Introduction to DirectX Raytracing:
Overview and Introduction to Ray Tracing Shaders

Chris Wyman, NVIDIA

Twitter: @_cwyman_
E-mail: chris.wyman@acm.org

More information: http://intro-to-dxr.cwyman.org
Next Steps

- Pete gave a nice overview of basics:
  - What is ray tracing? Why use ray tracing?
- Now we want to ask:
  - How do I do it?
Next Steps

- Pete gave a nice overview of basics:
  - What is ray tracing? Why use ray tracing?
- Now we want to ask:
  - How do I do it?
- Of course, you could start from scratch:
  - Write a CPU ray tracer; plenty of resources
  - Write a GPU ray tracer; can be tricky & ugly
Next Steps

- Pete gave a nice overview of basics:
  - What is ray tracing? Why use ray tracing?

- Now we want to ask:
  - How do I do it?

- Of course, you could start from scratch:
  - Write a CPU ray tracer; plenty of resources
  - Write a GPU ray tracer; can be tricky & ugly

- Use vendor-specific APIs
  - Hide ugly, low-level implementation details
  - Poor scaling cross-vendor, interact w/ raster
Use Standardized API: DirectX Raytracing

- Of course, that’s why you are here:
  - Today’s goal: show how to use DX Raytracing
Use Standardized API: DirectX Raytracing

- Of course, that’s why you are here:
  - Today’s goal: show how to use DX Raytracing

- Part of a widely-used API, DirectX

- Shares resources:
  - No data copying between raster and ray tracing

- Works across multiple vendors, either:
  - Via vendor-provided software or hardware
  - Via standardized compatibility layer (on DX12 GPUs)
Overview: Modern Graphics APIs

- Two main parts:
Overview: Modern Graphics APIs

• Two main parts:
  – **GPU device code (aka “shaders”):**
    • Includes parallel rendering and other parallel tasks
    • Simplified; writing parallel code *looks like* serial code
Overview: Modern Graphics APIs

• Two main parts:
  – **GPU device code (aka “shaders”):**
    • Includes parallel rendering and other parallel tasks
    • Simplified; writing parallel code *looks like* serial code
  – **CPU host code (often “DirectX API”):**
    • Manages memory resources (disk → CPU ↔ GPU)
    • Sets up, controls, manages, spawns GPU tasks
    • Defines shared graphics data structures (like ray accelerations structures)
    • Allows higher-level graphics algorithms requiring multiple passes
Overview: Modern Graphics APIs

- Two main parts:
  - **GPU device code (aka “shaders”):**
    - Includes parallel rendering and other parallel tasks
    - Simplified; writing parallel code *looks like* serial code
  - **CPU host code (often “DirectX API”):**
    - Manages memory resources (disk → CPU ↔ GPU)
    - Sets up, controls, manages, spawns GPU tasks
    - Defines shared graphics data structures (like ray accelerations structures)
    - Allows higher-level graphics algorithms requiring multiple passes

This rest of this talk focuses on DirectX Raytracing shaders!
Overview: Modern Graphics APIs

- Two main parts:
  - **GPU device code (aka “shaders”):**
    - Includes parallel rendering and other parallel tasks
    - Simplified; writing parallel code *looks like* serial code
  - **CPU host code (often “DirectX API”):**
    - Manages memory resources (disk → CPU ↔ GPU)
    - Sets up, controls, manages, spawns GPU tasks
    - Defines shared graphics data structures (like ray accelerations structures)
    - Allows higher-level graphics algorithms requiring multiple passes

Shawn will focus on host code later this morning
What Are Shaders?

• *Developer controlled* pieces of the graphics pipeline
  – The parts not automatically managed by the graphics API, driver, or hardware
  – Where you get to write your GPU code
What Are Shaders?

• **Developer controlled** pieces of the graphics pipeline
  – The parts not automatically managed by the graphics API, driver, or hardware
  – Where you get to write your GPU code

• Typically written in a C-like high-level language
  – *In DirectX, shaders are written in the High Level Shading Language (HLSL)*
What Are Shaders?

• **Developer controlled** pieces of the graphics pipeline
  – The parts not automatically managed by the graphics API, driver, or hardware
  – Where you get to write your GPU code

• Typically written in a C-like high-level language
  – *In DirectX, shaders are written in the High Level Shading Language (HLSL)*

• Individual shaders can represent instructions for complete GPU tasks
  – *E.g., DirectX’s compute shaders*
What Are Shaders?

- **Developer controlled** pieces of the graphics pipeline
  - The parts not automatically managed by the graphics API, driver, or hardware
  - Where you get to write your GPU code

- Typically written in a C-like high-level language
  - *In DirectX, shaders are written in the High Level Shading Language (HLSL)*

- Individual shaders can represent instructions for complete GPU tasks
  - *E.g., DirectX’s compute shaders*

- Or they can represent a subset of a more complex pipeline
  - *E.g., transforming geometry to cover the right pixels in DirectX’s vertex shaders*
DirectX Rasterization Pipeline

• What do shaders do in today’s widely-used rasterization pipeline?
DirectX Rasterization Pipeline

• What do shaders do in today’s widely-used rasterization pipeline?

