GPU TECHNOLOGY CONFERENCE

March 17-21, 2019 | Silicon Valley

Integrating the NVIDIA Material Definition Language MDL in Your Application

Lutz Kettner Director, Rendering Software and Material Definition

Sandra Pappenguth
Moritz Kroll
Matthias Raab
Kai Rohmer

PRESENTED BY NVIDIA

Agenda

MDL – a short introduction

MDL SDK overview

Loading and compiling materials

Executing texturing functions

Executing distribution functions

Distilling materials to a fixed target model

MDL – a short introduction

NVIDIA Material Definition Language (MDL)

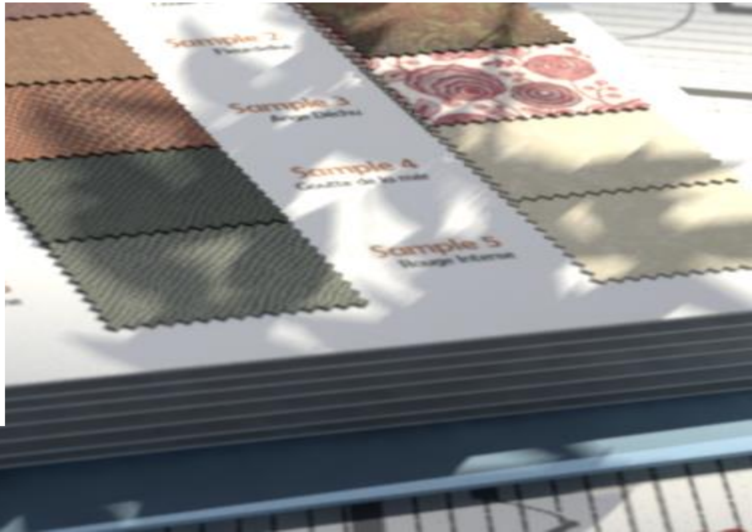
What is this?

Programming language to define **physically-based** materials

A declarative material definition based on a powerful material model

Procedurally programmable functions that compute values for the parameters of the material model

Developed by NVIDIA



MDL

Key Features

Independent of rendering algorithms
-> purely descriptive material model

Powerful set of elemental distribution functions plus modifiers and combiners

Well defined module and package concept

Designed for modern highly-parallel machine architectures

MDL SDK overview

MDL SDK 2019

Overview

C++ Library to enable MDL support in your application

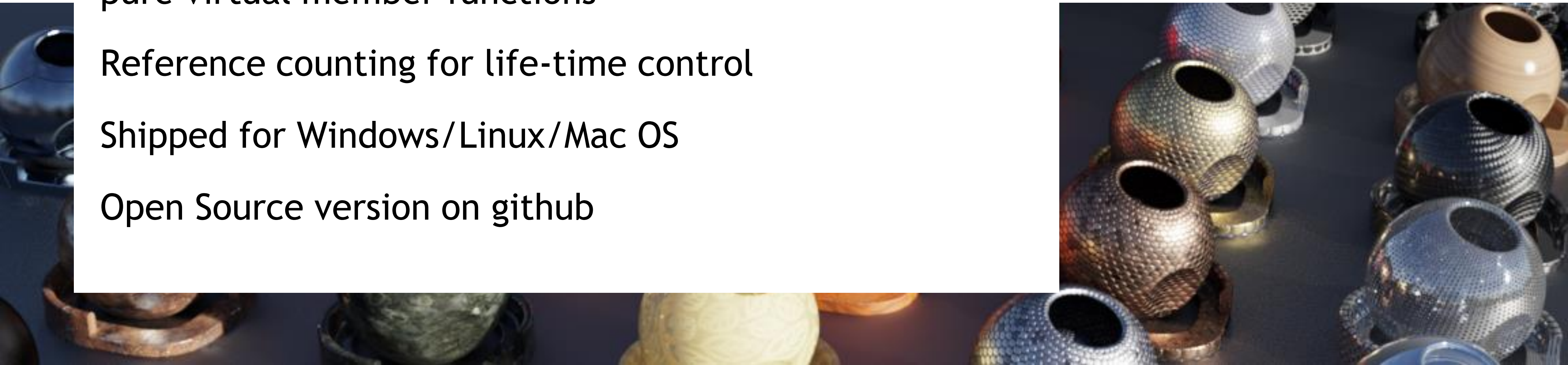
Binary compatibility across shared library boundaries

