thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

STRUCTURE OF GPU SHADERS

Specifically DirectX HLSL, but many similarities elsewhere

FIVE TYPES OF RAY TRACING SHADERS

- Ray tracing pipeline split into ***five*** shaders:
  - *A **ray generation shader** define how to start tracing rays*

FIVE TYPES OF RAY TRACING SHADERS

- Ray tracing pipeline split into ***five*** shaders:
  - *A **ray generation shader*** *define how to start tracing rays*
  - ***Intersection shader(s)*** *define how rays intersect geometry*

FIVE TYPES OF RAY TRACING SHADERS

- Ray tracing pipeline split into ***five*** shaders:
  - *A **ray generation shader*** *define how to start tracing rays*
  - ***Intersection shader(s)*** *define how rays intersect geometry*
  - ***Miss shader(s)*** *shading for when rays miss geometry*

FIVE TYPES OF RAY TRACING SHADERS

- Ray tracing pipeline split into ***five*** shaders:
  - *A **ray generation shader*** *define how to start tracing rays*
  - ***Intersection shader(s)*** *define how rays intersect geometry*
  - ***Miss shader(s)*** *shading for when rays miss geometry*
  - ***Closest-hit shader(s)*** *shading at the intersection point*

FIVE TYPES OF RAY TRACING SHADERS

- Ray tracing pipeline split into ***five*** shaders:

— *A **ray generation shader***	*define how to start tracing rays*
— ***Intersection shader(s)***	*define how rays intersect geometry*
— ***Miss shader(s)***	*shading for when rays miss geometry*
— ***Closest-hit shader(s)***	*shading at the intersection point*
— ***Any-hit shader(s)***	*run once per hit** (e.g., for transparency)*

FIVE TYPES OF RAY TRACING SHADERS

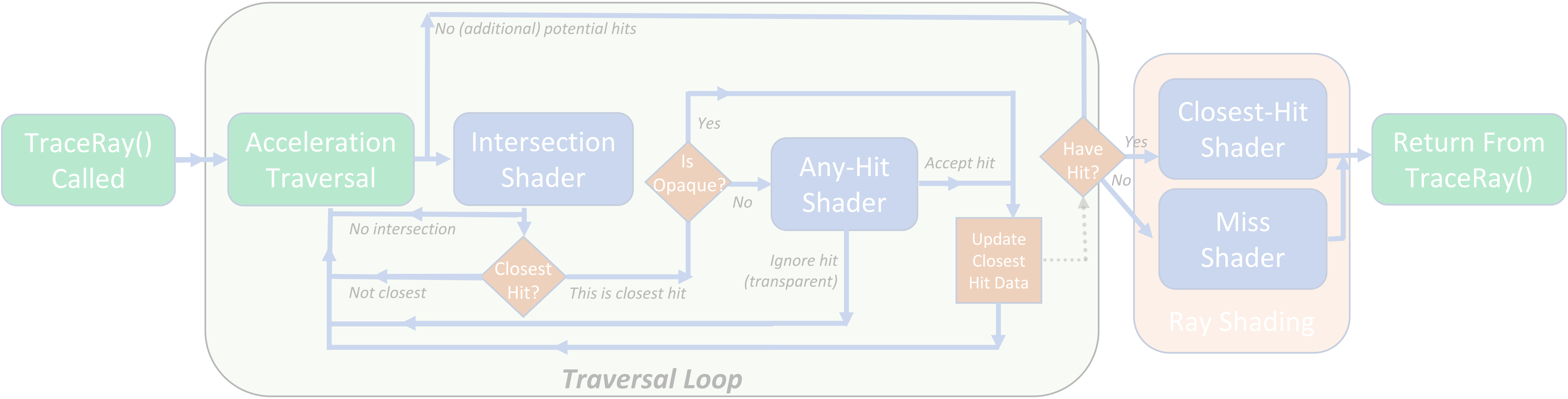
- Ray tracing pipeline split into ***five*** shaders:
  - *A **ray generation shader***
  - ***Intersection shader(s)***
  - ***Miss shader(s)***
  - ***Closest-hit shader(s)***
  - ***Any-hit shader(s)***

← Controls other shaders

← Defines object shapes (one shader per type)

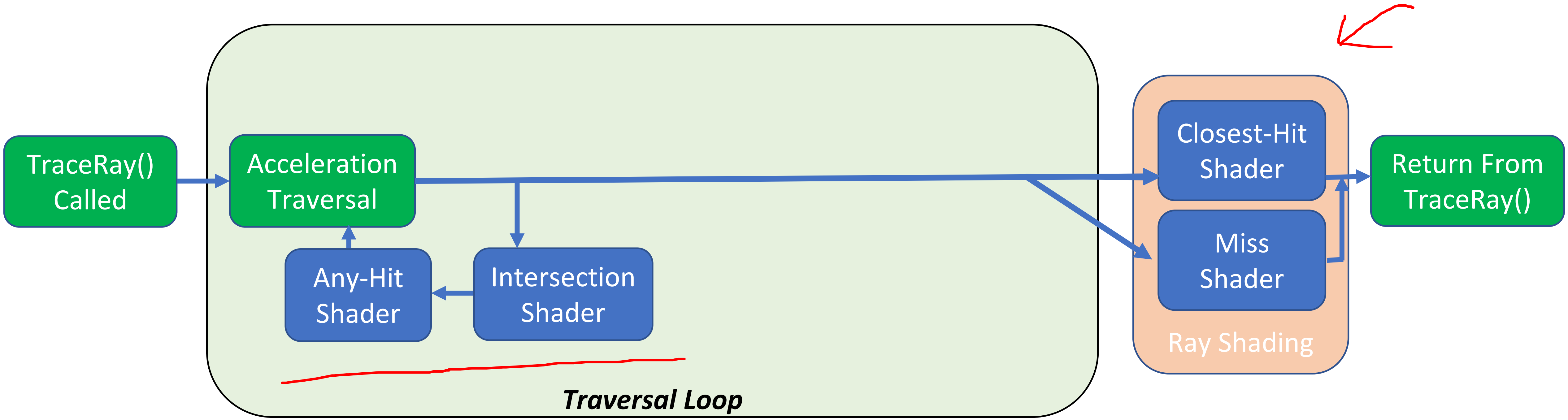
← Controls per-ray behavior (often many types)

