DirectX-Specs

DirectX Raytracing (DXR) Functional Spec

v1.17 10/25/2021

Contents

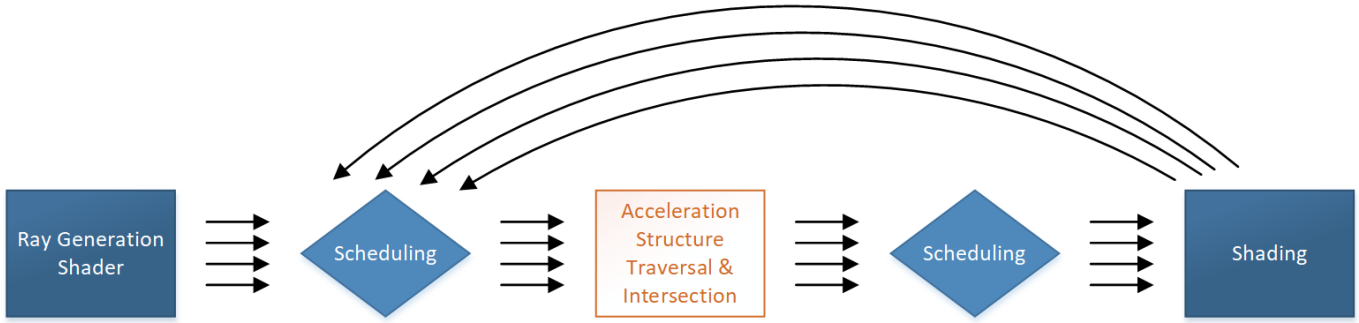
Intro

This document describes raytracing support in D3D12 as a first class peer to compute and graphics (rasterization). Similar to the rasterization pipeline, the raytracing pipeline strikes a balance between programmability, to maximize expressiveness for applications; and fixed function, to maximize the opportunity for implementations to execute workloads efficiently.

Overview

The system is designed to allow implementations to process rays independently. This includes the various types of shaders (to be described), which can only ever see a single input ray and cannot see or depend on the order of processing of other rays in flight. Some shader types can generate multiple rays over the course of a given invocation, and if desired look at the result of a ray's processing. Regardless, generated rays that are in-flight can never be dependent on each other.

This ray independence opens up the possibility of parallelism. To exploit this during execution, a typical implementation would balance between scheduling and other tasks.



(The above diagram is only a loose approximation of what an implementation might do – don't read it too deeply.)

The scheduling portions of execution are hard-wired, or at least implemented in an opaque way that can be customized for the hardware. This would typically employ strategies like sorting work to maximize coherence across threads. From an API point of view, ray scheduling is built-in functionality.

The other tasks in raytracing are a combination of fixed function and fully or partially programmable work:

The largest fixed function task is traversing acceleration structures that have been built out of geometry provided by the application, with the goal of efficiently finding potential ray intersections. Triangle intersection is also supported in fixed function.

Shaders expose application programmability in several areas:

- generating rays
- determining intersections for implicit geometry (as opposed to the fixed function triangle intersection option)
- processing ray intersections (such as surface shading) or misses

The application also has a high level of control over exactly which out of a pool of shaders to run in any given situation, as well as flexibility in the resources such as textures that each shader invocation has access to.

Design goals

- Implementation agnostic
    - Support for hardware with or without dedicated raytracing acceleration via single programming model

    - Expected variances in hardware capability are captured in a clean feature progression, if necessary at all

- Embrace relevant D3D12 paradigms
    - Applications have explicit control of shader compilation, memory resources and overall synchronization
    - Applications can tightly integrate raytracing with compute and graphics
    - Incrementally adoptable
- Friendly to tools such as PIX
    - Running tools such as API capture / playback don't incur unnecessary overhead to support raytracing

Walkthrough

The following walkthrough broadly covers most components of this feature. Further details will be described later in the document, including dedicated sections listing APIs and HLSL details.

Initiating raytracing

First a pipeline state containing raytracing shaders must be set on a command list, via SetPipelineState1().

Then, just as rasterization is invoked by Draw() and compute is invoked via Dispatch(), raytracing is invoked via DispatchRays(). DispatchRays() can be called from graphics command lists, compute command lists or bundles.

Tier 1.1 implementations also support GPU initiated DispatchRays() via ExecuteIndirect().

Tier 1.1 implementations also support a variant of raytracing that can be invoked from any shader stage (including compute and graphics shaders), but does not involve any other shaders - instead processing happens logically inline with the calling shader. See Inline raytracing.

Ray generation shaders

DispatchRays() invokes a grid of ray generation shader invocations. Each invocation (thread) of a ray generation shader knows its location in the overall grid, can generate arbitrary rays via

TraceRay(), and operates independently of other invocations. So there is no defined order of execution of threads with respect to each other.

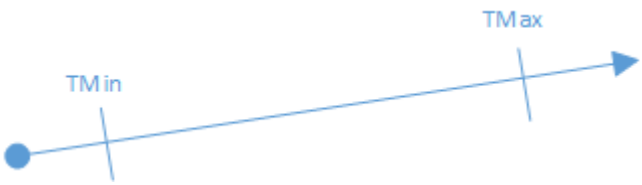
HLSL details are here.

Rays

A ray is: an origin, direction and parametric interval (TMin, TMax) in which intersections may occur at T locations along the interval. To be concrete, positions along the ray are: origin + T*direction (the direction does not get normalized).

There is some nuance to the exact bounds used, (TMin..TMax) vs [TMin...TMax] for intersections to count, depending on the geometry type, defined in ray extents.

A ray is accompanied by a user defined payload that is modifiable as the ray interacts with geometry in a scene and also visible to the caller of TraceRay() upon its return. In the case of the Inline raytracing variation, the payload isn't an explicit entity, rather it is just a part of whatever user variables the caller of RayQuery::TraceRayInline() has in its execution scope.



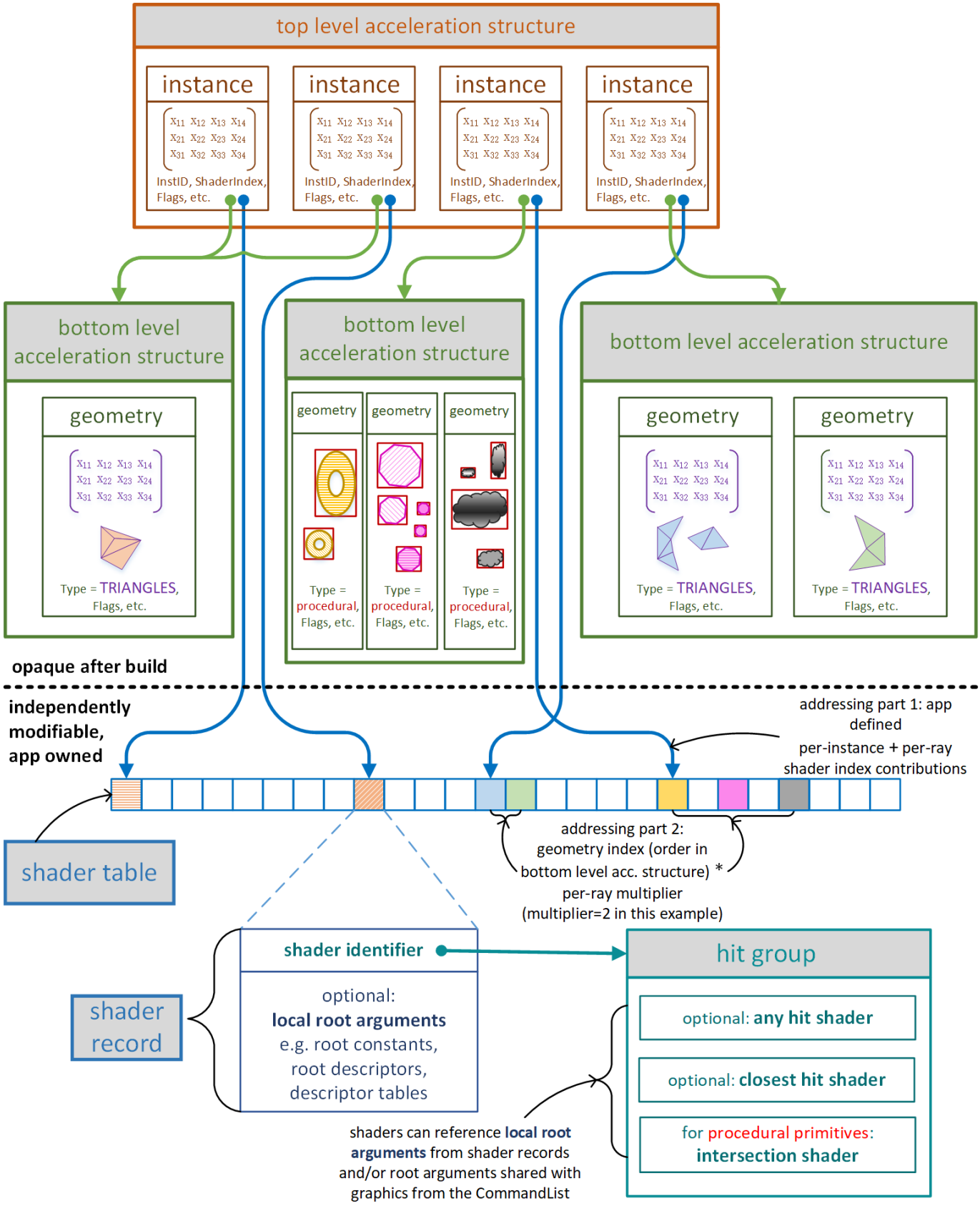
The TMin value tracked by the system never changes over the lifetime of a ray. On the other hand, as intersections are discovered (in arbitrary spatial order), the system reduces TMax to reflect the closest intersection so far. When all intersections are complete, TMax represents the closest intersection, the relevance of which appears later.

Raytracing output

In raytracing, shaders output results, such as color samples for an image, manually through UAVs.

Ray-geometry interaction diagram

Upcoming sections describe this picture, plus concepts not shown that aren't specific to geometry, like "miss shaders".



Geometry and acceleration structures

Geometry for a scene is described to the system using two levels of acceleration structures: Bottom-level acceleration structures each consist of a set of geometries that are building blocks

for a scene. A top-level acceleration structure represents a set of instances of bottom-level acceleration structures.

Within a given bottom-level acceleration structure there can be any number: (1) triangle meshes, or (2) procedural primitives initially described only by an axis aligned bounding box (AABB). A bottom-level acceleration structure can only contain a single geometry type. These geometry types are described more later. Given a definition of a set of these geometries (via array of D3D12_RAYTRACING_GEOMETRY_DESC), the application calls BuildRaytracingAccelerationStructure() on a CommandList to ask the system to build an opaque acceleration structure representing it into GPU memory owned by the application. This acceleration structure is what the system will use to intersect rays with the geometry.

Given a set of bottom-level acceleration structures, the application then defines a set of instances (by pointing to D3D12_RAYTRACING_INSTANCE_DESC structures living in GPU memory). Each instance points to a bottom-level acceleration structure and includes some other information for specializing the instance. A couple of examples of the specializing information included in an instance definition are: a matrix transform (to place the instance in the world), and a user defined InstanceID (identifying the unique instance to shaders).

Instances are sometimes referred to in this specification as geometry instances for clarity.

This set of geometry instance definitions is given to the implementation (via BuildRaytracingAccelerationStructure() to generate an opaque top-level acceleration structure into GPU memory owned by the application. This acceleration structure represents what the system traces rays against.

An application can use multiple top-level acceleration structures simultaneously, binding them to relevant shaders as input resources (see RaytracingAccelerationStructure in HLSL). That way a given shader can trace rays into different sets of geometry if desired.

> The two level hierarchy for geometry lets applications strike a balance between intersection performance (maximized by using larger bottom-level acceleration structures) and flexibility (maximized by using more, smaller bottom-level acceleration structures and more instances in a top-level acceleration structure).

See Acceleration structure properties for a discussion of rules and determinism.

Acceleration structure updates

Apps can request to make an acceleration structure updateable, or request an update to an updateable acceleration structure, via D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS in BuildRaytracingAccelerationStructure().

The updateable acceleration structures (before and after they have been updated) will not be as optimal in terms of raytracing performance as building a static acceleration structure from scratch. An update will be faster, however, than building an acceleration structure from scratch.

For updates to be viable to implement, constraints are in place on what an app is allowed to change. For instance, with triangle geometry in bottom-level acceleration structures only vertex positions can be updated. There is much more freedom of update allowed for instance descriptions in top-level acceleration structures. For more detail see Acceleration structure update constraints.

Built-in ray-triangle intersection - triangle mesh geometry

As mentioned above, geometry in a bottom-level acceleration structure can be represented as triangle meshes which use built-in ray-triangle intersection support that passes triangle barycentrics describing the intersection to subsequent shaders.

Intersection shaders - procedural primitive geometry

An alternative representation for geometry in a bottom-level acceleration structure is an axis aligned bounding box which contains a procedural primitive. The surface is defined by running an application defined intersection shader to evaluate intersections when a ray hits the bounding box. The shader defines the attributes describing intersections to pass on to subsequent shaders, including the current T value.

Using intersection shaders instead of the build-in ray-triangle intersection is less efficient but offers far more flexibility.

HLSL details are here.

Minor intersection shader details

Intersection shaders may be executed redundantly. There is no guarantee that for a given ray that the intersection shader only executes once for a given procedural primitive encountered in the acceleration structure. Multiple invocations for a given ray and primitive would be redundant (wasteful), yet implementations are free to have this behavior if the implementation believes the tradeoff is worth it for some reason. The implication of this is apps must be careful about authoring side effects into intersection shaders, such as doing UAV writes from them or in particular finding different intersections each invocation. The results may differ depending on the implementation.

Regardless if multiple invocations of intersection shaders occur for a given ray, the implementation must always honor the app's choice of flags on the geometry, which may include `D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION`. With this flag, the any hit shader (described next) must only execute once for a given intersection on a given ray.

Any hit shaders

A unique shader can be defined to run whenever a ray intersects a geometry instance within the current ray extents, regardless of position along the ray relative to other intersections. This is an any hit shader.

Any hit shaders can read intersection attributes, modify ray payload, indicate a hit should be ignored (IgnoreHit()), accept the hit and continue (by exiting execution) or accept the hit and tell the system to stop searching for more intersections (AcceptHitAndEndSearch()).

There is no defined order of execution of any hit shaders for the intersections along a ray path. If an any hit shader accepts a hit, its T value becomes the new TMax. So depending on the order that intersections are found all else being equal, different numbers of any hit shader invocations would occur.

The system cannot execute multiple any hit shaders for a given ray at the same time - as such, any hit shaders can freely modify their ray payload without worrying about conflicting with other shaders.

Any hit shaders are useful, for instance, when geometry has transparency. A particular case is transparency in shadow determination, where if the any hit shader finds that the current hit location is opaque it can tell the system to take this hit but stop searching for more intersections (just looking for anything in a ray's path). In many cases though, any hit shaders are not needed, yielding some execution efficiency: In the absence of an any hit shader for a given geometry instance that has an intersection T within the current ray interval, the implementation simply accepts the intersection and reduces TMax of the current ray interval to T.

Unlike some of the other shader types to be described, any hit shaders cannot trace new rays, as doing so here would lead to an unreasonable explosion of work for the system.

HLSL details are here.

Closest hit shaders

A unique shader can be defined to run for each geometry in an instance if it produced the closest accepted intersection within the ray extents. This is a closest hit shader.

Closest hit shaders can read intersection attributes, modify ray payload, and generate additional rays.

> A typical use of a closest hit shader would be to evaluate the color of a surface and either contribute to the ray payload or store data to memory (via UAV).

Any hit shaders (if any) along a ray's path are all executed before a closest hit shader (if any). In particular, if both shader types are defined for the geometry instance at the closest hit's T value, the any hit shader will always run before the closest hit shader.

HLSL details are here.

Miss shaders

For rays that do not intersect any geometry, a miss shader can be specified. Miss shaders can modify ray payload and generate additional rays. Since there was no intersection, there are no intersection attributes available.

HLSL details are here.

Hit groups

A hit group is one or more shaders consisting of: {0 or 1 intersection shader, 0 or 1 any hit shader, 0 or 1 closest hit shader}. Individual geometries in a given instance each refer to a hit group to provide their shader code. The point of the grouping is to allow implementations to be able to compile and execute the group efficiently as rays interact with geometry.

Ray generation shaders and miss shaders aren't part of hit groups because they aren't involved directly with geometry.

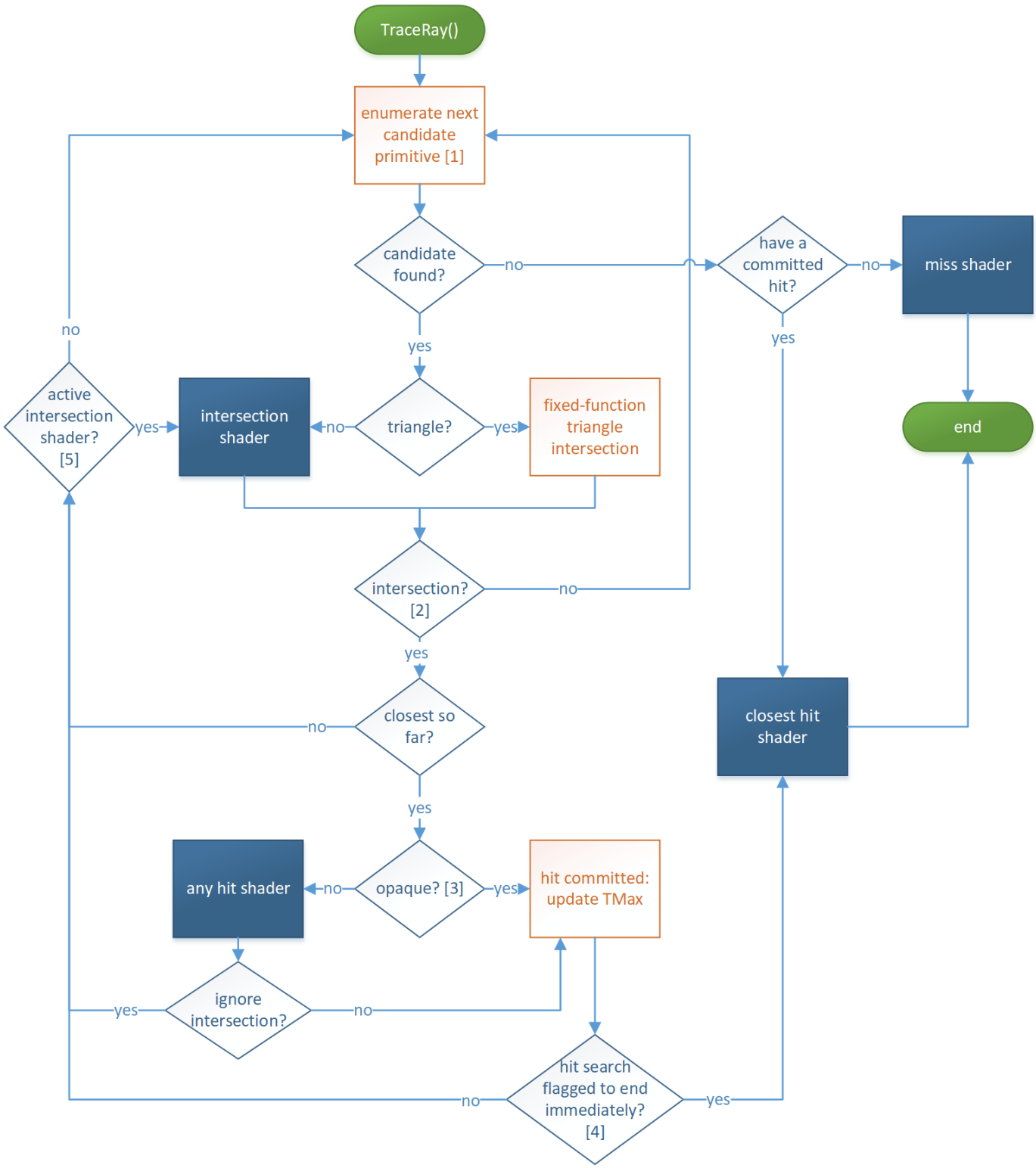
If a hit group contains an intersection shader, it can only be used with procedural primitive geometry. If a hit group does not contain an intersection shader, it can only be used with triangle geometry.

A hit group with no shaders at all is also possible, by simply using NULL as it's shader identifier (concept described later). This counts as opaque geometry.

> An empty hit group can be useful, for example, if the app doesn't want to do anything for hits and only cares about the miss shader running when nothing has been hit.

TraceRay control flow

This is what happens when a shader calls TraceRay():



[1] This stage searches acceleration structures to enumerate primitives that may intersect the ray, conservatively: If a primitive is intersected by the ray and is within the current ray extents, it is guaranteed to be enumerated eventually. If a primitive is not intersected by the ray or is outside the current ray extents, it may or may not be enumerated. Note that TMax is updated when a hit is committed.

[2] If the intersection shader is running and calls ReportHit(), the subsequent logic handles the intersection and then returns to the intersection shader via [5].

[3] Opaqueness is determined by examining the geometry and instance flags of the intersection as well as the ray flags. Also if there is no any hit shader, the geometry is considered opaque.

[4] The search for hits is ended at this point if either the `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` ray flag is set, or if the any hit shader called AcceptHitAndEndSearch(), which aborts the execution of the any hit shader at the AcceptHitAndEndSearch() call site. Since at least this hit was committed, whichever hit is closest so far has the closest hit shader run on it, if present (and not disabled via `RAY_FLAG_SKIP_CLOSEST_HIT_SHADER`).

[5] If the primitive that was intersected was not a triangle, an intersection shader is still active and resumes execution, given it may contain more calls to ReportHit().

Flags per ray

TraceRay() supports a selection of ray flags to override transparency, culling, and early-out behavior.

> To illustrate the utility of ray flags, consider how they would help implement one of multiple approaches to rendering shadows. Suppose an app wants to trace rays to distant light sources to accumulate light contributions for rays that don't hit any geometry, using tail recursion.
>
> TraceRay() could be called with `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH | RAY_FLAG_SKIP_CLOSEST_HIT_SHADER` flags from the ray generation shader, followed by exiting the shader withn nothing else to do. Any hit shaders, if present on geometry, would execute to determine transparency, though these shader invocations could be skipped if desired by also including `RAY_FLAG_FORCE_OPAQUE`.
>
> If any geometry hit is encountered (not necessarily the closest hit), ray processing stops, due to `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH`. A hit has been committed/found, but there is no closest hit shader invocation, due to `RAY_FLAG_SKIP_CLOSEST_HIT_SHADER`. So processing of the ray ends with no action.
>
> Rays that don't hit anything cause the miss shader to run, where light contribution is evaluated and written to a UAV. So in this scenario, geometry in the acceleration structure acted to cull miss shader invocations, ignoring every other type of shader (unless needed for transparency evaluation).
>
> Skipping shaders can alternatively be accomplished by setting shader bindings to NULL (shader bindings details are discussed later on). But the use of ray flags in this example means the implementation doesn't even have to look up shader bindings (only to find that they are NULL). Which also means the app doesn't have to bother configuring NULL bindings anywhere.

Instance masking

Geometry instances in top-level acceleration structures each contain an 8-bit user defined InstanceMask. TraceRay() has an 8-bit input parameter InstanceInclusionMask which gets ANDed with the InstanceMask from any geometry instance that is a candidate for intersection. If the result of the AND is zero, the intersection is ignored.

> This feature allows apps to represent different subsets of geometry within a single acceleration structure as opposed to having to build separate acceleration structures for each subset. The app can choose how to trade traversal performance versus overhead for maintaining multiple acceleration structures.
>
> An example would be culling objects that an app doesn't want to contribute to a shadow determination but otherwise remain visible.
>
> Another way to look at this is:
>
> The bits in InstanceMask define which "groups" an instance belongs to. (If it is set to zero the instance will always be rejected!)
>
> The bits in the ray's InstanceInclusionMask define which groups to include during traversal.

Callable shaders

Callable shaders are meant to assist with pathological shader permutations or shader networks, at the potential expense of some execution efficiency.

Callable shaders are defined through a shader table, described later, but basically a user defined function table. The table is identified by providing a GPU virtual address (CallableShaderTable in D3D12_DISPATCH_RAYS_DESC) to DispatchRays() calls. The contents of the table contain shader identifiers retrieved from state objects (described later) via GetShaderIdentifier().

A given callable shader is called (via CallShader() in HLSL) by indexing into the shader table to pick which callable shader to call from any of the raytracing shaders. A shader invocation making a call just produces one invocation of a callable shader, like a subroutine call with arbitrary in/out parameters. So when the call returns, the caller continues as would be expected. Callable shaders are separately compiled from other shaders, so compilers can't make any assumptions about caller/callee other than the agreed on function signature. The implementation chooses how to make use of a stack of user defined maximum size to store parameters (that it didn't decide to pass via registers) and/or live state – see Pipeline stack.

Implementations are expected to schedule callable shaders for execution separately from the calling shader, as opposed to the code being optimally inlined with the caller. This is similar to

the way tracing rays causes other shaders to run. So using this feature to execute a tiny program may not be worth the minimum overhead of scheduling the shader to run.

> In the absence of callable shaders as a feature, applications could achieve the same result by tracing rays with a NULL acceleration structure, which causes a miss shader to run, repurposing the ray payload and potentially the ray itself as function parameters. Except doing this miss shader hack would be wasteful in terms of defining a ray that is guaranteed to miss for no reason. Rather than supporting this hack, callable shaders are seen as a cleaner equivalent.
>
> The bottom line is implementations should not have difficulty supporting callable shaders given the system has to support miss shaders anyway. At the same time, apps must not expect execution efficiency that would greatly exceed that of invoking a miss shader from a raytrace (minus the actual ray processing overhead).

Resource binding

Since rays can go anywhere, in raytracing not only must all shaders for a scene be simultaneously available to execute, but also their resource bindings. In fact, the selection of what shader to run (by shader Identifier, described later) is considered just another resource binding along with traditional root signature bindings: descriptor tables, root descriptors and root constants.

Descriptor heaps set on CommandLists via SetDescriptorHeaps() are shared by raytracing, graphics and compute.

Local root signatures vs global root signatures

For raytracing shaders, bindings can be defined by one or both of the following root signatures:

- A *local* root signature, whose arguments come from shader tables, described later, enabling each shader to have unique arguments.
- A *global* root signature whose arguments are shared across all raytracing shaders and compute PSOs on CommandLists, set via SetComputeRootSignature() (or equivalent indirect state setting API if it ever exists).

Each raytracing shader used together can use different local root signatures but must use the same global root signature. The “global” root signature identical to the root signature used for compute state on command lists.

Different sets of shaders collected together in a State object (described later), may have different global root signatures, as long as during a DispatchRays() call (or equivalent indirect API if it ever

exists) any shaders that get invoked use the same global root signature that is set on the CommandList as described above.

Unlike global root signatures, local root signatures have a larger limit on the number of entries they can hold, bounded by the maximum supported shader record stride of 4096 bytes, minus 32 bytes for shader identifier size = 4064 bytes max local root signature footprint. Shader identifiers and shader records are described further below.

The shader "register" bindings (e.g. t0, u0 etc.) specified by a local root signature can't overlap with those in a global root signature for a given shader.

Note about static samplers: Local root signatures can define static samplers (just like global root signatures can), except that each local root signature used in a Raytracing pipeline state object (described later) must define any static samplers it uses identically as other local root signatures that define the same ones. So if any local root signature makes a definition for, say, sampler s0, all local root signatures that define s0 must use the same definition. Further, the total number of unique static samplers across local root signature and global root signature must fit in the static sampler limit for D3D12's resource binding model.

> The reason that local root signatures must not have any conflicting static sampler definitions is to enable shaders to be compiled individually on implementations that have to emulate static samplers using descriptor heaps. Such implementations can pick a fixed location in a sampler descriptor heap to place a static sampler, knowing that other shaders that might use a different local root signature and define the same sampler will use the same slot. Static samplers in the global root signature can also be handled the same way (given that as mentioned above, register bindings can't overlap across global and local root signatures.

There is a discussion on shader visibility flags in root signatures here.

Shader identifier

A shader identifier is an opaque data blob of 32 bytes that uniquely identifies (within the current device / process) one of the raytracing shaders: ray generation shader, hit group, miss shader, callable shader. The application can request the shader identifier for any of these shaders from the system. It can be thought of as a pointer to a shader.

If the raytracing process encounters a NULL shader identifier from an app when looking for a shader to run, no shader is executed for that purpose, and the raytracing process continues. In the case of a hit group, a NULL shader identifier simply means no shader is executed for any of the types of shaders it contains.

An application might create the same shader multiple times. This could be the same code but with same or different export names, potentially across separate raytracing pipelines or

collections of code (described later). In this case the seemingly identical shaders may or may not return the same identifier depending on the implementation. Regardless, execution behavior will be consistent with the specified shader code.

Shader record

```
shader record = {shader identifier, local root arguments for the shader}
```

A shader record simply refers to a region of memory owned by an application in the above layout. Since an application can retrieve a shader identifier for any raytracing shader, it can create shader records any way it wants, anywhere it wants. If a shader uses a local root signature, its shader record contains the arguments for that root signature. The maximum stride for shader records is 4096 bytes.

Shader tables

```
shader table = {shader record A}, {shader record B} ...
```

A shader table is a set of shader records in a contiguous region of memory. The start address must be aligned to 64 bytes (D3D12_RAYTRACING_SHADER_TABLE_BYTE_ALIGNMENT).

Raytracing indexes in to shader tables (in various ways) to enable running unique shaders and resource bindings for all the different parts of a scene. Only the particular shader records that will be accessed need to be validly populated.

There is no API object for a shader table; the app merely identifies a region in memory as being a shader table. Rather, the parameters to DispatchRays() include pointers to memory that let apps identify (among other things) the following types of shader tables:

- ray generation shader (single entry since only one shader record is needed)
- hit groups
- miss shaders
- callable shaders

Indexing into shader tables

The location in shader tables to find the appropriate shaders to use at a given geometry intersection is computed as the sum of various offsets provided by the application at different places, for flexibility.

Details are provided in Addressing calculations within shader tables, but basically the process starts at DispatchRays(), which provides base addresses and record strides for shader tables. Then each geometry and each geometry instance definition in a raytracing acceleration structure contribute values to the indexing. And the final contributions are provided by TraceRay() calls within shaders allow further differentiation of which shaders and arguments (bindings) to use with a given geometry instance, without having to change the geometries / instances or acceleration structures themselves.

Shader record stride

The application indicates a data stride it wants the system to use for records as a parameter to DispatchRays(). All shader table indexing arithmetic is done as multiples of this record stride. It can be any multiple of 32 bytes (D3D12_RAYTRACING_SHADER_RECORD_BYTE_ALIGNMENT) of size [0...4096] bytes.

If the stride is nonzero, the stride must be at least as large as the largest shader record. So there is some unused memory between shader records when they are smaller than the stride.

If the stride is 0, all indexing points to the same shader record. This is unlikely to be interesting, particularly given this would cause the local root signature to behave in a global way redundantly with the explicit global root signature. This could be handy for testing or manual debugging though.

Shader table memory initialization

When the system indexes in to a shader table using the stride and arrives at a record, a valid shader identifier must be there, followed by the appropriate amount of local root arguments. Individual local root arguments need only be initialized if the shader executing references them.

In a given record in a shader table, the root arguments that follow the shader identifier must match the local root signature the specified shader was compiled with. The argument layout is defined by packing each argument with padding as needed to align each to its individual (defined) size, and in the order declared in the local root signature. For instance, root descriptors and descriptor handles (identifying descriptor tables) are each 8 bytes in size and therefore need to be at the nearest 8 byte aligned offset from the start of the record after whatever argument precedes it.

Inline raytracing

TraceRayInline() is an alternative to TraceRay() that doesn't use any separate shaders - all shading is handled by the caller. Both styles of raytracing use the same acceleration structures.

`TraceRayInline()`, as a member of the RayQuery object, actually does very little itself - it initializes raytracing parameters. This sets up the shader to call other methods of the RayQuery object to work through the actual raytracing process.

Shaders can instantiate `RayQuery` objects as local variables, each of which acts as a state machine for ray query. The shader interacts with the `RayQuery` object's methods to advance the query through an acceleration structure and query traversal information. Accesses to the acceleration structure (e.g. box and triangle intersection) are abstracted and thus left to the hardware. Surrounding these fixed-function acceleration structure accesses, all necessary app shader code, for handling both enumerated candidate hits and the final result of a query (e.g. hit vs miss) can be contained in the individual shader driving the RayQuery.

`RayQuery` objects can be used in any shader stage, including compute shaders, pixel shaders etc. These can even be using in any raytracing shaders: any hit, closest hit etc., combining both raytracing styles.

Inline raytracing is supported by Tier 1.1 raytracing implementations.

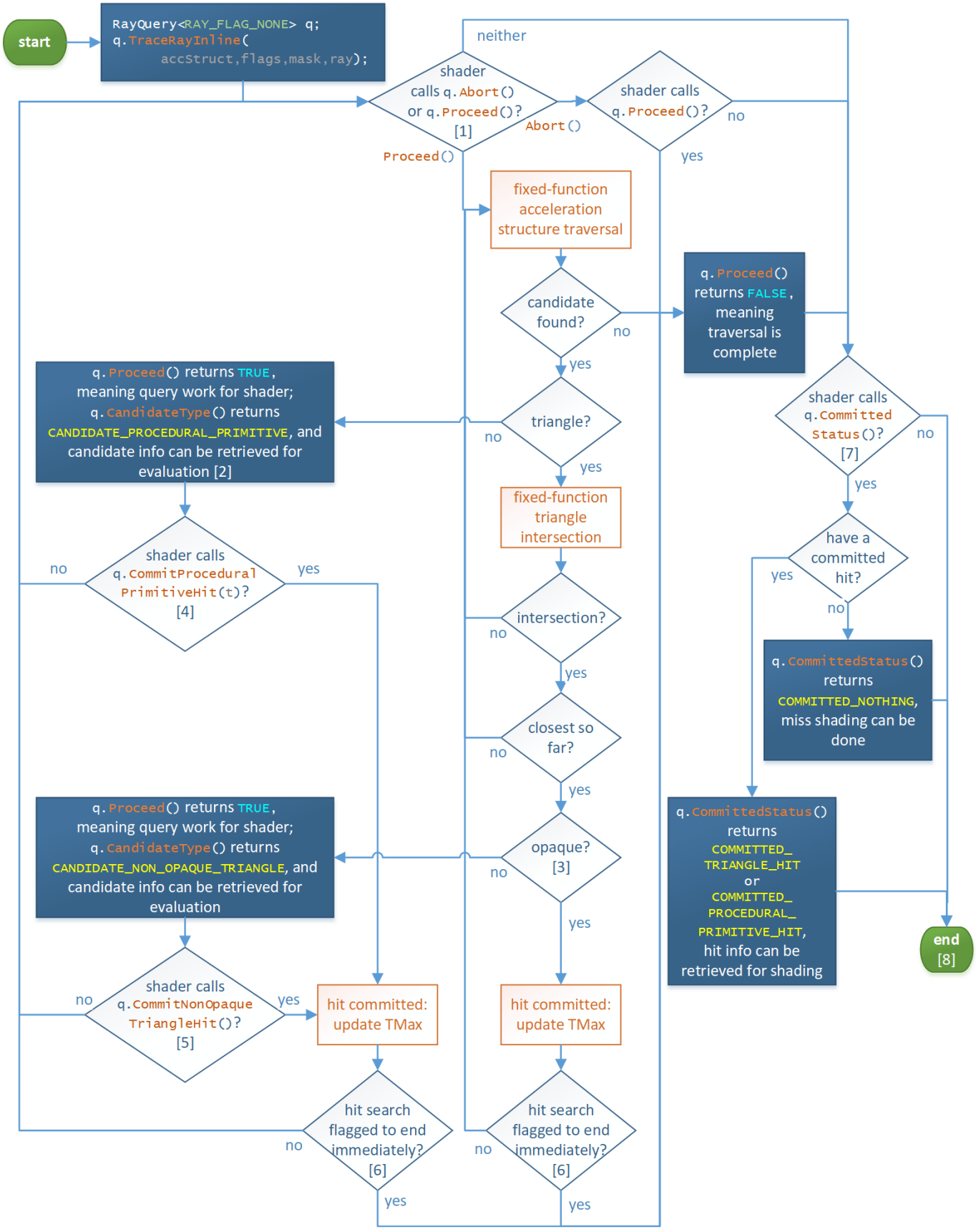
Pseudocode examples are here.

> The motivations for this second parallel raytracing system are both the any-shader-stage property as well as being open to the possibility that for certain scenarios the full dynamic-shader-based raytracing system may be overkill. The tradeoff is that by inlining shading work with the caller, the system has far less opportunity to make performance optimizations on behalf of the app. Still, if the app can constrain the complexity of its raytracing related shading work (while inlining with other non raytracing shaders) this path could be a win versus spawning separate shaders with the fully general path.
>
> One simple scenario for inline raytracing is tracing rays with the `RayQuery` object initialized with template flags: `RAY_FLAG_CULL_NON_OPAQUE | RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES | RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH`, using an acceleration structure with only triangle based geometry. In this case the system can see that it is only being asked to find either a hit or a miss in one step, which it could potentially fast-path. This could enable basic shadow determination from any shader stage as long as no transparency is involved. Should more complexity in traversal be required, of course the full state machine is available for completeness and generality.
>
> It is likely that shoehorning fully dynamic shading via heavy uber-shading through inline raytracing will have performance that depends extra heavily on the degree of coherence across threads. Being careful not to lose too much performance here may be a burden

> largely if not entirely for the the application and it's data organization as opposed to the system.

TraceRayInline control flow

The image below depicts what happens when a shader uses a RayQuery object to call RayQuery::TraceRayInline() and related methods for performing inline raytracing. It offers similar functionality to the full dynamic-shader-based TraceRay() control flow, except refactored to make more sense when driven from a single shader. The orange boxes represent fixed function operations while the blue boxes represent cases where control has returned to the originating shader to drive what happens next, if anything. In each of these states of shader control, the shader can choose to further interact with the `RayQuery` object via a subset of RayQuery intrinsics currently valid based on the current state.



[1] RayQuery::Proceed() searches the acceleration structure to enumerate primitives that may intersect the ray, conservatively: If a primitive is intersected by the ray and is within the current ray extents interval, it is guaranteed to be enumerated eventually. If a primitive is not intersected by the ray or is outside the current ray extents, it may or may not be enumerated. Note that TMax is updated when a hit is committed. RayQuery::Proceed() represents where the bulk of system acceleration structure traversal is implemented (including code inlining where applicable). RayQuery::Abort() is an optional shortcut for the shader to be able to cause traversal to appear to

be complete, via RayQuery::Proceed() returning `FALSE`. So it is just a convenient way to exit the shader's traversal loop. A shader can instead choose to break out of its traversal logic manually as well with normal shader code branching (invisible to the system). This works since, as discussed in [5], the shader can call RayQuery::CommittedStatus() and related methods for retrieving committed hit information at any time after a RayQuery::TraceRayInline call.

[2] Consider the case where the geometry is not triangle based. Instead of fixed function triangle intersection RayQuery::Proceed() returns control to the shader. It is the responsibility of the shader to evaluate all procedural intersections for this acceleration structure node, including resolving transparency for them if necessary without the system seeing what's happening. The net result in terms of traversal is to callRayQuery::CommitProceduralPrimitiveHit() at most once if the shader finds an opaque hit that is closest so far.

[3] Opaqueness is determined by examining the geometry and instance flags of the intersection as well as the selection of ray flags (template parameter and dynamic flags OR'd together).

[4] While in `CANDIDATE_PROCEDURAL_PRIMITIVE` state it is ok to call RayQuery::CommitProceduralPrimitiveHit() zero or more times for a given candidate, as long as each time called, the shader has manually ensured that the latest hit being committed is within the current ray extents. The system does not do this work for procedural primitives. So new committed hits will update TMax multiple times as they are encountered. For simplicity, the flow diagram doesn't visually depict the situation of multiple commits per candidate. Alternatively the shader could enumerate all procedural hits for the current candidate and just call `CommitProceduralPrimitiveHit()` once for the closest hit found that the shader has calculated will be the new TMax.

[5] While in `CANDIDATE_NON_OPAQUE_TRIANGLE` state, the system has already determined that the candidate would be the closest hit so far in the ray extents (e.g. would be new TMax if committed). It is ok for the shader to call RayQuery::CommitNonOpaqueTriangleHit() zero or more times for a given candidate. If called more than once, subsequent calls simply have not effect as the hit has already been committed. For simplicity, the flow diagram doesn't visually depict the situation of multiple commits per candidate.

