**D3D12 Raytracing Functional Spec**

v0.09 3/12/2018

# Contents

[Contents 1](#_Toc508639801)

[1 Intro 5](#_Toc508639802)

[2 Overview 5](#_Toc508639803)

[3 Design goals 6](#_Toc508639804)

[3.1 Largest open issue 6](#_Toc508639805)

[4 Walkthrough 7](#_Toc508639806)

[4.1 Initiating raytracing 7](#_Toc508639807)

[4.2 Ray generation shaders 7](#_Toc508639808)

[4.3 Rays 7](#_Toc508639809)

[4.4 Raytracing output 7](#_Toc508639810)

[4.5 Ray-geometry interaction diagram 8](#_Toc508639811)

[4.6 Geometry and acceleration structures 8](#_Toc508639812)

[4.7 Acceleration structure updates 9](#_Toc508639813)

[4.8 Built-in ray-triangle intersection (triangle mesh geometry) 9](#_Toc508639814)

[4.9 Intersection shaders (procedural primitive geometry) 10](#_Toc508639815)

[4.9.1 Minor intersection shader details 10](#_Toc508639816)

[4.10 Any hit shaders 10](#_Toc508639817)

[4.11 Closest hit shaders 10](#_Toc508639818)

[4.12 Miss shaders 11](#_Toc508639819)

[4.13 Hit groups 11](#_Toc508639820)

[4.14 TraceRay() control flow 12](#_Toc508639821)

[4.15 Ray flags 13](#_Toc508639822)

[4.16 Instance masking 13](#_Toc508639823)

[4.17 Callable shaders 14](#_Toc508639824)

[4.18 Resource binding 14](#_Toc508639825)

[4.18.1 Local root signatures vs root signatures 14](#_Toc508639826)

[4.19 Shader identifier 15](#_Toc508639827)

[4.20 Shader record 15](#_Toc508639828)

[4.21 Shader tables 15](#_Toc508639829)

[4.22 Indexing into shader tables 16](#_Toc508639830)

[4.22.1 Shader record stride 16](#_Toc508639831)

[4.22.2 Shader table memory initialization 16](#_Toc508639832)

[5 Shader management 16](#_Toc508639833)

[5.1 Problem space 16](#_Toc508639834)

[5.1.1 Implementations juggle many shaders 16](#_Toc508639835)

[5.1.2 Applications control shader compilation 17](#_Toc508639836)

[5.2 State objects 17](#_Toc508639837)

[5.2.1 Subobjects 17](#_Toc508639838)

[5.2.2 State object types 18](#_Toc508639839)

[5.2.3 DXIL libraries and state objects example 20](#_Toc508639840)

[5.2.4 Subobject association behavior 21](#_Toc508639841)

[5.2.5 State object caching 23](#_Toc508639842)

[6 System limits and fixed function behaviors 24](#_Toc508639843)

[6.1 Addressing calculations within shader tables 24](#_Toc508639844)

[6.1.1 Hit group table indexing 24](#_Toc508639845)

[6.1.2 Miss shader table indexing 24](#_Toc508639846)

[6.1.3 Callable shader table indexing 24](#_Toc508639847)

[6.1.4 Out of bounds shader table indexing 24](#_Toc508639848)

[6.2 Acceleration structure properties 24](#_Toc508639849)

[6.2.1 Data rules 24](#_Toc508639850)

[6.2.2 Determinism based on fixed acceleration structure build input 25](#_Toc508639851)

[6.2.3 Determinism based varying acceleration structure build input 25](#_Toc508639852)

[6.2.4 Preservation of triangle set 26](#_Toc508639853)

[6.2.5 AABB volume 26](#_Toc508639854)

[6.2.6 Fixed function ray-triangle intersection specification 26](#_Toc508639855)

[6.2.7 Acceleration structure update constraints 27](#_Toc508639856)

[6.2.8 Acceleration structure memory restrictions 28](#_Toc508639857)

[6.3 Ray recursion limit 29](#_Toc508639858)

[6.4 Pipeline stack 29](#_Toc508639859)

[6.4.1 Optimal pipeline stack size calculation 30](#_Toc508639860)

[6.4.2 Default pipeline stack size 31](#_Toc508639861)

[6.4.3 Pipeline stack limit behavior 32](#_Toc508639862)

[6.5 Shader limitations resulting from independence 32](#_Toc508639863)

[6.5.1 Wave Intrinsics 32](#_Toc508639864)

[7 Raytracing emulation 33](#_Toc508639865)

[8 Tools support 34](#_Toc508639866)

[8.1 Buffer bounds tracking 34](#_Toc508639867)

[8.2 Acceleration structure processing 34](#_Toc508639868)

[8.3 Shader Cycle Counter 35](#_Toc508639869)

[8.3.1 Basic Semantics 35](#_Toc508639870)

[8.3.2 Interpreting Cycle Counts 35](#_Toc508639871)

[8.3.3 Shader Compiler Constraints 36](#_Toc508639872)

[8.3.4 DXIL Details 36](#_Toc508639873)

[9 API 37](#_Toc508639874)

[9.1 Experimental feature exposure 37](#_Toc508639876)

[9.2 Device methods 37](#_Toc508639877)

[9.2.1 CreateStateObject() 37](#_Toc508639878)

[9.2.2 GetShaderIdentifierSize() 41](#_Toc508639879)

[9.2.3 GetRaytracingAccelerationStructurePrebuildInfo() 41](#_Toc508639880)

[9.3 Command list methods 44](#_Toc508639881)

[9.3.1 BuildRaytracingAccelerationStructure() 44](#_Toc508639882)

[9.3.2 EmitRaytracingAccelerationStructurePostBuildInfo() 54](#_Toc508639883)

[9.3.3 CopyRaytracingAccelerationStructure() 56](#_Toc508639884)

[9.3.4 DispatchRays() 59](#_Toc508639885)

[9.4 StateObjectConfig methods 61](#_Toc508639886)

[9.4.1 GetShaderIdentifier() 61](#_Toc508639887)

[9.4.2 GetShaderStackSize() 62](#_Toc508639888)

[9.4.3 GetPipelineStackSize() 62](#_Toc508639889)

[9.4.4 SetPipelineStackSize() 62](#_Toc508639890)

[9.5 Additional resource states 63](#_Toc508639891)

[9.6 Additional root signature flags 63](#_Toc508639892)

[9.6.1 D3D12\_ROOT\_SIGNATURE\_FLAG\_LOCAL\_ROOT\_SIGNATURE 63](#_Toc508639893)

[9.6.2 Note on shader visibility 63](#_Toc508639894)

[9.7 Additional SRV type 64](#_Toc508639895)

[9.8 Constants 65](#_Toc508639896)

[10 HLSL 66](#_Toc508639897)

[10.1 Types, enums, subobjects and concepts 66](#_Toc508639898)

[10.1.1 Ray flags 66](#_Toc508639899)

[10.1.2 Ray description structure 67](#_Toc508639900)

[10.1.3 RaytracingAccelerationStructure 68](#_Toc508639901)

[10.1.4 Subobject definitions 68](#_Toc508639902)

[10.1.5 Intersection attributes structure 69](#_Toc508639903)

[10.1.6 Ray payload structure 69](#_Toc508639904)

[10.1.7 Call parameter structure 69](#_Toc508639905)

[10.2 Shaders 69](#_Toc508639906)

[10.2.1 Ray generation shader 69](#_Toc508639907)

[10.2.2 Intersection shader 70](#_Toc508639908)

[10.2.3 Any hit shader 71](#_Toc508639909)

[10.2.4 Closest hit shader 71](#_Toc508639910)

[10.2.5 Miss shader 72](#_Toc508639911)

[10.2.6 Callable shader 73](#_Toc508639912)

[10.3 Intrinsics 73](#_Toc508639913)

[10.3.1 CallShader() 73](#_Toc508639914)

[10.3.2 TraceRay() 73](#_Toc508639915)

[10.3.3 ReportHit() 74](#_Toc508639916)

[10.3.4 IgnoreHit() 75](#_Toc508639917)

[10.3.5 AcceptHitAndEndSearch() 75](#_Toc508639918)

[10.4 System value intrinsics 76](#_Toc508639919)

[10.4.1 Ray dispatch system values 76](#_Toc508639920)

[10.4.2 Ray system values 76](#_Toc508639921)

[10.4.3 Primitive/object space system values 77](#_Toc508639922)

[10.4.4 Hit specific system values 78](#_Toc508639923)

# 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
		- Including fallback code for developers that implements raytracing on the stock D3D12 API
	+ 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

## Largest open issue

There are many smaller open issues highlighted throughout this document. The largest open/unresolved issue with the current design is worth calling out up front though:

It has been observed that the current definition of the API might not be the best fit for a lot of current or even near-term future hardware that does not have flexibility to schedule dynamic ray workloads.  The API steers implementations towards uber-shader formulations to handle all raytracing shaders.  IHVs have the burden to find clever solutions for their individual implementations to avoid or minimize well known impact of uber-shaders on GPU performance.  The raytracing fallback implementation has the most difficult task, given it uses the stock D3D12 API.  Alternative API designs, perhaps exposing scheduling steps more explicitly, might be a better fix to the problem rather than relying on hardware to fix it.

This is not a unanimous opinion, though it must be considered.

# 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, fleshing out over time as the feature solidifies.

## Initiating raytracing

Just as rasterization is invoked by Draw() and compute is invoked via Dispatch(), raytracing is invoked via [DispatchRays](#_DispatchRays())(). DispatchRays() can be called from graphics command lists, compute command lists or bundles.

## 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.](#_DispatchRays())

HLSL details are [here](#_Ray_Generation_Shader).

## 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).

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()](#_TraceRay) upon its return .



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 though 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 either: (1) triangle meshes, and (2) procedural primitives initially described only by an axis aligned bounding box (AABB). These options are described more later. Given a definition of a set of these geometries (via array of [D3D12\_RAYTRACING\_GEOMETRY\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_DESC_1)), the application calls [BuildRaytracingAccelerationStructure()](#_BuildRayTracingAccelerationStructur) 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](#_D3D12_RAY_TRACING_INSTANCE_DESC_1) 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()](#_BuildRayTracingAccelerationStructur)) 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](#_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](#_Acceleration_structure) 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](#_D3D12_RAY_TRACING_ACCELERATION_STRU) in [BuildRaytracingAccelerationStructure()](#_BuildRayTracingAccelerationStructur).

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](#_Acceleration_structure_update).

## 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](#_Intersection_shader_1).

### 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 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](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) on the geometry, which may include D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_NO\_DUPLICATE\_ANYHIT\_INVOCATION. With this flag, the [Any hit shader](#_Any_hit_shaders) (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 ray’s current [TMin, TMax] interval, 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 an hit should be ignored ([IgnoreHit()](#_IgnoreIntersection())), accept the hit and continue (by exiting execution) or accept the hit and tell the system to stop searching for more intersections ([AcceptHitAndEndSearch()](#_TerminateRay)).

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, it’s 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.

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 the 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](#_Any_Hit_Shader_1).

## Closest hit shaders

A unique shader can be defined to run for each geometry in an instance if it produced the closest accepted intersection in a ray’s [TMin, TMax] interval. 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](#_Closest_Hit_Shader_1).

## 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](#_Miss_Shader).

## 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](#_Shader_identifier) (concept described later).

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](#_Miss_Shader) running when nothing has been hit.

