RenderMan in Production

SIGGRAPH 2002 Course 16

course organizer: Larry Gritz, Exluna, Inc.

Lecturers:

Tony Apodaca, Pixar
Larry Gritz, Exluna, Inc.
Matt Pharr, Exluna, Inc.
Dan Goldman, Industrial Light + Magic
Hayden Landis, Industrial Light + Magic
Guido Quaroni, Pixar
Rob Bredow, Sony Pictures Imageworks

July 21, 2002

Lecturers

Tony Apodaca is a Senior Technical Director at Pixar Animation Studios. In the 20th century, he was Director of Graphics R&D at Pixar, was co-creator of the RenderMan Interface Specification, and lead the development of PhotoRealistic RenderMan. Tony has been at Pixar since 1986, where his film credits span *Red's Dream* through *Monsters, Inc.* He received a Scientific and Technical Academy Award from the Academy of Motion Picture Arts and Sciences for work on PhotoRealistic RenderMan, and with Larry was co-author of the book *Advanced RenderMan: Creating CGI for Motion Pictures*. He holds an MEng degree from Rensselaer Polytechnic Institute and has spoken at eight previous SIGGRAPH courses on RenderMan (as course organizer for five of them).

Rob Bredow is a CG Supervisor at Sony Pictures Imageworks currently working on *Stuart Little 2*. While at Sony, Rob has been also involved in creating many of the complex visual effects featured in both *Castaway* and the first *Stuart Little*. Rob is an acknowledged expert in the field of effects animation, shading and rendering and has presented at several conferences, including the Advanced RenderMan course at SIGGRAPH 2000. Rob's other credits include Visual Effects Supervisor on the feature film *Megiddo* and Director of R&D at VisionArt, where he developed visual effects for Independence Day, Godzilla, and Star Trek.

Dan Goldman received his Bachelors and Masters in Computer Science from Stanford University in 1994 and 1995. Since 1992, he has been at various times a software engineer, technical director, sequence supervisor, and computer graphics supervisor at Industrial Light and Magic. Most recently he was the computer graphics research and development supervisor for *Star Wars Episode II: Attack of the Clones*. He also worked on *Men In Black*, *Star Wars Episode I: The Phantom Menace*, *Galaxy Quest*, and *The Perfect Storm*. He is currently pursuing his PhD in computer graphics at the University of Washington, and continues work part-time as a software engineer for ILM.

Larry Gritz is a co-founder of Exluna, Inc. and technical lead of the Entropy renderer. Previously, Larry was the head of the rendering research group at Pixar and editor of the RenderMan Interface 3.2 Specification, as well as serving as a TD on film projects including both *Toy Story* movies. Larry has a BS from Cornell University and MS and Ph.D. degrees from the George Washington University. Larry was the original creator of BMRT, and was co-author (along with Tony) of the book *Advanced RenderMan: Creating CGI for Motion Pictures* (Morgan-Kaufmann, 1999), and has spoken at five previous SIGGRAPH courses on RenderMan (as course organizer for four of them).

Hayden Landis is a sequence supervisor at ILM. He joined ILM in 1997 as a TD working on *Men In Black*. Prior to ILM, Hayden worked at Xaos Inc. and Rocket Science, Inc. Some of his more involved projects have included *Pearl Harbor*, *Wild Wild West*, *Men In Black*, *Snake Eyes*, *Deep Impact*, and *Flubber*.

Matt Pharr is a co-founder of Exluna where he is currently working on hardware-based rendering technology. Previously at Exluna, he worked on the design and implementation of the Entropy rendering architecture and the 3ds max to Entropy bridge product. He has published three papers at SIGGRAPH on topics of light scattering algorithms and rendering of large scenes. He was a part-time member of Pixar's Rendering R&D group from 1998-2000, contributing to development of rendering software there. He has a B.S degree in Computer Science from Yale University and an M.S. from Stanford (with a PhD slowly nearing completion.)

Guido Quaroni has worked at Pixar since January 1997. At the studio he worked on *Toy Story 2* in the modeling, shading and FX department. He also worked on *Monsters Inc.* as sequence supervisor (and the voice of "Tony the Grocer"). At this time he is working in the studio tools departments focusing his attention on the shading pipeline.

Contents

Chapter 1

The Lore of the TDs

Tony Apodaca

Pixar

aaa@pixar.com

Before prospective feature film production technical directors in charge of computer graphics imagery ("CGI TDs"), such as yourselves, can start making the knockout special effects that we've come to expect from individuals of your caliber and future reputation, there are a few basics you need to know.

Okay, let's be honest. Being a superstar TD requires a lot more than extensive training, straight A's in all your math and programming courses, and memorization of all of Frank Oz's lines in *Episode 5*. In fact, being a superstar TD doesn't require **any** of those things. No, it requires being steeped in the secret knowledge, the *Lore of the TDs*. There is much that the mere 3D Modeling Artist does not know. There is much that the mere Production Supervisor cannot grasp. Only you, who are willing to devote your life to the mysterious ways of the TD can hope to one day reach the nirvana, the pinnacle of knowledge, the celebrated gurudom, that is the *Effects Supervisor*.

If you are destined to be one of the chosen ones, then perhaps this course, this 90 minute lecture, this moment of inspiration, will help you make your first step to a larger world. [1]

1.1 Basic Computer Graphics

This morning, we'll start with a basic grounding in some fundamental concepts of computer graphics. There are innumerable textbooks on computer graphics, mathematics, optics and physics which have information relevant in your travails as a production TD. In all likelihood, you have a couple of those textbooks on your shelves already. However, experience has shown us that most of those textbooks are also filled to the brim with information you won't ever need. Breshenham's circle drawing algorithm, anyone? And sometime the key bits of knowledge and lore are buried so deeply in that information overload that they get overlooked. While a short lecture such as this can't possibly cover in detail every concept that you'll have to master, I thought I would highlight the ones that I find most relevant on a day-to-day basis. If I gloss over details, or just simply confuse you, I apologize in advance. Consult the standard tomes for more depth.

1.1.1 Trigonometry and Vector Algebra

To quote one of America's most famous and enduring icons, "Math is hard." Yes, it is. Fortunately, despite appearances, TDs only rarely use math more complex than trigonometry in their day to day

[1] Oops, sorry, that was Alec Guinness in *4*.

lives. And since I assume everyone in this audience graduated high school (or at least college) with a semester of calculus under their belts, trigonometry should be a snap. Well, trigonometry and vector algebra. Okay, the occasional integral equation, but really, only occasionally....

Trig

The single most important mathematical equation that you could hope to know, that will solve the deepest mysteries of computer graphics, is this one:

$$U \cdot V \;=\; |U|\,|V|\;\cos(\theta_{U \rightarrow V})$$

It says that the dot product of two vectors is equal to the product of the magnitude of the two vectors times the cosine of the angle between them. If you completely understand this equation, you understand most of the important parts of trigonometry. So, let's take it apart.

First, there is no problem with what a vector is, right?

$$\begin{aligned} V &= V_x \cdot \bar{X} \;+\; V_y \cdot \bar{Y} \;+\; V_z \cdot \bar{Z} \\ &= V_x \cdot (1,0,0) \;+\; V_y \cdot (0,1,0) \;+\; V_z \cdot (0,0,1) \\ &= (V_x, V_y, V_z) \end{aligned}$$

A *vector* is quantity which is composed of two or more *components*. The number of components that a vector contains is called its *dimensionality*. In our notation, we use a capital letter to represent a vector, and a subscripted capital letter to represent one of the components. In our case, we are often interested in three-dimensional *direction vectors*, whose components represent distances along the $\bar{X}$, $\bar{Y}$ and $\bar{Z}$ axes. A quantity with only one component, a normal real number, is often called a *scalar* by vector algebraists, because multiplying a direction vector by a scalar lengthens or "scales" the vector. Adding vectors is done by adding the components. Real simple.

Second, do we remember what the dot product of two vectors is? Sure we do. It is the sum of the products of the elements.

$$U \cdot V \;=\; (U_x \cdot V_x \;+\; U_y \cdot V_y \;+\; U_z \cdot V_z)$$

There are two different ways to multiply two vectors together. Multiplying the two vectors componentwise, and summing the results, gives us a scalar which is the *dot product*. This is the simplest way you can multiply two vectors, since you just have to multiply the elements and add. The only tricky bit is that the result is a scalar. The other way to multiply two vectors is called the *cross product*, and that has its uses which we'll talk about later.

Next, what do the vertical bars mean? Magnitude. That is, the length of the vectors. Any time a vector is written between two bars like this $|V|$, this refers the the length of the vector, which, thanks to Pythagoras, we know is $\sqrt{{V_x}^2 \;+\; {V_y}^2 \;+\; {V_z}^2}$. A vector whose length is exactly equal to 1.0 (aka *unit length*) is called a *normalized* vector (not to be confused with a *normal vector*), and is often written with a bar over it like so: $\bar{V}$.

