thrive SIGGRAPH2019 LOS ANGELES • 28 JULY – 1 AUGUST

Introduction to Real-time Ray Tracing Part 2

GOING FAST: PARALLELIZING YOUR RAY TRACER

Chris Wyman

Principal Research Scientist

NVIDIA

SOME PRELIMINARIES

Ideas needed before GPU ray tracing

YOU JUST GOT THE BASICS

YOU JUST GOT THE BASICS

- But additional features ***expected*** for GPU rendering

YOU JUST GOT THE BASICS

- But additional features ***expected*** for GPU rendering
  - Typically, increased complexity; not just a few primitives

YOU JUST GOT THE BASICS

- But additional features ***expected*** for GPU rendering
  - Typically, increased complexity; not just a few primitives
  - Render triangle meshes
  - Just collections of triangles approximating 3D shapes
  - Easy enough; intersect each triangle in turn

YOU JUST GOT THE BASICS

- But additional features ***expected*** for GPU rendering
  - Typically, increased complexity; not just a few primitives
  - Render triangle meshes
  - Just collections of triangles approximating 3D shapes
  - Easy enough; intersect each triangle in turn
  - Mesh files usually contain material information
  - Often small-scale detail stored in textures

HOW TO HANDLE MATERIALS AND TEXTURES?

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color
    - Texture coordinates
    - Material parameters
    - Et cetera

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color
    - Texture coordinates
    - Material parameters
    - Et cetera

Our texture:

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color
    - Texture coordinates
    - Material parameters
    - Et cetera

Triangle vertices have:
texture coordinates

(0,0) (1,0)

(0,1)

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color
    - Texture coordinates
    - Material parameters
    - Et cetera

Triangle vertices have:
texture coordinates

(0,0)
(1,0)

(0,1)

Coordinate here:
Interpolates coordinates at vertices

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color
    - Texture coordinates
    - Material parameters
    - Et cetera

Triangle vertices have:
texture coordinates

Coordinate here:
Interpolates coordinates at vertices

Same interpolation as position,
normal, color, etc.

Use coord to index in the image array

HOW TO HANDLE MATERIALS AND TEXTURES?

- Ray-primitive intersection
  - Not just binary: Did we hit? Yes / No
  - Also need to store ***attributes*** at the hit point, e.g.:
    - Positions
    - Normal
    - Color
    - Texture coordinates
    - Material parameters
    - Et cetera
  - All attribute interpolation work the same way

BASICS OF OPTIMIZATION

Before jumping to GPU, take some baby steps

BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics

BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

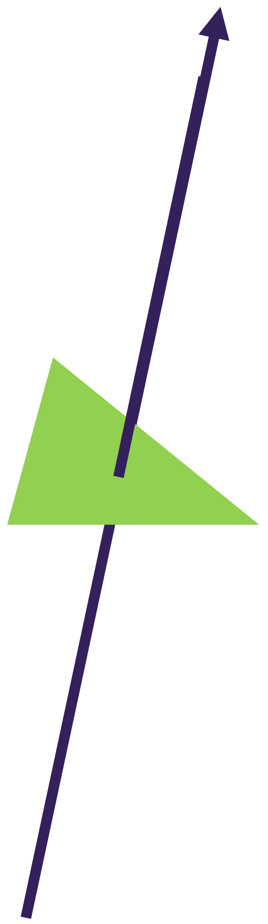
- Consider basic ray tracing algorithm:
  - Take a ray through your scene

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

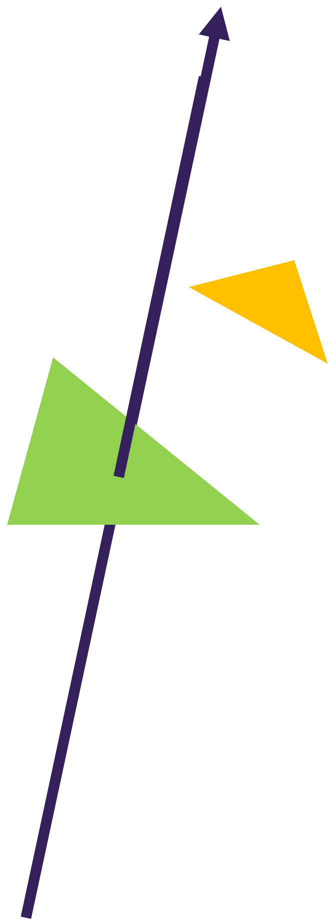
- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

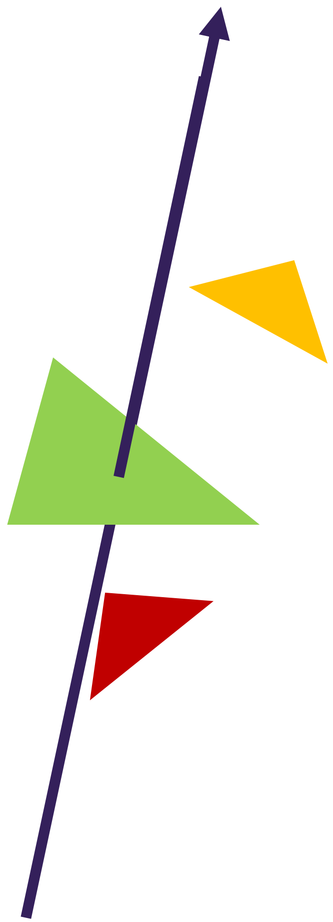
- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection
    - Repeat



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

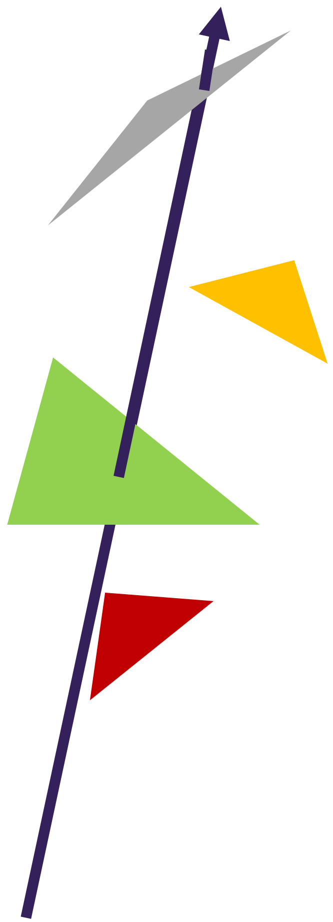
- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection
    - Repeat



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

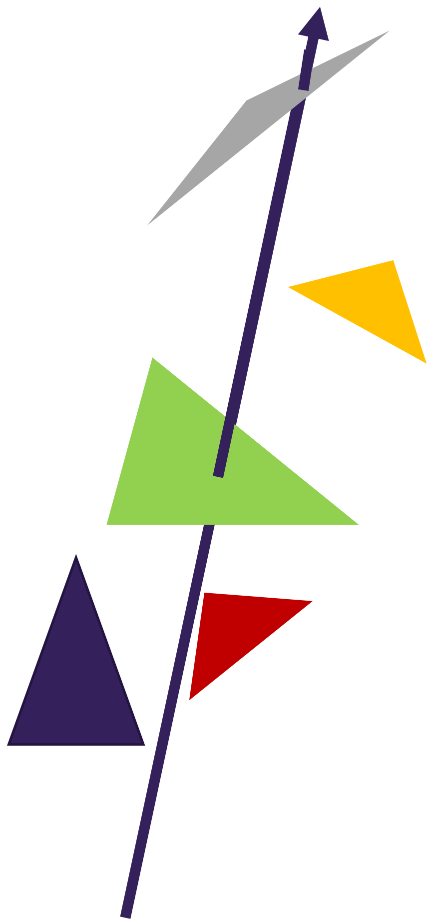
- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection
    - Repeat



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

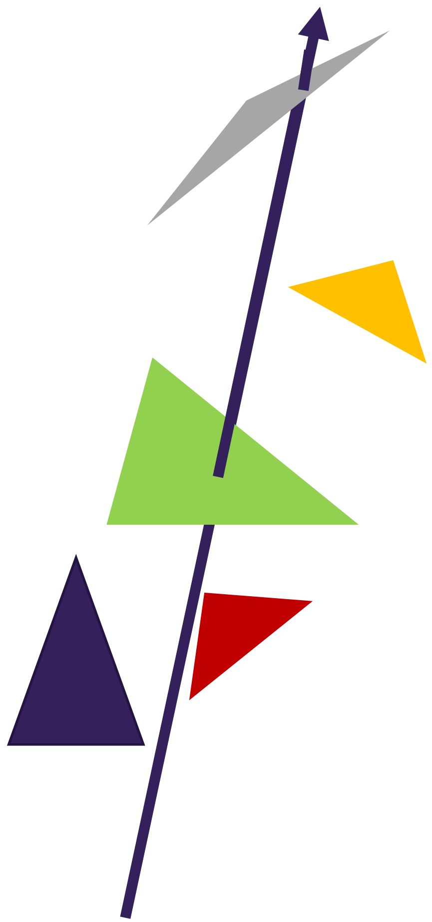
- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection
    - Repeat



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - — Why is tracing rays slow at all?

