GTC 2018 San Jose, S8518 Tutorial

AN INTRODUCTION TO NVIDIA OPTIX

Ankit Patel, Detlef Roettger, 2018-03-26

NVIDIA

AGENDA

OptiX Overview

Programming with OptiX

New Example Applications

Motion Blur

DL Denoiser

OptiX

NVIDIA GPU Ray Casting API

High-level GPU accelerated ray-casting API

C-API to setup scene and data

Multiple program domains and per ray payload under developer's control

Flexible single ray programming model

Supports multi-GPU and NVLINK

Develop "to the algorithm"

https://developer.nvidia.com/optix

volume scattering and dispersion

hair intersection and shading

Programming with OptiX

Prerequisites

Windows, Linux, Mac OS

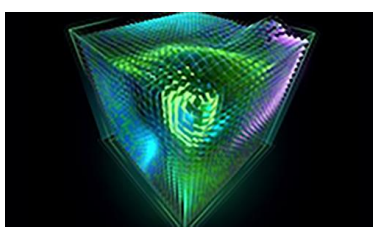
NVIDIA GPU (Kepler, Maxwell, Pascal, Volta)

Display Driver supporting CUDA 9.0

OptiX SDK

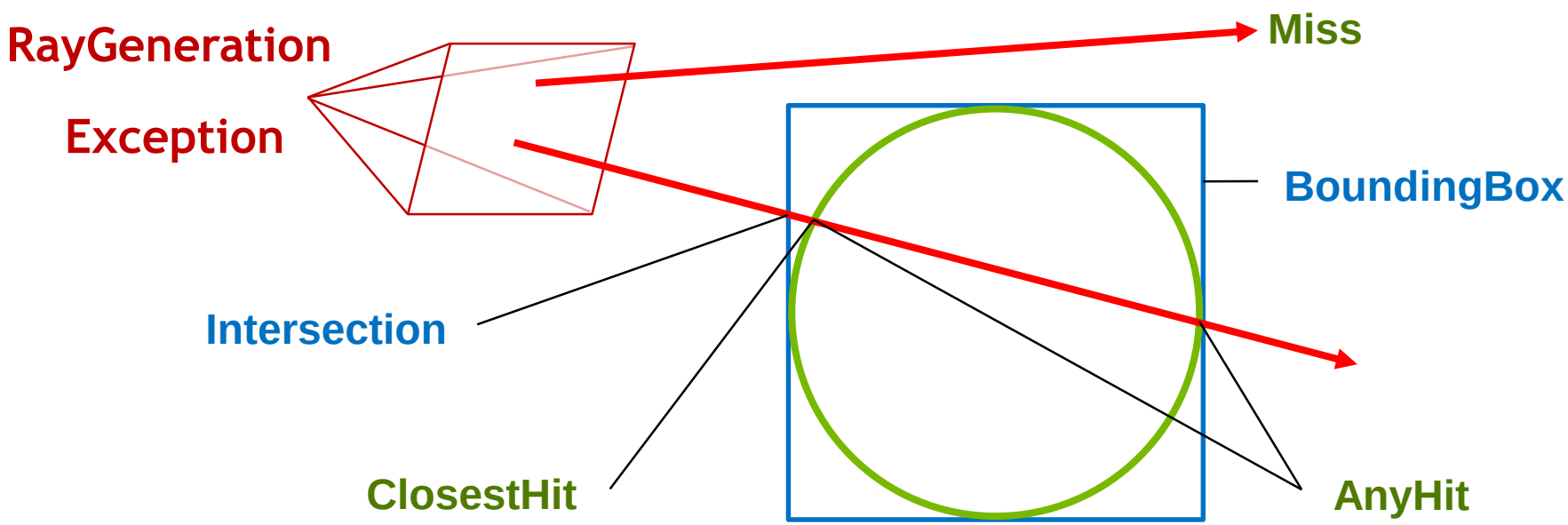
CUDA Toolkit

Host compiler supported by the CUDA Toolkit



OptiX Program Domains

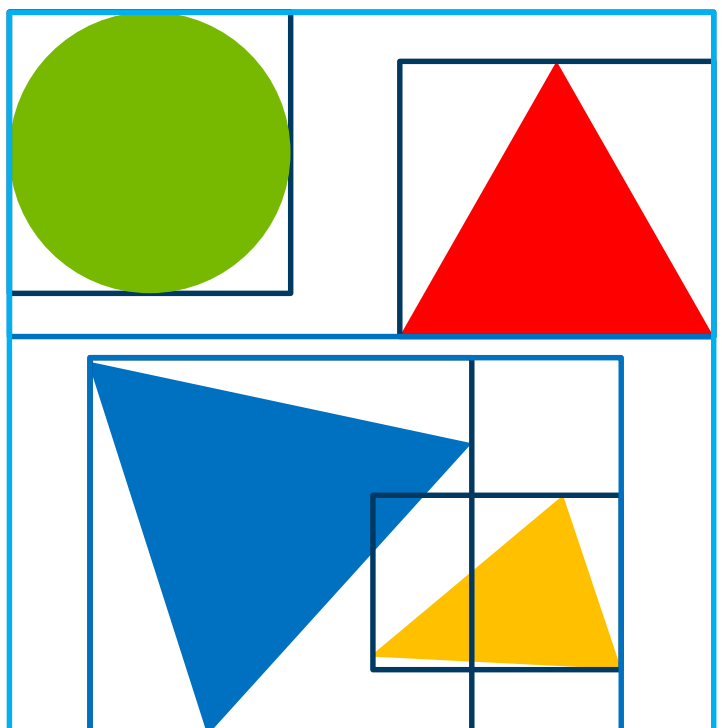
Developer controls the algorithm via CUDA C++ programs

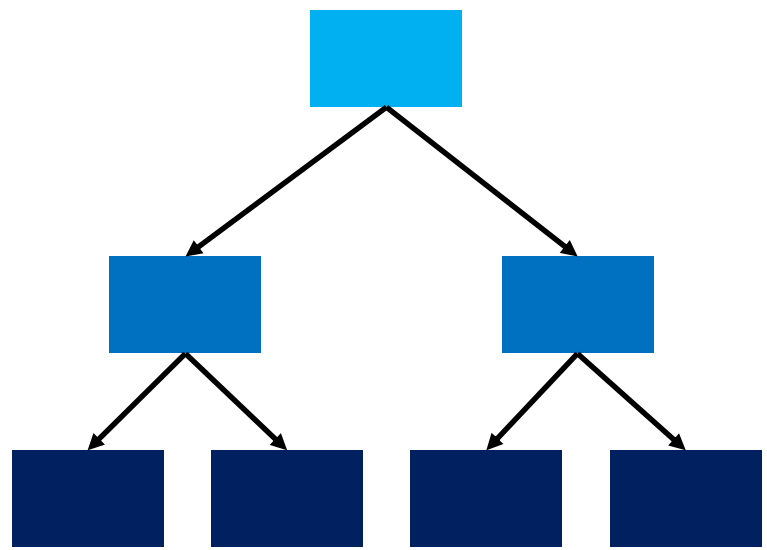


* per geometric primitive type
* per entry point
* per ray type

Acceleration Structures

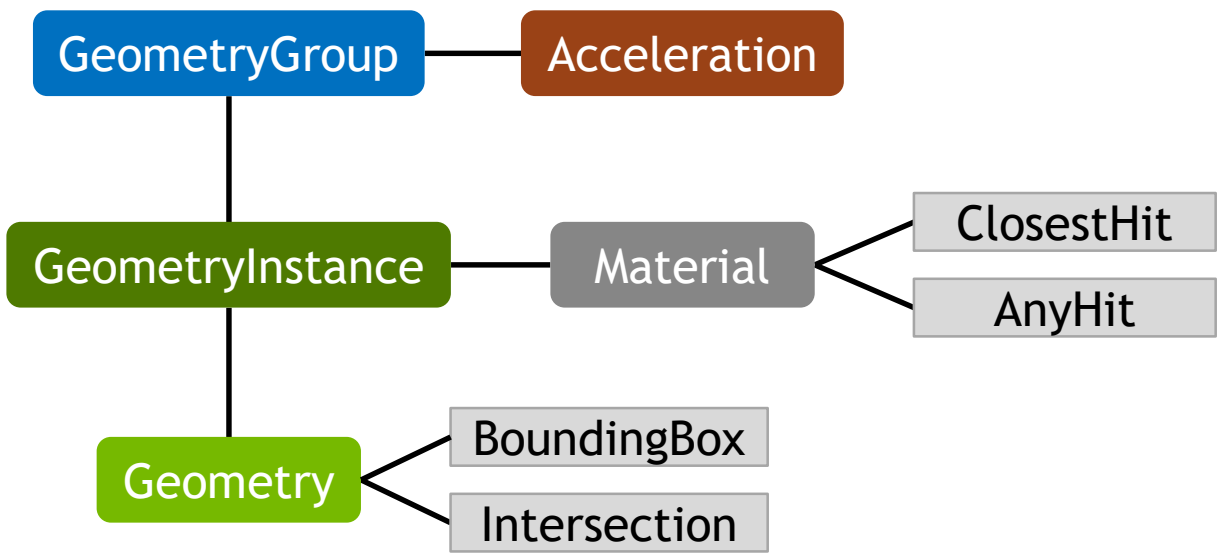
Bounding Volume Hierarchy (BVH)





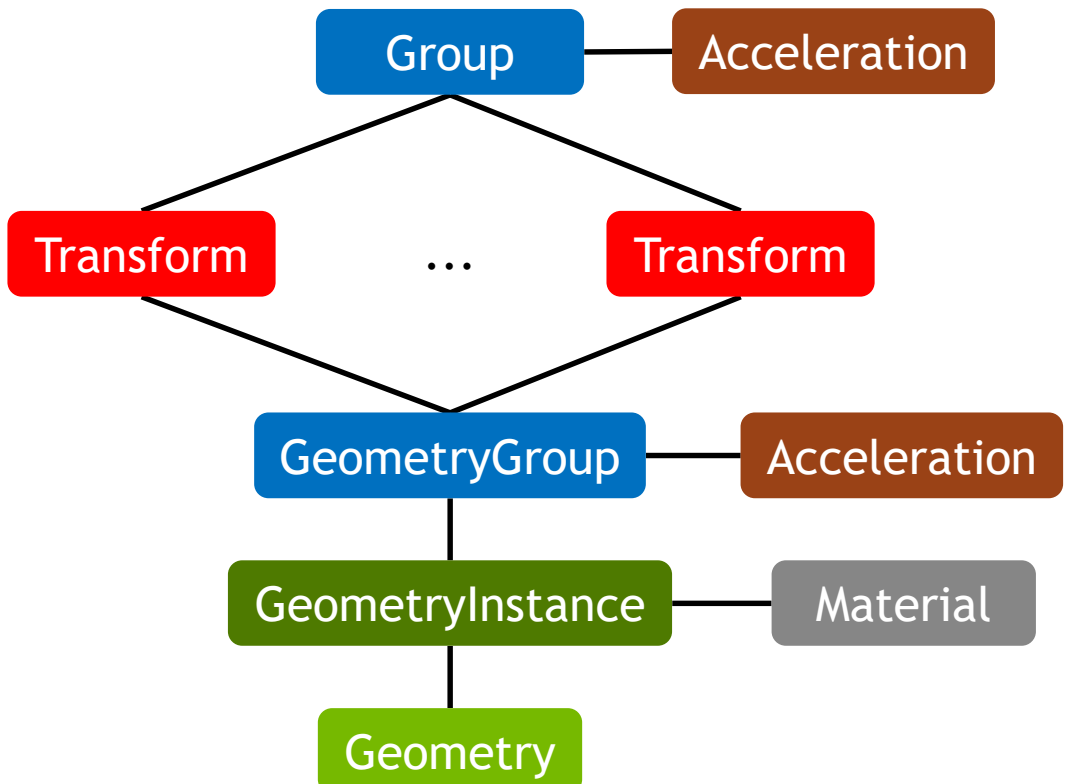
OptiX Scene Hierarchy

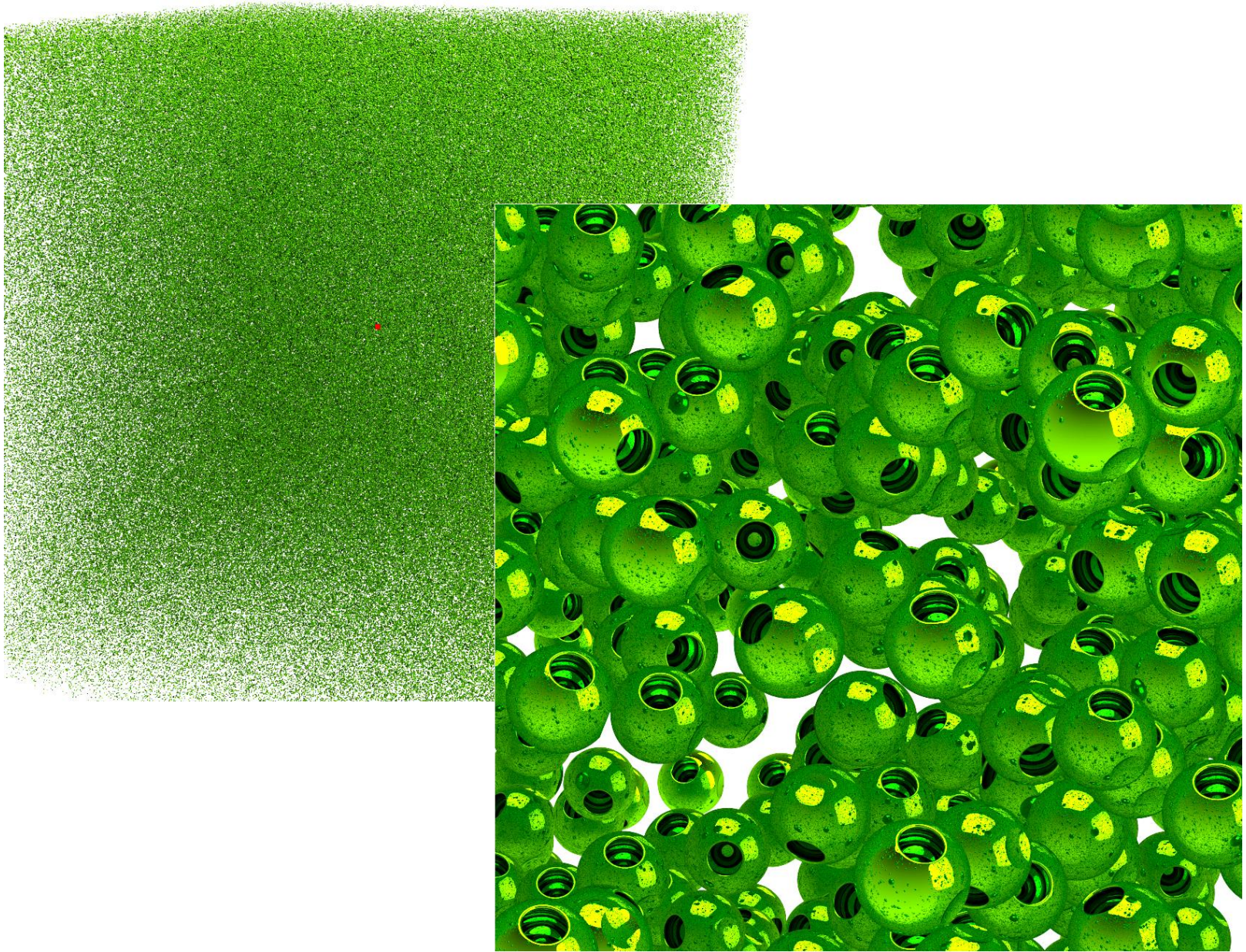
OptiX C-API to create and connect scene data