Finally, we know what the cosine of the angle between two vectors means. But what if the vectors are located far apart from each other? What if one is over here, and the other is way over there. How do you measure the angle between them? Well, the answer is, there is no such thing as "way over there" for direction vectors. Direction vectors are only directions, their "positions" don't matter, or even exist! You can move their starting point to whereever you want without changing them. So, to measure the angle between two vectors, you just think of them starting from the same point, and measure that angle. Capish? If the vectors point in the same direction, the angle between them is 0.0. The cosine of 0.0 is 1.0. If the vectors point in opposite directions, the angle is 180 degrees, also known as π radians, and the cosine of that is -1.0. If the vectors are perpendicular to each other, the angle is 90 degrees, $\frac{\pi}{2}$ radians, and the cosine of that angle is 0.0.

So, putting it all together. The dot product of two vectors is a scalar. It is proportional to the length of the vectors, and to the cosine of the angle between them. And because we can rearrange the master equation into this:

$$\theta_{U \to V} \;=\; \cos^{-1} \frac{U \cdot V}{|U|\,|V|}$$

we now have the single most common way of computing angles. **That** is why the dot product so important to computer graphics.

Homogeneous Coordinates

In your work as TDs, you will spend most of your professional career worrying about three very similar, yet totally distinct, types of 3-dimensional vectors. If you understand how and why they are different from each other, you'll be way ahead of your less erudite peers.

Points are vectors, and in our case they are 3-dimensional vectors. They represent positions in 3-D. *Direction vectors* are also 3-dimensional vectors. Generally if someone says casually that some quantity is a "vector", they mean a direction vector. We can measure both the "direction" and the "magnitude" of a direction vector, and it is often the case that we only care about one or the other in the equations that we are juggling. Points and direction vectors are related by a simple and elegant fact: the values of the components of a point are identical to the values of the direction vector that points from the origin to the point.

$$(P_x, P_y, P_z) \;=\; (0,0,0) \;+\; (V_x, V_y, V_z)$$
$$\text{where } P_x \;=\; V_x, \text{ etc.}$$

A long time ago, eons ago, back in ancient days before Nathan Vegdahl was born, a group of computer graphics practitioners noticed that a lot of the math that had to be done with points and vectors, like transforming them from one coordinate system to another, was easier if the points and vectors were represented in 4-dimensions. No, they were not adding *time* to the mix. Instead, they were appealing to a particularly elegant 4-dimensional space known as *homogeneous coordinates*. Homogeneous coordinates are elegant because it handles perspective projection with the same math that it handles normal translation or rotation. Homogeneous math doesn't get hung up on the problem that perspective projections make parallel lines intersect, which totally screws up our nice clean Euclidean 3-dimensional space.

Now, *here* is the magic. To put a point into 4-dimensional homogeneous coordinates, you merely tack on a 1.0 as the fourth (usually called w) component. To put a direction vector into homogeneous coordinates, you merely tack on, stay with me here, a 0.0 as the w component. Got that? So now, the pedestrian equation above becomes:

$$(P_x, P_y, P_z, 1.0) \;=\; (0,0,0,1) + (V_x, V_y, V_z, 0.0)$$

Et voila!

Suddenly, much that was mysterious comes into focus. Why are transformations always specified as 4×4 matrices instead of 3×3, or perhaps a 3×4? Because they are specifying homogeneous transformations. Consider the math involved with 3-dimensional transformation matrices. In order to transform a point by a 3×4 rotation and translation matrix as you'd find in a CG textbook, you have to know to multiply the point by the "upper 3×3", and then add in the translation from the bottom row. If you transform a direction vector, you don't bother to add the translation. When you put a point through a perspective projection, you calculate the x and y components, and then "divide by z". You simply can't put a direction vector through perspective. Instead, you must transform two points and then subtract them. So many special rules to remember!

With a homogeneous transformation matrix, all these little tricks go away. Points and vectors transform identically through the transformation matrix. The 0.0 or 1.0 in the w component handle the "translation problem". Perspective? No problem, just go for it.

You've probably heard little rules of thumb like "You can add two vectors, but you can't add two points" or "You can subtract two points, but you get a vector". But does it make sense to multiply a point by a scalar? Such questions are a lot easier to answer in homogeneous coordinates. Oh, you need to know one other thing: by definition, $(x, y, z, t) == (\frac{x}{t}, \frac{y}{t}, \frac{z}{t}, 1)$

$$
\begin{array}{lcll}
(x,y,z,0)+(r,s,t,0) & = & (x+r,y+s,z+t,0) & \text{A vector!} \\
(x,y,z,1)+(r,s,t,1) & = & (x+r,y+s,z+t,2) & \\
 & = & (\frac{x+r}{2},\frac{y+s}{2},\frac{z+t}{2},1) & \text{The midpoint?!} \\
(x,y,z,1)-(r,s,t,1) & = & (x-r,y-s,z-t,0) & \text{A vector!}
\end{array}
$$

Normal Vectors? Hardly!

The oddest of our triumverate of 3-vectors is the *normal vector*. Normal vectors are not called normal because they are straightforward, because they are not. They are not called normal because they are normalized (unit length) because they don't have to be. They are called normal because "normal" is a synonym for "perpendicular". Normal vectors are perpendicular to a surface. That is, they are perpendicular to the tangent plane at the surface.

What is a tangent plane, you might rightfully ask. Believe me, you probably understand what a normal vector is better than you'll understand my goofy explanation of what the tangent plane is. The idea of a tangent line is easier — it is a line that just touches the surface at the point, but doesn't penetrate the surface (at least, not near the point). The tangent plane is a plane that does the same. It represents the orientation of the surface. So, the normal vector in some sense points as far away from the surface as is possible.

How are normal vectors computed? For most primitives, it is relatively easy to compute a couple vectors in the tangent plane, because the tangents are related to the derivatives of the equations of the surface. Once you have two tangents, you apply the old *cross product* trick.

Remember we said there was another way to multiply two vectors? Here it is. The cross product is harder to compute than the dot product, but not that difficult once you learn the trick. The point is that it produces a vector, not a scalar like dot product. In fact, the vector that it produces is perpendicular to both of the original vectors. This is a very cool property, because generating an axis vector that is perpendicular to something is a very useful thing to be able to do. For example, consider when I have two vectors, both in the tangent plane. The surface normal is perpendicular to the tangent plane, and so by definition it is perpendicular to every vector in the tangent plane. Yep, just cross those two vectors and I've got the surface normal.

One of the hardest things to understand about normal vectors is how to transform them. It is a constant source of frustration and postings on the Highend3D mailing list. Here is the problem. They don't transform like points. They also don't transform like direction vectors. They are a special case, because they aren't really directions, per se. Surface normals need to stay perpendicular to the tangent plane, and so the right way to compute them is to transform the tangent plane, and then see where the surface normal ended up. To see why, look at this diagram:

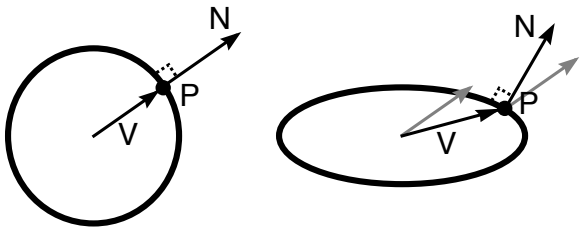


Figure 1.1: Transforming a point, direction vector and surface normal.

As you can see, if you transform a circle by a stretching transformation, points on the surface stretch into an ellipse, direction vectors stretch the same way, but surface normals actually contract. If you measure carefully, you would discover that they shrink by the reciprocal of the stretch. If the circle stretched by 3, the surface normals would contract to $\frac{1}{3}$.

Lots of math far beyond the scope of this lecture leads smart mathematicians to the conclusion that you can transform surface normals *if* you use the transpose of the inverse of the transformation matrix. Even more math proves that this is exactly the same as the inverse of the transpose of the transformation matrix. Sadly, a little more math yet proves that the inverse transpose matrix *is often equal* to the matrix. Just not always. Which is why people think they can get away with transforming normal vectors the same way as direction vectors — sometimes it just happens to work. But if you want to be a pro that never has to deal with bizarre inexplicable shading mistakes on objects the first time they squash and stretch, always remember that normal vectors transform differently than direction vectors. Keep track of which of your vectors are directions and which are normals, always use the right transformation function. You'll be glad you did.

1.1.2 Topology

So, if you think that vector algebra is difficult mathematics, wait 'til you hear about topology. At the layman's level (and believe me, we are not going to get into this any deeper than the layman's level), topology is the mathematics of shapes. Since computer graphics is all about drawing pictures of shapes, you'd think that a thorough of topology would be important for doing great computer graphics. Fortunately for us, that's not the case. But understanding a few of the buzzwords that we topology novices throw around is important for polishing that superstar TD aura.