## TraceRay() control flow

This is what happens when a shader calls [TraceRay](#_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 ray's current [TMin…TMax] interval, it is guaranteed to be enumerated eventually. If a primitive is not intersected by the ray or is outside the current [TMin…TMax] interval, 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()](#_ReportIntersection), 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](#_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](#_Ray_Flags) is set, or if the any hit shader called [AcceptHitAndEndSearch()](#_TerminateRay), which aborts the execution of the any hit shader at the AcceptHitAndEndSearch[AcceptHitAndEndSearch()](#_TerminateRay) 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()](#_ReportIntersection)ReportHit.

## Ray flags

[TraceRay()](#_TraceRay) supports a selection of [Ray flags](#_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()](#_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](#_Ray_Generation_Shader), followed by exiting the shader with 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](#_Closest_Hit_Shader_1) 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](#_Miss_shaders) 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](#_Shader_identifier)). 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](#_D3D12_RAY_TRACING_GEOMETRY_TRIANGLE) in top level acceleration structures each contain an 8-bit user defined InstanceMask. [TraceRay()](#_TraceRay) has an 8-bit input parameter InstanceInclusionMask which gets ANDed with the InstanceMask from any geometry 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

This feature is squarely in the “food for thought” category, more so than anything else in the spec. Could it work? Or something like it?

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](#_Shader_tables), 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](#_D3D12_DISPATCH_RAYS_DESC_1)) to [DispatchRays()](#_DispatchRays()_1) calls. The contents of the table contain shader identifiers retrieved from [state objects](#_State_objects) (described later) via [GetShaderIdentifier()](#_GetShaderIdentifier()).

A given callable shader is called (via [CallShader()](#_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](#_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 that do not hit anything and just cause 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 processing a ray that is guaranteed to miss for no reason, unless it were to be defined that tracing against a NULL acceleration structure is guaranteed to miss. 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 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 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 root signature.

Different sets of shaders collected together in a [State object](#_State_objects) (described later), may have different root signatures, as long as during a [DispatchRays()](#_DispatchRays()_1) call (or equivalent indirect API if it ever exists) any shaders that get invoked use the same root signature that is set on the CommandList as described above.

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

There is a discussion on shader visibility flags in root signatures [here](#_Note_on_shader).

## Shader identifier

A shader identifier is an opaque data blob of relatively small size reported by the implementation 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](#_Hit_groups), 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](#_State_object_types)). 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.

## 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.

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](#_DispatchRays())() include pointers to memory that let apps identify (among other things) the following types of shader tables:

* [ray generation shader](#_Ray_generation_shaders) (single entry since only one shader record is needed
* [hit groups](#_Hit_groups)
* [miss shaders](#_Miss_shaders)
* [callable shaders](#_Callable_shaders_1)

## 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](#_Addressing_calculations_within_1), but basically the process starts at [DispatchRays()](#_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()](#_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()](#_DispatchRays()). All shader table indexing arithmetic is done as multiples of this record stride. It can be any multiple of 16 bytes including 0.

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 normal 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.

# Shader management

## Problem space

### Implementations juggle many shaders

In a given [DispatchRays](#_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()](#_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()](#_CreateStateObject()) on a D3D12 device.

### Subobjects

State objects are built out of subobjects. A subobject has a [type](#_D3D12_STATE_SUBOBJECT_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](#_Subobject_association_behavior) for detailed discussion, and see the relevant parts of the state object API here: [D3D12\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION](#_D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIAT) and [D3D12\_DXIL\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION](#_D3D12_DXIL_SUBOBJECT_TO_EXPORTS_ASS).

The full set of subobject types is defined in [D3D12\_STATE\_SUBOBJECT](#_D3D12_STATE_SUBOBJECT).

#### 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](#_Subobject_definitions).

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). This doesn’t apply to all subobject types, since some are only relevant in 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](#_D3D12_STATE_OBJECT_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](#_D3D12_STATE_OBJECT_TYPE_1) 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](#_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.

One of the inputs to [DispatchRays](#_DispatchRays())() is what state object to use, and this must be an RTPSO.

#### 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](#_D3D12_STATE_OBJECT_TYPE_1) 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 may not be able to actually compile anything and would be left with simply storing the subobjects. The minimum amount of configuration state that has to be in a collection for a driver to be able to do some shader compilation work is documented (later on).

One example is that if a collection has shaders that reference bindings but no root signature or local root signature present and associated with each shader, the driver can’t compile the shader. There are other requirements for a driver to be able to do useful compile work behind a collection in the case of raytracing as well, described later.

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.

#### 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, and/or other existing collections, 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](#_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.

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](#_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.

### State object caching

Details TBD, but all of the driver shader caching techniques exposed by D3D12 can be extended to support state objects.

# 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](#_D3D12_DISPATCH_RAYS_DESC_1).HitGroupTable.StartAddress + // from: [DispatchRays()](#_DispatchRays())

 [D3D12\_DISPATCH\_RAYS\_DESC](#_D3D12_DISPATCH_RAYS_DESC_1).HitGroupTable.StrideInBytes \* // from: [DispatchRays()](#_DispatchRays())

 (

 RayContributionToHitGroupIndex + // from shader: [TraceRay()](#_TraceRay)

 (MultiplierForGeometryContributionToHitGroupIndex \* // from shader: [TraceRay()](#_TraceRay)

 GeometryContributionToHitGroupIndex) + // system generated index of geometry in

 // bottom level acceleration structure (0,1,2,3..)

 [D3D12\_RAYTRACING\_INSTANCE\_DESC](#_D3D12_RAY_TRACING_INSTANCE_DESC_1).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..) mirroring the order a geometry was placed by the app in its bottom level acceleration structure.

### Miss shader table indexing

MissShaderRecordAddress =

 D3D12\_DISPATCH\_RAYS\_DESC.MissShaderTable.StartAddress + // from: [DispatchRays()](#_DispatchRays())

 D3D12\_DISPATCH\_RAYS\_DESC.MissShaderTable.StrideInBytes \* // from: [DispatchRays()](#_DispatchRays())

 MissShaderIndex // from shader: [TraceRay()](#_TraceRay)

### Callable shader table indexing

CallableShaderRecordAddress =

 D3D12\_DISPATCH\_RAYS\_DESC.CallableShaderTable.StartAddress + // from shader: [TraceRay()](#_TraceRay)

 D3D12\_DISPATCH\_RAYS\_DESC.CallableShaderTable.StrideInBytes \* // from shader: [TraceRay()](#_TraceRay)

 ShaderIndex // from shader: [CallShader()](#_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()](#_TraceRay) from a shader
	+ input to [BuildRaytracingAccelerationStructure()](#_BuildRayTracingAccelerationStructur):
		- 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()](#_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()](#_D3D12_RAY_TRACING_INSTANCE_DESC), 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()](#_TraceRay) 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 an irrelevant virtue – 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_RAY_TRACING_GEOMETRY_FLAGS_1) 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](#_Toc497824603) 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.

### Fixed function ray-triangle intersection specification

To be fleshed out and refined.

For manifold geometry:

A single ray striking an edge in a scene must report an intersection with only one of the incident triangles. For manifold geometry, a ray striking a vertex must report an intersection with only one of the incident triangles.



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.

Details are to be fleshed out further, involving defining the ray-triangle intersection equivalent of the top-left rule used in triangle rasterization. Unlike in triangle rasterization, where the top-left rule guarantees implementation independent behavior, for ray-triangle Intersection in raytracing, the rule might be defined in such a way that a given implementation can guarantee deterministic choices about which triangle to choose in shared intersection cases, but that choice may not be the same across implementations. If an implementation agnostic definition can be found, that would be ideal of course.

#### 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.

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, 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.

As the ray-triangle intersection specification above gets fleshed out, such as consideration of a top-left rule, it may have implications on whether the examples above are conformant without tweaks.

### 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](#_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](#_D3D12_RAY_TRACING_GEOMETRY_TRIANGLE) 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\_AABBS\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_AABBS_DE) 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](#_D3D12_BUILD_RAY_TRACING_ACCELERATIO_1) can change.

This refers to [D3D12\_RAYTRACING\_INSTANCE\_DESC](#_D3D12_RAY_TRACING_INSTANCE_DESC_1) structures in GPU memory. The number of instances, defined by the NumDescs member of [D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC](#_D3D12_BUILD_RAY_TRACING_ACCELERATIO_1), 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](#_Additional_resource_states), 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](#_Additional_resource_states) 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()](#_BuildRayTracingAccelerationStructur_1)) and operations that read from them (such as [DispatchRays()](#_DispatchRays()_1)) (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](#_D3D12_RAY_TRACING_ACCELERATION_STRU_3) 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.

## Ray recursion limit

Raytracing pipeline state objects must [declare](#_D3D12_RAY_TRACING_PIPELINE_CONFIG) a maximum ray recursion depth (in the range [0..[31](#_Constants)]). The ray generation shader is depth 0. Below the maximum recursion depth, shader invocations such as closest hit or miss shaders can call [TraceRay](#_TraceRay)() any number of times. At the maximum recursion depth, [TraceRay](#_TraceRay)() calls result in some form of exception mechanism (to be designed) being invoked. Perhaps this would cause a device removal with information reported to the app that a recursion overflow was the problem.

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.

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 too). The way a 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()](#_GetRayTracingShaderStackSize()). (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](#_D3D12_RAY_TRACING_PIPELINE_CONFIG), 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()](#_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) and **CSmax** (callable shader). The other relevant input is comes from the [D3D12\_RAYTRACING\_PIPELINE\_CONFIG](#_D3D12_RAY_TRACING_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 +

 MaxTraceRecursionDepth\*

 max( ISMax+AHSMax,

 max(CHSMax,MSMax)

 ) +

 2\*CSMax; // 2 is a completely arbitrary choice

### Pipeline stack limit behavior

If making a call exceeds the declared stack size, some form of exception mechanism (to be designed) is invoked, similar to ray recursion overflow. Perhaps this would cause a device removal with information reported to the app that a stack overflow was the problem).

There is no practical limit on declared stack size. The runtime drops calls to [SetPipelineStackSize()](#_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()](#_GetRayTracingShaderStackSize()) 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 which 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()](#_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()](#_CallShader)
* [TraceRay()](#_TraceRay)
* [ReportHit()](#_ReportIntersection)ReportHit()

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

* IgnoreHit() [IgnoreHit()](#_IgnoreIntersection())
* AcceptHitAndEndSearch() [AcceptHitAndEndSearch()](#_TerminateRay)

Issue: From feedback from compiler team, it may be best to have an explicit AllowWaveRepacking() intrinsic that the application can explicitly call where they want to allow it. This wouldn’t be allowed in divergent flow control. If an app calls any of the intrinsics above that are called out as repacking points, perhaps AllowWaveRepacking() must be called alongside them somehow if the app wants to use other wave intrinsics afterwards.

# Raytracing emulation

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 provided 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 force 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.](#_DispatchRays())

## Acceleration structure processing

* [EmitRaytracingAccelerationStructurePostBuildInfo() and CopyRaytracingAccelerationStructure() support dedicated modes for tools to operate on acceleration structures in the following ways:](#_EmitRayTracingAccelerationStructure)
	+ **serialization:**

Driver serializes acceleration structures to a (still) opaque format that tools 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 captured, 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. This also isn’t meant for caching acceleration structures (at least currently). The expectation for now is running a build from scratch will likely be faster than loading an acceleration structure from disk, though perhaps it can be argued that there a benefit to being able to do both builds and deserializing of separate acceleration structures in parallel for greater aggregate bandwidth. Deserialization requires the OS to be in developer mode.

* + **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 memory restrictions](#_Toc496112813) 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.

