REAL-TIME RAY TRACING WITH MDL

Ignacio Llamas & Maksim Eisenstein, 03.21.2019 - **GPU Technology Conference**

AGENDA

A brief Introduction to MDL

Using MDL in a Monte Carlo Renderer

Using MDL in a Real-Time Ray Tracing Renderer

A BRIEF INTRODUCTION TO MDL

The NVIDIA Material Definition Language (MDL)

A language to declaratively and procedurally define physically-based materials for physically-based rendering solutions.

Describes what we need to know to model how light interacts with a material

Does not specify how to render.

That's up to the renderer

HOW IS MDL DIFFERENT

HLSL, GLSL

Lower level, generic procedural shading languages

Not specific to materials in any way

MATERIALX + SHADERX

Format for network-based CG looks

Specific to a production pipeline and tools

No standard xDFs

OSL

Language for programmable shading in advanced renderers

Material closures, support a few BxDFs, VDFs, EDFs. No combiners

MDL

Energy-conserving BxDF, EDF and VDF

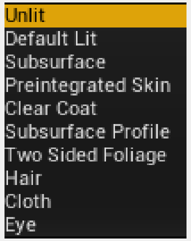
Elemental xDFs+Combiner xDFs = xDF graphs

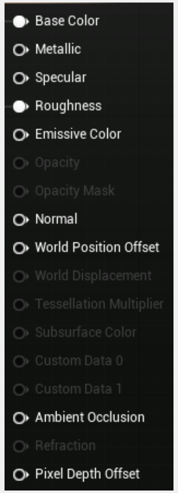
Declarative and Procedural language

REAL-TIME RENDERING ENGINE MATERIALS

Comparing to MDL

10 models, 17 params:
Lit: Diffuse + GGX
Clear Coat: Lit + 1 GGX
Others: varies





UE4

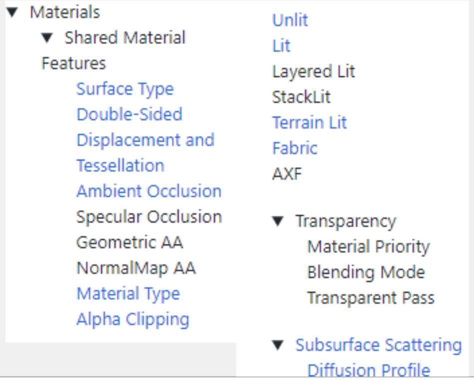
Generalization able to represent many materials:
10 elemental BSDFs
3 elemental EDFs
1 elemental VDF

2 mixing BSDFs, EDFs, VDFs
4 layering BSDFs

5 bsdf modifiers
1 edf modifier

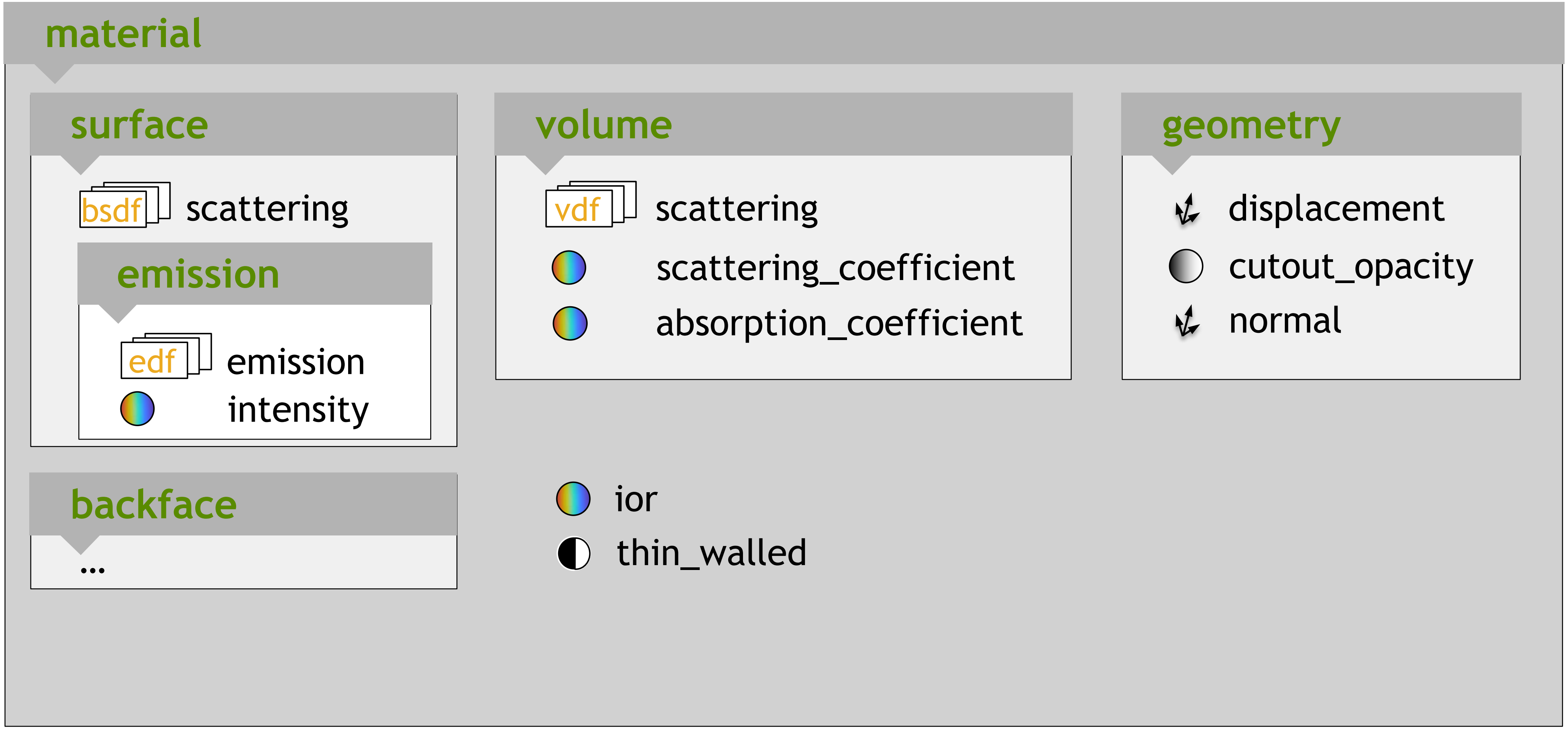
MDL

7 shading models:
Lit: Disney Diffuse + GGX



Unity

MDL MATERIAL MODEL



DEFINING A MATERIAL USING MDL

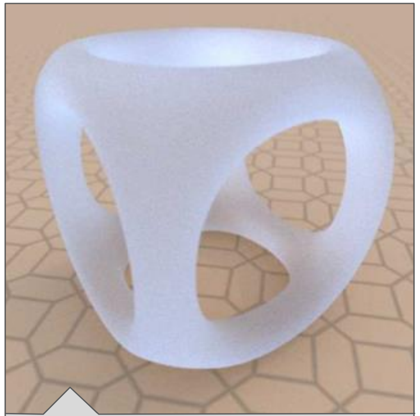
MDL is a 'C' like language. The material and its components viewed as a struct

```
struct material {
    bool              thin_walled  = false;
    material_surface  surface      = material_surface();
    material_surface  backface     = material_surface();
    color             ior          = color(1.0);
    material_volume   volume       = material_volume();
    material_geometry geometry     = material_geometry();
};

struct material_surface {
    bsdf              scattering   = bsdf();
    material_emission emission     = material_emission();
};
```

