NVIDIA® OptiX™ Ray Tracing Engine

Programming Guide

Version 2.5

1/27/2012

Table of Contents

Chapter 1. Introduction

OptiX Overview

GPUs are best at exploiting very high degrees of parallelism, and ray tracing fits that requirement perfectly. However, typical ray tracing algorithms can be highly irregular, which poses serious challenges for anyone trying to exploit the full raw computational potential of a GPU. The NVIDIA OptiX ray tracing engine and API address those challenges and provide a framework for harnessing the enormous computational power of both current- and future-generation graphics hardware to incorporate ray tracing into interactive applications. By using OptiX together with NVIDIA's CUDA™ architecture, interactive ray tracing is finally feasible for developers without a Ph.D. in computer graphics and a team of ray tracing engineers.

OptiX is not itself a raytracer. Instead, it is a scalable framework for building ray tracing based applications. The OptiX engine is composed of two symbiotic parts: 1) a host-based API that defines ray-tracing based data structures, and 2) a CUDA C-based programming system that can produce new rays, intersect rays with surfaces, and respond to those intersections. Together, these two pieces provide low-level support for "raw ray tracing". This allows user-written applications that use ray tracing for graphics, collision detection, sound propagation, visibility determination, etc.

Motivation

By abstracting the execution model of a generic ray tracer, OptiX makes it easier to assemble a ray tracing system, leveraging custom-built algorithms for object traversal, shader dispatch and memory management. Furthermore, the resulting system will be able to take advantage of future evolution in GPU hardware and OptiX SDK releases – similar to the manner that OpenGL and Direct3D provide an abstraction for the rasterization pipeline.

Wherever possible, the OptiX engine avoids specification of ray tracing behaviors and instead provides mechanisms to execute user-provided CUDA C code to implement shading (including recursive rays), camera models, and even color representations. Consequently, the OptiX engine can be used for Whitted-style ray tracing, path tracing, collision detection, photon mapping, or any other ray tracing-based algorithm. It is designed to operate either standalone or in conjunction with an OpenGL or DirectX application for hybrid ray tracing-rasterization applications.

Programming model

At the core of OptiX is a simple but powerful abstract model of a ray tracer. This ray tracer employs user-provided programs to control the initiation of rays, intersection of rays with surfaces, shading with materials, and spawning of new rays. Rays carry user-specified payloads that describe per-ray variables such as color, recursion depth, importance, or other attributes. Developers provide these functions to OptiX in the form of CUDA C-based functions. Because ray tracing is an inherently recursive algorithm, OptiX allows user programs to recursively spawn new rays, and the internal execution mechanism manages all the details of a recursion stack. OptiX also provides flexible dynamic function dispatch and a sophisticated variable inheritance mechanism so that ray tracing systems can be written very generically and compactly.

Ray tracing basics

"Ray tracing" is an overloaded term whose meaning can depend on context. Sometimes it refers to the computation of the intersection points between a 3D line and a set of 3D objects such as spheres. Sometimes it refers to a specific algorithm such as Whitted's method of generating pictures or the oil exploration industry's algorithm for simulating ground wave propagation. Other times it refers to a family of algorithms that include Whitted's algorithm along with others such as distribution ray tracing. OptiX is a ray tracing engine in the first sense of the word: it allows the user to intersect rays and 3D objects. As such it can be used to build programs that fit the other use of "ray tracing" such as Whitted's algorithm. In addition OptiX provides the ability for users to write their own programs to generate rays and to define behavior for when rays hit objects.

For graphics, ray tracing was originally proposed by Arthur Appel in 1968 for rendering solid objects. In 1980, Turner Whitted pursued the idea further by introducing recursion to enable reflective and refractive effects. Subsequent advances in ray tracing increased accuracy by introducing effects for depth of field, diffuse inter-reflection, soft shadows, motion blur, and other optical effects.

Simultaneously, numerous researchers have improved the performance of ray tracing using new algorithms for indexing the objects in the scene.

Realistic rendering algorithms based on ray tracing have been used to accurately simulate light transport. Some of these algorithms simulate the propagation of photons in a virtual environment. Others follow adjoint photons "backward" from a virtual camera to determine where they originated. Still other algorithms use bidirectional methods. OptiX operates at a level below such algorithmic decisions, so can be used to build any of those algorithms.

Ray tracing has often been used for non-graphics applications. In the computer-aided design community, ray tracing has been used to estimate the volume of complex parts. This is accomplished by sending a set of parallel rays at the part; the fraction of rays that hit the part gives the cross-sectional area, and the average length that those rays are inside the part gives the average depth. Ray tracing has also often been used to determine proximity (including collision) for complex moving objects. This is usually done by sending "feeler" rays from the surfaces of objects to "see" what is nearby. Rays are also commonly used for mouse-based object selection to determine what object is seen in a pixel, and for projectile-object collision in games. OptiX can be used for any of those applications.

The common feature in ray tracing algorithms is that they compute the intersection points of 3D rays (an origin and a propagation direction) and a collection of 3D surfaces (the "model" or "scene"). In rendering applications, the optical properties of the point where the ray intersects the model determine what happens to the ray (e.g., it might be reflected, absorbed or refracted). Other applications might not care about information other than where the intersection happens, or even if an intersection occurs at all. This variety of needs means it is desirable for OptiX to support a variety of ray-scene queries and user-defined behavior when rays intersect the scene.

One of ray tracing's nice features is that it is easy to support any geometric object that can be intersected with a 3D line. For example, it is straightforward to support spheres natively with no tessellation. Another nice feature is that ray tracing's execution is normally "sub-linear" in the number of objects---doubling the number of objects in the scene should less than double the running time. This is accomplished by organizing the objects into an acceleration structure that can quickly reject whole groups of primitives as not candidates for intersection with any given ray. For static parts of the scene, this structure can be reused for the life of the application. For dynamic parts of the scene, OptiX supports rebuilding the acceleration structure when needed. The structure only queries the bounding box of any geometric objects it contains, so new types of primitives can be added and the acceleration structures will continue to work without modification, so long as the new primitives can provide a bounding box.

For graphics applications, ray tracing has advantages over rasterization. One of these is that general camera models are easy to support; the user can associate points on the screen with any direction they want, and there is no requirement that rays originate at the same point. Another advantage is that important optical effects such as reflection and refraction can be supported with only a few lines of code Hard shadows are easy to produce with none of the artifacts typically associated with shadow maps, and soft shadows are not much harder. Furthermore, ray tracing can be added to more traditional graphics programs as a pass that produces a texture, letting the developer leverage the best of both worlds. For example, just the

specular reflections could be computed by using points in the depth buffer as ray origins. There are a number of such "hybrid algorithms" that use both z-buffer and ray tracing techniques.

For further information on ray tracing in graphics, see the following texts:

- The classic and still relevant book is "An Introduction to Ray Tracing" (Edited by A. Glassner, Academic Press, 1989).
- A very detailed beginner book is "Ray Tracing from the Ground Up" (K. Suffern, AK Peters, 2007).
- A concise description of ray tracing is in "Fundamentals of Computer Graphics" (P. Shirley and S. Marschner, AK Peters, 2009).
- A general discussion of realistic batch rendering algorithms is in "Advanced Global Illumination" (P. Dutré, P. Bekaert, K. Bala, AK Peters, 2006).
- A great deal of detailed information on ray tracing and algorithms that use ray tracing is in "Physically Based Rendering" (M. Pharr and G. Humphreys, Morgan Kaufmann, 2004).
- A detailed description of photon mapping is in "Realistic Image Synthesis Using Photon Mapping" (H. Jensen, AK Peters, 2001).
- A discussion of using ray tracing interactively for picking and collision detection, as well as a detailed discussion of shading and ray-primitive intersection is in "Real-Time Rendering" (T. Akenine-Möller, E. Haines, N. Hoffman, AK Peters, 2008).

Chapter 2. Programming Model Overview

The OptiX programming model consists of two halves: the host code and the GPU device programs. This chapter introduces the objects, programs, and variables that are defined in host code and used on the device.

Object Model

OptiX is an object-based C API that implements a simple retained mode object hierarchy. This object-oriented host interface is augmented with programs that execute on the GPU. The main objects in the system are:

- **Context** – An instance of a running OptiX engine
- **Program** – A CUDA C function, compiled to NVIDIA's PTX virtual assembly language
- **Variable** – A name used to pass data from C to OptiX programs
- **Buffer** – A multidimensional array that can be bound to a variable
- **TextureSampler** – One or more buffers bound with an interpolation mechanism
- **Geometry** – One or more primitives that a ray can be intersected with, such as triangles or other user-defined types
- **Material** – A set of programs executed when a ray intersects with the closest primitive or potentially closest primitive.
- **GeometryInstance** – A binding between Geometry and Material objects.
- **Group** – A set of objects arranged in a hierarchy
- **GeometryGroup** – A set of GeometryInstance objects
- **Transform** – A hierarchy node that geometrically transforms rays, so as to transform the geometric objects
- **Selector** – A programmable hierarchy node that selects which children to traverse
- **Acceleration** – An acceleration structure object that can be bound to a hierarchy node

These objects are created, destroyed, modified and bound with the C API and are further detailed in Chapter 3. The behavior of OptiX can be controlled by assembling these objects into any number of different configurations.

Programs

The ray tracing pipeline provided by OptiX contains several programmable components. These programs are invoked on the GPU at specific points during the execution of a generic ray tracing algorithm. There are eight types of programs:

- **Ray Generation** – The entry point into the ray tracing pipeline, invoked by the system in parallel for each pixel, sample, or other user-defined work assignment
- **Exception** – Exception handler, invoked for conditions such as stack overflow and other errors
- **Closest Hit** –Called when a traced ray finds the closest intersection point, such as for material shading
- **Any Hit** – Called when a traced ray finds a new potentially closest intersection point, such as for shadow computation
- **Intersection** – Implements a ray-primitive intersection test, invoked during traversal
- **Bounding Box** – Computes a primitive's world space bounding box, called when the system builds a new acceleration structure over the geometry
- **Miss** – Called when a traced ray misses all scene geometry
- **Visit** – Called during traversal of a Selector node to determine the children a ray will traverse

The input language for these programs is PTX. The OptiX SDK also provides a set of wrapper classes and headers for use with the NVIDIA C Compiler (nvcc) that enable the use of CUDA C as a way of generating appropriate PTX.

These programs are further detailed in Chapter 4.

Variables

OptiX features a flexible and powerful variable system for communicating data to programs. When an OptiX program references a variable, there is a well-defined set of scopes that will be queried for a definition of that variable. This enables dynamic overrides of variable definitions based on which scopes are queried for definitions.

For example, a closest hit program may reference a variable called *color*. This program may then be attached to multiple `Material` objects, which are, in turn, attached to `GeometryInstance` objects. Variables in closest hit programs first look for definitions directly attached to their `Program` object, followed by `GeometryInstance`, `Material` and `Context` objects, in that order. This enables a default *color* definition to exist on the `Material` object but specific instances using that material to override the default *color* definition.

See Section 3.4 for more information.

Execution Model

Once all of these objects, programs and variables are assembled into a valid context, ray generation programs may be launched. Launches take dimensionality and size parameters and invoke the ray generation program a number of times equal to the specified size.

Once the ray generation program is invoked, a special semantic variable may be queried to provide a runtime index identifying the ray generation program invocation. For example, a common use case is to launch a two-dimensional invocation with a width and height equal to the size, in pixels, of an image to be rendered.

See Section 4.3.2 for more information on launching ray generation programs from a context.

Chapter 3.Host API

Context

An OptiX context provides an interface for controlling the setup and subsequent launch of the ray tracing engine. Contexts are created with the `rtContextCreate` function. A context object encapsulates all OptiX resources -- textures, geometry, user-defined programs, etc. The destruction of a context, via the `rtContextDestroy` function, will clean up all of these resources and invalidate any existing handles to them.

`rtContextLaunch{1,2,3}D` serves as an entry point to ray engine computation. The launch function takes an entry point parameter, discussed in Section 3.1.1, as well as one, two or three grid dimension parameters. The dimensions establish a logical computation grid. Upon a call to `rtContextLaunch`, any necessary preprocessing is performed and then the ray generation program associated with the provided entry point index is invoked once per computational grid cell. The launch precomputation includes state validation and, if necessary, acceleration structure generation and kernel compilation. Output from the launch is passed back via OptiX buffers, typically but not necessarily of the same dimensionality as the computation grid.

```
RTcontext context;
rtContextCreate( &context );
unsigned int entry_point = ...;
unsigned int width = ...;
unsigned int height = ...;
// Set up context state and scene description
...
rtContextLaunch2D( entry_point, width, height );
rtContextDestroy( context );
```

While multiple contexts can be active at one time, this is usually unnecessary as a single context object can leverage multiple hardware devices. The devices to be used can be specified with `rtContextSetDevices`. By default, the highest compute capable set of compatible OptiX-capable devices is used. The following set of rules is currently used to determine device compatibility. These rules could change in the future. If incompatible devices are selected an error is returned from `rtContextSetDevices`.

- All SM 2.0+ devices can be run in multi-GPU configurations with other SM 2.0+ devices.
- All SM 1.2 and 1.3 devices can be run in multi-GPU configuration with other SM 1.2 and 1.3 devices.
- All SM 1.1 and 1.0 devices can only be run in single-GPU configurations.

Entry Points

Each context may have multiple computation entry points. A context entry point is associated with a single ray generation program as well as an exception program. The total number of entry points for a given context can be set with `rtContextSetEntryPointCount`. Each entry point's associated programs are set and queried by `rtContext{Set|Get}RayGenerationProgram` and `rtContext{Set|Get}ExceptionProgram`. Each entry point must be assigned a ray generation program before use; however, the exception program is an optional program that allows users to specify behavior upon various error conditions. The multiple entry point mechanism allows switching between multiple rendering algorithms as well as efficient implementation of techniques such as multi-pass rendering on a single OptiX context.

```
RTcontext context = ...;
rtContextSetEntryPointCount( context, 2 );

RTprogram pinhole_camera = ...;
RTprogram thin_lens_camera = ...;
RTprogram exception = ...;

rtContextSetRayGenerationProgram( context, 0,
                                  pinhole_camera );
rtContextSetRayGenerationProgram( context, 1,
                                  thin_lens_camera );
rtContextSetExceptionProgram( context, 0, exception );
rtContextSetExceptionProgram( context, 1, exception );
```

Ray Types

OptiX supports the notion of ray types, which is useful to distinguish between rays that are traced for different purposes. For example, a renderer might distinguish between rays used to compute color values and rays used exclusively for determining visibility of light sources (*shadow rays*). Proper separation of such conceptually different ray types not only increases program modularity, but also enables OptiX to operate more efficiently.