Access through abstract base classes with
pure virtual member functions

Reference counting for life-time control

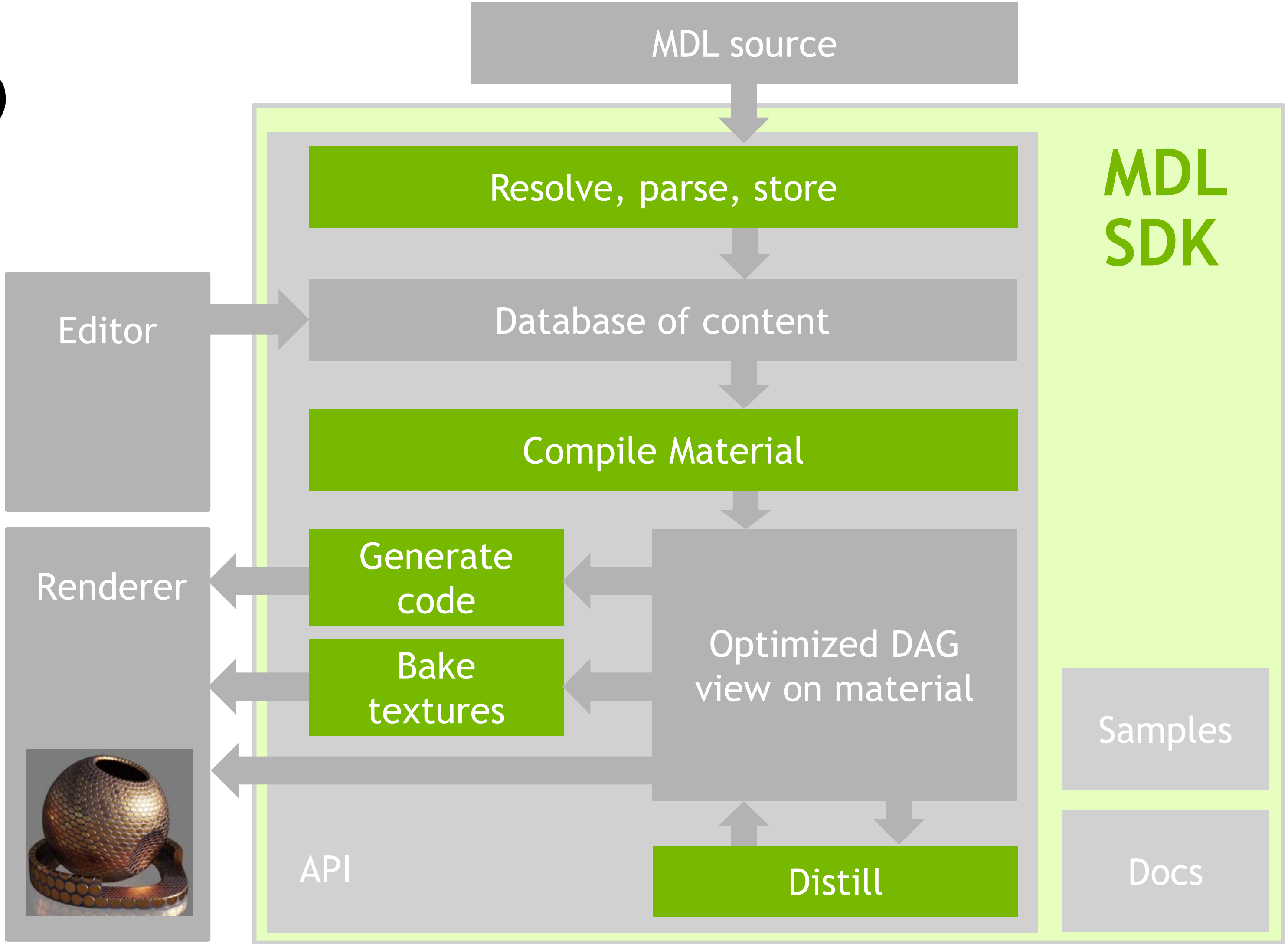
Shipped for Windows/Linux/Mac OS

Open Source version on github



MDL SDK 2019

What you get

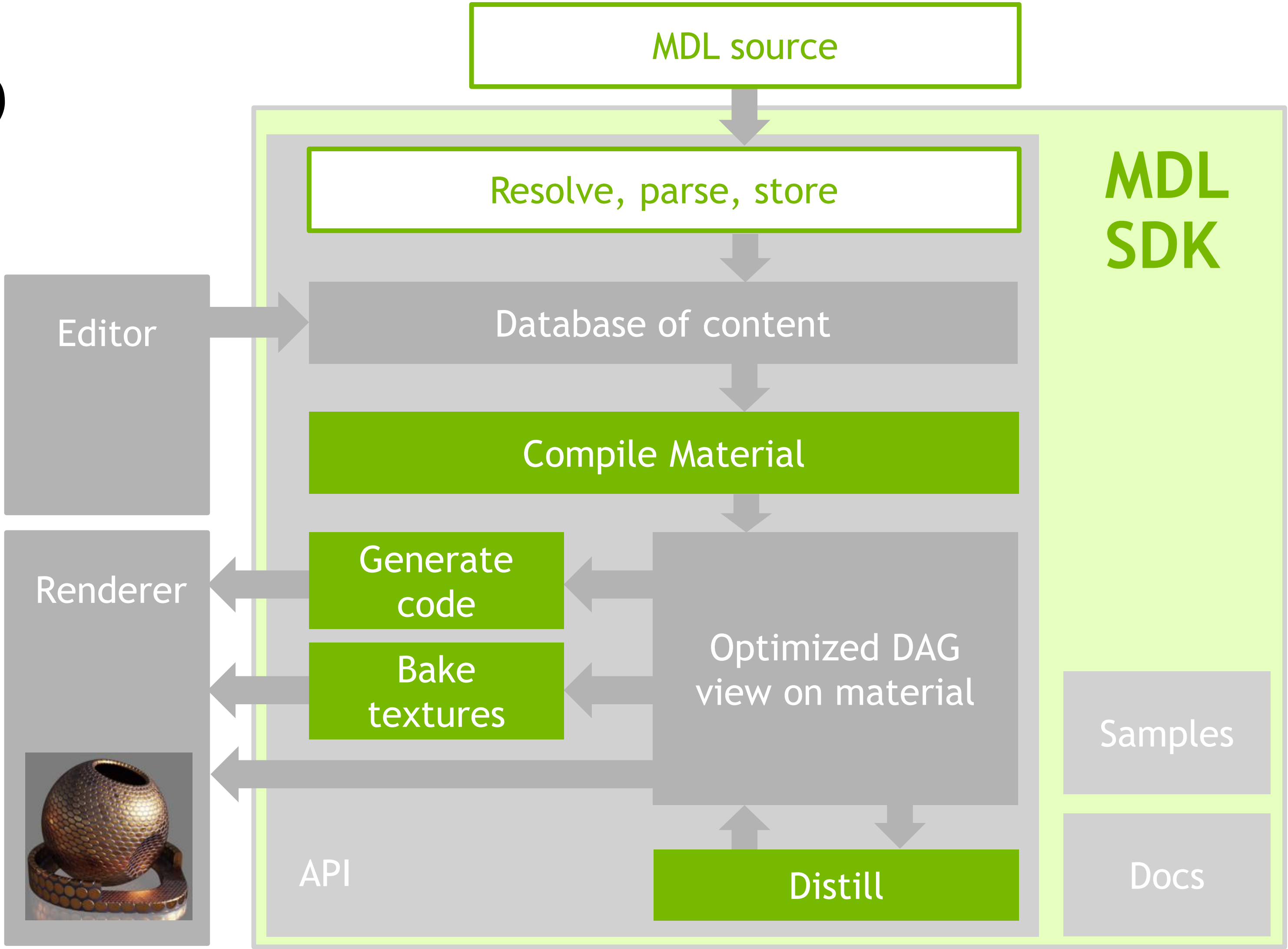


MDL SDK 2019

What you get

Loading of MDL materials

MDL 1.4 support



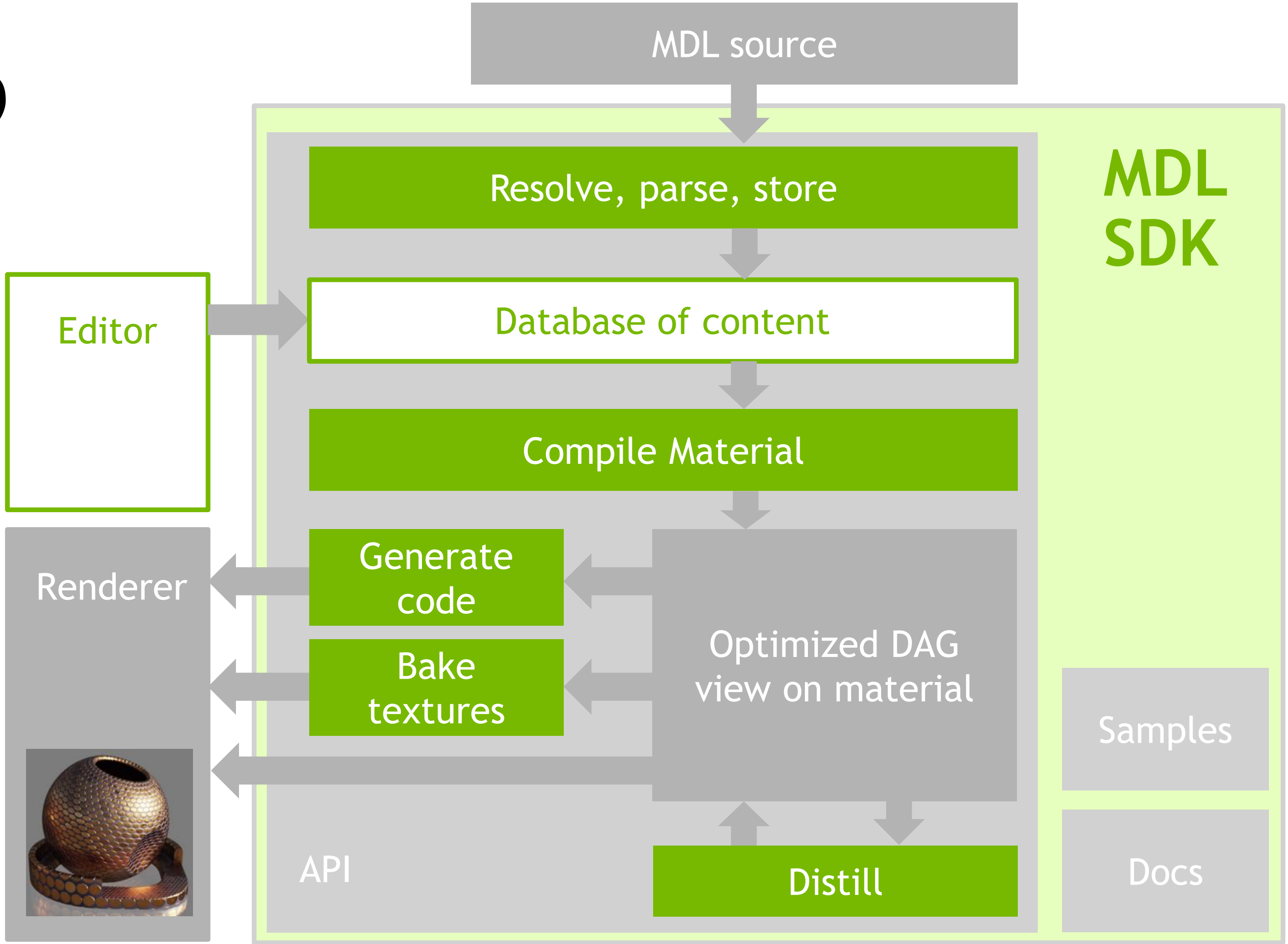
MDL SDK 2019

What you get

DB view on MDL definitions

MDL material/function editing and storage

