NVIDIA OptiX 7.0

Programming Guide

23 July 2019

Version 1.8

NVIDIA OptiX 7.0 – Programming Guide

Document build number 321303

Contents

1 Introduction

Three public APIs support NVIDIA ray tracing capabilities: DirectX Raytracing (DXR), Vulkan (VK_NV_ray_tracing extension) and the NVIDIA OptiXTM API. DXR and Vulkan enable ray tracing effects in raster-based gaming and visualization applications. NVIDIA OptiX supports ray tracing applications that use NVIDIA® CUDA® technology, for example: film and television VFX, computer-aided design for engineering and manufacturing, PLM, HPC, and Lidar simulation.

The new NVIDIA OptiX 7 API is similar to DXR in terms of abstraction level and features. However, NVIDIA OptiX 7 adds support for motion blur and multi-level transforms, which are features that are required in ray tracing applications aiming for production-quality rendering.

NVIDIA OptiX 7 includes functions to build acceleration structures and to compile networks of programs that perform ray tracing operations. It employs the same programming model as DXR, Vulkan and current versions of NVIDIA OptiX, including intersection, any-hit and closest-hit programs among others.

1.1 General description

The NVIDIA OptiX 7 API is a CUDA-centric API that is easily invoked by a CUDA-based application. The API is designed to be stateless, multi-threaded, asynchronous and explicitly controlled.

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

An NVIDIA OptiX 7 context controls a single GPU. The context does not hold bulk CPU allocations, but similarly to 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 10s of kilobytes) and are independent of the size of the GPU resources being used. See "Program pipeline creation" (page 31) for small exceptions to this rule.

Most of NVIDIA OptiX 7 is thread-safe; exceptions for specific API functions are documented.

The application invokes acceleration structure 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 with CUDA events to avoid race conditions on the GPU.

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

For efficiency and coherence reasons, the NVIDIA OptiX 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/thread, warp or streaming multiprocessor (SM). Consequently, applications cannot use shared memory, synchronizations, 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.

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, the term program is used in NVIDIA OptiX 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.

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.

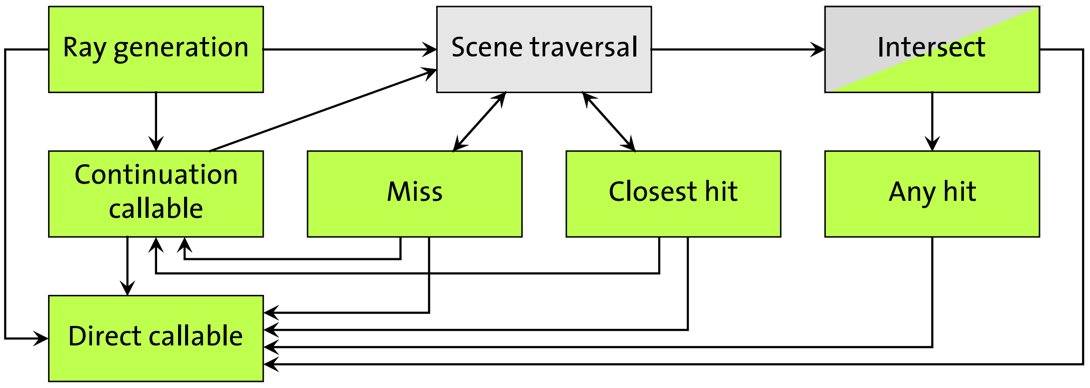

Fig. 2.1 – Relationship of OptiX programs. Gray boxes are fixed functions; green are user programs.

In the interest of brevity, this document sometimes refers to the program types as follows:

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

In Figure 2.1, gray boxes indicate fixed-function, hardware-accelerated operations, while green boxes indicate 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 in the shader binding table (SBT) is selected during execution by using offsets specified at build and run time. A record contains two data regions, *header* and *data*.

1. Record header
   - Opaque to the application, filled in by `optixSbtRecordPackHeader`
   - Used by NVIDIA OptiX to identify programmatic behavior.
   - For example a primitive would identify the intersection, any-hit, and closest-hit behavior for that primitive in the header.
2. Record data
   - Opaque to NVIDIA OptiX
   - 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 into and out of `optixTrace` and the programs invoked during ray traversal. Payload values are passed into 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 intersection 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 in NVIDIA OptiX 6, the allocation and transfer of buffers is explicitly controlled by user code.

2.3 Acceleration structures

NVIDIA OptiX acceleration structures are opaque data structures built on the device. Typically, they are BVH-based, but implementations and data layout of these structures may vary from one GPU architecture to another.

NVIDIA OptiX provides two basic types of acceleration structures:

1. Geometry Acceleration Structures (GAS), which are built over primitives (triangles or user-defined primitives).
2. Instance Acceleration Structures (IAS)
   - Are built over other traversable objects such as acceleration structures (GAS or IAS) or motion transform nodes

- Allow for instancing with per instance static transform.

2.4 Traversables

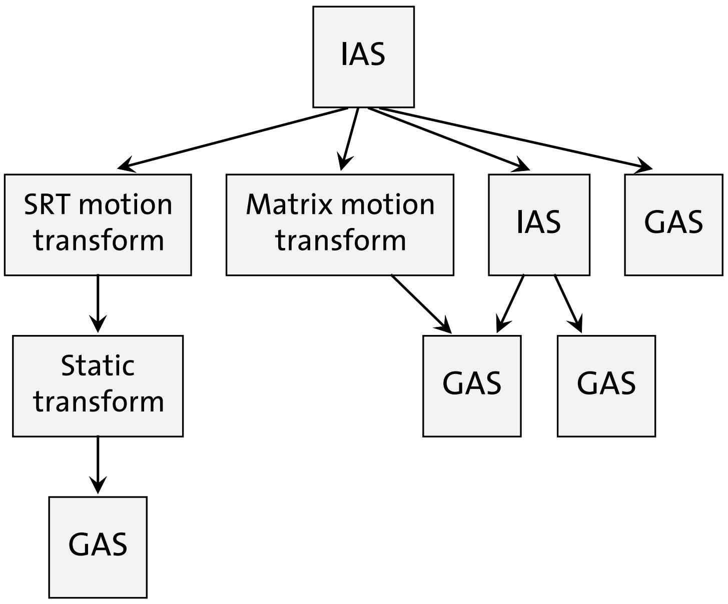

Fig. 2.2 – Example graph of traversables for a scene containing static as well as dynamic motion-transform driven objects.