First and foremost, there is the term *topology* itself, as it is used to describe different shapes. Shapes are divided into classes, and all shapes that share certain a certain property are together in a class. The property in question is that they can be "continuously deformed" into each other. By continuously deformed, they mean stretching, shrinking, bending, and otherwise reshaping the object, but absolutely not cutting, poking holes, filling holes or pushing the surface through itself (self-intersection). For example, if you take a normal six-sided closed cube, smash down the corners and push in the edges, you can get something that looks like a die (singular of dice). If you keep pushing and molding and deforming it, eventually you can get a sphere. So, a cube, a die and a sphere are all *topologically equivalent*. A torus, on the other hand, cannot be smashed or deformed in any way into a sphere, because of the hole in the middle. Therefore, it is in a different topological class. Topological equivalence has nothing to do with how many sides an object has, how wildly twisted its limbs are, or how much volume it fills.

In computer graphics, since we're not studying the subtle mathematics of topological equivalence classes and their relationship to Fermat's last theorem, we tend to use this term very loosely. For better or for worse, we sometimes use the word topology to describe the connectivity of the

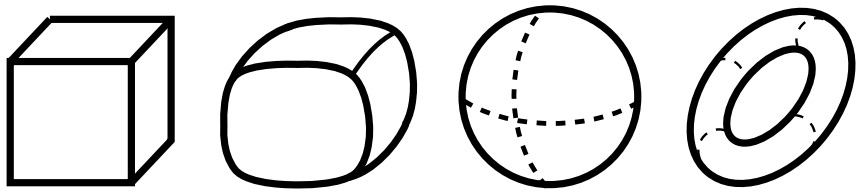


Figure 1.2: One of these things is not like the others.

edges of our objects. For example, we might say that a cube and an octahedron have different topology because they have different numbers of faces, edges and their vertices are connected together differently. This is important to us because our rendering algorithms must treat the object differently due to these connectivity differences, so for us, they are not "equivalent".

Another bastardized but common use of the word topology is in describing texture mapping coordinate systems. A hemisphere, for example, is topologically equivalent to a disk, and hence to a square. So is a sphere which is 99% closed with just a tiny hole at the top. But a topologist would claim that a closed sphere is not topologically equivalent because it is missing the hole. Well, for texture mapping, that's a small detail. We probably built our texture mapping coordinate system for the hemisphere using polar coordinates, or with some other trick, as though the whole sphere existed anyway. We certainly would do that with the 99% sphere. So a 100% sphere is just slightly larger, no big deal. In other words, we like to say that two geometric primitives are "texture mappingly equivalent" if you can slice it and unfold it, and then squash and strech and bend and whatever until you get two shapes on a plane that are the same. So, a sphere can be sliced down the "prime meridian", unwrapped, and with the appropriate stretching at the poles, teased into a square. But a cube can't be. A cube unfolds into a cross shape.

Well, now, the odd thing is that, of course, it easily could be. I mean, you *could* slice a cube the same way you slice the sphere, from the center of the top face to the center of the bottom face, then unwrapping, streching the poles and making a square. Sure, you could. But you wouldn't. Computer graphicists have an intuitive understanding that such a trick would introduce odd discontinuities in the texture coordinates on the top and bottom faces that weren't there in the original geometry. A slice has been made where there wasn't originally an edge. It is just, well, *wrong*, in some odd but universally understood way. So we say that in our line of work, a cube is not topologically a sphere, and anyone who creates projections that make them equivalent is performing a trick or is cheating or is gonna get themselves into a heap of trouble.

Another topology term that I've been talking around is *manifold*. It's stuffy, haughty sounding term that usually only college professors use. But the term is occasionally useful in polite conversation, and the concept is extremely important. A manifold is a connected set of points which is topologically the same dimension. For example, all the points that lie on a line segment, or in a bounded region of a plane. The important details are that all the points within some neighborhood are included, there are no arbitrary discontinuities, and that the dimensionality is uniform. At any point on the manifold there is a reasonable local coordinate system, and adjacent neighboring points all have similar local coordinate systems. Mathematicians call this "locally Euclidean". Manifolds are not restricted to shapes that are "globally Euclidean", because that excludes anything that is

curved, like a sphere, since the local coordinate system at every point on the manifold is different.

So, a manifold corresponds to what we generally think of as a "geometric primitive". It might be two dimensional, like a sheet, or three dimensional, like a spherical volume, It might be (in fact, often is) *embedded* in a space which has a higher dimension (a plane sitting in 3D). The fact that our geometric primitives correspond to mathematical manifolds is very useful when it comes to calculating various properties of the surface, like tangents, surface normals or texture coordinates. In fact, the computer graphics concept of topology is more akin to equivalent manifolds than to true topological equivalence, since we are always aware of and dealing with places where discontinuities mess up the local coordinate systems.

1.1.3 Walking the Transformation Stack

Most modern graphics APIs, RenderMan included, have a hierarchical transformation stack. This stack allows objects to be placed in the universe "relative" to their parent, grandparent and all their ancestors who came before, back to the mother of all coordinate systems – the *world* coordinate system. At every node in the transformation stack, the API maintains a transformation matrix which permits points (and direction vectors and normal vectors) to be transformed from the then-current "local" coordinate system back to world coordinates, and vice versa. This matrix is generally referred to as the *current transformation matrix*. The API calls which modify the current transformation matrix move the local coordinate system, so that objects which are added to the scene description can be in different locations and orientations.

In many modern graphics APIs, RenderMan included, there is a strange relationship between the position of the camera, and the position of the "world". Although TDs generally think of the camera as being an object to be placed in the world, and typically modeling packages generally maintain it as such, the renderer takes a slightly different view. Since the point of rendering is to figure out what is visible to the camera and make a picture of it, the camera is quite special. In fact, it is so special that it gets the priviledge of having the world laid out at its feet. In these immediate-mode APIs, the *camera coordinate system* is the Aristotlean center of the universe, while the world coordinate system is merely a convenient reference point from which to place *everything else* in the scene description.

From the mathematical point of view these are exactly equivalent. Everything is *relative*, after all. The matrix which puts the world in front of the camera is simply the inverse of the matrix that would have put the camera into the world. But by providing it in this camera-centric way, the immediate-mode API has the camera information it needs early enough to satisfy the requirement that the renderer knows everything that is needed about each geometric primitive when that primitive arrives.

So, a typical scene description contains a complicated sequence of stacked transformations that place all of the objects in the scene. If a modeling program is writing the sequence, and a rendering program is reading the sequence, this is all fine because the authors of both programs have figured out how to talk to each other using the API. But imagine the TD needs to listen in on the conversation. Or the TD wants to write his own transformation sequence by hand (horrors!). Reading a long sequence of transformations seems like it should be pretty straightforward, but for some reason it never really is, and your object always ends up being in Kalamazoo instead of in front of the camera. Why is that? Well, there are two equally valid styles of reading a sequence of transformations. Both ways lead to the same resulting view. However, it is very easy to get confused if you don't realize the difference between the two styles, and stick with the one that makes more sense to you.

The first style (which is my personally preferred style) is camera-centric. Your eyes are the camera, and your left hand out in front of your face is the "current coordinate system". Stick your fingers out in the official TD salute (thumb and index finger pointed like a gun, as the x and y axes, respectively, and raise your middle finger half way, and in that uncomfortable but noninsulting position it is the z axis). You read sequences of transformations from the top down, and each one

moves your hand around just as the transformation says. In this style, rotations spin your hand, and directions are always based on the current orientation of your hand. So if you rotate and then translate, you move your hand in a different direction than if you only translated. When you then place an object into the scene, it goes into your hand. The camera sees the object just as your eyes see it.

In the second style (which is Steve Upstill's favorite, in case you've read the *Companion* and been confused by him) is object-centric. You imagine the object to be at the origin of a large gridded space – its local coordinate system. You read sequences of transformations from the bottom up, and each one moves the object relative to the grid. In this style, directions are always relative to the stationary grid, so rotating the object revolves it about the origin of the grid, *not* around its own center, and does not change future directions of travel. When you reach the top of the sequence, the object is in its final place. The camera is then placed at the origin, and looks out over the z axis at the object.

As you can see, the main difference between these styles is the effects of rotation. In camera-centric style, your coordinate system spins around its own center. In object-centric style, objects revolve around a fixed center. In either case, always remember the alphabetical rule of rotation. A positive rotation around the x axis carries y toward z, position rotation around y carries z toward x, and positive rotation around z carries x toward y.

In both of these of these styles, you have to be able to deal with *mirror transformations*: transformations that change the directions of the fundamental axes. There are two types of mirror transformations. Scaling an axis by a negative number reverses its direction. Transformation matrices can also be used to swap (exchange) the directions of any two axes. In both cases, mirror transformations are said to change the *handedness* of the local coordinate system. In camera-centric style, changing handedness literally means changing hands. When a left-handed coordinate system is mirrored, your right hand can then be used to point the correct directions. In object-centric style, changing handedness means redrawing the stationary grid, so that the axes point in the new directions.