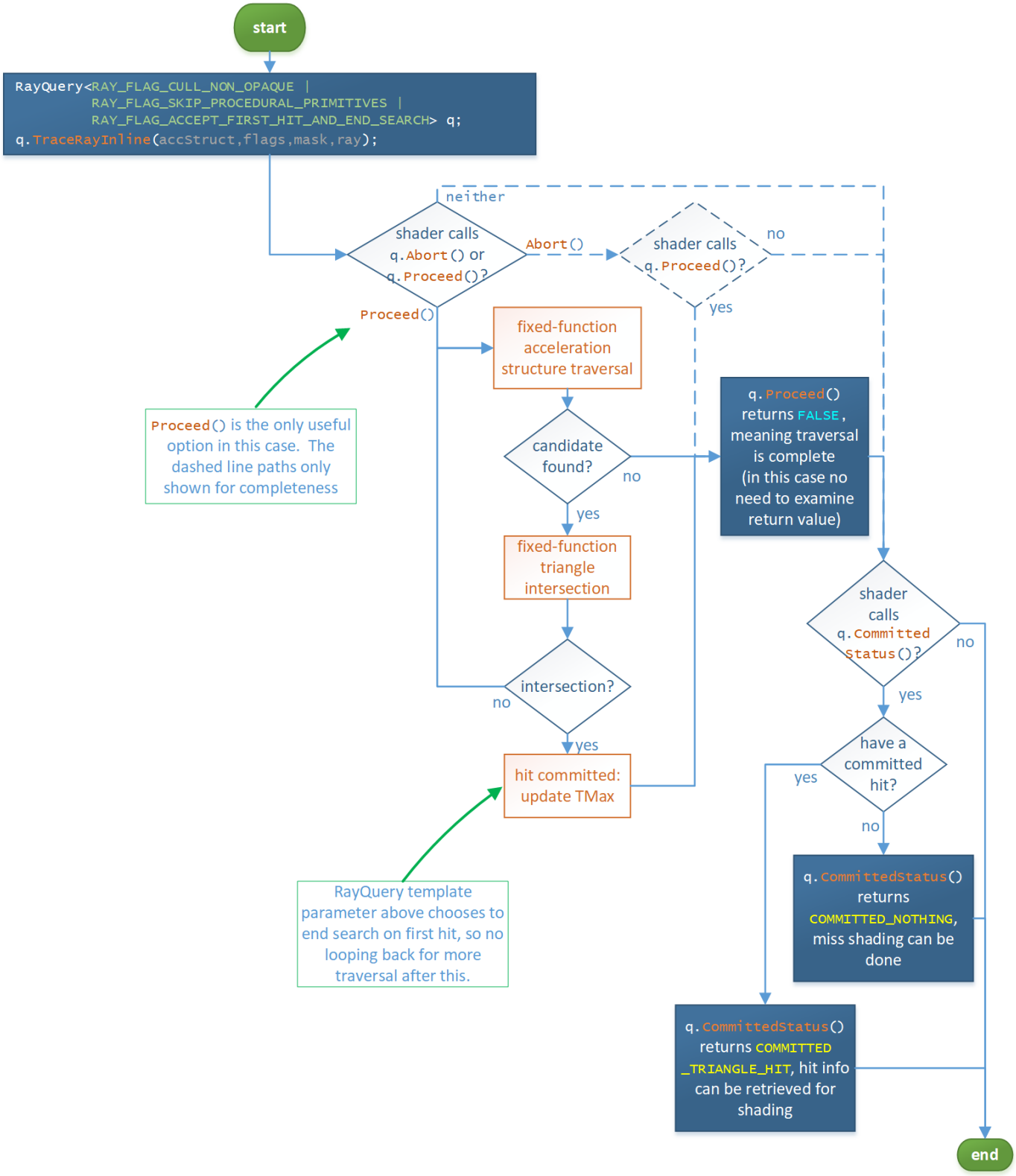
[6] The search for hits is ended at this point if the `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` ray flag is set.

[7] The shader can call RayQuery::CommittedStatus() from anywhere - a careful look at the flow diagram will reveal this. It doesn't have to be after traversal has completed. The status (including values returned from methods that report commited values) simply reflect the current state of the query. What is not depicted in the diagram is that if RayQuery::CommittedStatus() is called before traversal has completed, the shader can still continue with the ray query. One scenario where it can be interesting to call RayQuery::CommittedStatus() before RayQuery::Proceed() has returned false is if the shader has chosen to manually break out of its traversal loop without calling RayQuery::Abort() discussed in [1].

[8] The endpoint of the graph is trivially reachable anytime the shader has control simply by not calling methods that advance `RayQuery` state. The shader can arbitrarily choose to stop using the `RayQuery` object, or do final shading based on whatever the current state of the `RayQuery` object is. The shader can even reset the query regardless of its current state at any time by calling RayQuery::TraceRayInline() again to initialize a new trace.

Specialized TraceRayInline control flow

The image below depicts how a particular choice of template flags used with the initial declaration of `RayQuery` can prune down the full flow graph (depicted above). There is no shader participation needed in the search for intersections here. Further, the search is configured to end upon the first hit. Simplifications like this can free the system to generate more performant inline raytracing code.



Shader management

Problem space

Implementations juggle many shaders

In a given DispatchRays() invocation from a CommandList, the application must have a way to specify every shader that might be invoked, since rays can go anywhere. It would seem this is solved by the presence of shader tables that allow applications to arbitrarily select shaders and their root arguments.

However, implementations have the potential to run the arbitrary set of shaders more efficiently if they also get a chance to see the full set up front (before execution). So the design choice is to give implementations the ability to perform a quick link step. This link doesn't recompile the individual shaders but instead makes some scheduling decisions based on the characteristics of all of the shaders in the potentially referenced set. Where applications have freedom is to reference any of the shaders in a given pre-identified set from anywhere in shader tables.

Sets of shaders need to be pre-defined because it isn't viable to require drivers to inspect shader tables in order to figure out what the reachable set for a given DispatchRays() call. Shader tables can be modified freely by the application (with appropriate state barriers), after all, on the GPU timeline. The expectation is that any analysis of the set of shaders for the purpose of scheduling optimization for the group is best left as a CPU task for the driver.

This motivates the need to make some representation of all shaders reachable by a raytracing operation on the CPU timeline.

Applications control shader compilation

Applications must control when and where (on which threads) shader compilation occurs given the high CPU cost, particularly with large asset bases.

First there is the initial shader compile to DXIL binary, which can be done offline by an application (before any hardware driver sees it). The HLSL compiler supports DXIL libraries, allowing applications to easily store large compiled codebases in single files if desired.

Given shaders in one or more DXIL libraries, they must be submitted to drivers to compile on any given system where the shaders will run. Applications must be able to choose which subset of any given DXIL library a driver should compile at any given time; applications have the freedom to choose how to distribute driver shader compilation across threads, regardless of how groups of shaders happen to be packaged into DXIL libraries.

State objects

A state object represents a variable amount of configuration state, including shaders, that an application manages as a single unit and which is given to a driver atomically to process (e.g.

compile/optimize) however it sees fit. A state object is created via CreateStateObject() on a D3D12 device.

Subobjects

State objects are built out of subobjects. A subobject has a type and corresponding data. A couple of examples of subobject types: `D3D12_STATE_SUBOBJECT_TYPE_DXIL_LIBRARY` and `D3D12_STATE_SUBOBJECT_TYPE_LOCAL_ROOT_SIGNATURE*` .

Another notable subobject type is `D3D12_STATE_SUBOBJECT_TYPE_SUBOBJECT_TO_SHADERS_ASSOCIATION` , whose role is to associate another subobject with a list of DXIL exports. This enables, for example, multiple local root signatures to be present in a state object simultaneously, each associated with different shader exports. See Subobject association behavior for detailed discussion, and see the relevant parts of the state object API here: D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIATION and D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION.

The full set of subobject types is defined in D3D12_STATE_SUBOBJECT_TYPE.

Subobjects in DXIL libraries

DXIL libraries, compiled offline before state object creation, can also define many of the same kinds of subobjects that can be directly defined in state objects. The DXIL/HLSL versions of subobjects are defined here.

The reason that subobjects can be defined either in DXIL libraries or in state objects is to give the application the choice about how much state authoring to do offline (DXIL libraries) vs at runtime (state objects).

Shaders are **not** considered a subobject, so while they are present in DXIL libraries, they can't be directly passed into state objects. Instead, the way to get shaders into a state object is to put the containing DXIL library into the state object, as a DXIL library subobject, including the requested shader entrypoint names to include.

State object types

State objects have a type that dictates rules about the subobjects they contain and how the state objects can be used.

Raytracing pipeline state object

One of the state object types is `D3D12_STATE_OBJECT_TYPE_RAYTRACING_PIPELINE`, or raytracing pipeline state object - RTPSO for short. An RTPSO represents a full set of shaders that could be reachable by a DispatchRays call, with all configuration options resolved, such as local root signatures and other state.

An RTPSO can be thought of as an *executable* state object.

The input to SetPipelineState1() is a state object, where an RTPSO can be bound to the command list.

Graphics and compute state objects

In the future, graphics and compute pipelines could be defined in state object form for completeness. Initially the focus is on enabling raytracing. So for now the way graphics and compute PSOs are constructed is not changed.

Collection state object

Another state object type is `D3D12_STATE_OBJECT_TYPE_COLLECTION`, or collection for short. A collection can contain any amount of subobjects, but doesn't have constraints. Not all dependencies the included subobjects have must be resolved in the same collection. Even if dependencies are locally defined, the set of subobjects doesn't have to be the complete set of state that will eventually be used on the GPU. For instance, a collection may not include all shaders needed to raytrace a scene, though it could.

The purpose of a collection is to allow an application to pass an arbitrarily large or small collection of state to drivers to compile at once (e.g. on a given thread).

If too little configuration information is provided in the subobjects in a collection, the driver cannot compile anything, and would be left with simply storing the subobjects. This situation isn't allowed by default: collection creation will fail. The collection can opt out of needing all dependencies to be resolvable, causing the driver to defer compilation, by setting the `D3D12_STATE_OBJECT_FLAG_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITONS` flag in D3D12_STATE_OBJECT_FLAGS.

A collection must meet the following requirements for the driver to be able to compile it immediately:

- library functions called by shaders must be have code definitions
- resource bindings referenced by shadders must have local and/or global root signature subobjects defining the bindings
- raytracing shaders must have a D3D12_RAYTRACING_SHADER_CONFIG and a D3D12_RAYTRACING_PIPELINE_CONFIG subobject

The parts of the above list that involve subobject associations are discussed further at subobject association requirements.

For simplicity, collections can't be made out of other collections. Only executable state objects (e.g. RTPSOs) can take existing collections as a part of their definition.

State object lifetimes as seen by driver is a discussion useful for driver authors.

Collections vs libraries

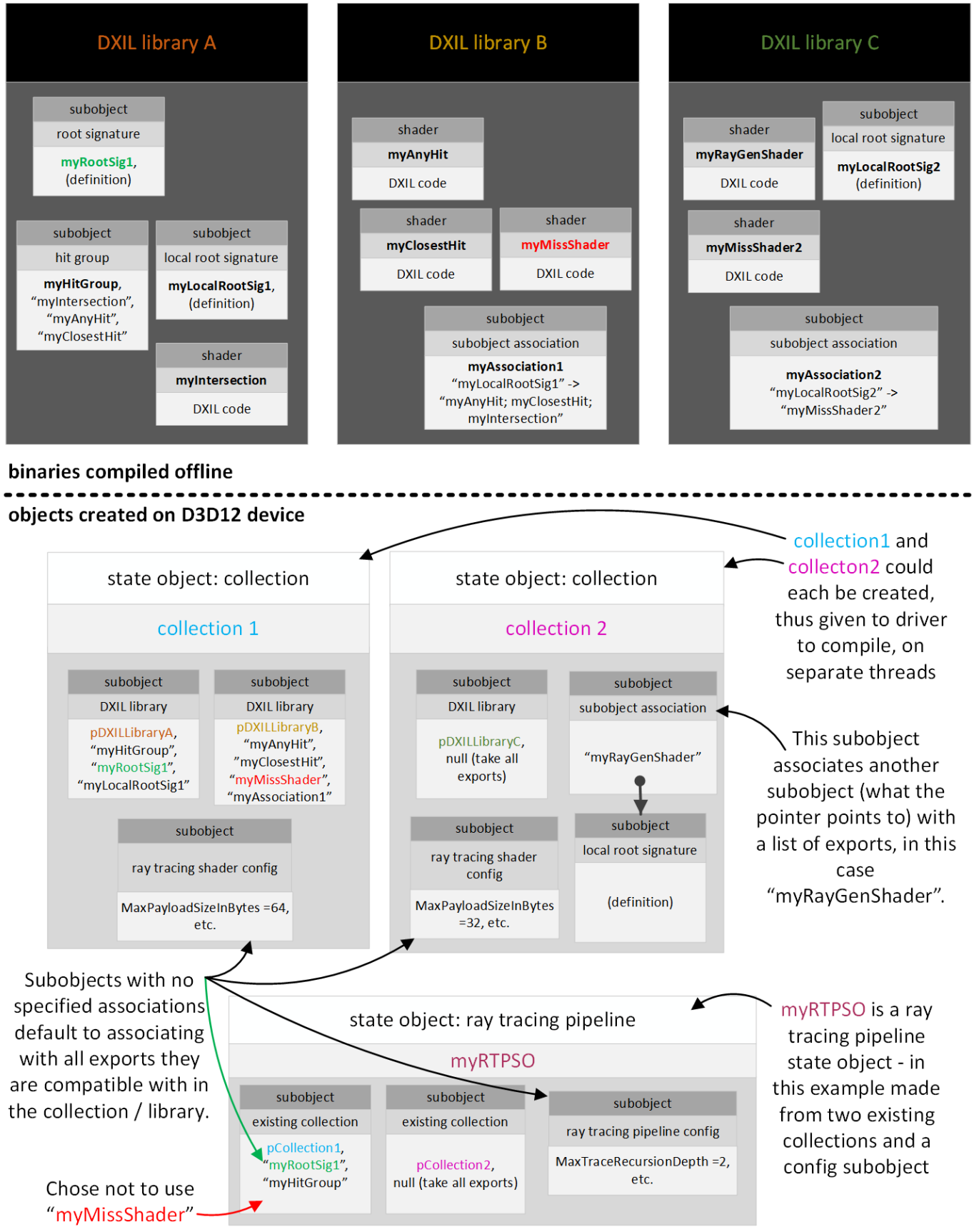
A collection is a bit like a library, but a different name is used to distinguish it from DXIL Libraries.

DXIL Libraries are hardware agnostic. In contrast, a collection's contents are given to a driver to process and can include a part of a DXIL Library (by listing a subset of exports to use), multiple DXIL Libraries, as well as other types of subobjects like root signatures.

An application can choose to have many tiny DXIL libraries each with a single compileable raytracing shader. It can choose to create a separate collection for each one across different threads on a CPU. Or it could make one collection per thread, distributing the set of shaders evenly across them. In either case, an RTPSO can then be constructed out of the set of collections.

Alternatively, if CPU time compiling shaders in the driver before use isn't a concern, an application can skip making collections and pass all of its DXIL Libraries directly into the creation of an RTPSO. An extreme case of this would be an application baking all of its shader assets (and other necessary subobjects like root signatures) in a single DXIL Library (e.g. one binary file), loading this into memory and passing it directly into the creation of an RTPSO. The driver would have to compile all of the shaders at once on one thread.

DXIL libraries and state objects example



Subobject association behavior

The introduction to Subobjects called out a specific type of subobject that associates another subobject with a set of shader exports.

This section describes how subobjects (like root signatures) get associated with shaders in DXIL libraries and state objects. This includes the way default associations (intended for convenience) and explicit association work. Also covered is how DXIL subobject associations can be overridden when an app includes a DXIL library in a state object.

Default associations

Default associations serve as a convenience for the common case where a given subobject (like a root signature) will be used with many shaders.

Terminology

For the subsequent discussion consider the following **scopes** of visibility where a set of shaders can be found:

- a given DXIL library
- a collection state object, which may get shaders from one or more DXIL libraries
- an executable state object (e.g. RTPSO), which may get shaders from one or more collections and/or DXIL libraries

A given scope can **contain** other inner scopes, or outer scopes can **enclose** it.

Declaring a default association

There are two ways declare a default association for a subobject to a set of shaders:

1) Declare a subobject in a given scope with no explicit associations in that scope that reference it. If this subobject is involved in an association defined in any *other* scope including enclosing or contained scopes, it doesn't affect that locally this subobject acts as a default association.

2) Define an association with an empty export list. The subobject specified may or may not be in the current scope. The subobject specified can also be unresolved (not defined in current, containing or enclosed scopes), unless the state object being defined is executable, e.g. RTPSO.

Behavior of a default association

In a default association a subobject is associated with all candidate exports in the current and contained scopes, but not to enclosing scopes. Candidates to be associated are exports for which the association would make sense, and that don't have an explicit association with another subobject of the same type already. There is one exception, where default association can **override** an existing association on an export, described later.

Explicit associations

Explicit associations associate a given subobject to a specific nonempty list of exports.

The subobject being associated (e.g. root signature) and/or the listed exports can be in any scope in the object.

In addition, neither the subobject being associated or the listed exports have to even be visible yet (may be unresolved references), unless the state object is executable, e.g. RTPSO.

Multiple associations of a subobject

A given subobject can be referenced in multiple association definitions, explicit and/or default. This way any given association definition doesn't need to be all-knowing (does not need to be aware of all shaders a subobject may be relevant to).

The use of multiple association declarations also enables, for instance, broadcasting a default association for a given subobject to be broadcast into multiple scopes. Each association declaration (in a different scope) this case would use an empty export (making it a default association) but reference the same subobject.

Conflicting subobject associations

If there are multiple explicit subject associations (with different subobject definitions) that map to a given shader export, this is a conflict. If a conflict is discovered during DXIL library creation, library creation will fail. Otherwise if a conflict is discovered during state object creation, that fails.

The determination of conflicts doesn't care what scope(s) hold the associations, subobjects being associated or shader export within a given state object (or single scope DXIL library). This is because by definition, explicit associations can reach anywhere in a state object (or DXIL library).

There is one exception where a conflict doesn't cause a failure and instead there is a precedence order that causes an override:

Exception: overriding DXIL library associations

Subobject associations (default or explicit) declared directly in state objects that target an export in a directly included DXIL library cause any other association to that export that was defined in any DXIL library (same or other library) to no longer apply to that export. And as a result, the subobject association declared directly in the state object "wins" and overrides the DXIL based association. This includes associations defined at state object scope via either

D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIATION (association at state object scope for a subobject also defined at state object scope) or D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION (association at state object scope a subject defined in a DXIL library).

Associations declared in state objects do not override any existing associations in contained collections (including DXIL libraries the contained collections may have).

The subobject being associated can be unresolved unless the state object is RTPSO (executable).

> The reason overriding is only defined for DXIL libraries directly passed into a given state object's creation is the following. Drivers never have to worry about compiling code that came from a DXIL library during state object creation only to have to recompile later because multiple subobject overrides happened. e.g. creating a collection that overrides associations in a DXIL library then creating an RTPSO that includes the collection and tries to override an association again is invalid (the second association becomes conflicting and state object creation fails).
>
> The value in supporting overriding of subobject associations is to give programmatic code (i.e. performing state object creation) one chance to override what is in a static DXIL library, without having to patch the DXIL library itself.

Subobject associations for hit groups

Hit groups reference a set of component shaders, such as a closest hit shader, any hit shader, and/or intersection shader. Subobject associations (like associating a local root signature to a shader) can be made directly to the individual component shaders used by a hit group and/or directly to the hit group. Making the association to the hit group can be convenient, as it applies to all the component shaders (so they don't need individual associations). If both a hit group has an association and its component shaders have associations, they must match. If a hit group doesn't have a particular subobject association, the associations for all component shaders must match. So different component shaders can't use different local root signatures, for instance.

Runtime resolves associations for driver

The runtime resolves what subobject associations ended up at any given export, accounting for defaults, overriding etc. and tells the driver the result during state object association. This ensures consistency across implementations.

Subobject association requirements

This table describes requirements for each subobject type that supports being associated with shader exports.

The “Match rule” column describes whether a subobject association is required or optional for any given shader and whether the definition of the subobject must match that of other shaders.

The “Match scope” column describes what set of shader code the matching requirement applies to.

Subobject type	Match rule	Match scope
Raytracing shader config	Required & matching for all exports	Full state object. More discussion at D3D12_RAYTRACING_SHADER_CONFIG.
Raytracing pipeline config	Required & matching for all exports	Full state object. More discussion at D3D12_RAYTRACING_PIPELINE_CONFIG, D3D12_RAYTRACING_PIPELINE_CONFIG1.
Global root signature	Optional, if present must match shader entry	Call graph reachable from shader entry. More discussion at D3D12_GLOBAL_ROOT_SIGNATURE.
Local root signature	Optional, if present must match shader entry	Call graph reachable from shader entry (not including calls through shader tables). More discussion at D3D12_LOCAL_ROOT_SIGNATURE.
Node mask	Optional, if present match for all exports	Full state object. More discussion at D3D12_NODE_MASK.
State object config	Optional, if present match for all exports	Local state object only, doesn’t need to match contained state objects. More discussion at D3D12_STATE_OBJECT_CONFIG

State object caching

Drivers are responsible for implementing caching of state objects using existing services in D3D12 to improve performance when state objects (or components in them) are reused across runs of an application.

Incremental additions to existing state objects

Tier 1.1 implementations support adding to existing state objects via AddToStateObject(). This incurs lower CPU overhead in streaming scenarios where new shaders need to be added to a state object that is already being used, rather than having to create a state object that is mostly redundant with an existing one. Details on the nuances of this option are described at AddToStateObject().

System limits and fixed function behaviors

Addressing calculations within shader tables

> The very fixed nature of shader table indexing described here is a result of IHV limitation. The hope is these limitations aren't too annoying for apps (which have to live with them). The extent to which the fixed function choices made here conflict with what an app actually wants may force app to do inefficient things like duplicating entries in shader tables to accomplish what they want. That said, such inefficiencies in shader table layout may not turn out to be an overall bottleneck. So this might be no worse than simply being slightly awkward to use.

Hit group table indexing

> HitGroupRecordAddress = D3D12_DISPATCH_RAYS_DESC.HitGroupTable.StartAddress + // from: DispatchRays()
> D3D12_DISPATCH_RAYS_DESC.HitGroupTable.StrideInBytes * // from: DispatchRays()
> (RayContributionToHitGroupIndex + // from shader: TraceRay()
> (MultiplierForGeometryContributionToHitGroupIndex * // from shader: TraceRay()
> GeometryContributionToHitGroupIndex) + // system generated index of geometry in bottom-level acceleration structure (0,1,2,3..)
> D3D12_RAYTRACING_INSTANCE_DESC.InstanceContributionToHitGroupIndex // from instance
>)

Setting MultiplierForGeometryContributionToHitGroupIndex > 1 lets apps group shaders for multiple ray types adjacent to each other per-geometry in a shader table. The acceleration structure doesn't need to know this is happening, as it merely stores an InstanceContributionToHitGroupIndex per-instance. GeometryContributionToHitGroupIndex is a fixed function sequential index (0,1,2,3..) incrementing per geometry, mirroring the order each geometry was placed by the app in the current bottom-level acceleration structure.

As of raytracing Tier 1.1 implementations, it can be interesting to set MultiplierForGeometryContributionToHitGroupIndex to 0, meaning geometry index does not contribute to shader table indexing at all. This can work if the shaders will be calling the GeometryIndex() intrinsic (added for Tier 1.1 implementations) to be able to distinguish geometries within shaders manually.

Miss shader table indexing

> MissShaderRecordAddress = D3D12_DISPATCH_RAYS_DESC.MissShaderTable.StartAddress + // from: DispatchRays()
> D3D12_DISPATCH_RAYS_DESC.MissShaderTable.StrideInBytes * // from: DispatchRays()
> MissShaderIndex // from shader: TraceRay()

Callable shader table indexing

> CallableShaderRecordAddress = D3D12_DISPATCH_RAYS_DESC.CallableShaderTable.StartAddress + // from: DispatchRays()
> D3D12_DISPATCH_RAYS_DESC.CallableShaderTable.StrideInBytes * // from: DispatchRays()
> ShaderIndex // from shader: CallShader()

Out of bounds shader table indexing

Behavior is undefined if shader tables are indexed out of range. The same applies to referencing a region within a shader table that is uninitialized or contains stale data.

Acceleration structure properties

Data rules

- Once an acceleration structure has been built, it does not retain any references to inputs to the build, including vertex buffers etc. pointed to by the app's acceleration structure description.
- Acceleration structures are self-contained aside from top-level acceleration structures pointing to bottom-level acceleration structures.
- Applications may not inspect the contents of an acceleration structure. Nothing stops a determined app from doing this, but the point is the data is implementation-dependent,

undocumented and therefore useless for an app to inspect.

- Once built, an acceleration structure is immutable with the exception of updates (incremental builds) done in-place.
- A top-level acceleration structure must be rebuilt or updated before use whenever bottom-level acceleration structures it references are rebuilt or updated.
- The valid operations on acceleration structures are the following:
    - input to TraceRay() and RayQuery::TraceRayInline() from a shader
    - input to BuildRaytracingAccelerationStructure():
        - as a bottom-level structure being referenced by a top-level acceleration structure build
        - as the source for an acceleration structure update (incremental build)
            - source can be the same as destination address to mean an in-place update
        - input to CopyRaytracingAccelerationStructure(), which has various modes for doing things like acceleration structure compaction or simply cloning the data structure
            - in particular, notice that copying acceleration structures in any other way is invalid
        - input to EmitRaytracingAccelerationStructurePostbuildInfo(), which reports information about an acceleration structure like how much space is needed for a compacted version.

Determinism based on fixed acceleration structure build input

Given a fixed world composed of triangles and AABBs, as well as identical shader code and data in the same order, multiple identical TraceRay() or RayQuery::TraceRayInline() calls produce identical results on the same device and driver. This requirement means that both the tracing of rays must be deterministic, and the acceleration structure must also be constructed such that it behaves deterministically.

Given the same triangle stream, AABB stream and any other configuration input to multiple acceleration structure builds (including the same instance and geometry transforms and other properties as applicable), the resulting acceleration structures' behavior must be the same on a given device and driver. The actual acceleration structures' contents may not be bit for bit identical, which could be revealed by a memory comparison. Matching acceleration structure data itself is of no value – they may contain internal pointers for instance that refer to differing addresses or data layout orderings without effect on behavior. So it is just the functional

behavior of the consistently constructed acceleration structures that must match. The same intersections will be found in the same order with the same order of shader invocations, assuming application shaders and data that could affect execution flow also match.

For acceleration structure updates (incremental builds), multiple identical update sequences with matching sets of inputs to each update result in the same consistency of acceleration structure behavior described above.

Determinism based varying acceleration structure build input

Aside from the obvious fact that changing the locations and amount of geometry used to build an acceleration will affect its behavior, there are subtler variations that can affect acceleration structure function.

Acceleration structure intersection finding and intersection ordering behaviors may change as a result of varying of any of the following factors across acceleration structure builds:

- vertex order (for triangles)
- primitive order (for triangles)
- AABB order
- instance order in a top-level acceleration structure
- geometry ordering in a bottom-level acceleration structure
- flags to acceleration structure build (or instance / geometry flags)
- acceleration structure update (incremental build) count and input history
- device/driver
- user defined values embedded in acceleration structures contributing to shader table indexing calculation or shader IDs. Implementations may find reason to, for instance, sort contents on these or somehow know which sets of content use the same values. Of course during an acceleration structure build the actual shader tables are not present, so the most an implementation could look at are the raw offset/ID values without trying to use them.

Acceleration structure intersection finding and intersection ordering behaviors do not change as a result of varying any of the following factors across acceleration structure builds:

- memory addresses of acceleration structures or build inputs (aside from data ordering tolerances described above)
- time

Preservation of triangle set

Implementations may not change the input set of triangles in an acceleration structure aside from the vertex order and primitive order. Merging, splitting, dropping triangles are not permitted.

Observable duplication of primitive in an acceleration structure is invalid. Observable meaning in any way that becomes visible during raytracing operations beyond just performance difference. Exceptions are:

- Intersection shader invocation counts, which are allowed to be duplicated.
- If an application has **not** set the flag `D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION` in a given geometry, multiple any hit invocations may be observed for a given primitive for a given ray.

Barycentrics provided in the intersection attributes structure for a triangle intersection must be relative to the original vertex order, since the app has to be able to look up vertex attributes on its own.

AABB volume

Implementations may replace the AABBs provided as input to an acceleration structure build with more or fewer AABBs (or other representation), with the only guarantee that the locations in space enclosed by the input AABBs are included in the acceleration structure.

In particular, applications must not depend on the planes in the AABBs input to acceleration structure build having any kind of clipping effect on the shapes defined by enclosed intersection shader invocations. An implementation may have chosen some larger volume than the input AABB for which to invoke intersection shaders. While there is freedom for implementations here, excessive bloat of bounding volumes would incur extreme performance penalties from unnecessary intersection shader invocation. So the extent of bounding volume bloating should be limited in practice.

Inactive primitives and instances

Triangles are considered “inactive” (but legal input to acceleration structure build) if the x component of each vertex is NaN. Similarly AABBs are considered inactive if AABB.MinX is NaN. The geometries and/or their transforms in bottom level acceleration structures can be used as one way to inject these NaN values and deactivate entire instances/geometries. Instances with NULL bottom level acceleration structure pointers are also considered legal but inactive.

All inactive primitives/AABBs/bottom level acceleration structures are discarded during the acceleration structure build phase and can thus not be intersected during traversal.

Inactive primitives may not become active in subsequent acceleration structure updates – results are undefined if attempted. Conversely, primitives that were active at the initial build operation may not be changed to inactive (and thus can't be discarded) in acceleration structure updates.

Inactive primitives are counted for PrimitiveIndex(), which affects the index values for neighboring active primitives. Similarly inactive instances are counted for InstanceIndex(), which affects the index for neighboring instances. This ensures that any array indexing an app wants to do based on index values for active primitives isn't affected by the presence of inactive primitives earlier in the array.

For triangles and AABBs, none of the input coordinates may be NaN unless the first coordinate is also NaN (that is, only when the primitive is inactive), otherwise behavior is undefined.

Triangle vertices using an SNORM format (see VertexFormat in D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC) cannot be inactive, given SNORM has no NaN representation.

Degenerate primitives and instances

The following are considered "degenerate" (after applying all transforms): Triangles that form a point or a line, point sized AABBs (Min.X == Max.X, same for Y and Z), and instances referencing (non-NULL) bottom level acceleration structures containing no active primitives.

All degenerate primitives and instances are still considered "active" and may participate in acceleration structure updates with new data, switching to/from degenerate status freely.

Degenerate instances use a point at the instance origin (specified by the instance transform) to guide the acceleration structure builder. Apps should give a best estimate of where any future updates might place the instance.

AABBs with inverted bounds (Min.X > Max.X or similar for Y and Z) are converted to a degenerate point at the center of the input bounds for all acceleration structure operations.

During traversal, degenerate AABBs may still report possible (false positive) intersections and invoke the intersection shader. The shader may check the validity of the hit by, for example, inspecting the bounds.

Degenerate triangles, on the other hand, do not generate any intersections.

Degenerate primitives may be discarded from an acceleration structure build, unless the `ALLOW_UPDATE` flag specified. As a consequence, the coordinates returned during AS visualization may be replaced with NaNs for these primitives.

An exception to the rule that degenerates cannot be discarded with `ALLOW_UPDATE` specified is primitives that have repeated index value can always be discarded (even with `ALLOW_UPDATE` specified). There is no value in keeping them since index values cannot be changed.

The primitive count in the returned GeometryDescs and the number of InstanceDescs may be affected by any discarded primitives. The index (buffer location) and output order of active primitives will still be correct, however.

Geometry limits

The runtime does not enforce these limits (defined too late). Exceeding them produced undefined behavior.

Maximum number of geometries in a bottom level acceleration structure: 2^24

Maximum number of primitives in a bottom level acceleration structure (sum across all geometries): 2^29, including any inactive or degenerate primitives (described above).

Maximum instance count in a top level acceleration structure: 2^24

Acceleration structure update constraints

The following describes the data that an app can change to the inputs of an acceleration structure update relative to the inputs / flags etc. used to build the source acceleration structure. Note that per acceleration structure Data rules, once built they never hold explicit references to the data used to build them, so it is fine for an update to provide data from different addresses in memory as long as the only changes in the data itself conform to the following restrictions.

> A rule of thumb is that the more that acceleration structure updates diverge from the original, the more that raytrace performance is likely to suffer. An implementation is expected to be able to retain whatever topology it might have in an acceleration structure during update.

Bottom-level acceleration structure updates

The VertexBuffer and/or Transform members of D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC can change. The Transform member cannot change between NULL <-> non-NULL, however. An app that wants to update Transform but doesn't have one initially can specify the identity matrix rather than NULL.

Essentially this means vertex positions can change.

The AABBs member of D3D12_RAYTRACING_GEOMETRY_DESC can change.

Nothing else can change, so note that in particular this means no changes to properties like the number of geometries, VertexCount, or AABBs, geometry flags, data formats, index buffer

contents and so on.

Note that if a bottom-level acceleration structure at a given address is pointed to by top-level acceleration structures ever changes, those top-level acceleration structures are stale and must either be rebuilt or updated before they are valid to use again.

Top-level acceleration structure updates

The InstanceDescs member of D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC can change.

This refers to D3D12_RAYTRACING_INSTANCE_DESC structures in GPU memory. The number of instances, defined by the NumDescs member of D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC, cannot change.

So aside from the number of instances used in the top-level acceleration structure being fixed, the definitions of each of the instances can be completely redefined during an acceleration structure update, including which bottom-level acceleration structure each instance points to.

Acceleration structure memory restrictions

Acceleration structures can only be placed in resources that are created in the default heap (or custom heap equivalent). Further, resources that will contain acceleration structures must be created in the state D3D12_RESOURCE_STATE_RAYTRACING_ACCELERATION_STRUCTURE, and must have resource flag `D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS`. The `ALLOW_UNORDERED_ACCESS` requirement simply acknowledges both: that the system will be doing this type of access in its implementation of acceleration structure builds behind the scenes, and from the app point of view, synchronization of writes/reads to acceleration structures is accomplished using UAV barriers (discussed later).

For the following discussion, these resources are referred to as acceleration structure buffers (ASBs).

ASBs cannot be transitioned into any other state, or vice versa, otherwise the runtime will put the command list into removed state.

If a placed buffer is created that is an ASB, but there is an existing buffer overlapping the VA range that is not an ASB, or vice versa, this is an error enforceable by debug layer error.

Regarding reserved buffers, if a tile is ever mapped into an ASB and a non-ASB simultaneously this is an error enforceable by debug layer error. Mapping a tile into or out of an acceleration structure invalidates that tile's contents.

> The reason for segregating ASBs from non-ASB memory is to enable tools/PIX to be able to robustly capture applications that use raytracing. The restriction avoids instability/crashing from tools attempting to serialize what they think are opaque acceleration structures that might have been partially overwritten by other data because the app repurposed the memory without tools being able to track it. The key issue here is the opaqueness of acceleration structure data, requiring dedicated APIs for serializing and deserializing their data to be able to preserve application state.

Synchronizing acceleration structure memory writes/reads

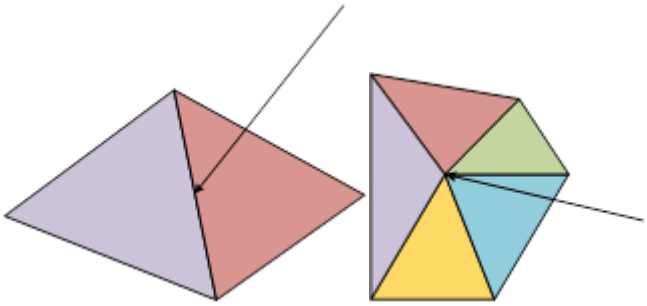
Given that acceleration structures must always be in D3D12_RESOURCE_STATE_RAYTRACING_ACCELERATION_STRUCTURE as described above, resource state transitions can't be used to synchronize between writes and reads (and vice versa) of acceleration structure data. Instead, the way to accomplish this is using UAV barriers on resources holding acceleration structure data between operations that write to an acceleration structure (such as BuildRaytracingAccelerationStructure() and operations that read from them (such as DispatchRays() (and vice versa).

> The use of UAV barriers (as opposed to state transitions) for synchronizing acceleration structure accesses comes in handy for scenarios like compacting multiple acceleration structures. Each compaction reads an acceleration structure and then writes the compacted result to another address. An app can perform a string of compactions to tightly pack a collection of acceleration structures that all may be in the same resource. No resource transitions are necessary. Instead all that's needed are a single UAV barrier after one or more original acceleration structure builds are complete before passing them into a sequence of compactions for each acceleration structure. Then another UAV barrier after compactions are done and the acceleration structures are referenced by DispatchRays() for raytracing.

Fixed function ray-triangle intersection specification

For manifold geometry:

- A single ray striking an edge in a scene must report an intersection with only one of the incident triangles. A ray striking a vertex must report an intersection with only one of the incident triangles. Which triangle is chosen may vary for different rays intersecting the same edge.



In the above examples of a shared edge intersection and a shared vertex intersection, only one triangle must be reported as intersected in each case.

For non-manifold geometry:

- When a ray strikes an edge shared by more than two triangles all triangles on one side of the edge (from the point of view of the ray) are intersected. When a ray strikes a vertex shared by separate surfaces, one triangle per surface is intersected. When a ray strikes a vertex shared by separate surfaces and strikes edges in the same place, the intersections for the points and the intersections for the edges each appear based on the individual rules for points and edges.

Watertightness

The implementation must use a ray-triangle intersection that is watertight. Regardless of where in the 32-bit float precision range ray-triangle intersections occur, gaps between triangles sharing edges must never appear. The scope of "sharing" for this definition of watertightness spans only a given bottom level acceleration structure for geometries that have matching transforms.

One example of an implementation of watertight ray-triangle intersection is here:

> http://jcgt.org/published/0002/01/05/paper.pdf

The following is another example focused on being efficient with an acceleration structure implementation while maintaining watertightness:

> https://software.intel.com/en-us/articles/watertight-ray-traversal-with-reduced-precision

It is expected that implementations of watertight ray triangle intersections are possible without having to resort to costly paths such as double precision fallbacks. This includes including following a form of top-left rule discussed below to remove double hits on edges.

Top-left rule

In triangle rasterization, the top-left rule guarantees consistent in/out determination on triangle edges across implementations with no holes or double hits on shared edges. This is possible

because there is a consistent space involved (“screen space”) and snapping vertex positions to fixed point precision is done during rasterization, eliminating variance.

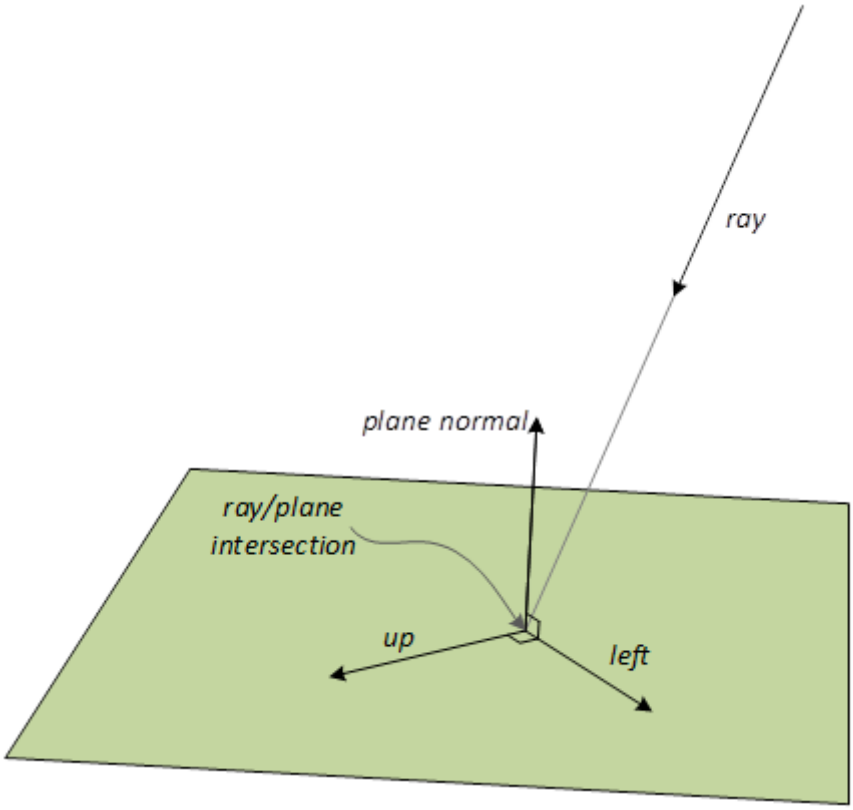
These conditions are not present with raytracing. So while some form of top-left rule (or equivalent) must be used by each implementation doing ray triangle intersection to decide whether intersections on edges are in or out, this only guarantees watertightness for that implementation and not exactly the same results across implementations.

What follows is an example top-left rule that can be applied by an implementation. The requirement is merely that implementations do something like this to guarantee the intersection properties described earlier.