• Run a shader, the *vertex shader*, on each vertex sent to the graphics card
  – This usually transforms it to the right location relative to the camera
DirectX Rasterization Pipeline

- What do shaders do in today’s widely-used rasterization pipeline?

- Group vertices into triangles, then run *tessellation shaders* to allow GPU subdivision of geometry
  - Includes 3 shaders with different goals, the *hull shader*, *tessellator shader*, and *domain shader*
DirectX Rasterization Pipeline

• What do shaders do in today’s widely-used rasterization pipeline?

• Run a shader, the *geometry shader*, on each tessellated triangle
  – Allows computations that need to occur on a complete triangle, e.g., finding the geometric surface normal
DirectX Rasterization Pipeline

- What do shaders do in today’s widely-used rasterization pipeline?

  - Rasterize our triangles (i.e., determine the pixels they cover)
    - Done by special-purpose hardware rather than user-software
    - Only a few developer controllable settings
What do shaders do in today’s widely-used rasterization pipeline?

- Run a shader, the **pixel shader** (or **fragment shader**), on each pixel generated by rasterization
  - This usually computes the surface’s color
DirectX Rasterization Pipeline

• What do shaders do in today’s widely-used rasterization pipeline?

  - Merge each pixel into the final output image (e.g., doing blending)
    - Usually done with special-purpose hardware
    - Hides optimizations like memory compression and converting image formats
DirectX Rasterization Pipeline

1. Squint a bit, and that pipeline looks like:

- Input: Set of Triangles
  - Shader(s) to transform vertices into displayable triangles
  - Rasterizer
  - Shader to compute color for each rasterized pixel
  - Output (ROP)

Output: Final Image
DirectX Ray Tracing Pipeline

- So what might a simplified ray tracing pipeline look like?
DirectX Ray Tracing Pipeline

• So what might a simplified ray tracing pipeline look like?

Please note:
A very simplified representation
DirectX Ray Tracing Pipeline

- So what might a simplified ray tracing pipeline look like?

- One advantage of ray tracing:
  - Algorithmically, much easier to add recursion

Please note:
A very simplified representation
DirectX Ray Tracing Pipeline

- Pipeline is split into five new shaders:
DirectX Ray Tracing Pipeline

- Pipeline is split into *five* new shaders:
  - A *ray generation shader* defines how to start ray tracing

Runs once per algorithm (or per pass)
DirectX Ray Tracing Pipeline

• Pipeline is split into **five** new shaders:
  
  – *A ray generation shader* defines how to start ray tracing
  
  – *Intersection shader(s)* define how rays intersect geometry

Runs once per algorithm (or per pass)

Defines geometric shapes, widely reusable
DirectX Ray Tracing Pipeline

- Pipeline is split into **five** new shaders:
  - A **ray generation shader** defines how to start ray tracing
  - **Intersection shader(s)** define how rays intersect geometry
  - **Miss shader(s)** define behavior when rays miss geometry

**Diagram**
- Input: Set of pixels
- Take input pixel position, generate ray(s)
- Intersect rays with scene
- Shade hit points; (optional) generate recursive ray(s)
- Output

- Runs once per algorithm (or per pass)
- Defines geometric shapes, widely reusable
DirectX Ray Tracing Pipeline

- Pipeline is split into five new shaders:
  - A **ray generation shader** defines how to start ray tracing
  - **Intersection shader(s)** define how rays intersect geometry
  - **Miss shader(s)** define behavior when rays miss geometry
  - **Closest-hit shader(s)** run once per ray (e.g., to shade the final hit)

  - **Runs once per algorithm (or per pass)**
  - **Defines geometric shapes, widely reusable**
DirectX Ray Tracing Pipeline

- Pipeline is split into **five** new shaders:
  - A **ray generation shader** defines how to start ray tracing
  - **Intersection shader(s)** define how rays intersect geometry
  - **Miss shader(s)** define behavior when rays miss geometry
  - **Closest-hit shader(s)** run once per ray (e.g., to shade the final hit)
  - **Any-hit shader(s)** run once per hit (e.g., to determine transparency)

\[\text{Runs once per algorithm (or per pass)}\]
\[\text{Defines geometric shapes, widely reusable}\]

\[\text{Note: Read spec for more advanced usage, since meaning of “any” may not match your expectations}\]
DirectX Ray Tracing Pipeline