OptiX Scene Hierarchy

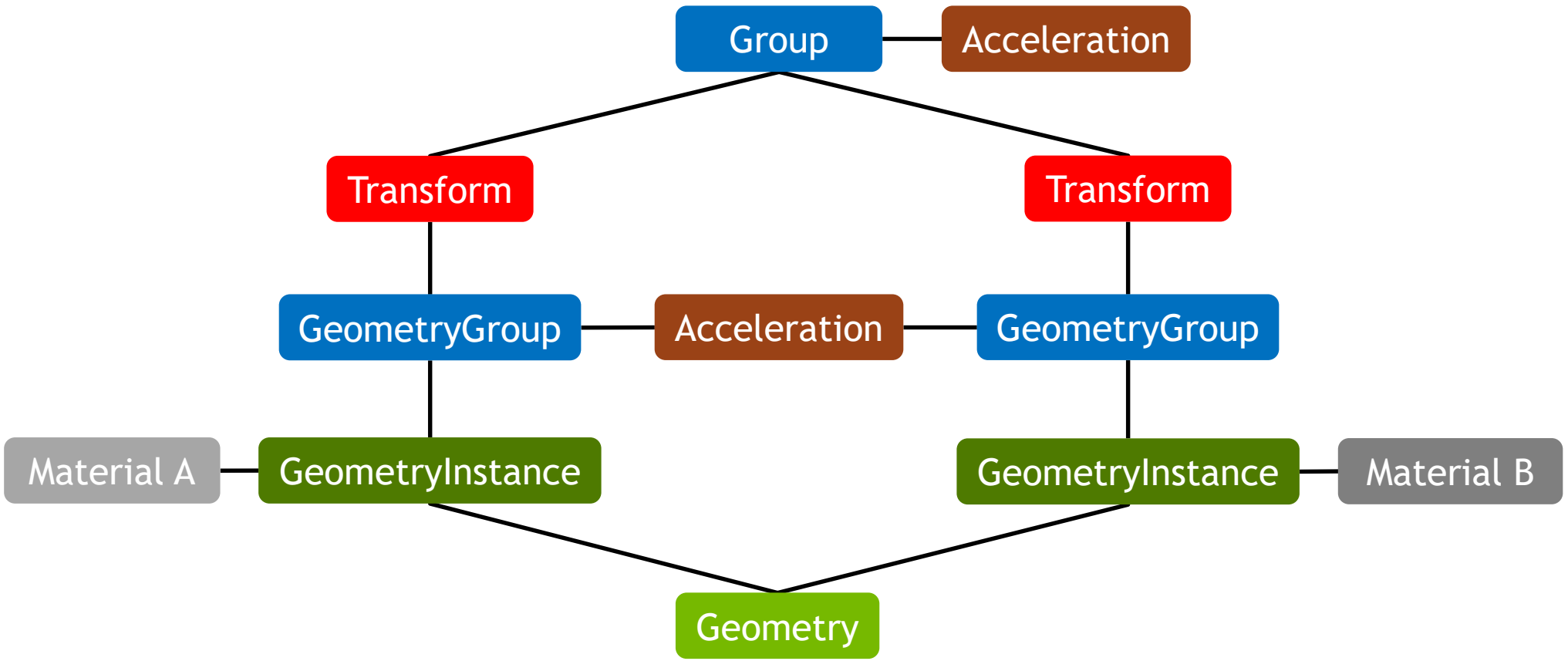
Instancing Sub-Trees





OptiX Scene Hierarchy

Acceleration Structure Sharing



OptiX Introduction Example Programs

optixIntro_01

optix::Context::create()

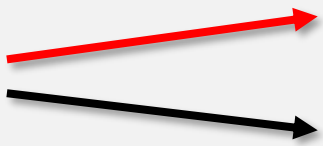
setDevices(begin, end);

RayGeneration

setEntryPointCount(num_entry_points)

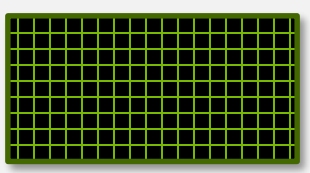
createProgramFromPTXFile(filename, program_name)

setRayGenerationProgram(index, program)



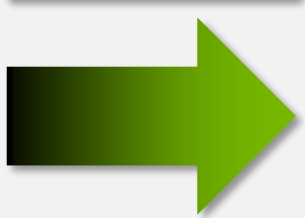
setRayTypeCount(num_ray_types)

setStackSize(bytes)



createBuffer(type, format, width, height)

context["variable_name"]->set(buffer);



launch(entry_point, width, height)

buffer->map(level, flags)

buffer->unmap()

RayGeneration Program

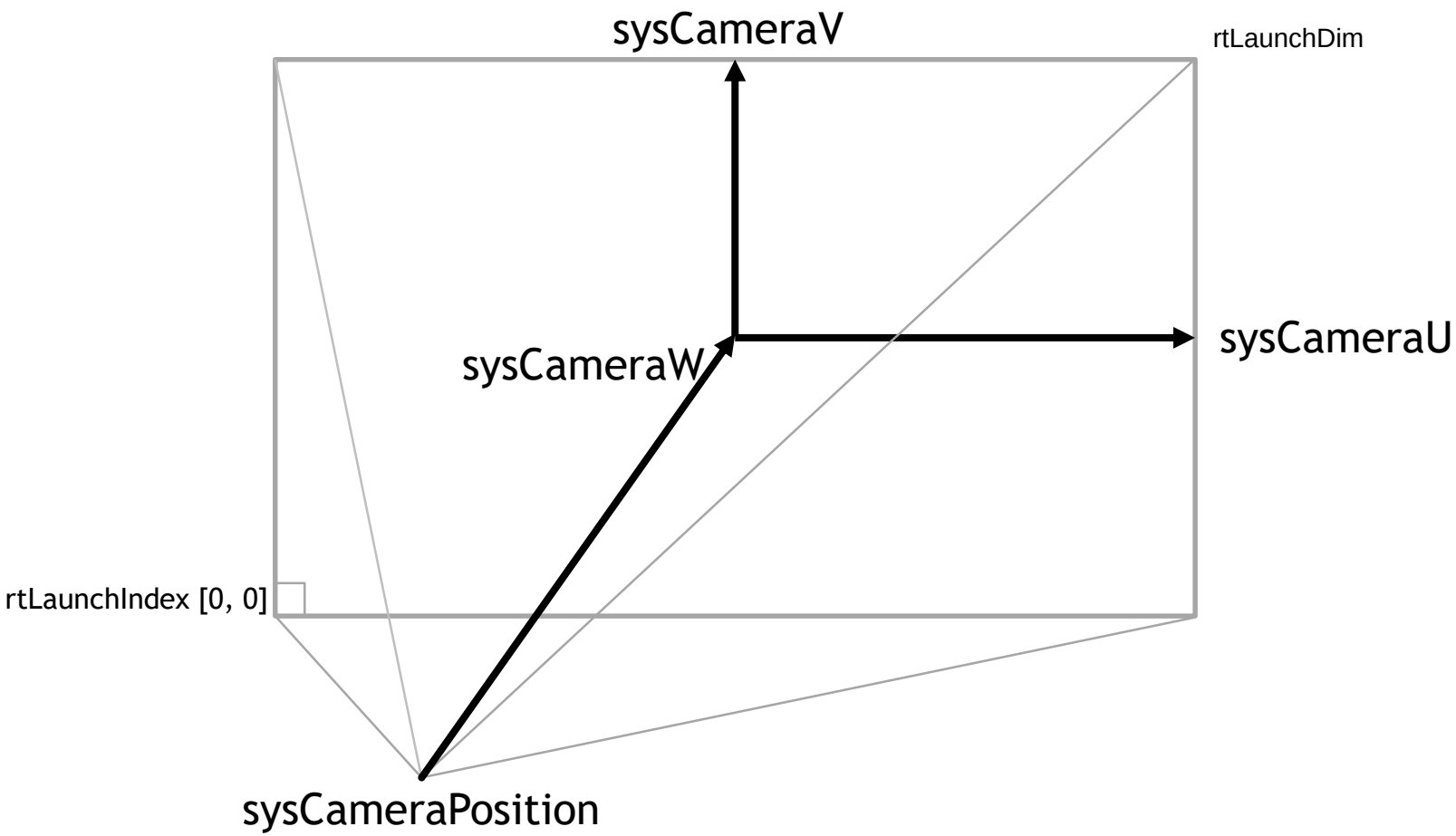
```
rtBuffer<float4, 2> sysOutputBuffer; // RGBA32F

rtDeclareVariable(uint2, theLaunchIndex, rtLaunchIndex, );

rtDeclareVariable(float3, sysColorBackground, , );

RT_PROGRAM void raygeneration()
{
  sysOutputBuffer[theLaunchIndex] = make_float4(sysColorBackground, 1.0f);
}
```

Exception Program

```
rtBuffer<float4, 2> sysOutputBuffer; // RGBA32F

rtDeclareVariable(uint2, theLaunchIndex, rtLaunchIndex, );

RT_PROGRAM void exception()
{
  rtPrintExceptionDetails();

  sysOutputBuffer[theLaunchIndex] = make_float4(1000000.0f, 0.0f, 1000000.0f, 1.0f);
}
```

optixIntro_02

Pinhole Camera



Tracing Rays

Group

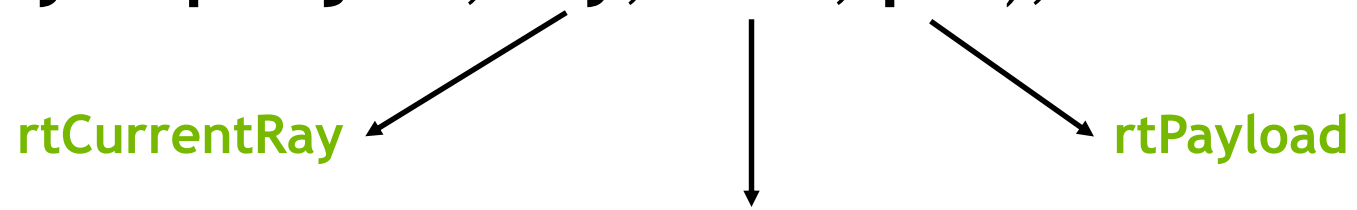
```
context["sysTopObject"]->set(group);
```

```
rtDeclareVariable(rtObject, sysTopObject, , );

optix::Ray ray = optix::make_Ray(origin, direction, raytype, t_min, t_max);

PerRayData prd;

rtTrace(sysTopObject, ray, prd);
```

Variable Semantics

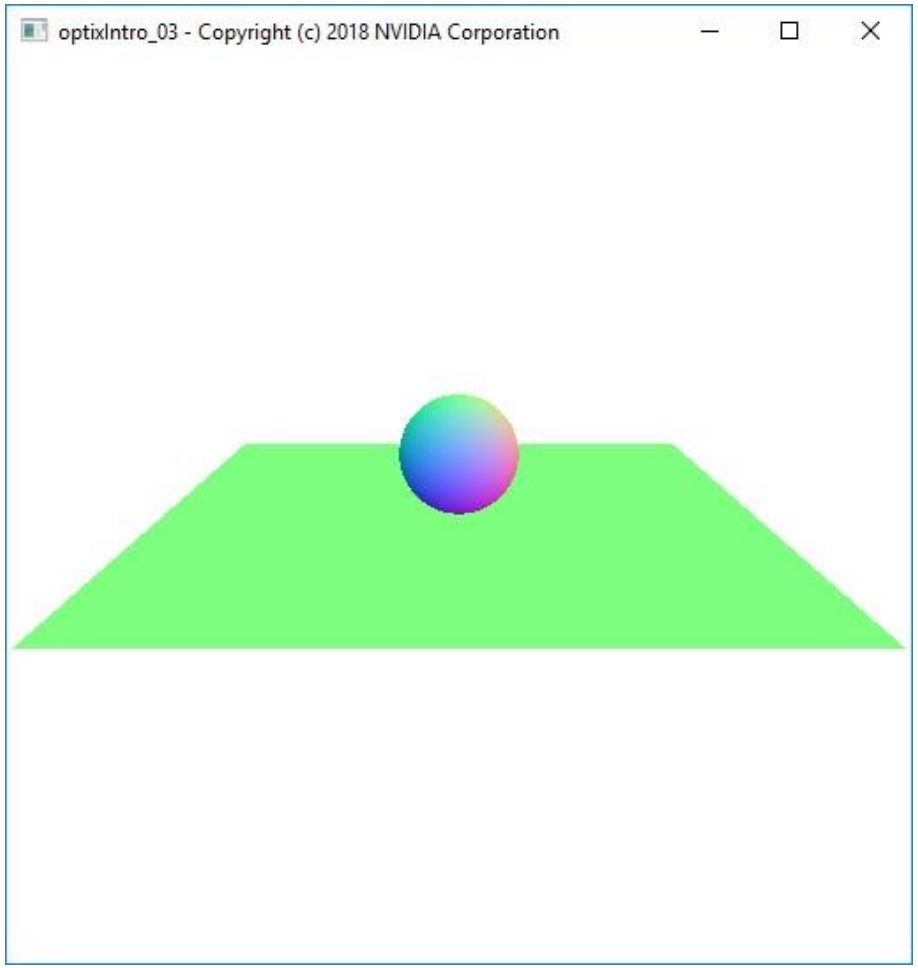
Access per ray data in other program domains

rtTrace(sysTopObject, ray, time, prd);




```
rtDeclareVariable(optix::Ray, theRay,      rtCurrentRay, );
rtDeclareVariable(float,      theTime,     rtCurrentTime, );
rtDeclareVariable(PerRayData, thePrd,      rtPayload, );
rtDeclareVariable(float,      theDistance, rtIntersectionDistance, );
```

optixIntro_03



BoundingBox Program

```
RT_PROGRAM void boundingbox_triangle_indexed(int primitiveIndex, float result[6])
{
  // uint3 indices     = ... ; // vertex indices of the triangle at primitiveIndex
  // float3 v0, v1, v2 = ... ; // vertex positions

  const float  area = optix::length(optix::cross(v1 - v0, v2 - v0));

  optix::Aabb* aabb = (optix::Aabb*) result;

  if (0.0f < area && !isinf(area)) {
    aabb->m_min = fminf(fminf(v0, v1), v2);
    aabb->m_max = fmaxf(fmaxf(v0, v1), v2);
  } else {
    aabb->invalidate();
  }
}
```