Example top-left rule implementation

Determining a coordinate system

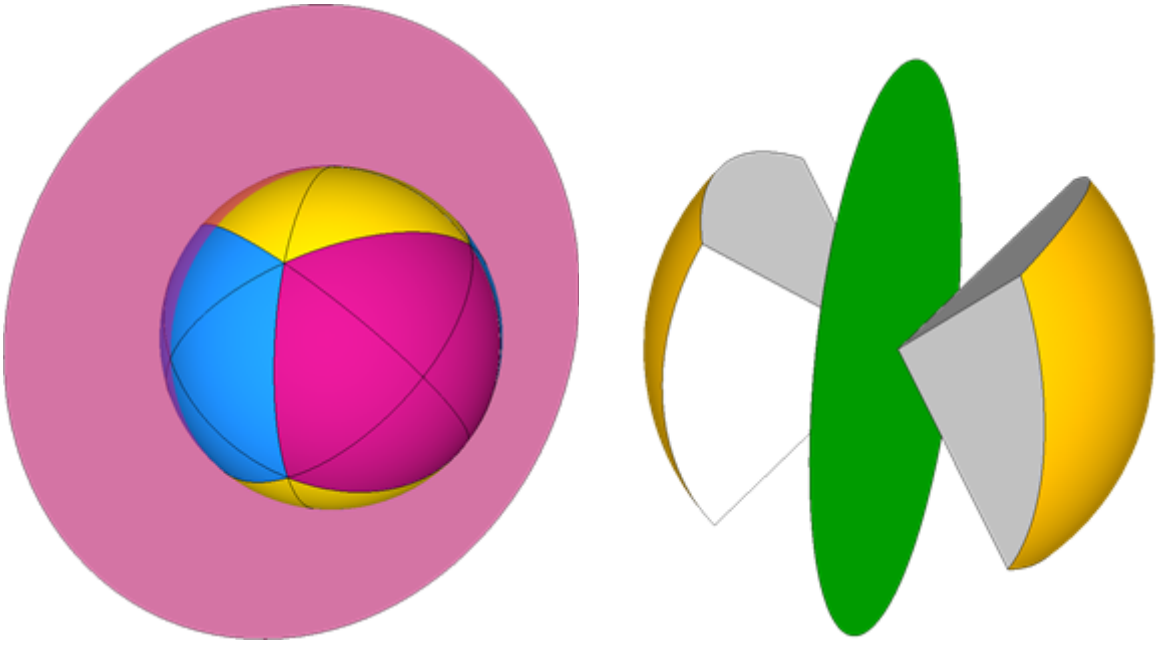
- For each ray, choose a plane where ray-triangle intersections are performed. Obviously, the plane must be intersected by the ray and may not contain the ray. This plane is only a function of the ray itself (origin and direction) and might not be the same plane chosen by any other ray.
- Compute the ray/plane intersection.
- Originating at the intersection point, choose a direction within the plane, also a function of only the ray. Consider this direction to be left.
- Take the cross product of left direction and the plane normal to establish the up direction. This yields the coordinate system needed to implement the top-left rule.



Hypothetical scheme for establishing plane for ray-tri intersection

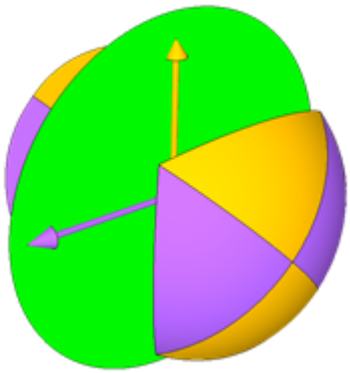
Using the dominant ray direction is one way to establish a plane within which to perform intersection. This involves choosing from one of three plane orientations, matching the three major axes. The offset of the intersection plane is not important.

The below images show example ray direction to intersection plane mappings. Ray directions on the unit sphere map to three possible intersection planes.



Choose the left direction within the plane based on next largest magnitude component of the normal.

In the below image, this corresponds to the colored triangular regions and corresponding left directions within the plane. There are two possible left directions within each of the three possible planes.



Triangle intersection

- Projected the triangle onto the ray's plane.
- Test the triangle against the ray/plane intersection. If the ray/plane intersection is strictly interior to the triangle, report an intersection.
- If the ray/plane intersection lies directly on one of the projected triangle's edges, apply the top-left rule to establish whether the triangle is considered intersected:

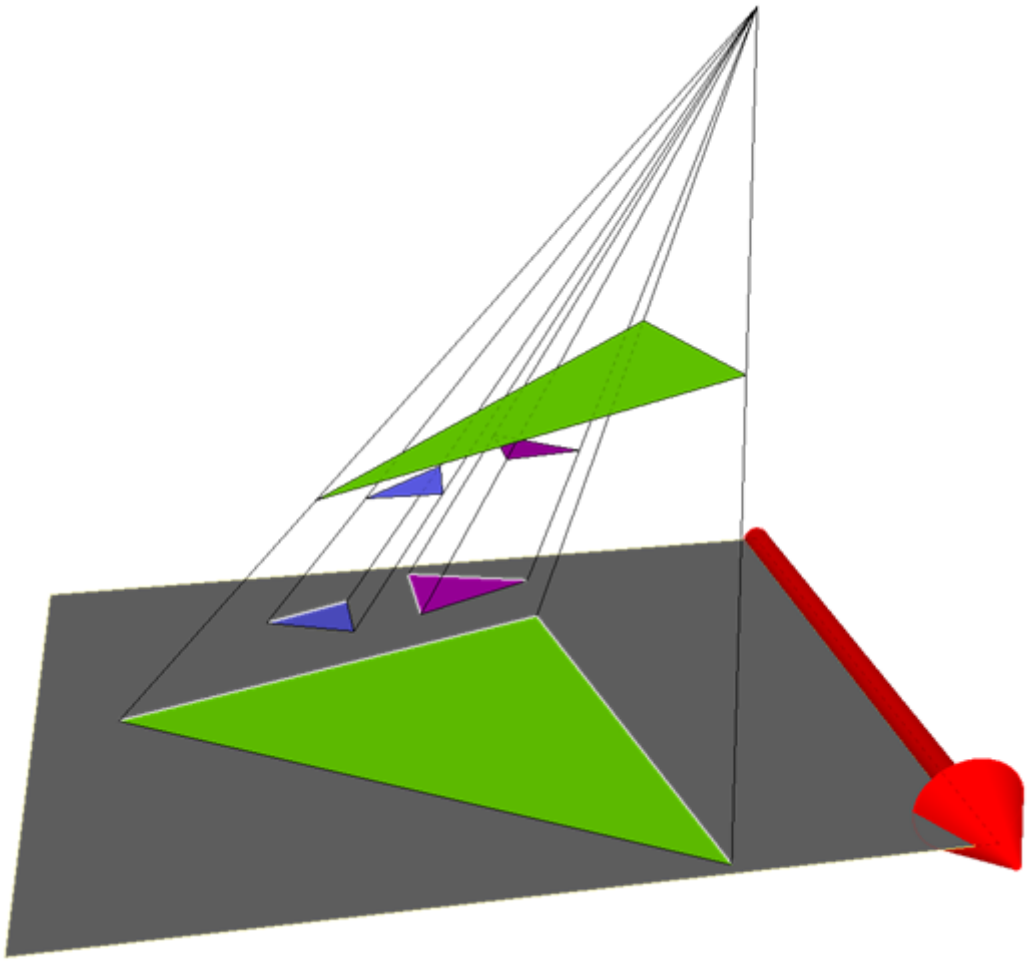
  ***Top edge**: If a projected edge is exactly parallel to the left direction, and the* up *direction points away from the projected triangle's interior in the space of the ray's plane, then it is a "top" edge.*

  ***Left edge**: If a projected edge is not exactly parallel to the left direction, and the* left *direction points away from the projected triangle's interior in the space of the ray's plane, then it is a "left" edge. A triangle can have one or two left edges.*

  ***Top-left rule**: If the ray-plane intersection falls exactly on the edge of a projected triangle, the triangle is considered intersected if the edge is a "top" edge or a "left" edge. If two edges from the same projected triangle (a vertex) coincide with the ray-plane intersection, then if both edges are "top" or "left" then the triangle is considered intersected.*

Examples of classifying triangle edges

In the below image, a single left direction (red) is shown in the intersection plane. Inclusive edges are shown in black and exclusive edges shown in white. Note that the blue and magenta triangles have edges parallel to the left direction. In one case, the edge in question is an upper edge in the other case the edge is a lower edge, inclusive and exclusive respectively.



As a series of rays strike an edge, the left direction can change 90°. Because of the change in left direction, the inclusive/exclusive classification of an edge can change along the length of the edge. Classification may also change if the intersection plane changes (also a function of ray direction) along the length of an edge.

Ray extents

Ray-triangle intersection can only occur if the intersection t-value satisfies TMin < t < TMax.

Ray-procedural-primitive intersection can only occur if the intersection t-value satisfies TMin <= t <= TMax.

> The reason procedural primitives use an inclusive bounds test is to give apps a choice about how to handle exactly overlapping intersections. For instance, an app could choose to compare primitiveID or somesuch for currently committed hit versus a candidate for intersection to decide whether to accept a new overlapping hit or not.

Ray TMin must be nonnegative and <= TMax. +INF is a valid TMin/TMax value (really only makes sense for TMax).

No part of ray origin, direction, T range can be NaN.

The runtime does not enforce these limits (they may get validated in GPU based validation eventually).

Violating these rules produces undefined behavior.

Ray recursion limit

Raytracing pipeline state objects must declare a maximum ray recursion depth (in the range [0..31]). The ray generation shader is depth 0. Below the maximum recursion depth, shader invocations such as closest hit or miss shaders can call TraceRay() any number of times. At the maximum recursion depth, TraceRay() calls result in the device going into removed state.

The current level of recursion cannot be retrieved from the system, due to the overhead that might be required to be able to report it to shaders. If applications need to track the level of ray recursion it can be done manually in the ray payload.

This recursion depth limit does not include callable shaders, which are not bounded except in the context of overall pipeline stack allocation.

> Apps should pick a limit that is as low as absolutely necessary. There may be performance implications in how the implementation chooses to handle upper limits set at obvious thresholds – e.g. 0 means no tracing of rays at all (perhaps only using callable shaders or not even that), 1 means single bounce rays, and numbers above 1 might imply a different implementation strategy.
>
> It isn't expected that most apps would ever need to declare very large recursion limits. The upper limit of 31 is there to put a bound on the number of bits hardware has to reserve for a counter – inexpensive yet large enough range to likely never have to worry about.

Pipeline stack

Raytracing shaders including callable shaders may consume memory out of a driver managed stack allocation. This memory is internally allocated/reserved by the driver during command list recording, such that command list recording will fail if the driver wouldn't be able to execute the command list due to the selected stack size. The stack memory requirement is expressed in terms of how much memory a call chain starting from an individual ray generation shader thread can consume, considering tracing rays, the various shaders that can be invoked in the process, including callable shaders, and nesting/recursion.

> In practice typical systems will support many thousands of threads in flight at once, so the actual memory footprint for driver managed stack storage will be much larger than the space required for just one thread. This multiplication factor is an implementation detail not

> directly exposed to the app. That said, it is in the app's best interest (if memory footprint is important) to make an optimal choice for the one number it has control over – individual thread stack size – to match what the app actually needs.

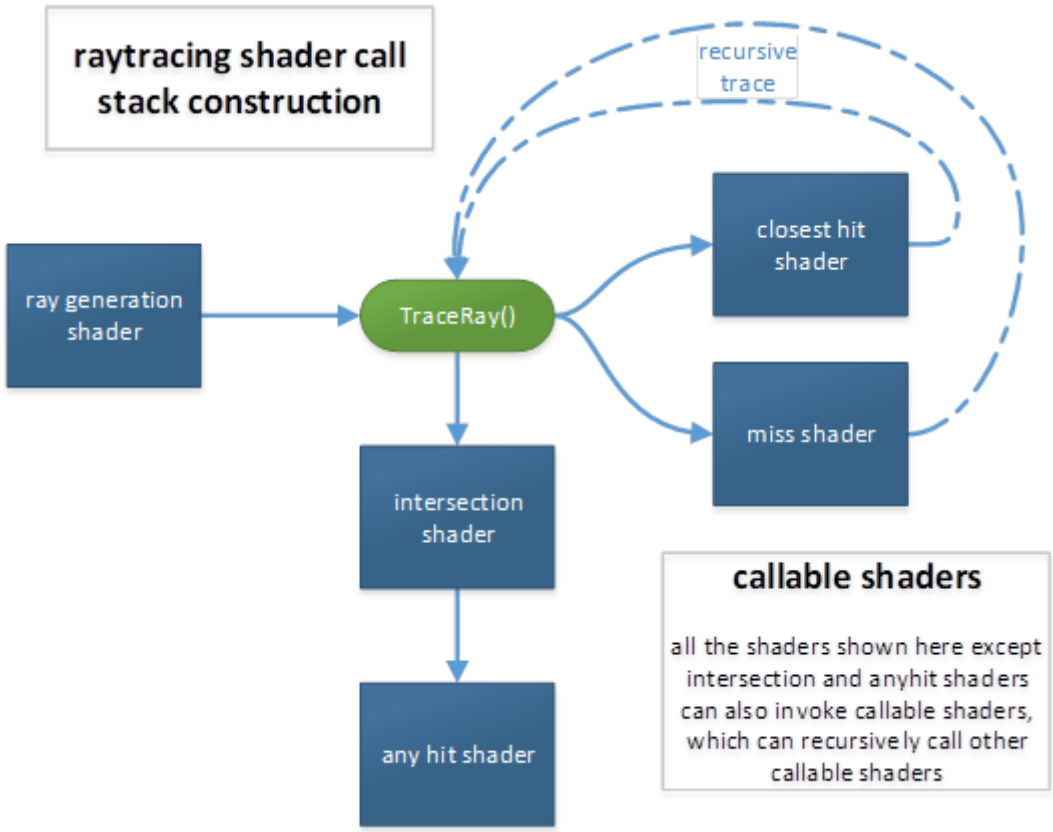
Raytracing pipeline state objects can optionally set a maximum pipeline stack size, otherwise a default value is used, which is typically overly conservative (though could underestimate if callable shaders recurse more than 2 levels). The way an app can calculate an optimal stack size if desired is described next, followed by an explanation of how the default is selected.

> Here is an example of a situation where it really matters for an app to manually calculate the stack size rather than rely on the default: Suppose there is a complex closest hit shader with lots of state doing complex shading that recursively shoots a shadow ray that's known to hit only trivial shaders with very small stack requirements. The default calculation described further below doesn't know this and will assume all levels of recursion might invoke the expensive closest hit shader, resulting in wasted stack space reservation, multiplied by the number of threads in flight on the GPU.

Optimal pipeline stack size calculation

Apps can retrieve the stack space requirement for individual shaders in a raytracing pipeline via GetShaderStackSize(). (The result will be the same for a given shader if it appears in other raytracing pipelines.) If the app combines these sizes with what it may know about the worst case call stack amongst individual shaders during raytracing, along with the MaxTraceRecursionDepth it declared, it can calculate the correct stack size. This is something the system cannot do on its own.

The diagram below depicts how an app author can reason about calculating an optimal stack size based on the shaders being used in the raytracing pipeline and which ones might potentially be reachable (which only the app author can reasonably know).



The app can set the overall stack storage per thread for a raytracing pipeline state via SetPipelineStackSize(). The specification for that method describes rules about when and how often the stack size can be set for a pipeline state.

Default pipeline stack size

The system initializes raytracing pipeline state objects with a default pipeline stack size computed as follows. This is calculation is intentionally simplified because it cannot account for what combination of shaders might actually execute (as that depends on application content and shader table layout, which are both unknown from the perspective of a raytracing pipeline state). The default stack size calculation takes the worst case combination of shaders in the raytracing pipeline in terms of individual stack sizes and factors in the maximum recursion level. For callable shaders, a default assumption is that every raytracing shader calls the callable shader with the maximum stack size to a recursion depth of 2.

The net result is that for raytracing pipelines with no callable shaders, the default stack size is guaranteed to safely fit the maximum declared level of recursion and the worst case combination of shaders. With callable shader in the mix, the default might be unsafe if the app happens to make more than 2 levels of recursive calls to the worst case callable shader on top of maxing out all other shaders.

The exact calculation is as follows. First a definition of the input variables.

For each shader type in the pipeline state, for which multiple instance of that shader type might be in the pipeline state, find the maximum individual shader stack size for that shader type. For

this discussion, let us name these maximum values:

RGSMax (for max ray generation shader stack size) **ISMax** (intersection shader) **AHSMax** (any hit shader) **CHSMax** (closest hit shader) **MSMax** (miss shader) **CSMax** (callable shader)

The other relevant input is comes from the D3D12_RAYTRACING_PIPELINE_CONFIG subobject in the pipeline state: **MaxTraceRecursionDepth**.

Using these values, considering the raytracing shader call stack construction diagram above, and arbitrarily estimating callable shaders to be called 2 levels deep per shader stage, the default stack size calculation becomes:

```
DefaultPipelineStackSizeInBytes =
  RGSMax
  + max( CHSMax, MSMax, ISMax+AHMax ) * min( 1, MaxTraceRecursionDepth )
  + max( CHSMax, MSMax ) * max( MaxTraceRecursionDepth - 1, 0 )
  + 2 * CSMax // if CS aren't used, this term will just be 0.
              // 2 is a completely arbitrary choice

  // Observe that ISMax and AHMax are only counted once, which results in
  // the split clauses involving MaxTraceRecursionDepth. Intersection and
  // anyhit shaders can't recurse.
```

Pipeline stack limit behavior

If making a call exceeds the declared stack size the device goes into removed state, similar to ray recursion overflow.

There is no practical limit on declared stack size. The runtime drops calls to SetPipelineStackSize() for extreme stack size values, >= 0xffffffff though (the parameter is actually UINT64 for this purpose). This is to catch the app blindly passing the return value of calling GetShaderStackSize() with invalid parameters, which returns 0xffffffff, either directly into SetPipelineStackSize or into a calculation summing stack sizes, multiple of which could be invalid values.

Shader limitations resulting from independence

Given that raytracing shader invocations are all independent of each other, features within shaders that explicitly rely on cross-shader communication are not allowed, with the exception of Wave Intrinsics described further below. Examples of features not available to shaders during raytracing: 2x2 shader invocation based derivatives (available in pixel shaders), thread execution syncing (available in compute).

Wave Intrinsics

Wave intrinsics are allowed in raytracing shaders, with the intent that they are for tools (PIX) logging. That said, applications are also not blocked from using wave intrinsics in case they might find safe use.

Implementations may repack threads at certain (well defined) points in raytracing shader execution such as calls to TraceRay(). As such, the results of wave intrinsics called within a shader are valid only until a potential thread repacking point is encountered in program execution order. In wave intrinsics have scopes of validity that are bounded by repacking points as well as the start/end of the shader.

Repacking points that bound wave intrinsic scope:

- CallShader()
- TraceRay()
- ReportHit()

Other intrinsics that result in a bound due to ending the shader invocation:

- IgnoreHit()
- AcceptHitAndEndSearch()

Note that use of RayQuery objects for inline raytracing does not count as a repacking point.

Execution and memory ordering

When TraceRay() or CallShader() are called, any resulting shader invocations complete by the time the call returns.

Memory operations (stores, atomics) performed by the caller can be guaranteed to be visible to the callee with a memory barrier in either caller or callee. `DeviceMemoryBarrier()` is the relevant barrier call for raytracing shaders. Input payload/parameter data is passed to callees by-value so does not require a barrier.

Memory operations performed by the callee can similarly be guaranteed to be visible to the caller (and any subsequent calls made by the caller) with a memory barriers in any shader. Output payload/parameter data is returned to the caller by-value so does not require a barrier.

General tips for building acceleration structures

The following generalized advice apples to the use of BuildRaytracingAccelerationStructure(). While it is possible that over time as more diverse device support appears the advice needs to be refined, even as-is this should be a useful reminder of the various options at play.

- **Prefer triangle geometry over procedural primitives**
    - Ray-triangle intersection can be hardware accelerated
- **Mark geometry as OPAQUE whenever possible**
    - If geometry doesn't require any-hit shader code to execute (e.g. for alpha testing), then always make sure it's marked as OPAQUE to utilize the raytracing hardware as effectively as possible. It doesn't matter whether the OPAQUE flag comes from the geometry descriptor (`D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE`), the instance descriptor (`D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_OPAQUE`), or through a ray flag (`RAY_FLAG_FORCE_OPAQUE`).
- **Merge many objects into fewer bottom-level acceleration structures**
    - In other words, take advantage of the fact that a build can accept more than one geometry descriptor and transform the geometry while building. This generally leads to the most efficient data structures, especially when objects' AABBs overlap each other. In addition, it reduces the number of BuildRaytracingAccelerationStructure invocations which leads to higher GPU utilization and lower overall CPU overhead. Consider merging e.g. for objects that consist of multiple meshes (and need to be rebuilt/updated at the same time), and for any static or almost static geometry.
- **Only build/update per frame what's really needed**
    - Acceleration updates aren't free, so objects that haven't deformed between frames shouldn't trigger one. This is sometimes not trivial to detect in an engine, but the effort can pay off twice since it may also be able to skip a skinning/vertex update pass.
- **Consider multiple update sources for skinned meshes**
    - Acceleration structure updates can happen either in-place, or use separate source and destination buffers. For some geometry (e.g. a hero character), it can make sense to build multiple high quality acceleration structures in different key poses upfront (e.g. during level load time), and then refit every frame using the closest matching keyframe as a source.
- **Rebuild top-level acceleration structure every frame**
    - Only updating instead of rebuilding is rarely the right thing to do. Rebuilds for a few thousand instances are very fast, and having a good quality top-level acceleration structure can have a significant payoff (bad quality has a higher cost further up in the tree).

- **Use the right build flags**
    - The next section is a guideline for common use cases

Choosing acceleration structure build flags

- Start by choosing a D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS combination from here:

#	PREFER_FAST_TRACE	PREFER_FAST_BUILD	ALLOW_UPDATE	Properties	Example
1	no	yes	no	Fastest possible build. Slower trace than #3 and #4.	Fully dynamic geometry like particles, destruction, changing prim counts or moving wildly (explosions etc), where per-frame rebuild is required.
2	no	yes	yes	Slightly slower build than #1, but allows very fast update.	Lower LOD dynamic objects, unlikely to be hit by too many rays but still need to be refitted per frame to be correct.
3	yes	no	no	Fastest possible trace. Slower build than #1 and #2.	Default choice for static level geometry.

#	PREFER_FAST_TRACE	PREFER_FAST_BUILD	ALLOW_UPDATE	Properties	Example
4	yes	no	yes	Fastest trace against update-able acceleration structure. Updates slightly slower than #2. Trace a bit slower than #3.	Hero character, high-LOD dynamic objects that are expected to be hit by a significant number of rays.

</small>

- Then consider adding these flags:

  ALLOW_COMPACTION

  Whenever compaction is desired. It's generally a good idea to do this on all static geometry to reclaim (potentially significant) amounts of memory.

  For updateable geometry, it makes sense to compact those BVHs that have a long lifetime, so the extra step is worth it (compaction and update are not mutually exclusive!).

  For fully dynamic geometry that's rebuilt every frame (as opposed to updated), there's generally no benefit from using compaction.

  One potential reason to NOT use compaction is to exploit the guarantee of BVH storage requirements increasing monotonically with primitive count – this does not hold true in the context of compaction.

  MINIMIZE_MEMORY

  Use only when under general mem pressure, e.g. if otherwise a DXR path won't run at all because things don't fit. Usually costs build and trace perf.

Determining raytracing support

See CheckFeatureSupport() and D3D12_RAYTRACING_TIER. This reports the level of device raytracing support.

Raytracing emulation is completely independent of the above – it is just a software library that sits on top of D3D.

Raytracing emulation

> The following emulation feature proved useful during the experimental phase of DXR design. But as of the first shipping release of DXR, the plan is to stop maintaining this codebase. The cost/benefit is not justified and further, over time as more native DXR support comes online, the value of emulation will diminish further. That said, the description of what was done is left in case a strong justification crops up to resurrect the feature.

The raytracing fallback layer is a library that provides support for raytracing on devices that do not have native driver/hardware support using a DX12 compute-shader-based solution. The library is built as a wrapper around the DX12 API and has distinct (but similar) interfaces from the DXR API. The library will also have an internal switch that allows it to use the DXR API when driver support exists, and fallback to compute when it doesn't. A desired outcome is that a fallback is useful on existing hardware without native raytracing support (and without driver implemented emulation) at least for limited scope scenarios that complement/support traditional graphics based rendering techniques. Emulation can also serve as a form of reference implementation that runs on GPUs (as opposed to the WARP software rasterizer) that raytracing capable devices can be compared against.

The fallback layer will provide via a public GitHub repo where developers can build and incorporate the library into their own engines. The repo will be open for both engine developers and hardware developers to submit pull requests to improve the code base. A benefit to having the fallback layer outside of the OS is that it can be snapped with releases of the DXIL compiler and developers are free to tailor their snap of the fallback layer with optimizations specific to their codebase. The fallback layer can even be used on older Windows OS's that do not have DXR API support.

The implementation will use a series of compute shaders to build acceleration structures on the GPU time line. Recursive shader invocations (via TraceRay or CallShader) will be handled by linking all shaders into a large state-machine shader that will emulate these invocations as function calls where parameters are saved off to a GPU-allocated stack. Performance is still TBD, but a goal is to ensure that common cases enable enough performance for small-scale techniques.

Tooling with the fallback layer will work with PIX. Use of PIX's raytracing-specific debugging tools (acceleration structure visualization for example) however will be limited to hardware with driver

support. The fallback layer's raytracing invocations will be shown as the underlying compute shaders dispatches.

While the fallback layer interface tries to stay faithful to the original DXR API, there are several points of divergence. The primary difference is DXR's requirement of reading pointers in shaders, primarily at ray dispatch when traversing from a top-level acceleration structure to a bottom-level acceleration structure. To mitigate this, the fallback layer forces the developer to provide pointers not as GPU VA's but instead as "descriptor heap index" and "byte offset" pairs, referred to as an emulated pointer. These are also required for handling local root descriptors and array-of-pointer layouts. The expectation is that this provides an abstracted form of pointer support that will allow for minimal porting from the native DXR API.

Tools support

Some parts of the design have been tweaked to be friendly to tools such as debug layers and PIX doing capture, playback and analysis. In addition to the design tweaks listed below, several generic techniques can be applied by tools (not specific to raytracing), such as shader patching, root signature patching and more generally, API hooking.

Buffer bounds tracking

- DispatchRays() has input parameters are pointers (GPUVA) to shader tables. Size parameters are also present so that tools can tell how much memory is being used.

Acceleration structure processing

- EmitRaytracingAccelerationStructurePostbuildInfo() and CopyRaytracingAccelerationStructure() support dedicated modes for tools to operate on acceleration structures in the following ways:
    - **serialization:**

      Driver serializes acceleration structures to a (still) opaque format that tools (or an app) can store to a file.

    - **deserialization:**

      Driver deserializes the serialized format above on a later playback of a captured application. The result is an acceleration structure that functions as the original did when serialized, and is the same size or smaller than the original structure before

serialization. This only works on the same device / driver that the serialization was performed on.

- **visualization:**

  Convert an opaque acceleration structure to a form that can be visualized by tools. This is a bit like the inverse of an acceleration structure build, where the output in this case is non-opaque geometry and/or bounding boxes. Tools can display a visualization of any acceleration structure at any point during an application run without having to incur overhead tracking how it was built.

  The format of the output may not exactly match the inputs the application originally used to generate the acceleration structure, per the following:

  For triangles, the output for visualization represents the same set of geometry as the application's original acceleration structure, other than any level of order dependence or other variation permitted by the acceleration structure build spec. Transform matrices may have been folded into the geometry. Triangle format may be different (with no loss of precision) – so if the application used float16 data, the output of visualization might be float32 data.

  For AABBs, any spatial volume contained in the original set of AABBs must be contained in the set of output AABBs, but may cover a larger volume with either a larger or smaller number of AABBS.

  Visualization requires the OS to be in developer mode.

Note that serialization and deserialization may still be needed by PIX even if D3D12 gets support for repeatable VA assignment for allocations across application runs. If PIX wants to modify workloads at all during playback, VA can't be preserved.

- See Acceleration structure update constraints for discussion on resource state requirements for acceleration structures that have been put in place. These restrictions, combined with the fact that all manipulations of acceleration structures must go through dedicated APIs for manipulating them mean that PIX can robustly trust the contents of an acceleration structure are valid.

API

Device methods

Per D3D12 device interface semantics, these device methods can be called by multiple threads simultaneously.

CheckFeatureSupport

```
HRESULT CheckFeatureSupport(
    D3D12_FEATURE Feature,
    [annotation("_Inout_updates_bytes_(FeatureSupportDataSize)")]
    void* pFeatureSupportData,
    UINT FeatureSupportDataSize
    );
```

This isn't a raytracing specific API, just the generic D3D API for querying feature support. To query for raytracing support, pass `D3D12_FEATURE_D3D12_OPTIONS5` for Feature, and point pFeatureSupportData to a D3D12_FEATURE_D3D12_OPTIONS5 variable. This has a member D3D12_RAYTRACING_TIER RaytracingTier.

CheckFeatureSupport Structures

D3D12_FEATURE_D3D12_OPTIONS5

```
// D3D12_FEATURE_D3D12_OPTIONS5
typedef struct D3D12_FEATURE_DATA_D3D12_OPTIONS5
{
    [annotation("_Out_")] BOOL SRVOnlyTiledResourceTier3;
    [annotation("_Out_")] D3D12_RENDER_PASS_TIER RenderPassesTier;
    [annotation("_Out_")] D3D12_RAYTRACING_TIER RaytracingTier;
} D3D12_FEATURE_DATA_D3D12_OPTIONS5;
```

The D3D12 options struct that reports RaytracingTier, the raytracing support level (among other unreleated features). See D3D12_RAYTRACING_TIER.

D3D12_RAYTRACING_TIER

```
typedef enum D3D12_RAYTRACING_TIER
{
    D3D12_RAYTRACING_TIER_NOT_SUPPORTED = 0,
    D3D12_RAYTRACING_TIER_1_0 = 10,
    D3D12_RAYTRACING_TIER_1_1 = 11,
} D3D12_RAYTRACING_TIER;
```

Level of raytracing support on the device. Queried via CheckFeatureSupport().

Value	Definition
D3D12_RAYTRACING_TIER_NOT_SUPPORTED	No support for raytracing on the device. Attempts to any raytracing related object will fail and using raytra related APIs on command lists results in undefined b
D3D12_RAYTRACING_TIER_1_0	The device supports the full raytracing functionality c in this spec, except features added in higher tiers liste
D3D12_RAYTRACING_TIER_1_1	Adds: - Support for indirect DispatchRays() calls (ra generation shader invocation) via ExecuteIndirect().<, - Support for incremental additions to existing sta objects via AddToStateObject(). - Support for inline raytracing via RayQuery objects declarable in any sha stage. - GeometryIndex() intrinsic added to re raytracing shaders, for applications that wish to distir geometries manually in shaders in addition to or inst burning shader table slots. - Additional ray fla RAY_FLAG_SKIP_TRIANGLES and RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES . - New of raytracing pipeline config subobject, D3D12_RAYTRACING_PIPELINE_CONFIG1, adding a fl D3D12_RAYTRACING_PIPELINE_FLAGS. The equivaler subobject in HLSL is RaytracingPipelineConfig1. The a flags, D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_TRIANGLE D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_PROCEDURAL_PR (minus D3D12_ when defined in HLSL) behave like OI equivalent RAY_FLAGS above into any TraceRay() call raytracing pipeline, except that these do not show up RayFlags() call from a shader. Implementations may b make pipeline optimizations knowing that one of the primitive types can be skipped. - Additional ve formats supported for acceleration structure build inp part of D3D12_RAYTRACING_GEOMETRY_TRIANGLES

CreateStateObject

```
HRESULT CreateStateObject(
    _In_ const D3D12_STATE_OBJECT_DESC* pDesc,
    _In_ REFIID riid, // ID3D12StateObject
```

```
    _COM_Outptr_ void** ppStateObject
    );
```

See State objects for an overview.

Parameter	Definition
`const D3D12_STATE_OBJECT_DESC* pDesc`	Description of state object to create. See D3D12_STATE_OBJECT_DESC. To help generate this see the `CD3D12_STATE_OBJECT_DESC` helper in class in d3dx12.h.
`REFIID riid`	__uuidof(ID3D12StateObject)
`_COM_Outptr_ void** ppStateObject`	Returned state object.
`Return: HRESULT`	`S_OK` for success. `E_INVALIDARG`, `E_OUTOFMEMORY` on failure. The debug layer provides detailed status information.

CreateStateObject Structures

Helper/sample wrapper code is available to make using the below structures for defining state objects much simpler to use.

D3D12_STATE_OBJECT_DESC

```
typedef struct D3D12_STATE_OBJECT_DESC
{
    D3D12_STATE_OBJECT_TYPE Type;
    UINT NumSubobjects;
    _In_reads_(NumSubobjects) const D3D12_STATE_SUBOBJECT* pSubobjects;
} D3D12_STATE_OBJECT_DESC;
```

Member	Definition
`D3D12_STATE_OBJECT_TYPE Type`	See D3D12_STATE_OBJECT_TYPE.
`UINT NumSubobjects`	Size of pSubobjects array.
`_In_reads(NumSubobjects) const D3D12_STATE_SUBOBJECT* pSubobjects`	Array of subobject definitions. See D3D12_STATE_SUBOBJECT.

D3D12_STATE_OBJECT_TYPE

```
typedef enum D3D12_STATE_OBJECT_TYPE
{
    D3D12_STATE_OBJECT_TYPE_COLLECTION = 0,
    // Could be added in future: D3D12_STATE_OBJECT_TYPE_COMPUTE_PIPELINE = 1,
    // Could be added in future: D3D12_STATE_OBJECT_TYPE_GRAPHICS_PIPELINE = 2,
    D3D12_STATE_OBJECT_TYPE_RAYTRACING_PIPELINE = 3,
} D3D12_STATE_OBJECT_TYPE;
```

Value	Definition
D3D12_STATE_OBJECT_TYPE_COLLECTION	Collection state object.
D3D12_STATE_OBJECT_TYPE_RAYTRACING_PIPELINE	Raytracing pipeline state object.

D3D12_STATE_SUBOBJECT

```
typedef struct D3D12_STATE_SUBOBJECT
{
    D3D12_STATE_SUBOBJECT_TYPE Type;
    const void* pDesc;
} D3D12_STATE_SUBOBJECT;
```

Subobject within a state object description.

Parameter	Definition
D3D12_STATE_SUBOBJECT_TYPE Type	See D3D12_STATE_SUBOBJECT_TYPE.
const void* pDesc	Pointer to state object description of the specified type.

D3D12_STATE_SUBOBJECT_TYPE

```
typedef enum D3D12_STATE_SUBOBJECT_TYPE
{
  D3D12_STATE_SUBOBJECT_TYPE_STATE_OBJECT_CONFIG = 0,
  D3D12_STATE_SUBOBJECT_TYPE_GLOBAL_ROOT_SIGNATURE = 1,
  D3D12_STATE_SUBOBJECT_TYPE_LOCAL_ROOT_SIGNATURE = 2,
  D3D12_STATE_SUBOBJECT_TYPE_NODE_MASK = 3,
  // 4 unused
  D3D12_STATE_SUBOBJECT_TYPE_DXIL_LIBRARY = 5,
  D3D12_STATE_SUBOBJECT_TYPE_EXISTING_COLLECTION = 6,
  D3D12_STATE_SUBOBJECT_TYPE_SUBOBJECT_TO_EXPORTS_ASSOCIATION = 7,
  D3D12_STATE_SUBOBJECT_TYPE_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION = 8,
```

```
    D3D12_STATE_SUBOBJECT_TYPE_RAYTRACING_SHADER_CONFIG = 9,
    D3D12_STATE_SUBOBJECT_TYPE_RAYTRACING_PIPELINE_CONFIG = 10,
    D3D12_STATE_SUBOBJECT_TYPE_HIT_GROUP = 11,
    D3D12_STATE_SUBOBJECT_TYPE_MAX_VALID,
} D3D12_STATE_SUBOBJECT_TYPE;
```

Set of subobject types, each with a corresponding struct definition.

Parameter	
D3D12_STATE_SUBOBJECT_TYPE_STATE_OBJECT_CONFIG	D3D12_STATE_OBJECT
D3D12_STATE_SUBOBJECT_TYPE_GLOBAL_ROOT_SIGNATURE	D3D12_GLOBAL_ROOT
D3D12_STATE_SUBOBJECT_TYPE_LOCAL_ROOT_SIGNATURE	D3D12_LOCAL_ROOT_
D3D12_STATE_SUBOBJECT_TYPE_NODE_MASK	D3D12_NODE_MASK
D3D12_STATE_SUBOBJECT_TYPE_DXIL_LIBRARY	D3D12_DXIL_LIBRARY_
D3D12_STATE_SUBOBJECT_TYPE_EXISTING_COLLECTION	D3D12_EXISTING_COL
D3D12_STATE_SUBOBJECT_TYPE_SUBOBJECT_TO_EXPORTS_ASSOCIATION	D3D12_SUBOBJECT_TO
D3D12_STATE_SUBOBJECT_TYPE_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION	D3D12_DXIL_SUBOBJE
D3D12_STATE_SUBOBJECT_TYPE_RAYTRACING_SHADER_CONFIG	D3D12_RAYTRACING_
D3D12_STATE_SUBOBJECT_TYPE_RAYTRACING_PIPELINE_CONFIG	D3D12_RAYTRACING_
D3D12_STATE_SUBOBJECT_TYPE_HIT_GROUP	D3D12_HIT_GROUP_D

D3D12_STATE_OBJECT_CONFIG

```
typedef struct D3D12_STATE_OBJECT_CONFIG
{
    D3D12_STATE_OBJECT_FLAGS Flags;
} D3D12_STATE_OBJECT_CONFIG;
```

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

This subobject defines general properties of a state object. The presence of this subobject in a state object is **optional**. If present, all exports in the state object must be associated with the same subobject (or one with a matching definition). This consistency requirement does not apply across existing collections that are included in a larger state object, with the exception of the presence of the `D3D12_STATE_OBJECT_FLAG_ALLOW_STATE_OBJECT_ADDITIONS` flag, detailed below.

Member	Definition
`D3D12_STATE_OBJECT_FLAGS Flags`	See D3D12_STATE_OBJECT_FLAGS.

D3D12_STATE_OBJECT_FLAGS

```
typedef enum D3D12_STATE_OBJECT_FLAGS
{
    D3D12_STATE_OBJECT_FLAG_NONE = 0x0,
    D3D12_STATE_OBJECT_FLAG_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITONS = 0x1,
    D3D12_STATE_OBJECT_FLAG_ALLOW_EXTERNAL_DEPENDENCIES_ON_LOCAL_DEFINITIONS = 0x2,
    D3D12_STATE_OBJECT_FLAG_ALLOW_STATE_OBJECT_ADDITIONS = 0x4,
} D3D12_STATE_OBJECT_FLAGS;
```

Value	
`D3D12_STATE_OBJECT_FLAG_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITONS`	This appl The references (de collection is ir depending or external subo absence of th dependencies associations b have enough to keep arour

Value	
	D3D12_STATE_ flag is set). Sc (e.g. RTPSO), link at most.< across separa the AddToSta

	This appl If a shaders / fun (e.g. call) exp (default), exp parts of conta footprint for t uncompiled c some externa said, if not all code in this c

Value	
D3D12_STATE_OBJECT_FLAG_ALLOW_EXTERNAL_DEPENDENCIES_ON_LOCAL_DEFINITIONS	and may still association d export does r presence or a allowed or nc association fc that counts a be set. < shader entryp visible as entr it. In the case tables for ray code across s via the AddTc
D3D12_STATE_OBJECT_FLAG_ALLOW_STATE_OBJECT_ADDITIONS	The prese pipeline, allov either as the presence of tl imported by whether they state object n objects that a

D3D12_GLOBAL_ROOT_SIGNATURE

```
typedef struct D3D12_GLOBAL_ROOT_SIGNATURE
{
    ID3D12RootSignature* pGlobalRootSignature;
} D3D12_GLOBAL_ROOT_SIGNATURE;
```

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

This subobject defines a global root signature that will be used with associated shaders. The presence of this subobject in a state object is **optional**. The combination of global and/or local root signatures associated with any given shader function must define all resource bindings declared by the shader (with no overlap across global and local root signatures).

If any given function in a call graph is associated with a particular global root signature, any other functions in the graph must either be associated with the same global root signature or

none, and the shader entry (the root of the call graph) must be associated with the global root signature.

Different shaders can use different global root signatures (or none) within a state object, however any shaders referenced during a particular DispatchRays() operation from a CommandList must have specified the same global root signature as what has been set on the CommandList as the compute root signature. So it is valid to define a single large state object with multiple global root signatures associated with different subsets of the shaders – apps are not forced to split their state object just because some shaders use different global root signatures.

Member	Definition
`ID3D12RootSignature* pGlobalRootSignature`	Root signature that will function as a global root signature. State object holds a reference.

