Introduction to Shading

Contents

Structure of this Chapter

- Learn how to add shadows to an image.

Lights & Shadows

In the adjacent image we can see a diffuse sphere and a diffuse plane illuminated by a directional light source. What's clearly missing now to this image to make it more realistic is the shadow of the sphere on the plane. Fortunately, simulating the action of objects casting shadows on over objects in the scene is pretty straightforward to simulate in CG. Well in fact, it depends on the algorithm you are using to solve the visibility problem.

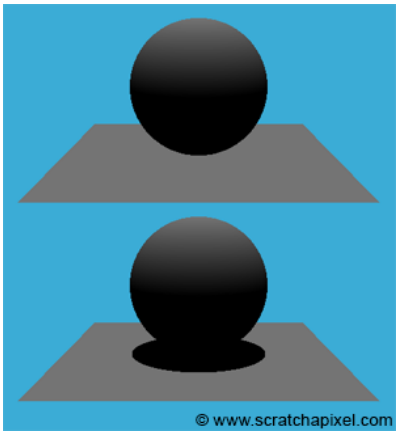

Figure 1: shadows are important as a visual clue to evaluate the position of objects with respect to each other.

- If you use rasterization, adding shadows to the scene can hardly be done in one single pass. It requires to precompute the visibility of objects from the point of view of the light which is stored in a special sort of image called a **shadow map**. You need to compute one shadow map per light in the scene, and this process needs to happen before you render the final image. We are planning to write a lesson on this technique in the future.
- If you use ray-tracing, computing shadowing can be done while the main image is being rendered (it doesn't require to precompute a special image such as a shadow map). All we need to do is cast a ray from the object visible through a particular pixel of the frame, from the point of intersection to the light source. If this ray which we call a **shadow ray** intersects an object on its way to the light, then the point that we are shading is in the shadow of that object.

As mentioned many times before, in the real world, light travels from the light source to the eye. Shadows are the consequence of some objects along the light path blocking the light from reaching another surface that is located further along the way. Though, as you know by now, in CG, it is more efficient to trace the path of light from the eye to the light source. Thus, we first trace a camera or primary ray, then find an intersection point (the surface that is visible for the pixel from which the primary ray was cast) and then trace a shadow ray from that point of intersection to the light. The difference between what happens in nature, and how we compute shadows in CG is shown in figure 2.

Adding shadows is really simple in a ray-tracer. If the primary ray intersects an object, we then compute the intersection point and trace one more ray in the light direction (or more precisely, in the opposite light direction). This is simple, since we already have all the code we need to compute the intersection of a ray with the objects making up the scene. For now, we will only learn how to add shadows from a directional light source. Later in this chapter, we will learn how to do the same thing for spherical lights. Here is the code:

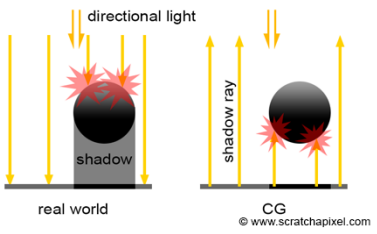

Figure 2: in nature light rays emitted by light sources by the first opaque object they encounter

on their way. In CG, we cast rays from the shaded point in light direction. If the ray intersects an object then the point is in shadow.

```
001 Vec3f castRay(
002     const Vec3f &orig, const Vec3f &dir,
003     const std::vector<std::unique_ptr<Object>> &objects,
004     const std::unique_ptr<DistantLight> &light,
005     const Options &options)
006 {
007     Vec3f hitColor = options.backgroundColor;
008     IsectInfo isect;
009     if (trace(orig, dir, objects, isect)) {
010         Vec3f hitPoint = orig + dir * isect.tNear;
011         Vec3f hitNormal;
012         Vec2f hitTexCoordinates;
013         isect.hitObject->getSurfaceProperties(hitPoint, dir, isect.index,
014             isect.uv, hitNormal, hitTexCoordinates);
015         Vec3f L = -light->dir;
016         IsectInfo isectShad;
017         bool vis = !trace(hitPoint + hitNormal * options.bias, L,
018             objects, isectShad, kShadowRay);
019         hitColor = vis * isect.hitObject->albedo * light->intensity *
020             light->color * std::max(0.f, hitNormal.dotProduct(L));
021     }

    return hitColor;
}
```