- Pipeline is split into **five** new shaders:
  - A **ray generation shader** defines how to start ray tracing
  - **Intersection shader(s)** define how rays intersect geometry
  - **Miss shader(s)** define behavior when rays miss geometry
  - **Closest-hit shader(s)** run once per ray (e.g., to shade the final hit)
  - **Any-hit\(^1\) shader(s)** run once per hit (e.g., to determine transparency)

\(^1\)Note: Read spec for more advanced usage, since meaning of “any” may not match your expectations
DirectX Ray Tracing Pipeline

Pipeline is split into five new shaders:
- A **ray generation shader** defines how to start ray tracing
- Intersection shader(s) define how rays intersect geometry
- **Miss shader(s)** define behavior when rays miss geometry
- Closest-hit shader(s) run once per ray (e.g., to shade the final hit)
- Any-hit\(^1\) shader(s) run once per hit (e.g., to determine transparency)

An new, unrelated sixth shader:
- A **callable shader** can be launched from another shader stage

---

\(^1\)Note: Read spec for more advanced usage, since meaning of “any” may not match your expectations
Ray Generation Shader

- Write code to:
  - Specify what ray(s) to trace for each pixel

- In particular:
  - Launch ray(s) by calling new HLSL `TraceRay()` intrinsic
  - Accumulate ray color into image after ray tracing finishes
What Happens When Tracing a Ray?

Input: Set of Pixels

Ray Generation Shader

TraceRay() HLSL Call

Shade rays

Very Abstractly: “Ray Tracing” Happens

Output: Final Image

• Let’s zoom in
  – To look at what happens during ray tracing

TraceRay() Called

Ray Generation Shader

TraceRay() HLSL Call

Shade rays

Very Abstractly: “Ray Tracing” Happens

Traversal Loop

Return From TraceRay()

Ray Shading
What Happens When Tracing a Ray?

- A good mental model:
  - First, we traverse our scene to find what geometry our ray hits
What Happens When Tracing a Ray?

- A good mental model:
  - First, we traverse our scene to find what geometry our ray hits
  - When we find the closest hit, shade at that point using the closest-hit shader
    - This shader is a ray property; in theory, each ray can have a different closest-hit shader.
What Happens When Tracing a Ray?

- If our ray misses all geometry, the **miss shader** gets invoked
  - *Can consider this a shading routine that runs when you see the background*
    - Again, the miss shader is specified per-ray

```
TraceRay() Called

Closest-Hit Shader
Miss Shader
Ray Shading

Traversal Loop

Return From TraceRay()
```
How Does Scene Traversal Happen?

- Traverse the scene acceleration structure to ignore trivially-rejected geometry
  - An opaque process, with a few developer controls
  - Allows vendor-specific algorithms and updates without changing render code
How Does Scene Traversal Happen?

- If all geometry trivially ignored, ray traversal ends
How Does Scene Traversal Happen?

- If all geometry trivially ignored, ray traversal ends
- For potential intersections, an **intersection shader** is invoked
  - *Specific to a particular geometry type (e.g., one shader for spheres, one for Bezier patches)*
  - *DirectX includes a default, optimized intersection for triangles*
How Does Scene Traversal Happen?

- No shader-detected intersection? Detected intersection not the closest hit so far?
  - Continue traversing through our scene
How Does Scene Traversal Happen?

- Detected hit might be transparent? Run the any-hit shader\(^1\)
  - A ray-specific shader, specified in conjunction with the closest-hit shader
  - Shader can call IgnoreHit() to continue traversing, ignoring this surface

\(^1\)Please note: I did not name this shader!
How Does Scene Traversal Happen?

- Update the closest hit point with newly discovered hit
- Continue traversing to look for closer intersections
Ray Traversal Pipeline

• Continue traversing scene until no closer hits discovered
  – Had a valid hit along the ray? Shade via the closest-hit shader
  – No valid hits? Shade via the miss shader
Summary: DirectX Ray Tracing Shaders

- Control where your rays start?

See the ray generation shader
Summary: DirectX Ray Tracing Shaders

- Control where your rays start? See the ray generation shader
- Control when your rays intersect geometry? See the geometry's intersection shader
Summary: DirectX Ray Tracing Shaders

- Control where your rays start?
  See the ray generation shader

- Control when your rays intersect geometry?
  See the geometry's intersection shader

- Control what happens when rays miss?
  See your ray's miss shader
Summary: DirectX Ray Tracing Shaders

- Control where your rays start? See the ray generation shader
- Control when your rays intersect geometry? See the geometry's intersection shader
- Control what happens when rays miss? See your ray's miss shader
- Control how to shade your final hit points? See your ray's closest-hit shader
Summary: DirectX Ray Tracing Shaders

• Control where your rays start?  See the ray generation shader
• Control when your rays intersect geometry?  See the geometry's intersection shader
• Control what happens when rays miss?  See your ray’s miss shader
• Control how to shade your final hit points?  See your ray’s closest-hit shader
• Control how transparency behaves?  See your ray’s any-hit shader
What Goes Into a DirectX Ray Tracing Shader?

