Author: Ionut Nedelcu

Ray Tracing Support

Unity Engine Features

Mesh Sampling in Ray Tracing Shaders

The following is a technical document describing how vertex attributes are read and interpolated in HLSL code when authoring ray tracing shaders in Unity.

The target audience for this document is graphics engineers who develop ray tracing effects in Unity.

Overview

Unity 2019.3 introduced experimental ray tracing support using DirectX 12 graphics API and new shader types specific to DirectX Raytracing (DXR) are now supported.

In regular rasterization based rendering, a vertex shader will define its vertex data inputs using vertex shader semantics (e.g. POSITION, TEXCOORD0) and the GPU will read vertex attributes from the input vertex streams and convey the data to GPU registers. This is done in an opaque way and the shader author is not concerned with how the data is read. However, in ray tracing there is no such mechanism and the vertex and index data needs to be manually read from Mesh vertex streams and the vertex attribute values have to eventually be decoded.

When ray-triangle intersections are detected during acceleration structure traversal, “any hit shaders” if available and a “closest hit shader” will be invoked depending on the position of the intersection point relative to the origin of the ray. Typically, *PrimitiveIndex()* HLSL intrinsic is used to retrieve the index of the triangle within the mesh geometry inside the bottom-level acceleration structure instance. Using this value, one can read the 3 indices of the vertices forming the triangle that was hit by the ray and finally interpolate various vertex attributes that are needed for different purposes like lighting calculations using normals, reading textures using UVs, etc. Vertex attribute interpolation is described later in this document.

Important notes:

- Mesh sampling is only available in closest hit shaders and any hit shaders that are part of the shader passes within a Material shader (a *.shader file) used for rendering Meshes.
- There is no data duplication when using ray tracing. The same Mesh and Texture data is used for both regular rasterization and ray tracing.

- As of Unity 2021.2, only Mesh Renderers and Skinned Mesh Renderers are supported in ray tracing.

Built-in Shader Parameters

The vertex stream array and the index buffer along with mesh and vertex attributes information buffers can be accessed in HLSL code using the **UnityRayTracingMeshUtils.cginc** built-in include file which is part of the Unity Editor installation. The shader include file defines the following built-in structures and buffers:

```
#define kMaxVertexStreams 8

struct MeshInfo
{
    uint vertexSize[kMaxVertexStreams];
    uint baseVertex;
    uint vertexStart;
    uint indexSize;
    uint indexStart;
};

struct VertexAttributeInfo
{
    uint Stream;
    uint Format;
    uint ByteOffset;
    uint Dimension;
};

StructuredBuffer<MeshInfo>              unity_MeshInfo_RT;
StructuredBuffer<VertexAttributeInfo>   unity_MeshVertexDeclaration_RT;

ByteAddressBuffer                       unity_MeshVertexBuffers_RT[kMaxVertexStreams];
ByteAddressBuffer                       unity_MeshIndexBuffer_RT;
```

When executing ray tracing dispatches using RayTracingShader.Dispatch or CommandBuffer.DispatchRays commands, Unity will bind the appropriate buffers of the Mesh used by each ray tracing instance that's part of a RayTracingAccelerationStructure.

There can be up to 8 vertex streams containing different vertex attributes in the unity_MeshVertexBuffers_RT array and a single index buffer bound to unity_MeshIndexBuffer_RT.

Typically, Mesh assets define the number vertex streams used and the format of the vertex attributes but a Mesh object can also be created from C# using various Mesh scripting APIs. For MeshRenderers, additional vertex attributes can also be specified using MeshRenderer.additionalVertexStreams.
unity_MeshInfo_RT and unity_MeshVertexDeclaration_RT are helper buffers used to exactly locate the vertex attributes within vertex streams.

When Renderers use multiple Materials, each sub-Mesh corresponding to a Material will use a different unity_MeshInfo_RT buffer, usually with a different MeshInfo.indexStart value.

unity_MeshVertexDeclaration_RT is a VertexAttributeInfo array containing information about each vertex attribute of a Mesh: the stream index into unity_MeshVertexBuffers_RT buffer array, a vertex attribute format, the vertex attribute byte offset within a vertex and the vertex attribute dimension.

One of the following defines can be used to index into the unity_MeshVertexDeclaration_RT structured array.

```
#define kVertexAttributePosition        0
#define kVertexAttributeNormal          1
#define kVertexAttributeTangent         2
#define kVertexAttributeColor           3
#define kVertexAttributeTexCoord0       4
#define kVertexAttributeTexCoord1       5
#define kVertexAttributeTexCoord2       6
#define kVertexAttributeTexCoord3       7
#define kVertexAttributeTexCoord4       8
#define kVertexAttributeTexCoord5       9
#define kVertexAttributeTexCoord6       10
#define kVertexAttributeTexCoord7       11
```