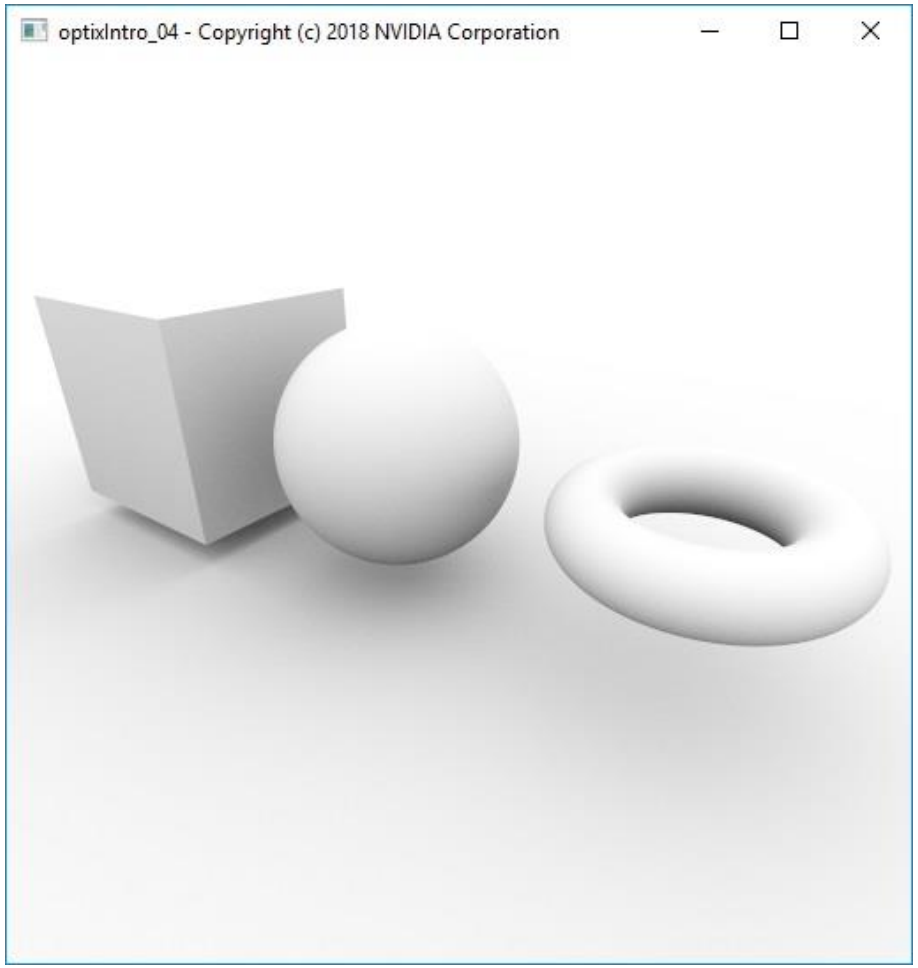
Intersection Program

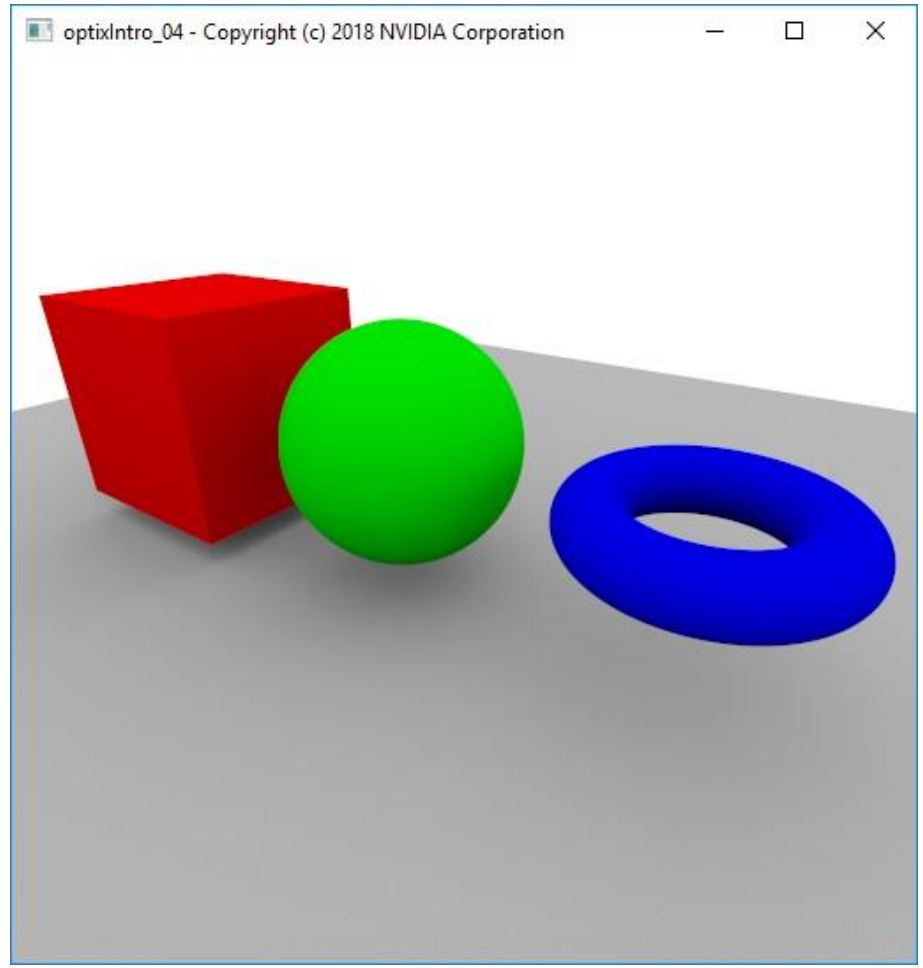
```
rtDeclareVariable(optix::Ray, theRay, rtCurrentRay, );

rtDeclareVariable(float3, varNormal, attribute NORMAL, );

RT_PROGRAM void intersection_triangle_indexed(int primitiveIndex)
{
  // uint3 indices     = ... ; // vertex indices of the triangle at primitiveIndex
  // float3 v0, v1, v2 = ... ; // vertex positions

  float3 n;
  float  t, beta, gamma;
  if (intersect_triangle(theRay, v0, v1, v2, n, t, beta, gamma)) {
    if (rtPotentialIntersection(t)) {
      // float3 n0, n1, n2 = ... ; // vertex normals
      const float alpha = 1.0f - beta - gamma;
      varNormal = n0 * alpha + n1 * beta + n2 * gamma; // interpolate shading normal
      rtReportIntersection(0);
    }
  }
}
```

ClosestHit Program

```
rtDeclareVariable(PerRayData, thePrd, rtPayload, );

rtDeclareVariable(optix::float3, varNormal, attribute NORMAL, );

RT_PROGRAM void closesthit()
{
  const float3 normal = optix::normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, varNormal));
  thePrd.radiance = normal * 0.5f + 0.5f;
}
```