Via transactions

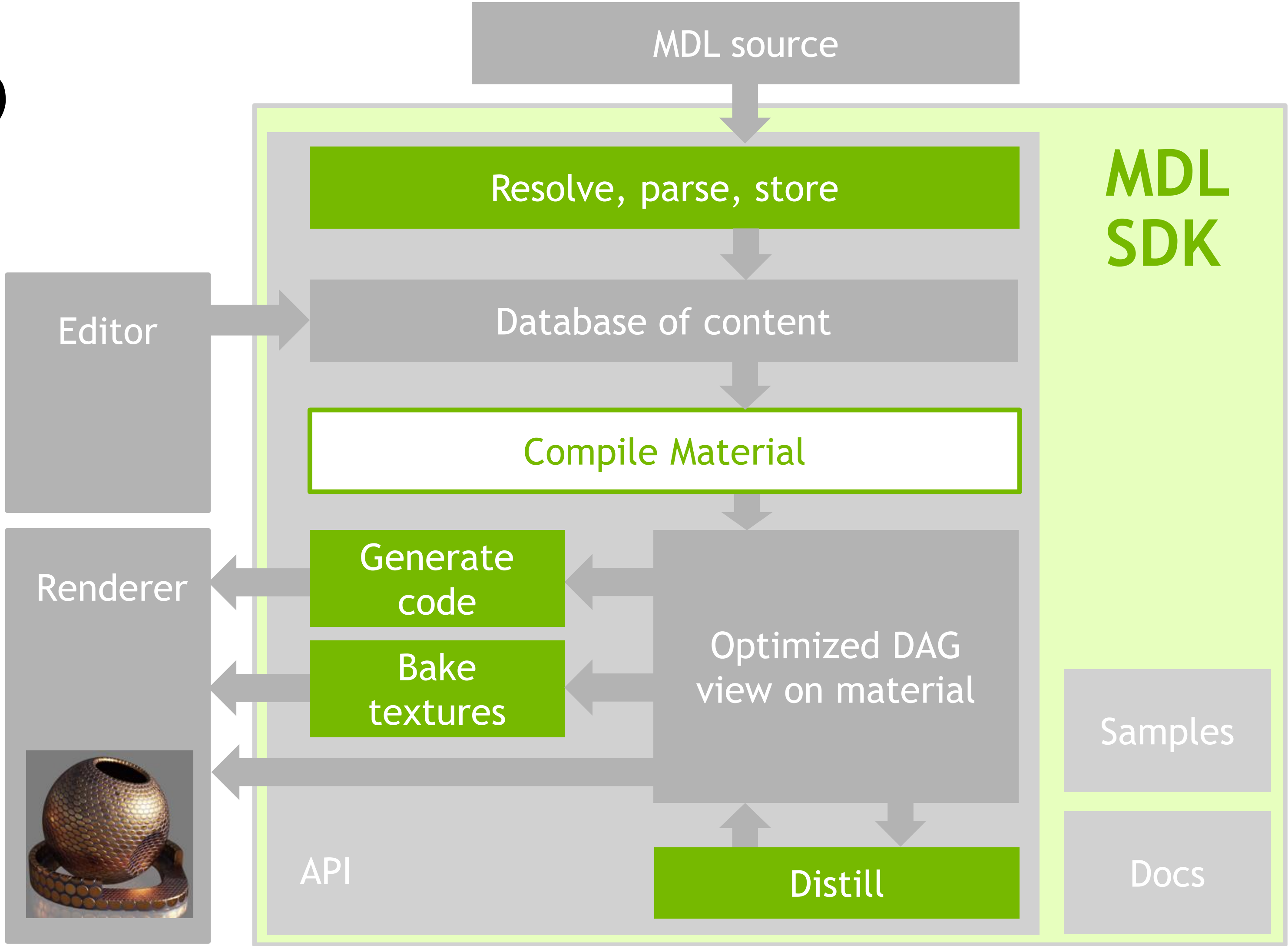


MDL SDK 2019

What you get

MDL 1.4 core compiler

Configuration, data import and export, backend access

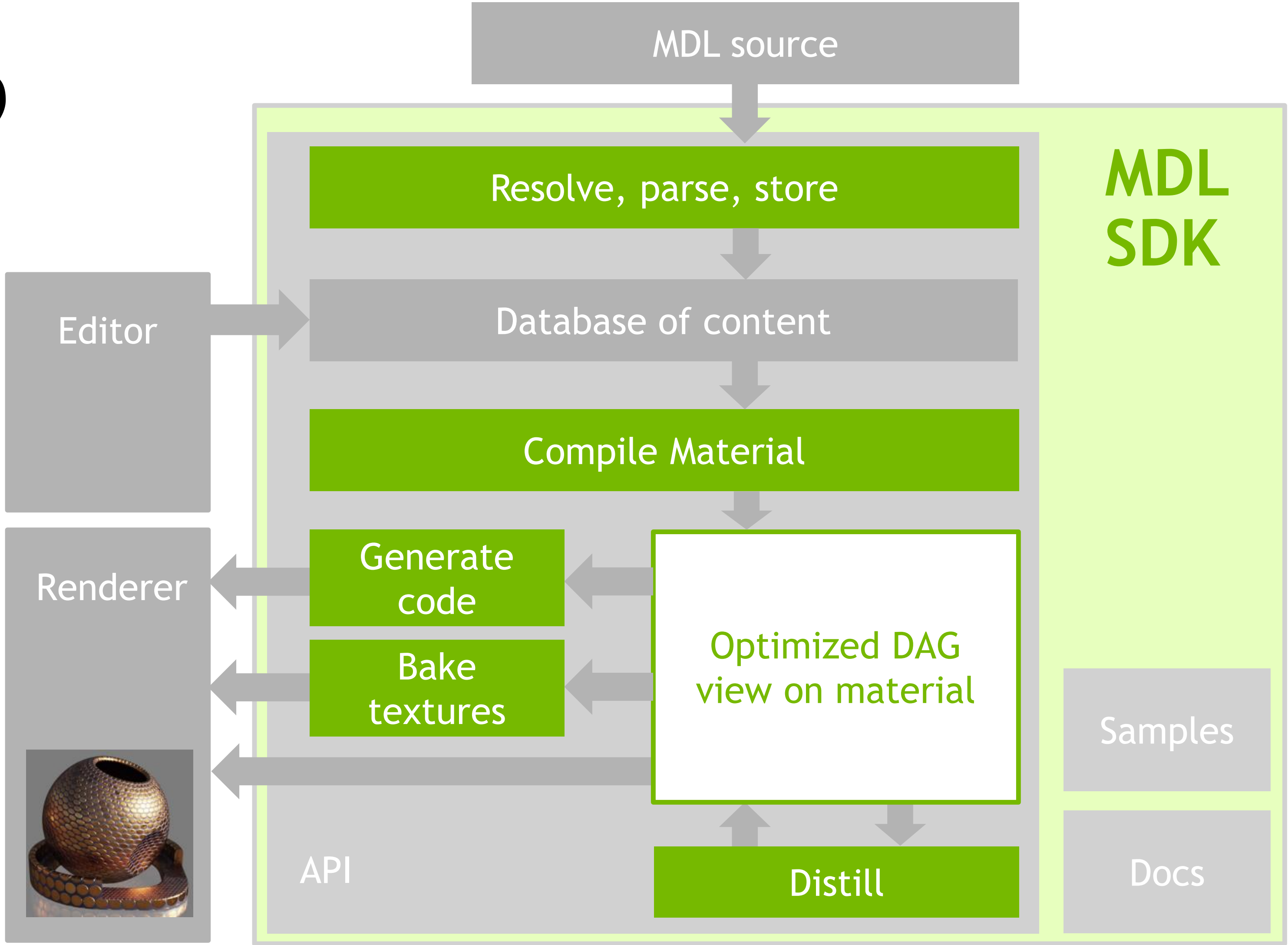


MDL SDK 2019

What you get

Compact, optimized DAG representation of a material instance

2 compilation modes

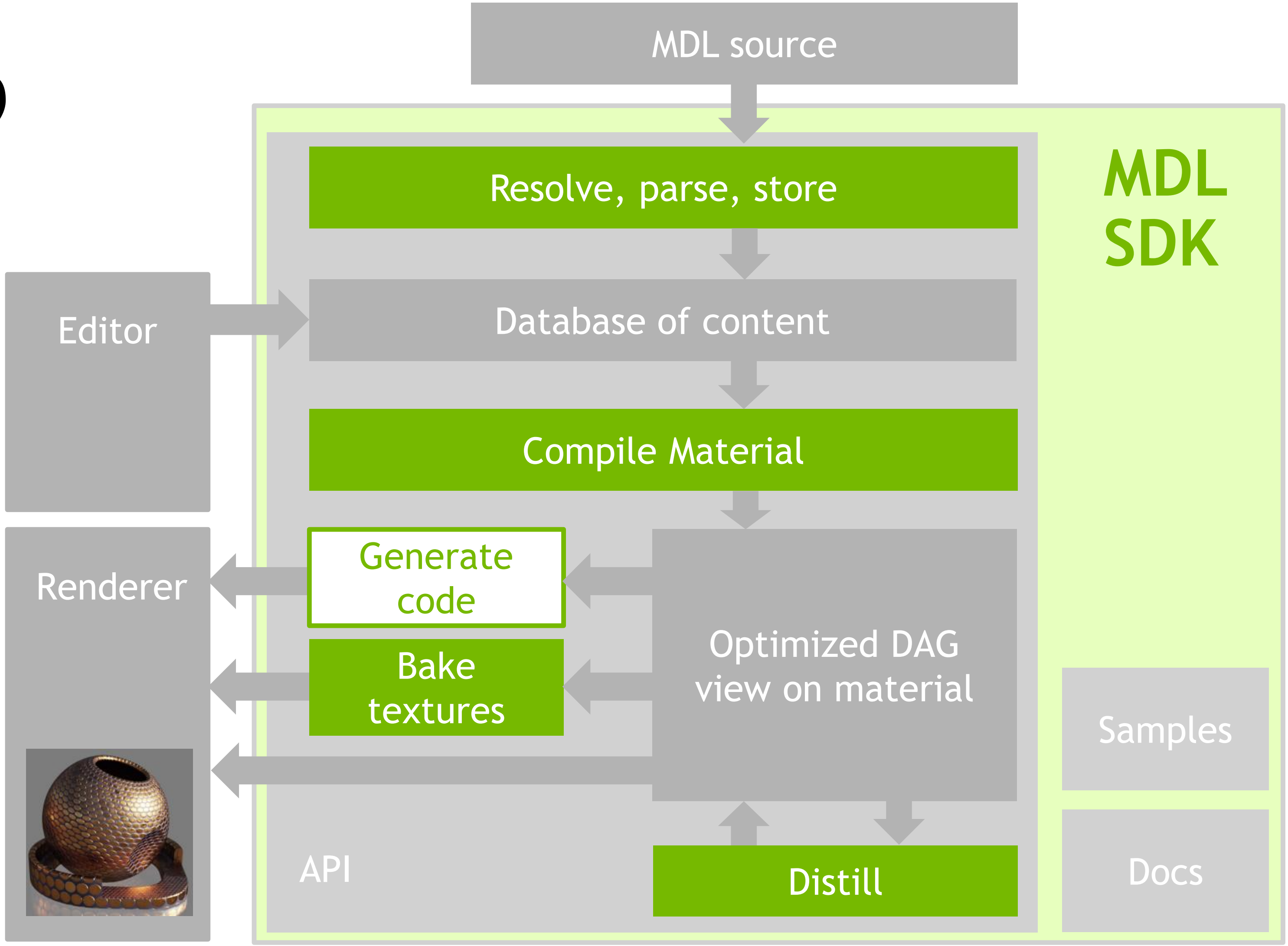


MDL SDK 2019

What you get

Backends for code generation

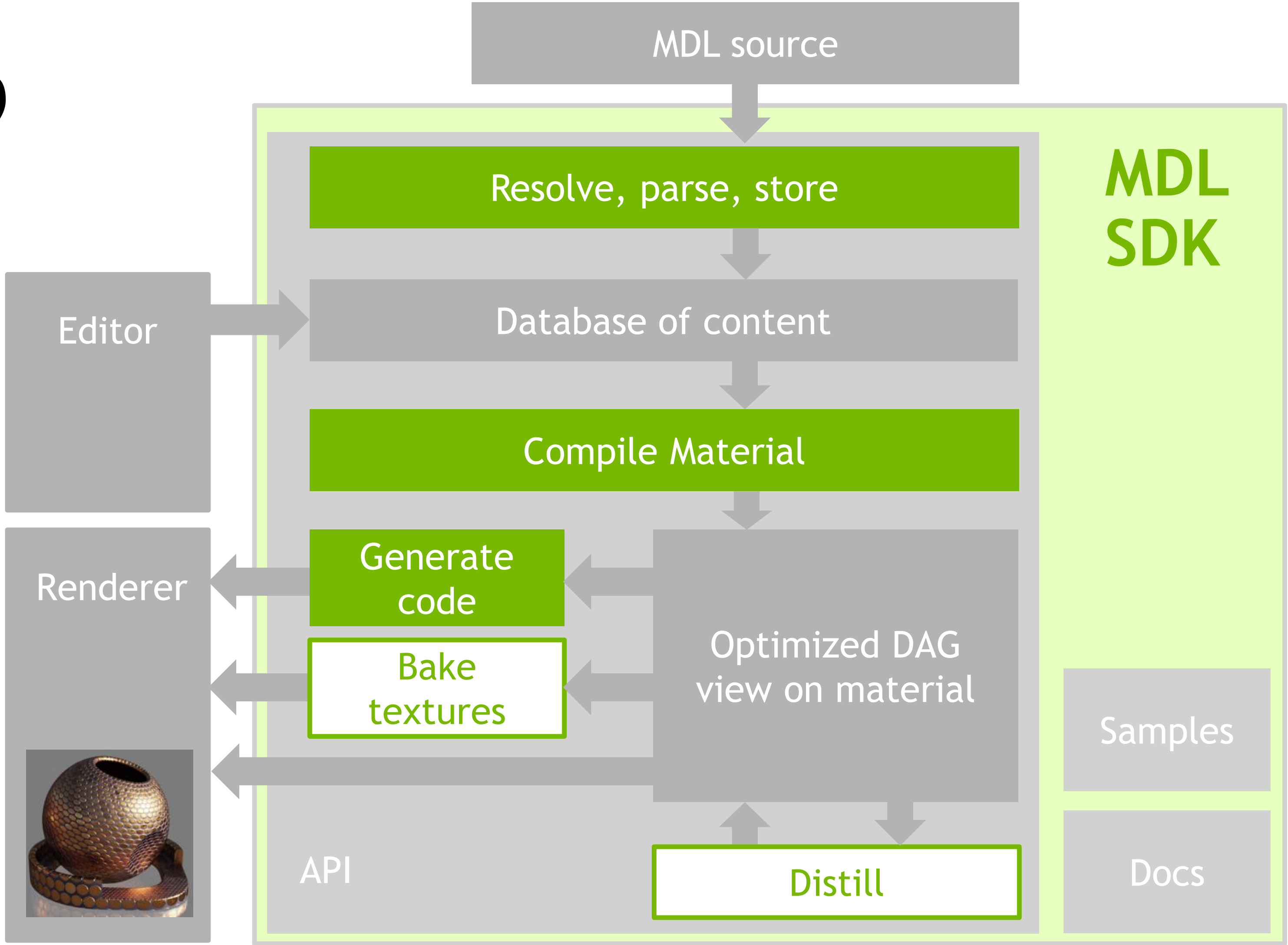
- CUDA PTX
- LLVM IR
- HLSL
- GLSL
- x86-64 CPU