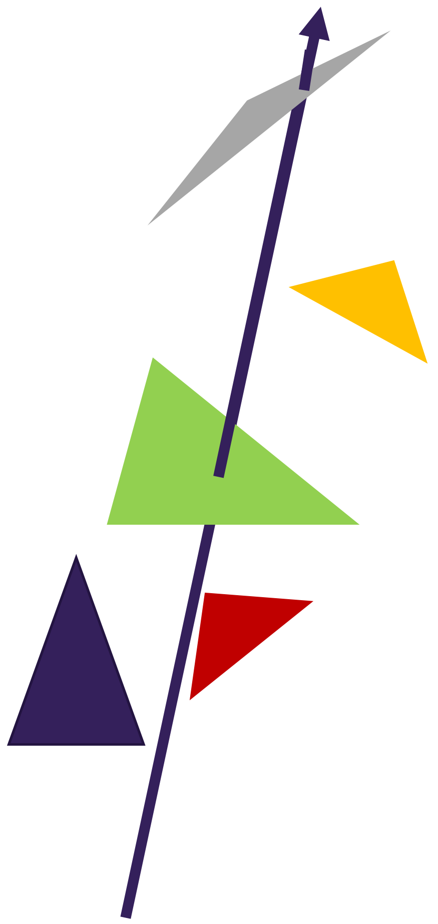
- Consider basic ray tracing algorithm:
  - — Take a ray through your scene
  - — Test triangle to find intersection
    - Repeat
  - — How do you know when you're done?



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection
    - Repeat
  - How do you know when you're done?
    - When you've tested every triangle?



BEFORE DIVING INTO PARALLELIZATION…

- Need to talk about some performance basics
  - Why is tracing rays slow at all?

- Consider basic ray tracing algorithm:
  - Take a ray through your scene
  - Test triangle to find intersection
    - Repeat
  - How do you know when you're done?
    - When you've tested every triangle?
    - Very expensive…
    - Every ray could test, 1 million (or more) triangles

WHAT’S OUR COMPUTATION BUDGET?

WHAT'S OUR COMPUTATION BUDGET?

- Let's be easy on ourselves:
  - Target just 1920 x 1080 at 60 fps

WHAT'S OUR COMPUTATION BUDGET?

- Let's be easy on ourselves:
  - Target just 1920 x 1080 at 60 fps
  - We need 125 million pixels per second!

WHAT'S OUR COMPUTATION BUDGET?

- Let's be easy on ourselves:
  - Target just 1920 x 1080 at 60 fps
  - We need 125 million pixels per second!

- With a ~10 TFLOP state-of-the-art GPU
  - If tracing one ray per pixel…
  - About 80,000 flops per ray

WHAT’S OUR COMPUTATION BUDGET?

- Let’s be easy on ourselves:
  - Target just 1920 x 1080 at 60 fps
  - We need 125 million pixels per second!

- With a ~10 TFLOP state-of-the-art GPU
  - If tracing one ray per pixel…
  - About 80,000 flops per ray

- An optimized triangle intersection: ~10 flops
  - Can afford ***at most*** 8,000 intersections per ray

WHAT'S OUR COMPUTATION BUDGET?

- Let's be easy on ourselves:
  - Target just 1920 x 1080 at 60 fps
  - We need 125 million pixels per second!

- With a ~10 TFLOP state-of-the-art GPU
  - If tracing one ray per pixel…
  - About 80,000 flops per ray

- An optimized triangle intersection: ~10 flops
  - Can afford ***at most*** 8,000 intersections per ray

- Conclusion: ***Don't test every triangle!***

KEY PRINCIPAL TO OPTIMIZATION:

- Make the ***common*** case fast

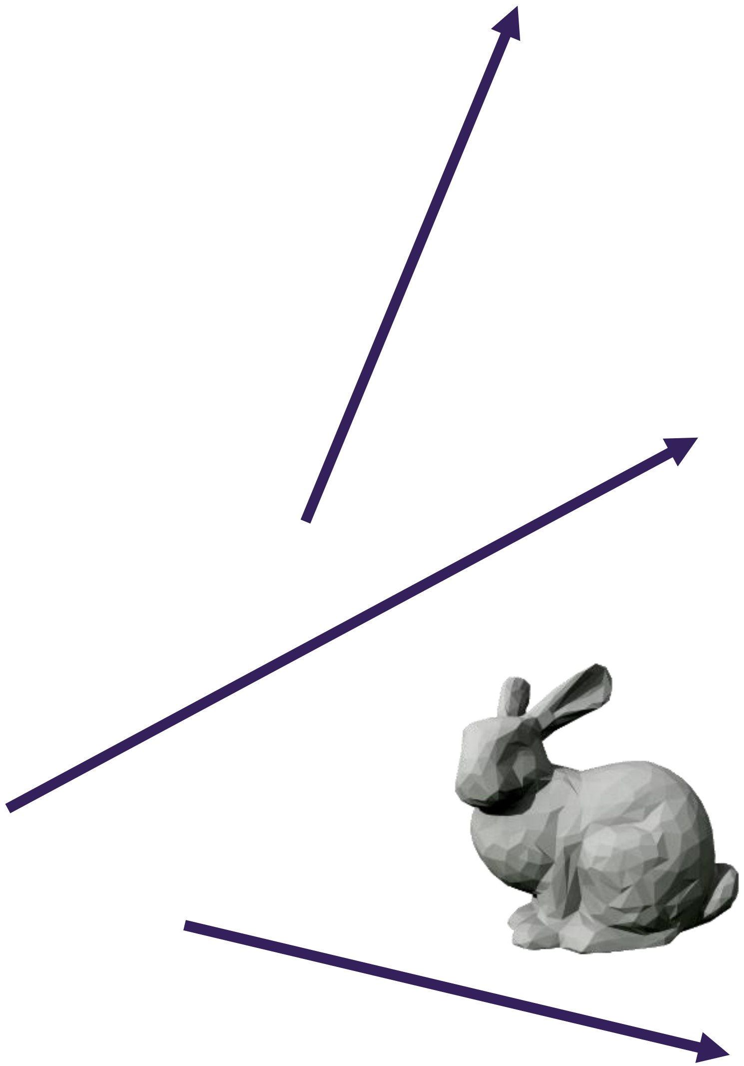
KEY PRINCIPAL TO OPTIMIZATION:

- Make the ***common*** case fast
- Common case in ray tracing?
  - Ray does not intersect a triangle

KEY PRINCIPAL TO OPTIMIZATION:

- Make the ***common*** case fast

- Common case in ray tracing?
  - Ray does not intersect a triangle
  - For any mesh, ray typically misses mesh

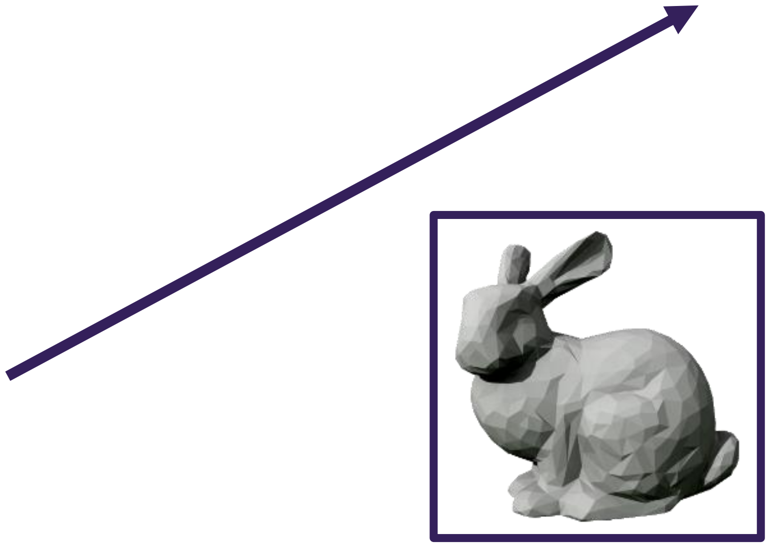


KEY PRINCIPAL TO OPTIMIZATION:

- Make the ***common*** case fast

- Common case in ray tracing?
  - Ray does not intersect a triangle
  - For any mesh, ray typically misses mesh

