DXR Tutorial 04

Raytracing Pipeline State

# Overview

Just like in regular rasterization code, we need to create a pipeline state that controls the fixed-function units and describes the shader which will be used. In this tutorial we will focus on the API that creates an ID3D12StateObjectPrototypePtr object. It’s a new DX12.1 state interface, and is created differently than ID3D12PipelineState. This entire operation happens during load-time inside **createRtPipelineState().**

# Shader-Libraries

dxcompiler, the new SM6.x compiler, introduces a new concept called shader-libraries. Libraries allow us to compile a file containing multiple shaders without specifying an entry point. We create shader libraries by specifying "lib\_6\_3" as the target profile, which requires us to use an empty string for the entry point. Using dxcompiler is straightforward but is not in the scope of this tutorial. You can take a look at compileShader() to see an example usage.

# Ray-Tracing Shaders

DXR introduces 5 new shader types – ray-generation, miss, closest-hit, any-hit and intersection. All the shaders for this tutorial can be found in *04-Shaders.hlsl*.

## Ray-Generation Shader

Ray-generation shader is the first stage in the ray-tracing pipeline. We will see in tutorial 06 that ray-tracing commands work on a 2D-grid. The ray-generation shader will be executed once per work item. This is where the user generates the primary-rays and dispatches ray-query calls.

Here is our ray-generation shader:

RaytracingAccelerationStructure gRtScene : register(t0);

RWTexture2D<float4> gOutput : register(u0);

[shader("raygeneration")]

void rayGen()

{

uint2 launchIndex = DispatchRaysIndex();

float3 col = linearToSrgb(float3(0.4, 0.6, 0.2));

gOutput[launchIndex] = float4(col, 1);

}

The most notable thing in this file is the first line. There’s a new HLSL data type called RaytracingAccelerationStructure. As you can guess by the name, it contains a resource view to a TLAS. As you can see, it shares the same register type as a regular shader resource view.

gOutput is a UAV to a 2D texture which we will use to output the results.

As for the shader itself, we first declare the type of the shader - [shader("raygeneration")]. Remember that when we create a shader-library we do not specify a shader profile. The compiler uses that declaration to figure out the shader type.

The last thing to note is **DispatchRaysIndex()**. This intrinsic will return the 2D-index of the current work item.

The rest of the shader simply writes a constant color to the screen. In later tutorials we will see how to implement a more interesting RGS which dispatches rays.

## Miss-Shader

A miss-shader will be called whenever a raytrace query did not hit any of the objects in the TLAS. Here’s our miss-shader:

struct Payload

{

bool hit;

};

[shader("miss")]

void miss(inout Payload payload)

{

payload.hit = false;

}

The miss-shader accepts a single inout argument – the ray payload. The ray-payload is a user-defined struct which is used to communicate data between the different shader stages. Note that the payload **must be a struct.**

OK. Now to the last program type.

## Hit-Group

A hit group is a collection of Closest-Hit, Any-Hit and Intersection Shaders. It is a single state element describing how to test for intersection and what should be the behavior in case an intersection is detected.

Any Hit Shader will be invoked for every intersection found in the traversal. Its main use is to programmatically decide whether an intersection should be accepted. For example, for alpha-tested geometry we would like to ignore the intersection if the alpha test failed. The any-hit shader will be ignored for acceleration structures created with the D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE flag. Note that there is no guarantee on the order that any-hit shaders are executed when multiple intersections are found. Don’t assume that the first invocation means the closest intersection to the origin.

Closest Hit Shader will be invoked exactly once per traversal if an intersection was found. As the name suggests, it will be invoked for the closest intersection found which falls in the ray range specified by the user (we will cover this in the next tutorial).

Intersection Shader will be invoked only when the primitive type is axis-aligned bounding-box. This topic is not covered in this tutorial series. For triangles, an internal triangle-intersection shader is used regardless of whether or not an intersection shader was specified. We can ignore this shader type for now.

As you might recall, our acceleration structures were built with the D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE flag and the primitive type is triangles. That means the only shader which is relevant in our case is the CHS.

struct IntersectionAttribs

{

float2 baryCrd;

};

[shader("closesthit")]

void chs(inout Payload payload, in IntersectionAttribs attribs)

{

payload.hit = true;

}

This shader is very similar to the miss-shader above. It accepts the payload and another input – the intersection attributes. The intersection attributes come from the intersection shader. In our case, we are using the built-in intersection shader, and the attributes are the *z* and *w* components of the barycentric coordinates of the primitive we hit. Note that the shader compiler expects the intersection attributes to be a struct – it can’t be an integral type.