1.1.4 Color Science

In nature, the color of light is represented by an energy spectrum. In CGI, the color of light is represented by three numbers. You might think that this is completely unlikely to work, and yet we all know that it does. Why is that? Let's try to find out.

Color Models

The representation of color a three numbers, aka *tristimulus values*, is common to almost all color representations. The most common one that we TDs are familiar with is *RGB*, which is semispectral in the sense that red, green and blue are colors that have representations in the spectrum. Artists, on the other hand, are much more likely to be comfortable with *CMY*, cyan, magenta and yellow, even though they almost certainly call them red, blue and yellow.

The fundamental difference between these two color spaces is that RGB are *additive* colors, while CMY are *subtractive* colors. Here "additive" refers to the fact that our display devices emit some red, some green and some blue, and the result that we see is due to the combination (addition) of all three lights together. So, as we are all aware, if you have something which is glowing red, and you add some glowing green, you get glowing yellow. "Subtractive" refers to the fact that with mixed paints or dyes, the color that you see is what is left from white after you've applied a paint that absorbs (subtracts) various colors. If you have something with everything but magenta removed, you see magenta, and then you add cyan, which removes everything but cyan (including magenta), you are left with a really dark (generally brownish) smudge.

There are many other mathematical models for color floating around the CG biz. For example, artists who have a hard time understanding that red plus green equals yellow sometimes find *HSV*,

hue, saturation and value more intuitive since it corresponds to concepts from their color theory classes. Sometimes color models are hardware dependent, such as the old *YIQ* model, which is the trivalue color model that is used for encoding NTSC television signals.

Metamers

So, why three? Well, each of those models has three for different peculiar reasons. But RGB has three for a very specific reason related to human color perception. In your eyeball, on your retina, there are three types of cells called *cones* which respond to spectral energy from certain narrow regions of the visible spectrum — one sensitive to reds, one sensitive to greens and one sensitive to blues. In other words, you actually "see" in RGB. This leads to an interesting phenomena: there are a variety of spectra which are different, but which stimulate the cones in an identical way. So, they are physically different colors, but they look exactly the same to us. Such colors are called *metamers*. So, if a display device wants us to see a particular color, it doesn't have to generate the exactly correct spectrum. It only has to create a spectrum which is a metamer of the correct one. By tuning the red, green and blue phosphors of a CRT display (or whatever color generating bits there are in displays of other technologies) to roughly match the sensitivity of your cones, the display can create anything your cones can see – anything you can see.

Well, this is all fine and dandy, but it doesn't work perfectly. First off, people are different (or so my father keeps saying). Second, it's really hard to match the cones with arbitrary technology even if we knew exactly what everyone's eyes did. Third, display devices can't generate as much pure honkin' power as the real world does, so displays are extremely limited in the overall brightness they can simulate. The result of all that is that the display can't really display every color you can see. The range of colors that it *can* produce is called its *gamut*. The gamut of a television is reasonably large, in comparision to say a printer, but is smaller than that of motion picture positive film, but it is really very small in comparison human perception.

Edge Detection

In addition to gamut problems, the special features of the human eye cause our beautiful CG images problems in another way, too. It turns out that the human eye is not terribly sensitive to absolute intensity, because the eye *accomodates*, meaning that it has a fancy autoexposure feature known as your pupil which allows you to see easily in a wide range of absolute brightness levels. It is not terribly sensitive to absolute color either, because your brain is really good at "filtering out" overall color shifts that might be due to the light sources not being purely white. And it is a lot less sensitive to color changes in the blues than in the greens, due to the way the cones are organized in your retina.

But it *does* notice contrast, relative changes in brightness from one part of the scene to another. In fact, the human eye is extremely sensitive to relative brightness changes. It goes so far as to accentuate them just to call attention to them. Is this some genetic throwback from the days when humans lived in trees in the jungle and had to see predators hiding in the grass? Who knows. But it is true that edges between differing contrast levels are enhanced perceptually, so that it appears that there is an even greater difference that there actually is.

On the "dark" side of the edge, there is a narrow band that looks even darker. On the "light" side of the edge, there is a narrow band that looks even lighter. These bands are called *Mach bands*, and while they are very difficult to print in a book such as this (because the halftoning messes up the edge), they are pretty easy to see on a display. Frightfully easy. Actually, they are so easy that it is hard to make the damn things go away!

Mach bands elimination is the main reason that renderers have to *dither* their images. Images are generally computed in full floating point precision, but when they go into the file or onto the screen, they have to be quantized down to the 256 or so integer input levels per color that the display

can handle. An image which has a slow ramp might be perfectly smooth in floating point, but in an 8-bit integer form it's all one value for a while, then suddenly jumps up to the next level, etc. in a stairstepping kind of way. Humans can easily see the contrast difference between these levels, and it looks awful. Hence dithering. Every floating point pixel is tweaked by a tiny random amount, generally half of one display level, before it is quantized. This means that some pixels will be artificially to bright, others artificially too dark. That will cause the stairstep edge to fuzz out, going up and down a couple times before finally going up for good. That's usually enough to turn off our perceptual edge-detection machinery, and the Mach band vanishes.

1.1.5 Local Illumination Models

Objects reflect light. Of course, they don't completely reflect all the light that strikes them. They absorb some, reflect some, and sometimes reemit some. The color of the light that leaves the surface of an object and gets into our eyes – the color that we see when we look at an object – this is the color that we say an object "is". Many things influence the behavior of light as it reflects off of an object made from a particular material. Some of these influences are purely geometric: which direction did the light come from; what is the shape of the object at the point of reflection; etc. Many of the influences are physical properties of the material itself: is it metallic, plastic, painted; how rough is the surface; does the surface have layers; etc.

The BRDF

If you were to take an object, and experimentally measure its light reflection properties, you could build a 4-D graph known as a *bi-directional reflection distribution function*, or *BRDF* for short. A BRDF is a function of two directions: an incoming light direction and an outgoing light direction. It tells you, for every incoming light direction, what fraction of light arriving from that direction will be reflected in each possible outgoing light direction. It is important to understand that the BRDF only cares about behavior at the point of reflection. Any other factors that may influence light – the intensity of the light source, the amount of fog in the scene, the sensitivity of the viewer to light, the color of nearby objects – do not influence the BRDF. The BRDF is only about the physics that occurs right at the spot where light is hitting and reflecting off of the surface. For this reason, the BRDF is the most general form of a *local illumination model*.

A local illumination model is a method (usually a straightforward mathematical equation, but it could be a more complicated algorithm) that estimates how much light leaves in a particular outgoing direction of interest, given the amount of light arriving from the directions that the renderer says it is arriving from. In other words, it tells us what color we would "see", if our eye was sitting along the outgoing direction. The history of computer graphics is littered with local illumination models. Some are famous, many are relegated to obscure Siggraph papers that noone references anymore. Probably every paper that proposes a local illumination model does so by defining a whole new set of obscure Greek symbols to represent specific geometric quantities or material properties which influence the model. But generally speaking, the model has something to do with these kinds of quantities:

- the color and intensity of the incoming light;
- the angle between the incoming light direction and the surface normal;
- the angle between the outgoing viewing direction and the surface normal;
- some measure of the roughness of the surface;
- some measure of the intrinsic color of the material;
- other material properties, like the index of refraction; etc.

Some local illumination models claim to be based in some way on real physics, by speculating that at a microscopic level the material has certain properties, and then determining precisely how

photons would react when interacting with such a surface. Other models are purely empirical, coming up with some function that seems to fit the data and then providing enough fudge factors to tweak to cover a range of possibilities. There are so many local illumination models because the physics of light interaction with materials is so complex that none of them are right. Each one does a reasonable job of approximating the BRDF of particular kinds of materials, under particular conditions, given certain assumptions. Some handle exotic materials with special properties really well, while others handle common materials very fast. For this reason, it is common for TDs to like to use one model when rendering plastic, another for metals, for wood, for skin, etc.

Reading BRDF Diagrams

At the end of any such paper, there will generally be diagrams which purport to show the intensity of the outgoing light that arises from a handful of incoming directions. In 2-D, one of these diagrams might look like this:

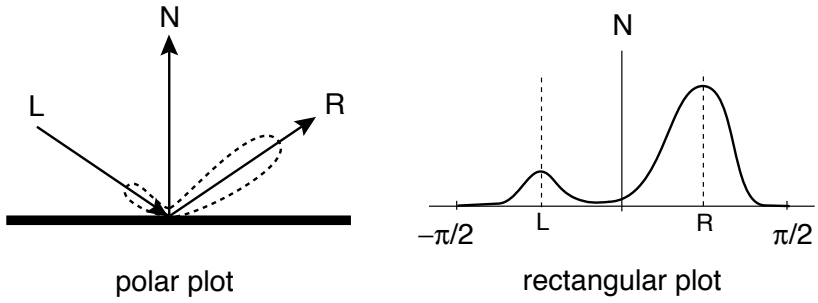


Figure 1.3: A typical BRDF diagram.

