Introduction to DirectX Raytracing

Part 2 – the API

Shawn Hargreaves
Microsoft

@shawnhargreaves
How To Use DXR, In Three Easy Steps

1. DispatchRays()
2. Configure a raytracing pipeline state
3. Build geometry into acceleration structures
... backing up: D3D12 basics

- D3D12 is a low level API, so you control memory allocations, object lifespans, and synchronization
- Create resources in GPU memory (textures, vertices, constant buffers)
- Copy data into them
- Record GPU instructions into a command list
  ```
  ID3D12GraphicsCommandList::Reset
  ID3D12GraphicsCommandList::SetGraphicsRootSignature
  ID3D12GraphicsCommandList::DrawInstanced
  ID3D12GraphicsCommandList::Dispatch
  ```
- Submit command lists to a queue for execution by the hardware
  ```
  ID3D12CommandQueue::ExecuteCommandLists
  ```
- Synchronize to know when work has finished
  ```
  ID3D12CommandQueue::Signal
  ID3D12Fence::GetCompletedValue
  ID3D12Fence::SetEventOnCompletion
  ```
Parallel Execution – Creating a Resource

Create buffer
Create upload heap
Write data to upload heap
CopyResource(upload -> buffer)
ExecuteCommandLists
queue->Signal(fence)
fence->SetEventOnCompletion
WaitForSingleObject(event)
Destroy upload heap

CopyResource
Signal
ExecuteCommandLists
D3D12 Binding Model (indirection ftw!)

- Descriptor = pointer to a GPU resource
- Descriptor table = indexable array of descriptors
- Descriptor heap = area of GPU memory containing multiple descriptor tables
- Root signature defines a binding convention, used by shaders to locate whatever data they need to access
  - Inlined root constants
  - Inlined root descriptors
  - Pointers to descriptor tables within the descriptor heap
New Requirements For Raytracing

• Acceleration structure format is opaque
  • Unlike traditional vertex data used for rasterization, there is no standard layout suitable for all implementations

• Rays can go anywhere
  • So all geometry and shaders must be simultaneously available

• Different shaders may want different resource bindings

More levels of indirection!
Build Acceleration Structures
Acceleration Structures

- Opaque geometry format optimized for ray traversal (e.g. BVH)
- Layout determined by driver and hardware
- Built at runtime on the GPU
- Immutable except for incremental in-place updates
Top vs. Bottom

• Bottom level acceleration structures
  • Contain triangles
  • Or procedural primitives defined by AABB plus intersection shader
  • Take time to build

• Top level acceleration structures
  • Point to one or more bottom level acceleration structures
  • Instancing = multiple references to the same bottom level structure
  • Fast to build

• Balance between:
  • Ray intersection performance: larger bottom level structures
  • Flexibility: more/smaller bottom level structures
Memory Management

- Because the format is implementation defined, you cannot know up front how big an acceleration structure will be

GetRaytracingAccelerationStructurePrebuildInfo()  
  - Runs on the CPU  
  - Returns a conservative estimate  
  - D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO
    .ResultDataMaxSizeInBytes  
    .ScratchDataSizeInBytes

BuildRaytracingAccelerationStructure()  
  - Runs on the GPU  
  - Returns actual size, in GPU memory after the command list has finished executing  
  - D3D12_RAYTRACING_ACCELERATION_STRUCTURE_POSTBUILD_INFO_TYPE
    _CURRENT_SIZE  
    _COMPACTED_SIZE
Acceleration Structure Build Flags

- D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG
  - _PREFER_FAST_TRACE
  - _PREFER_FAST_BUILD
  - _MINIMIZE_MEMORY
  - _ALLOW_COMPACTION
  - _ALLOW_UPDATE
  - _PERFORM_UPDATE
Compaction

• Suballocate out of larger buffers
• Use conservative sizes while generating command list for initial build
• After real size data is available, perform a compaction pass
• Don’t compact things that animate...
• Beware CPU/GPU stalls!
No Compaction

