Tutorials: Build a Path Tracer Step-by-Step

(Or Learning DirectX HLSL by example)

More information: http://intro-to-dxr.cwyman.org
Tutorial Goals

• Provide a simple abstraction to get you started
  – Allow easy startup, learning, and experimentation
Tutorial Goals

- Provide a simple abstraction to get you started
  - Allow easy startup, learning, and experimentation
- Provide code you can use as a starting point
  - See intro-to-dxr.cwyman.org
Tutorial Goals

- Provide a simple abstraction to get you started
  - Allow easy startup, learning, and experimentation
- Provide code you can use as a staring point
  - See intro-to-dxr.cwyman.org
- Show that using DirectX Raytracing can be straightforward
  - Once you get the low-level initialization and setup working
Tutorial Goals

• Provide a simple abstraction to get you started
  – Allow easy startup, learning, and experimentation

• Provide code you can use as a starting point
  – See intro-to-dxr.cwyman.org

• Show that using DirectX Raytracing can be straightforward
  – Once you get the low-level initialization and setup working

• Enable quick experiments with hybrid ray-raster rendering techniques
  – Tutorials also provide high-level abstraction over DX raster
Tutorial Goals

- Provide a simple abstraction to get you started
  - Allow easy startup, learning, and experimentation
- Provide code you can use as a staring point
  - See intro-to-dxr.cwyman.org
- Show that using DirectX Raytracing can be straightforward
  - Once you get the low-level initialization and setup working
- Enable quick experiments with hybrid ray-raster rendering techniques
  - Tutorials also provide high-level abstraction over DX raster
- Abstract the GPU ↔ CPU communication
Tutorial Code Structure

• Two pieces:
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
  – (Mostly) standard HLSL shaders
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
  pRes->requestTextureResource( ResourceManager::kOutputChannel );
  mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
  mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
  mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
  mpRays->compileRayProgram();
  if (mpScene) mpRays->setScene(mpScene);
}
```
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Ask for textures managed by render graph resource manager

*Here: We need to output to default output texture*
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Setup a wrapper around our ray tracing GPU launch
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
  pRes->requestTextureResource( ResourceManager::kOutputChannel );

  mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
  mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
  mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
  mpRays->compileRayProgram();
  if (mpScene) mpRays->setScene(mpScene);
}

Our shader file where the HLSL code is located
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

The names of our shader functions in the HLSL code
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
  pRes->requestTextureResource( ResourceManager::kOutputChannel );

  mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
  mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
  mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
  mpRays->compileRayProgram();
  if (mpScene) mpRays->setScene(mpScene);
}
```

Setup our miss shader

*In HLSL, will be miss shader #0, since we declare it here first*
Tutorial Code Structure

• Two pieces:
  - High-level C++, using a render graph style
    • Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Setup our hit group (using default triangle intersector)

In HLSL, will be hit group #0, since we declare it here first
Tutorial Code Structure

Two pieces:
- High-level C++, using a render graph style
  - Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", ",", "MyAnyHit" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Using no closest-hit shader? Skip it
cdot Two pieces:
  – High-level C++, using a render graph style
    • Contains an `initialize()` callback

code

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Using no any-hit shader? Skip it
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an `initialize()` callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Compile shaders & build internal data structures
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an initialize() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}
```

Make sure we know what our scene includes

So the wrapper can build an acceleration structure
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
```
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene( mpScene );
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars[“gOutTex”] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
```

Clear our output texture
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );

    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource( ResourceManager::kOutputChannel );
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene( mpScene );
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
    pRes->requestTextureResource(ResourceManager::kOutputChannel);
    mpRays = RayWrapper::create("simpleRays.rt.hlsl", "MyRayGen");
    mpRays->addMissShader("simpleRays.rt.hlsl", "MyMiss");
    mpRays->addHitShader("simpleRays.rt.hlsl", "MyClosestHit", "");
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTexture(ResourceManager::kOutputChannel);
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an initialize() callback
    • Contains an execute() callback

```cpp
void RayTrace::initialize( ResourceManager::SharedPtr pRes )
{
  pRes->requestTextureResource( ResourceManager::kOutputChannel );

  mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
  mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
  mpRays->addHitShader( "simpleRays.rt.hlsl", "MyClosestHit", "" );
  mpRays->compileRayProgram();
  if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
  Texture::SharedPtr pOutTex = pRes->getTextures( ResourceManager::kOutputChannel );
  pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

  auto vars = mpRays->getRayGenVars();
  vars["gOutTex"] = pOutTex;

  mpRays->execute( pCtx, pRes->getScreenSize() )
}
```
Tutorial Code Structure

---

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an `initialize()` callback
    - Contains an `execute()` callback

```cpp
void RayTrace::initialize(ResourceManager::SharedPtr pRes )
{
    ResourceManager::kOutputChannel);
    mpRays = RayWrapper::create( "simpleRays.rt.hlsl", "MyRayGen" );
    mpRays->addMissShader( "simpleRays.rt.hlsl", "MyMiss" );
    mpRays->addHitShader("simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
    Texture::SharedPtr pOutTex = pRes->getTextures(ResourceManager::kOutputChannel );
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute( pCtx, pRes->getScreenSize() )
}
```

To be clear:

This part is not the focus of this talk!
**Tutorial Code Structure**

---

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an `initialize()` callback
    - Contains an `execute()` callback

```cpp
void RayTrace::initialize(ResourceManager::SharedPtr pRes )
{
  pRes->requestTextureResource( ResourceManager::kOutputChannel );
  mpRays = RayWrapper::create("simpleRays.rt.hlsl", "MyRayGen" );
  mpRays->addMissShader("simpleRays.rt.hlsl", "MyMiss" );
  mpRays->addHitShader("simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit" );
  mpRays->compileRayProgram();
  if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
  Texture::SharedPtr pOutTex = pRes->getTexture( ResourceManager::kOutputChannel );
  pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

  auto vars = mpRays->getRayGenVars();
  vars["gOutTex"] = pOutTex;

  mpRays->execute( pCtx, pRes->getScreenSize() )
}
```

**To be clear:**
This part is not the focus of this talk!

**Tutorial goals:**
Quickly get going with GPU-accelerated ray tracing
Tutorial Code Structure

• Two pieces:
  - High-level C++, using a render graph style
    • Contains an initialize() callback
    • Contains an execute() callback

void RayTrace::initialize(ResourceManager::SharedPtr pRes )
{
  pRes->requestTextureResource( "resourceManager::kOutputChannel" );

  mpRays = mResourceManager->getRayTypes("simpleRays.rt.hlsl", "MyRayGen" );
  mpRays->addHitShader( "simpleRays.rt.hlsl", "MyMiss" );
  mpRays->addHitShader( "simpleRays.rt.hlsl", "MyAnyHit" );
  mpRays->compileRayProgram();
  if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute( RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes )
{
  Texture::SharedPtr pOutTex = pRes->getTexture( "resourceManager::kOutputChannel" );
  pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

  auto vars = mpRays->getRayGenVars();
  vars["gOutTex"] = pOutTex;

  mpRays->execute( pCtx, pRes->getScreenSize() )
}

To be clear:
This part is not the focus of this talk!

Tutorial goals:
Quickly get going with GPU-accelerated ray tracing

This wrapper:
Hides ugly host-side code
Gives an easy-to-use abstraction layer

Not designed to expose advanced functionality
**Tutorial Code Structure**

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an `initialize()` callback
    - Contains an `execute()` callback
  - There’s an `initialize()` callback
  - There’s an `execute()` callback

```cpp
void RayTrace::initialize(ResourceManager::SharedPtr pRes)
{
    pRes->requestTextureResource(ResourceManager::kOutputChannel);

    mpRays = RayWrapper::create("simpleRays.rt.hlsl", "MyRayGen");
    mpRays->addHitShader("simpleRays.rt.hlsl", "MyClosestHit", "MyAnyHit");
    mpRays->addHitShader("simpleRays.rt.hlsl", "MyMiss");
    mpRays->compileRayProgram();
    if (mpScene) mpRays->setScene(mpScene);
}

void RayTrace::execute(RenderContext::SharedPtr pCtx, ResourceManager::SharedPtr pRes)
{
    Texture::SharedPtr pOutTex = pRes->getTexture(ResourceManager::kOutputChannel);
    pRes->clearTexture( pOutTex, vec4(0,0,0,0) );

    auto vars = mpRays->getRayGenVars();
    vars["gOutTex"] = pOutTex;

    mpRays->execute(pCtx, pRes->getScreenSize());
}
```

**To be clear:**
This part is not the focus of this talk!

**Tutorial goals:**
Quickly get going with GPU-accelerated ray tracing

**This wrapper:**
Hides ugly host-side code
Gives an easy-to-use abstraction layer
Not designed to expose advanced functionality

Don’t like it?
Shawn will talk about raw host C++ API later

Want to use my abstractions?
Feel free to chat with me afterwards or online
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an `initialize()` callback
    - Contains an `execute()` callback
  - (Mostly) standard HLSL shaders

My focus for these tutorials
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an initialize() callback
    • Contains an execute() callback
  – (Mostly) standard HLSL shaders