HOW DO THESE FIT TOGETHER? [EYE CHART VERSION]



HOW DO THESE FIT TOGETHER? [LOGICAL VERSION]

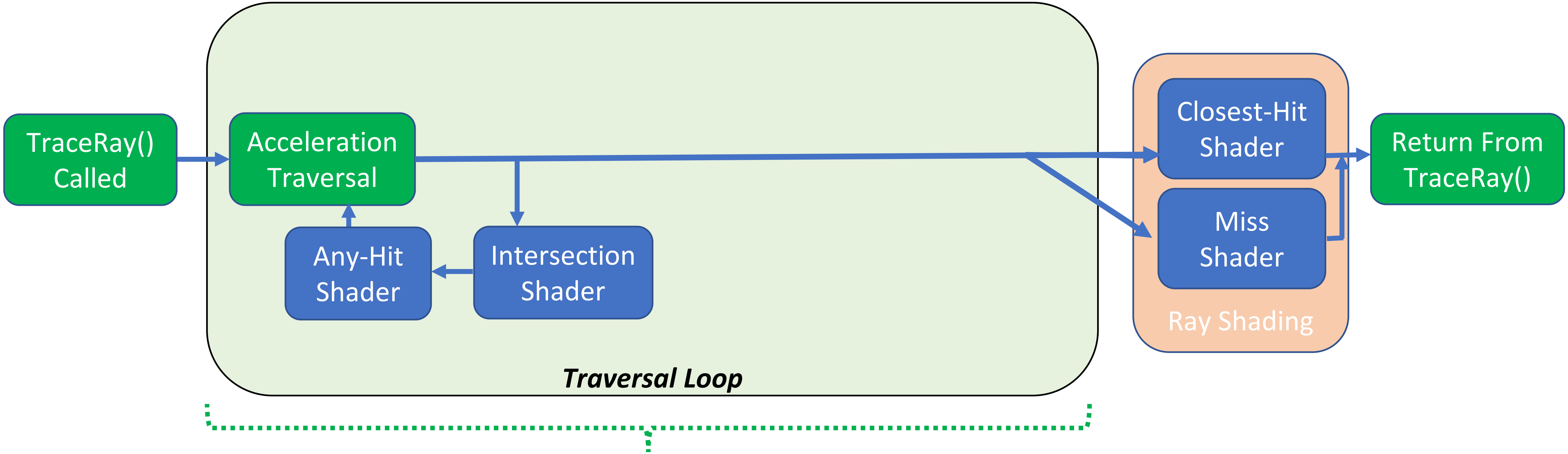
- Loop during ray tracing, testing hits until there's no more; then shade



thrive SIGGRAPH2019 LOS ANGELES • 28 JULY – 1 AUGUST

HOW DO THESE FIT TOGETHER? [LOGICAL VERSION]

- Loop during ray tracing, testing hits until there's no more; then shade



Some important details here; learn later for advanced functionality

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

```
[shader("raygeneration")]
void MyRayGen()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray         = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work
- Output image buffer
  - Communicates results with CPU

```
RWTexture<float4> gOutTex;
```

```
[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Information about scene
  - Passed as input from the CPU

```
RWTexture<float4> gOutTex;









[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray         = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work
- Each ray returns some value
  - Return payload is user-defined
  - Often, like this one, just a color
- Before tracing, initialize payload

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

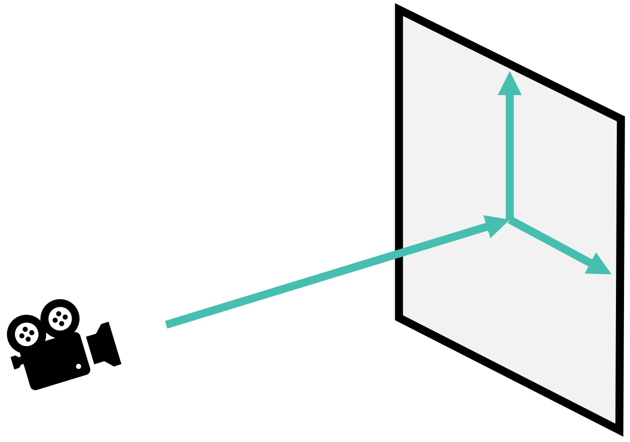
```
[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray          = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- You write a function here
  - Computes per-pixel ray direction
  - Based on location on screen