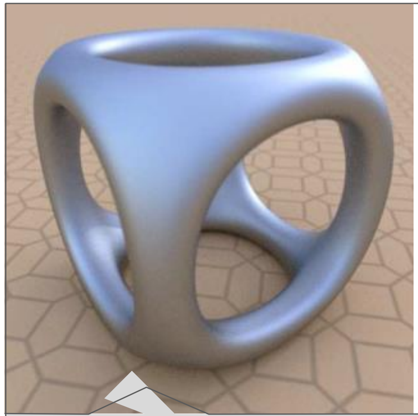
MDL ELEMENTAL DISTRIBUTION FUNCTIONS

Bidirectional
Scattering
Distribution
Functions

Diffuse Reflection

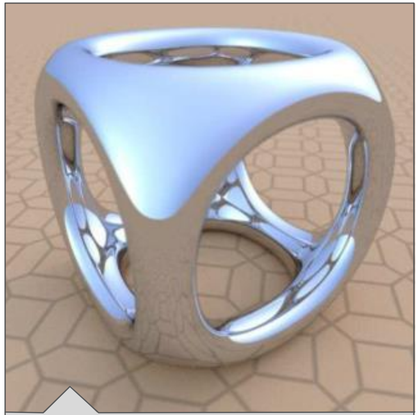


Diffuse Transmission

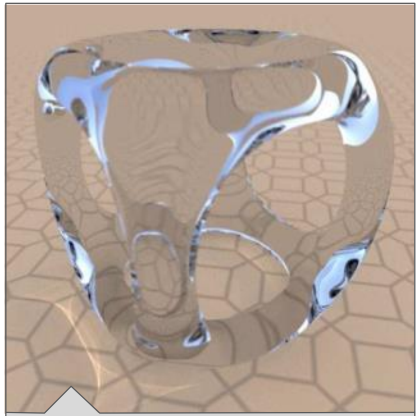


Glossy / Microfacet

Microfacet models:
- beckmann_smith
- ggx_smith
- beckmann_vcavities
- ggx_vcavities
- ward_geisler_moroder

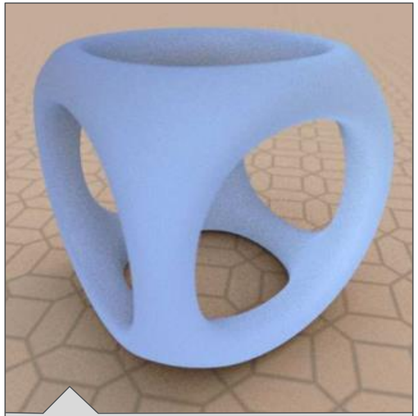


Specular Reflection



Spec. Refl.+Transm.