## Shader Cycle Counter

To assist tools such as PIX make sense of how shader execution progresses in an IHV agnostic way, a cycle counter can be read into shaders (and contribute to logging done by PIX via UAVs). This feature is useful for any shaders, not just raytracing shaders. That said, including this as part of the raytracing spec is just for convenience at the moment, given PIX will first make use of it with raytracing.

There appears no reason not to let applications use this feature as well, perhaps in developer mode only.

Note that this is not a new feature – it was part of D3D11 Shader Model 5. This spec merely moves the feature forward into DXIL.

What’s currently missing in the spec below is a way to correlate cycle counts from a given shader invocation to wall clock time, even if they can’t be directly compared against cycle count values in other shader invocations.

### Basic Semantics

The cycle counter is a 64-bit unsigned integer.

The cycle counter appears as an additional 2\*32-bit (64 bit total) input that can be declared in any shader model 6.2+. There are currently no native 64-bit integer arithmetic operations in shaders, although it is simple enough to emulate this. It may be fine for shaders to just look at the low 32-bits of the counter – this can be requested in the shader.

The counter is an implementation-dependent measure of cycles in the GPU engine, requiring care to interpret it usefully.

TBD: Way to correlate with wall-clock time.

### Interpreting Cycle Counts

The initial value of the counter is undefined.

A single reading of the cycle counter is meaningless. But any shader invocation can poll the counter value any number of times.

Computing a delta from cycle counter readings within a shader invocation is meaningful.

The rest of the text below in this section are parts of the D3D11 spec that need to be refined.

Computing a delta from cycle counter readings across separate shader invocations is not meaningful on all hardware. Developers must obtain information directly from IHVs about whether this is meaningful.

The only IHV agnostic approach to interpreting the counters is to limit calculation of deltas to within a given shader invocation, and only make comparisons of deltas within or between shader invocations.

There are plenty of reasons why test runs will execute differently. The obvious one is that execution of a shader can be interrupted by thread switching, so delta measurements will be arbitrarily larger than the number of cycles spent executing instructions in a given thread.

There is no supported way to find out the frequency of the counter. There is no way to correlate this shader internal counter with external timers such as asynchronous time queries. The counter measurements cannot be correlated with measurements on different hardware by other hardware vendors or even necessarily the same vendor.

If a GPU’s speed changes, such as for power saving, there is no way to know this happened, or its effect on cycle measurements.

Beyond these hints about the care needed to interpret the counter, the onus is on apps to research the properties of new hardware designs that may affect measurements.

### Shader Compiler Constraints

The HLSL shader compiler and driver compilers must treat reads of the cycle counter as execution barriers. Instructions can’t be moved across a counter read, and counter reads can’t be merged.

### DXIL Details

To be determined. For reference, the below was the definition for DXBC in Shader Model 5:

A new input register, vCycleCounter, can be declared in any version 5\_0 (and beyond) shader:

dcl\_input vCycleCounter.{x|xy}.

Reading x yields the 32 LSBs of the 64-bit count, and reading y yields the 32 MSBs.

This register can only be used as the source to a mov instruction, e.g. mov r0.w, vCycleCounter.x.

# API

## Experimental feature exposure

Not shown here yet are placeholder tweaks to the API for it to be surfaced initially as an experimental feature.

For the most part this involves the APIs being exposed via separate experimental interfaces such as ID3D12RaytracingDevice:: or ID3D12RaytracingCommandList:: for raytracing for now as opposed to sitting directly alongside the rest of the graphics and compute APIs on ID3D12Device\*:: or ID3D12CommandList\*::.

## Device methods

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

### CreateStateObject()

HRESULT CreateStateObject(

 \_In\_ const D3D12\_STATE\_OBJECT\_DESC\* pDesc,

 \_In\_ REFIID riid, // ID3D12StateObjectPrototype

 \_COM\_Outptr\_ void\*\* ppStateObject

);

See [State objects](#_State_objects) for an overview.

#### 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;

##### 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;

##### D3D12\_STATE\_SUBOBJECT

typedef struct D3D12\_STATE\_SUBOBJECT

{

 D3D12\_STATE\_SUBOBJECT\_TYPE Type;

 const void\* pDesc;

} D3D12\_STATE\_SUBOBJECT;

##### D3D12\_STATE\_SUBOBJECT\_TYPE

typedef enum D3D12\_STATE\_SUBOBJECT\_TYPE

{

 D3D12\_STATE\_SUBOBJECT\_TYPE\_FLAGS = 0, // D3D12\_STATE\_OBJECT\_FLAGS

 D3D12\_STATE\_SUBOBJECT\_TYPE\_ROOT\_SIGNATURE = 1, // ID3D12RootSignature\*

 D3D12\_STATE\_SUBOBJECT\_TYPE\_LOCAL\_ROOT\_SIGNATURE = 2, // ID3D12RootSignature\*

 D3D12\_STATE\_SUBOBJECT\_TYPE\_NODE\_MASK = 3, // UINT

 D3D12\_STATE\_SUBOBJECT\_TYPE\_CACHED\_STATE\_OBJECT = 4, // D3D12\_CACHED\_STATE\_OBJECT

 D3D12\_STATE\_SUBOBJECT\_TYPE\_DXIL\_LIBRARY = 5, // D3D12\_DXIL\_LIBRARY\_DESC

 D3D12\_STATE\_SUBOBJECT\_TYPE\_EXISTING\_COLLECTION = 6, // D3D12\_EXISTING\_COLLECTION\_DESC

 D3D12\_STATE\_SUBOBJECT\_TYPE\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION = 7,

 // D3D12\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION

 D3D12\_STATE\_SUBOBJECT\_TYPE\_DXIL\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION = 8,

 // D3D12\_DXIL\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION

 D3D12\_STATE\_SUBOBJECT\_TYPE\_RAYTRACING\_SHADER\_CONFIG = 9,

 // D3D12\_RAYTRACING\_SHADER\_CONFIG

 D3D12\_STATE\_SUBOBJECT\_TYPE\_RAYTRACING\_PIPELINE\_CONFIG = 10,

 // D3D12\_RAYTRACING\_PIPELINE\_CONFIG

 D3D12\_STATE\_SUBOBJECT\_TYPE\_HIT\_GROUP = 11, // D3D12\_HIT\_GROUP\_DESC

 D3D12\_STATE\_SUBOBJECT\_TYPE\_MAX\_VALID,

} D3D12\_STATE\_SUBOBJECT\_TYPE;

##### D3D12\_STATE\_OBJECT\_FLAGS

typedef enum D3D12\_STATE\_OBJECT\_FLAGS

{

 D3D12\_STATE\_OBJECT\_FLAGS\_NONE = 0x0,

 D3D12\_STATE\_OBJECT\_FLAG\_ALLOW\_EXTERNAL\_DEPENDENCIES\_ON\_LOCAL\_DEFINITIONS = 0x1,

 D3D12\_STATE\_OBJECT\_FLAG\_ALLOW\_LOCAL\_DEPENDENCIES\_ON\_EXTERNAL\_DEFINITONS = 0x2,

} D3D12\_STATE\_OBJECT\_FLAGS;

D3D12\_STATE\_OBJECT\_FLAG\_ALLOW\_EXTERNAL\_DEPENDENCIES\_ON\_LOCAL\_DEFINITIONS

This applies to state objects of type collection only.

If a collection is included in another state object (e.g. RTPSO), allow shaders / functions in the rest of the RTPSO to depend on (e.g. call) exports from this collection.

In the absence of this flag (default), exports from this collection cannot be directly referenced by other parts of containing state objects (e.g. RTPSO). This can reduce memory footprint for the collection slightly since drivers don’t need to keep uncompiled code in the collection on the off chance that it may get called by some external function that would then compile all the code together. That said, if not all necessary subobject associations have been locally defined for code in this collection, the driver may not be able to compile shader code yet and may still need to keep uncompiled code around.

A subobject association defined externally that associates a subobject (external or local) to a local export does not count as an external dependency on a local definition, so the presence or absence of this flag does not affect whether the association is allowed or not.

Also, regardless of the presence or absence of this flag, shader entrypoints (such as hit groups or miss shaders) in the collection are visible as entrypoints to a containing state object (e.g. RTPSO) if exported by it. In the case of an RTPSO, the exported entrypoints can be used in shader tables for raytracing.

D3D12\_STATE\_OBJECT\_FLAG\_ALLOW\_LOCAL\_DEPENDENCIES\_ON\_EXTERNAL\_DEFINITONS

This applies to state objects of type collection only.

The exports from this collection are allowed to have unresolved references (dependencies) that would have to be resolved (defined) when the collection is included in a containing state object (e.g. RTPSO). This includes depending on an externally defined subobject associations to associate an external or local subobject (e.g. root signature) to a local export.

In the absence of this flag (default), all exports in this collection must have their dependencies fully locally resolved, including any necessary subobject associations being defined locally. Advanced implementations/drivers will have enough information to compile the code in the collection and not need to keep around any uncompiled code (unless the D3D12\_STATE\_OBJECT\_FLAG\_ALLOW\_EXTERNAL\_DEPENDENCIES\_ON\_LOCAL\_DEFINITIONS flag is set). So that when the collection is used in a containing state object (e.g. RTPSO), minimal work needs to be done by the driver (ideally a “cheap” link at most).

##### D3D12\_EXPORT\_DESC

typedef struct D3D12\_EXPORT\_DESC

{

 LPWSTR Name;

 \_In\_opt\_ LPWSTR ExportToRename;

 D3D12\_EXPORT\_FLAGS Flags;

} D3D12\_EXPORT\_DESC;

##### D3D12\_EXPORT\_FLAGS

typedef enum D3D12\_EXPORT\_FLAGS

{

 D3D12\_EXPORT\_FLAG\_NONE = 0x0,

} D3D12\_EXPORT\_FLAGS;

##### D3D12\_DXIL\_LIBRARY\_DESC

typedef struct D3D12\_DXIL\_LIBRARY\_DESC

{

 D3D12\_SHADER\_BYTECODE DXILLibrary;

 UINT NumExports; // Optional, if 0 all exports in the library/collection will be surfaced

 \_In\_reads\_(NumExports) D3D12\_EXPORT\_DESC\* pExports;

} D3D12\_DXIL\_LIBRARY\_DESC;

##### D3D12\_EXISTING\_COLLECTION\_DESC

typedef struct D3D12\_EXISTING\_COLLECTION\_DESC

{

 ID3D12StateObjectPrototype\* pExistingCollection;

 UINT NumExports; // Optional, if 0 all exports in the library/collection will be surfaced

 \_In\_reads\_(NumExports) D3D12\_EXPORT\_DESC\* pExports;

} D3D12\_EXISTING\_COLLECTION\_DESC;

##### D3D12\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION

typedef struct D3D12\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION

{

 const D3D12\_STATE\_SUBOBJECT\* pSubobjectToAssociate;

 UINT NumExports;

 \_In\_reads\_(NumExports) LPWSTR\* pExports;

} D3D12\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION;

Associates a subobject defined directly in a state object (not necessarily the current one or one that has been seen yet) with shader exports. See [Subobject association behavior](#_Subobject_association_behavior) for detail.

##### D3D12\_DXIL\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION

typedef struct D3D12\_DXIL\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION

{

 LPWSTR pDXILSubobjectName;

 UINT NumExports;

 \_In\_reads\_(NumExports) LPWSTR\* 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](#_Subobject_association_behavior) for details.