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

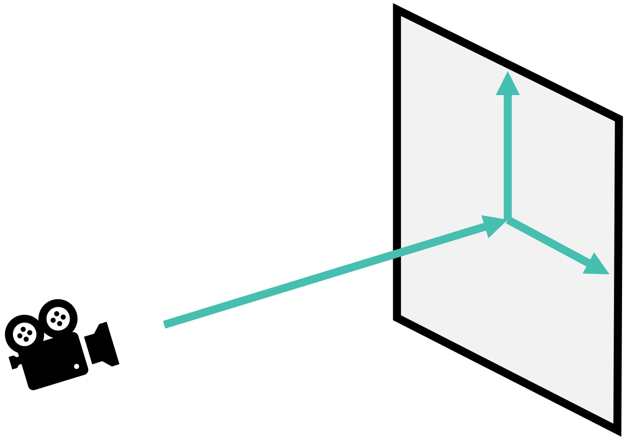


```
[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray          = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work
- You write a function here
  - Computes per-pixel ray direction
  - Based on location on screen
- Setup the ray to trace

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

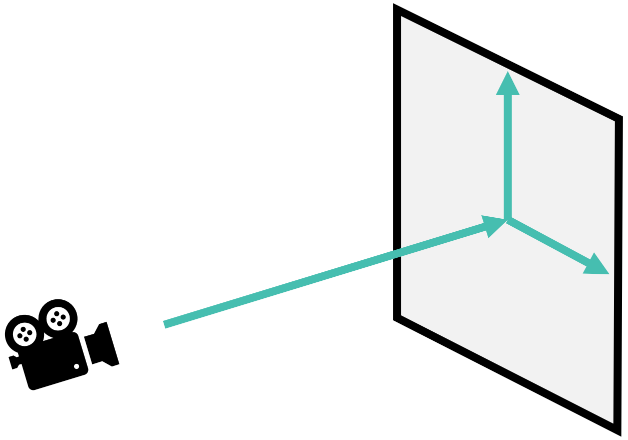


```
[shader(“raygeneration”)]
void MyRayGen()  {
    uint2  curPixel    = DispatchRaysIndex().xy;
    float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

    RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { float3(0, 0, 0) };

    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work
- You write a function here
  - Computes per-pixel ray direction
  - Based on location on screen
- Setup the ray to trace
  - Min and max distances to search

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```



```
[shader(“raygeneration”)]
void MyRayGen()  {
    uint2  curPixel    = DispatchRaysIndex().xy;
    float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

    RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { float3(0, 0, 0) };

    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Trace your ray here

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

```
[shader("raygeneration")]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Trace your ray here
  - Scene BVH

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };




[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Trace your ray here
  - Scene BVH
  - No special ray behaviors

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };









[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work
- Trace your ray here
  - Scene BVH
  - No special ray behaviors
  - What geometry should we test?
    - Bitmask; 0xFF → test all geometry

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

```
[shader(“raygeneration”)]
void MyRayGen()  {
    uint2  curPixel    = DispatchRaysIndex().xy;
    float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

    RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { float3(0, 0, 0) };

    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Trace your ray here
  - Scene BVH
  - No special ray behaviors
  - What geometry should we test?
    - Bitmask; 0xFF → test all geometry
  - Ray and payload from earlier

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

```
[shader(“raygeneration”)]
void MyRayGen()  {
    uint2  curPixel    = DispatchRaysIndex().xy;
    float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

    RayDesc ray         = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { float3(0, 0, 0) };

    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Which miss shader to use?
  - There’s a list of miss shaders
  - Specify index of the one to use

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}
```

```
[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work
- Which miss shader to use?
  - There’s a list of miss shaders
  - Specify index of the one to use
- In my tutorials, `MyMiss` is index 0
  - Why? First miss shader I loaded

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}
```

```
[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Which ***hit group*** to use?
  - May have 1 *any-hit shader*
  - May have 1 *closest-hit shader*
  - May have 1 *intersection shader*

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- Remember:
  - Ray generation shader starts work

- Which ***hit group*** to use?
  - May have 1 *any-hit shader*
  - May have 1 *closest-hit shader*
  - May have 1 *intersection shader*

- Here, has just one shader
  - It’s index 0 → specified first on load

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- How to read at high level:
  - For each pixel determine ray

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- How to read at high level:
  - For each pixel determine ray
  - Shoot the ray

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- How to read at high level:
  - For each pixel determine ray
  - Shoot the ray
  - If it misses? Return blue

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

- How to read at high level:
  - For each pixel determine ray
  - Shoot the ray
  - If it misses? Return blue
  - If it hits? Return red