```cpp
__import Raytracing;

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 );
}
```
Tutorial Code Structure

• Two pieces:
  - High-level C++, using a render graph style
    • Contains an initialize() callback
    • Contains an execute() callback
  - (Mostly) standard HLSL shaders

```c
[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 );
}
```
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an `initialize()` callback
    - Contains an `execute()` callback
  - (Mostly) standard HLSL shaders

```cpp
// Output texture specified in C++ code
RWTexture<float4> gOutTex;

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 );
}
Tutorial Code Structure

---

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback
  - (Mostly) standard HLSL shaders

Scene specific globals provided by our tutorial wrapper

```cpp
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 );
}
```
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback
  - (Mostly) standard HLSL shaders

User-defined ray payload

```cpp
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 );
}
```
Tutorial Code Structure

• Two pieces:
  – High-level C++, using a render graph style
    • Contains an `initialize()` callback
    • Contains an `execute()` callback
  – (Mostly) standard HLSL shaders

Utility function computes direction given:
  3 camera vectors
  2D pixel 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 );
}
Tutorial Code Structure

• Two pieces:
  - High-level C++, using a render graph style
    • Contains an initialize() callback
    • Contains an execute() callback
  - (Mostly) standard HLSL shaders

Min & max distance to search along ray
### Tutorial Code Structure

---

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback
  - (Mostly) standard HLSL shaders

---

Which miss shader to use?

*First one specified in C++ code*

```cpp
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, BuiltinIntersectAttribs 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 );
}
```
**Tutorial Code Structure**

- **Two pieces:**
  - High-level C++, using a render graph style
    - *Contains an initialize() callback*
    - *Contains an execute() callback*
  - *(Mostly) standard HLSL shaders*

---

**Which hit group to use?**

*First one specified in C++ code*

*May have 1 any hit, 1 closest hit, and 1 intersection shader*

---

```cpp
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, BuiltinIntersectAttribs 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 );
}
```
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback
  - (Mostly) standard HLSL shaders

---

```cpp
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, BuiltInIntersectAttribs 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 );
}
```

Total number of hit groups
Tutorial Code Structure

- Two pieces:
  - High-level C++, using a render graph style
    - Contains an initialize() callback
    - Contains an execute() callback
  - (Mostly) standard HLSL shaders

Only non-standard shader code here

__import Raytracing;

```c
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, BuiltinIntersectAttribs 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 );
}
```

Allows access to variables for our scene and geometry
(Any framework needs some similar mechanism)
__import Raytracing;

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, BuiltinIntersectAttribs 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);
}

For this scene

This code renders this
__import Raytracing;

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, BuiltInIntersectAttribs attrs) {
    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 );
}
How to Read This Code, In English

This code renders this

For this scene

Given the current pixel ID, compute the direction through the center of the pixel

__import Raytracing;

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, BuiltinIntersectAttribs 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 );
}
__import Raytracing;

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, BuiltinIntersectAttribs attrs) {
    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);
}
How to Read This Code, In English

This code renders this:

If we hit something, return red...

For this scene:

For this scene
This code renders this

If we miss everything, return blue...

For this scene

__import Raytracing;

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, BuiltinIntersectAttribs 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 );
}
How to Read This Code, In English

__import Raytracing;

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, BuiltinIntersectAttribs 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 );
}

For this scene

This code renders this

Store the result out to texture ‘gOutTex’

For this scene

This code renders this

Store the result out to texture ‘gOutTex’
__import Raytracing;

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, BuiltinIntersectAttribs 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 );
}
Tutorial: Ray Traced G-Buffer
• What is a G-Buffer?
  – Common in *deferred rendering*
  – Save visible geometry in first pass
  – Shade at visible geometry in subsequent pass
Tutorial: Ray Traced G-Buffer

- What is a G-Buffer?
  - Common in *deferred rendering*
  - Save visible geometry in first pass
  - Shade at visible geometry in subsequent pass
Tutorial: Ray Traced G-Buffer

• What is a G-Buffer?
  – Common in *deferred rendering*
  – Save visible geometry in first pass
  – Shade at visible geometry in subsequent pass

• What might shader code for this look like?
Tutorial: Ray Traced G-Buffer

- What is a G-Buffer?
  - Common in deferred rendering
  - Save visible geometry in first pass
  - Shade at visible geometry in subsequent pass

- What might shader code for this look like?

```c
RWTexture<float4> gPosOut, gNormOut, gMatlOut;

[shader("closesthit")]
void MyClosestHit(inout Payload, BuiltInIntersectAttributes attrs) {
    uint2 launchIdx = DispatchRaysIndex();
    ShadingData shade = getShadingData( PrimitiveIndex(), attrs );
    gPosOut[launchIdx] = float4( shade.posW, 1.0f );
    gNormOut[launchIdx] = float4( shade.N, 1.0f );
    gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}
```
Tutorial: Ray Traced G-Buffer

- What is a G-Buffer?
  - Common in deferred rendering
  - Save visible geometry in first pass
  - Shade at visible geometry in subsequent pass
- What might shader code for this look like?

```cpp
__import Raytracing;

RWTexture<float4> gPosOut, gNormOut, gMatlOut;

struct Payload { bool dummyValue; };

[shader("closesthit")]
void MyClosestHit(inout Payload, BuiltinIntersectAttrs attrs) {
    uint2 launchIdx = DispatchRaysIndex();
    ShadingData shade = getShadingData( PrimitiveIndex(), attrs );
    gPosOut[launchIdx] = float4( shade.posW, 1.0f );
    gNormOut[launchIdx] = float4( shade.N, 1.0f );
    gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}

[shader("raygeneration")]
void MyRayGen() {
    float3 pixelRayDir = normalize(getRayDirFromPixelID(DispatchRaysIndex()));
    RayDesc ray = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { false };
    TraceRay(gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload);
}
```

Write out geometry properties in closest hit shader

(Behaves a lot like pixel shader in raster)
### Tutorial: Ray Traced G-Buffer

- **What is a G-Buffer?**
  - Common in *deferred rendering*
  - Save visible geometry in first pass
  - Shade at visible geometry in subsequent pass
- **What might shader code for this look like?**

```cpp
RWTexture<float4> gPosOut, gNormOut, gMatlOut;

[shader("closesthit")]
void MyClosestHit(inout Payload, BuiltinIntersectAttribs attribs) {
  uint2 launchIdx = DispatchRaysIndex();
  ShadingData shade = getShadingData(PrimitiveIndex(), attribs);
  gPosOut[launchIdx] = float4( shade.posW, 1.0f );
  gNormOut[launchIdx] = float4( shade.N, 1.0f );
  gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}
```

**Tutorial-provided utility for accessing scene data**
Tutorial: Ray Traced G-Buffer

• What is a G-Buffer?
  – Common in deferred rendering
  – Save visible geometry in first pass
  – Shade at visible geometry in subsequent pass

• What might shader code for this look like?

```cpp
__import Raytracing;

RWTexture<float4> gPosOut, gNormOut, gMatlOut;

[shader("closesthit")]
void MyClosestHit(inout Payload, BuiltinIntersectAttribs attribs) {
  uint2 launchIdx = DispatchRaysIndex();
  ShadingData shade = getShadingData( PrimitiveIndex(), attribs );

  gPosOut[launchIdx] = float4( shade.posW, 1.0f );
  gNormOut[launchIdx] = float4( shade.N, 1.0f );
  gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}

[shader("raygeneration")]
void MyRayGen() {
  float3 pixelRayDir = normalize( getRayDirFromPixelID( DispatchRaysIndex() ) );
  RayDesc ray = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
  RayPayload payload = { false };
  TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );
}
```

To know what material & texture properties to return,
need the triangle ID and hit location
Tutorial: Ray Traced G-Buffer

