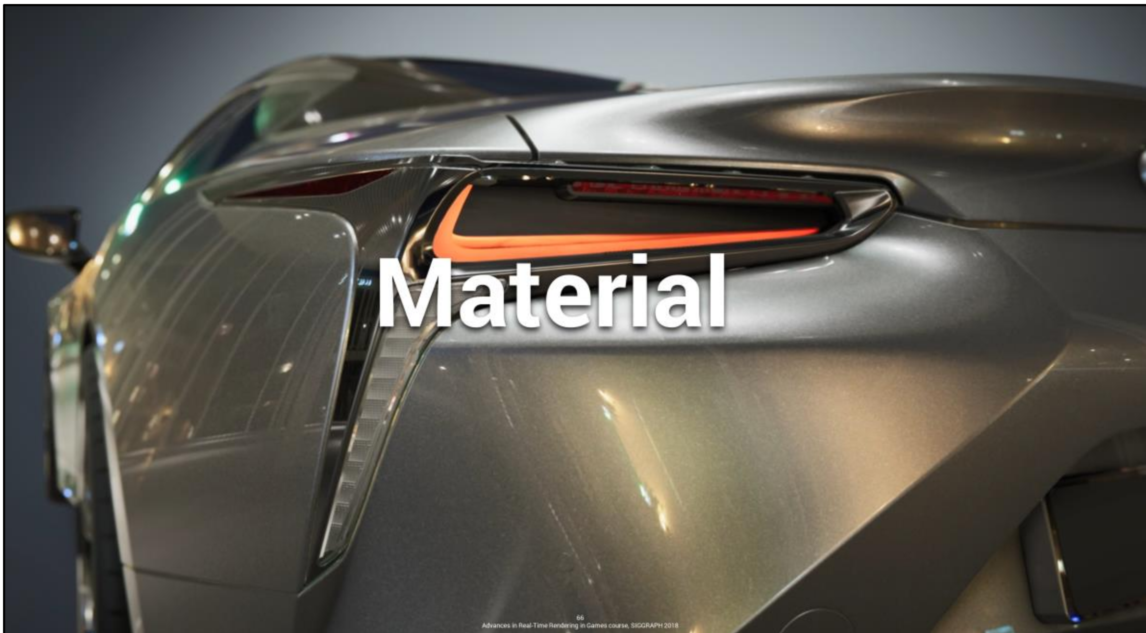
unity

Decals off mean with have remove the decals code. Goal is to measure the overhead induce by the DBuffer and cluster approach when there is 0 decals and when there is no decal code.
As we can see currently with our approach there is extra cost induced that is non negligeable. But then it scale well. Transparent material can chose to receive or not deals.
For DBuffer approache we perform an extra step of ‘decal classification’ that save a bit of performance.

These measurement have been done on a scene compose of multiple simple objects. Mean the decals that require to fetch multiple textures hurt a lot. With real world scene with complex layering where the material have plenty of ALU and fetch severals tetures already the difference between decals off and decals 0 is way lower. Also the extra cost show here is for the whole scene.



The test scene for performance number are a typical area of our demo Fontainebleau with various foliage and tree + a complex layered ground with some tessellation.


Material

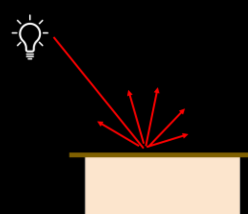
HDRP BRDF

- Lit shader
  - Default shader of HDRP
  - Deferred Material (Can switch to Forward Material)
  - Sum of material features

unity

HDRP BRDF

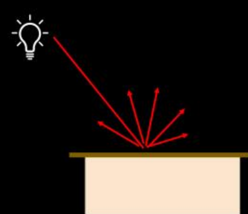
- Lit shader
  - Diffuse term: Burley’s diffuse a.ka. Disney Diffuse [Burley 2012]
  - BaseColor/Metallic parametrization



unity

HDRP BRDF

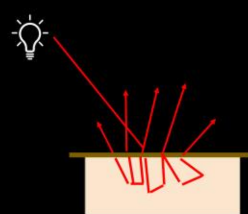
- Lit shader
  - Diffuse term: Burley’s diffuse a.ka. Disney Diffuse [Burley 2012]
  - Or DiffuseColor/SpecularColor parametrization



unity

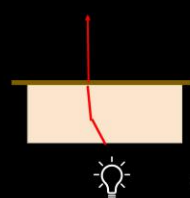
HDRP BRDF

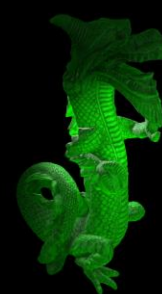
- Lit shader
  - SSS term: Disney SSS



HDRP BRDF

- Lit shader
  - Translucent term: based on Disney diffuse

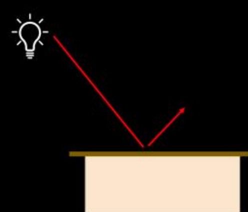




unity

HDRP BRDF

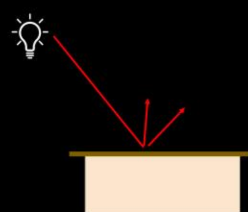
- Lit shader
  - Specular term: Multiscattering Isotropic GGX [Heitz 2016]



unity

HDRP BRDF

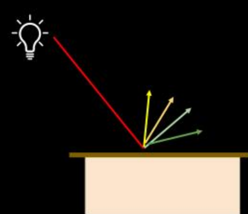
- Lit shader
  - Specular term: Multiscattering Anisotropic GGX [Heitz 2014]
    - A hack



unity

HDRP BRDF

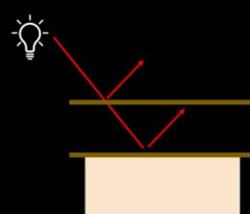
- Lit shader
  - Iridescence term: Replace Fresnel term [Belcour 2017]



unity

HDRP BRDF

- Lit shader
  - Clear coat specular term: Multiscattering Isotropic GGX
    - A hack



unity

HDRP BRDF

- Lit shader
  - Material ID in HDRP: Bitmask of Material Features
  - Ex: Standard + Translucency
  - Ex: Standard + ClearCoat + Anisotropy
  - Ex: Standard + Iridescence + Subsurface scattering
- GBuffer constrain
  - Exclusive Material features due to storage space
  - Iridescence, Anisotropy and Subsurface scattering / Translucency

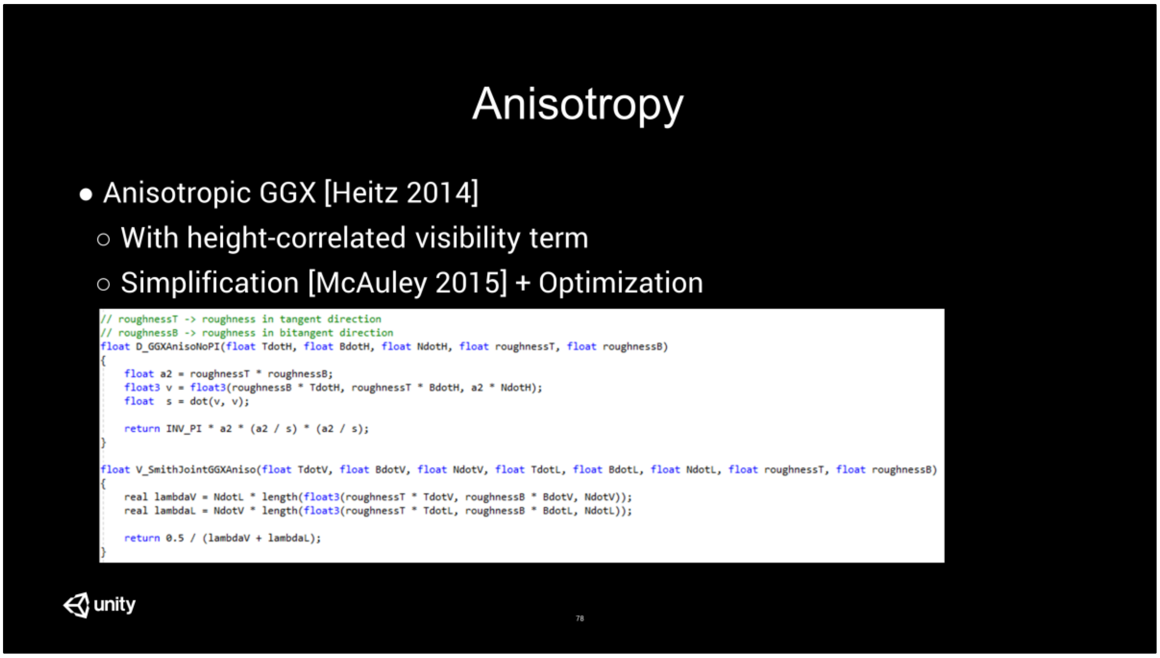
GBuffer Design

- Standard

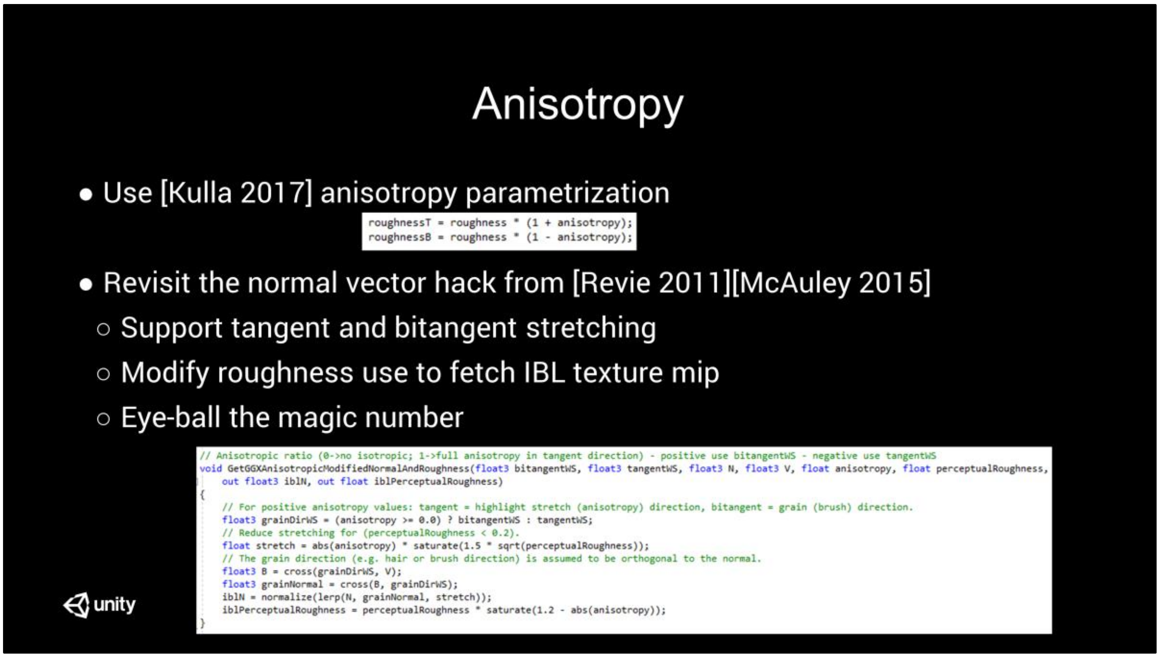
Standard	R	G	B	A
RT0 RGBA8 sRGB	BaseColor.rgb			Specular Occlusion
RT1 RGBA8	Normal.xy (Octahedral 12/12)			Perceptual Smoothness
RT2 RGBA8	Fresnel0.rgb			FeaturesMask(3) / CoatMask(5)

- Clear coat available with all variants
- No metallic – Decompress to Fresnel0
  - Optimization + Handle both parametrization (Metallic / Specular color)

unity

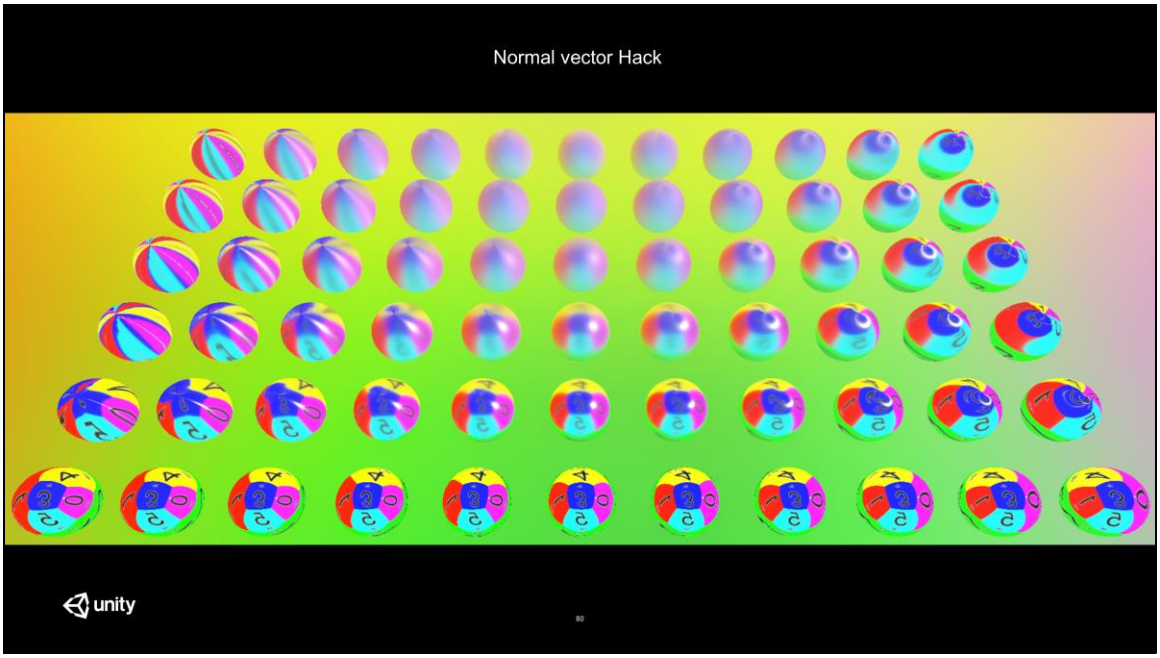


[Heizt 2014] Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs
[McAuley15] S. McAuley, The rendering of far cry 4, Cedec 2015

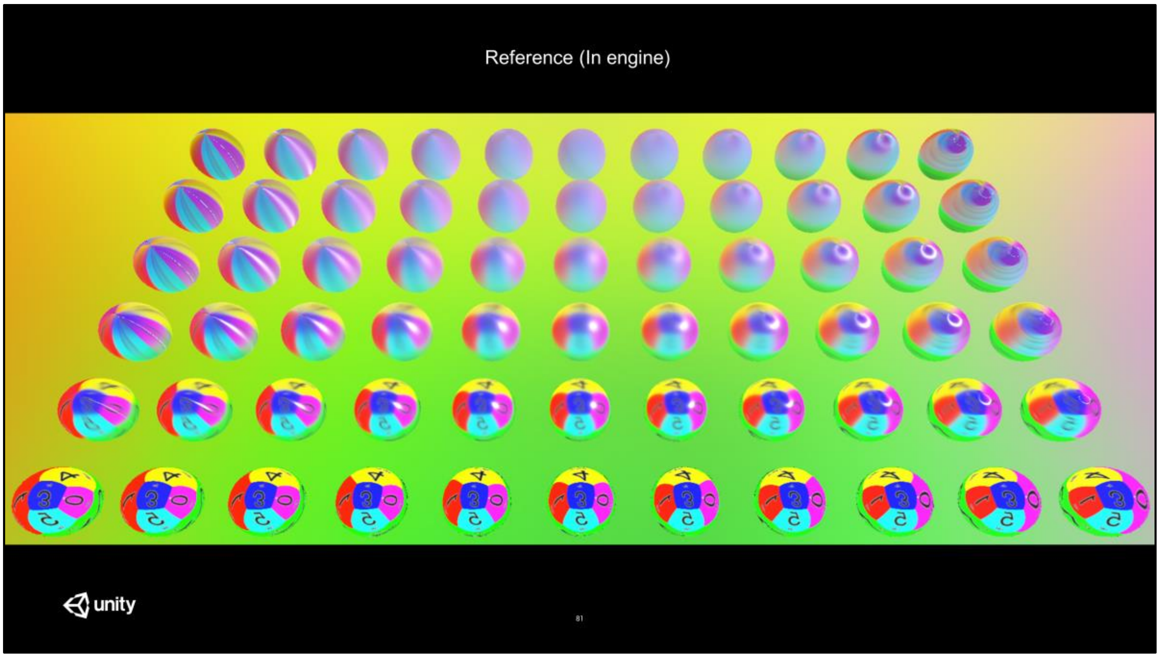


[Revie11] D. Revie, Implementing Fur Using Deferred Shading, GPU Pro 2
[McAuley15] S. McAuley, The rendering of far cry 4, Cedec 2015

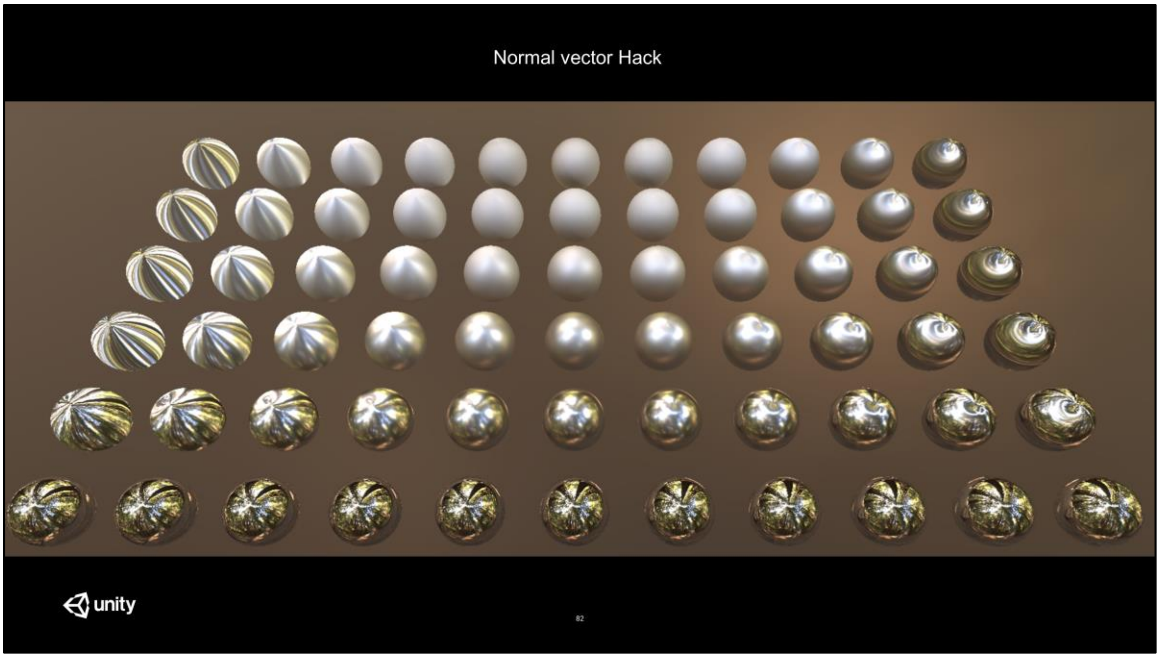
Hack purely empirical :)



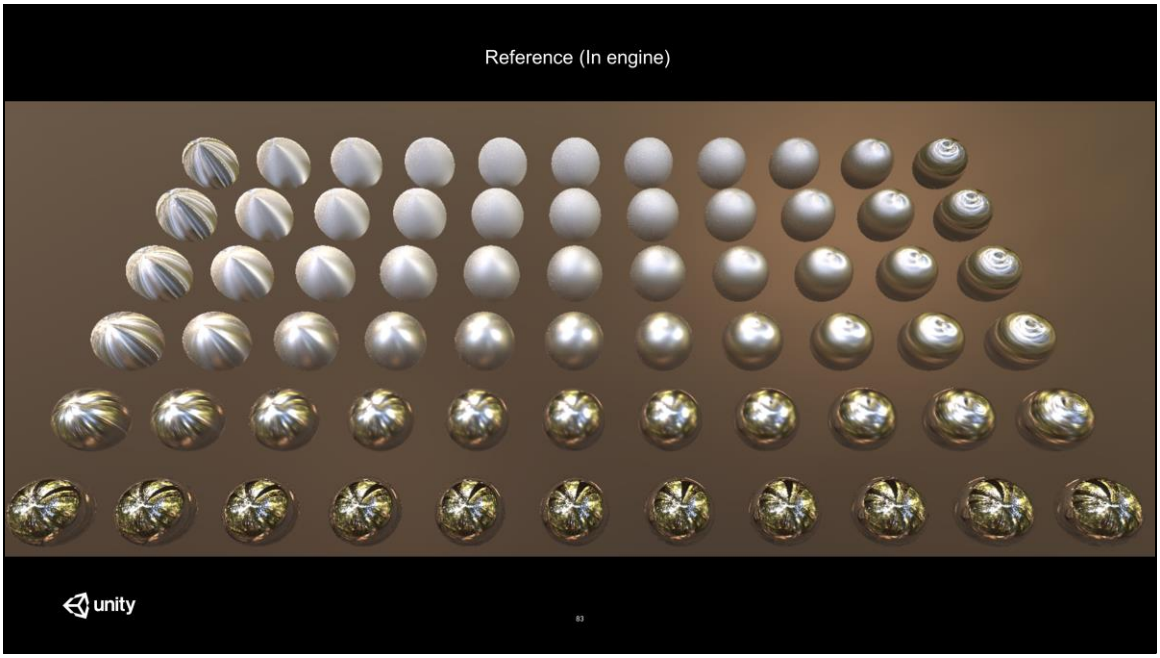
Use this calibration cubemap to check the stretching
Anisotropy left to right: -1 to 1
Perceptual Smoothness bottom to top: 1 to 0



Looks rather incorrect, but with high frequency cubemap….



Looks ok and there is almost something that (from far) could looks like reference...



in the future we would like at anisotropic filtering instead of this hack.

GBuffer Design

- Anisotropy

Anisotropy	R	G	B	A
RT0 RGBA8 sRGB	BaseColor.rgb			Specular Occlusion
RT1 RGBA8	Normal.xy (Octahedral 12/12)			Perceptual Smoothness
RT2 RGBA8	Anisotropy	Tangent frame angle(11) / Metallic(5)		FeaturesMask(3) / CoatMask(5)

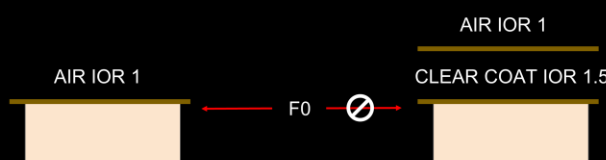
- Use the angle between the actual tangent frame and a default one

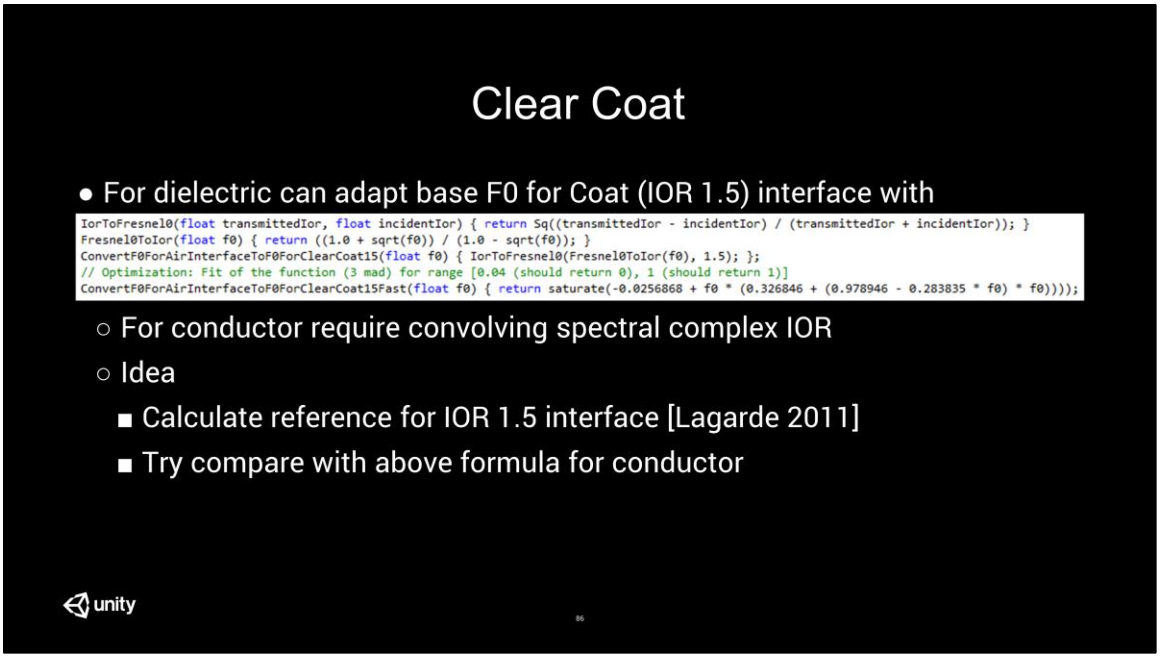
```
// ENCODE tangent frame
// Reconstruct the default tangent frame.
float3x3 frame = GetLocalFrame(normalWS);
// Compute the rotation angle of the actual tangent frame with respect to the default one.
float sinFrame = dot(tangentWS, frame[1]);
float cosFrame = dot(tangentWS, frame[0]);
uint  storeSin = abs(sinFrame) < abs(cosFrame) ? 4 : 0;
uint  quadrant = ((sinFrame < 0) ? 1 : 0) | ((cosFrame < 0) ? 2 : 0);
// sin [and cos] are approximately linear up to [after] 45 degrees.
float sinOrCos = min(abs(sinFrame), abs(cosFrame)) * sqrt(2);
outGBuffer2.rgb = float3(surfaceData.anisotropy * 0.5 + 0.5, sinOrCos,
                         PackFloatInt8bit(metallic, storeSin | quadrant, 8));
```

```
// DECODE tangent frame
float unused;
uint tangentFlags;
UnpackFloatInt8bit(inGBuffer2.b, 8, unused, tangentFlags);
// Get the rotation angle of the actual tangent frame with respect to the default one.
uint  quadrant = tangentFlags;
uint  storeSin = tangentFlags & 4;
float sinOrCos = inGBuffer2.g * rsqrt(2);
float cosOrSin = sqrt(1 - sinOrCos * sinOrCos);
float sinFrame = storeSin ? sinOrCos : cosOrSin;
float cosFrame = storeSin ? cosOrSin : sinOrCos;
sinFrame = (quadrant & 1) ? -sinFrame : sinFrame;
cosFrame = (quadrant & 2) ? -cosFrame : cosFrame;
// Rotate the reconstructed tangent around the normal.
tangentWS = sinFrame * frame[1] + cosFrame * frame[0];
bitangentWS = cross(frame[2], frame[0]);
```

Clear Coat

- Not physical - Simplified approach
  - Isotropic GGX on top of base
  - Fixed index of refraction (IOR) of 1.5 (F0 of 0.04) and roughness of 0.03
  - Calculate base specular with F0 accounting for coat interface
    - Game used Schlick Fresnel with input F0 (i.e Specular Color)
      - F0 in game is calculated with air (IOR 1) interface





[Lagarde 2011] Sébastien Lagarde. Feeding a physically based shading model.
F0 is Fresnel0 i.e reflectance at incident angle.