Traversal can accept the following traversable objects:

- IAS
- GAS (as a root for graph with a single GAS (see "Single GAS traversal" (page 26))
- Transform
- Matrix Motion
- Scaling, Rotation, Translation (SRT) Motion
- Static (no motion)

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

Note: DXR and VulkanRT use the terms TLAS (top-level acceleration structure) and BLAS (bottom-level acceleration structure). BLAS is the same as a GAS, and TLAS is similar to IAS. The single GAS traversable, motion transform nodes and nested IAS are not supported in DXR or VulkanRT. In 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.

2.5 Ray tracing with NVIDIA OptiX 7

A functional ray tracing system is implemented by combining four pieces as follows:

1. Build one or more acceleration structures that represent a geometry mesh in the scene and select one or more SBT records for each mesh. See "Acceleration structures" (page 15).
2. Build a pipeline of programs that contains all of the programs to be invoked during a ray tracing launch. See "Program pipeline creation" (page 31).
3. Build an SBT table that includes references to these programs and their parameters. See "Shader binding table" (page 41).
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 49). Device-side functionality is described in Device-side functions.

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. OptiX 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 utilization feedback.

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 build 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 internally.

In NVIDIA OptiX 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 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 library uses a function table approach to facilitate the introduction of new features in future releases while maintaining backwards 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 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 the OS-specific code 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;
...             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, we provide optional stubs that wrap the addresses in the function table into C functions. These stubs are made available by including the header file `optix_stubs.h`. With these stubs the previous example simplifies to:

Listing 3.4

```
CUcontext fromContext;
...            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 optix device context is created by specifying the CUDA context associated with the device. For convenience, zero can be passed and OptiX will use the current CUDA context.

Listing 4.1

```
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`. For now the options include parameters for specifying a callback function, log and data. (See "Sending messages with a callback function" (page 12).)

A handful 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 casual snoopers.

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, mixing only GPUs of the same SM 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 OptiX 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 newline at the end) and 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 and/or NVIDIA OptiX 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

If enabled, when creating `OptixModule`, `OptixProgramGroup`, and `OptixPipeline` objects any compilation of the input programs will be cached to disk. 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 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 set to 1, it enables the use of the disk cache.
- When set to 0, the disk cache is disabled. Note that no in-memory cache is used, so no caching behavior will be observed in this case.
- 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 and a message will be reported to the log callback if enabled. In this case the call to `optixDeviceContextCreate` will not return an error. To ensure that cache initialization succeeded on context creation, the status can be queried using `optixDeviceContextGetCacheEnabled`. If this indicates that caching is disabled, the cache can be reconfigured and then enabled using `optixDeviceContextSetCacheEnabled`. If the cache database cannot be initialized with `optixDeviceContextSetCacheEnabled`, an error will be returned. Garbage collection will be performed on the next write to the cache database and not immediately when the cache is enabled.

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

- Sets the folder location where the disk cache will be created. The location must be provided as a NULL-terminated string. The directory will be created if it does not exist.

  The cache database will be created immediately if the cache is currently enabled. Otherwise the cache database will be created later when the cache is enabled. An error will be returned if 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:
  - 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.
  - MacOS X: `/Library/Application Support/NVIDIA/OptixCache`

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

- Sets the low and high water marks for disk cache garbage collection. Setting either limit to zero will disable garbage collection. Garbage collection only happens when the cache database is written and is triggered whenever the cache data size exceeds the high water mark and proceeds until the size reaches the low water mark. Garbage collection will always free enough space to allow inserting the new entry without exceeding the low water mark. An error will be returned if either limit is non-zero and the high water mark smaller 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 will use its values when writing to the cache database.

Corresponding `get*` functions are also 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
```

Instance acceleration structures (IAS) and geometry acceleration structures (GAS) are created on the device using a set of functions that facilitate overlapping and pipelining of builds. These functions use one or more `OptixBuildInput` structs to specify the geometry plus a set of parameters to control the build. To ensure compatibility with future versions, the `OptixBuildInput` structure should be zero initialized before populating it with specific build inputs.

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 over built-in triangles
: `OPTIX_BUILD_INPUT_TYPE_TRIANGLES`

A geometry acceleration structure over custom primitives
: `OPTIX_BUILD_INPUT_TYPE_CUSTOM_PRIMITIVES`

For GAS builds, each build input can specify a set of triangles 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 that include a ray transformation and an `OptixTraversableHandle` that refers to a GAS, 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 "Compaction" (page 24) for more information.

`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, and 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 (AS) over these inputs. This acceleration structure can be of type IAS or GAS 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. Accordingly, 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 any need to wait for the build to finish. By producing handles at acceleration time, custom handles can also be generated based on input to the builder.

The AS 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, IAS builds do continue to refer to other IAS and GAS 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 values of zero (set by `memset` here) implies that there is no motion blur

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

...
```

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

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 26) for more information. 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 more efficient format (potentially of a different size to the input). 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 and contain 12 floats. If specified, the transformation is applied to all vertices at build time with no runtime traversal overhead.

Acceleration structures over custom primitives are supported by referencing an array of primitive AABB (axis aligned bounding box) buffers in device memory, with one buffer per motion key. The layout of an AABB is defined in the struct `OptixAabb`. Here is an example of how to specify the build input for custom primitives:

Listing 5.3

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

`optixAccelBuild` accepts multiple triangle inputs or multiple AABB inputs per call, but mixing triangle and AABB build inputs is not allowed.

Each build input maps to one or more consecutive SBT records that control program dispatch. See "Shader binding table" (page 41) for more information. If multiple SBT records 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. 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 2 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`
: Apply the default behavior of calling the AH program, possibly multiple times allowing the AS-builder to apply all optimizations.

`OPTIX_GEOMETRY_FLAG_REQUIRE_SINGLE_ANYHIT_CALL`
: Disables some AS-builder specific optimizations. 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 degrade 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 IS, AH and CH program. The application can choose to offset this index for all primitives in a build input with no overhead at runtime. This can be particularly useful when data for consecutive build inputs is stored consecutively in device memory. The primitiveIndexOffset 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 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
- An SBT offset
- Instance flags
- An 8-bit visibility mask
- A 24-bit user-supplied ID.

Unlike the triangle and AABB inputs, `optixAccelBuild` only accepts a single build input per build call.

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 GAS connected to an instance. The flags override any instance flags set during the traversal of parent IASs.

`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 AH calls for triangle and primitive intersections. Can be overridden by ray flags.

`OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT`
: Forces AH calls for triangle and 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 53).)