The question is: just what the heck do those bumps mean? For a very long time, I didn't understand these diagrams because I couldn't grok what the coordinate system was, and noone bothered to mention it after about 1978. The trick is that these are *polar* diagrams. The axes are not "left" and "up" like most graphs, but rather "angle" and "length". In particular, the independent variable is the direction of the outgoing light vector, represented as an angle around the point of reflection, which is oriented so that the surface normal points straight up. The dependent variable is intensity of the outgoing light, which is represented by the length of the outgoing vector. The graph is the dotted line, which indicates the length (intensity) at each angle (outgoing direction). The graph also indicates the incoming light vector ($\bar{L}$) for which this graph pertains, since generally the graph is very different at different incoming angles. Sometimes the mirror reflection vector ($\bar{R}$) is also shown, just for reference, as in this example.

Sometimes you'll see these graphs in 3-D, but usually they are in 2-D because they are generally the same if you rotate the whole diagram around the surface normal. In the uncommon cases where that is not true, the surface is called *anisotropic*, because it is not ("an") the same ("iso") when rendered in Hawaii ("tropics").

Diffuse and Specular

For most materials, the BRDF is dominated by two almost separable components. There is a general scattering of light in all directions, so that all viewers, no matter where they stand, generally see a little bit of *something* when they look at an object. This scattering is known as the *diffuse* component. Then there is a very focused, mirror reflection. Many materials which do not seem to reflect a

recognizable image (in other words, materials which are not mirrors) still reflect bright lights with a noticable "shiny spot". This focused reflection is known as the *specular* component.

Diffuse reflection is generally very evenly distributed. It is also the dominant reflection component for rough objects. One of the original approximations made at the very dawn of 3-D raster computer graphics was to assume that the light reflecting off of objects was in fact completely evenly distributed, and the local illumination model of choice in 1974 was Lambertian diffuse reflection. Lambert's model of diffuse reflection says that the reflectivity of an object is not dependent on orientation – neither orientation of the incoming light direction nor of the outgoing light direction. It reflects photons equally in from all directions, to all directions. The brightness of any given spot on the object only depends on how small it appears to the light source – the cross-sectional area. This is always confusing the first (or second) time, so think of it this way: consider two equal sized little regions on the surface of the object. One region faces the light, and so it appears large from the light's point of view. It has a large cross-sectional area, and will get hit by lots of photons. The other region is edge-on to the light, so it appears small, like a little sliver. It has a small cross-sectional area, and collects only a few photons. So, even though the individual photons get scattered uniformly in all directions, the first region will appear brighter because it has more photons to scatter. This is why Lambert's equation uses $(\bar{N} \cdot \bar{L})$, because the cross-sectional area is directly related to the angle between $\bar{N}$ and $\bar{L}$.

Specular reflection, on the other hand, is very focused. Light that comes in from one direction leaves almost entirely in some other one direction, or in a small neighborhood around it. The size of the beam is a function of the roughness of the surface. Because the light is restricted to a narrow outgoing beam, it is very bright if the viewer is in that beam, and nonexistent if the viewer is outside that beam. The most common effect caused by specular reflection are *specular highlights*. Specular highlights are, in fact, the mirror reflection images of the light sources in the scene. So, if the light sources are spheres, like light bulbs or suns, the specular highlights should be circular. But if the light sources are bright windows, perhaps the specular highlights ought to be squares. Interestingly, they often are not rendered that way. This is because the early crude approximations for specular reflection, such as Phong's original model, actually simulated fuzzy reflections of a point light source, rather than true mirror reflections of an area light source, which is the way the real world works. The highlight was round because the function used to shape the beam fades out in concentric circles determined only by the angle from the central reflection axis. Now, it is true that a rough surface has naturally fuzzy reflections, and this will cause a true mirror reflection of a round light bulb to fade out in a nice smooth way. But a fuzzy mirror reflection would also take into account the size and closeness of the light. Typically, specular functions just fake it by giving the user the ability to crank up the surface roughness much larger than it really ought to be, which means you can get the hack the model to get a believable look, even if it is not right. A wide variety of specular illumination models have been developed since Phong's groundbreaking but nevertheless hacky original, so it is baffling to me that people still like to use Phong specular. I guess they'll never be guru TDs like you!

Many modern local illumination models still have components that are fundamentally diffuse and specular, even though the specific equations and behaviors of those two components are not as naively uniform as the originals. These two components represent opposite ends of the spectrum in terms of photonic identity – diffuse represents energy coming from all directions, mixing and bouncing off uniformly in all directions, a melange of indistinguishable photons; specular represents the life of individual photons as they travel well-charted paths from a specific light source, bouncing around the scene and eventually reaching the viewer's eye. Because of this, different rendering algorithms specialize in simulating the effects of each. The study of new rendering algorithms is always partly driven by finding new and better ways of simulating either diffuse or specular reflection, or things in between.

1.1.6 Aliasing and Antialiasing

Everyone has heard of *aliasing* and *antialiasing*. Everyone has probably heard of Alias|Wavefront, too, but that's a different thing. Just why do computer graphics images alias, what is an alias, why does antialiasing antialias, and since when is "alias" a verb? These are the truest mysteries of computer graphics.

Aliasing

Aliasing is a term that is borrowed from signal processing, a branch of electrical engineering which predates computer graphics by several generations, so they understand it a lot better than we do. An "alias" of an intended signal occurs when equipment that is processing the signal can't handle it for some reason, and so mangles it, and makes it look like an entirely different signal. That different signal is called the alias.

The most common reason that signal processing equipment cannot handle a signal is because of its frequency spectrum. Any piece of electronic equipment, analog or digital, can only handle signals within particular frequency ranges. Commercial AM radios can only receive transmissions between 530kHz and 1700kHz. Stereo amplifiers can only amplify signals between about 20Hz and 20kHz. Outside these design ranges, their effectiveness falls off rapidly. Biological equipment has the same problem — the human eye can only register signals between 400 and 790 terraHz.

Digital electronic equipment often uses *sampling* to capture analog signals. By sampling, we mean measuring and storing the amplitude of the signal at some regular interval, like say, 44100 times per second. The problem with sampling is that it doesn't store the whole signal. It only stores those sampled amplitudes, like a series of dots on a page, under the assumption that you'll be able to "connect the dots" later on and get the original signal back. Or at least, something that you consider to be close enough to the original signal that it's worth it to spend time downloading it off the Net. That process of connecting the dots is known as *reconstruction*. Reconstruction is a very important subject, because if you do it badly, you can screw up a perfectly good mp3. And good reconstruction is not always easy, as we'll see later. But at this point, we'll skip one hundred years of math due to some smart, but non-TD guys, like Fourier and Nyquest and Shannon, and get to the punchline: even if you have the absolutely perfect method of reconstruction, you can never get the original signal back if you don't sample often enough. You've just thrown away too much of the original. Which brings us back to aliasing: your intended signal has been mangled and now it comes out sounding like something entirely different — an alias. Ta da!

Reconstruction

We've mentioned reconstruction, so let's look at this step more carefully. Reconstruction is the process of creating a continuous signal out of the sampled version. Reconstruction is generally done with *filters*, which are equations that specify, for every point in the continuous output, how to add up some linear combination of nearby samples to create the output value. Filters are identified by their shape, that is, the graph of the equation, and their width, the span of the non-zero portion of the graph, which in turn identifies how many nearby samples will be necessary to do the calculation.

Filters are applied to a series of samples with a process known as *convolution*. Convolution is tricky to explain. The basic idea is that you slide the filter continuously along the axis that contains the samples, and at every point along the axis, the output of the filter is the sum for all samples of the product of the height of the sample with the height of the filter that is covering that sample. Since the filter is nonzero only in a fixed region, and the samples are nonzero only at their sampled positions, this is just a simple finite sum. If either were nonzero over a broader space, we'd have to call it an integral (bleah!).

So, for example, consider the simplest filter, a *box filter* of width 1.0. A box filter looks like the shape in the center of the top row of the diagram. It is zero everywhere except between -0.5 and 0.5,

where its value is 1.0. Because of this shape, it always covers exactly one sample no matter where it is on the axis, so when it is convolved with (slid over) the samples, it spreads each sample out to cover the entire region between ± 0.5 around the sample. In the middle row of the diagram, we see a triangle filter with width 2.0. This filter always covers two samples, and as it slides towards a sample the filter increases, making that sample add more to the value. As it slides away from a sample the filter decreases again, making that sample contribute less. This means the output ramps between the samples.

