DXR Tutorial 13

Adding a Second Ray Type

# Overview

So far, we’ve been dealing with a single ray type – the primary ray. In ray-tracing, we usually want to trace rays originating at the hit-point. These are called secondary rays. We use them to check if the light-source hits the surface, compute reflection, AO and more.

In this tutorial we learn how to add a new ray type. We will add support for shadows using shadow-rays. We will use a simplified version of the shadow-ray, where only the plane is a shadow receiver. When a primary-ray hits the plane, we will trace a ray from the hit-point in the direction of the light source using the same TLAS. If the ray hits a geometry, then the hit point is in shadow. Otherwise it is lit.

# The Shadow Ray

A “ray” is a combination of a miss-program and a hit-program. For shadows, we only care if there was a hit or no. Both the closest-hit and miss shaders can be found in *`13-shaders.hlsl`*

struct ShadowPayload

{

bool hit;

};

[shader("closesthit")]

void shadowChs(inout ShadowPayload payload, in IntersectionAttribs attribs)

{

payload.hit = true;

}

[shader("miss")]

void miss(inout ShadowPayload payload)

{

payload.hit = false;

}

The payload contains a single Boolean value.

Theoretically, it’s more efficient to use an any-hit shader instead of closest-hit shader for shadow-rays. In our case it will not work, since we create the acceleration structures with D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE flag, which means the AHS will not be executed.

# Ray-Tracing Pipeline State Object

We need to make the following changes to **createRtPipelineState()**:

* In createDxilLibrary (), add the new entry points to the list.
* Create a new HitProgram for the shadowChs().
* We are going to use the empty-root signature with the shadow miss and hit-program, so we add them to emptyRootAssociation.
* Create a new ShaderConfig object for the shadow shaders. The attributes size is 2 floats, while the payload size is a single uint. Associate this object to the shaders using an ExportAssociation.
* Change the maxTraceRecursionDepth in the PipelineConfig object to 2. We’re going to call **TraceRay()** once from the ray-gen shader and once from the plane-CHS.
* Create and associate a root-signature with the plane-CHS. This happens in **createPlaneHitRootDesc()**. This root-signature contains a single SRV which will be used to bind the TLAS to the shader.

# Shader-Table Layout

By now it should be clear that the shader-table layout and indexing controls which shaders will be invoked when a ray hit a geometry or missed everything in the scene.

For reference, here is the **hit-program** indexing computation:

entryIndex =

InstanceContributionToHitGroupIndex +

GeometryIndex\* MultiplierForGeometryContributionToShaderIndex +

RayContributionToHitGroupIndex)

And this is for the **miss-program** it’s missShaderIndex passed to **TraceRay()**.

Let’s look at the new shader-table layout, and we’ll follow up with explanation on the indexing.

**Hit**

**Primary**

**Instance 2**

**Geom 0**

**Hit**

**Shadow**

**Instance 2**

**Geom 0**

**Hit**

**Primary**

**Instance 1**

**Geom 0**

**Hit**

**Shadow**

**Instance 1**

**Geom 0**

**Hit**

**Primary**

**Instance 0**

**Geom 1**

**Hit**

**Shadow**

**Instance 0**

**Geom 1**

**Hit**

**Shadow**

**Instance 0**

**Geom 0**

**RayGen**

**Miss**

**Primary Ray**

**Miss**

**Shadow Ray**

**Hit**

**Primary**

**Instance 0**

**Geom 0**

We have 11 entries:

Entry 0 - Ray-gen program

Entry 1 - Miss program for the primary ray

Entry 2 - Miss program for the shadow ray

Entries 3,4 - Hit programs for triangle 0 (primary followed by shadow)

Entries 5,6 - Hit programs for the plane (primary followed by shadow)

Entries 7,8 - Hit programs for triangle 1 (primary followed by shadow)

Entries 9,10 - Hit programs for triangle 2 (primary followed by shadow)

This is a common layout when multiple rays are required. In our case the records are tightly packed and use a single buffer, but that’s not mandatory. The layout follows these 2 conventions:

* Records for each geometry are consecutive.
* The shadow-ray record always follows its matching primary-ray record.