Instance build inputs have an optional `OptixBuildInputInstanceArray`::aabbs member. This is a pointer to a device buffer of (motion) AABBs, one per instance per motion key. The AABBs for different motion keys of a single instance are laid out consecutive in memory. Input AABBs are ignored for instances where the AS has no motion blur (that is, numKeys equals 0;

and the instance references an AS via `OptixInstance`::traversableHandle. (See "Motion blur" (page 26).) In this case, it is possible to query the AABB directly from the AS and the application can leave such AABBs uninitialized. When using an OPTIX_BUILD_INPUT_TYPE_INSTANCE_POINTERS build input the application provides a buffer of pointers to AABBs, one per input. (See "Instance build inputs" (page 19).) When none of the instances require an AABB, the entire AABBs buffer is not required and the application may set OptixBuildInputInstanceArray::aabbs to zero.

The sbtOffset is the instance's offset into the SBT array specified with the hitgroupRecordBase parameter of the OptixShaderBindingTable. If the child is a OptixStaticTransform, OptixMatrixMotionTransform or OptixSRTMotionTransform traversable object instead of a GAS then the sbtOffset corresponds to the GAS or IAS traversable at the end of the chain of traversable objects (for example IAS->MMT->ST->GAS). In the case where the eventual instance references a IAS the sbtOffset should be set to zero.

5.3 Build flags

An AS build can be controlled through the OptixBuildFlags. To enable random vertex access on an AS, use OPTIX_BUILD_FLAG_ALLOW_RANDOM_VERTEX_ACCESS. (See "Triangle mesh random access" (page 57).) To steer tradeoffs between build performance, runtime traversal performance and AS memory usage, use OPTIX_BUILD_FLAG_PREFER_FAST_TRACE, OPTIX_BUILD_FLAG_PREFER_FAST_BUILD.

The flags OPTIX_BUILD_FLAG_PREFER_FAST_TRACE and OPTIX_BUILD_FLAG_PREFER_FAST_BUILD are mutually exclusive. Multiple flags that are not mutually exclusive can be ORed together.

5.4 Dynamic updates

Building an AS can be computationally costly. Applications may choose to update an existing acceleration structure using modified vertex data or bounding boxes. Updating an existing AS is generally much faster than rebuilding; however, the quality of the AS may degrade if the data changes too much after an update (e.g. explosions or other chaotic transitions, even if for only parts of the mesh). This may result in slower traversal performance compared to an AS built from scratch from the modified input data.

To allow updates of an AS, 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 AS, 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, 0, nullptr);
```

Note that the update may require more or less temporary memory than the original build.

When updating an existing acceleration structure, only the device pointers and/or their buffer content may be changed. In particular, it is invalid to change the number of build inputs, the build input types, build flags, traversable handles for instances (for IAS), 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 AS will likely degrade substantially.
- During an animation operation, geometry that should be invisible to the camera should not be “removed” from the scene by moving it very far away or by making its geometry degenerate; otherwise, the AS is degraded.

In such cases, it is more efficient to re-build the GAS and/or IAS, or to use the respective masking and flags.

5.5 Relocation

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 compacted ones.

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

In order to relocate an acceleration structure, an `OptixAccelRelocationInfo` object is filled using `optixAccelGetRelocationInfo` and the traversable handle of the source accel. 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 accel may be copied to that device.

If an accel with instances is relocated, the traversable handles for the instances should be specified to `optixAccelRelocate` with the instanceTraversableHandles and numInstanceTraversableHandles parameters. No updating of the bounds is done, so instanceTraversableHandles should correspond to either the same instances as the source (when relocating the IAS without relocating the GAS) or the corresponding relocated instances (if relocating both the GAS and IAS).

Here is an example of relocating the GAS and IAS to new CUDA allocations on the same device:

Listing 5.10

```
CUdeviceptr d_relocatedGas;
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 GAS relocation");
      exit(2);
   }
}
OptixTraversableHandle relocatedGasHandle;
optixAccelRelocate(context, 0, &gasInfo, 0, 0, d_relocatedGas,
   gasBufferSizes.outputSizeInBytes, &relocatedGasHandle);

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

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

CUdeviceptr d_instanceTravHandles;
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(),
```

This is unnecessary because we are copying to the same device, but is here to illustrate the API.

Relocate IAS

```
    sizeof(OptixTraversableHandle) * instanceHandles.size(),
    cudaMemcpyHostToDevice);

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

5.6 Compaction

Compaction of an AS after construction is achieved by a post-process on the existing acceleration structure. This can significantly reduce memory usage, but it requires an additional pass. It is recommended that build and compact operations are performed in batches to ensure that device synchronization does not limit performance. The compacted size depends on the AS type and its properties as well as the device architecture.

To enable compaction, the build flag `OPTIX_BUILD_FLAG_ALLOW_COMPACTION` must be set in the `OptixAccelBuildOptions` as passed to `optixAccelBuild` as well as the emit property `OPTIX_PROPERTY_TYPE_COMPACTED_SIZE`. 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 AS using `optixAccelCompact`. It is recommended to guard the `optixAccelCompact` call by an `if (compactedSize < outputSize)` to avoid the compaction pass in cases where the compaction does not pay off. Note that this check requires to copy the compacted size (as queried with the `optixAccelBuild`) from the device memory to host memory.

A compacted AS 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;

accelOptions.buildFlags = OPTIX_BUILD_FLAG_ALLOW_COMPACTION;

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

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

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

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

The compacted AS does not reference the uncompacted input data. The application is free to reuse the uncompacted AS without invalidating the compacted AS. The copy output must not overlap the input, as it does not work in-place.

A compacted AS supports dynamic updates only if the uncompacted source AS was built with the `OPTIX_BUILD_FLAG_ALLOW_UPDATE` build flag. (See "Dynamic updates" (page 21).) The amount of temporary memory required for a dynamic update is the same for the uncompacted AS and compacted AS.

5.7 Traversables

In addition to GAS traversables the instances in a IAS may also reference transform traversables. These 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.