Both the number of different ray types as well as their behavior is entirely defined by the client application. The number of ray types to be used is set with `rtContextSetRayTypeCount()`.

The following properties may differ among ray types:

- The ray payload
- The closest hit program of each individual material
- The any hit program of each individual material
- The miss program

The ray payload is an arbitrary user-defined data structure associated with each ray. This is commonly used, for example, to store a result color, the ray's recursion depth, a shadow attenuation factor, and so on. It can be regarded as the result a ray delivers after having been traced, but it can also be used to store and propagate data between ray generations during recursive ray tracing.

The closest hit and any hit programs assigned to materials correspond roughly to shaders in conventional rendering systems: they are invoked when an intersection between a ray and a geometric primitive is found. Since those programs are assigned to materials per ray type, not all ray types must define behavior for both program types. See Sections 4.5 and 4.6 for a more detailed discussion of material programs.

The miss program is executed when a traced ray is determined to not hit any geometry. A miss program could, for example, return a constant sky color or sample from an environment map.

As an example of how to make use of ray types, a Whitted-style recursive ray tracer might define the ray types listed in Table 1:

Ray Type Purpose	Payload	Closest Hit	Any Hit	Miss
Radiance	`RadiancePL`	Compute color, keep track of recursion depth	n/a	Environment map lookup
Shadow	`ShadowPL`	n/a	Compute shadow attenuation and terminate ray if opaque	n/a

Table 1 Example Ray Types

The ray payload data structures in the above example might look as follows:

```
// Payload for ray type 0: radiance rays
struct RadiancePL
{
  float3 color;
  int    recursion_depth;
};

// Payload for ray type 1: shadow rays
struct ShadowPL
{
  float attenuation;
};
```

Upon a call to `rtContextLaunch()`, the ray generation program traces radiance rays into the scene, and writes the delivered results (found in the `color` field of the payload) into an output buffer for display:

```
RadiancePL payload;
payload.color = make_float3( 0.f, 0.f, 0.f );
```

```
payload.recursion_depth = 0; // initialize recursion
depth

Ray ray = ...     // some camera code creates the ray
ray.ray_type = 0; // make this a radiance ray

rtTrace( top_object, ray, payload );

// Write result to output buffer
writeOutput( payload.color );
```

A primitive intersected by a radiance ray would execute a closest hit program which computes the ray's color and potentially traces shadow rays and reflection rays. The shadow ray part is shown in the following code snippet:

```
ShadowPL shadow_payload;
shadow_payload.attenuation = 1.0f; // initialize to
visible

Ray shadow_ray = ...     // create a ray to light
source
shadow_ray.ray_type = 1; // make this a shadow ray

rtTrace( top_object, shadow_ray, shadow_payload );

// Attenuate incoming light ('light' is some user-
defined
// variable describing the light source)
float3 rad = light.radiance *
shadow_payload.attenuation;

// Add the contribution to the current radiance ray's
// payload (assumed to be declared as 'payload')
payload.color += rad;
```

To properly attenuate shadow rays, all materials use an any hit program which adjusts the attenuation and terminates ray traversal. The following code sets the attenuation to zero, assuming an opaque material:

```
shadow_payload.attenuation = 0; // assume opaque
material
rtTerminateRay(); // it won't get any darker, so
terminate
```

Global State

Aside from ray type and entry point counts, there are several other global settings encapsulated within OptiX contexts.

Each context holds a number of attributes that can be queried and set using `rtContext{Get|Set}Attribute`. For example, the amount of memory an OptiX context has allocated on the host can be queried by specifying `RT_CONTEXT_ATTRIBUTE_USED_HOST_MEMORY` as attribute parameter.

```
RTcontext context = ...;
RTsize used_host_memory;
rtContextGetAttribute( context,
RT_CONTEXT_ATTRIBUTE_USED_HOST_MEMORY, sizeof(RTsize),
&used_host_memory );
```

Currently, `rtContextGetAttribute` supports the following attributes: `RT_CONTEXT_ATTRIBUTE_MAX_TEXTURE_COUNT`, `RT_CONTEXT_CPU_NUM_THREADS`, `RT_CONTEXT_USED_HOST_MEMORY`, `RT_CONTEXT_AVAILABLE_DEVICE_MEMORY`, `RT_CONTEXT_ATTRIBUTE_GPU_PAGING_FORCED_OFF`, `RT_CONTEXT_ATTRIBUTE_GPU_PAGING_ACTIVE`. `rtContextSetAttribute` allows for setting the number of CPU threads used for various tasks such as acceleration structure builds via `RT_CONTEXT_CPU_NUM_THREADS` and allows disabling large memory paging via `RT_CONTEXT_ATTRIBUTE_GPU_PAGING_FORCED_OFF`. All other attributes are read-only.

To support recursion, OptiX uses a small stack of memory associated with each thread of execution. `rtContext{Get|Set}StackSize` allows for setting and querying the size of this stack. The stack size should be set with care as unnecessarily large stacks will result in performance degradation while overly small stacks will cause overflows within the ray engine. Stack overflow errors can be handled with user defined *exception programs*.

The `rtContextSetPrint*` functions are used to enable C-style `printf` printing from within OptiX programs, allowing these programs to be more easily debugged. The CUDA C function `rtContextSetPrintEnabled` turns on or off printing globally while `rtContextSetPrintLaunchIndex` toggles printing for individual computation grid cells. Print statements have no adverse effect on performance while printing is globally disabled, which is the default behavior.

Print requests are buffered in an internal buffer, the size of which can be specified with `rtContextSetPrintBufferSize`. Overflow of this buffer will cause truncation of the output stream. The output stream is printed to the standard output after all computation has completed but before `rtContextLaunch` has returned.

```
RTcontext context = ...;
rtContextSetPrintEnabled( context, 1 );
rtContextSetPrintBufferSize( context, 4096 );
```

Within an OptiX program, the `rtPrintf` function works similarly to C's `printf`. Each invocation of `rtPrintf` will be atomically deposited into the print output buffer, but separate invocations by the same thread or by different threads will be interleaved arbitrarily.

```
rtDeclareVariable( uint2, launch_index, rtLaunchIndex,
);
RT_PROGRAM void any_hit()
{
   rtPrintf( "Hello from index %u, %u!\n",
```

```
            launch_index.x, launch_index.y );
}
```

The context also serves as the outermost scope for OptiX variables. Variables declared via `rtContextDeclareVariable` are available to all OptiX objects associated with the given context. To avoid name conflicts, existing variables may be queried with either `rtContextQueryVariable` (by name) or `rtContextGetVariable` (by index), and removed with `rtContextRemoveVariable`.

`rtContextValidate` can be used at any point in the setup process to check the state validity of a context and all of its associated OptiX objects. This will include checks for the presence of necessary programs (e.g., an intersection program for a geometry node), invalid internal state such as unspecified children in graph nodes and the presence of variables referred to by all specified programs. Validation is always implicitly performed upon a context launch.

`rtContextCompile` can be used to explicitly request a compilation of the computation kernel associated with a context object. Use of `rtContextCompile` is not strictly necessary since any changes to a context's scene specification or programs will cause a compilation upon the next invocation of `rtContextLaunch`. `rtContextCompile` does allow the user to control the timing of the compilation, but the context should normally be finalized before compilation because any subsequent changes will cause a recompile within `rtContextLaunch`.

`rtContextSetTimeoutCallback` specifies a callback function of type `RTtimeoutcallback` that is called at a specified maximum frequency from OptiX API calls that can run long, such as acceleration structure builds, compilation, and kernel launches. This allows the application to update its interface or perform other tasks. The callback function may also ask OptiX to cease its current work and return control to the application. This request is complied with as soon as possible. Output buffers expected to be written to by an `rtContextLaunch` are left in an undefined state, but otherwise OptiX tracks what tasks still need to be performed and resumes cleanly in subsequent API calls.

```
// Return 1 to ask for abort, 0 to continue.
// An RTtimeoutcallback.
int CBFunc()
{
  update_gui();

  return bored_yet();
}

…
// Call CBFunc at most once every 100 ms.
rtContextSetTimeoutCallback( context, CBFunc, 0.1 );
```

`rtContextGetErrorString` can be used to get a description of any failures occurring during context state setup, validation or launch execution.

Buffers

OptiX uses buffers to pass data between the host and the device. Buffers are created by the host prior to invocation of `rtContextLaunch` using the `rtBufferCreate` function. This function also sets the buffer type as well as optional flags. The type and flags are specified as a bitwise OR combination.

The buffer type determines the direction of data flow between host and device. Its options are enumerated by `RTbuffertype`:

- `RT_BUFFER_INPUT` - Only the host may write to the buffer. Data is transferred from host to device and device access is restricted to be read-only.
- `RT_BUFFER_OUTPUT` - The converse of `RT_BUFFER_INPUT`. Only the device may write to the buffer. Data is transferred from device to host.
- `RT_BUFFER_INPUT_OUTPUT` - Allows read-write access from both the host and the device.

Buffer flags specify certain buffer characteristics and are enumerated by `RTbufferflags`:

- `RT_BUFFER_GPU_LOCAL` - Can only be used in combination with `RT_BUFFER_INPUT_OUTPUT`. This restricts the host to write operations as the buffer is not copied back from the device to the host. The device is allowed read-write access. However, writes from multiple devices are not coherent, as a separate copy of the buffer resides on each device.

Before using a buffer, its size, dimensionality and element format must be specified. The format can be set and queried with `rtBuffer{Get|Set}Format` and format options are enumerated by the `RTformat` type. Formats exist for C and CUDA C data types such as `unsigned int` and `float3`. Buffers of arbitrary elements can be created by choosing the format `RT_FORMAT_USER` and specifying an element size with the `rtBufferSetElementSize` function. The size of the buffer is set with `rtBufferSetSize{1,2,3}D` which also specifies the dimensionality implicitly.

```
RTcontext context = ...;
RTbuffer buffer;
typedef struct { float r; float g; float b; } rgb;
rtBufferCreate( context, RT_BUFFER_INPUT_OUTPUT,
&buffer );
rtBufferSetFormat( RT_FORMAT_USER );
rtBufferSetElementSize( sizeof(rgb) );
rtBufferSetSize2D( buffer, 512, 512 );
```

Host access to the data stored within a buffer is performed with the `rtBufferMap` function. This function returns a pointer to a one dimensional array representation of the buffer data. All buffers must be unmapped via `rtBufferUnmap` before context validation will succeed.

```
// Using the buffer created above
unsigned int width, height;
rtBufferGetSize2D( buffer, &width, &height );
void* data;
rtBufferMap( buffer, &data );
```

```
rgb* rgb_data = (rgb*)data;
for( unsigned int i = 0; i < width*height; ++I ) {
    rgb_data[i].r = rgb_data[i].g = rgb_data[i].b =
0.0f;
}
rtBufferUnmap( buffer );
```

Access to buffers within OptiX programs uses a simple array syntax. The two "template" arguments in the declaration below are the element type and the dimensionality, respectively.

```
rtBuffer<rgb, 2> buffer;
...
uint2 index = ...;
float r = buffer[index].r;
```

Textures

OptiX textures provide support for common texture mapping functionality including texture filtering, various wrap modes, and texture sampling. `rtTextureSamplerCreate` is used to create texture objects. Each texture object is associated with one or more buffers containing the texture data. The buffers may be 1D, 2D or 3D and can be set with `rtTextureSamplerSetBuffer`.

`rtTextureSamplerSetFilteringModes` can be used to set the filtering methods for minification, magnification and mipmapping. Wrapping for texture coordinates outside of [0, 1] can be specified per-dimension with `rtTextureSamplerSetWrapMode`. The maximum anisotropy for a given texture can be set with `rtTextureSamplerSetMaxAnisotropy`. A value greater than 0 will enable anisotropic filtering at the specified value. `rtTextureSamplerSetReadMode` can be used to request all texture read results be automatically converted to normalized float values.

The OptiX API has been designed to allow support for texture arrays and mip-mapping via the `rtTextureSamplerSetArraySize`, `rtTextureSamplerSetMipLevelCount` and `rtTextureSamplerSetIndexingMode`. However, OptiX 2.0 supports only a single mip level and a single element texture array. Future releases will fully support these features.

```
RTcontext context = ...;
RTbuffer tex_buffer = ...; // 2D buffer
RTtexturesampler tex_sampler;
rtTextureSamplerCreate( context, &tex_sampler );
rtTextureSamplerSetWrapMode( tex_sampler, 0,
                             RT_WRAP_CLAMP_TO_EDGE );
rtTextureSamplerSetWrapMode( tex_sampler, 1,
                             RT_WRAP_CLAMP_TO_EDGE );
rtTextureSamplerSetFilteringModes( tex_sampler,
                                   RT_FILTER_LINEAR,
                                   RT_FILTER_LINEAR,
```

```
                                          RT_FILTER_NONE );
rtTextureSamplerSetIndexingMode( tex_sampler,

RT_TEXTURE_INDEX_NORMALIZED_COORDINATES );
rtTextureSamplerSetReadMode( tex_sampler,

RT_TEXTURE_READ_NORMALIZED_FLOAT );
rtTextureSamplerSetMaxAnisotropy( tex_sampler, 1.0f );
rtTextureSamplerSetMipLevelCount( tex_sampler, 1 );
rtTextureSamplerSetArraySize( tex_sampler, 1 );
rtTextureSamplerSetBuffer( tex_sampler, 0, 0,
tex_buffer );
```

OptiX programs can access texture data with CUDA C's built-in `tex1D`, `tex2D` and `tex3D` functions.

```
rtTextureSampler<uchar4, 2,
cudaReadModeNormalizedFloat> t;
...
float2 tex_coord = ...;
float4 value = tex2D( t, tex_coord.x, tex_coord.y );
```

For further discussion of using textures within OptiX programs see Section 4.1.

Variables

OptiX variables provide a mechanism to pass named input parameters into user-defined programs. All variables are bound to an existing OptiX object upon creation. Contexts, programs, selectors, geometry instances, geometries and materials are all variable containers. These objects also determine the scope of their variables. A variable is visible to the object it is bound to as well as all of that object's children. Therefore, a context serves as a global scope as all variables bound to a context are visible to all other objects. Variables lower down in the object hierarchy will overshadow variables above them, allowing variable overrides. For a complete discussion of variable scoping, see Section 4.1.4.