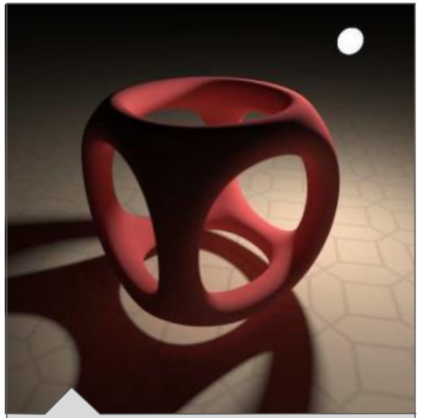
Measured BSDF



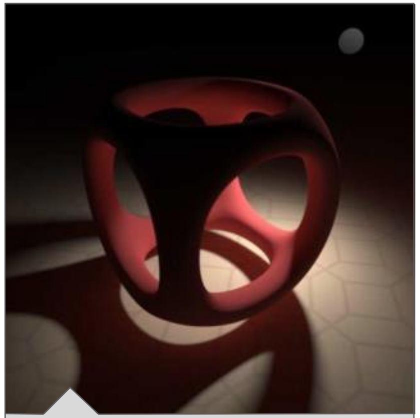
Backscatter Glossy

MDL ELEMENTAL DISTRIBUTION FUNCTIONS

Emissive
Distribution
Functions



Diffuse



Spot

IES Profile

Volume
Distribution
Functions

Henyey-Greenstein

NVIDIA

MDL DISTRIBUTION FUNCTION MODIFIERS

Tint

Thin Film

Directional Factor

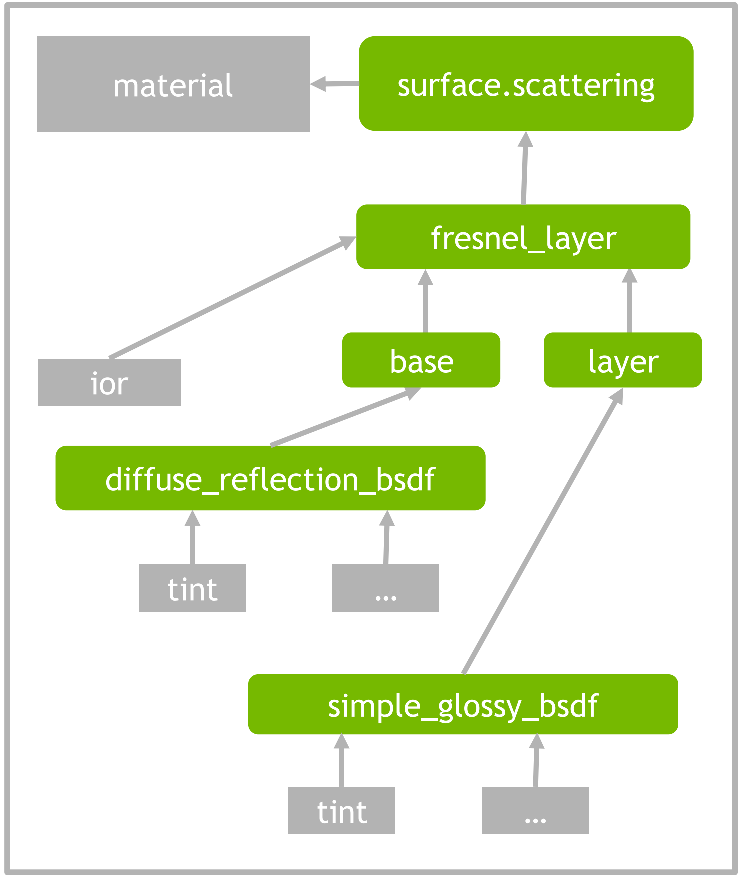
Measured Curve Factor

MDL DISTRIBUTION FUNCTIONS COMBINERS

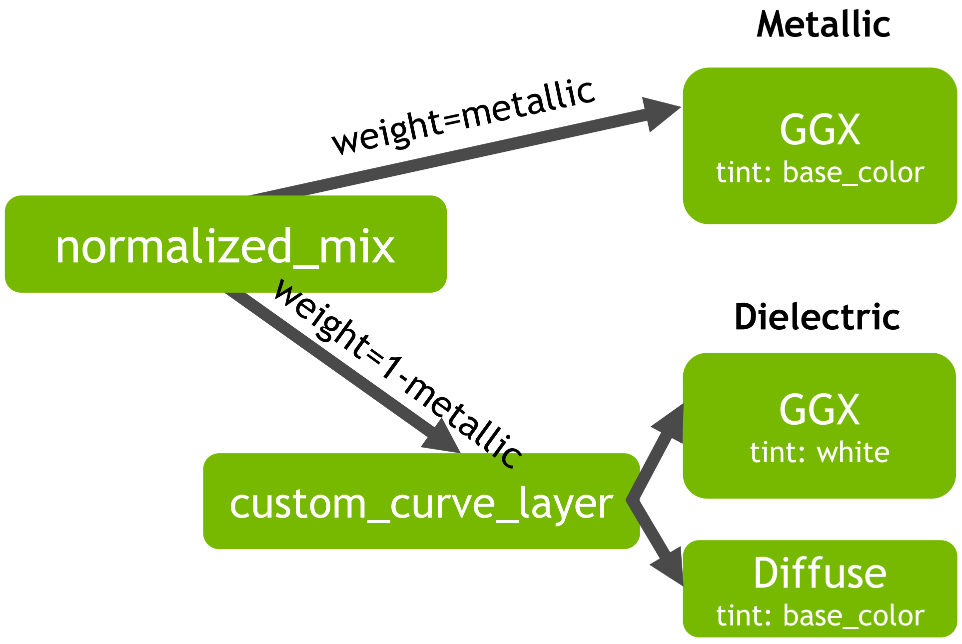
Normalized Mix
Clamped Mix
Weighted Layer

Fresnel Layer
Custom Curve Layer
Measured Curve Layer

EXAMPLE: DIFFUSE + GGX MDL



EXAMPLE: UE4 LIT

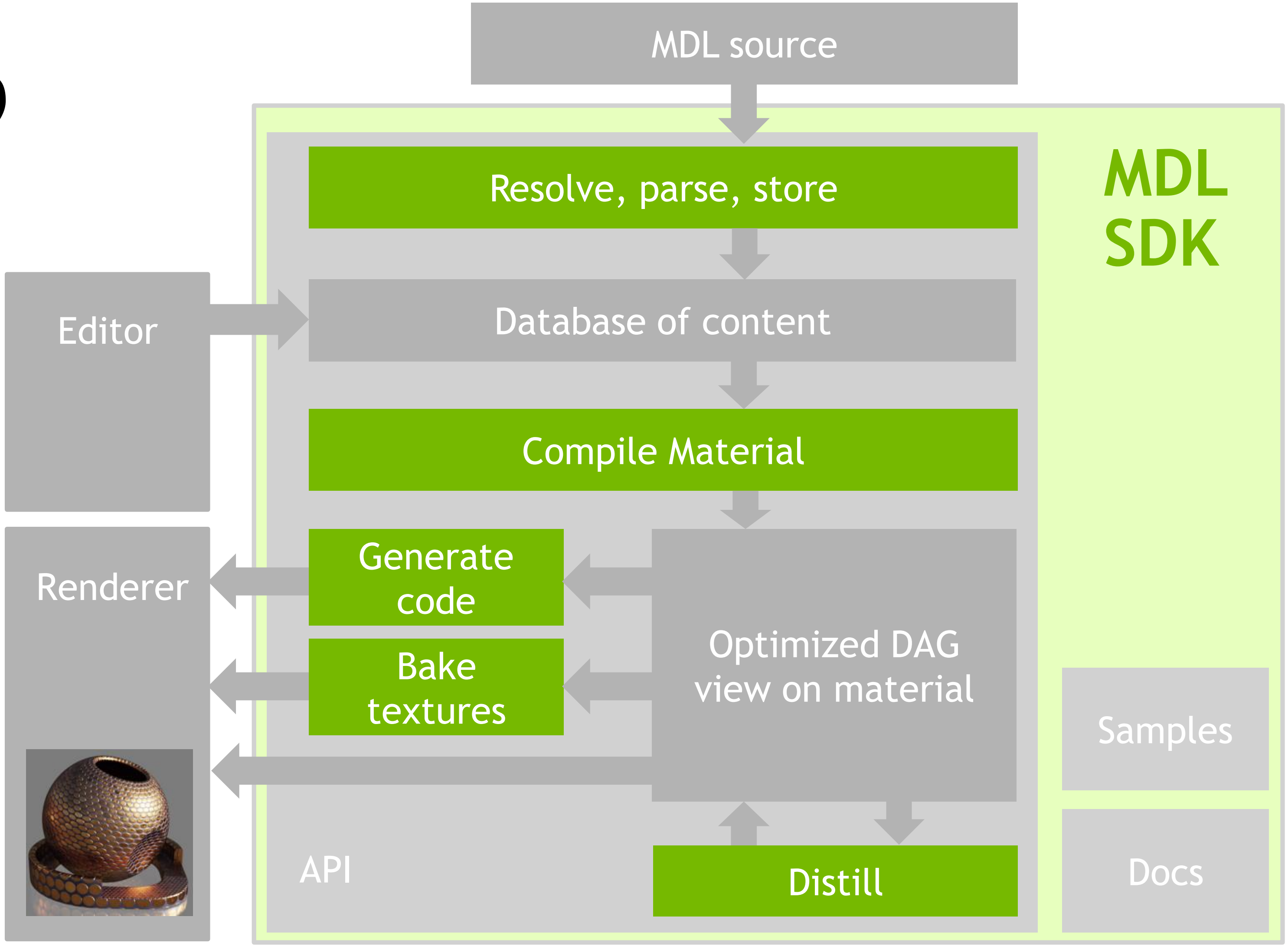



```
export material Lit(
  color base_color = color(0.8, 0.8, 0.8),
  float metallic = 0.0,
  float specular = 0.5,
  float roughness = 0.2,
  color emissive_color = color(0.0, 0.0, 0.0)
  float opacity_mask = 1.0,
  float3 normal = state::normal(),
) = let {

   float alpha = roughness * roughness;
   float grazing_refl = math::max((1.0 - roughness), 0.0);

   bsdf dielectric_component = df::custom_curve_layer(
       weight: specular,
       normal_reflectivity: 0.08,
       grazing_reflectivity: grazing_refl,
       layer: df::microfacet_ggx_vcavities_bsdf(roughness_u: alpha),
       base: df::diffuse_reflection_bsdf(tint: base_color));

   bsdf metallic_component = df::microfacet_ggx_vcavities_bsdf(tint: base_color, roughness_u: alpha);

   bsdf dielectric_metal_mix = df::normalized_mix(
           components: df::bsdf_component[](
                   df::bsdf_component(component: metallic_component, weight: metallic),
                   df::bsdf_component(component: dielectric_component, weight: 1.0-metallic)
               ));
}
in material(
   surface: material_surface(
       scattering: dielectric_metal_mix,
       emission: material_emission (
                emission:  df::diffuse_edf (),
                intensity: emissive_color)),

   geometry: material_geometry(
       cutout_opacity: opacity_mask,
       normal: normal));
```

