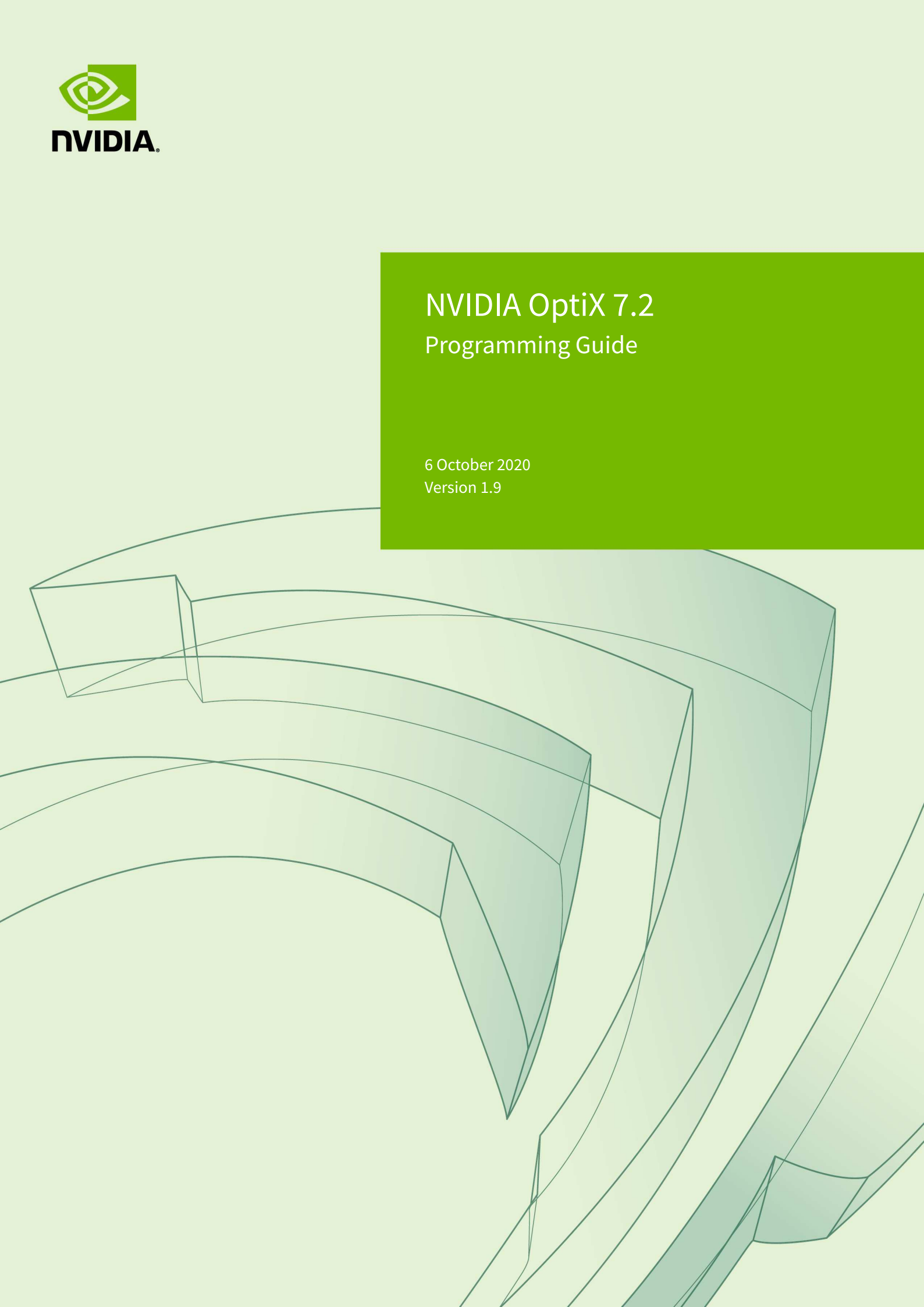

NVIDIA OptiX 7.2

Programming Guide

6 October 2020

Version 1.9

NVIDIA OptiX 7.2 – Programming Guide

Document build number 338715

Contents

Preface . . . 1
Terms used in this document . . . 1
1 Overview . . . 3
2 Basic concepts and definitions . . . 5
2.1 Program . . . 5
2.2 Program and data model . . . 5
2.2.1 Shader binding table . . . 6
2.2.2 Ray payload . . . 6
2.2.3 Primitive attributes . . . 7
2.2.4 Buffer . . . 7
2.3 Acceleration structures . . . 7
2.4 Traversing the scene graph . . . 7
2.5 Ray tracing with NVIDIA OptiX 7 . . . 9
3 Implementation principles . . . 11
3.1 Error handling . . . 11
3.2 Thread safety . . . 11
3.3 Stateless model . . . 11
3.4 Asynchronous execution . . . 11
3.5 Opaque types . . . 11
3.6 Function table and entry function . . . 11
4 Context . . . 15
4.1 Sending messages with a callback function . . . 16
4.2 Compilation caching . . . 16
5 Acceleration structures . . . 19
5.1 Primitive build inputs . . . 21
5.2 Curve build inputs . . . 24
5.3 Instance build inputs . . . 25
5.4 Build flags . . . 26
5.5 Dynamic updates . . . 26
5.6 Relocation . . . 28
5.7 Compacting acceleration structures . . . 29
5.8 Traversable objects . . . 31
5.8.1 Traversal of a single geometry acceleration structure . . . 32
5.9 Motion blur . . . 32
5.9.1 Motion acceleration structure . . . 33
5.9.2 Motion matrix transform . . . 33
5.9.3 Motion scale/rotate/translate transform . . . 34
6 Program pipeline creation . . . 37
6.1 Program input . . . 38
6.2 Module creation . . . 39
6.3 Pipeline launch parameter . . . 40
6.3.1 Parameter specialization . . . 41

6.4 Program group creation . . . 43
6.5 Pipeline linking . . . 46
6.6 Pipeline stack size . . . 47
6.6.1 Constructing a path tracer . . . 49
6.7 Compilation cache . . . 49
7 Shader binding table . . . 51
7.1 Records . . . 51
7.2 Layout . . . 52
7.3 Acceleration structures . . . 53
7.3.1 SBT instance offset . . . 53
7.3.2 SBT geometry-AS index . . . 53
7.3.3 SBT trace offset . . . 55
7.3.4 SBT trace stride . . . 55
7.3.5 Example SBT for a scene . . . 55
7.4 SBT record access on device . . . 57
8 Curves . . . 59
8.1 Differences between curves and triangles . . . 59
8.2 Splitting curve segments . . . 60
8.3 Curves and the hit program . . . 60
8.4 Interpolating curve endpoints . . . 60
8.5 Limitations . . . 61
9 Ray generation launches . . . 63
10 Limits . . . 65
11 Device-side functions . . . 67
11.1 Launch index . . . 69
11.2 Trace . . . 69
11.3 Payload access . . . 71
11.4 Reporting intersections and attribute access . . . 72
11.5 Ray information . . . 73
11.6 Undefined values . . . 73
11.7 Intersection information . . . 74
11.8 SBT record data . . . 75
11.9 Vertex random access . . . 75
11.10 Geometry acceleration structure motion options . . . 76
11.11 Transform list . . . 76
11.12 Terminating or ignoring traversal . . . 78
11.13 Exceptions . . . 78
12 Callables . . . 83
13 NVIDIA AI Denoiser . . . 85
13.1 Functions and data structures for denoising . . . 86
13.1.1 Structure and use of image buffers . . . 86
13.1.2 Selecting the denoiser training model . . . 87
13.1.3 Allocating denoiser memory . . . 88

13.1.4 Using the denoiser . . . 89
13.1.5 Calculating the HDR average color of the AOV model . . . 90
13.1.6 Calculating the HDR intensity parameter . . . 91
13.2 Using image tiles with the denoiser . . . 92
14 Demand-loaded sparse textures . . . 97
14.1 Motivation . . . 97
14.2 Background . . . 98
14.2.1 Sparse textures . . . 98
14.3 Library overview . . . 100
14.4 Texture fetch . . . 100
14.5 Page request processing . . . 102
14.6 Host-side library . . . 102
14.7 Implementation . . . 104

Preface

DirectX Raytracing (DXR),[1] Vulkan[2] (through the `VK_NV_ray_tracing` extension) and the NVIDIA OptiX™ API[3] employ a similar programming model to support ray tracing capabilities. DXR and Vulkan enable ray tracing effects in raster-based gaming and visualization applications. NVIDIA OptiX 7 is intended for ray tracing applications that use NVIDIA® CUDA® technology, such as:

- Film and television visual effects
- Computer-aided design for engineering and manufacturing
- Light maps generated by path tracing
- High-performance computing
- LIDAR simulation

NVIDIA OptiX 7 also includes support for motion blur and multi-level transforms, features required by ray-tracing applications designed for production-quality rendering.

Terms used in this document

This document and the OptiX API use abbreviations for the software components of OptiX.

The nine types of user-defined ray interactions, called *programs*, are abbreviated as follows:

Program type	*Abbreviation*
Ray generation	RG
Intersection	IS
Any-hit	AH
Closest-hit	CH
Miss	MS
Exception	EX
Direct callable	DC
Continuation callable	CC

The NVIDIA OptiX 7 program types resemble shaders in traditional rendering systems; the term “shader” is sometimes used in the names of API elements.

1. `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html`
2. `https://www.khronos.org/vulkan/`
3. `https://developer.nvidia.com/optix/`

The geometry of the scene to be rendered is optimized for ray tracing through *acceleration structures*:

Acceleration structure type	*Abbreviation in this document*	*Abbreviation in the API*
Geometry acceleration structure	geometry-AS	GAS
Instance acceleration structure	instance-AS	IAS
Acceleration structures in general		AS
Bottom-level acceleration structure (DXR and Vulkan)	BLAS	
Top-level acceleration structure (DXR and Vulkan)	TLAS	

The relationship of NVIDIA OptiX 7 programs and the elements of the acceleration structures with which they interact are defined in the *shader binding table*, abbreviated as "SBT". (Note that "shader" in this context refers to an NVIDIA OptiX 7 "program.") No other terms associated with the shader binding table are abbreviated.

Other abbreviations in the document include:

Term	*Abbreviation*
application programming interface	API
axis-aligned bounding box	AABB
graphics processing unit	GPU
high dynamic range	HDR
just-in-time	JIT
low dynamic range	LDR
multiple instruction, multiple data	MIMD
parallel thread execution	PTX
scaling, rotation, translation	SRT
streaming assembly [language]	SASS
streaming multiprocessor	SM

In this document and in the names of API elements, the "host" is the processor that begins execution of an application. The "device" is the GPU with which the host interacts. A "build" is the creation of an acceleration structure on the device as initiated by the host.

1 Overview

The NVIDIA OptiX 7 API is a CUDA-centric API that is invoked by a CUDA-based application. The API is designed to be stateless, multi-threaded and asynchronous, providing explicit control over performance-sensitive operations like memory management and shader compilation.

It supports a lightweight representation for scenes that can represent instancing, vertex- and transform-based motion blur, with built-in triangles, built-in swept curves, and user-defined primitives. The API also includes highly-tuned kernels and neural networks for machine-learning-based denoising.

An NVIDIA OptiX 7 context controls a single GPU. The context does not hold bulk CPU allocations, but like CUDA, may allocate resources on the device necessary to invoke the launch. It can hold a small number of handle objects that are used to manage expensive host-based state. These handle objects are automatically released when the context is destroyed. Handle objects, where they do exist, consume a small amount of host memory (typically less than 100 kilobytes) and are independent of the size of the GPU resources being used. For exceptions to this rule, see "Program pipeline creation" (page 37).

The application invokes the creation of acceleration structures (called *builds*), compilation, and host-device memory transfers. All API functions employ CUDA streams and invoke GPU functions asynchronously, where applicable. If more than one stream is used, the application must ensure that required dependencies are satisfied by using CUDA events to avoid race conditions on the GPU.

Applications can specify multi-GPU capabilities with a few different recipes. Multi-GPU features such as efficient load balancing or the sharing of GPU memory via NVLINK must be handled by the application developer.

For efficiency and coherence, the NVIDIA OptiX 7 runtime—unlike CUDA kernels—allows the execution of one task, such as a single ray, to be moved at any point in time to a different lane, warp or streaming multiprocessor (SM). (See section "Kernel Focus"[4] in the CUDA Toolkit Documentation.[5]) Consequently, applications cannot use shared memory, synchronization, barriers, or other SM-thread-specific programming constructs in their programs supplied to OptiX.

The NVIDIA OptiX 7 programming model provides an API that future-proofs applications: as new NVIDIA hardware features are released, existing programs can use them. For example, software-based ray tracing algorithms can be mapped to hardware when support is added or mapped to software when the underlying algorithms or hardware support such changes.

4. `https://docs.nvidia.com/cuda/cuda-gdb/index.html#kernel-focus`

5. `https://docs.nvidia.com/cuda/`

2 Basic concepts and definitions

2.1 Program

In NVIDIA OptiX 7, a *program* is a block of executable code on the GPU that represents a particular shading operation. This is called a *shader* in DXR and Vulkan. For consistency with prior versions of NVIDIA OptiX 7, the term *program* is used in the current documentation. This term also serves as a reminder that these blocks of executable code are programmable components in the system that can do more than shading. See "Program input" (page 38).

2.2 Program and data model

NVIDIA OptiX 7 implements a single-ray programming model with ray generation, any-hit, closest-hit, miss and intersection programs.

The ray tracing pipeline provided by NVIDIA OptiX 7 is implemented by 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. See "Ray generation launches" (page 63).

Intersection
: Implements a ray-primitive intersection test, invoked during traversal. See "Traversing the scene graph" (page 7) and "Ray information" (page 73).

Any-hit
: Called when a traced ray finds a new, potentially closest, intersection point, such as for shadow computation. See "Ray information" (page 73).

Closest-hit
: Called when a traced ray finds the closest intersection point, such as for material shading. See "Constructing a path tracer" (page 48).

Miss
: Called when a traced ray misses all scene geometry. See "Constructing a path tracer" (page 48).

Exception
: Exception handler, invoked for conditions such as stack overflow and other errors. See "Exceptions" (page 78).

Direct callables
: Similar to a regular CUDA function call, direct callables are called immediately. See "Callables" (page 83).

Continuation callables
: Unlike direct callables, continuation callables are executed by the scheduler. See "Callables" (page 83).

The ray-tracing "pipeline" is based on the interconnected calling structure of the eight programs and their relationship to the search through the geometric data in the scene, called a *traversal*. Figure 2.1 is a diagram of these relationships:

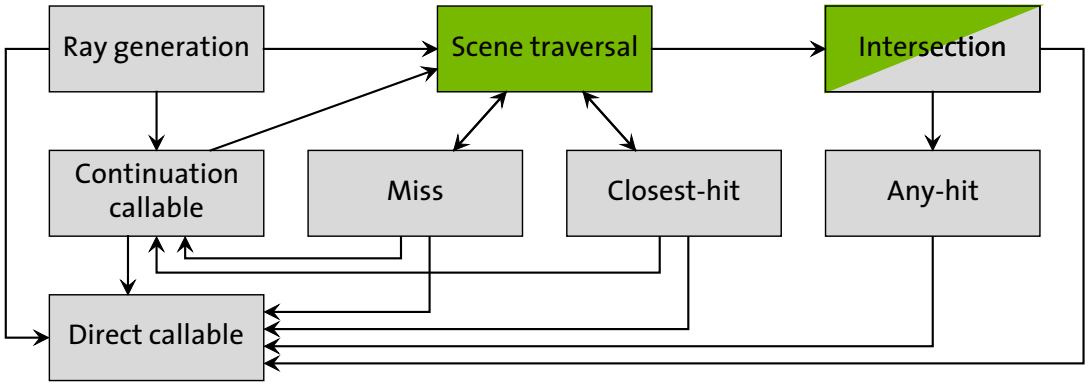

Fig. 2.1 – Relationship of NVIDIA OptiX 7 programs. Green represents fixed functions; gray represents user programs.

In Figure 2.1, green represents fixed-function, hardware-accelerated operations, while gray represents user programs. The built-in or user-provided exception program may be called from any program or scene traversal in case of an exception if exceptions are enabled.

2.2.1 Shader binding table

The *shader binding table* connects geometric data to programs and their parameters. A *record* is a component of the shader binding table that is selected during execution by using offsets specified when acceleration structures are created and at runtime. A record contains two data regions, *header* and *data*.

Record header

- Opaque to the application, filled in by `optixSbtRecordPackHeader`
- Used by NVIDIA OptiX 7 to identify programmatic behavior. For example, a primitive would identify the intersection, any-hit, and closest-hit behavior for that primitive in the header.

Record data

- Opaque to NVIDIA OptiX 7
- User data associated with the primitive or programs referenced in the headers can be stored here, for example, program parameter values.

2.2.2 Ray payload

The *ray payload* is used to pass data between `optixTrace` and the programs invoked during ray traversal. Payload values are passed to and returned from `optixTrace`, and follow a copy-in/copy-out semantic. There is a limited number of payload values, but one or more of these values can also be a pointer to stack-based local memory, or application-managed global memory.

2.2.3 Primitive attributes

Attributes are used to pass data from intersection programs to the any-hit and closest-hit programs. Triangle intersection provides two predefined attributes for the barycentric coordinates (U,V). User-defined intersections can define a limited number of other attributes that are specific to those primitives.

2.2.4 Buffer

NVIDIA OptiX 7 represents GPU information with a pointer to GPU memory. References to the term "buffer" in this document refer to this GPU memory pointer and the associated memory contents. Unlike NVIDIA OptiX 6, the allocation and transfer of buffers is explicitly controlled by user code.

2.3 Acceleration structures

NVIDIA OptiX 7 acceleration structures are opaque data structures built on the device. Typically, they are based on the *bounding volume hierarchy*[6] model, but implementations and the data layout of these structures may vary from one GPU architecture to another.

NVIDIA OptiX 7 provides two basic types of acceleration structures:

Geometry acceleration structures

- Built over primitives (triangles, curves, or user-defined primitives)

Instance acceleration structures

- Built over other objects such as acceleration structures (either type) or motion transform nodes
- Allow for instancing with a per-instance static transform

2.4 Traversing the scene graph

To determine the intersection of geometric data by a ray, NVIDIA OptiX 7 searches a graph of nodes composed of acceleration structures and transformations. This search is called a *traversal*; the nodes in the graph are called *traversable objects* or *traversables*.

Traversable objects consist of the two acceleration structure types and three types of transformations:

- An instance acceleration structure
- A geometry acceleration structure (as a root for graph with a single geometry acceleration structure (see "Traversal of a single geometry acceleration structure" (page 31))
- Static transform
- Matrix motion transform
- Scaling, rotation, translation (SRT) motion transform

6. `https://en.wikipedia.org/wiki/Bounding_volume_hierarchy`

The two motion transformations provide the required positional information for dynamic transformations used in motion blur calculation. Dynamic and static transformations can be combined in a graph. For example, Figure 2.2 combines both types:

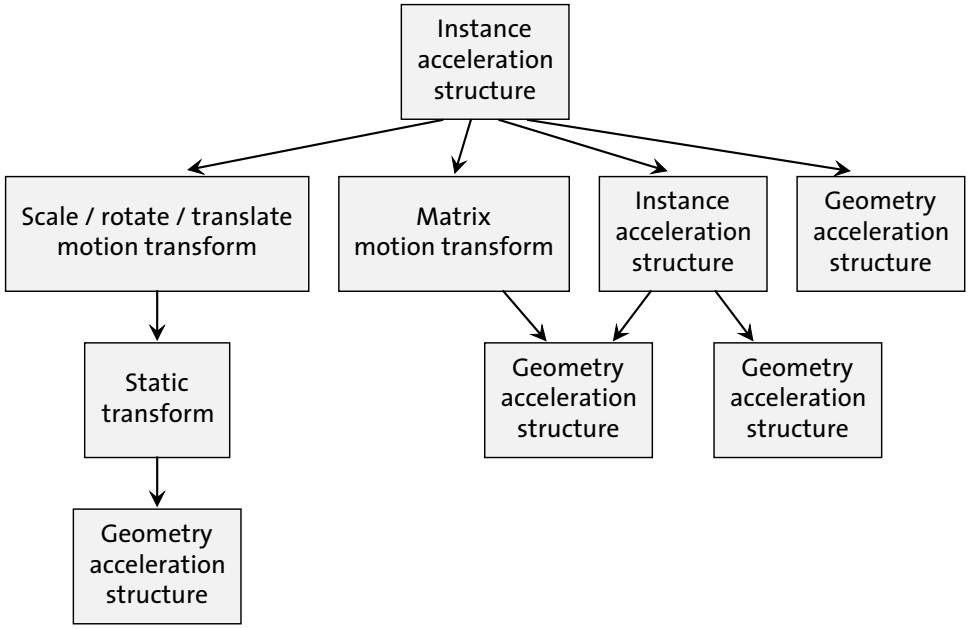

Fig. 2.2 – Example graph of traversables for a scene containing static as well as dynamic motion-transform driven objects

Traversable handles are 64-bit opaque values that are generated from device memory pointers for the graph objects. The handles identify the connectivity of these objects. All calls to `optixTrace` begin at a traversable handle.

Note: DXR and VulkanRT use the terms *top-level acceleration structure* and *bottom-level acceleration structure*. A bottom-level acceleration structure is the same as a geometry acceleration structure; a top-level acceleration structure is similar to an instance acceleration structure. A traversable consisting of a single geometry acceleration structure, motion transform nodes and nested instance acceleration structures are not supported in DXR or VulkanRT. In NVIDIA OptiX 7, the terms were changed due to the additional possible configurations of scene graphs beyond the strict two-level, top-bottom configurations supported by DXR and VulkanRT. (See DirectX Raytracing (DXR) Functional Spec.[7])

7. `https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#geometry-and-acceleration-structures`

2.5 Ray tracing with NVIDIA OptiX 7

A functional ray tracing system is implemented by combining four components as described in the following steps:

1. Create one or more acceleration structures that represent a geometry mesh in the scene and select one or more records in the shader binding table for each mesh. See "Acceleration structures" (page 19).
2. Create a pipeline of programs that contains all programs that will be invoked during a ray tracing launch. See "Program pipeline creation" (page 37).
3. Create a shader binding table that includes references to these programs and their parameters. See "Shader binding table" (page 51).
4. Launch a device-side kernel that will invoke a ray generation program with a multitude of threads calling `optixTrace` to begin traversal and the execution of the other programs. See "Ray generation launches" (page 63). Device-side functionality is described in "Device-side functions" (page 67).

Ray tracing work can be interleaved with other CUDA work to generate data, move data to and from the device, and move data to other graphics APIs. It is the application's responsibility to coordinate all work on the GPU. NVIDIA OptiX 7 does not synchronize with any other work.

3 Implementation principles

3.1 Error handling

Errors are reported using enumerated return codes. An optional log callback can be registered with the device context to receive any additional logging information.

Functions that compile can optionally take a string character buffer to report additional messaging for errors, warnings and resource use.

3.2 Thread safety

Almost all host functions are thread-safe. Exceptions to this rule are identified in the API documentation. A general requirement for thread-safety is that output buffers and any temporary or state buffers are unique. For example, you can create more than one acceleration structure concurrently from the same input geometry, as long as the temporary and output device memory are disjoint. Temporary and state buffers are always part of the parameter list if they are needed to execute the method.

3.3 Stateless model

Given the same input, the same output should be generated. GPU state is not held by NVIDIA OptiX 7 internally.

In NVIDIA OptiX 7 functions, a `CUstream` is associated with the `CUcontext` used to create the `OptixDeviceContext`. Applications can expect the `CUcontext` to remain the same after invoking NVIDIA OptiX 7 functions.

3.4 Asynchronous execution

Work performed on the device is issued on an application-supplied `CUstream` using asynchronous CUDA methods. The host function blocks execution until all work has been issued on the stream, but does not do any synchronization or blocking on the stream itself.

3.5 Opaque types

The API employs several opaque types, such as `OptixModule` and `OptixPipeline`. Such values should be treated like pointers, insofar as copying these does not create new objects.

3.6 Function table and entry function

The NVIDIA OptiX 7 library uses a function table approach to assist in the introduction of new features in future releases while maintaining backward compatibility. To that end, it defines a struct `OptixFunctionTable` that holds pointers to all functions of the host API for a particular version. The current version is specified in the `OPTIX_ABI_VERSION` macro definition.

Listing 3.1

```
struct OptixFunctionTable
{
    OptixResult (*optixDeviceContextCreate)(
        ...                 Parameter list details omitted
        );
    ...                     Struct members for other host API functions omitted
};
```

The NVIDIA OptiX 7 driver library exports the symbol `OptixQueryFunctionTable`. This function is used to obtain pointers to the actual API functions:

Listing 3.2

```
OptixQueryFunctionTable_t* optixQueryFunctionTable;

...                     OS-specific code to load the library and to assign the address of
                        OptixQueryFunctionTable to optixQueryFunctionTable omitted

OptixFunctionTable optixFunctionTable = {};
OptixResult result = optixQueryFunctionTable(
    OPTIX_ABI_VERSION, 0, 0, 0, &optixFunctionTable,
    sizeof(OptixFunctionTable));

...                     Error check omitted
```

Note that the three "0 "arguments in the example above allow for future extensions of the entry function without changing its signature and are currently unused. A complete example implementation of this functionality including code specific to the operating system is provided as source code in `optixInit()` found in the header file `optix_stubs.h`.