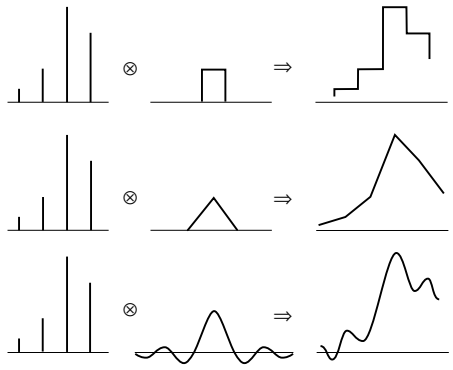


Figure 1.4: Convolution with simple filters.

Signal processing experts know much about the characteristics of various filters. For example, they know that the most perfect filter is a sync filter with infinite width. By "most perfect" we mean that it always does the theoretically best possible job of recreating the input signal given the samples. Unfortunately, infinite width means that *every* sample has at least a small contribution to *every* output value. That makes it pretty impractical for everyday use. But it means also means that any other filter we might choose is in some way suboptimal. For example, box filters *blur* the output more than they need to. Filters that counteract the blurring by trying to stay sharp will instead *ring*, meaning that the output oscillates and can't stay steady. Every filter will have some balance between blurring and ringing.

Some reconstruction filters have a property known as *negative lobes*, as we see in the bottom row of the diagram. These occur when the graph of the filter has some values less than zero. These filters are the ones that ring, which is not necessarily a bad thing if a small amount of ringing can be traded for removing a large amount of blurring. However, some people specifically refuse to use a negative lobe filter because the oscillation of the ringing can make output values of the filter go negative when the input samples are very low/quiet/dark. They cannot abide by the thought of a negative pixel value, and all the horrible things that will happen to the universe if a negative value were to collide with a positive value and annihilate each other in a massive matter-antimatter explosion. (By the way, those perfect sync filters have negative lobes, in case anyone asks.)

Chosing a reconstruction filter is always a delicate balance between evils. Some filters blur, some ring, some do some of both, some are definitely better than others, and you can never find one that is perfect. But finding a good one is critical to doing good antialiasing, because doing way excellent sampling gets you nowhere if you mess up all those beautiful samples by passing them through a crappy reconstruction filter. Try a few and decide which you like best.

Pixels

Now let's apply all that to computer graphics. Let's start with examining your computer screen. Yep, you've got that superhigh-res ultrasmall-dot microblack TFT-LCD with XJ-17. Your monitor has a resolution of nearly 2000 pixels horizontally and 1500 pixels vertically, at a magnificent 85 dpi. State of the art. This means that you are giving your eyes the veritable feast of a whopping 10kHz of visual signal information (assuming you sit about as far from your monitor as I do). My goodness, that seems a bit low, don't you think? I mean, you give your ears four times that much data, and even then people complain that they can hear the "digital artifacts" in the music. Surely your eyes deserve more respect that that!

This is the source of almost all of our problems. Our eyes are extremely sensitive devices, and our visual acuity is easily good enough to see a single isolated pixel on the screen. After all, printers are considered passe' if they don't do at least 600 dpi. This low-frequency device is not capable of handling stored visual signals higher than about 5kHz, and so mangles them, and presents us with *aliases* of the intended visual signal.

The most common form of pixel aliasing is *jaggies*. This is the artifact that causes a perfectly straight line or edge to appear as a stairstep pattern. In fact, jaggies are actually a reconstruction problem, not a sampling problem. If you look at the reconstruction filter diagram again, you can see that box filters turn samples into output signals with sharp jumps. Because our visual accuity is good enough that we can see the individual pixels, we can see these sharp jumps, and these are the jaggies. Using a wider box filter creates a related problem called *ropiness*, where a continuous line appears to turn into a rope-like spiral pattern.

Figure 1.5: Jaggies, ropiness and Moiré patterns.

Another common form of pixel aliasing is *Moiré patterns*. Moiré patterns occur when a group of parallel lines become very thin and close together in the distance, and when the sampling pattern can no longer capture them they start to come out as odd curves, dotted lines, thick line segments, and other obviously aliased patterns. This is classical aliasing because the problem is that the visual frequency of the thin lines increases dramatically as the lines converge in perspective, and at some point the sampling grid can no longer sample them appropriately. The frequency of the sampling *beats* against the frequency of the lines, so every distance will get its own alias, the beat frequency at that distance.

The obvious way to solve this problem is to increase the frequency of the samples. With more samples, you can capture a higher frequency signal, and you should be golden, right? Alas, this will only help sometimes. As the lines approach the vanishing point, their visual frequency increases without bound. You can never sample enough to capture them. (And this is generally true of computer graphics, not just goofy special cases like Moiré patterns. There are almost always frequencies present that are higher than whatever you would reasonably be willing to sample for.)

Other Aliasing Artifacts

Hopefully by now you understand that aliasing artifacts are a direct and inevitable result of sampling and reconstruction. This means that any other process that renderers perform that involve sampling can, and do, exhibit aliasing artifacts. Let's consider a few common ones.

For example, in modeling, you digitize a finite set of points on the surface of a clay model, and then in the computer you reconstruct them by connecting the points with polygons or some curved spline surface. Clearly, if you don't sample often enough, such that you skip over some high-frequency oscillations on the surface, then you will create an alias of the original surface. You might lose some interesting wiggly groove patterns on the surface, for example. If you do a really bad job, you might skip right over serious indentations or cracks in the surface. But even if you do a good job, but you look at the object too closely, your samples will get far apart, and you'll depend on the reconstruction to create data that isn't there. In other words, it will guess. If your reconstruction is poor, like with straight lines, then you'll see "polygonal artifacts". If your reconstruction is with fancy splines, you might get a surface that wobbles around near the original surface but doesn't actually lie flat upon it. Of course, you'll never know that if the object is so small on the screen that the samples you took fit nicely inside the pixels; in other words, if your sampling rate is higher than the pixel's sampling rate.

Probably everyone in this business has heard of *temporal aliasing*. This occurs in animation when our camera samples time discretely. The classic example of this phenomena is the *wagon wheel* effect. A camera is photographing a fast spinning wheel, at regularly spaced intervals (generally every $\frac{1}{24}$th of a second). If the wheel is rotating around so that it makes a complete revolution in exactly $\frac{1}{24}$ of a second, then the camera will see it in exactly the same position in every frame. It will look exactly like it is standing perfectly still. The signal of the wheel spinning very fast as been converted (aliased) into a signal of a stationary wheel. If the wheel spins just slightly slower, the wheel appears to move slowly *backwards*, an even more humorous aliasing artifact.

Temporal aliasing is generally combatted by the technique of *motion blur*. Motion blur certainly helps the problem, since it replaces discrete sampling of time with area sampling. But despite common wisdom to the contrary, it does not *solve* the temporal aliasing problem. After all, as we just saw, motion picture cameras alias, too! All motion blur does is change the artifact from one we laugh at (strobbing) into one that we are used to seeing (blurry streaks).

Another example is color. In computer graphics we almost always represent our colors with three values. Sometimes we use *RGB*, sometimes we use *HSV* or *XYZ* or *LAB*, and sometimes we even deign to borrow from television and use *YIQ*. Nature, on the other hand, represents color with a continuous spectrum. So, it is natural to guess that if we sample this continuous spectrum with only three samples, we will sometimes get *color aliasing*. A spectrum is sampled, some type of processing occurs on the samples, then the color is reconstructed, and the color comes out looking entirely different. Some people think that this phenomena is uncommon in computer graphics, that somehow because RGB is as old as the industry, or because of metamers, or because "experiments with 9-channel color were done and it didn't help", that somehow three channels is the divinely chosen number. They couldn't be more wrong. Anyone who has ever struggled with monitor gamma correction, color matching of displayed images with printed images, or even just seen their images projected on television, knows that reconstruction from three sampled values to physical color is quite literally never the same twice, and is a terrible problem.

Now consider software which attempts to simulate the changes of colors as they reflect off of or transmit through surfaces of a different color. In the real world, this process is effectively multiplying two whole spectra together. In CG, we just multiply our three lonely sample points. It should be obvious that any interesting interactions and subtle variations that happen in the full spectrum in between our samples, will go completely unnoticed by the sparcely sampled version. Have you ever seen a truly believable photorealistic image of stained glass? Yes, color aliasing is with us all the time.

As you can see, anytime a continuous signal is sampled, aliasing can and will occur. The artifacts can be somewhat ameliorated by wise choices in sampling rates, or special sampling techniques or clever reconstruction, that converts "unacceptable artifacts" into "less objectionable" or "more familiar" artifacts, but information is always lost and signals are always changed. That is the nature of aliasing.

1.1.7 Antialiasing

So, we don't want to see all these aliases. What do we do about it? Well, if we can't display an image with all of the frequency content that exists, let's consider what the next best picture might be.

Go back to the Moiré pattern generator, parallel black and white lines receeding to the vanishing point. It seems like the best solution that you'd want would be getting a sort of uniform grey that represented the average color of the black and white lines in the right proportions. That just cannot happen with normal uniform sampling and standard reconstruction. There is no way to force the samples to hit the lines and hit the background with exactly the right percentages everywhere. They land where they land, and you're stuck with it.