- Perhaps:
  - First intersect a mesh bounding box

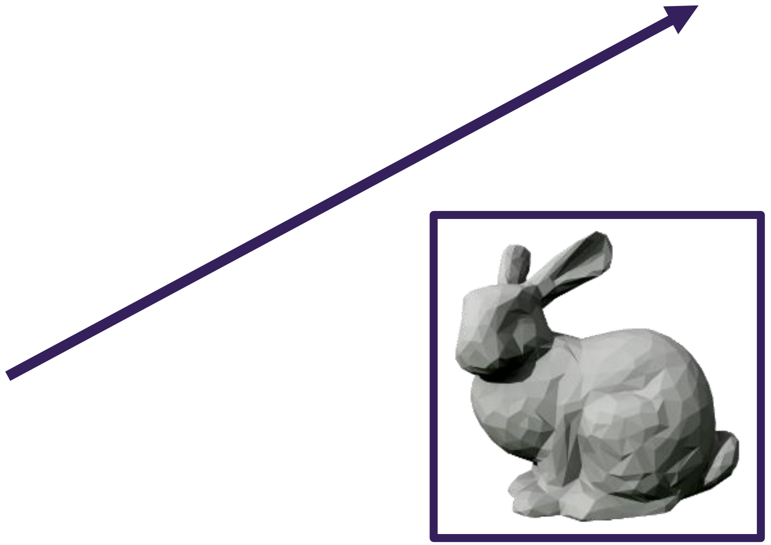


KEY PRINCIPAL TO OPTIMIZATION:

- Make the ***common*** case fast

- Common case in ray tracing?
  - Ray does not intersect a triangle
  - For any mesh, ray typically misses mesh

- Perhaps:
  - First intersect a mesh bounding box
  - Most rays avoid testing thousands of triangles
  - But, extra box test when hit bunny

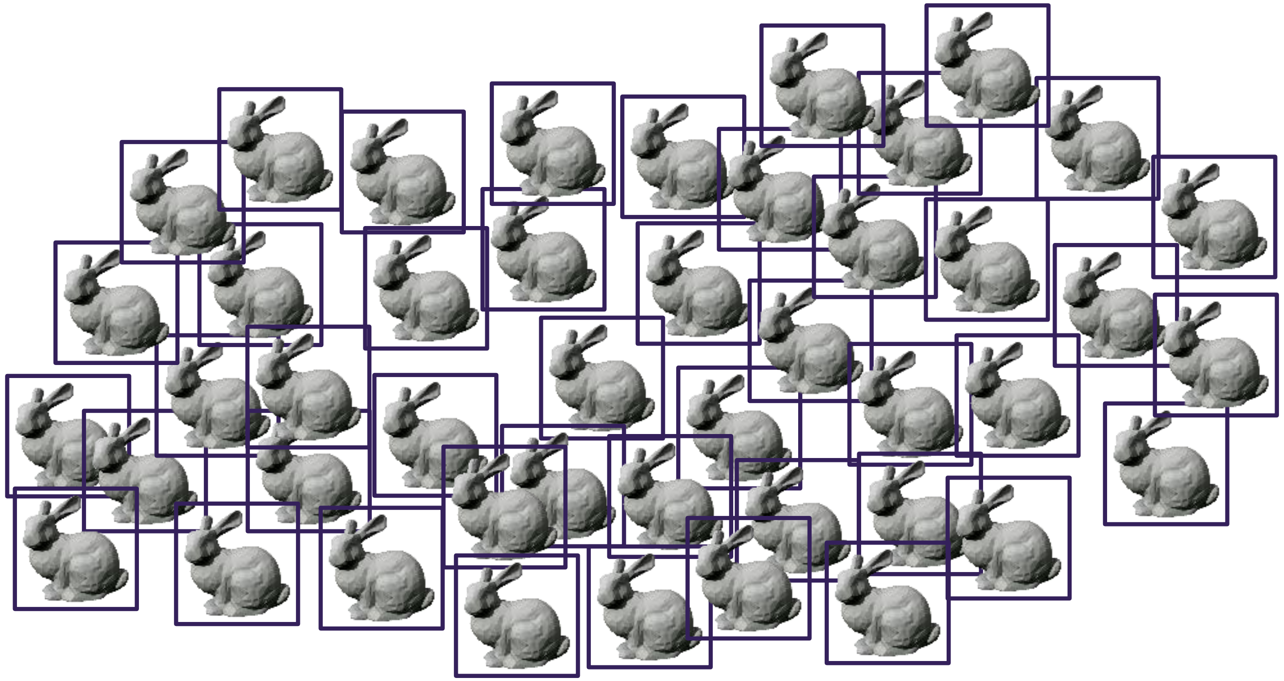


KEY PRINCIPAL TO OPTIMIZATION:

- What if you have thousands of bunnies?

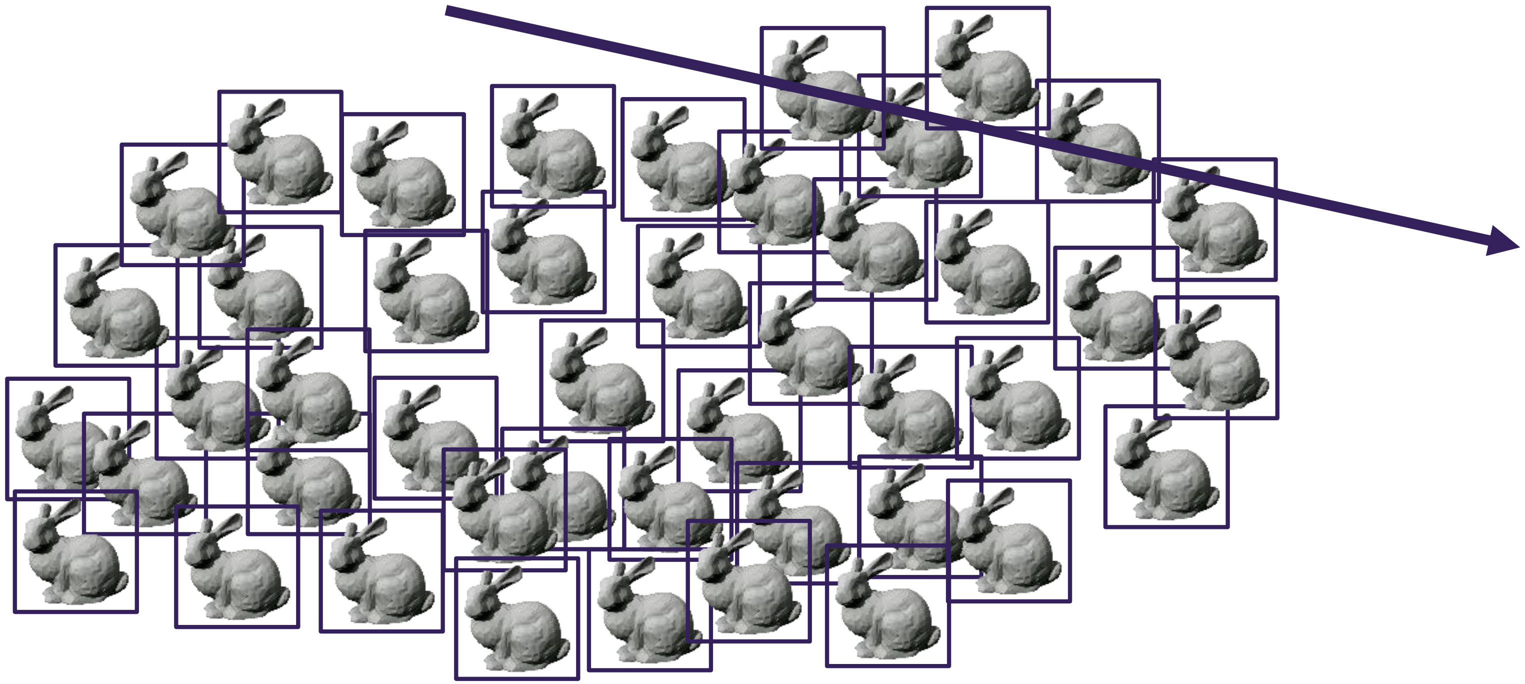
KEY PRINCIPAL TO OPTIMIZATION:

- What if you have thousands of bunnies?