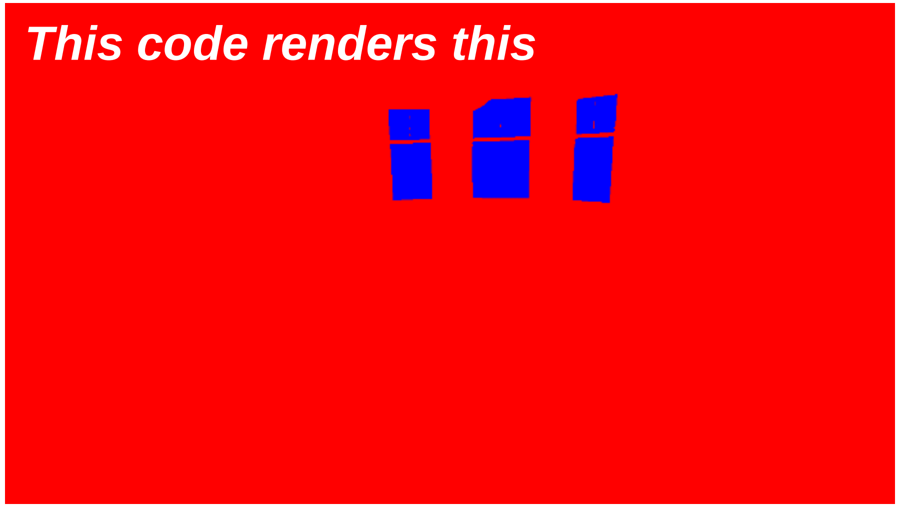
```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
    payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
    data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
    uint2  curPixel    = DispatchRaysIndex().xy;
    float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

    RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { float3(0, 0, 0) };

    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER

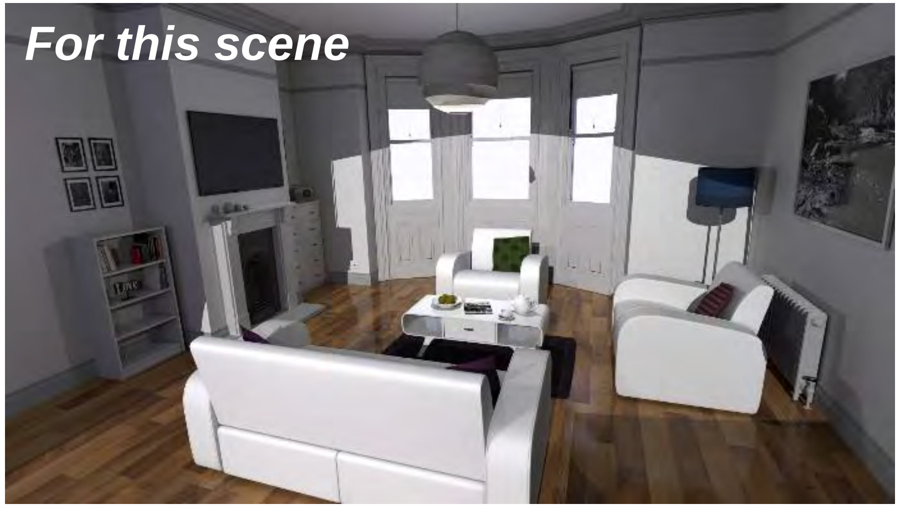
- How to read at high level:
  - For each pixel determine ray
  - Shoot the ray
  - If it misses? Return blue
  - If it hits? Return red
  - Output our result

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray        = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

REALLY SIMPLE GPU RAY TRACER



This code renders this



For this scene

```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[shader(“miss”)]
void MyMiss(inout RayPayload payload)  {
     payload.color = float3( 0, 0, 1 );
}

[shader(“closesthit”)]
void MyClosestHit(inout RayPayload data,
                  BuiltinTriangleIntersectAttribs attribs)  {
     data.color = float3( 1, 0, 0 );
}

[shader(“raygeneration”)]
void MyRayGen()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     RayDesc ray         = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
     RayPayload payload = { float3(0, 0, 0) };

     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );

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

WHAT ABOUT A REAL EXAMPLE?

WHAT ABOUT A REAL EXAMPLE?

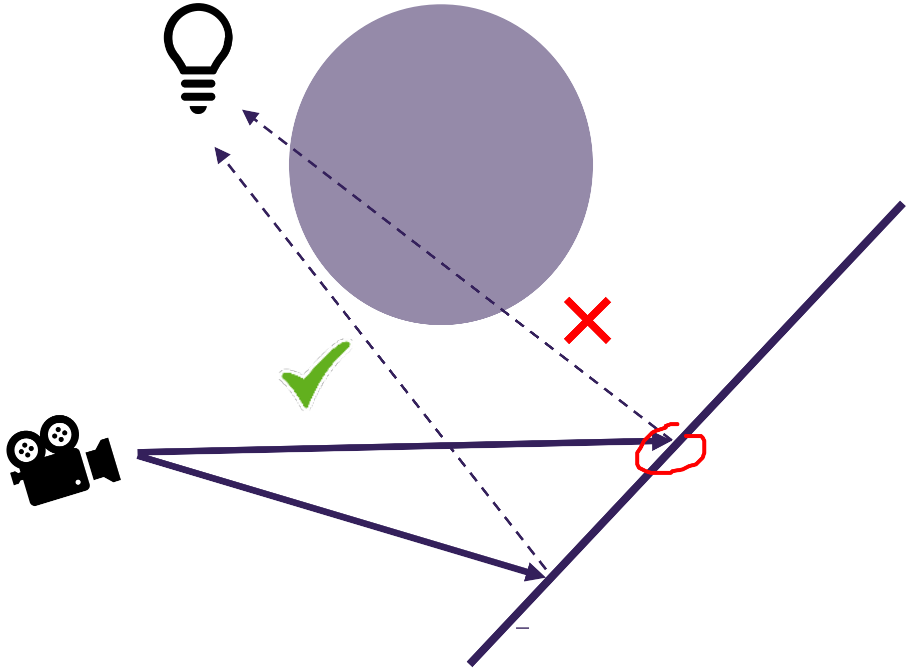
- Examples from my DXR tutors: http://intro-to-dxr.cwyman.org
  - Click on “code walkthrough”
  - Not quite equivalent to any of those, but close

WHAT ABOUT A REAL EXAMPLE?

- How about adding shadows?

