2018/2019, 4th quarter

INFOGR: Graphics / *NeedToKnow*

Lecture 7 – Ray Tracing (2)

Author: Jacco Bikker

TL;DR

In this installment of 'NeedToKnow': more ray tracing. The concepts of surface normals and interpolated vertex normals are introduced and applied to shading. Reflections and recursive reflections for Whitted-style ray tracing are added.

This is the last lecture before the midterm exam.

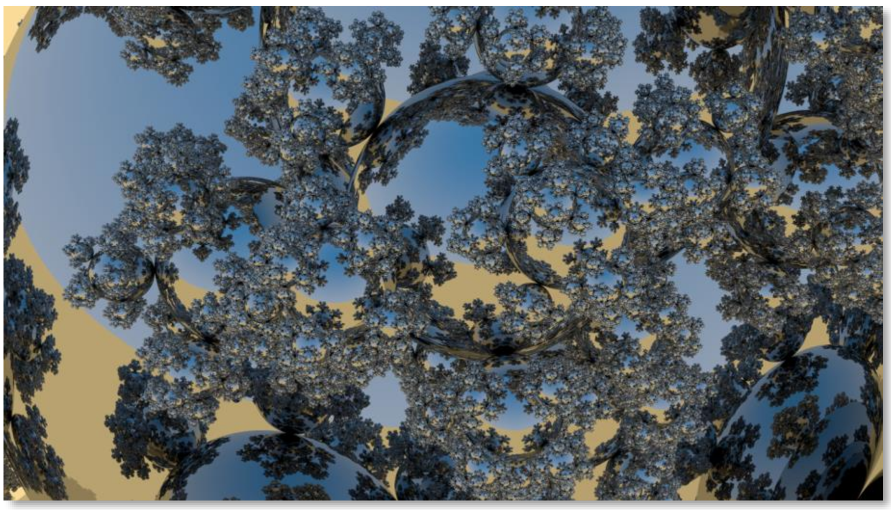


Figure 1: How to render a million reflective spheres in real-time remains a mystery.

Recap

In the previous installment of NeedToKnow we explored the principles of light transport. In a simple setup, a brief pulse of photons starts at a *pointlight* and spreads out as a sphere with a radius that increases at the speed of light. When the expanding sphere of photons encounters 'obstacles', these obstacles may *absorb*, *transmit* or *reflect* photons.

To calculate how many photons are reflected we need to know how many photons arrived. We are calculating this for a point, but since it's all about densities, we can establish photon density *per unit area* for a point. This density is proportional to the brightness of the light, divided by the squared distance.

The photons typically arrive at an angle. Depending on this angle the photons are spread out over a larger area. The density is multiplied by $N \cdot L$ to account for this.

Normals

A plane is conveniently defined using an implicit equation:

$Ax + By + Cz + D = 0$ (or: $\left(P \cdot \vec{N}\right) + D = 0$)

Quick intuitive verification of the maths: a horizontal (floor) plane at level 0 has a normal of $(0,1,0)$ (i.e., it points straight up) and consists of all the points for which $y = 0$. Based on the normal and the level, we get:

$0x + 1y + 0z + D = 0$

which reduces to: $y = 0$, which indeed defines all the points on the plane.

The second form now also makes sense: $\left(P \cdot \vec{N}\right)$ is $Ax + By + Cz$, if (A, B, C) is $\vec{N}$.

With this basic math we can do the calculations for light transport. The distance of a point P to a plane is simply calculated as $distance_{P,plane} = \left(P \cdot \vec{N}\right) + D$, and the cosine factor for light arriving at an angle becomes $\vec{L} \cdot \vec{N}$, where $\vec{L}$ is a (normalized!) vector from the point we are shading <u>to</u> the light. This is all illustrated in Figure 2.

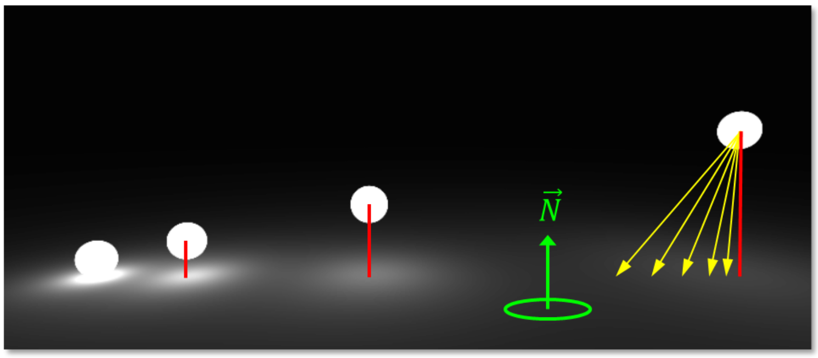


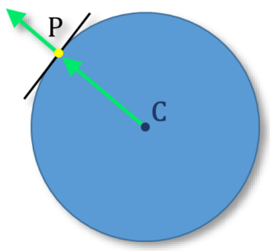
Figure 2: Distance attenuation and N dot L shading.

