CS 314: Lights, More Shading, Recursive Raytracing

Robert Bridson

October 21, 2008

1 Normals in Diffuse Shading

Last time we finished with the basic formula for diffuse (or matte, or Lambertian) shading: the color of a lit diffuse surface is

$$CL(\vec{l} \cdot \hat{n})$$

where C is the RGB colour vector of the surface material (how much light is reflected in each colour band), L is the RGB colour vector giving the intensity of the incoming light, $\vec{l}$ is a unit-length vector pointing towards the light source, and $\hat{n}$ is the unit-length surface normal. It does not depend on the viewing direction, since diffuse surfaces reflect light equally in all directions.

We should be a bit more precise. The surface normal $\hat{n}$ should be the outward pointing normal if this is a surface of a volumetrically-defined object (i.e. which has a well-defined outside and inside). If the surface is perfectly thin, and there's nothing to tell $\hat{n}$ and $-\hat{n}$ apart, then you must pick the one that faces the viewing direction—i.e. the one that has a negative dot-product with the ray traced from the camera to this point.

On top of this, we need to make sure we handle the case where $\vec{l} \cdot \hat{n} < 0$, i.e. where the surface we are looking at is oriented away from the light—and should get zero illumination, not negative colour. Thus we could use instead

$$CL \max(0, \vec{l} \cdot \hat{n})$$

2 Lighting

Before proceeding deeper into shading models, let's stop and consider the sources of light in a rendering. We'll start with fairly useful models that are simple to implement.

2.1 Distant Lights

If a light source is much, much further away from the scene than the size of the scene—for example, the Sun when rendering scenes on Earth—then a very good approximation is that the light is infinitely distant, and all incoming light rays are perfectly parallel, carrying a constant intensity: if you move around on the ground (modulo the curvature of the Earth) or if you move higher towards the Sun, the direction and intensity of sunlight hardly changes at all.

We represent such a **distant light** with a unit length direction vector $\vec{l}$ telling us where the light is coming from (or alternatively where it is going towards) and an RGB colour vector L representing the intensity of light in each colour channel. These numbers can be plugged directly into the diffuse shading model above, for example.

2.2 Point Lights

Another very simple light source is the **point light**: light emanating from an infinitesimally small point in space. This is a useful model for real lights that are very small but local to a scene, like a lightbulb in a room.

We can represent a point light with a location, $\vec{p}$, and a total power output L which is an RGB colour vector. Obviously then the direction to the light at a surface point $\vec{x}$ is:

$$\vec{l} = \frac{\vec{p} - \vec{x}}{\|\vec{p} - \vec{x}\|}$$

The intensity of the light at the surface is a little more subtle, however. Imagine a sphere of radius r centred on the point light: the light intensity across the inside of this sphere should be constant (assuming the point light emits light equally in all directions). The total power absorbed by this sphere should be independent of r, and be equal to the power emitted by the light: it is capturing every photon the light emits, and a photon doesn't change its energy depending on how far it goes. The surface area of the sphere is $4\pi r^2$, and therefore the intensity of light at the surface (total power or number of photons divided by area) must scale like $1/r^2$: an **inverse square law**. Thus for any surface point, we have to find the distance to the light D, and scale the power of the light (an RGB vector) by $1/D^2$ to get the light intensity there.

(As an aside: you might be wondering where the 4π factor went. If we were being fully correct physically, making sure energy is conserved etc., then we would have to be careful with these factors; however, right now we're just aiming for plausible formulas and so we strip off terms like that.)

2.3 Ambient Light

So far we've discussed **direct** illumination: lights shine directly on the surface. However, in most real scenes a significant amount of illumination is actually indirect, the result of light reflected off other surfaces in the scene before hitting the surface of interest. Unfortunately there is probably an infinite variety of different paths light can take indirectly to get from a source to a surface via other surfaces: every time light hits a diffuse surface it is scattered in all directions. Accurately approximating all of these possible paths is fairly challenging.

For many applications, a very crude approximation can suffice: **ambient light**. The idea is that at every point in the scene, there is roughly an equal amount of indirect light bouncing around in every direction, due to the huge number of different paths. We can then ask the user to specify a single RGB vector, the ambient light intensity, that gets added to the illumination on each diffuse surface. It doesn't have an associated direction, so there is no dot-product to scale it by.

This of course is ridiculous in some scenarios: for example, an almost completely closed box will be much darker inside than the room around it, and a single constant ambient light isn't going to model that. However, it's a great first hack at the problem, efficiently filling in missing light in all the places direct lighting doesn't reach.

2.4 Artistic Lights

If the ultimate goal of the rendering is just to get an attractive image, as we've more or less assumed so far, nothing constrains us from modifying any formulas in a non-physical way that can help the artist. For example, the $1/D^2$ law in the point-light may be awkward because objects too close to the light get very brightly lit—which is accurate, but potentially inconvenient if we don't want them highlighted that way—and so a different fall-off function like $1/(D^2 + \epsilon)$ might be nicer.

2.5 Area Lights

On the other hand, in some situations we can get a much nicer look if we hew closer to physical reality—or we may require some degree of physical accuracy from the rendering (for example, if it's being used by an architect to evaluate a design). Real lights are neither infinitely distant nor infinitesimally small points, but instead are a volume or area of space which emits light, such as the filament in a light bulb. In other cases there might be a small light source which passes through a diffusing material which can then be thought of as a much bigger light source, such as the sky.

These light sources, spread out over a region, can be classed specifically as volume lights, area lights, or linear lights (light emitted along a curve), but generally are lumped together as **area lights**. We can't handle them directly in a plain raytracer, unfortunately, since an area light implies infinitely many different incoming light directions at a surface point, and we can only handle finitely many—but we'll come back to that issue later.