More information: http://intro-to-dxr.cwyman.org
Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think `main()` in C/C++
Starting a DXR Shader

• As any program, need an entry point where execution starts
  – Think `main()` in C/C++

• Shader entry points can be *arbitrarily named*

```c
[shader("raygeneration")]

void PinholeCameraRayGen() // No parameters required
{ ... <Place code here> ... }
```
Starting a DXR Shader

• As any program, need an entry point where execution starts
  – Think main() in C/C++

• Shader entry points can be arbitrarily named

• Type specified by HLSL attribute: [shader(“shader-type”)]
  – Remember the ray generation shader is where ray tracing starts
Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think main() in C/C++
- Shader entry points can be arbitrarily named
- Type specified by HLSL attribute: [shader("shader-type")]
  - Remember the ray generation shader is where ray tracing starts
- Starting other shader types look like this:

```cpp
void PinholeCameraRayGen()  // No parameters required
{ ... <Place code here> ... }

void PrimitiveIntersection () // No parameters required
{ ... <Place code here> ... }
```
Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think `main()` in C/C++
- Shader entry points can be arbitrarily named
- Type specified by HLSL attribute: `[shader(“shader-type”)]`
  - Remember the ray generation shader is where ray tracing starts
- Starting other shader types look like this:

```c
[shader(“raygeneration”)]
void PinholeCameraRayGen() // No parameters required
{ ... <Place code here> ... }

[shader(“intersection”)]
void PrimitiveIntersection () // No parameters required
{ ... <Place code here> ... }

[shader(“miss”)]
void RayMiss(inout RayPayload data) // User-defined struct
{ ... <Place code here> ... }
```
Starting a DXR Shader

- As any program, need an entry point where execution starts
  - *Think* `main()` in C/C++
- Shader entry points can be *arbitrarily named*
- Type specified by HLSL attribute: `[shader("shader-type")]
  - *Remember the ray generation shader is where ray tracing starts*
- Starting other shader types look like this:

  ```hlsl
  [shader("raygeneration")]
  void PinholeCameraRayGen() // No parameters required
  { ... <Place code here> ... }
  ```

  ```hlsl
  [shader("intersection")]
  void PrimitiveIntersection() // No parameters required
  { ... <Place code here> ... }
  ```

  ```hlsl
  [shader("miss")]
  void RayMiss(inout RayPayload data) // User-defined struct
  { ... <Place code here> ... }
  ```

  ```hlsl
  [shader("anyhit")]
  void RayAnyHit(inout RayPayload data,
                 IntersectAttribs attribs)
  { ... <Place code here> ... }
  ```
Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think `main()` in C/C++
- Shader entry points can be arbitrarily named
- Type specified by HLSL attribute: `[shader(“shader-type”)]`
  - Remember the ray generation shader is where ray tracing starts
- Starting other shader types look like this:

```cpp
[shader(“raygeneration”)]
void PinholeCameraRayGen() // No parameters required
{ ... <Place code here> ... }

[shader(“intersection”)]
void PrimitiveIntersection() // No parameters required
{ ... <Place code here> ... }

[shader(“miss”)]
void RayMiss(inout RayPayload data) // User-defined struct
{ ... <Place code here> ... }

[shader(“anyhit”)]
void RayAnyHit(inout RayPayload data,
            IntersectAttribs attribs)
{ ... <Place code here> ... }

[shader(“closesthit”)]
void RayClosestHit(inout RayPayload data,
                     IntersectAttribs attribs)
{ ... <Place code here> ... }
```
Starting a DXR Shader

As any program, need an entry point where execution starts
  - Think main() in C/C++

Shader entry points can be arbitrarily named

Type specified by HLSL attribute: [shader(“shader-type”)]
  - Remember the ray generation shader is where ray tracing starts

Starting other shader types look like this:
  - RayPayload is a user-defined (and arbitrarily named structure)

```c
[shader(“raygeneration”)]
void PinholeCameraRayGen() // No parameters required
{ ... <Place code here> ... }

[shader(“intersection”)]
void PrimitiveIntersection() // No parameters required
{ ... <Place code here> ... }

[shader(“miss”)]
void RayMiss(inout RayPayload data) // User-defined struct
{ ... <Place code here> ... }

[shader(“anyhit”)]
void RayAnyHit(inout RayPayload data, IntersectAttribs attrs)
{ ... <Place code here> ... }

[shader(“closesthit”)]
void RayClosestHit(inout RayPayload data, IntersectAttribs attrs)
{ ... <Place code here> ... }
```
Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think main() in C/C++
- Shader entry points can be arbitrarily named
- Type specified by HLSL attribute: \[ \text{shader(“shader-type”)} \]
  - Remember the ray generation shader is where ray tracing starts
- Starting other shader types look like this:
  - RayPayload is a user-defined (and arbitrarily named structure)
  - IntersectAttribs has data reported on hits (by intersection shader)
What is a Ray Payload?