After a successful call to `optixQueryFunctionTable`, the function table can be used as follows, for example, for context creation:

Listing 3.3

```
CUcontext fromContext = nullptr;
...                     fromContext initialization omitted

OptixDeviceContextOptions options = {};
...                     options initialization omitted

OptixResult result = optixFunctionTable.optixDeviceContextCreate(
  fromContext, &options, &context);

...                     Error check omitted
```

Since the explicit call qualifications with the function table instance can be inconvenient, optional stubs that wrap the addresses in the function table into C functions are provided. These stubs are made available by including the header file `optix_stubs.h`. With these stubs the previous example can be simplified as follows:

Listing 3.4

```
CUcontext fromContext = nullptr;
...          fromContext initialization omitted

OptixDeviceContextOptions options = {};
...          options initialization omitted

OptixResult result = optixDeviceContextCreate(
   fromContext, &options, &context);

...          Error check omitted
```

Using these stubs is purely optional and applications are free to implement their own solution to make the addresses in the function table more easily available.

4 Context

The API functions described in this section are:

```
optixDeviceContextCreate
optixDeviceContextDestroy
optixDeviceContextGetProperty
optixDeviceContextSetLogCallback
optixDeviceContextSetCacheEnabled
optixDeviceContextSetCacheLocation
optixDeviceContextSetCacheDatabaseSizes
optixDeviceContextGetCacheEnabled
optixDeviceContextGetCacheLocation
optixDeviceContextGetCacheDatabaseSizes
```

A *context* is created by `optixDeviceContextCreate` and is used to manage a single GPU. The NVIDIA OptiX 7 device context is created by specifying the CUDA context associated with the device. For convenience, zero can be passed and NVIDIA OptiX 7 will use the current CUDA context.

Listing 4.1

```
OptixDeviceContext context = nullptr;
cudaFree(0);    Initialize CUDA for this device on this thread

CUcontext cuCtx = 0;    Zero means take the current context
optixDeviceContextCreate(cuCtx, 0, &context);
```

Additional creation time options can also be specified with `OptixDeviceContextOptions`, including parameters for specifying a callback function, log and data. (See "Sending messages with a callback function" (page 16).)

A small set of context properties exist for determining sizes and limits. These are queried using `optixDeviceContextGetProperty`. Such properties include maximum trace depth, maximum traversable graph depth, maximum primitives per build input, and maximum number of instances per acceleration structure.

The context may retain ownership of any GPU resources necessary to launch the ray tracing kernels. Some API objects will retain host memory. These are defined with create/destroy patterns in the API. The application must invoke `optixDeviceContextDestroy` to clean up any host or device resources associated with the context. If any other API objects associated with this context still exist when the context is destroyed, they are also destroyed.

A context can hold a decryption key. When specified, the context requires user code passed into the API to be encrypted using the appropriate session key. This minimizes exposure of the input code to security attacks.

Note: The context decryption feature is available upon request from NVIDIA.

An application may combine any mixture of supported GPUs as long as the data transfer and synchronization is handled appropriately. Some applications may choose to simplify multi-GPU handling by restricting the variety of these blends, for example, by mixing only GPUs of the same streaming multiprocessor version to simplify data sharing.

4.1 Sending messages with a callback function

A log callback and pointer to host memory can also be specified during context creation or later by using `optixDeviceContextSetLogCallback`. This callback will be used to communicate various messages. It must be thread-safe if multiple NVIDIA OptiX 7 functions are called concurrently.

This callback must be a pointer to a function of the following type:

Listing 4.2

```
typedef void (*OptixLogCallback)(
   unsigned int level,
   const char* tag,
   const char* message,
   void* cbdata);
```

The log level indicates the severity of the message. The tag is a terse message category description (for example, `SCENE STAT`). The message is a null-terminated log message (without a newline character at the end) and the value of `cbdata`, the pointer provided when setting the callback function.

The following log levels are supported:

`disable`
: Disables all messages. The callback function is not called in this case.

`fatal`
: A non-recoverable error. The context, as well as NVIDIA OptiX 7 itself, may no longer be in a usable state.

`error`
: A recoverable error, for example, when passing invalid call parameters.

`warning`
: Hints that the API might not behave exactly as expected by the application or that it may perform slower than expected.

`print`
: Status and progress messages.

4.2 Compilation caching

Compilation of input programs will be cached to disk when creating `OptixModule`, `OptixProgramGroup`, and `OptixPipeline` objects if caching has been enabled. Subsequent compilation can reuse the cached data to improve the time to create these objects. The cache can be shared between multiple `OptixDeviceContext` objects, and NVIDIA OptiX 7 will take

care of ensuring correct multi-threaded access to the cache. If no sharing between `OptixDeviceContext` objects is desired, the path to the cache can be set differently for each `OptixDeviceContext`.

The disk cache can be controlled with:

`optixDeviceContextSetCacheEnabled(..., int enabled)`

When `enabled` has a value of 1, the disk cache is enabled; a value of 0 disables it. Note that no in-memory cache is used when caching is disabled.

The cache database is initialized when the device context is created and when enabled through this function call. If the database cannot be initialized when the device context is created, caching will be disabled; a message is reported to the log callback if caching is enabled. In this case, the call to `optixDeviceContextCreate` does not return an error. To ensure that cache initialization succeeded on context creation, the status can be queried using `optixDeviceContextGetCacheEnabled`. If caching is disabled, the cache can be reconfigured and then enabled using `optixDeviceContextSetCacheEnabled`. If the cache database cannot be initialized with `optixDeviceContextSetCacheEnabled`, an error is returned. Garbage collection is performed on the next write to the cache database, not when the cache is enabled.

`optixDeviceContextSetCacheLocation(..., const char* location)`

The disk cache is created in the directory specified by `location`. The value of `location` must be a NULL-terminated string. The directory is created if it does not exist.

The cache database is created immediately if the cache is currently enabled. Otherwise the cache database is created later when the cache is enabled. An error is returned if it is not possible to create the cache database file at the specified location for any reason (for example, if the path is invalid or if the directory is not writable) and caching will be disabled. If the disk cache is located on a network file share, behavior is undefined.

The location of the disk cache can be overridden with the environment variable `OPTIX_CACHE_PATH`. This environment variable takes precedence over the value passed to this function when the disk cache is enabled.

The default location of the cache depends on the operating system:

Operating system	*Pathname*
Windows	`%LOCALAPPDATA%\NVIDIA\OptixCache`
Linux	`/var/tmp/OptixCache_`*username*, or `/tmp/OptixCache_`*username* if the first choice is not usable. The underscore and username suffix are omitted if the username cannot be obtained.

`optixDeviceContextSetCacheDatabaseSizes(`
`..., size_t lowWaterMark, size_t highWaterMark)`

Parameters `lowWaterMark` and `highWaterMark` set the low and high water marks for disk cache garbage collection. Setting either limit to zero disables garbage collection. Garbage collection only happens when the cache database is written. It is triggered whenever the cache data size exceeds the high water mark and proceeding until the size reaches the low water mark. Garbage collection always frees enough space to allow the insertion of the new entry within the boundary of the low water mark. An error is returned if either limit is nonzero and the high water mark is lower than the

low water mark. If more than one device context accesses the same cache database with different high and low water mark values, the device context uses its values when writing to the cache database.

Corresponding `get*` functions are supplied to retrieve the current value of these cache properties.

5 Acceleration structures

The API functions described in this section are:

```
optixAccelComputeMemoryUsage
optixAccelBuild
optixAccelRelocate
optixConvertPointerToTraversableHandle
```

NVIDIA OptiX 7 provides *acceleration structures* to optimize the search for the intersection of rays with the geometric data in the scene. Acceleration structures can contain two types of data: geometric primitives (a *geometry-AS*) or instances (an *instance-AS*). Acceleration structures are created on the device using a set of functions. These functions enable overlapping and pipelining of acceleration structure creation, called a *build*. The functions use one or more `OptixBuildInput` structs to specify the geometry plus a set of parameters to control the build.

Acceleration structures have size limits, listed in "Limits" (page 65). For an instance acceleration structure, the number of instances has an upper limit. For a geometry acceleration structure, the number of geometric primitives is limited, specifically the total number of primitives in its build inputs, multiplied by the number of motion keys.

The following build input types are supported:

Instance acceleration structures

`OPTIX_BUILD_INPUT_TYPE_INSTANCES`
`OPTIX_BUILD_INPUT_TYPE_INSTANCE_POINTERS`

A geometry acceleration structure containing built-in triangles

`OPTIX_BUILD_INPUT_TYPE_TRIANGLES`

A geometry acceleration structure containing built-in curve primitives

`OPTIX_BUILD_INPUT_TYPE_CURVES`

A geometry acceleration structure containing custom primitives

`OPTIX_BUILD_INPUT_TYPE_CUSTOM_PRIMITIVES`

For geometry-AS builds, each build input can specify a set of triangles, a set of curves, or a set of user-defined primitives bounded by specified axis-aligned bounding boxes. Multiple build inputs can be passed as an array to `optixAccelBuild` to combine different meshes into a single acceleration structure. All build inputs for a single build must agree on the build input type.

Instance acceleration structures have a single build input and specify an array of instances. Each instance includes a ray transformation and an `OptixTraversableHandle` that refers to a geometry-AS, a transform node, or another instance acceleration structure.

To prepare for a build, the required memory sizes are queried by passing an initial set of build inputs and parameters to `optixAccelComputeMemoryUsage`. It returns three different sizes:

`outputSizeInBytes`
: Size of the memory region where the resulting acceleration structure is placed. This size is an upper bound and may be substantially larger than the final acceleration structure. (See "Compacting acceleration structures" (page 29).)

`tempSizeInBytes`
: Size of the memory region that is temporarily used during the build.

`tempUpdateSizeInBytes`
: Size of the memory region that is temporarily required to update the acceleration structure.

Using these sizes, the application allocates memory for the output and temporary memory buffers on the device. The pointers to these buffers must be aligned to a 128-byte boundary. These buffers are actively used for the duration of the build. For this reason, they cannot be shared with other currently active build requests.

Note that `optixAccelComputeMemoryUsage` does not initiate any activity on the device; pointers to device memory or contents of input buffers are not required to point to allocated memory.

The function `optixAccelBuild` takes the same array of `OptixBuildInput` structs as `optixAccelComputeMemoryUsage` and builds a single acceleration structure from these inputs. This acceleration structure can contain either geometry or instances, depending on the inputs to the build.

The build operation is executed on the device in the specified CUDA stream and runs asynchronously on the device, similar to CUDA kernel launches. The application may choose to block the host-side thread or synchronize with other CUDA streams by using available CUDA synchronization functionality such as `cudaStreamSynchronize` or CUDA events. The traversable handle returned is computed on the host and is returned from the function immediately, without waiting for the build to finish. By producing handles at acceleration time, custom handles can also be generated based on input to the builder.

The acceleration structure constructed by `optixAccelBuild` does not reference any of the device buffers referenced in the build inputs. All relevant data is copied from these buffers into the acceleration output buffer, possibly in a different format.

The application is free to release this memory after the build without invalidating the acceleration structure. However, instance-AS builds will continue to refer to other instance-AS and geometry-AS instances and transform nodes.

The following example uses this sequence to build a single acceleration structure:

Listing 5.1

```
OptixAccelBuildOptions accelOptions = {};
OptixBuildInput buildInputs[2];

CUdeviceptr tempBuffer, outputBuffer;
size_t tempBufferSizeInBytes, outputBufferSizeInBytes;

memset(accelOptions, 0, sizeof(OptixAccelBuildOptions));
accelOptions.buildFlags = OPTIX_BUILD_FLAG_NONE;
```



```
accelOptions.operation = OPTIX_BUILD_OPERATION_BUILD;
accelOptions.motionOptions.numKeys = 0;
```
A numKeys value of zero specifies no motion blur

```
memset(buildInputs, 0,  sizeof(OptixBuildInput) * 2);
```
Initialize buildInputs memory to 0

```
...
```
Setup build inputs; see below.

```
OptixAccelBufferSizes bufferSizes = {};
optixAccelComputeMemoryUsage(optixContext, &accelOptions,
   buildInputs, 2, &bufferSizes);

void* d_output;
void* d_temp;

cudaMalloc(&d_output, bufferSizes.outputSizeInBytes);
cudaMalloc(&d_temp, bufferSizes.tempSizeInBytes);

OptixTraversableHandle outputHandle = 0;
OptixResult results = optixAccelBuild(optixContext, cuStream,
    &accelOptions, buildInputs, 2, d_temp,
    bufferSizes.tempSizeInBytes, d_output,
    bufferSizes.outputSizeInBytes, &outputHandle, nullptr, 0);
```

To ensure compatibility with future versions, the `OptixBuildInput` structure should be initialized with zeros before populating it with specific build inputs.

5.1 Primitive build inputs

A triangle build input references an array of triangle vertex buffers in device memory, one buffer per motion key (a single triangle vertex buffer if there is no motion). (See "Motion blur" (page 32).) Optionally, triangles can be indexed using an index buffer in device memory. Various vertex and index formats are supported as input, but may be transformed to an internal format (potentially of a different size than the input) that is more efficient.

For example:

Listing 5.2

```
OptixBuildInputTriangleArray& buildInput =
   buildInputs[0].triangleArray;
buildInput.type = OPTIX_BUILD_INPUT_TYPE_TRIANGLES;
buildInput.vertexBuffers = &d_vertexBuffer;
buildInput.numVertices = numVertices;
buildInput.vertexFormat = OPTIX_VERTEX_FORMAT_FLOAT3;
buildInput.vertexStrideInBytes = sizeof(float3);
buildInput.indexBuffer = d_indexBuffer;
buildInput.numIndexTriplets = numTriangles;
buildInput.indexFormat = OPTIX_INDICES_FORMAT_UNSIGNED_INT3;
buildInput.indexStrideInBytes = sizeof(int3);
buildInput.preTransform = 0;
```

The `preTransform` is an optional pointer to a 3x4 row-major transform matrix in device memory. The pointer needs to be aligned to 16 bytes; the matrix contains 12 floats. If specified, the transformation is applied to all vertices at build time with no runtime traversal overhead.

A curves build input is similar to a triangle build input; see "Curve build inputs" (page 24).

The acceleration structure build input for custom primitives uses the type `OptixBuildInputCustomPrimitiveArry`. Each custom primitive is represented by an *axis-aligned bounding box* (AABB), which is a rectangular solid defined by ranges of x, y, and z values, and which must completely enclose the primitive. The memory layout of an AABB is defined in the struct `OptixAabb`. The AABBs are organized in an array of buffers in device memory, with one buffer per motion key. The precise shape of each custom primitive will be depend on the intersection program in its SBT record.

Listing 5.3

```
OptixBuildInputCustomPrimitiveArray& buildInput =
    buildInputs[0].customPrimitiveArray;
buildInput.type = OPTIX_BUILD_INPUT_TYPE_CUSTOM_PRIMITIVES;
buildInput.aabbBuffers = d_aabbBuffer;
buildInput.numPrimitives = numPrimitives;
```

The `optixAccelBuild` function accepts multiple build inputs per call, but they must be all triangle inputs, all curve inputs, or all AABB inputs. Mixing build input types in a single geometry-AS is not allowed.

Each build input maps to one or more consecutive records in the shader binding table (SBT), which controls program dispatch. (See "Shader binding table" (page 51).) If multiple records in the SBT are required, the application needs to provide a device buffer with per-primitive SBT record indices for that build input. If only a single SBT record is requested, all primitives reference this same unique SBT record. Note that there is a limit to the number of referenced SBT records per geometry-AS. (Limits are discussed in "Limits" (page 65).)

For example:

Listing 5.4

```
buildInput.numSbtRecords = 2;
buildInput.sbtIndexOffsetBuffer =
    d_sbtIndexOffsetBuffer;
buildInput.sbtIndexOffsetSizeInBytes =
    sizeof(int);
buildInput.sbtIndexOffsetStrideInBytes =
    sizeof(int);
```

Values must be in range [0,1] for two SBT records

1-4 byte unsigned integer offsets allowed

Each build input also specifies an array of `OptixGeometryFlags`, one for each SBT record. The flags for one record apply to all primitives mapped to this SBT record.

For example:

Listing 5.5

```
unsigned int flagsPerSBTRecord[2];
flagsPerSBTRecord[0] = OPTIX_GEOMETRY_FLAG_NONE;
flagsPerSBTRecord[1] = OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT;
...
buildInput.flags = flagsPerSBTRecord;
```

The following flags are supported:

`OPTIX_GEOMETRY_FLAG_NONE`
: Applies the default behavior when calling the any-hit program, possibly multiple times, allowing the acceleration-structure builder to apply all optimizations.

`OPTIX_GEOMETRY_FLAG_REQUIRE_SINGLE_ANYHIT_CALL`
: Disables some optimizations specific to acceleration-structure builders. By default, traversal may call the any-hit program more than once for each intersected primitive. Setting the flag ensures that the any-hit program is called only once for a hit with a primitive. However, setting this flag may change traversal performance. The usage of this flag may be required for correctness of some rendering algorithms; for example, in cases where opacity or transparency information is accumulated in an any-hit program.

`OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT`
: Indicates that traversal should not call the any-hit program for this primitive even if the corresponding SBT record contains an any-hit program. Setting this flag usually improves performance even if no any-hit program is present in the SBT.

Primitives inside a build input are indexed starting from zero. This primitive index is accessible inside the intersection, any-hit, and closest-hit programs. If the application chooses to offset this index for all primitives in a build input, there is no overhead at runtime. This can be particularly useful when data for consecutive build inputs is stored consecutively in device memory. The `primitiveIndexOffset` value is only used when reporting the intersection primitive.

For example:

Listing 5.6

```
buildInput[0].aabbBuffers = d_aabbBuffer;
buildInput[0].numPrimitives = ...;
buildInput[0].primitiveIndexOffset = 0;

buildInput[1].aabbBuffers = d_aabbBuffer +
   buildInput[0].numPrimitives * sizeof(float) * 6;
buildInput[1].numPrimitives = ...;
buildInput[1].primitiveIndexOffset = buildInput[0].numPrimitives;
```

5.2 Curve build inputs

In addition to triangles and custom primitives, NVIDIA OptiX 7 supports curves as geometric primitives. Curves are used to represent long thin strands, such as for hair, fur, and carpet fibers. Scenes may contain thousands or millions of curves, and they will often be no wider than a couple of pixels in the final image.

Each curve is a swept surface defined by a three-dimensional series of vertices, called *control points*, and a possibly varying radius. The NVIDIA OptiX 7 API provides curves with these characteristics:

- The cross-section of the curve primitive is a circle.
- Curve geometry is defined by a cubic uniform B-spline curve, a quadratic uniform B-spline curve, or a series of linear segments.
- A radius is specified at each control point. The radius is interpolated along the curve using the same spline basis as position.
- Cubic and quadratic splines have flat end-caps, at each end of the curve strand, like the caps of a cylinder. Linear curves have spherical end-caps, with spherical "elbows" for smooth joints between segments.

Spline curves are composed of a series of polynomial segments. Each segment is defined by two, three, or four control points, depending on the curve type:

Curve type	*Control points per segment*
Piecewise linear	2
Quadratic	3
Cubic	4

NVIDIA OptiX 7 considers each polynomial segment to be a primitive, with its own primitive ID.

A curve build input (`OptixBuildInputCurveArray`) references an array of vertex buffers in device memory, one buffer per motion key (a single vertex buffer if there is no motion). (See "Motion blur" (page 32).) Parallel to this, there is an array of radius buffers in device memory, one buffer per motion key, providing a radius value at each control vertex at each motion key. There is also a (required) index buffer in device memory.

The B-spline control points of each curve strand will appear sequentially in the vertex buffer. The index array contains one index per segment, namely, the index of the segment's first control point. For example, a cubic curve with three segments will have six vertices. The index array might contain `{10, 11, 12}`, in which case the 3 segments will have control points: `{v[10], v[11], v[12], v[13]}`, `{v[11], v[12], v[13], v[14]}` and `{v[12], v[13], v[14], v[15]}`.

End caps appear at the ends of strands. NVIDIA OptiX 7 detects the strands by checking the overlap of segment control points. Within a B-spline strand, adjacent segments overlap all but one of their control points. In other words, unless `indexArray[N+1]` is equal to `indexArray[N]+1`, segment N is the end of one strand and segment N+1 is the beginning of another.

See also "Differences between curves and triangles" (page 59) .

5.3 Instance build inputs

An instance build input specifies a buffer of `OptixInstance` structs in device memory. These structs can be specified as an array of consecutive structs or an array of pointers to those structs. Each instance description references:

- A child traversable handle
- A static 3x4 row-major object-to-world matrix transform
- A user ID
- An SBT offset
- A visibility mask
- Instance flags

Unlike the triangle and AABB inputs, `optixAccelBuild` only accepts a single instance build input per build call. There are upper limits to the possible number of instances (the size of the buffer of the `OptixInstance` structs), the SBT offset, the visibility mask, as well as the user ID. (These limits are discussed in "Limits" (page 65).)

An example of this sequence:

Listing 5.7

```
OptixInstance instance = {};
float transform[12] = {1,0,0,3,0,1,0,0,0,0,1,0};
memcpy(instance.transform, transform, sizeof(float)*12);
instance.instanceId = 0;
instance.visibilityMask = 255;
instance.sbtOffset = 0;
instance.flags = OPTIX_INSTANCE_FLAG_NONE;
instance.traversableHandle = gasTraversable;

void* d_instance;

cudaMalloc(&d_instance, sizeof(OptixInstance));

cudaMemcpy(d_instance, &instance,
   sizeof(OptixInstance),
   cudaMemcpyHostToDevice);

OptixBuildInputInstanceArray* buildInput =
   &buildInputs[0].instanceArray;
buildInput->type = OPTIX_BUILD_INPUT_TYPE_INSTANCES;
buildInput->instances = d_instance;
buildInput->numInstances = 1;
```

The `OPTIX_BUILD_INPUT_TYPE_INSTANCE_POINTERS` build input is a variation on the `OPTIX_BUILD_INPUT_TYPE_INSTANCES` build input where `instanceDescs` references a device memory array of pointers to `OptixInstance` data structures in device memory.

Instance flags are applied to primitives encountered while traversing the geometry-AS connected to an instance. The flags override any instance flags set during the traversal of parent instance-ASs.

`OPTIX_INSTANCE_FLAG_DISABLE_TRIANGLE_FACE_CULLING`
: Disables face culling for triangles. Overrides any culling ray flag passed to `optixTrace`.

`OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING`
: Flips the triangle orientation during intersection. Also affects any front/backface culling.

`OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT`
: Disables any-hit calls for primitive intersections. Can be overridden by ray flags.

`OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT`
: Forces any-hit calls for primitive intersections. Can be overridden by ray flags.

`OPTIX_INSTANCE_FLAG_DISABLE_TRANSFORM`
: Disables the static matrix transformation.

The visibility mask is combined with the ray mask to determine visibility for this instance. If the condition `rayMask & instance.mask == 0` is true, the instance is culled. The visibility flags may be interpreted as assigning rays and instances to one of eight groups. Instances are traversed only when the instance and ray have at least one group in common. (See "Trace" (page 69).)

The `sbtOffset` describes one of two offset values. Typically, the `sbtOffset` is the instance's offset into the SBT array specified with the `hitgroupRecordBase` parameter of `OptixShaderBindingTable`. However, if the child is an `OptixStaticTransform`, `OptixMatrixMotionTransform` or `OptixSRTMotionTransform` traversable object instead of a geometry-AS, then the `sbtOffset` value corresponds to the geometry or instance acceleration structure traversable at the end of the chain of traversable objects. For example, a possible chain of traversable objects would be an instance-AS, an `OptixMatrixMotionTransform`, an `OptixStaticTransform`, and a geometry-AS. If the resulting instance references an instance-AS, set `sbtOffset` to zero.

5.4 Build flags

An acceleration structure build can be controlled using the values of the `OptixBuildFlags` enum. To enable random vertex access on an acceleration structure, use `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS`. (See "Vertex random access" (page 75).) To steer trade-offs between build performance, runtime traversal performance and acceleration structure memory usage, use `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` and `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD`. For curve primitives in particular, these flags control splitting; see "Splitting curve segments" (page 59).

The flags `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` and `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD` are mutually exclusive. To combine multiple flags that are not mutually exclusive, use the logical "or" operator.

5.5 Dynamic updates

Building an acceleration structure can be computationally costly. Applications may choose to update an existing acceleration structure using modified vertex data or bounding boxes. Updating an existing acceleration structure is generally much faster than rebuilding. However, the quality of the acceleration structure may degrade if the data changes too much after an update, for example, through explosions or other chaotic transitions—even if for only parts of the mesh. The degraded acceleration structure may result in slower traversal

performance as compared to an acceleration structure built from scratch from the modified input data.