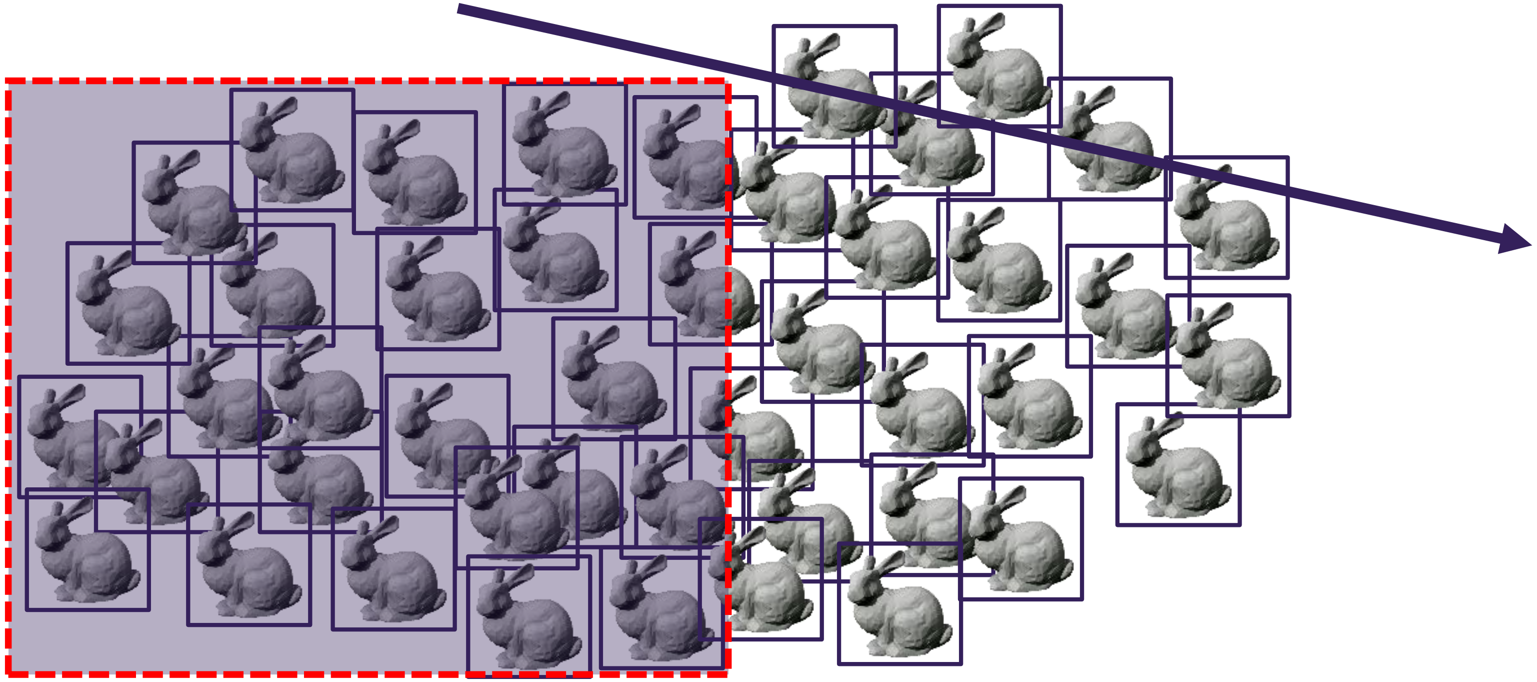
KEY PRINCIPAL TO OPTIMIZATION:

- What if you have thousands of bunnies?
  - Common case: Ray misses most bunnies



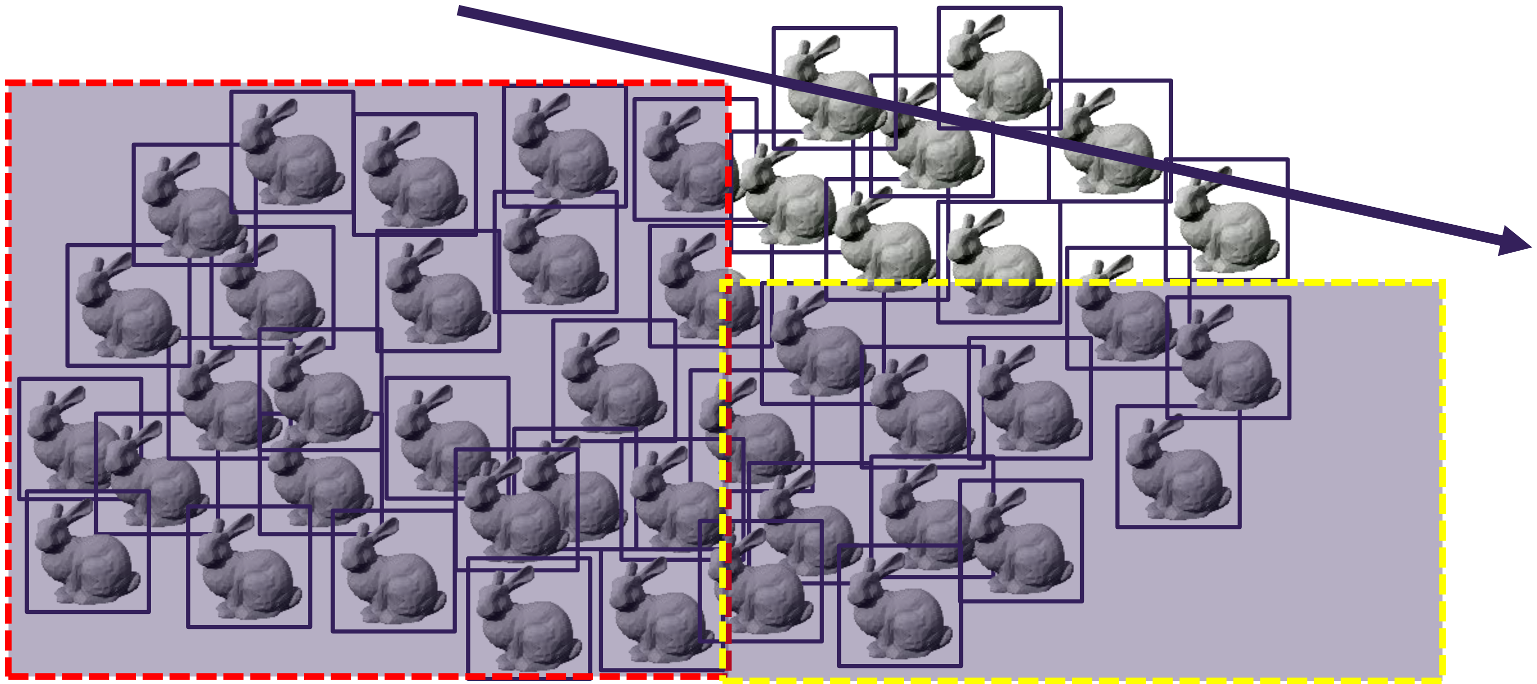
KEY PRINCIPAL TO OPTIMIZATION:

- What if you have thousands of bunnies?
  - Common case: Ray misses most bunnies
  - Can skip testing this half…



KEY PRINCIPAL TO OPTIMIZATION:

- What if you have thousands of bunnies?
  - Common case: Ray misses most bunnies
  - Can skip testing this half… and this quarter… with a few more boxes



KEY PRINCIPAL TO OPTIMIZATION:

- Build a tree of bounding boxes
  - Known as a “bounding volume hierarchy” or BVH

KEY PRINCIPAL TO OPTIMIZATION:

- Build a tree of bounding boxes
  - Known as a “bounding volume hierarchy” or BVH

- When using a principled tree build
  - Reduces number of required intersections
  - From O(N) to O(log N)

KEY PRINCIPAL TO OPTIMIZATION:

- Build a tree of bounding boxes
  - Known as a “bounding volume hierarchy” or BVH
- When using a principled tree build
  - Reduces number of required intersections
  - From O(N) to O(log N)
- With a binary tree, 1 million ray-triangle tests becomes:
  - Around 20 ray-box tests
  - A few ray-triangle tests in leaf nodes

KEY PRINCIPAL TO OPTIMIZATION:

- Production ray tracers ***always*** use some acceleration structure

KEY PRINCIPAL TO OPTIMIZATION:

- Production ray tracers ***always*** use some acceleration structure

- But, ***which*** structure? How do you best build it?
  - Literally decades of research

KEY PRINCIPAL TO OPTIMIZATION:

- Production ray tracers ***always*** use some acceleration structure

- But, ***which*** structure? How do you best build it?
  - Literally decades of research
  - Continuing to today (e.g., “Wide BVH Traversal with a Short Stack,” Vaidyanathan et al. 2019)

KEY PRINCIPAL TO OPTIMIZATION:

- Production ray tracers ***always*** use some acceleration structure
- But, ***which*** structure? How do you best build it?
  - Literally decades of research
  - Continuing to today (e.g., "Wide BVH Traversal with a Short Stack," Vaidyanathan et al. 2019)
- When starting real-time ray tracing, best bet:
  - Use someone else's code
  - Quality of your BVH easily affects performance by 2x, 3x, or >10x
    - Varies per scene!
  - Luckily most APIs will build structure

GOING PARALLEL

Coding for massively parallel GPUs

RAY TRACING: EMBARRASSINGLY PARALLEL

- Defined: Little to no effort needed to separate into parallel tasks

RAY TRACING: EMBARRASSINGLY PARALLEL

- Defined: Little to no effort needed to separate into parallel tasks

- Rendering often a prototypical example of *embarrassingly parallel*
  - One obvious way: assign one CPU or GPU core per pixel

RAY TRACING: EMBARRASSINGLY PARALLEL

- On CPU, call fork() or spawn() to create multiple threads
  - Each thread works on separate pixels
  - Wait for all threads to complete
  - Some threads take longer → may need load balancing

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2        curPixel = DispatchRaysIndex().xy;
   RayDesc      ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload   payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]   = float4( payload.rayColor, 1.0f );
}
```

Identify the current pixel

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2        curPixel = DispatchRaysIndex().xy;
   RayDesc      ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload   payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]   = float4( payload.rayColor, 1.0f );
}
```

Setup the ray

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  — Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Initialize ray return values

