DXR Tutorial 09

Constant Buffers

# Overview

In the previous tutorial we computed the hit-point colors based on constants defined in the shader. In this tutorial we will learn how to use constant-buffers with DXRT and use one to get the vertex colors from.

We already learned everything we need to know for working with constant-buffers. This tutorial is more of an exercise - feel free to try adding constant-buffer support all by yourself.

Let’s dive directly into the code.

# Closest-Hit Shader

Let’s start with modifying the shader. The changes are straightforward – we start by adding a constant-buffer definition.

cbuffer PerFrame : register(b0)

{

float3 A[3];

float3 B[3];

float3 C[3];

}

As you can see, we have 3 sets of vertex colors, one per triangle. We then use the instanceID to fetch the colors from the buffer and compute the result.

void chs(inout RayPayload payload, in BuiltInTriangleIntersectionAttributes attribs)

{

uint instanceID = InstanceID();

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

attribs. barycentrics.x,

attribs. barycentrics.y);

payload.color = A[instanceID] \* barycentrics.x +

B[instanceID] \* barycentrics.y +

C[instanceID] \* barycentrics.z;

}

# Modifying the RTPSO Creation

Up until now, we created the hit-program with an empty root-signature. Now that the closest-hit shader requires a constant-buffer, we need a different root-signature.

We will create a root-signature with a single entry – a root-descriptor for a Constant-Buffer View (CBV). If you’ll take a look at **createRtPipelineState**(), you’ll see that we added 2 new sub-objects:

* A LocalRootSignature for the hit-program created by calling **createHitRootDesc().**
* An ExportAssociation that associates the hit-program local root-signature to the hit-group.

We also removed the hit-group name from the empty root-signature association.

# Creating the Constant-Buffer

Creating the constant-buffer is done the same way as for rasterization. This happens in **createConstantBuffer()** and will not be explained here.

If you look at the **onload()** function, you’ll notice that we create the constant-buffer **before** we create the shader-table. That’s because we need the constant-buffer GPU address in hand when initializing the shader-table.

# The Shader Table

There are potentially 2 modifications we need to make in **createShaderTable()**. The first one is obvious – we need to set the CBV into the root-table. The other is subtler – we need to modify the shader-table record size.

Remember that all shader-table records share the same size. The size we chose was based on the largest required root-table size. Up until now, the ray-generation shader required the largest table. Its root-signature had a single descriptor-table, which is 8 bytes.

Now the closest-hit shader uses a root-descriptor, but luckily for us a root-descriptor is 8 bytes, so our shader-table record size can stay the same.

Finally, we need to set the constant-buffer address into the root-table

// 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), progIdSize);

uint8\_t\* pCbDesc = pHitEntry + progIdSize; // Adding `progIdSize` gets us to the location of the CB entry

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

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

The first line skips the ray-gen and miss program entries. We then skip the program identifier to get the address of the root-descriptor. We then set the constant-buffer GPU virtual address.

The spec requires root-descriptors to be aligned on an 8-byte address. The assertion in the code makes sure that this is the case.

And we’re done!