• What is a G-Buffer?
  – Common in deferred rendering
  – Save visible geometry in first pass
  – Shade at visible geometry in subsequent pass

• What might shader code for this look like?

```cpp
RWTexture<float4> gPosOut, gNormOut, gMatlOut;

[shader(“closesthit”)]
void MyClosestHit(inout Payload, BuiltinIntersectAttribs attrs) {
    uint2 launchIdx = DispatchRaysIndex();
    ShadingData shade = getShadingData( PrimitiveIndex(), attrs );
    gPosOut[launchIdx] = float4( shade.posW, 1.0f );
    gNormOut[launchIdx] = float4( shade.N, 1.0f );
    gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}

[shader(“raygeneration”)]
void MyRayGen() {
    float3 pixelRayDir = normalize( getRayDirFromPixelID( DispatchRaysIndex() ) );
    RayDesc ray = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { false };
    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );
}
```

Ray generation shader similar to before
(except no writing to texture)
Tutorial: Ray Traced G-Buffer

- What is a G-Buffer?
  - Common in deferred rendering
  - Save visible geometry in first pass
  - Shade at visible geometry in subsequent pass
- What might shader code for this look like?

```cpp
#include Raytracing;

RWTexture<float4> gPosOut, gNormOut, gMatlOut;
struct Payload { bool dummyValue; };

[shader("closesthit")]
void MyClosestHit(inout Payload, BuiltInIntersectAttributes attrs) {
  uint2 launchIdx = DispatchRaysIndex();
  ShadingData shade = getShadingData( PrimitiveIndex(), attrs );
  gPosOut[launchIdx] = float4( shade.posW, 1.0f );
  gNormOut[launchIdx] = float4( shade.N, 1.0f );
  gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}

[shader("raygeneration")]
void MyRayGen() {
  float3 pixelRayDir = normalize( getRayDirFromPixelID( DispatchRaysIndex() ) );
  RayDesc ray = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
  RayPayload payload = { false };
  TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );
}
```

Currently: payload must be a struct

Tutorial: Ray Traced G-Buffer

• What is a G-Buffer?
  – Common in deferred rendering
  – Save visible geometry in first pass
  – Shade at visible geometry in subsequent pass

• What might shader code for this look like?

```cpp
RWTexture<float4> gPosOut, gNormOut, gMatlOut;
struct Payload { bool dummyValue; };

// Assume the user clears G-buffer in C++ (So use an empty or null miss shader)

void MyMiss(inout Payload) { }

void MyClosestHit(inout Payload, BuiltinIntersectAttribs attribs) {
    uint2 launchIdx = DispatchRaysIndex();
    ShadingData shade = getShadingData( PrimitiveIndex(), attribs );
    gPosOut[launchIdx] = float4( shade.posW, 1.0f );
    gNormOut[launchIdx] = float4( shade.N, 1.0f );
    gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}

void MyRayGen() {
    float3 pixelRayDir = normalize( getRayDirFromPixelID( DispatchRaysIndex() ) );
    RayDesc ray = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
    RayPayload payload = { false };
    TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );
}
```
Tutorial: Ray Traced G-Buffer

• What is a G-Buffer?
  – Common in deferred rendering
  – Save visible geometry in first pass
  – Shade at visible geometry in subsequent pass

• What might shader code for this look like?

```cpp
#import Raytracing;

RWTexture<float4> gPosOut, gNormOut, gMatlOut;
struct Payload { bool dummyValue; };

[shader("miss")]
void MyMiss(inout Payload) { }

[shader("closesthit")]
void MyClosestHit(inout Payload, BuiltinIntersectAttribs attrs) {
  uint2 launchIdx = DispatchRaysIndex();
  ShadingData shade = getShadingData( PrimitiveIndex(), attrs );
  gPosOut[launchIdx] = float4( shade.posW, 1.0f );
  gNormOut[launchIdx] = float4( shade.N, 1.0f );
  gMatlOut[launchIdx] = float4( shade.diffuse, 1.0f );
}

[shader("raygeneration")]
void MyRayGen() {
  float3 pixelRayDir = normalize( getRayDirFromPixelID( DispatchRaysIndex() ) );
  RayDesc ray = { gCamera.posW, 0.0f, pixelRayDir, 1e+38f };
  RayPayload payload = { false };
  TraceRay( gRtScene, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload );
}
Tutorial: Ray Traced G-Buffer

- What is a G-Buffer?
  - Common in deferred rendering
  - Save visible geometry in first pass
  - Shade at visible geometry in subsequent pass

- What might shader code for this look like?

For this scene

1. Position buffer
2. Normal buffer
3. Diffuse material buffer
Tutorial: Ray Traced Ambient Occlusion
Tutorial: Ray Traced Ambient Occlusion

• What is ambient occlusion?
  – Approximates incident light over hemisphere
  – Gives a (very) soft shadow

Ambient occlusion with one shadow ray per pixel
Tutorial: Ray Traced Ambient Occlusion

• What is ambient occlusion?
  – Approximates incident light over hemisphere
  – Gives a (very) soft shadow

• Simplest implementation:
  – Shoot random ray over hemisphere
  – See if any occluders within specified radius
  – No? Return 1
  – Yes? Return 0

Ambient occlusion with one shadow ray per pixel
Tutorial: Ray Traced Ambient Occlusion

• How is this code different?
Tutorial: Ray Traced Ambient Occlusion

- How is this code different?
- Runs in two passes
  - Generate a G-buffer (as last tutorial)
  - Spawn rays over hemisphere to compute AO
Tutorial: Ray Traced Ambient Occlusion

• How is this code different?
• Runs in two passes
  - Generate a G-buffer (as last tutorial)
  - Spawn rays over hemisphere to compute AO

We just showed how to do this!
Tutorial: Ray Traced Ambient Occlusion

• How is this code different?
• Runs in two passes
  – Generate a G-buffer (as last tutorial)
  – Spawn rays over hemisphere to compute AO

We now focus on this pass
Ambient Occlusion Shaders

- Let's look at our ray generation shader

```plaintext
__import Raytracing;

Texture<float4> gPosition, gNormal;
RWTexture<float4> gOutColor;

cbuffer ConstHostData {
  float gMaxAO;
};

[shader("raygeneration")]
void MyRayGen() {
  uint2 launchIdx = DispatchRaysIndex();
  uint randSeed = initRand(launchIdx);
  float4 wsPos = gPosition[launchIdx];
  float4 wsNorm = gNormal[launchIdx];
  float aoVal = 1.0f;
  if (!isBackgroundPixel(wsPos)) {
    float3 aoDir = getRandomDir(randSeed, wsNorm);
    aoVal = shootAoRay(wsPos.xyz, aoDir, 1.0e-4f, gMaxAO);
  }
  gOutColor[launchIdx] = float4(aoVal, aoVal, aoVal, 1.0f);
}
```
Ambient Occlusion Shaders

• Let's look at our ray generation shader

For current pixel, init random # generator
(Fairly standard; see code for details)
Ambient Occlusion Shaders

• Let's look at our ray generation shader

Load pixel position and normal from G-buffer

```cpp
__import Raytracing;

Texture<float4> gPosition, gNormal;

[shader("raygeneration")]
void MyRayGen() {
    uint2 launchIdx = DispatchRaysIndex();
    uint randSeed = initRand(launchIdx);

    float4 wsPos = gPosition[launchIdx];
    float4 wsNorm = gNormal[launchIdx];
```
Ambient Occlusion Shaders

- Let's look at our ray generation shader

Set default AO value to output in background
Ambient Occlusion Shaders

- Let's look at our ray generation shader

Elsewhere, pick a random direction for AO ray

See provided code for details

Should be a cosine-weighted random sample on the hemisphere
Ambient Occlusion Shaders

Let's look at our ray generation shader

Shoot our AO ray
From our G-buffer location `wsPos` in our random direction with minimum hit distance of $10^{-4}$ and max of $gMaxAO$
**Ambient Occlusion Shaders**

Let's look at our ray generation shader.