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Trace your ray

RAY TRACING: EMBARRASSINGLY PARALLEL

- GPU programming model hides thread spawning and load-balancing
  - Code *appears* serial, but you have access to current pixel index

```
// Simple DirectX-like ray tracing shader
RWTexture<float4> rayColors;

[shader("raygeneration")]
void SimpleRayTracer() {
   uint2       curPixel = DispatchRaysIndex().xy;
   RayDesc     ray      = { GetRayOrigin(curPixel), 0.0f, GetRayDir(curPixel), 1e+38f };
   RayPayload  payload  = { float3(0, 0, 0) };
   TraceRay( ..., ray, payload );
   rayColors[curPixel]  = float4( payload.rayColor, 1.0f );
}
```

Output your results

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - ***Not*** the same as getting best performance

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - *Not* the same as getting best performance
- Many performance considerations:
  - # intersections, data structures, coherence, caching, load-balancing, SIMD

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - ***Not*** the same as getting best performance
- Many performance considerations:
  - # intersections, data structures, coherence, caching, load-balancing, SIMD
- APIs can leverage best-known methods behind your back

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - ***Not*** the same as getting best performance
- Many performance considerations:
  - # intersections, data structures, coherence, caching, load-balancing, SIMD
- APIs can leverage best-known methods behind your back
- APIs allow you to shoot yourself in the foot without knowing it

DOING IT YOURSELF V.S. EXISTING APIS

- Embarrassingly parallel ≠ easy to parallelize
  - ***Not*** the same as getting best performance
- Many performance considerations:
  - # intersections, data structures, coherence, caching, load-balancing, SIMD
- APIs can leverage best-known methods behind your back
- APIs allow you to shoot yourself in the foot without knowing it
- APIs come at many levels (e.g., use of CUDA without ray tracing API)

SOME RAY TRACING APIS

- Hardware vendor specific:
  - OptiX, Embree, FireRays
- Cross-vendor APIs:
  - DirectX Raytracing, Vulkan RT
- Game engine APIs:
  - Unity, Unreal
- Different:
  - Audiences, learning curves, flexibility, performance, built-in optimizations

TODAY: USING DIRECTX FOR SAMPLE CODE

- Why?
  - DirectX widely used API for interactive graphics
  - Similar to Vulkan model
  - Abstracts some bits tricky for novices’ ray tracers
  - Tutorial frameworks for easy experimentation

DIRECTX RAY TRACING RESOURCES

- Some DirectX Ray Tracing tutorials:
  - Tutorial framework that hides the C++ API (http://intro-to-dxr.cwyman.org)
    - Easy to get started, not targeted at optimal performance
    - Used for my sample code today
    - Builds on Falcor for abstraction
  - Lower-level tutorial covering DirectX API
    - From the "Introduction to DirectX Ray Tracing" *Ray Tracing Gems* article
  - A simple getting started blog post
  - Microsoft's DXR samples
    - A DirectX Raytracing functional specification

WE'LL FOCUS ON GPU SHADER CODE

- Why?
  - Focus on tracing rays, identifying where to trace rays
  - Where interesting rendering algorithms mostly live

WE'LL FOCUS ON GPU SHADER CODE

- Why?
  - Focus on tracing rays, identifying where to trace rays
  - Where interesting rendering algorithms mostly live

- The CPU has vital infrastructure…
  - But it's largely reusable stuff like asset loaders
  - Not interesting (to me) to re-write

WE'LL FOCUS ON GPU SHADER CODE

- Why?
  - Focus on tracing rays, identifying where to trace rays
  - Where interesting rendering algorithms mostly live
- The CPU has vital infrastructure…
  - But it's largely reusable stuff like asset loaders
  - Not interesting (to me) to re-write
- For parallel GPU ray tracer, CPU code is mostly glue:
  - Pass configuration and data to GPU
  - Launch GPU processes

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*

thrive SIGGRAPH2019 LOS ANGELES • 28 JULY – 1 AUGUST

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

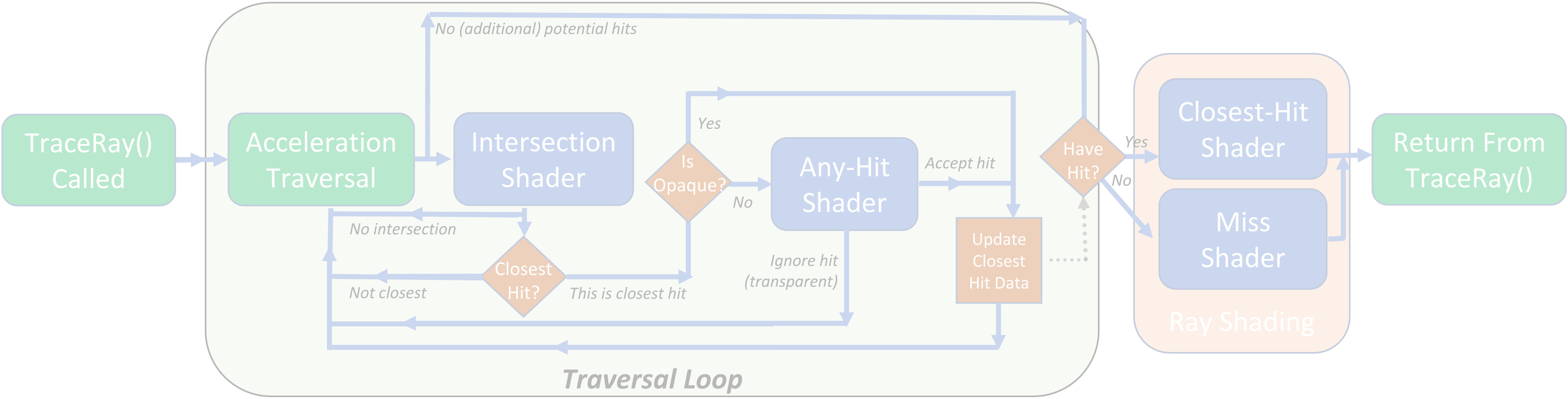
- Ray tracing pipeline split into ***five*** shaders:
  - *A* ***ray generation shader*** — *define how to start tracing rays*
  - ***Intersection shader(s)*** — *define how rays intersect geometry*
  - ***Miss shader(s)*** — *shading for when rays miss geometry*
  - ***Closest-hit shader(s)*** — *shading 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*** ← Controls other shaders
  - — ***Intersection shader(s)*** ← Defines object shapes (one shader per type)
  - — ***Miss shader(s)***
  - — ***Closest-hit shader(s)***
  - — ***Any-hit shader(s)***

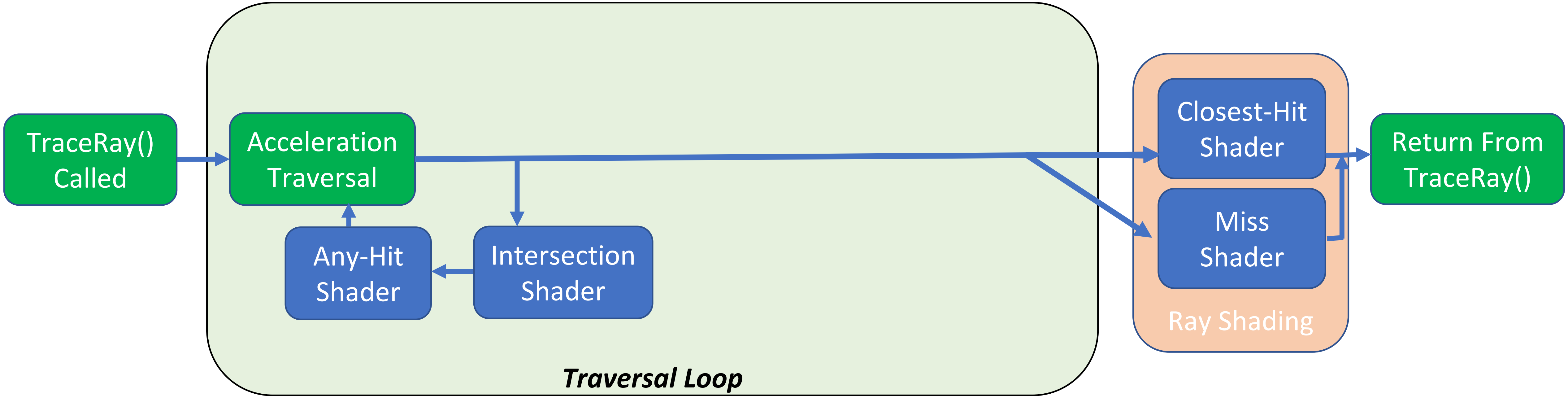
← Controls per-ray behavior (often many types)

HOW DO THESE FIT TOGETHER? [EYE CHART VERSION]