##### D3D12\_HIT\_GROUP\_DESC

typedef struct D3D12\_HIT\_GROUP\_DESC

{

 LPWSTR HitGroupExport;

 \_In\_opt\_ LPWSTR AnyHitShaderImport;

 \_In\_opt\_ LPWSTR ClosestHitShaderImport;

 \_In\_opt\_ LPWSTR IntersectionShaderImport;

} D3D12\_HIT\_GROUP\_DESC;

##### D3D12\_RAYTRACING\_SHADER\_CONFIG

typedef struct D3D12\_RAYTRACING\_SHADER\_CONFIG

{

 UINT MaxPayloadSizeInBytes;

 UINT MaxAttributeSizeInBytes;

} D3D12\_RAYTRACING\_SHADER\_CONFIG;

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

UINT **MaxPayloadSizeInBytes**

The maximum storage for scalars (counted as 4 bytes each) in ray payloads in raytracing pipelines that contain this program.

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](#_Constants).

##### D3D12\_RAYTRACING\_PIPELINE\_CONFIG

typedef struct D3D12\_RAYTRACING\_PIPELINE\_CONFIG

{

 UINT MaxTraceRecursionDepth;

} D3D12\_RAYTRACING\_PIPELINE\_CONFIG;

A raytracing pipeline needs one raytracing pipeline configuration. If multiple pipeline configurations present they must all match in content. There isn’t benefit to such duplication (at least not foreseen at the time of this feature design). For example defining it once per [collection](#_Collection_state_object) doesn’t help drivers do early shader compilation before a raytracing pipeline is created. This is unlike [D3D12\_RAYTRACING\_SHADER\_CONFIG](#_D3D12_RAYTRACING_SHADER_CONFIG) which does benefit from duplication per collection.

UINT **MaxTraceRecursionDepth**

Limit on ray recursion for the raytracing pipeline. See [Ray recursion limit](#_Ray_recursion_limit_1).

### GetShaderIdentifierSize()

UINT GetShaderIdentifierSize();

Retrieve the size of the unique [shader identifiers](#_Shader_identifier) returned by [GetShaderIdentifier()](#_GetShaderIdentifier()), which the application can copy into [shader records](#_Shader_record) in [shader tables](#_Shader_tables). This implies that the size of shader records (not counting any local root arguments they need) varies depending on the system.

Return value: UINT

Size of a shader identifier in bytes, guaranteed to be a nonzero multiple of 8 bytes no larger than 64 bytes.

### GetRaytracingAccelerationStructurePrebuildInfo()

void GetRaytracingAccelerationStructurePrebuildInfo(

 \_In\_ D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC\* 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()](#_BuildRayTracingAccelerationStructur). The result of this function lets the application provide the correct amount of output storage and scratch storage to [BuildRaytracingAccelerationStructure()](#_BuildRayTracingAccelerationStructur) given the same geometry

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.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\* **pInfo**

Result ( [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO](#_D3D12_RAY_TRACING_ACCELERATION_STRU_5) ) of query.

#### Structures

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

##### D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC

typedef struct D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC

{

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE Type;

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAGS Flags;

 UINT NumDescs;

 D3D12\_ELEMENTS\_LAYOUT DescsLayout;

 union

 {

 const D3D12\_RAYTRACING\_GEOMETRY\_DESC\* pGeometryDescs;

 const D3D12\_RAYTRACING\_GEOMETRY\_DESC\*const\* ppGeometryDescs;

 };

} D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC;

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE **Type**

Type ([D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE](#_D3D12_RAY_TRACING_ACCELERATION_STRU)) of acceleration structure.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAGS **Flags**

Flags ( [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAGS](#_D3D12_RAY_TRACING_ACCELERATION_STRU_4) ) to apply.

UINT **NumDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TOP\_LEVEL, number of instances that will be in the acceleration structure.

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL, number of elements pGeometryDescs or ppGeometryDescs refer to (which one is used depends on DescsLayout).

NumDescs may be 0, in which case the information returned applies to an empty acceleration structure.

D3D12\_ELEMENTS\_LAYOUT **DescsLayout**

How geometry descs are specified (see [D3D12\_ELEMENTS\_LAYOUT](#_D3D12_ELEMENTS_LAYOUT)): an array of descs or an array of pointers to descs. This parameter is only looked at if Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE\_BOTTOM\_LEVEL, otherwise it is ignored.

const D3D12\_RAYTRACING\_GEOMETRY\_DESC\* **pGeometryDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL, and DescsLayout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY, this field is used and points to NumDescs contiguous [D3D12\_RAYTRACING\_GEOMETRY\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_DESC_1) structures on the CPU describing individual geometries.

If Type is not D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL or DescsLayout is not D3D12\_ELEMENTS\_LAYOUT\_ARRAY, this parameter is unused (space repurposed in a union).

pGeometries may be NULL if NumDescs is 0.

The implementation is allowed to look at all the CPU parameters in the geometry descriptions. It may not inspect/dereference any GPU virtual addresses that are part of the descriptions. Except it can check to see if a pointer is NULL or not, such as the optional Transform in [D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_TRIANGLE_1), without dereferencing it.

In other words, the calculation of resource requirements for the acceleration structure does not depend on the actual geometry data (such as vertex positions), rather it can only depend on overall properties (such as the number of triangles).

const D3D12\_RAYTRACING\_GEOMETRY\_DESC\*const\* **ppGeometryDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL, and DescsLayout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY\_OF\_POINTERS, this field is used and points to an array of NumDescs pointers to [D3D12\_RAYTRACING\_GEOMETRY\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_DESC_1) structures on the CPU describing individual geometries.

If Type is not D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL or DescsLayout is not D3D12\_ELEMENTS\_LAYOUT\_ARRAY\_OF\_POINTERS, this parameter is unused (space repurposed in a union).

ppGeometryDescs may be NULL if NumDescs is 0.

The implementation is allowed to look at all the CPU parameters in the geometry descriptions. It may not inspect/dereference any GPU virtual addresses that are part of the descriptions. Except it can check to see if a pointer is NULL or not, such as the optional Transform in [D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_TRIANGLE_1), without dereferencing it.

In other words, the calculation of resource requirements for the acceleration structure does not depend on the actual geometry data (such as vertex positions), rather it can only depend on overall properties (such as the number of triangles).

##### 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;

UINT64 **ResultDataMaxSizeInBytes**

Size required to hold the result of an acceleration structure build based on the specified inputs.

UINT64 **ScratchDataSizeInBytes**

Scratch storage on GPU required during acceleration structure build based on the specified inputs

UINT64 **UpdateScratchDataSizeInBytes**

Scratch storage on GPU required during an acceleration structure update based on the specified inputs. This only needs to be called for the original acceleration structure build, and defines the scratch storage requirement for every acceleration structure update (other than the initial build).

If the D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_ALLOW\_UPDATE flag is not specified, this parameter returns 0.

## 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);

Perform an acceleration structure build on the GPU.

See [Geometry and acceleration structures](#_Geometry_and_acceleration) for an overview.

See [Acceleration structure properties](#_Acceleration_structure_properties) for a discussion of rules and determinism.

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

const D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC\* **pDesc**

Description of the acceleration structure to build.

#### Structures

##### D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC

typedef struct D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC

{

 D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE DestAccelerationStructureData;

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE Type;

 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\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAGS Flags;

 \_In\_opt\_ D3D12\_GPU\_VIRTUAL\_ADDRESS SourceAccelerationStructureData;

 D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE ScratchAccelerationStructureData;

} D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC;

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE **DestAccelerationStructureData**

Location to store resulting acceleration structure. [GetRaytracingAccelerationStructurePrebuildInfo()](#_GetRayTracingAccelerationStructureP) reports the amount of memory required for the result here given a set of acceleration structure build parameters.

The start address must be aligned to 256 bytes ( [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BYTE\_ALIGNMENT](#_Constants)).

The memory pointed to must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE **Type**

Type of acceleration structure to build (see [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE](#_D3D12_RAY_TRACING_ACCELERATION_STRU)).

UINT **NumDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TOP\_LEVEL, number of instances (laid out based on DescsLayout).

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL, number of elements pGeometryDescs or ppGeometryDescs refer to (which one is used depends on DescsLayout).

D3D12\_ELEMENTS\_LAYOUT **DescsLayout**

How geometry descs are specified (see [D3D12\_ELEMENTS\_LAYOUT](#_D3D12_ELEMENTS_LAYOUT)): an array of descs or an array of pointers to descs.

const D3D12\_GPU\_VIRTUAL\_ADDRESS **InstanceDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TOP\_LEVEL, this refers to NumDescs [D3D12\_RAYTRACING\_INSTANCE\_DESC](#_D3D12_RAY_TRACING_INSTANCE_DESC_1) structures in GPU memory describing instances. Each instance must be aligned to 16 bytes ([D3D12\_RAYTRACING\_INSTANCE\_DESC\_BYTE\_ALIGNMENT](#_Constants)).

If DescLayout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY, InstanceDescs points to an array of instance descs in GPU memory.

If DescLayout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY\_OF\_POINTERS, InstanceDescs points to an array in GPU memory of D3D12\_GPU\_VIRTUAL\_ADDRESS pointers to instance descs.

If Type is not D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TOP\_LEVEL, this parameter is unused (space repurposed in a union).

The memory pointed to must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

const D3D12\_RAYTRACING\_GEOMETRY\_DESC\* **pGeometryDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL, and DescsLayout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY, this field is used and points to NumDescs contiguous [D3D12\_RAYTRACING\_GEOMETRY\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_DESC_1) structures on the CPU describing individual geometries.

If Type is not D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL or DescsLayout is not D3D12\_ELEMENTS\_LAYOUT\_ARRAY, this parameter is unused (space repurposed in a union).

The reason pGeometryDescs is a CPU based parameter opposed to InstanceDescs which live on the GPU is, at least for early implementations, the CPU needs to look at some of the information such as triangle counts in pGeometryDescs in order to schedule acceleration structure builds. Perhaps in the future more of the data can live on the GPU.

const D3D12\_RAYTRACING\_GEOMETRY\_DESC\*\* **ppGeometryDescs**

If Type is D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL, and DescsLayout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY\_OF\_POINTERS, this field is used and points to an array of NumDescs pointers to [D3D12\_RAYTRACING\_GEOMETRY\_DESC](#_D3D12_RAYTRACING_GEOMETRY_DESC) structures on the CPU describing individual geometries.

If Type is not D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BOTTOM\_LEVEL or DescsLayout is not D3D12\_ELEMENTS\_LAYOUT\_ARRAY\_OF\_POINTERS, this parameter is unused (space repurposed in a union).

ppGeometryDescs is a CPU based parameter for the same reason as pGeometryDescs described above. The only difference is this option lets the app have sparsely located geometry descs if desired.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAGS **Flags**

[D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAGS](#_D3D12_RAY_TRACING_ACCELERATION_STRU) to use for the build.

D3D12\_GPU\_VIRTUAL\_ADDRESS **SourceAccelerationStructureData**

Address of an existing acceleration structure if an acceleration structure update (incremental build) is being requested, by setting D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PERFORM\_UPDATE in the Flags parameter. Otherwise this address must be NULL.

If this address is the same as DestAccelerationStructureData, the update is to be performed in-place. Any other form of overlap of the source and destination memory is invalid and produces undefined behavior.

The address must be aligned to 256 bytes ( [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BYTE\_ALIGNMENT](#_Constants)), which is a somewhat redundant requirement as any existing acceleration structure passed in here would have already been required to be placed with such alignment anyway.

