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

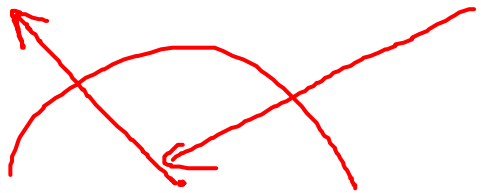
- Want global illumination?
  - Add a random outgoing ray
  - Recursive call: `shootColorRay()`
  - Account for BRDF
  - Add contributions together

- 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

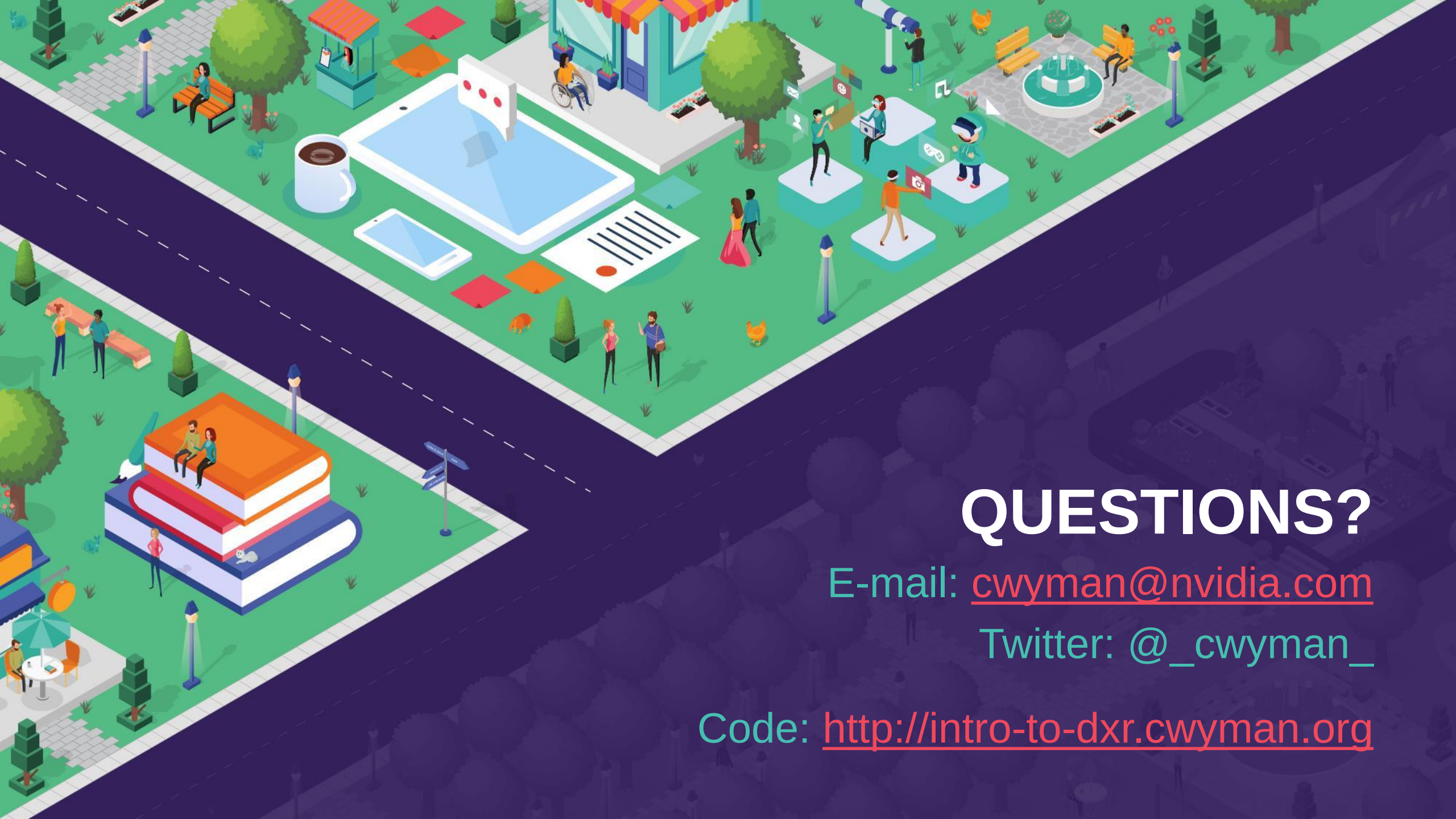
NAÏVE LIGHT SAMPLING:

- 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

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