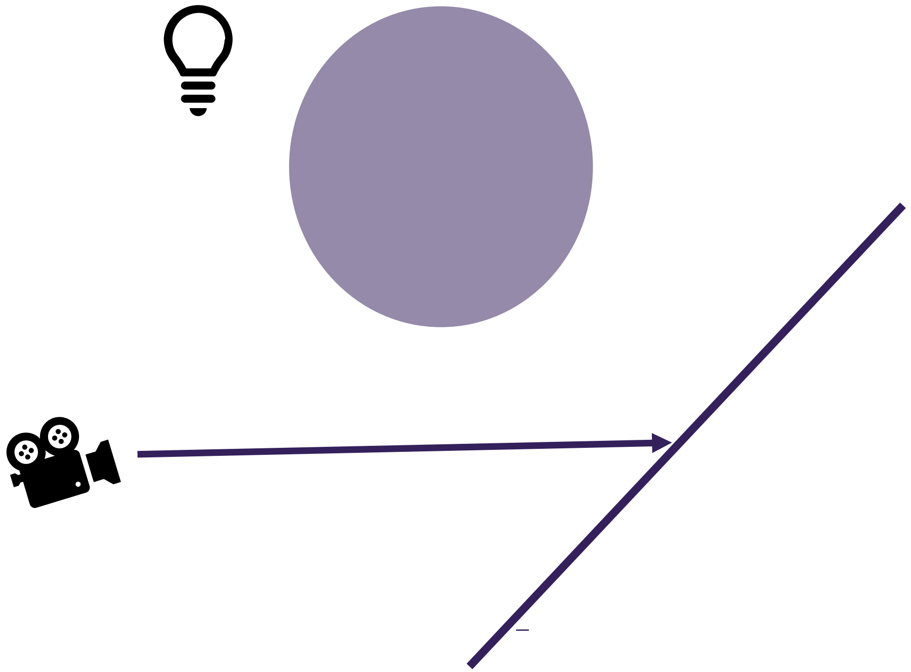
WHAT ABOUT A REAL EXAMPLE?

- How about adding shadows?
  - For each pixel, determine if light visible
  - Shoot a ray towards light



HOW DOES THIS WORK?

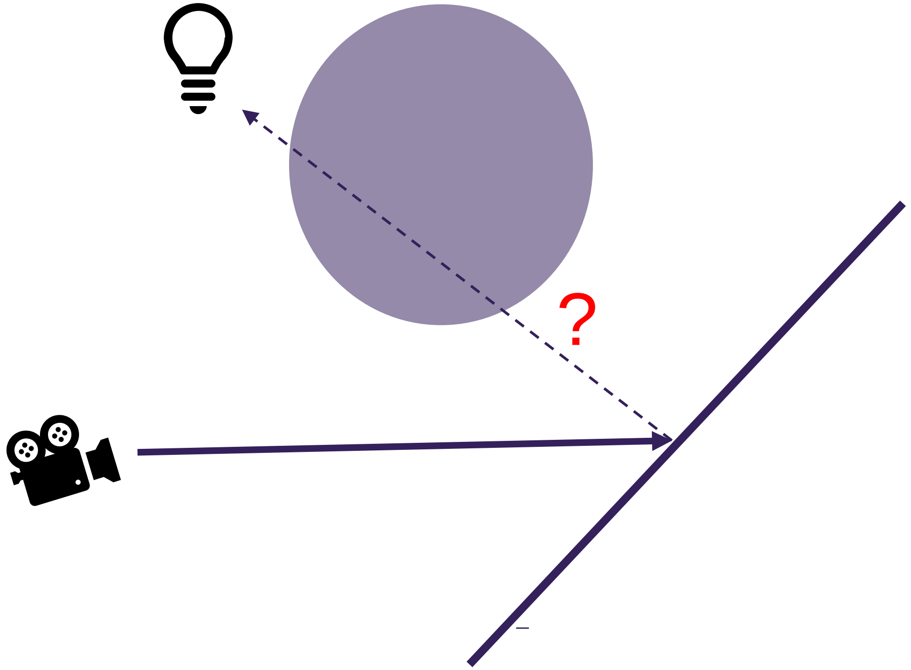
- Trace a ray from the camera



thrive SIGGRAPH2019 LOS ANGELES • 28 JULY – 1 AUGUST

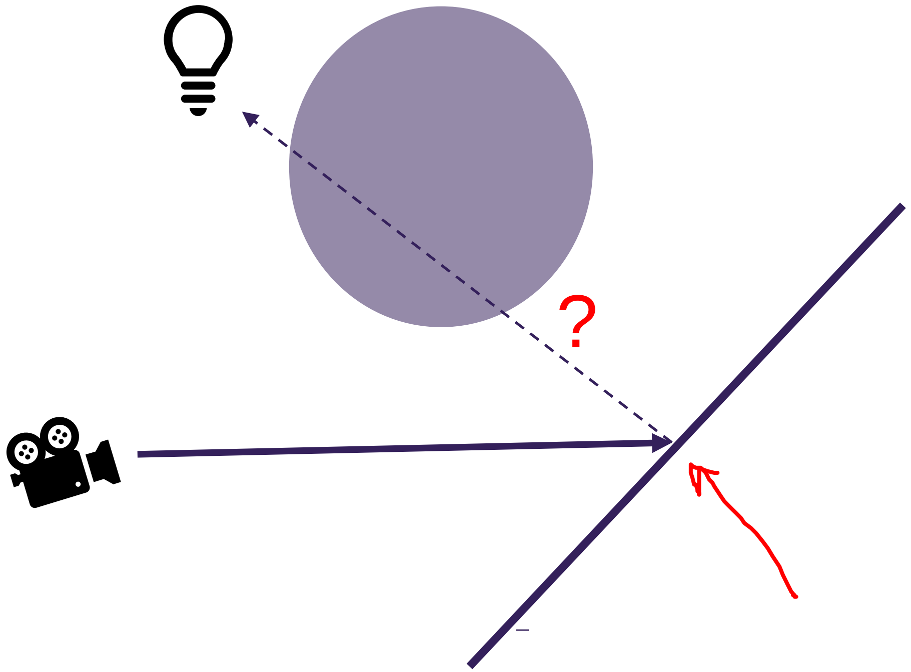
HOW DOES THIS WORK?

- Trace a ray from the camera
  - At the shading point (i.e., the closest hit)
  - Trace another ray towards the light



HOW DOES THIS WORK?