1. GetRaytracingAccelerationStructurePrebuildInfo()
2. Allocate _PREBUILD_INFO.ResultDataMaxSizeInBytes
3. BuildRaytracingAccelerationStructure()

Wasted space:

_PREBUILD_INFO.ResultDataMaxSizeInBytes - _POSTBUILD_INFO_COMPACTED_SIZE
In-place Compaction

Allocate multiple acceleration structures at offsets within larger buffer:

1. BuildRaytracingAccelerationStructure to (buffer + bufferOffset)
2. Wait for build to finish executing
3. bufferOffset += _POSTBUILD_INFO_CURRENT_SIZE

Wasted space:

_POSTBUILD_INFO_CURRENT_SIZE - _POSTBUILD_INFO_COMPACTED_SIZE

Beware stalls!
Full Compaction

1. For each acceleration structure:
   • `BuildRaytracingAccelerationStructure` to `(tempBuffer + tempOffset)`
   • `tempOffset += _PREBUILD_INFO.ResultDataMaxSizeInBytes`

2. Wait for builds to finish executing

3. For each acceleration structure:
   • `CopyRayTracingAccelerationStructure(buffer + bufferOffset, tempBufferLocation, _ACCELERATION_STRUCTURE_COPY_MODE_COMPACT)`
   • `bufferOffset += _POSTBUILD_INFO_COMPACTED_SIZE`

• No wasted space
• Starting point for more sophisticated dynamic allocation schemes
Animation

• Rigid body animation
  • Update just the top level structure
  • Change transform matrices
  • Add or remove bottom level references as required

• Skinned animation
  • Transform vertices into a buffer (e.g. using compute shader)
  • Apply new positions as an incremental update to an existing bottom level acceleration structure
  • Performance degrades the more things have moved
    • Skin in object space, not world space!
    • Periodic full rebuild after significant deformations?
Instance Masking

• 8 bit mask per top level acceleration structure entry
• TraceRay intrinsic takes an instance mask parameter
• Intersection only processed if
  \[(\text{TraceRay mask} \& \text{acceleration structure mask}) \neq 0\]
• For all rays to intersect everything, set both masks to \(0xFF\)
• Allows different rays to process different geometry, without having to build entirely separate top level acceleration structures
• e.g. one of the bits could represent “casts shadow”
Configure Raytracing
Pipeline State
Shader Tables

• Rays can go anywhere and hit anything
• Different objects can have different materials
• Need to run different shaders depending on which object a ray hit

• Solution:
  • Array of pointers to shaders
  • Index into the array is determined by which object was hit
Arrays of Pointers to Shaders

- Shader Identifier = ‘pointer’ to a shader (32 byte blob)
- Hit Group = { intersection shader, any hit shader, closest hit shader }
- Shader Record = { shader identifier, local root arguments }
- Shader Table = { shader record A }, { shader record B }, ...

- No dedicated API for creating shader tables
  - These are just memory that can be filled however you like
# Example – HitGroup Shader Table

<table>
<thead>
<tr>
<th>Shader Record #1 - Terrain</th>
<th>Shader Identifier (32 bytes)</th>
<th>Root Arguments</th>
<th>Pad to 32 byte alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hit Group Identifier</td>
<td>DescriptorHandle terrainTextureTable; (8 bytes)</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DescriptorHandle lightmapTable; (8 bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RootDescriptor constantBuffer; (8 bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>float fogDistance; (4 bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intersection = null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hit = null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Closest hit = shadeTerrain</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shader Record #2 - Water</th>
<th>Shader Identifier (32 bytes)</th>
<th>Root Arguments</th>
<th>Pad to 32 byte alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hit Group Identifier</td>
<td>DescriptorHandle animTexTable; (8 bytes)</td>
<td>20 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int currentTextureFrame; (4 bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intersection = null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hit = waterAny</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Closest hit = waterClosest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shader Record #3 - Volumetric Clouds</th>
<th>Shader Identifier (32 bytes)</th>
<th>Root Arguments</th>
<th>Pad to 32 byte alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hit Group Identifier</td>
<td>RootDescriptor constantBuffer; (8 bytes)</td>
<td>16 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RootDescriptor volumeTexture; (8 bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intersection = cloudIntersection</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hit = cloudAny</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Closest hit = null</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shader Table Indexing

