

# OptiX: A General Purpose Ray Tracing Engine

Steven G. Parker<sup>1\*</sup> James Bigler<sup>1</sup> Andreas Dietrich<sup>1</sup> Heiko Friedrich<sup>1</sup> Jared Hoberock<sup>1</sup> David Luebke<sup>1</sup>  
David McAllister<sup>1</sup> Morgan McGuire<sup>1,2</sup> Keith Morley<sup>1</sup> Austin Robison<sup>1</sup> Martin Stich<sup>1</sup>  
NVIDIA<sup>1</sup> Williams College<sup>2</sup>



**Figure 1:** Images from various applications built with OptiX. Top: Physically based light transport through path tracing. Bottom: Ray tracing of a procedural Julia set, photon mapping, large-scale line of sight and collision detection, Whitted-style ray tracing of dynamic geometry, and ray traced ambient occlusion. All applications are interactive.

## Abstract

The NVIDIA® OptiX™ ray tracing engine is a programmable system designed for NVIDIA GPUs and other highly parallel architectures. The OptiX engine builds on the key observation that most ray tracing algorithms can be implemented using a small set of programmable operations. Consequently, the core of OptiX is a domain-specific just-in-time compiler that generates custom ray tracing kernels by combining user-supplied programs for ray generation, material shading, object intersection, and scene traversal. This enables the implementation of a highly diverse set of ray tracing-based algorithms and applications, including interactive rendering, offline rendering, collision detection systems, artificial intelligence queries, and scientific simulations such as sound propagation. OptiX achieves high performance through a compact object model and application of several ray tracing-specific compiler optimizations. For ease of use it exposes a single-ray programming model with full support for recursion and a dynamic dispatch mechanism similar to virtual function calls.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; D.2.11 [Software Architectures]: Domain-specific architectures; I.3.1 [Computer Graphics]: Hardware Architectures—;

**Keywords:** ray tracing, graphics systems, graphics hardware

## 1 Introduction

To address the problem of creating an accessible, flexible, and efficient ray tracing system for many-core architectures, we introduce OptiX, a general purpose ray tracing engine. This engine combines a programmable ray tracing pipeline with a lightweight scene representation. A general programming interface enables the implementation of a variety of ray tracing-based algorithms in graphics and non-graphics domains, such as rendering, sound propagation, collision detection and artificial intelligence.

In this paper, we discuss the design goals of the OptiX engine as well as an implementation for NVIDIA Quadro®, GeForce®, and Tesla® GPUs. In our implementation, we compose domain-specific compilation with a flexible set of controls over scene hierarchy, acceleration structure creation and traversal, on-the-fly scene update, and a dynamically load-balanced GPU execution model. Although OptiX currently targets highly parallel architectures, it is applicable to a wide range of special- and general-purpose hardware and multiple execution models.

To create a system for a broad range of ray tracing tasks, several

\* e-mail: sparker@nvidia.com

trade-offs and design decisions led to the following contributions:

- **A general, low level ray tracing engine.** The OptiX engine focuses exclusively on the fundamental computations required for ray tracing and avoids embedding rendering-specific constructs. The engine presents mechanisms for expressing ray-geometry interactions and does not have built-in concepts of lights, shadows, reflectance, etc.
- **A programmable ray tracing pipeline.** The OptiX engine demonstrates that most ray tracing algorithms can be implemented using a small set of lightweight programmable operations. It defines an abstract ray tracing execution model as a sequence of user-specified programs. This model, when combined with arbitrary data stored with each ray, can be used to implement a variety of sophisticated rendering and non-rendering algorithms.
- **A simple programming model.** The OptiX engine provides the execution mechanisms that ray tracing programmers are accustomed to using and avoids burdening the user with the machinery of high-performance ray tracing algorithms. It exposes a familiar recursive, single-ray programming model rather than ray packets or explicit SIMD-style constructs. The engine abstracts any batching or reordering of rays, as well as algorithms for creating high-quality acceleration structures.
- **A domain-specific compiler.** The OptiX engine combines just-in-time compilation techniques with ray tracing-specific knowledge to implement its programming model efficiently. The engine abstraction permits the compiler to tune the execution model for available system hardware.
- **An efficient scene representation.** The OptiX engine implements an object model that uses dynamic inheritance to facilitate a compact representation of scene parameters. A flexible node graph system allows the scene to be organized for maximum efficiency, while still supporting instancing, level-of-detail and nested acceleration structures.

## 2 Related Work

While numerous high-level ray tracing libraries, engines and APIs have been proposed [Wald et al. 2007b], efforts to date have been focused on specific applications or classes of rendering algorithms, making them difficult to adapt to other domains or architectures. On the other hand, several researchers have shown how to map ray tracing algorithms efficiently to GPUs and the NVIDIA® CUDA™ architecture [Aila and Laine 2009; Horn et al. 2007; Popov et al. 2007], but these systems have focused on performance rather than flexibility.

CPU-based real-time ray tracing systems were first developed in the 1990’s on highly parallel supercomputers [Green and Paddon 1990; Muuss 1995; Parker et al. 1999]. Subsequent improvements in acceleration structures [Goldsmith and Salmon 1987; MacDonald and Booth 1989] and traversal techniques [Wald et al. 2001; Havran 2001; Reshetov et al. 2005; Wald et al. 2007a] enabled interactive ray tracing on desktop-class machines [Bigler et al. 2006; Georgiev and Slusallek 2008]. These systems were built using C and/or C++ programming languages with traditional object-oriented programming models, rather than the generic shader-based system described in this paper.

The RPU [Woop et al. 2005] is a special purpose hardware system for interactive ray tracing that provides some degree of programmability for using geometry, vertex and lighting shaders written in assembly language. Caustic Graphics [Caustic Graphics 2009] recently demonstrated a special purpose accelerator board but has not

published details about the mechanisms for programming shader programs.

OpenRT utilized a binary plug-in interface to provide surface, light, camera and environment shaders [Dietrich et al. 2003] but did not strive for the generality attempted here. Other interactive ray tracing systems such as Manta [Bigler et al. 2006], Razor [Djeu et al. 2007], and Arauna [Bikker 2007] also provide APIs that are system specific and not intended as general purpose solutions.

## 3 A Programmable Ray Tracing Pipeline

The core idea of the OptiX engine is that most ray tracing algorithms can be implemented using a small set of programmable operations. This is a direct analog to the programmable rasterization pipelines employed by OpenGL and Direct3D. At a high level, those systems expose an abstract rasterizer containing lightweight callbacks for vertex shading, geometry processing, tessellation, and fragment shading operations. An ensemble of these program types, typically used in multiple passes, can be used to implement a broad variety of rasterization-based algorithms.

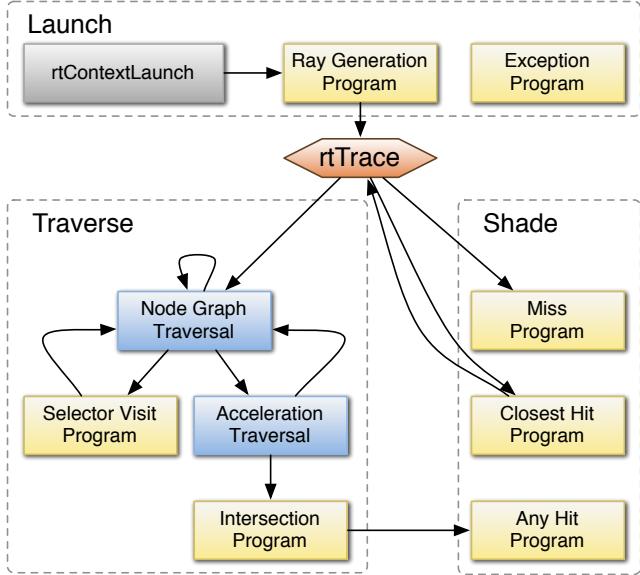
We have identified a corresponding abstract ray tracing execution model along with lightweight operations that can be customized to implement a wide variety of ray tracing-based algorithms. [NVIDIA 2010a]. These operations, or programs, can be combined with a user-defined data structure (*payload*) associated with each ray. The ensemble of programs conspire to implement a particular client application’s algorithm.