USING MDL IN A MONTE CARLO RENDERER

MDL SDK 2019

What you get



WORKING WITH A COMPILED MATERIAL

Inspect: Examine graph structure of compiled material

Compile: Use MDL backends to generate target code for

- texturing functions
- distribution functions

Distill: Use Distiller API to

- convert material to a fixed material model like UE4
- bake texturing functions into textures

FROM MATERIAL TO RENDERING CODE

Implementing the declarative part of the material

Actual shading code for material description can be highly renderer specific

- Renderer may analyze declarative part of compiled material instance (in particular all BSDFs)
  - Renderer can implement its own building blocks for all of MDL’s *df* module
  - Renderer needs to wire up BSDF hierarchy and parameters within its own data structures
  - Renderer can “interpret” that at runtime
- Or we just let the MDL SDK create code for the BSDFs

MDL-GENERATED CODE FOR SURFACE BSDFS

Essential blocks for a physically based Monte Carlo renderer

bsdf_init:	Shared initialization for the current shading point
bsdf_evaluate:	Evaluation of the BSDF for a given outgoing and incoming direction
bsdf_sample:	Importance sampling of an incoming for a given outgoing direction
bsdf_pdf:	Probability density computation of that importance sampling
edf_eval:	Evaluation of the EDF for a given outgoing direction

CALLING MDL-GENERATED CODE

Contract 1: Renderer to MDL Shader Code Interface

```
void bsdf_init(Shading_state_material state, inout packed_tex_results p);
```

Stores 'texturing function' results in 'p'. Reuse in _sample/eval/pdf

```
void bsdf_sample(Shading_state_material state, inout Packed_tex_results res,
                 inout uint seed, in float3 V, inout float3 L,
                 inout float3 bsdfOverPdf, inout float pdf);

float3 bsdf_eval(Shading_state_material state, inout Packed_tex_results res,
                 in float3 V, in float3 L);

float bsdf_pdf(Shading_state_material state, inout Packed_tex_results res,
               in float3 V, in float3 L /* direction to light */ );
```

EXECUTING CODE GENERATED BY MDL SDK

Contract 1: Renderer to MDL - Renderer-Provided Shading State

```
struct Shading_state_material {
    float3      normal;                  // state::normal()
    float3      geom_normal;             // state::geom_normal()
    float3      position;                // state::position()
    float       animation_time;          // state::animation_time()
    float3      text_coords[N];          // state::texture_coordinate() table
    float3      tangent_u[N];            // state::texture_tangent_u() table
    float3      tangent_v[N];            // state::texture_tangent_v() table

    float4x4    world_to_object;         // world-to-object transform matrix
    float4x4    object_to_world;         // object-to-world transform matrix
    uint        object_id;               // state::object_id()

    uint        arg_block_offset;        // offset to arguments in user buffer
    uint        ro_data_segment_offset;  // offset to read-only data in user buffer
};
```

EXECUTING CODE GENERATED BY MDL SDK

Contract 2: **MDL to Renderer Interface.** Texture and Parameter Access

```
float mdl_read_argblock_as_float(uint offs)
{
    return asfloat(gSceneParams.blockBuffer.Load(offs>>2));
}

int mdl_read_argblock_as_int(uint offs)
{
    return asint(gSceneParams.blockBuffer.Load(offs>>2));
}

uint mdl_read_argblock_as_uint(uint offs)
{
    return asuint(gSceneParams.blockBuffer.Load(offs>>2));
}

bool mdl_read_argblock_as_bool(uint offs)
{
    uint val = gSceneParams.blockBuffer.Load(offs>>2);
    return (val & (0xff << (8 * (offs & 3)))) != 0;
}
```

```
float mdl_read_rodata_as_float(uint offs)
{
    return asfloat(gSceneParams.blockBuffer.Load(offs>>2));
}

int mdl_read_rodata_as_int(uint offs)
{
    return asint(gSceneParams.blockBuffer.Load(offs>>2));
}

uint mdl_read_rodata_as_uint(uint offs)
{
    return gSceneParams.blockBuffer.Load(offs>>2);
}

bool mdl_read_rodata_as_bool(uint offs)
{
    uint val = gSceneParams.blockBuffer.Load(offs >> 2);
    return (val & (0xff << (8 * (offs & 3)))) != 0;
}
```

EXECUTING CODE GENERATED BY MDL SDK

Contract 2: **MDL to Renderer Interface.** Texture and Parameter Access

```
static uint mdlArgBlockByteOffset;

uint convertMdlTexIndexToInternalIndex(uint tex)
{
    uint textureDescriptorsRangeStart =
      gBlockBuffer.Load(mdlArgBlockByteOffset >>2);

    return  textureDescriptorsRangeStart + tex - 1;
}

bool tex_isvalid(uint tex)
{
    return tex != 0;
}
```

```
uint tex_width_2d(uint tex, int2 uvTile)
{
    const uint texIdx = convertMdlTexIndexToInternalIndex(tex);
    return getTextureWidth(texIdx);
}

uint tex_height_2d(uint tex, int2 uvTile)
{
    const uint texIdx = convertMdlTexIndexToInternalIndex(tex);
    return getTextureHeight(texIdx);
}

float4 tex_lookup_float4_2d(uint tex, float2 coord,
                            int wrapU, int wrapV,
                            float2 cropU, float2
cropV)
{
    const uint texIdx = convertMdlTexIndexToInternalIndex(tex);

    const int samplerIndex = getSamplerIndex(wrapU, wrapV);

    return textures[texIdx].SampleLevel(mdlSamplers[samplerIndex], coord, 0);
}
```

A SIMPLE PATH TRACER WITH MDL

With just BSDF scattering and Emission. No light sampling

Simplest unidirectional path tracer just needs **bsdf_init, bsdf_sample** and **edf_eval**
MDL SDK generates all the shader code necessary for these functions.

```
for every pixel: float3 color = 0;
  for every sample:
   float3 throughput = 1.0f; float3 radiance = 0;
   float3 L = generatePrimaryRay(pixel);
   for every bounce:
     Shading_state_material state; Packed_tex_results texRes;
     TraceRay(L, … , /*out*/state);
     bsdf_init(state, /*out*/ texRes);
     bsdf_sample(state, texRes, seed, V, /*out*/L, /*out*/bsdfOverPdf, /*out*/pdf);
     float3 emission = edf_eval(state);
     radiance += throughput * emission;
     throughput *= bsdfOverPdf;
     if (pdf == 0) break;
   color += radiance;
  color /= sampleCount;
```