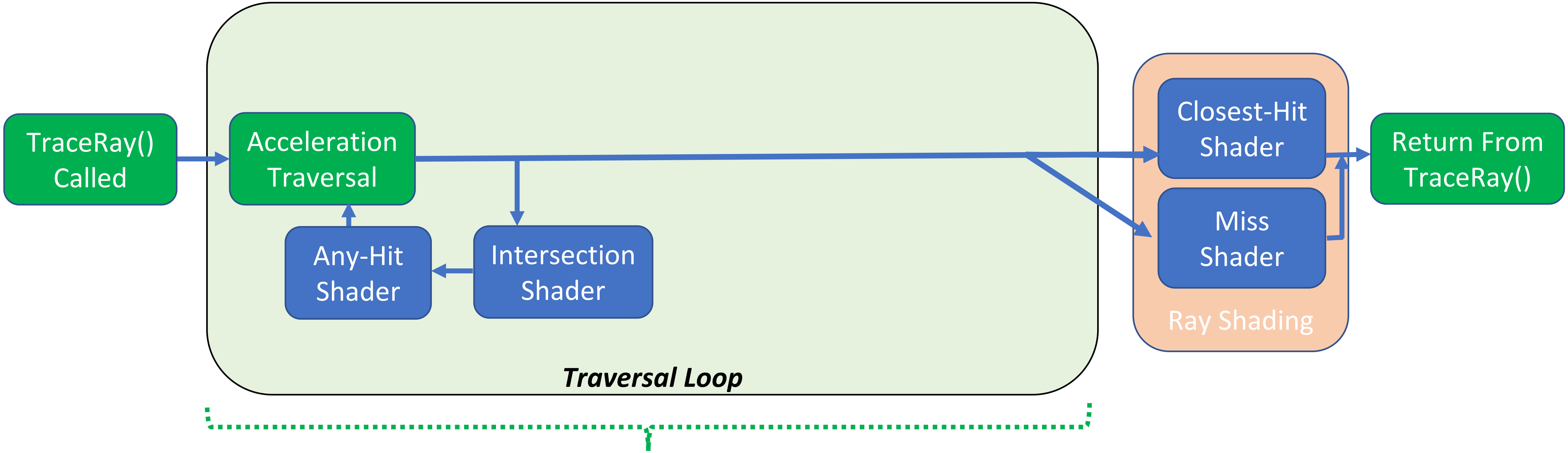
HOW DO THESE FIT TOGETHER? [LOGICAL VERSION]

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



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

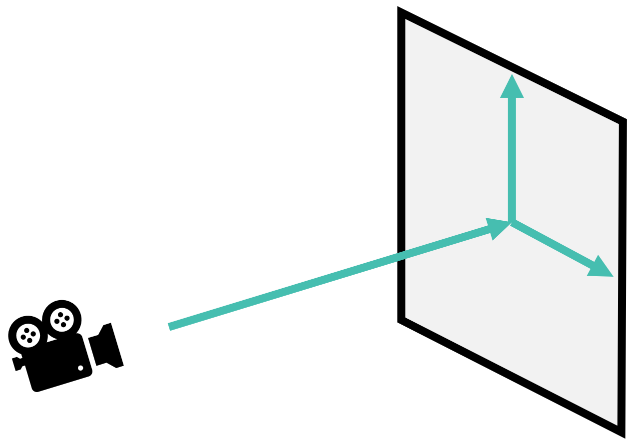
```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };
```

```
[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; };
```

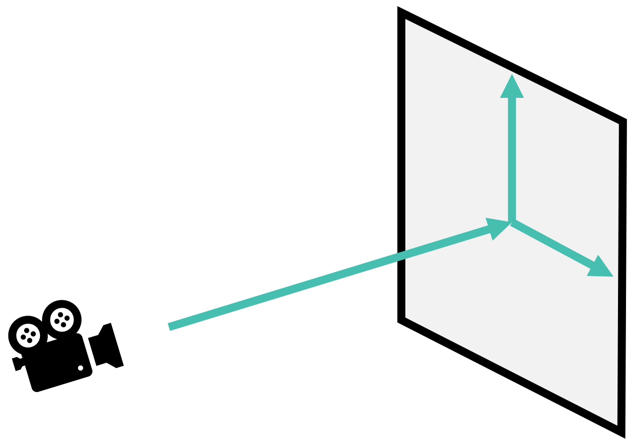


```
[shader("raygeneration")]
void MyRayGen()  {
     uint2  curPixel    = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     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; };
```

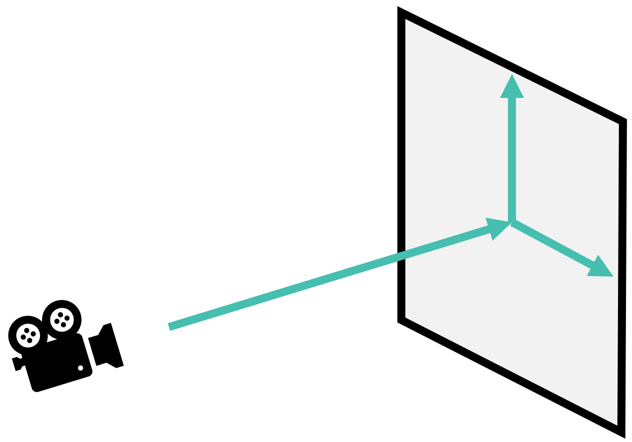


```
[shader("raygeneration")]
void MyRayGen()  {
     uint2  curPixel     = DispatchRaysIndex().xy;
     float3 pixelRayDir = normalize( getRayDirFromPixelID( curPixel ) );

     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

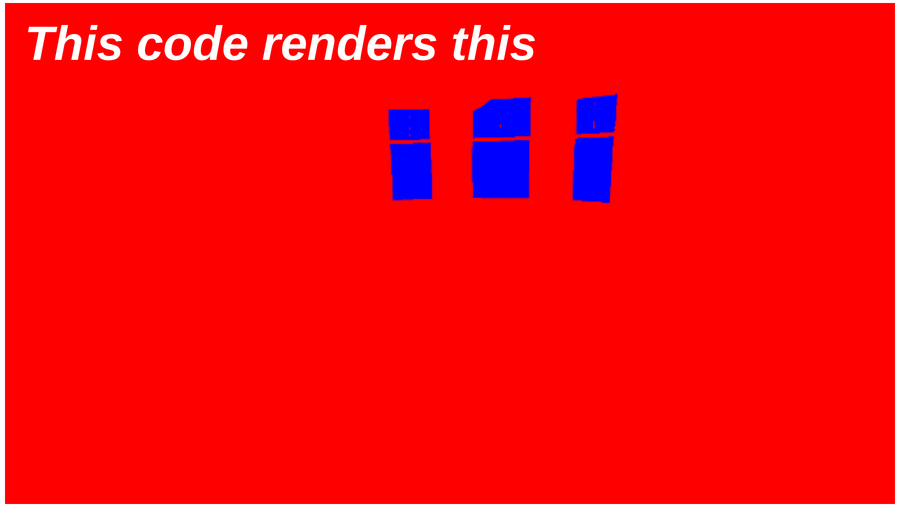
```
RWTexture<float4> gOutTex;
struct RayPayload { float3 color; };