```c
__import Raytracing;

Texture<float4> gPosition, gNormal;
RWTexture<float4> gOutColor;

cbuffer ConstHostData {
  float gMaxAO;
};

[shader("raygeneration")]
void MyRayGen() {
  uint2 launchIdx = DispatchRaysIndex();
  uint randSeed = initRand( launchIdx );

  float4 wsPos = gPosition[launchIdx];
  float4 wsNorm = gNormal[launchIdx];
  float aoVal = 1.0f;

  if ( !isBackgroundPixel( wsPos ) ) {
    float3 aoDir = getRandomDir( randSeed, wsNorm );
    aoVal = shootAoRay( wsPos.xyz, aoDir, 1.0e-4f, gMaxAO );
  }

  gOutColor[launchIdx] = float4( aoVal, aoVal, aoVal, 1.0f );
}
```

Constant buffer with data provided by our C++ program.
Ambient Occlusion Shaders

- So how do we shoot our ambient occlusion ray?
  - Note: In same shader file as last slide

```cpp
float shootAoRay(float3 orig, float3 dir, float minT, float maxT) {
  RayDesc ray = {
    orig, minT, dir, maxT
  };
  AOPayload 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.aoVal;
}
```
Ambient Occlusion Shaders

• So how do we shoot our ambient occlusion ray?
  – Note: In same shader file as last slide

Set up our ray and payload

Trace the ray and return the result

```c
struct AOPayload {
    float aoVal; // 0.0 means ‘hit’, 1.0 means ‘miss’
};

float shootAoRay(float3 orig, float3 dir, float minT, float maxT) {
    RayDesc ray = { orig, minT, dir, maxT }; 
    AOPayload pay = { 0.0f }; 
    TraceRay( gRtScene, flags, 0xFF, 0, 1, 0, ray, pay );
    return pay.aoVal;
}
...
Ambient Occlusion Shaders

• So how do we shoot our ambient occlusion ray?
  – Note: In same shader file as last slide

Note: Assume the shadow ray hits unless otherwise told
Ambient Occlusion Shaders

- So how do we shoot our ambient occlusion ray?
  - Note: In same shader file as last slide

```c
struct AOPayload {
  float aoVal; // 0.0 means ‘hit’, 1.0 means ‘miss’
};

[shader(“miss”)] void AoMiss(inout AOPayload pay) {
  pay.aoVal = 1.0f;
}

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

float shootAoRay(float3 orig, float3 dir, float minT, float maxT) {
  RayDesc ray = { orig, minT, dir, maxT };
  AOPayload 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.aoVal;
}
```

Our ray never executes closest hit
(Why? We assume rays hit, so no need)
 Ambient Occlusion Shaders

- So how do we shoot our ambient occlusion ray?
  - Note: In same shader file as last slide

```
struct AOPayload {
  float aoVal; // 0.0 means ‘hit’, 1.0 means ‘miss’
};

void AoMiss(inout AOPayload pay) {
  pay.aoVal = 1.0f;
}

void AoAnyHit(inout AOPayload pay, BuiltinIntersectAttribs attribs) {
  if (alphaTestFails(attribs))
    IgnoreHit();
}

float shootAoRay(float3 orig, float3 dir, float minT, float maxT) {
  RayDesc ray = { orig, minT, dir, maxT };  
  AOPayload 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.aoVal;
}
```

Once we find any hit, stop
(Why? We then know our assumption is true)
Ambient Occlusion Shaders

- So how do we shoot our ambient occlusion ray?
  - Note: In same shader file as last slide

Ray misses? Set AO flag appropriately
So how do we shoot our ambient occlusion ray?

- Note: In same shader file as last slide

On any surface, check if alpha test passes
(If not, geometry is transparent here; ignore the hit)
 Ambient Occlusion Shaders

• So how do we shoot our ambient occlusion ray?
  – Note: In same shader file as last slide

...?

A utility function queries scene textures
(See tutorial code for details)
Ambient Occlusion Results

// Remember our shader’s HLSL variables?
Texture<float4> gPosition, gNormal;
RWTexture<float4> gOutColor;

cbuffer ConstHostData {
    float gMaxAO;
};

... Key differences in this tutorial’s C++ code: sending data to HLSL
auto vars = mpRays->getRayGenVars();
vars[“gPosition”] = pRes->getTexture( “GBufferPositions” );
vars[“gNormal”] = pRes->getTexture( “GBufferNormals” );
vars[“gOutColor”] = pRes->getTexture( ResourceManager::kOutputChannel );
vars[“ConstHostData”][“gMaxAO”] = float( myMaxAmbientOcclusionDistance );
...
Want Less Noise? Shoot More Rays!

Amazon Bistro (64 rays per pixel)

UE4 Sun Temple (64 rays per pixel)
Shooting >1 Ray Per Pixel

... cbuffer ConstHostData {
    float gMaxAO;
};

[shader("raygeneration")]
void AoRayGen() {
    uint2 launchIdx = DispatchRaysIndex();
    uint randSeed = initRand(launchIdx);
    float4 wsPos = gPosition[launchIdx];
    float4 wsNorm = gNormal[launchIdx];
    float aoVal = 1.0f;
    if (!isBackgroundPixel(wsPos)) {
        float3 aoDir = getRandomDir(randSeed, wsNorm);
        aoVal = shootAoRay(wsPos.xyz, aoDir, 1.0e-4f, gMaxAO);
    }
    float aoVal = aoCount / float(gNumRays);
    gOutColor[launchIdx] = float4(aoVal, aoVal, aoVal, 1.0f);
}
Shooting >1 Ray Per Pixel

Amazon Bistro (64 rays per pixel)

UE4 Sun Temple (64 rays per pixel)

cbuffer ConstHostData {
  float gMaxAO;
  uint gNumRays;
};

[shader("raygeneration")]
void AoRayGen() {
  uint2 launchIdx = DispatchRaysIndex();
  uint randSeed = initRand( launchIdx );
  float4 wsPos = gPosition[launchIdx];
  float4 wsNorm = gNormal[launchIdx];
  float aoCount = float( gNumRays );

  if ( !isBackgroundPixel( wsPos ) ) {
    aoCount = 0.0f;
    for (int i = 0; i < gNumRays; i++) {
      float3 aoDir = getRandomDir( randSeed, wsNorm );
      aoCount += shootAoRay( wsPos.xyz, aoDir, 1.0e-4f, gMaxAO );
    }
  }

  float aoVal = aoCount / float( gNumRays );
  gOutColor[launchIdx] = float4( aoVal, aoVal, aoVal, 1.0f );
}
Tutorial: Diffuse Shadows And Global Illumination
Diffuse Lighting Only
Diffuse Plus Shadows
Tutorial: Diffuse Shadows And Global Illumination

• How is this code different?
Tutorial: Diffuse Shadows And Global Illumination

• How is this code different?
• Shoots two types of rays:
  – Shadow rays (test visibility only)
  – Indirect rays (return a color in selected direction)
Tutorial: Diffuse, Shadows, & GI

- How is this code different?
- Shoots two **types** of rays:
  - Shadow rays (test visibility only)
  - Indirect rays (return a color in selected direction)
- Shadow rays **identical** to AO rays
  - Both test visibility in a specified direction

```c
struct AOPayload {
    float aoVal; // 0.0 means ‘hit’, 1.0 means ‘miss’
};

[shader(“miss”)]
void AoMiss(inout AOPayload pay) {
    pay.aoVal = 1.0f;
}

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

float shootAoRay(float3 orig, float3 dir, float minT, float maxT) {
    RayDesc ray = { orig, minT, dir, maxT };
    AOPayload 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.aoVal;
}
```

...
Tutorial: Diffuse, Shadows, & GI

• How is this code different?

• Shoots two types of rays:
  – Shadow rays (test visibility only)
  – Indirect rays (return a color in selected direction)

• Shadow rays identical to AO rays
  – Both test visibility in a specified direction
  – Unless you want to rename for clarity

  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;
  }

  ...
Tutorial: Diffuse, Shadows, & GI

• How is this code different?

• Shoots two types of rays:
  – Shadow rays (test visibility only)
  – Indirect rays (return a color in selected direction)

• Shadow rays identical to AO rays
  – Both test visibility in a specified direction
  – Unless you want to rename for clarity
  – And changes due to different number of ray shaders
    • For me, shadows are hit group #0, miss shader #0
    • For me, there are 2 hit groups (1 for shadows, 1 for color)

... 

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, shadHitGrp, totalHitGrps, shadMiss, ray, pay);
    return pay.visibility;
}

...
Tutorial: Diffuse, Shadows, & GI

• How is this code different?

