7 Shading Tricks

Writing advanced shaders is a major effort — even a job description — so we can only scratch the surface of the craft here. We've already seen one aspect, the part of a surface shader corresponding to a BRDF, for which many theoretical and even directly measured models are available. You're already familiar with the notion of using texture maps to provide spatial variation on a surface, which can obviously be generalized in the shader context: any sort of texture map can provide input for any parameter in a shader. In this section we'll look at a few more common techniques in use.

7.1 Environment Maps for Mirrors

Mirror surfaces reflect the scene around them, which is inherently difficult for Reyes or Z-buffer rasterization — these algorithms at their core only render one primitive at a time, and don't necessarily even have the rest of the scene available when shading. Even for raytracers, mirror reflections pose a real problem if the surrounding scene is not computer generated but instead is supposed to come from real footage: the raytracer can't trace rays into the real world! The technique of *environment mapping* [BN76] luckily can solve this issue in many scenarios.

The underlying assumption for environment maps is that the mirror object be small relative to the distance between it and the rest of the scene it's reflecting. This of course might not be true, in which case the mathematical validity is in question — but the method might still look plausible, particularly if the object is curved and the reflections are expected to look distorted and hard-to-understand for the viewer anyhow. This assumption means that the rest of the scene looks almost the same from any point on the object — there isn't much "parallax".

We store a single panaromic image of the rest of the scene, taken from a single point in or near the object: this is recording what the incoming light

in every direction is for that point. We call this image the *environment map*. As a matter of implementation, this image can be stored a number of ways — underlying it as a mapping of the sphere of directions around the point to 2D, which could be done by parameterizing the image with latitude and longitude, or projecting six parts of the sphere onto the faces of a cube (a so-called "cube map"), for example.

With the assumption, this environment map should be a good approximation to the incoming lighting at any point on the mirror object, so we can just reuse it for all of them. The mirror shader than simply takes the incoming view direction, reflects it across the surface normal, converts that reflection direction to a look-up in the environment map, and returns the colour found there. The code is almost trivial.

Making the environment map in the first place is a bit more interesting. If the surrounding scene is CG, then it can be directly rendered of course (especially with the cube map representation: we just need to render the six faces of the cube with six different perspective cameras). If the surrounding scene is the real world set, a panaromic photograph has to be made. There are several techniques available for this: multiple photographs pointing in different directions can be stitched together, or a fish-eye lens used to capture a large part of the surroundings (though not all of it). The cheapest method of all is to get a mirrored ball, such as a Christmas tree ornament, and take a zoomed-in photo of that from some distance away — virtually every reflected direction is captured in the mirrored ball. Stitching together two such photos from different standpoints can eliminate the appearance of the camera in the reflections.

7.2 Image-Based Lighting

The idea behind environment mapping is incredibly powerful: "bake" the surrounding illumination into an image, so a simple look-up suffices to evaluate it. Baking complex calculations or real world data into images for

fast re-use in shaders is used all the time. A great example, generalizing the mirror shader with environment maps, is image-based lighting. The environment map is a good approximation of all the incoming light for a point on the surface. If the surface is not a mirror, but has some more complex BRDF (or is even just a plain Lambertian surface), that lighting data is still useful.

Classic rendering uses synthetic point lights, directional lights, etc. but using these to accurately model the real lighting present in a real scene, or a complex CG scene, is pretty cumbersome. If the real lighting needs a thousand point lights for a good enough approximation, it's also going to be fairly inefficient to render — naïvely there would have to be a loop over all lights for every point being shaded. Image-based lighting can solve these issues.

Where a mirror BRDF works to select a single incoming light direction to determine the outgoing light, which amounts to a single look-up into the environment map, a more general BRDF performs a weighted integral over many light directions — in the case of diffuse surfaces, all light directions in the hemisphere above the surface point. We can accurately approximate the integral with a weighted sum over the appropriate pixels in the environment map.

Of course, if the environment map is high resolution, doing this large sum every time the shader is invoked is going to be ruinously slow. However, it's perfectly reasonable to speed this up with other methods. For example, a mipmap can provide a low-resolution version appropriate for diffuse lighting (where basically the average or sum of light over a large area is required, not details on how it varies from pixel to pixel), or the environment map can be prefiltered to match the required BRDF so a simple look-up is again all that's required.

7.3 Ambient Occlusion

We didn't talk about this in class, and it won't appear in the final exam or assignments, but it's so remarkably useful and popular that I would be remiss not to mention it.

