Rendering and Shading

Robert Bridson

November 11, 2011

While rendering is often treated as a separate topic from animation, it is still an integral part of the process, and the special requirements of animation (and film visual effects in particular) do guide certain choices on the rendering side, and vice versa. We'll first look at a few basic requirements on rendering algorithms, before the commonly used methods themselves, and then focus on the process of shading which tends to dominate both the computer and user time spent on rendering.

1 Supersampling

We will focus on film visual effects, as it arguably sets the bar highest for rendering sophistication: if the computer generated imagery is to seamlessly blend into real footage, it has to be photorealistic. As we said before, the key insight is to set up rendering as a simulation of a real camera's action and the physics of light in the scene. Of course, we may frequently want to permit unphysical things to happen for artistic reasons, such as eliminating inconvenient shadows, in the same vein as real movie sets modify reality with the addition of special lights and props. This should be an artistic option, though, not an unavoidable restriction.

Review the action of a real camera. When a frame of film is recorded, the *shutter* is opened and light from the scene shines into the camera body through the *aperture*, is refracted through the *lens* system, and hits the image sensor (or initially unexposed film). The sensor is divided into pixels roughly speaking (or, if it's actual film will be scanned into pixels later), i.e. its area is split into a grid of sub-areas. Each sub-area accumulates a measure of how much light strikes it, until the shutter closes again and the frame is finished. The total amount of light recorded over the whole area of a pixel for the whole duration of the time that the shutter is open is what is saved as the value of that pixel in the final image.

A pixel in a real camera's image is clearly not just the colour of a single light ray hitting the centre of the pixel at a single instant in time. It is rather an integral over the light rays that hit anywhere in the pixel area, go through any point in the camera aperture, and at all times while the shutter is open.[1] The effects of integrating over these three parameters (where on the pixel, where in the aperture, when in time) are quite visible:

- Integrating over the entire area of the pixel is a form of *antialiasing*, softening out the jagged edges between distinct regions of the image (e.g. where both a black region and a white background overlap a pixel, a proportional shade of grey will be measured) and reducing Moiré patterns and other artifacts seen in simple point samples of rapidly varying images. In a real digital camera, this softening may be enhanced for better quality (less grainy) images by the addition of a translucent diffuser in front of the colour filter / sensor package. In rendering, the antialiasing properties can be similarly enhanced by changing the integral over a pixel to use a smooth weighting kernel spread out over the pixel and its near neighbours.
- Integrating over the aperture causes *depth-of-field* defocus. For a point in the scene which is in focus, all light paths through the aperture end up converging on the same point on the sensor. For a point in the scene that's too close or too far from the camera, however, the light rays extending from it to different points on the aperture end up hitting the sensor at different places, blurring the light over an area.[2]
- Integrating over the shutter time causes *motion blur*. If something moves fast enough, it will send light to a whole path of different pixels on the sensor.

[1]Going into the quantum physics level, it's perhaps better modeled as a sum of distinct photons that make it through the aperture to somewhere on the pixel while the shutter is open, but we generally will happily stick with a continuous integral to simplify life.

[2]The precise area over which it is blurred is in fact the same shape as the aperture, but scaled to a size depending on how far out of focus it is. If you've ever seen a tiny but bright point of light out-of-focus in an image, you'll notice it appears as a disc or polygon—that's actually the shape of the camera aperture you're seeing.

To seamlessly integrate computer generated images into real footage with all these effects, the renderer has to be able to approximate them. In some cases they can be adequately approximated with image operations (special blurs), but especially for antialiasing this is often not good enough.