The memory pointed to must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE **ScratchAccelerationStructureData**

Location where the build will store temporary data. [GetRaytracingAccelerationStructurePrebuildInfo()](#_GetRayTracingAccelerationStructureP) reports the amount of scratch memory the implementation will need for a given set of acceleration structure build parameters.

The start address must be aligned to 256 bytes ( [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BYTE\_ALIGNMENT](#_Constants)).

Contents of this memory going into a build on the GPU timeline are irrelevant and will not be preserved. After the build is complete on the GPU timeline, the memory is left with whatever undefined contents the build finished with.

The memory pointed to must be in state D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS.

##### 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;

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](#_Geometry_and_acceleration) and visualized in [Ray-geometry interaction diagram](#_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;

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_NONE

No options specified for the acceleration structure build.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_ALLOW\_UPDATE

Build the acceleration structure such that it supports future updates (via the flag D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PERFORM\_UPDATE) instead of the app having to entirely rebuild. This option may result in increased memory consumption, build times and lower raytracing performance. Future updates, however, should be faster than building the equivalent acceleration structure from scratch.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_ALLOW\_COMPACTION

Enables the option to compact the acceleration structure by calling [CopyRaytracingAccelerationStructure()](#_CopyRayTracingAccelerationStructure) with the compact mode (see [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE](#_D3D12_RAY_TRACING_ACCELERATION_STRU_3)).

This option may result in increased memory consumption and build times. After future compaction, however, the resulting acceleration structure should consume a smaller memory footprint (certainly no larger) than building the acceleration structure from scratch.

This flag is compatible with all other flags. If specified as part of an acceleration structure update, the source acceleration structure must have also been built with this flag.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PREFER\_FAST\_TRACE

Construct a high quality acceleration structure that maximizes raytracing performance at the expense of additional build time. A rough rule of thumb is the implementation should take about 2-3 times the build time than default in order to get better tracing performance.

This flag is recommended for static geometry in particular. It is also compatible with all other flags except for D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PREFER\_FAST\_BUILD.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PREFER\_FAST\_BUILD

Construct a lower quality acceleration structure, trading raytracing performance for build speed. A rough rule of thumb is the implementation should take about 1/2 to 1/3 the build time than default at a sacrifice in tracing performance.

This flag is compatible with all other flags except for D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PREFER\_FAST\_TRACE.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_MINIMIZE\_MEMORY

Minimize the amount of scratch memory used during the acceleration structure build as well as the size of the result. This option may result in increased build times and/or raytracing times.

This is orthogonal to the D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_ALLOW\_COMPACTION flag (and explicit acceleration structure compaction that it enables). Combining the flags can mean both the initial acceleration structure as well as the result of compacting it use less memory.

The impact of using this flag for a build is reflected in the result of calling [GetRaytracingAccelerationStructurePrebuildInfo()](#_GetRayTracingAccelerationStructureP) before doing the build to retrieve memory requirements for the build.

This flag is compatible with all other flags.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_PERFORM\_UPDATE

Perform an acceleration structure update, as opposed to building from scratch. This is faster than a full build, but can negatively impact raytracing performance, especially if the positions of the underlying objects have changed significantly from the original build of the acceleration structure before updates.

See [Acceleration structure update constraints](#_Acceleration_structure_update) for a discussion of what is allowed to change in an acceleration structure update.

If the addresses of the source and destination acceleration structures are identical, the update is performed in-place. Any other overlapping of address ranges of the source and destination is invalid. For non-overlapping source and destinations, the source acceleration structure is unmodified. The memory requirement for the output acceleration structure is the same as in the input acceleration structure.

This flag is compatible with all other flags. ALLOW\_UPDATE may or may not be set, it makes no difference (updates will continue to be allowed). The other flags selections, aside from ALLOW\_UPDATE and PERFORM\_UPDATE, must match the flags in the source acceleration structure.

Acceleration structure updates can be performed in unlimited succession.The source acceleration structure must have either been created with D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_ALLOW\_UPDATE, \_PERFORM\_UPDATE, or if otherwise cloned, the originating acceleration structure had to have been produced in one of these ways. Essentially, the acceleration structure had to have been constructed with the expectation that updates can occur.

##### D3D12\_ELEMENTS\_LAYOUT

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

typedef enum D3D12\_ELEMENTS\_LAYOUT

{

 D3D12\_ELEMENTS\_LAYOUT\_ARRAY = 0x0,

 D3D12\_ELEMENTS\_LAYOUT\_ARRAY\_OF\_POINTERS = 0x1

} D3D12\_ELEMENTS\_LAYOUT;

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;

D3D12\_RAYTRACING\_GEOMETRY\_TYPE **Type**

[D3D12\_RAYTRACING\_GEOMETRY\_TYPE](#_D3D12_RAYTRACING_GEOMETRY_TYPE) for this geometry.

D3D12\_RAYTRACING\_GEOMETRY\_FLAGS **Flags**

[D3D12\_RAYTRACING\_GEOMETRY\_FLAGS](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) for this geometry.

D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC **Triangles**

[D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_TRIANGLE_1) describing triangle geometry if Type is D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_TRIANGLES. Otherwise this parameter is unused (space repurposed in a union).

D3D12\_RAYTRACING\_GEOMETRY\_AABBS\_DESC **AABBs**

[D3D12\_RAYTRACING\_GEOMETRY\_AABBS\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_AABBS_DE) describing AABB geometry if Type is D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_PROCEDURAL\_PRIMITIVE\_AABBS. Otherwise this parameter is unused (space repurposed in a 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;

D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_TRIANGLES

The geometry consists of triangles described by [D3D12\_RAYTRACING\_GEOMETRY\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_DESC_1).

D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_PROCEDURAL\_PRIMITIVE\_AABBS

The geometry procedurally is defined during raytracing by [intersection shaders](#_Intersection_shaders_(procedural). So for the purpose of acceleration structure builds, the geometry’s bounds are described with axis-aligned bounding boxes via [D3D12\_RAYTRACING\_GEOMETRY\_AABBS\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_AABBS_DE_1).

##### 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;

D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_NONE

No options specified.

D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE

When rays encounter this geometry, the geometry acts as if no any hit shader is present. It is recommended to use this flag liberally, as it can enable important ray processing optimizations. Note that this behavior can be overridden on a per-instance basis with [D3D12\_RAYTRACING\_INSTANCE\_FLAGS](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) and on a per-ray basis using [Ray flags](#_Ray_Flags) in [TraceRay()](#_TraceRay).

D3D12\_RAYTRACING\_FLAG\_NO\_DUPLICATE\_ANYHIT\_INVOCATION

By default, the system is free to trigger an [any hit shader](#_Any_hit_shaders) more than once for a given ray-primitive intersection. This flexibility helps improve the traversal efficiency of acceleration structures in certain cases. For instance, if the acceleration structure is implemented internally with bounding volumes, the implementation may find it beneficial to store relatively long triangles in multiple bounding boxes rather than a larger single box.

However, some application use cases require that intersections be reported to the any hit shader at most once. This flag enables that guarantee for the given geometry, potentially with some performance impact.

This flag applies to all [geometry types](#_D3D12_RAY_TRACING_GEOMETRY_AABBS_DE).

##### D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC

typedef struct D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC

{

 D3D12\_GPU\_VIRTUAL\_ADDRESS Transform;

 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.

D3D12\_GPU\_VIRTUAL\_ADDRESS **Transform**

Address of a 3x4 affine transform matrix in row major layout to be applied to the vertices in the VertexBuffer during an acceleration structure build. The contents of VertexBuffer are not modified. If a 2D vertex format is used, the transformation is applied with the third vertex component assumed to be zero.

If Transform is NULL the vertices will not be transformed. Using Transform may result in increased computation and/or memory requirements for the acceleration structure build.

The memory pointed to must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

DXGI\_FORMAT **IndexFormat**

Format of the indices in the IndexBuffer. Must be one of:

 DXGI\_FORMAT\_UNKNOWN (when IndexBuffer is NULL)

DXGI\_FORMAT\_R32\_UINT

DXGI\_FORMAT\_R16\_UINT

DXGI\_FORMAT **VertexFormat**

Format of the vertices (positions) in VertexBuffer. Must be one of:

DXGI\_FORMAT\_R32G32\_FLOAT (third component assumed 0)

DXGI\_FORMAT\_R32G32B32\_FLOAT

DXGI\_FORMAT\_R16G16\_FLOAT (third component assumed 0)

DXGI\_FORMAT\_R16G16B16A16\_FLOAT (A16 component is ignored, other data can be packed there, such as setting vertex stride to 6 bytes)

UINT **IndexCount**

Number of indices in IndexBuffer. Must be 0 if IndexBuffer is NULL.

UINT **VertexCount**

Number of vertices (positions) in VertexBuffer.

D3D12\_GPU\_VIRTUAL\_ADDRESS **IndexBuffer**

Array of vertex indices. If NULL, triangles are non-indexed. Just as with graphics, the address must be aligned to the size of IndexFormat.

The memory pointed to must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE. Note that if an app wants to share index buffer inputs between graphics input assembler and raytracing acceleration structure build input, it can always put a resource into a 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 alignment on 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 (whereas there is a limit for graphics), other than that the bottom 32bits of the value are all that are used – the field is UINT64 purely to make neighboring fields align cleanly/obviously everywhere. Each vertex position is expected to be at the start address of the stride range and any excess space is ignored by acceleration structure builds. This excess space might contain other app data such as vertex attributes, 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 in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE. Note that if an app wants to share vertex buffer inputs between graphics input assembler and raytracing acceleration structure build input, it can always put a resource into a combination of read states simultaneously, e.g. D3D12\_RESOURCE\_STATE\_VERTEX\_AND\_CONSTANT\_BUFFER | D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

##### D3D12\_RAYTRACING\_GEOMETRY\_AABBS\_DESC

typedef struct D3D12\_RAYTRACING\_GEOMETRY\_AABBS\_DESC

{

 UINT AABBCount;

 D3D12\_GPU\_VIRTUAL\_ADDRESS\_AND\_STRIDE AABBs;

} D3D12\_RAYTRACING\_GEOMETRY\_AABBS\_DESC;

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](#_D3D12_GPU_VIRTUAL_ADDRESS_AND_STRID) describing the GPU memory location where an array of [AABB descriptions](#_D3D12_RAYTRACING_AABB) is to be found, including the data stride between AABBs. The address and stride must each be aligned to 4 bytes ([D3D12\_RAYTRACING\_AABB\_BYTE\_ALIGNMENT](#_Constants)). 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;

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

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.

typedef struct D3D12\_RAYTRACING\_INSTANCE\_DESC

{

 FLOAT Transform[12];

 UINT InstanceID : 24;

 UINT InstanceMask : 8;

 UINT InstanceContributionToHitGroupIndex : 24;

 UINT Flags : 8;

 D3D12\_GPU\_VIRTUAL\_ADDRESS AccelerationStructure;

} D3D12\_RAYTRACING\_INSTANCE\_DESC;

FLOAT **Transform[12]**

A 3x4 transform matrix in row major layout representing the instance-to-world transformation.

UINT **InstanceID**

An arbitrary 24-bit value that can be accessed via [InstanceID()](#_InstanceID()) in shader types listed in [System values and special semantics](#_System_values_and_1).

UINT **InstanceMask**

An 8-bit mask assigned to the instance, which can be used to include/reject groups of instances on a per-ray basis. See the InstanceInclusionMask parameter in [TraceRay()](#_TraceRay). If the value is zero, the instance will never be included, so typically this should be set to some nonzero value..