Helper Functions To The Rescue

There is a couple of HLSL helper functions in the **UnityRayTracingMeshUtils.cginc** include that will handle the complexity when reading vertex attributes and vertex indices:

```
uint3 UnityRayTracingFetchTriangleIndices(uint primitiveIndex)
```

- Reads the 3 vertex indices of the triangle that the ray hit.
- Called using *PrimitiveIndex()* HLSL intrinsic as a parameter.
- Supports both 16-bit and 32-bit index buffers.

```
float2 UnityRayTracingFetchVertexAttribute2(uint vertexIndex, uint attributeType)
float3 UnityRayTracingFetchVertexAttribute3(uint vertexIndex, uint attributeType)
float4 UnityRayTracingFetchVertexAttribute4(uint vertexIndex, uint attributeType)
```

- Reads various vertex attributes with different dimensions.
- Called using the result of the UnityRayTracingFetchTriangleIndices function and one of the kVertexAttribute* defines.
- Returns a zero vector if the vertex attribute does not exist in one of the vertex streams.
- Supports most common vertex attribute formats (see below).
- If the actual vertex attribute dimension is smaller than the one intended by the function call then the extra channels are zeroed out (e.g. calling UnityRayTracingFetchVertexAttribute4(index, kVertexAttributeTexCoord0) but the actual TexCoord0 has only 2 channels it will return a float4(u, v, 0, 0)).

bool UnityRayTracingHasVertexAttribute(uint attributeType) - Unity 2021.2 and above

- Checks if the vertex attribute attributeType is present in one of the unity_MeshVertexBuffers_RT vertex streams.
- One use case is when reading UVs for sampling baked lightmaps. These UVs are bound to TexCoord1 but they are not mandatory perhaps because they were not generated at import time for example. The standard in Unity is to use regular TexCoord0 UVs for sampling baked lightmaps when TexCoord1 UVs are missing.

In Unity 2021.2, the following vertex attribute formats (for VertexAttributeInfo.Format) are supported by UnityRayTracingFetchVertexAttribute functions as defined in **UnityRayTracingMeshUtils.cginc**:

```
#define kVertexFormatFloat         0
#define kVertexFormatFloat16       1
#define kVertexFormatUNorm8        2
#define kVertexFormatUNorm16       4
#define kVertexFormatSNorm16       5
```

These vertex attribute formats are not currently supported:

```
#define kVertexFormatSNorm8        3
#define kVertexFormatUInt8         6
#define kVertexFormatSInt8         7
#define kVertexFormatUInt16        8
#define kVertexFormatSInt16        9
#define kVertexFormatUInt32        10
#define kVertexFormatSInt32        11
```

Vertex Attribute Interpolation

When an “any hit shader” or a “closest hit shader” is invoked, the fixed-function ray-triangle intersection mechanism will pass the ray payload and the triangle barycentric coordinates of the intersection as an input to these shader functions.

E.g.

```
[shader("closesthit")]
void ClosestHitMain(inout RayPayload payload, in AttributeData attribs)
```

The standard format of the AttributeData structure is:

```
struct AttributeData
{
    float2 barycentrics;
};
```

Given the vertex attributes v0, v1 and v2 for the 3 vertices of a triangle, barycentrics.x is the weight for v1 and barycentrics.y is the weight for v2. For example, the shader code can interpolate by doing:

```
float3 barycentricCoords = float3(1.0 - attribs.barycentrics.x - attribs.barycentrics.y, attribs.barycentrics.x,
attribs.barycentrics.y);

v = v0 * barycentricCoords.x + v1 * barycentricCoords.y + v2 * barycentricCoords.z;
```