• Shoots two types of rays:
  – Shadow rays (test visibility only)
  – Indirect rays (return a color in selected direction)

• Shadow rays identical to AO rays
  – Both test visibility in a specified direction
  – Unless you want to rename for clarity
  – And changes due to different number of ray shaders
    • For me, shadows are hit group #0, miss shader #0
    • For me, there are 2 hit groups (1 for shadows, 1 for color)

```cpp
... 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, 2, 0, ray, pay);
  return pay.visibility;
}
...
Tutorial: Diffuse, Shadows, & GI

• Color rays are a bit more complex. How?
Tutorial: Diffuse, Shadows, & GI

• Color rays are a **bit** more complex. How?
  - Payload contains a color, per-pixel random seed

```c
struct IndirectPayload {
  float3 color; // will store ray color
  uint rndSeed; // current random seed
};
```
Tutorial: Diffuse, Shadows, & GI

• Color rays are a **bit** more complex. How?
  - Payload contains a color, per-pixel random seed
  - Miss shader needs to return the background color

```cpp
struct IndirectPayload {
    float3 color;  // will store ray color
    uint     rndSeed; // current random seed
};

[shader("miss")]
void IndirectMiss(inout IndirectPayload pay) {
    pay.color = GetBackgroundColor(WorldRayDirection());
}
```
Tutorial: Diffuse, Shadows, & GI

• Color rays are a **bit** more complex. How?
  – Payload contains a color, per-pixel random seed
  – Miss shader needs to return the background color
  – Any hit identical (i.e., discard transparent surfaces)

```cpp
struct IndirectPayload {
  float3 color;  // will store ray color
  uint rndSeed;  // current random seed
};

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

[shader(“anyhit”)]
void IndirectAnyHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
  if (alphaTestFails(attribs))
    IgnoreHit();
}
```
Tutorial: Diffuse, Shadows, & GI

• Color rays are a **bit** more complex. How?
  - Payload contains a color, per-pixel random seed
  - Miss shader needs to return the background color
  - Any hit identical (i.e., discard transparent surfaces)
  - **We have** a closest hit shader
    • At the hitpoint, gets material information then shades

```c
struct IndirectPayload {
    float3 color;  // will store ray color
    uint   rndSeed;  // current random seed
};

[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, BuiltinIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed );
}
```
Tutorial: Diffuse, Shadows, & GI

• Color rays are a bit more complex. How?
  – Payload contains a color, per-pixel random seed
  – Miss shader needs to return the background color
  – Any hit identical (i.e., discard transparent surfaces)
  – We have a closest hit shader
    • At the hitpoint, gets material information then shades

These routines are all specific to your material & shading system
**Tutorial: Diffuse, Shadows, & GI**

• Color rays are a **bit** more complex. How?
  – Payload contains a color, per-pixel random seed
  – Miss shader needs to return the background color
  – Any hit identical (i.e., discard transparent surfaces)
  – We **have** a closest hit shader
    • At the hitpoint, gets material information then shades
  – Shooting a ray is simpler

```cpp
struct IndirectPayload {
    float3 color; // will store ray color
    uint rndSeed; // current random seed
};

[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, BuiltinIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData(attribs);
    pay.color = DiffuseShade(hit.pos, hit.norm, hit.difColor, pay.rndSeed);
}

float3 shootColorRay(float3 orig, float3 dir, float minT, uint rndSeed) {
    RayDesc ray = { orig, minT, dir, 1.0e+38 };
    IndirectPayload pay = { float3(0.0f, 0.0f, 0.0f), rndSeed };
    TraceRay(gRtScene, RAY_FLAG_NONE, 0xFF, 1, 2, 1, ray, pay);
    return pay.color;
}
```
• Color rays are a bit more complex. How?
  – Payload contains a color, per-pixel random seed
  – Miss shader needs to return the background color
  – Any hit identical (i.e., discard transparent surfaces)
  – We have a closest hit shader
    • At the hitpoint, gets material information then shades
  – Shooting a ray is simpler
    • Use no special ray flags
Tutorial: Diffuse, Shadows, & GI

• Color rays are a bit more complex. How?
  - Payload contains a color, per-pixel random seed
  - Miss shader needs to return the background color
  - Any hit identical (i.e., discard transparent surfaces)
  - We have a closest hit shader
    • At the hitpoint, gets material information then shades
  - Shooting a ray is simpler
    • Use no special ray flags
    • Use correct hit & miss shaders for color rays
      - For me, color rays are type #1 (of 2)
Tutorial: Diffuse, Shadows, & GI

• Color rays are a **bit** more complex. How?
  - Payload contains a color, per-pixel random seed
  - Miss shader needs to return the background color
  - Any hit identical (i.e., discard transparent surfaces)
  - We **have** a closest hit shader
    - At the hitpoint, gets material information then shades
  - Shooting a ray is simpler
    - Use no special ray flags
    - Use correct hit & miss shaders for color rays
      - For me, color rays are type #1 (of 2)

```c
struct IndirectPayload {
    float3 color; // will store ray color
    uint rndSeed; // current random seed
};

[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, BuiltinIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData(attribs);
    pay.color = DiffuseShade(hit.pos, hit.norm, hit.difColor, pay.rndSeed);
}

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

Let’s take a look at this function
Tutorial: Diffuse, Shadows, & GI

- What happens for our diffuse shading?
Tutorial: Diffuse, Shadows, & GI

• What happens for our diffuse shading?
  – Pick a random light, so we don’t shoot one ray to each

```cpp
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
  // We will only shoot one shadow ray per frame, randomly to a light
  uint randomLight = uint( nextRand(seed) * gLightsCount );

  // What is the probability we picked that light?
  float sampleProb = 1.0f / float( gLightsCount );

  float3 lightIntensity = gLights[randomLight].intensity;
  float3 lightPos = gLights[randomLight].position;
  float distToLight = length(lightPos - pos);
  float3 dirToLight = normalize(lightPos - pos);

  // Compute our NdotL term; shoot our shadow ray in selected direction
  float NdotL = saturate( dot( norm, dirToLight ) );

  float isLit = shootShadowRay(pos, dirToLight, 1.0e-4f, distToLight);
  float3 rayColor = isLit * lightIntensity;

  // Return shaded color
  return (NdotL * rayColor * (difColor / M_PI)) / sampleProb;
}
```
Tutorial: Diffuse, Shadows, & GI

What happens for our diffuse shading?
- Pick a random light, so we don’t shoot one ray to each
- Load information about the selected light

```c
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
    // We will only shoot one shadow ray per frame, randomly to a light
    uint randomLight = uint( nextRand(seed) * gLightsCount );

    // What is the probability we picked that light?
    float sampleProb = 1.0f / float( gLightsCount );

    // Get information about this light; access your framework's scene structs
    float3 lightIntensity = gLights[randomLight].intensity;
    float3 lightPos = gLights[randomLight].position;
    float distToLight = length( lightPos - pos );
    float3 dirToLight = normalize( lightPos - pos );

    // Compute our NdotL term; shoot our shadow ray in selected direction
    float NdotL = saturate( dot( norm, dirToLight ) );

    // In range [0..1]
    float isLit = shootShadowRay( pos, dirToLight, 1.0e-4f, distToLight );

    float3 rayColor = isLit * lightIntensity;

    // Return shaded color
    return ( NdotL * rayColor * ( difColor / M_PI ) ) / sampleProb;
}
```
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
    // We will only shoot one shadow ray per frame, randomly to a light
    uint randomLight = uint( nextRand(seed) * gLightsCount );

    // What is the probability we picked that light?
    float sampleProb = 1.0f / float( gLightsCount );

    // Get information about this light; access your framework's scene structs
    float3 lightIntensity = gLights[randomLight].intensity;
    float3 lightPos = gLights[randomLight].position;
    float distToLight = length( lightPos - pos );
    float3 dirToLight = normalize( lightPos - pos );

    // Compute our NdotL term; shoot our shadow ray in selected direction
    float NdotL = saturate( dot( norm, dirToLight ) ); // In range [0..1]
    float isLit = shootShadowRay( pos, dirToLight, 1.0e-4f, distToLight );
    float3 rayColor = isLit * lightIntensity;

    // Return shaded color
    return ( NdotL * rayColor * (difColor / M_PI) ) / sampleProb;
}
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
    // We will only shoot one shadow ray per frame, randomly to a light
    uint randomLight = uint( nextRand(seed) * gLightsCount );

    // What is the probability we picked that light?
    float sampleProb = 1.0f / float( gLightsCount );

    // Get information about this light; access your framework's scene structs
    float3 lightIntensity = gLights[randomLight].intensity;
    float3 lightPos = gLights[randomLight].position;
    float distToLight = length( lightPos - pos );
    float3 dirToLight = normalize( lightPos - pos );