- Trace a ray from the camera
  - At the shading point (i.e., the closest hit)
  - Trace another ray towards the light
  - If it hits, shade the pixel as in shadow
  - If it misses, illuminate the pixel by the light



A REUSABLE SHADOW RAY

- Encapsulate a shadow ray
  - Create `shootShadowRay()`
  - Can call while shading

```
float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {




}
...
```

A REUSABLE SHADOW RAY

- Encapsulate a shadow ray
  - Create a ray
    - From some origin
    - In some direction
    - Check occlusions in $[t_{min} \ldots t_{max}]$

```
...
struct ShadowPayload {
     float visibility;   // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};




float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
     RayDesc       ray = { orig, minT, dir, maxT };
     ShadowPayload pay = { 0.0f };




}
...
```

A REUSABLE SHADOW RAY

- Encapsulate a shadow ray
  - Create a ray
    - From some origin
    - In some direction
    - Check occlusions in [t_{min}…t_{max}]
  - **Assume** shadows are occluded

```
...
struct ShadowPayload {
     float visibility;  // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};




float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
     RayDesc        ray = { orig, minT, dir, maxT };
     ShadowPayload pay = { 0.0f };




}
...
```

thrive SIGGRAPH2019 LOS ANGELES • 28 JULY - 1 AUGUST

A REUSABLE SHADOW RAY

- Encapsulate a shadow ray
  - — Create a ray
    - From some origin
    - In some direction
    - Check occlusions in [$t_{min}\ldots t_{max}$]
  - — **Assume** shadows are occluded
  - — Trace the ray
  - — Return 1 if lit, 0 otherwise

```
...
struct ShadowPayload {
     float visibility;  // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};
```

```
float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
     RayDesc       ray = { orig, minT, dir, maxT };
     ShadowPayload pay = { 0.0f };

     uint flags = RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH |
                  RAY_FLAG_SKIP_CLOSEST_HIT_SHADER;

     TraceRay( gRtScene, flags, 0xFF, 0, 1, 0, ray, pay );
     return pay.visibility;
}
...
```

A REUSABLE SHADOW RAY

- Encapsulate a shadow ray
  - — Create a ray
    - From some origin
    - In some direction
    - Check occlusions in $[t_{min} \ldots t_{max}]$
  - — **Assume** shadows are occluded
  - — Trace the ray
  - — Return 1 if lit, 0 otherwise
- Some shadow ray optimizations
  - — No shading; skip closest hit
  - — End at any occlusion
    - Need ***if*** not ***where***

```
...
struct ShadowPayload {
     float visibility;  // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};
```

```
float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
     RayDesc       ray = { orig, minT, dir, maxT };
     ShadowPayload pay = { 0.0f };

     uint flags = RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH |
                  RAY_FLAG_SKIP_CLOSEST_HIT_SHADER;

     TraceRay( gRtScene, flags, 0xFF, 0, 1, 0, ray, pay );
     return pay.visibility;
}
...
```

A REUSABLE SHADOW RAY

- Miss shader:
  - We missed…
  - Set visibility to 1.0

```
...
struct ShadowPayload {
     float visibility;  // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};

[shader(“miss”)]
void ShadowMiss(inout ShadowPayload pay) {
     pay.visibility = 1.0f;
}

float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
     RayDesc       ray = { orig, minT, dir, maxT };
     ShadowPayload pay = { 0.0f };

     uint flags = RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH |
                  RAY_FLAG_SKIP_CLOSEST_HIT_SHADER;

     TraceRay( gRtScene, flags, 0xFF, 0, 1, 0, ray, pay );
     return pay.visibility;
}
...
```

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY - 1 AUGUST

A REUSABLE SHADOW RAY

- Miss shader:
  - We missed…
  - Set visibility to 1.0

- Any hit shader:
  - Asks is occluder transparent?
  - If so, ignore this hit

```
...
struct ShadowPayload {
    float visibility;  // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};

[shader(“miss”)]
void ShadowMiss(inout ShadowPayload pay) {
    pay.visibility = 1.0f;
}

[shader(“anyhit”)]
void ShadowAnyHit(inout ShadowPayload pay, BuiltinIntersectAttribs attribs) {
    if (alphaTestFails(attribs))
        IgnoreHit();
}

float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
    RayDesc       ray = { orig, minT, dir, maxT };
    ShadowPayload pay = { 0.0f };

    uint flags = RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH |
                 RAY_FLAG_SKIP_CLOSEST_HIT_SHADER;

    TraceRay( gRtScene, flags, 0xFF, 0, 1, 0, ray, pay );
    return pay.visibility;
}
...
```

