DXR Tutorial 08

Instancing

# Overview

Now that we know how to invoke the ray-tracing pipeline, we can get into more advanced usages. We will start with something simple – instancing.

Instancing has 2 inputs:

1. Number of instances to render.
2. The transformation matrix for each instance.

In the rasterization API, we set those inputs by

1. Passing (InstanceCount > 1) to **DrawInstanced()** or **DrawIndexedInstanced()**.
2. Using *SV\_InstanceID* to control the transformation of each instance.

In DXR both inputs are set during the creation of the top-level acceleration-structure (TLAS).

# Acceleration Structures Revisited

Recall that we have 2 types of acceleration structures – top-level and bottom-level.

The bottom-level acceleration structure is the one that holds the geometric data – vertex and index buffers, strides, and vertex count. Conceptually, we can think of it as a mesh in local space.

The top-level acceleration structure then references the bottom-level acceleration structures we created. For each reference, we can optionally specify a local→world transformation matrix. Instancing is achieved by referencing the same bottom-level acceleration structure multiple times with different matrices.

# Code Walkthrough

We are going to modify our application to render 3 instances of the triangle.

There is no need to change the creation of the bottom-level acceleration structure. We only need to make a small change to the TLAS creation code - **createTopLevelAS**().

The first thing we need is to change the call to **GetRaytracingAccelerationStructurePrebuildInfo()**. We need to request the information for 3 instances – specified by the NumDescs field of D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC.

Then, we need to change the size of the D3D12\_RAYTRACING\_INSTANCE\_DESC buffer. We need size for 3 descriptors.

buffers.pInstanceDesc = createBuffer(pDevice, sizeof(D3D12\_RAYTRACING\_INSTANCE\_DESC) \* **3**, *D3D12\_RESOURCE\_FLAG\_NONE*, *D3D12\_RESOURCE\_STATE\_GENERIC\_READ*, kUploadHeapProps);

Next, let’s create the transformation matrices.

mat4 transformation[3];

transformation[0] = mat4(); // Identity

transformation[1] = translate(mat4(), vec3(-2, 0, 0));

transformation[2] = translate(mat4(), vec3(2, 0, 0));

Now we can go ahead and initialize the D3D12\_RAYTRACING\_INSTANCE\_DESC buffer.

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

{

instanceDescs[i].InstanceID = i;

instanceDescs[i].InstanceContributionToHitGroupIndex = 0;

instanceDescs[i].Flags = D3D12\_RAYTRACING\_INSTANCE\_FLAG\_NONE;

mat4 m = transpose(transformation[i]); // GLM is column major, the INSTANCE\_DESC is row major

*memcpy*(instanceDescs[i].Transform, &m, sizeof(instanceDescs[i].Transform));

instanceDescs[i].AccelerationStructure = pBottomLevelAS->GetGPUVirtualAddress();

instanceDescs[i].InstanceMask = 0xFF;

}

It’s very similar to the code we had before, but there are some things to note.

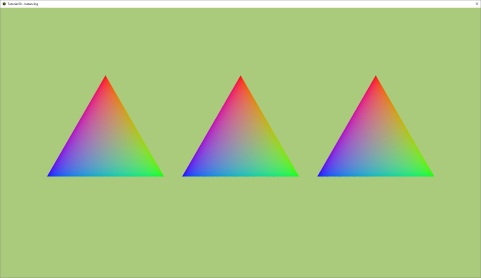
First, we set a different InstanceID per instance. It doesn’t have to be in sequential order (i.e. `i`). We can set it to whatever arbitrary value we want. The ray-tracing pipeline doesn’t use this value. It will be communicated to the hit-shader via the **InstanceID**() intrinsic.

Next, note that we are using the same InstanceContributionToHitGroupIndex. This means that we will use the same SBT record for all instances. That’s fine – we do not have any per-instance data in the hit-records.

Finally, we need to transpose our transformation matrix. This is an implementation detail – our math library uses column-major matrices while DRXT expects Transform in row-major format.

The last thing we need to change is the call to BuildRaytracingAccelerationStructure(). We need to set the NumDescs field of D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC to 3.

And we’re good to go. No other changes are required, we can run the application and see this image.

****

# InstanceID()

Actually, if you run the tutorial code you’ll see a different image then the one above. That’s because we also made a small change to the closest-hit shader (*08-Shaders.hlsl).* At the beginning of the shader, you can see the following line

uint instanceID = InstanceID();

This value will receive the value we specified when we created the TLAS (D3D12\_RAYTRACING\_INSTANCE\_DESC::InstanceID).

Based on this value we change the color interpolation order. The result is 3 different looking triangles.