• DispatchRays provides base address and stride for each of:
  • MissShaderTable
  • HitGroupTable
  • CallableShaderTable

• TraceRay intrinsic parameters:
  • RayContributionToHitGroupIndex
  • MultiplierForGeometryContributionToHitGroupIndex
  • MissShaderIndex

• Top level acceleration structure provides:
  • D3D12_RAYTRACING_INSTANCE_DESC.InstanceContributionToHitGroupIndex
Miss Shader Indexing

MissShaderRecordAddress =

\[ \text{start} + \text{stride} \times \text{index} \]

where:

- \( \text{start} = \text{D3D12 DISPATCH_RAYS_DESC.MissShaderTable.StartAddress} \)
- \( \text{stride} = \text{D3D12 DISPATCH_RAYS_DESC.MissShaderTable.StrideInBytes} \)
- \( \text{index} = \text{MissShaderIndex (TraceRay parameter)} \)
Hit Group Indexing

HitGroupRecordAddress =

\[
\text{start} + \text{stride} \times (\text{rayContribution} + \\
(\text{geometryMultiplier} \times \text{geometryContribution}) + \text{instanceContribution})
\]

where:

- \text{start} = \text{D3D12\_DISPATCH\_RAYS\_DESC.HitGroupTable.StartAddress}
- \text{stride} = \text{D3D12\_DISPATCH\_RAYS\_DESC.HitGroupTable.StrideInBytes}
- \text{rayContribution} = \text{RayContributionToHitGroupIndex (TraceRay parameter)}
- \text{geometryMultiplier} = \text{MultiplierForGeometryContributionToHitGroupIndex (TraceRay parameter)}
- \text{geometryContribution} = \text{index of geometry in bottom-level acceleration structure (0,1,2,3...)}
- \text{instanceContribution} = \text{D3D12\_RAYTRACING\_INSTANCE\_DESC.InstanceContributionToHitGroupIndex}
Compile and Link Shaders

• Offline
  • HLSL -> dxc.exe -> DXIL library

• At runtime
  • Load DXIL library code
  • Create subobjects and set associations
  • CreateStateObject(D3D12_STATE_OBJECT_TYPE_RAYTRACING_PIPELINE)
  • Look up shader identifiers
DXIL library

**MyRaygenShader**

**MyClosestHitShader**

**MyMissShader**

**HIT_GROUP_SUBOBJECT**
- Type = TRIANGLES
- IntersectionShader = null
- AnyHitShader = null
- ClosestHitShader = MyClosestHitShader

**SHADER_CONFIG_SUBOBJECT**
- PayloadSize = 16
- AttributeSize = 8

**LOCAL_ROOT_SIGNATURE_SUBOBJECT**
- RootSignature = xxx

**SUBOBJECT_TO_EXPORTS_ASSOCIATION_SUBOBJECT**

- Export = MyRaygenShader
- Subobject = LOCAL_ROOT_SIGNATURE_SUBOBJECT

**CreateStateObject()**

- GetShaderIdentifier(L"MyRaygenShader")
- GetShaderIdentifier(L"MyHitGroup")
- GetShaderIdentifier(L"MyMissShader")

