5 Shading for Raytracing

Shading is perhaps at its most interesting, and essential, in the context of Z-buffer rasterization and Reyes where any lighting effects, from shadows to reflections to the more exotic, must be done "manually" with shaders, which we'll see in the next section. By contrast, many of these effects come practically for free in ray-tracing[8] — and on the other hand, as we saw, engineering displacement shading is a little trickier in a raytracer. This can lead to a different flavour of shading for raytracers, particularly ones that aim to capture physically realistic effects.

From the standpoint of rendering as a simulation of the physics of light, one of the most basic concepts is the *Bidirectional Reflectance Distribution Function* (BRDF) of a point on surface. In a nutshell this describes how much light is reflected from any incoming angle to any outgoing angle. For example, an ideal Lambertian diffuse surface reflects the power of all incoming light equally in all outgoing directions; a perfect mirror reflects only the light coming in at the reflected angle around the normal to an outgoing direction. The notion of BRDF includes both of these examples and everything in between.

To define this a bit more precisely, first let's define how we actually measure the power of a light "ray". The radiometric quantity of greatest interest for raytracing is *radiance*, which measures the power of light per unit area on the emitting surface and per unit solid angle of the direction it's going in. At an intuitive level, you can view it as the "intensity" of light sampled on a ray.

When a light ray hits a surface head on, i.e. at angle 0° from the normal, the full radiance of the ray is felt. However, at other angles, the power of the light ray is spread out proportional to the cosine of the angle. If you imagine the light ray having an infinitesimal cross-sectional area, when it

[8] Of course, just as you can write a raytracer inside a Reyes shader, you can also use the same shading tricks for shadows etc. inside a raytracer, but there's not much point.

shines down along the normal the area that's light up on the surface is exactly the cross-section; when it shines at a shallower angle, the area on the surface that's lit up is stretched to be broader, diluting the concentration of light.[9]

Before getting to the precise definition of BRDF, it help to also define exactly how to measure the direction of incoming and outgoing light rays. The natural coordinate system to use is spherical coordinates, with θ being the angle between the ray and the surface normal, and ϕ being the other angle (with respect to some special tangent direction on the surface, but we'll ignore exactly how that's defined for now).

Putting this together, a light ray with radiance L_i coming in at angles (θ_i, ϕ_i) to some point on the surface will be providing $L_i \cos\theta_i$ light concentration. For any outgoing ray direction with angles (θ_o, ϕ_o), the BRDF multiples the incoming by some factor to calculate the contribution to the reflected radiance, more or less.

It gets a bit trickier since we have to account for all incoming light directions, integrating over all of their radiances, to get the total radiance out along some ray. Integrating the incoming directions over the hemisphere above the surface means a double integral from $\theta_i = 0$ to $\pi/2$, from $\phi_i = 0$ to 2π, This leads to the following equation for the outgoing radiance L_o as a function of its direction:

$$L_o(\theta_o, \phi_o) = \int_{\phi_i=0}^{2\pi} \int_{\theta_i=0}^{\pi/2} f(\theta_o, \phi_o, \theta_i, \phi_i) L_i(\theta_i, \phi_i) \cos\theta_i \, dA(\theta_i, \phi_i), \quad (3)$$

where $f(\theta_o, \phi_o, \theta_i, \phi_i)$ is the BRDF, and the area element $dA(\theta_i, \phi_i)$ is just $\sin\theta_i \, d\theta_i \, d\phi_i$ (remembering how to integrate with spherical coordinates).

The simplest example of a BRDF is the ideal Lambertian diffuse model:

$$f_{\text{Lambertian}}(\theta_o, \phi_o, \theta_i, \phi_i) = 1. \quad (4)$$

[9] This quantity, the power density of the incoming light at a point on a surface, is called *irradiance*.

Of course, a constant value between 0 and 1 could be used for a darker diffuse surface which also absorbs some of the light. In this case the reflection doesn't depend on either the incoming or outgoing angles at all.

The BRDF looks like a 4D function, but it's actually a bit more general — it's a "distribution". For example, a perfect mirror surface involves the Dirac delta:

$$f_{\text{mirror}}(\theta_o, \phi_o, \theta_i, \phi_i) = \frac{\delta(\theta_o - \theta_i, \phi_o - \phi_i - \pi)}{\cos\theta_i}. \tag{5}$$

This is zero when the outgoing direction is not the reflection of the incoming direction, but "infinite" enough that when they match, the integral ends up just copying the incoming radiance to the outgoing radiance on the reflected direction.

Glossy materials tend to be largest near the reflected direction, but nonzero at other angles too. Some materials like brushed metal or fabrics have a noticeable dependence on the ϕ angles, so they reflect light differently depending on how the surface is oriented. Many materials tend to reflect light more towards grazing (nearly tangential) angles, or back in the direction of the incoming light. There are also several other properties which any physically plausible BRDF should satisfy, such as energy conservation (the total power of the outgoing light can't be more than the total of the incoming light).

