DXR Tutorial 03

Acceleration Structure

# Overview

Now that we can clear the screen, it’s time to render something. The following 5 tutorials will do just that – we will write some code that will use raytracing to render a triangle to the screen.

The first thing we need to create are acceleration structures. An acceleration structure is an opaque data structure that represents the scene’s geometry. This structure is used in rendering time to intersect rays against. For more information on it and optimized usage please refer to the spec. In this tutorial we will focus on how to create it.

# The Whole Story

Most of the action happens inside **createAccelerationStructures().**It’s a new function we added which is called from **onLoad()**.

The first line of code there is

* mpVertexBuffer = createTriangleVB(mpDevice);

This is a standard triangle vertex-buffer, created using the regular DX12 API and so we will not go into details. The only thing to note is that we allocate buffer on the upload heap, but that’s just for convenience as it simplifies the code.

Next, we will create the bottom-level acceleration structure

* AccelerationStructureBuffers bottomLevelBuffers = createBottomLevelAS(mpDevice, mpCmdList, mpVertexBuffer);

# Bottom-Level Acceleration Structure

The BLAS is a data structure that represent a local-space mesh. It does not contain information regarding the world-space location of the vertices or instancing information.

The first thing in creating it is initializing a D3D12\_RAYTRACING\_GEOMETRY\_DESC struct:

D3D12\_RAYTRACING\_GEOMETRY\_DESC desc = {};

desc.Type = D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_TRIANGLES;

desc.Triangles.VertexBuffer.StartAddress = pVB->GetGPUVirtualAddress();

desc.Triangles.VertexBuffer.StrideInBytes = sizeof(vec3);

desc.Triangles.VertexFormat = DXGI\_FORMAT\_R32G32B32\_FLOAT;

desc.Triangles.VertexCount = 3;

desc.Flags = D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE;

We first set the type to D3D12\_RAYTRACING\_GEOMETRY\_TYPE\_TRIANGLES. This implies we will be using the built-in triangle intersection shader, but we will get to what that exactly means in tutorial 7.

Next, we set the GPU virtual address of the vertex-buffer.

The next 3 fields are equivalent to an input element layout descriptor. They describe the vertex stride, the offset of the position element inside the vertex and the position format. We only have a single element in our VB, which is the position, meaning VertexByteOffset equals 0. Each vertex is exactly 3 floats, and that’s the size and format of the vertex.

Next, we will set the number of vertices in the buffer. We only have 3.

The Flags field allows us to control some aspects of the acceleration structure. In this case, we know that the triangle is not transparent and so we set the D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE flag.

The spec recommends using this flag as much as possible. We will get to what this flag means exactly in tutorial 7.

Now that we are done with the descriptor, let’s create the buffer. As you know, in DX12, resource allocation and lifetime management is the user’s responsibility. DXR is no different in this regard – it will not allocate buffers for us, not even internal temporary buffers required during acceleration structure creation.

DXR requires 2 buffers:

1. Scratch buffer which is required for intermediate computation.
2. The result buffer which will hold the acceleration data.

To allocate these buffers, we need to know the required size. This is done using the following snippet:

D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC prebuildDesc = {};

prebuildDesc.DescsLayout = D3D12\_ELEMENTS\_LAYOUT\_ARRAY;

prebuildDesc.Flags = D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_NONE;

prebuildDesc.NumDescs = 1;

prebuildDesc.pGeometryDescs = &geomDesc;

prebuildDesc.Type = D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE\_BOTTOM\_LEVEL;

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO info;

ID3D12DeviceRaytracingPrototypePtr pRtDevice = pDevice;

pRtDevice->GetRaytracingAccelerationStructurePrebuildInfo(&prebuildDesc, &info);

We first initialize a D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC struct:

* DescsLayout – We are using an array, so the layout is D3D12\_ELEMENTS\_LAYOUT\_ARRAY.
* Flags – This field should match the value we will later use for building the acceleration structures. In our case we don’t use any special flags.
* The next 2 fields are the number of descriptors and the pointer to the descriptor array (in our case the array size is 1)
* Type – The type of the acceleration structure we are going to generate, bottom-level in our case.