MDL SDK 2019

What you get

Material distilling and baking

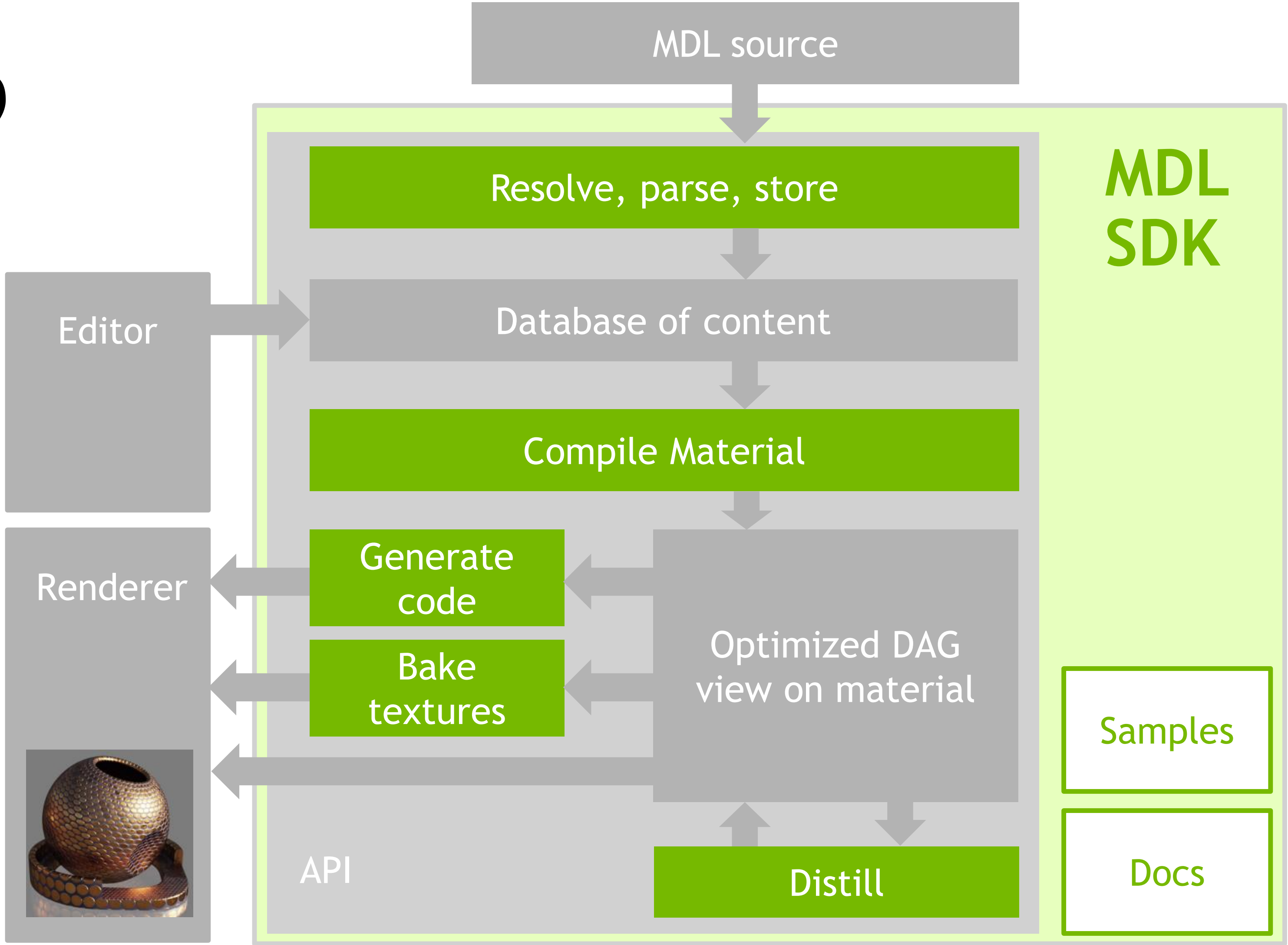


MDL SDK 2019

What you get

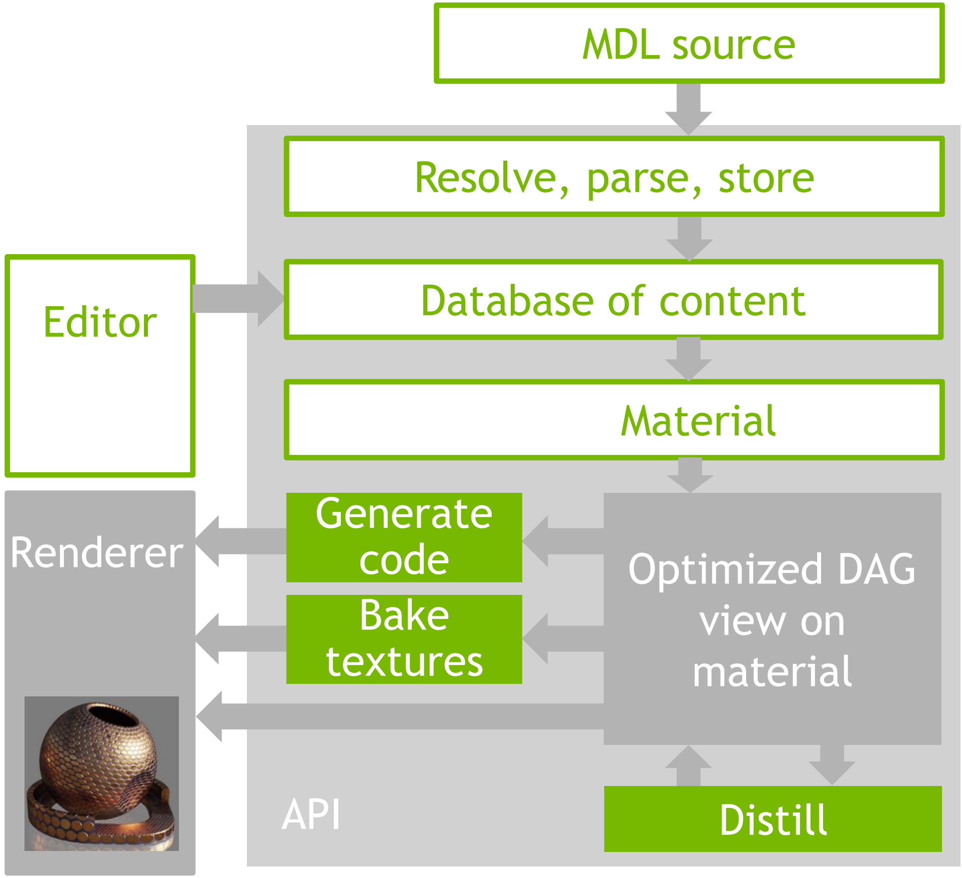
Example Code

Documentation



Loading and compiling materials

Loading and compiling materials



Steps towards a compiled material

Load a module

Load MDL module → Create material instance → Change Arguments → Compile material

```
// access MDL compiler
Handle<IMdl_compiler> mdl_compiler(
    mdl_sdk->get_api_component<IMdl_compiler>());

// load MDL module
mdl_compiler->load_module(transaction, "::example");
```

→ examples/mdl_sdk/modules

Steps towards a compiled material

Create a material instance

Load MDL module → **Create material instance** → Change Arguments → Compile material

```
// get material definition from database
Handle<const IMaterial_definition> material_definition(
    transaction->access<IMaterial_definition>("mdl::example::my_material"));

// instantiate material with default parameters and store in database
Handle<IMaterial_instance> material_instance(
    material_definition->create_material_instance(/*args=*/ nullptr));

// store in database
transaction->store(material_instance.get(), "my_material_instance");
```

→ examples/mdl_sdk/instantiation

Steps towards a compiled material

Edit a material instance

Load MDL module → Create material instance → **Change Arguments** → Compile material

```
// acquire MDL factory
Handle<IMdl_factory> mdl_factory(mdl_sdk->get_api_component<IMdl_factory>());

// create argument editor
Argument_editor arg_editor(transaction, “my_material_instance", mdl_factory);

// change the roughness parameter of the material instance
arg_editor.set_value("roughness", 0.8);

// attach a function call to the "tint" parameter via DB name
arg_editor.set_call("tint", "uv_as_color");
```

→ examples/mdl_sdk/instantiation

Steps towards a compiled material

Compile a material instance

Load MDL module → Create material instance → Change Arguments → **Compile material**

```
// access material instance
Handle<const IMaterial_instance> material_instance(
        transaction->access<const IMaterial_instance> ("my_material_instance");

// compile
Handle<ICompiled_material> compiled_material(
    material_instance->create_compiled_material(
        IMaterial_instance::CLASS_COMPILATION, ...));
```

→ examples/mdl_sdk/compilation

MDL Compilation Modes

Instance Compilation

All arguments are compiled into the resulting material

PRO: Allows for best optimization

CON: Every argument change requires full recompilation

Class Compilation

Many arguments remain parameters of the compiled material

PRO: Fast parameter updates, reuse of generated code for many variants of the same material

CON: Less optimization potential

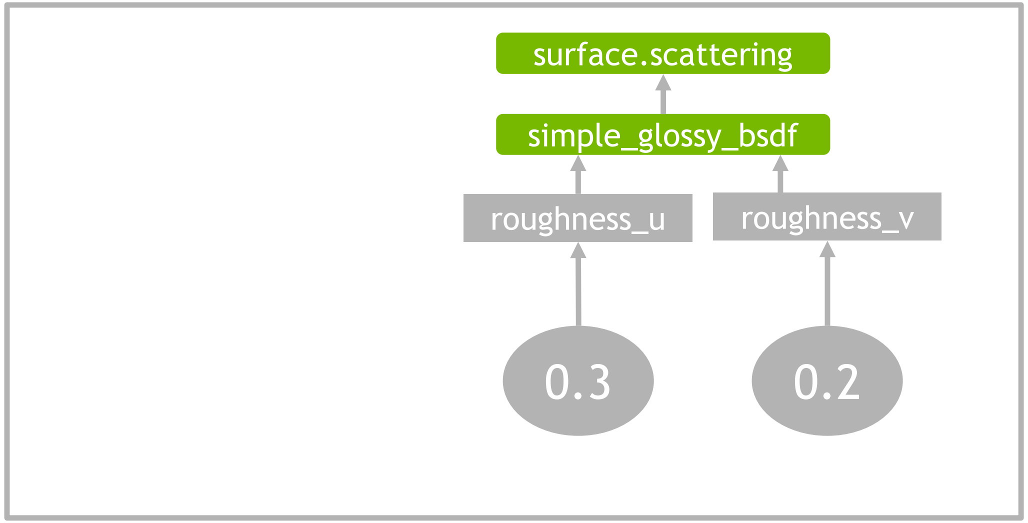
MDL Compilation Modes

```
material glossy(
   float ru: min(
              a: 0.3,
              b: 0.8),
   float rv: 0.2
)
= material(
  surface: material_surface(
    scattering:
      simple_glossy_bsdf(
        roughness_u: ru,
        roughness_v: rv)
  )
);
```

MDL Compilation Modes

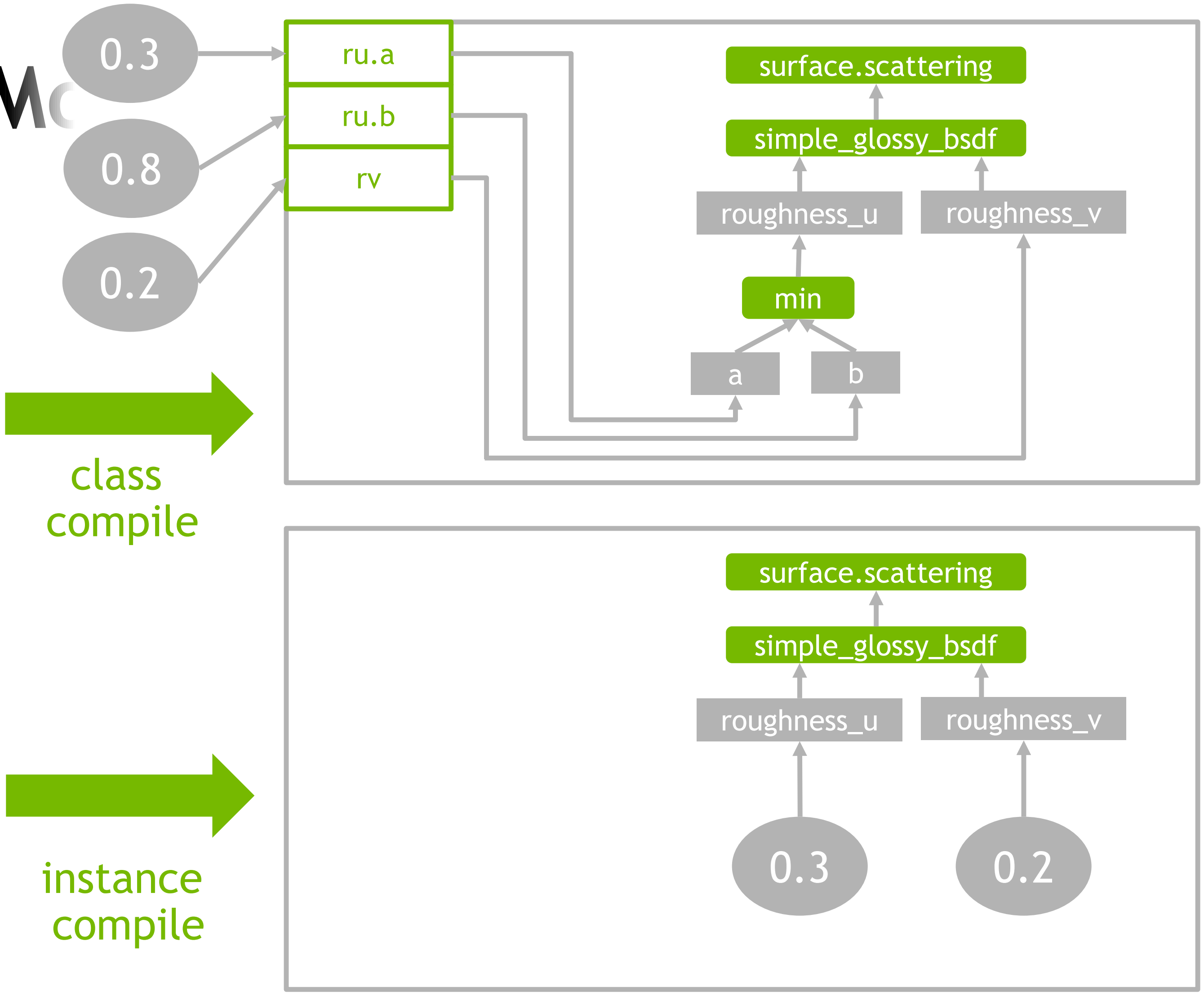
```
material glossy(
   float ru: min(
              a: 0.3,
              b: 0.8),
   float rv: 0.2
)
= material(
  surface: material_surface(
    scattering:
      simple_glossy_bsdf(
        roughness_u: ru,
        roughness_v: rv)
  )
);
```

instance
compile

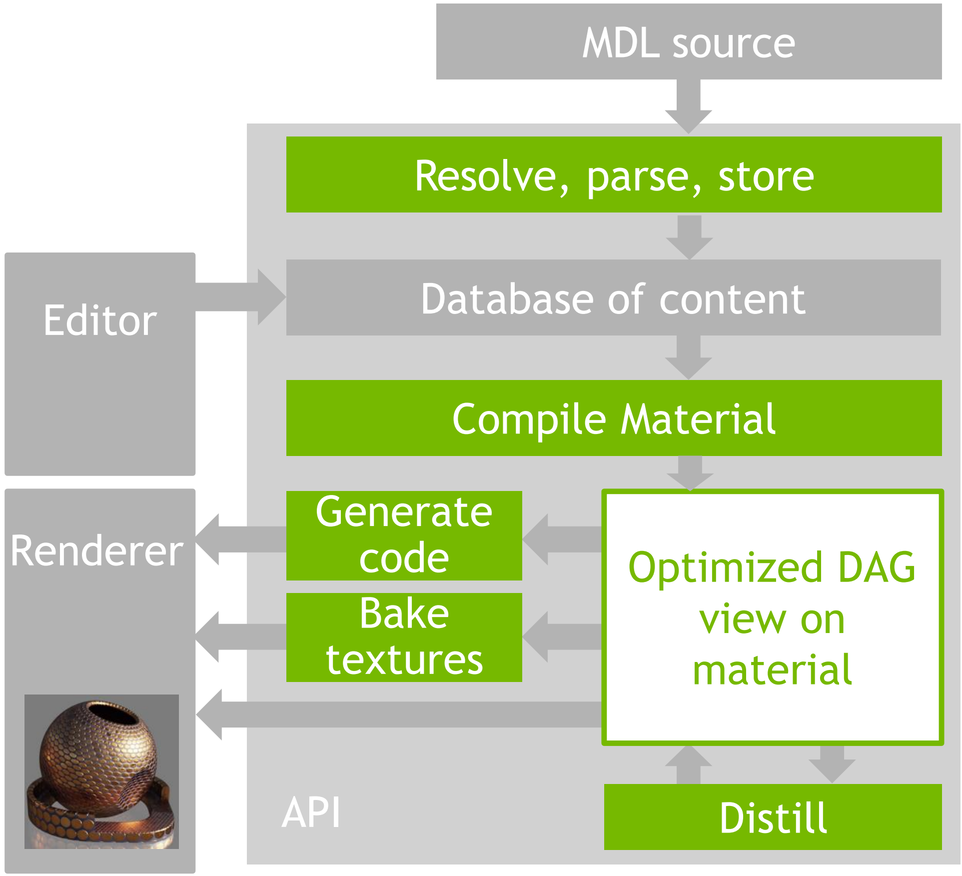


MDL Compilation Mc

```
material glossy(
    float ru: min(
              a: 0.3,
              b: 0.8),
    float rv: 0.2
)
= material(
  surface: material_surface(
    scattering:
      simple_glossy_bsdf(
        roughness_u: ru,
        roughness_v: rv)
  )
);
```