# Creating the RT Pipeline State Object

Now that we learnt about the new shader types, we can create our RTPSO. As mentioned before, creating an RTPSO is different from the way we created PSOs in DX12. Instead of a struct similar to D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC, we are going to build an array of D3D12\_STATE\_SUBOBJECT. Each sub-object describes a single element of the state. Most sub-objects reference other data structures, so we need to make sure all the referenced objects are valid when we call CreateStateObject(). For that reason, we are going to create a simple abstraction for each sub-object type.

Let’s go over the code in **createRtPipelineState()**.

*std*::array<D3D12\_STATE\_SUBOBJECT, 10> subobjects;

First, we allocate an array containing 10 sub-objects. We will see why 10 as we progress through the tutorial.

**DxilLibrary**

This is our abstraction for a sub-object of type D3D12\_STATE\_SUBOBJECT\_TYPE\_DXIL\_LIBRARY.

Let’s start from the struct members:

D3D12\_DXIL\_LIBRARY\_DESC dxilLibDesc = {};

D3D12\_STATE\_SUBOBJECT stateSubobject{};

ID3DBlobPtr pShaderBlob;

*std*::*vector*<D3D12\_EXPORT\_DESC> exportDesc;

*std*::*vector*<*std*::*wstring*> exportName;

As you can see, we store a bunch of D3D12 objects. As we will see in a second, a DXIL library sub-objects needs to reference multiple structs and we need to make sure they are valid when we create the RTPSO.

DxilLibrary(ID3DBlobPtr pBlob, const *WCHAR*\* entryPoint[], uint32\_t entryPointCount) : pShaderBlob(pBlob)

The library accepts a single ID3DBlob object which contains an SM6.1 library. This library can contain multiple entry points, and we need to specify which entry points we plan to use. In our case, we store shaders entry points (see createDxilLibrary()).

Next, we will initialize the D3D12\_STATE\_SUBOBJECT. pDesc has a void\* type, so we need to make sure we assign the right data structure to it. In this case, it’s a pointer to D3D12\_DXIL\_LIBRARY\_DESC.

stateSubobject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_DXIL\_LIBRARY;

stateSubobject.pDesc = &dxilLibDesc;

Next, we clear the library-desc and allocate space for the export-desc and export-names.

dxilLibDesc = {};

exportDesc.*resize*(entryPointCount);

exportName.*resize*(entryPointCount);

Now, assuming pBlob is not null, we can initialize the D3D12\_DXIL\_LIBRARY\_DESC object.

dxilLibDesc.DXILLibrary.pShaderBytecode = pBlob->GetBufferPointer();

dxilLibDesc.DXILLibrary.BytecodeLength = pBlob->GetBufferSize();

dxilLibDesc.NumExports = entryPointCount;

dxilLibDesc.pExports = exportDesc.*data*();

We need to set the blob address, blob size, number of exports (AKA entry-points) and a pointer to the array of export-desc. Since we already resized our exportDesc vector, we know that the address will not change. **Take care when dynam****ically resizing vectors**, it’s not uncommon to forget that the address changes which makes pExports invalid.

We then go over the entry-points and initialize the D3D12\_EXPORT\_DESC vector.

for (uint32\_t i = 0; i < entryPointCount; i++)

{

exportName[i] = entryPoint[i];

exportDesc[i].Name = exportName[i].c\_str();

exportDesc[i].Flags = D3D12\_EXPORT\_FLAG\_NONE;

exportDesc[i].ExportToRename = nullptr;

}

2 things to note:

1. We cache the entry-point name into a pre-allocated member vector of strings.
2. We set ExportToRename to nullptr. Later we will see that we need a way to identify each shader inside a state-object. This is usually done by passing the entry-point name to the required function. There could be cases where shaders from different blobs share the same entry-point name, making the identification ambiguous. To resolve this, we can use ExportToRename to give each shader a unique name. In our case, we set it to nullptr since each shader has a unique-entry point name.

Back in createRtPipelineState(), we create a DxilLibrary object and add it to the sub-object array.

// Create the DXIL library

DxilLibrary dxilLib = createDxilLibrary();

subobjects[index++] = dxilLib.stateSubobject;

We got ourselves our first sub-object object! As you can see, there’s a lot of memory management code here. This is a recurring theme with the new method of creating RTPSO.