Persp
NC2080
NC2080

USING MDL IN A REAL-TIME RAY TRACING RENDERER

MDL FOR REAL-TIME RAY TRACING

VS. MDL FOR MONTE CARLO RENDERING

Real-time Ray Tracing has different needs:

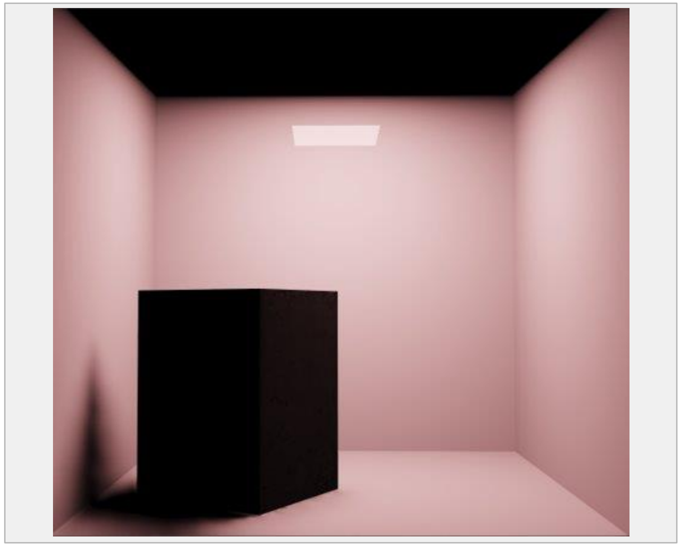
Cannot generate 100s or 1000s of samples per pixel per frame

Instead:

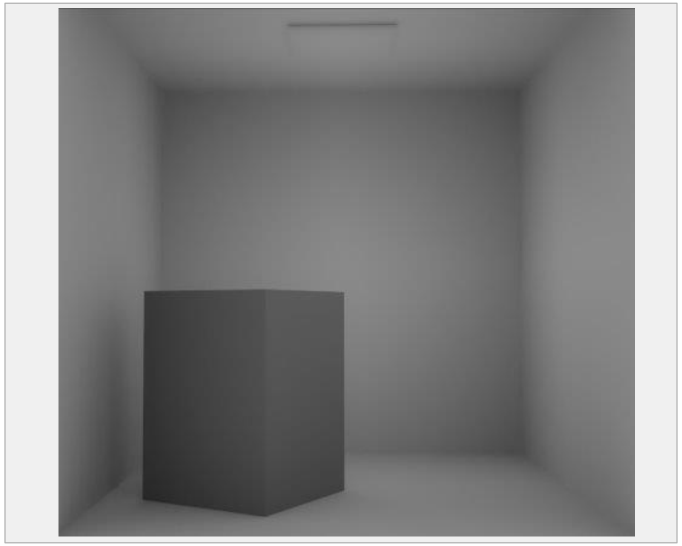
- Split light paths into segments and contribution types,
- Generate buffers with samples for these
- Denoise them and combine them into a final image

REAL-TIME RAY TRACING

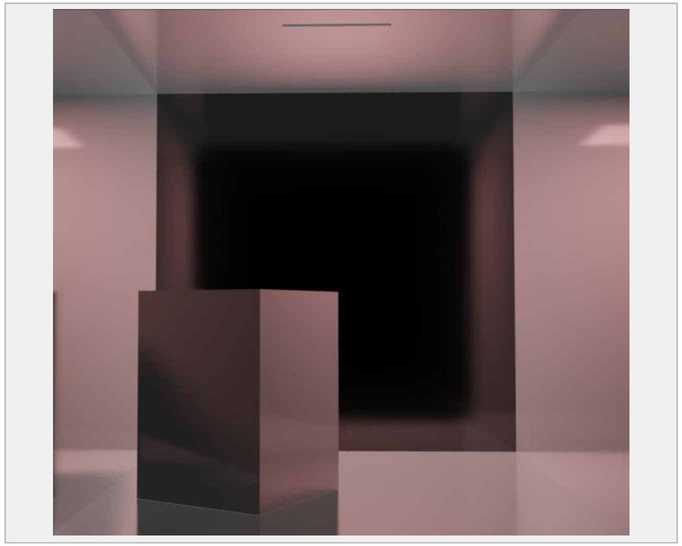
Splitting Light Paths



Direct Lighting from Analytical Lights



Indirect Diffuse / Ambient Occlusion



Indirect Specular: Reflections

REAL-TIME RAY TRACING

Direct Lighting from Analytical Lights

- For single sample:

$$\left(\int_\Omega d\omega_i \, L(\omega_i) \cos(\theta_i) \, f_{BRDF}\right) \cdot Denoise\left[L(\omega_j)\right]$$

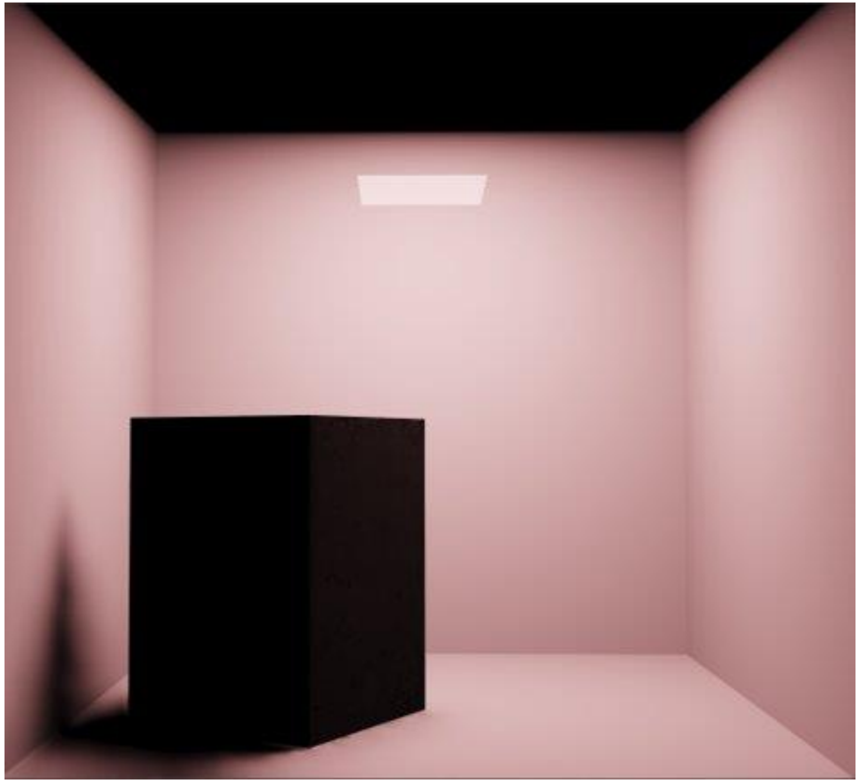
Linear Transform of Cosines approximation (LTC)