3 Evaluating Diffuse Shading

Once we have our light source models, we can finally put it together with the diffuse shading equation above, and properly evaluate the colour of a diffuse surface at a surface point $\vec{x}$ with normal $\hat{n}$:

$$C_{\text{lit}} = C_{\text{diffuse}} \left(L_{\text{ambient}} + \sum_{\text{light } i} L_i(\vec{x}) \max \left[0, \vec{l_i}(\vec{x}) \cdot \hat{n} \right] \right)$$

Here the symbols are:

- C_{lit}: the final output colour, an RGB vector
- C_{diffuse}: the diffuse colour of the surface material, an RGB vector
- L_{ambient}: the ambient light intensity, an RGB vector
- $L_i(\vec{x})$: the intensity of the i'th directional light (distant or point) evaluated at $\vec{x}$, an RGB vector
- $\vec{l_i}(\vec{x})$: the unit-length direction towards the i'th directional light evaluated at $\vec{x}$

Note that we can sum up all the different contributions from the light, since (at a more fundamental level we won't delve into) the equations of light are linear. This is basically the diffuse part of the OpenGL model.

3.1 Shadows

However, let's reconsider the sum over all lights. The main interesting elaboration is if one or more of the lights are blocked by another object in the scene, or in other words, if the surface is in the shadow of another object. In Z-buffer systems handling shadows is nontrivial, but we can really easily add them to a raytracer. Before including a light in the sum above, we just have to shoot a **secondary ray** from $\vec{x}$ to the light (so its origin is $\vec{x}$ and its direction is $\vec{l_i}$: if the ray intersects any other object, then the point is in shadow and we skip over the light.

This is the first of several uses for secondary rays (where the primary rays are the one we shoot from pixels to directly determine the image). For this use they get the special name **shadow rays**; we'll see other uses soon.

Secondary rays are shot starting from a surface point. However, this is a little dangerous numerically: depending on rounding error, it's quite possible that the surface itself will register as an intersection with the ray. One of the simplest hacks to get around this is to start the ray a little distance away from the surface, starting it at a parameter value of $t = \epsilon$ instead of $t = 0$, where $\epsilon > 0$ is a small value chosen by the programmer or user. It should be big enough to cover for rounding error but not so large as to cause visible artifacts—for typical length scales in a scene on the order of 1, an ϵ value of 10^{-5} is quite reasonable. Without this hack, occasionally a surface point will erroneously register as being in the shadow of itself, causing black dots to be scattered over the surface—this is sometimes known as **surface acne**.

Shadow rays are special for two additional reasons. One is that they might have a natural maximum parameter value: you want them to stop at a light source (such as a point light) and not continue past—since an object on the far side of a light won't cause a shadow at the point being tested. The other reason is that it doesn't matter which intersection we find—whether it's the first or not is irrelevant—which allows for potentially faster intersection code; we also don't care about the surface normal etc. at the intersection, just that an intersection exists. Most raytracers implement a special version of their intersection functions designed just for shadow rays that simply determines the existence of an intersection.

4 Mirror Surfaces

We turn now to almost the exact opposite of a diffuse surface, a perfect mirror. Here an incoming light ray is reflected in exactly one outgoing direction (the geometric reflection), instead of all directions uniformly. These materials are also very difficult for Z-buffer renderers, but fit beautifully in the raytracing framework.

The first thing is to work out the reflected direction of a vector $\vec{v}$ with respect to a surface with normal $\hat{n}$. At least conceptually we want to write $\vec{v}$ in terms of a normal component (orthogonal to the surface) and a tangential component (parallel to the surface): the reflection should keep the same tangential component but negate the normal component. The normal component of $\vec{v}$ is just $(\vec{v} \cdot \hat{n})\hat{n}$, and we can build a vector with it negated simply by subtracting *twice* the normal component from $\vec{v}$ (just like $-x = x - 2x$ for a real number x):

$$\vec{r} = \vec{v} - 2(\vec{v} \cdot \hat{n})\hat{n}$$

Suppose in the raytracer we have fired a ray with direction $\vec{d}$, it hits a mirror surface, and we want to know what colour is transmitted back along the ray. We simply need to build a new secondary ray with origin at the surface intersection point and a reflected direction $\vec{d} - 2(\vec{d} \cdot \hat{n})\hat{n}$, then compute the colour that the new ray sees: that colour should be reflected back to the original ray.

What we have decribed is a **recursive raytracer**: to find the colour on a given ray, if it hits a mirror surface we recurse on the secondary reflection ray and return that colour. If the secondary ray hits another mirror surface, the recursion goes even deeper.

Of course, most mirrors aren't perfect: they might absorb some of the light instead of reflecting it, and thus we can introduce a material parameter between 0 and 1 that scales the light intensity for each reflection by multiplication. We can make this material parameter different for each colour channel (so it's an RGB vector) to account for some materials such as gold, which reflects yellow colours much better than blue (hence it's apparent colour). In fact, we can further add the mirror model on top of diffuse shading to get surfaces like polished opaque plastic.

These secondary rays are full-fledged rays, unlike the shadow rays we saw before (which didn't need to find the first intersection), but they too need the hack starting them a little distance away from the surface to avoid rounding error problems.

Finally, it should be clear that in some mirror-rich scenes, the depth of recursion might be considerable. It's a wise idea to have a maximum limit on recursion to avoid the raytracer getting stuck (or overflowing the stack)—for example, by passing in a current maximum depth to the raytrace function which is passed as one smaller to recursive calls (which are made only if the depth is nonzero). Except in very special circumstances most people would have a hard time noticing if, say, the sixth recursive reflection and more are missing, so this maximum depth can usually be quite small.