`rt{Context|Program|…}DeclareVariable` can be used to declare a variable and set its name. A variable's value is specified by one of the `rtVariableSet*` functions, also determining the variable's type. The value and type of a variable may be queried via `rtVariableGet*` and `rtVariableGetType`.

```
RTcontext context = ...;
RTvariable variable;
rtContextDeclareVariable( context, "var1", &variable
);
rtVariableSet2f( 0.1f, 0.2f );
float value[2];
rtVariableGet2fv( variable, value );
RTobjecttype type;
rtVariableGetType( variable, &type );
```

OptiX programs can access variables and, optionally, provide them with default parameters in the following manner:

```
rtDeclareVariable( float2, var1, , );
rtDeclareVariable( float2, var2, , ) = {0.0f, 0.0f};
RT_PROGRAM void myProgram()
{
  float2 temp = var1;
}
```

All variables declared in user programs without default parameters must be declared and have their values set within the host code. Context validation will fail if this condition is not met. See Section 4.1.2 for further discussion of variable use within programs.

Graph Nodes

When a ray is traced from a program using the `rtTrace` function, a node is given that specifies the root of the graph. The host application creates this graph by assembling various types of nodes provided by the OptiX API. The basic structure of the graph is a hierarchy, with nodes describing geometric objects at the bottom, and collections of objects at the top.

The graph structure is not meant to be a scene graph in the classical sense. Instead, it serves as a way of binding different programs or actions to portions of the scene. Since each invocation of `rtTrace` specifies a root node, different trees or subtrees may be used. For example, shadowing objects or reflective objects may use a different representation – for performance or for artistic effect.

Graph nodes are created via `rt*Create` calls, which take the Context as a parameter. Since these graph node objects are owned by the context, rather than by their parent node in the graph, a call to `rt*Destroy` will delete that object's variables, but not do any reference counting or automatic freeing of its child nodes.

Figure 1 shows an example of what a graph might look like. The following sections will describe the individual node types.

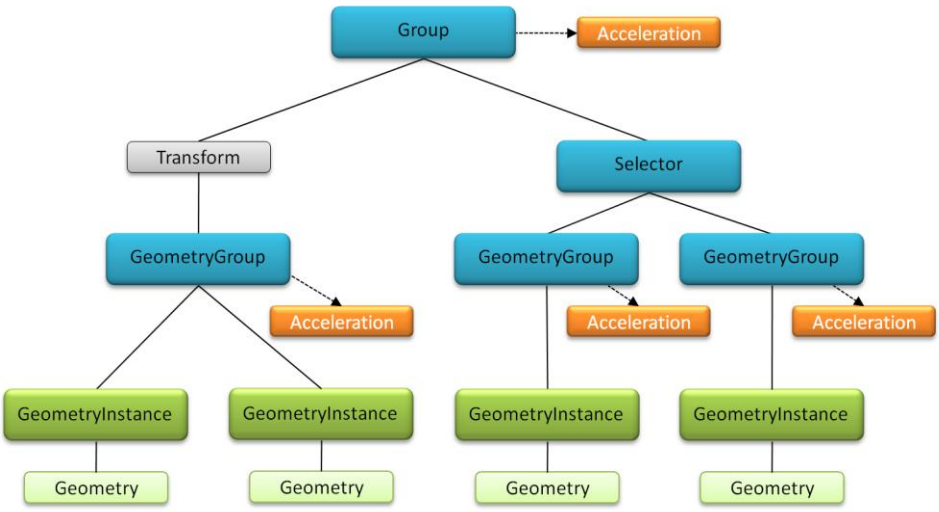

Figure 1 A sample node graph.

Geometry

A geometry node is the fundamental node to describe a geometric object: a collection of user-defined primitives against which rays can be intersected. The number of primitives contained in a geometry node is specified using `rtGeometrySetPrimitiveCount`.

To define the primitives, an intersection program is assigned to the geometry node using `rtGeometrySetIntersectionProgram`. The input parameters to an intersection program are a primitive index and a ray, and it is the program's job to return the intersection between the two. In combination with program variables, this provides the necessary mechanisms to define any primitive type that can be intersected against a ray. A common example is a triangle mesh, where the intersection program reads a triangle's vertex data out of a buffer (passed to the program via a variable) and performs a ray-triangle intersection.

In order to build an acceleration structure over arbitrary geometry, it is necessary for OptiX to query the bounds of individual primitives. For this reason, a separate bounds program must be provided using `rtGeometrySetBoundingBoxProgram`. This program simply computes bounding boxes of the requested primitives, which are then used by OptiX as the basis for acceleration structure construction.

Geometry nodes can be fully dynamic, i.e. the number of primitives as well as the variables on which the intersection and bounding box programs depend can vary between calls to `rtContextLaunch`. Whenever this is the case, acceleration structures containing references to the modified geometry node must be notified, which is achieved by calling `rtGeometryMarkDirty`. For more information on acceleration structure rebuilds, see Section 3.6.

The following example shows how to construct a geometry object describing a sphere, using a single primitive. The intersection and bounding box program are assumed to depend on a single parameter variable specifying the sphere radius:

```
RTgeometry geometry;
```

```
RTvariable variable;

// Set up geometry object.
rtGeometryCreate( context, &geometry );
rtGeometrySetPrimitiveCount( geometry, 1 );
rtGeometrySetIntersectionProgram( geometry,
                                  sphere_intersection
);
rtGeometrySetBoundingBoxProgram( geometry,
                                 sphere_bounds );

// Declare and set the radius variable.
rtGeometryDeclareVariable( geometry, "radius",
&variable );
rtVariableSet1f( variable, 10.0f );
```

Material

A material encapsulates the actions that are taken when a ray intersects a primitive associated with a given material. Examples for such actions include: computing a reflectance color, tracing additional rays, ignoring an intersection, and terminating a ray. Arbitrary parameters can be provided to materials by declaring program variables.

Two types of programs may be assigned to a material, closest hit programs and any hit programs. The two types differ in when and how often they are executed. The closest hit program, which is similar to a shader in a classical rendering system, is executed at most once per ray, for the closest intersection of a ray with the scene. It typically performs actions that involve texture lookups, reflectance color computations, light source sampling, recursive ray tracing, and so on, and stores the results in a ray payload data structure.

The any hit program is executed for each potential closest intersection found during ray traversal. The intersections for which the program is executed may not be ordered along the ray, but eventually all intersections of a ray with the scene can be enumerated if required (by calling `rtIgnoreIntersection` on each of them). Typical uses of the any hit program include early termination of shadow rays (using `rtTerminateRay`) and binary transparency, e.g., by ignoring intersections based on a texture lookup.

It is important to note that both types of programs are assigned to materials *per ray type*, which means that each material can actually hold more than one closest hit or any hit program. This is useful if an application can identify that a certain kind of ray only performs specific actions. For example, a separate ray type may be used for shadow rays, which are only used to determine binary visibility between two points in the scene. In this case, a simple any hit program attached to all materials under that ray type index can immediately terminate such rays, and the closest hit program can be omitted entirely. This concept allows for highly efficient specialization of individual ray types.

The closest hit program is assigned to the material by calling `rtMaterialSetClosestHitProgram`, and the any hit program is assigned with `rtMaterialSetAnyHitProgram`. If a program is omitted, an empty program is the default.

GeometryInstance

A geometry instance represents a coupling of a single geometry node with a set of materials. The geometry object the instance refers to is specified using `rtGeometryInstanceSetGeometry`. The number of materials associated with the instance is set by `rtGeometryInstanceSetMaterialCount`, and the individual materials are assigned with `rtGeometryInstanceSetMaterial`. The number of materials that must be assigned to a geometry instance is determined by the highest material index that may be reported by an intersection program of the referenced geometry.

Note that multiple geometry instances are allowed to refer to a single geometry object, enabling instancing of a geometric object with different materials. Likewise, materials can be reused between different geometry instances.

This example configures a geometry instance so that its first material index is `mat_phong` and the second one is `mat_diffuse`, both of which are assumed to be `rtMaterial` objects with appropriate programs assigned. The instance is made to refer to the `rtGeometry` object `triangle_mesh`.

```
RTgeometryinstance ginst;

rtGeometryInstanceCreate( context, &ginst );
rtGeometryInstanceSetGeometry( ginst, triangle_mesh );

rtGeometryInstanceSetMaterialCount( ginst, 2 );
rtGeometryInstanceSetMaterial( ginst, 0, mat_phong );
rtGeometryInstanceSetMaterial( ginst, 1, mat_diffuse
);
```

GeometryGroup

A geometry group is a container for an arbitrary number of geometry instances. The number of contained geometry instances is set using `rtGeometryGroupSetChildCount`, and the instances are assigned with `rtGeometryGroupSetChild`. Each geometry group must also be assigned an acceleration structure using `rtGeometryGroupSetAcceleration` (see Section 3.6).

The minimal sample use case for a geometry group is to assign it a single geometry instance:

```
RTgeometrygroup geomgroup;

rtGeometryGroupCreate( context, &geomgroup );
rtGeometryGroupSetChildCount( geomgroup, 1 );
rtGeometryGroupSetChild( geomgroup, 0,
geometry_instance );
```

Multiple geometry groups are allowed to share children, that is, a geometry instance can be a child of more than one geometry group.

Group

A group represents a collection of higher level nodes in the graph. They are used to compile the graph structure which is eventually passed to `rtTrace` for intersection with a ray.

A group can contain an arbitrary number of child nodes, which must themselves be of type `rtGroup`, `rtGeometryGroup`, `rtTransform`, or `rtSelector`. The number of children in a group is set by `rtGroupSetChildCount`, and the individual children are assigned using `rtGroupSetChild`. Every group must also be assigned an acceleration structure via `rtGroupSetAcceleration`.

A common use case for groups is to collect several geometry groups which dynamically move relative to each other. The individual position, rotation, and scaling parameters can be represented by transform nodes, so the only acceleration structure that needs to be rebuilt between calls to `rtContextLaunch` is the one for the top level group. This will usually be much cheaper than updating acceleration structures for the entire scene.

Note that the children of a group can be shared with other groups, that is, each child node can also be the child of another group (or of any other graph node for which it is a valid child). This allows for very flexible and lightweight instancing scenarios, especially in combination with shared acceleration structures (see Section 3.6).

Transform

A transform node is used to represent a projective transformation of its underlying scene geometry. The transform must be assigned exactly one child of type `rtGroup`, `rtGeometryGroup`, `rtTransform`, or `rtSelector`, using `rtTransformSetChild`. That is, the nodes below a transform may simply be geometry in the form of a geometry group, or a whole new subgraph of the scene.

The transformation itself is specified by passing a 4×4 floating point matrix (specified as a 16-element one-dimensional array) to `rtTransformSetMatrix`. Conceptually, it can be seen as if the matrix were applied to all the underlying geometry. However, the effect is instead achieved by transforming the rays themselves during traversal. This means that OptiX does not rebuild any acceleration structures when the transform changes.

This example shows how a transform object with a simple translation matrix is created:

```
RTtransform transform;
const float x=10.0f, y=20.0f, z=30.0f;

// Matrices are row-major.
const float m[16] = { 1, 0, 0, x,
                      0, 1, 0, y,
                      0, 0, 1, z,
                      0, 0, 0, 1 };

rtTransformCreate( context, &transform );
rtTransformSetMatrix( transform, 0, m, 0 );
```

Note that the transform child node may be shared with other graph nodes. That is, a child node of a transform may be a child of another node at the same time. This is often useful for instancing geometry.

Selector

A selector is similar to a group in that it is a collection of higher level graph nodes. The number of nodes in the collection is set by `rtSelectorSetChildCount`, and the individual children are assigned with `rtSelectorSetChild`. Valid child types are `rtGroup`, `rtGeometryGroup`, `rtTransform`, and `rtSelector`.

The main difference between selectors and groups is that selectors do not have an acceleration structure associated with them. Instead, a visit program is specified with `rtSelectorSetVisitProgram`. This program is executed every time a ray encounters the selector node during graph traversal. The program specifies which children the ray should continue traversal through by calling `rtIntersectChild`.

A typical use case for a selector is dynamic (i.e. per-ray) level of detail: an object in the scene may be represented by a number of geometry nodes, each containing a different level of detail version of the object. The geometry groups containing these different representations can be assigned as children of a selector. The visit program can select which child to intersect using any criterion (e.g. based on the footprint or length of the current ray), and ignore the others.

As for groups and other graph nodes, child nodes of a selector can be shared with other graph nodes to allow flexible instancing.

Acceleration Structures for Ray Tracing

Acceleration structures are an important tool for speeding up the traversal and intersection queries for ray tracing, especially for large scene databases. Most successful acceleration structures represent a hierarchical decomposition of the scene geometry. This hierarchy is then used to quickly cull regions of space not intersected by the ray.

There are many different types of acceleration structures, each with their own advantages and drawbacks. Furthermore, different scenes require different kinds of acceleration structures for optimal performance (e.g., static vs. dynamic scenes, generic primitives vs. triangles, and so on). The most common tradeoff is construction speed vs. ray tracing performance, and extreme solutions exist on both ends of the spectrum. For example, a high quality kd-tree or SBVH builder can take minutes to construct its acceleration structure. Once finished, though, rays can be traced more efficiently than with other types of acceleration structures, which in turn might be much faster to construct.

No single type of acceleration structure is optimal for all scenes. To allow an application to balance the tradeoffs, OptiX lets you choose between several kinds of supported structures. You can even mix and match different types of acceleration structures within the same node graph.

Acceleration objects in the Node Graph

Acceleration structures are individual API objects in OptiX, called `rtAcceleration`. Once an acceleration object is created with `rtAccelerationCreate`, it is assigned to either a group (using `rtGroupSetAcceleration`) or a geometry group (using `rtGeometryGroupSetAcceleration`). Every group and geometry group in the node graph needs to have an acceleration object assigned for ray traversal to intersect those nodes.

This example creates a geometry group and an acceleration structure and connects the two:

```
RTgeometrygroup geomgroup;
RTacceleration accel;

rtGeometryGroupCreate( context, &geomgroup );
rtAccelerationCreate( context, &accel );
rtGeometryGroupSetAcceleration( geomgroup, accel );
```

By making use of groups and geometry groups when assembling the node graph, the application has a high level of control over how acceleration structures are constructed over the scene geometry. If one considers the case of several geometry instances in a scene, there are a number of ways they can be placed in groups or geometry groups to fit the application's use case.