(Heitz et al., 2016), Real Shading in UE4 (Karis 2013)

- Generalization for multiple samples:

$$\left(\int_\Omega d\omega_i \, L(\omega_i) \cos(\theta_i) \, f_{BRDF}\right) \cdot \frac{Denoise\left[\sum_j \frac{V(\omega_j) L(\omega_j) \cos(\theta_j) f_{BRDF}}{f_{\Omega_i}}\right]}{Denoise\left[\sum_j \frac{L(\omega_j) \cos(\theta_j) f_{BRDF}}{f_{\Omega_i}}\right]}$$

Combining Analytic Direct Illumination and Stochastic Shadows (Heitz et al., 2018)
https://research.nvidia.com/publication/2018-05_Combining-Analytic-Direct



REAL-TIME RAY TRACING

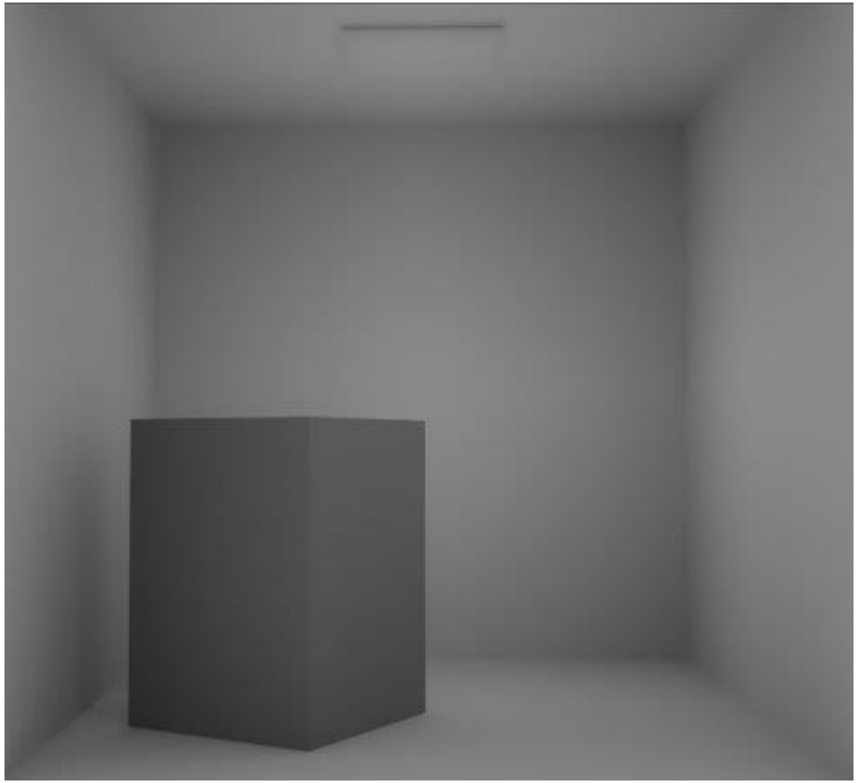
Indirect Diffuse and Ambient Occlusion

- **Ambient Occlusion**

'Diffuse reflectance' * 'Denoised AO ray visibility '
(short rays sampling hemisphere about normal)

- **Indirect Diffuse GI**

'Diffuse reflectance' * 'Denoised irradiance'



REAL-TIME RAY TRACING

Indirect Specular: Reflection / Refraction

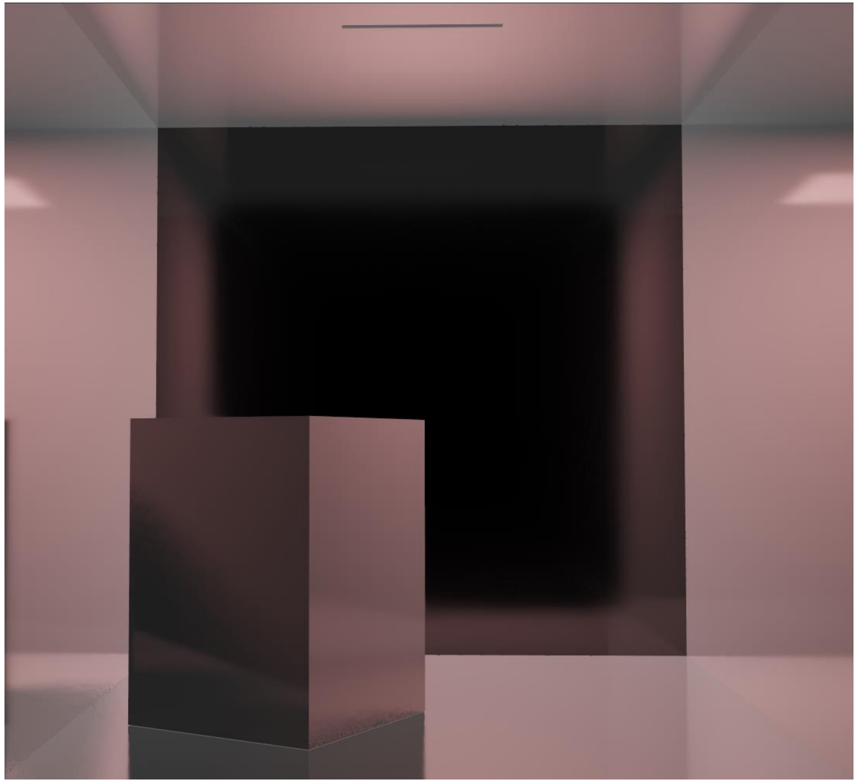
- **Indirect Specular: Reflections**

'Pre-integrated BSDF' * 'Denoised incoming radiance'

Getting More Physical in Call of Duty: Black Ops II (Lazarov 2013)