In some cases, we may get a negative value for $\vec{L} \cdot \vec{N}$. If this is the case, $\vec{L}$ must be a vector that arrives from underneath the surface, which means that the light source is below the surface. In that case, the surface blocks itself from the light. This has two consequences:

- the contribution of the light becomes 0
- we don't send a shadow ray, as this would be a waste of work.

Normals and Spheres

A normal for a sphere can be obtained easily. We typically want to know a normal for a point on the sphere. This normal is in the opposite direction of the vector from the point on the sphere to the center of the sphere. Don't forget to normalize.



Normal Interpolation

For a sphere we can calculate a perfect normal at any point on the sphere. For an object consisting of triangles things are not that simple.

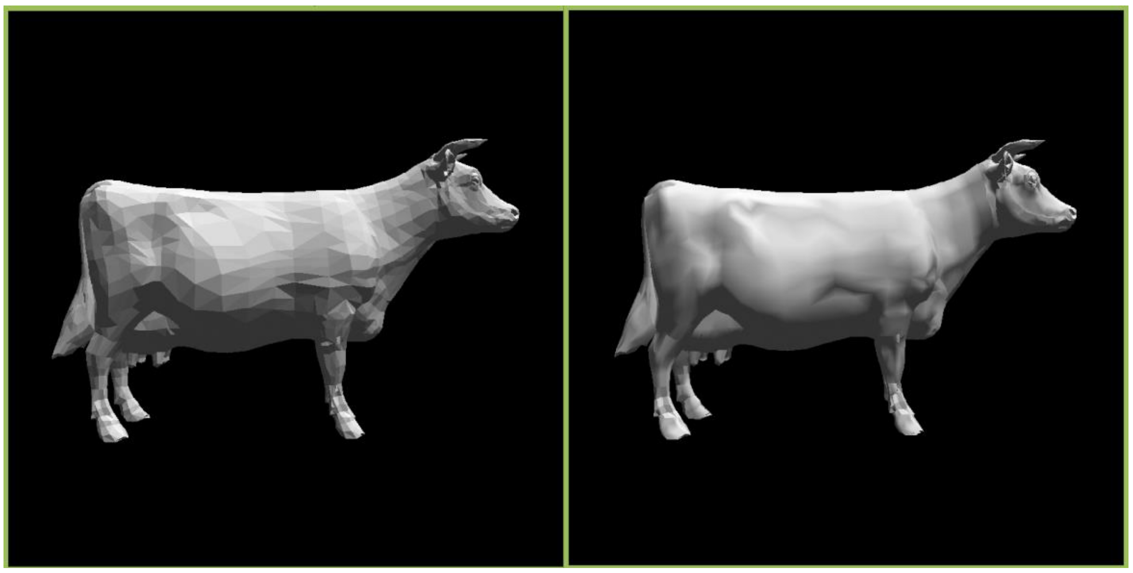


Figure 3: A cow rendered without and with normal interpolation.

The cow on the left does have perfect normals for each pixel, but the one on the right looks quite a bit better. It is shaded using a technique called *normal interpolation*, which relies on a counter-intuitive concept named *vertex normals*. The concept is illustrated in Figure 4.

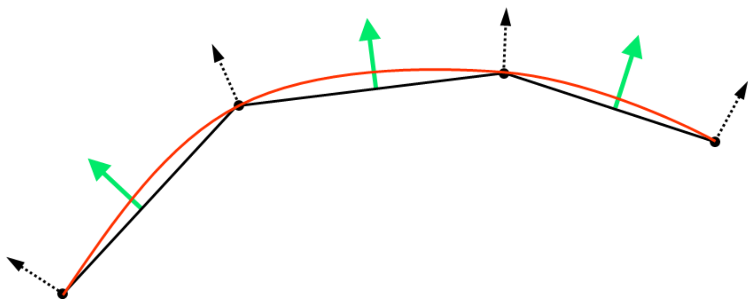


Figure 4: Normal interpolation.

