DXR Tutorial 10

Per-Instance Constant Buffer

# Overview

In the previous tutorial we learned how to bind a constant-buffer to the ray-tracing pipeline. The same constant-buffer was shared between the instances.

DXR provides a mechanism to bind different resources to different instances of the same geometry. As you recall, we bind the resources into a root-table that is part of the shader-table entry. By creating an shader-table entry per instance and understanding how shader-table indexing works, we can use different resource for each instance.

Before we get to it, let’s get the simple things out of the way.

# Closest-Hit Shader

Since we are using a different resource for each instance, we do not need to use **InstanceID**() anymore. We also change the constant-buffer to store a single set of vertex colors.

The code can be found in *’10-Shaders.hlsl’*:

cbuffer PerFrame : register(b0)

{

float3 A;

float3 B;

float3 C;

};

struct IntersectionAttribs

{

float2 baryCrd;

};

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);

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

}

There is no need to change the hit-program creation. The root-signature stays the same – a single root-descriptor for the CBV.

# Constant-Buffers

We now need to create 3 constant-buffer – one per instance. This is straightforward DX12 code and we will not go over it. You can see it in **createConstantBuffers()**.

# Shader-Table Indexing

To understand the changes required to support per-instance resources, we need to understand how the ray-tracing pipeline decides which shader-table record to use when executing programs. The indexing computation is different depending on the **type** of the program but they share several common elements specified.

* 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.

## Ray-Generation Program

This is an easy one. It’s the entry pointed by the StartAddress.

## Miss Program

The entry is (MissStartAddress + MissShaderIndex \* MissStrideInBytes).

## Hit Program

The entry index is

entryIndex =

InstanceContributionToHitGroupIndex +

GeometryIndex\* MultiplierForGeometryContributionToShaderIndex +

RayContributionToHitGroupIndex)

And the entry address is (HitStartAddress + entryIndex \* HitStrideInBytes)

There are 2 new elements here:

**InstanceContributionToHitGroupIndex**– This value is specified when creating the TLAS, as part of D3D12\_RAYTRACING\_INSTANCE\_DESC.

**GeometryIndex** – As you might recall, when creating a bottom-level acceleration structure we can specify multiple geometries by passing multiple D3D12\_RAYTRACING\_GEOMETRY\_DESC. The GeometryIndex is the index of the geometry inside the bottom-level acceleration structure. In our case, we have a single geometry so this value is always 0.

This indexing scheme allows some flexibility in the way the shader-table records are laid out. See the DXR specification for examples.

In our case, we will go with the following simple layout:

**Hit**

**Instance 2**

**Hit**

**Instance 0**

**Hit**

**Instance 1**

**Miss**

**RayGen**

This only requires us to change the **InstanceContributionToHitGroupIndex** field of D3D12\_RAYTRACING\_INSTANCE\_DESC structure when creating the TLAS.

We have no real use for either **MultiplierForGeometryContributionToShaderIndex** or **RayContributionToHitGroupIndex**. We will set both to zero.

You can see the change in line 386 (**createTopLevelAS()**). `i` is the instance index, in the range [0,3).

instanceDescs[i]. InstanceContributionToHitGroupIndex = i;

# Shader Table

The first thing we need to change in the shader-table size. We now need 5 entries. This affect the shader-table size we calculate (line 604):

uint32\_t shaderTableSize = mShaderTableEntrySize \* 5;

Finally, we need to initialize the 3 hit-program entries (lines 811-818):

// Entries 2-4 - The triangles' hit program. ProgramID and constant-buffer data

for (uint32\_t i = 0; i < 3; i++)

{

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

*memcpy*(pHitEntry, pRtsoProps->GetShaderIdentifier(kHitGroup), progIdSize);

uint8\_t\* pCbDesc = pHitEntry + progIdSize; // The location of the root-descriptor

assert(((uint64\_t)pCbDesc % 8) == 0); // Root descriptor must be stored at an 8-byte aligned address

\*(D3D12\_GPU\_VIRTUAL\_ADDRESS\*)pCbDesc = mpConstantBuffer[i]->GetGPUVirtualAddress();

}

This code is very similar to the code from tutorial 9. We calculate the address of the hit entry, then set the program identifier and the constant-buffer address of the current instance.

The last part of the code we need to change is in onFrameRender(). The Hit-Group table contains 3 entries, so we set raytraceDesc.HitGroupTable.*SizeInBytes* to mShaderTableEntrySize \* 3.

In our case, those changes result in the same image as in tutorial 9. However, now that we understand shader-table indexing we can start covering more advanced usages. We’ll get to that in the next tutorial.