D3D12_LOCAL_ROOT_SIGNATURE

```
typedef struct D3D12_LOCAL_ROOT_SIGNATURE
{
    ID3D12RootSignature* pLocalRootSignature;
} D3D12_LOCAL_ROOT_SIGNATURE;
```

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

This subobject defines a local root signature that will be used with associated shaders. The presence of this subobject in a state object is **optional**. The combination of global and/or local root signatures associated with any given shader function must define all resource bindings declared by the shader (with no overlap across global and local root signatures).

If any given function in a call graph (not counting calls across shader tables) is associated with a particular local root signature, any other functions in the graph must either be associated with the same local root signature or none, and the shader entry (the root of the call graph) must be associated with the local root signature.

This corresponds to the fact that the set of code reachable from a given shader entry gets invoked from a shader identifier in a shader record, where a single set of local root arguments apply. Of course different shaders can use different local root signatures (or none), as their shader identifiers will be in different shader records.

Member	Definition

Member	Definition
`ID3D12RootSignature* pLocalRootSignature`	Root signature that will function as a local root signature. State object holds a reference.

D3D12_DXIL_LIBRARY_DESC

```
typedef struct D3D12_DXIL_LIBRARY_DESC
{
    D3D12_SHADER_BYTECODE DXILLibrary;
    UINT NumExports;
    _In_reads_(NumExports) D3D12_EXPORT_DESC* pExports;
} D3D12_DXIL_LIBRARY_DESC;
```

Member	Definition
`D3D12_SHADER_BYTECODE DXILLibrary`	Library to include in the state object. Must have been compiled with library target 6.3 or higher. It is fine to specify the same library multiple times either in the same state object / collection or across multiple, as long as the names exported each time don't conflict in a given state object.
`UINT NumExports`	Size of pExports array. If 0, everything gets exported from the library.
`_In_reads(NumExports) D3D12_EXPORT_DESC* pExports`	Optional exports array. See D3D12_EXPORT_DESC.

D3D12_EXPORT_DESC

```
typedef struct D3D12_EXPORT_DESC
{
    LPCWSTR Name;
    _In_opt_ LPCWSTR ExportToRename;
    D3D12_EXPORT_FLAGS Flags;
} D3D12_EXPORT_DESC;
```

Member	Definition

Member	Definition
`LPWSTR Name`	Name to be exported. If the name refers to a function that is overloaded, a mangled version of the name (function parameter information encoded in name string) can be provided to disambiguate which overload to use. The mangled name for a function can be retrieved from HLSL compiler reflection (not documented in this spec). If ExportToRename field is non-null, Name refers to the new name to use for it when exported. In this case Name must be an unmangled name, whereas ExportToRename can be either a mangled or unmangled name. A given internal name may be exported multiple times with different renames (and/or not renamed).
`_In_opt_ LPWSTR ExportToRename`	If non-null, this is the name of an export to use but then rename when exported. Described further above.
`D3D12_EXPORT_FLAGS Flags`	Flags to apply to the export.

D3D12_EXPORT_FLAGS

```
typedef enum D3D12_EXPORT_FLAGS
{
    D3D12_EXPORT_FLAG_NONE = 0x0,
} D3D12_EXPORT_FLAGS;
```

No export flags are currently defined.

D3D12_EXISTING_COLLECTION_DESC

```
typedef struct D3D12_EXISTING_COLLECTION_DESC
{
    ID3D12StateObject* pExistingCollection;
    UINT NumExports;
    _In_reads_(NumExports) D3D12_EXPORT_DESC* pExports;
} D3D12_EXISTING_COLLECTION_DESC;
```

Member	Definition
`ID3D12StateObject* pExistingCollection`	Collection to include in state object. Enclosing state object holds a ref on the existing collection.

Member	Definition
`UINT NumExports`	Size of pExports array. If 0, all of the collection's exports get exported.
`_In_reads(NumExports) D3D12_EXPORT_DESC* pExports`	Optional exports array. See D3D12_EXPORT_DESC.

D3D12_HIT_GROUP_DESC

```
typedef struct D3D12_HIT_GROUP_DESC
{
    LPWSTR HitGroupExport;
    D3D12_HIT_GROUP_TYPE Type;
    _In_opt_ LPWSTR AnyHitShaderImport;
    _In_opt_ LPWSTR ClosestHitShaderImport;
    _In_opt_ LPWSTR IntersectionShaderImport;
} D3D12_HIT_GROUP_DESC;
```

Member	Definition
`LPWSTR HitGroupExport`	Name to give to hit group.
`D3D12_HIT_GROUP_TYPE Type`	See D3D12_HIT_GROUP_TYPE.
`_In_opt_ LPWSTR AnyHitShaderImport`	Optional name of anyhit shader. Can be used with all hit group types.
`_In_opt_ LPWSTR ClosestHitShaderImport`	Optional name of closesthit shader. Can be used with all hit group types.
`_In_opt_ LPWSTR IntersectionShaderImport`	Optional name of intersection shader. Can only be used with hit groups of type procedural primitive.

D3D12_HIT_GROUP_TYPE

```
typedef enum D3D12_HIT_GROUP_TYPE
{
    D3D12_HIT_GROUP_TYPE_TRIANGLES = 0x0,
    D3D12_HIT_GROUP_TYPE_PROCEDURAL_PRIMITIVE = 0x1,
} D3D12_HIT_GROUP_TYPE;
```

Specify the type of hit group. For triangles, the hit group can't contain an intersection shader. For procedural primitives, the hit group must contain an intersection shader.

> This enum exists to allow the possibility in the future of having other hit group types which may not otherwise be distinguishable from the other members of `D3D12_HIT_GROUP_DESC`. For instance, a new type might simply change the meaning of the intersection shader import to represent a different formulation of procedural primitive.

D3D12_RAYTRACING_SHADER_CONFIG

```
typedef struct D3D12_RAYTRACING_SHADER_CONFIG
{
  UINT MaxPayloadSizeInBytes;
  UINT MaxAttributeSizeInBytes;
} D3D12_RAYTRACING_SHADER_CONFIG;
```

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

A raytracing pipeline needs one raytracing shader configuration. If multiple shader configurations are present (such as one in each collection to enable independent driver compilation for each one) they must all match when combined into a raytracing pipeline.

Member	Definition
`UINT MaxPayloadSizeInBytes`	The maximum storage for scalars (counted as 4 bytes each) in ray payloads in raytracing pipelines that contain this program. Callable shader payloads are not part of this limit. This field is ignored for payloads that use payload access qualifiers.
`UINT MaxAttributeSizeInBytes`	The maximum number of scalars (counted as 4 bytes each) that can be used for attributes in pipelines that contain this shader. The value cannot exceed D3D12_RAYTRACING_MAX_ATTRIBUTE_SIZE_IN_BYTES.

D3D12_RAYTRACING_PIPELINE_CONFIG

```
typedef struct D3D12_RAYTRACING_PIPELINE_CONFIG
{
    UINT MaxTraceRecursionDepth;
} D3D12_RAYTRACING_PIPELINE_CONFIG;
```

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

A raytracing pipeline needs one raytracing pipeline configuration. If multiple shader configurations are present (such as one in each collection to enable independent driver compilation for each one) they must all match when combined into a raytracing pipeline.

Member	Definition
`UINT MaxTraceRecursionDepth`	Limit on ray recursion for the raytracing pipeline. See Ray recursion limit.

D3D12_RAYTRACING_PIPELINE_CONFIG1

```
typedef struct D3D12_RAYTRACING_PIPELINE_CONFIG1
{
    UINT MaxTraceRecursionDepth;
    D3D12_RAYTRACING_PIPELINE_FLAGS Flags;
} D3D12_RAYTRACING_PIPELINE_CONFIG1;
```

This adds a Flags field relative to its predecessor, D3D12_RAYTRACING_PIPELINE_CONFIG. `D3D12_RAYTRACING_PIPELINE_CONFIG1` requires Tier 1.1 raytracing support.

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

A raytracing pipeline needs one raytracing pipeline configuration. If multiple shader configurations are present (such as one in each collection to enable independent driver compilation for each one) they must all match when combined into a raytracing pipeline.

Member	Definition
`UINT MaxTraceRecursionDepth`	Limit on ray recursion for the raytracing pipeline. See Ray recursion limit.
`D3D12_RAYTRACING_PIPELINE_FLAGS Flags`	See D3D12_RAYTRACING_PIPELINE_FLAGS.

D3D12_RAYTRACING_PIPELINE_FLAGS

```
typedef enum D3D12_RAYTRACING_PIPELINE_FLAGS
{
    D3D12_RAYTRACING_PIPELINE_FLAG_NONE                         = 0x0,
    D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_TRIANGLES               = 0x100,
    D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_PROCEDURAL_PRIMITIVES   = 0x200,
```

```
} D3D12_RAYTRACING_PIPELINE_FLAGS;
```

Flags member of D3D12_RAYTRACING_PIPELINE_CONFIG1.

Value	Definition
D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_TRIANGLES	For any TraceRay() call wi raytracing pipeline, add in the RAY_FLAG_SKIP_TRIANGLES Ray The resulting combination of must be valid. The presence c flag in a raytracing pipeline c does not show up in a RayFla from a shader. Implementatic be able to optimize pipelines that a particular primitive typ not be considered.
D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_PROCEDURAL_PRIMITIVES	For any TraceRay() call wi raytracing pipeline, add in the RAY_FLAG_SKIP_PROCEDURAL_PR Ray flag. The resulting combi ray flags must be valid. The p of this flag in a raytracing pip config does not show up in a RayFlags() call from a shader. Implementations may be able optimize pipelines knowing th particular primitive type need considered.

D3D12_NODE_MASK

```
typedef struct D3D12_NODE_MASK
{
    UINT NodeMask;
} D3D12_NODE_MASK;
```

This is a subobject type that can be associated with shader exports. A summary of the rules about which subobject types can/must be associated with various shaders in a state object is here.

The node mask subobject identifies which GPU nodes the state object applies to. It is optional; in its absence the state object applies to all available nodes. If a node mask subobject has been associated with any part of a state object, a node mask association must be made to all exports in a state object (including imported collections) and all node mask subobjects that are referenced must have matching content.

D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIATION

```
typedef struct D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIATION
{
  const D3D12_STATE_SUBOBJECT* pSubobjectToAssociate;
  UINT NumExports;
  _In_reads_(NumExports) LPCWSTR* pExports;
} D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIATION;
```

Associates a subobject defined directly in a state object with shader exports. Depending on the choice of flags in the optional D3D12_STATE_OBJECT_CONFIG subobject for opting in to cross linkage, the exports being associated don't necessarily have to be present in the current state object (or one that has been seen yet) – to be resolved later, e.g. on RTPSO creation. See Subobject association behavior for detail.

Member	Definition
`const D3D12_STATE_SUBOBJECT* pSubobjectToAssociate`	Pointer to subobject in current state object to define an association to.
`UINT NumExports`	Size of export array. If 0, this is being explicitly defined as a default association. See Subobject association behavior. Another way to define a default association is to omit this subobject association for that subobject completely.
`_In_reads_(NumExports) LPCWSTR* pExports`	Exports to associate subobject with.

D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION

```
typedef struct D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION
{
    LPWCSTR pDXILSubobjectName;
    UINT NumExports;
    _In_reads_(NumExports) LPCWSTR* pExports;
} D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASSOCIATION;
```

Associates a subobject defined in a DXIL library (not necessarily one that has been seen yet) with shader exports. See Subobject association behavior for details.

Member	Definition
`LPWSTR pDXILSubobjectName`	Name of subobject defined in a DXIL library.
`UINT NumExports`	Size of export array. If 0, this is being explicitly defined as a default association. See Subobject association behavior. Another way to define a default association is to omit this subobject association for that subobject completely.
`_In_reads_(NumExports) LPCWSTR* pExports`	Exports to associate subobject with.

AddToStateObject

```
HRESULT AddToStateObject(
    _In_ const D3D12_STATE_OBJECT_DESC* pAddition,
    _In_ ID3D12StateObject* pStateObjectToGrowFrom,
    _In_ REFIID riid, // ID3D12StateObject
    _COM_Outptr_ void** ppNewStateObject
    );
```

Incrementally add to an existing state object. This incurs lower CPU overhead than creating a state object from scratch that is a superset of an existing one (e.g. adding a few more shaders). The reasons for lower overhead are:

- The D3D runtime doesn't have to re-validate the correctness of the existing state object. It only needs to check that what is being added is valid and doens't conflict with what is already in the state object.
- The driver ideally only needs to compile the shaders that have been added, and even that is avoided if what is being added happens to be an existing collection. In a clean driver implementation, there need only be a lightweight high level link step for the overall the state object.

> It wasn't deemed worth the effort or complexity to support incremental deletion, i.e. DeleteFromStateObject(). If an app finds that state object memory footprint is such a problem that it needs to periodically trim by shrinking state objects, it has to create the desired smaller state objects from scratch. In this case, if existing collections are used to piece together the smaller state object, at least driver overhead will be reduced, if not runtime overhead of parsing/validating the new state object as a whole.

Since `AddToStateObject` returns a new state object, this enables the application to free the original object when it is no longer needed (and no longer referenced by in flight work). A byproduct is that it is valid to create a branching lineage of state objects via `AddToStateObject`, even though the common case is likely only a single line of inheritance. A given state object only exposes exports that it added plus what it's parent exposed, and not what any siblings do. Shader identifiers from a parent remain valid for it's children, so do not need to be re-retrieved.

State object lifetimes as seen by driver is a discussion useful for driver authors.

The runtime uses a shared mutex across all state objects within a family to enforce thread safety: The `AddToStateObject` API takes a writer lock on state objects within a family. And APIs that retrieve shader export based information, GetShaderIdentifier() and GetShaderStackSize(), take a shared reader lock on state objects in a family.

> A straightforward way to minimize any thread contention is to retrieve any needed shader identifiers etc. from a state object right after it is created (and remembering that any other state objects created from this one share identifiers so they need not be re-retrieved). So that by the time `AddToStateObject` needs to be called, nothing will be at risk of blocking (unless `AddToStateObject` itself is called in parallel with related state objects for some reason).

The new state object starts off with the same pipeline stack size setting as the previous. (The runtime doesn't try to enforce thread safety against the app calling SetPipelineStackSize() to change stack size on the existing state object while it is being used in an `AddToStateObject()` call.) After `AddToStateObject()` returns a new state object, the app can call GetShaderStackSize() on newly added shaders to decide if the pipeline stack size for the new state object needs to be updated via SetPipelineStackSize().

There are some stipulations about how the additions to a state object must relate to what exists already in the state object (validated by the runtime):

- The state object description must be fully self-contained, e.g. not reference any contents of the existing state object. Said another way, the description must be valid to have been sent to CreateStateObject() on its own.
- Global subobjects definitions, such as D3D12_RAYTRACING_PIPELINE_CONFIG, must be consistent with how they are defined in the existing state object (e.g. same MaxTraceRecursionDepth).
- The original state object and the portion being added must both opt-in to being used with AddToStateObject(). This is accomplished by specifying the flag `D3D12_STATE_OBJECT_FLAG_ALLOW_STATE_OBJECT_ADDITIONS` as part of the flags member of D3D12_STATE_OBJECT_CONFIG. See flags for more detail such as what `D3D12_STATE_OBJECT_FLAG_ALLOW_STATE_OBJECT_ADDITIONS` means for collection state objects.

- Any exported shader entrypoints (i.e. that support shader identifiers) must have unique names that do not conflict with exports from the existing state object.

- Non shader entrypoints, such as library functions, are allowed to be defined over again (reusing a name with same or different code definition) relative to what already in the state object - necessary if they need to be visible locally. The same is true for any of the various subobjects that are defined.

> Disallowing cross-visibility between the existing state object and what is being added offers several benefits. It simplifies the incremental compilation burden on the driver, e.g. it isn't forced to touch existing compiled code. It avoids messy semantic issues like the presence of a new subobject affecting default associations that may have applied to the existing state object (if cross visibility were possible). Finally it simplifies runtime validation code complexity.

Parameter	Definition
`const D3D12_STATE_OBJECT_DESC* pAddition`	Description of state object contents to add to existing state object. See D3D12_STATE_OBJECT_DESC. To help generate this see the `CD3D12_STATE_OBJECT_DESC` helper in class in d3dx12.h.
`ID3D12StateObject* pStateObjectToGrowFrom`	Existing state object, which can be in use (e.g. active raytracing) during this operation. The existing state object must **not** be of type Collection - it is deemed too complex to bother defining behavioral semantics for this.
`REFIID riid`	__uuidof(ID3D12StateObject)
`_COM_Outptr_ void** ppNewStateObject`	Returned state object. Behavior is undefined if shader identifiers are retrieved for new shaders from this call and they are accessed via shader tables by any already existing or in flight command list that references some older state object. Use of the new shaders added to the state object can only occur from commands (such as DispatchRays or ExecuteIndirect calls) recorded in a command list **after** the call to `AddToStateObject`.
`Return: HRESULT`	`S_OK` for success. `E_INVALIDARG`, `E_OUTOFMEMORY` on failure. The debug layer provides detailed status information.

GetRaytracingAccelerationStructurePrebuildInfo

```
void GetRaytracingAccelerationStructurePrebuildInfo(
    _In_ const D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS* pDesc,
    _Out_ D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO *pInfo);
```

Query the driver for resource requirements to build an acceleration structure. The input acceleration structure description is the same as what goes into BuildRaytracingAccelerationStructure(). The result of this function lets the application provide the correct amount of output storage and scratch storage to BuildRaytracingAccelerationStructure() given the same geometry.

Builds can also be done with the same configuration passed to GetAccelerationStructurePrebuildInfo() overall except equal or smaller counts for any of: number of geometries/instances and number of vertices/indices/AABBs in any given geometry. In this case the storage requirements reported with the original sizes passed to GetRaytracingAccelerationStructurePrebuildInfo() will be valid – the build may actually consume less space but not more. This is handy for app scenarios where having conservatively large storage allocated for acceleration structures is fine.

This method is on the device as opposed to command list on the assumption that drivers must be able to calculate resource requirements for an acceleration structure build from only looking at the CPU visible portions of the call, without having to dereference any pointers to GPU memory containing actual vertex data, index data etc.

Parameter	Defi
`const D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS* pDesc`	Description of the acceleratic D3D12_BUILD_RAYTRACING_ACC This structure is shared with Build The implementation is parameters in this struct and nest inspect/dereference any GPU virtu to see if a pointer is NULL or not, D3D12_RAYTRACING_GEOMETRY dereferencing it. In othe resource requirements for the acc depend on the actual geometry d rather it can only depend on over of triangles, number of instances
`D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO* pInfo`	Result (D3D12_RAYTRACING_ACCELERAT

Parameter	of query. Defi

GetRaytracingAccelerationStructurePrebuildInfo Structures

In addition to below, see BuildRaytracingAccelerationStructure() for other structures (common to both APIs).

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO
{
    UINT64 ResultDataMaxSizeInBytes;
    UINT64 ScratchDataSizeInBytes;
    UINT64 UpdateScratchDataSizeInBytes;
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO;
```

Member	Definition
`UINT64 ResultDataMaxSizeInBytes`	Size required to hold the result of an acceleration structure b based on the specified inputs.
`UINT64 ScratchDataSizeInBytes`	Scratch storage on GPU required during acceleration structur based on the specified inputs.
`UINT64 UpdateScratchDataSizeInBytes`	Scratch storage on GPU required during an acceleration structure update based on the specified inputs. This only nee be called for the original acceleration structure build, and def the scratch storage requirement for every acceleration structu update (other than the initial build). If the `D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_ALLOW_` flag is not specified, this parameter returns 0.

CheckDriverMatchingIdentifier

```
D3D12_DRIVER_MATCHING_IDENTIFIER_STATUS
CheckDriverMatchingIdentifier(
```

```
    _In_ D3D12_SERIALIZED_DATA_TYPE SerializedDataType,
    _In_ const D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER* pIdentifierToCheck);
```

Suppose an app has data that has been serialized by a driver. In particular, the data is a serialized raytracing acceleration structure resulting from a call to CopyRaytracingAccelerationStructure() with mode `D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_SERIALIZE`, likely from a previous execution of the application. CheckDriverMatchingIdentifier() reports the compatibility of the serialized data with the current device/driver.

Parameter	Definition
`D3D12_SERIALIZED_DATA_TYPE SerializedDataType`	See D3D12_SERIALIZED_DATA_TYPE.
`const D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER* pIdentifierToCheck`	Identifier from the header of the serializ with the driver. See D3D12_SERIALIZED_DATA_DRIVER_MAT
`Return: D3D12_DRIVER_MATCHING_IDENTIFIER_STATUS`	See D3D12_DRIVER_MATCHING_IDENTI

CheckDriverMatchingIdentifier Structures

D3D12_SERIALIZED_DATA_TYPE

```
typedef enum D3D12_SERIALIZED_DATA_TYPE
{
    D3D12_SERIALIZED_DATA_RAYTRACING_ACCELERATION_STRUCTURE = 0x0,
} D3D12_SERIALIZED_DATA_TYPE;
```

Type of serialized data. At the moment there is only one:

Value	Definition
`D3D12_SERIALIZED_DATA_RAYTRACING_ACCELERATION_STRUCTURE`	Serialized data contains a raytracing acceleration structure.

D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER

```
typedef struct D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER
{
    GUID DriverOpaqueGUID;
```

```
    BYTE DriverOpaqueVersioningData[16];
} D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER;
```

Opaque data structure describing driver versioning for a serialized acceleration structure. This is a member of the header for serialized acceleration structure, D3D12_SERIALIZED_ACCELERATION_STRUCTURE_HEADER. Passing this identifier into CheckDriverMatchingIdentifier() tells an app if a previously serialized acceleration structure is compatible with the current driver/device, and can therefore be deserialized and used for raytracing.

D3D12_DRIVER_MATCHING_IDENTIFIER_STATUS

```
typedef enum D3D12_DRIVER_MATCHING_IDENTIFIER_STATUS
{
    D3D12_DRIVER_MATCHING_IDENTIFIER_COMPATIBLE_WITH_DEVICE = 0x0,
    D3D12_DRIVER_MATCHING_IDENTIFIER_UNSUPPORTED_TYPE = 0x1,
    D3D12_DRIVER_MATCHING_IDENTIFIER_UNRECOGNIZED = 0x2,
    D3D12_DRIVER_MATCHING_IDENTIFIER_INCOMPATIBLE_VERSION = 0x3,
    D3D12_DRIVER_MATCHING_IDENTIFIER_INCOMPATIBLE_TYPE = 0x4,
} D3D12_DRIVER_MATCHING_IDENTIFIER_STATUS;
```

Return value for CheckDriverMatchingIdentifier().

Value	Definiti
D3D12_DRIVER_MATCHING_IDENTIFIER_COMPATIBLE_WITH_DEVICE	Serialized data is compatible wi device/driver.
D3D12_DRIVER_MATCHING_IDENTIFIER_UNSUPPORTED_TYPE	D3D12_SERIALIZED_DATA_TYPE unsupported.
D3D12_DRIVER_MATCHING_IDENTIFIER_UNRECOGNIZED	Format of the data in D3D12_SERIALIZED_DATA_DRIV is unrecognized. This could indi the identifier was produced by a vendor.
D3D12_DRIVER_MATCHING_IDENTIFIER_INCOMPATIBLE_VERSION	Serialized data is recognized (lik hardware vendor), but its versio the current driver.
D3D12_DRIVER_MATCHING_IDENTIFIER_INCOMPATIBLE_TYPE	D3D12_SERIALIZED_DATA_TYPE is not compatible with the type as there is only a single defined error cannot not be produced.

CreateCommandSignature

```
HRESULT ID3D12Device::CreateCommandSignature(
    const D3D12_COMMAND_SIGNATURE_DESC* pDesc,
    ID3D12RootSignature* pRootSignature,
    REFIID riid, // Expected: ID3D12CommandSignature
    void** ppCommandSignature
);
```

This isn't a raytracing specific API, just the generic D3D API for creating command signatures to be used in ExecuteIndirect(). What is shown in this section are additions to enable indirect DispatchRays() calls from an indirect arguments buffer.

In particular the, D3D12_COMMAND_SIGNATURE_DESC field can be set up to enable DispatchRays.

CreateCommandSignature Structures

D3D12_COMMAND_SIGNATURE_DESC

```
typedef struct D3D12_COMMAND_SIGNATURE_DESC
{
    // The number of bytes between each drawing structure
    UINT ByteStride;
    UINT NumArgumentDescs;
    const D3D12_INDIRECT_ARGUMENT_DESC *pArgumentDescs;
    UINT NodeMask;
} D3D12_COMMAND_SIGNATURE_DESC;
```

Struct for defining a command signature via CreateCommandSignature(). The relevance to raytracing is the D3D12_INDIRECT_ARGUMENT_DESC array can have an entry for DispatchRays parameters as its last entry (and is mutually exclusive to the use of Draw or Dispatch arguments in a command signature).

D3D12_INDIRECT_ARGUMENT_DESC

```
typedef struct D3D12_INDIRECT_ARGUMENT_DESC
{
    D3D12_INDIRECT_ARGUMENT_TYPE Type;
    ...
} D3D12_INDIRECT_ARGUMENT_DESC;
```

Struct for defining an indirect argument in D3D12_COMMAND_SIGNATURE_DESC. Only showing the relevant field for raytracing, D3D12_INDIRECT_ARGUMENT_TYPE, which has an entry for DispatchRays.

D3D12_INDIRECT_ARGUMENT_TYPE

```
typedef enum D3D12_INDIRECT_ARGUMENT_TYPE
{
    ...
    D3D12_INDIRECT_ARGUMENT_TYPE_DISPATCH_RAYS,
    ...
} D3D12_INDIRECT_ARGUMENT_TYPE;
```

Parameter type for D3D12_INDIRECT_ARGUMENT_DESC. Only showing the relevant value for raytracing DispatchRays here:

Value	Definition
D3D12_INDIRECT_ARGUMENT_TYPE_DISPATCH_RAYS	DispatchRays argument in command signature. Each command in the indirect argument buffer includes D3D12_DISPATCH_RAYS_DESC values for the current indirect argument. When this argument is used, the command signature can't change vertex buffer or index buffer bindings (as those are related to the Draw pipeline only).

Command list methods

For all command list methods, at command list recording the runtime makes a deep copy of the parameters (not including data in GPU memory pointed to via GPU virtual addresses). So the application's CPU memory for the parameters is no longer referenced when the call returns. When the commands actually execute on the GPU timeline any GPU memory identified by GPU virtual addresses gets accessed, giving freedom for the application to change that memory independent of command list recording time.

BuildRaytracingAccelerationStructure

```
void BuildRaytracingAccelerationStructure(
    _In_ const D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC* pDesc,
```

```
    _In_ UINT NumPostbuildInfoDescs,
    _In_reads_opt_(NumPostbuildInfoDescs)
        const D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC* pPostbuildInfoD
);
```

Perform an acceleration structure build on the GPU. Also optionally output postbuild information immediately after the build.

> This postbuild information can also be obtained separately from an already built acceleration structure via EmitRaytracingAccelerationStructurePostbuildInfo(). The advantage of generating postbuild info along with a build is that a barrier isn't needed in between the build completing and requesting postbuild information, for the case where an app knows it needs the postbuild info right away.

See Geometry and acceleration structures for an overview.

See Acceleration structure properties for a discussion of rules and determinism.

Also see General tips for building acceleration structures.

Can be called on graphics or compute command lists but not from bundles.

Parameter	
`const D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC* pDesc`	Description of the accelerati D3D12_BUILD_RAYTRACING
`UINT NumPostbuildInfoDescs`	Size of postbuild info desc a
`const D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC* pPostbuildInfoDescs`	Optional array of descriptio properties of the acceleratic D3D12_RAYTRACING_ACCEI given postbuild info type,D3D12_RAYTRACING_A can only be selected for out

BuildRaytracingAccelerationStructure Structures

D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC

```
typedef struct D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC
{
    D3D12_GPU_VIRTUAL_ADDRESS DestAccelerationStructureData;
    D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS Inputs;
    _In_opt_ D3D12_GPU_VIRTUAL_ADDRESS SourceAccelerationStructureData;
    D3D12_GPU_VIRTUAL_ADDRESS ScratchAccelerationStructureData;
} D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC;
```

Member	
`D3D12_GPU_VIRTUAL_ADDRESS DestAccelerationStructureData`	Location to store resulting acc GetRaytracingAccelerationStructure required for the result here given a The address must be alig (D3D12_RAYTRACING_ACCELERATIC The memory pointed to must be in D3D12_RESOURCE_STATE_RAYTRAC
`D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS Inputs`	Description of the input data for the in its own struct since it is shared w GetRaytracingAccelerationStructure
`D3D12_GPU_VIRTUAL_ADDRESS SourceAccelerationStructureData`	Address of an existing accelera update (incremental build) is being D3D12_RAYTRACING_ACCELERATIO in the Flags parameter. Otherwise th address is the same as DestAccelera performed in-place. Any other form memory is invalid and produces un aligned to 256 bytes (D3D12_RAYTRACING_ACCELERATIC somewhat redundant requirement a here would have already been requ The memory pointed to m D3D12_RESOURCE_STATE_RAYTRAC
`D3D12_GPU_VIRTUAL_ADDRESS`	Location where the build will st data.GetRaytracingAccelerationStru scratch memory the implementatio structure build parameters. <p

ScratchAccelerationStructureData	(D3D12_RAYTRACING_ACCELERATIO Contents of this memory going intc
	will not be preserved. After the buil is left with whatever undefined cont memory pointed to must be in state

D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS

```
typedef struct D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS
{
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE Type;
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS Flags;
    UINT NumDescs;
    D3D12_ELEMENTS_LAYOUT DescsLayout;
    union
    {
        D3D12_GPU_VIRTUAL_ADDRESS InstanceDescs;
        const D3D12_RAYTRACING_GEOMETRY_DESC* pGeometryDescs;
        const D3D12_RAYTRACING_GEOMETRY_DESC*const* ppGeometryDescs;
    };
} D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS;
```

This structure is used by both BuildRaytracingAccelerationStructure() and GetRaytracingAccelerationStructurePrebuildInfo().

For GetRaytracingAccelerationStructurePrebuildInfo(), which isn't actually doing a build, any parameter that is referenced via `D3D12_GPU_VIRTUAL_ADDRESS` (in GPU memory), like InstanceDescs, will not be accessed by the operation. So this memory does not need to be initialized yet or be in a particular resource state. Whether GPU addresses are null or not *can* be inspected by the operation, even though the pointers are not dereferenced.

Member	Definition
`D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE Type`	Type of acceleration structure to buil D3D12_RAYTRACING_ACCELERATION
`D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS Flags`	`D3D12_RAYTRACING_ACCELERATION_STRU` use for the build.
`UINT NumDescs`	<p> If Type is `D3D12_RAYTRACING_ACCELERATION_STRU` number of instances (laid out based <p>If Type is `D3D12_RAYTRACING_ACCELERATION_STRU` number of elements pGeometryDesc refer to (which one is used depends

Member	Definition
`D3D12_ELEMENTS_LAYOUT DescsLayout`	How geometry descs are specified (s D3D12_ELEMENTS_LAYOUT): an array of pointers to descs.
`const D3D12_GPU_VIRTUAL_ADDRESS InstanceDescs`	If Type is `D3D12_RAYTRACING_ACCELERATION_STRU` refers to NumDescs D3D12_RAYTRA(structures in GPU memory describing instance must be aligned to 16 bytes (D3D12_RAYTRACING_INSTANCE_DE If DescLayout is `D3D12_ELEI` InstanceDescs points to an array of i memory. If DescLayout is `D3D12_ELEMENTS_LAYOUT_ARRAY_OF_POI` points to an array in GPU memory of `D3D12_GPU_VIRTUAL_ADDRESS` pointers If Type is not `D3D12_RAYTRACING_ACCELERATION_STRU` parameter is unused (space repurpos The memory pointed to must be `D3D12_RESOURCE_STATE_NON_PIXEL_SHA`
`const D3D12_RAYTRACING_GEOMETRY_DESC* pGeometryDescs`	If Type is `D3D12_RAYTRACING_ACCELERATION_STRU` and DescsLayout is `D3D12_ELEMENTS_` is used and points to NumDescs con D3D12_RAYTRACING_GEOMETRY_DE CPU describing individual geometrie not `D3D12_RAYTRACING_ACCELERATION_STRU` DescsLayout is not `D3D12_ELEMENTS_L`

Member	Definition
pGeometryDescs	parameter is unused (space repurpos *The reason pGeometryDescs is a as opposed to InstanceDescs which liv least for initial implementations, the some of the information such as trian pGeometryDescs in order to schedule builds. Perhaps in the future more of GPU.*
`const D3D12_RAYTRACING_GEOMETRY_DESC** ppGeometryDescs`	If Type is `D3D12_RAYTRACING_ACCELERATION_STRU` and DescsLayout is `D3D12_ELEMENTS_LAYOUT_ARRAY_OF_POI` used and points to an array of NumD D3D12_RAYTRACING_GEOMETRY_DE CPU describing individual geometrie not `D3D12_RAYTRACING_ACCELERATION_STRU` DescsLayout is not `D3D12_ELEMENTS_LAYOUT_ARRAY_OF_POI` is unused (space repurposed in a uni *ppGeometryDescs is a CPU based same reason as pGeometryDescs desc difference is this option lets the app h geometry descs if desired.*

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE

```
typedef enum D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE
{
  D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL = 0x0,
  D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL = 0x1
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE;
```

Value	Definition
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL	Top-level acceleration structure.
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL	Bottom-level acceleration structure.

Descriptions of these types are at Geometry and acceleration structures and visualized in Ray-geometry interaction diagram.

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS

```
typedef enum D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS
{
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_NONE = 0x00,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_ALLOW_UPDATE =0x01,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_ALLOW_COMPACTION= 0x02,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PREFER_FAST_TRACE= 0x04,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PREFER_FAST_BUILD= 0x08,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_MINIMIZE_MEMORY= 0x10,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PERFORM_UPDATE= 0x20,
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS;
```

Member	
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_NONE	No options specif
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_ALLOW_UPDATE	Build the acc D3D12_RAYTRACING having to entirely times and lower r building the equiv on an initial accel structure specifiec been built withou updates.
	Enables the o CopyRaytracingAc D3D12_RAYTRACI result in increasec the resulting acce larger) than buildi ALLOW_COMPACTION specifying ALLOW_

Member	
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_ALLOW_COMPACTION	finally compactinç The size requ compaction via Er D3D12_RAYTRACI in particular for a This flag structure update, other words as so other acceleration ALLOW_COMPACTION *might apply to sor certain informatio only help so much structure will no lo application could acceleration struct*
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PREFER_FAST_TRACE	Construct a h the expense of ad take about 2-3 tin This flag all other flags exc D3D12_RAYTRACING
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PREFER_FAST_BUILD	Construct a lo speed. A rough ru time than default other flags except D3D12_RAYTRACING
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_MINIMIZE_MEMORY	Minimize the well as the size of times. Th D3D12_RAYTRACING acceleration struc initial acceleration The impact o GetRaytracingAcc

Member	GetRaytracingAcc requirements for t
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PERFORM_UPDATE	<p> Perform an a faster than a full b positions of the u acceleration struc for a discussion o the addresses of t performed in-plac invalid. For non-o unmodified. The r the input accelera specified ALLOW_U selections, aside f acceleration struc succession, as lon D3D12_RAYTRACING update build cont

D3D12_ELEMENTS_LAYOUT

```
typedef enum D3D12_ELEMENTS_LAYOUT
{
    D3D12_ELEMENTS_LAYOUT_ARRAY = 0x0,
    D3D12_ELEMENTS_LAYOUT_ARRAY_OF_POINTERS = 0x1
} D3D12_ELEMENTS_LAYOUT;
```

Given a data set of n elements, describes how the locations of the elements are identified.

Member	Definition
D3D12_ELEMENTS_LAYOUT_ARRAY	For a data set of n elements, the pointer parameter simply points to the start of an of n elements in memory.
D3D12_ELEMENTS_LAYOUT_ARRAY_OF_POINTERS	For a data set of n elements, the pointer parameter points to an array of n pointers in memory, each pointing to an individual element of the set.

D3D12_RAYTRACING_GEOMETRY_DESC

```
typedef struct D3D12_RAYTRACING_GEOMETRY_DESC
{
    D3D12_RAYTRACING_GEOMETRY_TYPE Type;
    D3D12_RAYTRACING_GEOMETRY_FLAGS Flags;
    union
    {
        D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC Triangles;
        D3D12_RAYTRACING_GEOMETRY_AABBS_DESC AABBs;
    };
} D3D12_RAYTRACING_GEOMETRY_DESC;
```

Member	Definition
`D3D12_RAYTRACING_GEOMETRY_TYPE Type`	D3D12_RAYTRACING_GEOMETRY_TYPE for this g
`D3D12_RAYTRACING_GEOMETRY_FLAGS Flags`	D3D12_RAYTRACING_GEOMETRY_FLAGS for this
`D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC Triangles`	D3D12_RAYTRACING_GEOMETRY_TRIANGLES_D describing triangle geometry if Type is `D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES` . Oth parameter is unused (space repurposed in a unic
`D3D12_RAYTRACING_GEOMETRY_AABBS_DESC AABBs`	D3D12_RAYTRACING_GEOMETRY_AABBS_DESC (AABB geometry if Type is `D3D12_RAYTRACING_GEOMETRY_TYPE_PROCEDURAL_PRI` Otherwise this parameter is unused (space repur union).

D3D12_RAYTRACING_GEOMETRY_TYPE

```
typedef enum D3D12_RAYTRACING_GEOMETRY_TYPE
{
    D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES,
    D3D12_RAYTRACING_GEOMETRY_TYPE_PROCEDURAL_PRIMITIVE_AABBS
} D3D12_RAYTRACING_GEOMETRY_TYPE;
```

Value	Definitic
`D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES`	The geometry consists of trian D3D12_RAYTRACING_GEOME
`D3D12_RAYTRACING_GEOMETRY_TYPE_PROCEDURAL_PRIMITIVE_AABBS`	The geometry procedurally is raytracing by intersection sha purpose of acceleration struct geometry's bounds are descri

Value	Definitic
	bounding boxes via D3D12_RAYTRACING_GEOME

D3D12_RAYTRACING_GEOMETRY_FLAGS

```
typedef enum D3D12_RAYTRACING_GEOMETRY_FLAGS
{
    D3D12_RAYTRACING_GEOMETRY_FLAG_NONE = 0x0,
    D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE = 0x1,
    D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION = 0x2,
} D3D12_RAYTRACING_GEOMETRY_FLAGS;
```

Value	Definition
D3D12_RAYTRACING_GEOMETRY_FLAG_NONE	No options specified.
D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE	When rays encounter this geometr the geometry acts as if no any hit shader is present. It is recommend use this flag liberally, as it can enab important ray processing optimizations. Note that this behav can be overridden on a per-instanc basis with D3D12_RAYTRACING_INSTANCE_FI and on a per-ray basis using Ray fl in TraceRay() and RayQuery::TraceRayInline().
	<p>By default, the system is free t trigger an any hit shader more thar once for a given ray-primitive intersection. This flexibility helps improve the traversal efficiency of acceleration structures in certain ca For instance, if the acceleration structure is implemented internally with bounding volumes, the implementation may find it benefic

value	Definition
D3D12_RAYTRACING_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION	to store relatively dense triangles in multiple bounding boxes rather tha larger single box. Howeve some application use cases require intersections be reported to the an shader at most once. This flag enab that guarantee for the given geom potentially with some performance impact. This flag applies t geometry types.

D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC