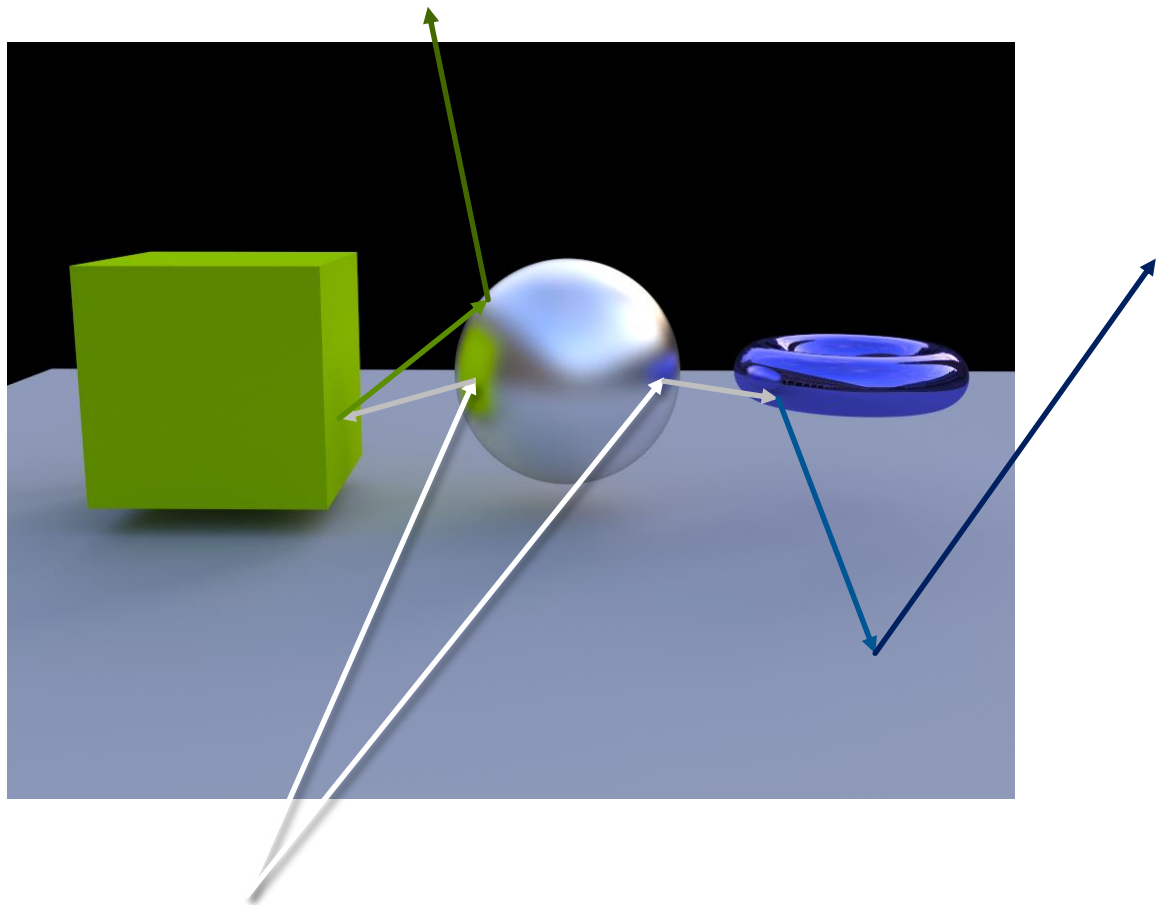
optixIntro_04



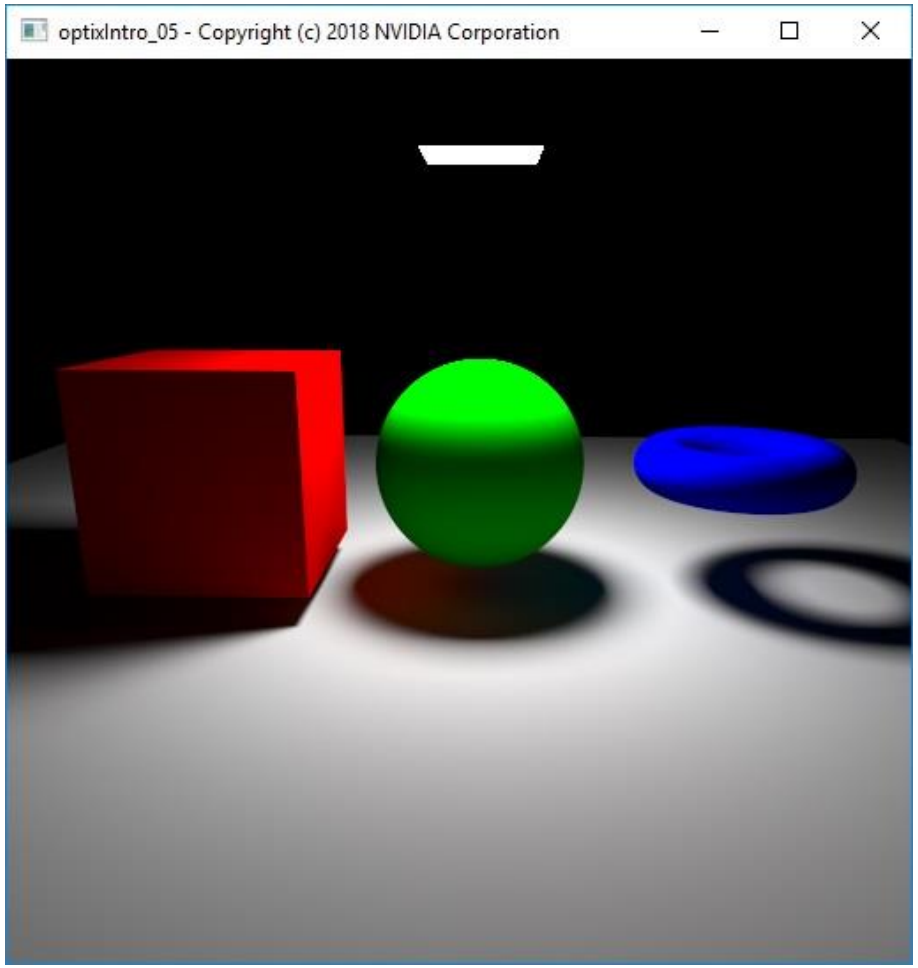


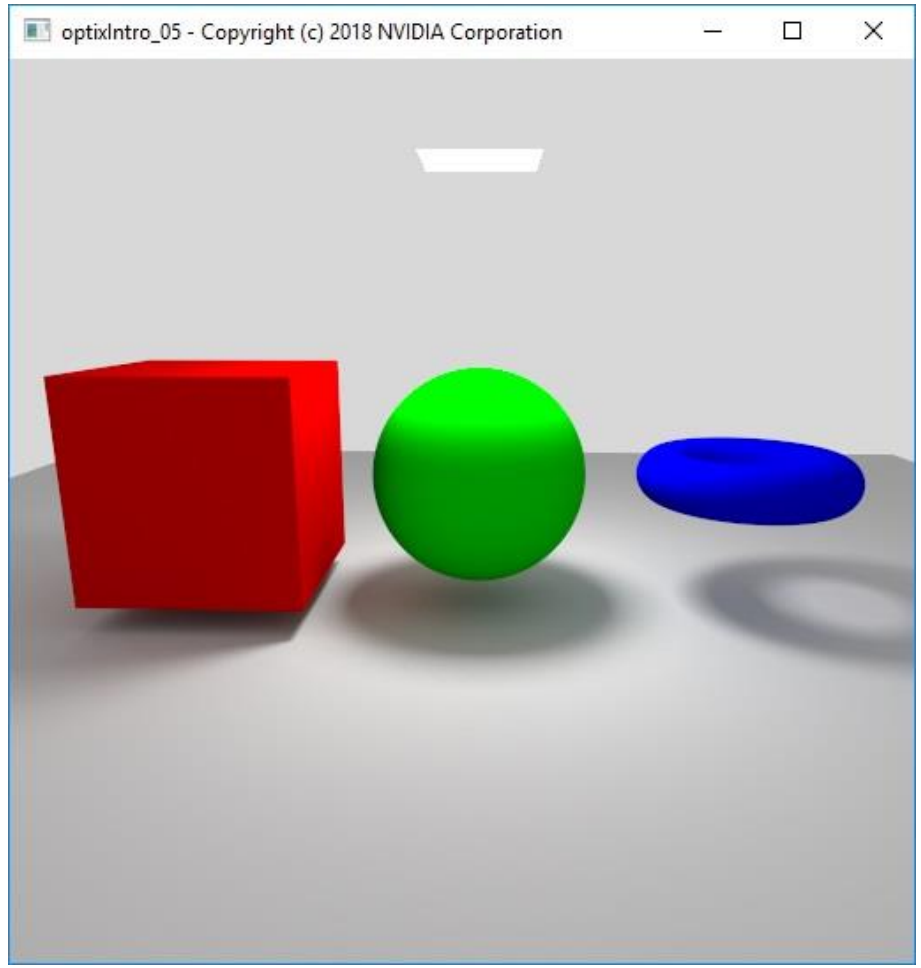
Integrator

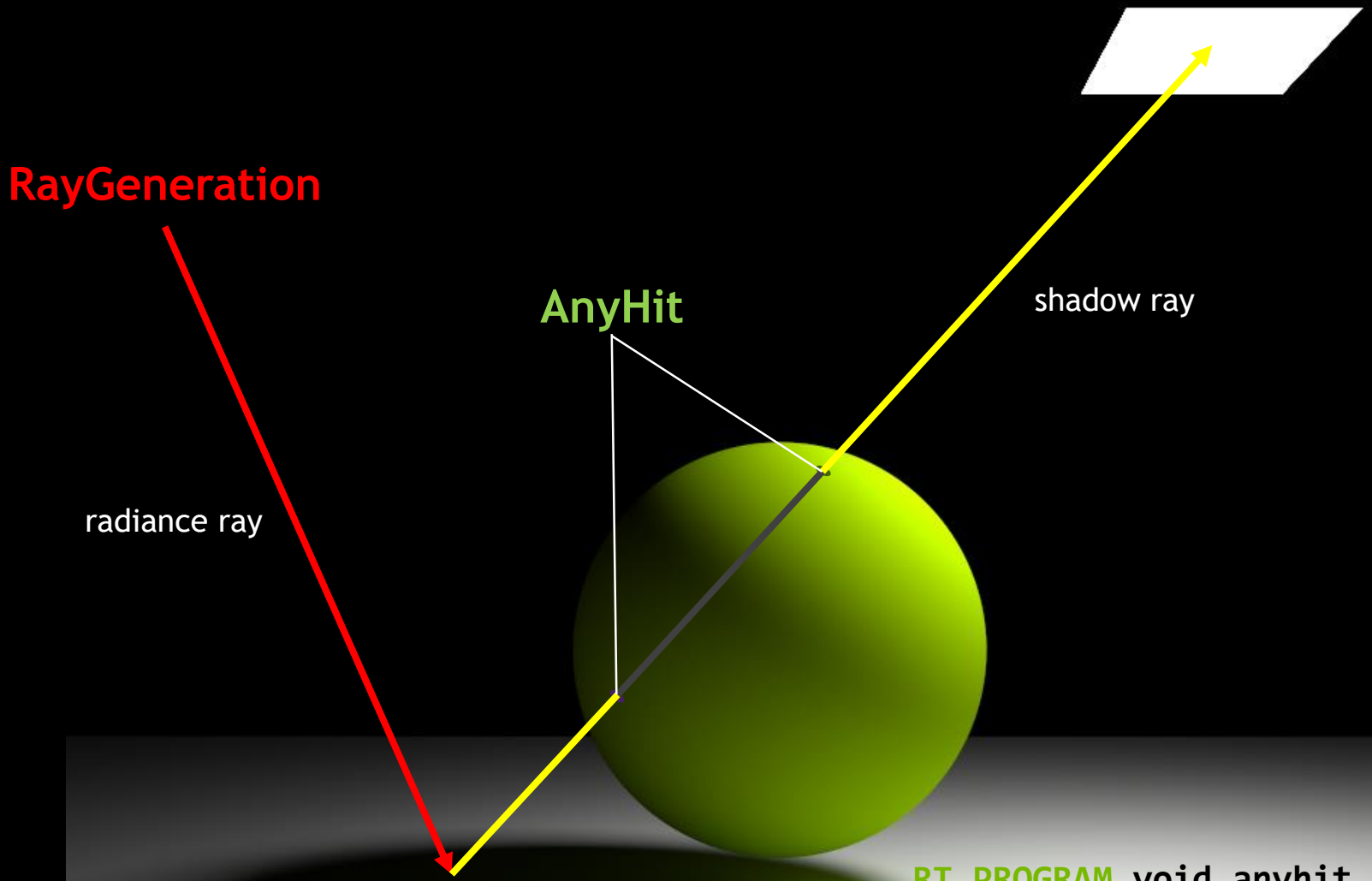
Unidirectional Path Tracer Throughput



optixIntro_05

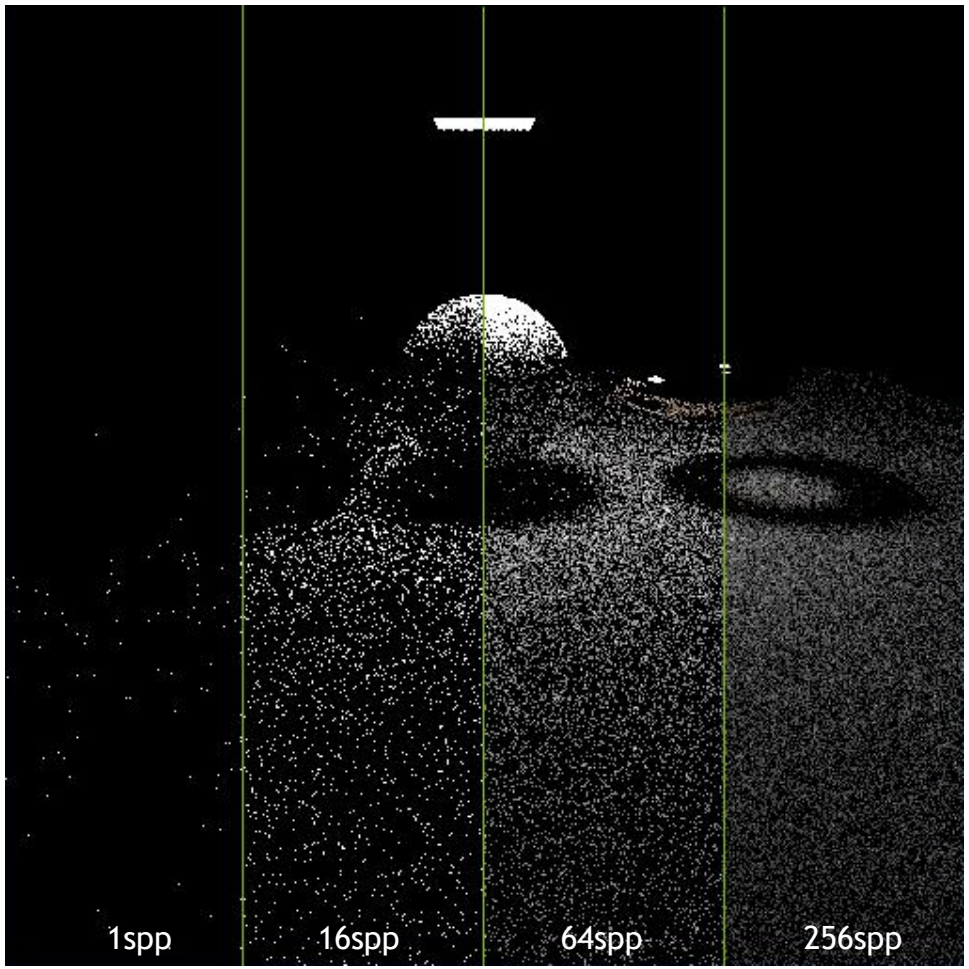


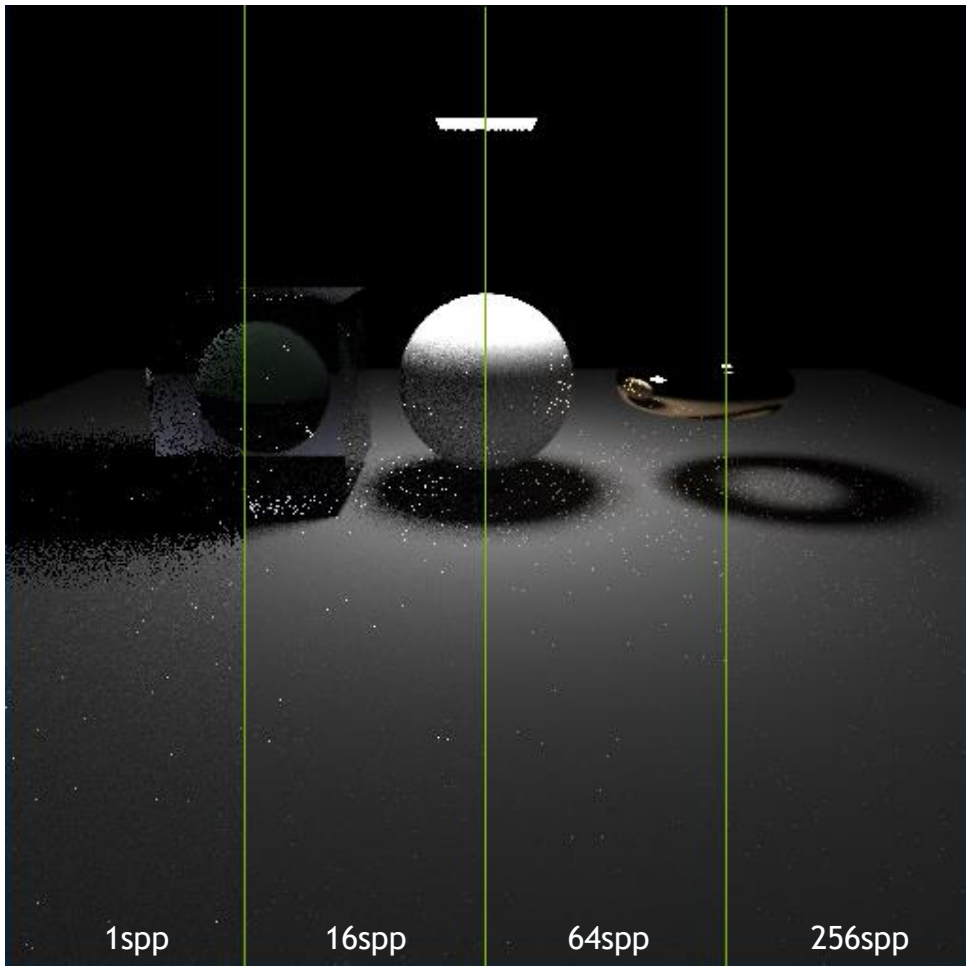




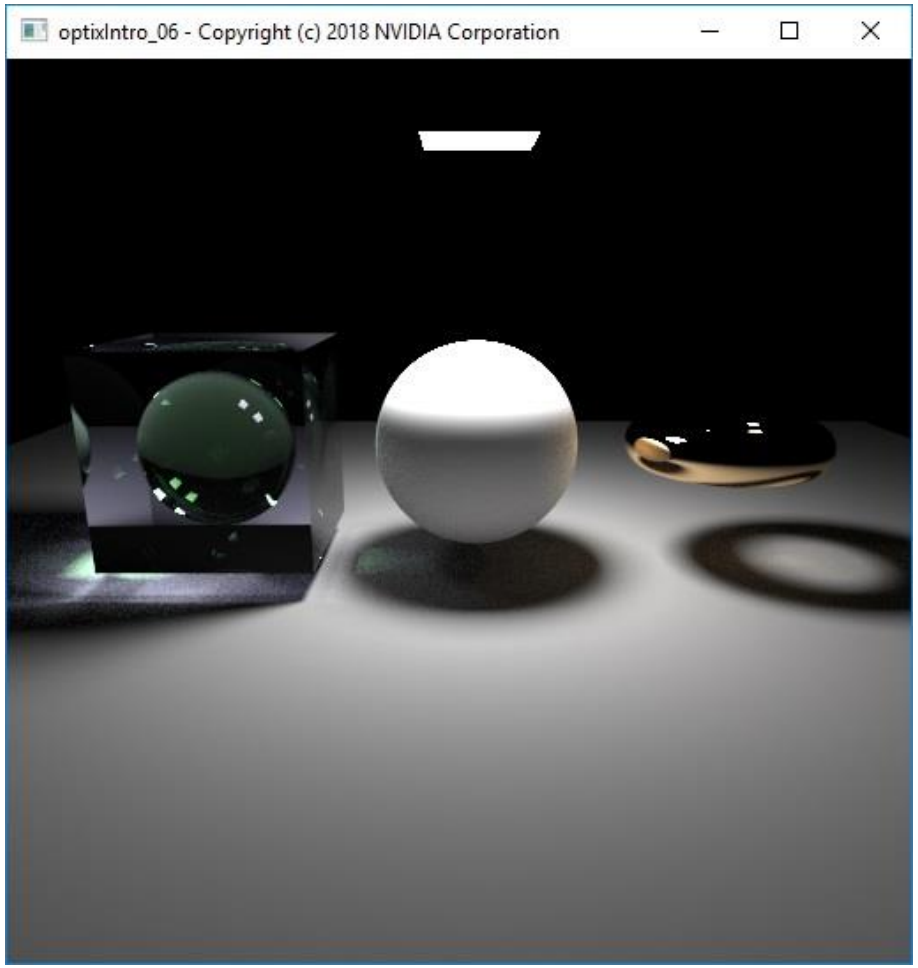

```
RT_PROGRAM void anyhit_shadow()
{
  thePrdShadow.visible = false;
  rtTerminateRay();
}
```