Working with a compiled material



Working with a compiled material

Graph of compiled material

Material model field

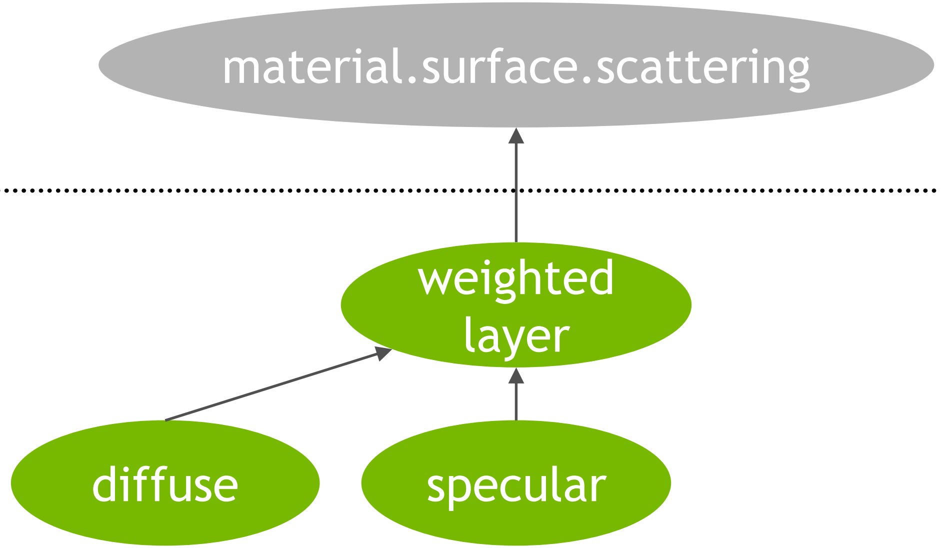
material.surface.scattering

Working with a compiled material

Graph of compiled material

Material model field

Distribution functions



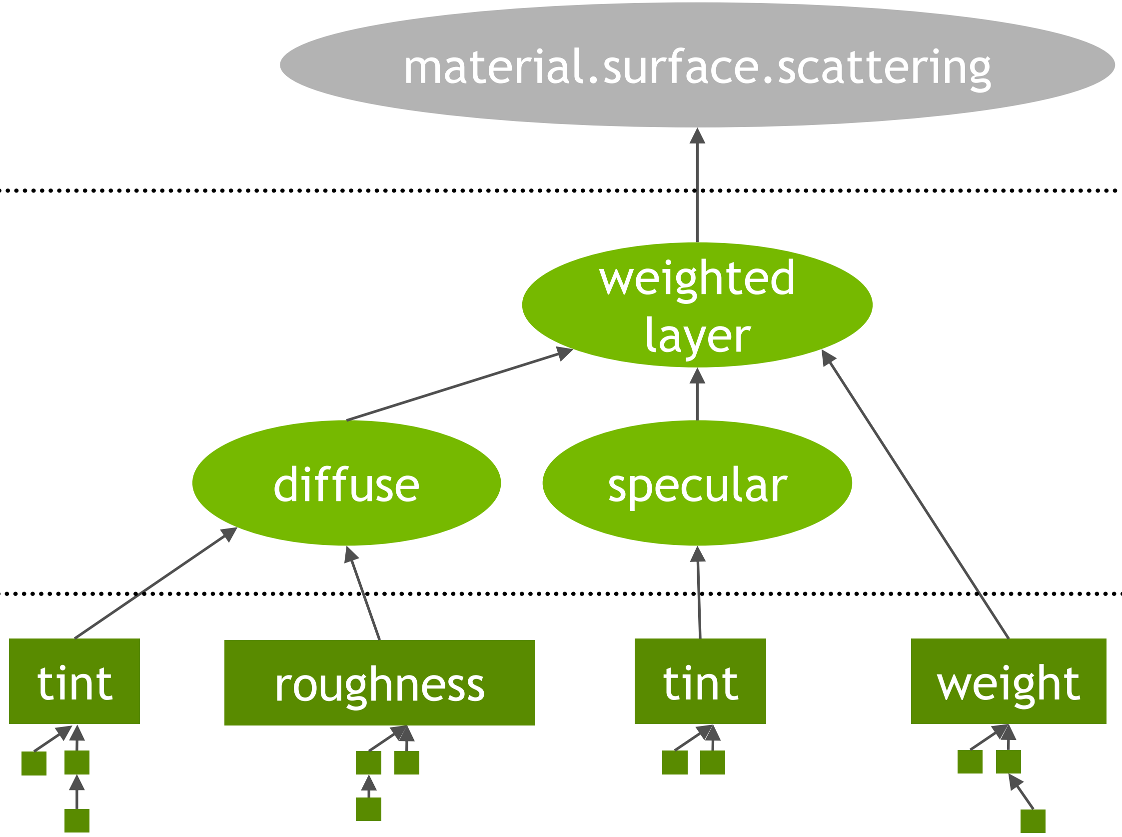
Working with a compiled material

Graph of compiled material

Material model field

Distribution functions

Texturing functions



Working with a compiled material

Inspect: Examine graph structure of compiled material

Compile: Use MDL backends to generate target code for

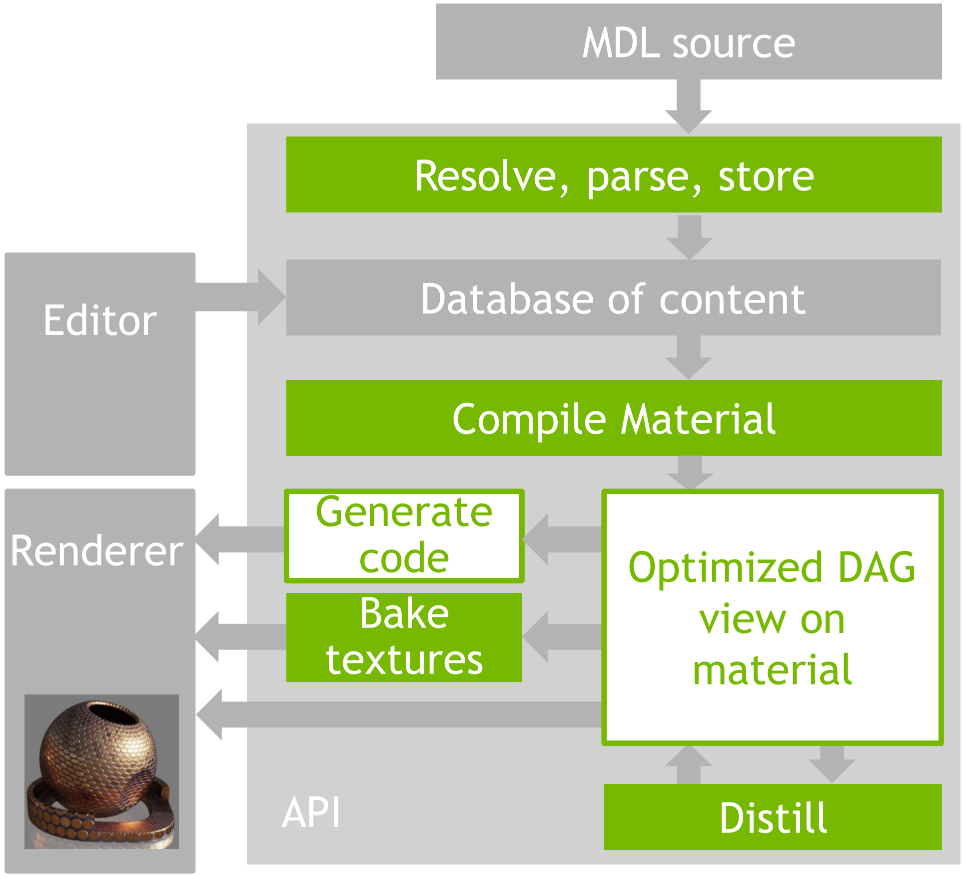
- texturing functions
- distribution functions (not for GLSL)

Distill: Use Distiller API to

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

Executing texturing functions

Executing texturing functions



Executing texturing functions

Using a backend

Select backend → Configure backend → Generate code → Execute code

```
// get a backend, e.g. the CUDA PTX backend
Handle<IMdl_backend> backend(
    mdl_compiler->get_backend(IMdl_compiler::MB_CUDA_PTX));

// set some backend specific options
backend->set_option("num_texture_results", "16");
backend->set_option("tex_lookup_call_mode", "direct_call");
...
```

→ examples/mdl_sdk/compilation

Executing texturing functions

Generating target code

Select backend → Configure backend → **Generate code** → Execute code

```
// translate function
Handle<ITarget_code> code(
    backend->translate_material_expression(
        transaction,
        compiled_material,
        "surface.scattering.tint",
        "my_material_expression",
        context));
```

Path to texturing function within compiled material

→ examples/mdl_sdk/compilation

Executing texturing functions

Executing target code

Select backend → Configure backend → Generate code → Execute code

- Access to shading state
- Argument passing in class compilation mode
- Runtime for resource access

→ examples/mdl_sdk/compilation

Executing texturing functions

MDL shading state provided by the renderer

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

    float4             *text_results;           // used by _bsdf_init()
    const char         *ro_data_segment;        // read-only data segment

    const float4       *world_to_object;        // world-to-object transform matrix
    const float4       *object_to_world;        // object-to-world transform matrix
    int                 object_id;              // state::object_id()
};
```

→ include/mi/neuraylib/target_code_types.h

Executing texturing functions

Calling your code on the CPU

```
// setup state
Shading_state_material state = {...};

// arguments of class-compiled material
Handle<ITarget_argument_block> arg_block_data = ...;

// call function
float3 result;
target_code->execute(
        function_index,
        state,
        /*tex_handler=*/ nullptr,
        arg_block_data.get(),
        &result);
```

→ examples/mdl_sdk/execution_native

Executing texturing functions

Passing arguments to class compiled materials

Generated `ITarget_code` contains

- `ITarget_argument_block`: Parameter values of compiled material
- `ITarget_value_layout`: Layout of argument block

i-th element in layout corresponds to i-th parameter of compiled material

→ examples/mdl_sdk/execution_hlsl

Executing texturing functions

Providing a runtime for resource-access

Renderer manages resource data

- MDL resources: textures, BSDF measurements, light profiles
- Large constant data blocks

Renderer provides access functions to this data (“Runtime”)

- Backend specific
- Example implementations shipped with the SDK
- Native backend provides optional built-in runtime

→ examples/mdl_sdk/execution_*

Executing texturing functions

Calling your code in CUDA

PTX backend generates string of PTX code

Can be linked to your kernel using `cuLinkAddData()`

Executing texturing functions

Calling your code in CUDA

```
// setup state
const Shading_state_material state = {...};

// texture lookup handler
const Resource_data res_data = {NULL, &my_texture_handler};

// arguments of class-compiled material
const char *arg_block_data = ...;

// call function
float3 result;
my_material_expression(&result, &state, &res_data, /*exception_state=*/ NULL,
    arg_block_data);
```

→ examples/mdl_sdk/execution_cuda

Executing texturing functions

Calling your code in OptiX

PTX backend generates string of PTX code

MDL utility code shipped with the OptiX SDK examples

- Creates callable programs
- Includes a runtime for bitmap texture access (and data upload to OptiX buffers)

```
mdl_helper = new Mdl_helper(optix_context);
...
optix::Program tint_prog = mdl_helper->compile_expression(...);

optix_material["my_material_expression"]->setProgramId(tint_prog);
```

Executing texturing functions

Calling your code in HLSL

HLSL backend generates string of HLSL code

Can be used in your HLSL shader (e.g. DXR closest-hit)

Executing texturing functions

Calling your code in HLSL