### 3.1 Programs

There are seven different types of programs in OptiX, each of which operates on a single ray at a time. In addition, a bounding box program operates on geometry to determine primitive bounds for acceleration structure construction. The combination of user programs and hardcoded OptiX kernel code forms the *ray tracing pipeline*, which is outlined in Figure 2. Unlike a feed-forward rasterization pipeline, it is more natural to think of the ray tracing pipeline as a call graph. The core operation, *rtTrace*, alternates between locating an intersection (*Traverse*) and responding to that intersection (*Shade*). By reading and writing data in user-defined ray payloads and in global device memory arrays (*buffers*, see section 3.5), these operations are combined to perform arbitrary computation during ray tracing.

**Ray generation** programs are the entry into the ray tracing pipeline. A single invocation of *rtContextLaunch* will create many instantiations of these programs. In the example in Figure 3, a ray generation program will create a ray using a pinhole camera model for a single pixel, start a trace operation, and store the resulting color in an output buffer. With this mechanism, one can also perform other operations such as creating photon maps, computing baked lighting, processing ray requests passed from OpenGL, shooting multiple rays for super-sampling, or implementing different camera models.

**Intersection** programs implement ray-geometry intersection tests. As the acceleration structures are traversed, the system will invoke an intersection program to perform the geometric query. The program determines if and where the ray touches the object and may compute normals, texture coordinates, or other attributes based on the hit position. An arbitrary number of attributes may be associated with each intersection. Intersection programs enable support for arbitrary surfaces such as spheres, cylinders, high-order surfaces, or even fractal geometries like the Julia set in Figure 1. However, even in a triangle-only system, one may encounter a wide



**Figure 2:** A call graph showing the control flow through the ray tracing pipeline. The yellow boxes represent user-specified programs and the blue boxes are algorithms internal to OptiX. Execution is initiated by the API call `rtContextLaunch`. A built-in function, `rtTrace`, can be employed by the ray generation program to cast rays into the scene. This function may also be called recursively by the closest hit program for shadow and secondary rays. The exception program is executed when the execution of a particular ray is terminated by an error such as excessive memory consumption.

variety of mesh representations. A programmable intersection operation facilitates direct access to the native format, which can help avoid copies when interoperating with rasterization-based systems.

**Bounding box** programs compute the bounds associated with each primitive to enable acceleration structures over arbitrary geometry. Given a primitive index, a simple program of this type may, for example, read vertex data from a buffer and compute a triangle’s bounding box. Procedural geometry can sometimes only estimate the bounds of a primitive. Such estimates are allowed as long as they are conservative, but loose bounds may degrade performance.

**Closest hit** programs are invoked once traversal has found the closest intersection of a ray with the scene geometry. This program type closely resembles surface shaders in classical rendering systems. Typically, a closest hit program will perform computations like shading, potentially casting new rays in the process, and store result data in the ray payload.

**Any hit** programs are called during traversal for every ray-object intersection that is found. The any hit program allows the material to participate in object intersection decisions while keeping the shading operations separate from the geometry operations. It may optionally terminate the ray using the built-in function `rtTerminateRay`, which will stop all traversal and unwind the call stack to the most recent invocation of `rtTrace`. This is a lightweight exception mechanism that can be used to implement early ray termination for shadow rays and ambient occlusion. Alternatively, the any hit program may ignore the intersection using `rtIgnoreIntersection`, allowing traversal to continue looking for other geometric objects. An intersection may be ignored, for instance, based on a texture channel lookup, thus implementing efficient alpha-mapped transparency without restarting traversal. An-

```
RT_PROGRAM void pinhole_camera() {
    Ray ray = PinholeCamera::makeRay( launchIndex );
    UserPayload payload;
    rtTrace( topObject, ray, payload );
    outputBuffer[launchIndex] = payload.result;
}
```

**Figure 3:** Example ray generation program (in CUDA C) for a single sample per pixel. The 2-dimensional grid location of the program invocation is given by the semantic variable `launchIndex`, which is used to create a primary ray using a pinhole camera model. Upon tracing a ray, the invoked material hit programs fill the result field of the user-defined payload structure. The variable `topObject` refers to the location in the scene hierarchy where ray traversal should start, typically the root of the node graph. At the location specified by `launchIndex`, the result is written to the output buffer to be displayed by the application.

other use case for the any hit program can be found in Section 8.1, where the application performs visibility attenuation for partial shadows cast by glass objects. Note that intersections may be presented out of order. The default any hit program is a no-op, which is often the desired operation.

**Miss** programs are executed when the ray does not intersect any geometry in the interval provided. They can be used to implement a background color or environment map lookup.

**Exception** programs are executed when the system encounters an exceptional condition, e.g., when the recursion stack exceeds the amount of memory available for each thread, or when a buffer access index is out of range. OptiX also supports user-defined exceptions that can be thrown from any program. The exception program can react, for example, by printing diagnostic messages or visualizing the condition by writing special color values to an output pixel buffer.

**Selector visit** programs expose programmability for coarse-level node graph traversal. For example, an application may choose to vary the level of geometric detail for parts of the scene on a per-ray basis. In this case, the visit program would examine the ray distance or a ray differential stored with the payload and make a traversal decision based on that data.

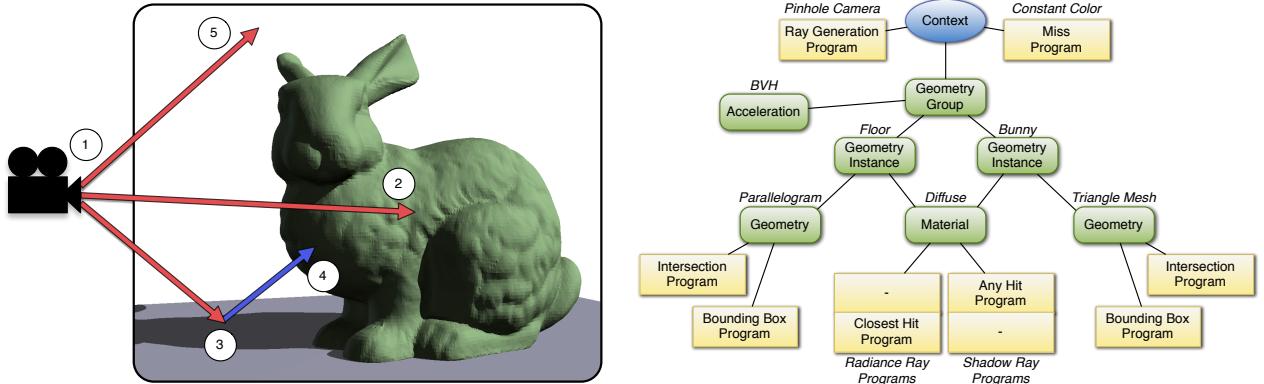
## 3.2 Scene representation

The OptiX engine employs a flexible structure for representing scene information and associated programmable operations, collected in a container object called the *context*. This representation is also the mechanism for binding programmable shaders to the object-specific data that they require. In conjunction with a special-purpose object model described in Section 3.3, a compact representation of scene data is achieved.

### 3.2.1 Hierarchy nodes

A scene is represented as a graph. This representation is very lightweight and controls the traversal of rays through the scene. It can also be used to implement instancing two-level hierarchies for animations of rigid objects, or other common scene structures. To support instancing and sharing of common data, the nodes can have multiple parents.

Four main node types can be used to provide the scene representation using a directed graph. Any node can be used as the root of scene traversal. This allows, for example, different representations to be used for different ray types.