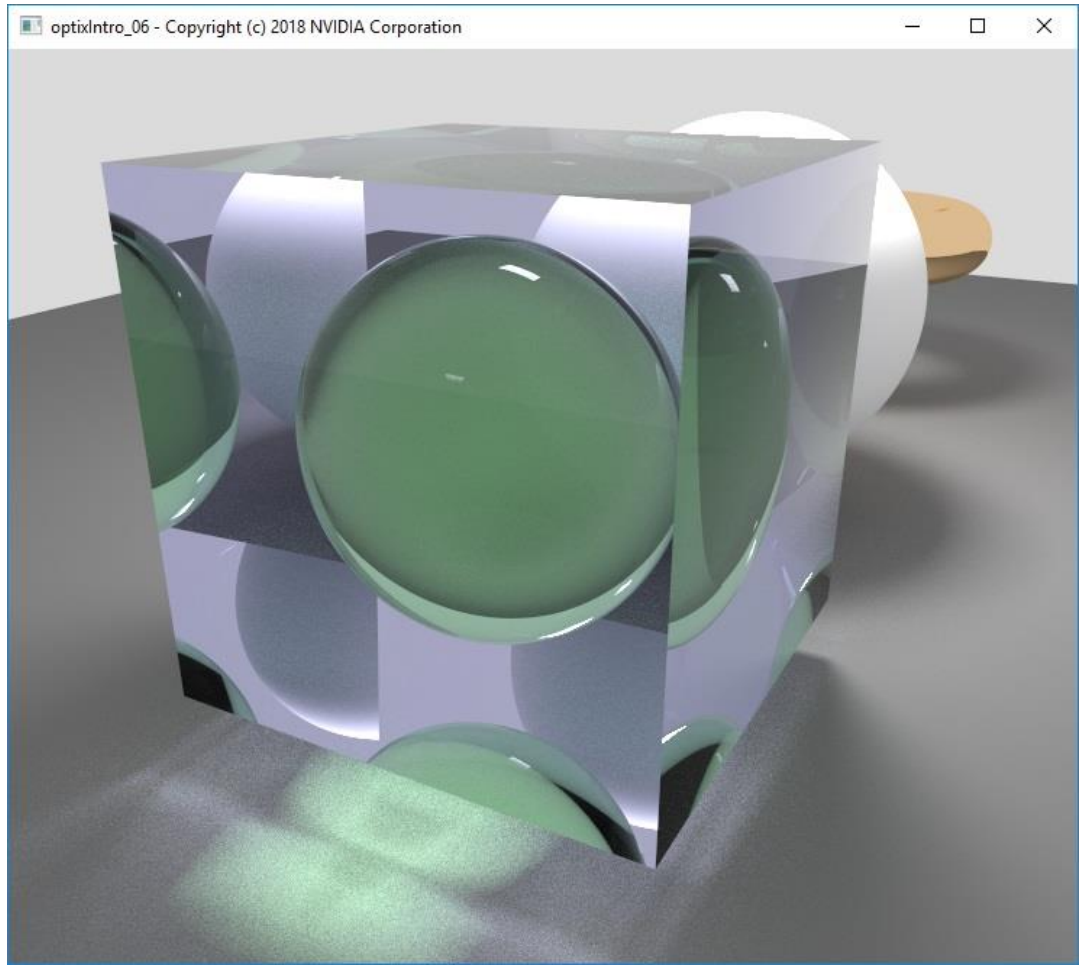
Next Event Estimation





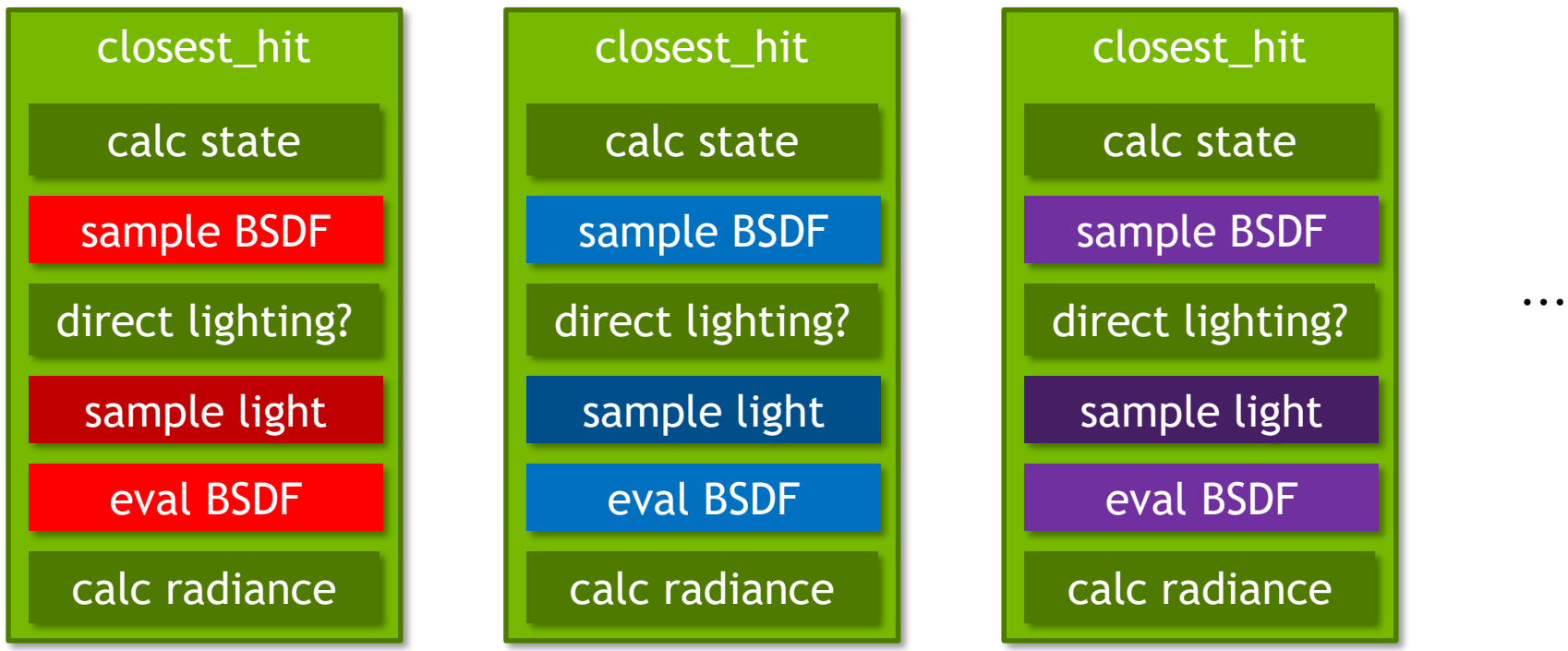
optixIntro_06





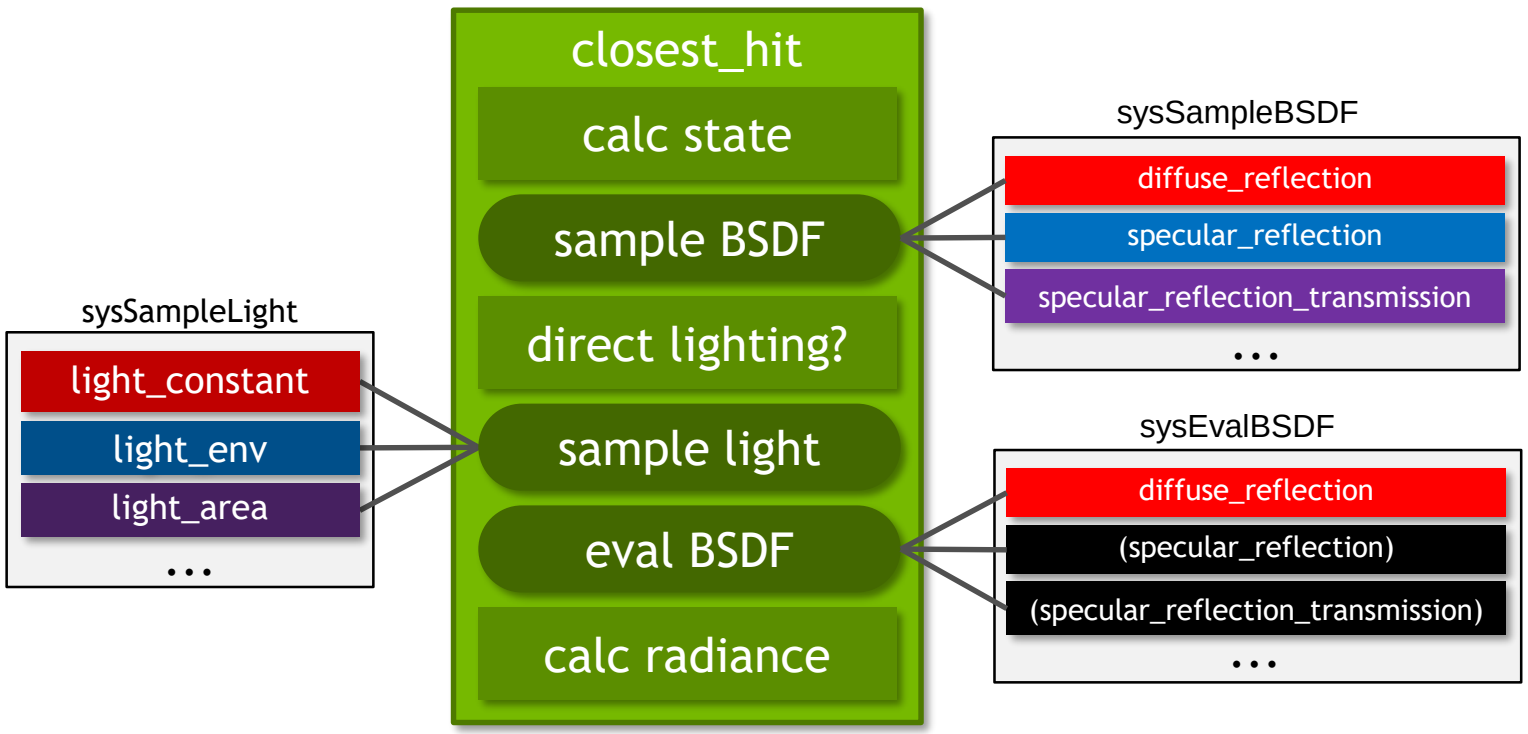
Adding more BSDF and Light Types

How much code do you really need?

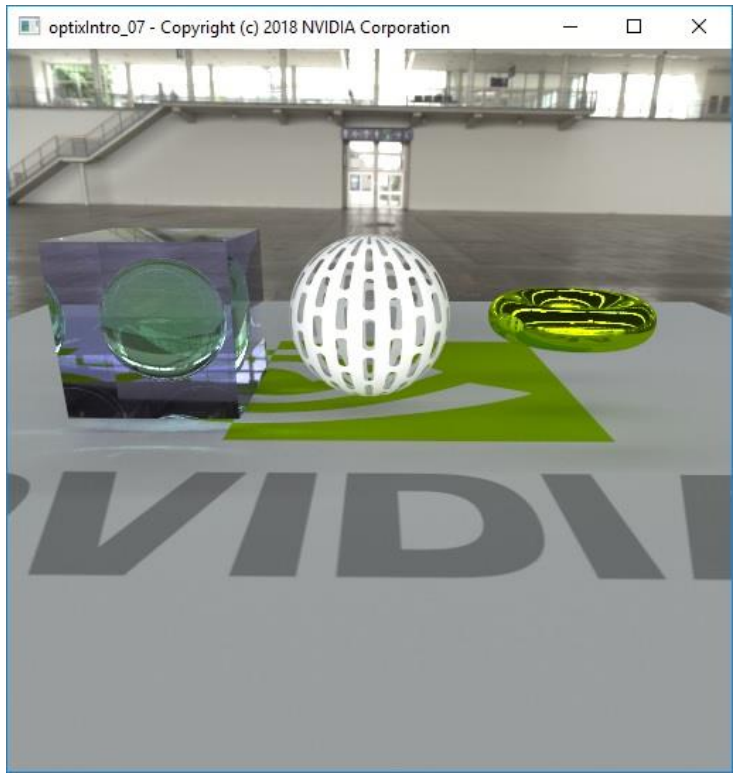


Buffers of Bindless Callable Program IDs

Implement fixed-function elements as bindless callable programs

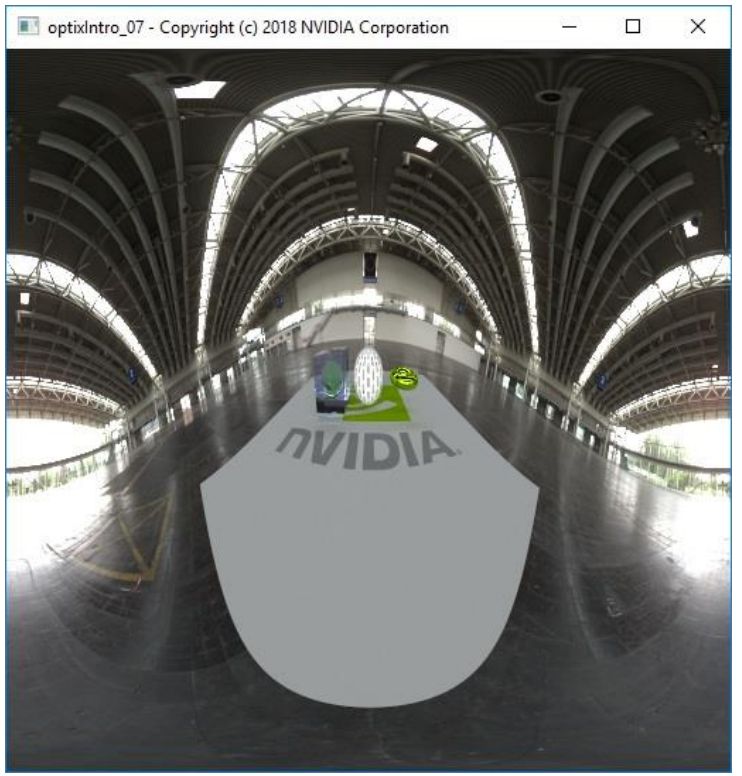


Lens Shaders



Pinhole

Fisheye



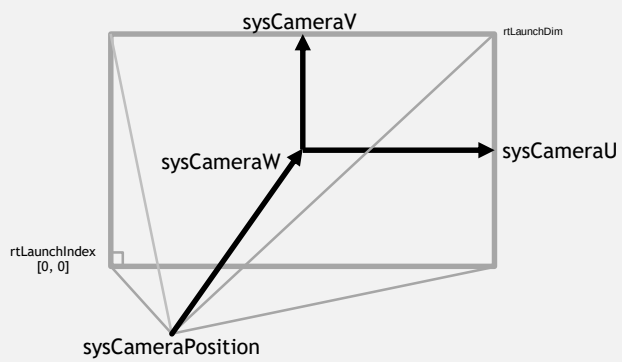
Spherical

Bindless Callable Program

Declaration of Buffer of IDs and Example

```
rtBuffer< rtCallableProgramId<return_typename(typename arg0, …, typename argN)> > name;
```

```
RT_CALLABLE_PROGRAM void lens_shader_pinhole(const float2 pixel,
                                             const float2 screen,
                                             const float2 sample,
                                             float3& origin,
                                             float3& direction)
{
  const float2 fragment = pixel + sample;
  const float2 ndc      = (fragment / screen) * 2.0f - 1.0f;

  origin    = sysCameraPosition;
  direction = optix::normalize(sysCameraU * ndc.x +
                               sysCameraV * ndc.y +
                               sysCameraW);
}
```