Sample Code

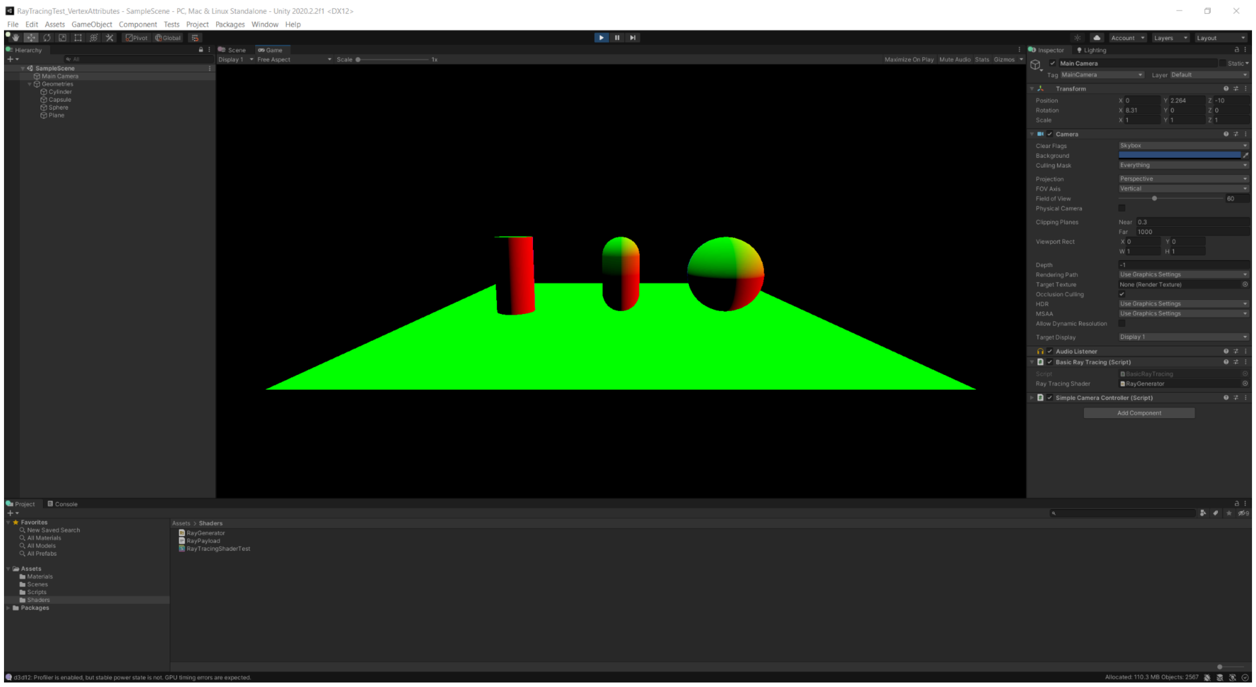
In the next example we will read the 3 vertices of the intersected triangle and interpolate them to get the interpolated world normal at the intersection point. Note that only the vertex normal is read from the vertex stream but other attributes like UVs or local vertex position can be read and interpolated in a similar fashion.

```
SubShader
{
    Pass
    {
        Name "Test"
        Tags{ "LightMode" = "RayTracing" }

        HLSLPROGRAM

        #include "UnityRaytracingMeshUtils.cginc"
        #include "RayPayload.hlsl"

        #pragma raytracing test

        struct AttributeData
        {
            float2 barycentrics;
        };
        struct Vertex
        {
            float3 normal;
        };
        Vertex FetchVertex(uint vertexIndex)
        {
            Vertex v;
            v.normal = UnityRayTracingFetchVertexAttribute3(vertexIndex, kVertexAttributeNormal);
            return v;
        }

        Vertex InterpolateVertices(Vertex v0, Vertex v1, Vertex v2, float3 barycentrics)
        {
            Vertex v;
            #define INTERPOLATE_ATTRIBUTE(attr) v.attr = v0.attr * barycentrics.x + v1.attr * barycentrics.y +
v2.attr * barycentrics.z
            INTERPOLATE_ATTRIBUTE(normal);
            return v;
        }

        [shader("closesthit")]
        void ClosestHitMain(inout RayPayload payload, in AttributeData attribs)
        {
            uint3 triangleIndices = UnityRayTracingFetchTriangleIndices(PrimitiveIndex());

            Vertex v0, v1, v2;
            v0 = FetchVertex(triangleIndices.x);
            v1 = FetchVertex(triangleIndices.y);
            v2 = FetchVertex(triangleIndices.z);

            float3 barycentricCoords = float3(1.0 - attribs.barycentrics.x - attribs.barycentrics.y,
attribs.barycentrics.x, attribs.barycentrics.y);

            Vertex v = InterpolateVertices(v0, v1, v2, barycentricCoords);

            float3 worldNormal = normalize(mul(v.normal, (float3x3)WorldToObject()));

            payload.color.xyz = worldNormal;
        }

        ENDHLSL
    }
}
```

The full project can be accessed in github:
https://github.com/INedelcu/RayTracingShader_VertexAttributeInterpolation

Author: Ionut Nedelcu

The project will simply generate the world normals of the scene geometry as output in Game View through the ray generation shader in RayGenerator.raytrace.



Main Camera Static
Tag MainCamera Layer Default

Transform
Position X 0 Y 2.264 Z -10
Rotation X 8.31 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera
Clear Flags: Skybox
Culling Mask: Everything
Projection: Perspective
FOV Axis: Vertical
Field of View: 60
Physical Camera: ☐
Clipping Planes: Near 0.3, Far 1000
Viewport Rect: X 0 Y 0 W 1 H 1
Depth: -1
Rendering Path: Use Graphics Settings
Target Texture: None (Render Texture)
Occlusion Culling: ✓
HDR: Use Graphics Settings
MSAA: Use Graphics Settings
Allow Dynamic Resolution: ☐
Target Display: Display 1

Audio Listener

Basic Ray Tracing (Script)
Script: BasicRayTracing
Ray Tracing Shader: RayGenerator

Simple Camera Controller (Script)

d3d12: Profiler is enabled, but stable power state is not. GPU timing errors are expected.

Allocated: 110.3 MB Objects: 2567

RayTracingShaderTest.shader contains the code from the previous sample.