**HitProgram**

HitProgram is an abstraction over a D3D12\_STATE\_SUBOBJECT\_TYPE\_HIT\_GROUP sub-object. A hit-group is a collection of intersection, any-hit and closest-hit shaders, at most one of each type. Since we don’t use custom intersection-shaders in these tutorials, our HitProgram object only accepts AHS and CHS entry point name.

HitProgram(*LPCWSTR* ahsExport, *LPCWSTR* chsExport, const *std*::*wstring*& name) : exportName(name)

{

desc = {};

desc.AnyHitShaderImport = ahsExport;

desc.ClosestHitShaderImport = chsExport;

desc.HitGroupExport = exportName.c\_str();

subObject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_HIT\_GROUP;

subObject.pDesc = &desc;

}

The code is pretty self-explanatory. The AnyHitShaderImport and ClosestHitShaderImport reference export names declared in the DxilLibrary we created. HitGroupExport is a **unique** name we will use to identify this hit-group in subsequent calls.

**LocalRootSignature**

DXR introduces a new concept called **Local Root Signature (LRS)**. In graphics and compute pipelines, we have a single, global root-signature used by all programs. For ray-tracing, in addition to that root-signature, we can create local root-signatures and bind them to specific shaders. As we will see in the next tutorial, the size of the root-signature affects the size of the **Shader Binding Table** and LRSs allow us to optimize the SBT.

Looking at createRayGenRootDesc(), you can see that creating an LRS is similar to a global root-signature generation, except we need to set the D3D12\_ROOT\_SIGNATURE\_FLAG\_LOCAL\_ROOT\_SIGNATURE flag.

The LRS abstraction is in LocalRootSignature:

LocalRootSignature(ID3D12DevicePtr pDevice, const D3D12\_ROOT\_SIGNATURE\_DESC& desc)

{

pRootSig = createRootSignature(pDevice, desc);

pInterface = pRootSig.*GetInterfacePtr*();

subobject.pDesc = &pInterface;

subobject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_LOCAL\_ROOT\_SIGNATURE;

}

As you can see, we store the ID3D12RootSignature object into a member variable, then set its address into pDesc.

Now that we created a LocalRootSignature object, we need to associate it with one of the shader. We do that using something called an **Export Assoication**.

**ExportAssociation**

An ExportAssociation object binds a sub-object into shaders and hit-groups. The object itself is very simple:

ExportAssociation(const *WCHAR*\* exportNames[], uint32\_t exportCount, const D3D12\_STATE\_SUBOBJECT\* pSubobjectToAssociate)

{

association.NumExports = exportCount;

association.pExports = exportNames;

association.pSubobjectToAssociate = pSubobjectToAssociate;

subobject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_SUBOBJECT\_TO\_EXPORTS\_ASSOCIATION;

subobject.pDesc = &association;

}

There’s one important detail we must remember when creating an ExportAssociation object- pSubobjectToAssociate must point to an object which is part of the array we are passing into CreateStateObject(). Let’s look at how we use the last 2 objects and see what that means:

// Create the ray-gen root-signature and association

LocalRootSignature rgsRootSignature(mpDevice, createRayGenRootDesc().desc);

subobjects[index] = rgsRootSignature.subobject;

uint32\_t rgsRootIndex = index++;

ExportAssociation rgsRootAssociation(&kRayGenShader, 1, &(subobjects[rgsRootIndex]));

subobjects[index++] = rgsRootAssociation.subobject;

First, we create the local root-signature for the ray-generation shader. We then insert its D3D12\_STATE\_SUBOBJECT into the subobjects array. Then, when we create our ExportAssociation object, **we pass the address of subobjects[rgsRootIndex]**. Be careful not making the mistake of storing the address of rgsRootSignature.subobject, it will not work as expected.

The next bit of code creates an empty local root-signature and binds it to the miss-shader and the hit-program. We need to bind an LRS for every object we export from our DxilLibrary .