**Figure 4:** Right: A complete OptiX context for a simple scene with a pinhole camera, two objects and shadows. The ray generation program implements the camera, while a miss program implements a constant white background. A single geometry group contains two geometry instances with a single BVH built over all underlying geometry in the triangle mesh and ground plane. Two types of geometry are implemented, a triangle mesh and a parallelogram, each with their own set of intersection and bounding box programs. The two geometry instances share a single material that implements a diffuse lighting model and fully attenuates shadow rays via closest hit and any hit programs, respectively. Left: Execution of the programs. 1. The ray generation program creates rays and traces them against the geometry group, which initiates BVH traversal. 2. If the ray intersects with geometry, the closest hit program will be called after the hit point is found. 3. The material will spawn shadow rays and trace them against scene geometry. 4. When an intersection along the shadow ray is found, the any hit program will terminate ray traversal and return to the calling program with shadow information. 5. If a ray does not intersect with any scene geometry, the miss program will be called.

**Group** nodes contain zero or more (but usually two or more) children of any node type. A group node has an acceleration structure associated with it and can be used to provide the top level of a two-level traversal structure.

**Geometry Group** nodes are the leaves of the graph and contain the primitive and material objects described below. This node type also has an acceleration structure associated with it. Any non-empty scene will contain at least one geometry group.

**Transform** nodes have a single child of any node type, plus an associated  $4 \times 3$  matrix that is used to perform an affine transformation of the underlying geometry.

**Selector** nodes have zero or more children of any node type, plus a single visit program that is executed to select among the available children. Although not implemented in the current version of the OptiX libraries, the node graph can be cyclic if the selector node is used carefully to avoid infinite recursion.

### 3.2.2 Geometry and material objects

The bulk of the data is stored in the geometry nodes at the leaves of the graph. These contain objects that define geometry and shading operations. They may also have multiple parents, allowing material and geometry information to be shared at multiple points in the graph; for a complete example, see Figure 4.

**Geometry Instance** objects bind a *geometry object* to a set of *material objects*. This is a common structure used by scene graphs to keep geometric and shading information orthogonal.

**Geometry** objects contain a list of geometric primitives. Each geometry object is associated with a bounding box program and an intersection program, both of which are shared among the geometry object's primitives.

**Material** objects hold information about shading operations, including programs called for each intersection as they are discovered

(any hit program) and for the intersection nearest to the origin of a given ray (closest hit program).

### 3.3 Object and data model

OptiX employs a special-purpose object model designed to minimize the constant data used by the programmable operations. In contrast to an OpenGL system, where only a single combination of shaders is used at a time. However, ray tracing can randomly access object and material data. Therefore, instead of the *uniform* variables employed by OpenGL shading languages, OptiX allows any of the objects and nodes described above to carry an arbitrary set of variables expressed as a typed name-value pair called a *variable*. Variables are set by the client application and have read-only access during the execution of a trace. Variables can be of scalar or vector integer and floating point types (e.g., *float3*, *int4*) as well as user-defined structs and references to buffers and texture samplers.

The inheritance mechanism for these variables is unique to OptiX. Instead of a class-based inheritance model with a single *self* or *this* pointer, the OptiX engine tracks the current geometry and material objects and the current traversal node. Variable values are inherited from the objects that are active at each point in the control flow. For example, an intersection program will inherit definitions from the *geometry* and *geometry instance* objects, in addition to global variables defined in the context. Conceptually, OptiX examines each of these objects for a matching name/value pair when a variable is accessed. This mechanism can be thought of as a generalization of nested scoping found in most programming languages. It can also be implemented quite efficiently in the just-in-time compiler.

As an example of how this is useful, consider an array of light sources called *lights*. Typically, a user would define *lights* in the context, the global scope of OptiX. This makes this value available in all shaders in the entire scene. However, if the lights for a particular object need to be overridden, another variable of the same name can be created and attached to the geometry instance associated with that object. In this way, programs connected to that

object would use the overridden value of *lights* rather than the value attached to the context. This is a powerful mechanism that can be used to minimize the scene data to enable high performance on architectures with minimal caches. The manner in which these cases are handled can vary dramatically from one renderer to another, so the OptiX engine provides the basic functionality to express any number of override rules efficiently.

A special type of variable, tagged with the keyword *attribute* can be used to communicate information from the intersection program to the closest- and any-hit programs. These are analogous to OpenGL *varying* variables, and are used for communicating texture coordinates, normals and other shading information from the intersection programs to the shading programs. These variables have special semantics — they are written by the intersection program but only the values associated with the closest intersection are kept. This mechanism enables the intersection operation to be completely separate from the shading operations, enabling multiple simultaneous primitives and/or mesh storage formats while still supporting texturing, shading normals, and object curvatures for ray differentials. Attributes that are not used by any closest- or any-hit program can be elided by the OptiX compiler.

### 3.4 Dynamic dispatch

To allow multiple ray-tracing operations to co-exist in a single execution, OptiX employs a user-defined *ray type*. A ray type is simply an index that selects a particular set of slots for *any hit* and *closest hit* programs to be executed when an intersection is found. This can be used, for example, to treat shadow rays separately from other rays.

Similarly, multiple *entry points* in an OptiX context enable an efficient way to represent different passes over the same set of geometry. For example, a photon mapper may use one entry point to cast photons into the scene and a second entry point to cast viewing rays.

### 3.5 Buffers and Textures

The key abstraction for bulk data storage is the multi-dimensional *buffer* object, which presents a 1-, 2- or 3-dimensional array of a fixed element size. A buffer is accessed through a C++ wrapper object in any of the programs. Buffers can be read-only, write-only or read-write and support atomic operations when supported by the hardware. A buffer is handle-based and does not expose raw pointers, thus enabling the OptiX runtime to relocate buffers for storage compaction, or for promotion to other memory spaces for performance. Buffers are typically used for output images, triangle data, light source lists, and other array-based data. Buffers are the sole means of outputting data from an OptiX program. In most applications, the ray generation program will be responsible for writing data to the output buffer, but any of the OptiX programs are allowed to write to output buffers at any location, but with no ordering guarantees.

A buffer can also be bound to a *texture sampler* object, which will utilize the GPU texturing hardware. Buffers and texture sampler objects are bound to OptiX variables and utilize the same scoping mechanisms as shader values. Additionally, both buffers and texture samplers can interoperate with OpenGL and DirectX, enabling efficient implementation of hybrid rasterization/ray-tracing applications.

## 4 System Overview

The OptiX engine consists of two distinct APIs, one for host-side and one for device-side code.<sup>1</sup> The *host API* is a set of C functions that the client application calls to create and configure a context, assemble a node graph, and launch ray tracing kernels. It also provides calls to manage devices used for kernel execution. The *program API* is the functionality exposed to user programs. This includes function calls for tracing rays, reporting intersections, and accessing data. In addition, several semantic variables encode state specific to ray tracing, e.g., the current distance to the closest intersection. Printing and exception handling facilities are also available for debugging.

Figure 5 outlines the control flow of an OptiX application. During setup, the application calls OptiX host API functions to provide scene data such as geometry, materials, acceleration structures, hierarchical relationships, and programs. A subsequent call to the *rtContextLaunch* API function passes control to OptiX, where changes in the context are processed. If required, a new ray tracing kernel is compiled from the given user programs. Acceleration structures are built (or updated) and data is synchronized between host and device memory. Finally, the ray tracing kernel is executed, invoking the various user programs as described in Section 3.

After execution of the ray tracing kernel has finished, its result data can be used by the application. Typically, this involves reading from output buffers filled by one of the user programs or displaying such a buffer directly, e.g., via OpenGL. An interactive or multi-pass application then repeats the process starting at context setup, where arbitrary changes to the context can be made, and the kernel is launched again.

## 5 Acceleration Structures

The core algorithm for finding an intersection between a ray and the scene geometry involves the traversal of acceleration structures. Such data structures are a vital component of virtually every ray tracing system. They are usually spatial or object hierarchies and are used by the traversal algorithm to efficiently search for primitives that potentially intersect a given ray. OptiX offers a flexible interface, suitable for a wide range of applications, to control its acceleration structures.