- Ray payload is an arbitrary user-defined, user-named structure
  - Contains intermediate data needed during ray tracing

```c
struct SimpleRayPayload {
    float3 rayColor;
};
```
What is a Ray Payload?

• Ray payload is an arbitrary user-defined, user-named structure
  – Contains intermediate data needed during ray tracing

```cpp
struct SimpleRayPayload {
  float3 rayColor;
};
```

Not familiar with HLSL?
Built-in types include scalar types: bool, int, uint, float
Also vectors of up to 4 components: bool1, int2, uint3, float4
And matrices up to 4x4 size: uint1x4, float2x2, int3x2, float4x4
What is a Ray Payload?

• Ray payload is an arbitrary user-defined, user-named structure
  – Contains intermediate data needed during ray tracing
  — Note: Keep ray payload as small as possible
  • Large payloads will reduce performance; spill registers into memory

struct SimpleRayPayload {
  float3 rayColor;
};

Not familiar with HLSL?
Built-in types include scalar types: bool, int, uint, float
Also vectors of up to 4 components: bool1, int2, uint3, float4
And matrices up to 4x4 size: uint1x4, float2x2, int3x2, float4x4
What is a Ray Payload?

Ray payload is an arbitrary user-defined, user-named structure
- Contains intermediate data needed during ray tracing
- **Note**: Keep ray payload as small as possible
  - Large payloads will reduce performance; spill registers into memory

A simple ray might look like this:
- Sets color to **blue** when the ray misses
- Sets color to **red** when the ray hits an object

```
struct SimpleRayPayload
{
  float3 rayColor;
};

[shader("miss")]
void RayMiss(inout SimpleRayPayload data)
{
  data.rayColor = float3( 0, 0, 1 );
}

[shader("closesthit")]
void RayClosestHit(inout SimpleRayPayload data, IntersectAttribs attribs)
{
  data.rayColor = float3( 1, 0, 0 );
}
```
What are the Intersection Attributes?

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?
What are the Intersection Attributes?

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?

- Specific to each intersection type
  - One structure for triangles, one for spheres, one for Bezier patches
What are the Intersection Attributes?

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?

- Specific to each intersection type
  - One structure for triangles, one for spheres, one for Bezier patches
  - DirectX provides a built-in for the fixed function triangle intersector

```c
struct BuiltinIntersectionAttribs
{
    // Barycentric coordinates of hit in
    float2 barycentrics; // the triangle are: (1-x-y, x, y)
}
```
What are the Intersection Attributes?

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?

- Specific to each intersection type
  - One structure for triangles, one for spheres, one for Bezier patches
  - DirectX provides a built-in for the fixed function triangle intersector
  - Could imagine custom intersection attribute structures like:

```c
struct BuiltinIntersectionAttribs {
    float2 barycentrics; // the triangle are: (1-x-y, x, y)
}

struct PossibleSphereAttribs {
    float2 thetaPhi; // the sphere (thetaPhi.x, thetaPhi.y)
}

struct PossibleVolumeAttribs {
    float3 vox; // return voxel coord: (vox.x, vox.y, vox.z)
}
```
What are the Intersection Attributes?

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?

- Specific to each intersection type
  - One structure for triangles, one for spheres, one for Bezier patches
  - DirectX provides a built-in for the fixed function triangle intersector
  - Could imagine custom intersection attribute structures like:

- Limited attribute structure size: max 32 bytes

```plaintext
struct BuiltinIntersectionAttribs {
  float2 barycentrics; // Barycentric coordinates of hit in
                         // the triangle are: (1-x-y, x, y)
}

struct PossibleSphereAttribs {
  float2 thetaPhi; // Giving (theta,phi) of the hit on
                   // the sphere (thetaPhi.x, thetaPhi.y)
}

struct PossibleVolumeAttribs {
  float3 vox; // Doing volumetric ray marching? Maybe
               // return voxel coord: (vox.x, vox.y, vox.z)
}
```
A Simple Example
A Simple Example

• Besides our shader, what data is needed on the GPU to shoot rays?
A Simple Example

• Besides our shader, what data is needed on the GPU to shoot rays?
• We need somewhere to write our output

// A standard DirectX unordered access view (a.k.a., “read-write texture”)
RWTexture<float4> outTex;
A Simple Example

- Besides our shader, what data is needed on the GPU to shoot rays?
- We need somewhere to write our output
- Where are we looking? We need camera data

```cpp
// A standard DirectX unordered access view (a.k.a., “read-write texture”)
RWTexture<float4> outTex;

// An HLSL “constant buffer”, to be populated from your host C++ code
cbuffer RayGenData {
    float3 wsCamPos;       // World space camera position
    float3 wsCamU, wsCamV, wsCamW;  // Camera right, up, and forward vectors
};
```
A Simple Example