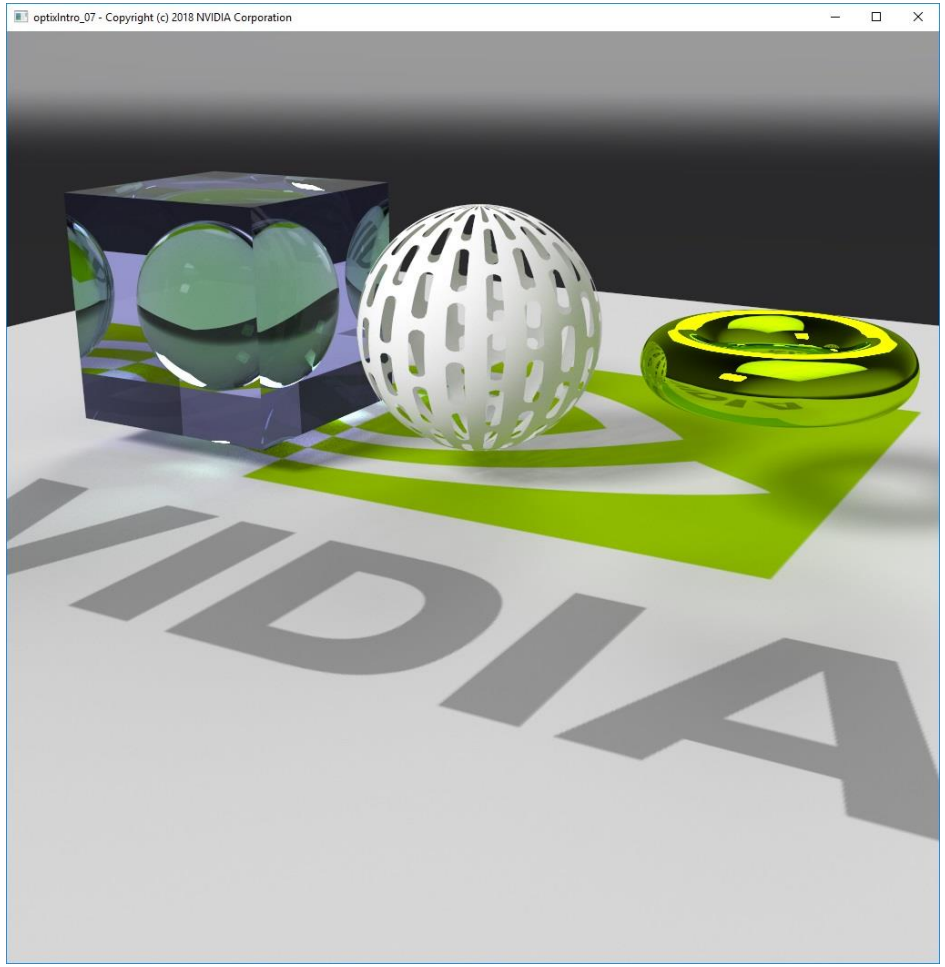


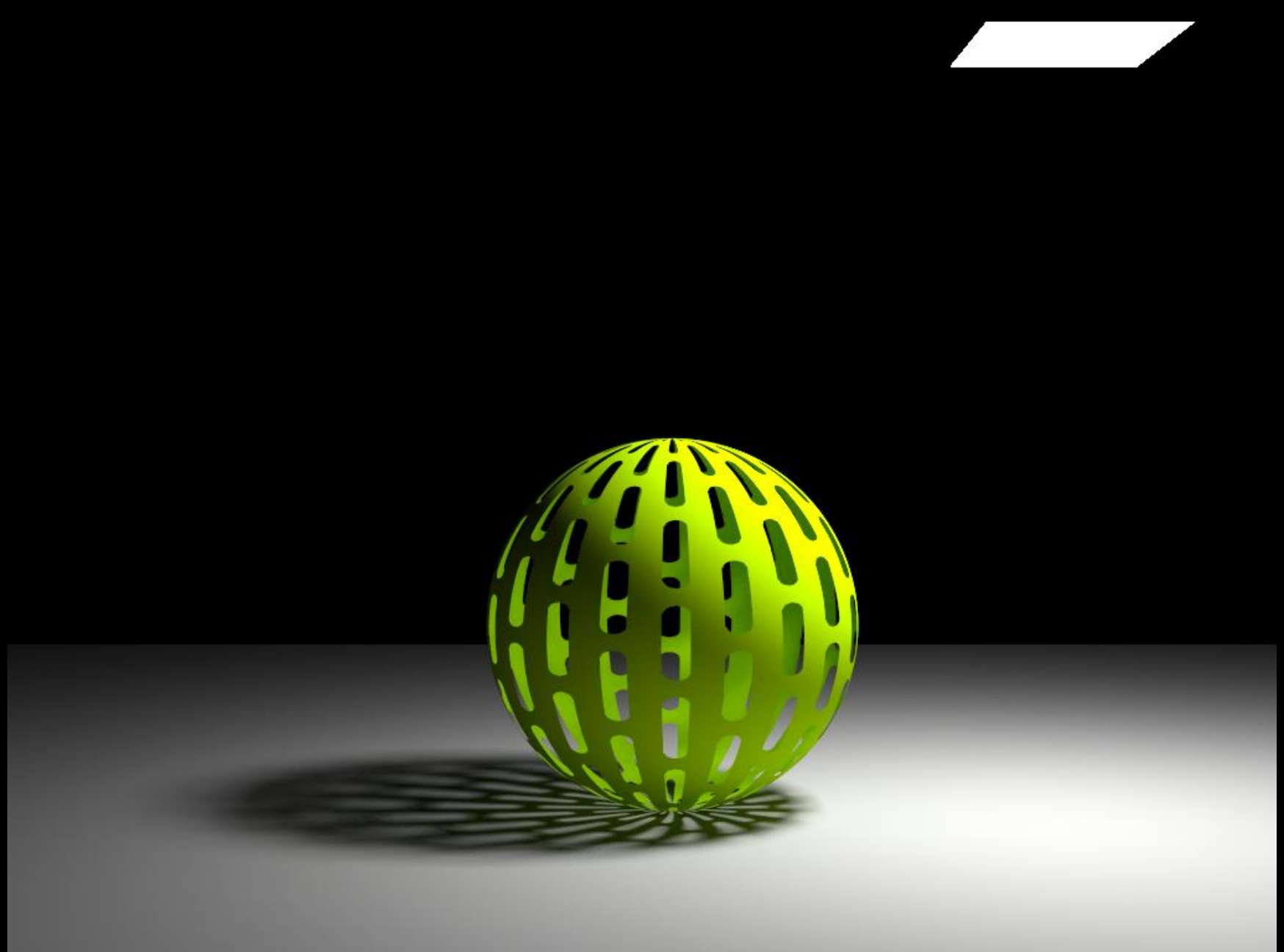
Variable Scopes

Organize your parameters

Program Type	Search Order			
ClosestHit AnyHit	Program	GeometryInstance	Material	Context
BoundingBox Intersection	Program	GeometryInstance	Geometry	Context
Raygeneration Exception Miss Bindless Callable Program	Program	Context		
Visit	Program	Node		

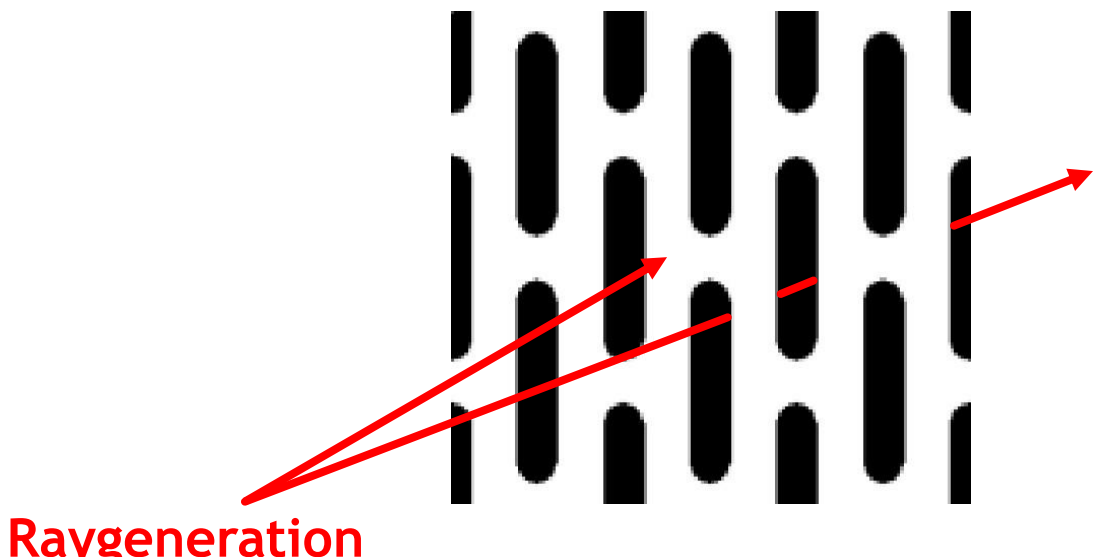
optixIntro_07





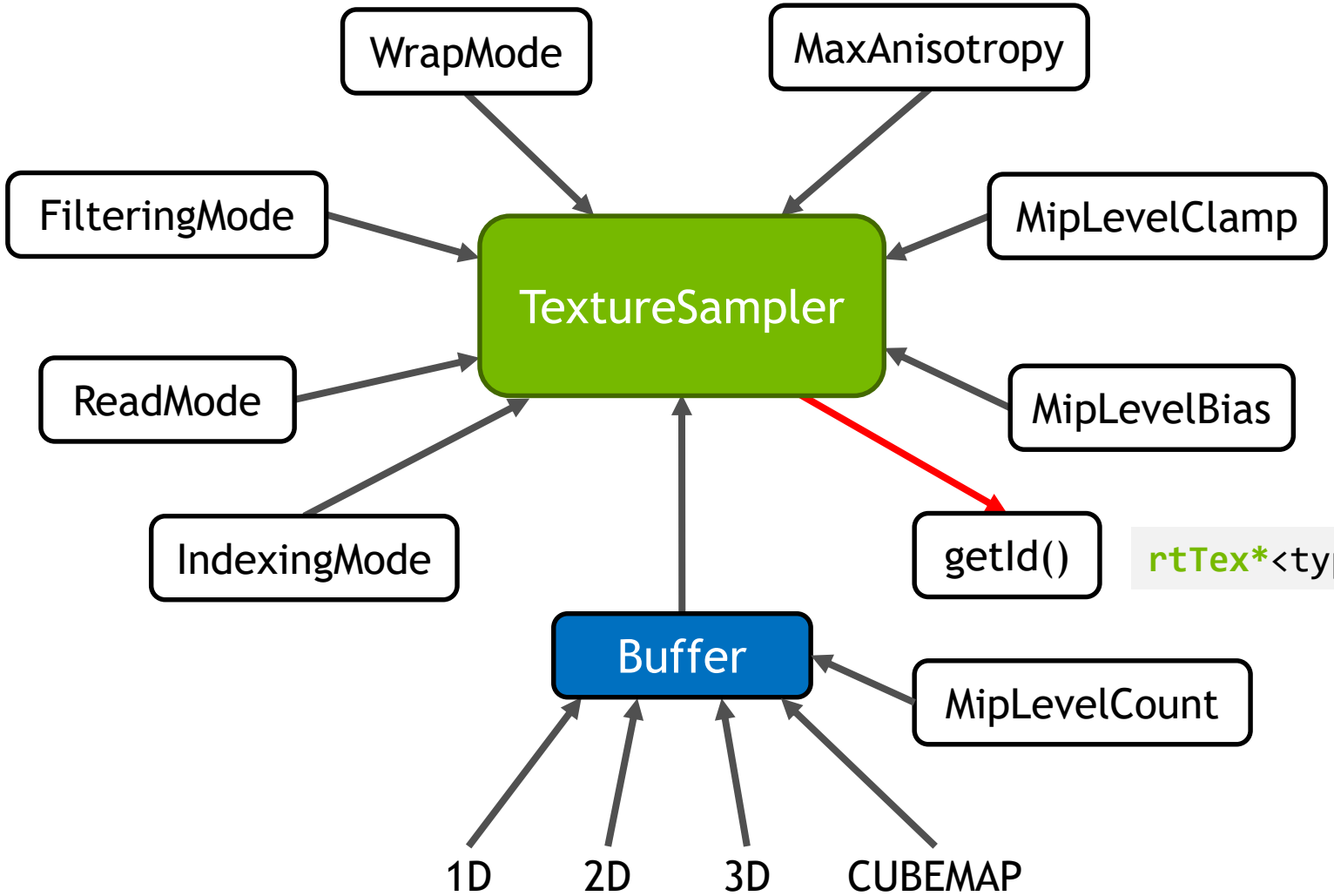
Cutout Opacity

Or how to use AnyHit programs



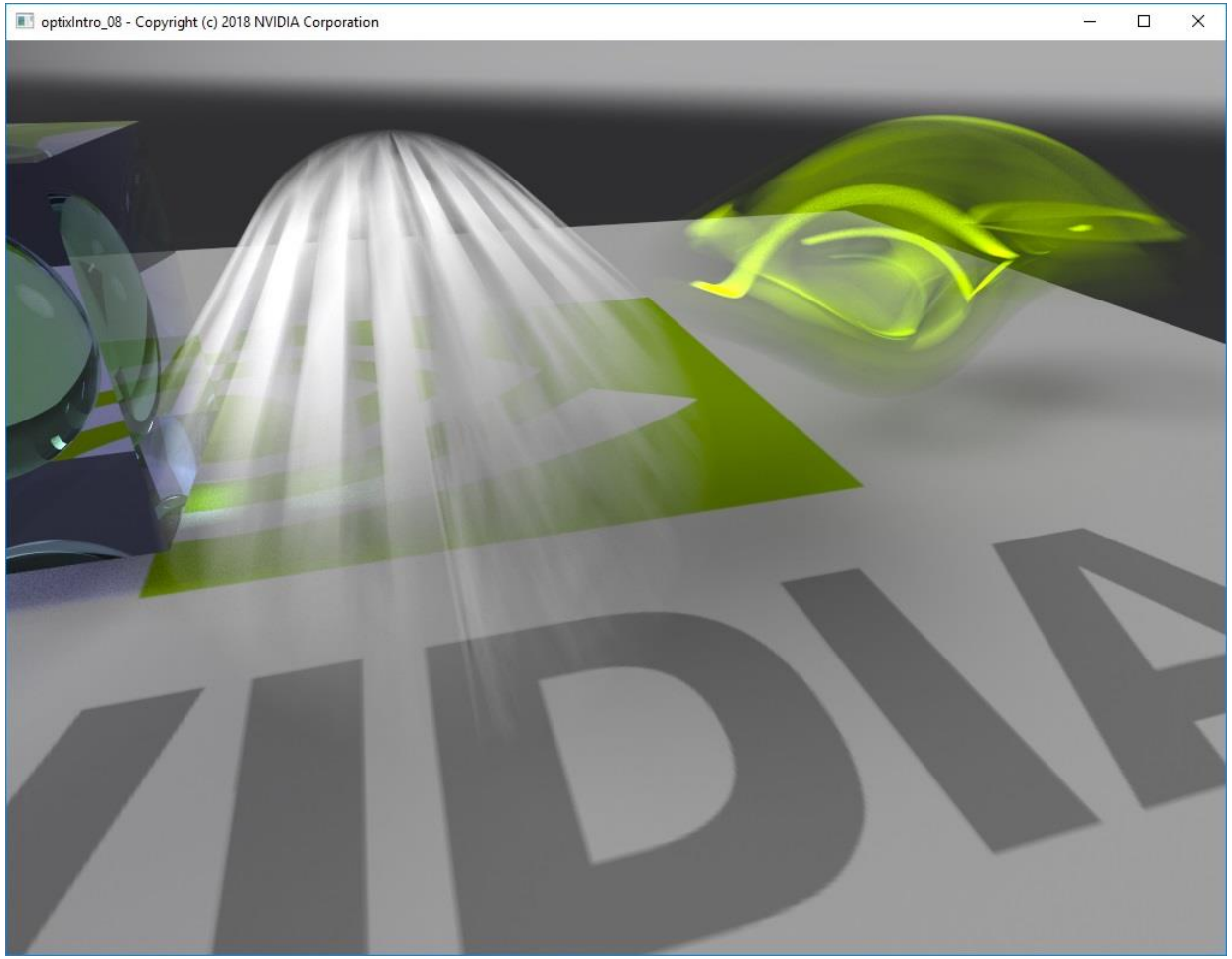

```
RT_PROGRAM void anyhit_cutout()
{
  if (getOpacity() < threshold)
    rtIgnoreIntersection();
}
```

```
RT_PROGRAM void anyhit_shadow_cutout()
{
  if (getOpacity() < threshold) {
    rtIgnoreIntersection();
  } else {
    thePrdShadow.visible = false;
    rtTerminateRay();
  }
}
```




```
rtTex*<typename>(id, u, v, …)
```

optixIntro_08



Motion Blur

Motion in Transform nodes

New functions for the Transform node

```
rtTransformSetMotionKeys(), rtTransformSetMotionRange(), rtTransformSetMotionBorderMode()
```

Two motion key types: A linearly interpolated 3x4 matrix or 16 elements from a Scale-Rotation-Translation (SRT) transformation

$$S = \begin{bmatrix} s_x & a & b & p_x \\ 0 & s_y & c & p_y \\ 0 & 0 & s_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Motion Blur

Motion in Geometry Nodes (Morphing)