The BRDF itself has several generalizations. One of the most important is the "spectral BRDF", which in fact is so common it's almost always assumed: it basically means the BRDF for a particular frequency of light. Materials often have colours, i.e. they reflect light differently depending upon the frequency. Typical graphics software can specify a different BRDF for each of the R, G, and B channels. Another generalization allows for light to be transmitted through a partially transparent surface, not just reflected, or even to bounce around inside a material before emerging at a different location ("subsurface scattering").

In most cases, other than perfect mirrors, exactly evaluating the integral in

equation (3) is really hard, especially if the incoming distribution of light is complicated. The classic raytracer solution is to use Monte Carlo sampling, just like we discussed for antialiasing, depth-of-field, and motion-blur in both Reyes and raytracers. A finite set of random incoming light directions are chosen (possibly just from a uniform distribution around the hemisphere, but it's much more efficient to concentrate the samples near directions where the most light will be reflected — search for the term "importance sampling" to learn more about this) and the radiance along those is evaluated through raytracing; the results are weighted appropriately with the BRDF etc. and summed to approximate the integral.

Coming back to the question of what a shader should do in a raytracer: the obvious choice is for it to evaluate the colour (or related attributes) of a point on a surface, perhaps using a custom BRDF and sending out sample rays to approximate the integral. Indeed, this could even be done in a Reyes or Z-buffer rasterizing renderer — there's nothing to stop someone from implementating the required raytracing inside their shaders (and in fact, it's a built-in function for the RenderMan Shading Language designed for Reyes).

However, this isn't an ideal solution. Figuring out the transport of light between surfaces is more the job of the renderer itself rather than the shader writer, and it can be pretty difficult to get high performance (or an easily tuned trade-off between performance and image quality) if the shaders are doing their own sampling outside of the control of the renderer as a whole. If each evaluation of a shader causes a number of rays to be sent out at random directions, there's also a real problem in terms of memory locality: each ray may well be tested against quite different parts of the scene, leading to a lot of memory traffic. A high performance raytracer instead needs to bundle together similar rays, that is ones that are likely to hit the same geometry, so they can be traced in parallel or at least with shared memory accesses. Putting the sampling code in the shader itself makes that trickier to manage.

The Open Shading Language [GSKC10] was designed for physically-oriented and high performance raytracing, and takes a different tactic with shading. An OSL shader is supposed to provide just the BRDF, leaving it to the renderer to figure out the sampling, rebundle similar rays into groups, etc. Also one of the potential benefits here of using a specialized shading language is that the renderer can have an easier time analyzing the code, say to figure out how to efficiently sample the different angles for the BRDF — e.g. if there is a Dirac delta like a mirror in there, that direction almost certainly should be sampled, while for a diffuse BRDF more uniform random sampling (or sampling directed according to the incoming light) could work the best.

6 Shader Aliasing

Another way of thinking about shaders is that they generalize texture maps. Texture maps store an image which is to be wrapped around a surface, i.e. when figuring out the colour at a point on the surface the renderer looks up the corresponding point in the texture image. A shader generalizes this to executing almost aribtrary code to figure that colour out instead of just looking it up in a table (image).

One of the big issues that has to be dealt with in using textures is *aliasing* in minification. If there is a pattern in the texture image that changes "faster" than the sampling rate, or in other words there are details in the texture image in between the points where you look up for rendering, bad artifacts are likely to appear in the final image. These aliasing artifacts typically appear as structured patterns in the image unrelated to the actual content — the small-scale details of the texture alias to large-scale errors.

The classic solution to aliasing is the use of filters that average over the small-scale details to produce a smoother, blurred texture image which can be sampled at the desired rate. The small-scale details are filtered out and thus can't alias. With texture images it's easy to perform filtering at differ-

ent levels to efficiently handle all of this.

Exactly the same problem can arise in programmed shaders. However, since the results of the shader are computed on the fly, not stored in a finite resolution image, it's a lot trickier to automatically filter them. Some rendering systems make it the job of the shader writer to handle the filtering explicitly in their code; others try to automatically adjust the renderer's supersampling rate to deal with it. In both cases, however, there has to be a notion of the rate of change of the shader's output in image space — e.g. if it's changing at a scale smaller than a pixel. This boils down to needing to know derivatives of the shader output.

Any decent shading language has support for derivatives. For Reyes, a common procedure is to use finite difference approximations across the shaded grid to provide the derivatives of the input variables relative to the shading rate — the shader writer can then use this along with the chain rule for their own expressions to figure out how to appropriate filter or blur (even in a very approximate sense) their output. In OpenGL, the shading language provides a construct to take the derivative of any variable in a fragment shader with respect to the pixel coordinates, and again that can be used by the shader writer to do their own filtering. The Open Shading Language for raytracers uses "Automatic Differentiation" (as mentioned previously for computing Jacobians for solving optimization problems) to produce exact derivatives from the source code itself.