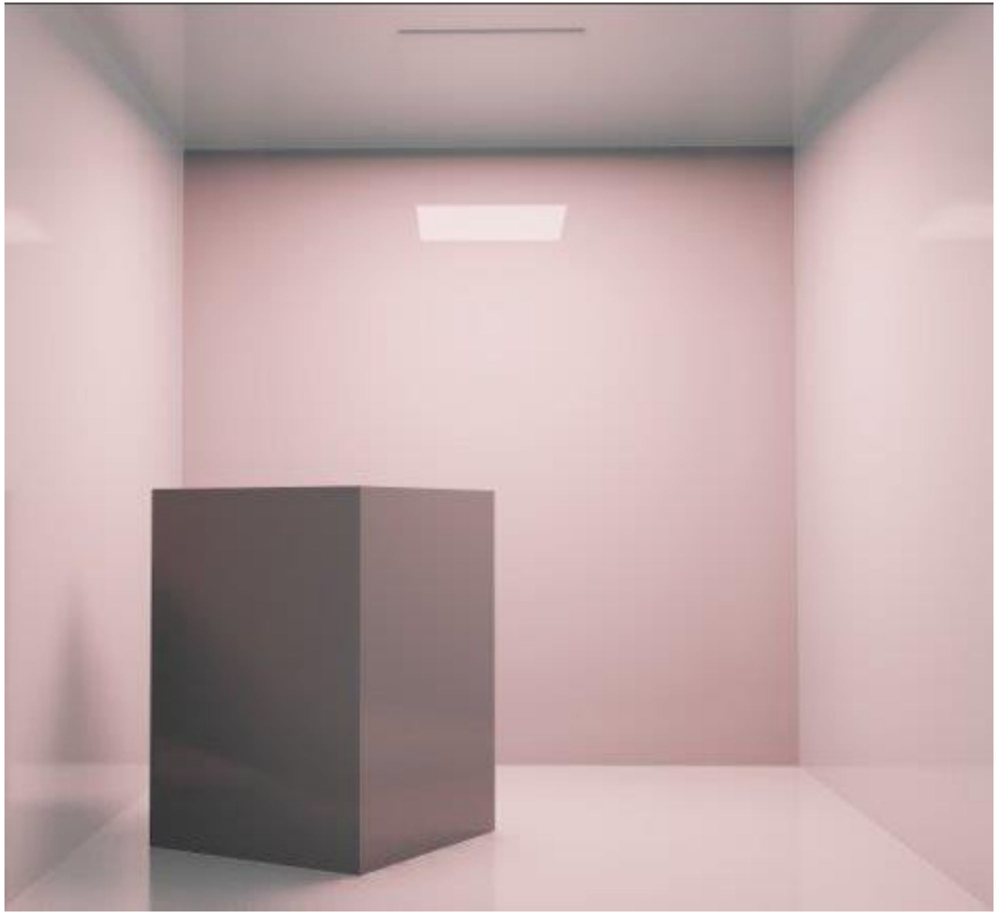
This approximation is for isotropic GGX only.
Generalizing to arbitrary BSDFs harder. Open issue.

- **Indirect Specular: Smooth Translucency**



REAL-TIME RAY TRACING

Combined Denoised Light Path Segments



REAL-TIME RAY TRACING WITH MDL

How do we do it?

Recall a few slides earlier:

"Actual shading code [...] can be highly renderer specific"

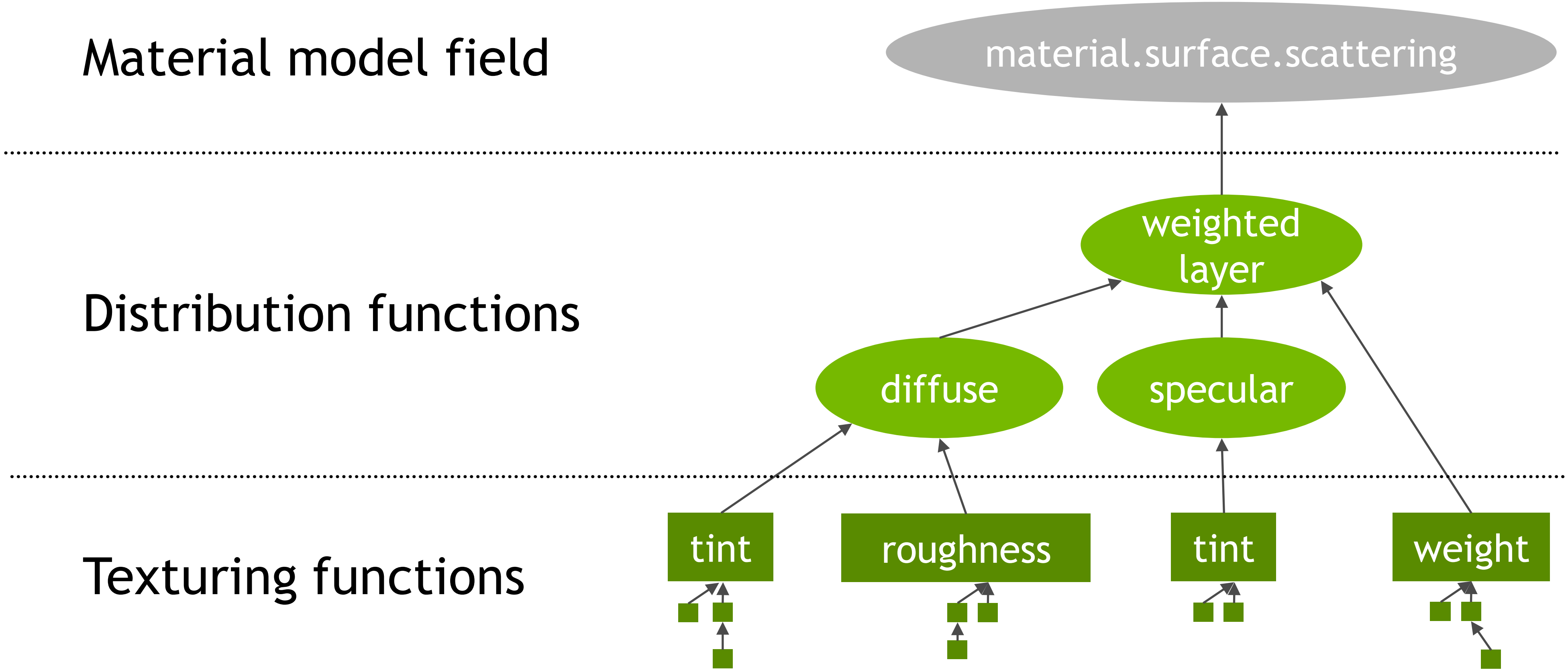
"A renderer may analyze the declarative part of the compiled material instance"

"Renderer can implement its own building blocks for all of MDL’s *df* module"

So... that's what we do.

WORKING WITH A COMPILED MATERIAL

Graph of compiled material



REAL-TIME RAY TRACING WITH MDL

Custom Code Generation

Partial port of MDL SDK libbsdf.cpp to HLSL

Map all glossy/microfacet BSDFs to GGX initially

Generate per-light-type BSDF analytic integral evaluation.