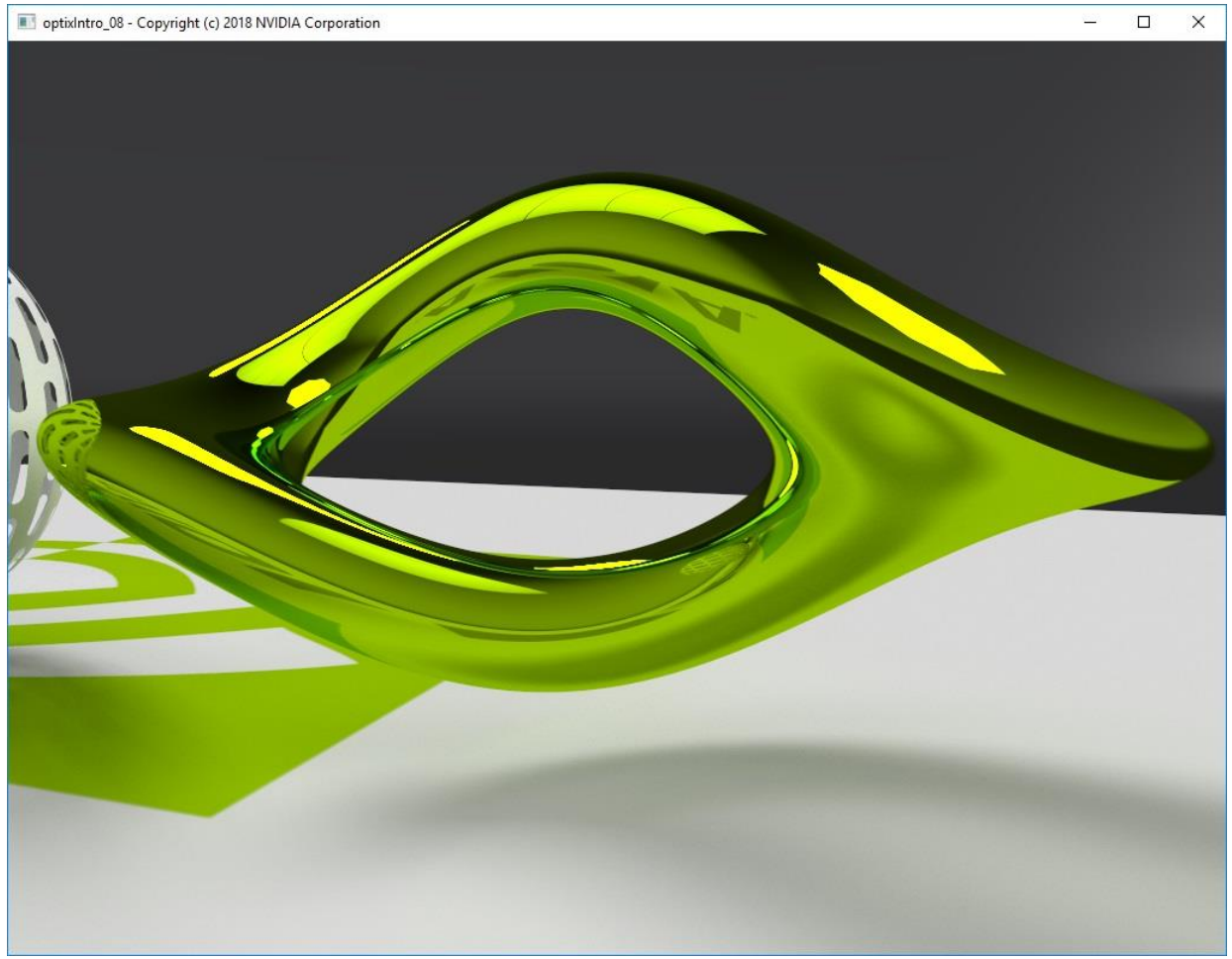
New BoundingBox program function signature

```
RT_PROGRAM void boundingbox_motionblur(int prim_index, int motion_index, float result[6]);
```

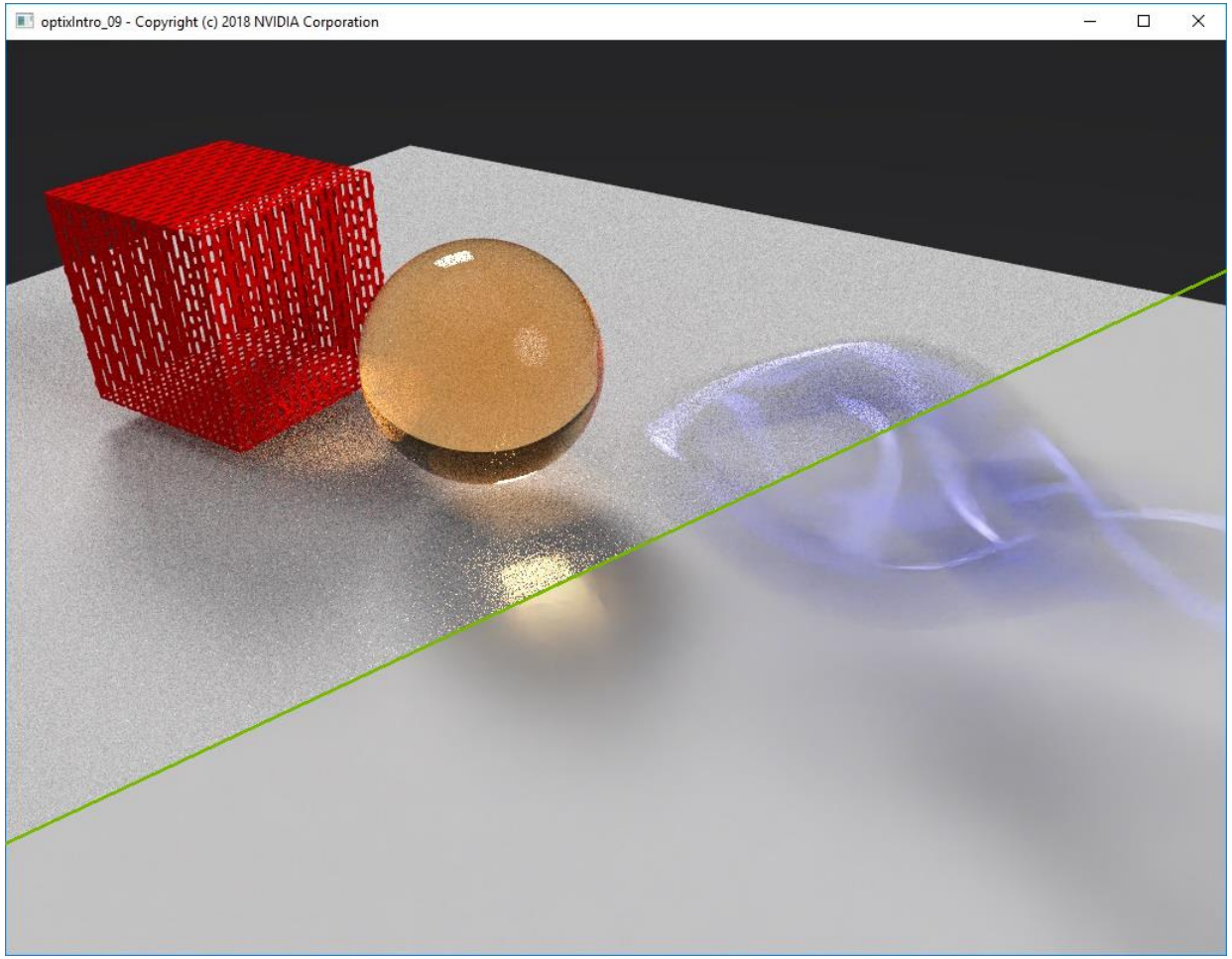
New functions for the Geometry node

```
rtGeometrySetMotionSteps(), rtGeometrySetMotionRange(), rtGeometrySetMotionBorderMode()
```

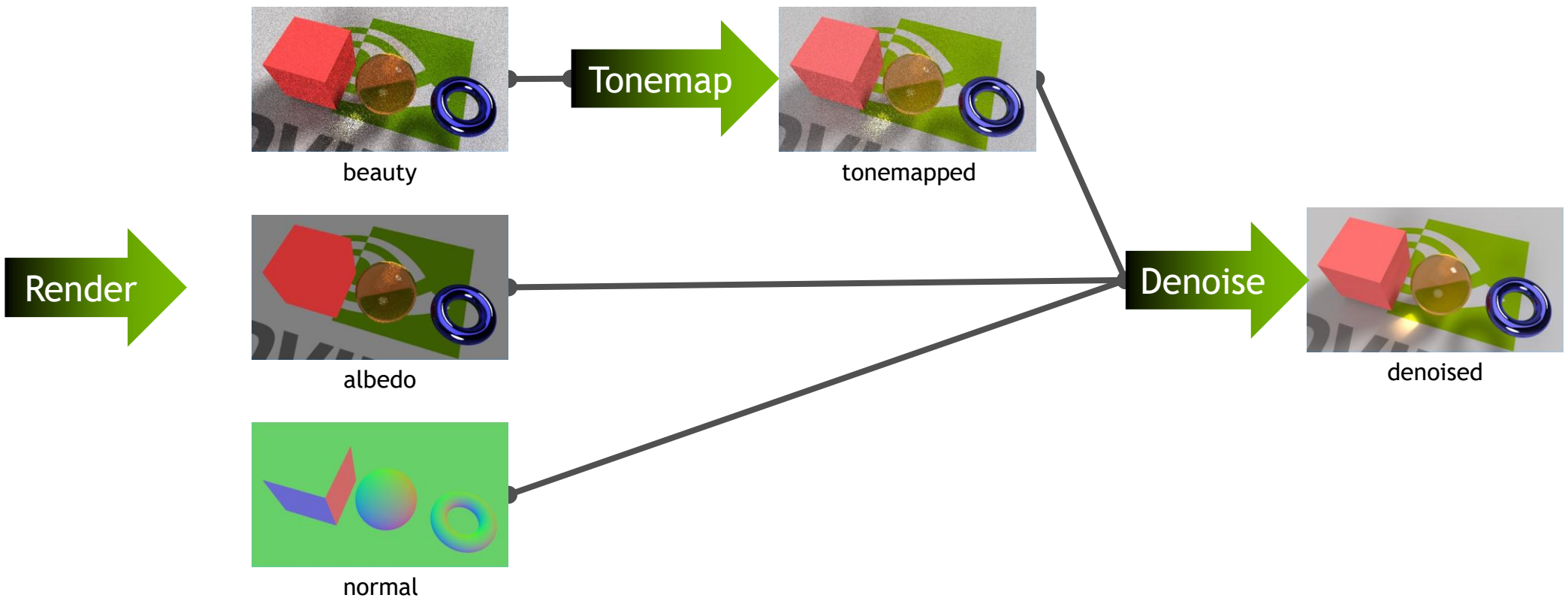
Rolling shutters



optixIntro_09

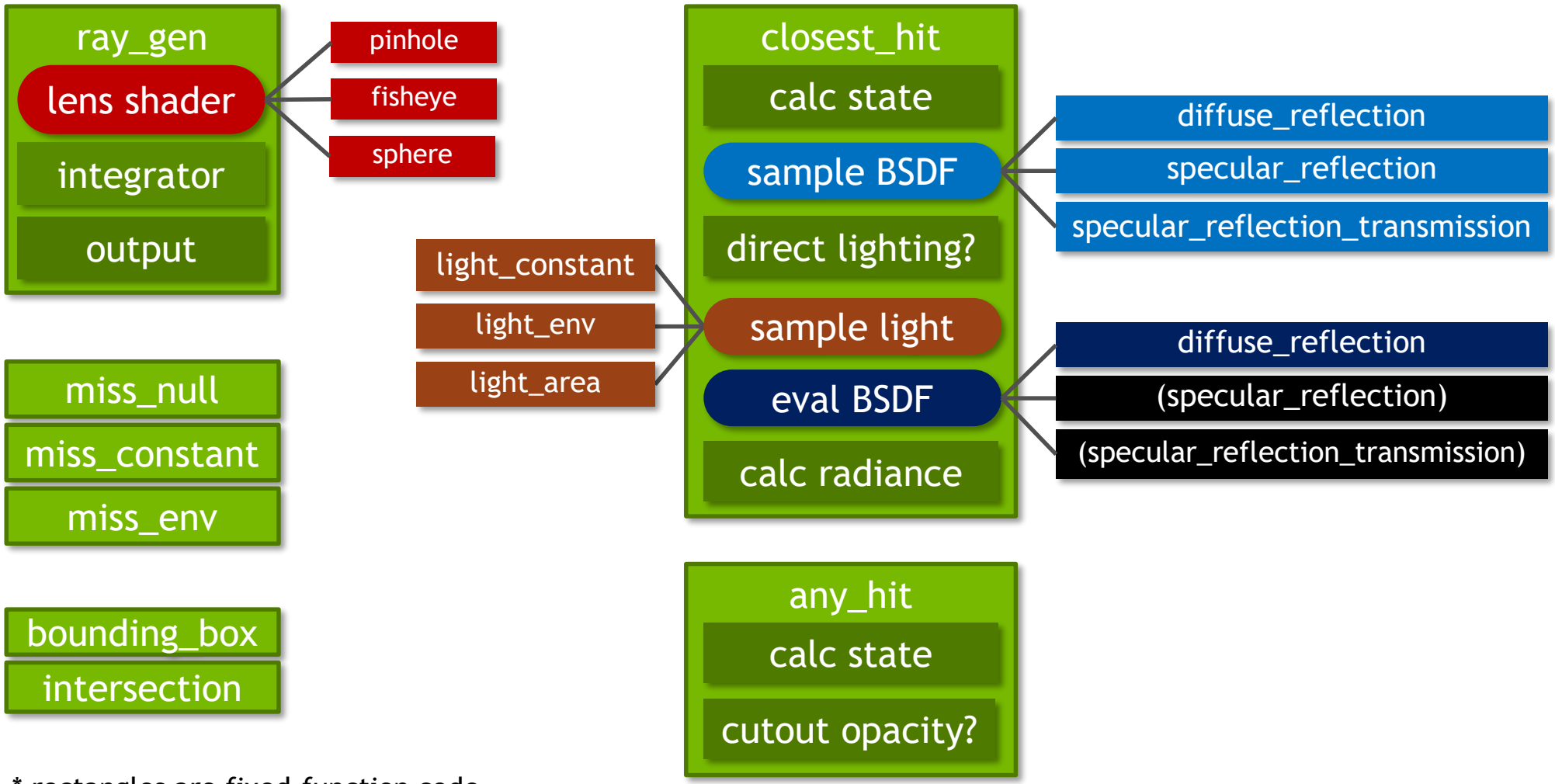


Post-Processing Pipeline



Post-Processing Setup

```
PostprocessingStage tonemapper = context->createBuiltinPostProcessingStage("TonemapperSimple");

tonemapper->declareVariable("input_buffer")->set(bufferOutput); // from the renderer
tonemapper->declareVariable("output_buffer")->set(bufferTonemapped);
tonemapper->declareVariable("exposure")->setFloat(1.0f);
tonemapper->declareVariable("gamma")->setFloat(2.2f);

PostprocessingStage denoiser = context->createBuiltinPostProcessingStage("DLDenoiser");

denoiser->declareVariable("input_buffer")->set(bufferTonemapped);
denoiser->declareVariable("input_albedo_buffer")->set(bufferAlbedo); // optional
denoiser->declareVariable("input_normal_buffer")->set(bufferNormal); // optional
denoiser->declareVariable("output_buffer")->set(bufferDenoised);

CommandList commandList = context->createCommandList();

commandList->appendLaunch(0, w, h); // Launch raygeneration at entry point 0 => Render
commandList->appendPostprocessingStage(tonemapper, w, h);
commandList->appendPostprocessingStage(denoiser, w, h);
commandList->finalize();

commandList->execute(); // Result in bufferDenoised.
```



* rectangles are fixed-function code
* round rectangles are bindless callable programs

Takeaway

OptiX is a high-level GPU ray casting SDK with a flexible programming model which allows to concentrate on the algorithm during developement

State-of-the-art acceleration structures and core features get added or improved with each new version