    // Compute our NdotL term; shoot our shadow ray in selected direction
    float NdotL = saturate( dot( norm, dirToLight ) ); // In range [0..1]
    float isLit = shootShadowRay( pos, dirToLight, 1.0e-4f, distToLight );
    float3 rayColor = isLit * lightIntensity;

    // Return shaded color
    return (NdotL * rayColor * (difColor / M_PI) ) / sampleProb;
}

• What happens for our diffuse shading?
  – Pick a random light, so we don’t shoot one ray to each
  – Load information about the selected light
  – Compute our cosine (NdotL) term
  – Shoot our shadow ray
  – Surface color depends on the light’s intensity
  – Compute total diffuse color
Tutorial: Diffuse, Shadows, & GI

• What happens for our diffuse shading?
  - Pick a random light, so we don’t shoot one ray to each
  - Load information about the selected light
  - Compute our cosine (NdotL) term
  - Shoot our shadow ray
  - Surface color depends on the light’s intensity
  - Compute total diffuse color

As for most shading, go back to the rendering equation for clarity

```c
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
  // We will only shoot one shadow ray per frame, randomly to a light
  uint randomLight = uint( nextRand(seed) * gLightsCount );

  // What is the probability we picked that light?
  float sampleProb = 1.0f / float( gLightsCount );

  // Get information about this light; access your framework's scene structs
  float3 lightIntensity = gLights[randomLight].intensity;
  float3 lightPos = gLights[randomLight].position;
  float distToLight = length( lightPos - pos );
  float3 dirToLight = normalize( lightPos - pos );

  // Compute our NdotL term; shoot our shadow ray in selected direction
  float NdotL = saturate( dot( norm, dirToLight ) ); // In range [0..1]
  float isLit = shootShadowRay( pos, dirToLight, 1.0e-4f, distToLight );
  float3 rayColor = isLit * lightIntensity;

  // Return shaded color
  return (NdotL * rayColor * (difColor / M_PI) ) / sampleProb;
}
```
Tutorial: Diffuse, Shadows, & GI

• What happens for our diffuse shading?
  – Pick a random light, so we don’t shoot one ray to each
  – Load information about the selected light
  – Compute our cosine (N\cdot L) term
  – Shoot our shadow ray
  – Surface color depends on the light’s intensity
  – Compute total diffuse color

As for most shading, go back to the rendering equation for clarity

\[ \int_{\Omega} f_r(\omega_i, \omega_o) L_i(\omega)(\omega_i \cdot \hat{n}) d\omega_i \]
Tutorial: Diffuse, Shadows, & GI

What happens for our diffuse shading?

- Pick a random light, so we don’t shoot one ray to each
- Load information about the selected light
- Compute our cosine ($N\cdot L$) term
- Shoot our shadow ray
- Surface color depends on the light’s intensity
- Compute total diffuse color

But we’re actually sampling it randomly
(aka Monte Carlo Integration)

$$\int_{\Omega} f_r(\omega_i, \omega_o)L_i(\omega) (\omega_i \cdot \hat{n}) d\omega_i \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f_r(\omega_i, \omega_o)L_i(\omega_i) (\omega_i \cdot \hat{n})}{p(\omega_j)}$$

```c
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
    // We will only shoot one shadow ray per frame, randomly to a light
    uint randomLight = uint( nextRand(seed) * gLightsCount );

    // What is the probability we picked that light?
    float sampleProb = 1.0f / float( gLightsCount );

    // Get information about this light; access your framework's scene structs
    float3 lightIntensity = gLights[randomLight].intensity;
    float3 lightPos = gLights[randomLight].position;
    float distToLight = length( lightPos - pos );
    float3 dirToLight = normalize( lightPos - pos );

    // Compute our NdotL term; shoot our shadow ray in selected direction
    float NdotL = saturate( dot( norm, dirToLight ) ); // In range [0..1]
    float isLit = shootShadowRay( pos, dirToLight, 1.0e-4f, distToLight );
    float3 rayColor = isLit * lightIntensity;

    // Return shaded color
    return (NdotL * rayColor * (difColor / M_PI) ) / sampleProb;
}
```
Tutorial: Diffuse, Shadows, & GI

• What happens for our diffuse shading?
  – Pick a random light, so we don’t shoot one ray to each
  – Load information about the selected light
  – Compute our cosine ($N \cdot L$) term
  – Shoot our shadow ray
  – Surface color depends on the light’s intensity
  – Compute total diffuse color

With only $N=1$ shadow ray

$$\frac{1}{N} \sum_{j=1}^{N} f_r(\omega_i, \omega_o) L_i(\omega_i) (\omega_i \cdot \hat{n})$$

```cpp
void DiffuseShade( float3 pos, float3 norm, float3 difColor, inout uint seed ) {
    // We will only shoot one shadow ray per frame, randomly to a light
    uint randomLight = uint( nextRand(seed) * gLightsCount );

    // What is the probability we picked that light?
    float sampleProb = 1.0f / float( gLightsCount );

    // Get information about this light; access your framework's scene structs
    float3 lightIntensity = gLights[randomLight].intensity;
    float3 lightPos = gLights[randomLight].position;
    float distToLight = length( lightPos - pos );
    float3 dirToLight = normalize( lightPos - pos );

    // Compute our $N \cdot L$ term; shoot our shadow ray in selected direction
    float NdotL = saturate( dot( norm, dirToLight ) ); // In range [0..1]
    float isLit = shootShadowRay( pos, dirToLight, 1.0e-4f, distToLight );
    float3 rayColor = isLit * lightIntensity;

    // Return shaded color
    return (NdotL * rayColor * (difColor / M_PI) ) / sampleProb;
}
```
Tutorial: Diffuse, Shadows, & GI

• Our ray generation shader, pretty straightforward:
Tutorial: Diffuse, Shadows, & GI

- Our ray generation shader, pretty straightforward:
  - Initialize random seed for current pixel

```cpp
[shader("raygeneration")]
void DiffuseGIRayGen() {
  uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
  uint randSeed = initRand(launchIdx);   // Get RNG for this pixel

  float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
  float3 wsNorm = gNormal[launchIdx].xyz;  // Load normal from G-buffer
  float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

  float3 color = DiffuseShade(wsPos, wsNorm, difColor, randSeed);

  float3 giDir = getCosWeightedRandomDir(randSeed, wsNorm);
  float3 giColor = shootColorRay(wsPos, giDir, 1.0e-4f, randSeed);

  color += difColor * giColor;

  gOutColor[launchIdx] = float4(color, 1.0f);
}
```
Tutorial: Diffuse, Shadows, & GI

• Our ray generation shader, pretty straightforward:
  – Initialize random seed for current pixel
  – Load pixel data from our G-buffer

```cpp
[shader("raygeneration")] void DiffuseGIRayGen() {
    uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
    uint randSeed = initRand(launchIdx); // Get RNG for this pixel

    float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
    float3 wsNorm = gNormal[launchIdx].xyz; // Load normal from G-buffer
    float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

    float3 color = DiffuseShade(wsPos, wsNorm, difColor, randSeed);

    float3 giDir = getCosWeightedRandomDir(randSeed, wsNorm);
    float3 giColor = shootColorRay(wsPos, giDir, 1.0e-4f, randSeed);

    color += difColor * giColor;

    gOutColor[launchIdx] = float4(color, 1.0f);
}
```

Tutorial: Diffuse, Shadows, & GI

- Our ray generation shader, pretty straightforward:
  - Initialize random seed for current pixel
  - Load pixel data from our G-buffer
  - Do our diffuse shade & shadows at this point

```cpp
[shader("raygeneration")]
void DiffuseGIRayGen() {
  uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
  uint randSeed = initRand( launchIdx ); // Get RNG for this pixel

  float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
  float3 wsNorm = gNormal[launchIdx].xyz; // Load normal from G-buffer
  float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

  // Do diffuse shading at the primary hit
  float3 color = DiffuseShade( wsPos, wsNorm, difColor, randSeed );
```

• Our ray generation shader, pretty straightforward:
  - Initialize random seed for current pixel
  - Load pixel data from our G-buffer
  - Do our diffuse shade & shadows at this point
Tutorial: Diffuse, Shadows, & GI