The best general purpose method we have to approximate these integrals is *supersampling*. For example, let's take a really simple example of an integral where we know the exact answer:

$$\int_0^1 x^2\,dx = \frac{1}{3}. \tag{1}$$

If we couldn't find the antiderivative $\frac{1}{3}x^3$ for some reason, we could approximate the answer by sampling the integrand at several points $\{x_1, \ldots, x_n\}$ in the interval $[0, 1]$, and taking their average:

$$\int_0^1 x^2\,dx \approx \frac{x_1^2 + x_2^2 + \cdots + x_n^2}{n} \tag{2}$$

As long as those samples are uniformly distributed (not biased towards one endpoint, say), it can be shown that our estimates will converge to the correct answer the more samples we take. Here's a table with some randomly chosen samples demonstrating the convergence:

samples	estimate
.221	.0488
.221, .969	.4939
.221, .969, .014	.32933
.221, .969, .014, .525	.31591
.221, .969, .014, .525, .378	.28130
.221, .969, .014, .525, .378, .758	.33018

The use of random samples gives this method the name *Monte Carlo*. Uniformly spaced samples would converge faster in this example, but is not as attractive for rendering since the remaining errors in the estimate might correlate across pixels in visible structured patterns, a bad artifact.[3]

[3]Even greater convergence is possible with careful *numerical quadrature* algorithms,

There are many techniques to improve the convergence of Monte Carlo without introducing artifacts. The most common example is using stratified or *jittered* patterns. For our one dimensional integral, this would amount to dividing up the interval $[0,1]$ into equal subintervals, and picking one random sample from each subinterval. This guarantees the samples are well-spaced across the domain, but are still quite random. Running the same experiment with jittered samples gives:

jittered samples	estimate
.365	.13322
.157, .733	.28097
.285, .421, .784	.29104
.229, .403, .547, .874	.31948

Going back to the rendering problem, for each pixel we can take multiple samples, varying the precise position within the pixel (perhaps using a jittered configuration from a uniform subgrid), the position on the aperture the light ray should go through, and the time the sample is taken during the shutter open period. The average colour over all these samples[4] gives the final estimate for the image. Cook et al. [CPC84] introduced this idea in the context of ray tracing (and extended it to calculate some other important effects which we will discuss later in this chapter) but at the camera sampling level it can equally apply to other rendering algorithms.

which place the samples and weight their contributions in certain optimal ways — but again these aren't as useful in rendering where the remaining errors can show up as faint but distracting patterns in the final image.

[4]Again, for high quality results it's best to use a weighted average with weights taken from a smooth kernel function like a Gaussian spread out over a small neighbourhood of pixels.

2 Performance

The scenes which are rendered, for film especially, have grown in detail and scope as fast or faster than computing power has grown over the past three decade. Rendering software has to be able to efficiently process (at the present) dozens of gigabytes of scene description for a single frame. At one time almost all the data was in the form of texture maps applied to relatively simple geometric primitives. Texture maps continue to be huge but the raw geometry has now, in some cases, also grown formidably large — scanners and physical simulation both are used to generate meshes or point clouds with hundreds of millions or more vertices, today.

Depth complexity, i.e. how many distinct surfaces project to different depths at the same pixel on average, is also quite relevant. Earlier there were strong arguments as to why depth complexity remains small even as overall scene complexity increases. For example, in the office where I am writing this, there are many richly detailed surfaces visible, but they don't overlap each other significantly if I don't go beyond the walls of the room. The rest of the building is invisible from my current viewpoint, so in a synthetic render of my office I wouldn't even bother modeling it — or it could be automatically ignored via "occlusion culling". However, there are now also common cases where deep depth complexity is unavoidable: outdoors in a forest or tall grass or a crowd, for example. Gracefully handling depth complexity is becoming important in some (but not all) scenarios.

Meanwhile, hardware has steadily moved in a trend where two aspects dominate performance: parallelism and data movement.

High-performance hardware exposes parallelism at several levels, such as vectorization (a type of SIMD / Same Instruction Multiple Data execution, where the same operation is applied to a vector of four numbers instead of one at a time) or multithreading (a type of MIMD / Multiple lnstructions Multiple Data, where completely independent threads of execution operate simultaneously). Algorithms must exploit this to see real gains from the hardware.