All traversable objects need to be 64-byte aligned in device memory. 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;
optixConvertPointerToTraversableHandle(
    optixContext, d_transform,
    OPTIX_TRAVERSABLE_TYPE_MATRIX_MOTION_TRANSFORM,
    &transformHandle);

OptixInstance instance;
instance.traversableHandle = transformHandle;

...             Setup instance description
```

5.7.1 Single GAS traversal

The traversable handle passed to `optixTrace` can be a traversable handle created from a GAS. This could be useful for scenes with single GAS objects as the root.

If the modules and pipeline only need to support single GAS traversables it is beneficial to set the `OptixPipelineCompileOptions`::traversableGraphFlags from `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_ANY` to `OPTIX_TRAVERSABLE_GRAPH_FLAG_ALLOW_SINGLE_GAS`.

This signals to OptiX that no other traversable types need to be supported.

5.8 Motion blur

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

There must be at least two motion keys. One exception is the AS build that generally accepts a `OptixAccelBuildOptions` with a nullified `OptixMotionOptions`. This effectively disables motion blur for the AS and ignores the motion begin and end time, along with the motion flags.

Traversables linked together in a scene graph may have a different number of motion keys, for example: a static IAS (zero motion key) linked to a mixture of GAS's with zero or multiple motion keys per GAS.

Motion boundary conditions are specified by using flags. By default, the motion is clamped at the boundaries, meaning it is static and visible. To make the traversable disappear outside the begin and/or end time, set `OPTIX_MOTION_FLAG_START_VANISH` and `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 6.0 programming guide.

5.8.1 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 AABB buffers (for `OptixBuildInputCustomPrimitiveArray` and `OptixBuildInputInstanceArray`) for all motion keys. Both 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;
```

5.8.2 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, depending on the number of motion keys. The struct specifies the header and the first two motion keys for convenience, but more than two keys can be used by computing 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 = (char*)
   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));
```

5.8.3 Motion SRT transform

The behavior of the motion SRT transform `OptixSRTMotionTransform` is similar to the matrix motion transform `OptixMatrixMotionTransform`; but 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 following 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 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 reverse its effect.

To obtain the effective transformation at time t, the elements of the components of S, R, and T will be 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
```

A pipeline contains all of the programs 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.

Programs are first compiled into modules of type `OptixModule`. One or more modules are then used to create an `OptixProgramGroup`. Those program groups are then linked into an `OptixPipeline` to enable them to work together 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. In addition, the size of the log message is reported as an output parameter. This is to detect truncated output. It is not advised to call the function again to get the full output since this could result in unnecessary and lengthy work, and in the case of cache hits different output. If there was an error, the information that would be reported in the log string is also reported via the device context log callback (when provided). The purpose of providing both mechanisms for these create functions is to allow a convenient mechanism for pulling out compilation errors from parallel creation without having to determine which output from the logger corresponded to which API invocation.

Symbols in `OptixModule` objects may be unresolved and contain extern references to variables and __device__ functions. During pipeline creation, these symbols can be resolved using the symbols defined in the pipeline modules. Duplicate symbols will trigger an error.

Most NVIDIA OptiX 7 API functions do not own any significant GPU state. Streaming Assembly (SASS), the executable binary programs in a pipeline, are an exception to this rule due to CUDA implementation details. 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, but is typically on the order of 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. Exercise caution in the number and size of the pipelines wherever possible.

Modules can be destroyed with `optixModuleDestroy`. Module lifetimes need to extend to the lifetimes of ProgramGroups that reference them. Modules may be destroyed after creating the `OptixPipeline` that uses them through the ProgramGroup objects.

6.1 Program input

Programs are specified using the Parallel Thread Execution instruction set (PTX). PTX can be generated by including NVIDIA OptiX 7 device headers in CUDA C code and compiling to PTX using the NVIDIA nvcc offline compiler or nvrtc JIT compiler. For example:

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

If specific architectural instructions are needed beyond the default, use the `nvcc` argument `-arch sm_XY` to enable those instructions. The SM target of the input PTX must be less than or equal to the SM version of the GPU for which the module will be compiled.

Using `-use_fast_math` is also highly recommended.

The NVIDIA OptiX programming model supports a Multiple Instruction, Multiple Data (MIMD) subset of CUDA. Execution must be assumed to be independent of other threads. Thus, shared memory usage and warp-wide or block-wide synchronizations such as barrier are not allowed in the input PTX. Apart from these constraints, all GPU instructions are allowed including math, texture, atomic operations, control flow, and memory loads/stores. 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.

The memory model is consistent only within execution of a single launch index starting at the ray-generation invocation and 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 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:

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 51).)

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 optimization level. `OptixPipelineCompileOptions` controls features of the API such as the usage of custom AH programs, 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;

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

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

6.3 Pipeline launch parameter

Launch-varying parameters or values that must be accessible from any module can be specified via a user defined variable named in `OptixPipelineCompileOptions`. In each module that needs access, this variable should be declared with `extern` or `extern "C"` linkage and the `__constant__` memory specifier. The size of the variable must match across all modules in a pipeline. Note that variables of equal size but differing types may trigger undefined behavior. For example:

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 – 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);
}
```

6.4 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 41).) There are several 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 is designated by its name as part of the `OptixProgramGroupDesc` struct passed to `optixProgramGroupCreate`. Most program groups may feature only a single program, but the HIT_GROUP can designate up to three programs for the CH, AH, and IS 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 reference that module.

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

Listing 6.4

```
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;
optixProgramGroupCreate(
    optixContext,
    &pgDesc,        programDescriptions
    1,              numProgramGroups
    &pgOptions,     programOptions
    logString, sizeof(logString),
    &sphereGroup);  programGroup
```

Multiple program groups of varying kinds can be constructed with a single call to `optixProgramGroupCreate`. The following code demonstrates constructing a raygen and miss program group.

Listing 6.5

```
OptixModule rg, miss;
...              RG and MS 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 obtained, they must be linked into an `OptixPipeline`. This object is then used to invoke a ray generation launch.

When the pipeline is linked, some fixed function components may be selected based on `OptixPipelineLinkOptions` and `OptixPipelineCompileOptions`, which were 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. For example:

Listing 6.6

```
OptixPipeline pipeline;

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.

Note that NVIDIA OptiX 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: direct stack and continuation stack. These sizes can be determined by the compiler for each program group. However, 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. 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 RG, MS, EX, CH, AH and IS programs and the continuation stack requirements resulting from EX programs are calculated internally and do not need to be configured. The direct stack requirements resulting from DC programs, as well as the continuation stack requirements resulting from RG, MS, CH, AH, IS, and CC programs need to be configured. If these are not configured explicitly, an internal default implementation is used. The default implementation is correct (but not necessarily optimal) as long as the maximum depth of call trees of CC and DC programs is at most two. Even in cases where the default implementation is correct, users can provide more precise stack requirements by using their knowledge about 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 programs. To set the stack sizes for a particular pipeline, use `optixPipelineSetStackSize`. Helper functions are available to implement these calculations.

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. We conclude the section with a real world example.

Let cssRG denote the maximum continuation stack size of all RG programs; similarly for MS, CH, AH, IS, and CC 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 CC and DC programs, respectively. Then a simple, conservative approach to compute the three parameters of `optixPipelineSetStackSize` is:

Listing 6.7

```
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 CH or MS programs including the call tree of continuation callables

```
continuationStackSize =
   cssRG
  + cssCCTree
  + maxTraceDepth * cssCHOrMSPlusCCTree
  + cssIS
  + cssAH;
```

This computation can be improved in several ways: For the computation of continuationStackSize we observe that 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 the more complicated, but better estimate:

Listing 6.8

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

The formulas above 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 AH and IS programs count as traversal, where as call sites in RG, MS, and CH count as state.