Clear Coat

- Given that the cost it is not so bad for runtime perf
  - Error increases with lower value
  - Use this approach to update base F0 when clear coat enabled

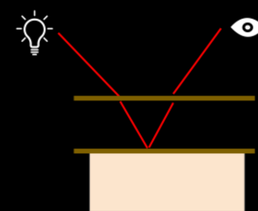
	F0 for air interface			F0 for IOR 1.5 interface			IorToFresnel0(Fresnel0ToIor(fresnel0), 1.5)		
	R	G	B	R	G	B	R	G	B
Silver	0.971519	0.959915	0.915324	0.960358	0.945675	0.893812	0.957583	0.940477	0.87569
Aluminium	0.913183	0.921494	0.924524	0.874831	0.887415	0.893018	0.872618	0.884564	0.888933
Gold	1	0.765557	0.336057	1	0.727192	0.232578	Div 0	0.669332	0.184471
Chromium	0.549585	0.556114	0.554256	0.415606	0.423679	0.429685	0.403956	0.411383	0.409266
Copper	0.955008	0.637427	0.538163	0.9342	0.549144	0.438454	0.933273	0.507081	0.391058

Relative error (%)	R	G	B
Silver	0.288954744	0.5496602956	2.027495715
Aluminium	0.2529631437	0.3212702062	0.457437588
Gold		7.956633186	20.68424357
Chromium	2.803135662	2.902197182	4.75208583
Copper	0.09922928709	7.659739522	10.80979989

unity

Clear Coat

- For all light types
  - Calculate base specular with F0 accounting for coat interface
  - Calculate Fresnel term for clear coat
  - Apply approximate energy conservation on base
  - Add coat specular contribution to base specular



```
baseF0 = ConvertF0ForAirInterfaceToF0ForClearCoat15Fast(fresnel0);
(...) // BaseSpecular calculation

float coatF = F_Schlick(CLEAR_COAT_F0, LdotH);
baseSpecular *= Sq(1.0 - coatF); // Scale base specular for ingouing and outgoing interface crossing
float DV = DV_SmithJointGGX(NdotH, NdotL, NdotV, CLEAR_COAT_ROUGHNESS);
baseSpecular += coatF * DV;
```

- IBL: Use Schlick Fresnel with NdotV + Extra IBL fetch

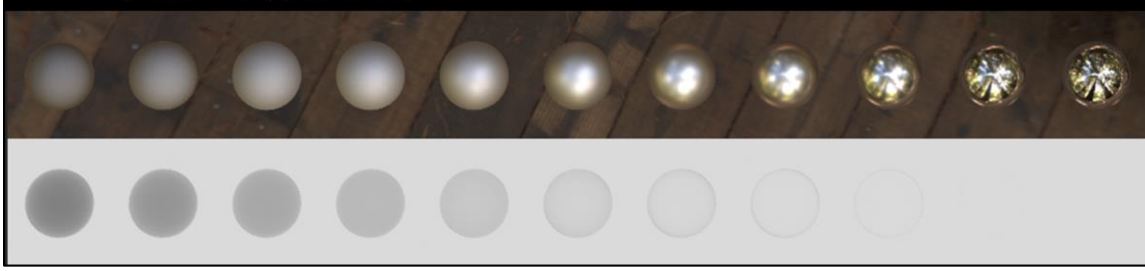
unity

Area Light use an extra LTC calculation
In the future we will use a more physical way based on our stacklit shader approach


unity
Rendered in real-time with Unity

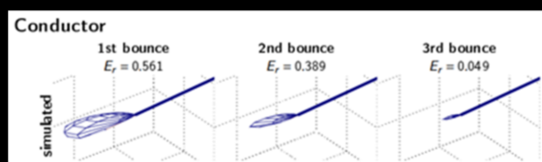
Multiscattering GGX

- Improve energy conservation
  - Lack of multi-scattering in GGX formulation
    - Up to 60% light lost (rough case) on furnace test (uniform HDRI)
- Current trend is to approximate with an added compensation term
  - [Kulla 2017] [Hill 2018]



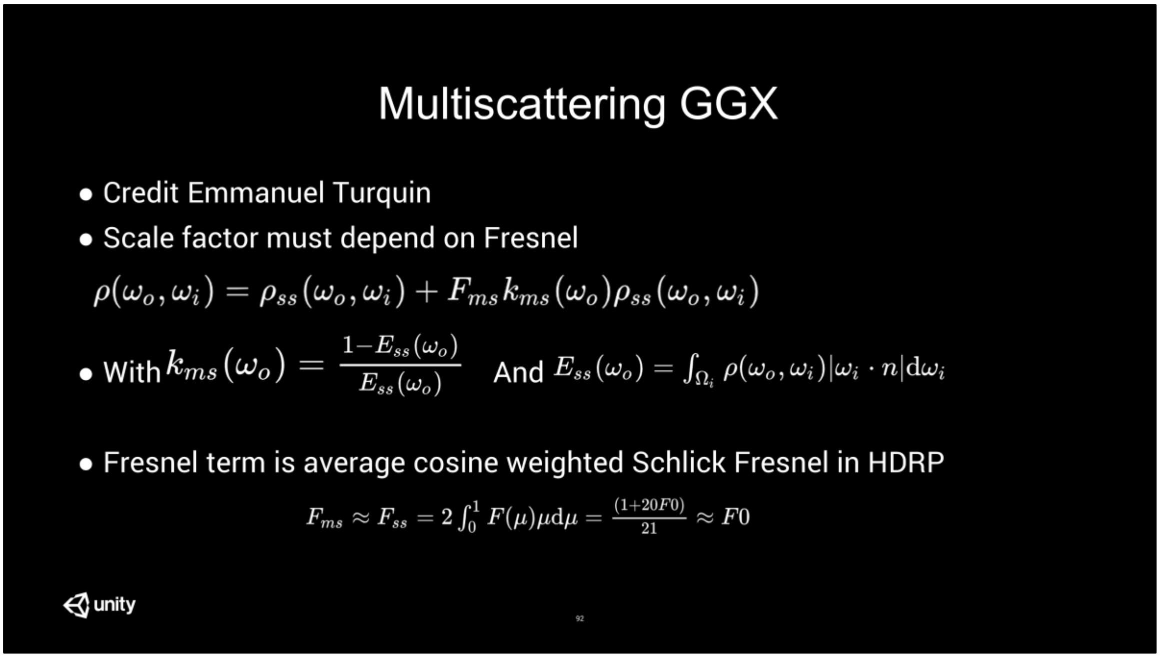
Multiscattering GGX

- [Heitz 2016] provided ground truth behavior
  - Rougher is more saturated for both diffuse and specular
  - Simulation show that bounce lobes are similar



    - Implies that the scale factor on single scattering GGX is enough

[Heitz 2016] Eric Heitz, Johannes Hanika, Eugene d’Eon and Carsten Dachsbacher
Multiple-Scattering Microfacet BSDFs with the Smith Model



F_{ms}\approx F_{ss}=2\int_{0}^{1}F(\mu)\mu \mathrm{d}{\mu } = \frac{(1 + 20F0)}{21}\approx F0
E_{ss}({\omega_o})=\int_{\Omega_i} \rho({\omega_o}, {\omega_i}) |{\omega_i \cdot n}| \mathrm{d}{\omega_i}
k_{ms}({\omega_o})= {\frac{1 - E_{ss}({\omega_o})}{E_{ss}({\omega_o})}}
\rho (\omega_o, \omega_i) = \rho_{ss} (\omega_o, \omega_i) + F_{ms}k_{ms}(\omega_o) \rho_{ss} (\omega_o, \omega_i)

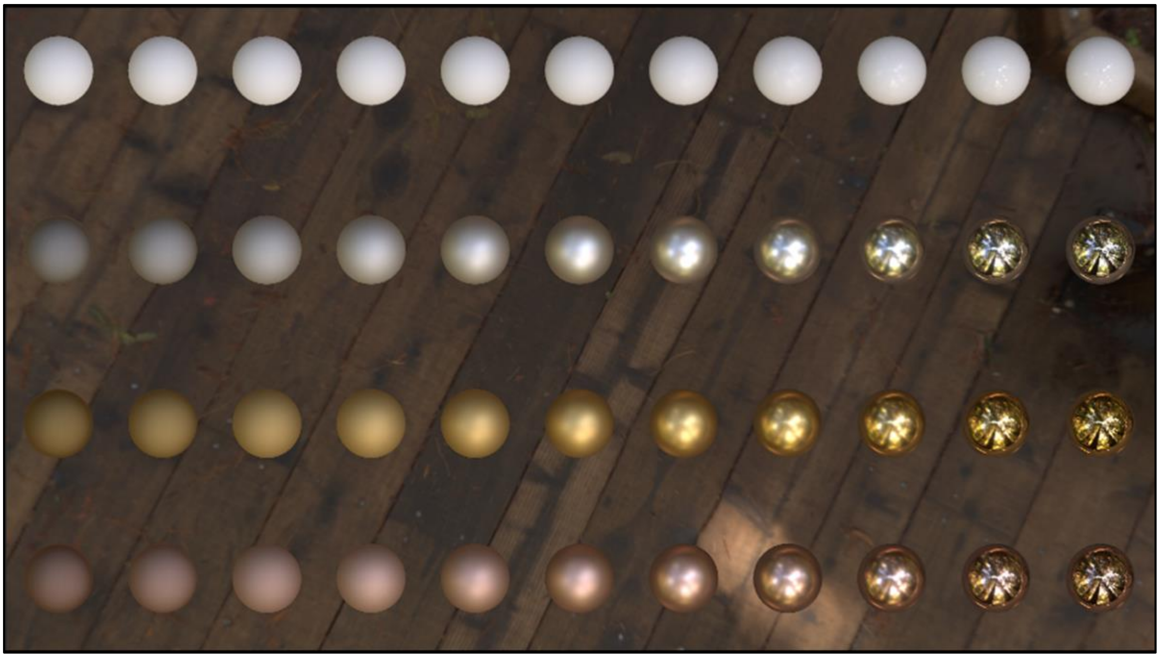
Multiscattering GGX

- Apply factor at end of lightloop on both direct and indirect specular
  - Work as it is a scale of original GGX lobe
  - Store $E_{ss}(\omega_o)$ in a texture. Share with cubemap preintegration.

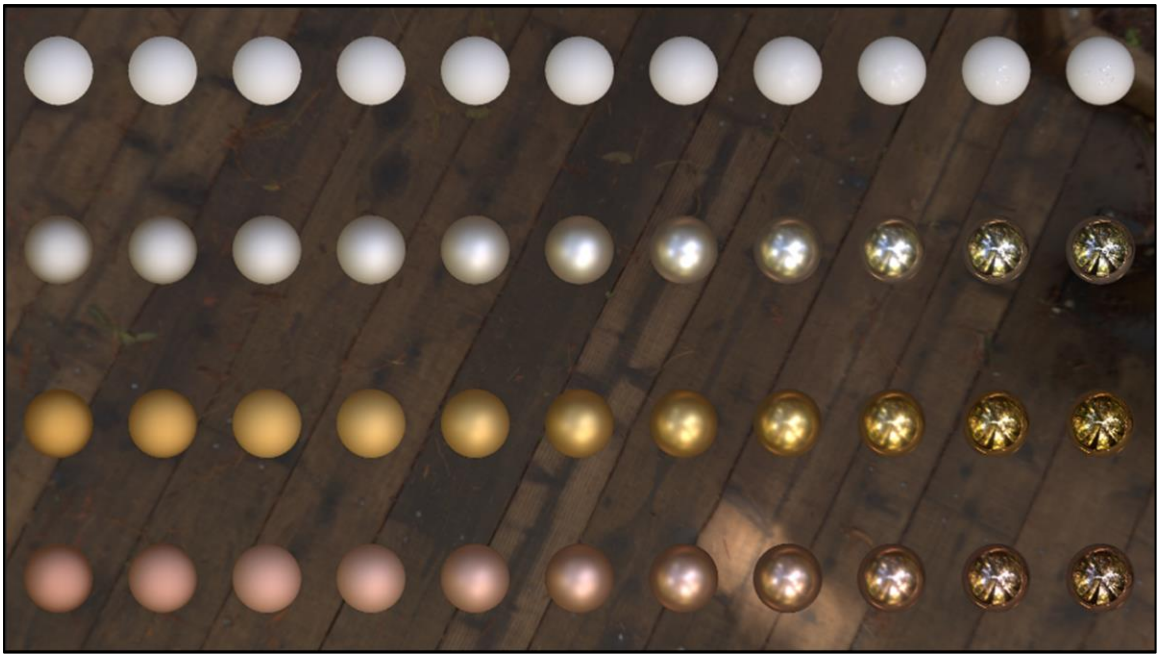
```
specularLighting = lighting.direct.specular + lighting.indirect.specularReflected;
// y = Integral{(BSDF / F) * <N,L> dw}
float3 preFGD = SAMPLE_TEXTURE2D_LOD(_PreIntegratedFGD_GGXDisneyDiffuse, s_linear_clamp_sampler, float2(NdotV, perceptualRoughness), 0).xyz;
float reflectivity = preFGD.y;
// Rescale the GGX to account for the multiple scattering.
specularLighting *= 1.0 + fresnel0 * ((1.0 / reflectivity) - 1.0);
```

- No multiscattering for Diffuse term
  - Disney diffuse is empirical and not energy-conserving
    - No darkening with increased roughness

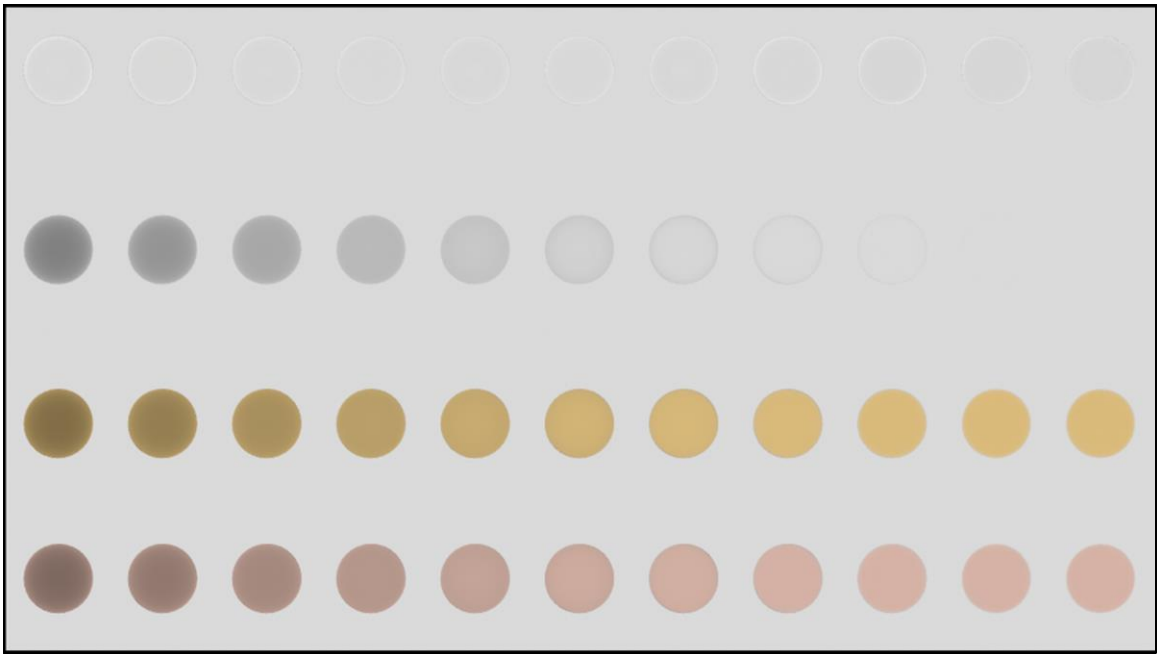
unity



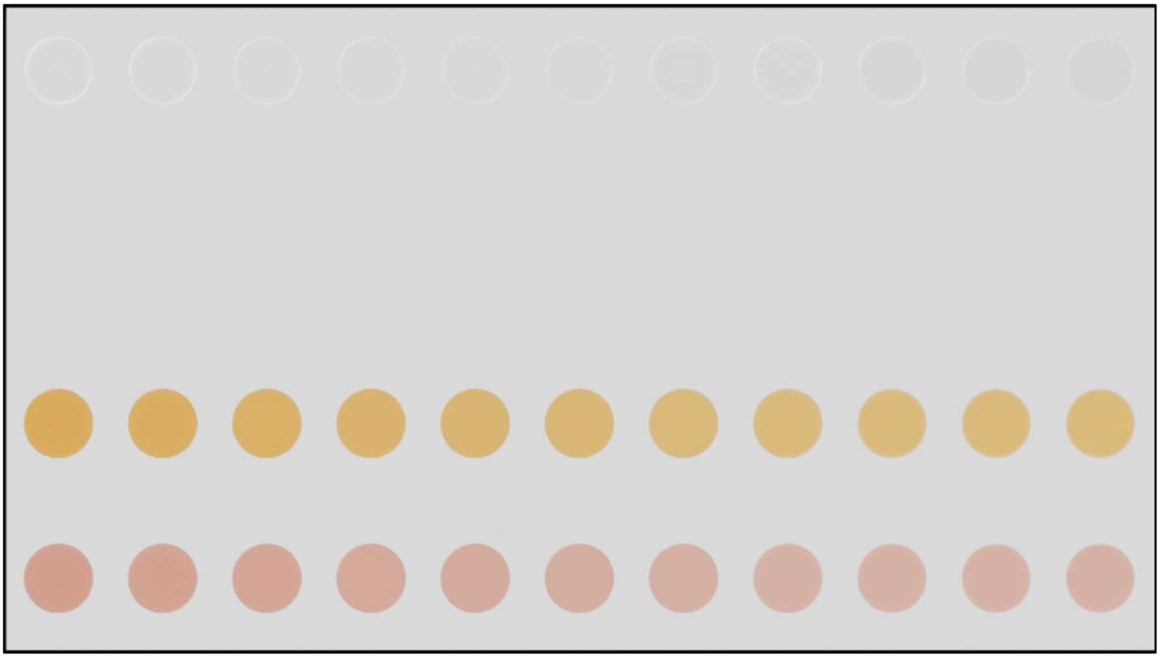
No multiple scattering
1st colum is dieletric pure white - Diffuse term here is Disney diffuse
2nd colum is F0 = 1 - Conductor
3nd is F0 = gold = 1 - Conductor
3nd is F0 = copper = 1 - Conductor



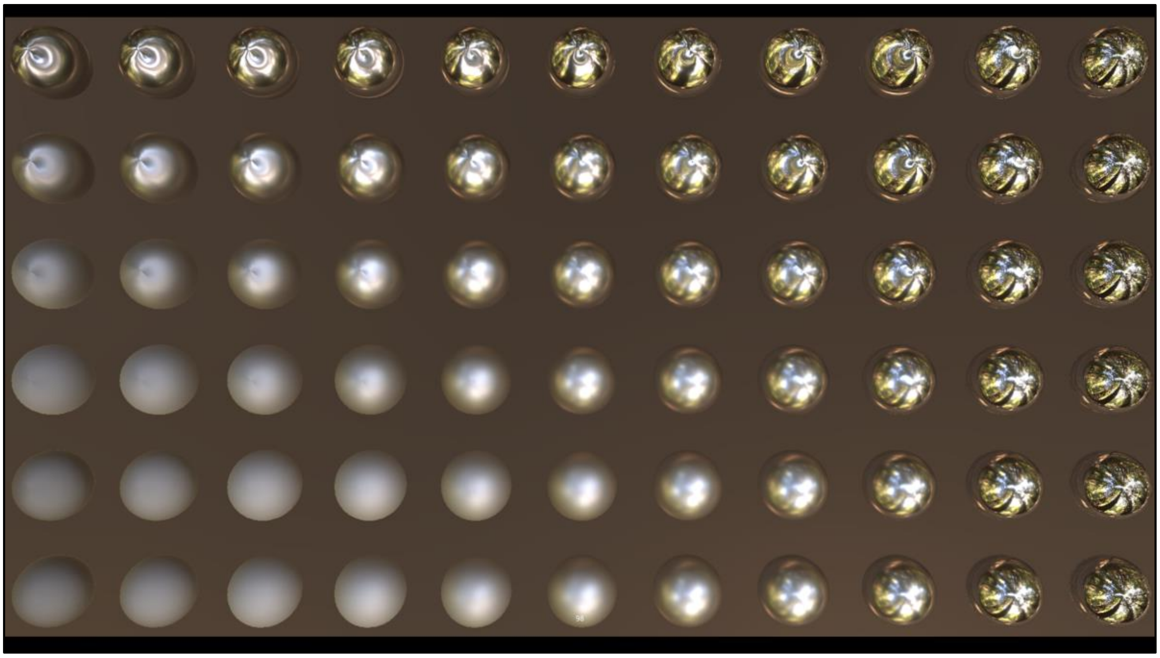
With multiple scattering
1st colum is dieletric pure white - Diffuse term here is Disney diffuse
2nd colum is F0 = 1 - Conductor
3nd is F0 = gold = 1 - Conductor
3nd is F0 = copper = 1 - Conductor



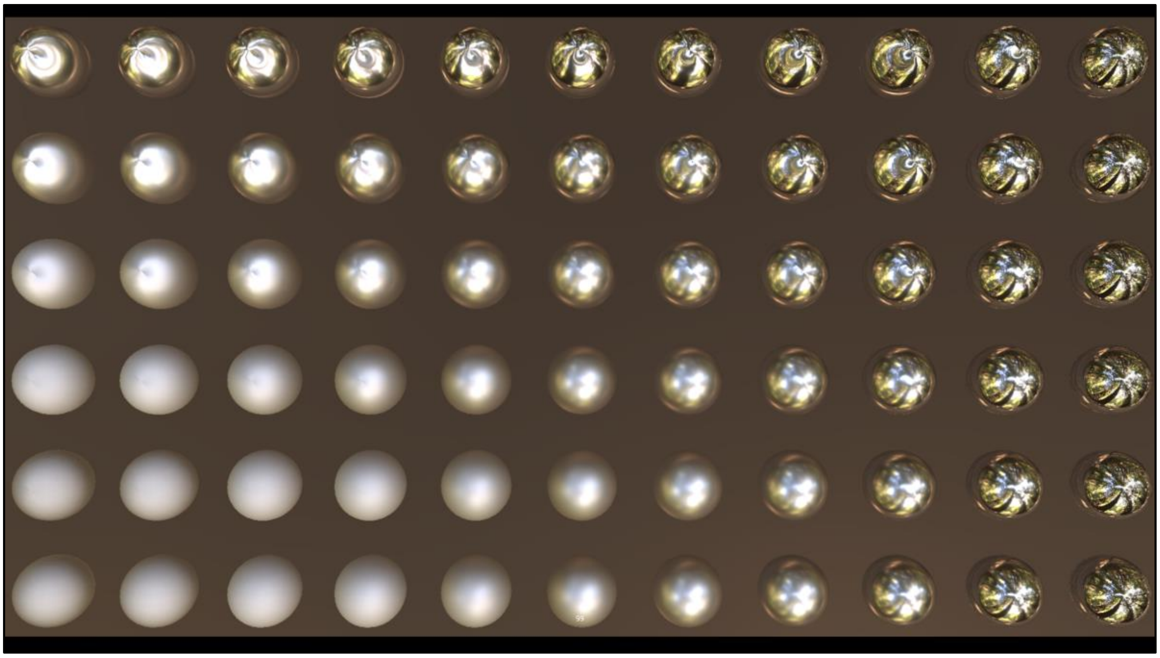
No multiple scattering
1st colum is dieletric pure white - Diffuse term here is Disney diffuse
2nd colum is F0 = 1 - Conductor
3nd is F0 = gold = 1 - Conductor
3nd is F0 = copper = 1 - Conductor