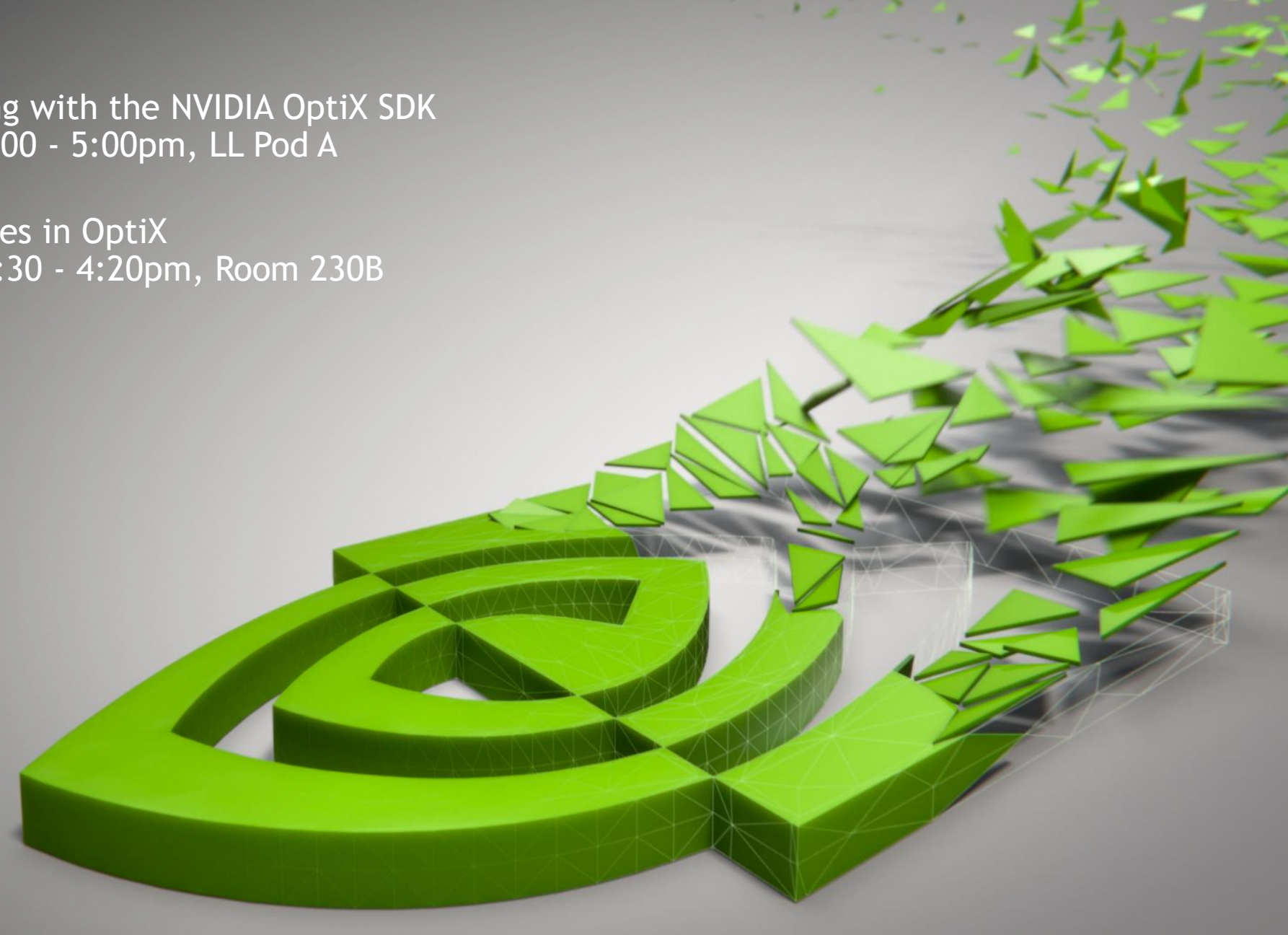
An easily extendable architecture for a global illumination path tracer using OptiX features available in version 5.0.1 has been presented

The nine OptiX examples accompanying this tutorial are going to be available on github

CE8105: Ray Tracing with the NVIDIA OptiX SDK
Monday, Mar 26, 4:00 - 5:00pm, LL Pod A

S8519: New Features in OptiX
Tuesday, Mar 27, 3:30 - 4:20pm, Room 230B

NVIDIA



BACKUP SLIDES

```
m_context = optix::Context::create();

m_context->setEntryPointCount(1);
m_context->setRayTypeCount(0);
m_context->setStackSize(m_stackSize);

m_bufferOutput = m_context->createBuffer(RT_BUFFER_OUTPUT, RT_FORMAT_FLOAT4, m_width, m_height);

m_context["sysOutputBuffer"]->set(m_bufferOutput);

optix::Program prgRaygen = m_context->createProgramFromPTXFile(
                                        ptxPath("raygeneration.cu"), "raygeneration");
m_context->setRayGenerationProgram(0, prgRaygen); // per entry point

optix::Program prgException = m_context->createProgramFromPTXFile(
                                           ptxPath("exception.cu"), "exception");
m_context->setExceptionProgram(0, prgException); // per entry point

m_context["sysColorBackground"]->setFloat(0.0f, 1.0f, 0.0f); // green

m_context->launch(0, m_width, m_height); // ==> uint2 rtLaunchDim

const void* data = m_bufferOutput->map(0, RT_BUFFER_MAP_READ);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, (GLsizei) m_width, (GLsizei) m_height, 0,
             GL_RGBA, GL_FLOAT, data);
m_bufferOutput->unmap();
```

rtTrace

```
rtBuffer<float4, 2> sysOutputBuffer; // RGBA32F

rtDeclareVariable(rtObject, sysTopObject, , );

RT_PROGRAM void raygeneration()
{
  PerRayData prd;

  prd.radiance = make_float3(0.0f);

  ... // Camera implementations calculate origin and direction here.

  optix::Ray ray = optix::make_Ray(origin, direction, 0, 0.0f, RT_DEFAULT_MAX);

  rtTrace(sysTopObject, ray, prd);

  sysOutputBuffer[theLaunchIndex] = make_float4(prd.radiance, 1.0f);
}
```

Miss Program

```
rtDeclareVariable(optix::Ray, theRay, rtCurrentRay, );
rtDeclareVariable(PerRayData, thePrd, rtPayload, );

rtDeclareVariable(float3, sysColorBottom, , );
rtDeclareVariable(float3, sysColorTop, , );

RT_PROGRAM void miss_gradient()
{
  const float t   = theRay.direction.y * 0.5f + 0.5f;
  thePrd.radiance = optix::lerp(sysColorBottom, sysColorTop, t);
}
```

```
rtDeclareVariable(float3, varGeoNormal, attribute GEO_NORMAL, );
rtDeclareVariable(float3, varNormal,    attribute NORMAL, );
rtDeclareVariable(optix::Ray, theRay, rtCurrentRay, );
rtDeclareVariable(PerRayData, thePrd, rtPayload, );

RT_PROGRAM void closesthit()
{
  float3 geoNormal = optix::normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, varGeoNormal));
  float3 normal    = optix::normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, varNormal));

  thePrd.pos = theRay.origin + theRay.direction * theIntersectionDistance;

  thePrd.flags |= (0.0f <= optix::dot(thePrd.wo, geoNormal)) ? FLAG_FRONTFACE : 0;
  if ((thePrd.flags & FLAG_FRONTFACE) == 0) {
    geoNormal = -geoNormal;
    normal    = -normal;
  }
  // Lambert sampling implementation:
  thePrd.radiance = make_float3(0.0f); // No emission, no direct lighting.
  unitSquareToCosineHemisphere(rng2(thePrd.seed), normal, thePrd.wi, thePrd.pdf);
  if (thePrd.pdf <= 0.0f || optix::dot(thePrd.wi, geoNormal) <= 0.0f) {
    thePrd.flags |= FLAG_TERMINATE;
    return;
  }
  MaterialParameter parameters = sysMaterialParameters[parMaterialIndex];
  thePrd.f_over_pdf = parameters.albedo * (M_1_PIf * optix::dot(thePrd.wi, normal) / thePrd.pdf);
}
```

```
rtDeclareVariable(PerRayData_shadow, thePrdShadow, rtPayload, );

RT_PROGRAM void anyhit_shadow()
{
  thePrdShadow.visible = false;
  rtTerminateRay();
}
```

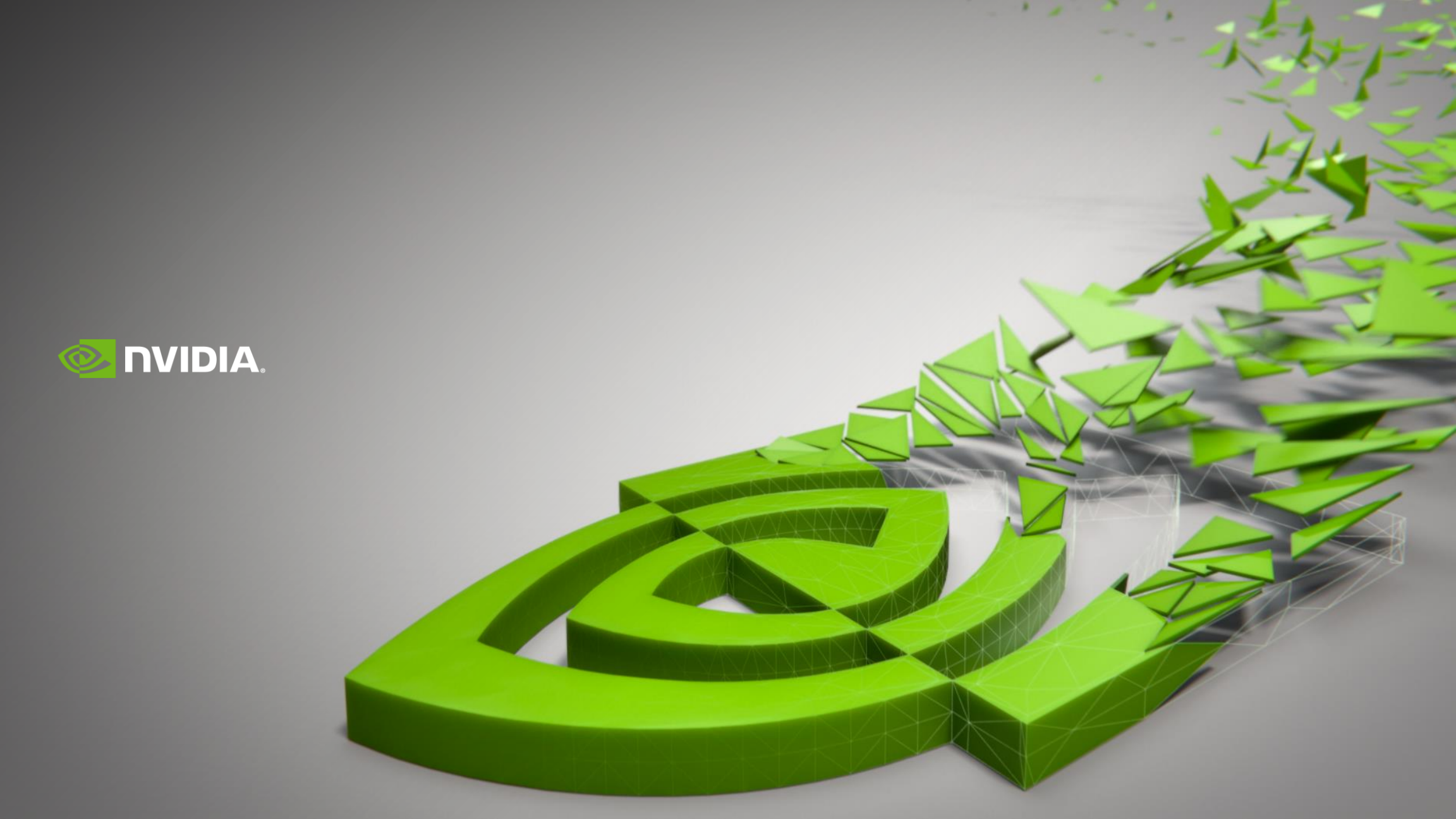
```
RT_FUNCTION void integrator(PerRayData& prd, float3& radiance)
{
  radiance = make_float3(0.0f);
  float3 throughput = make_float3(1.0f);

  int depth = 0; // Primary ray is path segment 0.
  while (depth < sysPathLength)
  {
    prd.wo    = -prd.wi;
    prd.flags = 0;

    optix::Ray ray = optix::make_Ray(prd.pos, prd.wi, 0, sysSceneEpsilon, RT_DEFAULT_MAX);
    rtTrace(sysTopObject, ray, prd);

    radiance += throughput * prd.radiance;

    if ((prd.flags & FLAG_TERMINATE) || prd.pdf <= 0.0f || isNull(prd.f_over_pdf))
    {
      break;
    }

    throughput *= prd.f_over_pdf; // == f * (fabsf(optix::dot(wi, normal)) / pdf);

    ++depth;
  }
}
```

```
#include <optix.h>
#include <optixu/optixu_math_namespace.h>

#include "material_parameter.h"
#include "per_ray_data.h"

RT_CALLABLE_PROGRAM void sample_bsdf_specular_reflection(MaterialParameter const& parameters,
                                                         State const& state,
                                                         PerRayData& prd)
{
  prd.wi = optix::reflect(-prd.wo, state.normal);
  if (optix::dot(prd.wi, state.geoNormal) <= 0.0f)
  {
    prd.flags |= FLAG_TERMINATE;
    return;
  }
  prd.f_over_pdf = parameters.albedo;
  prd.pdf        = 1.0f;
}

RT_CALLABLE_PROGRAM float4 eval_bsdf_specular_reflection(MaterialParameter const& parameters,
                                                         State const& state,
                                                         PerRayData const& prd,
                                                         float3 const& wiL)
{
  return make_float4(0.0f);
}
```

```
rtDeclareVariable(PerRayData_shadow, thePrdShadow, rtPayload, );

rtBuffer<MaterialParameter> sysMaterialParameters;
rtDeclareVariable(int, parMaterialIndex, , );

rtDeclareVariable(optix::float3, varTexCoord, attribute TEXCOORD, );

RT_PROGRAM void anyhit_shadow_cutout()
{
  float opacity = 1.0f;

  const int id = sysMaterialParameters[parMaterialIndex].cutoutID; // bindless texture ID
  if (id != RT_TEXTURE_ID_NULL) {
    opacity = intensity(make_float3(optix::rtTex2D<float4>(id, varTexCoord.x, varTexCoord.y)));
  }

  // Stochastic coutout opacity, think Monte Carlo!
  if (opacity < 1.0f && opacity <= rng(thePrdShadow.seed)) {
    rtIgnoreIntersection();
  } else {
    thePrdShadow.visible = false;
    rtTerminateRay();
  }
}
```

TextureSampler

```
optix::Buffer buffer = context->createBuffer(RT_BUFFER_INPUT, RT_FORMAT_UNSIGNED_BYTE4, w, h);

buffer->setMipLevelCount(1);

void *dst = buffer->map(0, RT_BUFFER_MAP_WRITE_DISCARD);
memcpy(dst, texels, w * h * getElementSize(RT_FORMAT_UNSIGNED_BYTE4));
buffer->unmap(0);

optix::TextureSampler sampler = context->createTextureSampler();

sampler->setWrapMode(0, RT_WRAP_REPEAT);
sampler->setWrapMode(1, RT_WRAP_REPEAT);
sampler->setWrapMode(2, RT_WRAP_REPEAT);
sampler->setFilteringModes(RT_FILTER_LINEAR, RT_FILTER_LINEAR, RT_FILTER_NONE);
sampler->setIndexingMode(RT_TEXTURE_INDEX_NORMALIZED_COORDINATES;
sampler->setReadMode(RT_TEXTURE_READ_NORMALIZED_FLOAT);
sampler->setMaxAnisotropy(1.0f);
sampler->setBuffer(buffer);
```


NVIDIA