Listing 6.9

```
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 CC programs. One of them is deep, but the involved CC programs need only a small continuation stack. The other one is shallow, but the involved CC programs needs a quite large continuation stack. Then the estimate of cssCCTree can be improved as follows:

Listing 6.10

```
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.

Lastly, let us consider a simple path tracer with two ray types: camera rays and shadow rays. We assume that there are only RG, MS, and CH programs, and no AH, IS, CC, or DC programs. The camera rays invoke only the miss and closest hit programs MS1 and CH1, respectively. CH1 might trace shadow rays, which invoke only the miss and closes hit programs MS2 and CH2, respectively. That is, the maximum trace depth is 2 and the initial formulas simplify to:

Listing 6.11

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

However, from our call graph structure we know that MS2 or CH2 can only be invoked from CH1. This allows for the following estimation:

Listing 6.12

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

which 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 11) for other options regarding the compilation cache.

Cache entries are typically restricted to only work on the same driver version and GPU type.

7 Shader binding table

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

7.1 SBT records

An SBT *record* consists of a header and a data block. The header content is opaque to the application. It holds information accessed by traversal execution to identify and invoke programs. The data block is not used by NVIDIA OptiX 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).

Use the API function, `optixSbtRecordPackHeader`, and the desired `OptixProgramGroup` object to fill the header of an SBT record. These SBT records must be uploaded to the device prior to an NVIDIA OptiX 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;
...
RayGenSbtRecord rgSBT;
rgSBT.data.color = make_float3(1.0f, 1.0f, 0.0f);
optixSbtRecordPackHeader(raygenPG, &rgSBT);
CUdeviceptr deviceRaygenSbt;
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.

7.2 SBT layout

An SBT is split into five sections,

one for each of the following types of program groups:

Group	*Programs*	*Value of enum* `OptixProgramGroupKind`
Ray generation	RG	`OPTIX_PROGRAM_GROUP_KIND_RAYGEN`
Exception	EX	`OPTIX_PROGRAM_GROUP_KIND_EXCEPTION`
Miss	MS	`OPTIX_PROGRAM_GROUP_KIND_MISS`
Hit	IS, AH, CH	`OPTIX_PROGRAM_GROUP_KIND_HITGROUP`
Callable	DC, CC	`OPTIX_PROGRAM_GROUP_KIND_CALLABLES`

OptiX program groups

(See also "Program group creation" (page 34).)

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

Listing 7.2

```
typedef struct OptixShaderBindingTable
{
    CUdeviceptr raygenRecord;
```
Device address of the SBT record of the ray generation program to start launch

```
    CUdeviceptr  exceptionRecord;
```
Device address of the SBT record of the exception shader

```
    CUdeviceptr  missRecordBase;
    unsigned int missRecordStrideInBytes;
    unsigned int missRecordCount;
```
Arrays of SBT records. The base address, stride in bytes and maximum index are defined.

```
    CUdeviceptr  hitgroupRecordBase;
    unsigned int hitgroupRecordStrideInBytes;
    unsigned int hitgroupRecordCount;

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

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 these sections are not required to be allocated contiguously.

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 this 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 DC when invoking `optixDirectCall` and CC when invoking `optixContinuationCall`.

Any-hit, closest-hit, intersection
: The computation of the index for the hit group (Intersection, Any-Hit, Closes-Hit) is done during traversal and is detailed in the following section.

7.3 Shader binding tables for geometric 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 computation during traversal:

```
sbt-index =
    sbt-istance-offset
    + (sbt-GAS-index * sbt-stride-from-trace-call)
    + sbt-offset-from-trace-call
```

SBT instance offset
: IAS instances (`OptixInstance`) store an SBT offset that is applied during traversal. This is zero for single GAS traversables (see "Single GAS traversal" (page 26)), since there is no corresponding IAS to hold this value. This value is limited to 24 bits (see the declaration of `OptixInstance`::sbtOffset).

SBT GAS index
: Each GAS build input references at least one SBT record. The first SBT GAS index for each GAS build input is the prefix sum of the number of SBT records. Therefore, the computed SBT GAS index is dependent on the order of the build inputs.

  Following is an example of a GAS with three build inputs. Each build input references one SBT record by specifying `numSBTRecords=1`. When intersecting geometry at trace time, the SBT GAS index used to compute the sbtIndex to select the hit group record will be as follows:

SBT GAS index	0	1	2
GAS Build input	Build input[0]		
		Build input[1]	
			Build input[2]

  In this simple example, the index for the build input equals the SBT GAS index. Hence, whenever a primitive from "Build input [1]" is intersected, the SBT GAS 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, given three build inputs with the first build input referencing four SBT records, the second only one SBT record, and the last build input referencing two SBT records. This yields the following SBT GAS indices when intersecting the corresponding GAS build input: An 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 and an index in the range of [5,6] if a primitive from "Build input [2]" is intersected.

SBT GAS index	0	1	2	3	4	5	6
GAS Build input	Build input[0] `numSBTRecords=4`				Build input[1] `numSBTRecords=1`	Build input[2] `numSBTRecords=2`	

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 GAS index as:

SBT GAS 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 GAS build. See "Acceleration structures" (page 15).

SBT trace offset

`optixTrace` takes the parameter SBToffset which allows for an SBT access shift for this specific ray and which is required to implement different ray types.

SBT trace stride

`optixTrace` takes the parameter SBTstride (in index space, not in bytes!) which multiplies the SBT GAS index and which is required to implement different ray types.

7.3.1 Example SBT for a scene

A SBT implements the program selection for a simple scene containing one IAS and two instances of the same GAS, which has two build inputs.

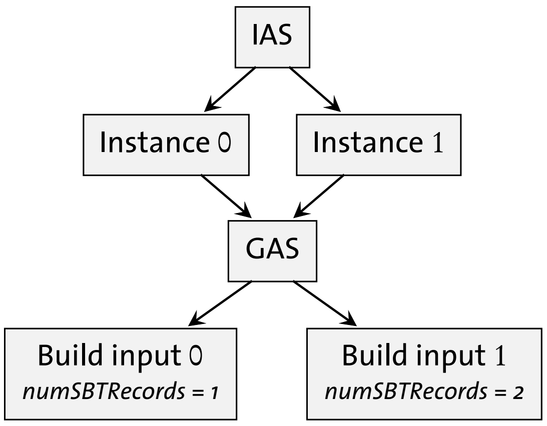

Fig. 7.1 – Structure of a simple scene

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 GAS have different transforms and SBT offsets to allow for material variation in each instance of the same GAS. Hence, the SBT needs to hold two miss records and 12 hit group records (3 for the GAS, $\times 2$ for the ray types, $\times 2$ for the two instances in the IAS).

The SBT looks as follows:

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 type.														
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 GAS 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, ...);
```

Note that program groups of different types (ray generation, miss, intersection, ...) do not need to be next to each other as in the example. The pointer to the first SBT record of each program group type is passed to `optixLaunch`, as described above, 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;   Triagle 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 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, IS, AH, CH, MS and EX programs until the invocations are complete.