- Besides our shader, what data is needed on the GPU to shoot rays?
- We need somewhere to write our output
- Where are we looking? We need camera data
- Need to know about our scene geometry

```cpp
// A standard DirectX unordered access view (a.k.a., “read-write texture”)
RWTexture<float4> outTex;

// An HLSL “constant buffer”, to be populated from your host C++ code
cbuffer RayGenData {
    float3 wsCamPos;  // World space camera position
    float3 wsCamU, wsCamV, wsCamW; // Camera right, up, and forward vectors
};

// Our scene’s ray acceleration structure, setup via the C++ DirectX API
RaytracingAccelerationStructure sceneAccelStruct;
```
A Simple Example

• Besides our shader, what data is needed on the GPU to shoot rays?
• We need somewhere to write our output
• Where are we looking? We need camera data
• Need to know about our scene geometry
• Also need information on how to shade the scene
  – More complex topic
  – Depends on your program’s or engine’s material format
  – Depends on your shading models
  – Leave for later, see full tutorial code for examples

// A standard DirectX unordered access view (a.k.a., “read-write texture”)
RWTexture<float4> outTex;

// An HLSL “constant buffer”, to be populated from your host C++ code
cbuffer RayGenData {
    float3 wsCamPos; // World space camera position
    float3 wsCamU, wsCamV, wsCamW; // Camera right, up, and forward vectors
};

// Our scene’s ray acceleration structure, setup via the C++ DirectX API
RaytracingAccelerationStructure sceneAccelStruct;
A Simple Example – Code

[shader("raygeneration")]
void PinholeCamera() {
  uint2 curPixel = DispatchRaysIndex().xy;
  uint2 totalPixels = DispatchRaysDimensions().xy;

  ...
}

CPU → GPU data declarations

- cbuffer RayGenData {
  // World-space camera data
  float3 wsCamPos;
  float3 wsCamU, wsCamV, wsCamW;
};
- RaytracingAccelerationStructure sceneAccelStruct;
- RWTexture<float4> outTex; // Output texture
- ...

What pixel are we currently computing?
How many rays, in total, are we generating?
A Simple Example – Code

```cpp
[shader("raygeneration")]

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;
}
```

Find pixel center in [0..1] x [0..1]
Compute normalized device coordinate (as in raster)
Convert NDC into pixel’s ray direction (using camera inputs)
A Simple Example – Code

```cpp
[shader("raygeneration")]
void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;
    ...
}
```

Collectively: Turn pixel ID into a ray direction
A Simple Example – Code

```cpp
[shader(“raygeneration”)]

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize( pixelRayDir );
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    ...}
```

Setup our ray

[RwTexture<float4> outTex; // Output texture
cbuffer RayGenData {
    // World-space camera data
    float3 wsCamPos;
    float3 wsCamU, wsCamV, wsCamW;
};
RaytracingAccelerationStructure sceneAccelStruct;]
A Simple Example – Code

---

shader("raygeneration")

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize(pixelRayDir);
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    ...
}

RayDesc is a new HLSL built-in type:

```hlsl
struct RayDesc {
    float3 Origin;   // Where the ray starts
    float TMin;      // Min distance for a valid hit
    float3 Direction; // Direction the ray goes
    float TMax;      // Max distance for a valid hit
};
```
A Simple Example – Code

```cpp
[shader("raygeneration")]
void PinholeCamera() {
  uint2 curPixel = DispatchRaysIndex().xy;
  uint2 totalPixels = DispatchRaysDimensions().xy;
  float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
  float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
  float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

  RayDesc ray;
  ray.Origin = wsCamPos;
  ray.Direction = normalize(pixelRayDir);
  ray.TMin = 0.0f;
  ray.TMax = 1e+38f;

  SimpleRayPayload payload = { float3(0, 0, 0) };

  // Setup our ray's payload
}
```
A Simple Example – Code

```cpp
// ... Shader ("raygeneration")

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;

    float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize(pixelRayDir);
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
              ray, payload );
}
```

...
A Simple Example – Code

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize(pixelRayDir);
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF, 
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER, 
              ray, payload );
}

A new intrinsic function in HLSL

Can call from ray generation, miss, and closest-hit shaders
A Simple Example – Code

```c
[shader("raygeneration")]

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize( pixelRayDir );
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
                HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
                ray, payload );
}
```

Our scene acceleration structure
A Simple Example – Code

```cpp
[shader("raygeneration")]

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize(pixelRayDir);
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
              ray, payload );
}
```