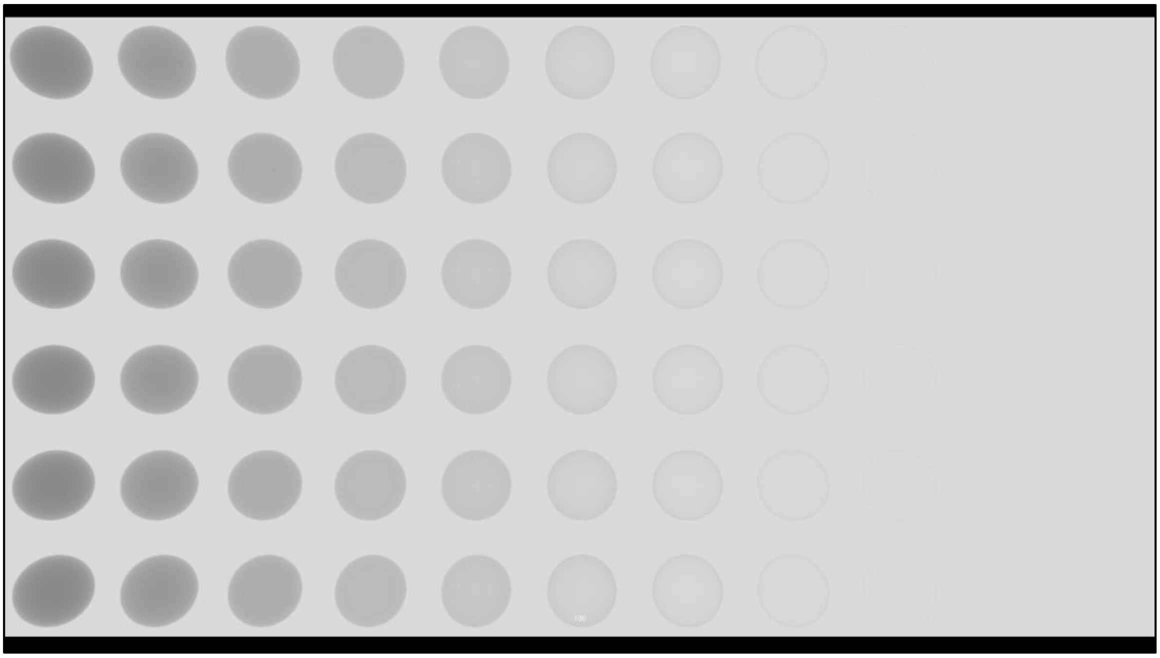
With multiple scattering
1st colum is dieletric pure white - Diffuse term here is Disney diffuse
2nd colum is F0 = 1 - Conductor
3nd is F0 = gold = 1 - Conductor
3nd is F0 = copper = 1 - Conductor



Anisotropy with no multiscattering



Anisotropy with multiscattering



Anisotropy with no multiscattering => We are using fake anisotropy with stretch hack, so the visual above is exactly the same than for no anisotropy.

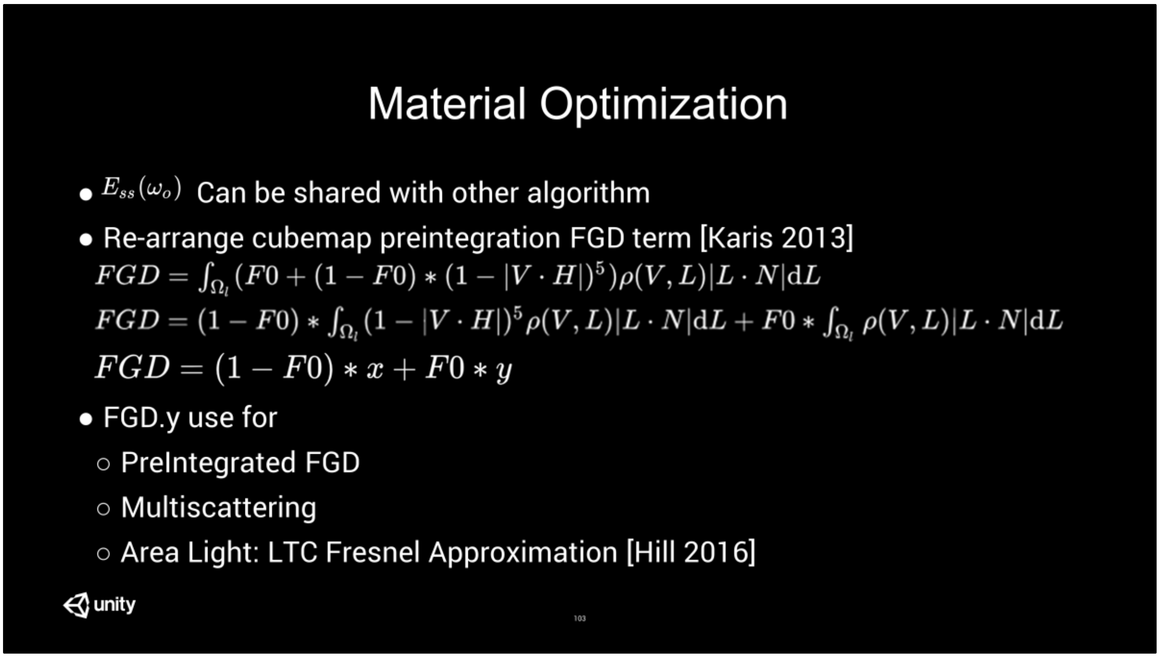
Anisotropy with multiscattering perfectly energy conserving! Totally fake but visual result isn’t bad.

Multiscattering GGX

- Made comparison with Mitsuba for validation
- Note: Mitsuba use spectral complex gold IOR + have inter-reflection

unity

Top HDRP, bottom Mitsuba
Comparison with Mitsuba is not so bad. But it is not simple to do fair comparison as Mitsuba is way more accurate and include light transport (reflection of sphere in sphere).



Quick pass on this one, just for reference

FGD =\int_{\Omega_i} \rho({\omega_o}, {\omega_i}) |{\omega_i \cdot n}| \mathrm{d}{\omega_i}
FGD=\int_{\Omega_l} (F0 + (1 - F0) * (1 - |{V \cdot H}|)^5)\rho({V}, {L}) |{L \cdot N}| \mathrm{d}{L}
FGD= (1 - F0) *\int_{\Omega_l} (1 - |{V \cdot H}|)^5\rho({V}, {L}) |{L \cdot N}| \mathrm{d}{L}+ F0*\int_{\Omega_l}\rho({V}, {L}) |{L \cdot N}| \mathrm{d}{L}
FGD= (1 - F0) *x+ F0 * y

[Hill 2016] LTC Fresnel approximation

Iridescence

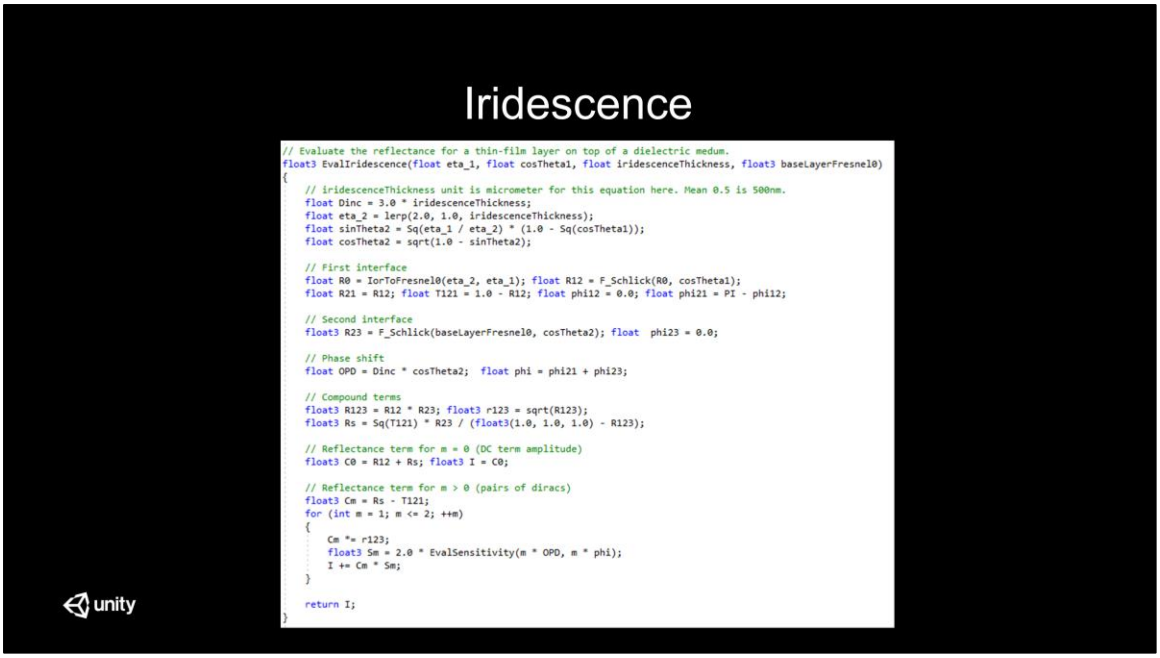
- Base on Unity Labs' research
  - Laurent Belcour: A Practical Extension to Microfacet Theory for the Modeling of Varying Iridescence
  - Code provided for BRDF explorer
  - Costly to evaluate for real-time, needs an approximation

unity

Iridescence

- Approximation for real time (Credit: Laurent Belcour)
  - Use Schlick Fresnel
    - Schick Fresnel wrong outside of IOR range 1.4 - 2.2 [Lagarde 2013]
    - IOR 1.0 suppose to cancel the effect - Use mask parameter instead
  - Use RGB color space only
    - Original code uses XYZ color
  - Simplify phase shift
  - Use less reflectance term

unity



Code provide as reference. EvalSensitivity is the same function than in the original provided code of the paper.

Iridescence

- EvalIridescence call to replace base Fresnel0
  - Done for all light types
  - Theoretically suppose to call it for each punctual lights - Too expensive
- Combine with clear coat (Hacky way)

```
float topIor = 1.0; // Default is air
if (HasFlag(bsdfData.materialFeatures, MATERIALFEATUREFLAGS_LIT_CLEAR_COAT))
{
    topIor = CLEAR_COAT_IOR;
    // HACK: Use the reflected direction to specify the
    // Fresnel coefficient for pre-convolved envmaps
    NdotV = sqrt(1.0 + Sq(1.0 / topIor) * (Sq(dot(bsdfData.normalWS, V)) - 1.0));
}
fresnel0 = EvalIridescence(topIor, NdotV, iridescenceThickness, fresnel0);
```

- Parametrization is still not friendly

unity

GBuffer Design

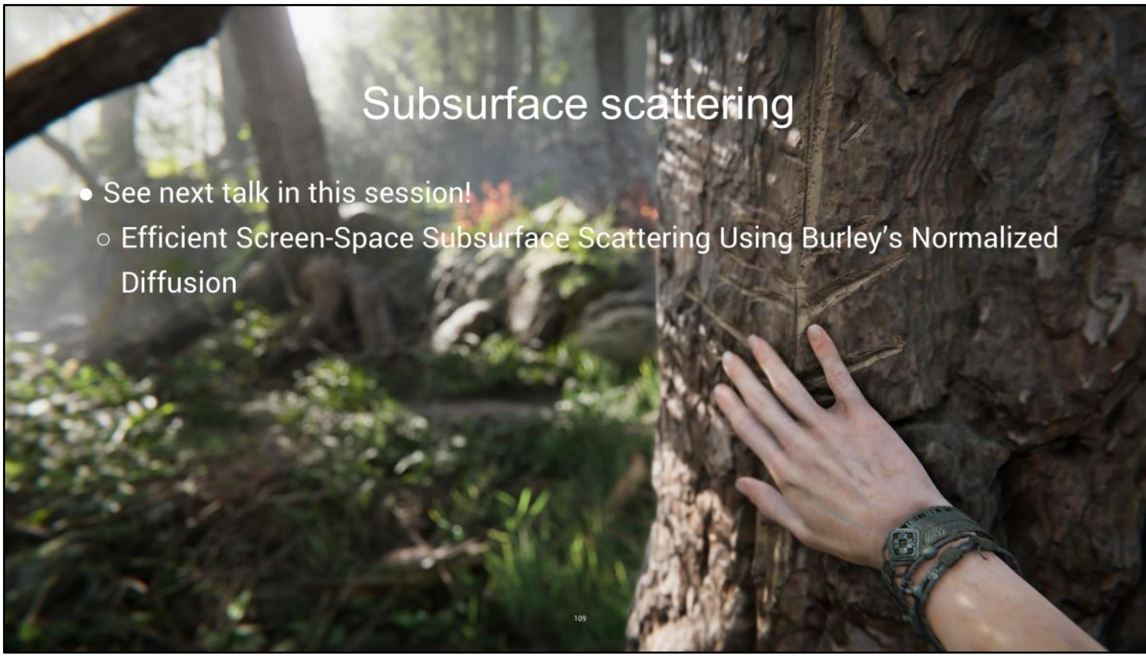
- Iridescence

Iridescence	R	G	B	A
RT0 RGBA8 sRGB	BaseColor.rgb			Specular Occlusion
RT1 RGBA8	Normal.xy (Octahedral 12/12)			Perceptual Smoothness
RT2 RGBA8	IOR	Thickness	Unused (3) / Metallic(5)	FeaturesMask(3) / CoatMask(5)

- 3 bit unused for optimization purposes
  - Match metallic encoding of Anisotropy

Subsurface scattering

- See next talk in this session!
  - Efficient Screen-Space Subsurface Scattering Using Burley's Normalized Diffusion



Translucency

- See next talk in this session!
  - Efficient Screen-Space Subsurface Scattering Using Burley's Normalized Diffusion
- Separate material features from Subsurface scattering
- HDRP support:
  - Subsurface scattering + Translucency
  - Subsurface scattering only
  - Translucency only
    - Foliage

GBuffer Design

- Subsurface scattering and/or Translucency

SSS + Transmission	R	G	B	A
RT0 RGBA8 sRGB	BaseColor.rgb			DiffusionProfile(4) / SubsurfaceMask(4)
RT1 RGBA8	Normal.xy (Octahedral 12/12)			Perceptual Smoothness
RT2 RGBA8	Specular Occlusion	Thickness	DiffusionProfile(4) / SubsurfaceMask(4)	FeaturesMask(3) / CoatMask(5)

- SSSSS pass require RT0 only
  - Swap specular occlusion location to save bandwidth
- Duplicate DiffusionProfile/SurfaceMask for optimization purposes

unity

The weird arrangement is to be able to save bandwidth and store all required information for SSS in one RT0. The lighting code don't need to read RT0 until the very end, so DiffusionProfile and SubsurfaceMask are duplicated to not have to read RT0 ahead.

Lit shader performance

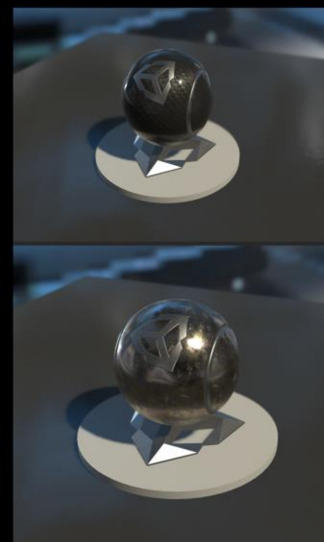
- Base PS4 - 1080p - Fullscreen quad with the material with simple input
- Affected by a Single light + Sky - No shadow
  - Use material and light classification
  - Subsurface scattering features write in two render targets
  - Forward pass sample global illumination (Extra cost)

All times are in miliseconds	Standard			Iridescent			Anisotropy		
	Punctual	Rectangular	ReflectionProbe	Punctual	Rectangular	ReflectionProbe	Punctual	Rectangular	ReflectionProbe
Deffered Mode (Lighting pass)	1.131269	2.32051	1.741142	1.774674	2.89209	1.836177	1.549801	3.039909	1.847629
Forward Mode (Forward pass)	2.294004	3.260511	2.267381	2.725634	3.694113	2.688607	2.455848	3.420852	2.390015
	SSS			**SSS + Transmission**			**Transmission**		
	Punctual	Rectangular	ReflectionProbe	Punctual	Rectangular	ReflectionProbe	Punctual	Rectangular	ReflectionProbe
Deffered Mode (Lighting pass)	1.549171	3.459631	1.808749	1.631744	3.521049	1.77821	1.576882	3.47842	1.740248
Forward Mode (Forward pass)	2.474051	3.4232871	2.462119	2.658449	4.026517	2.567901	2.424006	3.811705	2.368409
	StandardClearCoat			**AnisotropicClearCoat**			**IridescentClearCoat**		
	Punctual	Rectangular	ReflectionProbe	Punctual	Rectangular	ReflectionProbe	Punctual	Rectangular	ReflectionProbe
Deffered Mode (Lighting pass)	1.388901	3.35079	1.927229	2.601673	4.212729	1.86724	2.816857	4.497486	1.994913
Forward Mode (Forward pass)	2.437424	3.886043	2.55829	2.623948	4.065559	2.685529	2.899818	4.380542	2.984346

What can be observe is that our area light cost twice the price of a punctual light. Reflection probe have good performance with complex material (Due to various approximation we do)

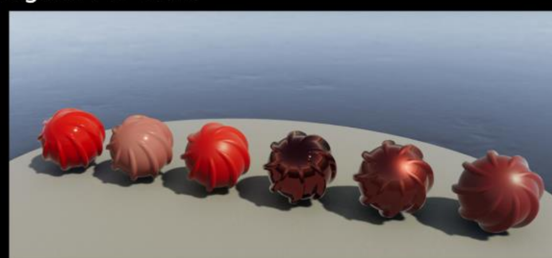
StackLit Shader

- Target VFX/Movie/Film
  - Accurate version of Lit
    - Remove some approximation
  - Support all materials features at the same times
    - Ex: Iridescence + SSS + Translucency + Coat
  - Always Forward material
- Vertical 2 layer shaders
  - Base + Coat Layer

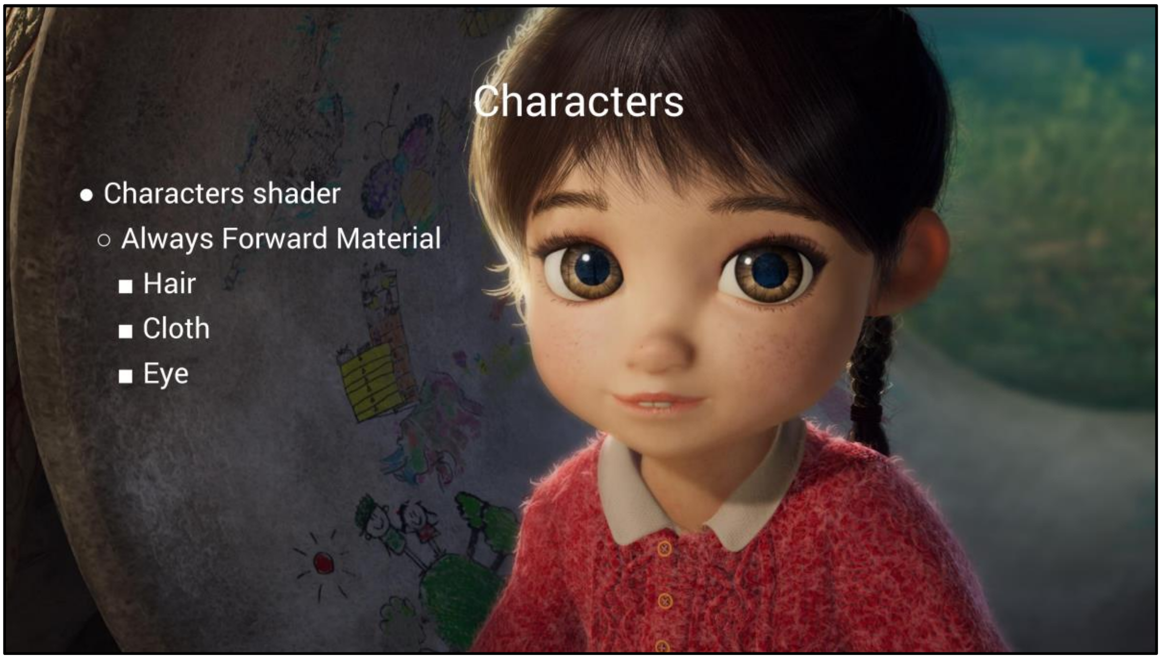


StackLit Shader

- Base on Unity Labs' research - Laurent Belcour
  - Efficient Rendering of Layered Materials using an Atomic Decomposition with Statistical Operators
    - Tuesday, 14 August 10:45am



unity



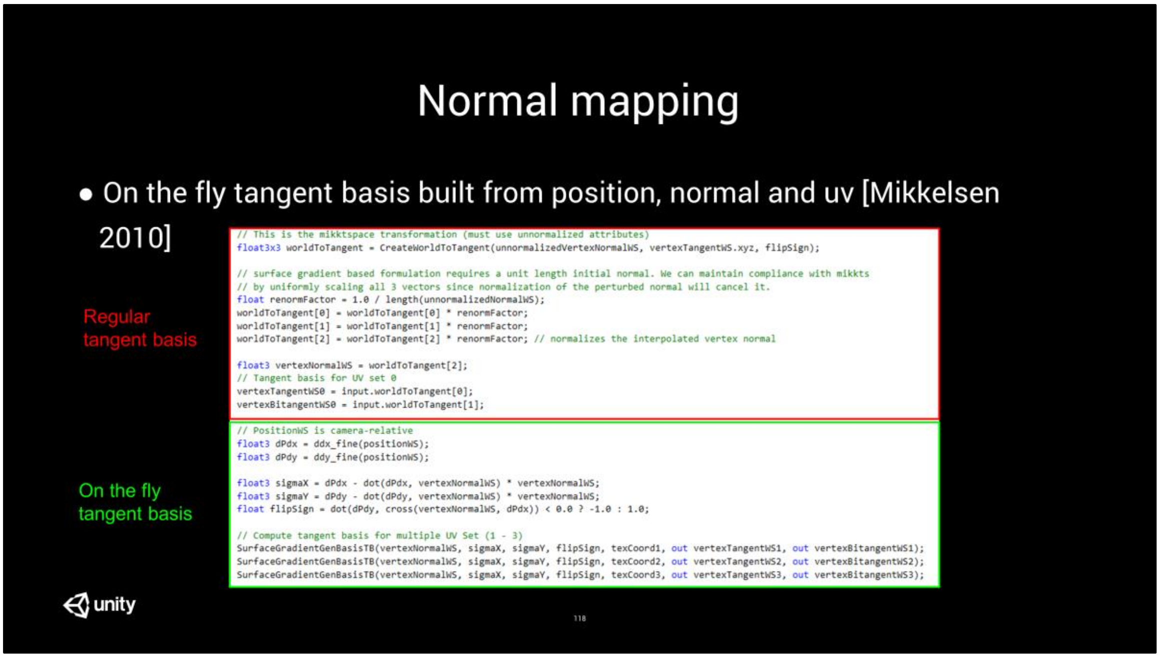
And here is another example showcasing organic material properties, from the Windup project done using HD from Yibing Jiang and the graphics team, which shows the power of anisotropic materials (used for hair), subsurface scattering, cloth BRDFs, and more advanced materials.

LayeredLit

- Facilities to mix several Lit materials together
  - Support various weights for masking
  - *Influence mode* targeting photogrammetry
- Describe in depth in “Photogrammetry workflow Layered Shader” ebook
- Want to build complex and rich environment
  - Mean complex layering of normal map
  - But...