```
typedef struct D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC
{
    D3D12_GPU_VIRTUAL_ADDRESS Transform3x4;
    DXGI_FORMAT IndexFormat;
    DXGI_FORMAT VertexFormat;
    UINT IndexCount;
    UINT VertexCount;
    D3D12_GPU_VIRTUAL_ADDRESS IndexBuffer;
    D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE VertexBuffer;
} D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC;
```

The geometry pointed to by this struct are always in triangle list from (indexed or non-indexed form). Strips are not supported for simplicity.

Member	Definition
D3D12_GPU_VIRTUAL_ADDRESS Transform3x4	Address of a 3x4 affine transform matrix in row r layout to be applied to the vertices in the VertexBuff during an acceleration structure build. The contents VertexBuffer are not modified. If a 2D vertex format i the transformation is applied with the third vertex component assumed to be zero. If Transfor NULL the vertices will not be transformed. Using Trar may result in increased computation and/or memory requirements for the acceleration structure build.</p The memory pointed to must be in state D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE . T address must be aligned to 16 bytes (D3D12_RAYTRACING_TRANSFORM3X4_BYTE_ALIGN

Member	Definition
DXGI_FORMAT IndexFormat	Format of the indices in the IndexBuffer. Must be of: DXGI_FORMAT_UNKNOWN (when IndexBuffer NULL) DXGI_FORMAT_R32_UINT DXGI_FORMAT_R16_UINT
DXGI_FORMAT VertexFormat	Format of the vertices (positions) in VertexBuffer be one of: DXGI_FORMAT_R32G32_FLOAT (third component assumed 0) DXGI_FORMAT_R32G32B32_FLOAT DXGI_FORMAT_R16G16_FLOAT (third component as 0) DXGI_FORMAT_R16G16B16A16_FLOAT (A16 component is ignored, other data can be packed the such as setting vertex stride to 6 bytes) DXGI_FORMAT_R16G16_SNORM (third component as 0) DXGI_FORMAT_R16G16B16A16_SNORM (A16 component is ignored, other data can be packed the such as setting vertex stride to 6 bytes) Tier devices support the following additional formats: DXGI_FORMAT_R16G16B16A16_UNORM (A16 compone ignored, other data can be packed there, such as set vertex stride to 6 bytes) DXGI_FORMAT_R16G16_UNORM (third component as 0) DXGI_FORMAT_R10G10B10A2_UNORM (A2 component is ignored, stride must be 4 bytes) DXGI_FORMAT_R8G8B8A8_UNORM (A8 component is ignored, other data can be packed there, such as set vertex stride to 3 bytes) DXGI_FORMAT_R8G8_U (third component assumed 0) DXGI_FORMAT_R8G8B8A8_SNORM (A8 component is

Member	Definition
	ignored, other data can be packed there, such as sett vertex stride to 3 bytes) DXGI_FORMAT_R8G8_S (third component assumed 0) *The 4 compor formats with ignored A* components were a way arou having to introduce 3 component variants of these for the DXGI format list (and associated infrastructure) on this scenario. So this just saved a minor mount of engi work given too much work to do overall.*
`UINT IndexCount`	Number of indices in IndexBuffer. Must be 0 if IndexI is NULL.
`UINT VertexCount`	Number of vertices (positions) in VertexBuffer. If an i buffer is present, this must be at least the maximum value in the index buffer + 1.
`D3D12_GPU_VIRTUAL_ADDRESS IndexBuffer`	Array of vertex indices. If NULL, triangles are nor indexed. Just as with graphics, the address must be a to the size of IndexFormat. The memory po to must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE`. N that if an app wants to share index buffer inputs betv graphics input assembler and raytracing acceleration structure build input, it can always put a resource int combination of read states simultaneously, e.g. `D3D12_RESOURCE_STATE_INDEX_BUFFER | D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE`.</
`D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE VertexBuffer`	Array of vertices including a stride. The alignme the address and stride must be a multiple of the component size, so 4 bytes for formats with 32bit components and 2 bytes for formats with 16bit components. There is no constraint on the stride (wh there is a limit for graphics), other than that the bott 32bits of the value are all that are used – the field is purely to make neighboring fields align cleanly/obvi everywhere. Each vertex position is expected to be at start address of the stride range and any excess spac ignored by acceleration structure builds. This excess might contain other app data such as vertex attribute which the app is responsible for manually fetching in

Member	Definition
	which the app is responsible for manually fetching in shaders, whether it is interleaved in vertex buffers or elsewhere. The memory pointed to must be state D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOUR Note that if an app wants to share vertex buffer inpu between graphics input assembler and raytracing acceleration structure build input, it can always put a resource into a combination of read states simultane e.g. D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFF D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE .</

D3D12_RAYTRACING_GEOMETRY_AABBS_DESC

```
typedef struct D3D12_RAYTRACING_GEOMETRY_AABBS_DESC
{
    UINT64 AABBCount;
    D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE AABBs;
} D3D12_RAYTRACING_GEOMETRY_AABBS_DESC;
```

Member	Definition
UINT AABBCount	Number of AABBs pointed to in the contiguous array at AABBs.
D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE AABBs	D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE describing the GPU memory location where an array of AABB descriptions is to be found, including the data stride between AABBs. The address and stride must each be aligned to 8 bytes (D3D12_RAYTRACING_AABB_BYTE_ALIGNMENT). The stride can be 0. The memory pointed to must be in state D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE .

D3D12_RAYTRACING_AABB

```
typedef struct D3D12_RAYTRACING_AABB
{
    FLOAT MinX;
    FLOAT MinY;
    FLOAT MinZ;
    FLOAT MaxX;
    FLOAT MaxY;
    FLOAT MaxZ;
} D3D12_RAYTRACING_AABB;
```

Member	Definition
`FLOAT MinX, MinY, MinZ`	The minimum X, Y and Z coordinates of the box.
`FLOAT MaxX, MaxY, MaxZ`	The maximum X, Y and Z coordinates of the box.

D3D12_RAYTRACING_INSTANCE_DESC

```
typedef struct D3D12_RAYTRACING_INSTANCE_DESC
{
    FLOAT Transform[3][4];
    UINT InstanceID : 24;
    UINT InstanceMask : 8;
    UINT InstanceContributionToHitGroupIndex : 24;
    UINT Flags : 8;
    D3D12_GPU_VIRTUAL_ADDRESS AccelerationStructure;
} D3D12_RAYTRACING_INSTANCE_DESC;
```

This data structure is used in GPU memory during acceleration structure build. This C++ struct definition is useful if generating instance data on the CPU first then uploading to the GPU. But apps are also free to generate instance descriptions directly into GPU memory from compute shaders for instance, following the same layout.

Member	Definition
`FLOAT Transform[3][4]`	A 3x4 transform matrix in row major layout representi to-world transformation. Implementations transform r to transforming all the geometry/AABBs.
`UINT InstanceID`	An arbitrary 24-bit value that can be accessed via Inst shader types listed in System value intrinsics.
`UINT InstanceMask`	An 8-bit mask assigned to the instance, which can be include/reject groups of instances on a per-ray basis. InstanceInclusionMask parameter in TraceRay() and RayQuery::TraceRayInline(). If the value is zero, the ins be included, so typically this should be set to some no

Member	Definition
	be included, so typically this should be set to some fi
UINT InstanceContributionToHitGroupIndex	Per-instance contribution to add into shader table ind the hit group to use. The indexing behavior is introdu Indexing into shader tables, detailed here: Addressing within shader tables, and visualized here: Ray-geomet diagram. Has no behavior with inline raytracing, howe still available to fetch from a RayQuery object for shac for any purpose.
UINT Flags	Flags from D3D12_RAYTRACING_INSTANCE_FLAGS to instance.

D3D12_GPU_VIRTUAL_ADDRESS AccelerationStructure	Address of the bottom-level acceleration structur instanced. The address must be aligned to 256 bytes (D3D12_RAYTRACING_ACCELERATION_STRUCTURE_B which is a somewhat redundant requirement as any e acceleration structure passed in here would have alre to be placed with such alignment anyway. Th pointed to must be in state D3D12_RESOURCE_STATE_RAYTRACING_ACCELERATIC

D3D12_RAYTRACING_INSTANCE_FLAGS

```
typedef enum D3D12_RAYTRACING_INSTANCE_FLAGS
{
    D3D12_RAYTRACING_INSTANCE_FLAG_NONE = 0x0,
    D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGLE_CULL_DISABLE = 0x1,
    D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGLE_FRONT_COUNTERCLOCKWISE =0x2,
    D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_OPAQUE = 0x4,
    D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_NON_OPAQUE = 0x8
} D3D12_RAYTRACING_INSTANCE_FLAGS;
```

Value	D
D3D12_RAYTRACING_INSTANCE_FLAG_NONE	No options specified.
	Disables front/back face Ray flags RAY_FLAG_CULL RAY_FLAG_CULL_FRONT_FA

D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGLE_CULL_DISABLE	effect on this instance. I precedence however - if D3D12_RAYTRACING_INSTA has no effect, and all tria
D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGLE_FRONT_COUNTERCLOCKWISE	This flag reverses fr useful if for example, the order differs from the de By default, a triangl appear clockwise from t its vertices appear count in object space in a left- Since these winding object space, they are ur transforms. For example with negative determina geometry) does not cha within the instance. Per- contrast, (defined in D3D12_RAYTRACING_GE get combined with the a space, so a negative det triangle winding.
D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_OPAQUE	The instance will act D3D12_RAYTRACING_GE been specified for all the acceleration structure re that this behavior can be RAY_FLAG_FORCE_NON_OPA to the D3D12_RAYTRACING_INSTA
D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_NON_OPAQUE	The instance will act D3D12_RAYTRACING_GE not been specified for ar bottom-level acceleratic instance. Note that this l the ray flag RAY_FLAG_FC

Value	exclusive to the D3D12_RAYTRACING_INSTA</p>

D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE

```
typedef struct D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE
{
    D3D12_GPU_VIRTUAL_ADDRESS StartAddress;
    UINT64 StrideInBytes;
} D3D12_GPU_VIRTUAL_ADDRESS_AND_STRIDE;
```

Member	Definition
UINT64 StartAddress	Beginning of a VA range.
UINT64 StrideInBytes	Defines indexing stride, such as for vertices. Only the bottom 32 bits get used. The field is 64 bits purely to make alignment of containing structures clean/obvious everywhere.

EmitRaytracingAccelerationStructurePostbuildInfo

```
void EmitRaytracingAccelerationStructurePostbuildInfo(
    _In_ const D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC* pDesc,
    _In_ UINT NumSourceAccelerationStructures,
    _In_reads_( NumSourceAccelerationStructures ) const D3D12_GPU_VIRTUAL_ADDRESS*
              pSourceAccelerationStructureData);
```

Emits post-build properties for a set of acceleration structures. This enables applications to know the output resource requirements for performing acceleration structure operations via CopyRaytracingAccelerationStructure().

Can be called on graphics or compute command lists but not from bundles.

Parameter	
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC* pDesc	Description of postbuild in
UINT NumSourceAccelerationStructures	Number of pointers to acce pSourceAccelerationStructu destination (output), which

Parameter	NumSourceAccelerationStr the structures depends on

`const D3D12_GPU_VIRTUAL_ADDRESS* pSourceAccelerationStructureData`	Pointer to array of GPU NumSourceAccelerationStr an existing acceleration stru (D3D12_RAYTRACING_ACC The memory poir D3D12_RESOURCE_STATE_I

EmitRaytracingAccelerationStructurePostbuildInfo Structures

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC
{
    D3D12_GPU_VIRTUAL_ADDRESS DestBuffer;
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TYPE InfoType;
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_DESC;
```

Description of the postbuild information to generate from an acceleration structure. This is used by both EmitRaytracingAccelerationStructurePostbuildInfo() and optionally by BuildRaytracingAccelerationStructure().

Member	Definition
`D3D12_GPU_VIRTUAL_ADDRESS DestBuffer`	Result storage. Size requ layout of the contents writte system depend on InfoType. The memory pointed to state `D3D12_RESOURCE_STATE_UNORD` The memory must be aligne

Member	Definition
	natural alignment for the me the particular output structu generated (e.g. 8 bytes for a the largest membes being U
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TYPE InfoType	Type of postbuild informatic

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TYPE

```
typedef enum D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TYPE
{
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_COMPACTED_SIZE = 0x0,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TOOLS_VISUALIZATION = 0x1,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_SERIALIZATION = 0x2,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_CURRENT_SIZE = 0x3,
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TYPE;
```

Member	
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_COMPACTED_SIZE	Space requ D3D12_RAY
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TOOLS_VISUALIZATION	Space requ See D3D12_RAY
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_SERIALIZATION	Space requ D3D12_RAY
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_CURRENT_SIZE	Size of the D3D12_RAY

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_COMPACTED_SIZE_DESC

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_COMPACTED_SIZE_DESC
{
    UINT64 CompactedSizeInBytes;
```

```
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_COMPACTED_SIZE_DESC;
```

Member	Definition
`UINT64 CompactedSizeInBytes`	Space requirement for acceleration structure after compaction. It is guaranteed that a compacted acceleration structure doesn't consume more space than a non-compacted acceleration structure. Pre-compaction, it is guaranteed that the size requirements reported by GetRaytracingAccelerationStructurePrebuildInfo() for a given build configuration (triangle counts etc.) will be sufficient to store any acceleration structure whose build inputs are reduced (e.g. reducing triangle counts). This is discussed in GetRaytracingAccelerationStructurePrebuildInfo(). This non-increasing property for smaller builds does not apply post-compaction, however. In other words, it is not guaranteed that having fewer items in an acceleration structure means it compresses to a smaller size than compressing an acceleration structure with more items.

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TOOLS_VISUALIZATION_DESC

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TOOLS_VISUALIZATION_
{
    UINT64 DecodedSizeInBytes;
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TOOLS_VISUALIZATION_DESC;
```

Member	Definition
`UINT64 DecodedSizeInBytes`	Space requirement for decoding an acceleration structure into a form that can be visualized by tools.

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_SERIALIZATION_DESC

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_SERIALIZATION_DESC
{
    UINT64 SerializedSizeInBytes;
    UINT64 NumBottomLevelAccelerationStructurePointers; // UINT64 to align arrays of this
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_SERIALIZATION_DESC;
```

Member	Definition
`UINT64 SerializedSizeInBytes	Size of the serialized acceleration structure, ir header. The header is D3D12_SERIALIZED_ACCELERATION_STRUCTU followed by followed by a list of pointers to b acceleration structures.
UINT64 NumBottomLevelAccelerationStructurePointers	How many 64bit GPUVAs will be at the st serialized acceleration structure (after D3D12_SERIALIZED_ACCELERATION_STRUCTURE_HE above). For a bottom-level acceleration struct be 0. For a top-level acceleration structure, th indicate the acceleration structures being refe When deserializing happens, these bottom level pointers must be initialized by t the serialized data (just after the header) to th locations where the bottom level acceleration will reside. These new locations pointed to at time need not have been populated with bot acceleration structures yet, as long as they ha initialized with the expected deserialized data before use in raytracing. During deserializatio reads the new pointers, using them to produc equivalent top-level acceleration structure to

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_CURRENT_SIZE_DESC

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_CURRENT_SIZE_DESC
{
    UINT64 CurrentSizeInBytes;
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_CURRENT_SIZE_DESC;
```

Member	Definition
UINT64 CurrentSizeInBytes**	Space used by current acceleration structure. If the acceleration structure hasn't had a compaction operation performed on it, this size the same one reported by GetRaytracingAccelerationStructurePrebuildInfo(), and if it has been compacted this size is the same reported for postbuild info with

Member	Definition
	D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_COMPACTED_ *While this appears redundant to other ways of querying sizes, one can be handy for tools to be able to determine how much memory occupied by an arbitrary acceleration structure sitting in memory.*

CopyRaytracingAccelerationStructure

```
void CopyRaytracingAccelerationStructure(
    _In_ D3D12_GPU_VIRTUAL_ADDRESS DestAccelerationStructureData,
    _In_ D3D12_GPU_VIRTUAL_ADDRESS SourceAccelerationStructureData,
    _In_ D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE Mode);
```

Since raytracing acceleration structures may contain internal pointers and have a device dependent opaque layout, copying them around or otherwise manipulating them requires a dedicated API so that drivers can handle the requested operation. This API takes a source acceleration structure and copies it to destination memory while applying the transformation requested by the Mode parameter.

Can be called on graphics or compute command lists but not from bundles.

Parameter	Definitic
D3D12_GPU_VIRTUAL_ADDRESS DestAccelerationStructureData	Destination memory. Required size EmitRaytracingAccelerationStructurePo necessary depending on the Mode.</p be 256 byte aligned (D3D12_RAYTRACING_ACCELERATION_ regardless of the Mode. Dest overlap source otherwise results are ur state that the memory pointed to must parameter - see D3D12_RAYTRACING_ACCELERATION_S definitions.
D3D12_GPU_VIRTUAL_ADDRESS	Acceleration structure or other typ based on the specified Mode. The data (such as if it is an acceleration structure the data pointed to (such as acceleratic data (such as acceleration structures) it a top-level acceleration structure, any b

SourceAccelerationStructureData	structures that it points to are not invo Memory must be 256 byte aligned (D3D12_RAYTRACING_ACCELERATION_ regardless of Mode. The resou pointed to must be in depends on the D3D12_RAYTRACING_ACCELERATION_S definitions.
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE Mode	Type of copy operation to perform. See D3D12_RAYTRACING_ACCELERATION_S

CopyRaytracingAccelerationStructure Structures

D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE

```
typedef enum D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE
{
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_CLONE = 0x0,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_COMPACT = 0x1,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_VISUALIZATION_DECODE_FOR_TOOLS = 0x2
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_SERIALIZE = 0x3,
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_DESERIALIZE = 0x4,
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE;
```

Value	
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_CLONE	(prese point the c sourc D3D
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_COMPACT	S sourc mem retrie - D3D other state

Value	
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_VISUALIZATION_DECODE_FOR_TOOLS	I D3D The s Emitl tools the ir descr descr roug for a conc primi numl not b geon table for to struc accel to cr mem

Value	
	D3D desti T
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_SERIALIZE	I D3D itself T a file sourc D3D desti V struc level accel accel
D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_DESERIALIZE	S head D3D secti the a level as lo accel prese resul runni least mem desti D3D

D3D12_SERIALIZED_ACCELERATION_STRUCTURE_HEADER

```
typedef struct D3D12_SERIALIZED_RAYTRACING_ACCELERATION_STRUCTURE_HEADER
{
    D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER DriverMatchingIdentifier;
    UINT64 SerializedSizeInBytesIncludingHeader;
    UINT64 DeserializedSizeInBytes;
    UINT64 NumBottomLevelAccelerationStructurePointersAfterHeader;
} D3D12_SERIALIZED_RAYTRACING_ACCELERATION_STRUCTURE_HEADER;
```

Header for a serialized raytracing acceleration structure.

Member	
`D3D12_SERIALIZED_DATA_DRIVER_MATCHING_IDENTIFIER DriverMatchingIdentifier`	See D3D12_SERIALIZED_DATA_DR
`UINT64 SerializedSizeInBytesIncludingHeader`	Size of serialized data.
`UINT64 DeserializedSizeInBytes`	Size that will be consumed when original acceleration structure bef
`UINT64 NumBottomLevelAccelerationStructurePointersAfterHeader`	Size of the array of `D3D12_GPU_VI` discussion in D3D12_RAYTRACING_ACCELERAT which also reports the same coun

D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_TOOLS_VISUALIZATION_HEADER

```
typedef struct D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_TOOLS_VISUALIZATION_HEADER
{
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE Type;
    UINT NumDescs;
} D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_TOOLS_VISUALIZATION_HEADER;

// Depending on Type field, NumDescs above is followed by either:
// D3D12_RAYTRACING_INSTANCE_DESC InstanceDescs[NumDescs]
// or D3D12_RAYTRACING_GEOMETRY_DESC GeometryDescs[NumDescs].
```

This describes the GPU memory layout of an acceleration structure visualization. It is a bit like the inverse of the inputs to an acceleration structure build, focused on simply the instance or geometry details depending on the acceleration structure type.

SetPipelineState1

```
void SetPipelineState1(_In_ ID3D12StateObject* pStateObject);
```

Set a state object on the command list. Can be called from graphics or compute command lists and bundles.

This is an alternative to SetPipelineState(*In* ID3D12PipelineState*) which is only defined for graphics and compute shaders. There is only one pipeline state active on a command list at a time, so either call sets the current pipeline state. The distinction between the calls is each sets particular types of pipeline state only. For now, at least, SetPipelineState1 is only for setting raytracing pipeline state.

Parameter	Definition
`ID3D12StateObject* pStateObject`	State object. For now this can only be of type: `D3D12_STATE_OBJECT_TYPE_RAYTRACING_PIPELINE` .

DispatchRays

```
void DispatchRays(_In_ const D3D12_DISPATCH_RAYS_DESC* pDesc);
```

Launch threads of a ray generation shader. See Initiating raytracing for an overview. Can be called from graphics or compute command lists and bundles.

A raytracing pipeline state must be set on the command list otherwise behavior of this call is undefined.

There are 3 dimensions passed in to set the grid size: width/height/depth. These dimensions are constrained such that width*height*depth <= 2^30. Exceeding this produces undefined behavior.

If any grid dimension is 0, no threads are launched.

Tier 1.1 implementations also support GPU initiated DispatchRays() via ExecuteIndirect().

Parameter	Definition
`const D3D12_DISPATCH_RAYS_DESC* pDesc`	Description of the ray dispatch. See D3D12_DISPATCH_RAYS_DESC.

DispatchRays Structures

D3D12_DISPATCH_RAYS_DESC

```
typedef struct D3D12_DISPATCH_RAYS_DESC
{
    D3D12_GPU_VIRTUAL_ADDRESS_RANGE RayGenerationShaderRecord;
    D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE MissShaderTable;
    D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE HitGroupTable;
    D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE CallableShaderTable;
    UINT Width;
    UINT Height;
    UINT Depth;
} D3D12_DISPATCH_RAYS_DESC;
```

Member	Definition
`D3D12_GPU_VIRTUAL_ADDRESS_RANGE RayGenerationShaderRecord`	Shader record for the Ray generation shader t must be in state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOU` must be aligned to 64 bytes (D3D12_RAYTRACING_SHADER_TABLE_BYTE_A
`D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE MissShaderTable`	Shader table for Miss shader. The stride is and must be aligned to 32 bytes (D3D12_RAYTRACING_SHADER_RECORD_BYTE and in the range [0...4096] bytes. Mer state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_` Address must be aligned to 64 bytes(D3D12_RAYTRACING_SHADER_TABLE_BY
`D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE HitGroupTable`	Shader table for Hit Group. The stride is re must be aligned to 32 bytes (D3D12_RAYTRACING_SHADER_RECORD_BYTE and in the range [0...4096] bytes. Me state `D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_` Address must be aligned to 64 bytes (D3D12_RAYTRACING_SHADER_TABLE_BYTE_A
	Shader table for Callable shaders. The str

Member	Definition
D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE CallableShaderTable	Shader table for Callable shaders. The str stride, and must be aligned to 32 bytes (D3D12_RAYTRACING_SHADER_RECORD_BYTE and in the range [0...4096] bytes. Mer state D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_ Address must be aligned to 64 bytes (D3D12_RAYTRACING_SHADER_TABLE_BYTE_A
UINT Width	Width of ray generation shader thread grid.
UINT Height	Height of ray generation shader thread grid.
UINT Depth	Depth of ray generation shader thread grid.

D3D12_GPU_VIRTUAL_ADDRESS_RANGE

```
typedef struct D3D12_GPU_VIRTUAL_ADDRESS_RANGE
{
    D3D12_GPU_VIRTUAL_ADDRESS StartAddress;
    UINT64 SizeInBytes;
} D3D12_GPU_VIRTUAL_ADDRESS_RANGE;
```

Member	Definition
UINT64 StartAddress	Beginning of a VA range.
UINT64 SizeInBytes	Size of a VA range.

D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE

```
typedef struct D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE
{
    D3D12_GPU_VIRTUAL_ADDRESS StartAddress;
    UINT64 SizeInBytes;
    UINT64 StrideInBytes;
} D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE;
```

Member	Definition
UINT64 StartAddress	Beginning of a VA range.
UINT64 SizeInBytes	Size of a VA range.
UINT64 StrideInBytes	Define record indexing stride within the memory range.

ExecuteIndirect

```
void ID3D12CommandList::ExecuteIndirect(
    ID3D12CommandSignature* pCommandSignature,
    UINT MaxCommandCount,
    ID3D12Resource* pArgumentBuffer,
    UINT64 ArgumentBufferOffset,
    ID3D12Resource* pCountBuffer,
    UINT64 CountBufferOffset
);
```

This isn't a raytracing specific API, just the generic D3D API for issuing indirect command execution.

What is relevant to raytracing is that `ID3D12CommandSignature` can be configured to support indirect calls to DispatchRays() - see CreateCommandSignature().

When a command signature configured for `DispatchRays` is passed to `ExecuteIndirect`, a raytracing pipeline state must be currently set on the command list otherwise behavior is undefined.

It is invalid for an application to modify the contents of shader tables from an ExecuteIndirect() call if that ExecuteIndirect() call references the same shader tables (e.g. via DispatchRays()).

ID3D12StateObjectProperties methods

ID3D12StateObjectProperties is an interface exported by ID3D12StateObject. The following methods are exposed from ID3D12StateObjectProperties:

GetShaderIdentifier

```
const void* GetShaderIdentifier(LPWSTR pExportName);
```

Retrieve the unique identifier for a shader that can be used in Shader records. This is only valid for Ray generation shaders, Hit groups, Miss shaders and Callable shaders. The state object can be a collection or raytracing pipeline state object, and the shader must not have any unresolved references or missing associations to required subobjects (in other words be fully compileable by a driver), otherwise nullptr is returned.

Parameter	Definition

Parameter	Definition
`LPWSTR pExportName`	Entrypoint in the state object for which to retrieve an identifier.
`Return value: const void*`	Returns a pointer to the shader identifier. The data pointed to is valid as long as the state object it came from is valid. The size of the data returned is D3D12_SHADER_IDENTIFIER_SIZE_IN_BYTES. Applications should copy and cache this data to avoid the cost of searching for it in the state object if it will need to be retrieved many times. The place the identifier actually gets used is in shader records within shader tables in GPU memory, which it is up to the app to populate. The data itself globally identifies the shader, so even if the shader appears in a different state object (with same associations like any root signatures) it will have the same identifier. If the shader isn't fully resolved in the state object, the return value is nullptr.

GetShaderStackSize

```
UINT64 GetShaderStackSize(LPCWSTR pExportName);
```

Retrieve the amount of stack memory required to invoke a raytracing shader in HLSL. Even ray generation shaders may return nonzero despite being at the bottom of the stack. See Pipeline stack for details on how this contributes to an app's pipeline stack size calculation. This only needs to be called if the app wants to configure stack size via SetPipelineStackSize() rather than relying on the conservative default size,

This is only valid for Ray generation shaders, Hit groups, Miss shaders and Callable shaders.

For Hit groups, stack size must be queried for the individual shaders comprising it: Intersection shaders, Any hit shaders, Closest hit shaders, as each likely has a different stack size requirement. The stack size can't be queried on these individual shaders directly, as the way they are compiled can be influenced by the overall hit group that contains them. The pExportName parameter described below includes syntax for identifying individual shaders within a hit group.

This API can be called on either collection state objects or raytracing pipeline state objects.

Parameter	Definition

Parameter	Definition
`LPCWSTR pExportName`	Shader entrypoint in the state object for which to retrieve stack size. For hit groups, an individual shader within the hit group must be specified as follows: “hitGroupName::shaderType”, where hitGroupName is the entrypoint name for the hit group and shaderType is one of: intersection, closesthit or anyhit (all case sensitive). E.g. “myTreeLeafHitGroup::anyhit”
`Return value: UINT64`	Amount of stack in bytes required to invoke the shader. If the shader isn’t fully resolved in the state object, or the shader is unknown or of a type for which a stack size isn’t relevant (such as a hit group) the return value is 0xffffffff. The reason for returning 32-bit 0xffffffff for a UINT64 return value is to ensure that bad return values don’t get lost when summed up with other values as part of calculating an overall Pipeline stack size. This is only UINT64 at the API. In the DDI this is a UINT; drivers cannot return massive values to the runtime.

GetPipelineStackSize

```
UINT64 GetPipelineStackSize();
```

Retrieve the current pipeline stack size. See Pipeline stack for the meaning of the size.

This call and SetPipelineStackSize() are not re-entrant. This means if calling either or both from separate threads, the app must synchronize on its own.

Parameter	Definition
`Return value: UINT64`	Current pipeline stack size in bytes. If called on non-executable state objects (e.g. collections), the return value is 0.

SetPipelineStackSize

```
void SetPipelineStackSize(UINT64 PipelineStackSizeInBytes);
```

Set the current pipeline stack size. See Pipeline stack for the meaning of the size, defaults, and how to pick a size. This method may optionally be called for a raytracing pipeline state.

This call is and GetPipelineStackSize() or any use of raytracing pipeline state objects, such as via DispatchRays() are not re-entrant. This means if the calling any of these from separate threads,

the app must synchronize on its own. At time of recording on command list, any given DispatchRays() call or ExecuteIndirect() call uses the most resent stack size setting. Similarly GetPipelineStackSize() call always returns the most recent stack size setting.

The runtime drops calls to state objects other than raytracing pipelines (such as collections).

Parameter	Definition
`UINT64 PipelineStackSizeInBytes`	Stack size in bytes to use during pipeline execution for each shader thread (of which there can be many thousands in flight on the GPU). If the value is >= 0xffffffff (max 32-bit UINT) the runtime drops the call (debug layer will print an error) as this is likely the result of summing up invalid stack sizes returned from GetShaderStackSize() called with invalid parameters (which return 0xffffffff). In this case the previously set stack size (or default) remains. This is only UINT64 at the API. In the DDI this is a UINT; as described above, massive stack sizes will not be requested from drivers.

Additional resource states

```
D3D12_RESOURCE_STATE_RAYTRACING_ACCELERATION_STRUCTURE = 0x400000
```

See discussion of this state in Acceleration structure update constraints.

Additional root signature flags

D3D12_ROOT_SIGNATURE_FLAG_LOCAL_ROOT_SIGNATURE

```
typedef enum D3D12_ROOT_SIGNATURE_FLAGS
{
    D3D12_ROOT_SIGNATURE_FLAG_NONE = 0x0,
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT = 0x1,
    D3D12_ROOT_SIGNATURE_FLAG_DENY_VERTEX_SHADER_ROOT_ACCESS = 0x2,
    D3D12_ROOT_SIGNATURE_FLAG_DENY_HULL_SHADER_ROOT_ACCESS = 0x4,
    D3D12_ROOT_SIGNATURE_FLAG_DENY_DOMAIN_SHADER_ROOT_ACCESS = 0x8,
    D3D12_ROOT_SIGNATURE_FLAG_DENY_GEOMETRY_SHADER_ROOT_ACCESS = 0x10,
    D3D12_ROOT_SIGNATURE_FLAG_DENY_PIXEL_SHADER_ROOT_ACCESS = 0x20,
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_STREAM_OUTPUT = 0x40,
    D3D12_ROOT_SIGNATURE_FLAG_LOCAL_ROOT_SIGNATURE = 0x80,
} D3D12_ROOT_SIGNATURE_FLAGS;
```

`D3D12_ROOT_SIGNATURE_FLAG_LOCAL_ROOT_SIGNATURE` indicates the root signature is to be used with raytracing shaders to define resource bindings sourced from shader records in shader tables. This flag cannot be combined with other root signature flags (list shown above) that are all related to the graphics pipeline – they don't make sense together. The absence of the flag means the root signature can be used with graphics or compute, where the compute version is also shared with raytracing's (global) root signature.

Local root signatures don't have restrictions on the number of root parameters that root signatures do.

This distinction between the two classes of root signatures is useful for drivers since their implementation of each layout could be different – one sourcing root arguments from CommandLists while the other sources them from shader tables.

Note on shader visibility

Root signatures used with raytracing share command list state with compute, as described in Local root signatures vs root signatures. As such, the only root parameter shader visibility that applies is `D3D12_SHADER_VISIBILITY_ALL`, meaning the root arguments set as part of compute command list state are also visible to raytracing.

Local root signatures can also only use `D3D12_SHADER_VISIBILITY_ALL`.

In other words, for both root signatures and local root signatures, there's nothing interesting to narrow down with shader visibility flags – local root arguments are simply always visible to all raytracing shaders (and compute for root signatures).

Additional SRV type

Acceleration structures are declared in HLSL via the RaytracingAccelerationStructure resource type, which can then be passed into TraceRay() and RayQuery::TraceRayInline(). From the API, these are bound either:

- via a descriptor heap based SRV with dimension `D3D12_SRV_DIMENSION_RAYTRACING_ACCELERATION_STRUCTURE` (whose description is simply a GPUVA, see below)
- as a root descriptor SRV, in which case no special indication is needed to distinguish it from other root descriptor SRVs, since all are described as simply a GPUVA

When creating descriptor heap based acceleration structure SRVs, the resource parameter must be NULL, as the memory location comes as a GPUVA from the view description

(D3D12_RAYTRACING_ACCELERATION_STRUCTURE_SRV) shown below. E.g. CreateShaderResourceView(NULL,pViewDesc).

```
typedef struct D3D12_RAYTRACING_ACCELERATION_STRUCTURE_SRV
{
    D3D12_GPU_VIRTUAL_ADDRESS Location;
} D3D12_RAYTRACING_ACCELERATION_STRUCTURE_SRV;

typedef enum D3D12_SRV_DIMENSION {
    D3D12_SRV_DIMENSION_UNKNOWN = 0,
    D3D12_SRV_DIMENSION_BUFFER = 1,
    D3D12_SRV_DIMENSION_TEXTURE1D = 2,
    D3D12_SRV_DIMENSION_TEXTURE1DARRAY = 3,
    D3D12_SRV_DIMENSION_TEXTURE2D = 4,
    D3D12_SRV_DIMENSION_TEXTURE2DARRAY = 5,
    D3D12_SRV_DIMENSION_TEXTURE2DMS = 6,
    D3D12_SRV_DIMENSION_TEXTURE2DMSARRAY = 7,
    D3D12_SRV_DIMENSION_TEXTURE3D = 8,
    D3D12_SRV_DIMENSION_TEXTURECUBE = 9,
    D3D12_SRV_DIMENSION_TEXTURECUBEARRAY = 10,
    D3D12_SRV_DIMENSION_RAYTRACING_ACCELERATION_STRUCTURE = 11,
} D3D12_SRV_DIMENSION;

typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC
{
    DXGI_FORMAT Format;
    D3D12_SRV_DIMENSION ViewDimension;
    UINT Shader4ComponentMapping;
    union
    {
        D3D12_BUFFER_SRV Buffer;
        D3D12_TEX1D_SRV Texture1D;
        D3D12_TEX1D_ARRAY_SRV Texture1DArray;
        D3D12_TEX2D_SRV Texture2D;
        D3D12_TEX2D_ARRAY_SRV Texture2DArray;
        D3D12_TEX2DMS_SRV Texture2DMS;
        D3D12_TEX2DMS_ARRAY_SRV Texture2DMSArray;
        D3D12_TEX3D_SRV Texture3D;
        D3D12_TEXCUBE_SRV TextureCube;
        D3D12_TEXCUBE_ARRAY_SRV TextureCubeArray;
        D3D12_RAYTRACING_ACCELERATION_STRUCTURE_SRV RaytracingAccelerationStructure;
    };
} D3D12_SHADER_RESOURCE_VIEW_DESC;
```

Constants

```
#define D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BYTE_ALIGNMENT 256

#define D3D12_RAYTRACING_INSTANCE_DESC_BYTE_ALIGNMENT 16
```

```
#define D3D12_RAYTRACING_TRANSFORM3X4_BYTE_ALIGNMENT 16

#define D3D12_RAYTRACING_MAX_ATTRIBUTE_SIZE_IN_BYTES 32

#define D3D12_RAYTRACING_SHADER_RECORD_BYTE_ALIGNMENT 32

#define D3D12_RAYTRACING_SHADER_TABLE_BYTE_ALIGNMENT 64

#define D3D12_RAYTRACING_AABB_BYTE_ALIGNMENT 8

#define D3D12_RAYTRACING_MAX_DECLARABLE_TRACE_RECURSION_DEPTH 31

#define D3D12_SHADER_IDENTIFIER_SIZE_IN_BYTES 32
```

HLSL

Types, enums, subobjects and concepts

Ray flags

Ray flags are passed to TraceRay() or RayQuery::TraceRayInline() to override transparency, culling, and early-out behavior. For an example, see here.

Ray flags are also a template paramter to RayQuery objects to enable static specialization - runtime behavior of RayQuery::TraceRayInline() takes on the OR of dynamic and static/template ray falgs.

Shaders that interact with rays can query the current flags via RayFlags() or RayQuery::RayFlags() intrinsics.

```
enum RAY_FLAG : uint
{
    RAY_FLAG_NONE = 0x00,
    RAY_FLAG_FORCE_OPAQUE = 0x01,
    RAY_FLAG_FORCE_NON_OPAQUE = 0x02,
    RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH = 0x04,
    RAY_FLAG_SKIP_CLOSEST_HIT_SHADER = 0x08,
    RAY_FLAG_CULL_BACK_FACING_TRIANGLES = 0x10,
    RAY_FLAG_CULL_FRONT_FACING_TRIANGLES = 0x20,
    RAY_FLAG_CULL_OPAQUE = 0x40,
    RAY_FLAG_CULL_NON_OPAQUE = 0x80,
    RAY_FLAG_SKIP_TRIANGLES = 0x100,
```

```
    RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES = 0x200,
};
```

(d3d12.h has equivalent `D3D12_RAY_FLAG_*` defined for convenience)

Value	Definition
`RAY_FLAG_NONE`	No options selected.
`RAY_FLAG_FORCE_OPAQUE`	All ray-primitive intersections encountered treated as opaque. So no any hit shaders will be regardless of whether or not the hit geometry s D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQU of the instance flags on the instance that was hi case of RayQuery::TraceRayInline(), the impact o shader invocations but instead on TraceRayInlin Mutually exclusive to `RAY_FLAG_FORCE` `RAY_FLAG_CULL_OPAQUE` and `RAY_FLAG_CULL_NON_`
`RAY_FLAG_FORCE_NON_OPAQUE`	All ray-primitive intersections encountered treated as non-opaque. So any hit shaders, if pr executed regardless of whether or not the hit ge D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQU of the instance flags on the instance that was hi case of RayQuery::TraceRayInline(), the impact o shader invocations but instead on TraceRayInlin Mutually exclusive to `RAY_FLAG_FORCE` `RAY_FLAG_CULL_OPAQUE` and `RAY_FLAG_CULL_NON_`
	The first ray-primitive intersection encounte

Value	Definition
	automatically causes AcceptHitAndEndSearch() immediately after the any hit shader (including
RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH	hit shader). The only exception is wher any hit shader calls IgnoreHit(), in which case th unaffected (such that the next hit becomes anot be the first hit. For this exception to apply, the a to actually be executed. So if the any hit shader because the hit is treated as opaque (e.g. due to RAY_FLAG_FORCE_OPAQUE or D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQU D3D12_RAYTRACING_INSTANCE_FLAG_OPAQUE AcceptHitAndEndSearch() is called. If a shader is present at the first hit, it gets invoked RAY_FLAG_SKIP_CLOSEST_HIT_SHADER is also prese that was found is considered “closest”, even tho potential hits that might be closer on the ray ma visited. A typical use for this flag is for only a single hit needs to be found. In RayQuery::TraceRayInline(), the impact of this fla above, except applied to TraceRayInline control the first committed hit, either from fixed functio from the shader, traversal completes.
RAY_FLAG_SKIP_CLOSEST_HIT_SHADER	Even if at least one hit has been committed group for the closest hit contains a closest hit sh execution of that shader. One example could help is illustrated here. This flag (makes no sense for) RayQuery::TraceRayInline() template parameter.
RAY_FLAG_CULL_BACK_FACING_TRIANGLES	Enables culling of back facing triangles. See D3D12_RAYTRACING_INSTANCE_FLAGS for sele triangles are back facing, per-instance. that specify D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGL this flag has no effect. On geometry ty D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANG no effect. This flag is mutually exclusive RAY_FLAG_CULL_FRONT_FACING_TRIANGLES .
	Enables culling of front facing triangles. See

Value	Definition
	D3D12_RAYTRACING_INSTANCE_FLAGS for sele triangles are back facing, per-instance.
RAY_FLAG_CULL_FRONT_FACING_TRIANGLES	that specify D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGI this flag has no effect. On geometry ty D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANG no effect. This flag is mutually exclusiv RAY_FLAG_CULL_BACK_FACING_TRIANGLES .
RAY_FLAG_CULL_OPAQUE	Culls all primitives that are considered opac their geometry and instance flags. This exclusive to RAY_FLAG_FORCE_OPAQUE , RAY_FLAG_FORCE_NON_OPAQUE , and RAY_FLAG_CULL
RAY_FLAG_CULL_NON_OPAQUE	Culls all primitives that are considered non- on their geometry and instance flags. T mutually exclusive to RAY_FLAG_FORCE_OPAQUE , RAY_FLAG_FORCE_NON_OPAQUE , and RAY_FLAG_CULL
RAY_FLAG_SKIP_TRIANGLES	Culls all triangles. This flag is mutu RAY_FLAG_CULL_FRONT_FACING_TRIANGLES , RAY_FLAG_CULL_BACK_FACING_TRIANGLES and RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES . <p specify D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGI it has no effect, as that flag is meant for disablir culling only. In other words RAY_FLAG_SKIP_TRIA overruled. This flag is only supported a implementations.
RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES	Culls all procedural primitives. This exclusive to RAY_FLAG_SKIP_TRIANGLES . <p supported as of Tier 1.1 implementations.

Ray description structure

The RayDesc structure is passed to TraceRay() or RayQuery::TraceRayInline() to define the origin, direction, and extents of the ray.

```
struct RayDesc
{
    float3 Origin;
    float TMin;
    float3 Direction;
```

```
    float TMax;
};
```

Raytracing pipeline flags

Flags used in Raytracing pipeline config1 subobject.

```
enum RAYTRACING_PIPELINE_FLAG : uint
{
    RAYTRACING_PIPELINE_FLAG_NONE                           = 0x0,
    RAYTRACING_PIPELINE_FLAG_SKIP_TRIANGLES                 = 0x100,
    RAYTRACING_PIPELINE_FLAG_SKIP_PROCEDURAL_PRIMITIVES     = 0x200,
};
```

For definitions of the flags, see the D3D12 API equivalent, D3D12_RAYTRACING_PIPELINE_FLAGS.