To allow updates of an acceleration structure, set `OPTIX_BUILD_FLAG_ALLOW_UPDATE` in the build flags.

For example:

Listing 5.8

```
accelOptions.buildFlags = OPTIX_BUILD_FLAG_ALLOW_UPDATE;
accelOptions.operation  = OPTIX_BUILD_OPERATION_BUILD;
```

To update the previously built acceleration structure, set `OPTIX_BUILD_OPERATION_UPDATE` and then call `optixAccelBuild` on the same output data. All other options are required to be identical to the original build. The update is done in-place on the output data.

For example:

Listing 5.9

```
accelOptions.operation = OPTIX_BUILD_OPERATION_UPDATE;

void* d_tempUpdate;
cudaMalloc(&d_tempUpdate, bufferSizes.tempUpdateSizeInBytes);

optixAccelBuild(optixContext, cuStream, &accelOptions,
   buildInputs, 2, d_tempUpdate,
   bufferSizes.tempUpdateSizeInBytes, d_output,
   bufferSizes.outputSizeInBytes, &outputHandle, nullptr, 0);
```

Updating an acceleration structure may require a different amount of memory than the original build.

When updating an existing acceleration structure, only the device pointers and/or their buffer content may be changed. You cannot change the number of build inputs, the build input types, build flags, traversable handles for instances (for an instance-AS), or the number of vertices, indices, AABBs, instances, SBT records or motion keys. Changes to any of these things may result in undefined behavior, including GPU faults.

Note the following:

- When using indices, changing the connectivity or, in general, using shuffled vertex positions will work, but the quality of the acceleration structure will likely degrade substantially.
- During an animation operation, geometry that should be invisible to the camera should not be "removed" from the scene, either by moving it very far away or by converting it into a degenerate form. Such changes to the geometry will also degrade the acceleration structure.

In these cases, it is more efficient to re-build the geometry-AS and/or the instance-AS, or to use the respective masking and flags.

5.6 Relocation

Geometry acceleration structures can be copied and moved, however they may not be used until `optixAccelRelocate` has been called to update the copied acceleration structure and generate the new traversable handle. Any acceleration structure may be relocated, including acceleration structures.

The copy does not need to be on the original device. This enables the copying of acceleration structure data to compatible devices without rebuilding the acceleration structure.

To relocate an acceleration structure, an `OptixAccelRelocationInfo` object is filled using `optixAccelGetRelocationInfo` and the traversable handle of the source acceleration structure. This object can then be used to determine if relocation to a device (as specified with an `OptixDeviceContext`) is possible. This is done using `optixAccelCheckRelocationCompatibility`. If the target device is compatible, the source acceleration structure may be copied to that device.

If an acceleration structure with instances is relocated, the traversable handles for the instances should be specified by `optixAccelRelocate` with the instanceTraversableHandles and numInstanceTraversableHandles parameters. The geometric bounds of the acceleration structure are not updated, so instanceTraversableHandles should correspond to either the same instances as the source (when relocating the instance-AS without relocating the geometry-AS) or the corresponding relocated instances (if relocating both types of acceleration structures).

The following example relocates the geometry and instance acceleration structure to new CUDA allocations on the same device:

Listing 5.10

```
CUdeviceptr d_relocatedGas = 0;
cudaMalloc((void**)&d_relocatedGas,
   gasBufferSizes.outputSizeInBytes);
cudaMemcpy((void*)d_relocatedGas, (void*)d_gasOutputBuffer,
   gasBufferSizes.outputSizeInBytes,
   cudaMemcpyDeviceToDevice);

OptixAccelRelocationInfo gasInfo = {};
optixAccelGetRelocationInfo(context, gasHandle, &gasInfo);
{
   int compatible = 0;
   optixAccelCheckRelocationCompatibility(
      context, &gasInfo, &compatible);
   if (compatible != 1) {
      fprintf(stderr,
         "Device isn't compatible for relocation "
         "of geometry acceleration structures.");
      exit(2);
   }
}
OptixTraversableHandle relocatedGasHandle = 0;
optixAccelRelocate(context, 0, &gasInfo, 0, 0, d_relocatedGas,
```

This is unnecessary because the copy operation's source and destination are on the same device, but is here to illustrate the API.

```
    gasBufferSizes.outputSizeInBytes, &relocatedGasHandle);

CUdeviceptr d_relocatedIas = 0;
cudaMalloc(
    (void**)&d_relocatedIas,
    iasBufferSizes.outputSizeInBytes);
cudaMemcpy((void*)d_relocatedIas, (void*)d_iasOutputBuffer,
    iasBufferSizes.outputSizeInBytes,
    cudaMemcpyDeviceToDevice);
OptixAccelRelocationInfo iasInfo = {};
optixAccelGetRelocationInfo(
    context, iasHandle, &iasInfo);
OptixTraversableHandle relocatedIasHandle = 0;

std::vector<OptixTraversableHandle>
    instanceHandles(g_instances.size());

CUdeviceptr d_instanceTravHandles = 0;
cudaMalloc(
    (void**)&d_instanceTravHandles,
    sizeof(OptixTraversableHandle) * instanceHandles.size());

for (unsigned int i = 0; i < g_instances.size(); ++i)
    instanceHandles[i] = relocatedGasHandle;

cudaMemcpy((void*)d_instanceTravHandles, instanceHandles.data(),
    sizeof(OptixTraversableHandle) * instanceHandles.size(),
    cudaMemcpyHostToDevice);

optixAccelRelocate(
    context, 0, &iasInfo,
    d_instanceTravHandles,
    instanceHandles.size(),
    d_relocatedIas,
    iasBufferSizes.outputSizeInBytes,
    &relocatedIasHandle);
```

Relocate the instance acceleration structure

5.7 Compacting acceleration structures

A post-process can compact an acceleration structure after construction. This process can significantly reduce memory usage, but it requires an additional pass. The build and compact operations are best performed in batches to ensure that device synchronization does not degrade performance. The compacted size depends on the acceleration structure type and its properties and on the device architecture.

To compact the acceleration structure as a post-process, do the following:

1. Build flag `OPTIX_BUILD_FLAG_ALLOW_COMPACTION` must be set in the `OptixAccelBuildOptions` as passed to `optixAccelBuild`.

2. The emit property `OPTIX_PROPERTY_TYPE_COMPACTED_SIZE` must be set in the `OptixAccelEmitDesc` as passed to `optixAccelBuild`. This property is generated on the device and it must be copied back to the host if it is required for allocating the new output buffer. The application may then choose to compact the acceleration structure using `optixAccelCompact`.
3. The `optixAccelCompact` call should be guarded by an `if (compactedSize < outputSize)` to avoid the compacting pass in cases where it is not beneficial. Note that this check requires a copy of the compacted size (as queried by `optixAccelBuild`) from the device memory to host memory.

A compacted acceleration structure may still be used for traversal and update operations or as input to `optixAccelRelocate`.

For example:

Listing 5.11

```
size_t *d_compactedSize;
OptixAccelEmitDesc property = {};
property.type = OPTIX_PROPERTY_TYPE_COMPACTED_SIZE;
property.result = d_compactedSize;

OptixTraversableHandle accelHandle = 0;

accelOptions.buildFlags = OPTIX_BUILD_FLAG_ALLOW_COMPACTION;

optixAccelBuild(optixContext, cuStream, &accelOptions,
   buildInputs, 2, d_tempUpdate, bufferSizes.tempSizeInBytes,
   d_output, bufferSizes.outputSizeInBytes, &accelHandle,
   &property, 1);

size_t compactedSize;
cudaMemcpy(&compactedSize, d_compactedSize,
   sizeof(size_t),
   cudaMemcpyDeviceToHost);

void *d_compactedOutputBuffer;
cudaMalloc(&d_compactedOutputBuffer, compactedSize);

if (compactedSize < bufferSizes.outputSizeInBytes) {
   OptixTraversableHandle compactedAccelHandle = 0;
   optixAccelCompact(optixContext, stream, accelHandle,
      d_compactedOutputBuffer, compactedSize,
      &compactedAccelHandle);
}
```

A compacted acceleration structure does not reference the uncompacted input data. The application is free to reuse the uncompacted acceleration structure without invalidating the compacted acceleration structure. Ensure that the copy output does not overlap the input, because it does not work in-place.

A compacted acceleration structure supports dynamic updates only if the uncompacted source acceleration structure was built with the `OPTIX_BUILD_FLAG_ALLOW_UPDATE` build flag. (See "Dynamic updates" (page 26).) The amount of temporary memory required for a dynamic update is the same for the uncompacted acceleration structure and compacted acceleration structure. Note that using the following build flags will lead to less memory savings when enabling the compacting post-process:

- `OPTIX_BUILD_FLAG_ALLOW_UPDATE`
- `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD`

5.8 Traversable objects

The instances in an instance-AS may reference transform traversables, as well as geometry-ASs. Transform traversables are fully managed by the application. The application needs to create these traversables manually in device memory in a specific form. The function `optixConvertPointerToTraversableHandle` converts a raw pointer into a traversable handle of the specified type. The traversable handle can then be used to link traversables together.

In device memory, all traversable objects need to be 64-byte aligned. Note that moving a traversable to another location in memory invalidates the traversable handle. The application is responsible for constructing a new traversable handle and updating any other traversables referencing the invalidated traversable handle.

The traversable handle is considered opaque and the application should not rely on any particular mapping of a pointer to the traversable handle.

For example:

Listing 5.12

```
OptixMatrixMotionTransform transform = {};

...          Setup motion transform

cudaMemcpy(d_transform, &transform,
    sizeof(OptixMatrixMotionTransform),
    cudaMemcpyHostToDevice);

OptixTraversableHandle transformHandle = 0;
optixConvertPointerToTraversableHandle(
    optixContext, d_transform,
    OPTIX_TRAVERSABLE_TYPE_MATRIX_MOTION_TRANSFORM,
    &transformHandle);

OptixInstance instance = {};
instance.traversableHandle = transformHandle;

...          Setup instance description
```

5.8.1 Traversal of a single geometry acceleration structure

The traversable handle passed to `optixTrace` can be a traversable handle created from a geometry-AS. This can be useful for scenes where single geometry-AS objects represent the root of the scene graph.

If the modules and pipeline only need to support single geometry-AS traversables, it is beneficial to change the `OptixPipelineCompileOptions::traversableGraphFlags` from `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_ANY` to `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_GAS`.

This signals to NVIDIA OptiX 7 that no other traversable types require support during traversal.

5.9 Motion blur

Motion blur is supported by `OptixMatrixMotionTransform`, `OptixSRTMotionTransform` and acceleration structure traversables. Several motion blur options are specified in the `OptixMotionOptions` struct: the number of motion keys, flags, and the beginning and ending motion times corresponding to the first and last key. The remaining motion keys are evenly spaced between the beginning and ending times.

Typically, motion transforms must specify at least two motion keys. One exception is the acceleration structure build that generally accepts an instance of an `OptixAccelBuildOptions` with field `OptixMotionOptions` set to `NULL`. This effectively disables motion blur for the acceleration structure and ignores the motion beginning and ending times, along with the motion flags.

Traversables linked together in a scene graph may have a different number of motion keys. For example, a static instance-AS (zero motion key) could be linked to a mixture of geometry-ASs with zero or multiple motion keys per structure.

Motion boundary conditions are specified by using flags. By default, the motion is clamped at the boundaries, meaning it is static and visible. To remove the traversable before the beginning time, set `OPTIX_MOTION_FLAG_START_VANISH`; to remove it after the ending time, set `OPTIX_MOTION_FLAG_END_VANISH`.

For example:

Listing 5.13

```
OptixMotionOptions motionOptions = {};
motionOptions.numKeys = 3;
motionOptions.timeBegin = -1f;
motionOptions.timeEnd = 1.5f;
motionOptions.flags = OPTIX_MOTION_FLAG_NONE;
```

For more information about these concepts, see the motion blur documentation in the NVIDIA OptiX 6.5 Programming Guide.[8]

8. `https://raytracing-docs.nvidia.com/optix6/guide_6_5/index.html#motion_blur_math#motion-blur`

5.9.1 Motion acceleration structure

Use `optixAccelBuild` to build a motion acceleration structure. The motion options are part of the build options (`OptixAccelBuildOptions`) and apply to all build inputs. Build inputs must specify primitive vertex buffers (for `OptixBuildInputTriangleArray` and `OptixBuildInputCurveArray`), radius buffers (for `OptixBuildInputCurveArray`), and AABB buffers (for `OptixBuildInputCustomPrimitiveArray` and `OptixBuildInputInstanceArray`) for all motion keys. These are interpolated during traversal to obtain the continuous motion vertices and AABBs between the begin and end time.

For example:

Listing 5.14

```
CUdeviceptr d_motionVertexBuffers[3];

OptixBuildInputTriangleArray buildInput = {};
buildInput.vertexBuffers = d_motionVertexBuffers;
buildInput.numVertices = numVertices;
```

5.9.2 Motion matrix transform

The motion matrix transform traversable (`OptixMatrixMotionTransform`) transforms the ray during traversal using a motion matrix. The traversable provides a 3x4 row-major object-to-world transformation matrix for each motion key. The final motion matrix is constructed during traversal by interpolating the elements of the matrices at the nearest motion keys.

The `OptixMatrixMotionTransform` struct has a dynamic size, dependent on the number of motion keys. The struct specifies the header and the first two motion keys for convenience; when using more than than two keys, compute the size required for additional keys.

For example:

Listing 5.15

```
#define NUM_MOTION_KEYS 3
float matrixKeys[NUM_MOTION_KEYS][12];
...

size_t transformSizeInBytes = sizeof(OptixMatrixMotionTransform)
   + (NUM_MOTION_KEYS-2) * 12 * sizeof(float);

OptixMatrixMotionTransform *transform =
   (OptixMatrixMotionTransform*) malloc(transformSizeInBytes);

transform->motionOptions.numKeys = NUM_MOTION_KEYS;
transform->motionOptions.timeBegin = -1f;
transform->motionOptions.timeEnd = 1.5f;
transform->motionOptions.flags = 0;
memcpy(transform->transform, matrixKeys,
NUM_MOTION_KEYS * 12 * sizeof(float));
```

5.9.3 Motion scale/rotate/translate transform

The behavior of the motion transform `OptixSRTMotionTransform` is similar to the matrix motion transform `OptixMatrixMotionTransform`. In `OptixSRTMotionTransform` the object-to-world transforms per motion key are specified as a scale, rotation and translation (SRT) decomposition instead of a single 3x4 matrix. Each motion key is a struct of type `OptixSRTData`, which consists of 16 floats:

Listing 5.16

```
typedef struct OptixSRTData {
  float sx, a, b, pvx, sy, c, pvy, sz, pvz, qx, qy, qz, qw, tx, ty, tz;
} OptixSRTData;
```

These 16 floats define the three components for scaling, rotation and translation whose product is the motion transformation:

- The scaling matrix S

$$S = \begin{bmatrix} sx & a & b & pvx \\ 0 & sy & c & pvy \\ 0 & 0 & sz & pvz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  defines an affine transformation that can include scale, shear, and a translation. The translation enables a pivot point to be defined for the scale, shear as well as the subsequent rotation.

- The quaternion R

$$R = \begin{bmatrix} qx & qy & qz & qw \end{bmatrix}$$

describes a rotation with angular component $\texttt{qw} = cos(\theta/2)$ and other components qx, qy and qz, where

$$\begin{bmatrix} qx & qy & qz \end{bmatrix} = sin(\theta/2) * \begin{bmatrix} ax & ay & az \end{bmatrix}$$

and where the axis $\begin{bmatrix} ax & ay & az \end{bmatrix}$ is normalized.

- The translation T

$$T = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \end{bmatrix}$$

defines another translation that is applied after the rotation. Typically, this translation includes the inverse translation from the matrix S to undo the pivot point transformation.

To obtain the effective transformation at time t, the elements of the components of S, R, and T are interpolated linearly. The components are then multiplied to obtain the combined transformation $C = T \times R \times S$. The transformation C is the effective object-to-world transformations at time t, and C^{-1} is the effective world-to-object transformation at time t.

Example 1 – Rotation about the origin

Use two motion keys. Set the first key to identity values. For the second key, define a quaternion from an axis and angle, for example, a 60-degree rotation about the z axis is given by:

$$Q = \begin{bmatrix} 0 & 0 & sin(\pi/6) & cos(\pi/6) \end{bmatrix}$$

Example 2 – Rotation about a pivot point

Use two motion keys. Set the first key to identity values. Represent the pivot point as a translation P, and define the second key as follows:

$$S' = P^{-1} \times S$$

$$T' = T \times P$$

$$C = T' \times R \times S'$$

Example 3 – Scaling about a pivot point

Use two motion keys. Set the first key to identity values. Represent the pivot as a translation $G = \begin{bmatrix} G_x & G_y & fG_z \end{bmatrix}$ and modify the pivot point described above:

$$P'_x = P_x + (-S_x * G_x + G_x)$$

$$P'_y = P_y + (-S_y * G_y + G_y)$$

$$P'_z = P_z + (-S_z * G_z + G_z)$$

6 Program pipeline creation

The following API functions are described in this section:

```
optixModuleCreateFromPTX
optixModuleDestroy
optixProgramGroupCreate
optixProgramGroupGetStackSize
optixPipelineCreate
optixPipelineDestroy
optixPipelineSetStackSize
```

Programs are first compiled into *modules* of type `OptixModule`. One or more modules are combined to create a *program group* of type `OptixProgramGroup`. Those program groups are then linked into an `OptixPipeline` on the GPU. This is similar to the compile and link process commonly found in software development. The program groups are also used to initialize the header of the SBT record associated with those programs.

The three create methods, `optixModuleCreateFromPTX`, `optixProgramGroupCreate`, and `optixPipelineCreate` take an optional log string. This string is used to report information about any compilation that may have occurred, such as compile errors or verbose information about the compilation result. To detect truncated output, the size of the log message is reported as an output parameter. It is not recommended that you call the function again to get the full output because this could result in unnecessary and lengthy work, or different output for cache hits. If an error occurred, the information that would be reported in the log string is also reported by the device context log callback (when provided).

Both mechanisms are provided for these create functions to allow a convenient mechanism for pulling out compilation errors from parallel creation operations without having to determine which output from the logger corresponds to which API invocation.

Symbols in `OptixModule` objects may be unresolved and contain `extern` references to variables and `__device__` functions.

These symbols can be resolved during pipeline creation using the symbols defined in the pipeline modules. Duplicate symbols will trigger an error.

A *pipeline* contains all programs that are required for a particular ray-tracing launch. An application may use a different pipeline for each launch, or may combine multiple ray-generation programs into a single pipeline.

Most NVIDIA OptiX 7 API functions do not own any significant GPU state; Streaming Assembly (SASS) instructions, which define the executable binary programs in a pipeline, are an exception. The `OptixPipeline` owns the CUDA resource associated with the compiled SASS and it is held until the pipeline is destroyed. This allocation is proportional to the amount of compiled code in the pipeline, typically tens of kilobytes to a few megabytes. However, it is possible to create complex pipelines that require substantially more memory,

especially if large static initializers are used. Wherever possible, Exercise caution in the number and size of the pipelines.

Module lifetimes need to extend to the lifetimes of program groups that reference them. After using modules to create an `OptixPipeline` through the `OptixProgramGroup` objects, modules may be destroyed with `optixModuleDestroy`.

6.1 Program input

NVIDIA OptiX 7 programs are encoded in the *parallel thread execution instruction set* (PTX)[9] language. To create PTX programs, you compile CUDA source files using the NVIDIA `nvcc` offline compiler[10] or `nvrtc` JIT compiler.[11] The CUDA code includes PTX device headers used during compilation.

For example:

```
nvcc -ptx -Ipath-to-optix-sdk/include --use_fast_math myprogram.cu -o myprogram.ptx
```

To specify architectural instructions not included in the default instruction set, use the `nvcc` argument `-arch sm_XY`. Using `--use_fast_math` is also recommended. The streaming multiprocessor (SM) target of the input PTX must be less than or equal to the SM version of the GPU for which the module is compiled.

The NVIDIA OptiX 7 programming model supports the *multiple instruction, multiple data* (MIMD) subset of CUDA. Execution must be independent of other threads. For this reason, shared memory usage and warp-wide or block-wide synchronization—such as barriers—are not allowed in the input PTX code. All other GPU instructions are allowed, including math, texture, atomic operations, control flow, and loading data to memory. Special warp-wide instructions like vote and ballot are allowed, but can yield unexpected results as the locality of threads is not guaranteed and neighboring threads can change during execution, unlike in the full CUDA programming model. Still, warp-wide instructions can be used safely when the algorithm in question is independent of locality by, for example, implementing warp-aggregated atomic adds.

The memory model is consistent only within the execution of a single launch index, which starts at the ray-generation invocation and only with subsequent programs reached from any `optixTrace` or callable program. This includes writes to stack allocated variables. Writes from other launch indices may not be available until after the launch is complete. If needed, atomic operations may be used to share data between launch indices, as long as an ordering between launch indices is not required. Memory fences are not supported.

The input PTX should include one or more NVIDIA OptiX 7 programs. The type of program affects how the program can be used during the execution of the pipeline. These program types are specified by prefixing the program's name with the following:

9. `https://docs.nvidia.com/cuda/parallel-thread-execution/`

10. `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/`

11. `https://docs.nvidia.com/cuda/nvrtc/`

Program type	*Function name prefix*
Ray generation	`__raygen__`
Intersection	`__intersection__`
Any-hit	`__anyhit__`
Closest-hit	`__closesthit__`
Miss	`__miss__`
Direct callable	`__direct_callable__`
Continuation callable	`__continuation_callable__`
Exception	`__exception__`

If a particular function needs to be used with more than one type, then multiple copies with corresponding program prefixes should be generated.

In addition, each program may call a specific set of device-side intrinsics that implement the actual ray-tracing-specific features. (See "Device-side functions" (page 67).)

6.2 Module creation

A module may include multiple programs of any program type. Two option structs control the parameters of the compilation process:

`OptixPipelineCompileOptions`
: Must be identical for all modules used to create program groups linked in a single pipeline.

`OptixModuleCompileOptions`
: May vary across the modules within the same pipeline.

These options control general compilation settings, for example, the level of optimization. `OptixPipelineCompileOptions` controls features of the API such as the usage of custom any-hit programs, curve primitives, motion blur, exceptions, and the number of 32-bit values usable in ray payload and primitive attributes. For example:

Listing 6.1

```
OptixModuleCompileOptions moduleCompileOptions = {};
moduleCompileOptions.maxRegisterCount =
   OPTIX_COMPILE_DEFAULT_MAX_REGISTER_COUNT;
moduleCompileOptions.optLevel =
   OPTIX_COMPILE_OPTIMIZATION_DEFAULT;
moduleCompileOptions.debugLevel =
   OPTIX_COMPILE_DEBUG_LEVEL_LINEINFO;

OptixPipelineCompileOptions pipelineCompileOptions = {};
pipelineCompileOptions.usesMotionBlur = false;
pipelineCompileOptions.traversableGraphFlags =
   OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_LEVEL_INSTANCING;
pipelineCompileOptions.numPayloadValues = 2;
pipelineCompileOptions.numAttributeValues = 2;
```

```
pipelineCompileOptions.exceptionFlags = OPTIX_EXCEPTION_FLAG_NONE;
pipelineCompileOptions.pipelineLaunchParamsVariableName = "params";
pipelineCompileOptions.usesPrimitiveTypeFlags = 0;

OptixModule module = nullptr;
char* ptxData = ...;
size_t logStringSize = sizeof(logString);

OptixResult res = optixModuleCreateFromPTX(
   optixContext,
   &moduleCompileOptions,
   &pipelineCompileOptions,
   ptxData, ptx.size(),
   logString, &logStringSize,
   &module);
```

The `numAttributeValues` field of `OptixPipelineCompileOptions` defines the number of 32-bit words that are reserved to store the attributes. This corresponds to the attribute definition in `optixReportIntersection`. See "Reporting intersections and attribute access" (page 72).

Note: For best performance when your scene contains nothing but built-in triangles, set `OptixPipelineCompileOptions::usesPrimitiveTypeFlags` to just `OPTIX_PRIMITIVE_TYPE_FLAGS_TRIANGLE`.

6.3 Pipeline launch parameter

You specify launch-varying parameters or values that must be accessible from any module through a user-defined variable named in `OptixPipelineCompileOptions`. In each module that needs access, declare this variable with `extern` or `extern "C"` linkage and the `__constant__` memory specifier. The size of the variable must match across all modules in a pipeline. Variables of equal size but differing types may trigger undefined behavior.

