DXR Tutorial 07

Basic Shaders

# Overview

The previous tutorials focused on the C++ side of things, which led us all the way to the ray-generation shader. In case you forgot – our acceleration structure contains a triangle. In this tutorial we will learn how to trace rays and render the triangle to the screen. Most of this tutorial will focus on the shaders.

# A Very Simplified Execution Description

You probably know already, but here’s how ray-tracing in DXRT-world works:

In the ray-generation shader, we would like to create rays. Each ray will be tested for intersection against the acceleration structure. If the ray hits nothing, the miss shader will be invoked. If there was a hit, the closest hit shader will be invoked for the closest intersection.

The traversal and intersection test happen in a fixed-function unit, and we will treat them as a black box.

# Tracing Rays

DXR introduces a new HLSL struct – RayDesc, and a new intrinsic – **TraceRay()** which can be used by the shader to initiate a ray-tracing query.

Let’s look at RayDesc.

struct RayDesc

{

float3 Origin;

float TMin;

float3 Direction;

float TMax;

};

Origin and Direction are, well, the ray’s world-space origin and direction.

TMin and TMax define the parametric distance of the ray interval (where 0 means the origin).

We initialize this structure at the start of the function.

First, we get the current work-item index and the dispatch dimensions.

uint2 launchIndex = DispatchRaysIndex();

uint2 launchDim = DispatchRaysDimensions();

float2 crd = float2 (launchIndex);

float2 dims = float2 (launchDim);

We then use these values to calculate the ray’s direction.

We do not have a camera, so we set the origin to be “2”in front of the triangle (which was centered around the origin) in world space.

float2 d = ((crd / dims) \* 2.f - 1.f);

float aspectRatio = dims.x / dims.y;

RayDesc ray;

ray.Origin = float3(0, 0, -2);

ray.Direction = normalize(float3(d.x \* aspectRatio, -d.y, 1));

ray.TMin = 0;

ray.TMax = 100000;

The direction creation is basic ray-tracing code.

TMin and TMax are set to arbitrary values. We just need to make sure that the triangle will be in range (which at a distance of roughly 2 units away from the camera, it will).

Now that we have a ray, we need to start the traversal. We do that by calling **TraceRay**().

RayPayload payload;

TraceRay(gRtScene, 0, 0xFF, 0, 0, 0, ray, payload);

The first parameter is the top-level acceleration-structure SRV.

The second parameter is the ray flags. These flags allow us to control the traversal behavior, for example enable back-face culling.

The 3rd parameter is the ray-mask. It can be used to cull entire objects when tracing rays. We will not cover this topic in the tutorials. **0xFF** means no culling.

Parameter 4 and 5 are RayContributionToHitGroupIndex and MultiplierForGeometryContributionToHitGroupIndex. They are used for shader-table indexing. We will cover them in later tutorials, for now we will set both to 0.

The 6th Parameter is the miss-shader index. This index is relative to the base miss-shader index we passed when calling **DispatchRays**(). We only have a single miss-shader, so we will set the index to 0.

The 7th parameter is the RayDesc object we created.

We will discuss the 8th parameter in the following section.

Tracing rays from the ray-generation shader has an implication on the C++ code. Up until now we assumed that the TraceRay() is not being called, and we set the maxTraceRecursionDepth in PipelineConfig to 0. Since the ray-generation shader calls TraceRay(), we need to set maxTraceRecursionDepth to 1.

# Payload

The last TraceRay() parameter is called a **payload**. TraceRay() is actually a template function, with a single template argument PayloadType. It must be a struct type and can have arbitrary layout. The payload is used to communicate data between the different shader stages. We can use it to pass data into the hit or miss shaders and get the result from them.

In our case, we have no data to pass into the shaders. We only want to read back a float3 value so we define a struct called RayPayload with a single float3 field.

The payload has implications on the C++ code. When creating the ray-tracing programs, we need to set the maximum payload size into D3D12\_RAYTRACING\_SHADER\_CONFIG. We must use the same value for all programs used with a single pipeline-state.

Our payload is 12 bytes – we pass this value when creating the ShaderConfig objects.

# Miss Shader

The expected behavior from our miss shader is to return the clear color.

[shader("miss")]

void miss(inout RayPayload payload)

{

payload.color = float3(0.4, 0.6, 0.2);

}

The only notable thing in this code is the fact that the payload is declared with the inout modifier;

# Closest Hit Shader

struct IntersectionAttribs

{

float2 baryCrd;

};

[shader("closesthit")]

void chs(inout RayPayload payload, in IntersectionAttribs attribs)

{

float3 barycentrics = float3 (1.0 – attribs.baryCrd.x – attribs.baryCrd.y,

attribs.baryCrd.x,

attribs.baryCrd.y);

const float3 A = float3(1, 0, 0);

const float3 B = float3(0, 1, 0);

const float3 C = float3(0, 0, 1);

payload.color = A \* barycentrics.x + B \* barycentrics.y + C \* barycentrics.z;

}

The CHS will be called only if the ray hit a primitive, and only for the closest hit-point. Like the miss-shader, it accepts the ray-payload. The second parameter is the attributes provided by the intersection shader. We are using the built-in intersection shader, so the IntersectionAttribs struct contains a single float2 and this value represents the y and z components of the barycentric coordinates.

Once we launch the application, we can finally see our triangle!