RaytracingAccelerationStructure

RaytracingAccelerationStructure is a resource type that can be declared in HLSL. It is bound as a raw buffer SRV in a descriptor table or root descriptor SRV. The declaration in HLSL is as follows:

```
RaytracingAccelerationStructure MyScene[] : register(t3,space1);
```

This example shows an unbounded size array of acceleration structures, which implies coming from a descriptor heap since root descriptors can't be indexed.

The RaytracingAccelerationStructure (for instance MyScene[i]) is passed to TraceRay() or RayQuery::TraceRayInline() to indicate the top-level acceleration resource built using BuildRaytracingAccelerationStructure(). It is an opaque resource with no methods available to shaders.

Subobject definitions

In addition to creating subobjects at runtime with the API, they can also be defined in HLSL and made available through compiled DXIL libraries.

Hit group

The hit group is a group of zero or one intersection, anyhit, and closesthit shaders referenced by name (string), rather than a single shader entry function. Use an empty string to omit a shader type.

Example:

```
HitGroup my_group_name("intersection_main", "anyhit_main",
"closesthit_main");
```

Root signature

A named root signature that can used globally in a raytracing pipeline or associated with shaders by name. The root signature is global for all shaders in a DispatchRays call.

```
RootSignature my_rs_name("root signature definition");
```

Local root signature

A named local root signature that can be associated with shaders. A local root signature defines the structure of additional root arguments read from the shader record in the shader table.

```
LocalRootSignature my_local_rs_name("local root signature definition");
```

Subobject to entrypoint association

An association between one subobject, such as a local root signature and a list of shader entry points. The subobject is referenced by name in a string, and the list of entry points is supplied as a semicolon-separated list of function names in a string.

```
SubobjectToEntrypointAssociation
my_association_name("subobject_name","function1;function2;function3");
```

Raytracing shader config

Defines the maximum sizes in bytes for the ray payload and intersection attributes. See the API equivalent: D3D12_RAYTRACING_SHADER_CONFIG.

```
RaytracingShaderConfig shader_config_name(maxPayloadSizeInBytes,maxAttributeSizeInBytes);
```

Raytracing pipeline config

Defines the maximum TraceRay() recursion depth. See the API equivalent: D3D12_RAYTRACING_PIPELINE_CONFIG.

```
RaytracingPipelineConfig config_name(maxTraceRecursionDepth);
```

Raytracing pipeline config1

Defines the maximum TraceRay() recursion depth as well as raytracing pipeline flags. See the API equivalent: D3D12_RAYTRACING_PIPELINE_CONFIG1.

```
RaytracingPipelineConfig1 config_name(maxTraceRecursionDepth,
                                      RAYTRACING_PIPELINE_CONFIG_FLAG_*);
```

The available flags (`RAYTRACING_PIPELINE_CONFIG_FLAG_*`) are defined at: Raytracing pipeline flags.

This subobject is only available on devices with Tier 1.1 raytracing support.

Intersection attributes structure

Intersection attributes come from one of two sources:

(1) Triangle geometry via fixed-function triangle intersection. In this case the structure used is the following:

```
struct BuiltInTriangleIntersectionAttributes
{
    float2 barycentrics;
};
```

any hit and closest hit shaders invoked using fixed-function triangle intersection must use this structure for hit attributes.

Given attributes `a0` , `a1` and `a2` for the 3 vertices of a triangle, `barycentrics.x` is the weight for `a1` and `barycentrics.y` is the weight for `a2` . For example, the app can interpolate by doing: `a = a0 + barycentrics.x \* (a1-a0) + barycentrics.y\* (a2 -- a0)` .

(2) Intersection with axis-aligned bounding boxes for procedural primitives in the raytracing acceleration structure triggers an intersection shader. That shader provides a user-defined intersection attribute structure to the ReportHit() call. The any hit and closest hit shaders bound in the same hit group with this intersection shader must use the same structure for hit attributes, even if the attributes are not referenced. The maximum attribute structure size is 32 bytes (D3D12_RAYTRACING_MAX_ATTRIBUTE_SIZE_IN_BYTES).

Ray payload structure

This is a user-defined structure that is provided as an inout argument in the TraceRay() call, and as an inout parameter in the shader types that may access the ray payload (any hit, closest hit, and miss shaders). Any shaders that access the ray payload must use the same structure as the one provided at the originating TraceRay() call. Even if one of these shaders doesn't reference the ray payload at all, it still must specify the matching payload as the originating TraceRay() call.

See payload access qualifiers for a discussion of annotations on payload members introduced in Shader Models 6.6 and 6.7.

Call parameter structure

This is a user-defined structure provided as an inout argument in the CallShader() call, and as an inout parameter for the Callable shader. The structure type used in the callable shader must match the structure provided to the corresponding CallShader() call.

Shaders

These shaders are functions compiled into a library (target lib_6_3), and identified by an attribute [shader("*shadertype*")] on the shader function.

See Intrinsics and System value intrinsics to see what is allowed for each shader type.

Certain features supported in graphics or compute shader types are not supported in raytracing shader types. See Shader limitations resulting from independence.

Ray generation shader

shader type: `raygeneration`

Overview is here.

Ray generation shaders call TraceRay() to generate rays (aside: RayQuery::TraceRayInline() works too, as it can be called from anywhere). The initial user-defined ray payload for each ray is provided to the TraceRay() call site. CallShader() may also be used in ray generation shaders to invoke callable shaders.

Rough Example:

```
struct SceneConstantStructure { ... };

ConstantBuffer<SceneConstantStructure> SceneConstants;
RaytracingAccelerationStructure MyAccelerationStructure : register(t3);
struct MyPayload { ... };

[shader("raygeneration")]
void raygen_main()
{
    RayDesc myRay = {
        SceneConstants.CameraOrigin,
        SceneConstants.TMin,
        computeRayDirection(SceneConstants.LensParams, DispatchRaysIndex(),
        DispatchRaysDimensions()),
        SceneConstants.TMax
        };

    MyPayload payload = { ... }; // init payload

    TraceRay(
        MyAccelerationStructure,
        SceneConstants.RayFlags,
        SceneConstants.InstanceInclusionMask,
        SceneConstants.RayContributionToHitGroupIndex,
        SceneConstants.MultiplierForGeometryContributionToHitGroupIndex,
        SceneConstants.MissShaderIndex,
        myRay,
        payload);

    WriteFinalPixel(DispatchRaysIndex(), payload);
}
```

Intersection shader

shader type: `intersection`

Overview is here.

Used to implement custom intersection primitives, the intersection shader is invoked for rays intersecting an associated bounding volume (bounding box). The intersection shader does not have access to the ray payload, but defines the intersection attributes for each hit through the ReportHit() call. The handling of ReportHit() may stop the intersection shader early, if the Ray

Flag `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` is set, or AcceptHitAndEndSearch() is called from an any hit shader. Otherwise, it returns true if the hit was accepted or false if rejected (see ReportHit() for details). This means that an any hit shader, if present, must execute before control conditionally returns to the intersection shader.

Rough Example:

```
struct CustomPrimitiveDef { ... };
struct MyAttributes { ... };
struct CustomIntersectionIterator {...};

void InitCustomIntersectionIterator(CustomIntersectionIterator it) {...}

bool IntersectCustomPrimitiveFrontToBack(
    CustomPrimitiveDef prim,
    inout CustomIntersectionIterator it,
    float3 origin, float3 dir,
    float rayTMin, inout float curT,
    out MyAttributes attr);

[shader("intersection")]
void intersection_main()
{
    float THit = RayTCurrent();
    MyAttributes attr;
    CustomIntersectionIterator it;

    InitCustomIntersectionIterator(it);

    while(IntersectCustomPrimitiveFrontToBack(
          CustomPrimitiveDefinitions[LocalConstants.PrimitiveIndex],
          it, ObjectRayOrigin(), ObjectRayDirection(),
          RayTMin(), THit, attr))
    {
        // Exit on the first hit. Note that if the ray has
        // RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH or an
        // anyhit shader is used and calls AcceptHitAndEndSearch(),
        // that would also fully exit this intersection shader (making
        // the "break" below moot in that case).
        if (ReportHit(THit, /*hitKind*/ 0, attr) &&
             (RayFlags() & RAY_FLAG_FORCE_OPAQUE))
            break;
    }
}
```

Any hit shader

shader type: `anyhit`

Overview is here.

The any hit shader is invoked when intersections are not opaque. The any hit shaders must declare a payload parameter, followed by an attributes parameter. Each must be a user defined structure type matching types used for TraceRay and ReportHit respectively (or the BuiltInIntersectionAttributes structure when fixed function triangle intersection is used).

The any hit shader may do the following kinds of things:

- Read and modify the ray payload: (inout payload_t rayPayload)
- Read the intersection attributes: (in attr_t attributes)
- Call AcceptHitAndEndSearch(), which accepts the current hit, ends the any hit shader, ends the intersection shader (if any), and executes the closest hit shader on the closest hit so far (if active).
- Call IgnoreHit(), which ends the any hit shader and tells the system to continue searching for hits, including returning control to an intersection shader (if currently executing) returning false from the ReportHit() call site.
- Return without calling either of these intrinsics, which accepts the current hit and tells the system to continue searching for hits, including returns control to the intersection shader (if any), returning true at the ReportHit() call site to indicate that the hit was accepted.

Even if an any hit shader invocation is ended by IgnoreHit() or AcceptHitAndEndSearch(), any modifications made to the ray payload so far must still be retained.

Rough Example:

```
[shader("anyhit")]
void anyhit_main( inout MyPayload payload, in MyAttributes attr )
{
  float3 hitLocation = ObjectRayOrigin() + ObjectRayDirection() *
      RayTCurrent();

  float alpha = computeAlpha(hitLocation, attr, ...);

  // Processing shadow and only care if a hit is registered?
  if (TerminateShadowRay(alpha))
      AcceptHitAndEndSearch(); // aborts function

  // Save alpha contribution and ignoring hit?
  if (SaveAndIgnore(payload, RayTCurrent(), alpha, attr, ...))
      IgnoreHit(); // aborts function

  // do something else
  // return to accept and update closest hit
}
```

Closest hit shader

shader type: `closesthit`

Overview is here.

When the closest hit has been determined or ray intersection search ended, the closest hit shader is invoked (if enabled). This is where surface shading and additional ray generation will typically occur. Closest hit shaders must declare a payload parameter, followed by an attributes parameter. Each must be a user defined structure type matching types used for TraceRay and ReportHit respectively (or the BuiltInIntersectionAttributes structure when fixed function triangle intersection is used).

Closest hit shaders may:

- Read and modify the ray payload: (inout payload_t rayPayload)
- Read the closest Intersection Attributes: (in attr_t attributes)
- Use CallShader() and TraceRay() to schedule more work and read back results.

Rough Example:

```
[shader("closesthit")]
void closesthit_main(inout MyPayload payload, in MyAttributes attr)
{
    CallShader( ... ); // maybe

    // update payload for surface
    // trace reflection
    float3 worldRayOrigin = WorldRayOrigin() + WorldRayDirection() *
        RayTCurrent();

    float3 worldNormal = mul(attr.normal, (float3x3)ObjectToWorld3x4());
    RayDesc reflectedRay = { worldRayOrigin, SceneConstants.Epsilon,
                             ReflectRay(WorldRayDirection(), worldNormal),
                             SceneConstants.TMax };

    TraceRay(MyAccelerationStructure,
            SceneConstants.RayFlags,
            SceneConstants.InstanceInclusionMask,
            SceneConstants.RayContributionToHitGroupIndex,
            SceneConstants.MultiplierForGeometryContributionToHitGroupIndex,
            SceneConstants.MissShaderIndex,
            reflectedRay,
            payload);

    // Combine final contributions into ray payload
    // this ray query is now complete.

    // Alternately, could look at data in payload result based on that make
```

```
    // other TraceRay calls. No constraints on the code structure.
}
```

Alternatively to the above example of calculating a current ray hit position via "rayOrigin + rayDirection * RayTCurrent()", it is possible to calculate the hit position via surface parametrization using barycentrics and vertex positions. The former is faster to calculate, but more prone to floating point error as any error will offset the position along the ray direction often away from the surface. This is, in particular, true for large RayTCurrent() values. The other method, surface parametrization, is more precise with any computational error shifting the computed position mostly along the surface, but also more expensive as it requires loading the vertex positions and doing more calculations. The extra overhead may be worthwhile if precision is critical.

Miss shader

shader type: `miss`

Overview is here. The miss shader must include a user defined structure typed payload parameter matching the one supplied to TraceRay().

If no intersections are found or accepted, the miss shader is invoked. This is useful for background or sky shading. The miss shader may use CallShader() and TraceRay() to schedule more work.

Rough Example:

```
[shader("miss")]
void miss_main(inout MyPayload payload)
{
  // Use ray system values to compute contributions of background,
  // sky,etc...

  // Combine contributions into ray payload
  CallShader( ... ); // maybe

  TraceRay( ... ); // maybe

  // this ray query is now complete
}
```

Callable shader

shader type: `callable`

Overview is here.

The callable shader is invoked from another shader by using the CallShader() intrinsic. There is a parameter structure supplied at the CallShader() call site that must match the parameter structure used in the callable shader pointed to by the requested index into the callable shader table supplied through the DispatchRays() API. The callable shader must declare this parameter as inout. Additionally, the callable shader may read launch index and dimension inputs, see System value intrinsics.

Rough Example:

```
[shader("callable")]
void callable_main(inout MyParams params)
{
    // Perform some common operations and update params
    CallShader( ... ); // maybe
}
```

Intrinsics

intrinsics \ shaders	ray generation	intersection	any hit	closest hit	miss	callable
CallShader()	*			*	*	*
TraceRay()	*			*	*	
ReportHit()		*				
IgnoreHit()			*			
AcceptHitAndEndSearch()			*			

CallShader

This intrinsic function definition is equivalent to the following function template:

```
template<param_t>
void CallShader(uint ShaderIndex, inout param_t Parameter);
```

Parameter	Definition
`uint ShaderIndex`	Provides index into the callable shader table supplied through the DispatchRays() API. See Callable shader table indexing.
`inout param_t Parameter`	The user-defined parameters to pass to the callable shader. This parameter structure must match the parameter structure used in the callable shader pointed to in the shader table.

TraceRay

Send a ray into a search for hits in an acceleration structure, including various types of shader invocations where applicable. See TraceRay() control flow.

This intrinsic function definition is equivalent to the following function template:

```
Template<payload_t>
void TraceRay(RaytracingAccelerationStructure AccelerationStructure,
    uint RayFlags,
    uint InstanceInclusionMask,
    uint RayContributionToHitGroupIndex,
    uint MultiplierForGeometryContributionToHitGroupIndex,
    uint MissShaderIndex,
    RayDesc Ray,
    inout payload_t Payload);
```

Parameter	Definition
`RaytracingAccelerationStructure AccelerationStructure`	Top-level acceleration structure to use. Specifying a NULL acceleration structure forces a miss.
`uint RayFlags`	Valid combination of Ray flags. Only defined ray flags are propagated by the system, e.g. visible to the RayFlags() shader intrinsic.

Parameter	Definition
`uint InstanceInclusionMask`	Bottom 8 bits of InstanceInclusionMask are used to include/reject geometry instances based on the InstanceMask in each instance: `if(! ((InstanceInclusionMask & InstanceMask) & 0xff)) { ignore intersection } </p>`
`uint RayContributionToHitGroupIndex`	Offset to add into Addressing calculations within shader tables for hit group indexing. Only the bottom 4 bits of this value are used.
`uint MultiplierForGeometryContributionToShaderIndex`	Stride to multiply by GeometryContributionToHitGroupIndex (which is just the 0 based index the geometry was supplied by the app into its bottom-level acceleration structure). See Addressing calculations within shader tables for hit group indexing. Only the bottom 4 bits of this multiplier value are used.
`uint MissShaderIndex`	Miss shader index in Addressing calculations within shader tables. Only the bottom 16 bits of this value are used.
`RayDesc Ray`	Ray to be traced. See Ray-extents for bounds on valid ray parameters.
`inout payload_t Payload`	User defined ray payload accessed both for both input and output by shaders invoked during raytracing. After TraceRay completes, the caller can access the payload as well.

ReportHit

This intrinsic definition is equivalent to the following function template:

```
template<attr_t>
bool ReportHit(float THit, uint HitKind, attr_t Attributes);
```

Parameter	Definition
`float THit`	The parametric distance of the intersection.
`uint HitKind`	A value used to identify the type of hit. This is a user-specified value in the range of 0-127. The value can be read by any hit or closest hit shaders with the HitKind() intrinsic.
`attr_t Attributes`	Intersection attributes. The type attr_t is the user-defined intersection attribute structure. See Intersection attributes structure.

ReportHit returns true if the hit was accepted. A hit is rejected if THit is outside the current ray interval, or the any hit shader calls IgnoreHit(). The current ray interval is defined by RayTMin() and RayTCurrent().

IgnoreHit

```
void IgnoreHit();
```

Used in an any hit shader to reject the hit and end the shader. The hit search continues on without committing the distance (hitT) and attributes for the current hit. The ReportHit() call in the intersection shader (if any) will return false. Any modifications made to the ray payload up to this point in the any hit shader are preserved.

AcceptHitAndEndSearch

```
void AcceptHitAndEndSearch();
```

Used in an any hit shader to commit the current hit (hitT and attributes) and then stop searching for more hits for the ray. If there is an intersection shader running, that stops. Execution passes to the closest hit shader (if enabled) with the closest hit recorded so far.

System value intrinsics

System values are retrieved by using special intrinsic functions, rather than including parameters with special semantics in your shader function signature.

The following table shows where system value intrinsics.

values \ shaders	ray generation	intersection	any hit	closest hit	miss	callabl
Ray dispatch system values:						
uint3 DispatchRaysIndex()	*	*	*	*	*	*
uint3 DispatchRaysDimensions()	*	*	*	*	*	*
Ray system values:						
float3 WorldRayOrigin()		*	*	*	*	
float3 WorldRayDirection()		*	*	*	*	
float RayTMin()		*	*	*	*	
float RayTCurrent()		*	*	*	*	
uint RayFlags()		*	*	*	*	
Primitive/object space system values:						
uint InstanceIndex()		*	*	*		
uint InstanceID()		*	*	*		
uint GeometryIndex() (requires Tier 1.1 implementation)		*	*	*		
uint PrimitiveIndex()		*	*	*		
float3 ObjectRayOrigin()		*	*	*		
float3 ObjectRayDirection()		*	*	*		

values / shaders	ray generation	* intersection	any hit	closest hit	miss	callabl
float3x4 ObjectToWorld3x4()						
float4x3 ObjectToWorld4x3()		*	*	*		
float3x4 WorldToObject3x4()		*	*	*		
float4x3 WorldToObject4x3()		*	*	*		
Hit specific system values:						
uint HitKind()			*	*		

Ray dispatch system values

Launch system values are inputs available to every raytracing shader type. They return the values at the ray generation shader instance that led to the current shader instance.

DispatchRaysIndex

The current x and y location within the Width and Height made available through the DispatchRaysDimensions() system value intrinsic.

```
uint3 DispatchRaysIndex();
```

DispatchRaysDimensions

The Width, Height and Depth values from the `D3D12_DISPATCH_RAYS_DESC` structure provided to the originating DispatchRays() call.

```
uint3 DispatchRaysDimensions();
```

Ray system values

These system values are available to all shaders in the hit group and miss shaders.

WorldRayOrigin

The world-space origin for the current ray.

```
float3 WorldRayOrigin();
```

WorldRayDirection

The world-space direction for the current ray.

```
float3 WorldRayDirection();
```

RayTMin

This is a float representing the parametric starting point for the ray.

```
float RayTMin();
```

RayTMin defines the starting point of the ray according to the following formula: Origin + (Direction * RayTMin). Origin and Direction may be in either world or object space, which results in either a world or an object space starting point.

RayTMin is defined when calling TraceRay(), and is constant for the duration of that call.

RayTCurrent

This is a float representing the current parametric ending point for the ray.

```
float RayTCurrent();
```

RayTCurrent defines the current ending point of the ray according to the following formula: Origin + (Direction * RayTCurrent). Origin and Direction may be in either world or object space, which results in either a world or an object space ending point.

RayTCurrent is initialized by the TraceRay() call from RayDesc::TMax, and updated during the trace query as intersections are reported (in the any hit), and accepted.

In the intersection shader, it represents the distance to the closest intersection found so far. It will be updated after ReportHit() to the THit value provided if the hit was accepted.

In the any hit shader, it represents the distance to the current intersection being reported.

In the closest hit shader, it represents the distance to the closest intersection accepted.

In the miss shader, it is equal to TMax passed to the TraceRay() call.

RayFlags

This is a uint containing the current ray flags (only). It doesn't reveal any flags that may have been added externally via D3D12_RAYTRACING_PIPELINE_CONFIG1.

```
uint RayFlags();
```

This can be useful if, for instance, in an intersection shader an app wants to look at the current ray's culling flags and apply corresponding culling in its custom intersection code.

Primitive/object space system values

These system values are available once a primitive has been selected for intersection. They enable identifying what is being intersected by the ray, the object space ray origin and direction, and the transformation matrices between object and world space.

InstanceIndex

The autogenerated index of the current instance in the top-level structure.

```
uint InstanceIndex();
```

InstanceID

The user-provided InstanceID on the bottom-level acceleration structure instance within the top-level structure.

```
uint InstanceID();
```

GeometryIndex

The autogenerated index of the current geometry in the bottom-level acceleration structure.

This method is only available on Tier 1.1 implementations.

The MultiplierForGeometryContributionToHitGroupIndex parameter to TraceRay() can be set to 0 to stop the geometry index from contributing to shader table indexing if the shader just wants to rely on GeometryIndex() to distinguish geometries.

```
uint GeometryIndex();
```

PrimitiveIndex

The autogenerated index of the primitive within the geometry inside the bottom-level acceleration structure instance.

```
uint PrimitiveIndex();
```

For `D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES`, this is the triangle index within the geometry object.

For `D3D12_RAYTRACING_GEOMETRY_TYPE_PROCEDURAL_PRIMITIVE_AABBS`, this is the index into the AABB array defining the geometry object.

ObjectRayOrigin

Object-space origin for the current ray. Object-space refers to the space of the current bottom-level acceleration structure.

```
float3 ObjectRayOrigin();
```

ObjectRayDirection

Object-space direction for the current ray. Object-space refers to the space of the current bottom-level acceleration structure.

```
float3 ObjectRayDirection();
```

ObjectToWorld3x4

Matrix for transforming from object-space to world-space. Object-space refers to the space of the current bottom-level acceleration structure.

```
float3x4 ObjectToWorld3x4();
```

The only difference between this and `ObjectToWorld4x3()` is the matrix is transposed – use whichever is convenient.

ObjectToWorld4x3

Matrix for transforming from object-space to world-space. Object-space refers to the space of the current bottom-level acceleration structure.

```
float4x3 ObjectToWorld4x3();
```

The only difference between this and `ObjectToWorld3x4()` is the matrix is transposed – use whichever is convenient.

WorldToObject3x4

Matrix for transforming from world-space to object-space. Object-space refers to the space of the current bottom-level acceleration structure.

```
float3x4 WorldToObject3x4();
```

The only difference between this and WorldToObject4x3() is the matrix is transposed – use whichever is convenient.

WorldToObject4x3

Matrix for transforming from world-space to object-space. Object-space refers to the space of the current bottom-level acceleration structure.

```
float3x4 WorldToObject4x3();
```

The only difference between this and WorldToObject3x4() is the matrix is transposed – use whichever is convenient.

Hit specific system values

HitKind

Returns the value passed as HitKind in ReportHit(). If intersection was reported by fixed-function triangle intersection, HitKind will be one of `HIT_KIND_TRIANGLE_FRONT_FACE` (254) or `HIT_KIND_TRIANGLE_BACK_FACE` (255).

(d3d12.h has equivalent `D3D12_HIT_KIND_*` defined for convenience)

```
uint HitKind();
```

RayQuery

```
RayQuery<RAY_FLAGS>
```

`RayQuery` represents the state of an inline raytracing call into an acceleration structure via member function RayQuery::TraceRayInline(). It is "inline" in the sense that this form of raytracing doesn't automatically invoke other shader invocations during traversal as TraceRay() does. The shader observes `RayQuery` to be in one of a set of states, each of which exposes relevant methods to retrieve information about the current state or continue the query into the acceleration structure.

`RayQuery` takes a template parameter on instantiation: a set of RAY_FLAGS that are inline/literal at compile time. These flags enable implementations to produce more efficient inline code generation by drivers customized to the selected flags. For example, declaring `RayFlags<RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES> myRayQuery` means ray traversals done via `myRayQuery` will not enumerate procedural primitives. Applications can also custimize their code to rely on the impact of the selected flags - in this example not having to write code to handle the procedural primitives showing up. RayQuery::TraceRayInline() also has a RAY_FLAGS field which allows dynamic ray flags configuration - the template flags are OR'd with the dynamic flags during a raytrace.

Pseudocode examples are here.

The size of `RayQuery` is implementation specific and opaque. Shaders can have any number of live `RayQuery` objects during shader execution, but at the cost of an unknown (and relatively high) amount of hardware register storage, which can limit how many shaders can run in parallel. This unknown size is in contrast to traditional shader variables (e.g. `float4 foo`), whose register cost is obvious by the data type of the variable (e.g. 16 bytes for a `float4`).

If a variable of type `RayQuery` is assigned to another (which must have been declared with a matching template specification), a reference to the original is passed (rather than a clone), so they are both now operating on the same shared state machine. If a variable of type `RayQuery` is passed into a function as a parameter, it is passed by reference. If a variable of type `RayQuery` is overwritten by another (such as by assignment), the overwritten object disappears.

> A RayQuery::Clone() intrinsic was considered to enable forking an in progress ray traversal. But this appeared to be of no value, lacking a known interesting scenario.
>
> A proposed feature that was cut is the ability for full TraceRay() to return a RayQuery object. This would have been a middle ground between inline raytracing and the dynamic-shader-based form - e.g. apps could use dynamic any-hit shaders but then choose to do final hit/miss processing in the calling shader, without the dynamic shaders having to bother stuffing ray query metadata like hit distance in the ray payload - redundant information given that the system knows it. It turned out that for some implementations this would actually be slower than the application manually stuffing only needed values into the ray payload, and would require extra compilation for paths that need the ray query versus those that do not.
>
> This feature could come back in a more refined form in the future. This could be allowing the user to declare entries in the ray payload as system generated values for all query data (e.g. SV_CurrentT). Some implementations could choose to compile shaders (which see the payload declaration) to automatically store these values in the ray payload as declared, whereas other implementations would be free to pull these system values as needed from elsewhere if they are readily available (avoiding payload bloat).

RayQuery intrinsics

`RayQuery` supports the methods (*) in the following tables depending on its current state. The intrinsics are listed across several tables for readability (with entries repeated as appicable). Calling unsupported methods is invalid and produces undefined results.

The first table lists intrinsics available after the two steps in `RayQuery` initialization: First, declaring a RayQuery object (with RayFlags template parameter), followed by calling RayQuery::TraceRayInline() to initialize trace parameters but not yet begin the traversal. RayQuery::TraceRayInline() can be called over again any time to initialize a new trace, discarding the current one regardlessof what state it is in.

Intrinsic \ State	New `RayQuery` object	`TraceRayInline()` was called
TraceRayInline()	*	*
`bool` Proceed()		*

Intrinsic \ State	New `RayQuery` object	`TraceRayInline()` was called
Abort()		*
`COMMITTED_STATUS` CommittedStatus()		*

The following table lists intrinsics available when RayQuery::Proceed() returned `TRUE`, meaning a type of hit candidate that requires shader evaluation has been found. Methods named Committed*() in this table may not actually be available depending on the current CommittedStatus() (i.e what type of hit has been commited yet if any?) - this is further clarified in another table further below.

Intrinsic \ **CandidateType()**	`HIT_CANDIDATE_NON_OPAQUE_TRIANGLE`
TraceRayInline()	*
`bool` Proceed()	*
Abort()	*
`CANDIDATE_TYPE` CandidateType()	*
bool CandidateProceduralPrimitiveNonOpaque()	
CommitNonOpaqueTriangleHit()	*
CommitProceduralPrimitiveHit(float t)	
`COMMITTED_STATUS` CommittedStatus())	*
Ray system values:	
uint RayFlags()	*
float3 WorldRayOrigin()	*
float3 WorldRayDirection()	*
float RayTMin()	*
float CandidateTriangleRayT()	*
float CommittedRayT()	*
Primitive/object space system values:	
uint CandidateInstanceIndex()	*
uint CandidateInstanceID()	*

Intrinsic \ CandidateType()	HIT_CANDIDATE_NON_OPAQUE_TRIANGLE	
uint CandidateInstanceContributionToHitGroupIndex()	*	
uint CandidateGeometryIndex()	*	
uint CandidatePrimitiveIndex()	*	
float3 CandidateObjectRayOrigin()	*	
float3 CandidateObjectRayDirection()	*	
float3x4 CandidateObjectToWorld3x4()	*	
float4x3 CandidateObjectToWorld4x3()	*	
float3x4 CandidateWorldToObject3x4()	*	
float4x3 CandidateWorldToObject4x3()	*	
uint CommittedInstanceIndex()	*	
uint CommittedInstanceID()	*	
uint CommittedInstanceContributionToHitGroupIndex()	*	
uint CommittedGeometryIndex()	*	
uint CommittedPrimitiveIndex()	*	
float3 CommittedObjectRayOrigin()	*	
float3 CommittedObjectRayDirection()	*	
float3x4 CommittedObjectToWorld3x4()	*	
float4x3 CommittedObjectToWorld4x3()	*	

float3x4 CommittedWorldToObject3x4()	*	
float4x3 CommittedWorldToObject4x3()	*	
Hit specific system values:		
float2 CandidateTriangleBarycentrics()	*	
bool CandidateTriangleFrontFace()	*	
float2 CommittedTriangleBarycentrics()	*	
bool CommittedTriangleFrontFace()	*	

bool CommittedTriangleFrontFace()

The following table lists intrinsics available depending on the current current COMMITTED_STATUS (i.e. what type of hit has been commited, if any?). This applies regardless of whether RayQuery::Proceed() has returned `TRUE` (shader evaluation needed for traversal), or `FALSE` (traversal complete). If `TRUE`, additional methods than shown below are available based on the table above.

Intrinsic \ CommittedStatus()	COMMITTED_TRIANGLE_HIT	COMMITTED_P
TraceRayInline()	*	
`COMMITTED_STATUS` CommittedStatus()	*	
Ray system values:		
uint RayFlags()	*	
float3 WorldRayOrigin()	*	
float3 WorldRayDirection()	*	
float RayTMin()	*	
float CommittedRayT()	*	
Primitive/object space system values:		
uint CommittedInstanceIndex()	*	
uint CommittedInstanceID()	*	
uint CommittedInstanceContributionToHitGroupIndex()	*	
uint CommittedGeometryIndex()	*	
uint CommittedPrimitiveIndex()	*	
float3 CommittedObjectRayOrigin()	*	
float3 CommittedObjectRayDirection()	*	
float3x4 CommittedObjectToWorld3x4()	*	
float4x3 CommittedObjectToWorld4x3()	*	
float3x4 CommittedWorldToObject3x4()	*	
float4x3 CommittedWorldToObject4x3()	*	

Hit specific system values:

Intrinsic \ CommittedStatus()	COMMITTED_TRIANGLE_HIT	COMMITTED_P
float2 CommittedTriangleBarycentrics()	*	
bool CommittedTriangleFrontFace()	*	

RayQuery enums

COMMITTED_STATUS

```
enum COMMITTED_STATUS : uint
{
    COMMITTED_NOTHING,
    COMMITTED_TRIANGLE_HIT,
    COMMITTED_PROCEDURAL_PRIMITIVE_HIT
};
```

Return value for RayQuery::CommittedStatus().

Value	Definition
COMMITTED_NOTHING	No hits have been committed yet.
COMMITTED_TRIANGLE_HIT	Closest hit so far is a triangle, a result of either the shader previously calling RayQuery::CommitNonOpaqueTriangleHit() or a fixed function opaque triangle intersection.
COMMITTED_PROCEDURAL_PRIMITIVE_HIT	Closest hit so far is a procedural primitive, a result of the shader previously calling RayQuery::CommittProceduralPrimitiveHit().

CANDIDATE_TYPE

```
enum CANDIDATE_TYPE : uint
{
    CANDIDATE_NON_OPAQUE_TRIANGLE,
    CANDIDATE_PROCEDURAL_PRIMITIVE
};
```

Return value for RayQuery::CandidateType().

Value	Definition
`CANDIDATE_NON_OPAQUE_TRIANGLE`	Acceleration structure traversal has encountered a non opaque triangle (that would be the closest hit so far if committed) for the shader to evaluate. If the shader decides this is opaque, it needs to call RayQuery::CommitNonOpaqueTriangleHit().
`CANDIDATE_PROCEDURAL_PRIMITIVE`	Acceleration structure traversal has encountered a procedural primitive for the shader to evaluate. It is up to the shader to calculate all possible intersections for this procedural primitive and commit at most one hit that it sees would be the closest so far, via RayQuery::CommitProceduralPrimitiveHit().

RayQuery TraceRayInline

Initialize parameters for an inline raytrace, invoking no other shader invocations, into an acceleration structure. Actual traversal does not begin yet. The calling shader manages the progress of the query via returned RayQuery object.

See Inline raytracing for an overview and the TraceRayInline() control flow diagram for more detail. Pseudocode examples are here.

This intrinsic function definition is equivalent to the following function:

```
void RayQuery::TraceRayInline(
    RaytracingAccelerationStructure AccelerationStructure,
    uint RayFlags,
    uint InstanceInclusionMask,
    RayDesc Ray);
```

Parameter	Definition
`RaytracingAccelerationStructure AccelerationStructure`	Top-level acceleration structure to use. Specifying a NULL acceleration structure forces a miss.
`uint RayFlags`	Valid combination of Ray flags. Only defined ray flags are propagated by the system, e.g. visible to the RayFlags() shader intrinsic. These flags are OR'd with the RayQuery's ray flags, and the combination must be valid (see the definition of each flag).

Parameter	Definition
`uint InstanceInclusionMask`	Bottom 8 bits of InstanceInclusionMask are used to include/reject geometry instances based on the InstanceMask in each instance: `if(! ((InstanceInclusionMask & InstanceMask) & 0xff)) { ignore intersection } </p>`
`RayDesc Ray`	Ray to be traced. See Ray-extents for bounds on valid ray parameters.
`Return: RayQuery`	Opaque object defining the state of a ray trace operation with no shader invocations. The caller calls methods on this object to participate in the acceleration structure traversal for as long as desired or until it ends. See RayQuery.

RayQuery intrinsics illustrates when this is valid to call.

TraceRayInline examples

- Example 1: Trivially get a simple hit/miss from tracing a ray.
- Example 2: More general case, handling all the possible states.
- Example 3: Expensive scenario with simultaneous traces.

TraceRayInline example 1

(back to examples list)

Trivial example, just get a simple triangle hit/miss from tracing a ray.

Here is a diagram that matches this example, illustrating what happens behind the scenes: Specialized TraceRayInline control flow.

```
RaytracingAccelerationStructure myAccelerationStructure : register(t3);

float4 MyPixelShader(float2 uv : TEXCOORD) : SV_Target0
{
    ...
    // Instantiate ray query object.
    // Template parameter allows driver to generate a specialized
    // implementation.
    RayQuery<RAY_FLAG_CULL_NON_OPAQUE |
             RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES |
             RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH> q;
```

```
        // Set up a trace.  No work is done yet.
        q.TraceRayInline(
            myAccelerationStructure,
            myRayFlags, // OR'd with flags above
            myInstanceMask,
            myRay);

        // Proceed() below is where behind-the-scenes traversal happens,
        // including the heaviest of any driver inlined code.
        // In this simplest of scenarios, Proceed() only needs
        // to be called once rather than a loop:
        // Based on the template specialization above,
        // traversal completion is guaranteed.
        q.Proceed();

        // Examine and act on the result of the traversal.
        // Was a hit committed?
        if(q.CommittedStatus()) == COMMITTED_TRIANGLE_HIT)
        {
            ShadeMyTriangleHit(
                q.CommittedInstanceIndex(),
                q.CommittedPrimitiveIndex(),
                q.CommittedGeometryIndex(),
                q.CommittedRayT(),
                q.CommittedTriangleBarycentrics(),
                q.CommittedTriangleFrontFace() );
        }
        else // COMMITTED_NOTHING
             // From template specialization,
             // COMMITTED_PROCEDURAL_PRIMITIVE can't happen.
        {
            // Do miss shading
            MyMissColorCalculation(
                q.WorldRayOrigin(),
                q.WorldRayDirection());
        }
        ...
    }
```

TraceRayInline example 2

(back to examples list)

More general case, handling all the possible states.

This a diagram illustrates the full control flow to support this type of scenario: TraceRayInline control flow.

```
RaytracingAccelerationStructure myAccelerationStructure : register(t3);

struct MyCustomAttrIntersectionAttributes { float4 a; float3 b; }
```

```
[numthreads(64,1,1)]
void MyComputeShader(uint3 DTid : SV_DispatchThreadID)
{
    ...
    // Instantiate ray query object.
    // Template parameter allows driver to generate a specialized
    // implemenation.  No specialization in this example.
    RayQuery<RAY_FLAG_NONE> q;

    // Set up a trace
    q.TraceRayInline(
        myAccelerationStructure,
        myRayFlags,
        myInstanceMask,
        myRay);

    // Storage for procedural primitive hit attributes
    MyCustomIntersectionAttributes committedCustomAttribs;

    // Proceed() is where behind-the-scenes traversal happens,
    // including the heaviest of any driver inlined code.
    // Returns TRUE if there's a task for the shader to perform
    // as part of traversal
    while(q.Proceed())
    {
        switch(q.CandidateType())
        {
        case CANDIDATE_PROCEDURAL_PRIMITIVE:
        {
            float tHit;
            MyCustomIntersectionAttributes candidateAttribs;

            // For procedural primitives, opacity is handled manually -
            // if an intersection is determined to not be opaque, just don't consider it
            // as a candidate.
            while(MyProceduralIntersectionEnumerator(
                tHit,
                candidateAttribs,
                q.CandidateInstanceIndex(),
                q.CandidatePrimitiveIndex(),
                q.CandidatetGeometryIndex()))
            {
                if( (q.RayTMin() <= tHit) && (tHit <= q.CommittedRayT()) )
                {
                    if(q.CandidateProceduralPrimitiveNonOpaque() &&
                        !MyProceduralAlphaTestLogic(
                            tHit,
                            attribs,
                            q.CandidateInstanceIndex(),
                            q.CandidatePrimitiveIndex(),
                            q.CandidateGeometryIndex()))
                    {
                        continue; // non opaque
                    }
```

```
                q.CommitProceduralPrimitiveHit(tHit);
                committedCustomAttribs = candidateAttribs;
            }
        }
        break;
    }
    case CANDIDATE_NON_OPAQUE_TRIANGLE:
    {
        if( MyAlphaTestLogic(
            q.CandidateInstanceIndex(),
            q.CandidatePrimitiveIndex(),
            q.CandidateGeometryIndex(),
            q.CandidateTriangleRayT(),
            q.CandidateTriangleBarycentrics(),
            q.CandidateTriangleFrontFace() )
        {
            q.CommitNonOpaqueTriangleHit();
        }
        if(MyLogicSaysStopSearchingForSomeReason()) // not typically applicable
        {
            q.Abort(); // Stop traversing and next call to Proceed()
                       // will return FALSE.
                       // Post-traversal results will just be based
                       // on what has been encountered so far.
        }
        break;
    }
    }
}
switch(q.CommittedStatus())
{
case COMMITTED_TRIANGLE_HIT:
{
    // Do hit shading
    ShadeMyTriangleHit(
        q.CommittedInstanceIndex(),
        q.CommittedPrimitiveIndex(),
        q.CommittedGeometryIndex(),
        q.CommittedRayT(),
        q.CommittedTriangleBarycentrics(),
        q.CommittedTriangleFrontFace() );
    break;
}
case COMMITTED_PROCEDURAL_PRIMITIVE_HIT:
{
    // Do hit shading for procedural hit,
    // using manually saved hit attributes (customAttribs)
    ShadeMyProceduralPrimitiveHit(
        committedCustomAttribs,
        q.CommittedInstanceIndex(),
        q.CommittedPrimitiveIndex(),
        q.CommittedGeometryIndex(),
        q.CommittedRayT());
    break;
```

```
        }
        case COMMITTED_NOTHING:
        {
            // Do miss shading
            MyMissColorCalculation(
                q.WorldRayOrigin(),
                q.WorldRayDirection());
            break;
        }
        }
        ...
    }
```

TraceRayInline example 3

(back to examples list)

Expensive scenario with multiple simultaneous traces:

```
    RaytracingAccelerationStructure myAccelerationStructure : register(t3);

    float4 MyPixelShader(float2 uv : TEXCOORD) : SV_Target0
    {
        ...
        RayQuery<RAY_FLAG_FORCE_OPAQUE> rayQ1;
        rayQ1.TraceRayInline(
          myAccelerationStructure,
          RAY_FLAG_NONE,
          myInstanceMask,
          myRay1);

        RayQuery<RAY_FLAG_FORCE_OPAQUE> rayQ2;
        rayQ2.TraceRayInline(
          myAccelerationStructure,
          RAY_FLAG_NONE,
          myInstanceMask,
          myRay2);

        // traverse rayQ1 and rayQ2 state machines simultaneously for some reason

        ...

        // Assignment from another generates a reference:
        RayQuery<RAY_FLAG_FORCE_OPAQUE> rayQ3 = rayQ2; // template config must match
                                        // rayQ3 and rayQ2 are aliases of each other

        ...

        // Reusing rayQ1 for a new query:
        // This is the one part of this example that's more likely to be useful: tracing a ray
        // sequentially after a previous trace is finished (so just reusing it's state for cor
```

```
    // sequentially after a previous trace is finished (so just reusing it's state for co
    // It's also fine to just declare a different query object, relying on the compiler
    // to notice the lifetimes of each don't overlap.
    rayQ1.TraceRayInline(
      myAccelerationStructure,
      RAY_FLAG_NONE,
      myInstanceMask,
      myRay1);
    // This could be done any time, no need to wait for the original query to be in some s
    // to be allowed to replace with a new trace.
}
```

RayQuery Proceed

```
bool RayQuery::Proceed();
```

This is the main worker function for an inline raytrace. While RayQuery::TraceRayInline() initializes the parameters for a raytrace, `RayQuery::Proceed()` is the function that causes the system to search the acceleration structure for hit candidates.