A vertex normal is typically the (normalized) average of the *geometric normals* of the polygons that use the vertex. By interpolating the vertex normals over the object we get a smoothly varying normal. If we use this smooth normal in our $\vec{N} \cdot \vec{L}$ calculations, we get the desired smooth mesh.

Reflections

Using normal vectors we can also calculate *specular reflections*.

Given a surface normal vector N and a vector V to the eye the reflected vector is:

$$\vec{R} = \vec{V} - 2(\vec{V} \cdot \vec{N})\vec{N}$$

Interesting fact: if $\vec{V}$ and $\vec{N}$ are normalized vectors, $\vec{R}$ also has a magnitude of 1, so we don't have to normalize it (and thus we shouldn't; don't make the ray tracer slower than necessary).

Light Transport for Reflections

When a primary ray hits a mirror, it will 'see' what the reflected ray sees. The reflected ray is thus an extension of the primary ray. An *extension ray* may hit another mirror; ray tracing is thus a recursive process. In pseudocode:

```
vec3 Trace( Ray ray )
{
   I, N, material = scene.GetIntersection( ray );
   if (material.isMirror)
      return material.color * Trace( … );
   return DirectIllumination() * material.color;
}
```

Note that absorption should still be applied. For example, a shiny green ball absorbs red and blue light. Also note that we do *not* use a shadow ray. A shadow ray works because diffuse objects reflect light from all directions, but mirrors don't do this. The only direction from which relevant light arrives is the reflected direction.

The Full Whitted

We can now render diffuse and specular objects, and we can handle reflections. This (plus dielectrics) is what Whitted-style ray tracing is all about. We can sum up the whole process in two functions:

```
Color Trace( vec3 O, vec3 D )
{
   I, N, mat = IntersectScene( O, D );
   if (!I) return BLACK;
   return DirectIllumination( I, N ) * mat.diffuseColor;
}
```

The first function is what we have seen before. Based on a primary ray (with origin O and direction $\vec{D}$), we find an intersection point I, normal $\vec{N}$ and a material. If we hit nothing, the ray left the scene, and we're done.

Otherwise, we calculate the illumination arriving at point I.

```
Color DirectIllumination( vec3 I, vec3 N )
{
   vec3 L = lightPos – I;
   float dist = length( L );
   L *= (1.0f / dist);
   if (!IsVisibile( I, L, dist )) return BLACK;
   float attenuation = 1 / (dist * dist);
   return lightColor * dot( N, L ) * attenuation;
}
```

Function `DirectIllumination` is evaluated for each light source. It takes a primary (or secondary, etc.) intersection point I and a normal $\vec{N}$ at that intersection point. It then creates a shadow ray between the light and the intersection point. If the shadow ray hits nothing, the contribution of the light source is returned.

Adding reflections is now a simple change to the `Trace` function:

```
Color Trace( vec3 O, vec3 D )
{
   I, N, mat = IntersectScene( O, D );
   if (!I) return BLACK;
   if (mat.isMirror())
      return Trace( I, reflect( D, N ) ) * mat.diffuseColor;
   else
      return DirectIllumination( I, N ) * mat.diffuseColor;
}
```

Note that mirrors do not calculate direct illumination; everything they reflect arrives via the recursive call to `Trace`.

Recursion

Whitted-style ray tracing is also known as *recursive ray tracing*. The recursion comes at a price: one primary ray potentially reflects the light of many secondary and shadow rays. The branching path that we get is called the *ray tree*. Especially when we have dielectrics in the scene the tree can become quite complex.

This has some consequences:

- we need a cap on the recursion
- deeper rays tend to contribute little to the color of the pixel.

A deep ray tree is expensive to evaluate. We will solve this issue later, using path tracing.

Not in Whitted

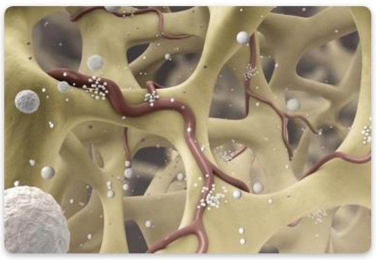
Finally, it's useful to consider what Whitted-style ray tracing cannot do:

- We do get shadows, but we do not get soft shadows.
- We get reflections, but they have to be specular.
- Mirrors reflect light from other mirrors, but diffuse surfaces do not reflect from other surfaces.
- Caustics are missing.
- How do we calculate depth of field and motion blur?

We will address these later.

THE END

That's it for this installment. If you have any questions, feel free to ask by email or on Slack!



INFOGR2019 – NEEDTOKNOW