Next, we need to call GetRaytracingAccelerationStructurePrebuildInfo() function. This function is not part of the regular ID3D12Device interface, but rather of ID3D12DeviceRaytracingPrototype. A simple assignment operator will convert the pointer, followed by a call to the function.

Once we get the information we can allocate the buffers:

buffers.pScratch = createBuffer(pDevice, info.ScratchDataSizeInBytes, *D3D12\_RESOURCE\_FLAG\_ALLOW\_UNORDERED\_ACCESS*, D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS, kDefaultHeapProps);

buffers.pResult = createBuffer(pDevice, info.ResultDataMaxSizeInBytes, *D3D12\_RESOURCE\_FLAG\_ALLOW\_UNORDERED\_ACCESS*, D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE, kDefaultHeapProps);

The buffers are allocated on the default heap, since we don’t need read/write access to them. We also create them with the *D3D12\_RESOURCE\_FLAG\_ALLOW\_UNORDERED\_ACCESS* (not mentioned in the spec but probably required). The spec requires the state of the buffers to be:

* D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS for the scratch buffer.
* D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE for the destination buffer.

Now that we have everything we need, we can create the acceleration structure. We start by initializing the AS descriptor. The values we set here must match the values used when calling GetRaytracingAccelerationStructurePrebuildInfo().

// Create the bottom-level AS

D3D12\_BUILD\_RAYTRACING\_ACCELERATION\_STRUCTURE\_DESC asDesc = {};

asDesc.DescsLayout = D3D12\_ELEMENTS\_LAYOUT\_ARRAY;

asDesc.pGeometryDescs = &geomDesc;

asDesc.DestAccelerationStructureData.StartAddress = buffers.pResult->GetGPUVirtualAddress();

asDesc.DestAccelerationStructureData.SizeInBytes = info.ResultDataMaxSizeInBytes;

asDesc.Flags = D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_NONE;

asDesc.NumDescs = 1;

asDesc.ScratchAccelerationStructureData.StartAddress = buffers.pScratch->GetGPUVirtualAddress();

asDesc.ScratchAccelerationStructureData.SizeInBytes = info.ScratchDataSizeInBytes;

asDesc.Type = D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE\_BOTTOM\_LEVEL;

The first 2 specify we are passing in an array of geometry descriptors and the location of the array.

This if followed by the GPU VA and size of the **destination** AS.

The Flags are set to none, we have 1 geometry descriptor in the array, we set the GPU VA and size of the scratch buffer. Lastly, we set the type of the requested AS.

Now that we have a descriptor ready, we can record a command.

ID3D12CommandListRaytracingPrototypePtr pRtCmdList = pCmdList;

pRtCmdList->BuildRaytracingAccelerationStructure(&asDesc);

Notice that there’s a new type of command-list interface - ID3D12CommandListRaytracingPrototype. We can simply cast it from our regular command-list by using the assignment operator.

Calling BuildRaytracingAccelerationStructure() will record a command into the list. **This command will note be processed until we submit the command list**, so make sure the scratch-buffer will not be released until execution finishes.

In the next section we will use the BLAS as an input for another BuildRaytracingAccelerationStructure() operation. We need to make sure that the write operation will finish before reading data from the result buffer. We do that using a regular UAV-barrier.

D3D12\_RESOURCE\_BARRIER uavBarrier = {};

uavBarrier.Type = D3D12\_RESOURCE\_BARRIER\_TYPE\_UAV;

uavBarrier.UAV.pResource = buffers.pResult;

pCmdList->ResourceBarrier(1, &uavBarrier);

# Top-Level Acceleration Structure

The TLAS is an opaque data structure that represents the entire scene. As you recall, BLAS represents objects in local space. The TLAS references the bottom-level structures, with each reference containing local-to-world transformation matrix.