Optical effects can loosely be divided into *direct illumination*, how incoming light reflects off a surface straight to the viewer, and *global illumination*, the more complex process of light interreflecting between several objects before arriving at the viewer. Most of what we've looked at so far is really in the realm of direct illumination, but certain global illumination effects are perceptually important if a little subtle.

Monte Carlo raytracing can be easily extended to handle global illumination via recursive rays — to evaluate the incoming light for the BRDF integral, more rays can be sent out into the scene, which themselves may hit objects and spawn further rays. However, Reyes and rasterizing renderers may not even have the full scene in memory at any time since they render each primitive separately; though of course a raytracer can be fitted into the shaders, it's more natural to seek approximations that more efficiently fit into the Reyes or rasterization mindset.

The simplest global illumination approximation you've already seen is the notion of "ambient light". If only direct illumination is calculated, any surface without a clear line-of-sight to a light source will be completely black, in total darkness. In virtually every real scenario, however, there will be *indirect* illumination: light from a light source partially reflects off every object it can directly reach (weighted by the BRDFs), and that reflected light in turn illuminates more objects, and further reflections continue to reach just about every visible point in the scene. Reflections off a diffuse surface go in all directions equally, so after a bounce or two, this light can be approximated as coming from all directions. This is the basic ambient light model: the artist sets a level for the ambient light in a scene, and every object is illuminated with at least that much even if it's shadowed from all other light

sources.

However, this ambient light model is too crude to be very realistic. A lot more of the interreflected light will reach a flat surface wide open to the rest of the scene than a narrow and deep crack. Looking around you should be able to see that the edges and especially corners of a room are generally a little darker, since a little less light of the interreflected light can reach them — there are less incoming directions for a corner than the middle of a wall. This variation in available light is subtle, and usually quite smooth, but is extremely important in establishing a realistic look.

The technique of *ambient occlusion* (AO) is a convincing approximation for this aspect of global illumination [Lan02]. In a pre-rendering stage an "ambient occlusion map" is calculated for each object in the scene. The AO value at a point on a surface is a fraction between 0 and 1 indicating (approximately) how much ambient light can reach that point. In other words, if you looked at the hemisphere of directions around the point, the AO value is the fraction of the hemisphere that isn't occluded by other geometry — where incoming light could come from. This can be estimated with a simplified raytracer (firing a random selection of rays from each mesh vertex, say) or with faster approximations appropriate for hardware rasterization (e.g. [BJ07]). In the actual shader call during rendering, the ambient light is scaled by the AO fraction at the given surface point, using just a quick look-up into an AO map or interpolation from vertex AO data.

7.4 Shadow Maps

Shadows are almost trivial to accurately capture with raytracing — the renderer can simply trace a shadow ray to a light to determine whether or not something blocks the light from reaching a given surface point. As always, this raytracing could be built into a shader for a Reyes or rasterization renderer as well. However, efficiently capturing shadows without raytracing remains one of the central challenges in real-time rasterization rendering

and is similarly difficult for Reyes.

Several solutions exist, with varying tradeoffs between complexity, performance, and image quality. We will focus on probably the simplest and most popular, *shadow maps* [Wil78].

Just as rasterization and raycasting can be seen as duals of each other based on changing the order of two nested loops (loop over pixels, loop over geometry that overlaps the pixels), shadow rays and shadow maps can be seen as duals with a similar switch in when and where shadow information is computed, but one extra approximation for the shadow maps.

In a raytracer, when shading a point on a surface with respect to some light, a shadow ray is cast towards the light to see if any opqaue object is closer to the light along that path: if so the light is blocked and the point is in shadow, and if not the light is shining on the surface so the lighting calculation can add to the final colour. The fundamental test here is if another object is closer to the light along the ray. Turning this around to the light's point of view (literally!), the light is going to illuminate the closest surfaces it can see in any direction, and leave the rest of the scene in shadow. The idea behind shadow mapping is to actually render the scene from the perspective of the light (using a perspective projection for a point light or an orthogonal projection for a directional light), recording the closest surface for every ray — in other words, saving a depth map, i.e. the Z-buffer from the render. In the primary rendering pass, a shader can transform the point into the camera's projection/modelview space and perform a look-up into the shadow map to see if it's the closest surface (lit) or further away (in shadow).