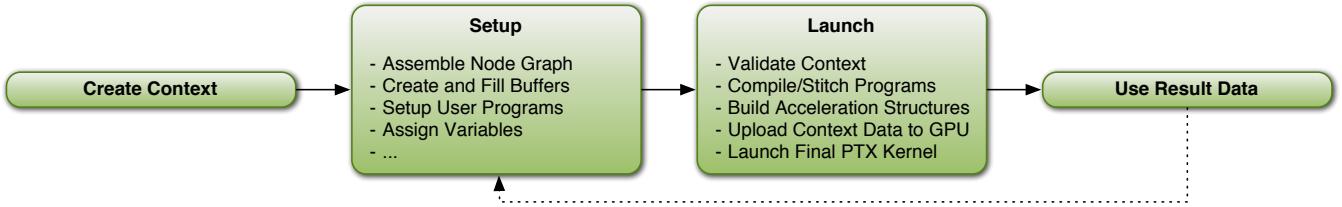
### 5.1 Interaction with the node graph

One of the reasons for collecting geometry data in a node graph is to facilitate the organization of the associated acceleration structures. Instead of maintaining all scene geometry within a single acceleration structure, it often makes sense to build several structures over different regions of the scene. For example, parts of the scene may be animated, requiring an acceleration structure to be rebuilt for every ray tracing pass. In this case, creating a separate structure for the static regions of the scene can increase efficiency. In addition to only constructing the static structure once, the application can typically invest a larger time budget into a higher quality build.

The OptiX engine associates acceleration structures with all groups and geometry groups in the node graph. Structures attached to geometry groups are low level, built over the geometric primitives the geometry group contains. Structures on groups are built over the bounds of the children of that group and thus represent high level

---

<sup>1</sup>We use the terms *host*, *device*, and *kernel* in the same way as commonly done in the CUDA environment: the *host* is the processor running the client application (usually a CPU). The *device* is the processor (usually a GPU) running the ray tracing code produced by OptiX, called the *kernel*.



**Figure 5:** Basic OptiX application control flow. The individual steps during context setup are controlled by the application, the launch procedure is handled by OptiX.

acceleration structures. These high level structures are useful to express hierarchical relationships between geometry that is modified at different rates.

**Instancing.** An important design goal for the acceleration structure system was support for flexible instancing. Here, instancing refers to low-overhead replication of scene geometry by referencing the same data more than once, without having to copy heavyweight data structures. As described in Section 3.2.1, nodes in the graph can be referenced multiple times, which naturally implements instancing. It is desirable to not only share geometry information among instances, but acceleration structures as well. At the same time, it should be possible to assign non-geometry data such as material programs and variables independently for each instance.

We chose to expose acceleration structures as separate API objects that are attached to groups and geometry groups. In the instancing case, it is possible to attach a single acceleration structure to multiple nodes, thus sharing its data and avoiding redundant construction of the same data structure. The method also results in efficient addition and removal of instances at runtime. Figure 6 shows an example of a node graph with instancing.

**Acceleration structures on combined geometry.** Dividing the scene into multiple acceleration structures reduces structure build time but also reduces ray traversal performance. In the limiting case of an entirely static scene, one would typically choose a single acceleration structure. One idea behind acceleration structures on geometry groups is to facilitate the application’s data management for that type of setup: instead of having to merge individual geometric objects into a monolithic chunk, they can stay organized as separate geometries and instances, and easily be collected within a single geometry group. The corresponding acceleration structure will be built over the individual primitives of any geometric objects, resulting in maximum efficiency as if all the geometry were combined. The OptiX engine will internally take care of the necessary

bookkeeping tasks, such as correct remapping of material indices.

A geometry group can also exploit certain per-object information when building its acceleration structure. For example, in a geometry group containing multiple objects, only a single one might have been modified between ray tracing passes. OptiX can take into account that information and omit some redundant operations (e.g. bounding box computations, see Section 5.3).

## 5.2 Types of acceleration structures

Ray tracing acceleration structures are an active area of research. There is no single type that is optimal for all applications under all conditions. The typical tradeoff between the different variants is ray tracing performance versus construction speed, and each application has a different optimal balance. Therefore, OptiX provides a number of different acceleration structure types that the application can choose from. Each acceleration structure in the node graph can be of a different type, allowing combinations of high-quality static structures with dynamically updated ones. Most types are also suitable for high level structures, i.e. acceleration structures attached to groups.

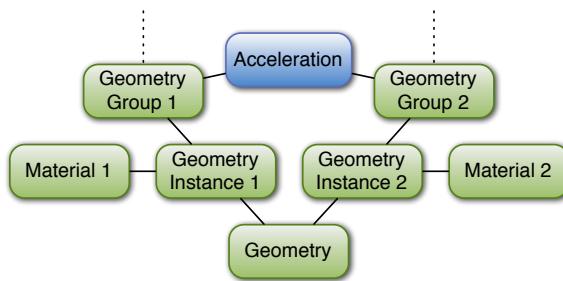
The currently implemented acceleration structures include algorithms focused on hierarchy quality (e.g. the SBVH [Stich et al. 2009]), on construction speed (e.g. the LBVH [Lauterbach et al. 2009]), and various balance levels in between.

## 5.3 Construction

Whenever the underlying geometry of an acceleration structure is changed, e.g. during an animation, it is explicitly marked for rebuild by the client application. OptiX then builds the so scheduled acceleration structures on the subsequent invocation of the *rtContextLaunch* API function.

The first stage in acceleration structure construction acquires the bounding boxes of the referenced geometry. This is achieved by executing for each geometric primitive in an object the bounding box program described in Section 3.1, which is required to return a conservative axis-aligned bounding box for its input primitive. Using these bounding boxes as elementary primitives for the acceleration structures provides the necessary abstraction to trace rays against arbitrary user-defined geometry (including several types of geometry within a single structure). To obtain the necessary bounding boxes for higher level group nodes in the tree, the union of the primitive bounding boxes is formed and propagated recursively.

The second construction stage consists of actually building the required acceleration structures given the obtained bounding boxes. The available host and device parallelism can be utilized in two ways. First, multiple acceleration structures in the node graph can be constructed in parallel, as they are independent. Second, a single acceleration structure build code can usually be parallelized (see e.g. [Shevtsov et al. 2007], [Zhou et al. 2008], [Lauterbach et al.



**Figure 6:** Node graph with instancing. Both geometry groups reference the same geometry object and share an acceleration structure, but use different materials. Geometry data is not duplicated.

2009]). The final acceleration structure data is placed in device memory for consumption by the ray traversal code.

## 5.4 Tuning

While acceleration structures in the OptiX engine are designed to perform well out of the box, it is sometimes necessary for the application to provide additional information to achieve the highest possible performance. The application can therefore set acceleration structure-specific properties that affect subsequent structure builds and ray traversals.

One example for such a property is the “refit” flag: if the geometry used by a BVH acceleration structure has changed only slightly, it is often sufficient to simply refit the BVH’s internal bounding boxes instead of rebuilding the full structure from scratch (see [Lauterbach et al. 2006]). The client application can enable this behavior on certain types of acceleration structures if it assumes the resulting total runtime will decrease. Such decisions are left to the application, as it usually possesses contextual information that is unavailable to OptiX.

Build procedures specialized to certain types of geometric primitives (as opposed to the axis-aligned bounding boxes discussed above) are a second case where properties are useful. The application may, for example, inform an SBVH acceleration structure that the underlying geometry consists exclusively of triangles, and where these triangles are located in memory. The SBVH can then perform a more exact method of constructing the hierarchy, which results in higher quality.

## 6 Domain-Specific Compilation

The core of the OptiX host runtime is a Just-In-Time (JIT) compiler that provides several important pieces of functionality. First, the JIT stage combines all of the user-provided shader programs into one or more kernels. Second, it analyzes the node graph to identify data-dependent optimizations. Third, it provides a domain-specific Application Binary Interface (ABI) and execution model that implements recursion and function pointer operations on a device that does not naturally support them. Finally, the resulting kernel is executed on the GPU using the CUDA driver API.