As you can see, we just call the trace function again (line 16) and set the variable `vis` to `false` if the trace function returns true (the intersection point is in shadow, thus the point is not illuminated by the directional light source) and `true` otherwise (the intersection point is not in shadow, thus the point is illuminated by the light source). All there is left to do is multiply the color of the intersection point by the result of this variable. If the point is in shadow, the point is black. If `vis` is set to true, then the color of the point is left unchanged (line 17).

Optimizations

Note that when we compute whether a point is in shadow of an object, the order in which the objects are intersected by the shadow ray doesn't matter. It doesn't matter if the object that the shadow ray intersects is not the closest object to the ray's origin. All we care about in this case of a shadow ray is to know if the ray intersects an object at all in which case the point from which the shadow ray was cast is in shadow. Practically, this means that for shadow rays, we could very well return from the `trace` function as soon as we intersect one object in the scene. This can potentially save some computation time.

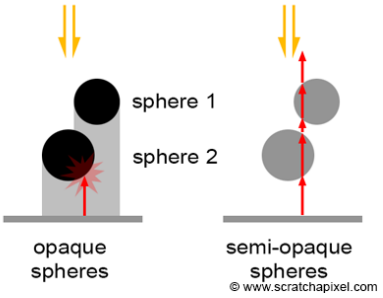

Figure 3: semi-transparent surfaces block light partially.

Though this optimization is only possible if all the objects in the scene are fully opaque. If some of the objects are transparent, then light rays which are potentially being attenuated while traveling through these semi-opaque objects, should definitely keep traversing the objects until they either encounter an opaque object on their way to the light or reach the light itself (only in the case of spherical lights, not in the case of distant light which are assumed to be located at infinity). In this section we will not show how to handle semi-opaque objects for shadow rays. Trying to optimize the code for shadow ray if you plan to support semi-opaque objects later on, is really not worth the pain but is left as an exercise if you wish to.

Shadow-Acne: Avoiding Self-Intersection

One problem often occurs though when ray-tracing is used to compute shadows. This problem is knows as **shadow-acne** and can

be seen in the adjacent image (top). It is a visual artefact that appears in the shape of small black dots on the surface of objects. Because of some small numerical errors introduced by the fact that numbers in our 3D engine can only be represented within a certain precision, sometimes, the intersection point is not exactly directly above the surface, but slightly below. When this happens and that a shadow ray is cast in the light direction, then rather than intersecting no object at all or some other object above the object surface, the shadow intersects the surface from which it is cast. In other words, we have a case of self-intersection. Of course an object can very well cast a shadow on itself, but in this particular case, it is clearly a mistake, as it only happens because the intersection point is below the surface rather than just on the surface. This situation is depicted in figure 4 (middle). The primary ray intersects a surface, but the point of intersection computed from the ray origin and direction as well as the intersection distance t happens to be slightly below the surface rather than just on top of it. Shooting a shadow ray from that position leads for the shadow ray to intersect the surface that the point should be on, and the point is marked as being in shadow.