- Our ray generation shader, pretty straightforward:
  - Initialize random seed for current pixel
  - Load pixel data from our G-buffer
  - Do our diffuse shade & shadows at this point
  - Shoot our global illum query; accumulate the result

```cpp
[shader("raygeneration")]
void DiffuseGIRayGen() {
  uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
  uint randSeed = initRand( launchIdx ); // Get RNG for this pixel

  float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
  float3 wsNorm = gNormal[launchIdx].xyz; // Load normal from G-buffer
  float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

  // Do diffuse shading at the primary hit
  float3 color = DiffuseShade( wsPos, wsNorm, difColor, randSeed );

  // Pick random direction for global illumination indirect ray; shoot ray
  float3 giDir = getCosWeightedRandomDir( randSeed, wsNorm );
  float3 giColor = shootColorRay( wsPos, giDir, 1.0e-4f, randSeed );

  // Accumulate properly weighted result into final color
  // Due to cosine-weighting, terms cancel, leaving simpler equation
  color += difColor * giColor;
}
```
Our ray generation shader, pretty straightforward:
- Initialize random seed for current pixel
- Load pixel data from our G-buffer
- Do our diffuse shade & shadows at this point
- Shoot our global illum query; accumulate the result
- Save to the output buffer
Tutorial: Diffuse, Shadows, & GI

Our ray generation shader, pretty straightforward:

- Initialize random seed for current pixel
- Load pixel data from our G-buffer
- Do our diffuse shade & shadows at this point
- Shoot our global illum query; accumulate the result
- Save to the output buffer

Deceptively simple... But it's not:

```cpp
RWTexture<float4> gOutColor;

[shader("raygeneration")]
void DiffuseGIRayGen() {
    uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
    uint randSeed = initRand(launchIdx); // Get RNG for this pixel
    float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
    float3 wsNorm = gNormal[launchIdx].xyz; // Load normal from G-buffer
    float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

    // Do diffuse shading at the primary hit
    float3 color = DiffuseShade(wsPos, wsNorm, difColor, randSeed);

    // Pick random direction for global illumination indirect ray; shoot ray
    float3 giDir = getCosWeightedRandomDir(randSeed, wsNorm);
    float3 giColor = shootColorRay(wsPos, giDir, 1.0e-4f, randSeed);

    // Accumulate properly weighted result into final color
    // Due to cosine-weighting, terms cancel, leaving simpler equation
    color += difColor * giColor;

    gOutColor[launchIdx] = float4(color, 1.0f);
}
```
Tutorial: Diffuse, Shadows, & GI

• Our ray generation shader, pretty straightforward:
  – Initialize random seed for current pixel
  – Load pixel data from our G-buffer
  – Do our diffuse shade & shadows at this point
  – Shoot our global illum query; accumulate the result
  – Save to the output buffer

Deceptively simple… But it’s not:

\[
\frac{1}{N} \sum_{j=1}^{N} f_r(\omega_j, \omega_o) L_i(\omega_j)(\omega_j \cdot \hat{n}) p(\omega_j)
\]

```
RWTexture<float4> gOutColor;

[shader("raygeneration")]
void DiffuseGIRayGen() {
    uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
    uint randSeed = initRand(launchIdx); // Get RNG for this pixel

    float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
    float3 wsNorm = gNormal[launchIdx].xyz; // Load normal from G-buffer
    float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

    // Do diffuse shading at the primary hit
    float3 color = DiffuseShade(wsPos, wsNorm, difColor, randSeed);

    // Pick random direction for global illumination indirect ray; shoot ray
    float3 giDir = getCosWeightedRandomDir(randSeed, wsNorm);
    float3 giColor = shootColorRay(wsPos, giDir, 1.0e-4f, randSeed);

    // Accumulate properly weighted result into final color
    // Due to cosine-weighting, terms cancel, leaving simpler equation
    color += difColor * giColor;

    gOutColor[launchIdx] = float4(color, 1.0f);
}
```
Tutorial: Diffuse, Shadows, & GI

• Our ray generation shader, pretty straightforward:
  – Initialize random seed for current pixel
  – Load pixel data from our G-buffer
  – Do our diffuse shade & shadows at this point
  – Shoot our global illum query; accumulate the result
  – Save to the output buffer

\[
\frac{1}{N} \sum_{j=1}^{N} f_r(\omega_j, \omega_o) L_i(\omega_j) (\omega_j \cdot \hat{n}) p(\omega_j)
\]

Key... For cos-weighted samples:

\[
p(\omega_j) = \frac{(\omega_j \cdot \hat{n})}{\pi}
\]

void DiffuseGIRayGen() {
  uint2 launchIdx = DispatchRaysIndex();  // What pixel are we?
  uint randSeed = initRand(launchIdx);    // Get RNG for this pixel
  float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
  float3 wsNorm = gNormal[launchIdx].xyz;  // Load normal from G-buffer
  float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

  // Do diffuse shading at the primary hit
  float3 color = DiffuseShade(wsPos, wsNorm, difColor, randSeed);

  // Pick random direction for global illumination indirect ray; shoot ray
  float3 giDir = getCosWeightedRandomDir(randSeed, wsNorm);
  float3 giColor = shootColorRay(wsPos, giDir, 1.0e-4f, randSeed);

  // Accumulate properly weighted result into final color
  // Due to cosine-weighting, terms cancel, leaving simpler equation
  color += difColor * giColor;

  gOutColor[launchIdx] = float4(color, 1.0f);
}

RWTexture<float4> gOutColor;

[shader("raygeneration")]

void DiffuseGIRayGen() {
  uint2 launchIdx = DispatchRaysIndex();  // What pixel are we?
  uint randSeed = initRand(launchIdx);    // Get RNG for this pixel
  float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
  float3 wsNorm = gNormal[launchIdx].xyz;  // Load normal from G-buffer
  float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

  // Do diffuse shading at the primary hit
  float3 color = DiffuseShade(wsPos, wsNorm, difColor, randSeed);

  // Pick random direction for global illumination indirect ray; shoot ray
  float3 giDir = getCosWeightedRandomDir(randSeed, wsNorm);
  float3 giColor = shootColorRay(wsPos, giDir, 1.0e-4f, randSeed);

  // Accumulate properly weighted result into final color
  // Due to cosine-weighting, terms cancel, leaving simpler equation
  color += difColor * giColor;

  gOutColor[launchIdx] = float4(color, 1.0f);
}
Tutorial: Diffuse, Shadows, & GI

- Our ray generation shader, pretty straightforward:
  - Initialize random seed for current pixel
  - Load pixel data from our G-buffer
  - Do our diffuse shade & shadows at this point
  - Shoot our global illum query; accumulate the result
  - Save to the output buffer

\[
\frac{1}{N} \sum_{j=1}^{N} f_r(\omega_j, \omega_o) L_i(\omega_j)(\omega_j \cdot n) \cdot p(\omega_j)
\]

Key… For cos-weighted samples:

\[
p(\omega_j) = \frac{(\omega_j \cdot n)}{\pi}
\]

```c
RWTexture<float4> gOutColor;

[shader("raygeneration")]
void DiffuseGIRayGen() {
    uint2 launchIdx = DispatchRaysIndex(); // What pixel are we?
    uint randSeed = initRand( launchIdx ); // Get RNG for this pixel

    float3 wsPos = gPosition[launchIdx].xyz; // Load position from G-buffer
    float3 wsNorm = gNormal[launchIdx].xyz; // Load normal from G-buffer
    float3 difColor = gDifColor[launchIdx].xyz; // Load material from G-buffer

    // Do diffuse shading at the primary hit
    float3 color = DiffuseShade( wsPos, wsNorm, difColor, randSeed );

    // Pick random direction for global illumination indirect ray; shoot ray
    float3 giDir = getCosWeightedRandomDir( randSeed, wsNorm );
    float3 giColor = shootColorRay( wsPos, giDir, 1.0e-4f, randSeed );

    // Accumulate properly weighted result into final color
    // Due to cosine-weighting, terms cancel, leaving simpler equation
    color += difColor * giColor;

    gOutColor[launchIdx] = float4( color, 1.0f );
}
```
Diffuse Global Illumination Results
Tutorial: Adding Multiple Bounces
Direct Lighting Only
Direct + 2 Bounces Diffuse GI
Tutorial: Adding Multiple Bounces

• Adding extra bounces can have big impact!
Tutorial: Adding Multiple Bounces

• Adding extra bounces can have big impact!
• No huge code change with recursive ray tracing
Tutorial: Adding Multiple Bounces

Remember our closest hit shader?

```c
struct IndirectPayload {
  float3 color; // will store ray color
  uint rndSeed; // current random seed
};

[shader("closesthit")]
void IndirectClosestHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
  ShadingData hit = getHitShadingData( attribs );

  pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed );
}
```
Tutorial: Adding Multiple Bounces

Remember our closest hit shader?

