DXR Tutorial 05

Shader Table

# Before We Begin

I strongly urge you to read the Shader-Table section in the spec. It covers many topics and details which are far beyond the scope of this tutorial. This tutorial will only cover the basic concepts. In the next tutorials we will see more advanced usages of the Shader-Table.

# Overview

In the previous tutorials, we created the *acceleration structures* that describe the scene’s geometry and the *ray-tracing pipeline state object* which specifies the programs that will be used.

The last piece required for rendering is the **Shader-Table**. It’s a GPU-visible buffer which is owned and managed by the application – allocation, data updates, etc. The shader-table is an array of **records** and it has 2 roles:

1. Describe the relation between the scene’s geometry and the program to be executed.
2. Bind resources to the pipeline.

The first role is required because we can have multiple hit and miss programs attached to the state object and we need to know which shader to execute when a geometry is hit (or nothing was hit).

The second role is required because:

1. We can create each program with a different local root-signature.
2. Each geometry might require a different set of resources (vertex-buffer, textures, etc.)

Note that the API allows to use multiple shader-tables in a single **DispatchRays()** call. For simplicity, we will use a single shader-table in this tutorial.

# Shader Table Records

Each shader-table record has 2 sections. It begins with an opaque program identifier (obtained by calling ID3D12StateObjectPropertiesPtr::GetShaderIdentifier()) followed by a root table containing the shader resource bindings.

The root-table is very similar to the regular rasterization root-table. The difference is that in our case we set the entries directly into the buffer instead of using setter methods. The sizes of the different entries are slightly different than those described in the [D3D12 root signature limits](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899209(v=vs.85).aspx):

* **Root Constants** are 4 bytes.
* **Root Descriptors** are 8 bytes.
* **Descriptor Tables** are **8 bytes**. This is different than the size required by the regular root-signature.

For root constants and root descriptors we set the same data as what would be passed to the setter functions. Descriptor table is different – we need to set the D3D12\_GPU\_DESCRIPTOR\_HANDLE::ptr field.

Another important thing is that root-descriptors must be stored at an 8-byte aligned address, so in some cases padding might be required.

# The Shader-Table Layout

The shader-table is an array of shader-table records. There are no rules on how the records should be laid out. There are several parameters which determine how indexing happens. For now, let’s focus on creating a shader-table which fits the programs and geometry we created. In later tutorials we will cover more advanced layouts.

The shader-table in this tutorial is created in **createShaderTable()**.

The first thing we need to do is create the buffer for the shader-table. For this, we need to figure out what the shader-table size is.

As mentioned before, all shader-table records must have the same size. We will choose that size based on the largest required entry. In our case, it’s straightforward – only the RayGen shader requires shader resources, so its record size is the largest.

Remember that each shader-table record starts with an opaque program identifier. Next, we need to add the size of the data required for the root-table. We created the Hit Program’s root-signature with a single descriptor-table entry. A descriptor-table consumes 8 bytes, so let’s add it to the entry size:

mShaderTableEntrySize += 8; // The ray-gen's descriptor table

Finally, the entry size must be aligned up to D3D12\_RAYTRACING\_SHADER\_RECORD\_BYTE\_ALIGNMENT.

mShaderTableEntrySize = align\_to(D3D12\_RAYTRACING\_SHADER\_RECORD\_BYTE\_ALIGNMENT, mShaderTableEntrySize);

We have 3 programs and a single geometry, so we need 3 entries (we’ll get to why the number of entries depends on the geometry count in later tutorials).

uint32\_t shaderTableSize = mShaderTableEntrySize \* 3;

Now that we have the size, we can create the buffer. For simplicity, we create on the upload heap.

mpShaderTable = createBuffer(mpDevice, shaderTableSize, *D3D12\_RESOURCE\_FLAG\_NONE*, *D3D12\_RESOURCE\_STATE\_GENERIC\_READ*, kUploadHeapProps);

The shader-table buffer can also be created on the default heap, in which case we need to transition it to the D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE before we call **DispatchRays()**.

Next, we map it and set the program identifiers. In our example, the first record will be for the RayGen program, followed by the miss program. The hit program record will be the last.

To get the shader identifier, we need to use the ID3D12StateObjectProperties interface. We get it using the following snippet:

ID3D12StateObjectPropertiesPtr pRtsoProps;

mpPipelineState->*QueryInterface*(*IID\_PPV\_ARGS*(&pRtsoProps));

Once we converted the pointer, we can get the identifiers:

// Entry 0 - ray-gen program ID and descriptor data

*memcpy*(pData, pRtsoProps->GetShaderIdentifier(kRayGenShader), *D3D12\_SHADER\_IDENTIFIER\_SIZE\_IN\_BYTES*);

// This is where we need to set the descriptor data for the ray-gen shader

// Entry 1 - miss program

*memcpy*(pData + mShaderTableEntrySize, pRtsoProps->GetShaderIdentifier(kMissShader), *D3D12\_SHADER\_IDENTIFIER\_SIZE\_IN\_BYTES*);

// Entry 2 - hit program. Program ID and one constant-buffer as root descriptor

uint8\_t\* pHitEntry = pData + mShaderTableEntrySize \* 2; // +2 skips the ray-gen and miss entries

*memcpy*(pHitEntry, pRtsoProps->GetShaderIdentifier(kHitGroup), *D3D12\_SHADER\_IDENTIFIER\_SIZE\_IN\_BYTES*);

The program identifier entry must be placed at the beginning of the record. Also note that we didn’t initialize the root-table for the RayGen shader yet.

That’s pretty much it. We got ourselves a Shader Table!