// Create the miss- and hit-programs root-signature and association

D3D12\_ROOT\_SIGNATURE\_DESC emptyDesc = {};

emptyDesc.Flags = D3D12\_ROOT\_SIGNATURE\_FLAG\_LOCAL\_ROOT\_SIGNATURE;

LocalRootSignature hitMissRootSignature(mpDevice, emptyDesc);

subobjects[index] = hitMissRootSignature.subobject;

uint32\_t hitMissRootIndex = index++;

const *WCHAR*\* missHitExportName[] = { kMissShader, hitProgram.exportName.*c\_str*() };

ExportAssociation missHitRootAssociation(missHitExportName, arraysize(missHitExportName),

&(subobjects[hitMissRootIndex]));

subobjects[index++] = missHitRootAssociation.subobject;

**ShaderConfig**

Next bit is the shader configuration. There are 2 values we need to set:

1. The payload size in bytes. This is the size of the payload struct we defined in the HLSL. In our case the payload is a single bool (**4-bytes in HLSL**).
2. The attributes size in bytes. This is the size of the data the hit-shader accepts as its intersection-attributes parameter. For the built-in intersection shader, the attributes size is 8-bytes (2 floats).

ShaderConfig(uint32\_t maxAttributeSizeInBytes, uint32\_t maxPayloadSizeInBytes)

{

shaderConfig.MaxAttributeSizeInBytes = maxAttributeSizeInBytes;

shaderConfig.MaxPayloadSizeInBytes = maxPayloadSizeInBytes;

subobject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_RAYTRACING\_SHADER\_CONFIG;

subobject.pDesc = &shaderConfig;

}

The code is self-explanatory. Once we create a ShaderConfig object, we need to associate it with our shaders, which is what the following snippet does.

Again, note that the ExportAssociation object accepts the address of a sub-object from the subobjects array.

// Bind the payload size to the programs

ShaderConfig shaderConfig(sizeof(float)\*2, sizeof(float)\*1);

subobjects[index] = shaderConfig.subobject;

uint32\_t shaderConfigIndex = index++;

const *WCHAR*\* shaderExports[] = { kMissShader, hitProgram.exportName.*c\_str*(), kRayGenShader };

ExportAssociation configAssociation(shaderExports, arraysize(shaderExports),

&(subobjects[shaderConfigIndex]));

subobjects[index++] = configAssociation.subobject;

**PipelineConfig**

The pipeline configuration is a global sub-object affecting all pipeline stages. In the case of raytracing, it contains a single value - MaxTraceRecursionDepth. This value simply tells the pipeline how many recursive raytracing calls we are going to make. Our object looks as follows:

PipelineConfig(uint32\_t maxTraceRecursionDepth)

{

config.MaxTraceRecursionDepth = maxTraceRecursionDepth;

subobject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_RAYTRACING\_PIPELINE\_CONFIG;

subobject.pDesc = &config;

}

Since our ray-generation shader doesn’t make any raytracing calls, we set this value to 0.

// Create the pipeline config

PipelineConfig config(0);

subobjects[index++] = config.subobject;

**GlobalRootSignature**

The last piece of the puzzle is the global root-signature. As the name suggests, this root-signature affects all shaders attached to the pipeline. The final root-signature of a shader is defined by both the global and the shader’s local root-signature. The code is straightforward. This is how we initialize the GlobalRootSignature object:

GlobalRootSignature(ID3D12DevicePtr pDevice, const D3D12\_ROOT\_SIGNATURE\_DESC& desc)

{

pRootSig = createRootSignature(pDevice, desc);

pInterface = pRootSig.*GetInterfacePtr*();

subobject.pDesc = &pInterface;

subobject.Type = D3D12\_STATE\_SUBOBJECT\_TYPE\_ROOT\_SIGNATURE;

}

And this is how we use it, setting the global root-signature to an empty signature:

GlobalRootSignature root(mpDevice, {});

mpEmptyRootSig = root.pRootSig;

subobjects[index++] = root.subobject;

**CreateStateObject**

Now that we initialized our array of D3D12\_STATE\_SUBOBJECT, we can finally create the ID3D12StateObjectPrototype object. This is done using the following, simple snippet:

D3D12\_STATE\_OBJECT\_DESC desc;

desc.NumSubobjects = index;

desc.pSubobjects = subobjects.*data*();

desc.Type = D3D12\_STATE\_OBJECT\_TYPE\_RAYTRACING\_PIPELINE;

ID3D12DeviceRaytracingPrototypePtr pRtDevice = mpDevice;

d3d\_call(pRtDevice->CreateStateObject(&desc, *IID\_PPV\_ARGS*(&mpPipelineState)));

And we’re done!

There are more details and fine-print related to how to use sub-object and create a state-objects. I strongly suggest you read the spec to get all the details, but for now we have enough to work with.