One approach would be to look at small regions of the floor, and analyze how much black and how much white are in that region, and return the appropriate grey. This technique is called *area sampling*. It is very appropriate if you can choose the size of the regions wisely (such as, the amount of the floor that all is visible in the same pixel), and if your model of the lines lends itself to calculating the areas of the intersections of the pattern pieces with the region. If your lines were made of long black and white strip objects, for example, you might be able to mathematically calculate the area of overlap. But if the stripes were the result of running some equation at any given point on the surface to see if it was black or white, well, it might be very difficult to determine the area within which the equation would always return the same answer.

Another popular technique is known as *prefiltering*. In this technique, whatever function is generating the pattern is simply not allowed to create a pattern with frequencies that are too high. For example, the black and white strips might not be allowed to be thinner than a certain width, and as they go back in distance they are forced to become thicker and thicker to compensate for the shrinking in perspective. Whatever function generates them would have to figure out how to handle the fact that they would start to overlap, and "do the right thing". If the mathematical equation was used, it would have to be tweaked so that points that were close together could not have very different values, and therefore the color couldn't change rapidly enough to cause the aliasing. This is in many ways the best solution, because it removes high frequencies before they even exist, and hence no aliasing is possible. Of course, as you may have guessed, it is very difficult for the pattern generators to "figure out what to do" and make sure that in all cases "points that are close together don't have very different values".

The final popular technique is to remove the restriction that the sampling happens at a predictable, regular interval. Instead, take the same number of samples, but spray them all over the place at random and see what they hit. This technique is called *stochastic sampling*, and was developed by Pixar in the mid 1980s. The idea is that if there is no way to force uniform samples to hit the black and the white with the right percentages everywhere, just fire at random and you'll get something close to the right answer. You'll never get exactly the right answer, but since there is no pattern to your sampling, there will be no pattern to your errors either. You won't get a Moiré pattern, but rather you'll get a jumble of *noise*.

If you remember the Mach band discussion, the human eye is pretty sensitive to seeing Moiré patterns and other contrasty edges. But it's not terribly sensitive to noisiness, which I guess our jungle perception just interprets as haze or something. Some say it has to do with the natural noisiness in the pattern of rods and cones on our retina, but I'm not sure. In any case, the result is, if you ask 100 people, 95 will say that the noise is a *less objectionable artifact* than the aliasing.

Unquestionably, antialiasing CGI is an art, not a science. In the afternoon sessions, various speakers will touch on the need to, and the techniques for, antialiasing various types of patterns in shaders. Listen carefully and read their notes twice. Doing antialiasing well is what separates the men from the boys.

1.1.8 Rendering Algorithms and Pipelines

For our purposes, we'll define a rendering algorithm as an algorithm which takes an abstract description of a scene, and of a camera in that scene, and creates a 2-dimensional image of what that scene would look like from that camera's point of view. It creates a synthetic photograph of the virtual scene. In order to make this picture, the rendering algorithm has two basic functions: it must determine what object is visible in each pixel; and it must determine what color that object is at that pixel. The first question can be called *visible surface determination*, or alternatively *hidden surface elimination*, or simply *hiding* for short. The second problem is *shading*.

Hidden Surface Elimination

Computer graphics practitioners have been writing rendering software and building rendering hardware for nearly 30 years. As you might imagine, in that time, a lot of different techniques have been tried. Relatively early on, a very brilliant practitioner observed that the fundamental difference between the many different hidden surface elimination algorithms that were being espoused at the time was that they sorted their scene description database in different orders. He was the first to recognize that "computer graphics is just sorting."

In time, computers got faster, memories got larger, scene databases got much larger, and our requirements for realism and features got much much stronger. Many of the early hidden surface algorithms didn't scale well, generally because they didn't sort in the right order. Nowadays the popular hidden surface algorithms can be divided into two categories: *object-order*, and *pixel-order*.

An object-order algorithm proceeds as follows:

```
for each object in the database
    determine which pixels it might be in
    for each such pixel
        calculate how far away the object is in that pixel
        if it is closer than any object yet seen
            put this object in the pixel
        endif
    done
done
```

Object-order algorithms are quite common. You probably have a hardware implementation of one of the oldest such algorithms in your PC at home — the Z-buffer. Reyes, the algorithm used by *PhotoRealistic RenderMan* is also an object-order algorithm. The differences between such algorithms lie in how they deal with the details of "which pixels it might be in" and "put the object in".

As you might expect, pixel-order algorithms proceed like this:

```
for each pixel on the screen
    determine which objects might be in the pixel
    for each such object
        determine how far away that object is
    done
    sort the distance results
    put the closest object in the pixel
done
```

The most common pixel-order algorithm is, you guessed it, ray tracing. However, many different variations of ray tracing exist, based on how they deal with "which objects might be in" and how they manage the sort.

Shading

In modern renderers, whether software or hardware based, the main features that make its images appear to be "photorealistic" are features which enhance the power or scope of its shading engine. It is completely true that the appearance of an object is as important, or even more important than, the shape of the object when it comes to making believable images. For this reason, a renderer's ability to create great images is fundamentally tied up with the power of its shading engine. Shading algorithms can also be divided into two categories: *local illumination* and *global illumination* algorithms.

Local illumination algorithms are algorithms which compute the color of objects based only on the local illumination model of the object itself. The renderer may have a wide variety of material properties and geometric quantities of the object at its disposal while computing its color, and also generally will have access to a common database of light sources and their properties, but it does not have access to the properties or characteristics of any other object in the scene. Each object stands alone, reflects light directly from the light sources towards the viewer using its own local illumination model, and has no knowledge or interaction with any other object in the database. Z-buffers are examples of such algorithms that shade each object in isolation.

Global illumination algorithms, on the other hand, are able to access information about how light is moving around the scene globally, such as how it is reflecting off of other objects in the scene. From this information, it can process not only light that is coming directly from lights, but also light that arrives indirectly after having interacted with other objects in the scene. I'm sure everyone is well aware that ray tracers and photon mappers are both considered global illumination algorithms, because they can process various forms of indirect illumination. Global illumination algorithms can be further categorized by how many levels of which kinds of indirect light interactions can be detected by the algorithm. Some handle only indirect specular light, while others handle indirect diffuse.

Another way a rendering algorithms' shading engine can be categorized is whether it is fixed or procedural. In a fixed shading model, the designers of the renderer have determined what the local illumination model will be, what parameters it will have, and by what methods and under what conditions those parameters can be manipulated. For example, an OpenGL renderer might support 8 lights, either distant or spot lights, with a Fast Phong lighting model on vertices with 7 surface parameters (`color`, `opacity`, `Ka`, `Kd`, `Ks`, `roughness`, `specularcolor`) plus texture-based color and reflection mapping on a per-pixel basis.

Alternatively, a renderer might have a procedural shading system, where for every object in the scene, a program is loaded which calculates the color at every point on the surface of the object. Such systems are extremely flexible from the user's point of view, since the user can write a program to do bizarre shading that no renderer implementer would have ever thought of. They are the heart of many photorealistic software renderers. Unfortunately, they are not practical in some renderer implementations, particularly hardware renderers that run their graphics computations on specially designed but inflexible GPUs.

Pipeline Order

There is one additional variable in the design of rendering algorithms. This is the question of when in the pipeline shading occurs. In particular, an important factor in managing the speed and memory efficiency of a rendering algorithm is to determine whether shading should happen before hiding or after hiding.

For example, in a standard ray tracer, the renderer doesn't bother to calculate the color of a ray hit until it has determined that it is the frontmost hit along that ray. Therefore, the shading of the object occurs only after the visible surface has been determined — shading *after* hiding. In a conventional Gouraud-shaded Z-buffer algorithm, the vertices of polygons are shaded immediately after they are transformed, and these colors are copied into any of the pixels that the Z-buffer determines should be visible – shading *before* hiding.

Some algorithms do not fit so easily into such naive categorization. For example, modern interactive graphics cards divide their shading calculations into phases, some of which happen before hiding and some after. This merely goes to show that in the constant battle to design graphics pipelines that are fast and efficient as well as powerful and flexible, the question of which sorting order is "best" will never be answered.

1.1.9 Texture

The desire to make rendered images of objects look more complex, less symmetrical and generally more *realistic* than our simple geometric primitives and straightforward local illumination models allow has been true from the early days of computer graphics. Over the years, the lore has accumulated a great variety of ways to apply texture to the surface of objects in order to enhance their realism. These techniques are the breakfast cereal of serious superstar TDs, so let's make sure we've all had our USDA-approved balanced breakfast.