For example, Figure 2 places all the geometry instances in a single geometry group. An acceleration structure on a geometry group will be constructed over the individual primitives defined by the collection of child geometry instances. This will allow OptiX to build an acceleration structure which is as efficient as if the geometries of the individual instances had been merged into a single object.

A different approach to managing multiple geometry instances is shown in Figure 3. Each instance is placed in its own geometry group, i.e. there is a separate acceleration structure for each instance. The resulting collection of geometry groups is aggregated in a top level group, which itself has an acceleration structure. Acceleration structures on groups are constructed over the bounding volumes of the child nodes. Because the number of child nodes is usually relatively low, high level structures are typically quick to update. The advantage of this approach is that when one of the geometry instances is modified, the acceleration structures of the other instances need not be rebuilt. However, because higher level acceleration structures introduce an additional level of complexity and are built only on the coarse bounds of their group's children, the graph in Figure 3 will likely not be as efficient to traverse as the one in Figure 2. Again, this is a tradeoff the application needs to balance, e.g. in this case by considering how frequently individual geometry instances will be modified.

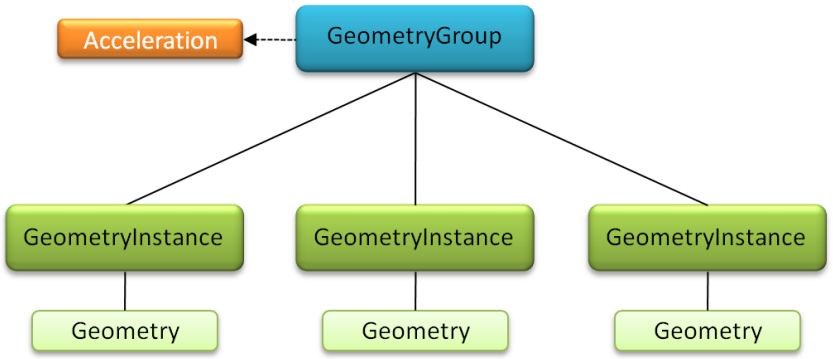

Figure 2 Multiple geometry instances in a geometry group

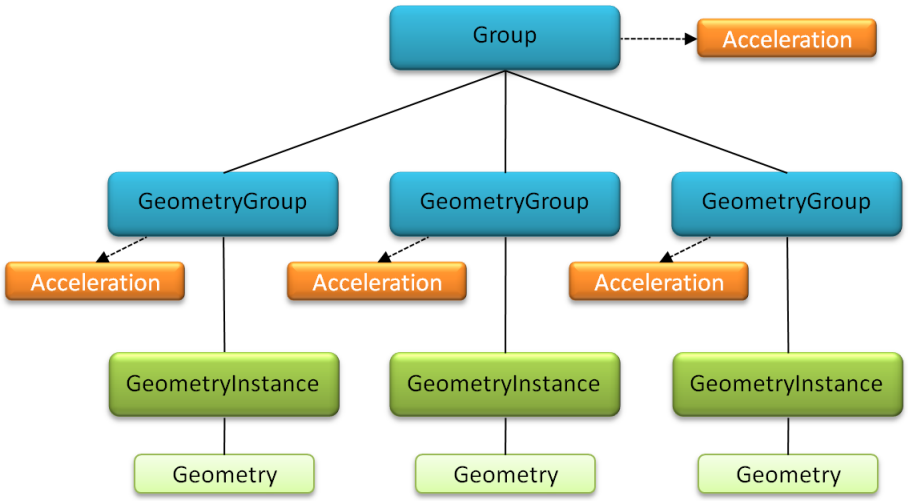

Figure 3 Multiple geometry instances, each in a separate geometry group

Builders and Traversers

An `rtAcceleration` consists of a *builder* and a *traverser*. The builder is responsible for collecting input geometry (in most cases, this geometry is the bounding boxes created by geometry nodes' bounding box programs) and computing a data structure that allows a *traverser* to accelerate a ray-scene intersection query. Builders and traversers are not application-defined programs. Instead, the application chooses an appropriate builder and its corresponding traverser from Table 2:

Builder / Traverser	Description
`Sbvh` / `Bvh` or `BvhCompact`	The Split-BVH (SBVH) is a high quality bounding volume hierarchy. While build times and memory footprint are highest, it is usually the method of choice for static geometry due to its high ray tracing performance. Improvements over regular BVHs are especially visible if the geometry is non-uniform (e.g. triangles of different sizes). This builder can be used for any type of geometry, but for optimal performance with triangle geometry, specialized properties should be set (see Table 3)[1].
`Bvh` / `Bvh` or `BvhCompact`[2]	The `Bvh` builder constructs a classic bounding volume hierarchy. It focuses on quality over construction performance and delivers a good middle ground between the `Sbvh` and `MedianBvh`. It also supports refitting for fast incremental updates (see Table 3). Bvh is often the best choice for acceleration structures built over groups.
`MedianBvh` / `Bvh` or `BvhCompact`	The `MedianBvh` builder uses a fast construction scheme to produce a medium-quality bounding volume hierarchy. It is typically useful for dynamic and semi-dynamic content, as well as for acceleration structures on groups.
`Lbvh` / `Bvh` or `BvhCompact`	The Lbvh builder uses the new HLBVH2 algorithm[3] to perform a very fast GPU-based bounding volume hierarchy build. It is ideal for many applications including very large or animated scenes where acceleration construction time dominates run time.
`TriangleKdTree` / `KdTree`	This builder constructs a high quality kd-tree, which in most cases is comparable to the SBVH in ray tracing performance. Build times and memory footprint are usually higher for the kd-tree. This builder is specialized for triangle geometry and thus needs to be configured using certain properties (see Table 3).
`NoAccel` / `NoAccel`	This is a dummy builder which does not construct an actual acceleration structure. Traversal loops over all elements and intersects each one with the ray. This is very inefficient for anything but very simple cases, but can sometimes outperform real acceleration structures, e.g. on a group with very few child nodes.

[1] More details on the SBVH can be found in Martin Stich, Heiko Friedrich, Andreas Dietrich. *Spatial Splits in Bounding Volume Hierarchies*. http://www.nvidia.com/object/nvidia_research_pub_012.html

[2] The BvhCompact traverser compresses the BVH data by a factor of four before uploading to the device and uses the compressed data structure in real-time during traversal of a bounding volume hierarchy. It is typically useful for large datasets to minimize the number of page misses when virtual memory is turned on.

[3] More information on the HLBVH2 can be found in Kirill Garanzha, Jacopo Pantaleoni, David McAllister. *Simpler and Faster HLBVH with Work Queues*. http://research.nvidia.com/publication/simpler-and-faster-hlbvh-work-queues

Table 2 Supported builders and traversers

Table 2 shows the builders and traversers currently available in OptiX. A builder is set using `rtAccelerationSetBuilder`, and the corresponding traverser, which must be compatible with the builder, is set with `rtAccelerationSetTraverser`. The builder and traverser can be changed at any time; switching builders will cause an acceleration structure to be flagged for rebuild.

This example shows a typical initialization of an acceleration object:

```
RTacceleration accel;

rtAccelerationCreate( context, &accel );
rtAccelerationSetBuilder( accel, "Bvh" );
rtAccelerationSetTraverser( accel, "Bvh" );
```

Properties

Fine-tuning acceleration structure construction can be useful depending on the situation. For this purpose, builders expose various named properties, which are listed in Table 3**:**

Property	Available in Builder	Description
`refit`	`Bvh`	If set to "1", the builder will only readjust the node bounds of the bounding volume hierarchy instead of constructing it from scratch. Refit is only effective if there is an initial BVH already in place, and the underlying geometry has undergone relatively modest deformation. In this case, the builder delivers a very fast BVH update without sacrificing too much ray tracing performance. The default is "0".

Property	Available in Builder	Description
`vertex_buffer_name`	`Sbvh` `TriangleKdTree`	The name of the buffer variable holding triangle vertex data. Each vertex consists of 3 floats. Mandatory for `TriangleKdTree`, optional for `Sbvh` (but recommended if the geometry consists of triangles). The default is "vertex_buffer".
`vertex_buffer_stride`	`Sbvh` `TriangleKdTree`	The offset between two vertices in the vertex buffer, given in bytes. The default value is "0", which assumes the vertices are tightly packed.
`index_buffer_name`	`Sbvh` `TriangleKdTree`	The name of the buffer variable holding vertex index data. The entries in this buffer are indices of type `int`, where each index refers to one entry in the vertex buffer. A sequence of three indices represents one triangle. If no index buffer is given, the vertices in the vertex buffer are assumed to be a list of triangles, i.e. every 3 vertices in a row form a triangle. The default is "index_buffer".
`index_buffer_stride`	`Sbvh` `TriangleKdTree`	The offset between two indices in the index buffer, given in bytes. The default value is "0", which assumes the indices are tightly packed.

Table 3 Acceleration structure properties

Properties are specified using `rtAccelerationSetProperty`. Their values are given as strings, which are parsed by OptiX. Properties take effect only when an acceleration structure is actually rebuilt. Setting or changing the property does not itself mark the acceleration structure for rebuild; see the next section for details on how to do that. Properties not recognized by a builder will be silently ignored.

```
// Enable fast refitting on a BVH acceleration.
rtAccelerationSetProperty( accel, "refit", "1" );
```

Acceleration Structure Builds

In OptiX, acceleration structures are flagged (marked "dirty") when they need to be rebuilt. During `rtContextLaunch`, all flagged acceleration structures are built

before ray tracing begins. Every newly created `rtAcceleration` object is initially flagged for construction.

An application can decide at any time to explicitly mark an acceleration structure for rebuild. For example, if the underlying geometry of a geometry group changes, the acceleration structure attached to the geometry group must be recreated. This is achieved by calling `rtAccelerationMarkDirty`. This is also required if, for example, new child geometry instances are added to the geometry group, or if children are removed from it.

The same is true for acceleration structures on groups: adding or removing children, changing transforms below the group, etc., are operations which require the group's acceleration to be marked as dirty. As a rule of thumb, every operation that causes a modification to the underlying geometry over which the structure is built (in the case of a group, that geometry is the children's axis-aligned bounding boxes) requires a rebuild. However, no rebuild is required if, for example, some parts of the graph change further down the tree, without affecting the bounding boxes of the immediate children of the group.

Note that the application decides independently for each single acceleration structure in the graph whether a rebuild is necessary. OptiX will not attempt to automatically detect changes, and marking one acceleration structure as dirty will not propagate the dirty flag to any other acceleration structures. Failure to mark acceleration structures as dirty when necessary may result in unexpected behavior – usually missing intersections or performance degradation.

Caching Acceleration Data

Depending on the choice of builder and the complexity of the underlying data, acceleration structure construction can be slow. OptiX provides a way to extract data from already-built acceleration structures, which allows the application to store that data for later reuse.

Acceleration data can be queried from an acceleration object using `rtAccelerationGetData` and can be restored using `rtAccelerationSetData`. The following sample shows how to request data from an acceleration structure:

```
RTsize size;
void* data;

rtAccelerationGetDataSize( accel, &size );
data = malloc( size );
rtAccelerationGetData( accel, data );
```

Note that data can only be extracted from an acceleration structure that is currently not flagged dirty (i.e. construction must have occurred), which guarantees that the data is valid. Therefore, it can be useful to force an acceleration structure build without actually performing any ray tracing. This can be done by calling `rtContextLaunch` with all dimension arguments set to zero:

```
rtContextLaunch1D( context, 0, 0 );
```

The data returned by `rtAccelerationGetData` will include information such as the used builder, traverser, and properties. Upon a successful call to

`rtAccelerationSetData`, all these data points will be restored and the acceleration structure will be flagged as non-dirty, as if a regular construction had been performed. This means, for example, that the current builder set in an acceleration object may change due to a call to `rtAccelerationSetData`.

Note, however, that it is only the information required to reconstruct the acceleration structure that is included in the returned data. In particular, actual geometry data and graph information of a group or geometry group are not included. The application should ensure that the restored acceleration structure matches the underlying geometry or the ensuing behavior is undefined.

It is also important to note that a well-written application should always be prepared for `rtAccelerationSetData` to fail. Among the reasons for failure of this call can be internal format changes from one version of OptiX to another, or incompatibilities between different platforms. It is usually straightforward to implement a correct handling of this case by simply marking the acceleration structure dirty if the call fails, which will cause the acceleration structure to be built on the fly instead of being reconstructed from cached data:

```
if( rtAccelerationSetData(accel,data,size) !=
RT_SUCCESS )
{
  rtAccelerationMarkDirty( accel );
}
```

Shared Acceleration Structures

Mechanisms such as a graph node being attached as a child to multiple other graph nodes make composing the node graph flexible, and enable interesting instancing applications. Instancing can be seen as inexpensive reuse of scene objects or parts of the graph by referencing nodes multiple times instead of duplicating them.

OptiX decouples acceleration structures as separate objects from other graph nodes. Hence, acceleration structures can naturally be shared between several groups or geometry groups, as long as the underlying geometry on which the structure is built is the same:

```
// Attach one acceleration to multiple groups.
rtGroupSetAcceleration( group1, accel );
rtGroupSetAcceleration( group2, accel );
rtGroupSetAcceleration( group3, accel );
```

Note that the application must ensure that each node sharing the acceleration structure has matching underlying geometry. Failure to do so will result in undefined behavior. Also, acceleration structures cannot be shared between groups and geometry groups.

The capability of sharing acceleration structures is a powerful concept to maximize efficiency, as shown in Figure 4. The acceleration node in the center of the figure is attached to both geometry groups, and both geometry groups reference the same geometry objects. This reuse of geometry and acceleration structure data minimizes both memory footprint and acceleration construction time. Additional geometry groups could be added in the same manner at very little overhead.