If candidates are found, they are either automatically processed in the case of opaque triangles, or the function returns to the shader (see `TRUE` below) for assistance in deciding what to do next - in particular when non-opaque triangles or procedural primitives are encountered.

Parameter	Definition
Return: `bool`	`TRUE` means there is a candidate for a hit that requires shader consideration. Calling RayQuery::CandidateType() after this reveals what the situation is. `FALSE` means that the search for hit candidates is finished. Once `RayQuery::Proceed()` has returned `FALSE` for a given RayQuery::TraceRayInline(), further calls to `RayQuery::Proceed()` will continue to return `FALSE` and do nothing, unless the shader chooses to issue a new RayQuery::TraceRayInline() call on the `RayQuery` object.

> For an inline raytrace to be at all useful, `RayQuery::Proceed()` must be called at least once, and typically called repeatedly until it returns `FALSE`, unless the shader chooses to just bail out of a search early for any reason.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery Abort

```
void RayQuery::Abort();
```

Force a ray query into the finished state. In particular, calls to RayQuery::Proceed() will return `FALSE`.

`RayQuery::Abort()` can be called any time after RayQuery::TraceRayInline() has been called.

RayQuery intrinsics also illustrates when this is valid to call.

> This method is provided for convenience for telling an outer loop that calls `RayQuery::Proceed()` such as below to break out, from within some nested code within the loop. The shader can accomplish the same thing manually as well, effectively abandoning a search. Either way, RayQuery::CommittedStatus() can be called to inform how to do any final shading.

```
while(rayQ.Proceed()) // `FALSE` after Abort() below
                      // (or search actually ended)
{
    ...
    {
        ...
        {
            rayQ.Abort();
        }
        ...
    }
    ...
}
```

RayQuery CandidateType

```
CANDIDATE_TYPE RayQuery::CandidateType()
```

When RayQuery::Proceed() has returned `TRUE` and there is a candidate hit for the shader to evaluate, `RayQuery::CandidateType()` reveals what type of candidate it is.

Parameter	Definition
`Return value: CANDIDATE_TYPE`	See CANDIDATE_TYPE.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateProceduralPrimitiveNonOpaque

```
bool RayQuery::CandidateProceduralPrimitiveNonOpaque()
```

When RayQuery::CandidateType() has returned `CANDIDATE_PROCEDURAL_PRIMITIVE`, this indicates whether the acceleration structure combined with ray flags consider this procedural primitive candidate to be non-opaque. The only action the system might have taken with this information is to cull opaque or non-opaque primitives if the shader requested it via RAY_FLAGS. If a candidate has been produced however, culling obviously did not apply. At this point it is completely up to the shader if it wants to act any further on this opaque/non-opaque information.

The shader could choose to not even both calling this method and make it's own opacity determination for the procedural hits it finds, ignoring whatever the acceleration structure / ray flags may have decided.

RayQuery CommitNonOpaqueTriangleHit

```
void RayQuery::CommitNonOpaqueTriangleHit()
```

When RayQuery::CandidateType() has returned `CANDIDATE_NON_OPAQUE_TRIANGLE` and the shader determines this hit location is actually opaque, it needs to call `RayQuery::CommitNonOpaqueTriangleHit()`. `RayQuery::CommitNonOpaqueTriangleHit()` can be called multiple times per non opaque triangle candidate - after the first call, subsequent calls simply have no effect for the current candidate.

The system will have already determined this candidate hit T would > the ray's TMin and < TMax (RayQuery::CommittedRayT()), so the shader need not manually verify these conditions.

Once committed, the system remembers hit properties like barycentrics, front/back facing of the hit, acceleration structure node information like PrimitiveIndex. As long as this remains the closest hit, RayQuery::CommittedTriangleBarycentrics() etc. will return the properties for this hit.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommitProceduralPrimitiveHit

```
void RayQuery::CommitProceduralPrimitiveHit(float tHit)
```

When RayQuery::CandidateType() has returned `CANDIDATE_PROCEDURAL_PRIMITIVE`, the shader is responsible for procedurally finding all intersections for this candidate. If the shader is manually accumulating transparency, it can do so manually without the system knowing this. The one responsibility the system expects of the shader is that it must only commit hit(s) when it manually determines that a given hit being reported satisfies TMin <= t <= TMax (

RayQuery::CommittedRayT()). When the range condiditon is satisfied, the shader can validly commit a hit via `RayQuery::CommitProceduralPrimitiveHit()` . See ray extents for some further discussion.

The shader must manually store (e.g. in local variables) any hit information that it might want later. The exception is data that the system tracks, such as the t value, or acceleration structure node information such as InstanceIndex, which can be retrieved via methods like RayQuery::CommittedInstanceIndex() for whatever the current closest commited hit is.

It is ok to call `RayQuery::CommitProceduralPrimitiveHit()` multiple times while processing a given candidate procedural primitive. In this case, each time it is called, the shader must ensure the hit being committed is within the most recent [TMin … TMax] (inclusive) range based on previous hits that have been committed. Another way to get the job done is to manually determine which one hit is closest in range (if any) and just commit once.

RayQuery intrinsics illustrates when this is valid to call.

Parameter	Definition
`float tHit`	t value for intersection that shader determined is closest so far and >= the ray's TMin.

RayQuery CommittedStatus

```
COMMITTED_STATUS CommittedStatus()
```

Status of the closest hit that has been committed so far (if any). This can be called any time after RayQuery::TraceRayInline() has been called.

RayQuery intrinsics illustrates when this is valid to call.

Parameter	Definition
`Return value: COMMITTED_STATUS`	See COMMITTED_STATUS.

RayQuery RayFlags

This is a uint containing the current ray flags (template flags ORd with dynamic flags).

```
uint RayQuery::RayFlags();
```

This can be useful if, for instance, during handling of a procedural primitive candidatean app wants to look at the current ray's culling flags and apply corresponding culling in its custom intersection code.

> It is arguable that this method appears redundant given the shader could just remember what flags it originally passed into RayQuery::TraceRayInline(). The rationale is that since the system needs to keep this information anyway, no need for the app to waste a live register storing it, so in effect it can eliminate redundancy.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery WorldRayOrigin

The world-space origin for the current ray.

```
float3 RayQuery::WorldRayOrigin();
```

> It is arguable that this method appears redundant given the shader could just remember what flags it originally passed into RayQuery::TraceRayInline(). The rationale is that since the system needs to keep this information anyway, no need for the app to waste a live register storing it, so in effect it can eliminate redundancy.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery WorldRayDirection

The world-space direction for the current ray.

```
float3 RayQuery::WorldRayDirection();
```

> It is arguable that this method appears redundant given the shader could just remember what flags it originally passed into RayQuery::TraceRayInline(). The rationale is that since the system needs to keep this information anyway, no need for the app to waste a live register storing it, so in effect it can eliminate redundancy.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery RayTMin

This is a float representing the parametric starting point for the ray.

```
float RayQuery::RayTMin();
```

RayTMin defines the starting point of the ray according to the following formula: `Origin + (Direction \* RayTMin)`. `Origin` and `Direction` may be in either world or object space, which results in either a world or an object space starting point.

RayTMin is defined when calling RayQuery::TraceRayInline(), and is constant for the duration of that call.

> It is arguable that this method appears redundant given the shader could just remember what flags it originally passed into RayQuery::TraceRayInline(). The rationale is that since the system needs to keep this information anyway, no need for the app to waste a live register storing it, so in effect it can eliminate redundancy.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateTriangleRayT

This is a float representing the parametric distance at which a candidate triangle for hit consideration lies.

```
float RayQuery::CandidateTriangleRayT();
```

`CandidateTriangleRayT()` defines the candidate point for non-opaque triangles along the ray according to the following formula: `Origin + (Direction * CandidateTriangleRayT)`. `Origin` and `Direction` may be in either world or object space, which results in either a world or an object space ending point.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedRayT

This is a float representing the parametric distance at which the closest committed hit so far lies.

```
float RayQuery::CommittedRayT();
```

`CommittedRayT()` defines the current TMax point along the ray according to the following formula: `Origin + (Direction * CommittedRayT)`. `Origin` and `Direction` may be in either world or object space, which results in either a world or an object space ending point.

`CommittedRayT()` is initialized by the RayQuery::TraceRayInline() call to `RayDesc::TMax`, and updated during the trace query as new intersections are commited.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateInstanceIndex

The autogenerated index of the current instance in the top-level structure for the current hit candidate.

```
uint RayQuery::CandidateInstanceIndex();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateInstanceID

The user-provided `InstanceID` on the bottom-level acceleration structure instance within the top-level structure for the current hit candidate.

```
uint RayQuery::CandidateInstanceID();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateInstanceContributionToHitGroupIndex

The user-provided `InstanceContributionToHitGroupIndex` on the bottom-level acceleration structure instance within the top-level structure for the current hit candidate.

```
uint RayQuery::CandidateInstanceContributionToHitGroupIndex();
```

With inline raytracing, this value has no functional effect, even though the name describes contributing to hit group indexing. The name refers to the behavior with dynamic-shader-based raytracing. It is simply made available via RayQuery for completeness, enabling inline raytracing to see everything in an acceleration structure.

> An app might use this a way to store another arbitrary user value per instance into instance data. Or it might be sharing an acceleration structure between dynamic-shader-based and inline raytracing: In the dynamic-shader-based case, the value participates in shader table indexing and in the inline case the app may achieve some similar equivalent effect manually indexing into its own data structures.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateGeometryIndex

The autogenerated index of the current geometry in the bottom-level acceleration structure for the current hit candidate.

```
uint RayQuery::CandidateGeometryIndex();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidatePrimitiveIndex

The autogenerated index of the primitive within the geometry inside the bottom-level acceleration structure instance for the current hit candidate.

```
uint RayQuery::CandidatePrimitiveIndex();
```

For `D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES`, this is the triangle index within the geometry object.

For `D3D12_RAYTRACING_GEOMETRY_TYPE_PROCEDURAL_PRIMITIVE_AABBS`, this is the index into the AABB array defining the geometry object.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateObjectRayOrigin

Object-space origin for the current ray. Object-space refers to the space of the current bottom-level acceleration structure for the current hit candidate.

```
float3 RayQuery::CandidateObjectRayOrigin();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateObjectRayDirection

Object-space direction for the current ray. Object-space refers to the space of the current bottom-level acceleration structure for the current hit candidate.

```
float3 RayQuery::CandidateObjectRayDirection();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateObjectToWorld3x4

Matrix for transforming from object-space to world-space. Object-space refers to the space of the current bottom-level acceleration structure for the current hit candidate.

```
float3x4 RayQuery::CandidateObjectToWorld3x4();
```

The only difference between this and `CandidateObjectToWorld4x3()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateObjectToWorld4x3

Matrix for transforming from object-space to world-space. Object-space refers to the space of the current bottom-level acceleration structure for the current hit candidate.

```
float4x3 RayQuery::CandidateObjectToWorld4x3();
```

The only difference between this and `CandidateObjectToWorld3x4()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateWorldToObject3x4

Matrix for transforming from world-space to object-space. Object-space refers to the space of the current bottom-level acceleration structure for the current hit canddiate.

```
float3x4 RayQuery::CandidateWorldToObject3x4();
```

The only difference between this and `CandidateWorldToObject4x3()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateWorldToObject4x3

Matrix for transforming from world-space to object-space. Object-space refers to the space of the current bottom-level acceleration structure for the current hit candidate.

```
float3x4 RayQuery::CandidateWorldToObject4x3();
```

The only difference between this and `CandidateWorldToObject3x4()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedInstanceIndex

The autogenerated index of the instance in the top-level structure for the closest hit committed so far.

```
uint RayQuery::CommittedInstanceIndex();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedInstanceID

The user-provided `InstanceID` on the bottom-level acceleration structure instance within the top-level structure for the closest hit committed so far.

```
uint RayQuery::CommittedInstanceID();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedInstanceContributionToHitGroupIndex

The user-provided `InstanceContributionToHitGroupIndex` on the bottom-level acceleration structure instance within the top-level structure for the closest hit committed so far.

```
uint RayQuery::CommittedInstanceContributionToHitGroupIndex();
```

With inline raytracing, this value has no functional effect, even though the name describes contributing to hit group indexing. The name refers to the behavior with dynamic-shader-based raytracing. It is simply made available via RayQuery for completeness, enabling inline raytracing to see everything in an acceleration structure.

> An app might use this a way to store another arbitrary user value per instance into instance data. Or it might be sharing an acceleration structure between dynamic-shader-based and inline raytracing: In the dynamic-shader-based case, the value participates in shader table indexing and in the inline case the app may implement some similar equivalent effect manually indexing into its own data structures.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedGeometryIndex

The autogenerated index of the geometry in the bottom-level acceleration structure for the closest hit committed so far.

```
uint RayQuery::CommittedGeometryIndex();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedPrimitiveIndex

The autogenerated index of the primitive within the geometry inside the bottom-level acceleration structure instance for the closest hit committed so far.

```
uint RayQuery::CommittedPrimitiveIndex();
```

For `D3D12_RAYTRACING_GEOMETRY_TYPE_TRIANGLES`, this is the triangle index within the geometry object.

For `D3D12_RAYTRACING_GEOMETRY_TYPE_PROCEDURAL_PRIMITIVE_AABBS`, this is the index into the AABB array defining the geometry object.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedObjectRayOrigin

Object-space origin for the ray. Object-space refers to the space of the bottom-level acceleration structure for the closest hit committed so far.

```
float3 RayQuery::CommittedObjectRayOrigin();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedObjectRayDirection

Object-space direction for the ray. Object-space refers to the space of the bottom-level acceleration structure for the closest hit committed so far.

```
float3 CommittedObjectRayDirection();
```

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedObjectToWorld3x4

Matrix for transforming from object-space to world-space. Object-space refers to the space of the bottom-level acceleration structure for the closest hit committed so far.

```
float3x4 RayQuery::CommittedObjectToWorld3x4();
```

The only difference between this and `CommittedObjectToWorld4x3()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedObjectToWorld4x3

Matrix for transforming from object-space to world-space. Object-space refers to the space of the bottom-level acceleration structure for the closest hit committed so far.

```
float4x3 RayQuery::CommittedObjectToWorld4x3();
```

The only difference between this and `CommittedObjectToWorld3x4()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedWorldToObject3x4

Matrix for transforming from world-space to object-space. Object-space refers to the space of the bottom-level acceleration structure for the closest hit committed so far.

```
float3x4 RayQuery::CommittedWorldToObject3x4();
```

The only difference between this and `CommittedWorldToObject4x3()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CommittedWorldToObject4x3

Matrix for transforming from world-space to object-space. Object-space refers to the space of the bottom-level acceleration structure for the closest hit committed so far.

```
float3x4 RayQuery::CommittedWorldToObject4x3();
```

The only difference between this and `CommittedWorldToObject3x4()` is the matrix is transposed – use whichever is convenient.

RayQuery intrinsics illustrates when this is valid to call.

RayQuery CandidateTriangleBarycentrics

```
float2 RayQuery::CandidateTriangleBarycentrics
```

Applicable for the current hit candidate, if RayQuery::CandidateType() returns `CANDIDATE_NON_OPAQUE_TRIANGLE`. Returns hit location barycentric coordinates.

RayQuery intrinsics illustrates when this is valid to call.

Parameter	Definition
Return value: float2	Given attributes `a0`, `a1` and `a2` for the 3 vertices of a triangle, `float2.x` is the weight for `a1` and `float2.y` is the weight for `a2`. For example, the app can interpolate by doing: `a = a0 + float2.x \* (a1-a0) + float2.y\* (a2 -- a0)`.

RayQuery CandidateTriangleFrontFace

```
bool RayQuery::CandidateTriangleFrontFace
```

Applicable for the current hit candidate, if RayQuery::CandidateType() returns `CANDIDATE_NON_OPAQUE_TRIANGLE`. Reveals whether hit triangle is front of back facing.

RayQuery intrinsics illustrates when this is valid to call.

Parameter	Definition
`Return value: bool`	`TRUE` means front face, `FALSE` means back face.

RayQuery CommittedTriangleBarycentrics

```
float2 RayQuery::CommittedTriangleBarycentrics
```

Applicable for the closest committed hit so far, if that hit is a triangle (which can be determined from RayQuery::CommittedStatus(). Returns hit location barycentric coordinates.

RayQuery intrinsics illustrates when this is valid to call.

Parameter	Definition
`Return value: float2`	Given attributes `a0`, `a1` and `a2` for the 3 vertices of a triangle, `float2.x` is the weight for `a1` and `float2.y` is the weight for `a2`. For example, the app can interpolate by doing: `a = a0 + float2.x \* (a1-a0) + float2.y\* (a2 -- a0)`.

RayQuery CommittedTriangleFrontFace

```
bool RayQuery::CommittedTriangleFrontFace
```

Applicable for the closest committed hit so far, if that hit is a triangle (which can be determined from RayQuery::CommittedStatus(). Reveals whether hit triangle is front of back facing.

RayQuery intrinsics illustrates when this is valid to call.

Parameter	Definition
`Return value: bool`	`TRUE` means front face, `FALSE` means back face.

Payload access qualifiers

Shader models 6.6 and 6.7 add payload access qualifiers (PAQs) to the ray payload structure. PAQs are annotations which describe the read and write semantics of a payload field, that is, which shader stages read or write a given field. The added semantic information can help implementations reduce register pressure and can avoid spilling of payload state to memory. This incentivizes the use of the narrowest-possible qualifiers for each payload field.

Availability

Prior to shader model 6.6, payload access qualifiers (PAQs) are not supported.

With SM 6.6, PAQs are disabled by default. The user may opt-in to the feature by using the `-enable-payload-qualifiers` command line flag in DXC.

With SM 6.7 and higher, PAQs are enabled by default. The user may opt-out of the feature by using the `-disable-payload-qualifiers` command line flag in DXC.

It is legal to mix annotated and unannotated payloads within the same library / state object.

Payload size

With payload access qualifiers (PAQs), the `MaxPayloadSizeInBytes` property of D3D12_RAYTRACING_SHADER_CONFIG is no longer needed. The field is ignored by drivers if PAQs are enabled (the default per above) in SM 6.7 or higher.

For SM 6.7 and higher but with PAQs disabled, `MaxPayloadSizeInBytes` is still used.

For SM 6.6, in order to ease the transition for driver implementers, applications must still set the `MaxPayloadSizeInBytes` regardless of whether PAQs are used or not.

Syntax

Any structure type used as payload must carry the `[raypayload]` type attribute as part of the type declaration as follows:

```
struct [raypayload] MyPayload{ ... };
```

Only structs marked as payload as shown above can be used as argument to TraceRay calls and as payload parameter in closesthit, anyhit or miss shaders.

Payload types require annotating each member variable with PAQs. The payload access qualifier (PAQ) syntax follows the syntax of resource bindings. A valid annotation follows this syntax:

```
Type Field : qualifier1([s0, s1, ...]) : qualifier2([s0, s1, ...]);
```

The two valid qualifiers are `read` and `write`. Each qualifier carries an argument list, containing the shader stages it applies to (`s0..sN` in the above definition). Valid shader stages are: `anyhit`, `closesthit`, `miss`, `caller`.

Each field must declare one `read` and one `write` qualifier. Qualifier argument lists can be empty.

PAQs can only be specified for scalar, array, struct, vector, or matrix types. For payload types containing other types (i.e., structs with PAQs), the PAQs must be specified in the nested type and directly annotating the member is not allowed.

Semantics

Generally speaking, the `read` qualifier indicates that a shader stage reads the payload field, and the `write` qualifier indicates that a shader stage writes to the field.

For the application developer, the following classification provides a description of the qualifier's behavior that should suffice as a mental model in most scenarios. For a more precise definition of the semantics, see the next section.

anyhit/closesthit/miss stages

qualifier	semantic
`read`	Indicates that for the given stage, the payload field is available for reading and will contain the last value written by a previous stage.
`write`	Indicates that the payload field will be overwritten by the given stage, if the stage executes. The field will be overwritten regardless of whether the executed shader explicitly assigns a value. If a value is not explicitly written by the shader, then an undefined value is written to the payload field at the end of the shader. If the given stage does not execute, the payload field remains unmodified. A shader stage may not execute for various reasons. Examples include ray flags (`FORCE_OPAQUE` does not execute anyhit, `SKIP_CLOSESTHIT` does not execute closesthit) or dynamic behavior (rays that miss all geometry do not execute closeshit, rays that hit geometry do not execute miss, etc). *Note that invoking a NULL shader identifier in the Shader Table is equivalent to executing an empty shader, so in that case the stage counts as executed.*
`read+write`	Indicates that the given stage may read and/or write the payload field. This is analogous to inout function parameters. If the shader does not explicitly assign a value or if the stage is not executed, the payload field remains unmodified.
`<none>`	The payload field is neither available for reading nor are its contents modified by the given stage.

caller stage

The `caller` stage denotes the caller of the TraceRay function (i.e., most commonly a raygeneration shader). Fields that represent "inputs" to TraceRay (for example a random seed) must first be written by the caller, so the payload access qualifier (PAQ) for such fields must include `write(caller)`. Fields that represent "outputs" from TraceRay (for example an output color) must be read by the caller, so the PAQ for such fields must include `read(caller)`. Fields can be both inputs and outputs to TraceRay by specifying both `read(caller)` and `write(caller)`.

Detailed semantics

Payload access qualifiers (PAQs) take effect at the transition between shader stages. A shader stage transition occurs when calling or returning from TraceRay, or any time an anyhit/closesthit/miss shader is entered or exited. Conceptually, each shader stage that receives a payload as an argument creates the payload parameter as a local working copy of the actual payload attached to the ray. The PAQs then determine which fields are copied between the local copy and the actual payload when entering and exiting the shader stage.

Specifically, the following semantics apply:

- At the beginning of a TraceRay call, any fields of the payload type that are marked `write(caller)` are copied from the payload argument passed to TraceRay into the actual payload. All other fields of the actual payload have undefined contents.
- At the beginning of execution of a shader stage that receives a payload, any fields that are marked `read` for that stage are copied from the actual payload to the parameter the shader receives. All other fields of the input parameter are left undefined.
- At the end of execution of a shader stage that receives a payload, any fields that are marked `write` for that stage are copied back from the parameter of the shader to the actual payload. Any values written to other fields of the parameter are ignored.
- At the end of execution of a TraceRay call, any fields of the payload type that are marked `read(caller)` are copied from the actual payload back to the payload argument passed to TraceRay. All other fields of the payload parameter will have undefined contents.

The implementation can organize the actual payload however it wants, and need only preserve the values of payload fields that have well-defined values and might possibly be read in the future.

Local working copy

Because payload access qualifiers (PAQs) only affect stage transitions, the compiler does not restrict how shaders may access the local working copy of the payload; the local copy behaves exactly like an un-annotated struct variable. This ensures that variables of payload type can freely

be assigned to each other. In particular, it also enables passing payloads as function arguments (in which case in/out semantics apply just like they would for regular struct parameters).

It is the responsibility of the developer to ensure that shaders honor the specified PAQs when accessing payload fields to achieve the desired behavior. To help guard the developer against unintended effects, the compiler will attempt to warn whenever accesses are detected that could lead to undefined values or ignored writes.

Shader stage sequence

The general sequence of relevant shader stages is as follows. (intersection shaders are left out because they cannot access payloads).

```
caller -> anyhit -> (closesthit|miss) -> caller
          ^  |
          |__|
```

Because multiple anyhit shaders can be executed during the course of a TraceRay operation, 'anyhit' both precedes and succeeds itself.

A TraceRay call may not invoke any shaders at all (e.g. a ray that hits triangle geometry marked as opaque, while also specifying the `SKIP_CLOSESTHIT` ray flag). In that case, payload access qualifiers still apply to the `caller -> caller` transition.

The following rules are enforced by the compiler at the payload declaration:

1) Any `read` stage must be preceded by a `write` stage 2) Any `write` stage must be succeeded by a `read` stage

Example

An example payload structure with payload access qualifiers is shown below:

```
struct Nested { float a, b, c; }

struct [raypayload] MyPayload
{
    // "Pure" output from closesthit or miss:
    float4 irradiance : read(caller) : write(closesthit, miss);

    // "Pure" input into all stages, not preserved over TraceRay
    uint seed : read(anyhit,closesthit,miss) : write(caller);

    // Simple input into all stages, preserved over TraceRay
    uint seed : read(anyhit,closesthit,miss,caller) : write(caller);
```

```
    // In-out flag, overwritten by miss (e.g. for shadow/visibility):
    bool hasHit : read(caller) : write(caller,miss);

    // Anyhits communicating amongst themselves, with initialization from caller
    float a : read(anyhit) : write(caller,anyhit);

    // Nested struct which itself does not have PAQs:
    Nested n : read(caller) : write(closesthit,miss);

    // Nested payload struct which itself has PAQs:
    MyBasePayload base;
};
```

Guidelines

Here are some guidelines to help developers specify payload access qualifier definitions correctly:

1) Does the caller need to initialize the field before calling TraceRay Add `caller` to `write`. 2) Does the caller use the returned field after calling TraceRay (including in cases like loops)? Add `caller` to `read`. 3) Does any shader of an anyhit/closesthit/miss stage read the field, but no shader in the same stage ever writes it? Add the corresponding shader stage to `read` but not `write`. 4) Do all shaders of an anyhit/closesthit/miss stage write the field unconditionally and never read it? Add the corresponding shader stage to `write` but not `read`. 5) Does any anyhit/closesthit/miss shader conditionally modify the field, or do some shaders in the stage write the field while others don't? Try to make the `write` unconditional in all shaders and apply guideline (4). If that is not possible, add the stage to both `read` and `write`. 6) Specify as few qualifiers/stages as possible for maximum performance. Try to make fields "pure inputs" or "pure outputs" (see later examples) where possible.

Optimization potential

1) Shortened lifetimes

```
struct [raypayload] MyPayload
{
    float ahInput : write(caller) : read(anyhit);
};
```

In this example, `ahInput` is not accessed after the anyhit stage, that is, its lifetime ends after anyhit. The implementation is therefore free to ignore this field for subsequent stages, which can reduce shader register pressure in closesthit and miss shaders.

2) Disjoint lifetimes

```
struct [raypayload] MyPayload
{
    float alpha : write(caller,anyhit) : read(closesthit);
    bool didHit : write(closesthit, miss) : read(caller);
    ...
};
```

In this example, the lifetime of `alpha` will end once the closesthit shader has read the value and resources (e.g. registers) allocated for `alpha` are free to be reused. Since the lifetimes of `alpha` and `didHit` are now provably disjoint, the implementation is allowed to reuse `alpha`'s register to propagate `didHit` back to the caller of TraceRay.

Advanced examples

Various accesses and recursive TraceRay

```
struct [raypayload] Payload
{
    float a : write(closesthit, miss) : read(caller);
    float b : write(miss) : read(caller);
    float c : write(caller, closesthit) : read(caller, closesthit);
    float d : write(caller) : read(closesthit, miss);
};

[shader("closesthit")]
void ClosestHit(inout Payload payload)
{
    float tmp1 = payload.a; // WARNING: reading undefined value ('a' not read(closesthit)
    float tmp2 = payload.b; // WARNING: reading undefined value ('b' not read(closesthit)
    float tmp3 = payload.c;
    float tmp4 = payload.d;

    payload.a = 3;
    payload.b = 3; // WARNING: write will be ignored after CH returns ('b' not write(close
    payload.c = 3;
    payload.d = 3; // WARNING: write will be ignored after CH returns ('d' not write(close

    Payload p;
    p.a = 3; // WARNING: value will be undefined inside TraceRay ('a' not write(caller))
    p.b = 3; // WARNING: value will be undefined inside TraceRay ('b' not write(caller))
    p.c = 3;
    p.d = 3;
    TraceRay(p);
    float tmp5 = p.a;
    float tmp6 = p.b;
    float tmp7 = p.c;
    float tmp8 = p.d; // WARNING: reading undefined value ('d' not read(caller))
```

```
    float tmp3 = p.a; // WARNING: reading undefined value ("a" not read(caller))

    Payload p2 = payload; // copying entire payload is OK, but inherits any potential unde
    TraceRay(p2);
}
```

Payload as function parameter

```
struct [raypayload] MyPayload
{
    float a : write(caller, closesthit, miss) : read(caller);
};

// could also use plain 'in' or 'out', behaves like normal struct parameter
void foo(inout MyPayload p)
{
    p.a = 123;
}

[shader("closesthit")]
void ClosestHit(inout MyPayload p)
{
    foo(p);
}
```

Forwarding payloads to recursive TraceRay calls

```
struct [raypayload] MyPayload
{
    float pureInput : write(caller) : read(closesthit);
    float pureOutput : write(closesthit) : read(caller);
};

[shader("closesthit")]
void ClosestHit(inout MyPayload p)
{
    // undefined contents due to lack of read(closesthit) and write(caller)
    float a = p.pureOutput;

    // This write will not be visible inside recursive closesthit stages, due to
    // the lack of write(caller) and read(closesthit) annotations. Once the recursive
    // TraceRay call returns, the value of p.pureOutput will be whatever the recursive
    // closesthit invocation assigned, or undefined if the recursive invocation did
    // not assign anything.
    p.pureOutput = 222;

    // This write will be visible inside the recursive closesthit stages, thanks to
```

```
    // write(caller). We are operating on the local copy of the payload here, so the
    // fact that p.pureInput does not specify write(closesthit) is irrelevant for this
    // point (it only matters at the stage transition out of closeshit).
    p.pureInput = 123;
    TraceRay(p);

    float c = p.pureInput; // now undefined due to lack of read(caller)
    float d = p.pureOutput; // result of recursive TraceRay, or undefined
                            // if the recursion didn't write

    p.pureOutput = 444; // visible to the caller
}
```

Pure input in a loop

```
struct [raypayload] MyPayload
{
    float pureInput : write(caller) : read(closesthit);
};

[shader("raygeneration")]
void Raygen()
{
    MyPayload p;

    p.pureInput = 123;

    while (condition)
    {
      // p.pureInput will be undefined after the first
      // loop iteration. Either specify read(caller)
      // to preserve the value or manually keep a copy
      // of the value live across the TraceRay call.
      TraceRay(p);
    }
}
```

Conditional pure output overwriting initial value

```
struct [raypayload] MyPayload
{
    float pureOutput : write(caller,closesthit) : read(caller);
};

[shader("closesthit")]
void ClosestHit(inout MyPayload p)
{
    if( condition )
```

```
            p.pureOutput = 123;
        else
        {
            // p.pureOutput becomes undefined because we do not
            // write it explicitly and omit read(closeshit)
        }
    }
```

Payload access qualifiers in DXIL

Payload Access Qualifiers (PAQs) are represented as metadata in DXIL, like type annotations. In contrast to type annotations, PAQs are attached to the `dx.dxrPayloadAnnotations` metadata node which references a single node that contains tag `kDxilPayloadAnnotationStructTag(0)` followed by a list of pairs consisting of an undef value of the payload type and a metadata node reference.

```
!dx.dxrPayloadAnnotations = !{!24}
|--->!24 = !{i32 0, %struct.Payload undef, !25, %struct.MyColor undef, !29}
     |--->!25 = !{!26, !27, !28}
     |     |--->!26 = !{i32 0, i32 0}
     |     |--->!27 = !{i32 0, i32 819}
     |     |--->!28 = !{i32 0, i32 51}
     |--->!29 = !{!30, !30, !30}
           |--->!30 = !{i32 0, i32 545}
```

The referenced metadata node must contain another reference to per-field metadata, hence every field in the payload type must be represented by one metadata note and the order must match the type's layout.

The per-field metadata contains the `kDxilPayloadFieldAnnotationAccessTag(0)` followed by a bitmask that stores the PAQs for this field. For each shader stage 4 bits are used in the bitmask. Bits 0-1 are used for the PAQs. Bit 0 is set it indicates that the shader stage has read access, bit 1 indicates write access.

The bits for each stage are:

Stage	Bits
Caller	0-3
Closesthit	4-7
Miss	8-11
Anyhit	12-15

Note: A field declared with another payload type must not have any bits set. The payload access information must be taken from the payload type, not the field.

DDI

General notes

Descriptor handle encodings

Recall that shader tables, containing application defined data, can hold 8 byte GPU descriptor handles when they are present in local root signatures.

Descriptor handles are generated by the app by adding a driver defined descriptor increment to a descriptor heap base address (also reported by the driver to the app). It is required that descriptor handles are globally unique across all descriptor heaps (regardless of descriptor heap type). This ensures that tools and the debug layer can cheaply understand the intent of the application when it places a descriptor in a shader table. It can be validated, for instance, that descriptors referenced by shader tables come from the descriptor heap currently bound on the command list (as opposed to being stale descriptor handles pointing into a previously used descriptor heap, or invalid descriptors).

State object DDIs

See State objects for an overview. DDIs generally mirror APIs (and follow DDI patterns in the rest of D3D12). Notable exceptions are called out below.

State subobjects

Most subobjects mirror the API, with exceptions listed here:

The runtime passes down any subjects that have been defined directly in DXIL libraries as plain subobjects in the DDI (the driver won't find the subobjects in the DXIL library that it gets).

The runtime also converts any subobject association definitions including default associations into an explicit list of associations for every exported function. So the driver doesn't have try to understand any rules about which subobjects need to be associated with which functions.

D3D12DDI_STATE_SUBOBJECT_TYPE

```
typedef enum D3D12DDI_STATE_SUBOBJECT_TYPE
{
    D3D12DDI_STATE_SUBOBJECT_TYPE_STATE_OBJECT_CONFIG = 0, // D3D12DDI_STATE_OBJECT_CONFIC
    D3D12DDI_STATE_SUBOBJECT_TYPE_GLOBAL_ROOT_SIGNATURE = 1, // D3D12DDI_GLOBAL_ROOT_SIGNA
    D3D12DDI_STATE_SUBOBJECT_TYPE_LOCAL_ROOT_SIGNATURE = 2, // D3D12DDI_LOCAL_ROOT_SIGNATU
    D3D12DDI_STATE_SUBOBJECT_TYPE_NODE_MASK = 3, // D3D12_NODE_MASK_0054
    // 4 unused
    D3D12DDI_STATE_SUBOBJECT_TYPE_DXIL_LIBRARY = 5, // D3D12DDI_DXIL_LIBRARY_DESC_0054
    D3D12DDI_STATE_SUBOBJECT_TYPE_EXISTING_COLLECTION = 6, // D3D12DDI_EXISTING_COLLECTION
    // skip value from API not needed in DDI
    // skip value from API not needed in DDI
    D3D12DDI_STATE_SUBOBJECT_TYPE_RAYTRACING_SHADER_CONFIG = 9, // D3D12DDI_RAYTRACING_SHA
    D3D12DDI_STATE_SUBOBJECT_TYPE_RAYTRACING_PIPELINE_CONFIG = 10, // D3D12DDI_RAYTRACING_
    D3D12DDI_STATE_SUBOBJECT_TYPE_HIT_GROUP = 11, //
    D3D12DDI_HIT_GROUP_DESC_0054

    // DDI only objects
    D3D12DDI_STATE_SUBOBJECT_TYPE_SHADER_EXPORT_SUMMARY = 0x100000, // D3D12DDI_FUNCTION_S
} D3D12DDI_STATE_SUBOBJECT_TYPE;
```

State object types driver sees. Note the list isn't exactly the same as what exists at the API: D3D12_STATE_OBJECT_TYPE. As discussed above, association subobjects at the API are converted into a per-function association list - D3D12DDI_FUNCTION_SUMMARY_0054.

The definitions of the various other DDI subobjects is omitted as they are identical to the API equivalents documented here. Minor exceptions where DDI subobject definitions differ slightly: in cases where in the API a subobject contains an API COM interface, the DDI contains the DDI handle. For example the ID3D12RootSignature* in D3D12_GLOBAL_ROOT_SIGNATURE appears in the DDI equivalent subobject, `D3D12DDI_GLOBAL_ROOT_SIGNATURE_0054` , with `D3D12DDI_HROOTSIGNATURE` hGlobalRootSignature.

D3D12DDI_STATE_SUBOBJECT_0054

```
typedef struct D3D12DDI_STATE_SUBOBJECT_0054
{
    D3D12DDI_STATE_SUBOBJECT_TYPE Type;
    const void* pDesc;
} D3D12DDI_STATE_SUBOBJECT_0054;
```

Parameter	Definition
`D3D12DDI_STATE_SUBOBJECT_TYPE` `Type`	Type of subobject. See D3D12DDI_STATE_SUBOBJECT_TYPE.

Parameter	Definition
`const void* pDesc`	Pointer to subobject whose contents depend on Type. For instance if Type is `D3D12DDI_STATE_SUBOBJECT_TYPE_LOCAL_ROOT_SIGNATURE`, the pDesc points to `D3D12DDI_GLOBAL_ROOT_SIGNATURE_0054`. Any data within the subobjects including shader bytecode is remains valid for the lifetime of the state object, as the memory is owned by the runtime, with no references to data the app passed in to CreateStateObject. So the app doesn't have to keep its creation data around and the driver doesn't need to make copies of any part of a state object definition.

D3D12DDI_STATE_SUBOBJECT_TYPE_SHADER_EXPORT_SUMMARY

D3D12DDI_STATE_SUBOBJECT_TYPE_SHADER_EXPORT_SUMMARY is a DDI subobject generated by the runtime during its state object validation and creation process. This provides any information the runtime determined, such as which subobjects have been associated with any given export, including resolving any defaults.

See Subobject association behavior for a discussion about how subobject associations work at the API. At the DDI, the driver simply sees the results of the association rules.

D3D12DDI_FUNCTION_SUMMARY_0054

```
typedef struct D3D12DDI_FUNCTION_SUMMARY_0054
{
    UINT NumExportedFunctions;
    _In_reads_(NumExportedFunctions) const
    D3D12DDI_FUNCTION_SUMMARY_NODE_0054* pSummaries;
    D3D12DDI_EXPORT_SUMMARY_FLAGS OverallFlags;
} D3D12DDI_FUNCTION_SUMMARY_0054;
```

Descriptions of all exported functions in a state object.

Parameter	Definition
`UINT NumExportedFunctions`	How many functions are exported by the current state object. Size of pSummaries array.

Parameter	Definition
`_In_reads_(NumExportedFunctions) const D3D12DDI_FUNCTION_SUMMARY_NODE_0054* pSummaries`	Array of D3D12DDI_FUNCTION_SUMMARY_NODE_0054.
`D3D12DDDI_EXPORT_SUMMARY_FLAGS OverallFlags`	Properties of the state object overall. See D3D12_EXPORT_SUMMARY_FLAGS. This is the aggregate of the flags for each individual export. See Flags in D3D12DDI_FUNCTION_SUMMARY_NODE_0054.

D3D12DDI_FUNCTION_SUMMARY_NODE_0054