### 6.1 OptiX programs

User-specified programs, often called a shader, are provided to the OptiX host API in the form of Parallel Thread Execution (PTX) functions [NVIDIA 2010b]. PTX is a virtual machine assembly language that is part of the CUDA architecture. It implements a low-level virtual machine, similar in many ways to the popular open source Low-Level Virtual Machine (LLVM) intermediate representation [Lattner and Adve 2004]. Like LLVM, PTX defines a set of simple instructions that provide basic operations for arithmetic, control flow and memory access. PTX also provides several higher-level operations such as texture access and transcendental operations. Also similar to LLVM, PTX assumes an infinite register file and abstracts many real machine instructions. A JIT compiler in the CUDA runtime will perform register allocation, instruction scheduling, dead-code elimination, and numerous other late optimizations as it produces machine code targeting a particular GPU architecture.

PTX is written from the perspective of a single thread and thus does not require explicit lane mask manipulation operations. This makes it straightforward to lower PTX from a high-level shading language, while giving the OptiX runtime the ability to manipulate and optimize the resulting code. While PTX also provides parallel

synchronization and communication instructions, these instructions are neither necessary for nor allowed by the OptiX runtime.

NVIDIA’s CUDA C/C++ compiler, *nvcc*, emits PTX and is currently the preferred mechanism for programming OptiX. Programs are compiled offline using *nvcc* and submitted to the OptiX API via a PTX string. By leveraging the CUDA C++ compiler, OptiX shader programs have a rich set of programming language constructs available including pointers, templates and overloading that come automatically by using C++ as the input language. A set of header files is provided that support the necessary variable annotations and pseudo-instructions for tracing rays and other OptiX operations. These operations are lowered to PTX in the form of a *call* instruction that gets further processed by the OptiX runtime.

While this provides a powerful C++-based shading language, it may not be useful in all applications. Alternatively, any compiler frontend that can emit PTX could be used. One could imagine frontends for Cg, HLSL, GLSL, MetaSL, OpenSL, RSL, GSL, OpenCL, etc., that could produce appropriate PTX for input into OptiX. In this manner, OptiX is shading-language agnostic, since multiple syntax variants could be used to generate programs for use with the runtime API.

### 6.2 PTX to PTX compilation

Given the set of PTX functions for a particular scene, the OptiX compiler rewrites the PTX using multiple PTX to PTX transformation passes, which are similar to the compiler passes that have proven successful in the LLVM infrastructure. In this manner, OptiX uses PTX as an intermediate representation rather than a traditional instruction set. This process implements a number of domain-specific operations including an ABI (calling sequence), link-time optimizations, and data-dependent optimizations. The fact that most data structures in a typical ray tracer are read-only provides a substantial opportunity for optimizations that would not be considered safe in a more general environment.

**Analysis.** The first stage of this process is to perform a static analysis of all of the PTX functions provided. This pass ensures that the variables referenced in each function have been provided by the node graph and are of consistent types. At the same time, we determine whether each of the data buffers is read-only or read-write to inform the runtime where the data should be stored. Finally, this pass can analyze the structure of the node graph in preparation for other data-specific optimizations shown below.

**Inline intrinsic operations.** The OptiX runtime provides several operations beyond the ones provided by CUDA. These instructions are replaced with an inlined function that implements the requested operations. Examples include access to the currently active ray origin, direction and payload, a read-write surface store abstraction, and accessing the transform stack. In addition, we process pseudo-instructions corresponding to exceptional control flow such as *rtTerminateRay* and *rtIgnoreIntersection*.

**Shader variable object model.** A program can reference a shader variable without additional syntax, just as a member variable would be accessed in C++. These accesses will manifest in PTX as a load instruction associated with specially tagged global variables. We detect accesses to these variables using a dataflow analysis pass and replace them with a load indexed from a pointer to the current geometry, material, instance or other API object as determined by the analysis pass. To implement dynamic inheritance of variables, a small table associated with each object determines the base pointer and associated offset.

---

```

for( int i = 0; i < 5; ++i ) {
    Ray ray = make_Ray( make_float3( i, 0, 0 ),
                        make_float3( 0, 0, 1 ),
                        0, 1e-4f, 1e20f );
    UserPayloadStruct payload;
    rtTrace( top_object, ray, payload );
}

```

---

**Figure 7:** A simple CUDA C program snippet that calls *rtTrace*, a function that requires a continuation, in a loop.

---

```

ld.global.u32    %node, [top_object+0];
mov.s32          %i, 0;
loop:
    call _rt_trace, ( %node, %i, 0, 0, 0, 0, 1,
                      0, 1e-4f, 1e20f, payload );
    add.s32          %i, %i, 1;
    mov.u32          %iend, 5;
    setp.ne.s32      %predicate, %i, %iend;
    @%predicate bra  loop;

```

---

**Figure 8:** PTX code corresponding to the program in Figure 7. The register *%i* is live across the call to *rtTrace*. Therefore, the continuation mechanism must restore it after the call returns.

**Continuations.** Consider the shader program shown in Figure 7 and the corresponding PTX shown in Figure 8. This program implements a simple loop to trace 5 rays from points (0,0,0), (1,0,0), (2,0,0), (3,0,0) and (4,0,0). While not a useful program, this example can be used to illustrate how continuations are used. To allow this loop to execute as expected, the variable *i* must be saved before temporarily abandoning the execution of this program to invoke the *rtTrace* function.

This is accomplished by implementing a backward dataflow analysis pass to determine the PTX registers that are live when the pseudo-instruction for *rtTrace* is encountered. A live register is one that is used as an argument for some subsequent instruction in the dataflow graph. We reserve slots on the stack for each of these variables, pack them into 16-byte vectors where possible, and store them on the stack before the call and restore them after the call. This is similar to a caller-save ABI that a traditional compiler would implement for a CPU-based programming language. In preparation for introducing continuations, we perform a loop-hoisting pass and a copy-propagation pass on each function to help minimize the state saved in each continuation.

Finally, the *rtTrace* pseudo-instruction is replaced with a branch to return execution to the state machine described below, and a label that can be used to eventually return control flow to this function. This transformation results in the pseudo-code shown in Figure 9. However, the non-structural *gotos* in this code will result in an irreducible control flow graph due to entering the loop both at the top of the loop and the *state2* label.

Irreducible control flow thwarts the mechanisms in the GPU to control the SIMD execution of this function, resulting in a dramatic slowdown for divergent code. Consequently, we split this function by cloning the nodes in the graph for each state. After performing dead-code elimination, the code sequence in Figure 10 is obtained. This control flow is more friendly to SIMD execution because it is well-structured. Divergence can be further reduced by introducing new states around the common code. This final transformation may or may not be worthwhile, depending on the cost of switching states and the degree of execution divergence.

---

```

state1:
    for( int i = 0; i < 5; ++i ) {
        Ray ray = make_Ray( ..., i, ... );
        UserPayloadStruct payload;
        push i;
        state = trace;
        goto mainloop;
    state2:
        pop i;
    }

```

---

**Figure 9:** Pseudo-code for the program in Figure 7 with inserted continuation.

---

```

state1:
    i = 0;
    Ray ray = make_Ray( ..., i, ... );
    UserPayloadStruct payload;
    push i;
    state = trace;
    goto mainloop;

state2:
    pop i;
    ++i;
    if( i > 5 ) {
        state = returnState;
        goto mainloop;
    }
    Ray ray = make_Ray( ..., i, ... );
    UserPayloadStruct payload;
    state = trace;
    goto mainloop;

```

---

**Figure 10:** Pseudo-code for the program in Figure 7 with continuation and a split to regain control flow graph reducibility.

### 6.3 Optimization

The OptiX compiler infrastructure provides a set of domain-specific and data-dependent optimizations that would be challenging to implement in a statically compiled environment. These include (performance increases for a variety of applications in parentheses):

- Elide transformation operations for node graphs that do not utilize a transformation node (up to a 7% performance improvement).
- Eliminate printing and exception related code if these options are not enabled in the current execution.
- Reduce continuation size by regenerating constants and intermediates after a restore. Since the OptiX execution model guarantees that object-specific variables are read-only, this local optimization does not require an interprocedural pass.
- Specialize traversal based on tree characteristics such as existence of degenerate leaves, degenerate trees, shared acceleration structure data, or mixed primitive types.
- Move small read-only data to constant memory or textures if there is available space (up to a 29% performance improvement).