For example, the header file in Listing 6.2 defines the variable to shared, named `params`, as an instance of the `Params` struct:

Listing 6.2 – Struct defined in header file `params.h`

```
struct Params
{
   uchar4* image;
   unsigned int image_width;
};
extern "C" __constant__ Params params;
```

Listing 6.3 shows that by including header file `params.h`, programs can access and set the values of the shared `params` struct instance:

Listing 6.3 – Use of header file `params.h` *in OptiX program*

```
#include "params.h"

extern "C"
__global__ void draw_solid_color()
{
    ...
    unsigned int image_index =
       launch_index.y * params.image_width + launch_index.x];
    params.image[image_index] = make_uchar4(255, 0, 0);
}
```

6.3.1 Parameter specialization

In some cases it could be beneficial to specialize modules in a pipeline to toggle specific features on and off. For example, users may wish to compile support in their shaders for calculating shadow rays, but may wish to disable this support if the scene parameters do not require them. Users could support this with either multiple versions of the PTX program or by reading a value from the pipeline launch parameters to indicate whether shadows are supported. Multiple versions of the PTX program would allow for the best performance, but come at the cost of maintaining and storing all those program versions. NVIDIA OptiX 7 provides a mechanism for specializing values in the pipeline launch parameters.

During compilation of the module, NVIDIA OptiX 7 will attempt to find loads to the pipeline launch parameter struct that are specified by `OptixPipelineCompileOptions::pipelineLaunchParamsVariableName` within a given range. These specified loads are then each replaced with a predefined value. Compiler optimization passes use those constant values.

The struct `OptixModuleCompileBoundValueEntry` in Listing 6.4 can specify an array of bytes that will replace a portion of the pipeline parameters:

Listing 6.4

```
struct OptixModuleCompileBoundValueEntry {
    size_t pipelineParamOffsetInBytes;
    size_t sizeInBytes;
    const void* boundValuePtr;
    const char* annotation;   Optional string to display
};
```

Listing 6.5 shows how array of `OptixModuleCompilePBoundValueEntry` structs can be specified in `OptixModuleCompileOptions` during module compilation:

Listing 6.5

```
struct OptixModuleCompileOptions {
    ...
    const OptixModuleCompileBoundValueEntry* boundValues;
    unsigned int numBoundValues;
};
```

Listing 6.6 is an example of specializing a module to disable shadow rays:

Listing 6.6 – Device code that references the pipeline launch parameters to determine if shadows are enabled

```
struct LP {
   bool useShadows;
};

extern "C" {
__constant__ LP params;
}

extern "C"
__global__ void __closesthit__ch()
{
   float3 shadowColor = make_float3( 1.f, 1.f, 1.f );
   if ( params.useShadows )
   {
      shadowColor = traceShadowRay(...);
   }
   ...
}
```

On the host side, Listing 6.7 shows the implementation of the launch parameters:

Listing 6.7 – Host code to specialize the pipeline launch parameters

```
LP launchParams = {};
launchParams.useShadows = false;

OptixModuleCompileBoundValueEntry useShadow = {};
useShadow.pipelineParamOffset = offsetof(LP, useShadows);
useShadow.sizeInBytes = sizeof( LP::useShadows );
useShadow.boundValuePtr = &launchParams.useShadows;

OptixModuleCompileOptions moduleCompileOptions = {};
moduleCompileOptions.boundValues = &useShadow;
moduleCompileOptions.numBoundValues = 1;
...
optixModuleCompileFromPTX( ..., moduleCompileOptions, ... );
```

This pipeline launch parameter specialization makes the code of Listing 6.8 possible:

Listing 6.8

```
extern "C"
__global__ void __closesthit__ch()
{
   float3 shadowColor = make_float3( 1.f, 1.f, 1.f );
   if ( false )
   {
```



```
        shadowColor = traceShadowRay(...);
    }
    ...
}
```

Subsequent optimization would remove unreachable code, producing Listing 6.9:

Listing 6.9

```
extern "C"
__global__ void __closesthit__ch()
{
    float3 shadowColor = make_float3( 1.f, 1.f, 1.f );
    ...
}
```

The bound values are intended to represent a constant value in the `pipelineParams`. NVIDIA OptiX 7 will attempt to locate all loads from the `pipelineParams` and correlate them to the appropriate bound value. However, there are cases where these loads and bound values cannot be safely or reliably correlated. For example, correlation is not possible if the pointer to the `pipelineParams` is passed as an argument to a non-inline function or if the offset of the load to the `pipelineParams` cannot be statically determined due to access in a loop. No module should rely on the value being specialized in order to work correctly. The values in the `pipelineParams` specified on `optixLaunch` should match the bound value. If validation mode is enabled on the context, NVIDIA OptiX 7 will verify that the bound values that are specified match the values in `pipelineParams` specified to `optixLaunch`.

If caching is enabled, changes in these values will result in newly compiled modules.

The `pipelineParamOffset` and `sizeInBytes` must be within the bounds of the `pipelineParams` variable or `OPTIX_ERROR_INVALID_VALUE` will be returned from `optixModuleCreateFromPTX`.

If more than one bound value overlaps or the size of a bound value is equal to 0, an `OPTIX_ERROR_INVALID_VALUE` will be returned from `optixModuleCreateFromPTX`.

The same set of bound values do not need to be used for all modules in a pipeline, but overlapping values between modules must have the same value. `OPTIX_ERROR_INVALID_VALUE` will be returned from `optixPipelineCreate` otherwise.

6.4 Program group creation

`OptixProgramGroup` objects are created from one to three `OptixModule` objects and are used to fill the header of the SBT records. (See "Shader binding table" (page 51).) There are five types of program groups.

```
OPTIX_PROGRAM_GROUP_KIND_RAYGEN
OPTIX_PROGRAM_GROUP_KIND_MISS
OPTIX_PROGRAM_GROUP_KIND_EXCEPTION
OPTIX_PROGRAM_GROUP_KIND_HITGROUP
OPTIX_PROGRAM_GROUP_KIND_CALLABLES
```

Modules can contain more than one program. The program in the module is designated by its entry function name as part of the `OptixProgramGroupDesc` struct passed to

`optixProgramGroupCreate`. Four program groups can contain only a single program; only `OPTIX_PROGRAM_GROUP_HITGROUP` can designate up to three programs for the closest-hit, any-hit, and intersection programs.

Programs from modules can be used in any number of `OptixProgramGroup` objects. The resulting program groups can be used to fill in any number of SBT records. Program groups can also be used across pipelines as long as the compilation options match.

The lifetime of a module must extend to the lifetime of any `OptixProgramGroup` that references that module.

A hit group specifies the intersection program used to test whether a ray intersects a primitive, together with the hit shaders to be executed when a ray does intersect the primitive. For built-in primitive types, a built-in intersection program should be obtained from `optixBuiltinISModuleGet()` and used in the hit group. As a special case, the intersection program is not required – and is ignored – for triangle primitives.

The following examples show how to construct a single hit-group program group:

Listing 6.10 – Construct hit-group for custom primitives

```
OptixModule shadingModule, intersectionModule;
...          shadingModule and intersectionModule created here by optixModuleCreateFromPTX

OptixProgramGroupDesc pgDesc = {};
pgDesc.kind = OPTIX_PROGRAM_GROUP_KIND_HITGROUP;
pgDesc.hitgroup.moduleCH = shadingModule;
pgDesc.hitgroup.entryFunctionNameCH = "__closesthit__shadow";
pgDesc.hitgroup.moduleAH = shadingModule;
pgDesc.hitgroup.entryFunctionNameAH = "__anyhit__shadow";
pgDesc.hitgroup.moduleIS = intersectionModule;
pgDesc.hitgroup.entryFunctionNameIS = "__intersection__sphere";

OptixProgramGroupOptions pgOptions = {};
OptixProgramGroup sphereGroup = nullptr;
optixProgramGroupCreate(
    optixContext,
    &pgDesc,          programDescriptions

    1,                numProgramGroups

    &pgOptions,       programOptions
    logString, sizeof(logString),
    &sphereGroup);    programGroup
```

Listing 6.11 – Construct hit-group for built-in curves primitives

```
OptixModule shadingModule, intersectionModule;
...          shadingModule created here by optixModuleCreateFromPTX
```

```
OptixBuiltinISOptions builtinISOptions = {};
builtinISOptions.builtinISModuleType =
   OPTIX_PRIMITIVE_TYPE_ROUND_CUBIC_BSPLINE;
OptixResult res = optixBuiltinISModuleGet(
   optixContext,
   &moduleCompileOptions,
   &pipelineCompileOptions,
   &builtinISOptions,
   &intersectionModule);

OptixProgramGroupDesc pgDesc= {};
pgDesc.kind = OPTIX_PROGRAM_GROUP_KIND_HITGROUP;
pgDesc.hitgroup.moduleCH = shadingModule;
pgDesc.hitgroup.entryFunctionNameCH = "__closesthit__curves";
pgDesc.hitgroup.moduleAH = nullptr;   Any-hit shader is optional
pgDesc.hitgroup.entryFunctionNameAH = nullptr;
pgDesc.hitgroup.moduleIS = intersectionModule;
pgDesc.hitgroup.entryFunctionNameIS = nullptr;   No name for built-in IS

OptixProgramGroupOptions pgOptions = {};
OptixProgramGroup curvesGroup = nullptr;
optixProgramGroupCreate(
   optixContext,
   &pgDesc,
   1,
   &pgOptions,
   logString, sizeof(logString),
   &curvesGroup);
```

Multiple program groups of varying kinds can be constructed with a single call to `optixProgramGroupCreate`. The following code demonstrates the construction of a ray-generation and miss program group.

Listing 6.12

```
OptixModule rg, miss;
...   Ray-generation and miss programs created here by optixModuleCreateFromPTX

OptixProgramGroupDesc pgDesc[2] = {};
pgDesc[0].kind = OPTIX_PROGRAM_GROUP_KIND_MISS;
pgDesc[0].miss.module = miss1;
pgDesc[0].miss.entryFunctionName = "__miss__radiance";
pgDesc[1].kind = OPTIX_PROGRAM_GROUP_KIND_RAYGEN;
pgDesc[1].raygen.module = rg;
pgDesc[1].raygen.entryFunctionName = "__raygen__pinhole_camera";
OptixProgramGroupOptions pgOptions = {};
OptixProgramGroup raygenMiss[2];
optixProgramGroupCreate(
   optixContext,
```

```
    &pgDesc,    programDescriptions
    2,          numProgramGroups
    &pgOptions, programOptions
    logString, sizeof(logString),
    raygenMiss); programGroup
```

Options defined in `OptixProgramGroupOptions` may vary across program groups linked into a single pipeline, similar to `OptixModuleCompileOptions`.

6.5 Pipeline linking

After all program groups of a pipeline are defined, they must be linked into an `OptixPipeline`. The resulting `OptixPipeline` object is then used to invoke a ray-generation launch.

When the `OptixPipeline` is linked, some fixed function components may be selected based on `OptixPipelineLinkOptions` and `OptixPipelineCompileOptions`. These options were previously used to compile the modules in the pipeline. The link options consist of the maximum recursion depth setting for recursive ray tracing, along with pipeline level settings for debugging. However, the value for the maximum recursion depth has an upper limit that overrides an limit set by the link options. (See "Limits" (page 65).)

For example, the following code creates and links an `OptixPipeline`:

Listing 6.13

```
OptixPipeline pipeline = nullptr;

OptixProgramGroup programGroups[3] =
   { raygenMiss[0], raygenMiss[1], sphereGroup };

OptixPipelineLinkOptions pipelineLinkOptions = {};
pipelineLinkOptions.maxTraceDepth = 1;
pipelineLinkOptions.debugLevel = OPTIX_COMPILE_DEBUG_LEVEL_FULL;
optixPipelineCreate(
   optixContext,
   &pipelineCompileOptions,
   &pipelineLinkOptions,
   programGroups,
   3,
   logString, sizeof(logString),
   &pipeline);
```

After calling `optixPipelineCreate`, the fully linked module is loaded into the driver.

NVIDIA OptiX 7 uses a small amount of GPU memory per pipeline. This memory is released when the pipeline or device context is destroyed.

6.6 Pipeline stack size

The programs in a module may consume two types of stack structure : a *direct stack* and a *continuation stack*. The resulting stack needed for launching a pipeline depends on the resulting call graph, so the pipeline must be configured with the appropriate stack size. These sizes can be determined by the compiler for each program group. A pipeline may be reused for different call graphs as long as the set of programs is the same. For this reason, the pipeline stack size is configured separately from the pipeline compilation options.

The direct stack requirements resulting from ray-generation, miss, exception, closest-hit, any-hit and intersection programs and the continuation stack requirements resulting from exception programs are calculated internally and do not need to be configured. The direct stack requirements resulting from direct-callable programs, as well as the continuation stack requirements resulting from ray-generation, miss, closest-hit, any-hit, intersection, and continuation-callable programs need to be configured. If these are not configured explicitly, an internal default implementation is used. When the maximum depth of call trees of continuation-callable and direct-callable programs is two or less, the default implementation is correct (but not necessarily optimal) Even in cases where the default implementation is correct, Users can always provide more precise stack requirements based on their knowledge of a particular call graph structure.

To query individual program groups for their stack requirements, use `optixProgramGroupGetStackSize`. Use this information to calculate the total required stack sizes for a particular call graph of NVIDIA OptiX 7 programs. To set the stack sizes for a particular pipeline, use `optixPipelineSetStackSize`. For other parameters, helper functions are available to implement these calculations. The following is an explanation about how to compute the stack size for `optixPipelineSetStackSize`, starting from a very conservative approach, and refining the estimates step by step.

Let `cssRG` denote the maximum continuation stack size of all ray-generation programs; similarly for miss, closest-hit, any-hit, intersection, and continuation-callable programs. Let `dssDC` denote the maximum direct stack size of all direct callable programs. Let maxTraceDepth denote the maximum trace depth (as in `OptixPipelineLinkOptions::maxTraceDepth`), and let `maxCCDepth` and `maxDCDepth` denote the maximum depth of call trees of continuation-callable and direct-callable programs, respectively. Then a simple, conservative approach to compute the three parameters of `optixPipelineSetStackSize` is:

Listing 6.14

```
directCallableStackSizeFromTraversal = maxDCDepth * dssDC;
directCallableStackSizeFromState     = maxDCDepth * dssDC;

cssCCTree =
   maxCCDepth * cssCC;
```

Upper bound on continuation stack used by call trees of continuation callables

```
cssCHOrMSPlusCCTree =
   max(cssCH, cssMS) + cssCCTree;
```

Upper bound on continuation stack used by closest-hit or miss programs, including the call tree of continuation-callable programs

```
continuationStackSize =
   cssRG
 + cssCCTree
```

```
  + maxTraceDepth * cssCHOrMSPlusCCTree
  + cssIS
  + cssAH;
```

This computation can be improved in several ways. For the computation of continuationStackSize, the stack sizes `cssIS` and `cssAH` are not used on top of the other summands, but can be offset against one level of `cssCHOrMSPlusCCTree`. This gives a more complex but better estimate:

Listing 6.15

```
continuationStackSize =
   cssRG
  + cssCCTree
  + max(1, maxTraceDepth) - 1) * cssCHOrMSPlusCCTree
  + min(maxTraceDepth, 1) * max(cssCHOrMSPlusCCTree,cssIS+cssAH);
```

The preceding formulas are implemented by the helper function `optixUtilComputeStackSizes`.

The computation of the first two terms can be improved if the call trees of direct callable programs are analyzed separately based on the semantic type of their call site. In this context, call sites in any-hit and intersection programs count as traversal, whereas call sites in ray-generation, miss, and closest-hit programs count as state.

Listing 6.16

```
directCallableStackSizeFromTraversal =
   maxDCDepthFromTraversal * dssDCFromTraversal;
directCallableStackSizeFromState =
   maxDCDepthFromState * dssDCFromState;
```

This improvement is implemented by the helper function `optixUtilComputeStackSizesDCSplit`.

Depending on the scenario, these estimates can be improved further, sometimes substantially. For example, imagine there are two call trees of continuation-callable programs. One call tree is deep, but the involved continuation-callable programs need only a small continuation stack. The other call tree is shallow, but the involved continuation-callable programs needs a quite large continuation stack. The estimate of `cssCCTree` can be improved as follows:

Listing 6.17

```
cssCCTree = max(maxCCDepth1 * cssCC1, maxCCDepth2 * cssCC2);
```

This improvement is implemented by the helper function `optixUtilComputeStackSizesCssCCTree`.

Similar improvements might be possible for all expressions involving maxTraceDepth if the ray types are considered separately, for example, camera rays and shadow rays.

6.6.1 Constructing a path tracer

A simple path tracer can be constructed from two ray types: camera rays and shadow rays. The path tracer will consist only of ray-generation, miss, and closest-hit programs, and will not use any-hit, intersection, continuation-callable, or direct-callable programs. The camera rays will invoke only the miss and closest-hit programs `MS1` and `CH1`, respectively. `CH1` might trace shadow rays, which invoke only the miss and closest-hit programs `MS2` and `CH2`, respectively. That is, the maximum trace depth is two and the initial formulas simplify to:

Listing 6.18

```
directCallableStackSizeFromTraversal = maxDCDepth * dssDC;
directCallableStackSizeFromState     = maxDCDepth * dssDC;
continuationStackSize =
   cssRG + 2 * max(cssCH1, cssCH2, cssMS1, cssMS2);
```

However, from the call graph structure it is clear that `MS2` or `CH2` can only be invoked from `CH1`. This restriction allows for the following estimate:

Listing 6.19

```
continuationStackSize
   = cssRG + max(cssMS1, cssCH1 + max(cssMS2, cssCH2));
```

This estimate is never worse than the previous one, but often better, for example, in the case where the closest-hit programs have different stack sizes (and the miss programs do not dominate the expression).

The helper function `optixUtilComputeStackSizesSimplePathTracer` implements this formula by permitting two arrays of closest-hit programs instead of two single programs.

6.7 Compilation cache

Compilation work is triggered automatically when calling `optixModuleCreateFromPTX` or `optixProgramGroupCreate`, and also potentially during `optixPipelineCreate`. This work is automatically cached on disk if enabled on the `OptixDeviceContext`. Caching reduces compilation effort for recurring programs and program groups. While it is enabled by default, users can disable it through the use of `optixDeviceContextSetCacheEnabled`. See "Context" (page 15) for other options regarding the compilation cache.

Generally, cache entries are compatible with the same driver version and GPU type only.

7 Shader binding table

The *shader binding table* (SBT) is an array that contains information about the location of programs and their parameters. The SBT resides in device memory and is managed by the application.

7.1 Records

A *record* is an array element of the SBT that consists of a header and a data block. The header content is opaque to the application, containing information accessed by traversal execution to identify and invoke programs. The data block is not used by NVIDIA OptiX 7 and holds arbitrary program-specific application information that is accessible in the program. The header size is defined by the `OPTIX_SBT_RECORD_HEADER_SIZE` macro (currently 32 bytes).

The API function `optixSbtRecordPackHeader` and a given `OptixProgramGroup` object are used to fill the header of an SBT record. The SBT records must be uploaded to the device prior to an NVIDIA OptiX 7 launch. The contents of the SBT header are opaque, but can be copied or moved. If the same program group is used in more than one SBT record, the SBT header can be copied using plain device-side memory copies. For example:

Listing 7.1

```
template <typename T>
struct Record
{
   __align__(OPTIX_SBT_RECORD_ALIGNMENT)
      char header[OPTIX_SBT_RECORD_HEADER_SIZE];
   T data;
};

typedef Record<RayGenData> RayGenSbtRecord;

OptixProgramGroup raygenPG = nullptr;
...
RayGenSbtRecord rgSBT;
rgSBT.data.color = make_float3(1.0f, 1.0f, 0.0f);
optixSbtRecordPackHeader(raygenPG, &rgSBT);
CUdeviceptr deviceRaygenSbt = 0;
cudaMalloc((void**)&deviceRaygenSbt, sizeof(RayGenSbtRecord));
cudaMemcpy((void**)deviceRaygenSbt, &rgSBT,
   sizeof(RayGenSbtRecord), cudaMemcpyHostToDevice);
```

SBT headers can be reused between pipelines as long as the compile options match between modules and program groups. The data section of an SBT record can be accessed on the device using the `optixGetSbtDataPointer` device function.

7.2 Layout

A shader binding table is split into five sections, where each section represents a unique program group type:

Group	*Program types in group*	*Value of enum* `OptixProgramGroupKind`
Ray generation	ray-generation	`OPTIX_PROGRAM_GROUP_KIND_RAYGEN`
Exception	exception	`OPTIX_PROGRAM_GROUP_KIND_EXCEPTION`
Miss	miss	`OPTIX_PROGRAM_GROUP_KIND_MISS`
Hit	intersection, any-hit, closest-hit	`OPTIX_PROGRAM_GROUP_KIND_HITGROUP`
Callable	direct-callable, continuation-callable	`OPTIX_PROGRAM_GROUP_KIND_CALLABLES`

OptiX program groups

See also "Program group creation" (page 43).

Pointers to the SBT sections are passed to the NVIDIA OptiX 7 launch:

Listing 7.2

```
typedef struct OptixShaderBindingTable
{
    CUdeviceptr raygenRecord = 0;

    CUdeviceptr  exceptionRecord = 0;

    CUdeviceptr  missRecordBase = 0;
    unsigned int missRecordStrideInBytes;
    unsigned int missRecordCount;

    CUdeviceptr  hitgroupRecordBase = 0;
    unsigned int hitgroupRecordStrideInBytes;
    unsigned int hitgroupRecordCount;

    CUdeviceptr  callablesRecordBase = 0;
    unsigned int callablesRecordStrideInBytes;
    unsigned int callablesRecordCount;
} OptixShaderBindingTable;
```

Device address of the SBT record of the ray generation program to start launch

Device address of the SBT record of the exception shader

Arrays of SBT records. The base address, stride in bytes and maximum index are defined.

All SBT records on the device are expected to have a minimum memory alignment, defined by `OPTIX_SBT_RECORD_ALIGNMENT` (currently 16 bytes). Therefore, the stride between records must also be a multiple of `OPTIX_SBT_RECORD_ALIGNMENT`. Each section of the SBT is an independent memory range and is not required to be allocated contiguously.

The selection of s SBT record depends on the program type and uses the corresponding base pointer. Since there can only be a single call to both ray-generation and exception programs, a stride is not required for these two program group types and the passed-in pointer is expected to point to the desired SBT records.

For other types, the SBT record at index *sbt-index* for a program group of type *group-type* is located by the following formula:

group-type`RecordBase` + *sbt-index* * *group-type*`RecordStrideInBytes`

For example, the third record (index 2) of the miss group would be:

```
missRecordBase + 2 * missRecordStrideInBytes
```

The index to records in the shader binding table is used in different ways for the miss, hit, and callables groups:

Miss
: Miss programs are selected for every `optixTrace` call using the `missSBTIndex` parameter.

Callables
: Callables take the index as a parameter and call the direct-callable when invoking `optixDirectCall` and continuation-callable when invoking `optixContinuationCall`.

Any-hit, closest-hit, intersection
: The computation of the index for the hit group (intersection, any-hit, closest-hit) is done during traversal. See "Acceleration structures" (page 53) for more detail.

7.3 Acceleration structures

The selection of the SBT hit group record for the instance is slightly more involved to allow for a number of use cases such as the implementation of different ray types. The SBT record index `sbtIndex` is determined by the following index calculation during traversal:

sbt-index =
    sbt-instance-offset
    + (*sbt-geometry-acceleration-structure-index* * *sbt-stride-from-trace-call*)
    + *sbt-offset-from-trace-call*

The index calculation depends upon the following SBT indices and offsets:

- Instance offset
- Geometry acceleration structure index
- Trace offset
- Trace stride

7.3.1 SBT instance offset

Instance acceleration structure instances (type `OptixInstance`) store an SBT offset that is applied during traversal. This is zero for single geometry-AS traversable because there is no corresponding instance-AS to hold the value. (See "Traversal of a single geometry acceleration structure" (page 31).) This value is limited to 24 bits (see the declaration of `OptixInstance::sbtOffset`).

7.3.2 SBT geometry-AS index

Each geometry acceleration structure build input references at least one SBT record. The first SBT geometry acceleration structure index for each geometry acceleration structure build

input is the prefix sum of the number of SBT records. Therefore, the computed SBT geometry acceleration structure index is dependent on the order of the build inputs.

The following example demonstrates a geometry acceleration structure with three build inputs. Each build input references one SBT record by specifying `numSBTRecords=1`. When intersecting geometry at trace time, the SBT geometry acceleration structure index used to compute the `sbtIndex` to select the hit group record will be is organized follows:

SBT geometry-AS index	*0*	*1*	*2*
Geometry-AS build input	Build input[0]		
		Build input[1]	
			Build input[2]

In this simple example, the index for the build input equals the SBT geometry acceleration structure index. Hence, whenever a primitive from "Build input [1]" is intersected, the SBT geometry acceleration structure index is one.

When a single build input references multiple SBT records (for example, to support multiple materials per geometry), the mapping corresponds to the prefix sum over the number of referenced SBT records.