Special traversal behavior for this ray? (Here: No)
A Simple Example – Code

```c
[shader("raygeneration")]

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize(pixelRayDir);
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
              ray, payload );
}

RWTexture<float4> outTex; // Output texture
cbuffer RayGenData { // World-space camera data
    float3 wsCamPos;
    float3 wsCamU, wsCamV, wsCamW;
};
RaytracingAccelerationStructure sceneAccelStruct;

struct SimpleRayPayload {
    float3 color;
};
```

Instance mask; 0xFF → test all geometry

This allows us to ignore some geometry via a mask
A Simple Example – Code

```c
[shader("raygeneration")]
void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamW;

    RayDesc ray;
    ray.Origin  = wsCamPos;
    ray.Direction = normalize( pixelRayDir );
    ray.TMin    = 0.0f;
    ray.TMax    = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF, HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER, ray, payload );
}
```

Which intersection, any-hit, closest-hit, and miss shaders to use?

*Known from C++ API setup & total number of shaders. This case: 0, 1, 0*
A Simple Example – Code

```cpp
[shader("raygeneration")] void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray(Direction = normalize( pixelRayDir );
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
             HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
             ray, payload );
}
```

What ray are we shooting?
A Simple Example – Code

[shader("raygeneration")]

void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5, 0.5)) / totalPixels;
    float2 ndc = float2(2, -2) * pixelCenter + float2(-1, 1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize(pixelRayDir);
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
              ray, payload );
}

What is the ray payload? Stores intermediate, per-ray data
A Simple Example – Code

```cpp
[shader(“raygeneration”)]
void PinholeCamera() {
    uint2 curPixel = DispatchRaysIndex().xy;
    uint2 totalPixels = DispatchRaysDimensions().xy;
    float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
    float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
    float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

    RayDesc ray;
    ray.Origin = wsCamPos;
    ray.Direction = normalize( pixelRayDir );
    ray.TMin = 0.0f;
    ray.TMax = 1e+38f;

    SimpleRayPayload payload = { float3(0, 0, 0) };

    TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF,
              HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
              ray, payload);

    outTex[curPixel] = float4( payload.color, 1.0f );
}
```
**Combine With Simple Ray Type**

```cpp
RWTexture<float4> outTex;

cbuffer RayGenData { float3 wsCamPos, wsCamU, wsCamV, wsCamW; }
RaytracingAccelerationStructure sceneAccelStruct;
struct SimpleRayPayload { float3 color; }

[shader("raygeneration")]
void PinholeCamera() {
  uint2 curPixel = DispatchRaysIndex().xy;
  uint2 totalPixels = DispatchRaysDimensions().xy;
  float2 pixelCenter = (curPixel + float2(0.5,0.5)) / totalPixels;
  float2 ndc = float2(2,-2) * pixelCenter + float2(-1,1);
  float3 pixelRayDir = ndc.x * wsCamU + ndc.y * wsCamV + wsCamZ;

  RayDesc ray;
  ray.Origin = wsCamPos;
  ray.Direction = normalize( pixelRayDir );
  ray.TMin = 0.0f;
  ray.TMax = 1e+38f;

  SimpleRayPayload payload = { float3(0, 0, 0) };

  TraceRay( sceneAccelStruct, RAY_FLAG_NONE, 0xFF, 
             HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER, 
             ray, payload );

  outTex[curPixel] = float4( payload.color, 1.0f );
}

[shader("miss")]
void RayMiss(inout SimpleRayPayload data) {
  data.color = float3( 0, 0, 1 );
}

[shader("closesthit")]
void RayClosestHit(inout SimpleRayPayload data, 
                   BuiltinIntersectionAttribs attribs) {
  data.color = float3( 1, 0, 0 );
}
```

- Now you have a complete DirectX Raytracing shader
  - (Both intersection shader and any-hit shader are optional)
- Shoots rays from app-specified camera
- Returns **red** if rays hit geometry, **blue** on background
What Can DXR HLSL Shaders Do?
What Can DXR HLSL Shaders Do?

- All the standard HLSL data types, texture resources, user-definable structures and buffers
  - See Microsoft documentation for more details and course tutorials for more examples
What Can DXR HLSL Shaders Do?

• All the standard HLSL data types, texture resources, user-definable structures and buffers
  – See Microsoft documentation for more details and course tutorials for more examples

• Numerous standard HLSL intrinsic or built-in functions useful for graphics, spatial manipulation, and 3D mathematics
  – Basic math (sqrt, clamp, isinf, log), trigonometry (sin, acos, tanh), vectors (normalize, length), matrices (mul, transpose)
  – See Microsoft documentation for full list and course tutorials for more examples
What Can DXR HLSL Shaders Do?