Furthermore, the rewrite passes can introduce substantial modifications to the code, which can be cleaned up by additional standard optimization passes such as dead-code elimination, constant propagation, loop-hoisting, and copy-propagation.

---

```

state = initialState;
while( state != DONE )
    switch(state) {
        case 1:      state = program1();      break;
        case 2:      state = program2();      break;
        ...
        case N:      state = programN();     break;
    }
}

```

---

**Figure 11:** Pseudo-code for a simple state machine approach to megakernel execution. The state to be selected next is chosen by a switch statement. The switch is executed repeatedly until the state variable contains a special value that indicates termination.

## 7 Execution Model

Various authors have proposed different execution models for parallel ray tracing. In particular, the monolithic kernel, or *megakernel*, approach proves successful on modern GPUs [Aila and Laine 2009]. This approach minimizes kernel launch overhead but potentially reduces processor utilization as register requirements grow to the maximum across constituent kernels. Because GPUs mask memory latency with multi-threading, this is a delicate tradeoff. OptiX implements a megakernel by linking together a set of individual user programs and traversing the state machine induced by execution flow between them at runtime.

As GPUs evolve, different execution models may become practical. For example, a streaming execution model [Gribble and Ramani 2008] may be useful on some architectures. Other architectures may provide hardware support for acceleration structure traversal or other common operations. Since the OptiX engine does not prescribe an execution order between the roots of the ray trees, these alternatives could be targeted with a rewrite pass similar to the one we presently use to generate a megakernel.

### 7.1 Megakernel execution

A straightforward approach to megakernel execution is simple iteration over a switch-case construct. Inside each case, a user program is executed and the result of this computation is the case, or state, to select on the next iteration. Within such a state machine mechanism, OptiX may implement function calls, recursion, and exceptions.

Figure 11 illustrates a simple state machine. The program states are simply inserted into the body of the switch statement. The state index, which we call a *virtual program counter* (*VPC*), selects the program snippet that will be executed next. Function calls are implemented by setting the VPC directly, virtual function calls are implemented by setting it from a table, and function returns simply restore the state to the continuation associated with a previously active function (the virtual return address). Furthermore, special control flow such as exceptions manipulate the VPC directly, creating the desired state transition in a manner similar to a lightweight version of the *setjmp / longjmp* functionality provided by C.

### 7.2 Fine-grained scheduling

While the straightforward approach to megakernel execution is functionally correct, it suffers serialization penalties when the state diverges within a single SIMT unit [Lindholm et al. 2008]. To mitigate the effects of execution divergence, the OptiX runtime uses a fine-grained scheduling scheme to reclaim divergent threads that would otherwise lay dormant. Instead of allowing the SIMT hardware to automatically serialize a divergent switch's execution,

---

```

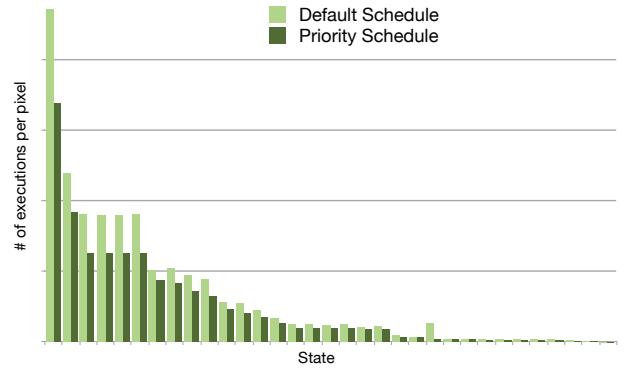
state = initialState;
while( state != DONE ) {
    next_state = scheduler();
    if(state == next_state)
        switch(state) {
            // Insert cases here as before
        }
}

```

---

**Figure 12:** Pseudo-code for megakernel execution through a state machine with fine-grained scheduling.

### SIMD Scheduling



**Figure 13:** The benefit of fine-grained scheduling with prioritization. Bars represent the number of state executions per pixel. A substantial reduction can be seen by scheduling the state transitions with a fixed priority, as described in Section 7.2.

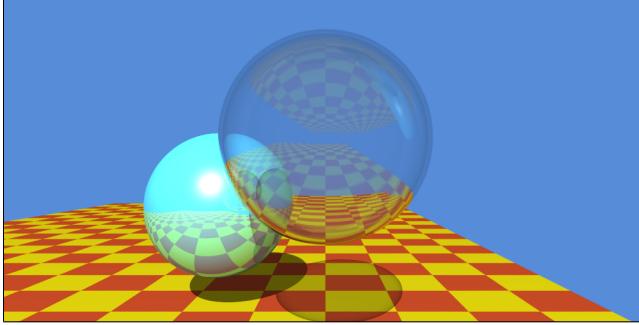
OptiX explicitly selects a single state for an entire SIMT unit to execute using a scheduling heuristic. Threads within the SIMT unit that do not require the state simply idle that iteration. The mechanism is outlined in Figure 12.

We have experimented with a variety of fine-grained scheduling heuristics. One simple scheme that works well determines a schedule by assigning a static prioritization over states. By scheduling threads with like states during execution, OptiX reduces the number of total state transitions made by a SIMT unit, which can substantially decrease execution time over the automatic schedule induced by the serialization hardware. Figure 13 shows an example of such a reduction.

### 7.3 Load balancing

In addition to minimizing SIMT execution divergence with a fine-grained scheduler, OptiX employs a three-tiered dynamic load balancing approach on GPUs. Each ray tracing kernel launch is presented as a queue of data parallel tasks to the physical execution units. The current execution model enforces independence between these tasks, enabling the load balancer to dynamically schedule work based on the characteristics of the workload and the execution hardware.

Work is distributed from the CPU host to one or more GPUs dynamically to enable coarse-grained load balancing between GPUs of differing performance. Once a batch of work has been submitted to a GPU, it is placed in a global queue. Each execution unit on the GPU is assigned a local queue that is filled from the GPU's global queue and dynamically distributes work to individual processing elements when they have completed their current job. This is an ex-



**Figure 14:** Recreation of Whitted’s sphere scene with user-specified programs: sphere and rectangle intersection; glass, procedural checker, and metal hit programs; sky miss program; and pinhole camera with adaptive anti-aliasing ray generation. Runs at over 30 fps on a GeForce GTX480 at 1k by 1k resolution.

tension of the scheme used by [Aila and Laine 2009] to incorporate dynamic load balancing between GPUs.

## 8 Application Case Studies

This section presents various use cases of OptiX by discussing the basic ideas behind a number of different applications.

### 8.1 Whitted-style ray tracing

The OptiX SDK contains several example ray tracing applications. One of these is an updated recreation of Whitted’s original sphere scene [1980]. This scene is simple, yet demonstrates important features of the OptiX engine.

The sample’s ray generation program implements a basic pinhole camera model. The camera position, orientation, and viewing frustum are specified by a set of program variables that can be modified interactively. The ray generation program begins the shading process by shooting a single ray per pixel or, upon user request, performing adaptive antialiasing via supersampling. The material *closest hit* programs are then responsible for recursively casting rays and computing a shaded sample color. After returning from the recursion, the ray generation program accumulates the sample color, stored in the ray payload, into an output buffer.

The application defines three separate pairs of intersection and bounding box programs, each implementing a different geometric primitive: a parallelogram for the floor, a sphere for the metal ball, and a thin-shell sphere for the hollow glass ball. The glass ball could have been modeled with two instances of the plain sphere primitive, but the flexibility of the OptiX program model gives us the freedom to implement a more efficient specialized version for this case. Each intersection program sets several attribute variables: a geometric normal, a shading normal, and, if appropriate, a texture coordinate. The attributes are utilized by material programs to perform shading computations.