## Shader-Table Indexing

As a reminder, here is a summary of the different values used to calculate an shader-table address:

* D3D12\_DISPATCH\_RAYS\_DESC contains StartAddress and StrideInBytes fields per shader type.
* RayContributionToHitGroupIndex – One of the parameters of the HLSL’s TraceRay() function. The maximum allowed value is 15.
* MultiplierForGeometryContributionToShaderIndex – One of the parameters of the HLSL’s TraceRay() function. The maximum allowed value is 15.
* MissShaderIndex – One of the parameters of the HLSL’s TraceRay () function.
* InstanceContributionToHitGroupIndex – This value is specified when creating the TLAS, as part of D3D12\_RAYTRACING\_INSTANCE\_DESC.

We will set these values as follows:

* MissStartAddress - the address of the second shader-table entry
* HitBaseIndex - the address of the third shader-table entry
* RayContributionToHitGroupIndex – The ray-index. 0 For the primary-ray, 1 for the shadow-ray. The simplest way to understand this is to look at the hit-program index computation mentioned above.
* MultiplierForGeometryContributionToShaderIndex – This only affects instances with multiple geometries. In our case, instance 0. This is the distance in records between geometries. In our case it’s the ray count (2).
* MissShaderIndex – Since our miss-shaders entries are stored contiguously in the shader-table, we can treat this value as the ray-index.
* InstanceContributionToHitGroupIndex
  + 0 for instance 0
  + 4 for instance 1
  + 6 for instance 2

# Code Changes

At this stage, you should be familiar enough with the code that we don’t need to go over it in much detail. Instead, we will point to the location of the changes.

## 13-SecondaryRayType.cpp

* Change the value of InstanceContributionToHitGroupIndex for each instance in **createTopLevelAS()**
* **createShaderTable()** – Create a larger shader-table with 11 entries and set the records based on the layout above.
* When calling **DispatchRays()**, use the new miss and hit programs’ parameters and shader-table sizes.

## Ray-Gen Shader

* Pass “2” as the MultiplierForGeometryContributionToShaderIndex when calling **rtTrace()**.
* The RayContributionToHitGroupIndex and MissShaderIndex stay 0. In effect is the ray-index and we’d like to trace a primary ray.

## Plane CHS

We completely reimplemented **planeChs()**. Let’s go over the code.

void planeChs(inout Raypayload payload, in IntersectionAttribs attribs)

The function signature stayed the same. Next, we need to fetch the hit-point properties using the following intrinsics:

float hitT = **RayTCurrent()**;

float3 rayDirW = **WorldRayDirection()**;

float3 rayOriginW = **WorldRayOrigin()**;

* hitT – The parametric distance along the ray direction between the ray’s origin and the intersection point.
* rayDirW – The world-space direction of the incoming ray. This is the value that was passed to **TraceRay()** by the ray-gen shader.
* rayOriginW – The world-space origin of the incoming ray. This is the value that was passed to **TraceRay()** by the ray-gen shader.

We start by finding the world-space position of the intersection point. This value is the origin of the new shadow-ray.

float3 posW = rayOriginW + hitT \* rayDirW;

RayDesc ray;

ray.Origin = posW;

We simulate a directional light, so we use a constant direction for the shadow-ray.

ray.Direction = normalize(float3(0.5, 0.5, -0.5));

We then set the ray’s extents. Note that we do not use 0 for **TMin** but set it into a small value. This is to avoid aliasing issues due to floating-point errors.

ray.TMin = 0.01;

ray.TMax = 100000;

Now we can trace the ray

ShadowPayload shadowPayload;

TraceRay(gRtScene, 0, 0xFF, 1 /\* ray index\*/, 0, 1, ray, shadowPayload);

Note that we set RayContributionToHitGroupIndex and MissShaderIndex to the same value(‘1’), which is the ray-index.

The result of this TraceRay() call will be used to compute the intersection point’s color.

float factor = shadowPayload.hit ? 0.1 : 1.0;

payload.color = float4(0.9f, 0.9f, 0.9f, 1.0f) \* factor;

Now that the coding is done, we can launch our application and see some shadows on the plane.