Normal mapping issues

- But normal mapping has many issues
  - Requires one tangent frame per UVs, even procedural UVs
  - Hard to handle volume bump mapping (Triplanar / Noise)
  - Multiple blending formulation [Brisebois 2012]
    - Some order-dependent
  - Impractical with procedural geometry
    - Hard to handle many blendshapes



Bump Mapping Unparametrized Surfaces on the GPU Morten S. Mikkelsen 2010
Built from UVSet, position and normal

Normal mapping

- Each UV set adds extra GPU cost [ALUs]

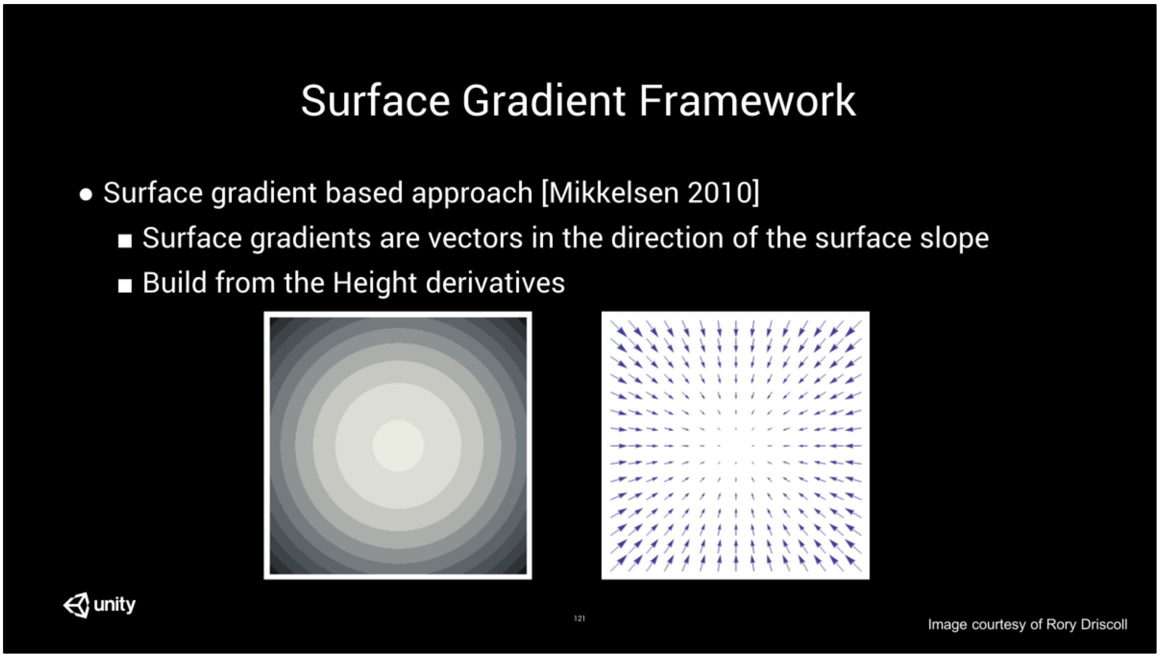
```
// This produces an orthonormal basis of the tangent and bitangent WITHOUT vertex level tangent/bitangent for any UV including procedurally generated
void SurfaceGradientGenBasisTB(real3 nrmVertexNormal, real3 sigmaX, real3 sigmaY, real flipSign, real2 texST, out real3 vT, out real3 vB)
{
    real2 dSTdx = ddx_fine(texST), dSTdy = ddy_fine(texST);

    real det = dot(dSTdx, real2(dSTdy.y, -dSTdy.x));
    real sign_det = det < 0 ? -1 : 1;

    // invC0 represents (dXds, dYds); but we don't divide by determinant (scale by sign instead)
    real2 invC0 = sign_det * real2(dSTdy.y, -dSTdx.y);
    vT = sigmaX * invC0.x + sigmaY * invC0.y;
    if (abs(det) > 0.0)
        vT = normalize(vT);
    vB = (sign_det * flipSign) * cross(nrmVertexNormal, vT);
}
```

unity

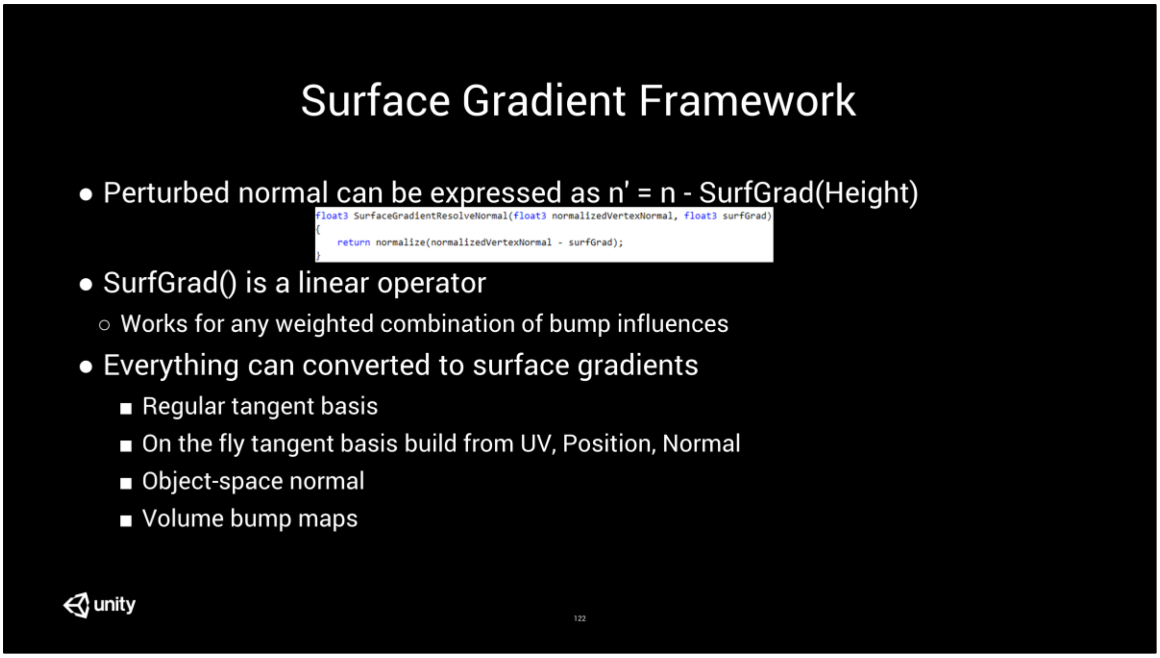
Normal mapping

- In practice
  - Tangent basis based on UV0 use Mikktspace [Mikkelsen 2008]
    - On the fly TBN not good at handling low poly mesh with hard surface
      - I.e when normal map is use to correct shape of the mesh
  - Generate on the fly basis for other UV1-3
  - Only solve small part of the normal mapping issues...

[Mikkelsen 2008] Morten S. Mikkelsen. Simulation of Wrinkled face Revisited.



Given a scalar height field (i.e. a two-dimensional array of scalar values), the gradient of that field is a 2D vector field where each vector points in the direction of greatest change. The length of the vectors corresponds to the rate of change.

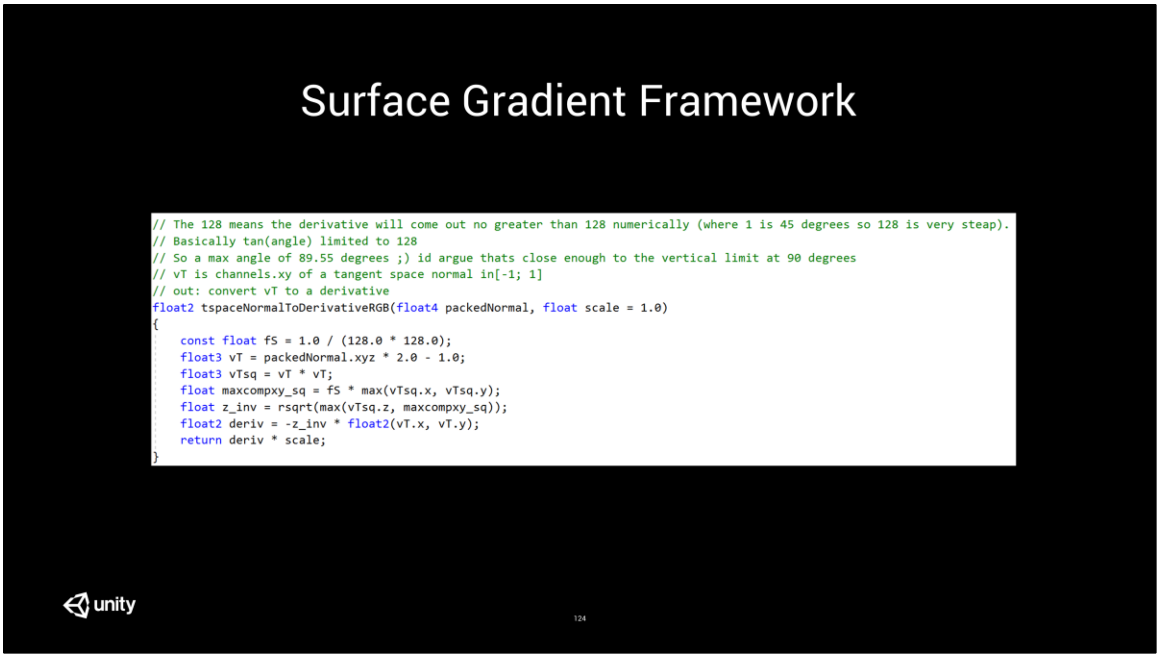


1. The surface gradient based approach [MM2010 - sfgrad] allows us to unify all of this into one framework.
2. It is shown in [MM2010 - sfgrad] that Blinn's perturbed normal can be expressed as n' = n - SurfGrad(H)
3. SurfGrad(H) is a linear operator and will work for any weighted combination of bump influences.
   - An object space normal can be converted on the fly (in the pixel shader) into a surface gradient.
   - Conventional mikktspace compliant vertex level tangent space can be converted on the fly to a surface gradient.
      - We can also generate a surface gradient on the fly from a uv, position and normal WITHOUT a vertex tangent space (though not mikktspace compliant).
      - For volume bump maps we can generate a surface gradient on the fly which as shown in [MM2010 - sfgrad] provides the correct result

Surface Gradient Framework

```
// surface gradient from an on the fly TBN (deriv obtained using tspaceNormalToDerivative()) or
// from conventional vertex level TBN (mikktspace compliant and deriv obtained using tspaceNormalToDerivative())
float3 SurfaceGradientFromTBN(float2 deriv, float3 vT, float3 vB)
{
    return deriv.x * vT + deriv.y * vB;
}

// surface gradient from an already generated "normal" such as from an object or world space normal map
// v does not need to be unit length as long as it establishes the direction.
float3 SurfaceGradientFromPerturbedNormal(float3 nrmVertexNormal, float3 v)
{
    float3 n = nrmVertexNormal;
    float s = 1.0 / max(FLT_EPS, abs(dot(n, v)));
    return s * (dot(n, v) * n - v);
}

// used to produce a surface gradient from the gradient of a volume bump function such as a volume of perlin noise.
// equation 2. in "bump mapping unparametrized surfaces on the GPU".
// Observe the difference in figure 2. between using the gradient vs. the surface gradient to do bump mapping (the original method is proved wrong in the paper!).
float3 SurfaceGradientFromVolumeGradient(float3 nrmVertexNormal, float3 grad)
{
    return grad - dot(nrmVertexNormal, grad) * nrmVertexNormal;
}

// triplanar projection considered special case of volume bump map
// derivs obtained using tspaceNormalToDerivative() and weights using computeTriplanarWeights().
float3 SurfaceGradientFromTriplanarProjection(float3 nrmVertexNormal, float3 triplanarWeights, float2 deriv_xplane, float2 deriv_yplane, float2 deriv_zplane)
{
    const float w0 = triplanarWeights.x, w1 = triplanarWeights.y, w2 = triplanarWeights.z;

    // assume deriv_xplane, deriv_yplane and deriv_zplane sampled using (z,y), (z,x) and (x,y) respectively.
    // positive scales of the look-up coordinate will work as well but for negative scales the derivative components will need to be negated accordingly.
    float3 volumeGrad = float3(w2 * deriv_zplane.x + w1 * deriv_yplane.y, w2 * deriv_zplane.y + w0 * deriv_xplane.y, w0 * deriv_xplane.x + w1 * deriv_yplane.x);

    return SurfaceGradientFromVolumeGradient(nrmVertexNormal, volumeGrad);
}
```

unity



conversion from tangent space normal to derivative allows us to rewrite tbn transformation as surfgradient since it represents a uniform scale

n = (nx, ny, nz) as derivative is d = (-nx/nz, -ny/nz)

So after final normalization we get tbn transform vT*n.x + vB*n.y + vN*n.z is the same as vN - (d.x*vT + d.y*vB) where the part in parenthesis is basically a tbn style surface gradient when used together with your other slide where tbn are uniformly scaled. The normal mapping slide with code involving worldToTangent. So in the former version vT, vB and vN are all unnormalized since interpolation. In the latter surfgrad variant they're uniformly scaled (as a trick to normalize vN since surfgrad formulation requires it). This factor is canceled out in final normalization along with division by nz to make the derivative which is also a uniform scale

Surface Gradient Framework

- Triplanar normal mapping can be tricky to implement
  - Often result in wrong orientation

unity

Surface Gradient Framework

- Surface gradient is simpler and don’t exhibit the issue

Surface Gradient Framework

- Nice solution to normal mapping issues
  - Example use
    - Layered material
      - Base + 3 layers
        - Base UV0
        - Layers UV0-2 or Planar
        - UV3 for details map
      - Various blend mask mode

unity

Surface Gradient Framework

- Performance number
  - Cost imply by on the fly tangent basis
  - Scene with complex material, tree, foliage, ground

Normal Gradient (Deffered)	
State	GBufferTime
On	6.964854
Off	6.570049

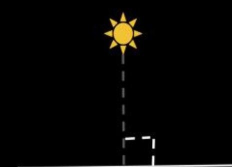
unity


Lighting
129
Advances in Real-Time Rendering in Games course, SIGGRAPH 2018

Physical light unit

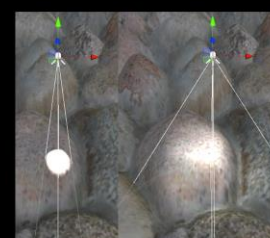
- Base on [Lagarde 2014] - Same formulation

Punctual Light	Luminous power (lm), Luminous intensity (cd)
Area Light	Luminous power (lm), Luminance (cd/m^2) or EV (K=12.5)
Emissive	Luminance (cd/m^2)
Environment	Illuminance (Lux) of upper hemisphere
Sun	Illuminance (Lux) at ground level with sun at Zenith



- Two modes for aperture of spot lights
  - Occlusion
  - Reflector
    - Intensity vary with aperture

```
// Occlusion
public static float ConvertPointLightLumenToCandela(float intensity)
{  return intensity / (4.0f * Mathf.PI); }
// Reflector
public static float ConvertSpotLightLumenToCandela(float intensity, float angle)
{ return intensity / (2.0f * (1.0f - Mathf.Cos(angle / 2.0f)) * Mathf.PI); }
```

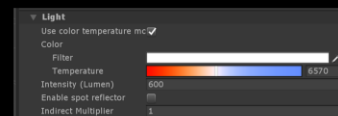


unity

People may not be aware but using inverse square attenuation mean that you use physical unit. I.e it is candela and directional light is in Lux (if divide by PI), else PI Lux

Physical light unit

- Punctual and Sun light color
  - Filter + Color temperature
  - Accurate Fit approximation (Max error: 0.008)




```
// Given a correlated color temperature (in Kelvin), estimate the RGB equivalent. Curve fit error is max 0.008.
// return color in linear RGB space
2 références | sebastienlagarde, il y a 32 jours | 1 auteur, 1 modification
public static Color CorrelatedColorTemperatureToRGB(float temperature)
{
    float r, g, b;

    // Temperature must fall between 1000 and 40000 degrees
    // The fitting require to divide kelvin by 1000 (allow more precision)
    float kelvin = Mathf.Clamp(temperature, 1000.0f, 40000.0f) / 1000.0f;
    float kelvin2 = kelvin * kelvin;

    // Using 6570 as a pivot is an approximation, pivot point for red is around 6580 and for blue and green around 6560.
    // Calculate each color in turn (Note, clamp is not really necessary as all value belongs to [0..1] but can help for extremum).
    // Red
    r = kelvin < 6.570f ? 1.0f : Mathf.Clamp((1.35651f + 0.216422f * kelvin + 0.000633715f * kelvin2) / (-3.24223f + 0.918711f * kelvin), 0.0f, 1.0f);
    // Green
    g = kelvin < 6.570f ?
        Mathf.Clamp((-399.809f + 414.271f * kelvin + 111.543f * kelvin2) / (2779.24f + 164.143f * kelvin + 84.7356f * kelvin2), 0.0f, 1.0f) :
        Mathf.Clamp((1370.38f + 734.616f * kelvin + 0.689955f * kelvin2) / (-4625.69f + 1699.87f * kelvin), 0.0f, 1.0f);
    //Blue
    b = kelvin > 6.570f ? 1.0f : Mathf.Clamp((348.963f - 523.53f * kelvin + 183.62f * kelvin2) / (2848.82f - 214.52f * kelvin + 78.8614f * kelvin2), 0.0f, 1.0f);

    return new Color(r, g, b, 1.0f);
}
```

unity

Physical light unit

- HDRI
  - Plenty of relative HDRI available but want absolute HDRI
    - Require measurement data [Lagarde 2016]
  - Enter desired illuminance value (Lux) instead for upper hemisphere
  - Compute ratio with effective value of HDRI and apply multiplier
    - Use brute force sphere uniform sampling on GPU in Editor

```
float GetUpperHemisphereLuxValue()
{
    float sum = 0.0; float dphi = 0.005; float dtheta = 0.005;
    for (float phi = 0; phi < 2.0 * PI; phi += dphi)
    {
        for (float theta = 0; theta < PI / 2.0; theta += dtheta)
        {
            float3 L = SphericalToCartesian(phi, cos(theta));
            real val = Luminance(SAMPLE_TEXTURECUBE_LOD(skybox, sampler_skybox, L, 0).rgb);
            sum += cos(theta) * sin(theta) * val;
        }
    }
    return sum * dphi * dtheta;
}
float luxMultiplier =  desiredLuxValue / upperHemisphereLuxValue;
```

[Lagarde 2016] An Artist-Friendly Workflow for Panoramic HDRI (Sébastien Lagarde)
Lux value can easily be measure with a lux meter
Typical value for clear sky HDRI without Sun: 20 000 lux

Physical light unit

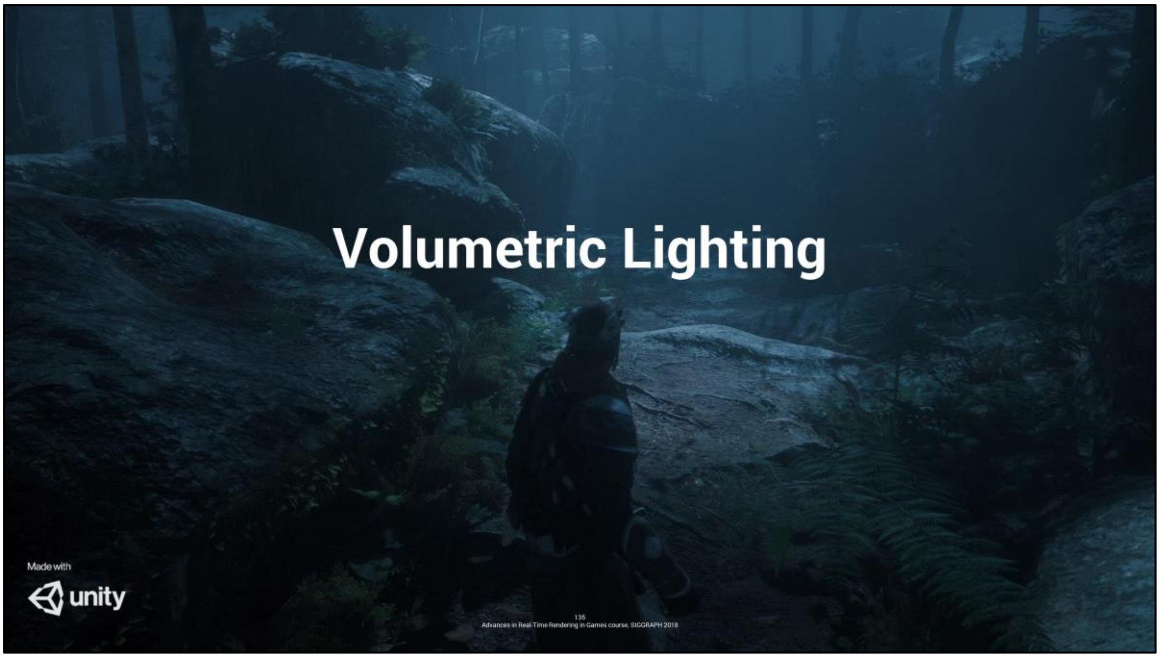
- Area lights [Heitz 2017]
  - Base on Linearly Transformed Cosine
  - Joint Unity Labs and Lucasfilm research
    - Support Rectangle and Line
    - Upcoming sphere and disc
- Line light have no area so integrate radiance along the line

```
public static float CalculateLineLightLumenToLuminance(float intensity, float lineWidth)
{
    //Unlike tube lights, our line lights have no surface area, so the integral becomes:
    //power = Integral{length, Integral{sphere, radiance}}
    //For an isotropic line light, radiance is constant, therefore:
    //power = length * (4 * Pi) * radiance,
    return intensity / (4.0f * Mathf.PI * lineWidth);
}
```

unity

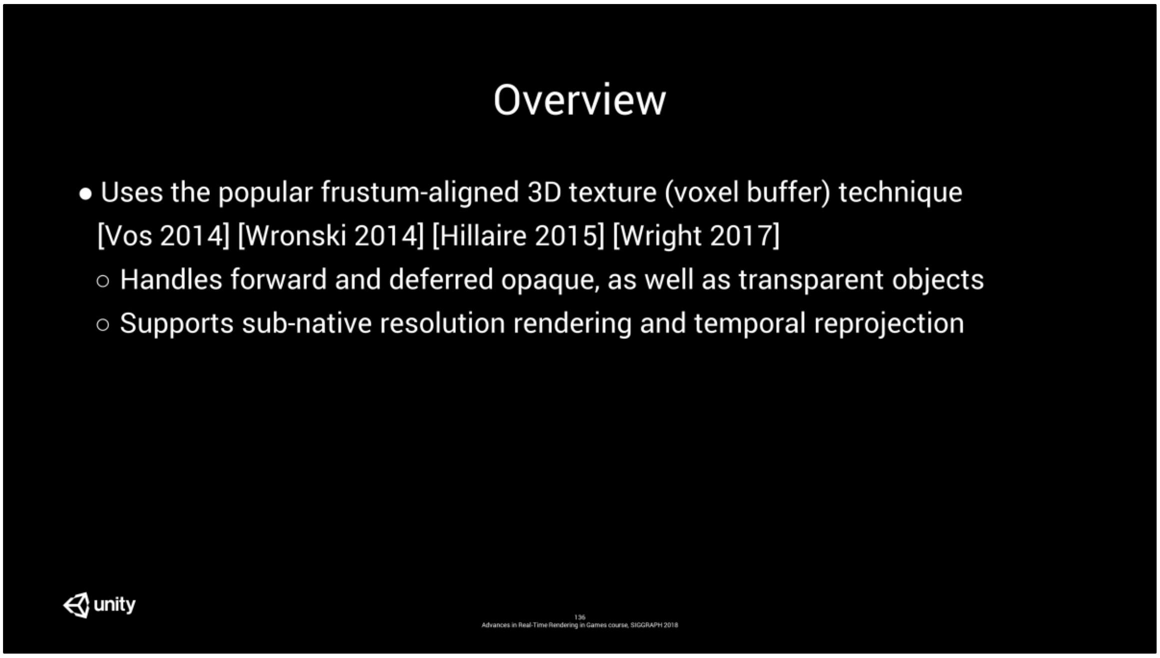
No shadow :(

Next up: volumetrics! See SIGGRAPH 2018 HD RP volumetrics.mp4
Do not remove this slide. It has a video.



Thank you, Sebastien.

In the remaining time, I will shed some light onto our implementation of volumetric lighting (along with some open problems).



We implemented the so-called “froxel” lighting algorithm, which is a popular AAA solution for volumetric lighting.

Some of its advantages include support of all surface types, as well as the ability to efficiently perform sub-native resolution rendering with temporal reprojection.