For example, consider three build inputs where the first build input references four SBT records, the second references one SBT record, and the last references two SBT records:

SBT geometry-AS index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
Geometry-AS build input	Build input[0] `numSBTRecords=4`						
					Build input[1] `numSBTRecords=1`		
						Build input[2] `SBTIndexOffset2`	

These three build inputs result in the following possible SBT geometry acceleration structure indices when intersecting the corresponding geometry acceleration structure build input:

- One index in the range of [0,3] if a primitive from "Build input [0]" is intersected
- Four if a primitive from "Build input [1]" is intersected
- One index in the range of [5,6] if a primitive from "Build input [2]" is intersected

The per-primitive SBT index offsets, as specified by using `sbtIndexOffsetBuffer`, are local to the build input. Hence, per-primitive offsets in the range [0,3] for the build input 0 and in the range [0,1] for the last build input, map to the SBT geometry acceleration structure index as follows:

SBT geometry-AS index	0	1	2	3	4	5	6
Build input[0] `SBTIndexOffset:`	[0]						
		[1]					
			[2]				
				[3]			
Build input[1] `SBTIndexOffset=nullptr`							
Build input[2] `SBTIndexOffset:`						[0]	
							[1]

Because build input 1 references a single SBT record, a `sbtIndexOffsetBuffer` does not need to be specified for the geometry acceleration structure build. See "Acceleration structures" (page 19).

7.3.3 SBT trace offset

The `optixTrace` function takes the parameter `SBToffset`, allowing for an SBT access shift for this specific ray. It is required to implement different ray types.

7.3.4 SBT trace stride

The parameter `SBTstride`, defined as an index offset, is multiplied by `optixTrace` with the SBT geometry acceleration structure index. It is required to implement different ray types.

7.3.5 Example SBT for a scene

In this example, a shader binding table implements the program selection for a simple scene containing one instance acceleration structure and two instances of the same geometry acceleration structure, where the geometry acceleration structure has two build inputs:

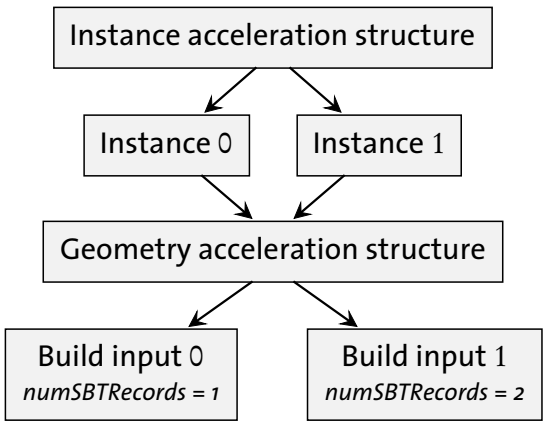

Fig. 7.1 – Structure of a simple scene

The first build input references a single SBT record, while the second one references two SBT records. There are two ray types: one for forward path tracing and one for shadow rays (next

event estimation). The two instances of the geometry acceleration structure have different transforms and SBT offsets to allow for material variation in each instance of the same geometry acceleration structure. Therefore, the SBT needs to hold two miss records and 12 hit group records (three for the geometry acceleration structure, $\times 2$ for the ray types, $\times 2$ for the two instances in the instance acceleration structure).

The SBT is structured in the following way:

Raygen	Miss		`instance0.sbtOffset = 0`						`instance1.sbtOffset = 6`					
RG0	MS0	MS1	hit0	hit1	hit2	hit3	hit4	hit5	hit6	hit7	hit8	hit9	hit10	hit11
The preceding programs are called for the following combination of geometry and ray types														
Instance:			0	0	0	0	0	0	1	1	1	1	1	1
Build input:			0	0	1	1	1	1	0	0	1	1	1	1
SBT index:			0	1	2	3	4	5	6	7	8	9	10	11
SBT instance offset:			0	0	0	0	0	0	6	6	6	6	6	6
SBT geometry-AS index:			0	0	1	1	2	2	0	0	1	1	2	2
Build input SBT index offset:			-	-	0	0	1	1	-	-	0	0	1	1
Trace offset/ray type:	0	1	0	1	0	1	0	1	0	1	0	1	0	1

To trace a ray of type 0 (for example, for path tracing):

Listing 7.3

```
optixTrace(IAS_handle,
   ray_org, ray_dir,
   tmin, tmax, time,
   visMask, rayFlags,
   0,  sbtOffset

   2,  sbtStride

   0,  missSBTIndex
   rayPayload0, ...);
```

Shadow rays need to pass in an adjusted `sbtOffset` as well as `missSBTIndex`:

Listing 7.4

```
optixTrace(IAS_handle,
   ray_org, ray_dir,
   tmin, tmax, time,
   visMask, rayFlags,
   1,  sbtOffset

   2,  sbtStride
```



```
    1,   missSBTIndex
    rayPayload0, ...);
```

Program groups of different types (ray generation, miss, intersection, and so on) do not need to be adjacent to each other as shown in the example. The pointer to the first SBT record of each program group type is passed to `optixLaunch`, as described previously, which allows for arbitrary spacing in the SBT between the records of different program group types.

7.4 SBT record access on device

To access the SBT data section of the currently running program, request its pointer by using an API function:

Listing 7.5

```
CUdeviceptr optixGetSbtDataPointer();
```

Typically, this pointer is cast to a pointer that represents the layout of the data section. For example, for a closest hit program, the application gets access to the data associated with the SBT record that was used to invoke that closest hit program:

Listing 7.6 – Data for closest hit program

```
struct CHData {
    int meshIdx;   Triangle mesh build input index
   float3 base_color;
};

CHData* material_info = (CHData*)optixGetSbtDataPointer();
```

The program is encouraged to rely on the alignment constraints of the SBT data section to read this data efficiently.

8 Curves

8.1 Differences between curves and triangles

Ray tracing curves with NVIDIA OptiX 7 is similar to the procedure for ray tracing triangles; see "Curve build inputs" (page 24) . The differences between curves and triangles include the following:

- In the triangle build input, the index buffer is optional. In the curves build input, it is mandatory.
- Each curves build input references just a single SBT record. Unlike triangles, there is no per-primitive SBT index. It is still possible to use multiple materials for curves in the same BVH, by using multiple build inputs, one per material. (Because there is only one SBT record, the `OptixGeometryFlags` are specified by only one int, rather than an array of ints.)
- There is no `preTransform` field for curves. If there were, a nonuniform scale or shear transformation would yield different results as a pre-transform than as an instance transform. Nonuniform instance transforms create elliptical cross sections, while `preTransform` would still have a circular cross section.
- Each shader binding table record for curves requires a hit group that uses a built-in intersection program for curves. In the `OptixProgramGroupHitgroup`, you must use a module returned by `optixBuiltinISModuleGet()` for `moduleIS`, and `nullptr` for `entryFunctionNameIS`.
- Curves must be explicitly enabled in the pipeline to be rendered; triangles and custom primitives are enabled by default. This is enabled in the `OptixPipelineCompileOptions::usesPrimitiveTypeFlags` by setting the relevant bits from `OptixPrimitiveTypeFlags`.

Listing 8.1

```
OptixPipelineCompileOptions pipelineCompileOptions = {};
...
pipelineCompileOptions.usesPrimitiveTypeFlags =
    OPTIX_PRIMITIVE_TYPE_FLAGS_TRIANGLE |
    OPTIX_PRIMITIVE_TYPE_FLAGS_CUSTOM |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_LINEAR |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_QUADRATIC_BSPLINE |
    OPTIX_PRIMITIVE_TYPE_FLAGS_ROUND_CUBIC_BSPLINE;
```

- To render motion blur for any primitive, motion blur must be enabled in the pipeline. But for curves, motion blur must also be enabled in the built-in intersection program, by setting the `OptixBuiltinISOptions::usesMotionBlur` flag.

8.2 Splitting curve segments

NVIDIA OptiX 7 can split curve segments into multiple sub-segments, and bound the sub-segments separately. This gives faster performance but costs more memory. Splitting can be controlled via `OptixBuildFlags`, using `OPTIX_BUILD_FLAG_PREFER_FAST_TRACE` for a splitting factor higher than default, and `OPTIX_BUILD_FLAG_PREFER_FAST_BUILD` for a lower splitting factor.

Note: Splitting means that the same primitive (the same curve segment) can be hit multiple times by a ray. Geometry with `OPTIX_GEOMETRY_FLAG_REQUIRE_SINGLE_ANYHIT_CALL` set is not split; in this case, each segment can only be hit once, but a longer curve strand composed of multiple segments can still be hit more than once.

8.3 Curves and the hit program

In hit programs, curve hits provide a single attribute: the curve parameter ("u") within the segment corresponding to the intersection, returned by `optixGetCurveParameter`(). When and only when an end cap is hit, the "u" parameter returned is exactly 0.0 or 1.0. As with all hit programs, `optixGetRayTmax`() returns the ray parameter ("t"), from which the hit point can be computed, and `optixGetPrimitiveIndex`() returns the segment's primitive index. To maximize performance, no other geometry attributes are passed or precomputed. Instead, the program must compute whatever curve geometry it requires (surface normals, etc.) using the vertex data.

To fetch the vertex positions and radius values for the segment, pass the primitive index to the function appropriate for the curve type:

```
optixGetLinearCurveVertexData
optixGetQuadraticBSplineVertexData
optixGetCubicBSplineVertexData
```

Sample code is provided to interpolate the curve points and radii, and to compute tangents, normals, and derivatives; see header file `SDK/cuda/curve.h`. A single hit program can (and should) handle multiple curve primitive types by checking the value of `optixGetPrimitiveType`(). For example, see `optixHair`.cu in the `optixHair` code sample.

The `optixGetCurveParameter` function returns the parameter value relative to a single polynomial segment of the curve. If needed, the hit program can map this to the parameter value relative to the entire multi-segment strand. For instance, this can be used to interpolate between two colors specified at the root and tip of a hair. This segment-to-strand mapping is the application's responsibility; see the `optixHair` code sample for an example implementation.

8.4 Interpolating curve endpoints

A B-spline curve typically does not interpolate (that is, does not touch) its control points. In particular it does not reach as far as its first and last control points:

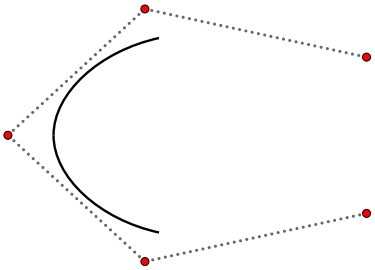

Fig. 8.1 – The curve does not reach the first and last control points

If desired, the application can modify the control point sequence to interpolate these points, by adding an additional control point at each end:

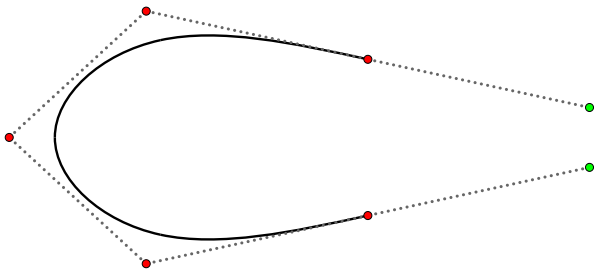

Fig. 8.2 – Adding points (in green) to extend the curve

Following the Universal Scene Description documentation,[12] let's call these "phantom" points, resulting in "pinned" curves. A phantom point is constructed by reflecting through the endpoint the point that preceded it. Given control points $p_1, p_2 ... p_n$, the phantom points can be defined geometrically by:

$$p_0 = p_1 + (p_1 - p_2)$$

$$p_{n+1} = p_n + (p_n - p_{n-1})$$

It is also possible to interpolate endpoints by repeating them. For cubic B-spline curves, a control point that is repeated 3 times will be interpolated. OptiX allows this, but the phantom point method is preferred, as it is numerically more stable, and permits control of curvature.

Note: The additional points must be added to the build input by the application. These points are not added by NVIDIA OptiX 7 itself.

8.5 Limitations

There are some caveats to be aware of when using curves and curve hit programs. We hope to ease some of these in future versions.

Internal rays

In the current implementation, if a ray starts inside a curve primitive (inside the tube), results are undefined, as tubes are considered solid. Therefore, secondary rays from curve hits should be launched from outside the radius of the curve.

12. `https://graphics.pixar.com/usd/docs/api/class_usd_geom_basis_curves.html#UsdGeomBasisCurves_Segment`

Multiple hits

For a ray that intersects a single curve segment more than once, NVIDIA OptiX 7 does not guarantee that all intersections will be reported as hits. In particular, the any-hit program will in general not be called more than once for a given segment. To implement transparency, the alternative approach to using an any-hit program is to use closest-hit and relaunch a continuation ray from each hit. However, because of the current limitation on internal rays, a continuation ray must be moved to the other side of the curve. In that case, overlapping curves may be missed.

Triple control points

As noted in "Interpolating curve endpoints" (page 60) , duplicating control points is possible in NVIDIA OptiX 7, but the use of phantom points is preferred. Duplicating the ending control points is numerically challenging (the derivative is zero) and the first segment is forced to be straight.

Convoluted cases

Curves for hair, fur, cloth fibers, etc., are typically thin with relatively gentle curvature. It is possible to construct curve segments with tight curvature (relative to width), self-intersections, or rapidly varying radii that will exhibit artifacts, particularly if rendered up close. In many cases, these can be remedied by splitting the segment into 3 smaller segments.

9 Ray generation launches

The API function described in this section is:

`optixLaunch`

A ray generation launch is the primary workhorse of the NVIDIA OptiX API. A launch invokes a 1D, 2D or 3D array of threads on the device and invokes ray generation programs for each thread. When the ray generation program invokes `optixTrace`, other programs are invoked to execute traversal, intersection, any-hit, closest-hit, miss and exception programs until the invocations are complete.

A pipeline requires device-side memory for each launch. This space is allocated and managed by the API. Because launch resources may be shared between pipelines, they are only guaranteed to be freed when the `OptixDeviceContext` is destroyed.

To initiate a pipeline launch, use the `optixLaunch` function. All launches are asynchronous, using CUDA streams. When it is necessary to implement synchronization, use the mechanisms provided by CUDA streams and events.

In addition to the pipeline object, the CUDA stream, and the launch state, it is necessary to provide information about the SBT layout, including: This includes:

- The base addresses for sections of the SBT that hold the records of different types
- The stride, in bytes, along with the maximum valid index for arrays of SBT records. The stride is used to calculate the SBT address for a record based on a given index. (See "Layout" (page 51).)

The value of the pipeline launch parameter is specified by the `pipelineLaunchParamsVariableName` field of the `OptixPipelineCompileOptions` struct. It is determined at launch with a `CUdeviceptr` parameter, named `pipelineParams`, that is provided to `optixLaunch`. Note the following restrictions:

- If the size specified by the `pipelineParamsSize` argument of `optixLaunch` is smaller than the size of the variable specified by the modules, the non-overlapping values of the parameter will be undefined.
- If the size is larger, an error will occur.

(See "Pipeline launch parameter" (page 40).)

The kernel creates a copy of `pipelineParams` before the launch, so the kernel is allowed to modify `pipelineParams` values during the launch. This means that subsequent launches can run with modified pipeline parameter values. Users cannot synchronize with this copy between the invocation of `optixLaunch` and the start of the kernel.

Note: Concurrent launches with different values for `pipelineParams` in the same pipeline triggers serialization of the launches. Concurrency requires a separate pipeline for each concurrent launch.

The dimensions of a launch must also be specified. If one-dimensional launches are required, use the width as the dimension of the launch and set both a height and a depth of 1. If two-dimensional launches are required, set the width and the height as the dimension of the launch and set a depth of 1.

Dimension	*Width*	*Height*	*Depth*
1D	*width*	1	1
2D	*width*	*height*	1
3D	*width*	*height*	*depth*

Specifying different dimensions for a launch

For example:

Listing 9.1

```
CUstream stream = nullptr;
cuStreamCreate(&stream);
CUdeviceptr raygenRecord, hitgroupRecords;

...          Generate acceleration structures and SBT records

unsigned int width = ...;
unsigned int height = ...;
unsigned int depth = ...;
OptixShaderBindingTable sbt = {};
sbt.raygenRecord = raygenRecord;
sbt.hitgroupRecords = hitgroupRecords;
sbt.hitgroupRecordStrideInBytes = sizeof(HitGroupRecord);
sbt.hitgroupRecordCount = numHitGroupRecords;
MyPipelineParams pipelineParams = ...;
CUdeviceptr d_pipelineParams = 0;

...          Allocate and copy the params to the device

optixLaunch(pipeline, stream,
   d_pipelineParams, sizeof(MyPipelineParams),
   &sbt, width, height, depth);
```


10 Limits

The previous chapters described properties that have an upper limit lower than the limit implied by the data type of the property. The values of these limits depend on the GPU generation for which the device context is created. These values can be queried at runtime using `optixDeviceContextGetProperty`.

Limit values may change from one OptiX SDK version to the next but not internally with updated NVIDIA drivers. Updated drivers use the limit values of the SDK version employed during application compilation as well as the GPU generation. The following table lists the NVIDIA OptiX 7 limit values for the currently supported GPU generations, including Turing:

Type of limit	*Limit*	*non-RTX cards*	*RTX-enabled Turing*
Acceleration structure	Maximum number of primitives per geometry acceleration structure including motion keys	2^{29}	2^{29}
	Maximum number of referenced SBT records per geometry acceleration structure	2^{24}*	2^{24}*
	Maximum number of instances per instance acceleration structure	2^{28}*	2^{28}*
	Number of bits for SBT offset	28*	28*
	Number of bits for user ID	28*	28*
	Number of bits for visibility mask	8	8
Pipeline	Maximum trace (recursion) depth	31	31
	Maximum traversable graph depth	31	31
Device functions	Number of bits for `optixTrace` – `visibilityMask`	8	8
	Number of bits for `optixTrace` – `SBTstride`	4	4
	Number of bits for `optixTrace` – `SBToffset`	4	4
	Number of bits for `optixReportIntersection` – `hitKind`	7	7
Hardware version	RT Cores version (read as x.x)	00	10

NVIDIA OptiX 7 limits

(*)Values marked with * were raised beginning in NVIDIA OptiX version 7.1. For code compiled with OptiX SDK version 7.0, the limit is 24.

For the instance properties SBT offset, user ID, and visibility mask, the higher bits of the 32-bit struct member must be set to zero. In case of the device functions, any bits higher than those specified in the table are ignored. Note that limits for device functions cannot be queried at runtime.

11 Device-side functions

The NVIDIA OptiX device runtime provides functions to set and get the ray tracing state and to trace new rays from within user programs. The following functions are available in all program types:[13]

```
optixGetTransformTypeFromHandle
optixGetInstanceIdFromHandle
optixGetInstanceTransformFromHandle
optixGetInstanceInverseTransformFromHandle
optixGetStaticTransformFromHandle
optixGetMatrixMotionTransformFromHandle
optixGetSRTMotionTransformFromHandle
optixGetGASMotionTimeBegin
optixGetGASMotionTimeEnd
optixGetGASMotionStepCount
optixGetPrimitiveType( hitKind )
optixIsFrontFaceHit( hitKind )
optixIsBackFaceHit( hitKind )
optixGetTriangleVertexData
optixGetLinearCurveVertexData
optixGetQuadraticBSplineVertexData
optixGetCubicBSplineVertexData
optixGetLaunchIndex
optixGetLaunchDimensions
optixGetSbtDataPointer
```

Other functions are available only in specific program types. The following table identifies the program types within which the OptiX device functions are valid.

13. The abbreviation “GAS” is used for geometry acceleration structures in function names.

Device function	ray generation	intersection	any hit	closest hit	miss	exception	direct callable	continuation callable
`optixTrace` `optixContinuationCall`	✓			✓	✓			✓

`optixSetPayload_0` ... `optixSetPayload_7`
`optixGetPayload_0` ... `optixGetPayload_7`
`optixGetWorldRayOrigin`
`optixGetWorldRayDirection`
`optixGetRayTmin`
`optixGetRayTmax`
`optixGetRayTime`
`optixGetRayFlags`
`optixGetRayVisibilityMask`
		✓	✓	✓	✓			

`optixGetObjectRayOrigin` `optixGetObjectRayDirection`		✓	✓					

`optixGetTransformListSize`
`optixGetTransformListHandle`
`optixGetSbtGASIndex`
		✓	✓	✓		✓		

`optixGetWorldToObjectTransformMatrix`
`optixGetObjectToWorldTransformMatrix`
`optixTransformPointFromWorldToObjectSpace`
`optixTransformPointFromObjectToWorldSpace`
`optixTransformVectorFromWorldToObjectSpace`
`optixTransformVectorFromObjectToWorldSpace`
`optixTransformNormalFromWorldToObjectSpace`
`optixTransformNormalFromObjectToWorldSpace`
`optixGetGASTraversableHandle`
`optixGetPrimitiveIndex`
`optixGetInstanceId`
`optixGetInstanceIndex`
`optixGetAttribute_0` ... `optixGetAttribute_7`
		✓	✓	✓				

`optixGetHitKind`
`optixGetPrimitiveType`
`optixIsFrontFaceHit`
`optixIsBackFaceHit`
`optixIsTriangleHit` (header function)
`optixIsTriangleFrontFaceHit` (header function)
`optixIsTriangleBackFaceHit` (header function)
`optixGetTriangleBarycentrics`
`optixGetCurveParameter`
			✓	✓				

`optixReportIntersection`		✓						
`optixTerminateRay` `optixIgnoreIntersection`			✓					
`optixDirectCall` `optixThrowException`	✓	✓	✓	✓	✓		✓	✓

`optixGetExceptionCode`
`optixGetExceptionInvalidTraversable`
`optixGetExceptionInvalidSbtOffset`
`optixGetExceptionInvalidRay`
`optixGetExceptionParameterMismatch`
`optixGetExceptionLineInfo`
`optixGetExceptionDetail_0` ...
`optixGetExceptionDetail_7`
						✓		

Any function in the module that calls an NVIDIA OptiX device-side function is inlined into the caller (with the exceptions noted below). This process is repeated until only the outermost function contains these function calls.

For example, consider a closest-hit program that calls a function called `computeValue`, which calls `computeDeeperValue`, which itself calls `optixGetTriangleBarycentrics`. The inlining process inlines the body of `computeDeeperValue` into `computeValue`, which in turn, is inlined into the closest-hit program. Recursive functions that call device-side API functions will generate a compilation error.

The following functions do not trigger inlining:

```
optixGetTransformTypeFromHandle
optixGetInstanceIdFromHandle
optixGetInstanceTransformFromHandle
optixGetInstanceInverseTransformFromHandle
optixGetStaticTransformFromHandle
optixGetMatrixMotionTransformFromHandle
optixGetSRTMotionTransformFromHandle
optixGetLaunchDimensions
optixGetGASMotionTimeBegin
optixGetGASMotionTimeEnd
optixGetGASMotionStepCount
optixGetTriangleVertexData
optixGetLinearCurveVertexData
optixGetQuadraticBSplineVertexData
optixGetCubicBSplineVertexData
```

11.1 Launch index

The *launch index* identifies the current thread, within the launch dimensions specified by `optixLaunch` on the host. The launch index is available in all programs.

Listing 11.1

```
uint3 optixGetLaunchIndex();
```

Typically, the ray generation program is only launched once per launch index.

In contrast to the CUDA programming model, program execution of neighboring launch indices is not necessarily done within the same warp or block, so the application must not rely on the locality of launch indices.

11.2 Trace

The `optixTrace` function initiates a ray tracing query starting with the given traversable and the provided ray origin and direction. If the given `OptixTraversableHandle` is null, only the miss program is invoked.

An arbitrary payload is associated with each ray that is initialized with this call; the payload is passed to all the intersection, any-hit, closest-hit and miss programs that are executed

during this invocation of trace. The payload can be read and written by each program using the eight pairs of `optixGetPayload` and `optixSetPayload` functions (for example, `optixGetPayload_0` and `optixSetPayload_0`). The payload is subsequently passed back to the caller of `optixTrace` and follows a copy-in/copy-out semantic. Payloads are limited in size and are encoded in a maximum of eight 32-bit integer values, which are held in registers where possible. To hold additional state, these values may also encode pointers to stack-based variables or application-managed global memory.

The rayTime argument sets the time allocated for motion-aware traversal and material evaluation. If motion is not enabled in the pipeline compile options, the ray time is ignored and removed by the compiler. To request the ray time, use the `optixGetRayTime` function. In a pipeline without motion, `optixGetRayTime` always returns 0.

The `rayFlags` argument can represent a combination of `OptixRayFlags`. The following flags are supported. Illegal combinations are noted.

`OPTIX_RAY_FLAG_NONE`
: No change from the behavior configured for the individual acceleration structure.

`OPTIX_RAY_FLAG_DISABLE_ANYHIT`
: Disables any-hit programs for the ray. Overrides potential instance flag `OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT` when intersecting instances. This flag is mutually exclusive with `OPTIX_RAY_FLAG_ENFORCE_ANYHIT`, `OPTIX_RAY_FLAG_CULL_DISABLED_ANYHIT`, `OPTIX_RAY_FLAG_CULL_ENFORCED_ANYHIT`.