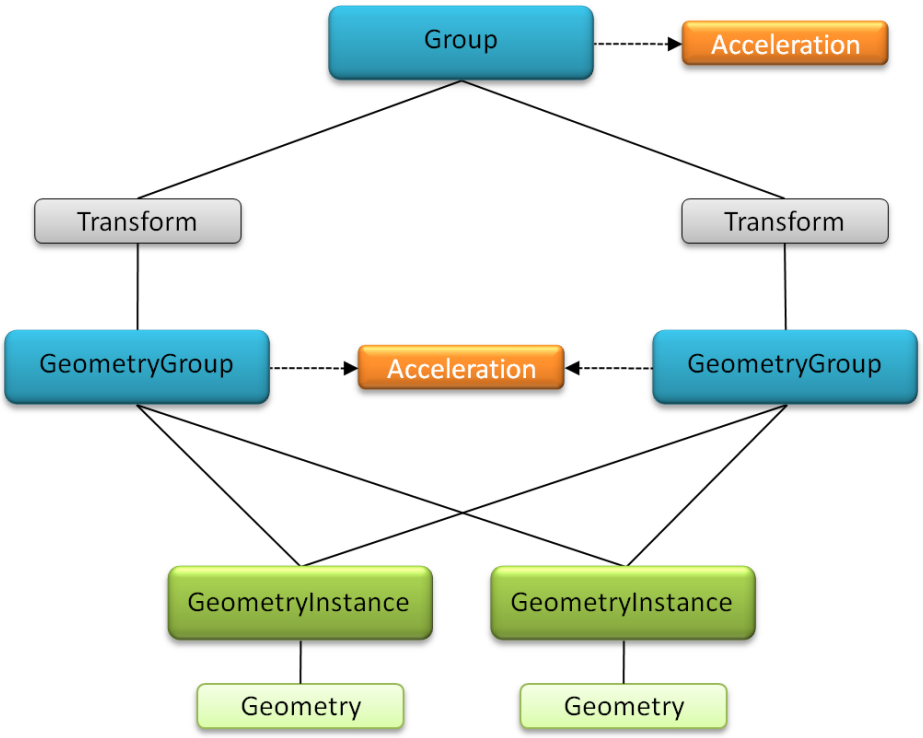

Figure 4 Two geometry groups sharing an acceleration structure and the underlying geometry objects.

Chapter 4. Programs

This chapter describes the different kinds of OptiX programs, which provide programmatic control over ray intersection, shading, and other general computation in OptiX ray tracing kernels. OptiX programs are associated with binding points serving different semantic roles during a ray tracing computation. Like other concepts, OptiX abstracts programs through its object model as *program objects*.

OptiX Program Objects

The central theme of the OptiX API is programmability. OptiX programs are written in CUDA C, and specified to the API through a string or file containing PTX, the parallel thread execution virtual assembly language associated with CUDA. The `nvcc` compiler that is distributed with the CUDA SDK is used to create PTX in conjunction with the OptiX header files.

These PTX files are then bound to Program objects via the host API. Program objects can be used for any of the OptiX program types discussed later in this section.

Managing Program Objects

OptiX provides two API entry points for creating Program objects: `rtProgramCreateFromPTXString`, and `rtProgramCreateFromPTXFile`. The former creates a new Program object from a string of PTX source code. The latter creates a new Program object from a file of PTX source on disk:

```
RTcontext context = ...;
const char *ptx_filename = ...;
const char *program_name = ...;
RTprogram program = ...;
rtProgramCreateFromPTXFile( context, ptx_filename,
                            function_name, &program );
```

In this example, `ptx_filename` names a file of PTX source on disk, and `function_name` names a particular function of interest within that source. If the program is ill-formed and cannot compile, these entry points return an error code.

Program objects may be checked for completeness using the `rtProgramValidate` function, as the following example demonstrates:

```
if( rtProgramValidate(context, program) != RT_SUCCESS
)
{
  printf( "Program is not complete." );
}
```

An error code returned from `rtProgramValidate` indicates an error condition due to the program object or any other objects bound to it.

Finally, the `rtProgramGetContext` function reports the context object owning the program object, while `rtProgramDestroy` invalidates the object and frees all resources related to it.

Communication Through Variables

OptiX program objects communicate with the host program through *variables*. Variables are declared in an OptiX program using the `rtDeclareVariable` macro:

```
rtDeclareVariable( float, x, , );
```

This declaration creates a variable named *x* of type `float` which is available to both the host program through the OptiX variable object API, and to the device program code through usual C language semantics. Notice that the last two arguments are left blank in this example. The commas must still be specified.

Variables declared in this way may be read and written by the host program through the `rtVariableGet*` and `rtVariableSet*` family of functions. When variables are declared this way, they are implicitly `const`-qualified from the device program's perspective. If communication from the program to the host is necessary, an `rtBuffer` should be used instead.

As of OptiX 2.0, variables may be declared inside arbitrarily nested namespaces to avoid name conflicts. References from the host program to namespace-enclosed OptiX variables will need to include the full namespace.

Program variables may also be declared with *semantics*. Declaring a variable with a semantic binds the variable to a special value which OptiX manages internally over the lifetime of the ray tracing kernel. For example, declaring a variable with the `rtCurrentRay` semantic creates a special read-only program variable that mirrors the value of the `Ray` currently being traced through the program flow:

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

Variables declared with a built-in semantic exist only during ray tracing kernel runtime and may not be modified or queried by the host program. Unlike regular variables, some semantic variables may be modified by the device program.

Declaring a variable with an *annotation* associates with it a read-only string which, for example, may be interpreted by the host program as a human-readable description of the variable. For example:

```
rtDeclareVariable( float, shininess, , "The shininess
of the sphere" );
```

A variable's annotation is the fourth argument of `rtDeclareVariable`, following the variable's optional semantic argument. The host program may query a variable's annotation with the `rtVariableGetAnnotation` function.

Internally Provided Semantics

OptiX currently manages four internal semantics for program variable binding. Table 4 summarizes in which types of program these semantics are available, along with their access rules from device programs and a brief description of their meaning.

Name	`rtLaunchIndex`	`rtCurrentRay`	`rtPayload`	`rtIntersectionDistance`
Access	read only	read only	read/write	read only
Description	The unique index identifying each thread launched by `rtContextLaunch{1|2|3}D`.	The state of the current ray.	The state of the current ray's payload of user-defined data.	The parametric distance from the current ray's origin to the closest intersection point yet discovered.
Ray Generation	Yes	No	No	No
Exception	Yes	No	No	No
Closest Hit	Yes	Yes	Yes	Yes
Any Hit	Yes	Yes	Yes	Yes
Miss	Yes	Yes	Yes	No
Intersection	Yes	Yes	No	Yes
Bounding Box	No	No	No	No
Visit	Yes	Yes	Yes	Yes

Table 4 Semantic Variables

Attribute Variables

In addition to the semantics provided by OptiX, variables may also be declared with user-defined semantics called *attributes*. Unlike built-in semantics, the value of variables declared in this way must be managed by the programmer. Attribute variables provide a mechanism for communicating data between the intersection program and the shading programs (e.g., surface normal, texture coordinates).

Attribute variables may *only* be written in an intersection program between calls to `rtPotentialIntersection` and `rtReportIntersection`. Although OptiX may not find all object intersections in order along the ray, the value of the attribute variable is guaranteed to reflect the value at the closest intersection at the time that the closest hit program is invoked. For this reason, programs should use attribute variables (as opposed to the ray payload) to communicate information about the local hit point between intersection and shading programs.

The following example declares an attribute variable of type `float3` named *normal*. The semantic association of the attribute is specified with the user-defined name *normal_vec*. This name is arbitrary, and is the link between the variable declared here and another variable declared in the closest hit program. The two attribute variables need not have the same name as long as their attribute names match.

```
rtDeclareVariable( float3, normal, attribute
normal_vec, );
```

Program Variable Scoping

OptiX program variables can have their values defined in two ways: static initializations, and (more typically) by variable declarations attached to API objects. A variable declared with a static initializer will only use that value if it does not find a definition attached to an API object. A declaration with static initialization is written:

```
rtDeclareVariable( float, x, , ) = 5.0f;
```

The OptiX variable scoping rules provide a valuable inheritance mechanism that is designed to create compact representations of material and object parameters. To enable this, each program type also has an ordered list of scopes through which it will search for variable definitions in order. For example, a closest hit program that refers to a variable named *color* will search the Program, GeometryInstance, Material and Context API objects for definitions created with the `rt*DeclareVariable()` functions, in that order. Similar to scoping rules in a programming language, variables in one scope will shadow those in another scope. Table 5 summarizes the scopes that are searched for variable declarations for each type of program.

Ray Generation	Program	Context		
Exception	Program	Context		
Closest Hit	Program	GeometryInstance	Material	Context
Any Hit	Program	GeometryInstance	Material	Context
Miss	Program	Context		
Intersection	Program	GeometryInstance	Geometry	Context
Bounding Box	Program	GeometryInstance	Geometry	Context
Visit	Program	Node		

Table 5 Scope search order for each type of program (from left to right)

It is possible for a program to find multiple definitions for a variable in its scopes depending upon where the program is called. For example, a closest hit program may be attached to several Material objects and reference a variable named *shininess*. We can attach a variable definition to the Material object as well as attach a variable definition to specific GeometryInstance objects that we create that reference that Material.

During execution of a specific GeometryInstance's closest hit program, the value of *shininess* depends on whether the particular instance has a definition attached: if the GeometryInstance defines *shininess*, then that value will be used. Otherwise, the value will be taken from the Material object. As you can see from Table 5 above, the program searches the GeometryInstance scope before the Material scope. Variables with definitions in multiple scopes are said to be *dynamic* and may incur a performance penalty. Dynamic variables are therefore best used sparingly.

Program Variable Transformation

Recall that rays have a projective transformation applied to them upon encountering Transform nodes during traversal. The transformed ray is said to be in *object space*, while the original ray is said to be in *world space*.

Programs with access to the `rtCurrentRay` semantic operate in the spaces summarized in Table 6:

Ray Generation	World
Closest Hit	World
Any Hit	Object
Miss	World
Intersection	Object
Visit	Object

Table 6 Space of rtCurrentRay for Each Program Type

To facilitate transforming variables from one space to another, OptiX's CUDA C API provides a set of functions:

```
__device__ float3 rtTransformPoint( RTtransformkind kind,
                                    const float3& p )
__device__ float3 rtTransformVector( RTtransformkind kind,
                                     const float3& v )
__device__ float3 rtTransformNormal( RTtransformkind kind,
                                     const float3& n )
__device__ void rtGetTransform( RTtransformkind kind,
                                float matrix[16] )
```

The first three functions transform a `float3`, interpreted as a point, vector, or normal vector, from object to world space or vice versa depending on the value of a `RTtransformkind` flag passed as an argument. `rtGetTransform` returns the four-by-four matrix representing the current transformation from object to world space (or vice versa depending on the `RTtransformkind` argument). For best performance, use the `rtTransform*()` functions rather than performing your own explicit matrix multiplication with the result of `rtGetTransform()`.

A common use case of variable transformation occurs when interpreting attributes passed from the intersection program to the closest hit program. Intersection programs often produce attributes, such as normal vectors, in object space. Should a closest hit program wish to consume that attribute, it often must transform the attribute from object space to world space:

```
float3 n = rtTransformNormal( RT_OBJECT_TO_WORLD, normal );
```

Which OptiX calls are supported where?

Not all OptiX function calls are supported in all types of user provided programs. For example, it doesn't make sense to spawn a new ray inside an intersection program, so this behavior is disallowed. A complete table of what device-side functions are allowed is given below:

	Ray Generation	Exception	Closest Hit	Any Hit	Miss	Intersection	Bounding Box	Visit
rtTransform*			●	●	●	●	●	●
rtTrace	●		●		●			
rtThrow	●		●	●	●	●	●	●
rtPrint	●	●	●	●	●	●	●	●
rtTerminateRay				●	●			
rtIgnoreIntersection				●				
rtIntersectChild								●
rtPotentialIntersection						●		
rtReportIntersection						●		

Ray Generation Programs

A *ray generation program* serves as the first point of entry upon a call to `rtContextLaunch{1|2|3}D`. As such, it serves a role analogous to the `main` function of a C program. Like C's `main` function, any subsequent computation performed by the kernel, from casting rays to reading and writing from buffers, is spawned by the ray generation program. However, unlike a serial C program, an OptiX ray generation program is executed many times in parallel – once for each thread implied by `rtContextLaunch{1|2|3}D`'s parameters.

Each thread is assigned a unique `rtLaunchIndex`. The value of this variable may be used to distinguish it from its neighbors for the purpose of, e.g., writing to a unique location in an `rtBuffer`:

```
rtBuffer<float, 1> output_buffer;
rtDeclareVariable( unsigned int, index, rtLaunchIndex,
);
...;
float result = ...;
output_buffer[index] = result;
```

In this case, the result is written to a unique location in the output buffer. In general, a ray generation program may write to any location in output buffers, as long as care is taken to avoid race conditions between buffer writes.

Entry Point Indices

To configure a ray tracing kernel launch, the programmer must specify the desired ray generation program using an *entry point index*. The total number of entry points for a context is specified with `rtContextSetEntryPointCount`:

```
RTcontext context = ...;
unsigned int num_entry_points = ...;
rtContextSetEntryPointCount( context, num_entry_points
);
```

OptiX requires that each entry point index created in this manner have a ray generation program associated with it. A ray generation program may be associated with multiple indices. Use the `rtContextSetRayGenerationProgram` function to associate a ray generation program with an entry point index in the range `[0, num_entry_points)`:

```
RTprogram prog = ...;
// index is >= 0 and < num_entry_points
unsigned int index = ...;
rtContextSetRayGenerationProgram( context, index, prog
);
```

Launching a Ray Generation Program

`rtContextLaunch{1|2|3}D` takes as a parameter the entry point index of the ray generation program to launch:

```
RTsize width = ...;
rtContextLaunch1D( context, index, width );
```

If no ray generation program has been associated with the entry point index specified by `rtContextLaunch{1|2|3}D`'s parameter, the launch will fail.

Ray Generation Program Function Signature

In CUDA C, ray generation programs return `void` and take no parameters. Like all OptiX programs, ray generation programs written in CUDA C must be tagged with the `RT_PROGRAM` qualifier. The following snippet shows an example ray generation program function prototype:

```
RT_PROGRAM void ray_generation_program( void );
```

Example Ray Generation Program

The following example ray generation program implements a pinhole camera model in a rendering application. This example demonstrates that ray generation programs act as the gateway to all ray tracing computation by initiating traversal through the `rtTrace` function, and often store the result of a ray tracing computation to an output buffer.