A pipeline requires device-side memory for each launch. This space is allocated and managed by the API. The launch resources may be shared between pipelines, so they will only be 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 launch state, it is necessary to provide information about the SBT layout to be launched. 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 "SBT layout" (page 41).)

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`, with these restrictions:

- If the size (specifed 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 33).)

Once the kernel has started, a copy of the memory pointed to by `pipelineParams` will have completed, so the kernel is allowed to modify the values during launch. This allows subsequent launches to run with altered pipeline parameter values. There are no opportunities for users to synchronize on this copy between the invocation of `optixLaunch` and the start of the kernel.

Note: Concurrent launches with different values for `pipelineParams` on the same pipeline will result in serialization of the launches at the moment. The current work around to achieve concurrency is to have a separate pipeline for each concurrent launch.

Finally, launch requires the dimension of the launch. If one-dimensional launches are required, use the width as the dimension of the launch and set both a height and depth of 1. If two-dimensional launches are required, set the width and height as the dimension of the launch and use a depth of 1. For example:

Listing 8.1

```
CUstream stream;
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;

...            Allocate and copy the params to the device

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

9 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:

```
optixGetTransformTypeFromHandle
optixGetInstanceIdFromHandle
optixGetInstanceTransformFromHandle
optixGetInstanceInverseTransformFromHandle
optixGetStaticTransformFromHandle
optixGetMatrixMotionTransformFromHandle
optixGetSRTMotionTransformFromHandle
optixGetTriangleVertexData
optixGetGASMotionTimeBegin
optixGetGASMotionTimeEnd
optixGetGASMotionStepCount
optixGetLaunchIndex
optixGetLaunchDimensions
optixGetSbtDataPointer
```

Other functions are available only in specific program types.

The following table shows in which program types the OptiX device functions are valid.

Device function	*RG*	*IS*	*AH*	*CH*	*MS*	*EX*	*DC*	*CC*
`optixTrace`	✓			✓	✓			✓
`optixSetPayload_0` … `optixSetPayload_7` `optixGetPayload_0` … `optixGetPayload_7`		✓	✓	✓	✓			
`optixGetWorldRayOrigin` `optixGetWorldRayDirection`		✓	✓	✓	✓			
`optixGetObjectRayOrigin` `optixGetObjectRayDirection`		✓	✓					
`optixGetRayTmin` `optixGetRayTmax` `optixGetRayTime` `optixGetRayFlags` `optixGetRayVisibilityMask`		✓	✓	✓	✓			
`optixGetTransformListSize` `optixGetTransformListHandle` `optixGetGASTraversableHandle`		✓	✓	✓	✓	✓		
`optixGetWorldToObjectTransformMatrix` `optixGetObjectToWorldTransformMatrix`		✓	✓	✓	✓			
`optixTransformPointFromWorldToObjectSpace` `optixTransformPointFromObjectToWorldSpace` `optixTransformVectorFromWorldToObjectSpace` `optixTransformVectorFromObjectToWorldSpace` `optixTransformNormalFromWorldToObjectSpace` `optixTransformNormalFromObjectToWorldSpace`		✓	✓	✓	✓			
`optixGetPrimitiveIndex`		✓	✓	✓		✓		
`optixGetInstanceId` `optixGetInstanceIndex`		✓	✓	✓				
`optixGetHitKind` `optixIsTriangleHit` (header function) `optixIsTriangleFrontFaceHit` (header function) `optixIsTriangleBackFaceHit` (header function) `optixGetTriangleBarycentrics`			✓	✓				
`optixReportIntersection`		✓						
`optixGetAttribute_0` … `optixGetAttribute_7`		✓	✓	✓				
`optixTerminateRay` `optixIgnoreIntersection`			✓					
`optixContinuationCall`	✓			✓	✓			✓
`optixDirectCall`	✓	✓	✓	✓	✓		✓	✓
`optixThrowException`	✓	✓	✓	✓	✓		✓	✓
`optixGetExceptionCode` `optixGetExceptionInvalidTraversable` `optixGetExceptionInvalidSbtOffset` `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 if a CH program calls a function called computeValue, which calls computeDeeperValue, which itself calls `optixGetTriangleBarycentrics`, then the body of computeDeeperValue is inlined into computeValue, which in turn, is inlined into the CH 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
```

9.1 Launch index

Available in all programs, it returns the current launch index within the launch dimensions specified by `optixLaunch` on the host.

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

Listing 9.1

```
uint3 optixGetLaunchIndex();
```

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

9.2 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, and the payload is passed to all the IS, AH, CH and MS 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 8 32-bit integer values, which are held in registers when possible. These values may also encode pointers to stack-based variables or application-managed global memory, to hold additional state.

The rayTime argument sets the time of the ray 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` will always return 0.

The visibility mask controls intersection against configurable masks of instances. (See "Instance build inputs" (page 19).) Intersections are computed if there is at least one matching bit in both masks.

The specified miss SBT index is used to identify the program that is invoked on a miss. (See "SBT layout" (page 41).) For example:

Listing 9.2

```
__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);
```

9.3 Payload access

In IS, AH, CH, and MS 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 with names that end in the payload index. For example, payload 0 is set and accessed by these two functions:

Listing 9.3

```
__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 via `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.

9.4 Reporting intersections and attribute access

To report an intersection with the current traversable, the IS program can use the `optixReportIntersection` function. The hitKind of the given intersection is communicated to the associated AH and CH program and allows the AH and CH 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.

There are up to eight

32-bit primitive attribute values available. IS programs can write the attributes when reporting an intersection using `optixReportIntersection`. CH and AH programs are then able to read these attributes. For example:

Listing 9.4

```
__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 AH, an application calls `optixIgnoreIntersection`. The CH program is called for the closest accepted intersection with the attributes reported for that intersection. Note that AH 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 will not be 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.

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 CH or IS shaders when the attributes are consumed. More sophisticated attributes are probably better handled

in the CH program. There are generally better memory bandwidth savings by deferring certain calculations to CH or reloading values once in CH.