The speed at which processors can crunch numbers now is vastly faster than the speed at which data can be fetched from or written to main memory, or even transferred between processors. Optimizing data movement (e.g. fetching the next block of data at the same time as processing the current block) and maximizing the amount of useful work that can be done relative to the number of accesses is crucial. Algorithms which randomly skip through memory pay a terrible price; "data locality" (working on data stored nearby in memory) must be considered for any data structure.

3 Rendering Algorithms

There are basically three major rendering algorithms at work today.[5] You should already be familiar with Z-buffer rasterization as exemplified by APIs such as OpenGL, and raytracing; the other big algorithm is *Reyes*, used primarily in the film industry. It's important to understand the differences between these, particularly in terms of *shading*: what parts of the rendering algorithm are programmable by the user?

Let's first review Z-buffer rasterization. The rendering pipeline consists essentially of the following steps (ignoring some that aren't relevant to our discussion, such as blending and or enhancing performance through tiling the image):

1 A single rendering primitive (usually triangle, perhaps point or line or quad) is specified with vertex data (typically position in 3D, perhaps also normals, texture coordinates, colours etc.).

2 Each vertex is processed independently to be ready for the image. Traditionally this would involve applying model-view and projection

[5] I ignore certain specialized scenarios such as volume rendering for scientific and medical visualization: I'm only focusing here on general-purpose methods for primarily rendering surfaces in an animation context.

transformations of the position to normalized device coordinates, similar processing of the normals, and related lighting calculations. More generally this is handled by a *vertex shader* (see below).

3 The primitive is clipped against the bounds of the view frustum, eliminating problems with triangles that touch or cross the camera plane (projected to infinity by perspective) and also saving subsequent steps from working with off-screen geometry.

4 The clipped primitive is rasterized to *fragments* (essentially sample points, usually one per pixel) where processed vertex data is interpolated. Traditionally calculations such as a texture look-up happen here; more generally a *fragment shader* (see below) calculates the fragment colour and depth (or other outputs).

5 The depth test is applied to each fragment vis a vis the Z-buffer, updating the final image values.

More recent hardware has augmented this with an additional step after the vertex shader, "geometry shading", which can generate new primitives — for example, replacing a large triangle with an intricately deformed high resolution surface. This is a big step forwards in capability, but not as widely used yet: compare it to the dicing/shading stages of Reyes below.

Z-buffer rasterization excels for scenes where the primitives are large (map to many fragments) and most of the complexity comes from textures: the rasterization has very low overhead and easily parallelizes, the texture accesses are nicely coherent. Even when the primitives are smaller, there is plenty of parallelism to be had between primitives (albeit needing careful handling of parallel access to the final image and Z-buffer). Since primitives are handled independently, they can be streamed through the renderer without requiring the entire sene be present in memory at the same time. However, performance suffers when a scene has high depth complexity: a lot of shading work is done for vertices and fragments which don't show up in the final image, since shading occurs before the depth test. If geometric complexity grows to the point where primitives are much smaller than a pixel, there's also a waste in processing and shading vertices which don't contribute to fragments.

The Reyes algorithm by Cook et al. [CCC87] is similar in that it rasterizes one primitive at a time, in essence, but has a few notable differences. It proceeds as follows, again skipping over some of the implementation details:

1 A single rendering primitive is specified at a relatively higher level (e.g. a cone), subject to transformations.

2 If the bounding box of a primitive is too large, it is *split* into smaller primitives which correspond to a quadrilateral in parameter space, and the rendering proceeds with them. If a primitive's bounding box lies outside the view frustum, it is culled — there is no clipping to the frustum in Reyes.

3 Each (sub-)primitive is *diced* into a regular grid of *micropolygons* along the natural parameterization axes, such that each micropolygon is a quad roughly half the size of a pixel (or some other user-specified fraction of a pixel). Note this is not a rasterization; the quads are not necessarily aligned with pixels, just of comparable or smaller size.