UINT **InstanceContributionToHitGroupIndex**

Per-instance contribution to add into shader table indexing to select the hit group to use. The indexing behavior is introduced here: [Indexing into shader tables](#_Indexing_into_shader), detailed here: [Addressing calculations within shader tables](#_Addressing_calculations_within_1), and visualized here: [Ray-geometry interaction diagram](#_Ray-geometry_interaction_diagram).

UINT **Flags**

Flags from [D3D12\_RAYTRACING\_INSTANCE\_FLAGS](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) to apply to the instance.

D3D12\_GPU\_VIRTUAL\_ADDRESS **AccelerationStructure**

Address of the bottom-level acceleration structure that is being instanced. The address must be aligned to 256 bytes ([D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BYTE\_ALIGNMENT](#_Constants)), which is a somewhat redundant requirement as any existing acceleration structure passed in here would have already been required to be placed with such alignment anyway.

The memory pointed to must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

##### 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;

D3D12\_RAYTRACING\_INSTANCE\_FLAG\_NONE

No options specified.

D3D12\_RAYTRACING\_INSTANCE\_FLAG\_TRIANGLE\_CULL\_DISABLE

Disables culling for this instance. The [Ray flags](#_Ray_Flags) RAY\_FLAG\_CULL\_BACK\_FACING\_TRIANGLES and RAY\_FLAG\_CULL\_FRONT\_FACING\_TRIANGLES will have no effect on this instance.

D3D12\_RAYTRACING\_INSTANCE\_FLAG\_TRIANGLE\_FRONT\_COUNTERCLOCKWISE

This flag reverses front and back facings, which is useful if for example, the application’s natural winding order differs from the default (described below).

By default, a triangle is front facing if its vertices appear clockwise from the ray origin and back facing if its vertices appear counter-clockwise from the ray origin, in object space in a left-handed coordinate system.

Since these winding direction rules are defined in object space, they are unaffected by instance transforms. For example, an instance transform matrix with negative determinant (e.g. mirroring some geometry) does not change the facing of the triangles within the instance. Per-geometry transforms, by contrast, (defined in [D3D12\_RAYTRACING\_GEOMETRY\_TRIANGLES\_DESC](#_D3D12_RAY_TRACING_GEOMETRY_TRIANGLE_1)), get combined with the associated vertex data in object space, so a negative determinant matrix there *does* flip triangle winding.

D3D12\_RAYTRACING\_INSTANCE\_FLAG\_FORCE\_OPAQUE

The instance will act as if [D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) had been specified for all the geometries in the bottom-level acceleration structure referenced by the instance. Note that this behavior can be overridden by the [ray flag](#_Ray_Flags) RAY\_FLAG\_FORCE\_NON\_OPAQUE.

Mutually exclusive to the D3D12\_RAYTRACING\_INSTANCE\_FLAG\_FORCE\_NON\_OPAQUE flag.

D3D12\_RAYTRACING\_INSTANCE\_FLAG\_FORCE\_NON\_OPAQUE

The instance will act as if [D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) had not been specified for any of the geometries in the bottom-level acceleration structure referenced by the instance. Note that this behavior can be overridden by the [ray flag](#_Ray_Flags) RAY\_FLAG\_FORCE\_OPAQUE.

Mutually exclusive to the D3D12\_RAYTRACING\_INSTANCE\_FLAG\_FORCE\_OPAQUE flag.

##### 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;

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\_ D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE DestBuffer,

 \_In\_ D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TYPE InfoType,

 \_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()](#_CopyRayTracingAccelerationStructure).

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

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE **DestBuffer**

Result storage. Size required and the layout of the contents written by the system depend on InfoType.

The memory pointed to must be in state D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TYPE **InfoType**

Type of post-build information to retrieve.

UINT **NumSourceAccelerationStructures**

Number of pointers to acceleration structure GPUVAs pointed to by pSourceAccelerationStructureData. This number also affects the destination (output), which will be a contiguous array of NumSourceAccelerationStructures output structures, where the type of the structures depends on InfoType.

const D3D12\_GPU\_VIRTUAL\_ADDRESS\* **pSourceAccelerationStructureData**

Pointer to array of GPUVAs of size NumSourceAccelerationStructures. Each GPUVA points to the start of an existing acceleration structure.

The memory pointed to must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

#### Structures

##### D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TYPE

typedef enum D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TYPE

{

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_COMPACTED\_SIZE,

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TOOLS\_VISUALIZATION,

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_SERIALIZATION,

} D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TYPE;

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_COMPACTED\_SIZE

Space requirements for an acceleration structure after compaction.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TOOLS\_VISUALIZATION

Space requirements for generating tools visualization for an acceleration structure (used by tools).

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_SERIALIZATION

Space requirements for serializing an acceleration structure (used by tools, at least for now).

##### 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;

UINT64 **CompactedSizeInBytes**

Space requirement for acceleration structure after compaction.

##### D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TOOLS\_VISUALIZATION\_DESC

typedef struct D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TOOLS\_VISUALIZATION\_DESC

{

 UINT64 DecodedSizeInBytes;

} D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_TOOLS\_VISUALIZATION\_DESC;

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 struct

} D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_SERIALIZATION\_DESC;

UINT64 **SerializedSizeInBytes**

Size of the serialized acceleration structure, including a header. The header is D3D12\_SERIALIZED\_ACCELERATION\_STRUCTURE\_HEADER shown below followed by followed by a list of pointers to bottom-level acceleration structures.

typedef struct D3D12\_SERIALIZED\_ACCELERATION\_STRUCTURE\_HEADER

{

 UINT64 SerializedSizeInBytesIncludingHeader;

 UINT64 DeserializedSizeInBytes;

 UINT64 NumBottomLevelAccelerationStructurePointersAfterHeader; // UINT64 to align

 // subsequent pointers

} D3D12\_SERIALIZED\_ACCELERATION\_STRUCTURE\_HEADER

UINT **NumBottomLevelAccelerationStructurePointers**

How many 64bit GPUVAs will be at the start of the serialized acceleration structure (after D3D12\_SERIALIZED\_ACCELERATION\_STRUCTURE\_HEADER above). For a bottom-level acceleration structure this will be 0. For a top-level acceleration structure, the pointers indicate the acceleration structures being referred to. When deserializing happens, these pointers must be initialized by the app in the serialized data (just after the header) to the new locations of the equivalent acceleration structures. These new locations pointed to at deserialize time need not have been populated with bottom level acceleration structures yet, as long as they have been initialized with the expected deserialized data structures before use in raytracing.. During deserialization, the driver reads the new pointers, using them to produce an equivalent top-level acceleration structure to the original.

### CopyRaytracingAccelerationStructure()

void CopyRayTracingAccelerationStructure(

 \_In\_ D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE 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.

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE **DestAccelerationStructureData**

Destination memory. Required size can be discovered by calling [EmitRaytracingAccelerationStructurePostBuildInfo()](#_D3D12_RAY_TRACING_INSTANCE_DESC) beforehand, if necessary depending on the Mode.

Destination start address must be 256 byte aligned ([D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BYTE\_ALIGNMENT](#_Constants)), regardless of the Mode.

Destination memory range cannot overlap source otherwise results are undefined.

The resource state that the memory pointed to must be in depends on the Mode parameter - see [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE](#_D3D12_RAY_TRACING_ACCELERATION_STRU_3) definitions.

D3D12\_GPU\_VIRTUAL\_ADDRESS **SourceAccelerationStructureData**

Acceleration structure to copy/transform based on the specified Mode. The source acceleration structure remains unchanged and still usable. The operation only involves the source acceleration structure directly specified and not others it may point to. E.g. in the case of a top-level acceleration structure, any bottom-level acceleration structures that it points to are not involved in the operation.

The resource state that the memory pointed to must be in depends on the Mode parameter - see [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE](#_D3D12_RAY_TRACING_ACCELERATION_STRU_3) definitions.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE **Mode**

Type of copy operation to perform.

#### 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;

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE\_CLONE

Copy an acceleration structure while fixing up any self-referential pointers that may be present so that the destination is a self-contained match for the source. Any external pointers to other acceleration structures remain unchanged from source to destination in the copy. The size of the destination is identical to the size of the source.

The source and destination memory must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

 D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE\_COMPACT

Similar to the clone mode, producing a functionally equivalent acceleration structure to source in the destination. Compact mode also fits the destination into a potentially smaller memory footprint (certainly no larger). The size required for the destination can be retrieved beforehand from [EmitRaytracingAccelerationStructurePostBuildInfo()](#_D3D12_RAY_TRACING_INSTANCE_DESC).

This mode is only valid if the source acceleration structure was originally built with the [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_ALLOW\_COMPACTION](#_D3D12_RAY_TRACING_ACCELERATION_STRU) flag, otherwise results are undefined.

The source and destination memory must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE\_VISUALIZATION\_DECODE\_FOR\_TOOLS

Destination takes the layout described in [D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TOOLS\_VISUALIZATION\_HEADER](#_D3D12_BUILD_RAY_TRACING_ACCELERATIO). The size required for the destination can be retrieved beforehand from [EmitRaytracingAccelerationStructurePostBuildInfo()](#_D3D12_RAY_TRACING_INSTANCE_DESC).

This mode is intended for tools such as PIX only, though nothing stops any app from using it. The output is essentially the inverse of an acceleration structure build.

For top-level acceleration structures, the output includes a set of instance descriptions that are identical to the data used in the original build and in the same order.

For bottom-level acceleration structures, the output includes a set of geometry descriptions *roughly* matching the data used in the original build. The output is only a rough match for the original in part because of the tolerances allowed in the specification for [acceleration structures](#_Acceleration_structure), and in part because reporting exactly the same structure as is conceptually encoded may not be simple.

AABBs returned for procedural primitives, for instance, could be more conservative (larger) in volume and even different in number than what is actually in the acceleration structure representation (because it may not be clean to expose the exact representation).

Geometries (each with its own geometry description) must appear in the same order as in the original build, as [shader table indexing](#_Hit_group_table) calculations depends on this.

This overall structure with is sufficient for tools/PIX to be able to give the application some visual sense of the acceleration structure the driver made out of the app’s input. Visualization can help reveal driver bugs in acceleration structures if what is shown grossly mismatches the data the application used to create the acceleration structure, beyond allowed tolerances.

The source memory must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

The destination memory must be in state D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS.

This mode is only permitted when developer mode is enabled on the OS.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE\_SERIALIZE

Destination takes the layout and size described in [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_SERIALIZATION\_DESC](#_D3D12_RAY_TRACING_ACCELERATION_STRU_1), via [EmitRaytracingAccelerationStructurePostBuildInfo()](#_D3D12_RAY_TRACING_INSTANCE_DESC). This serializes an acceleration structure so that tools/PIX can store to a file for later playback (via deserialization). While intended for tools/PIX, nothing stops any app from using this.

The source memory must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

The destination memory must be in state D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS.

When serializing a top-level acceleration structure the bottom-level acceleration structures it refers to do not have to still be present/intact in memory. Likewise bottom-level acceleration structures can be serialized independent of whether any top-level acceleration structures are pointing to them. Said another way, order of serialization of acceleration structures doesn’t matter.

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_COPY\_MODE\_DESERIALIZE

Source must be a serialized acceleration structure, with any pointers (directly after the header) fixed to point to their new locations, as discussed in the [D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_POSTBUILD\_INFO\_SERIALIZATION\_DESC](#_D3D12_RAY_TRACING_ACCELERATION_STRU_1) section.