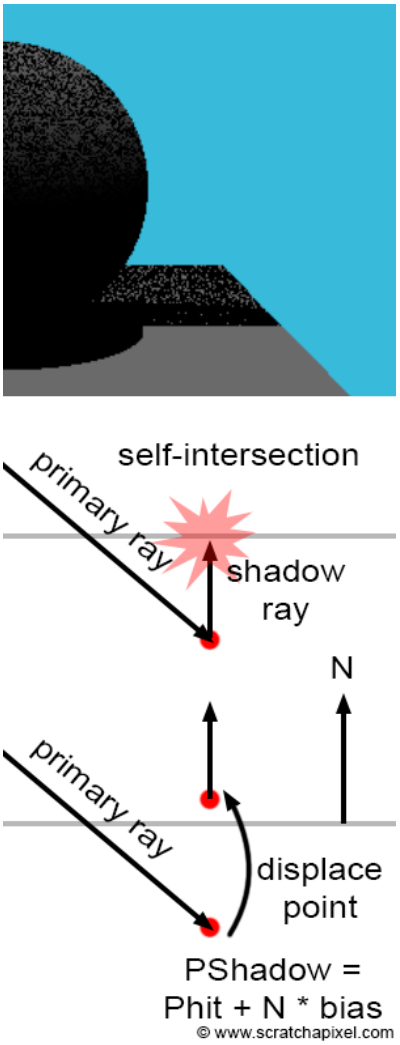

Figure 4: ray tracing acne is caused by numerical precision errors. Because of numerical precision the shadow ray origin is below the surface from which the ray is cast, causing a self-intersection.

Different methods can be used to alleviate this problem. One solution is to use double-precision rather than single-precision floating-point numbers. In other words, using `double` instead of `float`. Though this doesn't eliminate the problem totally. It just makes it less likely to happen. Another simpler and more robust solution, consists of systematically displacing the origin of the shadow ray in the direction of the surface normal, to move it with above the surface (figure 4, bottom). The amount by which you displace or move the point in the normal direction is left to the user and can be tweaked on a scene basis. This value is often refer to in ray-tracer as **shadow bias**.

> The term shadow bias is also used with shadow maps. Check the lesson on shadow maps [link].

In code, you would get:

```
001 struct Options
002 {
003     ...
004     float bias = 1e-4;
005 };
006
007 Vec3f castRay(...)
008 {
009     ...
010     bool vis = !trace(hitPoint + hitNormal * options.bias, L,
011          objects, isectShad, kShadowRay);
012     ...
    }
```

As you can see the bias is generally a very small value. The amount of bias required depends on different factors such as the scene scale, the object curvature, the object distance to the camera, etc. As mentioned before, this is the easiest way of getting rid of shadow-acne, though first it needs to be tweaked on scene to scene basis, and more importantly, by changing the actual position of the point

from which the shadow ray is cast, we get a result which is slightly different from the result we would get if we had cast the ray from the "actual" point on the surface. The higher the bias, the larger the difference between the shadow we should get and the one that appears in the image (as shown in figure 5). Tweaking the bias value is thus a balancing act. We want it high enough so that no acne shows up in the final frame at all, while keeping it as small as possible to reduce the effect of shadow bias (the technique of pushing the ray below or above the surface is also used when we cast reflection or refraction/transmission rays. Check the chapter from this lesson on simulating reflection and refraction).

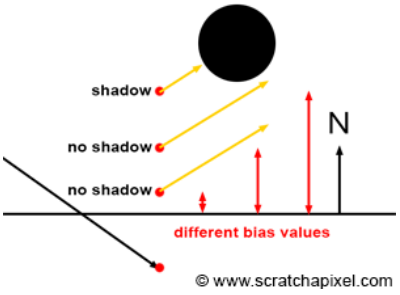

Figure 5: a large bias causes the point to appear in shadow when it shouldn't

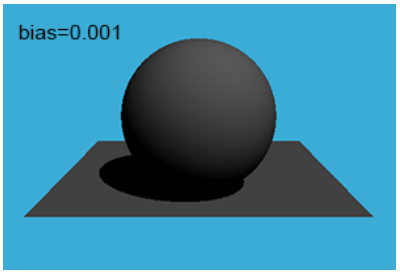

Figure 6: increasing the bias shifts the shadow position.