There are some difficulties with shadow maps. When a raytracer casts a shadow ray, it usually has to start the ray a tiny distance off the surface in case rounding error causes an erroneous intersection with the surface itself. For shadow maps, rounding error is further compounded with a much more significant interpolation error from the discrete Z-buffer samples in the shadow map (it's likely the point being shaded doesn't coincide

exactly with a shadow map sample, but instead falls between them and so the depth from light must be interpolated). For this reason, a *bias* is added to the comparison between the surface point and the shadow map, to avoid surfaces shadowing themselves (an error which causes sporadic black spots to appear on surfaces, hence the name "surface acne"). Of course, if the bias is too large, shadows will be incorrectly computed, and regions which should be in shadow will still be lit.

Used as is, shadow maps also may cause aliasing artifacts. The decision of lit vs. in shadow is strictly Boolean: is the interpolated shadow map depth less than or greater than the point's computed light depth plus bias? If the shadow map is low resolution, with a single shadow map "texel" applying to many pixels in the final image, the finite resolution of the depth map will be quite apparent. Going to very high resolution, so this simply doesn't occur, is one possible expensive solution; another is to supersample the depth map with nearby points to the point being shaded and taking the average of the shadowing results to generate a fraction between zero and one, so-called *percentage-closest filtering* [RSC87].

Many other extensions to shadow maps, as well as quite different algorithms such as "shadow volumes", have been proposed. The recent book by Eisemann et al. is a great resource for further reference [ESAW11].

7.5 Noise

Finally, any discussion of shaders must mention the concept of *noise*, an enormously important type of mathematical function used as a primitive building block for many complex shaders. Introduced by Ken Perlin [Per85b], noise provides a way to incorporate random variation into functions in a controlled, predictable way.

A classic example is something like the varying colour of a sawn block of wood. The pattern of the woodgrain has a lot of structure — smooth contours with a gradation from light to dark — but isn't perfectly regular. Ev-

ery tree grows a little differently, has knots at unpredictable locations, and so forth, so building a periodic pattern with sine curves or the like would look wrong. Humans are exquisitely talented at seeing repetitive patterns and symmetries, and when they crop up in places they shouldn't it destroys the suspension of disbelief. However, setting just a random colour at every point on the surface obviously wouldn't respect the expected structure at all, or even be smooth.

Noise provides a smoothly varying function with an element of randomness. A good noise function will have a predictable amplitude, say between -1 and 1, and a predictable scale over which is varies, say oscillating once between two consecutive integers on average. However, unlike a sine curve, it shouldn't be periodic or have any other humanly discernible pattern — and yet it still should be well-defined and repeatable (namely, if you evaluate the noise function twice with the same arguments you should get back exactly the same value).

There are many possible ways to do this, varying in performance and quality. Most of them use some variant of the following:

- Define the noise function as a Hermite spline with integer knots.
- Set the value of the function at knots to zero.
- Set the derivative of the function at a knot to a pseudo-randomly chosen value, e.g. with a hash function on the integer knot to index into a fixed array of preset values.

The details of exactly what spline and what hash function is used vary considerably. The critical thing to do well is the hash function, as it's responsible for injecting the apparent random variation of the noise function.

Noise functions can be built in any number of dimensions. We haven't discussed multi-variable splines yet, though they're tremendously important for geometric modeling, but really the only crucial conceptual extension is

that the "derivative" values in a multi-variable Hermite spline are going to at least be gradient vectors.

Once you have a noise function, it can be used in a plethora of ways to build complex shaders. At the very simplest, any regular mathematical expression based on sine or cosine which has the right structure, but suffers because it's exactly periodic, can be fixed by replacing sine with noise, or adding noise to the argument of sine, etc.

References

[BJ07] Jared Boberock and Yuntao Jia. High-quality ambient occlusion. In *GPU Gems 3 / Chapter 12*, pages 239–274. Addison-Wesley, 2007.

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19:542–547, October 1976.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21:95–102, August 1987.

[Coo84] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18:223–231, January 1984.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18:137–145, January 1984.

[ESAW11] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. *Real-Time Shadows*. A.K. Peters, 2011.

[GSKC10] Larry Gritz, Clifford Stein, Chris Kulla, and Alejandro Conty. Open shading language. In *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, pages 33:1–33:1, 2010.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *SIGGRAPH Comput. Graph.*, 24:289–298, September 1990.

[Lan02] H. Landis. Production-ready global illumination. In *SIGGRAPH Course Notes 16*, SIGGRAPH 2002, 2002.

[Per85a] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19:287–296, July 1985.

[Per85b] Ken Perlin. An image synthesizer. In *Proc. SIGGRAPH*, pages 287–296. ACM, 1985.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph.*, 21:283–291, August 1987.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12:270–274, August 1978.

[ZHR+09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. *ACM Trans. Graph.*, 28:155:1–155:11, December 2009.