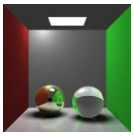


2018/2019, 4th quarter

INFOGR: Graphics / *NeedToKnow*

Lecture 6 – Ray Tracing

Author: Jacco Bikker

TL;DR

In this installment of 'NeedToKnow': ray tracing, shading, and a bit of textures.

Figure 1: This article was sponsored by NVIDIA.

Ray Tracing

In the previous installment of NeedToKnow the basics of ray tracing were discussed: how to define a ray, and how to aim it at a pixel on the screen. It is time to add some shading.

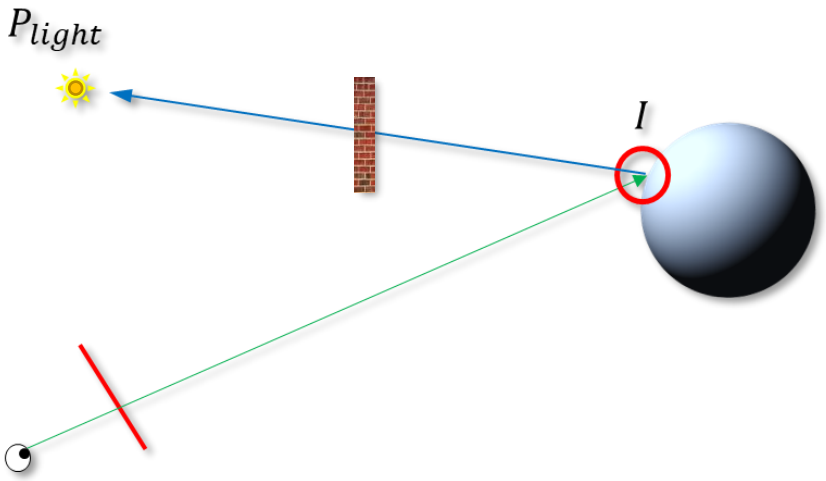
In the real world we see objects because there is light transport from a light source, via zero or more surfaces, to the sensor / camera / eye. A ray tracer mimics this process, although we typically work backwards. Doing things backwards allows us to ignore light paths that do not contribute to the image. It is also valid to go backwards, thanks to the *Helmholtz reciprocity principle* (eloquently defined by Wikipedia).

We have discussed three material types so far. The first is the *diffuse (or: Lambert) material*. When a ray reaches such a material, we attempt to complete the light transport path using a *shadow ray*. The second type are the *pure speculars*. Rays that encounter a pure specular bounce and proceed in a deterministic direction. And finally we have *dielectrics*. This includes water and glass. A dielectric typically transmits and reflects light transport paths. At a dielectric, the path thus splits.

Shadow Rays

The rays that originate from the camera are known as *primary rays*. We also call these rays *extension rays*: they extend the path, searching for the nearest surface along the ray. There is a second type of ray, which is a bit different. This is the shadow ray, or connection ray. These

rays are used to connect to a light source. A shadow ray does not care about the nearest intersection; it merely wants to know *if* there is an intersection. Since any intersection provides the desired answer, a shadow ray is typically cheaper to evaluate than an extension ray.



A shadow ray uses the same ray equation as an extension ray:

$$P(t) = O + t\vec{D}$$

However, instead of using infinity for t, we typically set t to the distance between the intersection point and the light, since we are not interested in intersections with geometry beyond the light.

Devils and Details

If there is no geometry between an intersection point and a light, the shadow ray should report 'no obstruction'. But what if we find an intersection at $t = 0$? The origin of the ray *is* on a surface after all. Requiring that $t > 0$ doesn't solve the problem: floating point inaccuracies will lead to *shadow acne*. We solve this issue using a small offset:

$$O += epsilon\ \vec{D}$$

The direction of the offset is $\vec{D}$. The magnitude of the offset is 'as small as possible'. As a rule of thumb, it is 10^{-5} to 10^{-6} times the world size. If we make it any smaller, floating point accuracy will reduce it to zero; if it is significantly larger, the gap becomes visible.

> A floating point number is stored in 32-bits and uses 1 bit for the sign, 8 for the *exponent* and 23 for the fraction (or *mantissa*). In scientific notation, and base 2, we then get: $(-1)^{sign} * 2^{exponent-127} * fraction$. The smallest fraction is thus $1/2^{23}$, which yields almost 7 digits of decimal precision, regardless of the exponent.