Note the variables `eye`, `U`, `V`, and `W`. Together, these four variables allow the host API to specify the position and orientation of the camera.

```
rtBuffer<uchar4, 2> output_buffer;
rtDeclareVariable( uint2, index, rtLaunchIndex, );
rtDeclareVariable( rtObject, top_object, , );
rtDeclareVariable(float3,        eye, , );
rtDeclareVariable(float3,        U, , );
rtDeclareVariable(float3,        V, , );
rtDeclareVariable(float3,        W, , );

struct Payload
{
  uchar4 result;
};

RT_PROGRAM void pinhole_camera( void )
{
  uint2 screen = output_buffer.size();

  float2 d = make_float2( index ) /
             make_float2( screen ) * 2.f - 1.f;
  float3 origin = eye;
  float3 direction = normalize( d.x*U + d.y*V + W );

  optix::Ray ray =
    optix::make_Ray( origin, direction, 0,
                     0.05f, RT_DEFAULT_MAX );

  Payload payload;
```

```
  rtTrace( top_object, ray, payload );

  output_buffer[index] = payload.result;
}
```

Exception Programs

OptiX ray tracing kernels invoke an *exception program* when certain types of serious errors are encountered. Exception programs provide a means of communicating to the host program that something has gone wrong during a launch. The information an exception program provides may be useful in avoiding an error state in a future launch or for debugging during application development.

Exception Program Entry Point Association

An exception program is associated with an entry point using the `rtContextSetExceptionProgram` function:

```
RTcontext context = ...;
RTprogram program = ...;
// index is >= 0 and < num_entry_points
unsigned int index = ...;
rtContextSetExceptionProgram( context, index, program
);
```

Unlike with ray generation programs, the programmer need not associate an exception program with an entry point. By default, entry points are associated with an internally provided exception program that silently ignores errors.

As with ray generation programs, a single exception program may be associated with many different entry points.

Exception Types

OptiX detects a number of different error conditions that result in exception programs being invoked. An exception is identified by its code, which is an integer defined by the OptiX API. For example, the exception code for the stack overflow exception is `RT_EXCEPTION_STACK_OVERFLOW`.

The type or code of a caught exception can be queried by calling `rtGetExceptionCode` from the exception program. More detailed information on the exception can be printed to the standard output using `rtPrintExceptionDetails`.

In addition to the built in exception types, OptiX provides means to introduce user-defined exceptions. Exception codes between `RT_EXCEPTION_USER` (`0x400`) and `0xFFFF` are reserved for user exceptions. To trigger such an exception, `rtThrow` is used:

```
// Define user-specified exception codes.
#define MY_EXCEPTION_0 RT_EXCEPTION_USER + 0
#define MY_EXCEPTION_1 RT_EXCEPTION_USER + 1
```

```
RT_PROGRAM void some_program()
{
  ...
  // Throw user exceptions from within a program.
  if( condition0 )
    rtThrow( MY_EXCEPTION_0 );
  if( condition1 )
    rtThrow( MY_EXCEPTION_1 );
  ...
}
```

In order to control the runtime overhead involved in checking for error conditions, individual types of exceptions may be switched on or off using `rtContextSetExceptionEnabled`. Disabling exceptions usually results in faster performance, but is less safe. By default, only `RT_EXCEPTION_STACK_OVERFLOW` is enabled. During debugging, it is often useful to turn on all available exceptions. This can be achieved with a single call:

```
...
rtContextSetExceptionEnabled(context,
RT_EXCEPTION_ALL, 1);
...
```

Exception Program Function Signature

In CUDA C, exception programs return `void`, take no parameters, and use the `RT_PROGRAM` qualifier:

```
RT_PROGRAM void exception_program( void );
```

Example Exception Program

The following example code demonstrates a simple exception program which indicates a stack overflow error by outputting a special value to an output buffer which is otherwise used as a buffer of pixels. In this way, the exception program indicates the `rtLaunchIndex` of the failed thread by marking its location in a buffer of pixels with a known color. Exceptions which are not caused by a stack overflow are reported by printing their details to the console.

```
rtDeclareVariable( int, launch_index, rtLaunchIndex,
);
rtDeclareVariable( float3, error, , ) =
make_float3(1,0,0);
rtBuffer<float3, 2> output_buffer;

RT_PROGRAM void exception_program( void )
{
  const unsigned int code = rtGetExceptionCode();

  if( code == RT_EXCEPTION_STACK_OVERFLOW )
    output_buffer[launch_index] = error;
  else
```

```
    rtPrintExceptionDetails();
}
```

Closest Hit Programs

After a call to the `rtTrace` function, OptiX invokes a *closest hit program* once it identifies the nearest primitive intersected along the ray from its origin. Closest hit programs are useful for performing primitive-dependent processing that should occur once a ray's visibility has been established. A closest hit program may communicate the results of its computation by modifying per-ray data or writing to an output buffer. It may also recursively call the `rtTrace` function. For example, a computer graphics application might implement a surface shading algorithm with a closest hit program.

Closest Hit Program Material Association

A closest hit program is associated with each (*material*, *ray_type*) pair. Each pair's default program is a no-op. This is convenient when an OptiX application requires many types of rays but only a small number of those types require special closest hit processing.

The programmer may change an association with the `rtMaterialSetClosestHitProgram` function:

```
RTmaterial material = ...;
RTprogram program = ...;
unsigned int type = ...;
rtMaterialSetClosestHitProgram( material, type,
program );
```

Closest Hit Program Function Signature

In CUDA C, closest hit programs return `void`, take no parameters, and use the `RT_PROGRAM` qualifier:

```
RT_PROGRAM void closest_hit_program( void );
```

Recursion in a Closest Hit Program

Though the `rtTrace` function is available to all programs with access to the `rtLaunchIndex` semantic, a common use case of closest hit programs is to perform recursion by tracing more rays upon identification of the closest surface intersected by a ray. For example, a computer graphics application might implement Whitted-style ray tracing by recursive invocation of `rtTrace` and closest hit programs. Care must be used to limit the recursion depth to avoid stack overflow.

Example Closest Hit Program

The following code example demonstrates a closest hit program that transforms the normal vector computed by an intersection program (not shown) from the intersected primitive's local coordinate system to a global coordinate system. The

transformed normal vector is returned to the calling function through a variable declared with the `rtPayload` semantic. Note that this program is quite trivial; normally the transformed normal vector would be used by the closest hit program to perform some calculation (e.g., lighting). See the OptiX Quickstart Guide for examples.

```
rtDeclareVariable( float3, normal, attribute
normal_vec, );

struct Payload
{
  float3 result;
};

rtDeclareVariable( Payload, ray_data, rtPayload, );

RT_PROGRAM void closest_hit_program( void )
{
  float3 norm;
  norm = rtTransformNormal( RT_OBJECT_TO_WORLD, normal
);
  norm = normalize( norm );
  ray_data.result = norm;
}
```

Any Hit Programs

Instead of the closest intersected primitive, an application may wish to perform some computation for *any* primitive intersection that occurs along a ray cast during the `rtTrace` function; this usage model can be implemented using *any hit programs*. For example, a rendering application may require some value to be accumulated along a ray at each surface intersection.

Any Hit Program Material Association

Like closest hit programs, an any hit program is associated with each (*material*, *ray_type*) pair. Each pair's default association is with an internally-provided any hit program which implements a no-op.

The `rtMaterialSetAnyHitProgram` function changes a `(material, ray_type)` pair's association:

```
RTmaterial material = ...;
RTprogram program = ...;
unsigned int type = ...;
rtMaterialSetAnyHitProgram( material, type, program );
```

Termination in an Any Hit Program

A common OptiX usage pattern is for an any hit program to halt ray traversal upon discovery of an intersection. The any hit program can do this by calling `rtTerminateRay`. This technique can increase performance by eliminating redundant traversal computations when an application only needs to determine whether *any* intersection occurs and identification of the *nearest* intersection is irrelevant. For example, a rendering application might use this technique to implement shadow ray casting, which is often a binary true or false computation.

Any Hit Program Function Signature

In CUDA C, any hit programs return `void`, take no parameters, and use the `RT_PROGRAM` qualifier:

```
RT_PROGRAM void any_hit_program( void );
```

Example Any Hit Program

The following code example demonstrates an any hit program that implements early termination of shadow ray traversal upon intersection. The program also sets the value of a per-ray payload member, `attenuation`, to zero to indicate the material associated with the program is totally opaque.

```
struct Payload
{
  float attenuation;
};

rtDeclareVariable( Payload, payload, rtPayload, );

RT_PROGRAM void any_hit_program( void )
{
  payload.attenuation = 0.f;

  rtTerminateRay();
}
```

Miss Programs

When a ray traced by the `rtTrace` function intersects no primitive, a *miss program* is invoked. Miss programs may access variables declared with the `rtPayload` semantic in the same way as closest hit and any hit programs.

Miss Program Function Signature

In CUDA C, miss programs return `void`, take no parameters, and use the `RT_PROGRAM` qualifier:

```
RT_PROGRAM void miss_program( void );
```

Example Miss Program

In a computer graphics application, the miss program may implement an environment mapping algorithm using a simple gradient, as this example demonstrates:

```
rtDeclareVariable( float3, environment_light, , );
rtDeclareVariable( float3, environment_dark, , );
rtDeclareVariable( float3, up, , );

struct Payload
{
  float3 result;
};

rtDeclareVariable( Payload, payload, rtPayload, );
rtDeclareVariable( optix::Ray, ray, rtCurrentRay, );

RT_PROGRAM void miss(void)
{
  float t = max( dot( ray.direction, up ), 0.0f );
  payload.result = lerp( environment_light,
                         environment_dark, t );
}
```

Intersection and Bounding Box Programs

Intersection and *bounding box programs* represents geometry by implementing ray-primitive intersection and bounding algorithms. These program types are associated with and queried from Geometry objects using `rtGeometrySetIntersectionProgram`, `rtGeometryGetIntersectionProgram`, `rtGeometrySetBoundingBoxProgram`, and `rtGeometryGetBoundingBoxProgram`.

Intersection and Bounding Box Program Function

Like the previously discussed OptiX programs, in CUDA C, intersection and bounding box programs return `void` and use the `RT_PROGRAM` qualifier. Because Geometry objects are collections of primitives, these functions require a parameter to specify the index of the primitive of interest to the computation. This parameter is always in the range `[0, N)`, where `N` is given by the argument to the `rtGeometrySetPrimitiveCount` function.

Additionally, the bounding box program requires an array of `floats` to store the result of the bounding box computation, yielding these function signatures:

```
RT_PROGRAM void intersection_program( int prim_index
);
RT_PROGRAM void bounding_box_program( int prim_index,
```

```
                                                    float result[6]
);
```

Reporting Intersections

Ray traversal invokes an intersection program when the current ray encounters one of a Geometry object's primitives. It is the responsibility of an intersection program to compute whether the ray intersects with the primitive, and to report the parametric *t-value* of the intersection. Additionally, the intersection program is responsible for computing and reporting any details of the intersection, such as surface normal vectors, through attribute variables.

Once the intersection program has determined the t-value of a ray-primitive intersection, it must report the result by calling a pair of OptiX functions, `rtPotentialIntersection` and `rtReportIntersection`:

```
__device__ bool rtPotentialIntersection( float tmin )
__device__ bool rtReportIntersection( unsigned int
material )
```

`rtPotentialIntersection` takes the intersection's t-value as an argument. If the t-value could potentially be the closest intersection of the current traversal the function narrows the *t-interval* of the current ray accordingly and returns `true`. If the t-value lies outside the t-interval the function returns `false`, whereupon the intersection program may trivially return.

If `rtPotentialIntersection` returns `true`, the intersection program may then set any attribute variable values and call `rtReportIntersection`. This function takes an `unsigned int` specifying the index of a material that must be associated with an any hit and closest hit program. This material index can be used to support primitives of several different materials flattened into a single Geometry object. Traversal then immediately invokes the corresponding any hit program. Should that any hit program invalidate the intersection via the `rtIgnoreIntersection` function, then `rtReportIntersection` will return `false`. Otherwise, it will return `true`.

The values of attribute variables must be modified only between the call to `rtPotentialIntersection` and the call to `rtReportIntersection`. The result of writing to an attribute variable outside the bounds of these two calls is undefined. The values of attribute variables written in this way are accessible by any hit and closest hit programs.

If the any hit program invokes `rtIgnoreIntersection`, any attributes computed will be reset to their previous values and the previous t-interval will be restored.

If no intersection exists between the current ray and the primitive, an intersection program need only return.

Specifying Bounding Boxes

Acceleration structures use *bounding boxes* to bound the spatial extent of scene primitives to accelerate the performance of ray traversal. A bounding box program's responsibility is to describe the minimal three dimensional axis-aligned

bounding box that contains the primitive specified by its first argument and store the result in its second argument. Bounding boxes are always specified in object space, so the user should not apply any transformations to them.

For correct results bounding boxes must merely contain the primitive. For best performance bounding boxes should be as tight as possible.

Example Intersection and Bounding Box Programs

The following code demonstrates how an intersection and bounding box program combine to describe a simple geometric primitive. The sphere is a simple analytic shape with a well-known ray intersection algorithm. In the following code example, the *sphere* variable encodes the center and radius of a three-dimensional sphere in a `float4`:

```
rtDeclareVariable( float4, sphere, , );
rtDeclareVariable( optix::Ray, ray, rtCurrentRay, );
rtDeclareVariable( float3, normal, attribute normal );

RT_PROGRAM void intersect_sphere( int prim_index )
{
  float3 center = make_float3( sphere.x, sphere.y,
                               sphere.z );
  float radius = sphere.w;
  float3 O = ray.origin - center;
  float b = dot( O, ray.direction );
  float c = dot( O, O ) - radius*radius;
  float disc = b*b - c;
  if( disc > 0.0f ) {
    float sdisc = sqrtf( disc );
    float root1 = (-b - sdisc);
    bool check_second = true;
    if( rtPotentialIntersection( root1 ) ) {
      shading_normal = geometric_normal =
        (O + root1*D) / radius;
      if( rtReportIntersection( 0 ) )
        check_second = false;
    }
    if( check_second ) {
      float root2 = (-b + sdisc);
      if( rtPotentialIntersection( root2 ) ) {
        shading_normal = geometric_normal =
          (O + root2*D) / radius;
        rtReportIntersection( 0 );
      }
    }
  }
}
```

Note that this intersection program ignores its `prim_index` argument and passes a material index of 0 to `rtReportIntersection`; it represents only the single primitive of its corresponding Geometry object.

The bounding box program for the sphere is very simple:

```
RT_PROGRAM void bound_sphere( int, float result[6] )
{
  float3 cen = make_float3( sphere.x, sphere.y,
sphere.z );
  float3 rad = make_float3( sphere.w, sphere.y,
sphere.z );

  // compute the minimal and maximal corners of
  // the axis-aligned bounding box
  float3 min = cen - rad;
  float3 max = cen + rad;
  // store results in order
  result[0] = min.x;
  result[1] = min.y;
  result[2] = min.z;
  result[3] = max.x;
  result[4] = max.y;
  result[5] = max.z;
}
```

Selector Programs

Ray traversal invokes selector *visit programs* upon encountering a Selector node to programmatically select which of the node's children the ray shall visit. A visit program dispatches the current ray to a particular child by calling the `rtIntersectChild` function. The argument to `rtIntersectChild` selects the child by specifying its index in the range `[0, N)`, where `N` is given by the argument to `rtSelectorSetChildCount`.