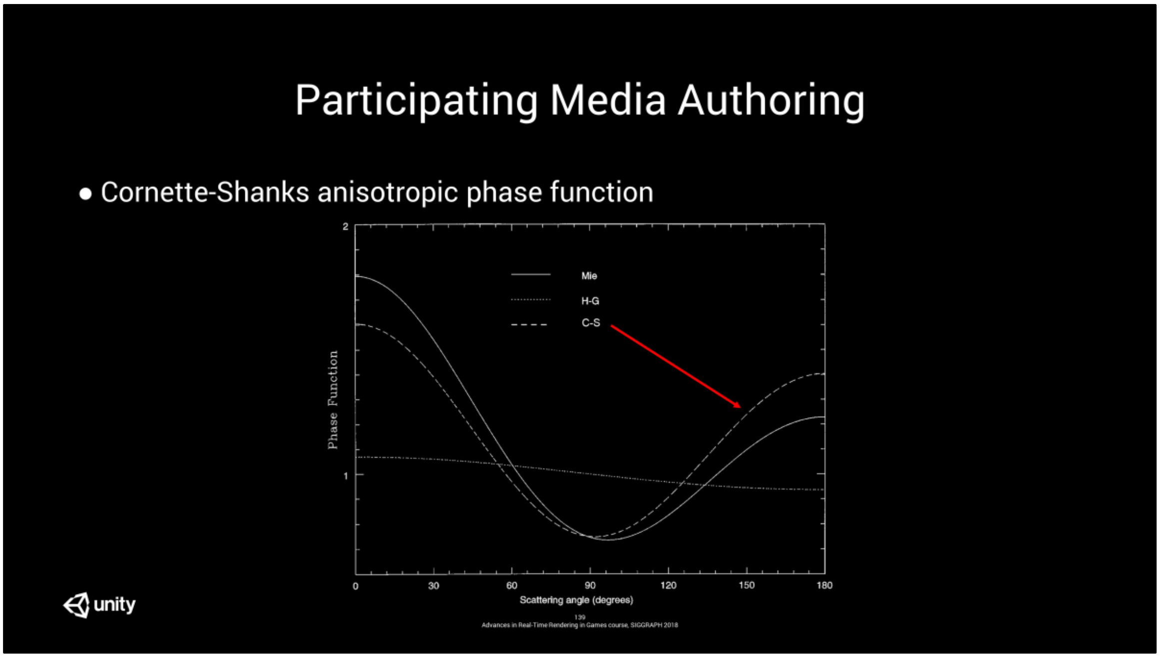
My goal is not to repeat the information that’s already been published, but rather describe how our implementation differs from the existing approaches, and provide the missing details.

We support 2 ways of adding fog to the scene:

- an artist can add global, unbounded fog, or
- a local density volume represented by an oriented bounding box with a grayscale 3D texture.

See SIGGRAPH 2018 HD RP volumetrics participating media authoring.mp4

In both cases, we expose an artist-friendly volumetric material parametrization of single scattering albedo and mean free path, which we then internally convert to the scattering and extinction coefficients.
For performance reasons, we only support monochromatic mean free paths, which means that extinction coefficients are also monochromatic.
As a result, while light bounces can tint scattered light, fog attenuation will never affect the color of light travelling along straight paths (such as camera rays).



We chose to support the Cornette-Shanks anisotropic phase function with the global anisotropy parameter. Compared to the Henyey-Greenstein, it provides a better match for the “true” Mie phase function.

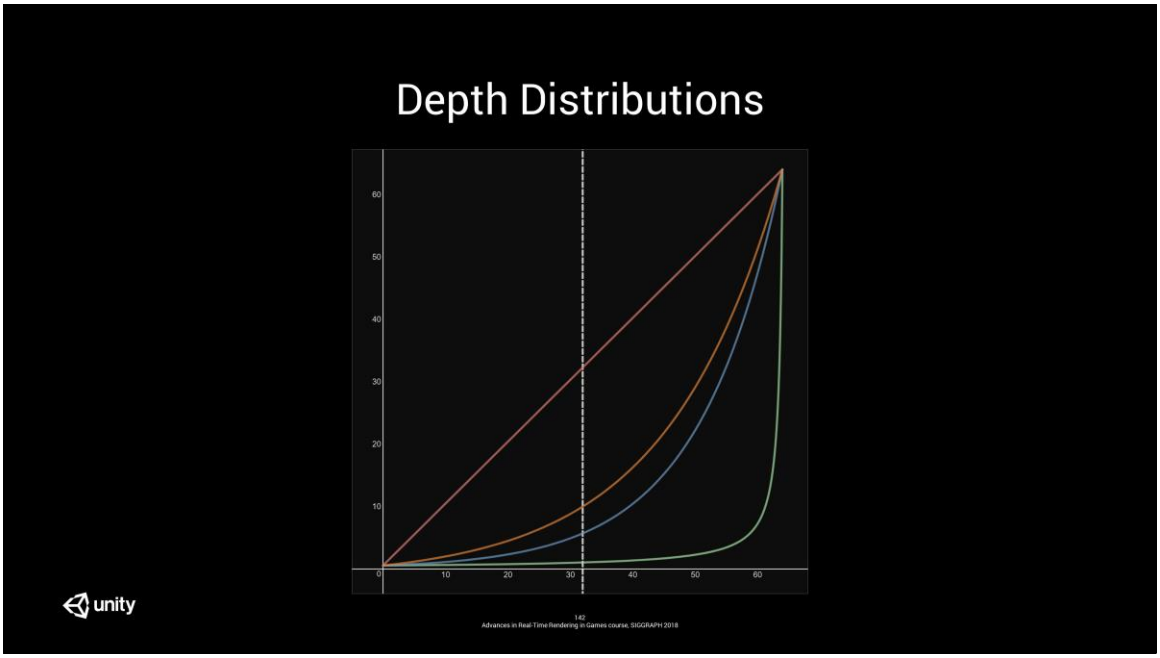
Note: Cornette-Shanks anisotropic phase function [Cornette 1992] [Toublanc 1996].

For example, this is how a spot light acts within highly forward-scattering fog. (see SIGGRAPH 2018 HDRP talk - spotlight with forward scatter fog.gif)

For local fog, we use 3D textures to represent participating media because volumetric lighting is evaluated at such a low rate that many involved signals quickly become undersampled and thus alias...

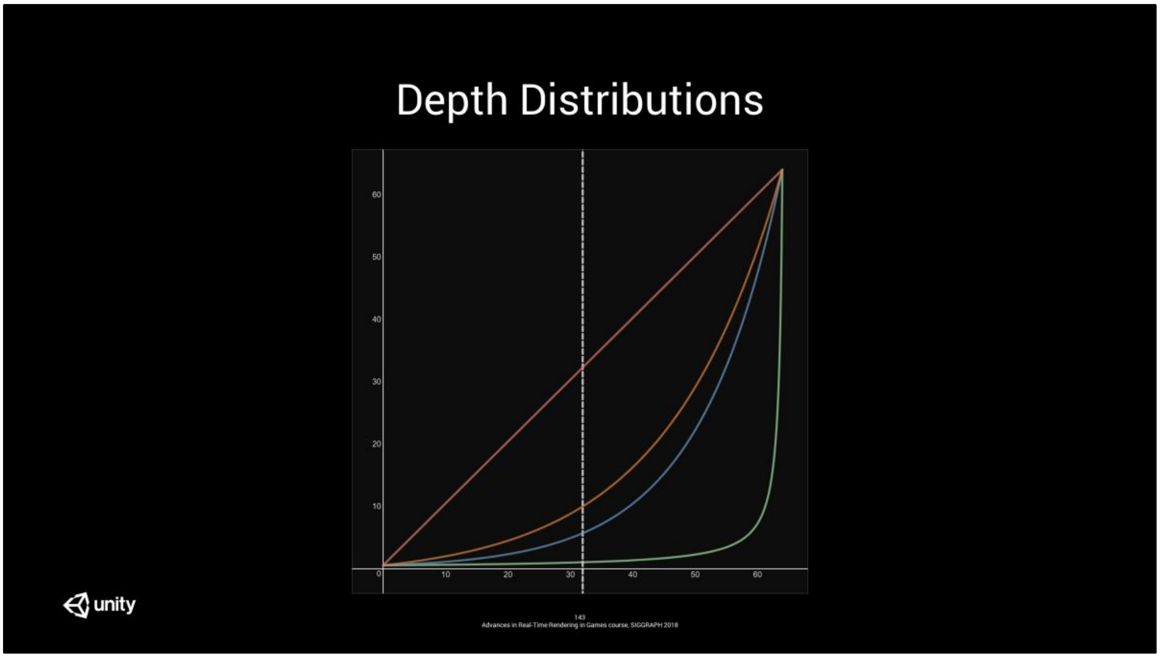
This includes shadow maps, light cookies and density textures. Luckily, for textures we can just* use MIP maps, while handling geometry LOD is more complicated.

** see "Open Problems and Future Work"*

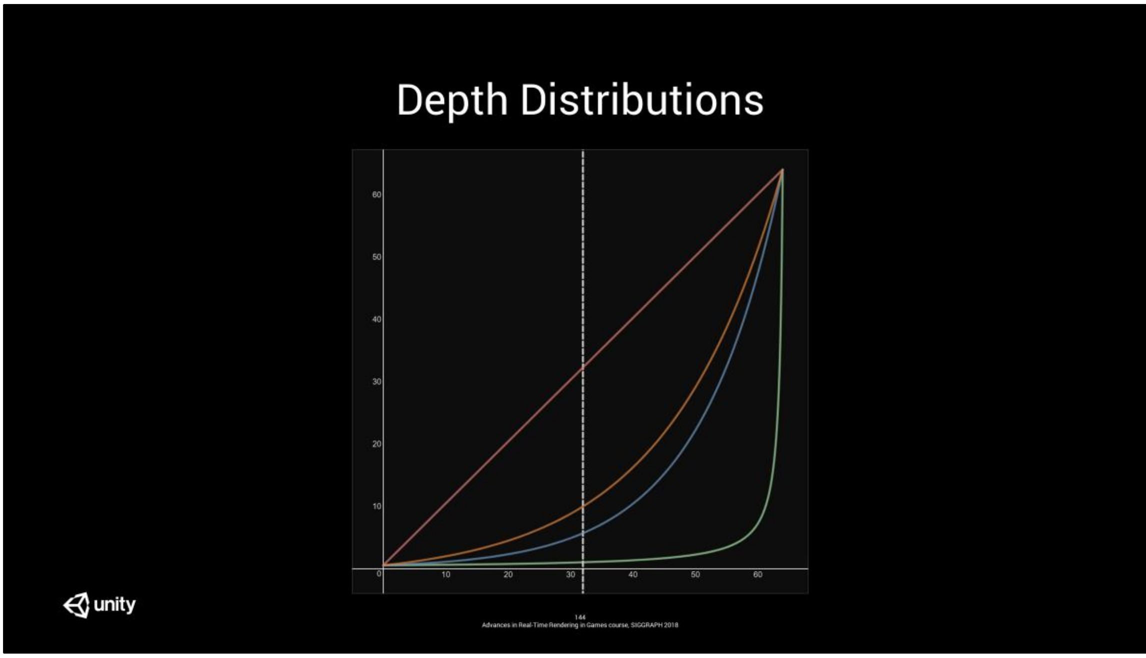


Our implementation is quite flexible when it comes to slice distributions.

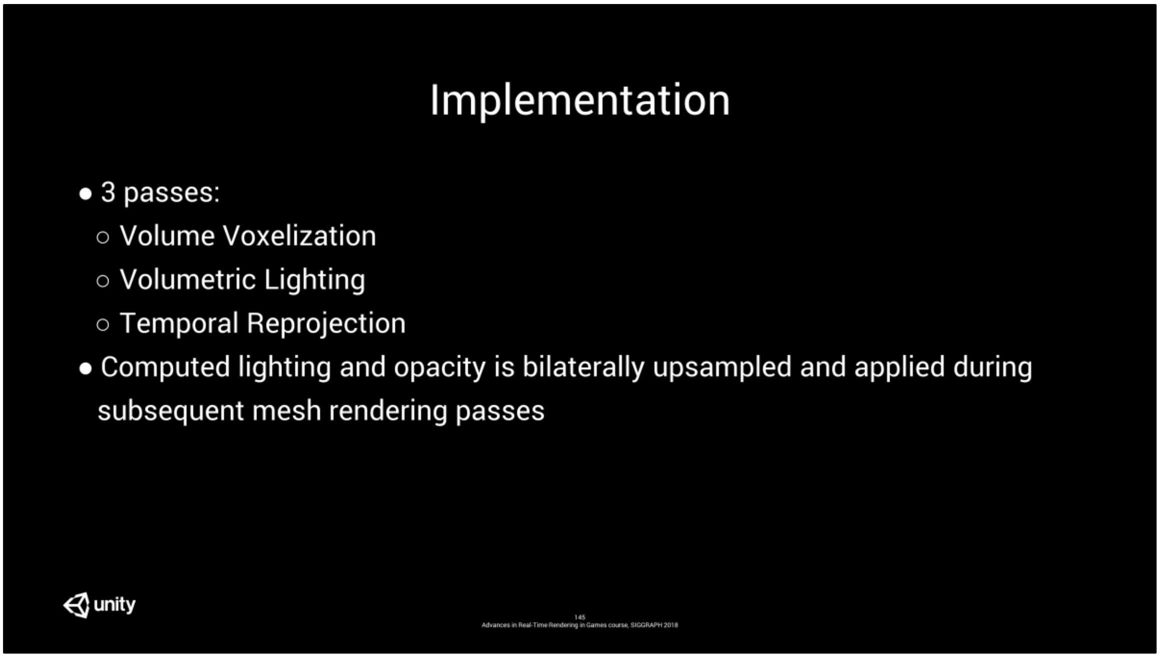
We started with work of Brano Kemen of Outerra, who described the logarithmic depth distribution in his blog post [Klemen 2012].
In some sense, his distribution is optimal. However, different content may have different needs, therefore we expose a tweak parameter which controls the *generalized* logarithmic depth distribution, which smoothly transitions between linear and logarithmic.



Now, how to read this graph (https://www.desmos.com/calculator/qrtatrlrba):
* on the X axis, you have the depth slice of the buffer, from 0 to 64;
* on the Y axis, you have the linear depth corresponding to this slice, from 0.5 to 64 meters;
* I've also drawn a vertical line in the middle, at 32 slices, which we'll examine.



In red, we have the typical inverse Z distribution, which is predictably awful, and covers the range of 0.5 to 1 meter.
In green, we have the standard logarithmic distribution, covering the distance of up to 5.6 meters.
In blue, we have the generalized distribution with the tweak parameter set to 0.5, which covers the distance of up to 10 meters.
Depending on the value of the tweak parameter, it can span the range from the logarithmic distribution in green to the linear distribution in purple.

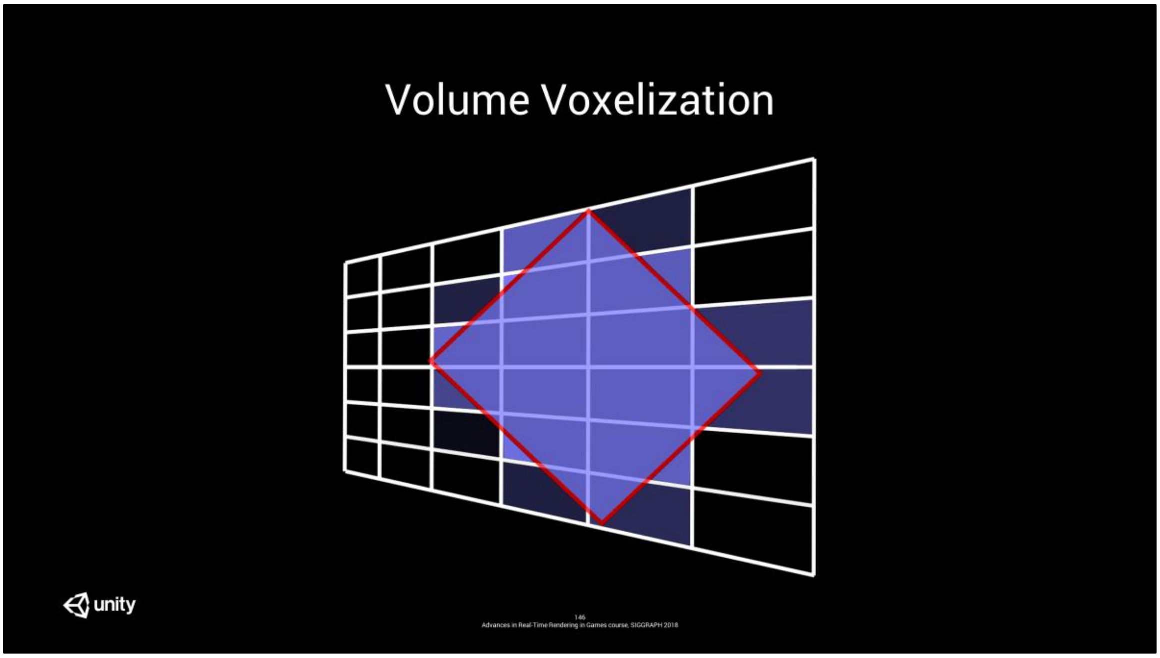


Our implementation is split into 3 passes.

During the 1st pass, we fill the density buffer by voxelizing density volumes.
During the 2nd pass, we solve the single scattering integral.
During the 3rd pass, we combine the results from the current frame with accumulated results of previous frames.

As a result, we obtain volumetric lighting and opacity buffers, which we bilaterally upsample and apply during mesh rendering.

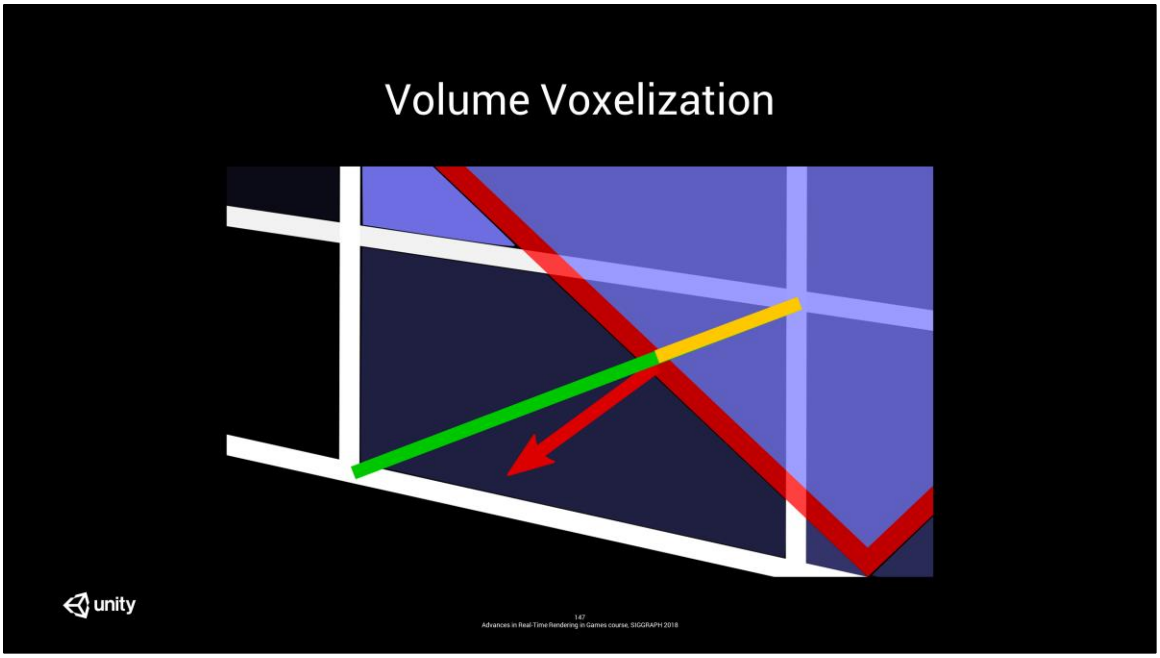
Note: opacity is (1 - transmittance).



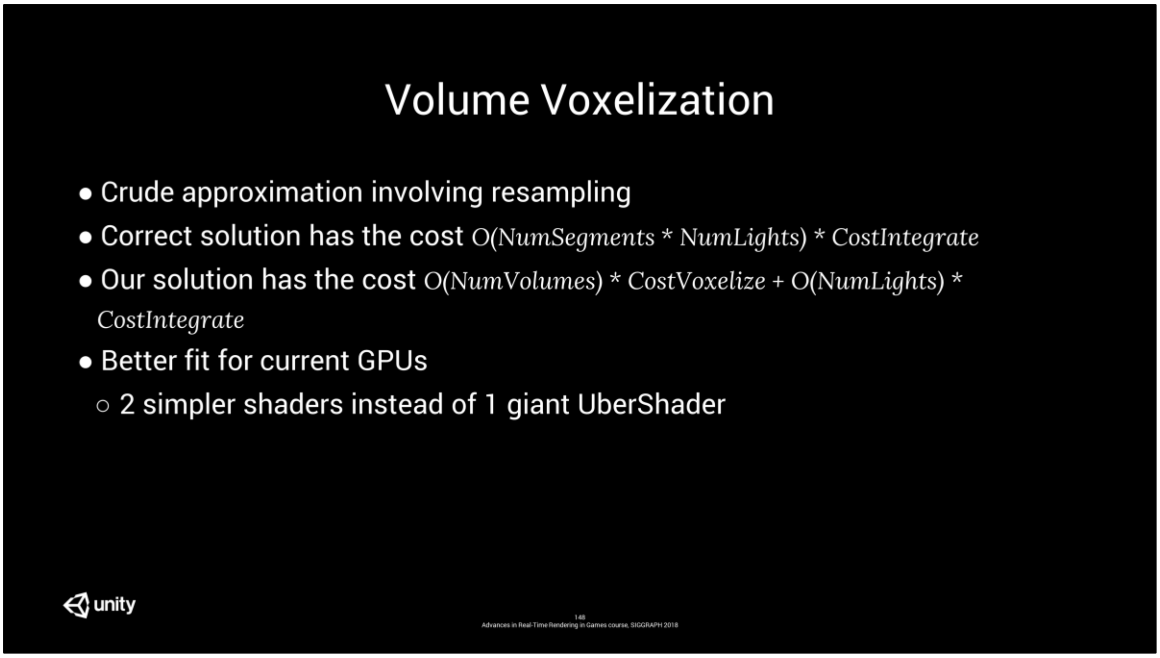
We start by performing conservative solid voxelization of density volumes. To put it plainly, we determine the set of voxels overlapping a box.

We want to compute partial coverage in order to anti-alias the resulting buffer and achieve temporal stability. While this may seem like a nice application for a conservative rasterizer, our goal is to use async compute.
We haven't yet found a paper which describes an efficient solution to this problem. Therefore, we came up with our own.
It is inspired by techniques presented in the paper of Samuli Laine titled "A Topological Approach to Voxelization" [Laine 2013].

Just as for clustered lighting, we start with a clustered pre-pass to prune and localize a set of volumes.
Then, for each voxel, we look up the set of volumes overlapping its cluster, and voxelize those.



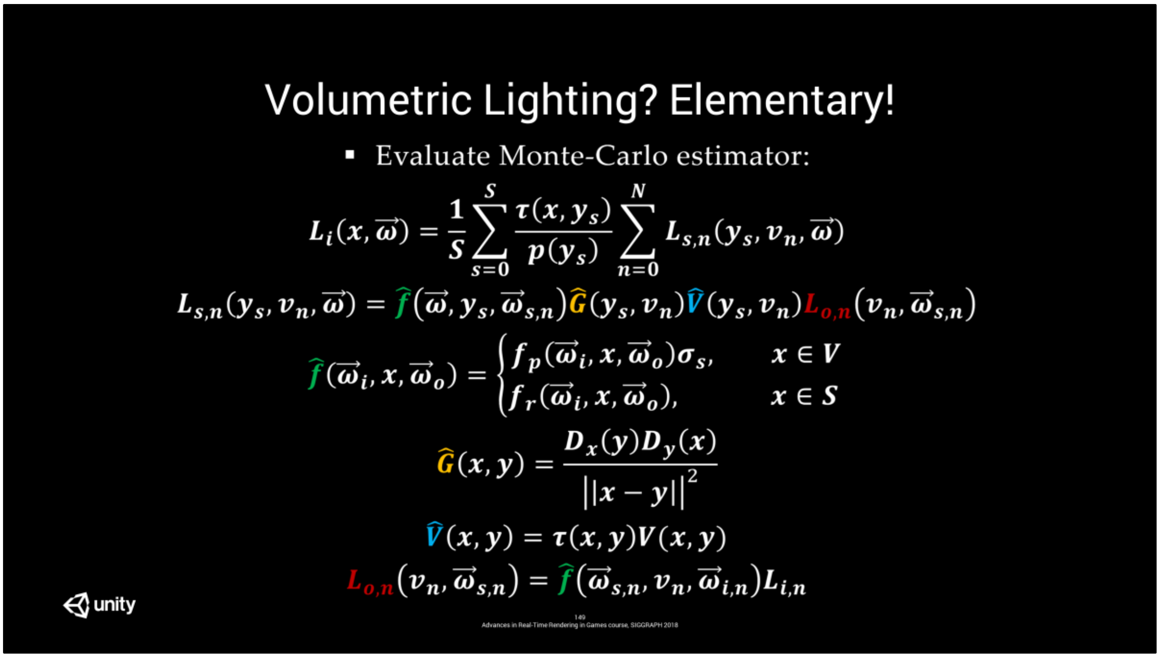
In order to determine partial coverage, we take the closest face of the box and compute its normal.
Then we take the diagonal of the voxel most aligned with this normal, and compute the overlap of this diagonal with the box.
This gives us an approximation of partial coverage of the voxel.



A solution involving voxelization is by its nature an approximation. The largest issue is resampling, which causes an irreversible loss of information.

Ideally, during the lighting pass, we would integrate over individual ray segments overlapping density volumes, skipping voxelization altogether.