9.5 Ray information

To query the properties of the currently active ray, use the following functions. In RG and EX programs, these functions are not supported because there is no currently active ray.

`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 IS or AH) 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 IS and AH.

`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 IS and CH, this is the smallest reported hitT or if no intersection has been recorded yet the tmax that was passed into `optixTrace`.
- In AH, this returns the hitT value as passed into `optixReportIntersection`.
- In MS 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`.

9.6 Undefined values

Advanced application writers seeking more fine-grained control over register usage may want to reduce total register utilization 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 9.5

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

9.7 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 application can query the hitKind reported in `optixReportIntersection` by using `optixGetHitKind`. For triangle meshes, the hitKind is one of:

`OPTIX_HIT_KIND_TRIANGLE_FRONT_FACE`
: The ray intersects the front face of the triangle.

`OPTIX_HIT_KIND_TRIANGLE_BACK_FACE`
: The ray intersects the back face of the triangle.

When traversing a scene with instances (that is, a scene containing IAS objects), two properties of the most recently visited instance can be queried in IS and AH. In the case of CH, 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 IAS'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` and `optixGetInstanceIndex` returns `0`.

9.8 Triangle mesh random access

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

Listing 9.6

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

The function writes the three triangle vertices to data. The sbtGasIdx parameter identifies the GAS local SBT index of this primitive. It is left to the application to obtain the sbtGasIdx for a primitive. For example, the application could store the sbtGasIdx as user data in the SBT record. (See "Shader binding table" (page 41).)

Function `optixGetTriangleVertexData` returns the three triangle vertices at the rayTime passed in. Motion interpolation is performed if motion is enabled on the GAS and pipeline.

The potential decompression step of triangle data may come with significant runtime overhead. Furthermore, enabling random access may cause the GAS to use slightly more memory.

The user can use `optixGetGASTraversableHandle`, `optixGetPrimitiveIndex` and `optixGetRayTime` to obtain the GAS traversable handle, primitive index and motion time associated with an intersection in the CH and AH programs:

Listing 9.7

```
const SbtData* rtData = (const SbtData*)optixGetSbtDataPointer();

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

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

NVIDIA OptiX may remove degenerate (unintersectable) triangles from the AS during construction. Calling `optixGetTriangleVertexData` on a degenerate triangle will not return the original triangle vertices but instead return NaN as triangle data.

9.9 GAS motion options

The function `optixGetTriangleVertexData` performs motion vertex interpolation for triangle position data. The user may also want to perform motion interpolation on other user-managed vertex data, such as interpolating vertices in a custom motion IS, or interpolating user-provided shading normals in the CH shader. To facilitate this, NVIDIA OptiX provides the following functions to obtain the motion options for a GAS:

```
optixGetGASMotionTimeBegin
optixGetGASMotionTimeEnd
optixGetGASMotionStepCount
```

For example, if the number of motion keys for the user vertex data equals the number of motion keys in the GAS, the user can compute the left key index and intra-key interpolation time with the following:

Listing 9.8

```
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;
```

9.10 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 a IAS. Function `optixGetInstanceIdFromHandle` returns the instance user ID, whereas `optixGetInstanceTransformFromHandle` and `optixGetInstanceInverseTransformFromHandle` return the instance transform and its inverse.

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

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

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

These pointers should be used only to read data associated with these nodes. Writing data to any traversables active during a launch produces undefined results. For example:

Listing 9.9 – 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

The application can implement a generic transformation evaluation function using these intrinsics, as outlined earlier in this section, or write a specialized version 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`.

In addition, the handles passed back from `OptixTraversableHandle` can be stored or passed into other functions which can decode them.

9.11 Terminating or ignoring traversal

In AH programs, two functions can be used 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 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.

9.12 Exceptions

Exceptions allow NVIDIA OptiX to check for invariants and, if violated, to report details about the violation.

To enable exception checks, set the `OptixPipelineCompileOptions`::exceptionFlags field with a bitwise combination of `OptixExceptionFlags`.

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`.

`OPTIX_EXCEPTION_FLAG_TRACE_DEPTH`
: Checks to see if the ray depth exceeds the value specified with `OptixPipelineLinkOptions`::maxTraceDepth before tracing a new ray. Some case of stack overflow are only detected if the exception for trace depth is enabled as well (or the value of `OptixPipelineLinkOptions`::maxTraceDepth is correct).

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

`OPTIX_EXCEPTION_FLAG_DEBUG`
: Enables a number of run time checks including checking for overflows for the traversable list and validating the set of traversables encountered at runtime.

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 will be provided by OptiX. This built-in exception program will only print the first five exceptions to stdout. 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 occured can be queried with:

Listing 9.10