DispatchRays()
void DispatchRays(D3D12_DISPATCH_RAYS_DESC* pDesc);

struct D3D12_DISPATCH_RAYS_DESC
{
  D3D12_GPU_VIRTUAL_ADDRESS_RANGE RayGenerationShaderRecord;
  D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE MissShaderTable;
  D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE HitGroupTable;
  D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE CallableShaderTable;
  UINT Width;
  UINT Height;
  UINT Depth;
};

struct D3D12_GPU_VIRTUAL_ADDRESS_RANGE
{
  D3D12_GPU_VIRTUAL_ADDRESS StartAddress;
  UINT64 SizeInBytes;
};

struct D3D12_GPU_VIRTUAL_ADDRESS_RANGE_AND_STRIDE
{
  D3D12_GPU_VIRTUAL_ADDRESS StartAddress;
  UINT64 SizeInBytes;
  UINT64 StrideInBytes;
};
Other Things You Should Know
Can haz plz? (for developers)

Experimental Preview
- Requires Windows 10 April 2018 update with suitable hardware and drivers
- Minor differences from final design

Official Release
- Coming in the next version of Windows (RS5)

Raytracing Fallback Layer

Sample code
D3D12_FEATURE_DATA_D3D12_OPTIONS5 featureData{};

if (SUCCEEDED(CheckFeatureSupport(D3D12_FEATURE_D3D12_OPTIONS5, &featureData, sizeof(featureData))))
{
    if (featureData.RaytracingTier >= D3D12_RAYTRACING_TIER_1_0)
    {
        ...
    }
}

Raytracing Fallback Layer

- Emulates the DXR API via compute shaders
  - Runs on systems without native DXR driver support
  - Requires Win10 Fall Creators Update (2017) and a GPU with retail DXIL capability

- Redirects to the real DXR implementation if driver support exists

- Standalone static lib

- Good way to get started, but limited substitute for a DXR driver in terms of performance
Compute shaders do not support raw GPU memory pointers
But the BuildRaytracingAccelerationStructure input structures are full of pointers...
Emulated GPU Pointers replace a GPU VA with descriptor heap index and byte offset pair
Apps must create views around memory they will access for raytracing

```c
UINT descriptorHeapIndex = 0;

if (!fallbackDevice->UsingRaytracingDriver())
{
    D3D12_CPU_DESCRIPTOR_HANDLE descriptorHeapHandle;
    descriptorHeapIndex = AllocateDescriptor(&descriptorHeapHandle);
    device->CreateUnorderedAccessView(resource, nullptr, desc, descriptorHeapHandle);
}

WRAPPED_GPU_POINTER ptr = fallbackDevice->GetWrappedPointerSimple(
    descriptorHeapIndex,
    resource->GetGPUVirtualAddress());
```

#### Pipeline

<table>
<thead>
<tr>
<th>Record</th>
<th>Layout</th>
<th>Root Signature</th>
<th>Root Argument Values</th>
<th>Table Offset (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>△</td>
<td>MyHitGroup1</td>
<td>ab1#53</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>△</td>
<td>MyHitGroup2</td>
<td>ab1#55</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Root Constant</td>
<td>Perem 0</td>
<td>9876</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Padding (4 bytes)</td>
<td></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Root CBV</td>
<td>Perem 1</td>
<td>res436</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Root SRV</td>
<td>Perem 2</td>
<td>&lt;invalid descriptor&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Root UAV</td>
<td>Perem 3</td>
<td>res437</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Descriptor Table</td>
<td>Perem 4</td>
<td>104</td>
</tr>
<tr>
<td>2</td>
<td>△</td>
<td>MyHitGroup3</td>
<td>ab1#53</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>&lt;invalid identifier&gt;</td>
<td>&lt;none&gt;</td>
<td>192</td>
</tr>
<tr>
<td>4</td>
<td>△</td>
<td>MyHitGroup4</td>
<td>ab1#54</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Root Constant</td>
<td>Perem 0</td>
<td>45</td>
</tr>
</tbody>
</table>

### Info (Shader Table)

- **General Info**
  - Shader Table Size: 320 bytes
  - Shader Record Size: 64 bytes
  - Shader Identifier Size: 16 bytes
  - Largest Record Size: 56 bytes
- **Root Constant Format**
  - □ unit (decimal)
  - ○ unit (hex)
  - □ float
  - □ signed int
Debugging DXR – http://blogs.msdn.com/pix