Shadows

Why Shadows are Important

Shadows are important in ray tracing for two main reasons (at least as far as we are concerned):

1. They allow us to provide either *quantitative* or *qualitative* answers to questions referring to the relative depth of objects in relation to the camera, number and types of light sources, relative positions of objects, amongst others.
2. They allow us to render more accurate scenes, as the absence of shadows makes images look "fake".

Definitions

For directional and point lights, a shadow is a part of the scene that doesn't receive *any* illumination from a given light source because one or more objects block the illumination. A different way of looking at this (which is used for real-time graphics) is that shadows are the parts of the scene that the light can't "see".

The shadows of point and directional lights are hard-edged because surface points either receive full illumination or no illumination from these types of lights. If we discuss real lights however, things get more complex, as these have a finite area. These lights will cast *soft shadows* and consist of the following components:

- *Umbra*: the area of the shadow where no light is visible,
- *Penumbra*: the area of the shadow where some light is visible.

In *very* broad terms, the smaller the light source in relation to the object, the smaller the penumbra. As the light source increases, the umbra shrinks down while the penumbra increases. This further justifies the notion that the shadow is the area that the light can't "see".

Implementation

Implementing shadows is relatively straight-forward. After we have computed the ambient colour, we then need to test whether or not the point is in shadow. If it is, then the final radiance for that pixel is the ambient colour. If it isn't, then the radiance is computed based on the material of the surface. The question now becomes: how do we know whether an object is in shadow or not?

The way we are going to do this is to cast *shadow rays*. These are a type of secondary ray, which describes the set of rays whose origin is not the camera (or the viewing plane). The shadow ray is computed by taking the point of intersection on the surface as the origin and the direction is then computed based on either the direction of the light source (for directional lights) or the vector that goes towards the light source (for point lights). We then cast this ray and

check for intersections against the objects in the scene. If the ray intersects with any objects, then the point is in shadow. If no objects are hit, the point is not in shadow. Notice that for this type of intersection we don't actually care about any of the information that we require for shading. All that we are interested in is whether the ray hits something or not.

The Epsilon Factor

We established that the origin of the shadow rays is the surface itself. It would therefore make sense to assume that if we evaluate $r(t)$ at $t = 0$ we would get back the point on the surface that we started with. What we will find however, is that this is not the case. The reason behind this is that any of the floating point types (`float` and `double`) all have a limited precision. This means that when performing the ray calculation, the number that comes out may not be *exactly* the one we're expecting. Instead, the number we get is $n \pm \epsilon$ for some small ϵ. So what is the problem? If the number we get is $n - \epsilon$ then that's fine, since the point is technically inside the surface itself and therefore there isn't a problem. The issue starts when we get $n + \epsilon$. In this case, the ray is computed as having started *above* the surface. This means that when the ray intersects against the object that we are currently shading, the `hit` function will return true and therefore the object will be deemed to be shadowed *by itself*! This phenomenon is known as *self-shadow aliasing* or *salt-pepper noise*.

This behaviour is intrinsic to any floating point operation that we perform (hence why you will find the use of `atlas::core::areEqual` in the lab code, since you should *never* compare floats directly), so what we have to do is reduce the error as much as possible. To do this, we will add a check in the `hit` functions for each geometry. If the computed value of t is less than some ϵ, we set $t = 0$ and say that there is no intersection. We could in theory use the same ϵ for all surfaces, but because each one is distinct (and their calculations introduce different types of errors) then it makes more sense for each one to have it's own. A starting value for your ray tracers can be $\epsilon = 0.0001$.

Costs

Shadows can be more expensive than the primary rays. For every hit point, we need a shadow ray for each directional and point light. If there are enough lights in the scene, the cost of the shadows will exceed the cost of the primary rays. The situation is similar for scenes with multiple area lights.