```
#include "mdl_target_code_types.hlsli"
#include "renderer_mdl_runtime.hlsli"

[ insert generated code here ]

[ ... ]

// setup state
Shading_state_material mdl_state = { ... };

// call function
float3 result = my_material_expression(mdl_state);
```

→ examples/mdl_sdk/execution_hlsl

Texture filtering

Providing derivatives to texture functions

Texture filtering may require derivatives of UV-input with respect to screen-space coordinates for anti-aliasing

UV-input typically driven by an expression of `state::texture_coordinate()`

MDL SDK offers automatic computation of derivatives of such expressions and passes them to the texture runtime

→ Enable with backend option "*texture_runtime_with_derivs*"

→ examples/mdl_sdk/df_cuda

Texture filtering

Providing derivatives to texture functions

Renderer needs to provide

- *Shading_state_material_with_derivs* as state
  Texture coordinates with derivatives struct type (*float2 val*, *dx*, *dy*)
- *tex_lookup_deriv_float4_2d* / *tex_lookup_deriv_float3_2d*
  Can call functions like tex2DGrad with provided derivatives

→ examples/mdl_sdk/df_cuda

Multiple texturing functions

Creating code for multiple texturing functions

Generating target code per expression may cause problems

- Common utility code is compiled several times
- Can cause name clashes if generated PTX/HLSL/GLSL code is used in a single program

Using a “link unit” solves this problem

- Add multiple expressions to a link unit
- Then translate link unit

Optimizing render state access

Customized state

GLSL backend offers a configurable state

- Access via state field,
- function (e.g. “normal()”),
- shader input variables (“normal”),
- or always zero

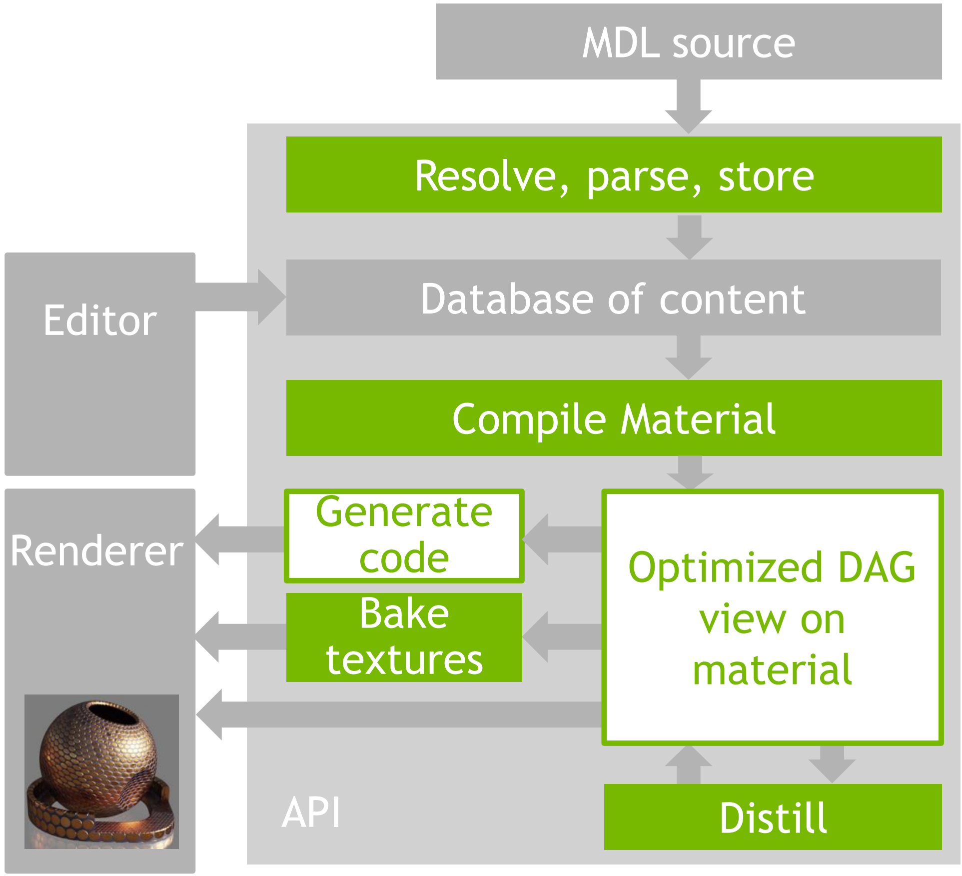
Native, PTX, and LLVM-IR backends allow customized access

- Pass pointer to your state data
- Provide accessor functions for all state fields as LLVM bitcode

→ examples/mdl_sdk/user_modules

Executing distribution functions

Executing distribution functions



From material to rendering code

Implementing the declarative part of the material

Actual shading code for material description can be highly renderer specific

- A renderer may analyze the declarative part of the 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

Compiling BSDFs

Generating functions for BSDFs

```
// for a single material
Handle<ITarget_code> target_code(backend->translate_material_df(
        transaction,
        compiled_material,
        "surface.scattering",
        "my_material",
        context));
```

Path to BSDF
within compiled material

Compiling BSDFs

Generates building blocks for a physically based renderer

init: Shared initialization for the current shading point

evaluate: Evaluation of the BSDF for a given outgoing and incoming direction

sample: Importance sampling of an incoming for a given outgoing direction

pdf: Probability density computation of that importance sampling

my_module::my_material →

```
void my_material_init(…)
void my_material_evaluate(…)
void my_material_sample(…)
void my_material_pdf(…)
```

Calling BSDFs

Function signatures

```
typedef void (Bsdf_init_function)     (Shading_state_material *state,
                                        const Resource_data *res_data,
                                        const void *exception_state,
                                        const char *arg_block_data);

typedef void (Bsdf_sample_function)   (Bsdf_sample_data *data,
                                        const Shading_state_material *state,
                                        const Resource_data *res_data,
                                        const void *exception_state,
                                        const char *arg_block_data);

typedef void (Bsdf_evaluate_function)(Bsdf_evaluate_data *data,
                                        ...);

typedef void (Bsdf_pdf_function)      (Bsdf_pdf_data *data,
                                        ...);
```

→ include/mi/neuraylib/target_code_types.h

Using BSDFs

Initialization function "_init"

Often there are multiple calls to evaluate and / or calls to both evaluate and sample for the same shading point

No need to compute texturing functions more than once

BSDF init function caches those results in `state->text_results`

Size of array configurable, excess texturing functions get recomputed

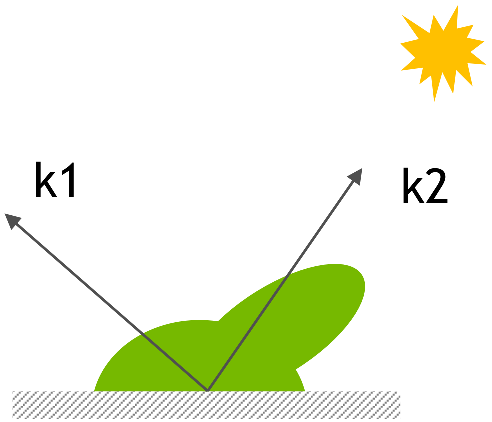
Further replaces `state->normal` by MDL's `material_geometry.normal`

Using BSDFs

Evaluation function “_evaluate”

```
struct Bsdf_evaluate_data {
    // Input fields
    float3        ior1;     // IOR current medium
    float3        ior2;     // IOR other side
    float3        k1;       // outgoing direction
    float3        k2;       // incoming direction

    // Output fields
    float3        bsdf;     // bsdf * dot(normal, k2)
    float         pdf;      // pdf (non-projected hemisphere)
};
```



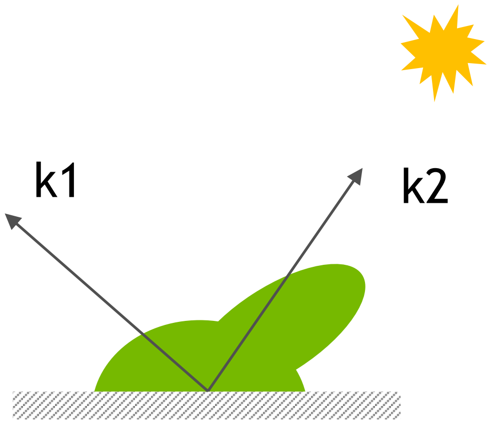
→ include/mi/neuraylib/target_code_types.h

Using BSDFs

Probability density function “_pdf”

```
struct Bsdf_pdf_data {
    // Input fields
    float3          ior1;     // IOR current medium
    float3          ior2;     // IOR other side
    float3          k1;       // outgoing direction
    float3          k2;       // incoming direction

    // Output fields
    float           pdf;      // pdf (non-projected hemisphere)
};
```



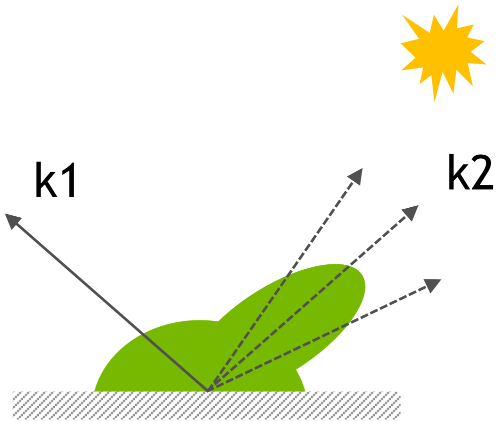
→ include/mi/neuraylib/target_code_types.h

Using BSDFs

Importance sampling function “_sample”

```
struct  {
    // Input fields
    float3          ior1;           // IOR current medium
    float3          ior2;           // IOR other side
    float3          k1;             // outgoing direction
    float3          xi;             // pseudo-random sample
                                    // number

    // Output fields
    float3          k2;             // incoming direction
    float           pdf;            // pdf (non-projected hemisphere)
    float3          bsdf_over_pdf;  // bsdf * dot(normal, k2) / pdf
    Bsdf_event_type event_type;     // the type of event for the generated sample
                                    // (e.g. BSDF_EVENT_GLOSSY_REFLECTION or
                                    //  BSDF_EVENT_ABSORB)
};
```



→ include/mi/neuraylib/target_code_types.h

Using BSDFs

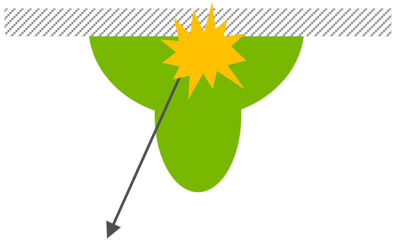
Simplistic path tracer example

```
// init
my_material_init(&state, ...);
// light sampling: evaluate
for (int l = 0; l < num_light_sources; ++l) {
    eval_data.k2 = get_light_direction(l, ...);
    contrib += current_weight * my_material_evaluate(&eval_data, &state, ...) *
        get_light_contribution(l, ...);
}
// continue path: importance sample BSDF
sample_data.xi = ...;
my_material_sample(&sample_data, &state, ...);
if (sample_data.event == BSDF_EVENT_ABSORB){
   absorb(...);
   return;
}
current_weight *= sample_data.bsdf_over_pdf;
ray.direction = sample_data.k2;
...
```

→ examples/mdl_sdk/df_cuda

Using EDFs

Analogously to using BSDFs



Same building block functions: **`init`**, **`evaluate`**, **`pdf`**, **`sample`**

with similar parameters: **`Edf_evaluate_data`**, **`Edf_sample_data`**, **`Edf_pdf_data`**

```
struct Edf_evaluate_data {
    // Input fields
    float3              k1;                 // outgoing direction

    // Output fields
    float               cos;                // dot(normal, k1)
    float3              edf;                // emission
    float               pdf;                // pdf (non-projected hemisphere)
};
```

→ include/mi/neuraylib/target_code_types.h

Using BSDFs, EDFs and texturing functions

Bundle functions that belong to one material to use one argument block

```
material glowing(color glow_int: color(10.0)) = material(
  surface: material_surface(
    scattering: simple_glossy_bsdf(roughness_u: 0.1),
    emission: material_emission(
        emission: diffuse_edf(),
         intensity: glow_int)));
```