A REUSABLE SHADOW RAY

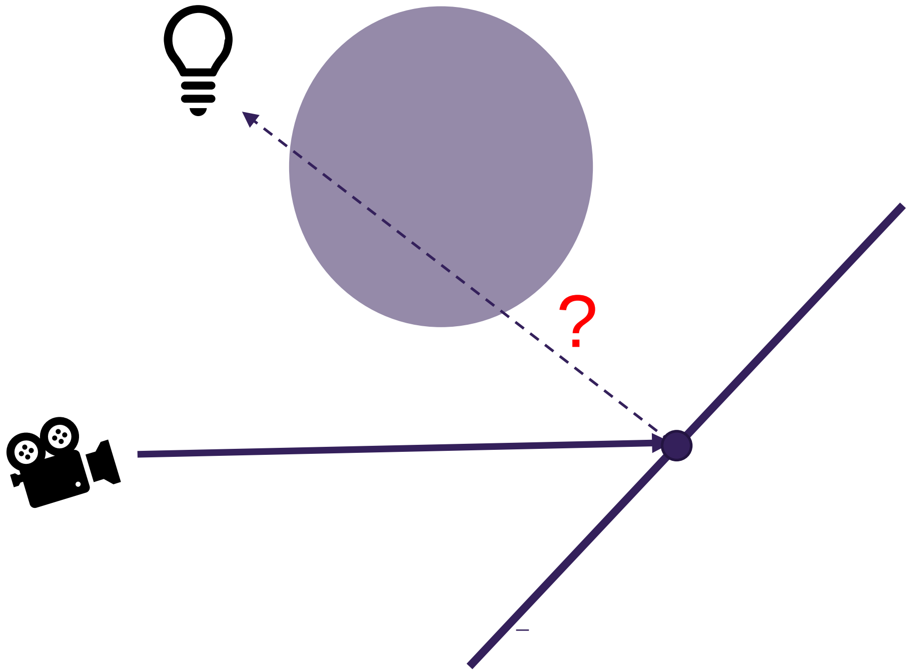
- Gives reusable shadow rays
  – Useful in many contexts

```
...
struct ShadowPayload {
     float visibility;  // 0.0 means ‘shadowed’, 1.0 means ‘lit’
};

[shader(“miss”)]
void ShadowMiss(inout ShadowPayload pay) {
     pay.visibility = 1.0f;
}

[shader(“anyhit”)]
void ShadowAnyHit(inout ShadowPayload pay, BuiltinIntersectAttribs attribs) {
     if (alphaTestFails(attribs))
          IgnoreHit();
}

float shootShadowRay(float3 orig, float3 dir, float minT, float maxT)  {
     RayDesc       ray = { orig, minT, dir, maxT };
     ShadowPayload pay = { 0.0f };

     uint flags = RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH |
                  RAY_FLAG_SKIP_CLOSEST_HIT_SHADER;

     TraceRay( gRtScene, flags, 0xFF, 0, 1, 0, ray, pay );
     return pay.visibility;
}
...
```

A REUSABLE SHADOW RAY

- Gives reusable shadow rays
  - — Useful in many contexts

- Like where?
  - — Maybe: want to shade this point



SHADING A DIFFUSE SURFACE

- To shade, we need:
  - Position at hit point
  - Normal at hit point
  - Material at hit point

```
float3 DiffuseShade( float3 hitPos, float3 hitNorm, float3 difColor ) {















}
```

SHADING A DIFFUSE SURFACE

- To shade, we need:
  - Position at hit point
  - Normal at hit point
  - Material at hit point
- Grab light information
  - Direction to light
  - How far away is it?

```
float3 DiffuseShade( float3 hitPos, float3 hitNorm, float3 difColor ) {
     // Get information about the light; access your framework’s scene structs
     float  distToLight    = length( gLight.position – hitPos );
     float3 dirToLight     = normalize( gLight.position – hitPos );













}
```

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

SHADING A DIFFUSE SURFACE

- To shade, we need:
  - Position at hit point
  - Normal at hit point
  - Material at hit point
- Grab light information
  - Direction to light
  - How far away is it?
- Trace our shadow ray

```
float3 DiffuseShade( float3 hitPos, float3 hitNorm, float3 difColor ) {
     // Get information about the light; access your framework’s scene structs
     float  distToLight    = length( gLight.position – hitPos );
     float3 dirToLight     = normalize( gLight.position – hitPos );

     // Shoot shadow ray with our encapsulated shadow tracing function
     float  isLit    = shootShadowRay(hitPos, dirToLight, 1.0e-4f, distToLight );









}
```

SHADING A DIFFUSE SURFACE

- To shade, we need:
  - Position at hit point
  - Normal at hit point
  - Material at hit point
- Grab light information
  - Direction to light
  - How far away is it?
- Trace our shadow ray
- Compute diffuse shading

```
float3 DiffuseShade( float3 hitPos, float3 hitNorm, float3 difColor ) {
     // Get information about the light; access your framework’s scene structs
     float  distToLight    = length( gLight.position – hitPos );
     float3 dirToLight     = normalize( gLight.position – hitPos );

     // Shoot shadow ray with our encapsulated shadow tracing function
     float  isLit    = shootShadowRay(hitPos, dirToLight, 1.0e-4f, distToLight );

     // Compute our NdotL term; shoot our shadow ray in selected direction
     float  NdotL    = saturate( dot( hitNorm, dirToLight ) );  // In range [0..1]

     // Return shaded color
     return isLit
            ? (NdotL * gLight.intensity * (difColor / M_PI) )
            : float3(0, 0, 0);
}
```

SHADING A DIFFUSE SURFACE

- To shade, we need:
  - Position at hit point
  - Normal at hit point
  - Material at hit point
- Grab light information
  - Direction to light
  - How far away is it?
- Trace our shadow ray
- Compute diffuse shading
- Want more complex material?
  - Insert different code here

```
float3 DiffuseShade( float3 hitPos, float3 hitNorm, float3 difColor ) {
     // Get information about the light; access your framework’s scene structs
     float  distToLight    = length( gLight.position – hitPos );
     float3 dirToLight     = normalize( gLight.position – hitPos );

     // Shoot shadow ray with our encapsulated shadow tracing function
     float  isLit    = shootShadowRay(hitPos, dirToLight, 1.0e-4f, distToLight );

     // Compute our NdotL term; shoot our shadow ray in selected direction
     float  NdotL    = saturate( dot( hitNorm, dirToLight ) );  // In range [0..1]

     // Return shaded color
     return isLit
            ? (NdotL * gLight.intensity * (difColor / M_PI) )
            : float3(0, 0, 0);
}
```

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?
- Encapsulate tracing a color ray

```
struct IndirectPayload {
     float3 color;     // will store ray color
};

[shader(“miss”)]
void IndirectMiss(inout IndirectPayload pay) {

}

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {



}

[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {




}

float3 shootColorRay(float3 orig, float3 dir, float minT )  {
     RayDesc         ray = { orig, minT, dir, 1.0e+38 };
     IndirectPayload pay = { float3( 0.0f ) };
     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay );
     return pay.color;
}
```