Destination gets an acceleration structure that is functionally equivalent to the acceleration structure that was originally serialized. It does not matter what order top-level and bottom-level acceleration structures (that the top-level refers to) are deserialized, as long as by the time a top-level acceleration structure is used for raytracing or acceleration structure updates it’s referenced bottom-level acceleration structures are present.

Deserialize only works on the same device and driver version otherwise results are undefined. This isn’t intended to be used for caching acceleration structures, as running a full acceleration structure build is likely to be faster than loading one from disk.

While intended for tools/PIX, nothing stops any app from using this, though at least for now deserialization requires the OS to be in developer mode.

The source memory must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

The destination memory must be in state [D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE](#_Additional_resource_states).

##### 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.

### DispatchRays()

void DispatchRays(\_In\_ ID3D12StateObjectPrototype\* pRaytracingPipeline,

 \_In\_ const D3D12\_DISPATCH\_RAYS\_DESC\* pDesc);

Launch threads of a ray generation shader. See [Initiating raytracing from a CommandList](#_Initiating_ray_tracing) for an overview. Can be called from graphics or compute command lists and bundles.

ID3D12StateObjectPrototype\* **pRaytracingPipeline**

State object of [type](#_D3D12_STATE_OBJECT_TYPE_1): D3D12\_STATE\_OBJECT\_TYPE\_RAYTRACING\_PIPELINE.

const D3D12\_DISPATCH\_RAYS\_DESC\* **pDesc**

Description of the ray dispatch.

#### 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;

} D3D12\_DISPATCH\_RAYS\_DESC;

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE **RayGenerationShaderRecord**

[Shader record](#_Shader_record) for the [Ray generation shader](#_Ray_generation_shaders) to use. Memory must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE\_AND\_STRIDE **MissShaderTable**

[Shader table](#_Shader_tables) for [Miss shader](#_Miss_Shader)s. The stride is record stride. Memory must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE\_AND\_STRIDE **HitGroupTable**

[Shader table](#_Shader_tables) for [Hit Group](#_Hit_groups)s. The stride is record stride. Memory must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

D3D12\_GPU\_VIRTUAL\_ADDRESS\_RANGE\_AND\_STRIDE **CallableShaderTable**

[Shader table](#_Shader_tables) for [Callable shaders](#_Callable_shaders). The stride is record stride. Memory must be in state D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE.

UINT **Width**

Width of ray generation shader thread grid.

UINT **Height**

Height 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;

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;

UINT64 **StartAddress**

Beginning of a VA range.

UINT64 **SizeInBytes**

Size of a VA range.

UINT64 **StrideInBytes**

Define record indexing stride within the memory range.

## StateObjectConfig methods

ID3D12StateObjectConfigPrototype is an interface exported by ID3D12StateObjectPrototype. The following methods are exposed from ID3D12StateObjectConfigPrototype.

### GetShaderIdentifier()

const void\* GetShaderIdentifier(LPWSTR pEntrypointName);

Retrieve the unique identifier for a shader that can be used in [Shader record](#_Shader_record)s. This is only valid for [Ray generation shaders](#_Ray_generation_shaders), [Hit groups](#_Hit_groups), [Miss shaders](#_Miss_shaders) and [Callable shaders](#_Callable_shaders). The state object can be a collection or raytracing pipeline state object, otherwise returns nullptr (not applicable yet since these are the only types of state objects for now).

LPWSTR **pEntrypointName**

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 given by [GetShaderIdentifierSize()](#_D3D12_STATE_OBJECT_TYPE). 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 pEntrypointName);

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](#_Callable_shader_stack) for details on how this contributes to an app’s pipeline stack size calculation.

This is only valid for [Ray generation shaders](#_Ray_generation_shaders), [Hit groups](#_Hit_groups), [Miss shaders](#_Miss_shaders) and [Callable shaders](#_Callable_shaders).

For [Hit groups](#_Hit_groups), stack size must be queried for the individual shaders comprising it: [Intersection shaders](#_Intersection_shaders_(procedural), [Any hit shaders](#_Toc505010241), [Closest 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 pEntrypointName 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.

LPCWSTR **pEntrypointName**

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](#_Hit_groups)) 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](#_Callable_shader_stack) size.

### GetPipelineStackSize()

UINT64 GetPipelineStackSize();

Retrieve the current pipeline stack size. See [Pipeline stack](#_Callable_shader_stack) for the meaning of the size.

This call and [SetPipelineStackSize()](#_SetPipelineStackSize()_1) are not re-entrant. This means if calling either or both from separate threads, the app must synchronize on its own.

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](#_Callable_shader_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()](#_GetPipelineStackSize()) or any use of raytracing pipeline state objects, such as via [DispatchRays()](#_Toc505010306) are not re-entrant. This means if the calling any of these from separate threads, the app must synchronize on its own. Any given [DispatchRays()](#_Toc505010306) call [GetPipelineStackSize()](#_SetPipelineStackSize()) call uses/returns the most recent stack size setting.

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

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()](#_GetRayTracingShaderStackSize()) called with invalid parameters (which return 0xffffffff). In this case the previously set stack size (or default) remains.

## Additional resource states

 D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE = 0x400000

See discussion of this state in [Acceleration structure memory restrictions](#_Acceleration_structure_memory).

## 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](#_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](#_Local_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](#_RayTracingAccelerationStructure) resource type, which can then be passed into [TraceRay()](#_TraceRay). 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\_MAX\_ATTRIBUTE\_SIZE\_IN\_BYTES 32

#define D3D12\_RAYTRACING\_SHADER\_RECORD\_BYTE\_ALIGNMENT 16

#define D3D12\_RAYTRACING\_AABB\_BYTE\_ALIGNMENT 4

#define D3D12\_RAYTRACING\_MAX\_DECLARABLE\_TRACE\_RECURSION\_DEPTH 31

# HLSL

## Types, enums, subobjects and concepts

### Ray flags

Ray flags are passed to [TraceRay()](#_TraceRay) to override transparency, culling, and early-out behavior. For an example, see [here](#_Callable_shaders_1). Shaders that interact with rays can query the current flags via [RayFlags()](#_RayFlags()) intrinsic.

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\_NONE

No options selected.

RAY\_FLAG\_FORCE\_OPAQUE

All ray-primitive intersections encountered in a raytrace are treated as opaque. So no [any hit](#_Any_Hit_Shader_1) shaders will be executed regardless of whether or not the hit geometry specifies [D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1), and regardless of the [instance flags](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) on the instance that was hit.

Mutually exclusive to RAY\_FLAG\_FORCE\_NON\_OPAQUE, RAY\_FLAG\_CULL\_OPAQUE and RAY\_FLAG\_CULL\_NON\_OPAQUE.

RAY\_FLAG\_FORCE\_NON\_OPAQUE

All ray-primitive intersections encountered in a raytrace are treated as non-opaque. So [any hit](#_Any_Hit_Shader_1) shaders, if present, will be executed regardless of whether or not the hit geometry specifies [D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1), and regardless of the [instance flags](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) on the instance that was hit.

Mutually exclusive to RAY\_FLAG\_FORCE\_OPAQUE, RAY\_FLAG\_CULL\_OPAQUE and RAY\_FLAG\_CULL\_NON\_OPAQUE.

RAY\_FLAG\_ACCEPT\_FIRST\_HIT\_AND\_END\_SEARCH

The first ray-primitive intersection encountered in a raytrace automatically causes [AcceptHitAndEndSearch()](#_TerminateRay) to be called immediately after the any hit shader (including if there is no any hit shader).

The only exception is when the preceding [any hit](#_Any_Hit_Shader_1) shader calls [IgnoreHit()](#_IgnoreIntersection()), in which case the ray continues unaffected (such that the next hit becomes another candidate to be the first hit. For this exception to apply, the any hit shader has to actually be executed. So if the any hit shader is skipped because the hit is treated as opaque (e.g. due to RAY\_FLAG\_FORCE\_OPAQUE or [D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) or [D3D12\_RAYTRACING\_INSTANCE\_FLAG\_OPAQUE](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) being set), then [AcceptHitAndEndSearch()](#_TerminateRay) is called.

If a [closest hit](#_Closest_Hit_Shader_1) shader is present at the first hit, it gets invoked (unless RAY\_FLAG\_SKIP\_CLOSEST\_HIT\_SHADER is also present). The one hit that was found is considered “closest”, even though other potential hits that might be closer on the ray may not have been visited.

A typical use for this flag is for shadows, where only a single hit needs to be found.

RAY\_FLAG\_SKIP\_CLOSEST\_HIT\_SHADER

Even if at least one hit has been committed, and the hit group for the closest hit contains a closest hit shader, skip execution of that shader.

One example where this flag could help is illustrated [here](#_Callable_shaders_1).

RAY\_FLAG\_CULL\_BACK\_FACING\_TRIANGLES

Enables culling of back facing triangles. See [D3D12\_RAYTRACING\_INSTANCE\_FLAGS](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) for selecting which triangles are back facing, per-instance.

On instances that specify [D3D12\_RAYTRACING\_INSTANCE\_FLAG\_TRIANGLE\_CULL\_DISABLE](#_D3D12_RAY_TRACING_INSTANCE_FLAGS), this flag has no effect.

On geometry types other than [D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_TRIANGLES](#_D3D12_RAY_TRACING_GEOMETRY_AABBS_DE), this flag has no effect.

This flag is mutually exclusive to RAY\_FLAG\_CULL\_FRONT\_FACING\_TRIANGLES.

RAY\_FLAG\_CULL\_FRONT\_FACING\_TRIANGLES

Enables culling of front facing triangles. See [D3D12\_RAYTRACING\_INSTANCE\_FLAGS](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) for selecting which triangles are back facing, per-instance.

On instances that specify [D3D12\_RAYTRACING\_INSTANCE\_FLAG\_TRIANGLE\_CULL\_DISABLE](#_D3D12_RAY_TRACING_INSTANCE_FLAGS), this flag has no effect.

On geometry types other than [D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_TRIANGLES](#_D3D12_RAY_TRACING_GEOMETRY_AABBS_DE), this flag has no effect.

This flag is mutually exclusive to RAY\_FLAG\_CULL\_BACK\_FACING\_TRIANGLES.

RAY\_FLAG\_CULL\_OPAQUE

Culls all primitives that are considered opaque based on their [geometry](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) and [instance](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) flags.

This flag is mutually exclusive to RAY\_FLAG\_FORCE\_OPAQUE, RAY\_FLAG\_FORCE\_NON\_OPAQUE, and RAY\_FLAG\_CULL\_NON\_OPAQUE.

RAY\_FLAG\_CULL\_NON\_OPAQUE

Culls all primitives that are considered non-opaque based on their [geometry](#_D3D12_RAY_TRACING_GEOMETRY_FLAGS_1) and [instance](#_D3D12_RAY_TRACING_INSTANCE_FLAGS) flags.

This flag is mutually exclusive to RAY\_FLAG\_FORCE\_OPAQUE, RAY\_FLAG\_FORCE\_NON\_OPAQUE, and RAY\_FLAG\_CULL\_OPAQUE.

### Ray description structure

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

struct RayDesc

{

 float3 Origin;

 float TMin;

 float3 Direction;

 float TMax;

};

### 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() 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](#_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](#_Ray_payload_structure) and [attribute structure](#_Intersection_attributes_structure_3). See the API equivalent: [D3D12\_RAYTRACING\_SHADER\_CONFIG](#_D3D12_RAYTRACING_SHADER_CONFIG).