`OPTIX_RAY_FLAG_ENFORCE_ANYHIT`
: Forces any-hit program execution for the ray. Overrides `OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT` and `OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT`.

`OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT`
: Terminates the ray after the first hit and executes the closest-hit program of that hit.

`OPTIX_RAY_FLAG_DISABLE_CLOSESTHIT`
: Disables closest-hit programs for the ray, but still executes the miss program in case of a miss.

`OPTIX_RAY_FLAG_CULL_BACK_FACING_TRIANGLES`
: Prevents intersection of triangle back faces (respects a possible face change due to instance flag `OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING`). This flag is mutually exclusive with `OPTIX_RAY_FLAG_CULL_FRONT_FACING_TRIANGLES`.

`OPTIX_RAY_FLAG_CULL_FRONT_FACING_TRIANGLES`
: Prevents intersection of triangle front faces (respects a possible face change due to instance flag `OPTIX_INSTANCE_FLAG_FLIP_TRIANGLE_FACING`). This flag is mutually exclusive with `OPTIX_RAY_FLAG_CULL_BACK_FACING_TRIANGLES`.

`OPTIX_RAY_FLAG_CULL_DISABLED_ANYHIT`
: Prevents intersection of geometry which disables any-hit programs (due to setting geometry flag `OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT` or instance flag `OPTIX_INSTANCE_FLAG_DISABLE_ANYHIT`). This flag is mutually exclusive with `OPTIX_RAY_FLAG_CULL_ENFORCED_ANYHIT`, `OPTIX_RAY_FLAG_ENFORCE_ANYHIT`, `OPTIX_RAY_FLAG_DISABLE_ANYHIT`.

`OPTIX_RAY_FLAG_CULL_ENFORCED_ANYHIT`
: Prevents intersection of geometry which have an enabled any-hit program (due to not setting geometry flag `OPTIX_GEOMETRY_FLAG_DISABLE_ANYHIT` or setting instance flag

OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT). This flag is mutually exclusive with OPTIX_RAY_FLAG_CULL_DISABLED_ANYHIT, OPTIX_RAY_FLAG_ENFORCE_ANYHIT, OPTIX_RAY_FLAG_DISABLE_ANYHIT.

Ray flags modify traversal behavior. For example, setting ray flag OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT causes the very first hit (that is not ignored in any-hit) to abort further traversal, defining it as the closest hit. This particular flag can be useful for shadow rays to allow for the early termination of a traversal without the need for a special any-hit program that calls optixTerminateRay.

The visibility mask controls intersection against configurable masks of instances. (See "Instance build inputs" (page 24).) Intersections are computed if there is at least one matching bit in both masks. The same limit applies in the number of available bits as for the instance visibility mask. See "Limits" (page 65).

The SBT offset and stride adjust the SBT indexing when selecting the SBT record for a ray intersection. Like the visibility mask, both parameters are limited. See "Limits" (page 65) and "Acceleration structures" (page 53).

The specified miss SBT index is used to identify the program that is invoked on a miss. (See "Layout" (page 51).) The argument must be a valid index for a SBT record for a miss program.

Listing 11.2

```
__device__ void optixTrace(OptixTraversableHandle handle,
    float3 rayOrigin,
    float3 rayDirection,
    float  tmin,
    float  tmax,
    float  rayTime,
    OptixVisibilityMask visibilityMask,
    unsigned int rayFlags,
    unsigned int SBToffset,
    unsigned int SBTstride,
    unsigned int missSBTIndex,
    unsigned int& p0,
    ...
    unsigned int& p7);
```

11.3 Payload access

In intersection, any-hit, closest-hit, and miss programs, the payload is used to communicate values from the optixTrace that initiates the traversal, to and from other programs in the traversal, and back to the caller of optixTrace. There are up to eight 32-bit payload values available. Getting and setting the eight payload values use functions whose names end with a payload index. For example, payload 0 is set and accessed by these two functions:

Listing 11.3

```
__device__ void optixSetPayload_0(unsigned int p);
__device__ unsigned int optixGetPayload_0();
```

Setting a payload value using `optixSetPayload` causes the updated value to be visible in any subsequent `optixGetPayload` calls until the return to the caller of `optixTrace`. Payload values that are not explicitly set in a program remain unmodified. Payload values can be set anywhere in a program. Values accessible using `optixSetPayload` and `optixGetPayload` are associated only with the payload passed into the most recent call to `optixTrace`.

In some use cases, register consumption can be reduced by writing `optixUndefinedValue`() to payload values that are no longer used. This advanced application feature enables the compiler to improve runtime performance by improving register usage.

Although the type of the payload is exclusively integer data, it is expected that users will wrap one or more of these data types into more readable data structures using `__int_as_float` and `__float_as_int`, or other data types where necessary.

11.4 Reporting intersections and attribute access

To report an intersection with the current traversable, the intersection program can use the `optixReportIntersection` function. The hitKind of the given intersection is communicated to the associated any-hit and closest-hit program and allows the any-hit and closest-hit programs to customize how the attributes should be interpreted. The lowest 7 bits of the hitKind are interpreted; values [128, 255] are reserved for internal use.

Up to eight 32-bit primitive attribute values are available. Intersection programs write the attributes when reporting an intersection using `optixReportIntersection`. Then closest-hit and any-hit programs are able to read these attributes. For example:

Listing 11.4

```
__device__ bool optixReportIntersection(
   float hitT,
   unsigned int hitKind,
   unsigned int a0, ... , unsigned int a7);

__device__ unsigned int optixGetAttribute_0();
```

To reject a reported intersection in an any-hit program, an application calls `optixIgnoreIntersection`. The closest-hit program is called for the closest accepted intersection with the attributes reported for that intersection. An any-hit program may not be called for all possible hits along the ray. When an intersection is accepted (not discarded by `optixIgnoreIntersection`), the interval for intersection and traversal is updated. Further intersections outside the new interval are not performed.

Although the type of the attributes is exclusively integer data, it is expected that users will wrap one or more of these data types into more readable data structures using `__int_as_float` and `__float_as_int`, or other data types where necessary.

Triangle intersections return two attributes, the barycentric coordinates (u,v) of the hit, which may be read with the convenience function `optixGetTriangleBarycentrics`. Curve

primitive intersections return one attribute, the curve parameter (u) within the polynomial curve segment, which may be read with the convenience function `optixGetCurveParameter`.

No more than eight values can be used for attributes. Unlike the ray payload that can contain pointers to local memory, attributes should not contain pointers to local memory. This memory may not be available in the closest-hit or intersection programs when the attributes are consumed. More sophisticated attributes are probably better handled in the closest-hit program. There are generally better memory bandwidth savings by deferring certain calculations to the closest-hit program or reloading values once in the closest-hit program.

11.5 Ray information

To query the properties of the currently active ray, use the following functions:

`optixGetWorldRayOrigin / optixGetWorldRayDirection`
: Returns the ray's origin and direction passed into `optixTrace`. It may be more expensive to call these functions during traversal (that is, in intersection or any-hit) than their object space counterparts.

`optixGetObjectRayOrigin / optixGetObjectRayDirection`
: Returns the object space ray direction or origin based based on the current transformation stack. These functions are only available in intersection and any-hit programs.

`optixGetRayTmin`
: Returns the minimum extent associated with the current ray. This is the tmin value passed into `optixTrace`.

`optixGetRayTmax`
: Returns the maximum extent associated with the current ray. Note the following:

  - In intersection and closest-hit programs, this is the smallest reported hitT or if no intersection has been recorded yet the tmax that was passed into `optixTrace`.
  - In any-hit programs, this returns the hitT value as passed into `optixReportIntersection`.
  - In miss programs, the return value is the tmax that was passed into `optixTrace`.

`optixGetRayTime`
: Returns the time value passed into `optixTrace`. Returns 0 if motion is disabled in the pipeline.

`optixGetRayFlags`
: Returns the ray flags passed into `optixTrace`.

`optixGetRayVisibilityMask`
: Returns the visibility mask passed into `optixTrace`.

Note: In ray-generation and exception programs, these functions are not supported because there is no currently active ray.

11.6 Undefined values

Advanced application writers seeking more fine-grained control over register usage may want to reduce total register use in heavy intersect and any-hit programs by writing an

unknown value to payload slots not used during traversal. NVIDIA OptiX provides the following function to make this straightforward:

Listing 11.5

```
__device__ unsigned int optixUndefinedValue();
```

11.7 Intersection information

The primitive index of the current intersection point can be queried using `optixGetPrimitiveIndex`. The primitive index is local to its build input.

The SBT index of the current intersection point can be queried using `optixGetSbtGASIndex`. The SBT index is local to its build input. (See "Shader binding table" (page 51).)

The application can query the 8-bit hit kind by using `optixGetHitKind`. The hit kind is analyzed by calling `optixGetPrimitiveType`, which tells whether a custom primitive, built-in triangle, or built-in curve primitive was hit, as well as whether the curve was linear, quadratic or cubic. For built-in primitives, `optixIsFrontFaceHit` and `optixIsBackFaceHit` tell whether the ray hit a front or back face of the primitive. For custom primitives, the hit kind is the value reported by `optixReportIntersection` when it was called in the intersection.

Note: It is generally more efficient to have one hit shader handle multiple primitive types (by switching on the value of `optixGetPrimitiveType`), rather than have several hit shaders that implement the same ray behavior but differ only in the type of geometry they expect.

For triangle hits, there are several notational shortcuts. The hit kind is either `OPTIX_HIT_KIND_TRIANGLE_FRONT_FACE` or `OPTIX_HIT_KIND_TRIANGLE_BACK_FACE`, depending on whether the ray came from the front or back of the triangle.

The device functions `optixIsTriangleHit`, `optixIsTriangleFrontFaceHit`, and `optixIsTriangleBackFaceHit` may also be used.

The functions `optixGetPrimitiveType`, `optixIsFrontFaceHit`, and `optixIsBackFaceHit` each have two forms. Without an argument, they analyze the current ray intersection; this form can only be used in a closest-hit or any-hit program. Or, an explicit hit kind argument may be used; this form can be called from any type of program.

When traversing a scene with instances, that is, a scene containing instance acceleration structure objects, two properties of the most recently visited instance can be queried in intersection and any-hit programs. In the case of closest-hit programs, the properties reference the instance most recently visited when the hit was recorded with `optixReportIntersection`. Using `optixGetInstanceId` the value supplied to the `OptixInstance::instanceId` can be retrieved. Using `optixGetInstanceIndex` the zero-based index within the instance acceleration structure's instances associated with the instance is returned. If no instance has been visited between the geometry primitive and the target for `optixTrace`, `optixGetInstanceId` returns `~0u` (the bitwise complement of a `unsigned int` zero) and `optixGetInstanceIndex` returns an `unsigned int` zero.

11.8 SBT record data

The data section of the current SBT record can be accessed using `optixGetSbtDataPointer`. It returns a pointer to the data, omitting the header of the SBT record (see "Shader binding table" (page 51)).

11.9 Vertex random access

Triangle vertices are baked into the triangle data structure of the geometry acceleration structure. When a triangle geometry acceleration structure is built with the `OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS` flag set, the application can query in object space the triangle vertex data of any triangle in the geometry acceleration structure. Because the geometry acceleration structure contains the triangle data, the application can safely release its own triangle data buffers on the device, thereby lowering overall memory usage.

The function `optixGetTriangleVertexData` returns the three triangle vertices at the `rayTime` passed in. Motion interpolation is performed if motion is enabled on the geometry acceleration structure and pipeline.

Listing 11.6

```
void optixGetTriangleVertexData(
    OptixTraversableHandle gas,
    unsigned int primIdx,
    unsigned int sbtGasIdx,
    float rayTime,
    float3 data[3]);
```

The user can call functions `optixGetGASTraversableHandle`, `optixGetPrimitiveIndex`, `optixGetSbtGASIndex` and `optixGetRayTime` to obtain the geometry acceleration structure traversable handle, primitive index, geometry acceleration structure local SBT index and motion time associated with an intersection in the closest-hit and any-hit programs. The function `optixGetTriangleVertexData` also performs motion vertex interpolation for triangle position data.

For example:

Listing 11.7

```
OptixTraversableHandle gas = optixGetGASTraversableHandle();
unsigned int primIdx = optixGetPrimitiveIndex();
unsigned int sbtIdx = optixGetSbtGASIndex();
float time = optixGetRayTime();

float3 data[3];
optixGetTriangleVertexData(gas, primIdx, sbtIdx, time, data);
```

NVIDIA OptiX may remove degenerate (unintersectable) triangles from the acceleration structure during construction. Calling `optixGetTriangleVertexData` on a degenerate triangle returns `NaN` as triangle data, not the original triangle vertices.

The potential decompression step of triangle data may come with significant runtime overhead. Enabling random access may cause the geometry acceleration structure to use slightly more memory.

Care has to be taken if `optixGetTriangleVertexData` is used with a primitive index other than the value returned by `optixGetPrimitiveIndex`. `optixGetTriangleVertexData` expects a local primitive index corresponding to the build input / `sbtGASIndex` plus the primitive index offset as specified in the build input at the acceleration structure build.

Curve primitive vertices and radii are also stored in the geometry acceleration structure, and may be retrieved in an analogous way using the function `optixGetLinearCurveVertexData`, `optixGetQuadraticBSplineVertexData`, or `optixGetCubicBSplineVertexData`, depending on the type of curve.

11.10 Geometry acceleration structure motion options

In addition to the motion vertex interpolation performed by `optixGetTriangleVertexData`, interpolation may also be desired for other user-managed vertex data, such as interpolating vertices in a custom motion intersection, or interpolating user-provided shading normals in the closest-hit shader. NVIDIA OptiX 7 provides the following functions to obtain the motion options for a geometry acceleration structure:

```
optixGetGASMotionTimeBegin
optixGetGASMotionTimeEnd
optixGetGASMotionStepCount
```

For example, if the number of motion keys for the user vertex data equals the number of motion keys in the geometry acceleration structure, the user can compute the left key index and intra-key interpolation time as follows:

Listing 11.8

```
OptixTraversableHandle gas = optixGetGASTraversableHandle();

float currentTime = optixGetRayTime();
float timeBegin = optixGetGASMotionTimeBegin(gas);
float timeEnd = optixGetGASMotionTimeEnd(gas);
int numIntervals = optixGetGASMotionStepCount(gas) - 1;

float time =
   (globalt - timeBegin) * numIntervals / (timeEnd - timeBegin);
time = max(0.f, min(numIntervals, time));
float fltKey = floorf(time);

float intraKeyTime = time - fltKey;
int leftKey = (int)fltKey;
```

11.11 Transform list

In a multi-level/IAS scene graph, one or more transformations are applied to each primitive. NVIDIA OptiX provides intrinsics to read a transform list at the current primitive. The transform list contains all transforms on the path through the scene graph from the root

traversable (passed to `optixTrace`) to the current primitive. Function `optixGetTransformListSize` returns the number of entries in the transform list and `optixGetTransformListHandle` returns the traversable handle of the transform entries.

Function `optixGetTransformTypeFromHandle` returns the type of a traversable handle and can be of one of the following types:

`OPTIX_TRANSFORM_TYPE_INSTANCE`
: An instance in an instance acceleration structure. Function `optixGetInstanceIdFromHandle` returns the instance user ID. Functions `optixGetInstanceTransformFromHandle` and `optixGetInstanceInverseTransformFromHandle` return the instance transform and its inverse.

`OPTIX_TRANSFORM_TYPE_STATIC_TRANSFORM`
: A transform corresponding to the `OptixStaticTransform` traversable. Function `optixGetStaticTransformFromHandle` returns a pointer to the traversable.

`OPTIX_TRANSFORM_TYPE_MATRIX_MOTION_TRANSFORM`
: A transform corresponding to the `OptixMatrixMotionTransform` traversable. Function `optixGetMatrixMotionTransformFromHandle` returns a pointer to the traversable.

`OPTIX_TRANSFORM_TYPE_SRT_MOTION_TRANSFORM`
: A transform corresponding to the `OptixSRTMotionTransform` traversable. Function `optixGetSRTMotionTransformFromHandle` returns a pointer to the traversable.

Only use these pointers to read data associated with these nodes. Writing data to any traversables that are active during a launch produces undefined results.

For example:

Listing 11.9 – Generic world to object transform computation

```
float4 mtrx[3];
for (unsigned int i = 0; i < optixGetTransformListSize(); ++i) {
   OptixTraversableHandle handle = optixGetTransformListHandle(i);
   float4 trf[3];
   switch(optixGetTransformTypeFromHandle(handle)) {
   case OPTIX_TRANSFORM_TYPE_INSTANCE: {
      const float4* trns =
         optixGetInstanceInverseTransformFromHandle(handle);
      trf[0] = trns[0];
      trf[1] = trns[1];
      trf[2] = trns[2];
   } break;
   case OPTIX_TRANSFORM_TYPE_STATIC_TRANSFORM : {
      const OptixStaticTransform* traversable =
         optixGetStaticTransformFromHandle(handle);
       ...          Compute trf
   } break;
   case OPTIX_TRANSFORM_TYPE_MATRIX_MOTION_TRANSFORM : {
      const OptixMatrixMotionTransform* traversable =
         optixGetMatrixMotionTransformFromHandle(handle);
```

```
        ...          Compute trf
    } break;
    case OPTIX_TRANSFORM_TYPE_SRT_MOTION_TRANSFORM : {
        const OptixSRTMotionTransform* traversable =
            optixGetSRTMotionTransformFromHandle(handle);
        ...          Compute trf
    } break;
    default:
        continue;
    }
    if (i == 0) {
        mtrx[0] = trf[0];
        mtrx[1] = trf[1];
        mtrx[2] = trf[2];
    } else {
        float4 m0 = mtrx[0], m1 = mtrx[1], m2 = mtrx[2];
        mtrx[0] = rowMatrixMul(m0, m1, m2, trf[0]);
        mtrx[1] = rowMatrixMul(m0, m1, m2, trf[1]);
        mtrx[2] = rowMatrixMul(m0, m1, m2, trf[2]);
    }
}
```

Right multiply rows with pre-multiplied matrix

An application can implement a generic transformation evaluation function using these intrinsics. A specialized evaluation function can also be defined that takes advantage of the particular structure of the scene graph, for example, when a scene graph features only one level of instances. The value returned by `optixGetTransformListSize` can be specialized with the `OptixPipelineCompileOptions::traversableGraphFlags` compile option by selecting which subset of traversables need to be supported. For example, if only one level of instancing is necessary and no motion blur transforms need to be supported, set traversableGraphFlags to `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_LEVEL_INSTANCING`.

Handles passed back from `OptixTraversableHandle` can be stored or passed to other functions in which they can be decoded.

11.12 Terminating or ignoring traversal

In any-hit programs, use the following functions to control traversal:

`optixTerminateRay`
: Causes the traversal execution associated with the current ray to immediately terminate. After termination, the closest-hit program associated with the ray is called.

`optixIgnoreIntersection`
: Causes the current potential intersection to be discarded. This intersection will not become the new closest hit intersection associated with the ray.

These functions do not return to the caller and they immediately terminate the program. Any modifications to ray payload values must be set before calling these functions.

11.13 Exceptions

Exceptions allow NVIDIA OptiX to check for invariants and to report details about violations.

To enable exception checks, set the `OptixPipelineCompileOptions::exceptionFlags` field with a bitwise combination of `OptixExceptionFlags`. Depending on the scenario and combination of flags, enabling exceptions can lead to severe overhead, so some flags should be mainly used in internal and debug builds.

There are several different kinds of exceptions, which are enabled based on the set of flags specified:

`OPTIX_EXCEPTION_FLAG_NONE`
: No exception set (default).

`OPTIX_EXCEPTION_FLAG_STACK_OVERFLOW`
: Checks for overflow in the continuation stack specified with the `continuationStackSize` parameter to `optixPipelineSetStackSize`. When this exception is enabled, the overhead is usually negligible.

`OPTIX_EXCEPTION_FLAG_TRACE_DEPTH`
: Before tracing a new ray, checks to see if the ray depth exceeds the value specified with `OptixPipelineLinkOptions::maxTraceDepth`. Some stack overflows are only detected if the exception for trace depth is enabled as well (or the value of `OptixPipelineLinkOptions::maxTraceDepth` is correct). When this exception is enabled, the overhead is usually negligible.

`OPTIX_EXCEPTION_FLAG_USER`
: Enables the use of `optixThrowException`().

`OPTIX_EXCEPTION_FLAG_DEBUG`
: Enables a number of run time checks including overflows for the traversable list, the validation of the set of traversables encountered at runtime, the validation of traced rays for nan or inf values, the validation of parameter counts for callable programs, and the validation of the sbt index when calling callable programs. When enabled, the overhead of this exception ranges from negligible to severe, depending on the type of GPU and the set of RTX features that are active.

If an exception occurs, the exception program is invoked. The exception program can be specified with an SBT record set in `OptixShaderBindingTable::exceptionRecord`. If exception flags are specified but no exception program is provided, a default exception program is provided by NVIDIA OptiX 7. This built-in exception program prints the first five exceptions that occurred to `stdout` to limit the amount of exception printing. Control does not return to the location that triggered the exception, and execution of the launch index ends.

In exception programs, the kind of exception that occurred can be queried with `optixGetExceptionCode`.

A number of exception codes are defined for the built-in exceptions.

Some exceptions provide additional information, accessed by API functions.

`OPTIX_EXCEPTION_CODE_STACK_OVERFLOW`
: Stack overflow of the continuation stack. No information functions.

`OPTIX_EXCEPTION_CODE_TRACE_DEPTH_EXCEEDED`
: The trace depth was exceeded. No information functions.

`OPTIX_EXCEPTION_CODE_TRAVERSAL_DEPTH_EXCEEDED`
: The traversal depth was exceeded. Information functions:

```
optixGetTransformListSize()
optixGetTransformListHandle()
```

`OPTIX_EXCEPTION_CODE_TRAVERSAL_INVALID_TRAVERSABLE`
: Traversal encountered an invalid traversable type. Information functions:

optixGetTransformListSize()
optixGetTransformListHandle()
optixGetExceptionInvalidTraversable()

`OPTIX_EXCEPTION_CODE_TRAVERSAL_INVALID_MISS_SBT`
The miss SBT record index is out of bounds. Information function:

optixGetExceptionInvalidSbtOffset()

`OPTIX_EXCEPTION_CODE_TRAVERSAL_INVALID_HIT_SBT`
The traversal hit sbt record index was out of bounds. Information functions:

optixGetTransformListSize()
optixGetTransformListHandle()
optixGetExceptionInvalidSbtOffset()
optixGetSbtGASIndex()

`OPTIX_EXCEPTION_CODE_INVALID_RAY`
The shader encountered a call to `optixTrace` with at least one of the float arguments being inf or nan. Information function:

optixGetExceptionInvalidRay()

`OPTIX_EXCEPTION_CODE_CALLABLE_PARAMETER_MISMATCH`
The shader encountered a call to either `optixDirectCall` or `optixContinuationCall` where there is a mismatch of the provided arguments and the parameters of the callable program which is called. Information function:

optixGetExceptionParameterMismatch()

`OPTIX_EXCEPTION_CODE_CALLABLE_INVALID_SBT`
The callable program sbt record index was out of bounds. Information functions:

optixGetExceptionInvalidSbtOffset()

`OPTIX_EXCEPTION_CODE_CALLABLE_NO_DC_SBT_RECORD`
The callable program sbt record does not contain a direct callable program. Information functions:

optixGetExceptionInvalidSbtOffset()

`OPTIX_EXCEPTION_CODE_CALLABLE_NO_CC_SBT_RECORD`
The callable program sbt record does not contain a continuation callable program. Information functions:

optixGetExceptionInvalidSbtOffset()

User exceptions can be thrown with values between 0 and $2^{30} - 1$. Zero to eight 32-bit-value details can also be used to pass information to the exception program using a set of functions of one to nine arguments:

Listing 11.10 – Function signatures for user exceptions

```
optixThrowException(
   unsigned int code);

optixThrowException(
   unsigned int code,
   unsigned int detail0);
```

```
optixThrowException(
    unsigned int code,
    unsigned int detail0,
    unsigned int detail1);

...             Exceptions for three to seven detail arguments

optixThrowException(
    unsigned int code,
    unsigned int detail0,
    unsigned int detail1,
    unsigned int detail2,
    unsigned int detail3,
    unsigned int detail4,
    unsigned int detail5,
    unsigned int detail6,
    unsigned int detail7);
```

The details can be queried in the exception program with eight functions, `optixGetExceptionDetail_0()` to `optixGetExceptionDetail_7()`.

These functions' behavior is undefined when exception detail for another exception code is queried or if user exception detail that is queried was not set with `optixThrowException`.

12 Callables

Callable programs allow for additional programmability within the standard set of NVIDIA OptiX 7 programs. They are invoked using their index in the shader binding table. Invoking a function with its index enables a function call to change its target at runtime without having to recompile the program. This increase in flexibility can enable, for example, different shading effects in response to user input or programs that can be customized based on the scene setup.