Selector Visit Program Function Signature

In CUDA C, visit programs return `void`, take no parameters, and use the `RT_PROGRAM` qualifier:

```
RT_PROGRAM void visit_program( void );
```

Example Visit Program

Visit programs may implement, for example, sophisticated level-of-detail systems or simple selections based on ray direction. The following code sample demonstrates an example visit program that selects between two children based on the direction of the current ray:

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

RT_PROGRAM void visit( void )
{
  unsigned int index = (unsigned int)( ray.direction.y
< 0 );
  rtIntersectChild( index );
```

```
}
```

Chapter 5.Building with OptiX

Libraries

OptiX comes with several header files and two supporting libraries, optix and optixu in 32- and 64-bit versions. On Windows these libraries are statically linked against the C runtime libraries and are suitable for use in any version of MS Visual Studio, though only VS 2005 and 2008 have been tested. In addition, if you wish to distribute the OptiX libraries with your application, the VS redistributables are not required by our DLL.

The OptiX libraries are numbered not by release version, but by binary compatibility. Incrementing this number means that a library will not work in place of an earlier version (e.g. optix.2.dll will not work when an optix.1.dll is requested). On Linux, you will find liboptix.so which is a soft link to liboptix.so.1 which is a soft link to liboptix.so.2.1.0, the actual library. liboptix.so.1 is the binary compatibility number similar to optix.1.dll. On MacOS X, liboptix.2.1.0.dylib is the actual library, and you will also find a soft link named liboptix.1.dylib (again, with the 1 indicating the level of binary compatibility), as well as liboptix.dylib.

Header Files

There are two principal methods to gain access to the OptiX API. Including `<optix.h>` in host and device code will give access strictly to the C API. Using `<optix_world.h>` in host and device code will provide access to the C and C++ API as well as importing additional helper classes, functions, and types into the `optix` namespace (including wrappers for CUDA's vector types such as `float3`).

Sample5 from the SDK provides two identical implementations using both the C (`<optix.h>`) and C++ (`<optixpp_namespace.h>`) API, respectively. Understanding this sample should give you a good sense of how the C++ wrappers work.

The `optixu` include directory contains several headers that augment the C API. The `namespace` versions of the header files (see the list of files below) place all the classes, functions, and types into the `optix` namespace. This allows better integration into systems which would have had conflicts within the global namespace. Backward compatibility is maintained if you include the old headers. It is not recommended to mix the old global namespace versions of the headers with the new `optix` namespace versions of the headers in the same project. Doing so can result in linker errors and type confusion.

- <optix_world.h> – General include file for the C/C++ APIs for host and device code, plus various helper classes, functions, and types all wrapped in the optix namespace.
- <optix.h> – General include file for the C API for host and device code.
- <optixu/optixu_math_namespace.h> – Provides additional operators for CUDA's vector types as well as a host of functions such as fminf, refract, and an ortho normal basis class.
- <optixu/optixupp_namespace.h> – C++ API for OptiX (backward compatibility with optixu:: namespace is provided in <optixpp.h>)
- <optixu/optixu_matrix_namespace.h> – Templated multi-dimensional matrix class with certain operations specialized for specific dimensions.
- <optixu/optixu_aabb_namespace.h> – Axis-Aligned Bounding Box class.
- <optixu/optixu_math_stream_namespace.h> – Standard template library stream operators for CUDA's vector types.
- <optixu/optixu_vector_types.h> – Wrapper around CUDA's <vector_types.h> header that defines the CUDA vector types in the optix namespace.
- <optixu/optixu_vector_functions.h> – Wrapper around CUDA's <vector_functions.h> header that defines CUDA's vector functions (e.g. make_float3) into the optix namespace.

PTX Generation

Programs supplied to the OptiX API must be written in PTX. This PTX could be generated from any mechanism, but the most common method is to use the CUDA Toolkit's nvcc compiler to generate PTX from CUDA C code.

When nvcc is used, make sure the desired device code bitness is targeted by using the -m32 or -m64 flag. The bitness of all PTX given to the OptiX API must match, and will determine the bitness of the generated device code. 64-bit PTX may only be used with 64-bit application binaries. Note that on devices that do not support 64 bit device pointers, the memory space available to the application will be limited to 32 bits despite the use of 64 bit PTX.

When using nvcc to generate PTX output specify the -ptx flag. Note that any host code in the CUDA file will not be present in the generated PTX file. Your CUDA files should include <optix_world.h> or <optix.h> to gain access to functions and definitions required by OptiX. In addition, there is <optixu/optixu_math_namespace.h> that defines many useful operations for vector types and ray tracing. <optixu/optixu_math_namespace.h> can be included in both host and device code. Note that <optix_world.h> includes this file automatically.

In order to run nvcc from within Visual Studio while building 64-bit host code, add "`-ccbin $(VCInstallDir)bin`" to tell nvcc where to find the 32-bit Visual Studio compiler. Visual Studio will replace `$(VCInstallDir)` with the path on any command executed through its build system.

In order to provide better support for compilation of PTX to different SM targets, OptiX uses the `.target` information found in the PTX code to determine compatibility with the currently utilized hardware. If you wish your code to run a sm_12 device, compiling the PTX with `-arch sm_13` will generate an error even if no sm_13 features are present in the code. Compiling to sm_12 will run on sm_12 and higher targets (e.g. sm_13 and sm_20).

SDK Build

Our SDK samples' build environment is generated by CMake. CMake is a cross platform tool that generates several types of build systems, such as Visual Studio projects and makefiles. The SDK comes with three text files describing the installation procedures on Windows, Macintosh, and Linux, currently named INSTALL-WIN.txt, INSTALL-MAC.txt and INSTALL-LINUX.txt respectively. See the appropriate file for your operating system for details on how to compile the SDK.

Chapter 6.Interoperability with OpenGL and Direct3D

OptiX supports the sharing of data between OpenGL/D3D applications and both `rtBuffers` and `rtTextureSamplers`. This way, OptiX applications can read data directly from objects such as vertex and pixel buffers, and can also write arbitrary data for direct consumption by graphics shaders. This sharing is referred to as *interop*.

OpenGL Interop

OptiX supports interop for OpenGL buffer objects, textures, and render buffers. OpenGL buffer objects can be read and written by OptiX program objects, whereas textures and render buffers can only be read.

Buffer Objects

OpenGL buffer objects like PBOs and VBOs can be encapsulated for use in OptiX with `rtBufferCreateFromGLBO`. The resulting buffer is a reference only to the OpenGL data; the size of the OptiX buffer as well as the format have to be set via `rtBufferSetSize` and `rtBufferSetFormat`. When the OptiX buffer is destroyed, the state of the OpenGL buffer object is unaltered. Once an OptiX buffer is created, the original GL buffer object is immutable, meaning the properties of the GL object like its size cannot be changed while registered with OptiX. However, it is still possible to read and write to the GL buffer object using the appropriate GL functions. If it is necessary to change properties of an object, first call `rtBufferGLUnregister` before making changes. After the changes are made the object has to be registered again with `rtBufferGLRegister`. This is necessary to allow OptiX to access the objects data again. Registration and unregistration calls are expensive and should be avoided if possible.

Textures and Render Buffers

OpenGL texture and render buffer objects must be encapsulated for use in OptiX with `rtTextureSamplerCreateFromGLImage`. This call may return with `RT_ERROR_MEMORY_ALLOCATION_FAILED` for textures that have a size of 0. Once an OptiX texture sampler is created, the original GL texture is immutable, meaning the properties of the GL texture like its size cannot be changed while registered with OptiX. However, it is still possible to read and write to the GL texture using the appropriate GL functions. If it is necessary to change properties of a GL texture, first call `rtTextureSamplerGLUnregister` before making

changes. After the changes are made the texture has to be registered again with `rtTextureSamplerGLRegister`. This is necessary to allow OptiX to access the textures data again. Registration and unregistration calls are expensive and should be avoided if possible.

Currently, only textures with the following GL targets are supported:

- `GL_TEXTURE_2D`
- `GL_TEXTURE_2D_RECT`
- `GL_TEXTURE_3D`

Supported attachment points for render buffers are:

- `GL_COLOR_ATTACHMENT<NUM>`

Not all OpenGL texture formats are currently supported by OptiX. A table that lists the supported texture formats can be found in Appendix A.

OptiX detects automatically the size, texture format, and number of mipmap levels of a texture. `rtTextureSamplerSetMipLevelCount`, `rtTextureSamplerSetArraySize`, and `rtTextureSampler(Set/Get)Buffer` cannot be called for OptiX interop texture samplers and will return `RT_ERROR_INVALID_VALUE`.

Direct3D Interop

OptiX also provides interop functionality for D3D9, D3D10, and D3D11 buffer objects, as well as textures/surfaces on appropriate Windows platforms. D3D buffer objects can be read and written by OptiX program objects, whereas textures and surfaces can only be read.

Before any subsequent call can be made to create OptiX interop buffers or texture samplers, `rtContextSetD3D<9/10/11>InteropDevice` has to be called in order to register the device with the OptiX context that performs the D3D commands. A context can only be bound to a single D3D device. The binding is immutable throughout the lifetime of a context.

Buffer Objects

Currently OptiX supports the following D3D buffer types:

- `IDirect3DIndexBuffer9`
- `IDirect3DVertexBuffer9`
- `ID3D10Buffer`
- `ID3D11Buffer`

OptiX buffer objects must be created from existing D3D resources with `rtBufferCreateFromD3D<9/10/11>Resource`. These calls may return with `RT_ERROR_MEMORY_ALLOCATION_FAILED` for buffers that have a size of 0. The resulting buffer is a reference only to the D3D data; the size of the OptiX buffer as well as the format have to be set via `rtBufferSetSize` and `rtBufferSetFormat`. Once an OptiX buffer is created, the original D3D

buffer object is immutable, meaning properties of the D3D object like its size cannot be changed while the buffer is registered with OptiX. However, it is still possible to read and write to the D3D buffer object using the appropriate D3D functions. If it is necessary to change properties of an object, unregister the buffer with `rtBufferD3D<9/10/11>Unregister`. After the changes are made the object has to be registered again with `rtBufferD3D<9/10/11>Register`. This is necessary to allow OptiX to access the objects data again. Registration and unregistration calls are expensive and should be avoided if possible.

`rtBufferGetD3D<9/10/11>Resource` can be used to query the bound D3D resource pointer. The appropriate size and format of the OptiX buffer must be set with `rtBufferSetSize<1/2/3>D` and `rtBufferSetFormat`.

D3D Buffer-Creation Flags

In certain situations OptiX requires host-side (CPU) access to interop objects. Because of this, D3D buffers should be created with the proper flags to allow OptiX access to the underlying data.

D3D9 buffers require no special flags.

For the current version of OptiX (2.1), CPU access flags are required to be set in the appropriate D3D buffer description for D3D10 and D3D11, e.g. D3D11_CPU_ACCESS_READ. A future version of OptiX will remove this requirement.

Textures and Surfaces

Currently OptiX supports the following D3D texture and surface types:

- `IDirect3DSurface9`
- `IDirect3DTexture9`
- `IDirect3DVolumeTexture9`
- `ID3D10Texture<1/2/3>D`
- `ID3D11Texture<1/2/3>D`

Cube maps, texture arrays as well as mipmap levels are not supported. OptiX texture sampler objects must be created from existing D3D resources with `rtTextureSamplerCreateFromD3D<9/10/11>Resource`. These calls may return with `RT_ERROR_MEMORY_ALLOCATION_FAILED` for textures that have a size of 0. Once an OptiX texture sampler is created, the original D3D texture is immutable, meaning the properties of the D3D texture like its size cannot be changed while registered with OptiX. However, it is still possible to read and write to the D3D texture using the appropriate D3D functions. If it is necessary to change properties of a D3D texture, unregister the texture with `rtTextureSamplerD3D<9/10/11>Unregister`. After the changes are made the texture has to be registered again with `rtTextureSampler<9/10/11>Register`. This is necessary to allow OptiX to access the textures data again. Registration and unregistration calls are expensive and should be avoided if possible.

`rtTextureSamplerGetD3D<9/10/11>Resource` can be used to query the bound D3D resource pointer. A list with the currently supported texture formats can be found in the Appendix A.

OptiX automatically detects the size, texture format, and number of mipmap levels of a texture. `rtTextureSamplerSetMipLevelCount`, `rtTextureSamplerSetArraySize, and rtTextureSampler(Set/Get)Buffer` cannot be called for OptiX interop texture samplers and will return `RT_ERROR_INVALID_VALUE`.

D3D Texture-Creation Flags

In certain situations OptiX requires host-side (CPU) access to interop objects. Because of this, D3D textures should be created with the proper flags to allow OptiX access to the underlying data.

OptiX requires D3D9 textures to be created with the memory pool argument using either D3DPOOL_MANAGED or D3DPOOL_DEFAULT in combination with D3DUSAGE_DYNAMIC.

For the current version of OptiX (2.1), CPU access flags are required to be set in the appropriate D3D texture description for D3D10 and D3D11, e.g. D3D11_CPU_ACCESS_READ. A future version of OptiX will remove this requirement.

Chapter 7.OptiXpp: C++ Wrapper for the OptiX C API

OptiXpp wraps each OptiX C API opaque type in a C++ class and provides relevant operations on that type. Most of the OptiXpp class member functions map directly to C API function calls. For example, `VariableObj::getContext` wraps `rtVariableGetContext` and `ContextObj::createBuffer` wraps `rtBufferCreate`.

Some functions perform slightly more complex sequences of C API calls. For example

```
ContextObj::createBuffer(unsigned int type, RTformat
format, RTsize width)
```

provides in one call the functionality of

```
rtBufferCreate
rtBufferSetFormat
rtBufferSetSize1D
```

See OptiX_Utility_Library_Reference.pdf or `optixpp_namespace.h` for a full list of the available OptiXpp functions. The usage of the API is described below.

OptiXpp Objects

The OptiXpp classes consist of a Handle class, a class for each API opaque type, and three classes that provide attributes to these objects.

Handle Class

All classes are manipulated via the reference counted `Handle` class. Rather than working with a `ContextObj` directly you would use a `Context` instead, which is simply a `typedef` for `Handle<ContextObj>`.

In addition to providing reference counting and automatic destruction when the reference count reaches zero, the `Handle` class provides a mechanism to create a handle from a C API opaque type, as follows:

```
    RTtransform t;
    rtTransformCreate( my_context, &t );
    Transform Tr = Transform::take( t );
```

The converse of `take` is `get`, which returns the underlying C API opaque type, but does not decrement the reference count within the handle.