4 Each grid is shaded: a surface shader runs at each grid vertex to determine its colour. The shader here may even change the position, though within the limits of the bounding box (ipso facto, the bounding box provided for a primitive has to take into account how far vertices may change their position when shaded).

5 The shaded micropolygons are rasterized against samples, with a Z-buffer to determine how to combine them. Typically in Reyes many samples are taken per pixel, possibly jittered in time and aperture position too, to provide antialiasing etc. If motion blur is in play, each quad needs to know both start and end positions during the shutter time so it can interpolate its position to the sample time if need be.

6 Samples are combined with a weighted average into pixel values.

The major algorithmic differences compared to OpenGL are that Reyes doesn't need clipping, and that (most of) the shading is all done at one stage before rasterization, decoupled from both the geometry resolution and the final image sampling rate. Reyes also has had a focus on quality from the start, supporting various curved primitives as well as supersampling to achieve antialiasing, motion-blur and depth-of-field; OpenGL is possibly evolving in a direction of supporting these (there's no fundamental reason other than available hardware performance that it can't) but is not there yet.[6] However, these aren't enormous differences — for example, several authors have worked out high performance implementations of Reyes on GPUs too, e.g. [ZHR$^+$09]. In particular, essentially the same performance comments apply equally to Reyes with the exception that the Reye decoupling of shading rate from geometry resolution (OpenGL vertex shaders) and image sampling rate (OpenGL fragment shaders) means performance can be tuned a bit better. In most film shots, shading accounts for the vast majority of render time.

Raytracing reorders the rendering process significantly, and in so doing supports extensions to capture just about all optical effects. A basic raytracer works as follows:

1 The entire scene description is loaded into a ray-query structure of some sort, such as a KD-tree, to permit fast ray tests.

2 For each image sample a primary ray is generated and traced into the scene: the colour of the sample is determined by which geometry it hits and the execution of shaders at those intersections. For opaque surfaces, only the frontmost intersection needs to be found and shaded.

[6]Full Screen Anti-Aliasing (FSAA) has been around a while, using a structured form of supersampling; matching the motion-blur and depth-of-field capabilities of Reyes via supersampling is still a way off.

3 Samples are combined with a weighted average into pixel values.

In terms of quality/flexibility the one difficult point about ray tracing compared to the other algorithms is displacement shading: a ray intersection can't simply be moved during shading as micropolygon vertices can in Reyes. Displacements must be computed before the intersection tests — scheduling that component of shading (at an appropriate resolution) in an efficient way isn't trivial. Likewise decoupling the shading rate from the sampling rate isn't obvious — however, it is naturally decoupled from the geometry resolution, even more so than Reyes which must shade even the smallest primitive at least into one micropolygon.

Raytracing enjoys several advantages asymptotically for scene and depth complexity. With typical acceleration structures and typical scenes, rendering cost per sample scales with the logarithm of depth complexity, not linearly (as it does with the other two algorithms). Raytracing can also guarantee that only intersections which contribute to the final image will be shaded, since the intersection tests occur before shading. When there are many primitives per pixel, or high depth complexity, raytracing has a natural advantage — we are arguably often in that scenario nowadays.[7] That said, the overhead of raytracing can be significantly greater than the other algorithms — the entire scene has to be available at once, rather than streamed through one primitive at a time, and in a naïve implementation each ray incurs a lot of memory access and arithmetic which are amortized across multiple samples/fragments/micropolygons in the other methods. To achieve high performance, nearby rays must be bundled into groups which are traced in parallel for example.

[7]Of course, more advanced algorithms are being developed to automatically and adaptively "simplify" overly complex geometry so there are never more than a few primitives per pixel, a geometric form of mipmapping called "Level of Detail" processing. Likewise, advanced culling methods can help reduce depth complexity. However, to reliably do this in general is a very hard problem, a subject of ongoing research.