Two types of callable programs exist in NVIDIA OptiX 7: *direct callables* and *continuation callables*. Unlike direct callables, continuation callables can call the function `optixTrace`. Direct callables are called immediately, but because of their additional capability, continuation callables need to be executed by the scheduler. This may result in additional overhead when running continuation callables.

Since continuation-callable programs can only be called from other continuation-callable programs, nested callables that need to call `optixTrace` must be marked as continuation callables. This has implications when creating a shader network from callables that trace rays. In such cases, refactoring the shader network to avoid calling `optixTrace` is recommended to improve overall performance.

The use of continuation callables can also lead to better overall performance, especially if the code block in question is part of divergent code execution. A simple example is a variety of material parameter inputs, such as different noise functions or a complex bitmap network. Using a continuation callable for each of these inputs allows the scheduler to more efficiently execute these complex snippets and to potentially resolve most of the divergent code execution.

Direct callables can be called from any program type except exception. Continuation callables can be called from ray-generation, closest-hit, and miss programs.

There are three parts to calling a callable program.

1. Implementation of the program you wish to call. That program needs to be annotated with the appropriate name prefix `__direct_callable__` or `__continuation_callable__`.
2. The inclusion of a shader binding table record for the program group that contains the callable program.
3. The call in a program of the callable using the functions `optixDirectCall` and `optixContinuationCall`, shown in Listing 12.1 (page 84).

Listing 12.1 – Device functions with variadic templates

```
template<typename ReturnT, typename... ArgTypes>
ReturnT optixDirectCall(
   unsigned int sbtIndex, ArgTypes... args);

template<typename ReturnT, typename... ArgTypes>
ReturnT optixContinuationCall(
   unsigned int sbtIndex, ArgTypes... args);
```

The device-code compilation of these variadic templates requires C++11 or later.

The `sbtIndex` argument is an index of the array of callable shader binding table entries specified with the `OptixShaderBindingTable::callablesRecordBase` parameter to `optixTrace`. The program group for a callable can contain both a direct-callable and continuation-callable program, though these programs share a single record in the shader binding table.

To keep compilation time low for large functions (such as complex noise calculations) that are called from multiple sections of the code base, it is possible to use non-inlined functions. This mechanism is useful when a callable turns out to be too costly, requiring longer compilation times or degrading runtime performance. It also avoids the need to add callables to the shader binding table and to schedule the function during runtime execution (which might be preferable in some cases).

13 NVIDIA AI Denoiser

Image areas that have not yet fully converged during rendering will often exhibit pixel-scale noise due to the insufficient amount of information gathered by the renderer. This grainy appearance in an image may be caused by low iteration counts, especially in scenes with complex lighting environments and material calculations.

The NVIDIA AI Denoiser can estimate the converged image from a partially converged image. Instead of improving image quality through a larger number of path tracing iterations, the denoiser can produce images of acceptable quality with far fewer iterations by post-processing the image.

The denoiser is based on statistical data sets that guide the denoising process. These data, represented by a binary blob called a *training model*, are produced from a large number of rendered images in different stages of convergence. The images are used as input to an underlying *deep learning* system. (See the NVIDIA Developer article "Deep Learning"[14] for more information about deep-learning systems.)

Because deep-learning training needs significant computational resources—even obtaining a sufficient number of partially converged images can be difficult—a general-purpose model is included with the OptiX software. This model is suitable for many renderers. However, the model may not yield optimal results when applied to images produced by renderers with very different noise characteristics compared to those used in the original training data.

Post-processing rendered images includes image filters, such as blurring or sharpening, or reconstruction filters, such as box, triangle, or Gaussian filters. Custom post-processing performed on a noisy image can lead to unsatisfactory denoising results. During post-processing, the original high-frequency, per-pixel noise may become smeared across multiple pixels, making it more difficult to detect and be handled by the model. Therefore, post-processing operations should be done *after* the denoising process, while reconstruction filters should be implemented by using filter importance-sampling.

In general, the pixel color space of an image that is used as input for the denoiser should match the color space of the images on which the denoiser was trained. However, slight variations, such as substituting sRGB with a simple gamma curve, should not have a noticeable impact. Images used for the training model included with the NVIDIA AI Denoiser distribution were output directly as HDR data.

14. `http://developer.nvidia.com/deep-learning`

13.1 Functions and data structures for denoising

Calling function `optixDenoiserCreate` creates a denoiser object:

Listing 13.1

```
OptixResult optixDenoiserCreate(
    OptixDeviceContext          context,
    const OptixDenoiserOptions* options,
    OptixDenoiser*              denoiser);
```

The denoiser options determine the type of denoiser input and the format of pixel data:

Listing 13.2

```
typedef struct OptixDenoiserOptions {
    OptixDenoiserInputKind inputKind;
    OptixPixelFormat       pixelFormat;
} OptixDenoiserOptions;
```

The type of input data is defined by the enum `OptixDenoiserInputKind`:

```
OPTIX_DENOISER_INPUT_RGB
OPTIX_DENOISER_INPUT_RGB_ALBEDO
OPTIX_DENOISER_INPUT_RGB_ALBEDO_NORMAL
```

The structure of pixel data is defined by the enum `OptixPixelFormat`:

```
OPTIX_PIXEL_FORMAT_HALF3
OPTIX_PIXEL_FORMAT_HALF4
OPTIX_PIXEL_FORMAT_FLOAT3
OPTIX_PIXEL_FORMAT_FLOAT4
```

After denoising, `optixDenoiserDestroy` should be called to destroy the denoiser object along with associated host resources:

Listing 13.3

```
OptixResult optixDenoiserDestroy(
    OptixDenoiser denoiser);
```

13.1.1 Structure and use of image buffers

An RGB image buffer that contain a noisy image is provided as input to the denoising process. The optional fourth (alpha) channel of the image can be configured so that it is ignored by the denoiser. Note that this buffer must contain values between 0 and 1 for each of the three color channels (for example, as the result of tone-mapping) and should be encoded in sRGB or gamma space with a gamma value of 2.2 when working in LDR.

When using HDR input instead, RGB values in the color buffer should be in a range from zero to 10,000, and on average not too close to zero, to match the built-in model. Images in HDR format can contain single, extremely bright, nonconverted pixels, called *fireflies*. Using a preprocess pass that corrects drastic under- or over-exposure along with clipping or filtering

of fireflies on the HDR image can improve the denoising quality dramatically. Note, however, that no tone-mapping or gamma correction should be performed on HDR data.

The optional, noise-free normal buffer (`OPTIX_DENOISER_INPUT_RGB_ALBEDO_NORMAL`) contains the surface normals of the primary hit in camera space. The buffer contains data in format `OPTIX_PIXEL_FORMAT_HALF3` or `OPTIX_PIXEL_FORMAT_FLOAT3`. These two formats can represent the x, y and z components of a vector. However, because the normal vectors are defined in screen space, the z component is ignored. The camera space is assumed to be right handed such that the camera is looking down the negative z axis, and the up direction is along the y axis. The x axis points to the right. An optional fourth channel of this buffer is ignored. It must have the same type and dimensions as the input buffer.

The normal buffer can improve denoising quality for scenes containing a high degree of geometric detail. This detail may be part of the geometric complexity of the surface itself, or may be the result of high-resolution images used in normal or bump mapping.

The optional, noise-free albedo image represents an approximation of the color of the surface of the object, independent of view direction and lighting conditions. In physical terms, the albedo is a single color value approximating the ratio of radiant exitance to the irradiance under uniform lighting. The albedo value can be approximated for simple materials by using the diffuse color of the first hit, or for layered materials, by using a weighted sum of the albedo values of the individual BRDFs. For some objects such as perfect mirrors or highly glossy materials, the quality of the denoising result might be improved by using the albedo value of a subsequent hit instead. The fourth channel of this buffer is ignored, but must have the same type and dimensions as the input buffer. Specifying albedo can dramatically improve denoising quality, especially for very noisy input images.

13.1.2 Selecting the denoiser training model

To select a training model for the denoiser, call function `optixDenoiserSetModel`:

Listing 13.4

```
OptixResult optixDenoiserSetModel(
    OptixDenoiser          denoiser,
    OptixDenoiserModelKind kind,
    void*                  data,
    size_t                 sizeInBytes);
```

The `OptixDenoiserModelKind` is one of the following enum values:

`OPTIX_DENOISER_MODEL_KIND_LDR`
: The input image data contains values of limited dynamic range with RGB values in the range 0.0 to 1.0. The `data` and `sizeInBytes` parameters of `optixDenoiserSetModel` are not used.

`OPTIX_DENOISER_MODEL_KIND_HDR`
: The input image data contains values of high dynamic range with RGB values in the range of 0.0 to approximately 10,000.0 or more. The `data` and `sizeInBytes` parameters of `optixDenoiserSetModel` are not used.

`OPTIX_DENOISER_MODEL_KIND_USER`

The input image data points to weights in host memory (not device memory). The total size of this memory region must be given in the `sizeInBytes` parameter. The `data` value must not be zero.

`OPTIX_DENOISER_MODEL_KIND_AOV`

Separate passes can be defined by many rendering systems using *arbitrary output variables*, or AOVs. AOV images from a rendering system can contain diffuse, emission, glossy, specular or other types of data. High dynamic range AOV images can be passed as input layers to `optixDenoiserInvoke` in addition to the beauty, RGB, albedo and normal images. The AOV mode can be used to denoise multiple layers simultaneously and may reduce processing time.

After denoising, the denoised output layers can be composited to a noise-free beauty layer, typically by adding all RGB values from the denoised layers. The AOVs must be specified after the albedo or normal layers, depending on the `OptixDenoiserModelKind` value. Inference is run on the first, noisy beauty layer and then applied to the noisy AOVs. The first output layer is always the denoised beauty layer, followed by denoised AOVs.

For example, if four layers are passed to `optixDenoiserInvoke` (beauty, albedo, normal, diffuse) and the `OPTIX_DENOISER_INPUT_RGB_ALBEDO_NORMAL` input kind is selected, then there is one AOV for the diffuse component. Two output layers are written: a beauty layer and a layer for the AOV.

13.1.3 Allocating denoiser memory

To allocate the required, temporary device memory to run the denoiser, call function `optixDenoiserComputeMemoryResources`:

Listing 13.5

```
OptixResult optixDenoiserComputeMemoryResources(
    const OptixDenoiser denoiser,
    unsigned int        inputWidth,
    unsigned int        inputHeight,
    OptixDenoiserSizes* returnSizes);
```

The memory requirements are returned in the fields of struct `OptixDenoiserSizes`:

Listing 13.6

```
typedef struct OptixDenoiserSizes {
    size_t       stateSizeInBytes;
    size_t       minimumScratchSizeInBytes;
    size_t       recommendedScratchSizeInBytes;
    unsigned int overlapWindowSizeInPixels;
} OptixDenoiserSizes;
```


The `OptixDenoiserSizes` values are used as input to function `optixDenoiserSetup`:

Listing 13.7

```
OptixResult optixDenoiserSetup(
    OptixDenoiser denoiser,
    CUstream      stream,
    unsigned int  inputWidth,
    unsigned int  inputHeight,
    CUdeviceptr   denoiserState,
    size_t        denoiserStateSizeInBytes,
    CUdeviceptr   scratch,
    size_t        scratchSizeInBytes);
```

13.1.4 Using the denoiser

To execute denoising on a given image, call function `optixDenoiserInvoke`:

Listing 13.8

```
OptixResult optixDenoiserInvoke(
    OptixDenoiser              denoiser,
    CUstream                   stream,
    const OptixDenoiserParams* params,
    CUdeviceptr                denoiserState,
    size_t                     denoiserStateSizeInBytes,
    const OptixImage2D*        inputLayers,
    unsigned int               numInputLayers,
    unsigned int               inputOffsetX,
    unsigned int               inputOffsetY,
    const OptixImage2D*        outputLayer,
    CUdeviceptr                scratch,
    size_t                     scratchSizeInBytes);
```

Function `optixDenoiserInvoke` executes the denoiser on a set of input data and produces one output image. State memory must be available during the execution of the denoiser or until `optixDenoiserSetup` is called with a new state memory pointer. Scratch memory passed as a `CUdeviceptr` must be exclusively available to the denoiser during execution. It is only used for the duration of `optixDenoiserInvoke`. Scratch and state memory sizes must have a size greater than or equal to the sizes returned by `optixDenoiserComputeMemoryResources`.

Input and output to the denoiser uses struct `OptixImage2D` to represent pixel data:

Listing 13.9

```
typedef struct OptixImage2D {
    CUdeviceptr      data
    unsigned int     width;
    unsigned int     height;
    unsigned int     rowStrideInBytes;
    unsigned int     pixelStrideInBytes;
```

```
    OptixPixelFormat format;
} OptixImage2D;
```

Parameters `inputOffsetX` and `inputOffsetY` are pixel offsets in the `inputLayers` image. These offsets specify the beginning of the image without overlap. When denoising an entire image without tiling, there is no overlap and `inputOffsetX` and `inputOffsetY` must be zero. When denoising a tile which is adjacent to one of the four sides of the entire image the corresponding offsets must also be zero; there is no overlap at the side adjacent to the image border. Parameters `inputWidth` and `inputHeight` correspond to the values passed to `optixDenoiserComputeMemoryResources`.

Note that the `OptixPixelFormat` value of the `inputLayers` must match the format that was specified during `optixDenoiserCreate`. The `outputLayer` must have the same width, height and pixel format as the input RGB(A) layer. All input layers must have the same width and height.

Further control over denoising is provided by the `OptixDenoiserParams` struct:

Listing 13.10

```
typedef struct OptixDenoiserParams {
    unsigned int denoiseAlpha;
    CUdeviceptr  hdrIntensity = 0;
    float        blendFactor;
    CUdeviceptr  hdrAverageColor;
} OptixDenoiserParams;
```

The `denoiseAlpha` field of `OptixDenoiserParams` can be used to disable the denoising of the (optional) alpha channel of the noisy image. The `blendFactor` field specifies an interpolation weight between the noisy input image (1.0) and the denoised output image (0.0). Calculation of `hdrIntensity` is described in "Calculating the HDR intensity parameter" (page 91).

Parameter `hdrAverageColor` is used when the `OPTIX_DENOISER_MODEL_KIND_AOV` model kind is set. This parameter is a pointer to three floats that are the average log color of the RGB channels of the input image. The default value (the null pointer) will produce results that are not optimal.

13.1.5 Calculating the HDR average color of the AOV model

The function `optixDenoiserComputeAverageColor` computes the average logarithmic value for each of the first three channels of the input image. When denoising tiles the intensity of the entire image should be computed—not per tile—for consistent results. This function needs scratch memory with a size of at least:

```
sizeof(int) * (3 + 3 * inputImage::width * inputImage::height)
```

When denoising entire images without tiling, the same scratch memory as passed to `optixDenoiserInvoke` could be used.

The `inputImage` parameter must contain three or four components of type half or float. The data type `unsigned char` is not supported.

Listing 13.11

```
OptixResult optixDenoiserComputeAverageColor(
    OptixDenoiser       denoiser,
    CUstream            stream,
    const OptixImage2D* inputImage,
    CUdeviceptr         outputAverageColor,
    CUdeviceptr         scratch,
    size_t              scratchSizeInBytes);
```

13.1.6 Calculating the HDR intensity parameter

The value of `hdrIntensity` in `OptixDenoiserParams` can be calculated in one of two ways. A custom application-side, preprocessing pass on the image data could provide a result value of type `float`.

Intensity can also be calculated by calling function `optixDenoiserComputeIntensity`:

Listing 13.12

```
OptixResult optixDenoiserComputeIntensity(
    OptixDenoiser       denoiser,
    CUstream            stream,
    const OptixImage2D* inputImage,
    CUdeviceptr         outputIntensity,
    CUdeviceptr         scratch,
    size_t              scratchSizeInBytes);
```

Function `optixDenoiserComputeIntensity` calculates the logarithmic average intensity of parameter `inputImage`. The calculated value is returned in parameter `outputIntensity` as a pointer to a single value of type `float`.

Intensity calculation can be dependent on `optixDenoiserInvoke`. The `params` parameter of `optixDenoiserInvoke` is a struct of type `OptixDenoiserParams`. By default, the `hdrIntensity` field of `OptixDenoiserParams` is a null pointer. If `hdrIntensity` is not a null pointer, it points to an array of RGB values that are multiplied by the output values in `outputIntensity`. This is useful for denoising HDR images which are very dark or bright. When denoising with tiles, the intensity of the entire image should first be computed for consistent results across tiles. (Tiling is described in the "Using image tiles with the denoiser" (page 92).)

For each RGB pixel in the inputImage the intensity is calculated and summed if it is greater than 1e-8f as follows:

$$intensity = log(r \times 0.212586 + g \times 0.715170 + b \times 0.072200)$$

The function returns:

$$\frac{0.18}{exp\left(\frac{sum\_of\_intensities}{number\_of\_summed\_pixels}\right)}$$

Execution of `optixDenoiserComputeIntensity` requires a scratch memory size in bytes of at least:

```
sizeof(int) * (2 + inputImage::width * inputImage::height)
```

When denoising entire images (without tiling) the same scratch memory that is passed to `optixDenoiserInvoke` can be used.

Further information about this tone-mapping strategy can be found in "Photographic Tone Reproduction for Digital Images"[15] by Erik Reinhard, et al.

13.2 Using image tiles with the denoiser

Denoising can be performed incrementally on subregions of an image to limit memory usage during the denoising process. An image is divided in to a set of *tiles* defined by their width and height, as well as a surrounding region of overlapping pixels to avoid discontinuities at tile boundaries.

This section defines function `create_tiles_for_denoising` that returns a vector of `Tile` structs through the reference argument `tiles`. The `Tile` struct contains input arguments for function `optixDenoiserInvoke` to denoise that subregion of the image.

Listing 13.13 is a header file for function `create_tiles_for_denoising`. The header also declares utility function `get_pixel_size`.

Listing 13.13 – File `create_tiles_for_denoising.h`

```
#include <algorithm>
#include <vector>
#include "optix.h"
#include "optix_host.h"
#include "optix_types.h"

struct Tile
{
  OptixImage2D input;
  OptixImage2D output;
  unsigned int inputOffsetX;
  unsigned int inputOffsetY;
};
```

Tile parameters for optixDenoiserInvoke

15. `http://www.cmap.polytechnique.fr/~peyre/cours/x2005signal/hdr_photographic.pdf`


```
OptixResult create_tiles_for_denoising(
  void*              inputBuffer,
  void*              outputBuffer,
  int                inputWidth,
  int                inputHeight,
  OptixPixelFormat   pixelFormat,
  int                overlap,
  int                tileWidth,
  int                tileHeight,
  std::vector<Tile>& tiles);
```

Tiling function

```
unsigned int get_pixel_size(
  OptixPixelFormat pixelFormat);
```

Calculate number of bytes in a pixel, based on pixel format

Figure 13.1 describes the geometry of the layout of input and output instances of `OptixImage2D` with respect to `inputBuffer` and `outputBuffer`, the arguments of `create_tiles_for_denoising`. Bold text indicates fields in the `Tile` struct.

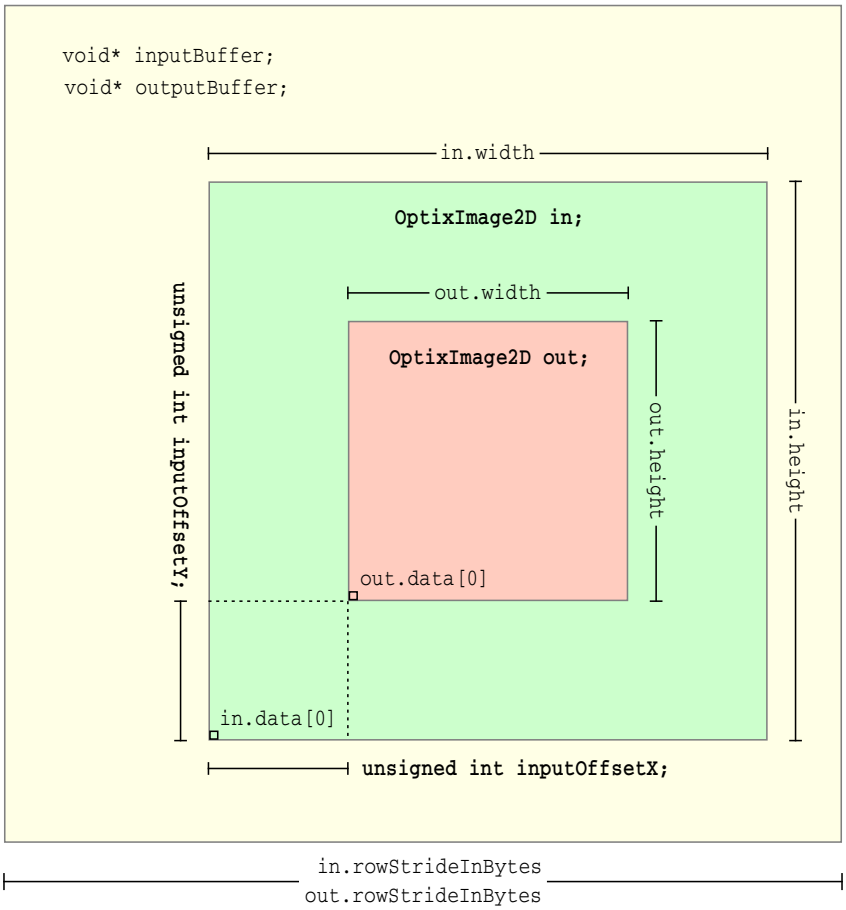

Fig. 13.1 – Geometry of denoiser tiling layout

The tiling code must accommodate tile sizes that do not evenly divide the image buffers. Figure 13.2 shows a tile that with a consistent overlap size on all edges.

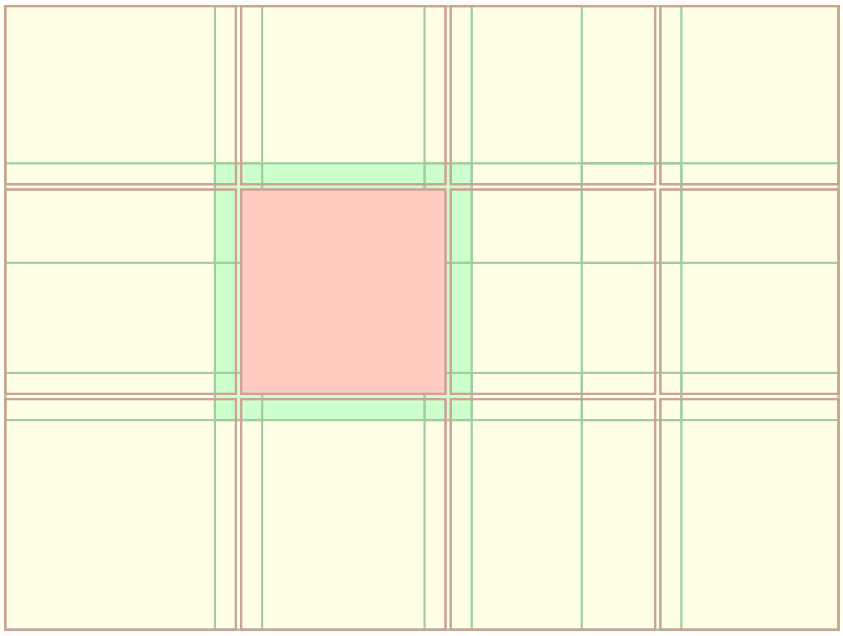

Fig. 13.2 – Output tile with overlap

Figure 13.3 show how the tiling mechanism varies the tile sizes based on their position in the input and output buffers.

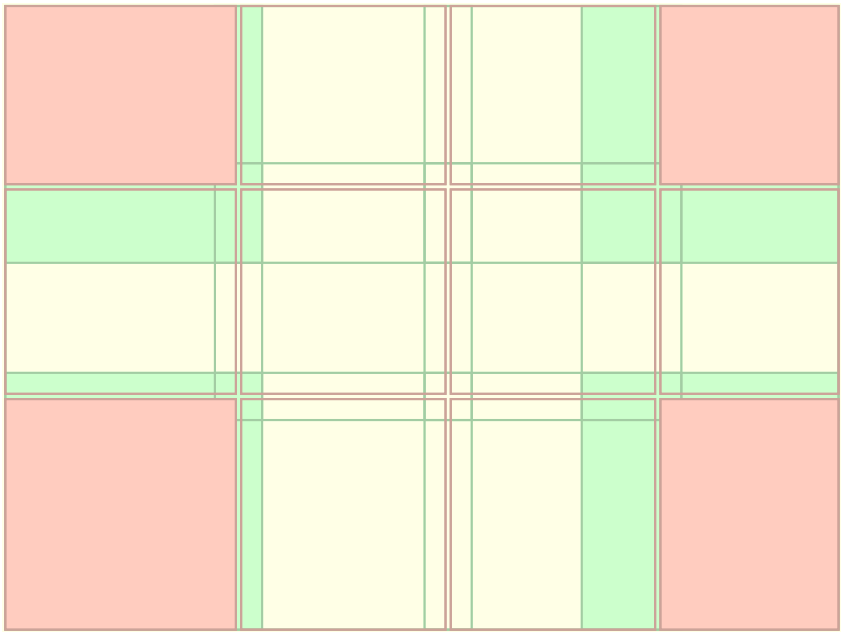

Fig. 13.3 – Tiles in corners showing variable overlap

Listing 13.14 (page 95) defines function `create_tiles_for_denoising`. The denoising parameters are calculated based on offsets into the image buffers and saved in the `Tile` structs. Each new `Tile` struct is added to the vector `tiles`.