- All the standard HLSL data types, texture resources, user-definable structures and buffers
  - See [Microsoft documentation](#) for more details and course tutorials for more examples
- Numerous standard HLSL intrinsic or built-in functions useful for graphics, spatial manipulation, and 3D mathematics
  - Basic math (sqrt, clamp, isinf, log), trigonometry (sin, acos, tanh), vectors (normalize, length), matrices (mul, transpose)
  - See [Microsoft documentation](#) for full list and course tutorials for more examples
- New intrinsic functions for ray tracing
  - Functions related to ray traversal: TraceRay(), ReportHit(), IgnoreHit(), and AcceptHitAndEndSearch()
  - Functions for ray state, e.g.: WorldRayOrigin(), RayTCurrent(), InstanceID(), and HitKind()
# New Ray Tracing Built-in Functions

<table>
<thead>
<tr>
<th>Ray Traversal Functions</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceRay()</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>Launch a new ray</td>
</tr>
<tr>
<td>ReportHit()</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Found a hit; test it; function returns true if hit accepted</td>
</tr>
<tr>
<td>IgnoreHit()</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>Hit point should be ignored, traversal continues</td>
</tr>
<tr>
<td>AcceptHitAndEndSearch()</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>Hit is good; stop search immediately, execute closest hit</td>
</tr>
</tbody>
</table>
# New Ray Tracing Built-in Functions

## Ray Traversal Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceRay()</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Launch a new ray</td>
</tr>
<tr>
<td>ReportHit()</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Found a hit; test it; function returns true if hit accepted</td>
</tr>
<tr>
<td>IgnoreHit()</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>Hit point should be ignored, traversal continues</td>
</tr>
<tr>
<td>AcceptHitAndEndSearch()</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>Hit is good; stop search immediately, execute closest hit</td>
</tr>
</tbody>
</table>

## Ray Launch Details

<table>
<thead>
<tr>
<th>Function</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>DispatchRaysDimensions()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>How many rays were launched (e.g., 1920 × 1080)</td>
</tr>
<tr>
<td>DispatchRaysIndex()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Why ray (in that range) is the shader currently processing</td>
</tr>
</tbody>
</table>
## New Ray Tracing Built-in Functions

### Ray Traversal Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceRay()</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Launch a new ray</td>
</tr>
<tr>
<td>ReportHit()</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Found a hit; test it; function returns true if hit accepted</td>
</tr>
<tr>
<td>IgnoreHit()</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>Hit point should be ignored, traversal continues</td>
</tr>
<tr>
<td>AcceptHitAndEndSearch()</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>Hit is good; stop search immediately, execute closest hit</td>
</tr>
</tbody>
</table>

### Ray Launch Details

<table>
<thead>
<tr>
<th>Function</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>DispatchRaysDimensions()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>How many rays were launched (e.g., 1920 × 1080)</td>
</tr>
<tr>
<td>DispatchRaysIndex()</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Why ray (in that range) is the shader currently processing</td>
</tr>
</tbody>
</table>

### Hit Specific Details

<table>
<thead>
<tr>
<th>Function</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>HitKind()</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>Information about what kind of hit we’re processing</td>
</tr>
</tbody>
</table>

(Developer data specified by your intersection shader. For triangles can be: HIT_KIND_TRIANGLE_FRONT_FACE or HIT_KIND_TRIANGLE_BACK_FACE)
# New Ray Tracing Built-in Functions

<table>
<thead>
<tr>
<th>Ray Introspection</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>RayTCurrent()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Current distance along the ray</td>
</tr>
<tr>
<td>RayTMin()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Min ray distance, as passed to this ray's TraceRay()</td>
</tr>
<tr>
<td>RayFlags()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>The flags passed to this ray's TraceRay()</td>
</tr>
<tr>
<td>WorldRayOrigin()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>The ray origin passed to this ray's TraceRay()</td>
</tr>
<tr>
<td>WorldRayDirection()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>The ray direction passed to this ray's TraceRay()</td>
</tr>
</tbody>
</table>
## New Ray Tracing Built-in Functions

<table>
<thead>
<tr>
<th>Current Object Introspection</th>
<th>Ray Gen</th>
<th>Intersect</th>
<th>Any Hit</th>
<th>Closest</th>
<th>Miss</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstanceIndex()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Instance index in acceleration structure (generated)</td>
</tr>
<tr>
<td>InstanceID()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Instance identifier in acceleration struct (user-provided)</td>
</tr>
<tr>
<td>PrimitiveIndex()</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>Index of primitive in geometry instance (generated)</td>
</tr>
<tr>
<td>ObjectToWorld()</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>Matrix to transform object-space to world-space</td>
</tr>
<tr>
<td>WorldToObject()</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>Matrix to transform world-space to object-space</td>
</tr>
<tr>
<td>ObjectRayOrigin()</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>Essentially: WorldToObject(WorldRayOrigin())</td>
</tr>
<tr>
<td>ObjectRayDirection()</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>Essentially: WorldToObject(WorldRayDirection())</td>
</tr>
</tbody>
</table>