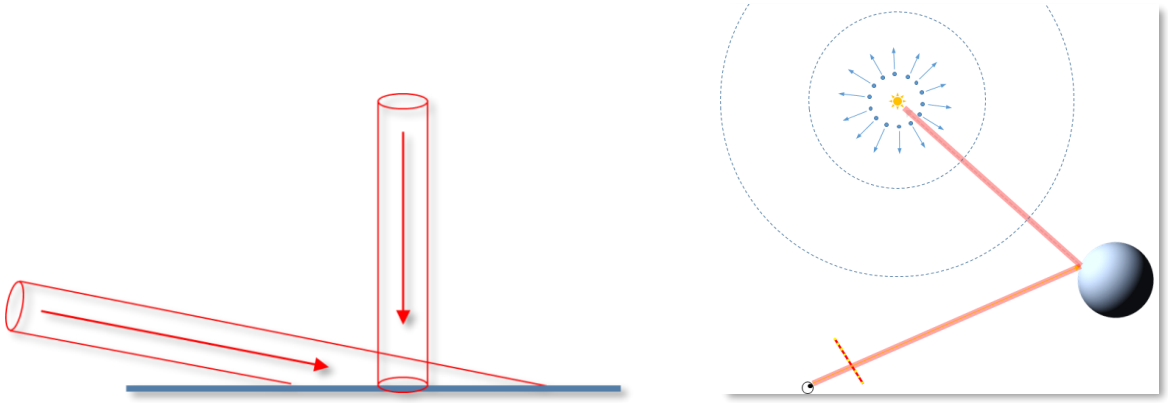
If we use an offset, we should reduce the length of the ray. And, while we are at it, we don't want to intersect the light either, so to be safe,

$$t-= 2\ epsilon$$

Counting Photons

How bright is a surface that gets illuminated by light sources?

- First, it depends on the number of visible light sources, and of course their brightness: each light adds to the illumination.
- Secondly, it depends on the distance r of each light source: light diminishes by $\frac{1}{r^2}$. This is known as *distance attenuation*.
- Next, it depends on the angle between the surface and a vector to the light. This is because the photons are distributed over a larger area when the surface is tilted away from the light. The relation is $cos\theta$, or simply the dot between the normal and the (normalized) vector to the light.
- And finally, it depends on how much the material reflects, i.e. the *color* or *albedo* of the material.



Diffuse shading thus becomes: `dot( N, L ) * (1 / (r * r)) * lightColor * materialColor.`

> A note on performance: the order of the calculations matters. 'ligtColor' is a 3-component vector (r, g, b), so multiplying it by a scalar requires three multiplications under the hood. In the proposed ordering, at least the first multiplication is a single operation. As soon as lightColor or materialColor is involved, the data type is promoted to vec3.
>
> Also: never ever calculate 'r squared' as 'pow(r, 2)'. The pow function is very expensive to evaluate. Even 'pow(r, 5)' is faster when written as r * r * r * r * r.

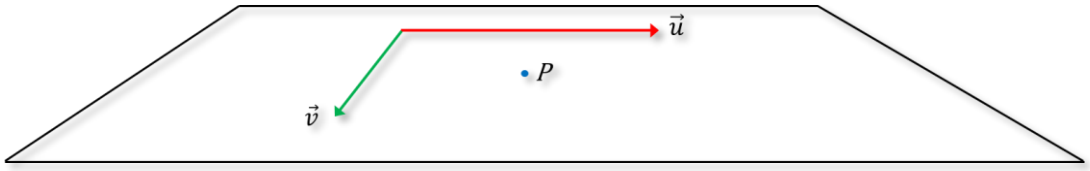
Textures

Material color often is defined locally, e.g. on this giraffe. We say that the material is *spatially variant*. To determine the color for a location, we first need to identify the location. A 2D mapping is convenient and saves us the cost of storing a 3D texture.

The 2D mapping is obtained using some math: for each triangle we chose two vectors in the plane of the triangle (ideally perpendicular but definitely not parallel), which define a basis. Any point on the plane (and thus the triangle) is now a combination of the basis vectors.

Once we can parameterize locations on the plane using two variables we can use this to generate a procedural texture, or, more commonly, a look-up texture.

Defining a point on the plane using two variables is easy enough, but for ray tracing, we often need the reverse operation: we already have a 3D position (calculated using a distance t along a ray and the ray equation $p(t) = O + t\vec{D}$), and we are looking for the two variables that we can use to read from the texture.