```
int optixGetExceptionCode();
```

Six exception codes are defined for the builtin 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 has been exceeded. No information functions.

`OPTIX_EXCEPTION_CODE_TRAVERSAL_DEPTH_EXCEEDED`
: The traversal depth is exceeded. Two information functions:

    `optixGetTransformListSize`()

    `optixGetTransformListHandle`()

`OPTIX_EXCEPTION_CODE_TRAVERSAL_INVALID_TRAVERSABLE`
: Traversal encountered an invalid traversable type. Three information functions:

    `optixGetTransformListSize`()

    `optixGetTransformListHandle`()

    `optixGetExceptionInvalidTraversable`()

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

`optixGetExceptionInvalidSbtOffset()`

`OPTIX_EXCEPTION_CODE_TRAVERSAL_INVALID_HIT_SBT`
: The traversal hit sbt record index out of bounds. Four information functions:

  `optixGetTransformListSize()`
  `optixGetTransformListHandle()`
  `optixGetExceptionInvalidSbtOffset()`
  `optixGetPrimitiveIndex()`

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 9.11 – 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()`.

Behavior is undefined if using an exception detail query for another exception code, or if user exception detail is queried that was not set with `optixThrowException`.

10 Callables

Two types of callable programs exist in NVIDIA OptiX 7: *direct callables* (DC) and *continuation callables* (CC). Direct callables are called immediately, similar to a regular CUDA function call, while continuation callables are executed by the scheduler. Thus, a continuation callable can feature more overhead when executed.

Continuation callable programs allow the use of `optixTrace`, while direct callable programs do not. Since CC programs can only be called from other CC 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.

On the other hand, the usage of continuation callables can 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.

While direct callables can be called from any program type except EX, continuation callables cannot be called from IS, AH or DC.

The two device functions relevant for calling a callable are `optixDirectCall` and `optixContinuationCall` (note the variadic template requires C++11 device code compilation):

Listing 10.1 – 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 sbtIndex argument is an index of the array of callable SBT entries specified with the `OptixShaderBindingTable`::callablesRecordBase parameter to `optixTrace`. The program group for a callable can contain both a DC and CC program, though these programs share a single record in the SBT.

Note that it is also possible to use non-inlined functions in order to keep compilation time low for large (such as complex noise) functions that are called from multiple sections of the code base. This additional mechanism can be useful in case a callable turns out to be too costly (longer compilation time and/or lower runtime performance -> Revisit as soon as Michael’s fix is in to address spills). It also avoids needing to add callables to the SBT and will also avoid that the function can be scheduled during runtime execution (which might be preferable in some cases).

11 NVIDIA AI Denoiser

Image areas that have not yet fully converged during rendering will often exhibit pixel-scale grainy noise due to the insufficient amount of information gathered by the renderer (for example, low iteration counts and/or a scene featuring complex lighting and materials).

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

The denoiser comes with built-in pre-trained models. These models, represented by a binary blob called training model, is the result of training the underlying Deep Learning system with a large group of rendered images in different stages of convergence. Since training needs significant computational resources and even obtaining a sufficient amount of such image pairs can be difficult, a general-purpose model is included. The provided models are suitable for many renderers in practice, but might not always lead to optimal results when applied to images produced by renderers with very different noise characteristics compared to those that were present in the original training data.

In addition note that any custom post-processing done on the noisy image, like image filters (for example, blur, sharpen) or reconstruction filters (for example, box, triangle, gauss) can lead to unsatisfactory results, as the original high-frequency/per-pixel noise might have been smeared across multiple pixels and is thus more difficult to detect and handle by the model. Thus, any post-processing operations should be done after the denoising process, while reconstruction filters should be implemented by using filter importance sampling instead.

One can also create a custom model by training the denoiser with a self-provided set of images and use the resulting training model in the NVIDIA AI Denoiser, but this process is not part of NVIDIA OptiX itself. To learn how to generate training data based on your renderer's images one can attend the course Rendered Image Denoising using Autoencoders, which is part of the NVIDIA Deep Learning Institute.

In general, the pixel color space of an image that is used as input for the denoiser should match the one of the images it was trained on, although slight variations such as substituting sRGB with a simple gamma curve should not have a noticeable impact. The images used for the training model included with the NVIDIA AI Denoiser distribution were output in tone mapped LDR, using a gamma value of 2.2, or directly in HDR data, respectively.

To denoise an existing image, use `optixDenoiserCreate` to create a denoiser object along with `optixDenoiserSetModel` to select the training model that will be used to denoise. After denoising, use `optixDenoiserDestroy` to destroy the object along with associated host resources.

Listing 11.1

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

`OptixDenoiserOptions` specifies which `OptixDenoiserInputKind` will be used later:

```
OPTIX_DENOISER_INPUT_RGB
OPTIX_DENOISER_INPUT_RGB_ALBEDO
OPTIX_DENOISER_INPUT_RGB_ALBEDO_NORMAL
```

where the input RGB, Albedo and normal buffers are of type `OptixPixelFormat`:

```
OPTIX_PIXEL_FORMAT_HALF3
OPTIX_PIXEL_FORMAT_HALF4
OPTIX_PIXEL_FORMAT_FLOAT3
OPTIX_PIXEL_FORMAT_FLOAT4
OPTIX_PIXEL_FORMAT_UCHAR3
OPTIX_PIXEL_FORMAT_UCHAR4
```

The RGB buffer will contain a noisy image that is to be denoised. The optional fourth (alpha) channel of the image can be configured to not be changed by the denoiser if needed. 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 builtin model. Note that no tone-mapping or gamma correction must be done for HDR data. Using a pre-process pass that corrects drastic under- or over-exposure along with clipping/filtering of fireflies (that is, single, extremely bright, non-converged pixels) on the HDR image beforehand can improve the denoising quality dramatically though.

The optional, noise-free normal buffer is expected to contain the surface normals of the primary hit in camera space. 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 be helpful in some situations where a lot of fine geometric detail and/or high resolution normal/bump maps are used.

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.

Note that specifying albedo can dramatically improve denoising quality, especially for very noisy input images.

To select a training model for the denoiser, use:

Listing 11.2

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

with `OptixDenoiserModelKind` being one of:

`OPTIX_DENOISER_MODEL_KIND_LDR`
`OPTIX_DENOISER_MODEL_KIND_HDR`
`OPTIX_DENOISER_MODEL_KIND_USER`

Note that when selecting `OPTIX_DENOISER_MODEL_KIND_USER`, one must specify data and sizeInBytes.

To be able to allocate the required, temporary device memory to run the denoiser, use:

Listing 11.3

```
OptixResult optixDenoiserComputeMemoryResources(
   const OptixDenoiser handle,
   unsigned int maximumOutputWidth,
   unsigned int maximumOutputHeight,
   OptixDenoiserSizes* returnSizes);

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

Then pass the denoiser state device memory allocated with a size of stateSizeInBytes to:

Listing 11.4

```
OptixResult optixDenoiserSetup(
   OptixDenoiser handle,
   CUstream      stream,
   CUdeviceptr   denoiserState,
   size_t        denoiserStateSizeInBytes);
```

To then actually execute the denoising on a given image, use:

Listing 11.5

```
OptixResult optixDenoiserInvoke(
   OptixDenoiser                handle,
   CUstream                     stream,
   const OptixDenoiserParams*   params,
   CUdeviceptr                  denoiserState,
   Size_t                       denoiserStateSizeInBytes,
   const \texttt{OptixImage2D}* inputLayers,
   unsigned int                 numInputLayers,
   const \texttt{OptixImage2D}* outputLayer,
   CUdeviceptr                  scratch,
   Size_t                       scratchSizeInBytes);

typedef struct \texttt{OptixImage2D}
{
   unsigned int     width, height;
   unsigned int     strideInBytes;
   OptixPixelFormat format;
} \texttt{OptixImage2D};

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

The scratch memory must be exclusively available to the denoiser during the execution.

Note that `OptixPixelFormat` of the inputLayers must match what was specified during `optixDenoiserCreate`, whereas outputLayer can be of a different format.

Struct field denoiseAlpha can be used to disable the denoising of the (optional) alpha channel of the noisy image, and blendFactor to interpolate between the noisy input image (1.0) and the denoised output image (0.0).

To fill in hdrIntensity, it must be either calculated during a custom application-side pre-processing pass or by using:

Listing 11.6

```
OptixResult optixDenoiserComputeIntensity(
   OptixDenoiser       handle,
   CUstream            stream,
   const \texttt{OptixImage2D}* inputImage,
   CUdeviceptr         outputIntensity);
```

where parameter outputIntensity is a pointer to a single float value.