```
// add all functions of interest at once
Target_function_description descs[3] = {
    Target_function_description("surface.scattering"),
    Target_function_description("surface.emission.emission"),
    Target_function_description("surface.emission.intensity")};

link_unit->add_material(compiled_material, descs, 3, context));
```

Using DFs

SDK Examples

Simple mini-renderer in CUDA, computing direct light on a sphere using

- BSDF importance sampling
- BSDF evaluation of importance sampled environment light
- Combined via multiple importance sampling
- Evaluate EDF and emission intensity

OptiX example included in OptiX SDK

- Includes utility code to generate callable programs for BSDF functions

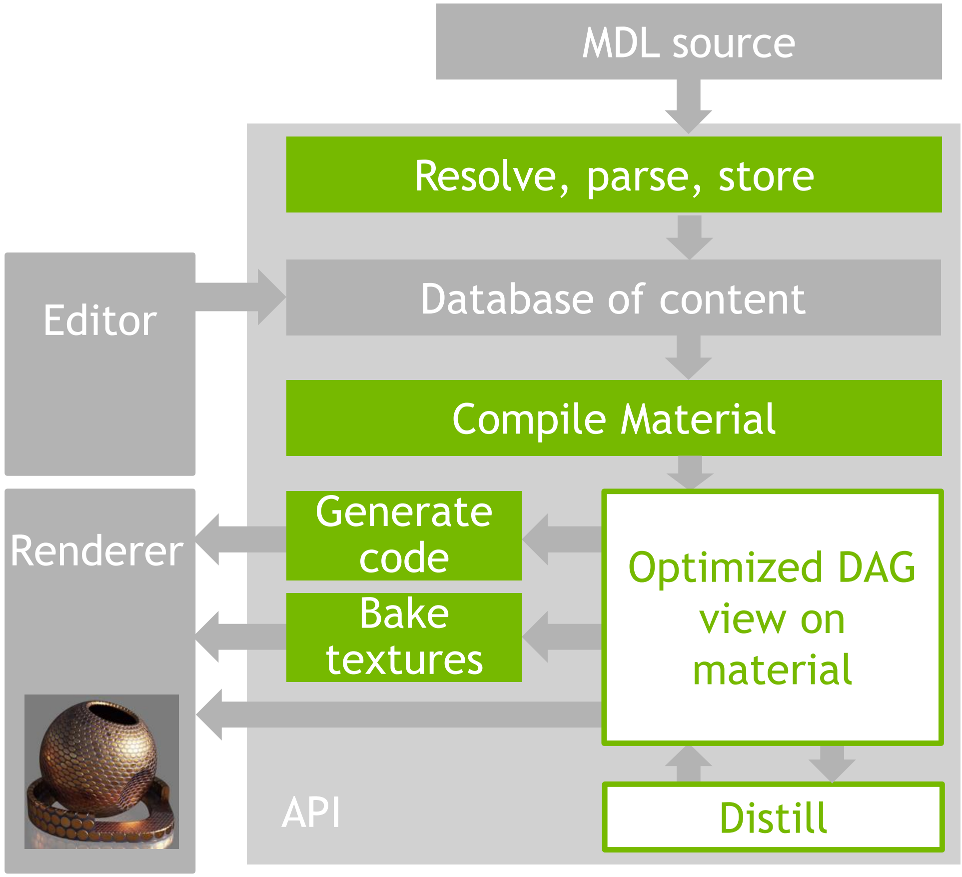
Using DFs

Live demo DXR path tracer

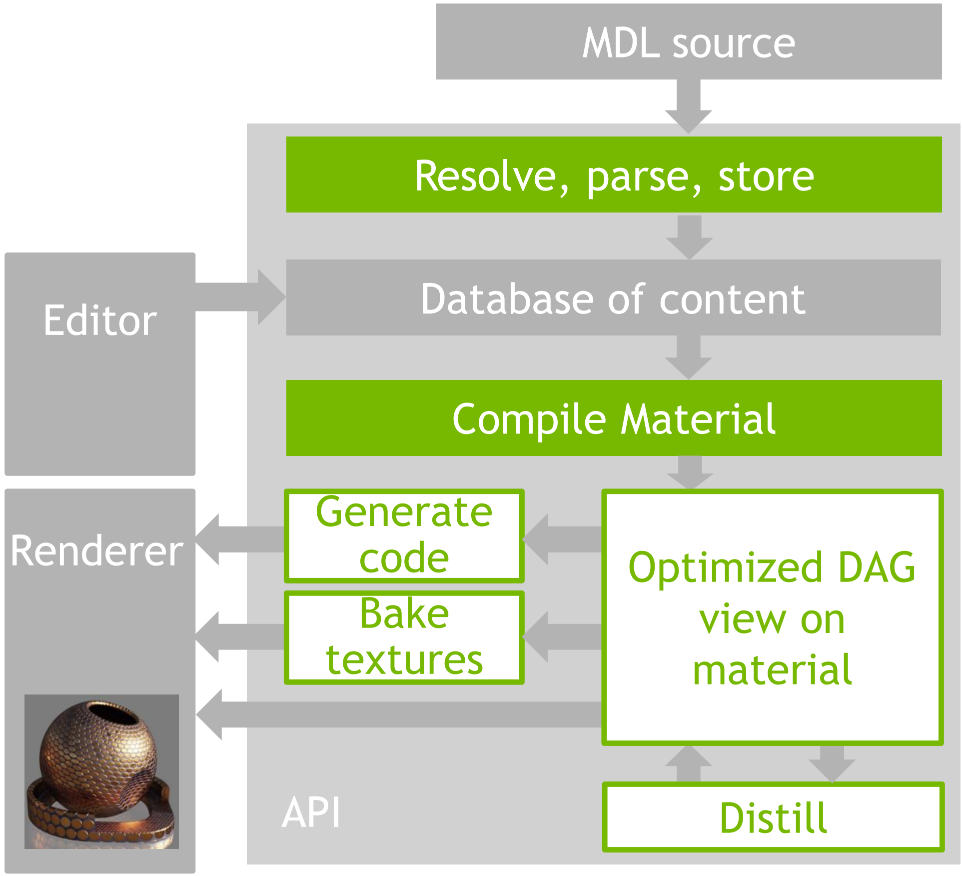
- Using BSDFs in HLSL
- GLTF model loading
- Interactive material parameter editing
- Image-based lighting

Distilling to a fixed target model

Distilling to a fixed target model



Distilling to a fixed target model



Distilling to a fixed material model

Overview

MDL Material	distill →	Fixed Material Model
Complex BSDF layering Complex procedurals		Simple BSDF structure One texture per parameter

MDL distilling → Adapts MDL for realtime applications

Term rewriting system with rule sets to simplify expressions

Available models: diffuse, diffuse/glossy, UE4, specular/glossy, transmissive PBR

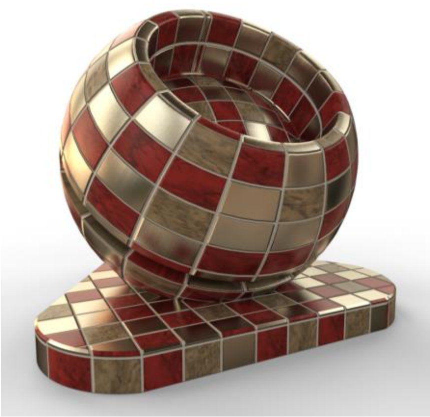
Distilling to a fixed material model

Overview

Distilling a metal

original

diffuse



diffuse/glossy

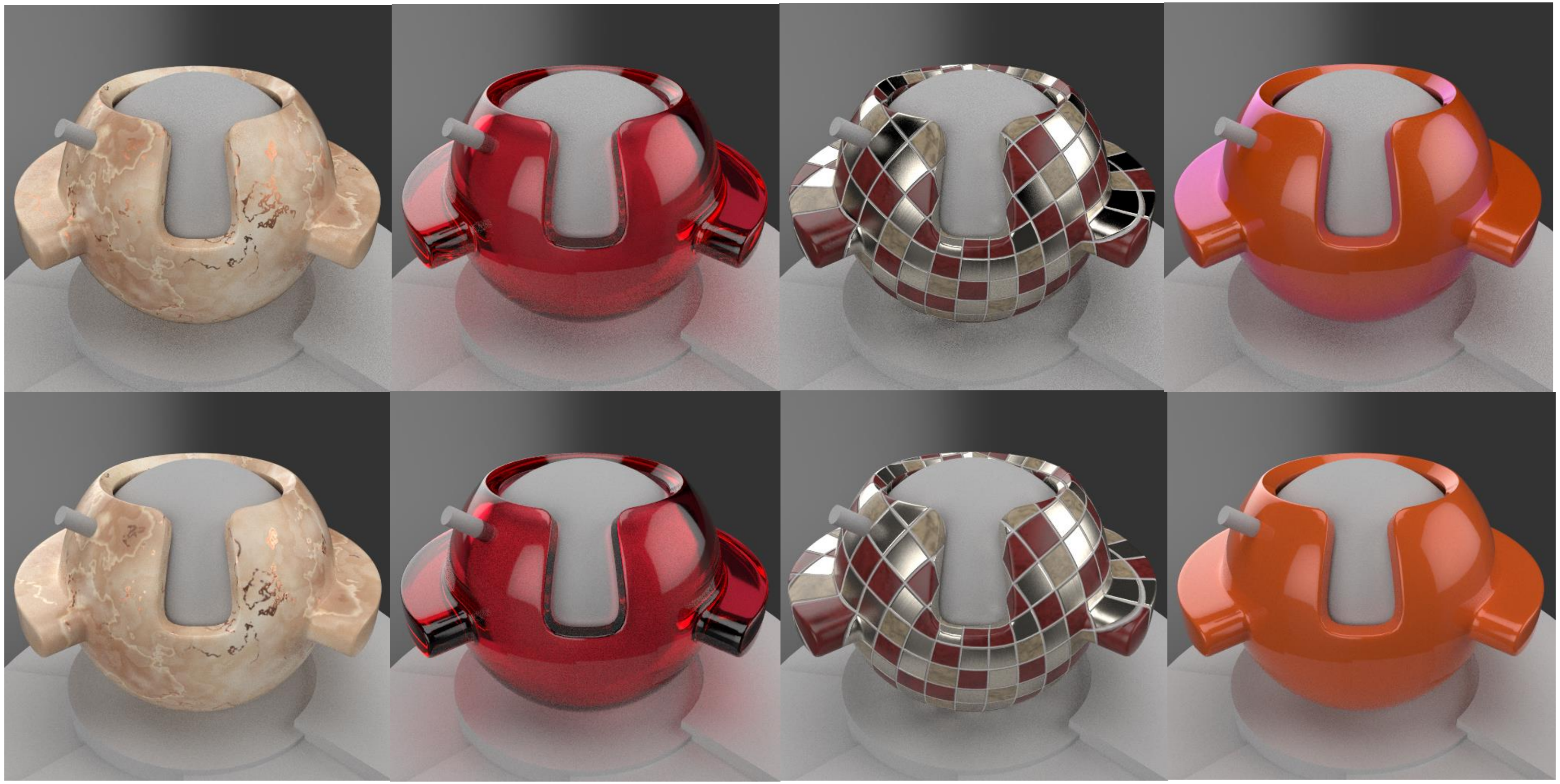
UE4

Distilling to a fixed material model

Overview

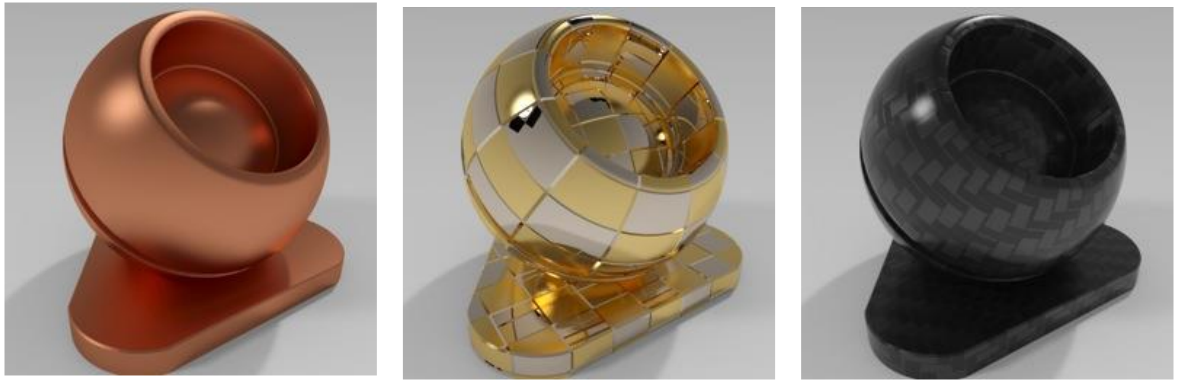
Original:
Iray MDL

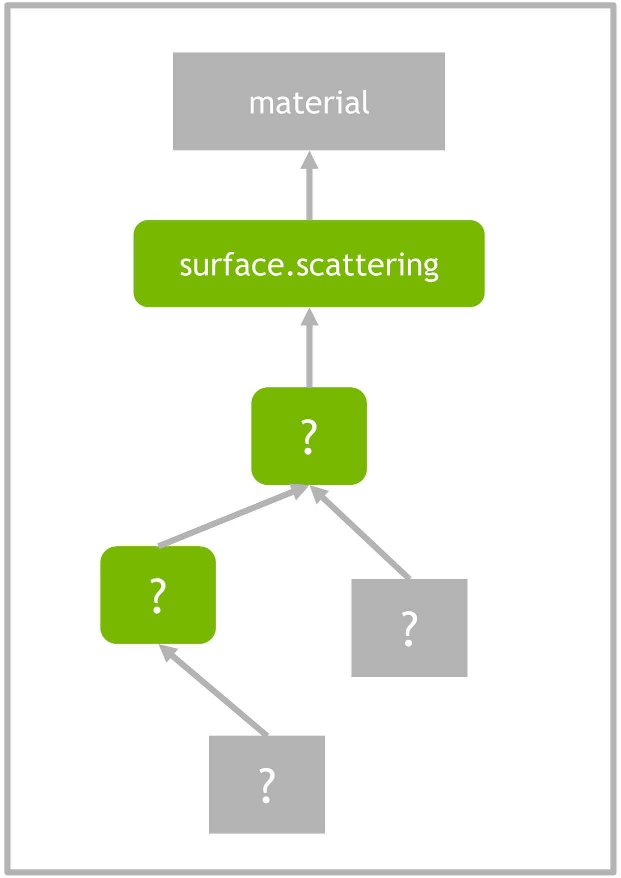
Projection:
Dassault Stellar
with Enterprise
PBR using trans-
missive PBR



Distilling to a fixed material model

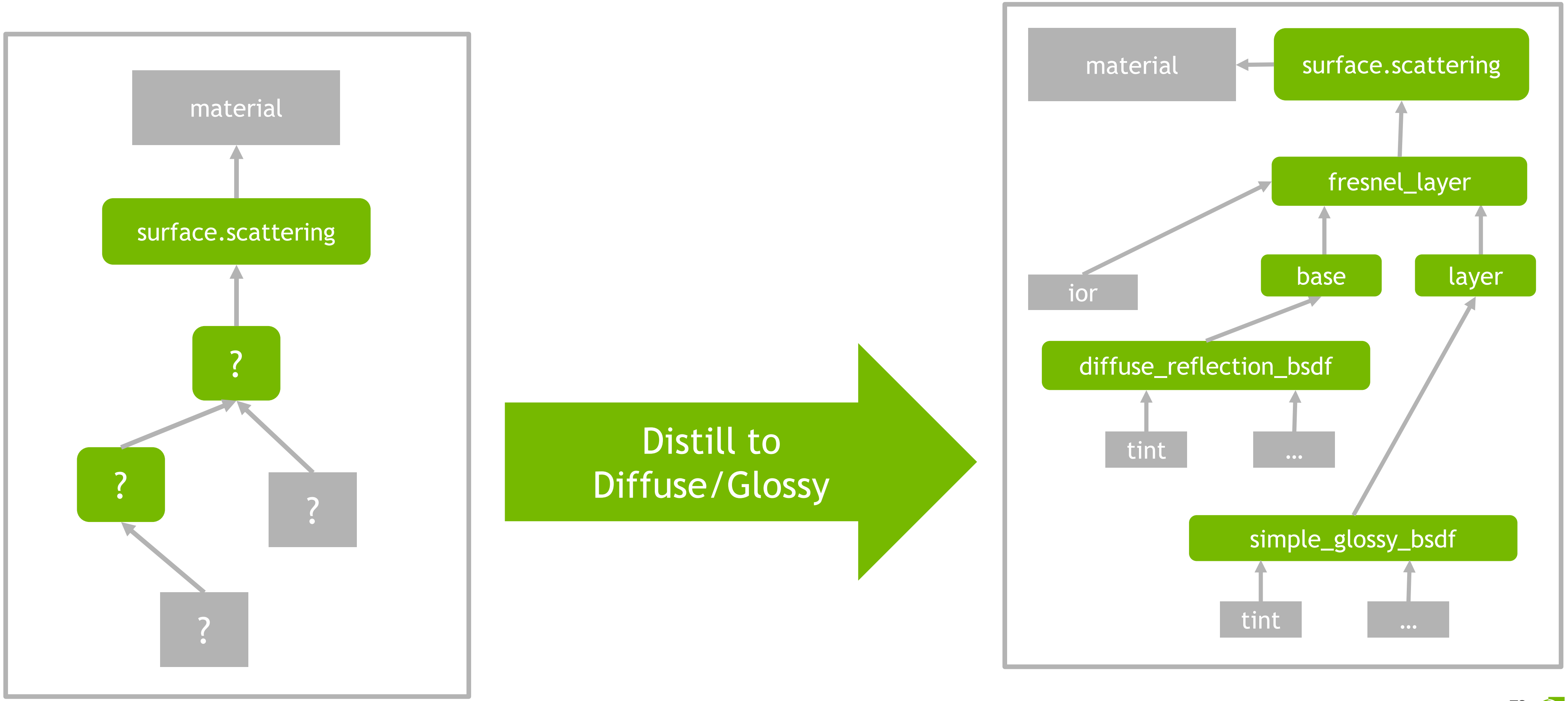
Overview





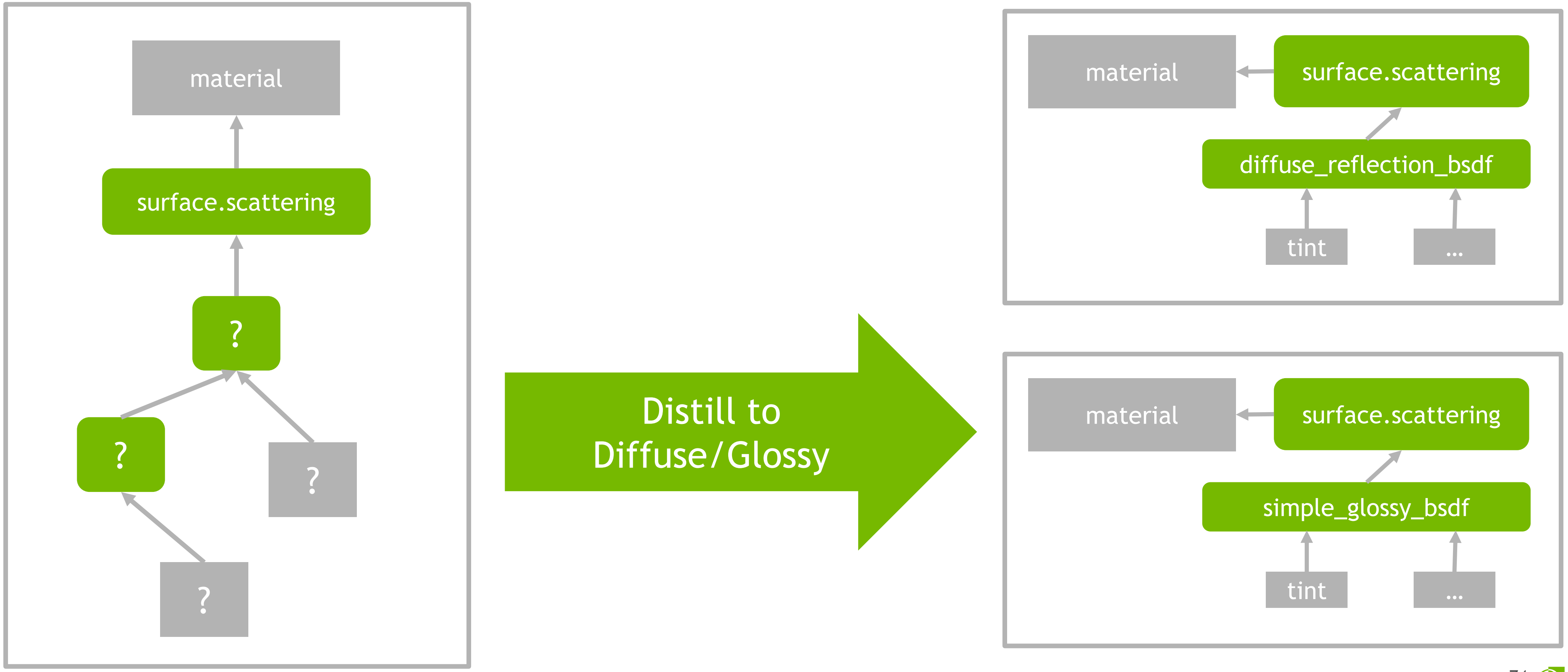
Distilling to a fixed material model

Overview



Distilling to a fixed material model

Overview



Distilling to a fixed material model

Code

Get Distiller API → Distill Material → Gather Expressions → Process Expressions