The above image shows that we can simply project point P on vector u to obtain a distance along $\vec{u}$ that we need to travel to get to P. The projection is done using the dot product. Likewise, we project P on vector $\vec{v}$. The texture coordinates are thus:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} P \cdot \vec{u} \\ P \cdot \vec{v} \end{pmatrix}$$

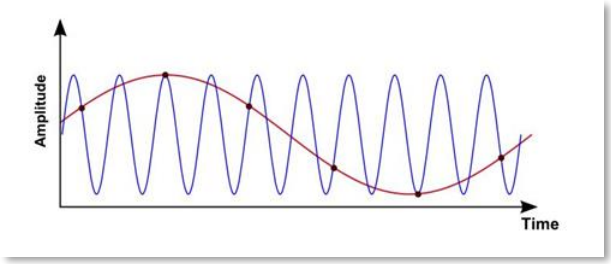
We still need to scale x and y by the texture dimensions. We may also want to *clamp* against the edges of the texture, or we can apply *tiling*.

> Tiling requires a modulo over x and y, so two per pixel. A modulo is as expensive as a division (i.e., quite substantial). For powers of 2 we can calculate the modulo in an efficient manner:
>
> x % 16 == x & 15, because 15 is '1111' in binary, so x & 15 yields the lowest four bits of any integer value. Unlike the modulo, the 'logical and' is very cheap to evaluate.
>
> Before hardware texture units became available many 3D engines therefore required texture sizes to be powers of two. This may also be a sensible requirement for your ray tracer, if you want it to run as fast as possible.

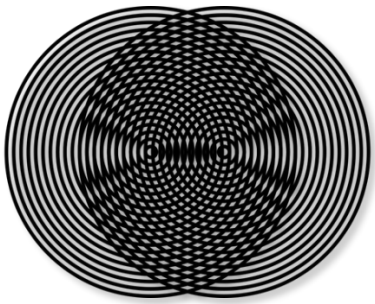
Sampling

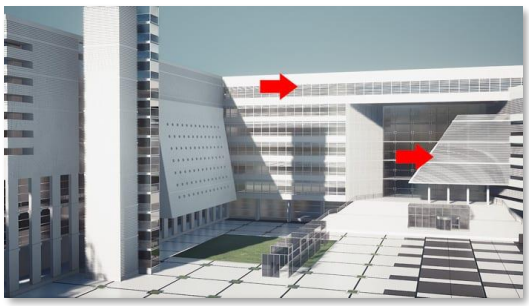
A texture map is a raster. When projecting it to another raster (the screen), we get al kinds of problems: aliasing, oversampling and undersampling.

Aliasing occurs when the frequency of the source signal exceeds the frequency of our samples:



On a raster this results in Moiré patterns.

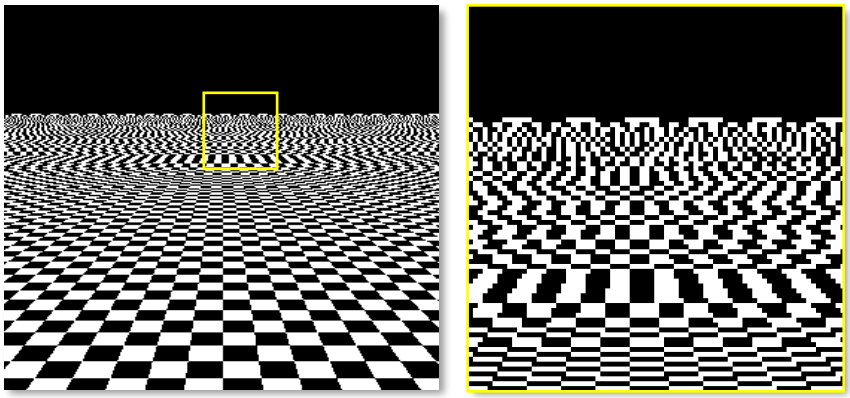




Oversampling occurs when we read the same discrete value many times:

A proper fix for this issue is to increase the resolution of the texture. If this is not an option, we can at least use interpolation to smooth out the hard edges of the individual texture pixels (*texels*). In two dimensions this interpolation is called *bilinear interpolation*.

Undersampling is the opposite effect: it happens when we skip discrete values.



A proper fix for undersampling is to read more than a single texel. This can be done using *MIP-mapping*. Here, the smaller maps are calculated by averaging four pixels at a time to one pixel. Reading this single pixel thus effectively is a 2x2 pixel read.

We can also combine this with bilinear filtering, to get *trilinear filtering*. For trilinear filtering, we blend between MIP-map levels.

THE END

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