**Two steps:**

- **Load data to shade**
- **Do direct lighting**

```c
[shader("closesthit")]
void IndirectClosestHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs);
    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed );
}
```
Tutorial: Adding Multiple Bounces

Let’s add a third step...

Two steps:

• Load data to shade
• Do direct lighting
• Add additional bounces

[shader("closesthit")]

void IndirectClosestHit(inout IndirectPayload pay, BuiltinIntersectAttribs attrs) {
    ShadingData hit = getHitShadingData( attrs);

    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed);

    if (pay.rayDepth < gMaxRayDepth)
    {
        float3 giDir = getCosWeightedRandomDir( randSeed, wsNorm );
        float3 giColor = shootColorRay( wsPos, giDir, 1.0e-4f, randSeed );
        pay.color += hit.difColor * giColor;
    }
}
Tutorial: Adding Multiple Bounces

Requires a new shader parameter
(Note this a hit shader variable, which is specified separately from a ray gen variable)

Requires remembering current ray depth

cbuffer HitShaderCB {
    uint gMaxRayDepth; // Specify your max ray depth
};

struct IndirectPayload {
    float3 color;  // will store ray color
    uint rndSeed; // current random seed
    uint rayDepth; // increment by 1 every time you bounce
};

[shader("closesthit")]
void IndirectClosestHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );

    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed );

    if (pay.rayDepth < gMaxRayDepth)
    {
        float3 giDir = getCosWeightedRandomDir( randSeed, wsNorm );
        float3 giColor = shootColorRay( wsPos, giDir, 1.0e-4f, randSeed );
        pay.color += hit.difColor * giColor;
    }
}
Want a Different Material Model?
Want a Different Material Model?

cbuffer HitShaderCB {
    uint gMaxRayDepth; // Specify your max ray depth
};

struct IndirectPayload {
    float3 color;  // will store ray color
    uint rndSeed;  // current random seed
    uint rayDepth; // increment by 1 every time you bounce
};

[shader(“closesthit”)]
void IndirectClosestHit(inout IndirectPayload pay, BuiltInIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed );
    if (pay.rayDepth < gMaxRayDepth)
        { float3 giDir = getCosWeightedRandomDir( randSeed, wsNorm );
          float3 giColor = shootColorRay( wsPos, giDir, 1.0e-4f, randSeed );
          pay.color += hit.difColor * giColor;
        }
}
Want a Different Material Model?

Write a new function to do direct shading

(See tutorial code for GGX model)

cbuffer HitShaderCB {
    uint gMaxRayDepth; // Specify your max ray depth
};

struct IndirectPayload {
    float3 color; // will store ray color
    uint rndSeed; // current random seed
    uint rayDepth; // increment by 1 every time you bounce
};

[shader("closesthit")] void IndirectClosestHit(inout IndirectPayload pay, BuiltinIntersectAttribs attribs) {
    ShadingData hit = getHitShadingData( attribs );
    pay.color = DiffuseShade( hit.pos, hit.norm, hit.difColor, pay.rndSeed );

    if (pay.rayDepth < gMaxRayDepth) {
        float3 giDir = getCosWeightedRandomDir( randSeed, wsNorm );
        float3 giColor = shootColorRay( wsPos, giDir, 1.0e-4f, randSeed );
        pay.color += hit.difColor * giColor;
    }
}
Want a Different Material Model?

Write new sampling code to do indirect light
(See tutorial code for GGX model)
One Last Example: Sphere Rendering

- All examples have used raster scenes: i.e., only triangles
One Last Example: Sphere Rendering

- All examples have used raster scenes: i.e., only triangles
- But one advantage of ray tracing: arbitrary shapes!

*Final example from Pete’s “Ray Tracing in One Weekend”*
One Last Example: Sphere Rendering

- Need: define how rays intersect a sphere
  - Plenty of documentation on required mathematics

```cpp
[shader(“intersection”)]
void SphereIntersect() {
    // Code for sphere intersection
}
```
One Last Example: Sphere Rendering

- Need: define how rays intersect a sphere
  - Plenty of documentation on required mathematics

- Custom intersection shaders:
  - You are responsible for sending data
  - I pass an array of sphere radii and centers

```c
Buffer<float4> gSphereData;

[shader(“intersection”)]
void SphereIntersect() {
    float4 sphData = gSphereData[ PrimitiveIndex() ];
    float3 sphCenter = sphData.xyz;
    float3 sphRadius = sphData.w;
}
```
One Last Example: Sphere Rendering

• Need: define how rays intersect a sphere
  – Plenty of [documentation](#) on required mathematics

• Custom intersection shaders:
  – You are responsible for sending data
  – I pass an array of sphere radii and centers

• Plug in center, radius, ray origin and direction
  – To standard intersection math

```cpp
Buffer<float4> gSphereData;

[shader(“intersection”)]
void SphereIntersect() {
  float4 sphData = gSphereData[ PrimitiveIndex() ];
  float3 sphCenter = sphData.xyz;
  float sphRadius = sphData.w;
  float3 toCenter = WorldRayOrigin() - sphCenter;
  float a = dot( WorldRayDirection(), WorldRayDirection() );
  float b = 2.0f * dot( WorldRayDirection(), toCenter );
  float c = dot( toCenter, toCenter ) - sphRadius * sphRadius;
  if (b * b >= 4.0f * a * c) {
    float sqrtVal = sqrt( b * b - 4.0f * a * c );
    SphereAttribs sphAttr = { sphCenter };
    ReportHit( ( -b - sqrtVal ) / (2.0f * a), 0, sphAttr );
    ReportHit( ( -b + sqrtVal ) / (2.0f * a), 0, sphAttr );
  }
}
```
One Last Example: Sphere Rendering

- Need: define how rays intersect a sphere
  - Plenty of documentation on required mathematics

- Custom intersection shaders:
  - You are responsible for sending data
  - I pass an array of sphere radii and centers

- Plug in center, radius, ray origin and direction
  - To standard intersection math

- ReportHit() at all possible hit points
  - Attribute is whatever data you need to shade

```cpp
struct SphereAttribs {
    float3 sphCenter;
};
Buffer<float4> gSphereData;

[shader("intersection")]
void SphereIntersect() {
    float4 sphData = gSphereData[ PrimitiveIndex() ];
    float3 sphCenter = sphData.xyz;
    float sphRadius = sphData.w;
    float3 toCenter = WorldRayOrigin() - sphCenter;

    float a = dot( WorldRayDirection(), WorldRayDirection() );
    float b = 2.0f * dot( WorldRayDirection(), toCenter );
    float c = dot( toCenter, toCenter ) - sphRadius * sphRadius;

    if (b * b >= 4.0f * a * c) {
        float sqrtVal = sqrt( b * b - 4.0f * a * c );
        SphereAttribs sphAttr = { sphCenter };
        ReportHit( ( -b - sqrtVal ) / (2.0f * a), 0, sphAttr );
        ReportHit( ( -b + sqrtVal ) / (2.0f * a), 0, sphAttr );
    }
}
```
One Last Example: Sphere Rendering

• Your hit shaders use this attribute for shading

Get material information

Find sphere normal from attributes

Shade

(see tutorial code for details)

Buffer<float4> gSphereMatls;

[shader("closesthit")]
void SphereClosestHit(inout Payload pay, SphereAttribs attribs)
{
  float4 sphMatl = gSphereMatls[ PrimitiveIndex() ];

  float3 hitPt = WorldRayOrigin() + RayTCurrent() * WorldRayDirection();
  float3 sphNorm = normalize( hitPt - attribs.sphCenter );

  if ( isDiffuseMaterial( sphMatl ) )
      pay.color = ...
  else if ( isMetalMaterial( sphMatl ) )
      pay.color = ...
  else if ( isGlassMaterial( sphMatl ) )
      pay.color = ...
}
Summary

Image: My Tutorial 14 running on freely available Amazon Bistro scene
Summary

• Overview of DirectX Raytracing shaders
  – Five new stages: ray generation, intersection, miss, closest-hit, and any-hit shaders
  – New builtin HLSL functions for ray tracing

• Walked through a number of samples demonstrating their use
  – Full code, additional tutorials, and runnable executables available online (intro-to-dxr.cwyman.org)

• I hope your takeaway:
  – The shader side of DirectX Raytracing: very similar to coding serial ray tracing in C++
  – The ugly API code can be encapsulated to allow for fast prototyping in a GPU-accelerated path tracer

Image: My Tutorial 14 running on freely available Amazon Bistro scene
Questions?