```
// Acquire distilling API used for material distilling and baking
Handle<IMdl_distiller_api> distiller_api(
    mdl_sdk->get_api_component<IMdl_distiller_api>());

// Distill the compiled material to the diffuse_glossy material model
Handle<const ICompiled_material> distiller_material(
    distiller_api->distill_material(compiled_material.get(),
       "diffuse_glossy"));
```

→ examples/distilling

Distilling to a fixed material model

Code

Get Distiller API → Distill Material → **Gather Expressions** → Process Expressions

```
// Get diffuse color
const char* diffuse_path;

switch(get_call_semantic(distilled_material, "surface.scattering")){
case DS_INTRINSIC_DF_DIFFUSE_REFLECTION_BSDF:
    diffuse_path = "surface.scattering.tint";
    break;
case DS_INTRINSIC_DF_FRESNEL_LAYER:
    diffuse_path = "surface.scattering.base.tint";
    break;
…
}
```

→ examples/distilling

Distilling to a fixed material model

Code

Get Distiller API → Distill Material → Gather Expressions → **Process Expressions**

```
// Bake …
Handle<const IBaker> baker(distiller_api->create_baker(
    distilled_material.get(), diffuse_path));

Handle<ICanvas> canvas = ...
baker->bake_texture(canvas.get());

// … or generate code
link_unit->add_material_expression(
    compiled_material.get(), diffuse_path, "get_diffuse");
```

→ examples/distilling_glsl

Distilling to a fixed material model

SDK Examples

Simple console application

- Distills a material to the desired target model
- Analyses result and bakes expressions to textures

Simple OpenGL example

- Distills input material to UE4
- Generates GLSL code for all relevant texturing functions
- Integrates generated code with a UE4 like GLSL shader
- Renders an IBL lit sphere

MDL SDK

How to get

- Download from https://developer.nvidia.com/mdl-sdk
- An introduction to MDL can be found at www.mdlhandbook.com

Further Information on MDL

www.nvidia.com/mdl raytracing-docs.nvidia.com/mdl/index.html

Documents

NVIDIA Material Definition Language

- *Technical Introduction*
- *Handbook*
- *Language Specification*

GTC On-Demand

on-demand-gtc.gputechconf.com

MDL@GTC

Mon 9 AM SJCC 230B	*Sharing Physically Based Materials Between Renderers with MDL*
Mon 10 AM SJCC 230B	*Integrating the NVIDIA Material Definition Language MDL in Your Application*
Mon 11 AM Hilton Hotel Almaden 2	*A New PBR Material Serving Mobile, Web, Real-Time Engines and Ray Tracing*
Tue 9 AM Hilton Hotel Almaden 2	*Multi-Platform Photo-Real Rendering: Utilizing NVIDIA'S MDL and Allegorithmic's Substance Suite for Product Imaging*
Thu 10 AM SJCC 230C	*Real-Time Ray Tracing with MDL Materials*