```
typedef struct D3D12DDI_FUNCTION_SUMMARY_NODE_0054
{
    LPCWSTR ExportNameUnmangled;
    LPCWSTR ExportNameMangled;
    UINT NumAssociatedSubobjects;
    _In_reads_(NumAssociatedSubobjects) const
        D3D12DDI_STATE_SUBOBJECT_0054*const* ppAssociatedSubobjects;
    D3D12DDI_EXPORT_SUMMARY_FLAGS Flags;
} D3D12DDI_FUNCTION_SUMMARY_NODE_0054;
```

For each shader export (not function export) in a state object being created, this struct tells the driver which subobjects have been associated with it, and some flags indicating status like unresolved bindings remaining (which is valid for collection state objects).

Parameter	Definition
`LPCWSTR ExportNameUnmangled`	Unmangled name of the export.
`LPCWSTR ExportNameMangled`	Mangled name of the export.
`UINT NumAssociatedSubobjects`	Number of subobjects associated with the export. Size of ppAssociatedSubobjects array.
`_In_reads_(NumAssociatedSubobjects) const D3D12DDI_STATE_SUBOBJECT_0054*const* ppAssociatedSubobjects`	Array of pointers to subobjects, D3D12DDI_STATE_SUBOBJECT_0054, that are associated with the current export.
`D3D12DDI_EXPORT_SUMMARY_FLAGS Flags`	Status of the current export. See D3D12_EXPORT_SUMMARY_FLAGS.

D3D12_EXPORT_SUMMARY_FLAGS

```
typedef enum D3D12DDI_SHADER_EXPORT_SUMMARY_FLAGS
{
    D3D12DDI_SHADER_EXPORT_SUMMARY_FLAG_NONE = 0,
    D3D12DDI_SHADER_EXPORT_SUMMARY_FLAG_UNRESOLVED_RESOURCE_BINDINGS = 0x1,
    D3D12DDI_SHADER_EXPORT_SUMMARY_FLAG_UNRESOLVED_FUNCTIONS = 0x2,
    D3D12DDI_EXPORT_SUMMARY_FLAG_UNRESOLVED_ASSOCIATIONS = 0x4,
} D3D12DDI_SHADER_EXPORT_SUMMARY_FLAGS;
```

Flags indicating properties the runtime has determined about a given shader export (including the graph of functions it may call). Unresolved resource bindings or unresolved functions can only appear for collection state objects, since for executable state objects (e.g. RTPSOs), the runtime ensures all dependencies are resolved.

Regardless of flags settings, it remains possible for the driver to find code incompatibility while linking code across DXIL libraries that the runtime missed, since the runtime isn't doing full linking. This is likely to be rare. For instance, a shader in one DXIL library might call a function where a parameter is a user defined type that has been defined locally. The function being called may appear in a different DXIL library with the same function signature but having the user defined type defined differently there. Without doing full linking, the runtime will miss this, in which case the driver would have to fail state object creation.

State object lifetimes as seen by driver

Collection lifetimes

Consider a state object (call it `S`) that references another state object such as an existing]collection (call it `E`).

In the runtime, `S` holds a reference on `E` so that even if the app destroys `E`, the driver won't see `E` destroyed until `S` gets destroyed. This also means the DDI structures describing `E` when it was originally created stay alive as long as `E` is kept alive. So in the driver both `E` and `S` can freely keep references into the data structures describing the contents of `E` without having to copy them.

AddToStateObject parent lifetimes

Consider a state object (call it `C`) that is a child of a parent state object (call it `P`), as a result of incremental addition via AddToStateObject().

In the runtime, `C` holds a reference on `P`'s creation description DDI structures only. So in the driver both `C` and `P` can freely keep references to the data structures describing the contents of `P` without having to copy them.

However, in the runtime `C` does not hold a reference on the DDI object/handle for `P`. So if the app destroys `P`, the driver is told to destroy `P`, even if `C` is still alive. This means if the driver needs to keep any driver internal data related to `P` alive because `C` depends on it, it needs to do the appropriate reference tracking manually. By letting the driver see the destroy of `P` even while its descendent `C` is alive, the driver has a chance to destroy any dependency-free data it might have related to `P`.

> In a typical AddToStateObject() scenario, and app might be making regular additions while discarding the (smaller) parents that are no longer needed. The driver gets to observe that this is happening while still being able to rely on DDI data structures for all the parts of the incrementally grown state object staying alive.

Reporting raytracing support from the driver

D3D12DDI_RAYTRACING_TIER

```
typedef enum D3D12DDI_RAYTRACING_TIER
{
    D3D12DDI_RAYTRACING_TIER_NOT_SUPPORTED = 0,
    D3D12DDI_RAYTRACING_TIER_1_0 = 10,
} D3D12DDI_RAYTRACING_TIER;
```

Level of raytracing support. Currently all or nothing.

This is the RaytracingTier member of `D3D12DDI_D3D12_OPTIONS_DATA_0054`.

Potential future features

This is a non-exhaustive list of potential future features. The list will likely grow and evolve.

Traversal shaders

Bottom-level acceleration structures currently support two types: triangle geometry or procedural geometry (via AABBs).

We could add a third type of bottom-level acceleration structure: traversal. The goal of this node would be to make some procedural decision about what other acceleration structure (if any) to forward the ray that arrived at the node. This could allow a dynamic LOD selection, for example,

by choosing a separate acceleration structure to forward the ray to that contains appropriate LOD geometry.

A traversal node would be defined as AABBs (like procedural geometry) with an index into a shader table where a traversal shader is to be found. When one of these AABBs is hit by a ray the referenced traversal shader is invoked. The traversal shader might be very simple, perhaps only choosing to "forward" the ray into another acceleration structure of its choice (or drop the ray).

The following intrinsic looks a like TraceRay() with fewer parameters:

```
void ForwardRay(RaytracingAccelerationStructure AccelerationStructure,
        uint RayContributionToHitGroupIndex,
        uint MultiplierForGeometryContributionToHitGroupIndex,
        uint MissShaderIndex,
        RayDesc Ray);
```

Ray processing for the forwarded ray behaves just like TraceRay(), with some exceptions. The closest hit shader is only invoked for the lowest T across the original and any forwarded rays. If AcceptHitAndEndSearch() is invoked in the forwarded ray, that ends searching in the parent ray too (followed by closest hit selection as usual).

- There's no payload in the parameter list, as that is forwarded from the original ray.
    - The traversal shader itself may want to be able to inspect the payload (read-only?)
- Other parameters to TraceRay() such as payload, ray flags, instance cull mask get forwarded with the new ray.
- The ray can be defined arbitrarily, likely to be a transformed version of the parent ray
- There would be a user declared recursion limit on ForwardRay recursion, perhaps separate from the user declared TraceRay recursion limit.
    - And/or perhaps ForwardRay specifies a **bottom**-level acceleration structure to use (including defining instance data).

More efficient acceleration structure builds

The current spec requires apps to produce vertex positions in final position into GPU memory before acceleration structure build (unless the available instancing transforms in the acceleration structure definition would happen to do the job).

It could be more efficient if the staging of transformed (e.g. skinned) vertices to memory could be avoided. Perhaps ongoing acceleration structure builds could be directly fed with geometry as it is produced via compute shaders, stream output, UAV writes and/or some new shader stage dedicated to thread-cooperative vertex/primitive processing. Perhaps the system takes geometry directly as shaders process it for the purpose of rasterization and simultaneously encode it into an opaque ongoing acceleration structure build. The system could exploit any data coherency the app has likely already baked into the order it processes geometry, and potentially eliminate wasteful writing/reading of staging geometry to GPU memory.

It is possible that the complexity of more advanced acceleration structures will not be worth it, like if perf bottlenecks tend to be somewhere else.

Beam tracing

The Ray generation shader could have a different mode of operation where the grid location for each thread represents an explicitly defined region of space. The region could be the frustum that represents the volume that an eye would see through each “pixel” in a parameterized definition of the curvature of a viewport. Beams could be intersected with geometry in acceleration structures and produce some kind of rasterization of the intersected geometry, perhaps with some involvement of depth buffering and multisampling. Shader selection could come from shader tables just as in the current raytracing proposal. Since the system would understand the parameterization of the view volume and how it is diced into beams it could do things like make memory layout and rasterization order optimizations (spanning “pixels”) similar to what traditional rasterizers do.

An application of this would be for aligning geometry projection and shading frequency with display optics as opposed to being stuck shoehorning desired projection through plane(s) of pixels as is done with traditional rasterization.

ExecuteIndirect improvements

DispatchRays in command signature

This is now supported - see CreateCommandSignature().

Draw and Dispatch improvements

Going further, perhaps the shader table concept that enables dynamic shader selection in raytracing could be applied back to graphics and compute. So that even for command signatures

that include Draw*()/Dispatch*() there could be both:

1) an option to include a shader identifier for a graphics or compute pipeline state into a command signature

2) an option to set an index into a shader table that picks the pipeline state to use, including local root arguments if desired

```
- as part of this, command signatures should also be improved to
  allow descriptor table setting, so the full set of root
  parameter types can be changed which means adding descriptor
  table support
```

BuildRaytracingAccelerationStructure in command signature

It might even be interesting to be able to use ExecuteIndirect() to issue acceleration structure builds. How this looks might tie into how a design for More efficient acceleration structure builds plays out.

Change log

Version|Date|Change log

- |-|- v0.01|9/27/2017|Initial draft. v0.02|10/20/2017| Changed shader record alignment to 16 bytes Changed argument alignment within shader records: each argument must be aligned to its size Better naming for various parameters that contribute to shader indexing. Ray recursion overflow or callable shader stack overflow now result in a to-be-defined exception mechanism being invoked (likely involving device removal). Initial assumption was that wave intrinsics should be disallowed in raytracing shaders (preservingray scheduling independence). Switched to allowing them with the intent that PIX and careful apps can use them. The HLSL compiler may need to warn about how fragile this can be if misused in raytracing shaders. Loosened the acceleration structure visualization mode for tools (PIX) to allow for tolerances and variation allowed within acceleration structure builds (to be fleshed out further elsewhere later). Changed raytracing acceleration structure alignment from 16 to 256 bytes. Clarified that raytracing shaders that can input ray payloads must declare their layout, even if they aren't looking at it. Fleshed out discussion of acceleration structure determinism Fleshed out raytracing API method documentation (state objects still to be done) Added some detail to ray-triangle intersection specification (still more to be done) Added some details to callable shader description Added shader cycle counter spec to tools section, ported from the spec that is in D3D11 for shader

model 5. What's missing is a way to correlate shader cycle counts to wall clock time. v0.03|11/02/2017| Specified that wave intrinsics have scope of validity bounded by thread repacking points in the shader (as well as start/end). Thread repacking points: CallShader(), TraceRay(), ReportIntersection(). Renames of some of the pieces in the state object API to be more consistent with DXIL library concepts. Added a MissShaderIndex parameter to TraceRay(). So now the selection of miss shaders out of the miss shader table is no longer coupled to how the hit group is selected. Added RAY_FLAG_SKIP_CLOSEST_HIT_SHADER. Also added an example in the ray flags section of the overall walkthrough, showing the usefulness of ray flags in general, including this new flag. Also documented each ray flag. Added an instance culling section in the walkthrough (the feature was already present), highlighting the usefulness. In the callable shaders discussion in the walkthrough, clarified that callable shaders are expected to be implemented by scheduling them for separate execution as opposed to being inlined. Discussed of how callable shaders are really no different than a special case of running a miss shader, minus any geometry interaction. So given an implementation has to support miss shaders, supporting callable shaders is in some ways a subset. And this should scope app expectations about how they behave accordingly. Added a Watertightness requirement to the ray-triangle intersection spec (which is still a work in progress overall). Snippet: It is expected that implementations of watertight ray triangle intersections, including following a form of top-left rule to remove double hits on edges, are possible without having to resort to costly paths such as double precision fallbacks. A proposal is in the works and will appear in this spec. Added D3D12_RESOURCE_STATE_SHADER_TABLE Added bit size limits for the parameters to TraceRay that contribute to shader table indexing. Added section describing potential future features, such as: traversal shaders, more efficient acceleration structure builds, beam tracing, ExecuteIndirect() improvements. This list will grow. Clarified that if the raytracing process encounters a null shader identifier, no shader is executed for that purpose and the raytracing process continues. v0.04|11/07/2017|<li>Called out the largest open issue with this spec as it stands now (mention of which was missing before). In short it has been observed that the current design may not be the best fit for a lot of current and near-future hardware given the way it steers towards ubershaders. While this isn't a unanimous opinion, it must be considered. This note appears both in the Open Issues section and repeated in the Design Goals section at the front – look there for more detail. Stopped allowing GeometryContributionToHitGroupIndex to change in bottom-level acceleration structure updates. Removed RayRecursionLevel() given the overhead it might cause for implementations to be able to report it. If apps need it they can track it manually in ray payload. Stated there isn't a limit on the user declared callable shader stack size. The question still remains of what the failure mode should be if too big a stack causes out of memory. Clarified that acceleration structure serialization/deserialization (intended for PIX) isn't meant for caching acceleration structures. It is likely that doing a full build of an acceleration structure will be faster than loading it from disk. Also stated that deserialize and visualize modes for acceleration structure copies are only allowed when the OS is in Developer Mode. Minor cleanup of

acceleration structure update flag behaviors: If an acceleration structure is built with ALLOW_UPDATE and then PERFORM_UPDATE is done, the result still allows update. ALLOW_UPDATE cannot be specified with PERFORM_UPDATE – further updates are implied to be allowed for the result of an update. For acceleration structure builds, added a D3D12_ELEMENTS_LAYOUT parameter describing how instances/geometries are laid out. There are two options: D3D12_ELEMENTS_LAYOUT_ARRAY and D3D12_ELEMENTS_LAYOUT_ARRAY_OF_POINTERS. This is similar to the flexibility on the array of rendertargets in OMSetRenderTargets in graphics. For wave intrinsics in raytracing shaders added open issue that an explicit AllowWaveRepacking() intrinsic may be required to be paired with intrinsics like TraceRay() if an app wants to use other wave intrinsics in the shader. HLSL syntax cleanup and minor renaming, such as: Renamed PrimitiveID() to PrimitiveIndex() since it is more like InstanceIndex(), being autogenerated as opposed to being user defined, like InstanceID() Started documenting the required resource state for resources in some of the APIs (not finished yet). Also listed which APIs can work on graphics vs compute vs bundle command lists. v0.05|11/15/2017| Clarified that on acceleration structure update, while the transform member of geometry descriptions can change, it can't change from NULL <-> non-NULL. Clarified that for tools visualization of an acceleration structure, the returned geometry may have transforms folded in and may change topology/format of the data (with no loss of precision), so strips might become lists and float16 positions might become float32. DispatchRays() can be called from graphics or compute command lists (had listed graphics only) Consistent with above, raytracing shaders that use a root signature (instead of, or in addition to a local root signature) get root arguments from the compute root arguments on the command list. Clarified in shader table memory initialization section that descriptor handles (identifying descriptor tables) take up 8 bytes. For RayTracingAccelerationStructure clarified that the HLSL declaration includes :register(t#) binding assignment syntax (just like any other SRV binding). Also clarified that acceleration structures can be bound as root descriptors as well as via descriptor heap. Switched the way descriptor based acceleration structure SRVs are declared. Instead of just being an acceleration structure flag on the buffer view type, switched to having a new view type: D3D12_SRV_DIMENSION_RAYTRACING_ACCELERATION_STRUCTURE. The description for this view type contains a GPUVA, which makes more sense than legacy buffer SRV creation fields: FirstElement, NumElements etc. – these don't mean anything for acceleration structures. v0.06|1/29/2018| Changed LPCSTR parameters in state objects (for exports names) to LPWSTR to be consistent with DXIL APIs Minor enum renames Made a few fields D3D12_GPU_VIRTUAL_ADDRESS_RANGE (instead of just _ADDRESS) for APIs that are writing to memory, to help tools and drivers be simpler Added a NO_DUPLICATE_ANYHIT_INVOCATION geometry flag (enabling faster acceleration structures if absent) Removed GeometryContributionToHitGroupIndex field in D3D12_RAYTRACING_GEOMETRY_DESC in favor of it being a fixed function value: the 0 based index of the geometry in the order the app supplied it into the bottom- level acceleration structure. Having this value be user programmable turned was not hugely valuable, and there was IHV pushback against supporting it until a stronger need came up

(or perhaps some better shader table indexing scheme is proposed that can be implemented just as efficiently).
Added struct defining acceleration structure serialization header
Removed topology and strip cut value from geometry desc. Just triangle lists now.
Clarified that during raytracing visualization, geometries may appear in a different order than in the original build, as this order has no bearing on raytracing behavior.
Specified that intersections shaders are allowed to execute redundantly for a given ray and procedural primitive (redundant, but implementation is free to make that choice). Detailed a bunch of caveats related to this.
Defined ray recursion limit to be declared by app in the range [0...31].
GetShaderIdentifierSizeInBytes() returns a nonzero multiple of 8 bytes.
Added RayFlags() intrinsic, returning current ray flags to all shaders that interact with rays.
Changed callable shader stack limit declaration to just a shader stack limit that applies to all raytracing shaders except ray generation shaders (since they are the root).
Minor corrections to shader example code.
Renamed InstanceCullMask to InstanceInclusionMask (and flipped the sense of the instance masking test). Was: InstanceCullMask & InstanceMask -> cull. Changed to !(InstanceInclusionMask & InstanceMask) -> cull. Feedback was the change made the feature more useful typically, despite the slightly awkward implication that if apps set InstanceMask to 0 in an instance definition it will never get included.
For the HitGroupTable parameter to DispatchRays, specified: all shader records in the table must contain a valid shader identifier (NULL is valid). In contrast, local root arguments in the shader records need only be initialized if they could be referenced during raytracing (same requirement for any type of shader table).
Required that GPU descriptor handles are globally unique across all descriptor heaps regardless of type. This makes it much cheaper for tools/debug layer to validate descriptor handles in shader tables, including making sure.
Before deserializing a serialized top-level acceleration structure the app patches the pointers in the header to where bottom-level acceleration structures will be. Those bottom-level acceleration structures pointed to need not have been deserialized yet (spec previously required it), as long as they are deserialized by the app before raytracing uses the acceleration structure.
Removed D3D12_RESOURCE_STATE_SHADER_TABLE in favor of simply reusing D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE.
Required that resources that will contain acceleration structures must be created in the D3D12_RESOURCE_STATE_RAYTRACING_ACCELERATION_STRUCTURE state and must stay in that state forever. Synchronization of read/write operations to acceleration structures is done by UAV barrier. See “Acceleration structure memory restrictions” for more details.
For all raytracing APIs that input or output to GPU memory, defined for each relevant parameter what resource state it needs to be in. In summary: all acceleration structure accesses (read or write) require ACCELERATION_STRUCTURE state as described above. Any other read only GPU input to a raytracing operation must be in NON_PIXEL_SHADER_RESOURCE state, and destination for writes (such as post build info or serialized acceleration structures) must be in UNORDERED_ACCESS.
Added D3D12_ROOT_SIGNATURE_FLAG_LOCAL_ROOT_SIGNATURE so that apps can distinguish whether a root signature is meant for graphics/compute vs being a local root signature. This

distinction between the two types of root signatures is useful for drivers since their implementation of each layout could be different. Also, local root signatures don't have limits on the number of root parameters that root signatures do. Related to adding the local root signature flag above, removed the D3D12_SHADER_VISIBILITY_RAYTRACING option for root parameter visibility. In a root signature (as opposed to local root signature), the root arguments are shared with compute, which just uses D3D12_SHADER_VISIBILITY_ALL. Clarified in the D3D12_RAYTRACING_INSTANCE_FLAG_TRIANGLE_FRONT_COUNTERCLOCKWISE description that winding order applies in object space. So instance transforms with a negative determinant (e.g. mirroring an instance) don't change the winding within the instance. Per-geometry transforms, by contrast, are combined with geometry in object space, so negative determinant transforms do flip winding in that case. In the DDI section, added discussion of a shader export summary that the runtime will generate for drivers, indicating for each shader export in a collection or RTPSO which subobjects have been associated with it. In the case of a collection it also indicates if the export has any unresolved bindings or functions. With all this information, a driver can quickly decide if it should even bother trying to compile the shader yet. There are more details than summarized here. Specified that collections can't be made from existing collections (for simplicity). Only executable state objects (e.g. RTPSOs) can be constructed using existing collections as part of their definition. v0.07|2/1/2018| Renames to go with marketing branding: "DirectX Raytracing": RAY_TRACING -> RAYTRACING, RayTracing -> Raytracing, ray tracing -> raytracing etc. Getting rid of the space is meant to at least reduce the likelihood of people concatenating the name to RT, a confusingly overloaded acronym (runtime, WindowsRT etc.). Renamed various shader intrinsics for clarity: TerminateRay() -> AcceptHitAndEndSearch(), IgnoreIntersection() -> IgnoreHit(), ReportIntersection() -> ReportHit(), RAY_FLAG_TERMINATE_ON_FIRST_HIT -> RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH Added "Subobject association behavior" section discussing details about the subobject association feature in state objects. Lots of detail into default associations, explicit associations, overriding associations, what scopes are affected etc. This might need more refinement or clarification in the future but should be a good start. The destination resource state for acceleration structure serialize is UNORDERED_ACCESS (spec had mistakenly stated NON_PIXEL_SHADER_RESOURCE Defined that the order of serializing top-level and/or bottom-level acceleration structures doesn't matter (and the dependencies don't have to still be present/intact in memory). Same for deserializing. Got rid of "TBD: GeometryIndex()" HLSL intrinsic, as this value is not available to shaders. GeometryContributionToHitGroupIndex is only used in shader table indexing, and if apps want the value in the shader, they need to put a constant in their root signature manually. Removed stale comment indicating there's 4 bytes padding between geometry descs in raytracing visualization output to allow natural alignment when viewed on CPU. That padding became unnecessary as of spec v0.06 when the GeometryContributionToHitGroupIndex was removed from geometry desc struct.

v0.08|2/6/2018| Series of wording cleanups and clarifications throughout spec Clarified that observable duplication of primitives is what is not allowed in acceleration structures. Observable as in something that becomes visible during raytracing operations beyond just performance differences. Listed exceptions where duplication can happen: intersection shader invocation counts, and when the app has not set the NO_DUPLICATE_ANYHIT_INVOCATION flag on a given geometry Changed the behavior of pipeline stack configuration. Moved it out of the raytracing pipeline config subobject (which was a part of the pipeline state) to being mutable on a pipeline state via GetPipelineStackSize()/SetPipelineStackSize(). Added detailed discussion on the default stack size calculation (which is conservative and likely wasteful) as well as how apps can optionally choose to set it with a more optimal value based on knowing their specific calling pattern. Clarified that shader identifiers globally identify shaders, so that even if it appears in a different state object (with the same associations like any root signatures) it will have the same identifier. In addition to being in acceleration_structure state, resources that will hold acceleration structures must also have resource flag allow_unordered_access. This merely acknowledges that it is UAV barriers that are used for read/write synchronization. And that behind the scenes, implementations will be doing UAV style accesses to build acceleration structures (despite the API resource state being always "acceleration structure" v0.09|3/12/2018| For a (tools) acceleration structure visualization, geometries must appear in the same order as in build (given that hit group table indexing is influenced by the order of each geometry in the acceleration structure, so the order needs to be preserved for visualization). Vertex format DXGI_FORMAT_R16G16B16_FLOAT doesn't actually exist. Specified that DXGI_FORMAT_R16G16B16A16_FLOAT can be specified, and the A16 component gets ignored. Other data can be packed over the A16 part, such as setting the vertex stride to 6 bytes for instance. For GetShaderStackSize(), required that to get the stack size for a shader within a hit group, the following syntax is used for the entrypoint string name: hitGroupName::shaderType, where hitGroupName is the name of the hit group export and shaderType is one of: intersection, closesthit, anyhit (all case sensitive). E.g. "myLeafHitGroup::anyhit". Stack size is specific to individual shaders, but the overall hit group a shader belongs to can influence the stack size as well. Guaranteed that shader identifier size (returned by GetShaderIdentifierSize() can be no larger than 64 bytes. The Raytracing Emulation section has been updated with more concrete details on the Fallback Layer. It will be a stand-alone library that will be open-sourced publically that will closely mirror the DXR API. Developers will be free to pull the repro and tailor it for their own needs, or even send pull-requests with contributions. Removed row_major modifier on matrix intrinsics definition (irrelevant). Removed system value semantics, added language to clarify parameter requirements. Added state object flags: D3D12_STATE_OBJECT_FLAG_ALLOW_EXTERNAL_DEPENDENCIES_ON_LOCAL_DEFINITIONS and D3D12_STATE_OBJECT_FLAG_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITIONS. Both of these apply to collection state objects only. In the absence of the first flag (default),

drivers are not forced to keep uncompiled code around just because of the possibility that some other entity in a containing collection might call a function in the collection. In the absence of the second flag (default), contents of collections must be fully resolved locally (including necessary subobject associations) such that advanced implementations can fully compile code in the collection (not having to wait until RTPSO creation). These flags will be added to the OS post-March 2018/GDC, and in the meantime the default behavior (absence of the flags) is the only available current behavior. Added discussion: Subobject associations for hit groups. Explains that subobject associations can be made to the component shaders in a hit group and/or to the hit group directly. Basically it is invalid if there are any conflicts – see the text for full detail. v0.91|7/27/2018| Cleared out references to experimental/prototype API and open issues Local root signatures don't have the size constraint that global root signatures do (since local root signatures just live in shader tables in app memory). All local root signatures referenced in an RTPSO that define static samplers must have matching definitions for any given s# that they define, and the total number of s# defined across global and local root signatures must fit within bind model limit. Shader table start address must be 64 byte aligned Shader record stride must be a multiple of 32 bytes Noted that subobjects defined in DXIL did not make this first release. Added table (Subobject association requirements) describing rules for each subobject type. Drivers are responsible for implementing caching of state objects using existing services in D3D12. Added discussions of "Inactive primitives and instances" as well as "Degenerate primitives and instances" Fleshed out "Fixed function ray-triangle intersection specification" with a hypothetical top-left rule description. Implementations must do something like it to ensure no double hits or holes along shared edges. Exceeding ray recursion limit or overflowing pipeline stack result in device removed. Refined "Default pipeline stack size" calculation Added "Determining raytracing support" Cut "Shader Cycle Counter" feature, could bring it back in a future release. Series of minor tweaks to API/DDI definitions to get to final release state, including fleshing out documentation for individual APIs/structs/enums. Fleshed out descriptions of D3D12_STATE_OBJECT_FLAG_ALLOW_LOCAL_DEPENDENCIES_ON_EXTERNAL_DEFINITIONS and
D3D12_STATE_OBJECT_FLAG_ALLOW_EXTERNAL_DEPENDENCIES_ON_LOCAL_DEFINITIONS. The absence of these flags in a state object configuration (which is the default) means drivers can compile code in collections without waiting until RTPSO create (aside from final linking on a clean implementation), and conversely don't have to keep extra stuff around in each collection on the chance that other parts of an RTPSO will depend on whats inside a given collection. Apps can opt in to cross-linkage basically.
D3D12_NODE_MASK state object definition was missing.
Added D3D12_HIT_GROUP_TYPE member to hit groups.
Removed GetShaderIdentifierSizeInBytes(). Shader identifers are now a fixed 32 bytes.
Refactored GetRaytracingAccelerationStructurePrebuildInfo() and BuildRaytracingAccelerationStructure() APIs to share the same input struct as a convenience. The latter API just has additional parameters to specify (such as actual buffer pointers).
Added the functionality of GetAccelerationStructurePostbuildInfo() as optional output

parameters to BuildRaytracingAccelerationStructure(). So an app can request post build info directly as an additional output of a build (without extra barrier calls between doing the build and getting postbuild info), or it can still separately get postbuild info for an already built acceleration structure.
BUILD_FLAG_ALLOW_UPDATE can only be specified on an original build or an update where the source acceleration structure also specified ALLOW_UPDATE. Similarly, BUILD_FLAG_ALLOW_COMPACTION can only be specified on an initial acceleration structure build or on an update where the input acceleration structure also specified ALLOW_COMPACTION.
Renamed “D3D12_GPU_VIRTUAL_ADDRESS Transform” parameter to D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC to Transform3x4 to be more clear about the matrix layout. Similarly renamed “FLOAT Transform[12]” member of D3D12_RAYTRACING_INSTANCE_DESC to “FLOAT Transform[3][4]”
Added support for 16 bit SNORM vertex formats for acceleration structure build input.
Added buffer alignment (and or stride) requirements to various places in API where buffer locations are provided.
Added option to query the current size of an already built acceleration structure “POSTBUILD_INFO_CURRENT_SIZE”. Useful for tools inspecting arbitrary acceleration structures.
Added D3D12_SERIALIZED_DATA_DRIVERS_MATCHING_IDENTIFIER to serialized acceleration structure headers. This is opaque versioning information the driver generates that allow an app to serialize acceleration structures to disk and later restore. On restore, the identifier can be compared with what the current device reports via new CheckDriverMatchingIdentifier() API to see if a serialized acceleration structure is compatible with the current device (and can be deserialized).
Added SetPipelineState1() command list API for setting RTPSOs (factored out of the parameters to DispatchRays(). Calling SetPipelineState1() unsets any pipeline state set by SetPipelineState() (graphics or compute). And vice versa.
Made the grid dimensions in DispatchRays 3D to be consistent with compute Dispatch(). Similarly in HLSL, DispatchRaysIndex() returns uint3 and DispatchRaysDimensions() returns uint3.
Clarified when it is valid to call GetShaderIdentifier() to get the identifier for a given export from a state object.
Replaced float3x4 ObjectToWorld() and WorldToObject() intrinsics with ObjectToWorld3x4(), ObjectToWorld4x3(), WorldToObject3x4() and WorldToObject4x3(). The 3x4 and 4x3 variants just let apps pick whichever transpose of the matrix they want to use.
Documented the various DDIs that are not just trivial passthroughs of APIs. Most interesting of which is D3D12DDI_FUNCTION_SUMMARY_0054, a DDI only subobject in state object creation that the runtime generates, listing for each export what subobject got associated to it among other things. This way drivers don’t need to understand the various rules for how subobject associations (with defaults) work.
v1.0|10/1/2018| Clarified that ray direction does not get normalized. Clarified that tracing rays with a NULL acceleration structure causes a miss shader to run. Set maximum shader record stride to 4096 bytes. Reduced maximum local root signature size from unlimited to max-stride (4096) – shader identifier size (32) = 4064 bytes max. Corrected stale reference to DispatchRays() taking a state object as input -> corrected to SetPipelineState1 as the API where state objects are set. The grid dimensions passed to DispatchRays() must satisfy width*height*depth <= 2^30. Exceeding this produces undefined behavior. Removed stale mention that collections can be

nested (they cannot at least for now).
Added geometry limits section: Max geos in a BLAS: 2^24, Max prims in BLAS across geos: 2^29, max instance count in TLAS 2^24.
Added Ray extents section: TMin must be nonnegative and <= TMax. +Inf is ok for either (really only makes sense for TMax). Ray-trinagle intersection can only occur if the intersection T-value satisfies TMin < T < TMax. No part of ray origin/direction/T range can be NaN.
Added General tips for building acceleration structures section
Added Choosing acceleration structure build flags section
Added preamble to Raytracing emulation section pointing out that the fallback is no longer being maintained given the cost/benefit is not justified etc.
Various minor typos fixed
Fleshed out details of ALLOW_COMPACTION, including that specifying the flag may increase acceleration structure size.
Also for postbuild info, clarified some guarantees about CompactedSizeInBytes, such as compacted acceleration structures don't use more space than non compacted ones. And there isn't a guarantee that an acceleration structure with fewer items than another one will compact to a smaller size if both are compacted (the property does hold if they are not compacted).
Per customer request, clarified for D3D12_RAYTRACING_INSTANCE_DESC that implementations transform rays as opposed to transforming all geometry/AABBs.
Only defined ray flags are propagated by the system, e.g. visible to the RayFlags() shader intrinsic.
Clarified that the scope of shared edges where watertight ray triangle intersection applies only within a given bottom level acceleration structure for geometries that share the same transform.
v1.01|11/27/2018| Clarified that determination of inactive/degenerate primitives includes all geometry transforms. For instance, per-geometry bottom level transforms are one way that NaNs can be injected to cheaply cause groups of geometry to be deactivated in builds. Some clarifications in "Conflicting subobject associations" section regarding DXIL defined subobjects (a feature for a future release). Clarified that mixing of geometry types within a bottom-level acceleration structure is not supported.
v1.02|3/4/2019|Added a comment on ways of calculating a hit position and their tradeoffs to Closest Hit Shader section. v1.03|3/25/2019| Ported DXR spec to Markdown. Removed wording that DXIL subobjects aren't in initial release, as they have now been implemented as of 19H1. v1.04|4/18/2019| Support for indirect DispatchRays() calls (ray generation shader invocation) via ExecuteIndirect(). Support for incremental additions to existing state objects via AddToStateObject(). Support for inline raytracing via TraceRayInline(). Added D3D12_RAYTRACING_TIER_2, which requires the above features. v1.05|5/15/2019|<li>Renamed `InlineRayQuery` to simply `RayQuery` in support of adding this object as a return value for TraceRay() (in addition to TraceRayInline()). For `TraceRay()`, the returned `RayQuery` object represents the final state of ray traversal after any shader invocations have completed. The caller can retrieve information about the final state of traversal (CLOSEST_HIT or MISS) and relevant metadata such as hit attributes, object ids etc. that the system already knows about without having to manually store this information in the ray payload. Added stipulations on the state object definition passed into the AddToStateObject() API based around the incremental addition being a valid self contained definition of a state object, even though it is being combined with an existing one. The list of stipulations are intended to simplify driver

compiler burden, runtime validation burden, as well as simplify code linkage/dependency semantics. Added `D3D12_STATE_OBJECT_FLAG_ALLOW_STATE_OBJECT_ADDITIONS` flag so that *executable* state objects (e.g. raytracing pipelines) must opt-in to being used with AddToStateObject(). The flags discussion includes a description of what it means for *collection* state objects to set this flag. Renamed D3D12_RAYTRACING_TIER_2_0 to D3D12_RAYTRACING_TIER_1_1 to slightly better convey the relative significance of the new features to the set of functionality in TIER_1_0. Removed RAY_QUERY_STATE_UNINITIALIZED from RAY_QUERY_STATE in favor of simply defining that accessing an uninitialized `RayQuery` object produces undefined behavior, and the HLSL compiler will attemp to disalllow it. v1.06|5/16/2019|<li>Removed a feature proposed in previous spec update: ability for TraceRay() to return a RayQuery object. For some implementations this would actually be slower than the application manually stuffing only needed values into the ray payload, and would require extra compilation for paths that need the ray query versus those that do not. This feature could come back in a more refined form in the future. This could be allowing the user to declare entries in the ray payload as system generated values for all query data (e.g. SV_CurrentT). Some implementations could choose to compile shaders to automatically store these values in the ray payload (as declared), whereas other implementations would be free to retrieve these system values from elsewhere if they are readily available (avoiding payload bloat). v1.07|6/5/2019| For Tier 1.1, GeometryIndex() intrinsic added to relevant raytracing shaders, for applications that wish to distinguish geometries manually in shaders in adddition to or instead of by burning shader table slots. Revised AddToStateObject() API such that it returns a new ID3D12StateObject interface. This makes it explicitly in the apps control when to release the original state object (if ever). Clarified lots of others detail about the semantics of the operation, particularly now that an app can produce potentially a family of related state objects (though a straight line of inheritance may be the most likely in practice). Refactored inline raytracing state machine. Gave up up original proposal's conceptual simplicity (matching the shader based model closely) in favor of a model that's functionally equivalent but makes it easier for drivers to generate inline code. The shader has a slightly higher burden having to do a few more things manually (outside the system's tracking). The biggest win should be that there is one logical place where the bulk of system implemented traversal work needs to happen: RayQuery::Proceed(), a method which the shader will typically use in a loop condition until traversal is done. Some helpful links: Inline raytracing overview TraceRayInline control flow RayQuery object RayQuery intrinsics TraceRayInline examples v1.08|7/17/2019| Removed distinction between inline ray flags and ray flags. Both dynamic ray flags for RayQuery::TraceRayInline() and TraceRay() as well as the template parameter for RayQuery objects share the same set of flags. Added ray flags: `RAY_FLAG_SKIP_TRIANGLES` and `RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES` to allow completely skipping either class of primitives. These flags are valid everywhere ray flags can be used, as summarized above. Allowed RayQuery::CommitProceduralPrimitiveHit() and RayQuery::CommitNonOpaqueTriangleHit() to be called zero or more times per candidate (spec was 0 or 1 times). If `CommitProceduralPrimitiveHit(t)` is called multiple times per

candidate, it requires the shader to have ensure that each time the new t being committed is closest so far in ray extents (would be the new TMax). If `CommitNonOpaqueTriangleHit()` is called multiple times per candidate, subsequent calls have no effect, as the hit has already been committed. RayQuery::CandidatePrimitiveIndex() was incorrectly listed as not supported for HIT_CANDIDATE_PROCEDURAL_PRIMITIVE in the RayQuery intrinsics tables. New version of raytracing pipeline config subobject, D3D12_RAYTRACING_PIPELINE_CONFIG1, adding a flags field, D3D12_RAYTRACING_PIPELINE_FLAGS. The HLSL equivalent subobject is RaytracingPipelineConfig1. The available flags, `D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_TRIANGLES` and `D3D12_RAYTRACING_PIPELINE_FLAG_SKIP_PROCEDURAL_PRIMITIVES` (minus `D3D12_` when defined in HLSL) behave like OR'ing the above equivalent RAY_FLAGS listed above into any TraceRay() call in a raytracing pipeline. Implementations may be able to make some pipeline optimizations knowing that one of the primitive types can be skipped. Cut RayQuery::Clone() instrinsic, as it doesn't appear to have any useful scenario. Ray extents section has been updated to define a different range test for triangles versus procedural primitives. For triangles, intersections occur in the (TMin…TMax) (exclusive) range, which is no change in behavior. For procedural primitives, the spec wasn't explicit on what the behavior should be, and now the chosen behavior is that intersections occur in [TMin…TMax] (inclusive) range. This allows apps the choice about how to handle exactly overlapping hits if they want a particular one to be reported. v1.09|11/4/2019| Added Execution and memory ordering discussion defining that TraceRay() and/or CallShader() invocations finish execution when the caller resumes, and that for UAV writes to be visible memory barriers must be used.. Added State object lifetimes as seen by driver section to clarify how driver sees the lifetimes of collection state objects as well as state objects in a chain of AddToStateObject() calls growing a state object while the app is also destroying older versions. In BuildRaytracingAccelerationStructure() API, tightened the spec for the optional output postbuild info that the caller can request. The caller can pass an array of postbuild info descriptions that they want the driver to fill out. The change is to require that any given postbuild info type can only be requested at most once in the array. v1.1|11/7/2019|<li>Added accessor functions RayQuery::CandidateInstanceContributionToHitGroupIndex and RayQuery::CommittedInstanceContributionToHitGroupIndex so that inline raytracing can see the `InstanceContributionToHitGroupIndex` field in instance data. This is done in the spirit of letting inline raytracing see everything that is in an acceleration structure. Of course in inline raytracing there are no shader tables, so there is no functional behavior from this value. Instead it allows the app to use this value as arbitrary user data. Or when sharing acceleration structures between dynamic-shader-based raytracing, to manually implementing in the inline raytracing case the effective result of the shader table indexing in the dynamic-shader-based raytracing case. In Wave Intrinsics section, clarified that use of RayQuery objects for inline raytracing does not count as a repacking point. v1.11|3/3/2020| For Tier 1.1, additional vertex formats supported for acceleration structure build input as part of D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC.

Cleaned up spec wording - D3D12_RAYTRACING_PIPELINE_CONFIG and D3D12_RAYTRACING_SHADER_CONFIG have the same rules: If multiple shader configurations are present (such as one in each collection to enable independent driver compilation for each one) they must all match when combined into a raytracing pipeline. For collections, spelled out the list of conditions that must be met for a shader to be able to do compilation (as opposed to having to defer it). Clarified that ray recursion limit doesn't include callable shaders. Clarified that in D3D12_RAYTRACING_SHADER_CONFIG, the MaxPayloadSizeInBytes field doesn't include callable shader payloads. It's only for ray payloads. v1.12|4/6/2020| For `D3D12_RAYTRACING_ACCELERATION_STRUCTURE_COPY_MODE_VISUALIZATION_DECODE_FOR_TOOLS`, clarified that the result is self contained in the destination buffer. v1.13|7/6/2020| For D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC, clarified that if an index buffer is present, this must be at least the maximum index value in the index buffer + 1. Clarified that a null hit group counts as opaque geometry. v1.14|1/12/2021| Clarified that RayFlags() does not reveal any flags that may have been added externally via D3D12_RAYTRACING_PIPELINE_CONFIG1. v1.15|3/26/2021|<li>Added payload access qualifiers section, introducing a way to annotate members of ray payloads to indicate which shader stages read and/or write individual members of the payload. This lets implementations optimize data flow in ray traversal. It is opt-in for apps starting with shader model 6.6, and if present in shaders appears as metadata that is ignored by existing drivers. For shader model 6.7 and higher these payload access qualifiers are required to be used by default (opt-out). v1.16|7/29/2021| For any hit shaders, clarified that for a given ray, the system can't execute multiple any hit shaders at the same time. As such shaders can modify the ray payload freely without worrying about conflicting with other shader invocations. v1.17|10/25/2021|<li>In Degenerate primitives and instances, added the clarification: An exception to the rule that degenerates cannot be discarded with `ALLOW_UPDATE` specified is primitives that have repeated index value can always be discarded (even with `ALLOW_UPDATE` specified). There is no value in keeping them since index values cannot be changed.