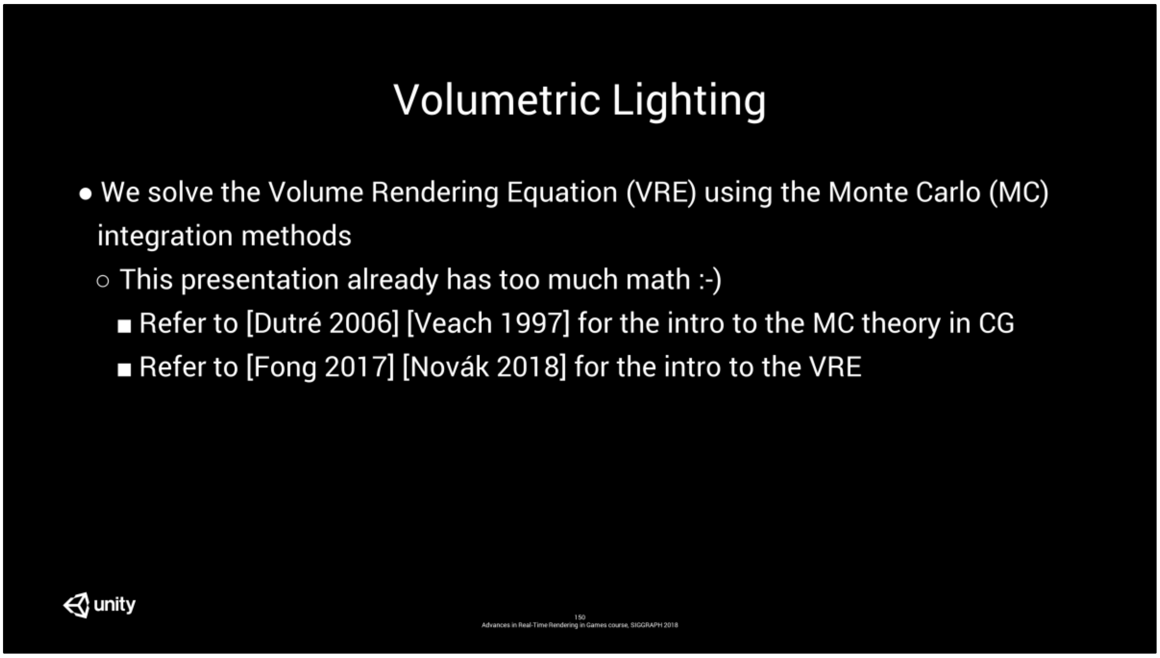
However, since volumetric lighting is already quite expensive, we prefer to have an approximation involving two simpler shaders over a giant ubershader with nested loops.



We solve the Volume Rendering Equation using Monte Carlo. It’s “just” a plain old recursive multidimensional integral.

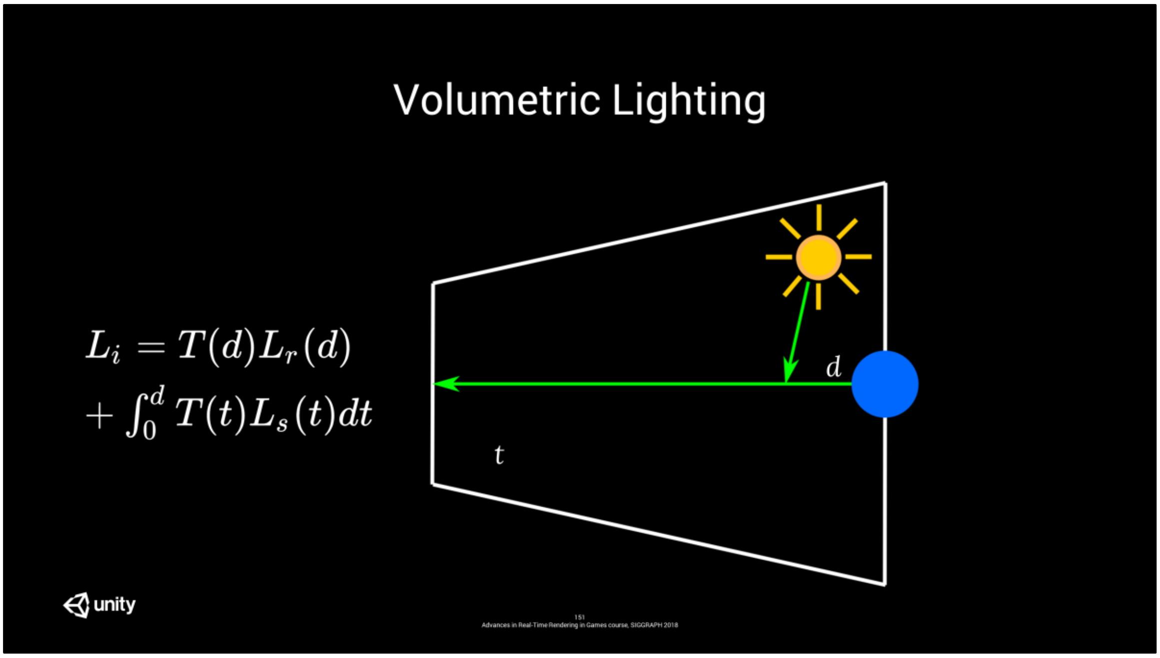
Instead of spending 10 minutes on this slide, …

Note: joke slide, don’t waste time deciphering this one. :-)



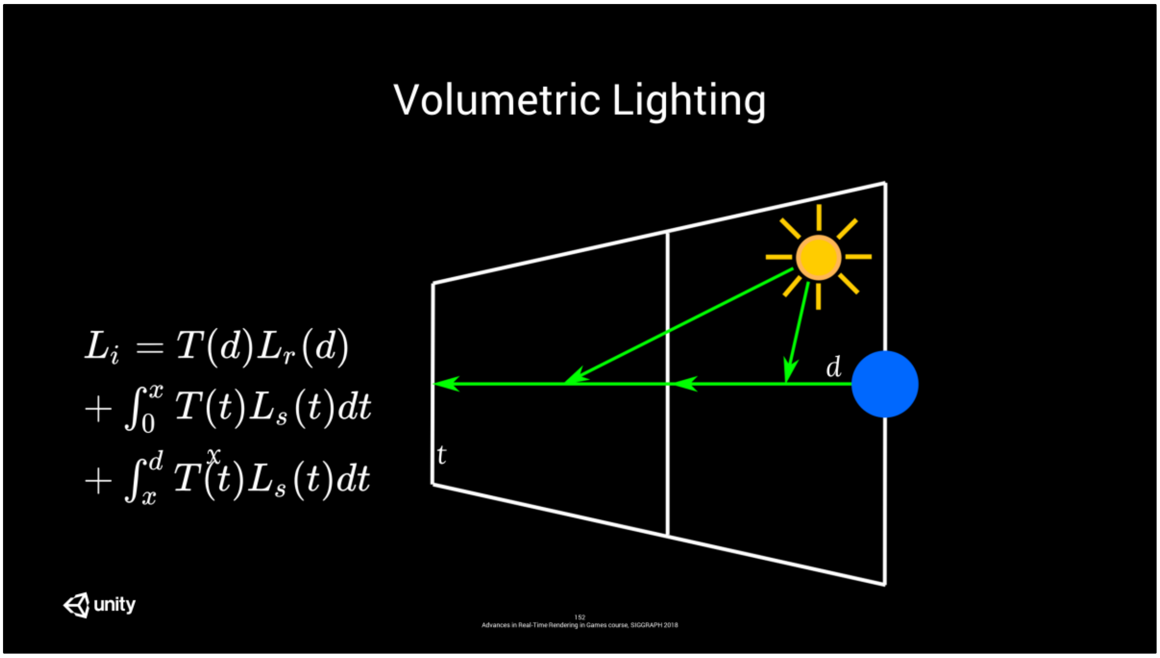
… I will only cover the way the math applies to our use case of voxel buffer lighting.

If you feel lost, please check out the references.



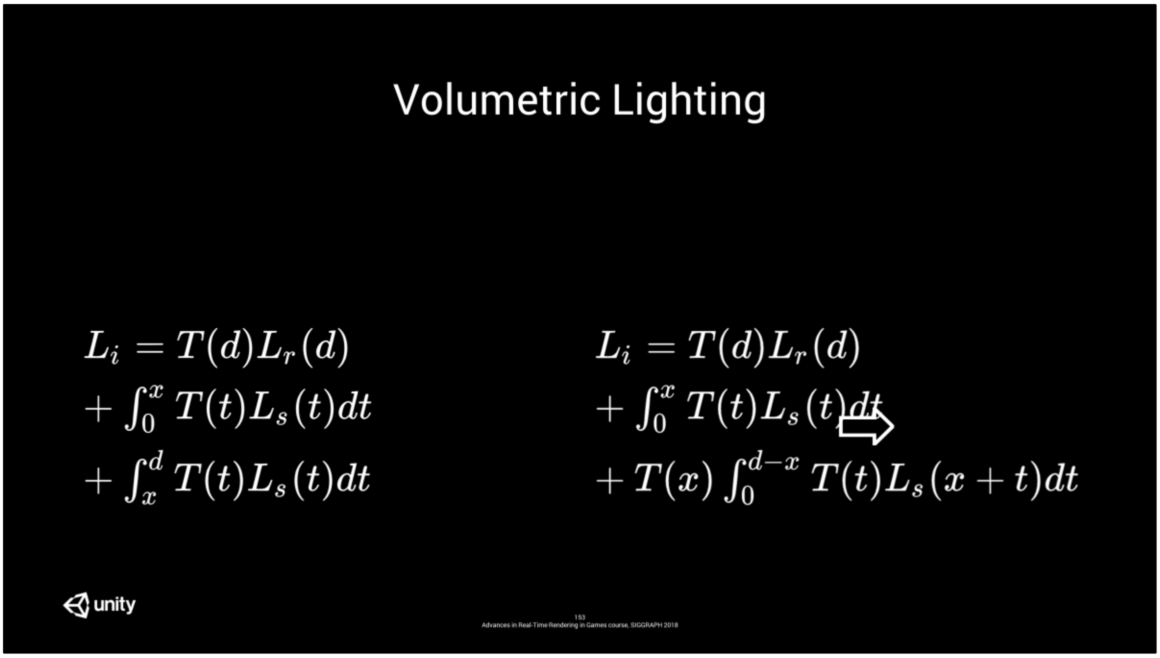
Imagine that *d* is the distance to the closest opaque surface along the ray.

In that case, the amount of incoming radiance *L_i* is the sum of reflected radiance *L_r* attenuated by transmittance *T*, and the integral of in-scattered radiance *L_s* continuously attenuated by transmittance *T* along the ray.

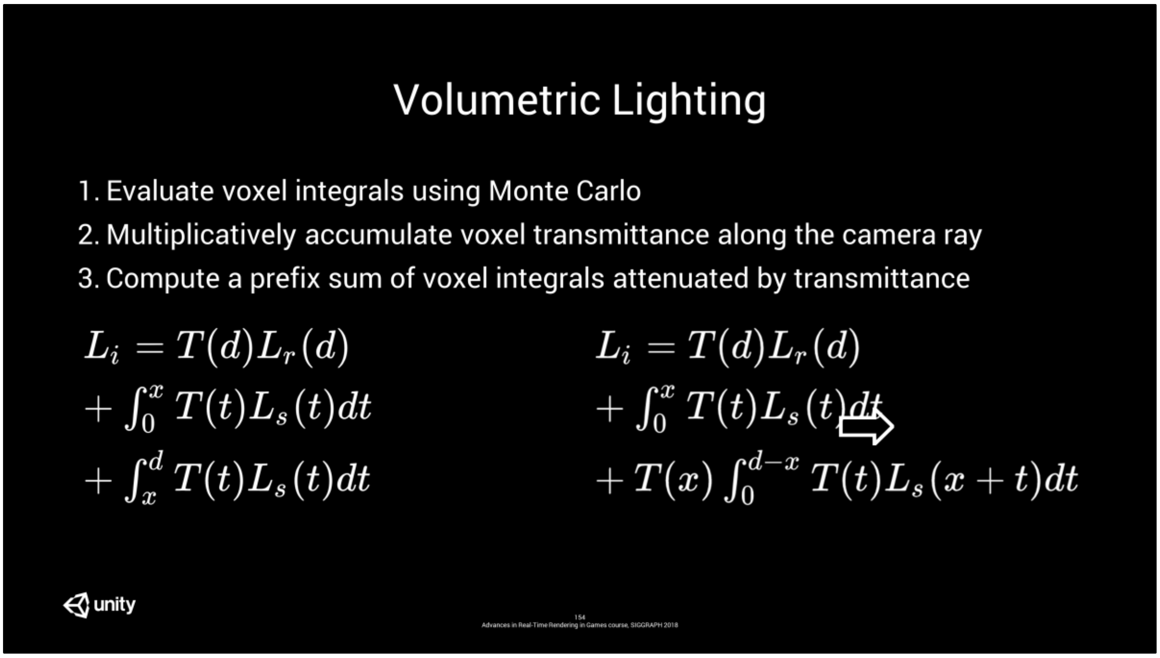


In case there are two voxels along the ray, it’s trivial to split the integral in two.

5s pause

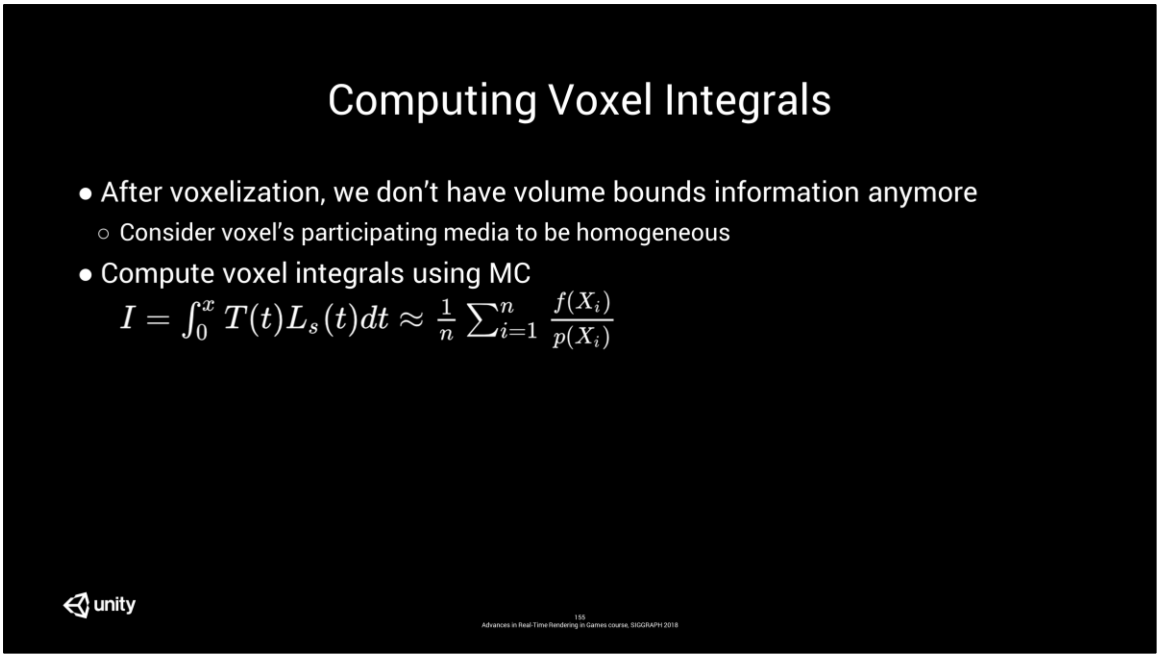


Finally, we can utilize the multiplicative property of transmittance to obtain independent voxel integrals.



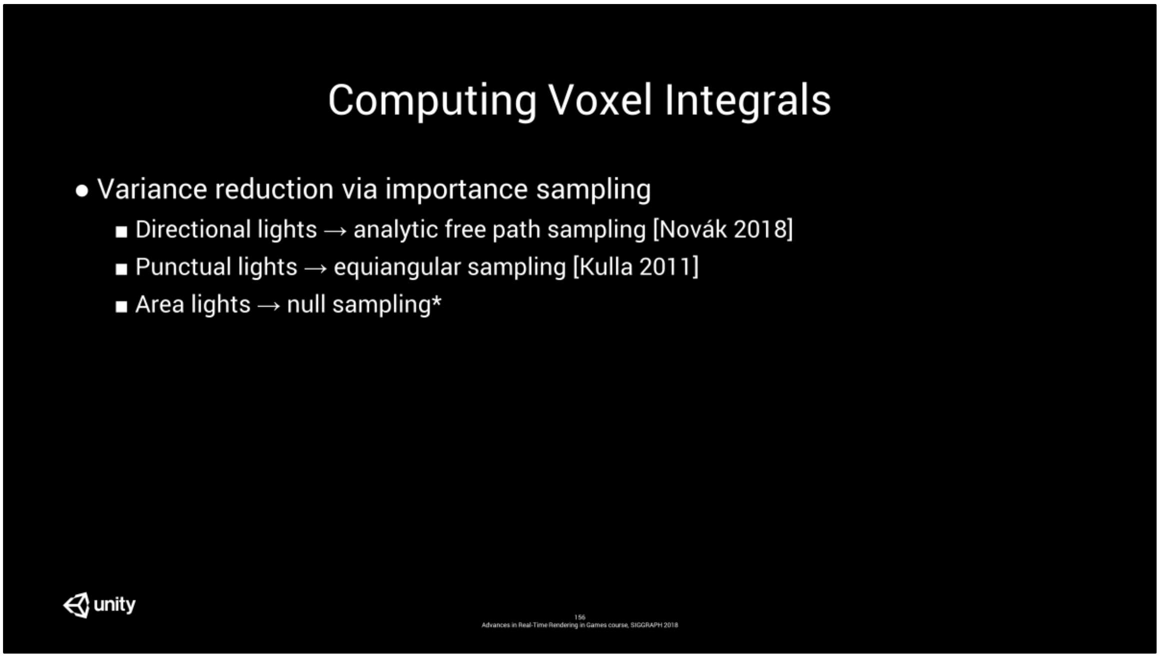
Therefore, our lighting algorithm is conceptually very simple:

- evaluate voxel integrals using Monte Carlo
- multiplicatively accumulate voxel transmittance along the camera ray
- finally, compute a prefix sum of voxel integrals attenuated by transmittance

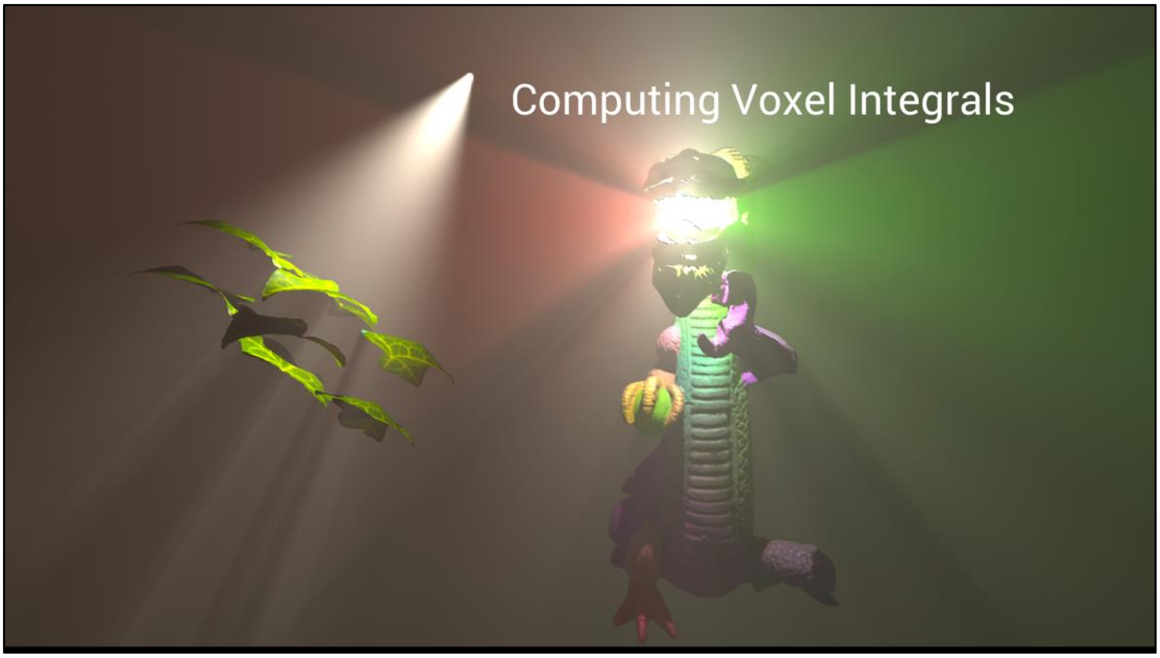


Since we pre-voxelize density volumes, we can consider voxel's participating media to be homogeneous. This considerably simplifies integral evaluation.

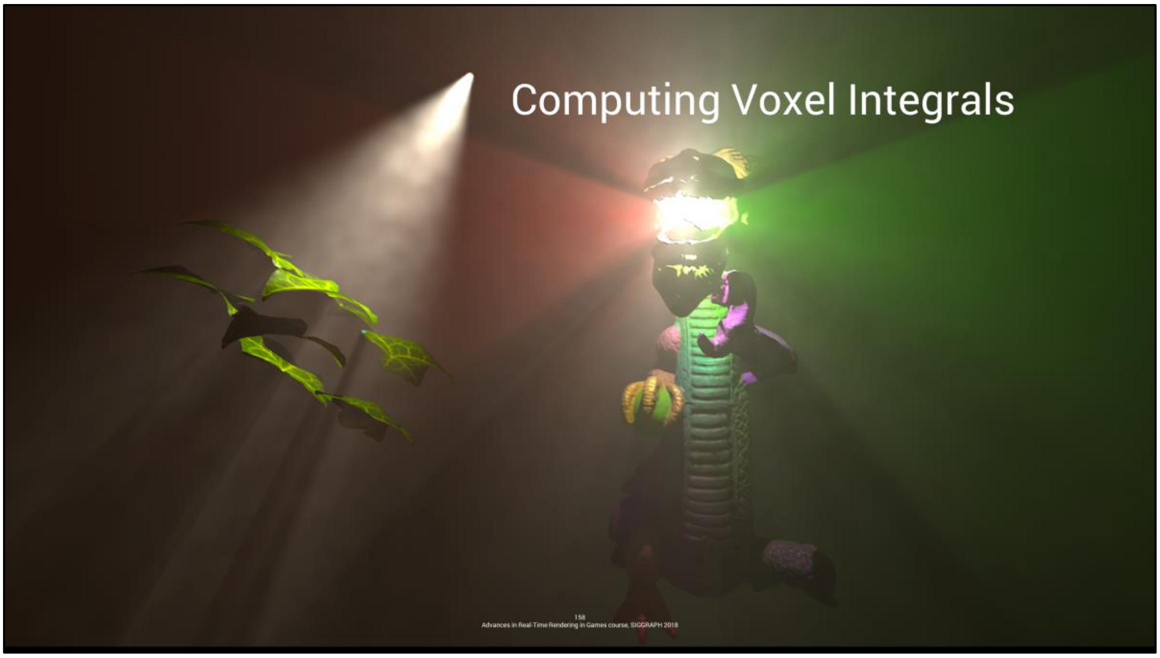
We use Monte Carlo tools for the job. In particular, we use importance sampling for variance reduction.



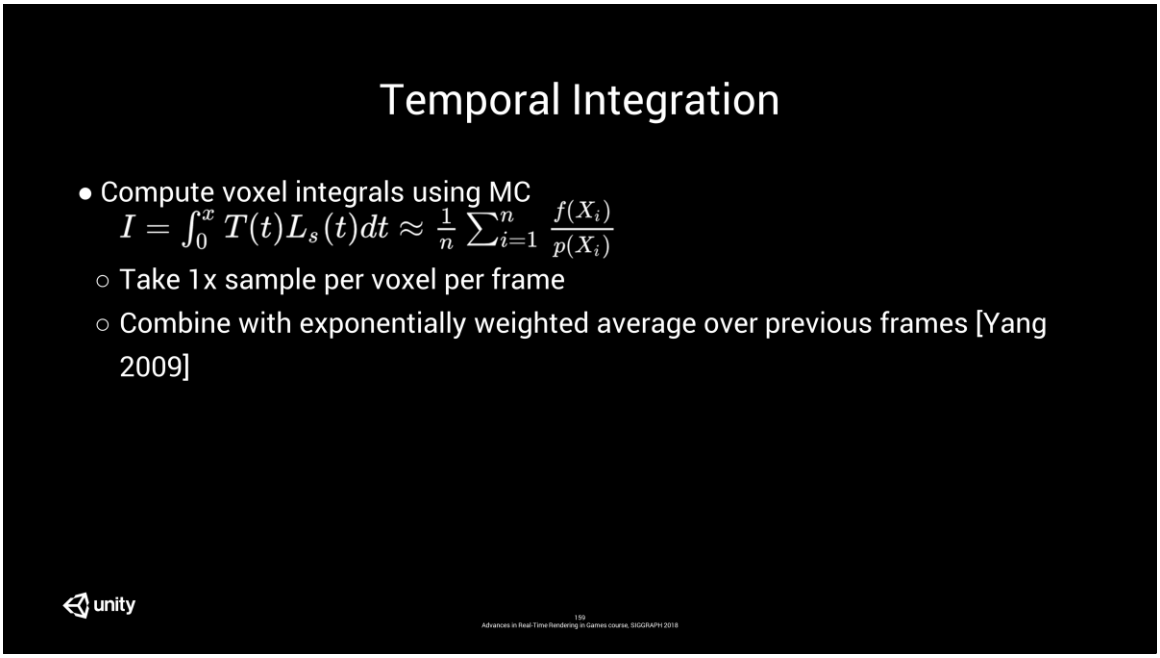
For directional and box projector lights, we use analytic distance sampling.
For punctual and spot lights, we use equiangular sampling, which is designed to handle inverse square attenuation.
For area lights, we use null sampling, which means we take 0 samples because area lights are not yet supported (sorry, Eric).