RaytracingShaderConfig shader\_config\_name(maxPayloadSizeInBytes, maxAttributeSizeInBytes);

#### Raytracing pipeline config

Defines the maximum [TraceRay()](#_TraceRay) recursion depth. See the API equivalent: [D3D12\_RAYTRACING\_PIPELINE\_CONFIG](#_D3D12_RAYTRACING_PIPELINE_CONFIG).

RaytracingPipelineConfig config\_name(maxTraceRecursionDepth);

### Intersection attributes structure

Intersection attributes come from one of two sources:

1. Triangle geometry uses 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](#_Any_Hit_Shader) hit attributes.

1. Intersection with axis-aligned bounding boxes for procedural primitives in the raytracing acceleration structure triggers an [intersection shader](#_Intersection_shaders_(procedural). That shader provides a user-defined intersection attribute structure to the [ReportHit()](#_ReportIntersection) call. The [any hit](#_Toc505010241) and [closest hit](#_Closest_hit_shaders) shaders bound in the same [hit group](#_Hit_groups) with this [intersection shader](#_Intersection_shaders_(procedural) 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](#_Constants)).

### Ray payload structure

This is a user-defined structure that is provided as an inout argument in the [TraceRay()](#_TraceRay) call, and as an inout parameter in the shader types that may access the ray payload ([any hit](#_Toc505010241), [closest hit](#_Closest_hit_shaders), and [miss shaders](#_Miss_shaders)). Any shaders that access the ray payload must use the same structure as the one provided at the originating [TraceRay()](#_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()](#_TraceRay) call.

### Call parameter structure

This is a user-defined structure provided as an inout argument in the [CallShader()](#_CallShader) call, and as an inout parameter for the [callable shader](#_Callable_shaders_2). The structure type used in the callable shader must match the structure provided to the corresponding [CallShader()](#_CallShader) call.

## Shaders

These shaders are functions compiled into a library (target lib\_6\_1 for preview release), and identified by an attribute [shader("*shadertype*")] on the shader function.

See [Intrinsics](#_Intrinsics_1) and [System Values](#_System_values_and) 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](#_Shader_limitations_resulting).

### Ray generation shader

shader type: raygeneration

Overview is [here](#_Ray_generation_shaders).

Ray generation shaders call [TraceRay()](#_TraceRay) to generate rays. The initial user-defined ray payload for each ray is provided to the [TraceRay()](#_TraceRay) call site. [CallShader](#_CallShader())() may also be used in ray generation shaders to invoke [callable shaders](#_Callable_shader_2).

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](#_Intersection_shaders_(procedural).

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()](#_ReportIntersection) call. The handling of [ReportHit](#_ReportIntersection())() may stop the intersection shader early, if the [Ray Flag](#_Ray_Flags) RAY\_FLAG\_ACCEPT\_FIRST\_HIT\_AND\_END\_SEARCH is set, or [AcceptHitAndEndSearch()](#_TerminateRay) is called from an [any hit](#_Any_Hit_Shader_1) shader. Otherwise, it returns true if the hit was accepted or false if rejected (see [ReportHit()](#_ReportIntersection) for details). This means that an [any hit](#_Any_Hit_Shader_1) shader, if present, must execute before control conditionally returns to the intersection shader.

Rough Example:

struct CustomPrimitiveDef { ... };

struct MyAttributes { ... };

bool IntersectCustomPrimitiveFrontToBack(

 CustomPrimitiveDef prim,

 float3 origin, float3 dir,

 float rayTMin, inout float curT,

 out MyAttributes attr);

[shader("intersection")]

void intersection\_main()

{

 float THit = RayTCurrent();

 MyAttributes attr;

 while(IntersectCustomPrimitiveFrontToBack(

 CustomPrimitiveDefinitions[LocalConstants.PrimitiveIndex],

 ObjectRayOrigin(), ObjectRayDirection(),

 RayTMin(), THit, attr))

 {

 if (ReportHit(THit, /\*hitKind\*/ 0, attr))

 break;

 }

}

### Any hit shader

shader type: anyhit

Overview is [here](#_Any_hit_shaders).

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](#_Intersection_shader_1) (if any), and executes the [closest hit](#_Closest_Hit_Shader_1) shader on the closest hit so far (if active).
* Call [IgnoreHit()](#_IgnoreIntersection()), which ends the any hit shader and tells the system to continue searching for hits, including returning control to an [intersection shader](#_Intersection_shader_1) (if currently executing) returning false from the ReportHit[ReportHit()](#_ReportIntersection) 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](#_Intersection_shader_1) (if any), returning true at the ReportHit[ReportHit()](#_ReportIntersection) call site to indicate that the hit was accepted.

Even if an any hit shader invocation is ended by IgnoreHit[IgnoreHit()](#_IgnoreIntersection()) or [AcceptHitAndEndSearch](#_TerminateRay)(), 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](#_Closest_hit_shaders).

When the closest hit has been determined or ray intersection search [ended](#_TerminateRay), 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](#_CallShader())() and [TraceRay](#_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)ObjectToWorld());

 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.

}

### Miss shader

shader type: miss

Overview is [here](#_Miss_shaders). The miss shader must include a user defined structure typed payload parameter matching the one supplied to [TraceRay()](#_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](#_CallShader)() and [TraceRay](#_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](#_Callable_shaders_1).

The callable shader is invoked from another shader by using the [CallShader](#_CallShader)() intrinsic. There is a parameter structure supplied at the [CallShader](#_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](#_DispatchRays())() API. The callable shader must declare this parameter as inout. Additionally, the callable shader may read [launch index](#_RayGenIndex()) and [dimension](#_RayGenDimensions()) inputs, see [System Values](#_System_values_and).

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()](#_CallShader()) | ✓ |  |  | ✓ | ✓ | ✓ |
| [TraceRay()](#_TraceRay) | ✓ |  |  | ✓ | ✓ |  |
| [ReportHit()](#_ReportIntersection) |  | ✓ |  |  |  |  |
| [IgnoreHit()](#_IgnoreIntersection())IgnoreHit() |  |  | ✓ |  |  |  |
| [AcceptHitAndEndSearch()](#_TerminateRay)AcceptHitAndEndSearch() |  |  | ✓ |  |  |  |

### CallShader()

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

template<param\_t>

void CallShader(uint ShaderIndex, inout param\_t Parameter);

uint **ShaderIndex**

Provides index into the callable shader table supplied through the [DispatchRays](#_DispatchRays())() API – see [Callable shader table indexing](#_Callable_Shader_Table).

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](#_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);

RaytracingAccelerationStructure **AccelerationStructure**

Top-level acceleration structure to use.

uint **RayFlags**

Valid combination of [Ray flags](#_Ray_Flags).

uint **InstanceInclusionMask**

Bottom 8 bits of InstanceInclusionMask are used to include/reject geometry instances based on the InstanceMask in each [instance](#_D3D12_RAY_TRACING_INSTANCE_DESC_1):

if(!((InstanceInclusionMask & InstanceMask) & 0xff)) { ignore intersection }

uint **RayContributionToHitGroupIndex**

Offset to add into [Addressing calculations within shader tables](#_Addressing_calculations_within_1) 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](#_Addressing_calculations_within_1) 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](#_Addressing_calculations_within_1). Only the bottom 16 bits of this value are used.

RayDesc **Ray**

[Ray](#_Ray_description_structure) to be traced.

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);

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](#_Any_Hit_Shader_1) or [closest hit](#_Closest_Hit_Shader_1) shaders with the [HitKind()](#_HitKind()) intrinsic.

attr\_t **Attributes**

Intersection attributes. The type attr\_t is the user-defined intersection attribute structure. See [Intersection attributes structure](#_Intersection_attributes_structure_1).

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()](#_IgnoreIntersection()). The current ray interval is defined by [RayTMin()](#_RayTMin()) and [RayTCurrent()](#_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()](#_ReportIntersection) 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](#_Intersection_shader_1) running, that stops. Execution passes to the [closest hit shader](#_Closest_Hit_Shader_1) (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 | callable |
| *Ray dispatch system values:* |  |  |  |  |  |  |
| uint2 [DispatchRaysIndex()](#_DispatchRaysIndex()) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| uint2 [DispatchRaysDimensions()](#_DispatchRaysDimensions()) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| *Ray system values:* |  |  |  |  |  |  |
| float3 [WorldRayOrigin()](#_WorldRayOrigin()) |  | ✓ | ✓ | ✓ | ✓ |  |
| float3 [WorldRayDirection()](#_WorldRayDirection()) |  | ✓ | ✓ | ✓ | ✓ |  |
| float [RayTMin()](#_RayTMin()) |  | ✓ | ✓ | ✓ | ✓ |  |
| float [RayTCurrent()](file:///C%3A/Users/amarp/AppData/Roaming/Microsoft/Word/D3D12%20Ray%20Tracing%20Functional%20Spec306411510659907511/HYPERLINK#_CurrentRayT()) |  | ✓ | ✓ | ✓ | ✓ |  |
| uint [RayFlags()](#_RayFlags()) |  | ✓ | ✓ | ✓ | ✓ |  |
| *Primitive/object space system values:* |  |  |  |  |  |  |
| uint [InstanceIndex()](#_CurrentRayT()) |  | ✓ | ✓ | ✓ |  |  |
| uint [InstanceID()](#_InstanceID()) |  | ✓ | ✓ | ✓ |  |  |
| uint [PrimitiveIndex()](#_PrimitiveIndex()) |  | ✓ | ✓ | ✓ |  |  |
| float3 [ObjectRayOrigin()](#_ObjectRayOrigin()) |  | ✓ | ✓ | ✓ |  |  |
| float3 [ObjectRayDirection()](#_ObjectRayDirection()) |  | ✓ | ✓ | ✓ |  |  |
| float3x4 [ObjectToWorld()](#_ObjectToWorld()) |  | ✓ | ✓ | ✓ |  |  |
| float3x4 [WorldToObject()](#_WorldToObject()) |  | ✓ | ✓ | ✓ |  |  |
| *Hit specific system values:* |  |  |  |  |  |  |
| uint [HitKind()](#_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()](#_DispatchRaysDimensions()) system value intrinsic.

uint2 DispatchRaysIndex();

#### DispatchRaysDimensions()

The Width and Height values from the D3D12\_DISPATCH\_RAYS\_DESC structure provided to the originating [DispatchRays](#_DispatchRays())() call.

uint2 DispatchRaysDimensions();

### Ray system values

These system values are available to all shaders in the [hit group](#_Hit_groups) and [miss shaders](#_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](#_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](#_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](#_Intersection_shader_1), 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](#_Any_Hit_Shader_1) shader, it represents the distance to the current intersection being reported.

In the [closest hit](#_Closest_Hit_Shader_1) shader, it represents the distance to the closest intersection accepted.

In the [miss shader](#_Miss_Shader), it is equal to TMax passed to the [TraceRay()](#_TraceRay) call.

#### RayFlags()

This is a uint containing the current [ray flags](#_Ray_Flags).

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();

#### 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.

float3 ObjectRayOrigin();

#### ObjectRayDirection()

Object-space direction for the current ray.

float3 ObjectRayDirection();

#### ObjectToWorld()

Matrix for transforming from object-space to world-space.

float3x4 ObjectToWorld();

#### WorldToObject()

Matrix for transforming from world-space to object-space.

float3x4 WorldToObject();

### Hit specific system values

#### HitKind()

Returns the value passed as HitKind in [ReportHit()](#_ReportIntersection)ReportHit(). If intersection was reported by fixed-function triangle intersection, HitKind will be one of HIT\_KIND\_TRIANGLE\_FRONT\_FACE or HIT\_KIND\_TRIANGLE\_BACK\_FACE.

uint HitKind();