Let’s take a look at **createTopLevelAS().**

Like bottom-level AS creation, we need to create the result and scratch buffers. The code is very similar, the only difference is how we query the required sizes. This happens in the following snippet:

D3D12\_GET\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO\_DESC prebuildDesc = {};

prebuildDesc.DescsLayout = D3D12\_ELEMENTS\_LAYOUT\_ARRAY;

prebuildDesc.Flags = D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_BUILD\_FLAG\_NONE;

prebuildDesc.NumDescs = 1;

prebuildDesc.Type = D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE\_TOP\_LEVEL;

D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_PREBUILD\_INFO info;

ID3D12DeviceRaytracingPrototypePtr pRtDevice = pDevice;

pRtDevice->GetRaytracingAccelerationStructurePrebuildInfo(&prebuildDesc, &info);

The only difference is the Type field – we are requesting information for creating a TLAS.

Next, we will create the scratch and result buffer. Nothing new here.

Now we can proceed to describe the instances used for the TLAS. We do that by filling a buffer of D3D12\_RAYTRACING\_INSTANCE\_DESC. We pass an array of such descriptors to the **BuildRaytracingAcceleration()** function. This array describes the scene.

The first thing to know about this array, is that it can’t simply reside on the regular C++ heap. We need to pass this array to **BuildRaytracingAcceleration()** in a GPU buffer (either on the upload or default heap). Since it’s a DX resource accessed by the GPU, all the regular synchronization and lifetime management rules apply.

We only have a single instance, so we create a buffer with that size, then map it to write. Next, we will initialize it.

pInstanceDesc->InstanceID = 0;

pInstanceDesc->InstanceContributionToHitGroupIndex = 0;

pInstanceDesc->Flags = D3D12\_RAYTRACING\_INSTANCE\_FLAG\_NONE;

mat4 m;

*memcpy*(pInstanceDesc->Transform, &m, sizeof(pInstanceDesc->Transform));

pInstanceDesc->AccelerationStructure = pBottomLevelAS->GetGPUVirtualAddress();

The first field is InstanceID. It doesn’t affect raytracing at all, and the runtime ignores it while tracing rays. It’s simply a user-defined value that will communicated to the shader via the **InstanceID()** intrinsic.

InstanceContributionToHitGroupIndex is the offset of the instance inside the shader-binding-table. Let’s set it to 0 for now. This value will be explained in tutorial 5.

There are numerous options for the Flags. Refer to the spec for more details, in the tutorial we will just set it to D3D12\_RAYTRACING\_INSTANCE\_FLAG\_NONE.

Next is the transformation matrix. It’s a 3x4 affine transform matrix in row-major layout. This transformation will be applied to each vertex in the bottom-level structure. In this case, we are setting an identity matrix. This value can also be nullptr, which is equivalent to setting an identity matrix, but may result in better performance.

The last field – AccelerationStructure – is the GPU virtual address of the bottom-level acceleration structure containing the vertex data.

After we finish the initialization, we can unmap the desc-buffer and call **BuildRaytracingAccelerationStructure().** The code is almost identical to the one used to create the BLAS, except:

1. We need to set the instance-descriptor buffer GPU VA into the InstanceDescs field.
2. We need to set the Type field to D3D12\_RAYTRACING\_ACCELERATION\_STRUCTURE\_TYPE\_TOP\_LEVEL

Just as we did for the BLAS, we need to insert a UAV barrier for the result buffer. This step is required because we need to make sure that the write operation performed in **BuildRaytracingAccelerationStructure**() finishes before the read operation in **DispatchRays**() (will be shown in tutorial 6).

# Back to createAccelerationStructures()

We created some buffers and recorded commands to create bottom-level and top-level acceleration structures. We now need to execute the command-list. To simplify resource lifetime management, we will submit the list and wait until the GPU finishes its execution. This is not required by the spec – the list can be submitted whenever if the resources are kept alive until execution finishes.

The last part is releasing resources that are no longer required and keep references to the resources which will be used for rendering.

Remember that we are using smart COM-pointers, so keeping reference is as simple as storing a copy of the smart-pointer. This happens in the following code

mpTopLevelAS = topLevelBuffers.pResult;

mpBottomLevelAS = bottomLevelBuffers.pResult

Note that we need to store both top-level and bottom-level structures. The scratch buffers and the instance-desc buffers will be released automatically once the local variable holding their smart pointer goes out of scope.

And that’s it! We have acceleration structures, which means one major concept of DXRT is behind us!