DXR Tutorial 06

Raytrace()

# Overview

At this stage, we have all the pieces of the puzzle required for ray-tracing. We have the scene, also known as the Acceleration Structure, we have the state, also known as the Ray Tracing Pipeline State Object, and we have the Shader Table. It’s time to put them to use.

At the end of this tutorial, we will be able to clear the screen using all the objects above. OK, that might not sound very impressive, but it’s an imperative step. Let’s begin.

# Shader Resources

As you might recall, our ray-generation shader writes the result to a 2D UAV. It also requires the top-level acceleration structure (TLAS). We need to create the following resources:

1. A 2D texture that will serve as the output texture.
2. A CBV/UAV/SRV heap with space for 2 entries.
3. A UAV for the output texture.
4. An SRV for the TLAS.

Steps 1-3 are trivial and only require regular DX12 calls which will not be explained. Let’s talk about the 4th step (the code is in createShaderResources()).

Creating an SRV for the TLAS is slightly different than regular SRV. D3D12\_SHADER\_RESOURCE\_VIEW\_DESC has a new field – RaytracingAccelerationStructure – which is used to pass the GPU virtual address of the TLAS. Other than that, the code is very similar to regular SRV creation:

D3D12\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc = {};

srvDesc.ViewDimension = D3D12\_SRV\_DIMENSION\_RAYTRACING\_ACCELERATION\_STRUCTURE;

srvDesc.Shader4ComponentMapping = D3D12\_DEFAULT\_SHADER\_4\_COMPONENT\_MAPPING;

srvDesc.RaytracingAccelerationStructure.Location = mpTopLevelAS->GetGPUVirtualAddress();

D3D12\_CPU\_DESCRIPTOR\_HANDLE srvHandle = mpSrvUavHeap->GetCPUDescriptorHandleForHeapStart();

srvHandle.ptr += mpDevice>GetDescriptorHandleIncrementSize(D3D12\_DESCRIPTOR\_HEAP\_TYPE\_CBV\_SRV\_UAV);

mpDevice->CreateShaderResourceView(nullptr, &srvDesc, srvHandle);

You may have noticed that the first entry in the heap is the UAV and second entry is the SRV. This matches the ray-generation shader expected root-table layout – we created its root-signature with a single table with 2 entries (refresh your memory by looking at createRayGenRootDesc()).

# Binding the Resources

Now that we have the resources, we need to bind them to the program. To do that, we need to get back to the shader table initialization code – createShaderTable().

We already initialized all the program identifiers. We need only add the descriptor table entry for the ray-generation shader. The descriptor-table entry comes immediately after the program identifier and its 8-bytes long. The data we need to set is the pointer of the GPU descriptor handle of the table. In our case, the table **is** the entire heap and we can use the heap-start handle. We end up with the following code for the ray-gen entry:

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

*memcpy*(pData, pRtsoProps->GetShaderIdentifier(kRayGenShader), progIdSize);

uint64\_t heapStart = mpSrvUavHeap->GetGPUDescriptorHandleForHeapStart().ptr;

\*(uint64\_t\*)(pData + progIdSize) = heapStart;

# 

# Other Required Changes

These are not DXRT specific, so we’ll mentioned briefly:

* We need to set the descriptor heap in beginFrame()
* We added code to onFrameRender() which copies the output texture into the back-buffer and handle resource barriers correctly.

# Our Ray-Generation Shader

As promised in tutorial 04, let’s go over our ray-generation shader.

[shader("raygeneration")]

void rayGen()

{

uint3 launchIndex = DispatchRaysIndex();

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

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

}

Assuming you are familiar with SM6.0, the code should look familiar. The only new thing here is **DispatchRaysIndex()**. The intrinsic returns the index of the work item being processed. In our case, as we will see in a second, that maps directly to screen coordinates.

# Let’s Raytrace!

Now that we have everything in place, we can finally raytrace. First, we need to initialize a D3D12\_DISPATCH\_RAYS\_DESC struct. It has 4 sections which we will cover next.

The first section is simply the width and height for the ray-generation shader thread grid.

D3D12\_DISPATCH\_RAYS\_DESC raytraceDesc = {};

raytraceDesc.Width = mSwapChainSize.x;

raytraceDesc.Height = mSwapChainSize.y;

raytraceDesc.Depth = 1;

The next section describes the ray-generation entry. We only have a single ray-generation entry per shader-table. In our case it’s the first entry in the shader-table buffer.

raytraceDesc.RayGenerationShaderRecord.StartAddress =

mpShaderTable->GetGPUVirtualAddress() + 0 \* mShaderTableEntrySize;

raytraceDesc.RayGenerationShaderRecord.SizeInBytes = mShaderTableEntrySize;

Next comes the miss-shaders entries. We can have multiple entries – all of them must share the same buffer and we need to specify the stride between miss-shader entries. In our case, the first miss-shader entry is the second entry in the shader-table and the stride is our shader-table entry size.

*size\_t* missOffset = 1 \* mShaderTableEntrySize;

raytraceDesc.MissShaderTable.StartAddress =

mpShaderTable->GetGPUVirtualAddress() + missOffset;

raytraceDesc.MissShaderTable.StrideInBytes = mShaderTableEntrySize;

raytraceDesc.MissShaderTable.SizeInBytes = mShaderTableEntrySize; // Only a single miss-entry

Finally comes the hit-programs entries. It’s very similar to the miss-shaders entry. In our case, the first hit-entry is the second entry in the shader-table.

*size\_t* hitOffset = 2 \* mShaderTableEntrySize;

raytraceDesc.HitGroupTable.StartAddress = mpShaderTable ->GetGPUVirtualAddress() + hitOffset;

raytraceDesc.HitGroupTable.StrideInBytes = mShaderTableEntrySize;

raytraceDesc.HitGroupTable.SizeInBytes = mShaderTableEntrySize;

Next, we need to set the global-root signature and initialize it. Since we are not using a global-root signature in our tutorials, we will set an empty compute root-signature.

mpCmdList->SetComputeRootSignature(mpEmptyRootSig);

Finally, we need to set our RT pipeline state object. This is done through the new ID3D12GraphicsCommandList4::SetPipelineState1() function.

mpCmdList->SetPipelineState1(mpPipelineState.GetInterfacePtr());

Now that we have everything ready, we can call **DispatchRays()**.

mpCmdList->DispatchRays(&raytraceDesc);

That’s all. We can now clear the screen using the ray-generation shader! That’s not too exciting, but we’ve made a lot of progress. The next step is to render something more interesting. But that’s a different tutorial…