[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




```
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?

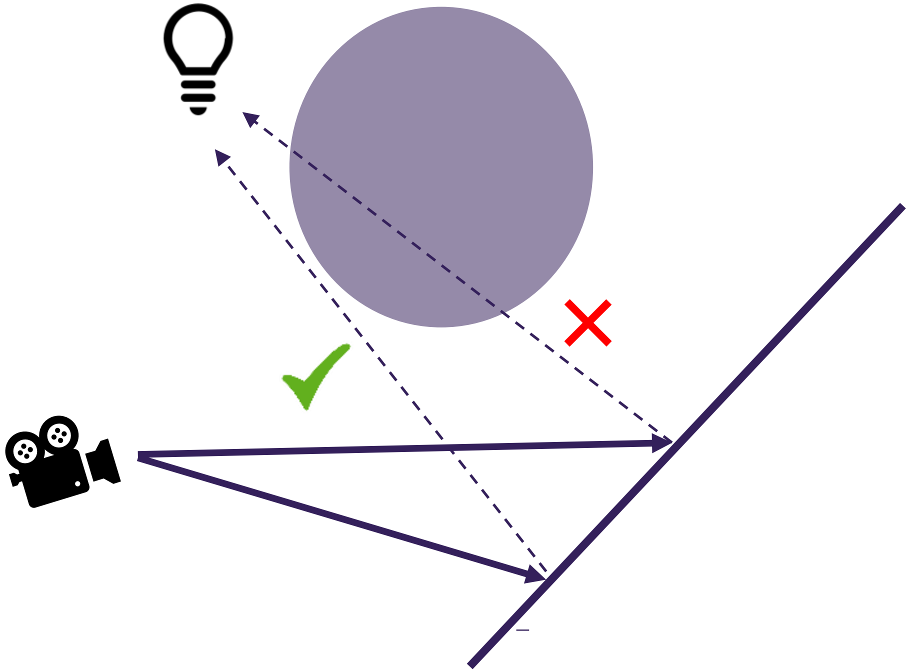
- Examples from my DXR tutors: http://intro-to-dxr.cwyman.org
  - Click on “code walkthrough”
  - Not quite equivalent to any of those, but close

WHAT ABOUT A REAL EXAMPLE?

- How about adding shadows?

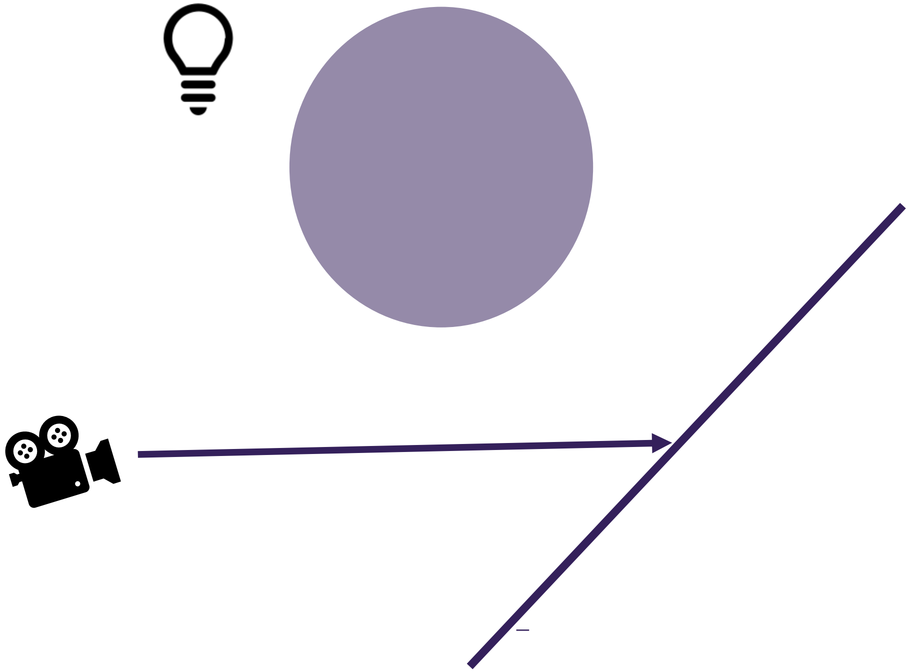
WHAT ABOUT A REAL EXAMPLE?

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



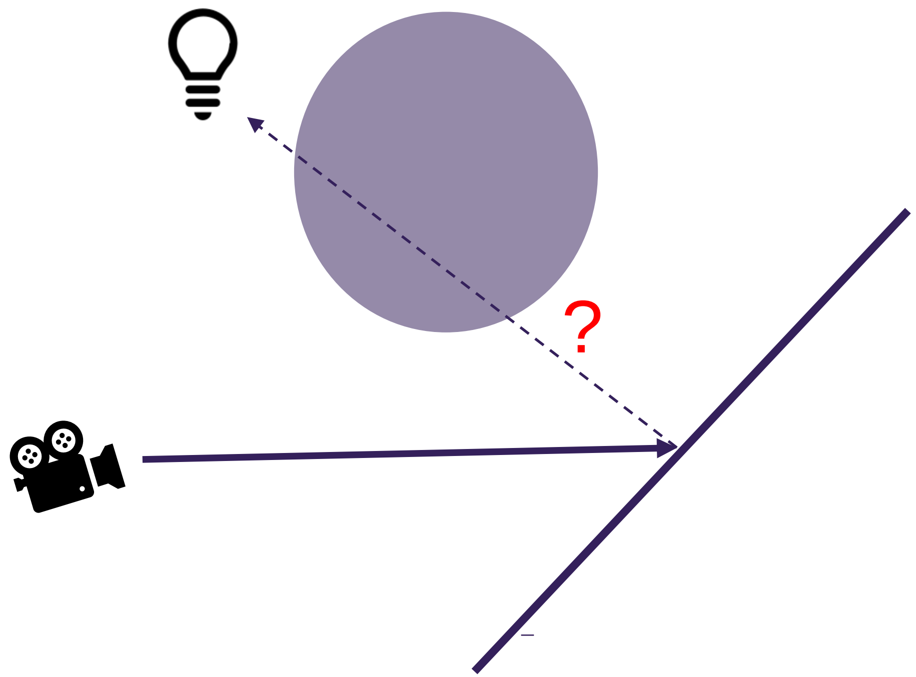
HOW DOES THIS WORK?

- Trace a ray from the camera



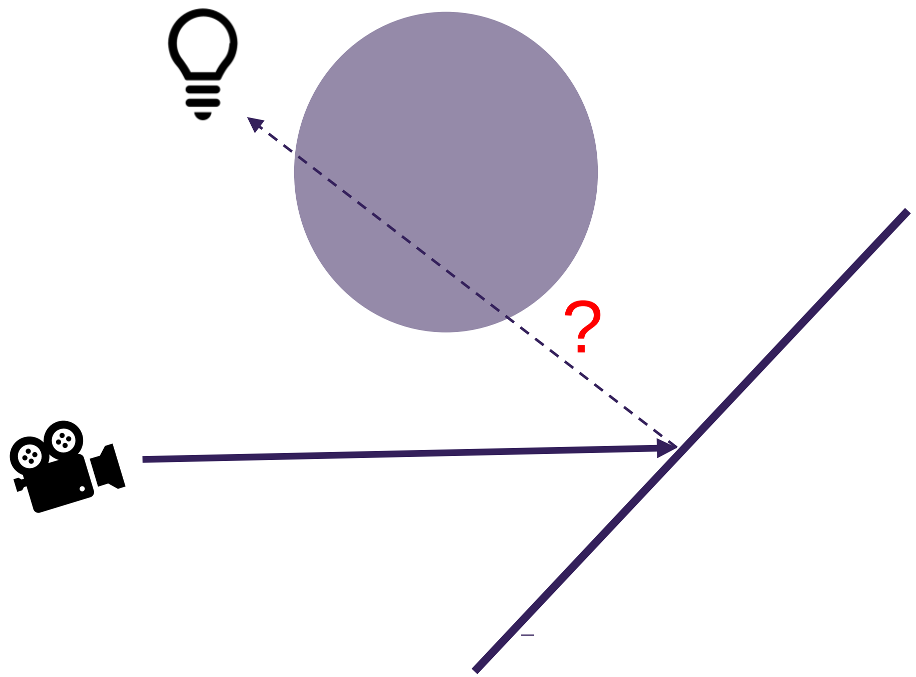
HOW DOES THIS WORK?

- 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)  {




}
...
```

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}…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 $[t_{min} \ldots t_{max}]$
  - **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 };
}
...
```

A REUSABLE SHADOW RAY

- Encapsulate a shadow ray
  - Create a ray
    - From some origin
    - In some direction
    - Check occlusions in $[t_{min}\dots t_{max}]$
  - **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 $[t_{min}\dots t_{max}]$
  - **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;
}
...
```

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

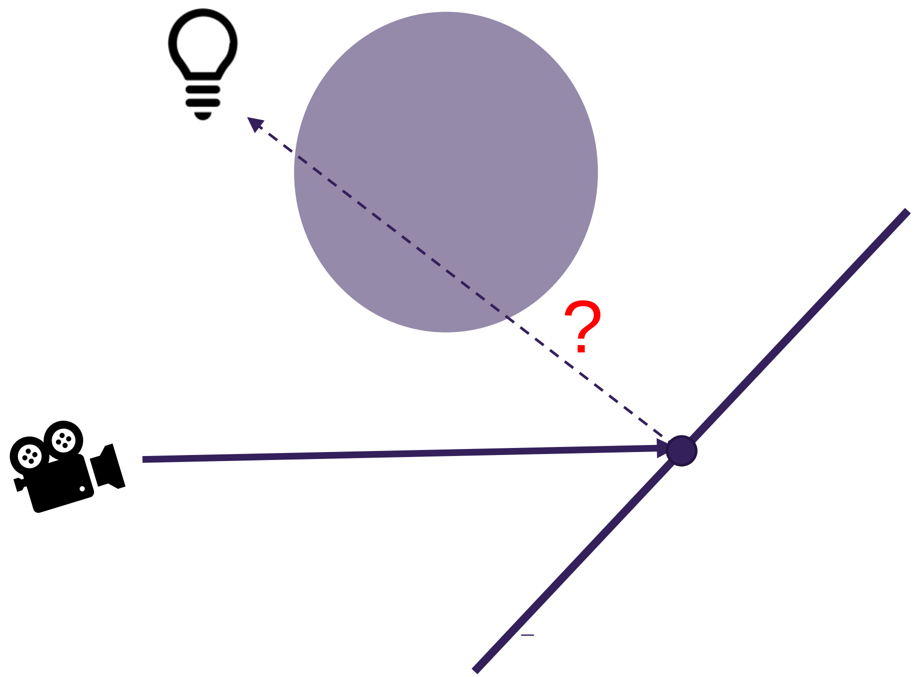
- Gives reusable shadow rays
  - Useful in many contexts

```
...
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 );








}
```

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 );
}
```

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

GOING FURTHER

More complex materials, multi-bounce lighting, etc.

GOING FURTHER

- Take code for color ray & tweak

```
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;
}
```

GOING FURTHER

- Take code for color ray & tweak
  - Mostly here:

```
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;
}
```

GOING FURTHER

- Want global illumination?