The ray type mechanism is employed to differentiate radiance from shadow rays. The application attaches a trivial program that immediately terminates a ray to the materials’ *any hit* slots for shadow rays. This early ray termination yields high efficiency for mutual visibility tests between a shading point and the light source. The glass material is an exception, however: here, the *any hit* program is used to attenuate a visibility factor stored in the ray payload. As a result, the glass sphere casts a subtler shadow than the metal sphere.

---

```

float3 throughput = make_float3( 1, 1, 1 );
payload.nextRay = camera.getPrimaryRay();
payload.shootNextRay = true;

while( payload.shootNextRay == true ) {
    rtTrace( payload.nextRay, payload );
    throughput *= payload.throughput;
}
sampleContribution = payload.lightColor * throughput;

```

---

**Figure 15:** Pseudo-code for iterative path tracing in Design Garage.

### 8.2 NVIDIA Design Garage

NVIDIA Design Garage is a sophisticated interactive rendering demonstration intended for public distribution. The top image of Figure 1 was rendered using this software. The core of Design Garage is a physically-based Monte Carlo path tracing system [Kajiya 1986] that continuously samples light paths and refines an image estimate by integrating new samples over time. The user may interactively view and edit a scene as an initial noisy image converges to the final solution.

To control stack utilization, Design Garage implements path tracing using iteration within the ray generation program rather than recursively invoking *rtTrace*. The pseudocode of Figure 15 summarizes.

In Design Garage, each material employs a closest hit program to determine the next ray to be traced, and passes that back up using a specific field in the ray payload. The closest hit program also calculates the throughput of the current light bounce, which is used by the ray generation to maintain the cumulative product of throughput over the complete light path. Multiplying the color of the light source hit by the last ray in the path yields the final sample contribution.

OptiX’s support for C++ in ray programs allows materials to share a generic closest hit implementation parameterized upon a BSDF type. This allows us to implement new materials as BSDF classes with methods for importance sampling as well as BSDF and probability density evaluation. Design Garage implements a number of different physically-based materials, including metal and automotive paint. Some of these shaders support normal and specular maps.

While OptiX implements all ray tracing functionality of Design Garage, an OpenGL pipeline implements final image reconstruction and display. This pipeline performs various post processing stages such as tone mapping, glare, and filtering using standard rasterization-based techniques.

### 8.3 Image Space Photon Mapping

Image Space Photon Mapping (ISPM) [McGuire and Luebke 2009] is a real-time rendering algorithm that combines ray tracing and rasterization strategies (Figure 16). We ported the published implementation to the OptiX engine. That process gives insight into the differences between a traditional vectorized serial ray tracer and OptiX.

The ISPM algorithm computes the first segment of photon paths from the light by rasterizing a “bounce map” from the light’s reference frame. It then propagates photons by ray tracing with Russian Roulette sampling until the last scattering event before the eye. At each scattering event, the photon is deposited into an array that is the “photon map”. Indirect illumination is then gathered in image space by rasterizing a small volume around each photon from the



**Figure 16:** ISPM real-time global illumination. A recursive closest hit program in OptiX implements the photon trace.

eye’s viewpoint. Direct illumination is computed by shadow maps and rasterization.

Consider the structure of a CPU-ISPM photon tracer. It launches one persistent thread per core. These threads process photon paths from a global, lockless work queue. ISPM photon mapping generates incoherent rays, so traditional packet strategies for vectorizing ray traversal do not help with this process. For each path, the processing thread enters a while-loop, depositing one photon in a global, lockless photon array per iteration. The loop terminates upon photon absorption.

Under OptiX-ISPM we also maintain global lockless input and output buffers. Trace performance increases with the success of fine-grain scheduling of programs into coherent SIMD units and decreases with the size of state communicated between programs. Mimicking a traditional CPU-style of software architecture would be inefficient under OptiX because it would require passing all material parameters between the ray generation and hit programs and a variable iteration while-loop in the closest hit program. OptiX-ISPM therefore follows an alternative design that treats all propagation iterations as co-routines. It contains a single ray generation program with one thread per photon path. A recursive closest hit program implements the propagate-and-deposit iterations. This allows threads to yield between iterations so that the fine-grained scheduler can regroup them.

We note that the broad approach taken here is a way of unifying a raster graphics API like OpenGL or DirectX with ray tracing primitives without extending the raster API. Deferred shading is a form of yielding, where the geometry buffers are like a functional-programming *continuation* that holds the state of an interrupted pixel shader. Those buffers are treated as input by the OptiX API. It writes results out to another buffer, and we then effectively resume the shading process by rendering volumes over the geometry buffers with a new pixel shader.

#### 8.4 Collision Detection

OptiX is intended to be useful for non-rendering applications as well. The center panel in Figure 1 shows an OpenGL visualization from a collision detection and line-of-sight engine built on the OptiX engine. In this example, the engine is simulating 4096 mov-

	Scene Triangles	Conference 283 k	Fairy Forest 174 k	Sibenik 80 k
GTX285	Aila-Laine (Manual)	Primary 137 Mrays/s A.O. 120	78 Mrays/s 89	112 Mrays/s 99
	OptiX (Compiled)	Primary 91 A.O. 72	59 45	86 65
	<b>Compilation penalty</b>		<b>34% to 40%</b>	<b>24% to 49%</b>
GTX480	Aila-Laine (Manual)	Primary 252 Mrays/s A.O. 193	143 Mrays/s 140	222 Mrays/s 173
	OptiX (Compiled)	Primary 192 A.O. 129	103 78	161 114
	<b>Compilation penalty</b>		<b>24% to 33%</b>	<b>28% to 44%</b>
				<b>27% to 34%</b>

**Table 1:** The cost of OptiX API flexibility and abstraction is a reduction in performance compared to a domain-specific hand-optimized GPU ray tracer [Aila and Laine 2009]. On our benchmark scenes, this penalty is about 25-35% of peak Mrays/s as of the time of this writing.

ing objects, tracing rays against a static 1.1 million polygon scene. The engine traces 512 collision probe rays from each object center using a closest hit program, and  $4096^2 / 2$  line-of-sight rays between all pairs of objects using an any hit program. Including time to process the collision results and perform object dynamics, the engine achieves 25 million rays/second on GeForce GTX 280 and 48 million rays per second on GTX 480. While a ray casting approach is not robust to all collision operations, it is an oft-used technique because of its simplicity.

## 9 Performance Results

All results in this section were rendered at HD 1080p ( $1920 \times 1080$ ) resolution. To evaluate the basic performance reachable by OptiX kernels, we recreated some of the experiments performed in [Aila and Laine 2009] using the same scenes and camera positions. We compared our generated kernels against these manually optimized kernels to measure the overhead created by software abstraction layers. We measured raw ray tracing and intersection times, ignoring times for scene setup, kernel compilation, acceleration structure builds, buffer transfers, etc. Equivalent acceleration structures and timing mechanisms were used in both systems. Table 1 shows the

	<i>Whitted-style</i>	<i>Path Tracing</i>
FX5800	1.0 fps	0.3 fps
2 x FX5800	2.0	0.6
GTX480	4.5	1.5

**Table 2:** Design Garage application performance at HD 1080p for a 910 k-triangle sports car scene, on a variety of GPU configurations. Frame rates include ray tracing, shading, and postprocessing. The path traced result is shown in Figure 1 (top).

results for runs on NVIDIA GeForce GTX 285 and GeForce GTX 480 GPUs averaged over the same 5 viewpoints used in the original paper. While, as expected, the flexibility and programmability of OptiX comes at a price, the performance gap is still acceptable. The largest gap exists for ambient occlusion rays, which is partially due to a remaining deficiency in the benchmark. In particular, we did not perform ray sorting and used a lower number of secondary rays per pixel for our measurements.