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?
- Encapsulate tracing a color ray
  - Setup a ray
  - Initialize return color to black

```
struct IndirectPayload {
    float3 color;    // will store ray color
};

[shader(“miss”)]
void IndirectMiss(inout IndirectPayload pay) {

}

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {


}

[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {


}

float3 shootColorRay(float3 orig, float3 dir, float minT )  {
    RayDesc         ray = { orig, minT, dir, 1.0e+38 };
    IndirectPayload pay = { float3( 0.0f ) };
    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay );
    return pay.color;
}
```

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?
- Encapsulate tracing a color ray
  - Setup a ray
  - Initialize return color to black
  - Trace ray, then return its color

```
struct IndirectPayload {
    float3 color;    // will store ray color
};

[shader(“miss”)]
void IndirectMiss(inout IndirectPayload pay) {

}

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {



}

[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                   BuiltinTriangleIntersectAttribs attribs) {



}

float3 shootColorRay(float3 orig, float3 dir, float minT )  {
    RayDesc        ray = { orig, minT, dir, 1.0e+38 };
    IndirectPayload pay = { float3( 0.0f ) };
    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay );
    return pay.color;
}
```

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?
- Encapsulate tracing a color ray
  - Setup a ray
  - Initialize return color to black
  - Trace ray, then return its color
  - For every hit, check transparency

```
struct IndirectPayload {
     float3 color;     // will store ray color
};

[shader(“miss”)]
void IndirectMiss(inout IndirectPayload pay) {


}

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
     if (alphaTestFails(attribs))
          IgnoreHit();
}
[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {



}

float3 shootColorRay(float3 orig, float3 dir, float minT )  {
     RayDesc          ray = { orig, minT, dir, 1.0e+38 };
     IndirectPayload pay = { float3( 0.0f ) };
     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay );
     return pay.color;
}
```

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?
- Encapsulate tracing a color ray
  - Setup a ray
  - Initialize return color to black
  - Trace ray, then return its color
  - For every hit, check transparency
  - On miss, return background

```
struct IndirectPayload {
     float3 color;     // will store ray color
};

[shader(“miss”)]
void IndirectMiss(inout IndirectPayload pay) {
     pay.color = GetBackgroundColor( WorldRayDirection() );
}

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
     if (alphaTestFails(attribs))
          IgnoreHit();
}

[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                       BuiltinTriangleIntersectAttribs attribs) {



}

float3 shootColorRay(float3 orig, float3 dir, float minT )  {
     RayDesc          ray = { orig, minT, dir, 1.0e+38 };
     IndirectPayload pay = { float3( 0.0f ) };
     TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay );
     return pay.color;
}
```

USE A SHADE FUNCTION

- Where to use `DiffuseShade()`?
- Encapsulate tracing a color ray
  - Setup a ray
  - Initialize return color to black
  - Trace ray, then return its color
  - For every hit, check transparency
  - On miss, return background
  - On closest hit, shade

```
struct IndirectPayload {
    float3 color;    // will store ray color
};

[shader(“miss”)]
void IndirectMiss(inout IndirectPayload pay) {
    pay.color = GetBackgroundColor( WorldRayDirection() );
}

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
    if (alphaTestFails(attribs))
        IgnoreHit();
}

[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                      BuiltinTriangleIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor );
}

float3 shootColorRay(float3 orig, float3 dir, float minT )  {
    RayDesc         ray = { orig, minT, dir, 1.0e+38 };
    IndirectPayload pay = { float3( 0.0f ) };
    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay );
    return pay.color;
}
```

PUTTING IT TOGETHER…

PUTTING IT TOGETHER…

- Go back to ray gen shader
  - Similar to simple one we started with

```
[shader(“raygeneration”)]
void BasicRayTracer()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     float3 pixelColor  = shootColorRay( gCamera.posW, pixelRayDir, 0.0f );

     outTex[curPixel]   = float4( pixelColor, 1.0f );
}
```

PUTTING IT TOGETHER…

- Go back to ray gen shader
  - Similar to simple one we started with
  - Get current pixel, it’s ray direction

```
[shader(“raygeneration”)]
void BasicRayTracer()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     float3 pixelColor  = shootColorRay( gCamera.posW, pixelRayDir, 0.0f );

     outTex[curPixel]   = float4( pixelColor, 1.0f );
}
```

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY - 1 AUGUST

PUTTING IT TOGETHER…

- Go back to ray gen shader
  - Similar to simple one we started with
  - Get current pixel, it’s ray direction
  - Shoot a color ray in that direction

```
[shader(“raygeneration”)]
void BasicRayTracer()  {
      uint2  curPixel    = DispatchRaysIndex().xy;
      float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

      float3 pixelColor  = shootColorRay( gCamera.posW, pixelRayDir, 0.0f );

      outTex[curPixel]   = float4( pixelColor, 1.0f );
}
```

PUTTING IT TOGETHER…

- Go back to ray gen shader
  - Similar to simple one we started with
  - Get current pixel, it’s ray direction
  - Shoot a color ray in that direction
  - Output the final result

```
[shader(“raygeneration”)]
void BasicRayTracer()  {
      uint2  curPixel     = DispatchRaysIndex().xy;
      float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

      float3 pixelColor  = shootColorRay( gCamera.posW, pixelRayDir, 0.0f );

      outTex[curPixel]   = float4( pixelColor, 1.0f );
}
```

DEMO?

- Full code, binaries, and walk through:
  - http://intro-to-dxr.cwyman.org