> **Using LTC approximation** (Heitz et al. 2016)
>
> Sphere, Rectangle, Disk, Line, Distant Light with Cone Angle (using virtual sphere)

Generate custom HLSL functions, similar to MDL SDK Distiller:

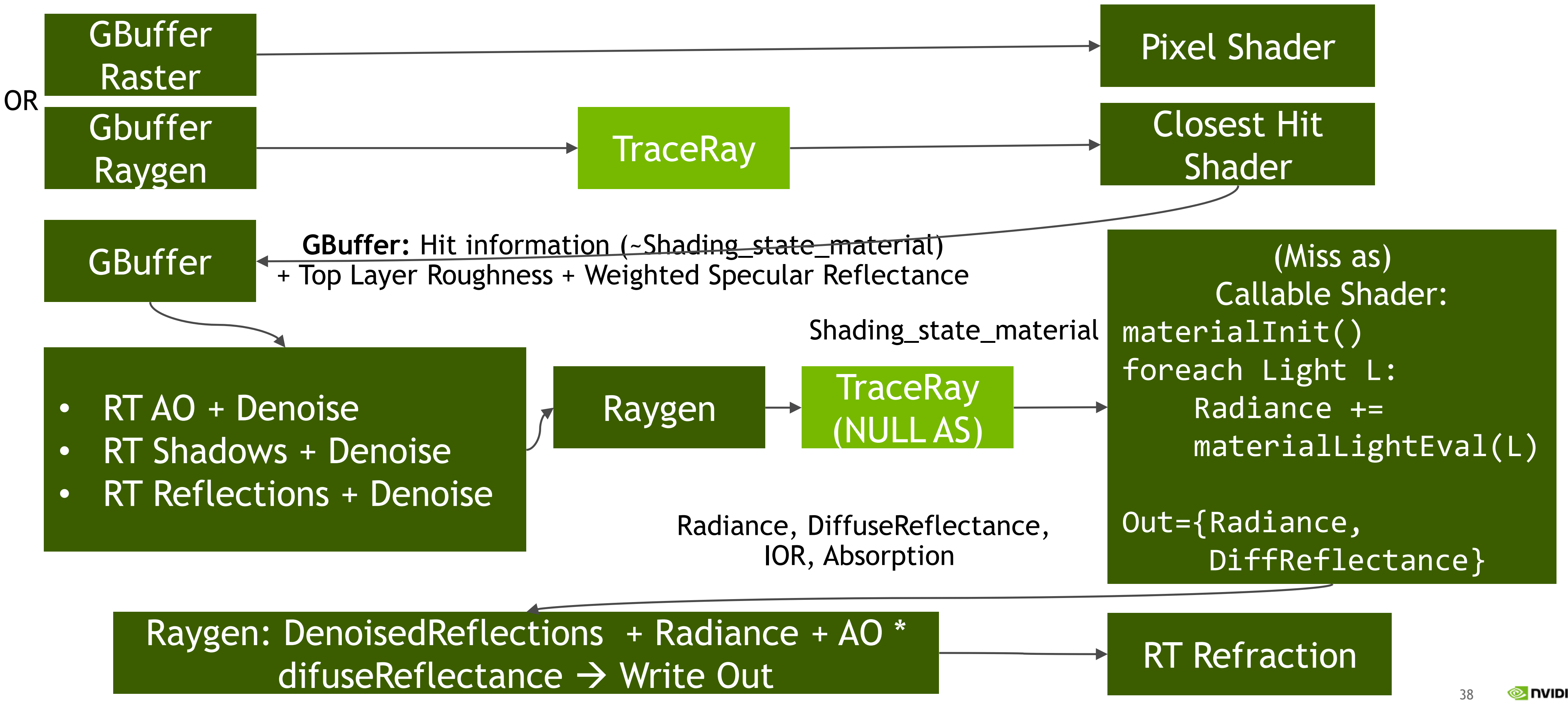
- Weighted Diffuse Tint (reflectance) for all diffuse layers
- Weighted Specular Reflectance for all specular/glossy layers. Used for reflections
- Roughness for Top N Layers. Used for reflections

WRAPPING THE MDL SDK INTO A SIMPLER API

Our 'MDL Translator' library

```
Result addMdlSearchPaths(const char* mdlPaths[], size_t numPaths);

Module* createModule(const char* file, CompilationMode compileMode);
void destroyModule(Module* module);

Material* createMaterial(const Module* module, const char* materialName);
void destroyMaterial(Material* material);
MaterialOpacity getMaterialOpacity(const Material* material);
bool getMaterialCutOutOpacityIsConstant(const Material* material);

const ShaderCode* generateShaderCode(Material* material);
const char* getShaderCode(const ShaderCode* materialCode);

size_t getReadOnlyBlockSize(const ShaderCode* shaderCode);
const void* getReadOnlyBlockData(const ShaderCode* shaderCode);

size_t getParamBufferSize(const ShaderCode* shaderCode);
const char* getParamsBuffer(const ShaderCode* shaderCode);
const carb::renderer::MaterialParam* getParamDescArray(const ShaderCode* shaderCode, uint32_t* count);

size_t getTextureCount(const ShaderCode* shaderCode);
const char* getTextureName(const ShaderCode* shaderCode, size_t index);
TextureShape getTextureShape(const ShaderCode* shaderCode, size_t index);
TextureGammaMode getTextureGammaMode(const ShaderCode* shaderCode, size_t index);
```

RENDERING DATA FLOW



SECONDARY BOUNCES

Transparency

- In current implementation we handled only smooth materials for this interaction
  - This includes glass, plastics and water
  - Also handled thin walled (two-sided, volume-less) surfaces
- Use new MDL SDK API (C++) to query whether the material is transparent
- Generated functions to get IOR and VDF absorption
- Both events, refraction and reflection, are handled. Since the material is smooth the energy ratios are governed by Fresnel equations, and ray directions by Snell's law
- Non zero roughness means we'll have to sample distributions for directions, and rely on filtering to clean up the result
  - For MC integration we need to sample a direction from a pdf of our choice, this requires distilling a roughness value

SECONDARY BOUNCES

Reflections

- Distill roughness value and sample the microfacet pdf
- Generate reflection ray and trace it
- Afterwards use the callable shader to get radiance value, store it in a buffer for denoising
- Reflection denoising:

$$\left(\int_\Omega d\omega_i \cos\theta_i \cdot f_{BRDF}\right) \cdot \frac{Denoise\left[\sum_j \frac{L_j \cos(\theta_j)\, f_{BRDF}}{f_{\Omega_i}}\right]}{Denoise\left[\sum_j \frac{\cos(\theta_j)\, f_{BRDF}}{f_{\Omega_i}}\right]}$$

- In our implementation we did the denoising after the division, this effectively cancels out all terms, save for radiance, when a single sample is used
- We used GGX preintegrated BRDF. In theory any BRDF can be pre-integrated, but the result can be multi dimensional, as pre-integration removes only the incoming light direction

THANKS AND ACKNOWLEDGEMENTS

Ardavan Kanani: MDL Integration / MDL Translator library, LTC implementation

MDL Team: Lutz Kettner, Jan Jordan, Moritz Kroll, Michael Beck, Sandra Pappenguth

NVIDIA Real-Time Rendering Research Team: Slang (Tim Foley), TAA (Marco Salvi), Tonemapping

Omniverse Team

Ray Tracing Technology Team

NVIDIA®

NVIDIA