Table 2 shows performance numbers for Design Garage (see Section 8.2) on NVIDIA Quadro FX5800 and GeForce GTX 480 GPUs, which is more indicative of a real scene than the above test. This application is challenging for several reasons. First, it is a physically-based path tracing code with complex sampling, multiple materials, and many other features. This results in a large kernel that requires many registers, thus reducing the number of threads that can run in parallel on the GPU. Second, threads are more likely to diverge early due to diffuse or glossy light bounces that result in different material shaders being executed, causing reduced SIMD efficiency. Third, a division of scene geometry into multiple acceleration structures (to support animation) additionally increases the number of operations for ray traversal compared to a monolithic data structure. Nevertheless, the OptiX engine can successfully combine all of these various programs and still make Design Garage fast enough to offer interactive modification of the scene and convergence to a photorealistic image within seconds.

We also compared the OptiX-ISPM implementation to the published CPU implementation on an Intel Core 2 Quad computer with a GTX485 GPU rendering at HD 1080p resolution. We evaluated 20 scenes, including the “Sponza atrium” and “NS2” [McGuire and Luebke 2009]. Table 3 summarizes performance results for four representative scenes. All were rendered with  $4 \times 4$  subsampling in the global gathering step. *Local illumination time* includes geometry buffers and shadow maps. *I/O time* measures the data transfers between OpenGL and CPU or CUDA memory. *Net time* is Local + Global + Trace + I/O. The typical speedup was about  $4 \times$  for the trace and  $2.5 \times$  overall. “NS2” yielded the lowest net speedup, with the OptiX photon trace  $3.0 \times$  faster than the CPU one and the net time  $1.8 \times$  faster. Note that being on the same side of the PCI bus is as important as computational performance. Avoiding CPU-GPU data transfer can reduce I/O time by as much as 50%. Improving data exchange efficiency between the two APIs will further reduce the cost of this data transfer.

## 10 Limitations and Future Work

Currently, the OptiX runtime supports double-precision operations in programs, but rays are stored in single-precision. For some applications it would be desirable to have a double-precision ray. Extensions to the OptiX buffer mechanism would make some applications easier, such as operations for append, reductions, and sorting values. In some applications, a dynamic memory allocation mechanism may also be required.

Scene	Sponza	Frog	Ironworks	NS2
Tris	66 k	104 k	178 k	405 k
Emitted Photons	262 k	121 k	200 k	100 k
Photon Hits	489 k	183 k	390 k	178 k
McGuire-Luebke (Quad Core2)				
Trace time	<b>106 ms</b>	<b>56 ms</b>	<b>119 ms</b>	<b>37 ms</b>
I/O time	44	22	43	14
OptiX (GTX480)				
Trace time	<b>15</b>	<b>12</b>	<b>17</b>	<b>12</b>
I/O time	17	10	14	5
OpenGL				
Local Illum.	13	16	23	15
Global Gather	13	6	20	9
OptiX Trace Speedup	<b>4.7 x</b>	<b>3.6 x</b>	<b>5.2 x</b>	<b>3.0 x</b>
OptiX Net Speedup	3.0 x	2.3 x	2.8 x	1.8 x

**Table 3:** Comparison of trace time and OpenGL↔ray tracer data transfer at HD 1080p resolution for CPU [McGuire and Luebke 2009] and OptiX ISPM implementations. Both have the same local illumination and global gather times. The OptiX photon trace is about  $2.5 \times$  faster than the CPU one.

As with most compilers, there are endless opportunities for additional optimization passes that will be added as we gain experience with the system on important applications. In addition, it would be interesting to see additional shading languages target OptiX through PTX.

OptiX acceleration structures are built using the bounding box program or a special API that supports only triangle data. To create better acceleration structures for programmable geometry, it would be beneficial to generalize the acceleration structure builds to allow additional programmable operations. This might include a user-defined box/primitive overlap test among other operations. OptiX supports several types of acceleration structures, but currently does not provide a mechanism for the user to implement their own.

## 11 Conclusion

The OptiX system provides a general-purpose and high performance ray tracing API. OptiX balances ease of use with performance by presenting a simple programming model, based on a programmable ray tracing pipeline for single-ray user programs, that can be compiled into an efficient self-scheduling megakernel. Thus the heart of OptiX is a JIT compiler that processes *programs*, snippets of user-specified code in the PTX language. OptiX associates these programs with nodes in a graph that defines the geometric configuration and acceleration data structures against which rays are traced. Our contributions include a low-level ray tracing API and associated programming model, the concept of a programmable ray tracing pipeline and the associated set of program types, a domain-specific JIT compiler that performs the megakernel transformations and implements several domain-specific optimizations, and a lightweight scene representation that lends itself to high-performance ray tracing and supports, but does not restrict, the structure of the application scene graph. The OptiX ray tracing engine is a shipping product and already supports a wide range of applications. We illustrate the broad applicability of OptiX with multiple examples ranging from simplistic to fairly complex.

## Acknowledgments

The car, frog, and engine model in Figure 1 are courtesy of TurboSquid. The bunny model in Figures 16 and 4 is courtesy of the Stanford University Graphics Lab. Phil Miller was instrumental in keeping the effort on track. The authors appreciate valuable comments from Dr. Greg Humphreys and benefited greatly from groundwork and numerous conversations on ray tracing with members of NVIDIA Research and the SceniX team.

## References

- AILA, T., AND LAINE, S. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics 2009*, 145–149.
- BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 187–196.
- BIKKER, J. 2007. Real-time Ray Tracing Through the Eyes of a Game Developer. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, 1–10.
- CAUSTIC GRAPHICS, 2009. Introduction to CausticRT. [http://www.caustic.com/pdf/Introduction\\_to\\_CausticRT.pdf](http://www.caustic.com/pdf/Introduction_to_CausticRT.pdf).
- DIETRICH, A., WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, 23–31.
- DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. 2007. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. TR-07-52.
- GEORGIEV, I., AND SLUSALLEK, P. 2008. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*.
- GOLDSMITH, J., AND SALMON, J. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications* 7, 5, 14–20.
- GREEN, S. A., AND PADDON, D. J. 1990. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer* 6, 2, 62–73.
- GRIBBLE, C. P., AND RAMANI, K. 2008. Coherent Ray Tracing via Stream Filtering. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 59–66.
- HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.
- HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 167–174.
- KAJIYA, J. T. 1986. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 143–150.
- LATTNER, C., AND ADVE, V. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the 2004 International Symposium on Code Generation and Optimization*.
- LAUTERBACH, C., EUI YOON, S., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 39–45.
- LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum (Proceedings of Eurographics)* 28, 2, 375–384.
- LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 39–55.
- MACDONALD, J. D., AND BOOTH, K. S. 1989. Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface*, 152–63.
- MCGUIRE, M., AND LUEBKE, D. 2009. Hardware-Accelerated Global Illumination by Image Space Photon Mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics*.
- MUSS, M. J. 1995. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*.
- NVIDIA, 2010. NVIDIA OptiX Ray Tracing Engine Programming Guide Version 2.0. <http://developer.nvidia.com/object-optix-home.html>.
- NVIDIA, 2010. PTX: Parallel Thread Execution ISA Version 2.0. [http://developer.download.nvidia.com/compute/cuda/3\\_0/toolkit/docs/ptx\\_isa\\_2.0.pdf](http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf).
- PARKER, S. G., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B. E., AND HANSEN, C. D. 1999. Interactive Ray Tracing. In *SISD*, 119–126.
- POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum* 26, 3 (Sept.). (Proceedings of Eurographics), to appear.
- RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 1176–1185.
- SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum* 26, 3, 395–404.
- STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics 2009*, 7–13.
- WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive Rendering with Coherent Ray Tracing. In *Computer Graphics Forum (Proceedings of Eurographics 2001)*, vol. 20.
- WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* 26, 1.
- WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the Art in Ray Tracing Animated Scenes. In *STAR Proceedings of Eurographics 2007*, 89–116.
- WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6, 343–349.
- WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics (Proceeding of ACM SIGGRAPH)* 24, 3, 434–444.
- ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-Time KD-Tree Construction on Graphics Hardware. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)*, 1–11.