4 Shading

Shading has a special meaning in the context of rendering. While it originally just referred to the code which calculated the colour of a point on a surface based on material information and lighting (hence the name "shading"), it now refers to essentially any user-customizable part of determining the appearance of elements in a scene. While the choice and implemention of the core rendering algorithm, as discussed above, is of utmost importance, the bulk of the programming effort (and computation) in typical rendering tasks lies in shading.

Shading began with relatively simple models of appearance, such as constant colour, Lambertian (a.k.a. diffuse, or matte) shading, mirror reflections, the Phong glossy model, etc. Early renderers, and even many today, provided a library of such models and allowed the user to pick which was in use, and set its parameters. The code implementing such a model would be called a "shader".

The major difficulty with this approach is when none of the provided shaders quite capture the phenomena of interest. The real world is full of materials with extremely complex view- and lighting-dependent appearance. For example, even a simple piece of white writing paper could be approximated as Lambertian: it's fairly diffuse. However, if you look at it from a glancing angle (view direction nearly parallel to the paper) it becomes much more specular; if you look at it closely the detailed texture of fibres matted together becomes apparent; if there's light on the other side the translucency of thin paper is obvious. Properly capturing the appearance of ink or toner or pencil graphite on the paper, to appear photorealistic in a close-up shot, adds a whole other level of complexity. It's unlikely that every such scenario will be anticipated in advance by the programmers of the renderer.

Another possibility that some renderers have adopted is to provide a dynamic library, "plug-in" mechanism where users can program their own shaders to match the renderer's API with a standard language such as

C++. This opens up wide new possibilities for users, who can add support for their special requirements. However, the advantage of having the full power of a general purpose programming language can also be a disadvantage: all sorts of bugs, including crash bugs, security holes, and serious performance disasters, are possible. General purpose programming languages also may make common shading operations (such as working with vectors, transforms, textures) more difficult or verbose than desired. Such plug-ins are typically "black boxes" of compiled code, and can't be directly analyzed by the renderer in useful ways without tremendous effort in disassembly — a topic to which we'll return below. Finally, they're restricted to the regular hardware for which compilers are available: execution on a GPU is generally out of the question.

The best solution has been to instead develop a "Domain Specific Language" for shading, i.e. a *shading language*. The evolution of shading languages went through Cook's shade trees [Coo84] and Perlin's original image synthesizer [Per85], but was realized in full by Hanrahan and Lawson [HL90] for Pixar's RenderMan Reyes renderer. A shading language is a specialized programming language which only exposes aspects of computation relevant to rendering, provides convenient support for common shading operations (such as vector and matrix operations etc.), and can be analyzed and executed by the renderer as needed to compute images.

Shading languages open up enormous possibilities for rendering. Obviously they can implement mathematical expressions for new models of appearance, or combine existing expressions in novel ways, but they can do a lot more. This is particularly true when you begin to view rendering as capable of more than simply generating a standard image file of coloured pixels. For example, in concert with shaders, a renderer can save more than just colour (or not even colour): the "alpha" or transparency of each pixel, the depth of a pixel (what is stored in the Z-buffer), the ID number of the closest primitive to the camera in each pixel, the velocity of the geometry at each pixel, and many more variables are possible outputs. These in turn can be used creatively as inputs to further rendering (and shading) or image-processing passes, some of which we'll look at in more detail below. As mentioned above for vertex/geometry shaders and Reyes surface shaders, displacement shaders can even modify the geometry being rendered, e.g. turning a simple sphere into a baseball with raised sitches and other geometric details. Shaders can likewise take inputs from many sources: parameters set by a user, the position of the point being shaded, the normal and texture coordinates at that point, any number of texture map images (some of which may be abstract information derived from previous rendering passes), the set of lights in the scene. Shaders might even be given access to the full scene geometry as well: a common trick in rasterizing renderers is to write a mini-ray-tracer in the shading language to permit effects like volume rendering of smoke which are hard to do with rasterization.