```
[shader("closesthit")]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    float3 directLight = DiffuseShade( hit.pos, hit.norm, hit.difColor );




}
```

GOING FURTHER

- Want global illumination?
  - Add a random outgoing ray
  - Recursive call: `shootColorRay()`
  - Account for BRDF
  - Add contributions together

- A basic ***path tracer***

```
[shader("closesthit")]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    float3 directLight = DiffuseShade( hit.pos, hit.norm, hit.difColor );

    float3 bounceDir = selectRandomDirection();
    float3 indirectColor = shootColorRay( hit.pos, bouncDir );
    float3 indirectLight = DiffuseIndirect( bounceDir, indirectColor );

    pay.color = directLight + indirectLight;
}
```

GOING FURTHER

- Want global illumination?
  - Add a random outgoing ray
  - Recursive call: `shootColorRay()`
  - Account for BRDF
  - Add contributions together
- A basic ***path tracer***
  - Usually encapsulate BRDF
  - Direct light done with BRDF::evaluate()

```
[shader("closesthit")]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    float3 directLight = DiffuseShade( hit.pos, hit.norm, hit.difColor );

    float3 bounceDir = selectRandomDirection();
    float3 indirectColor = shootColorRay( hit.pos, bouncDir );
    float3 indirectLight = DiffuseIndirect( bounceDir, indirectColor );

    pay.color = directLight + indirectLight;
}
```

GOING FURTHER

- Want global illumination?
  - Add a random outgoing ray
  - Recursive call: `shootColorRay()`
  - Account for BRDF
  - Add contributions together

```
[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    float3 directLight = DiffuseShade( hit.pos, hit.norm, hit.difColor );

    float3 bounceDir = selectRandomDirection();
    float3 indirectColor = shootColorRay( hit.pos, bouncDir );
    float3 indirectLight = DiffuseIndirect( bounceDir, indirectColor );

    pay.color = directLight + indirectLight;
}
```

- A basic ***path tracer***
  - Usually encapsulate BRDF
  - Direct light done with `BRDF::evaluate()`
  - Indirect done with `BRDF::scatter()`
    - Also sometimes called `sample()`

GOING FURTHER

- Want global illumination?
  - Add a random outgoing ray
  - Recursive call: `shootColorRay()`
  - Account for BRDF
  - Add contributions together

```
[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay,
                        BuiltinTriangleIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    float3 directLight = DiffuseShade( hit.pos, hit.norm, hit.difColor );

    float3 bounceDir = selectRandomDirection();
    float3 indirectColor = shootColorRay( hit.pos, bouncDir );
    float3 indirectLight = DiffuseIndirect( bounceDir, indirectColor );

    pay.color = directLight + indirectLight;
}
```

- A basic ***path tracer***
  - Usually encapsulate BRDF
  - Direct light done with `BRDF::evaluate()`
  - Indirect done with `BRDF::scatter()`
    - Also sometimes called `sample()`

- Makes it easy to plug in new materials

MANY LIGHTS?

MANY LIGHTS?

- Don't just evaluate BRDF for one light

```
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);
}
```

MANY LIGHTS?

- Don’t just evaluate BRDF for one light
  - Loop per light

```
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);
}
```

MANY LIGHTS?

- Don’t just evaluate BRDF for one light
  - Loop per light
- Thousands of lights? Becomes expensive

```
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);
}
```

MANY LIGHTS?

- Don’t just evaluate BRDF for one light
  - Loop per light

- Thousands of lights? Becomes expensive
- What if: emissive triangles, spheres, bunnies?

MANY LIGHTS?

- Don't just evaluate BRDF for one light
  - Loop per light

- Thousands of lights? Becomes expensive
- What if: emissive triangles, spheres, bunnies?

- Need to *sample* your lights
  - Pick a random location on some light
  - Evaluate direct lighting from that point

NAÏVE LIGHT SAMPLING:

- Lots of point lights (e.g., N points):
  - Randomly pick number in [1…N], use that light for shading

NAÏVE LIGHT SAMPLING:

- Lots of point lights (e.g., N points):
  - Randomly pick number in [1…N], use that light for shading
- One surface light:
  - Pick a point uniformly over the surface
  - E.g., on a quad, pick both (u, v) randomly in [0…1]

NAÏVE LIGHT SAMPLING:

- Lots of point lights (e.g., N points):
  - Randomly pick number in [1…N], use that light for shading
- One surface light:
  - Pick a point uniformly over the surface
  - E.g., on a quad, pick both (u, v) randomly in [0…1]
- For many emissive surfaces (e.g., N surfaces):
  - First pick number in [1…N], then pick random point on surface

NAÏVE LIGHT SAMPLING:

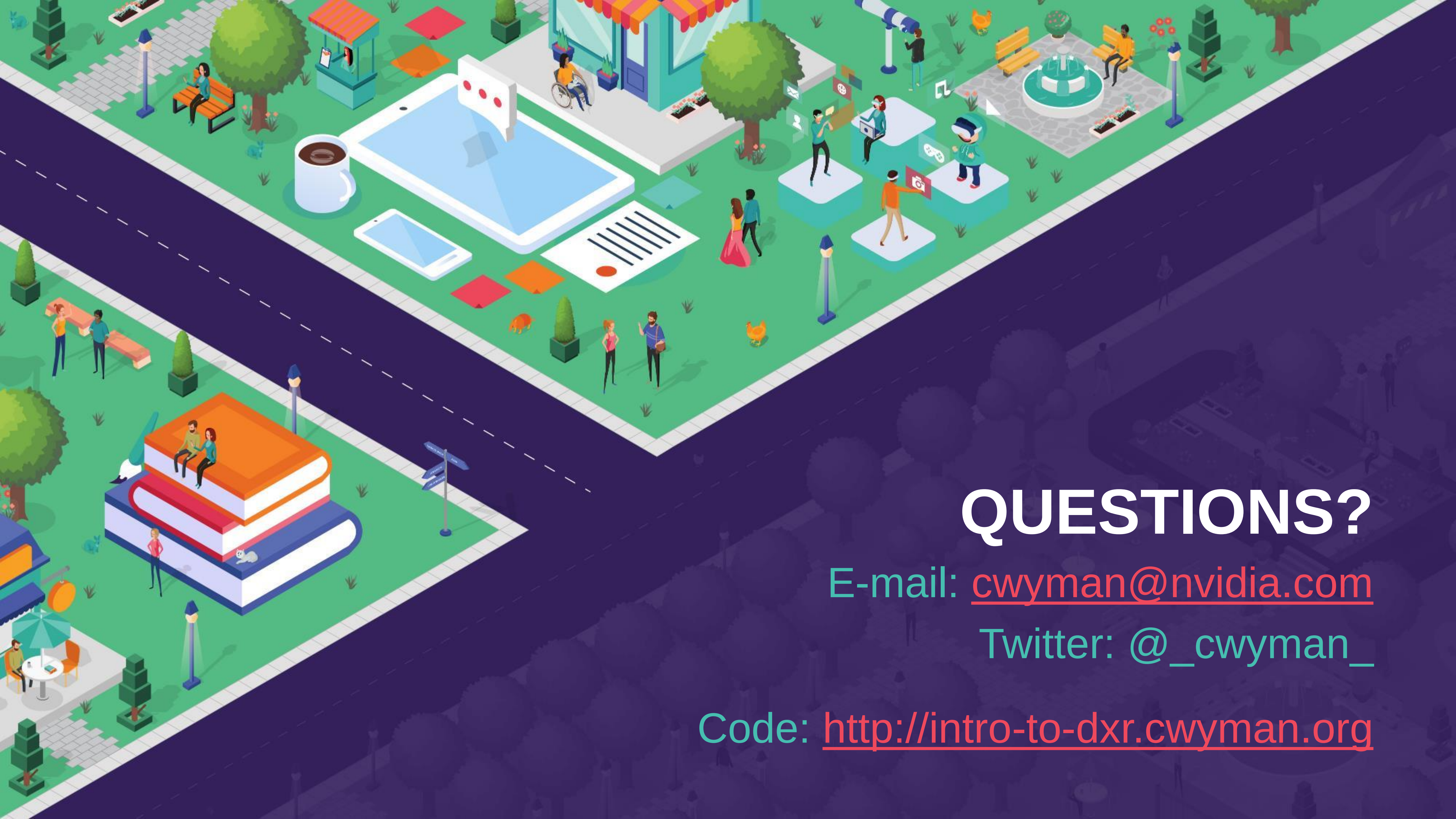
- Lots of point lights (e.g., N points):
  - Randomly pick number in [1…N], use that light for shading
- One surface light:
  - Pick a point uniformly over the surface
  - E.g., on a quad, pick both (u, v) randomly in [0…1]
- For many emissive surfaces (e.g., N surfaces):
  - First pick number in [1…N], then pick random point on surface
  - Alternatively weight choice of light based on area

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

UP NEXT

Morgan McGuire

With more on materials, sampling, and how to think about GPU ray tracing performance



QUESTIONS?

E-mail: cwyman@nvidia.com

Twitter: @_cwyman_

Code: http://intro-to-dxr.cwyman.org