```
Transform Tr;
...
rtTransformDestroy( Tr->get( ) );
```

These functions are typically used when calling C API functions, though such is rarely necessary since OptiXpp provides nearly all OptiX functionality.

Attribute Classes

The attributes are API, Destroyable, and Scoped.

API: All object types have the API attribute. This attribute provides the following functions to objects:

- `getContext()` – Return the context to which this object belongs
- `checkError()` – Check the given result code and throw an error with appropriate message if the code is not `RTsuccess`. `checkError` is often used as a wrapper around a call to a function that makes OptiX API calls:

```
my_context->checkError( sutilDisplayFilePPM( ... ) );
```

Destroyable: This attribute provides the following functions to objects:

- `destroy()` – Equivalent to `rt*Destroy()`
- `validate()` – Equivalent to `rt*Validate()`

Scoped: This attribute applies only to API objects that are containers for `RTvariables`. It provides functions for accessing the contained variables. The most basic access is via `operator[]`, as follows:

```
my_context["new_variable"]->setFloat( 1.0f );
```

This access returns the variable, but first creates it within the containing object if it does not already exist.

This array operator syntax with the string variable name argument is probably the most powerful feature of OptiXpp, as it greatly reduces the amount of code necessary to access a variable.

The following functions are also available to Scoped objects:

- `declareVariable()` – Declare a variable associated with this object
- `queryVariable()` – Query a variable associated with this object by name
- `removeVariable()` – Remove a variable associated with this object
- `getVariableCount()` – Query the number of variables associated with this object, typically so as to iterate over them

- `getVariable()` – Query variable by index, typically while iterating over them

The following table lists all of the OptiXpp objects and their attributes.

Object	API	Destroyable	Scoped
Context	yes	yes	yes
Program	yes	yes	yes
Buffer	yes		
Variable	yes		
TextureSampler	yes	yes	
Group	yes	yes	
GeometryGroup	yes	yes	
GeometryInstance	yes	yes	yes
Geometry	yes	yes	yes
Material	yes	yes	yes
Transform	yes	yes	
Selector	yes	yes	

Table 7 OptiXpp Opaque Types and Their Attributes

API Objects

In addition to the methods provided by the attribute classes that give commonality to the different API objects each object type also has a unique set of methods. These functions cover the complete set of functionality from the C API, although not all methods will be described here. See `optixpp_namespace.h` for the complete set.

Context

The Context object provides `create*` functions for creating all other opaque types. These are owned by the context and handles to the created object are returned:

```
Context my_context;
Buffer Buf = my_context->createBuffer(RT_BUFFER_INPUT,
RT_FORMAT_FLOAT4, 1024, 1024);
```

Context also provides `launch()` functions, with overloads for 1D, 2D, and 3D kernel launches. It provides many other functions that wrap `rtContext*` C API calls.

Buffer

The Buffer class provides a `map` call that returns a pointer to the buffer data, and provides an `unmap` call. It also provides `set` and `get` functions for the buffer format, element size, and 1D, 2D, and 3D buffer size. Finally, it provides `registerGLBuffer` and `unregisterGLBuffer`.

Variable

The Variable class provides `getName`, `getAnnotation`, `getType`, and `getSize` functions for returning properties of the variable. It also contains a multitude of `set*` functions that set the value of the variable and its type, if the type is not already set:

```
my_context["my_dim3"]->setInt( 512, 512, 1024 );
```

The Variable object also offers `set` functions for setting its value to an API object, and provides `setUserData` and `getUserData`.

TextureSampler

The TextureSampler class provides functions to set and get the attributes of an `RTtexturesampler`, such as `setWrapMode`, `setMipLevelCount`, etc.

It also provides `setBuffer`, `getBuffer`, `registerGLTexture`, and `unregisterGLTexture`.

Group and GeometryGroup

The remaining API object classes are for OptiX node types. They offer member functions for setting and querying the nodes to which they attach.

The Group class provides `setAcceleration`, `getAcceleration`, `setChildCount`, `getChildCount`, `setChild`, and `getChild`.

GeometryInstance

`RTgeometryinstance` is a binding of Geometry and Material. Thus, GeometryInstance provides functions to `set` and `get` both the `Geometry` and the `Materials`. This includes `addMaterial`, which increments the material count and appends the given Material to the list.

Geometry

The unique functions provided by the Geometry class `set` and `get` the `BoundingBoxProgram`, the `IntersectionProgram` and the `PrimitiveCount`. It also offers `markDirty` and `isDirty`.

Material

A Material consists of a `ClosestHitProgram` and an `AnyHitProgram`, and is a container for the variables appertaining to these programs. It contains `set` and `get` functions for these programs.

Transform

An `RTtransform` node applies a transformation matrix to its child, so the Transform class offers `setChild`, `getChild`, `setMatrix`, and `getMatrix` methods.

Selector

A Selector node applies a Visit program to operate on its multiple children. Thus, the Selector class includes functions to `set` and `get` the `VisitProgram`, `ChildCount`, and `Child`.

Exceptions

The Exception class of OptiXpp encapsulates an error message. These errors are often the direct result of a failed OptiX C API function call and subsequent `rtContextGetErrorString` call. Nearly all methods of all object types can throw an exception using the Exception class. Likewise, the `checkError()` function can throw an Exception.

Additionally, the Exception class can be used explicitly by user code as a convenient way to throw exceptions of the same type as OptiXpp.

Call `Exception::makeException` to create an Exception.

Call `getErrorString()` to return an std::string for the error message as returned by `rtContextGetErrorString()`.

Chapter 8.Performance Guidelines

Subtle changes in your code can dramatically alter performance. This list of performance tips should help when using OptiX.

- Where possible use floats instead of doubles. This also extends to the use of literals and math functions. For example, use `0.5f` instead of `0.5` and `sinf` instead of `sin` to prevent automatic type promotion. To check for automatic type promotion, search the PTX files for the ".f64" instruction modifier.
- OptiX will try to partition thread launches into tiles that are the same dimensionality as the launch. To have maximal coherency between the threads of a tile you should choose a launch dimensionality that is the same as the coherence dimensionality of your problem. For example, the common problem of rendering an image has 2D coherency (adjacent pixels both horizontally and vertically look at the same part of the scene), so a 2D launch is appropriate. Conversely, a collision detection problem with many agents each looking in many directions may appear to be 2D (the agents in one dimension and the ray directions in another), but there is rarely coherence between different agents, so the coherence dimensionality is one, and performance will be better by using a 1D launch.
- Do not build an articulate scene graph with `Groups`, `Transforms` and `GeometryInstances`. Try to make the topology as shallow and minimal as possible. For example, for static scenes the fastest performance is achieved by having a single `GeometryGroup`, where transforms are flattened to the geometry. For scenes where `Transforms` are changing all the static geometry should go in one `GeometryGroup` and each `Transform` should have a single `GeometryGroup`. Also, if possible, combine multiple meshes into a single mesh.
- Each new Program object can introduce execution divergence. Try to reuse the same program with different variable values. However, don't take this idea too far and attempt to create an "über shader". This will create execution divergence within the program. Experiment with your scene to find the right balance.
- Try to minimize live state across calls to `rtTrace` in programs. For example, in a closest hit program temporary values *used* after a recursive call to `rtTrace` should be *computed* after the call to `rtTrace`, rather than before, since these values must be saved and restored when calling `rtTrace`, impacting performance. `RTvariables` declared outside of the program body are exempt from this rule.

- No acceleration structure is best in all situations. For static geometry groups, use `Sbvh` or `TriangleKdTree`. For dynamic geometry groups or large number of elements, experiment with `Lbvh` (recommended) and `MedianBvh`. For Group nodes (e.g. higher level graph nodes), `Bvh` is the best choice in many cases, but if there are few enough children `NoAccel` can be useful.

- In multi-GPU environments `INPUT_OUTPUT` and `OUTPUT` buffers are stored on the host. In order to optimize writes to these buffers, types of either 4 bytes or 16 bytes (e.g. `float`, `uint`, or `float4`) should be used when possible. One might be tempted to make an output buffer used for the screen out of `float3`'s (RGB), however using a `float4` buffer will result in improved performance (e.g. `output_buffer[launch_index] = make_float4( result_color ))`. This also affects defined types (see the progressivePhotonMap sample for an example of accessing user defined structs with `float4`s).

- In multi-GPU environments `INPUT_OUTPUT` buffers may be stored on the device, with a separate copy per device by using the `RT_BUFFER_GPU_LOCAL` buffer attribute. This is useful for avoiding the slower reads and writes by the device to host memory. `RT_BUFFER_GPU_LOCAL` is useful for scratch buffers, such as random number seed buffers and variance buffers.

- Use iteration instead of recursion where possible (e.g. path tracing with no ray branching). See the path_tracer sample for an example of how to use iteration instead of recursion when tracing secondary rays.

- For best performance, use the `rtTransform*` functions rather than explicitly transforming by the matrix returned by `rtGetTransform`.

- Disable exceptions that are not needed. While it is recommended to turn on all available exception types during development and for debugging, the error checking involved e.g. to validate buffer index bounds is usually not necessary in the final product.

- Avoid recompiling the OptiX kernel. These recompiles can be triggered when certain changes to the input programs or variables occur. For example, swapping the ClosestHit program of a Material between two programs will cause a recompile on each swap because the kernel consists of different code, whereas creating two Materials, one with each program, and swapping between the two Materials will not cause a recompile because only the node graph is changing, not the code. Creating dummy nodes with the alternate programs is one way to provide all of the code at once. Also avoid changing the layout of variables attached to scope objects.

- It is possible for a program to find multiple definitions for a variable in its scopes depending upon where the program is called. Variables with definitions in multiple scopes are said to be *dynamic* and may incur a performance penalty.
  For example, a closest hit program may be attached to several Material objects and reference a variable named *shininess*. We can attach a variable definition to the Material object as well as attach a variable definition to

specific GeometryInstance objects that we create that reference that Material. During execution of a specific GeometryInstance's closest hit program, the value of *shininess* depends on whether the particular instance has a definition attached: if the GeometryInstance defines *shininess*, then that value will be used. Otherwise, the value will be taken from the Material object.

- When creating PTX code using nvcc, adding `--use-fast-math` as a compile option can reduce code size and increase the performance for most OptiX programs. This can come at the price of slightly decreased numerical floating point accuracy. See the nvcc documentation for more details.

Chapter 9.Caveats

Keep in mind the following caveats when using OptiX.

- Setting a large stack size will consume GPU device memory. Try to minimize the stack as much as possible. Start with a small stack and with the use of an exception program that will make it obvious you have exceeded your memory, increase the stack size until the stack is sufficiently large.
- The use of `__shared__` memory within a program is not recommended. This is currently untested.
- Don't use PTX `bar()` or CUDA `syncthreads()`. It will lock up your machine.
- `threadIdx` in CUDA can map to multiple launch indices (e.g. pixels). Use the `rtLaunchIndex` semantic instead.
- Use of the CUDA `malloc()`, `free()`, and `printf()` functions within a program is not supported. Attempts to use these functions will result in an illegal symbol error.
- Currently, OptiX is **not** guaranteed to be thread-safe. While it may be successful in some applications to use OptiX contexts in different host threads, it may fail in others. OptiX should therefore only be used from within a single host thread.

Appendix A. Supported Interop Texture Formats

OpenGL Texture Format	D3D Format	DXGI Format
GL_RGBA8	D3DFMT_R32F	DXGI_FORMAT_R8_SINT
GL_RGBA16	D3DFMT_L16	DXGI_FORMAT_R8_SNORM
GL_R32F	D3DFMT_L8	DXGI_FORMAT_R8_UINT
GL_RG32F	D3DFMT_A8	DXGI_FORMAT_R8_UNORM
GL_RGBA32F	D3DFMT_G32R32F	DXGI_FORMAT_R16_SINT
GL_R8I	D3DFMT_G16R16	DXGI_FORMAT_R16_SNORM
GL_R8UI	D3DFMT_V16U16	DXGI_FORMAT_R16_UINT
GL_R16I	D3DFMT_A8L8	DXGI_FORMAT_R16_UNORM
GL_R16UI	D3DFMT_V8U8	DXGI_FORMAT_R32_SINT
GL_R32I	D3DFMT_A32B32G32R32F	DXGI_FORMAT_R32_UINT
GL_R32UI	D3DFMT_A16B16G16R16	DXGI_FORMAT_R32_FLOAT
GL_RG8I	D3DFMT_A8R8G8B8	DXGI_FORMAT_R8G8_SINT
GL_RG8UI	D3DFMT_X8R8G8B8	DXGI_FORMAT_R8G8_SNORM
GL_RG16I	D3DFMT_A8B8G8R8	DXGI_FORMAT_R8G8_UINT_
GL_RG16UI	D3DFMT_X8B8G8R8	DXGI_FORMAT_R8G8_UNORM
GL_RG32I	D3DFMT_Q16W16V16U16	DXGI_FORMAT_R16G16_SINT
GL_RG32UI	D3DFMT_Q8W8V8U8	DXGI_FORMAT_R16G16_SNORM
GL_RGBA8I		DXGI_FORMAT_R16G16_UINT
GL_RGBA8UI		DXGI_FORMAT_R16G16_UNORM
GL_RGBA16I		DXGI_FORMAT_R32G32_SINT
GL_RGBA16UI		DXGI_FORMAT_R32G32_UINT
GL_RGBA32I		DXGI_FORMAT_R32G32_FLOAT
GL_RGBA32UI		DXGI_FORMAT_R8G8B8A8_SINT
		DXGI_FORMAT_R8G8B8A8_SNORM
		DXGI_FORMAT_R8G8B8A8_UINT
		DXGI_FORMAT_R8G8B8A8_UNORM
		DXGI_FORMAT_R16G16B16A16_SINT
		DXGI_FORMAT_R16G16B16A16_SNORM
		DXGI_FORMAT_R16G16B16A16_UINT
		DXGI_FORMAT_R16G16B16A16_UNORM
		DXGI_FORMAT_R32G32B32A32_SINT
		DXGI_FORMAT_R32G32B32A32_UINT
		DXGI_FORMAT_R32G32B32A32_FLOAT

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com