Listing 13.14 – File `create_tiles_for_denoising.cpp`

```
#include "create_tiles_for_denoising.h"

OptixResult create_tiles_for_denoising(
   void*              inputBuffer,
   void*              outputBuffer,
   int                inputWidth,
   int                inputHeight,
   OptixPixelFormat   pixelFormat,
   int                overlap,
   int                tileWidth,
   int                tileHeight,
   std::vector<Tile>& tiles)
{
   unsigned int pixelSizeInBytes =
      get_pixel_size(pixelFormat);
   unsigned int rowStrideInBytes =
      inputWidth * pixelSizeInBytes;
```

Byte counts for offsets in `inputBuffer` and `outputBuffer`

```
   int input_width =
      std::min(tileWidth + 2 * overlap, inputWidth);
   int input_height =
      std::min(tileHeight + 2 * overlap, inputHeight);
```

Tile size includes overlap, but isn't larger than the input

```
   int input_y = 0, copied_y = 0;
   do {
      int inputOffsetY = 0;
      int copy_y = tileHeight + overlap;

      if (input_y != 0) {
         inputOffsetY =
            std::max(overlap,
                    input_height - (inputHeight - input_y));
         copy_y = std::min(tileHeight, inputHeight - copied_y);
      }
```

Adjust tile size and offset after first tile

```
      int input_x = 0, copied_x = 0;
      do {
         int inputOffsetX = 0;
         int copy_x = tileWidth + overlap;

         if (input_x != 0) {
            inputOffsetX =
               std::max(overlap,
                       input_width - (inputWidth - input_x));
            copy_x = std::min(tileWidth, inputWidth - copied_x);
         }
```

Adjust tile size and offset after first tile

```
         Tile tile {};
```

Struct for the current tile

```
            tile.input.data =
              (CUdeviceptr)(
                (char*)inputBuffer
                + (input_y - inputOffsetY) * rowStrideInBytes
                + (input_x - inputOffsetX) * pixelSizeInBytes);
            tile.input.width = input_width;
            tile.input.height = input_height;
            tile.input.rowStrideInBytes = rowStrideInBytes;
            tile.input.format = pixelFormat;
```

Tile input parameters

```
            tile.output.data =
              (CUdeviceptr)(
                (char*)outputBuffer
                + input_y * rowStrideInBytes
                + input_x * pixelSizeInBytes);
            tile.output.width = copy_x;
            tile.output.height = copy_y;
            tile.output.rowStrideInBytes = rowStrideInBytes;
            tile.output.format = pixelFormat;
```

Tile output parameters

```
            tile.inputOffsetX = inputOffsetX;
            tile.inputOffsetY = inputOffsetY;
```

Offset from input to output

```
            tiles.push_back(tile);
```

Add current tile to vector of tiles

```
          input_x += input_x == 0 ? tileWidth + overlap : tileWidth;
          copied_x += copy_x;
        } while (input_x < inputWidth);
        input_y += input_y == 0 ? tileHeight + overlap : tileHeight;
        copied_y += copy_y;
      } while (input_y < inputHeight);
      return OPTIX_SUCCESS;
    }

    unsigned int get_pixel_size(OptixPixelFormat pixelFormat)
    {
      switch(pixelFormat) {
      case OPTIX_PIXEL_FORMAT_HALF3:
        return 3 * sizeof(unsigned short);
      case OPTIX_PIXEL_FORMAT_HALF4:
        return 4 * sizeof(unsigned short);
      case OPTIX_PIXEL_FORMAT_FLOAT3:
        return 3 * sizeof(float);
      case OPTIX_PIXEL_FORMAT_FLOAT4:
        return 4 * sizeof(float);
      default:
        return OPTIX_ERROR_INVALID_VALUE;
      }
    }
```


14 Demand-loaded sparse textures

This section describes the OptiX Demand Loading library, which is a standalone CUDA library that is distributed as open-source code in the OptiX SDK. (It is free for commercial use.) Although it is part of the OptiX SDK, the Demand Loading library has no OptiX dependencies, and it is technically not part of the OptiX API.

The OptiX Demand Loading library allows hardware-accelerated sparse textures to be loaded on demand, which greatly reduces memory requirements, bandwidth, and disk I/O compared to preloading textures. It works by maintaining a page table that tracks which texture tiles have been loaded into GPU memory. An OptiX closest-hit program fetches from a texture by calling library device code that checks the page table to see if the required tiles are present. If not, the library records a page request, which is processed by library host code after the kernel has terminated. Kernel launches are repeated until no tiles are missing, typically using adaptive sampling to avoid redundant work. Although it is currently focused on texturing, much of the library is generic and can be adapted to load arbitrary data on demand, such as per-vertex data like colors or normals.

Before describing the design and implementation of the library in more detail, we first provide some motivation and background information.

14.1 Motivation

The size of film-quality texture assets has been a longstanding obstacle to rendering final-frame images for visual effects shots and animated films. It's not unusual for a hero character to have 50-100 GB of textures, and many sets and props are equally detailed (sometimes for no good reason, except that it would be more expensive to author at a more appropriate resolution).

Film models are usually painted at extremely high resolution in case of extreme closeup. A character's face is typically divided into several regions (for example, eye, nose, mouth), each of which is painted at 4K or 8K resolution. In addition, numerous layers are often used, including coarse and fine displacement and various material layers such as dirt, each of which is a separate collection of textures.

It's not practical to preload entire film-quality textures into GPU memory. Besides wasting GPU memory, it also requires time and bandwidth to read the textures from disk and transfer them to GPU memory. Nor is it possible to downscale the images, since the level of detail that is required depends on the distance of the textured object from the camera. Simply clamping the maximum resolution can lead to a loss of detail that is not acceptable in a film-quality renderer.

To make matters worse, it's not unusual for a production shot to have thousands of textured objects that are outside the camera view or completely occluded by other objects. It's generally not possible to know *a priori* which textures are required without actually rendering a scene.

All of these factors make it difficult to load textures into GPU memory before rendering. Our approach is a multi-pass one: the initial passes identify missing texture data, which is loaded from disk on demand and transferred to GPU memory between passes.

14.2 Background

Mipmapping is a well known technique for dealing with high resolution textures. The idea is to precompute downscaled images, as illustrated in Figure 14.1, that can be used when a lower level of detail is acceptable, such as when a textured object is far from the camera.

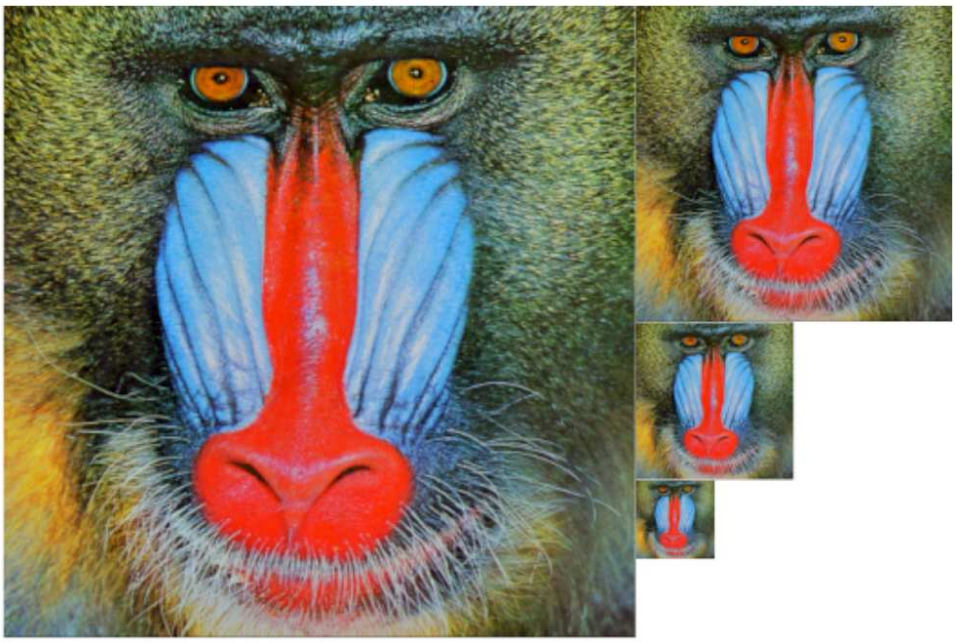

Fig. 14.1 – Mipmapped texture showing downscaled images

In a ray tracer, choosing which level of detail to use is usually accomplished using ray differentials to determine the gradients of the texture coordinates. The texture hardware on the GPU then uses those gradients to choose two miplevels, and then it performs a filtered lookup in each miplevel, and blends between the two.

Needless to say, that's a very expensive operation. But it's necessary to avoid temporal aliasing when the camera or the geometry is moving, which would otherwise create a pop when the choice of miplevels suddenly changes from one frame to the next. GPU acceleration for mipmapped texturing is often taken for granted, but it offers a huge performance gain compared to texturing on the CPU.

14.2.1 Sparse textures

Mipmapped textures are usually tiled, which divides each miplevel into a number of smaller sub-images. This allows a sparse memory layout to be used. Instead of loading an entire miplevel onto the GPU, only the individual tiles that are required are loaded, which saves memory and bandwidth.

For example, consider rendering a quad angled away from the camera, as shown in Figure 14.2 (page 99). The region that is closest to the camera must use tiles from a high resolution miplevel to ensure adequate detail. However, tiles from coarser miplevels can be used farther from the camera.

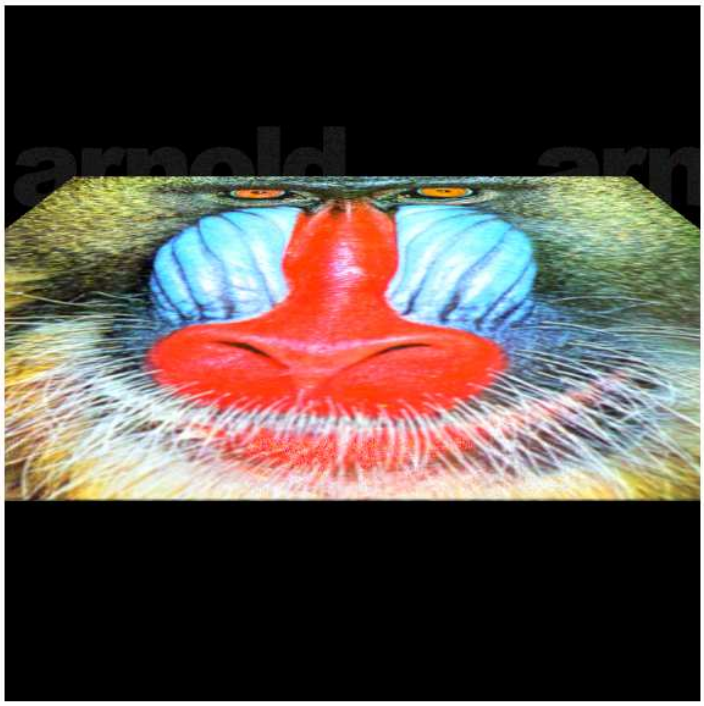

Fig. 14.2 – Texture-mapped quad angled away from the camera

For example, the tiles highlighted in Figure 14.3 illustrate which tiles might be required:

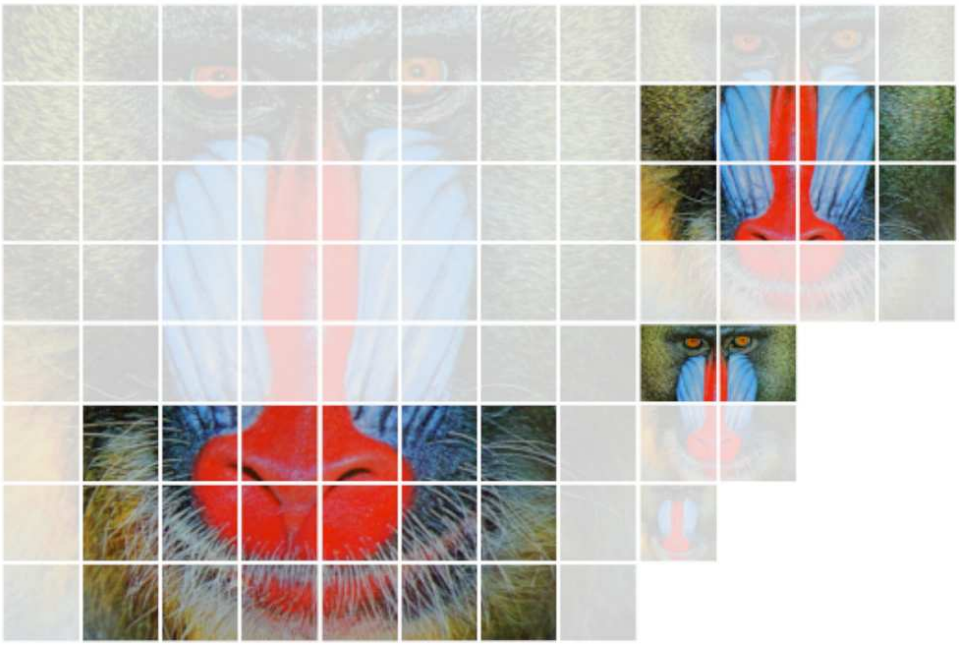

Fig. 14.3 – Mipmap tiles potentially required by Figure 14.3

Sparse textures are now supported in CUDA, starting with the CUDA 11.1 toolkit. Under the hood, CUDA sparse textures employ the virtual memory system in a clever way. The main complication is that texture filtering often requires samples from multiple tiles for a single texture fetch. In that case, the GPU texture units require those tiles to be contiguous in memory. But it would be wasteful to reserve enough space to allow that.

The trick is to use the virtual memory system to provide the illusion of contiguous tiles to the GPU texture units. That's accomplished by binding the texture sampler to virtual memory, and then mapping individual tiles as virtual pages.

Figure 14.4 illustrates how that works. On the left each tile is labeled with its virtual address. The page table shown in the middle maps that virtual address to a tile stored at an arbitrary offset in the physical backing storage, shown on the right.

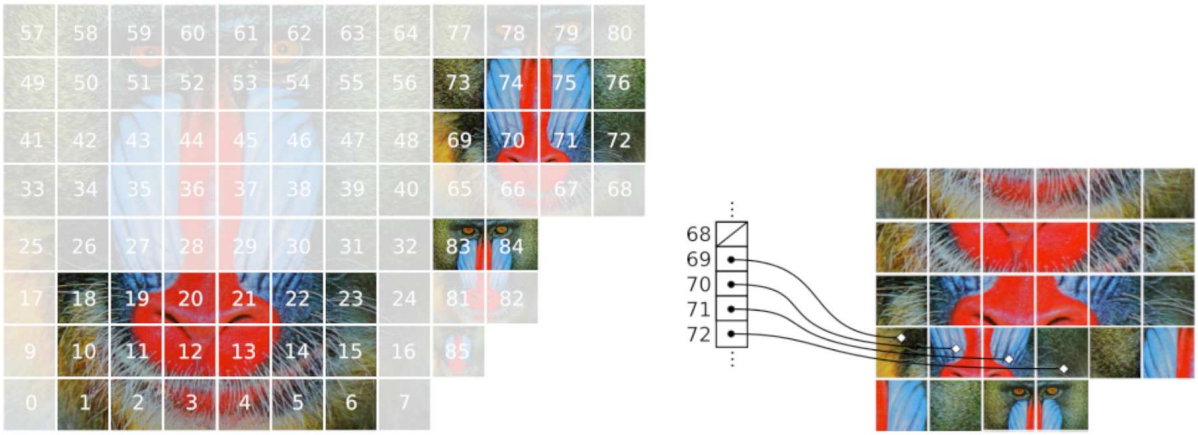

Fig. 14.4 – Tiles label-led by virtual addresses stored by offsets in the backing storage

Of course, the level of indirection is not free, but it's hardware accelerated by the virtual memory system. The benefit is that the backing storage is quite compact, compared to the memory that would be required if physical memory was allocated for all the tiles in contiguous order.

14.3 Library overview

The OptiX Demand Loading library provides a framework for loading CUDA sparse textures on demand. It is based on a multi-pass approach to rendering:

A page table tracks which tiles are resident on the GPU. As an OptiX kernel discovers missing tiles, it records page requests. After a batch of requests has been accumulated, the kernel exits to allow the page requests to be processed on the CPU. For each request, a tile is loaded from disk, decompressed, and copied to GPU memory. Once the required tiles have been loaded into GPU memory, the page table is updated and the OptiX kernel is relaunched Any pixels with missing texture tiles are resampled. Each pass might encounter other missing tiles, for example due to dependent texture reads or newly followed ray paths, so the process is repeated until there are no more misses.

14.4 Texture fetch

The OptiX Demand Loading library provides the following CUDA function (implemented entirely as header code), which used to fetch from a demand-loaded sparse texture with four floating-point channels. (Additional overload functions are available for other formats. Only 2D textures are currently supported.)

Listing 14.1

```
float4 tex2DGrad(
    const DemandTextureContext& context,
    unsigned int textureId,
    float x, float y,
    float2 ddx, float2 ddy,
    bool* isResident);
```

The arguments are as follows:

`context`
: Contains the page table and other data.

`textureId`
: Used to determine the offset in the page table at which the texture tiles are tracked.

`x, y`
: The texture coordinates (normalized from zero to one).

`ddx, ddy`
: The gradients of the texture coordinates, which are used to determine the level of detail required (that is, which miplevel). The gradients are usually calculated from ray differentials in an OptiX closest-hit shader.

`isResident`
: An output parameter, which is set to true if the required tiles are resident in GPU memory.

The following steps are performed, as illustrated by Figure 14.5.

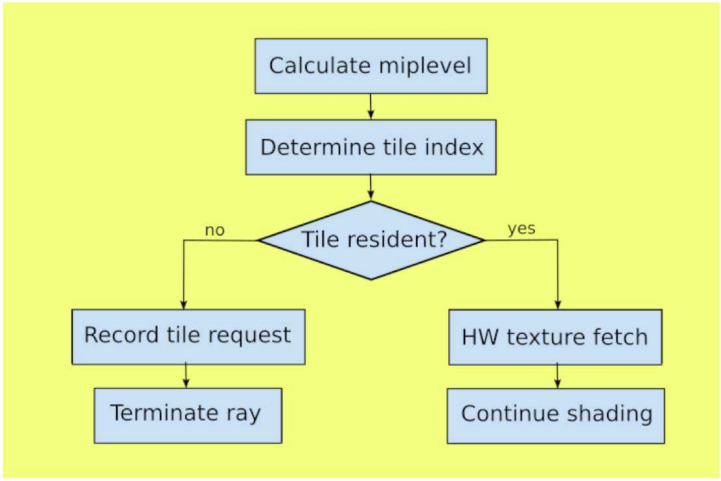

Fig. 14.5 – Texture fetch steps

1. First the required miplevels are calculated, which is based on the gradients. As usual, two miplevels are typically selected, and the final result is a weighted average.
2. The texture footprint is then calculated, using the texture coordinates to determine the offsets of the texture samples within the required miplevel. Linear filtering employs multiple samples, so up to four tiles might be required per miplevel.

3. For each required tile, its page table index is calculated by adding the tile index to the offset of this texture's page table entries.
4. If the required tiles are resident, the usual texture fetch instruction is executed, as shown on the right side of the illustration. That provides fully hardware accelerated sampling and filtering.
5. If a tile is missing a tile request is recorded (as shown on the left). Typically the ray is then terminated by throwing an OptiX user exception. But this step is user-programmable, in the closest hit program. So alternatively a default color could be substituted to allow shading to continue. Of course that's an approximation that might only be acceptable in a preview render or as an interim step during progressive refinement.

14.5 Page request processing

After a batch of tile requests has accumulated on the GPU, the OptiX kernel exits to allow them to be processed on the CPU.

As illustrated in Figure 14.6, the tile requests are processed by a parallel for loop that reads tiles from disk, decompresses them, and copies them into GPU memory. The OptiX Demand Loading library provides support for reading OpenEXR textures; incorporating other third-party libraries is straightforward.

When all the requests are processed, the OptiX kernel is relaunched for another rendering pass. Additional page requests might arise, for example due to dependent texture reads, so the entire process is repeated until there are no more requests.

This fits quite naturally into an adaptive sampling framework. It's not necessary to keep track of which rays failed due to missing tiles. Those rays make no contribution to the framebuffer, so the adaptive sampler will naturally resample the missing pixels in the next kernel launch.

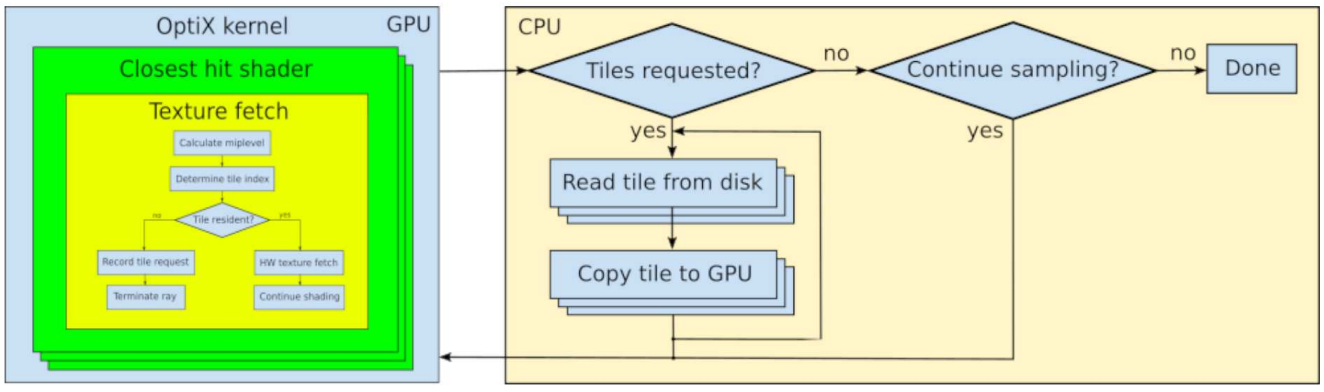

Fig. 14.6 – An adaptive sampling framework

14.6 Host-side library

In addition to the texture fetch function described above, which is implemented entirely as CUDA header code, the OptiX Demand Loading library provides a C++ interface for host side operations.

The primary interface is the `DemandTextureManager`, which is obtained from the following function by providing a list of active devices and a few configuration options:

Listing 14.2

```
DemandTextureManager*
    createDemandTextureManager(
        const std::vector<unsigned int>& devices,
        const DemandTextureManagerConfig& config);
```

The following method creates a demand-loaded sparse texture. The texture descriptor specifies the filter mode and wrap mode, etc.

Listing 14.3

```
const DemandTexture& createTexture(
    std::shared_ptr<ImageReader> image,
    const TextureDescriptor& textureDesc);
```

The texture initially has no backing storage. The `ImageReader` argument has a `readTile()` method that serves as a callback during page request processing.

The returned `DemandTexture` object is mostly opaque. It provides a `getId()` method that provides the texture identifier, which is required by the texture fetch function described above. The texture id would typically be associated with an object in the OptiX shader binding table (SBT).

Before launching an OptiX kernel, the host program calls the following `DemandTextureManager` method, which updates device-side data structures including the page table and an array of sparse texture samplers. The method returns a context (by result parameter) that is required by the texture fetch function. The host program typically passes the context to the OptiX kernel as a launch parameter after copying it to device memory.

Listing 14.4

```
void launchPrepare(
    unsigned int deviceIndex,
    DemandTextureContext& demandTextureContext);
```

After an OptiX kernel has exited, the host program calls the following `DemandTextureManager` method, which processes any accumulated page requests. As described above, the tile requests are processed by a parallel for loop that reads tiles from disk, decompresses them, and copies them into GPU memory.

Listing 14.5

```
int processRequests();
```

When rendering is complete, the following function is used to destroy the `DemandTextureManager`, which frees all its resources, including device-side sparse texture memory.

Listing 14.6

```
void destroyDemandTextureManager(DemandTextureManager* manager);
```

14.7 Implementation

The full source for the OptiX Demand Loading library is provided in the OptiX SDK, along with a sample that illustrates its use. It is free for commercial use. The source code is accompanied by documentation (generated by Doxygen) that describes the implementation in detail. Questions are welcome on the NVIDIA OptiX forums.

We expect that some developers will fork the library to customize it, while others will rely on NVIDIA for continued feature development. The source code might move to a public repository in the future to facilitate such uses, with periodic library deliveries as part of the OptiX SDK.


NVIDIA