Here is some programmer art with equiangular sampling in action. This is global fog...

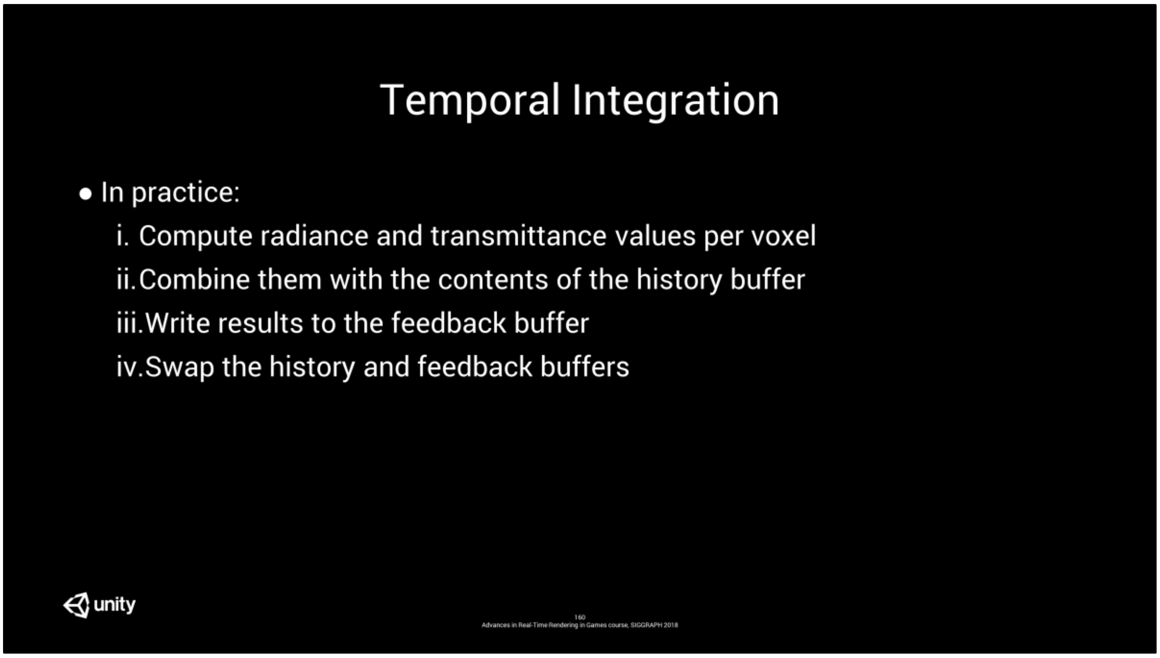


And this one uses a density volume, giving the fog a spatially-varying texture.



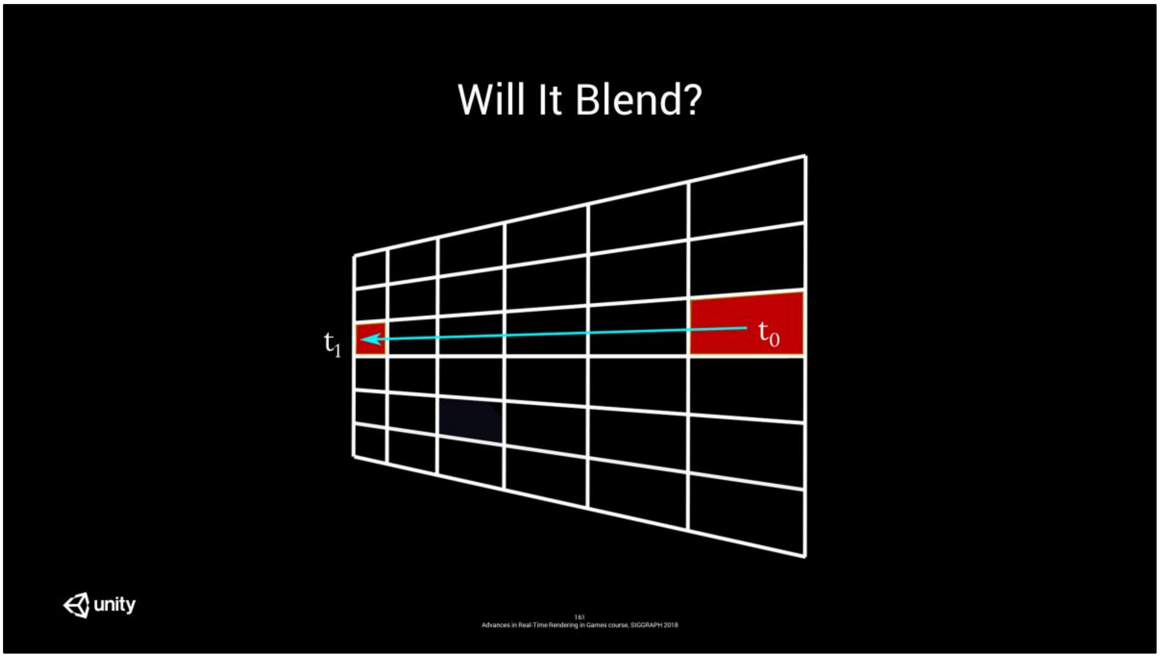
Monte Carlo integration usually involves taking several samples. However, taking more than one sample per voxel every frame is typically too expensive, especially on the current generation of console hardware.

Therefore, we take a single sample per voxel per frame instead, and then combine it with exponentially weighted average over previous frames.



In practice, we compute radiance and transmittance estimates per voxel, combine them with the contents of the history buffer, and write the results into the feedback buffer.

We perform reprojection in the world space, trilinearly interpolating radiance and transmittance estimates from 8 closest voxels.



How do we perform temporal blending?

Let’s say that we computed voxel radiance and transmittance estimates at time 0, and want to reproject and combine them with estimates at time 1.
If we have a fast forward camera motion, we may end up reprojecting from the voxel at the back to the voxel at the front.
What’s immediately obvious is that their dimensions are very different. Therefore, radiance and transmittance estimates are not going to be similar.

For instance, you may experience brightness of your entire screen changing as a result of fast camera motion.

So, what do we do?

The idea is to somehow “normalize” both radiance and transmittance estimates w.r.t. the voxel dimensions to obtain blendable densities.
However, the radiance integral over the length of the voxel and transmittance are not linear functions of length.
Transmittance is an exponential of optical depth, which is a linear function of length, so we can easily use that.

As for the radiance integral, the story is more complicated.
On the graph, as a function of length, I plotted the estimate of the integral given by the weight of analytic distance sampling [https://www.desmos.com/calculator/divvz5q57p].
The solid line represents the estimate, and the dotted line represents its 1st derivative w.r.t. length.

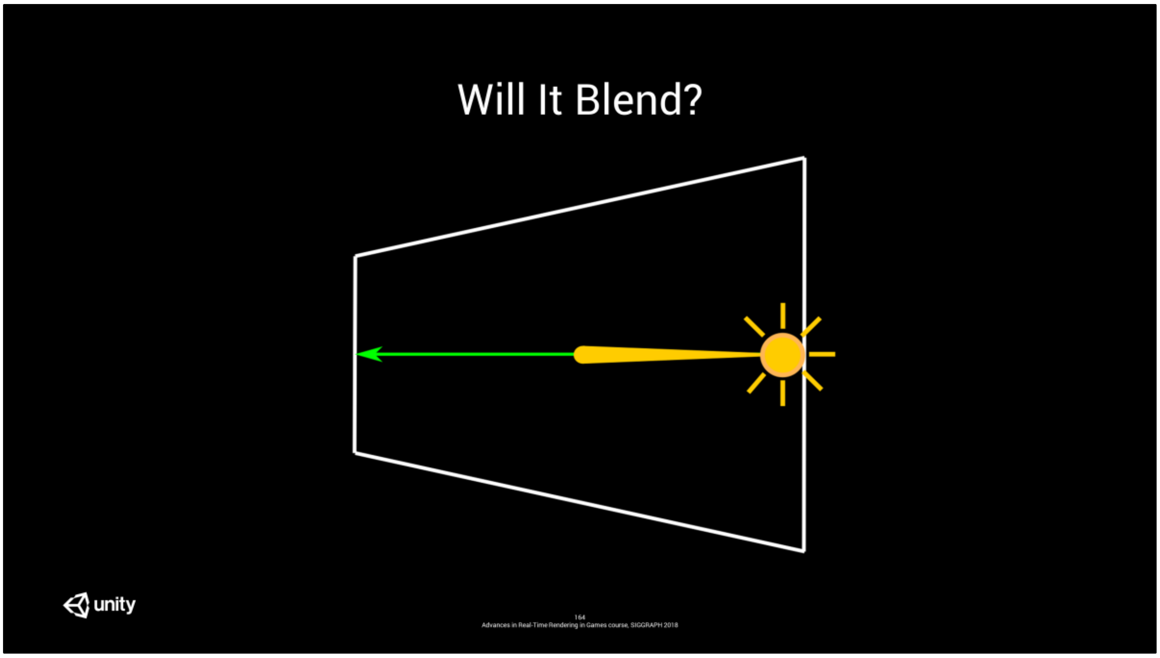
The derivative is non-constant, so the function is not linear. However, for small displacements the linear approximation is not too bad.

We can improve upon this a bit by dividing the estimate by the length, as shown here in green. It's much closer to being linear.

Another idea is to integrate incoming radiance along the unit interval rather than the actual length of the voxel, and use that for reprojection.
While both of these methods are relatively simple for directional lights, correctly handling punctual lights with equiangular sampling remains an open problem.

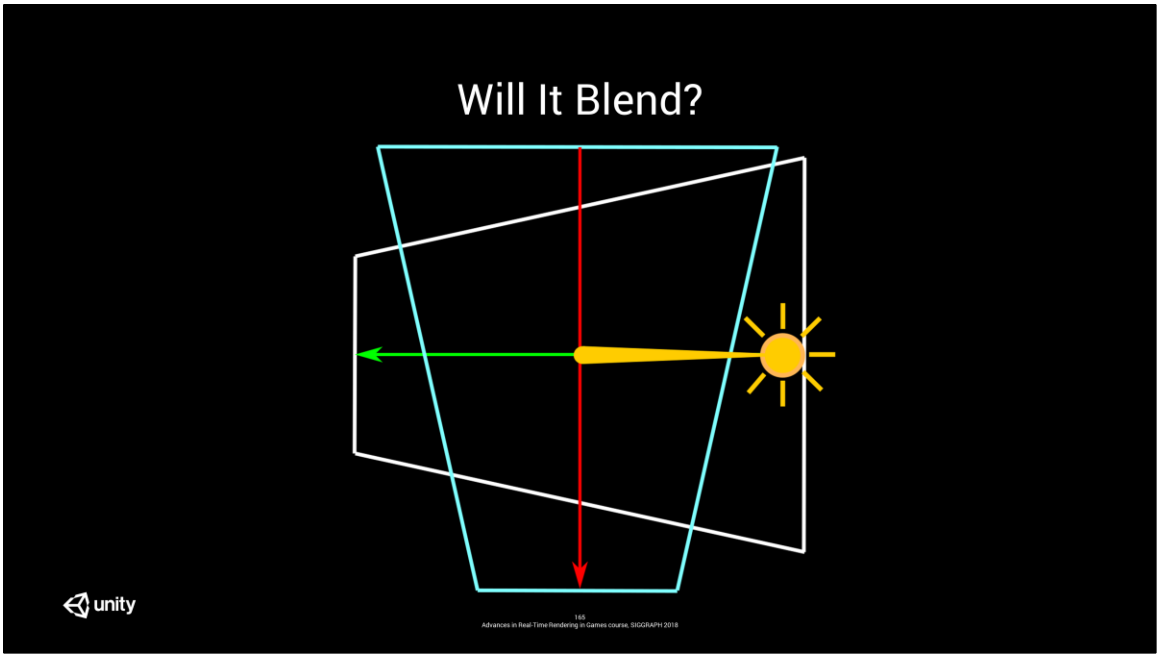
Also, I suspect a more elegant, generic solution exists. If you have one, please let me know!

Given a reprojected voxel with "normalized" radiance and transmittance, we can rescale it back using the length of our current voxel, and then blend it with the estimate from the current frame. The correct way of volume blending is given by Tom Duff in his paper titled "Deep Compositing Using Lie Algebras" [Duff 2017].



However, there is a catch: all of this works assuming that the phase function is isotropic. Generally speaking, it's not.

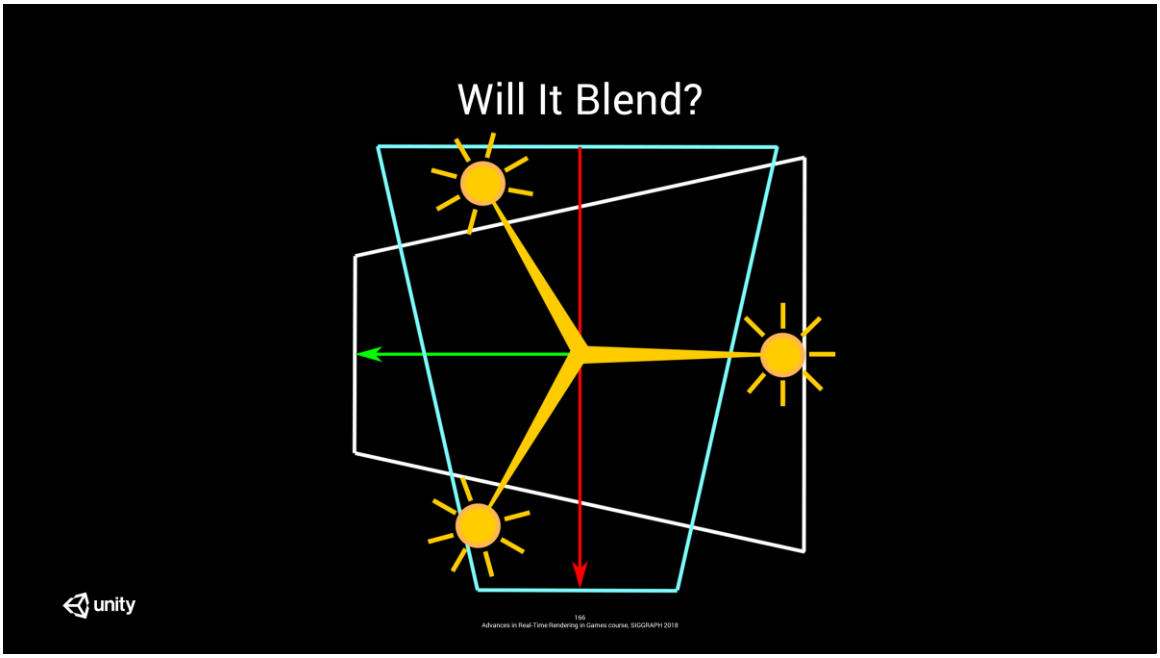
Let's say that we have a strongly forward-scattering phase function. If we look straight at a directional light, we obtain a high radiance estimate for the voxel.



If we rotate our camera, and therefore our voxel, by 90 degrees, we obtain a very low radiance estimate, since our medium is forward scattering, and we are facing the light at the right angle.

Therefore, the high value reprojected from the previous frame is no longer valid in the current context.

You can try to be clever, and say that since we know the light direction, we can rescale the phase function of the previous frame to fit the direction of the current frame...



However, once you have several lights illuminating the voxel, it's "game over" for this approach.

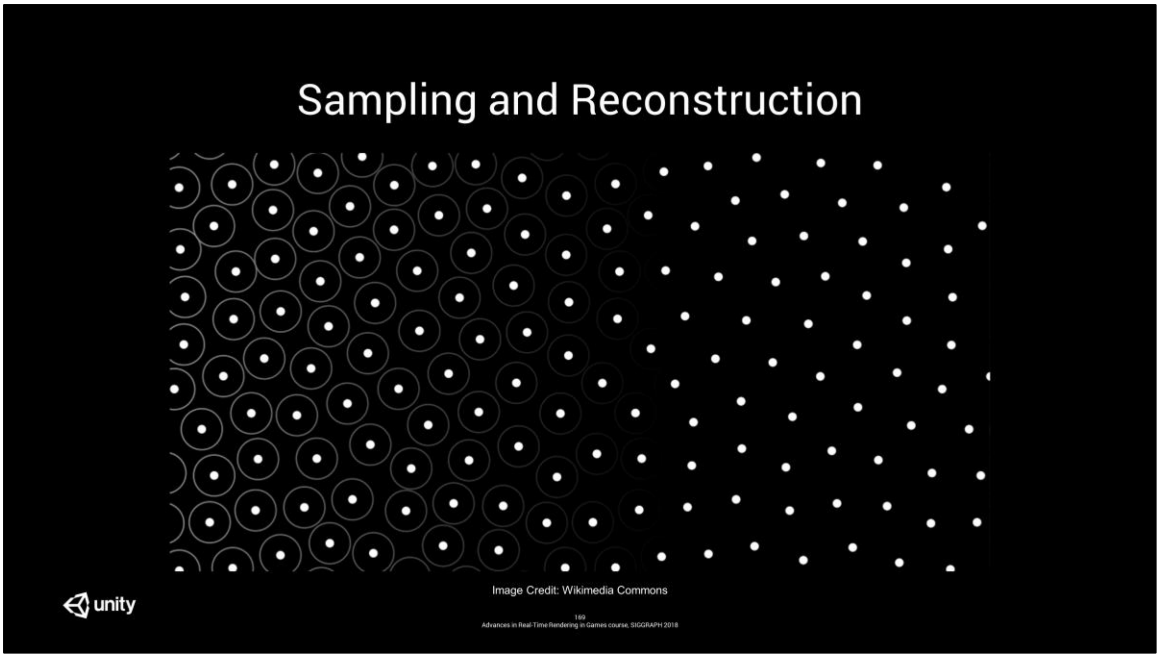
So, what is the solution?
I can only offer a workaround which, nonetheless, works reasonably well in practice.
When computing the voxel integral, we compute two estimates… One multiplied by the phase function, and one that is not. We store the isotropic version in the history buffer, and that is what we reproject.
To reconstruct the influence of the phase function during the current frame, we divide the estimate with the phase function by the one without it, and use the ratio to rescale the reprojected radiance.
This excludes anisotropy from the temporal integration process, which is of course bad, but it also removes a lot of jarring reprojection artifacts.

One of the most obvious results of temporal integration is the reduction in shadow aliasing and banding, as you can see here.

flip back and forth

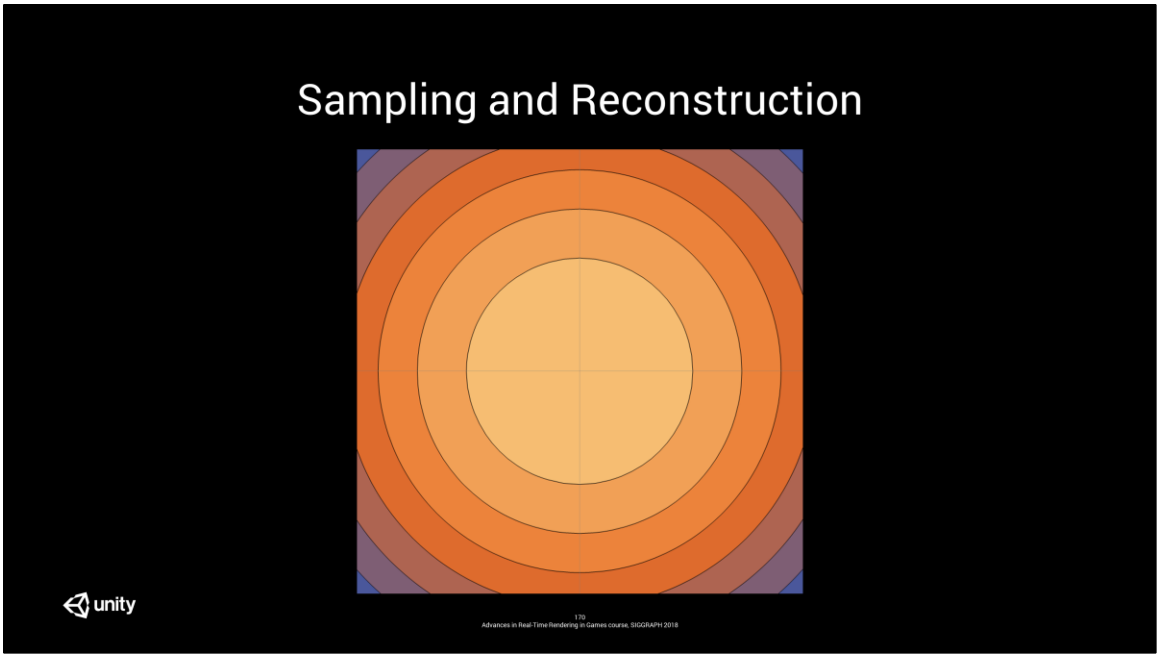
One of the most obvious results of temporal integration is the reduction in shadow aliasing and banding, as you can see here.

flip back and forth



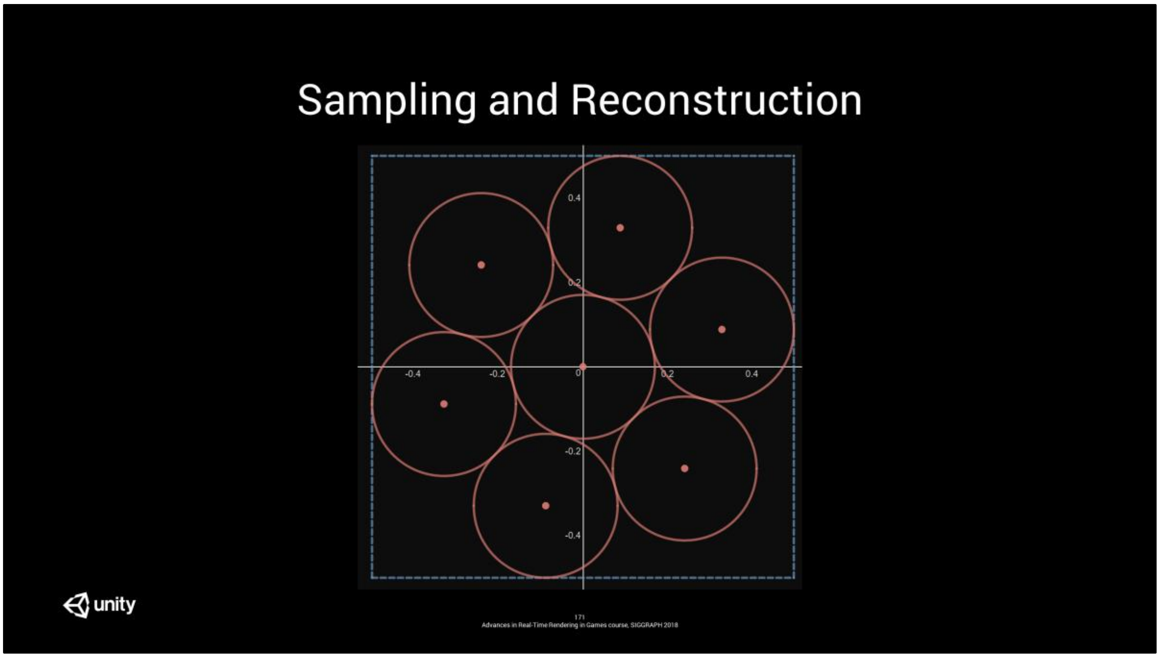
High quality sampling is essential for quick convergence of the Monte Carlo algorithm. We also need to ensure that our low resolution buffers are well anti-aliased, since all issues will be magnified by upsampling.

For high convergence rates, we want our sampling pattern to be rather uniform.
We also want the spectrum of our sampling pattern to contain most of its energy in high frequencies, which are less perceptible to the human observer, **and** which are going to be attenuated by the low-pass component of the reconstruction filter [Mitchell 1991].
These are blue-noise, or Poisson-hypersphere properties.

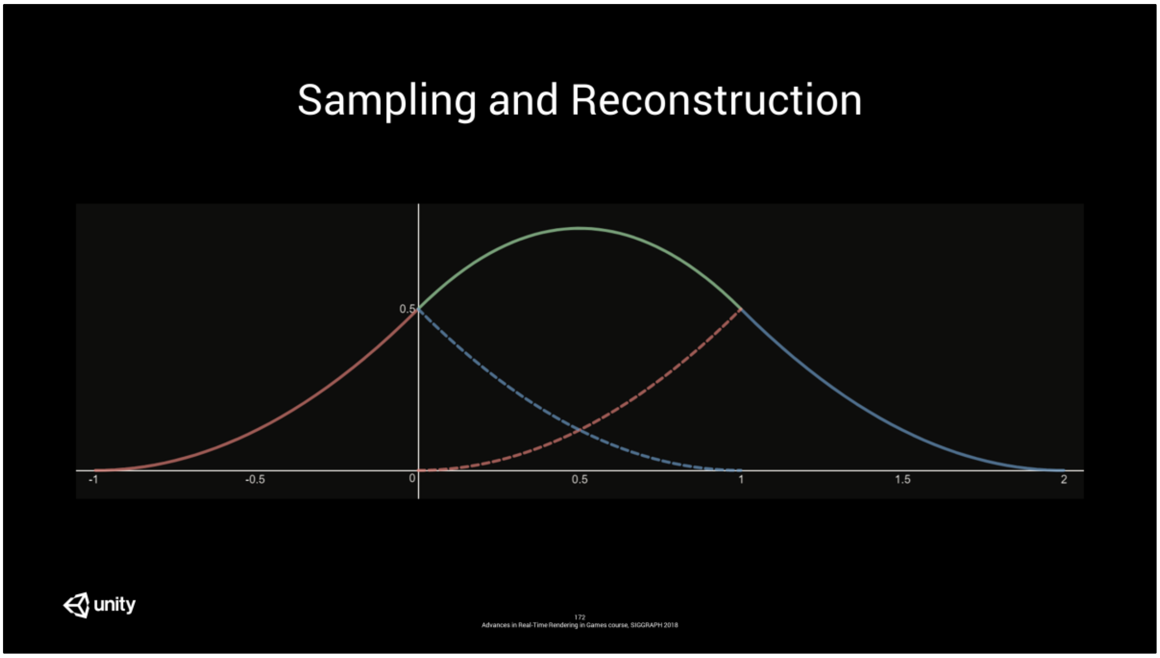


Additionally, as perceptively noted by Timothy Lottes, the shape of the sampling pattern should approximate a good reconstruction filter. While it's difficult to have spatially-varying weights in the temporal integration context, we can at least make sure that the footprint of the pattern is circular rather than square [Smith 1995].

Finally, while using a random pattern can mask structured artifacts with noise, it becomes more difficult to control the quality of the resulting distribution.



Therefore, we decided to use a deterministic pattern called hexagonal sphere-packed lattice [Wiki H]. It is the highest density sphere packing, and it fits all of our criteria.
We slightly rotate the pattern by 15 degrees to minimize the discrepancy along the X and Y axes.
Currently, each pixel uses the same pattern, but we would like to try per-pixel rotations in the future.
To avoid visible jitter, we make sure to traverse the samples in the order which keeps the average of coordinates as close to the center as possible.
Finally, Don Mitchell's paper [Mitchell 1991] tells us that in addition to the Poisson sphere properties in 3D, the distribution should satisfy Poisson rod properties after projections onto individual axes.
With that in mind, for now, we simply use a uniform distribution along the Z axis. We are planning to explore sphere packing in 3D in the future.



To upsample the volumetric lighting buffer, we perform biquadratic filtering in the screen plane [Getreuer 2011], and simple linear filtering in Z.
It’s 4 bilinear taps in total.
The idea is to limit both the memory bandwidth and the spatial extent of the filter, which tends to be quite large due to the low resolution of the buffer.
Additionally, we use bilateral filtering, which basically means that the coordinate of the texture look-up depends on the depth of the closest surface.
It would be interesting to experiment with generalized filters in the future [Nehab 2014].

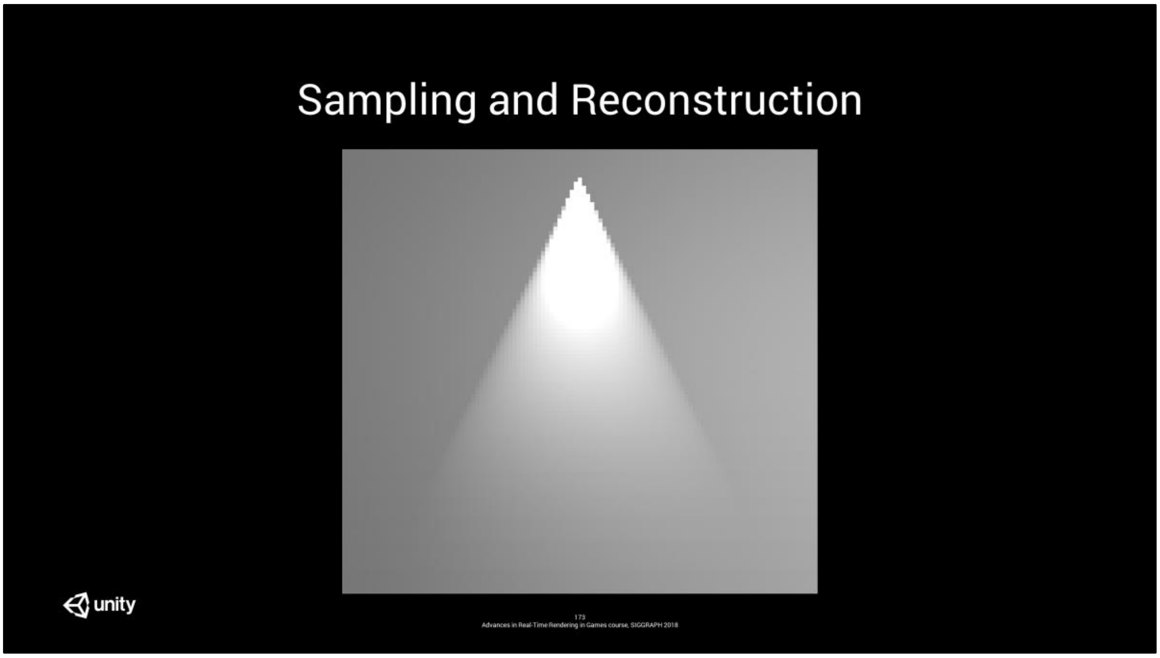
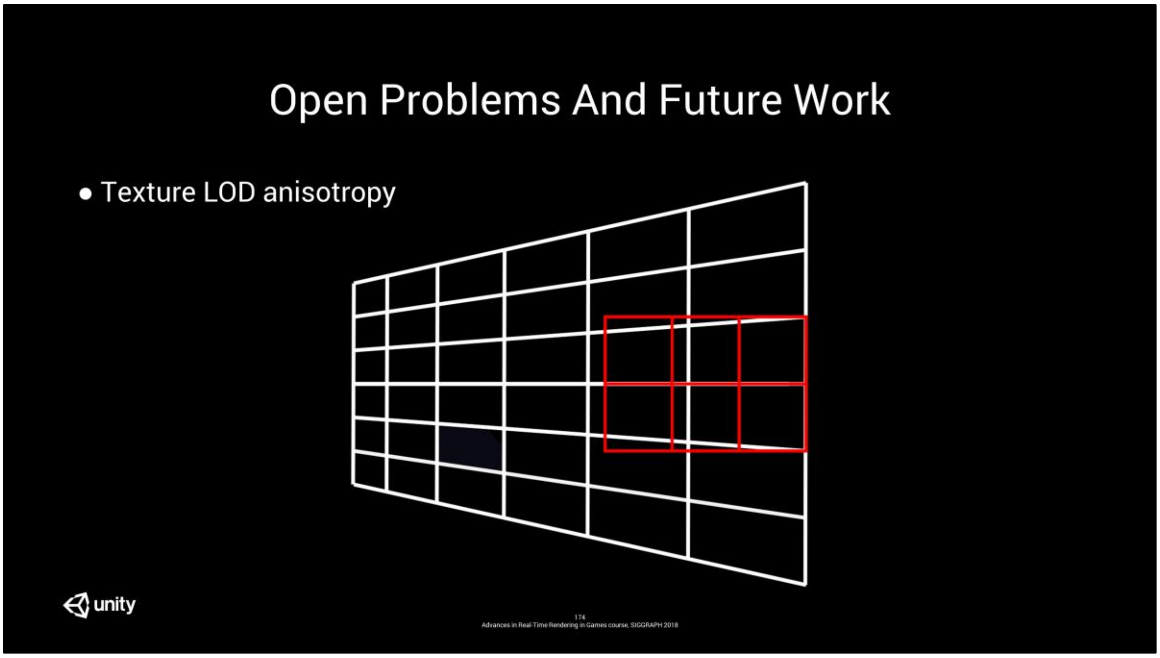
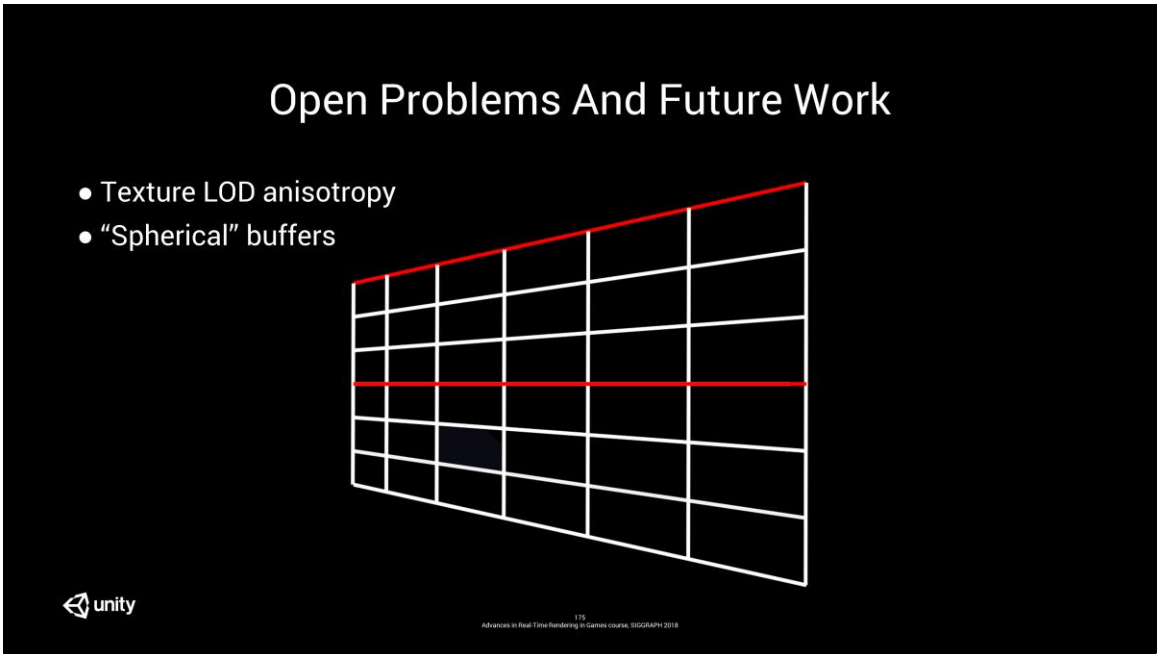


Image upsampling should be performed in the perceptually-linear space [Nehab 2014].
Therefore, we upsample tone-mapped radiance and transmittance rather than physically-linear optical depth.
Interestingly, the same should be done for anti-aliasing [Persson 2008]. However, the Monte Carlo formulation of the temporal integrator expects physically-linear rather than perceptually-linear values.
So there’s a certain conflict between correctness and aliasing in this case.

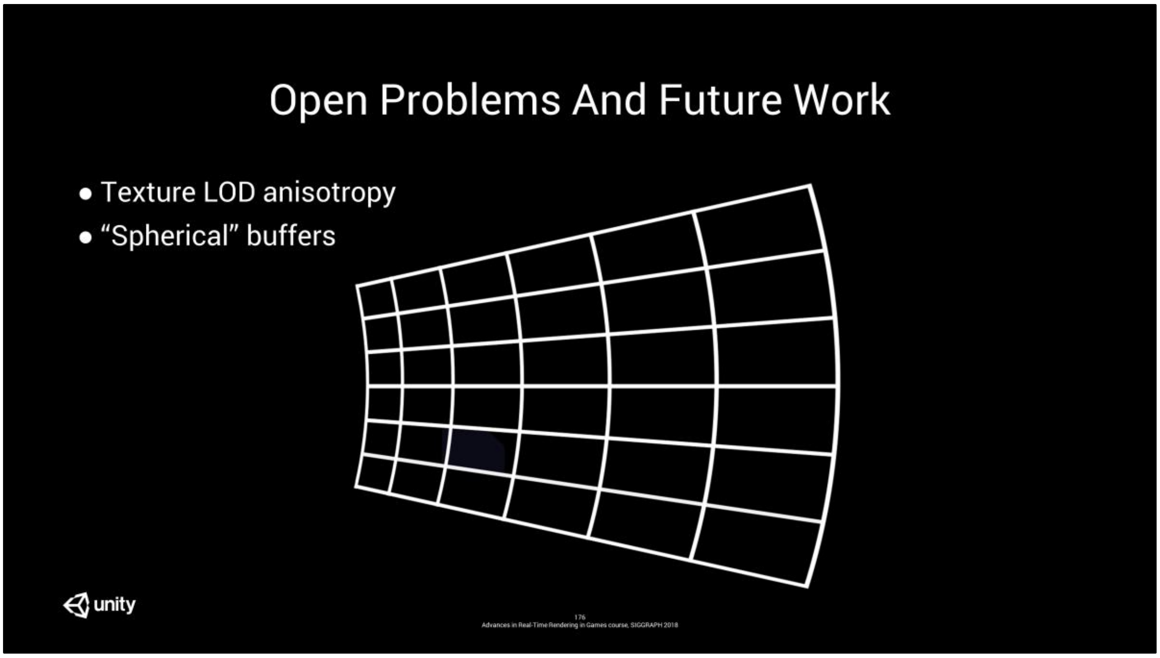


Finally, I’d like to make a few comments about open problems and future work.

Voxels usually have highly anisotropic footprints in the texture space. However, the hardware doesn’t offer anisotropic filtering support for 3D textures, which means that we overfilter in practice.
Manually writing a texture filter loop in the shader is not particularly appealing from the performance perspective.
It is worth noting that we have exactly the same issue with anisotropic cubemap filtering.



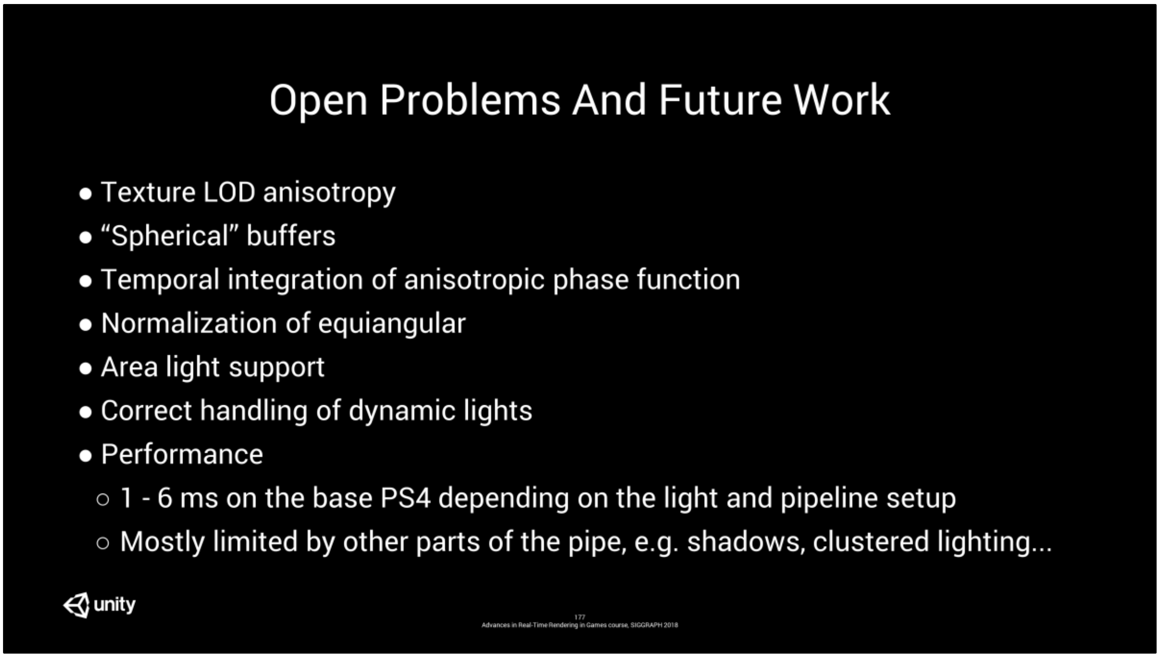
Another issue is that, since volumetric buffers are shaped like a frustum, distances from the near to the far plane increase as you move away from the center of the screen.
This results in different radiance and transmittance estimates even for constant lighting and participating media.
It can show up as darkened corners during cubemap rendering, for example.



The solution is to use spherical buffers, parametrized by distance from the eye rather than depth.

Additionally, rotational reprojection becomes nearly perfect, which is a nice bonus.

We had a concern that it could make light culling more complicated, but turns out that it may actually be more efficient than the traditional methods [Zhang 2018].



As I've already mentioned, temporal support of anisotropy and equiangular needs improve.
We need to support area lights.
Dynamic lights need to be handled correctly. Marco's Variance Clipping seems promising [Salvi 2016].
Finally, performance needs to improve. The current numbers are mostly limited by the cost of shadows, and clustered lighting not being scalarized on GCN.

Thank You
And to all the Unity team and people involved
in the work of High Definition Render Pipeline
HDRP team
Postprocess team
Graphic fondation team
Lighting team
Spotlight team
MWU team
Platform team
Unity Labs
Natalya Tatarchuk
Emmanuel Turquin
Ronan Marchalot
Stéphane Laroche
178
Advances in Real-Time Rendering in Games course, SIGGRAPH 2018

Please note that the source code of hdrp is available on github at this link above. You can retrieve the code mention in these slides

References

1. [Jensen 2001] Henrik W. Jensen, Stephen R. Marschner, Marc Levoy and Pat Hanrahan. *A Practical Model for Subsurface Light Transport.*
2. [Mikkelsen 2010] Morten S. Mikkelsen. *Skin Rendering by Pseudo-Separable Cross Bilateral Filtering*.
3. [Jimenez 2010] Jorge Jimenez, David Whelan, Veronica Sundstedt and Diego Gutierrez. *Real-Time Realistic Skin Translucency*.
4. [Jimenez 2013] Jorge Jimenez and Javier von der Pahlen. *Next-Generation Character Rendering*.
5. [Jimenez 2014] Jorge Jimenez, Károly Zsolnai, Adrian Jarabo, Christian Freude, Thomas Auzinger, Xian-Chun Wu, Javier von der Pahlen, Michael Wimmer and Diego Gutierrez. *Separable Subsurface Scattering*.
6. [d'Eon 2007] Eugene d'Eon, David Luebke and Eric Enderton. *Efficient Rendering of Human Skin*.
7. [Barré-Brisebois 2011] Colin Barré-Brisebois and Marc Bouchard. *Approximating Translucency for a Fast, Cheap and Convincing Subsurface Scattering Look*.
8. [Burley 2015] Brent Burley and Per H. Christensen. *Approximate Reflectance Profiles for Efficient Subsurface Scattering*.
9. [Wang 1995] Lihong Wang, Steven L. Jacques and Liqiong Zheng. *Monte Carlo Modeling of Light Transport in Multi-Layered Tissues*.
10. [Press 2007] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. *Numerical Recipes, 3rd Edition*.
11. [Morton 1966] G. M. Morton. *A Computer-Oriented Geodetic Data Base, and a New Technique in File Sequencing.*
12. [Hannay 2004] John H. Hannay and John F. Nye. *Fibonacci Numerical Integration on a Sphere*.

References

13. [Marques 2015] Ricardo Marques, Christian Bouville, Luís P. Santos and Kadi Bouatouch. *Efficient Quadrature Rules for Illumination Integrals*.
14. [Garawany 2016] Ramy E. Garawany. *Deferred Lighting in Uncharted 4*.
15. [Aaltonen 2017] Sebastian Aaltonen. *Optimizing Gpu Occupancy and Resource Usage with Large Thread Groups*.
16. [Vos 2014] Nathan Vos. *Volumetric Light Effects in Killzone: Shadow Fall*.
17. [Wronski 2014] Bart Wronski. *Volumetric Fog: Unified, Compute Shader Based Solution to Atmospheric Scattering*.
18. [Hillaire 2015] Sébastien Hillaire. *Physically Based and Unified Volumetric Rendering in Frostbite*.
19. [Wright 2017] Daniel Wright. *Volumetric Fog, Unreal Engine Livestream*.
20. [Fong 2017] Julian Fong, Magnus Wrenninge, Christopher Kulla, and Ralf Habel. *Production Volume Rendering*.
21. [Cornette 1992] William M. Cornette and Joseph G. Shanks. *Physically Reasonable Analytic Expression for the Single-Scattering Phase Function*.
22. [Toublanc 1996] Dominique Toublanc. *Henyey-Greenstein and Mie Phase Functions in Monte Carlo Radiative Transfer Computations*.
23. [Klemen 2012] Brano Klemen. *Maximizing Depth Buffer Range and Precision*.
24. [Laine 2013] Samuli Laine. *A Topological Approach to Voxelization*.

unity

References

25.[Veach 1997] Eric Veach. *Monte Carlo Methods for Light Transport Simulation.*

26.[Dutré 2006] Philip Dutré, Kavita Bala and Philippe Bekaert. *Advanced Global Illumination, 2nd Edition.*

27.[Novák 2018] Jan Novák, Iliyan Georgiev, Johannes Hanika and Wojciech Jarosz. *Monte Carlo Methods for Volumetric Light Transport Simulation.*

28.[Heitz 2014] Eric Heitz. *Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs.*

29.[Heitz 2016] Eric Heitz, Johannes Hanika, Eugene d'Eon and Carsten Dachsbacher. *Multiple-Scattering Microfacet BSDFs with the Smith Model.*

30.[Heitz 2017] Eric Heitz and Stephen Hill. *Real-Time Line- and Disk-Light Shading.*

31.[Lagarde 2014] Sébastien Lagarde and Charles de Rousiers. *Moving Frostbite to PBR.*

32.[Lagarde 2016] Sébastien Lagarde. *An Artist-Friendly Workflow for Panoramic HDRI.*

33.[Brisebois 2012] Colin Barré-Brisebois and Stephen Hill. *Blending in Detail.*

34.[Belcour 2017] Laurent Belcour and Pascal Barla. *A Practical Extension to Microfacet Theory for the Modeling of Varying Iridescence.*

35.[Kulla 2011] Christopher Kulla and Marcos Fajardo. *Importance Sampling of Area Lights in Participating Media.*

36.[Kulla 2017] Christopher Kulla and Alejandro Conty. *Revisiting Physically Based Shading at Imageworks.*

unity

References

37.[Yang 2009] Lei Yang, Diego Nehab, Pedro Sander, Pitchaya Sitthi-amorn, Jason Lawrence and Hugues Hoppe. *Amortized Supersampling*.

38.[Duff 2017] Tom Duff. *Deep Compositing Using Lie Algebras*.

39.[Mitchell 1991] Don P. Mitchell. *Spectrally Optimal Sampling for Distribution Ray Tracing*.

40.[Smith 1995] Alvy Ray Smith. *A Pixel Is Not A Little Square*.

41.[Wiki H] Wikipedia. https://en.wikipedia.org/wiki/Close-packing_of_equal_spheres

42.[Getreuer 2011] Pascal Getreuer. *Linear Methods for Image Interpolation*.

43.[Nehab 2014] Diego Nehab. *A Fresh Look at Generalized Sampling*.

44.[Persson 2008] Emil Persson. *Post-Tonemapping Resolve for High Quality HDR Antialiasing in D3D10*.

45.[Zhang 2018] Eric Zhang. *Improved Tile-Based Light Culling with Spherical-Sliced Cones*.

46.[Salvi 2016] Marco Salvi. *An Excursion in Temporal Supersampling*.

47.[Mikkelsen 2008] Morten S. Mikkelsen. *Simulation of Wrinkled Surfaces Revisited.*

48.[Mikkelsen 2010] Morten S. Mikkelsen. *Bump Mapping Unparametrized Surfaces on the GPU*.

49.[Mikkelsen 2016] Morten S. Mikkelsen. *Fine Pruned Tiled Light Lists*.

50.[Olsson 2012] Ola Olsson, Markus Billeter and Ulf Assarsson. *Clustered Deferred and Forward Shading.*

unity

References

37.[Burley 2012] Brent Burley. *Physically-Based Shading at Disney*.
38.[McAuley 2015] Steve McAuley. The rendering of far cry 4.
39.[Revie 2011] Donald Revie, Implementing Fur Using Deferred Shading.
40.[Lagarde 2011] Sébastien Lagarde. Feeding a physically based shading model.
41.[Lagarde 2013] Sébastien Lagarde. Memo on Fresnel equations.
42.[Brinck 2016] Waylon Brinck and Andrew Maximov. Technical art of Uncharted 4.
43.[Sousa 2016] Tiago Sousa and Jean Geffroy. The devil is in the details: idTech 666.
44.[Cantlay 2007] Iain Cantlay. High-Speed, Off-Screen Particles
45.[Coffin 2011] Christina Coffin, SPU-Based Deferred Shading in BATTLEFIELD 3 for Playstation 3
46.[Hill 2016] Stephen Hill. LTC Fresnel Approximation.

unity