Texture Maps

In the beginning there were *texture maps*, and they were good. Unfortunately, they were badly named. When texture maps were invented in the late 1970's, they were simple color images that were pasted onto (parametric) primitives, like posters on a wall. The bad thing was that these images didn't really add "texture" in the artistic or architectural or any other sense, but the name stuck anyway, so now many people commonly call images that apply diffuse color to a surface a "texture map". Fortunately, those of us in the photorealism business have seen enough other tricks with images that we apply the name "texture mapping" to generically mean the application of any image in any part of the shading calculation. What the unwashed would call texture maps we call *diffuse color maps*.

A couple years later, computer graphics was introduced to its first truly spectacular shading trick, the *bump map*. With this technique, a texture is used to alter the direction of the surface normal just prior to calculation of the local illumination model. Because the local illumination model uses the surface normal to determine the orientation relative to the light source, tweaking it caused the surface to look bumpy. The first published image of a bump mapped object was of an orange. Now *there* was some texture!

In the early 1980's, an obscure little computer graphics research facility in Long Island called NYIT made a series of fundamental contributions to computer graphics trickery. One of the most lasting was the *reflection map*. With this technique, a 360 degree photograph of the space around an object is captured by some means (for example, photographing a gazing sphere). The object is shaded by first calculating the mirror reflection angle off of the surface and then using that direction to index into the map. Fake reflection, with no ray tracing needed! Another similar technique which was lost for decades but recently rediscovered was the *illumination map*, where a picture of the light sources and other bright spots all around each object is created, and this map is used instead of normal light sources to illuminate the object. Fast environmental lighting, no light sources needed!

The mid 1980's was a hotbed of new uses of texture maps. We saw the first discussion of *refraction maps*, where a reflection map image was indexed by the refracted ray direction (computed from Snell's Law) rather than the reflected ray direction, in order to compute better looking transparent objects. People started applying texture maps to other parameters of the local illumination model.

For example, if the local illumination model had a specular coefficient `Ks`, then replace a stodgy old constant `Ks` with a value from a texture map that varies over the whole surface. This could be called a *specular map*, and would give you an object which was more specular in some places than in others. Similarly, you could have a *diffuse map*, a *roughness map*, etc. Eventually people realized that all parameters of the illumination function could be textured, and should be. We would now call this entire class of textures *parameter maps*.

Textures that make the object more transparent in some parts and more opaque in other parts are known as *opacity maps*. The first use of opacity maps was a spectacular Siggraph paper that simulated clouds in a flight simulator. It was the first paper to definitively prove that complex photorealistic images could be made with extremely simple shapes that had subtle shading.

Displacement maps were introduced to replace bump maps. The main disadvantage to a bump map is that it doesn't actually change the silhouette of the object. The orange looks bumpy, because the normals point as though the surface had protrusions and dimples, but the silhouette is still precisely circular. Displacement maps solve this problem because they actually move the surface itself, rather than just tweak the surface normal. However, since moving the surface means changing which bit of the object is visible in each pixel, use of this technique requires a renderer that does shading *before* hiding. Renderers that do hiding before shading must make special (and usually painful) accomodations for objects that use displacement mapping.

Another very useful texture map is the *conditional map*. Consider an object which has parts made of two different materials, for example, a wooden box with a pearl inlay. The appropriate way to create the diffuse color and illumination parameters for the wooden parts and the pearl parts are very different, so it probably is insufficient to merely have all of the texture maps contain appropriate values at the various parts. Instead, the object can be shaded once as though it were all wood, and again as though it were all pearl, and the texture map is used to control which of the two versions is shown at each point on the surface.

On the lighting side, one of the most dramatic developments was the *shadow map*. In this technique, an image is created from the point of view of the light source, but it does not contain colors but rather the contents of the depth buffer (specifically, the distance to the nearest object in each pixel). When computing the contribution of that light to the main image, every point being shaded is compared to the values in the shadow map. If the point is farther way from the light than the value in the shadow map, then it is obscured by something (is in its shadow), and receives no light. If the point is closer to the light than the shadow map value, then the object is fully illuminated. This way we get shadows in our scenes, with no ray tracing needed!

Another lighting trick is the use of *cookies* or *slides*. These are texture maps that are used to modify the color and intensity of a light source as it shines onto the scene. The name "cookie" derives from the theatrical lighting term *cucaloris*, which is a board with holes in it that you place in front of a spot light. Stage lighting has lots of tricks – barn doors, bounce cards, gels, gobos, etc. – which are used to subtly control lighting. And one way or another all of these tricks have made their way into the virtual lighting setups of computer graphics. The name "slide" derives from the thing you stick in a slide projector, of course.

Procedural Textures

In the old days, textures were always precomputed, painted or captured, and stored as images. Nowadays we call such images texture maps, to distinguish them from a very powerful technique first developed in the mid 1980's called *procedural textures*. A procedural texture is used the same way as a texture map, in that it provides a value that can be used to tweak a normal, modify a parameter, scale a light parameter, etc. The difference is the way that value is generated. Rather than indexing into an existing rectangular 2-D image, a small program is run which generates the value. Procedural textures take parameters, like any other program, which can be the position of the point to be textured, parametric coordinates, multiple sets of parametric coordinates, or any other values that

might be interesting in making patterns.

Procedural texturing has certain great advantages over traditional texture mapping. The first is scale independence. Texture maps have limited size and limited resolution. If you want to make a picture of a very large object, you might have to repeat (or *tile*) a texture map many times to cover it. Such repeats are generally very visible in the final image. If you render a closeup view of an object, you might see a single texture pixel (*texel*) spread out over a large region, again very obvious. Procedural textures can be independent of scale because programs can easily be designed to generate a different value at every point in space, whether these points are far apart or very very close together.

The second advantage is size. Traditional texture maps can be megabytes of data to be stored on disk and loaded into the renderer. Procedural textures are typically a few to a few dozen (or perhaps at most a few hundred) lines of code. They are small, as even the most complex procedural texture ever written is described in fewer bytes than even a trivial diffuse color map. Natural textures which are often three-dimensional, like the rings of a tree, the density of water vapor in a cloud, or the striations of veins in marble, are extremely difficult to model realistically with a 2D texture map. You could try to use a 3D volume map, but they are almost impossibly huge. 3D savvy procedural textures, however, are no larger or more complex than 2D ones.

The third advantage is source. Texture maps come from somewhere else. Some painter painted it, or some photographer photographed it, or some art director provided an image from a book to scan. How completely boring! Procedural textures come from the studly programming prowess of superstar TDs, honing their skills, working day and night for the high five that follows the unveiling of a particularly cool image. Painters? We don't need no stinkin' painters!

There are certain disadvantages to procedural textures, though. The first is that they are difficult to antialias. The problem of antialiasing of texture maps was essentially solved about 20 years ago, and while not every renderer does exactly the right thing with reconstruction, they are generally good enough to use. Procedural textures need procedural antialiasing, a black art which is subtle and difficult and still developing. This afternoon we will spend a lot of time discussing antialiasing of patterns, and shader writers spend much of their lives working with this problem.

The second is *directability*. Directors are fickle. They never like our beautiful procedural textures just as they are. They always want it a little less orange on this bit, or a extra dimple right here beside the thingamabob, and lose the dark splotch here between the guy's upper left and lower left eyes, because it looks like another eye is trying to pop out of the skin. Procedural textures are often very difficult to control to such exacting requirements. Rewriting the program to get rid of the splotch without getting rid of all the splotches everywhere, or without making a new splotch appear under the lower right eye instead, is often very difficult.

As appearance modelers, the tension between use of canned texture maps and procedural textures is another subtle battleground. In the end, a combination of techniques is usually used, depending on scale, importance, ease of programming and freedom from artifacts.

Texture Projections

Whether texture maps or procedural textures are used, one of the main questions in applying the texture to the surface is quite simply how to identify which points on the surface get which bit of texture. Since texture is commonly two dimensional, and the surface of the object is a two dimensional manifold embedded in a three dimensional space (see, I *told* you that word would come in handy), it seems like it would be pretty straightforward to make a correspondence between the surface and the texture. In fact, it is so easy that there are probably half a dozen common ways to do it.

The most common way is texture coordinates attached to the vertices that define the surface. These texture coordinates, commonly referred to as `s` and `t`, are assigned by the modeling system to each vertex of the geometric primitives, whether they be polygonal meshes, patches or subdivision

meshes. At all points between the vertices, the renderer is responsible for appropriately interpolating the values of the surrounding vertices, to get a texture space which is beautifully continuous and free of undesired stretching, kinking or other artifacts. For parametric primitives, the obvious texture coordinate system is simply the parametric coordinates `u` and `v` themselves, since these are known by the mathematics to be a reasonably artifact-free space.

The other most common way is to take advantage of the position of the object in 3D, and convert the 3D coordinates of points on the surface into 2D coordinates of positions in the texture. Such conversions of 3D into 2D are called *projections*. There are any variety of ways to project textures, based entirely on the mathematical equation that is used to convert the three position coordinates (`x`, `y`, `z`) into the two texture coordinates (`s`, `t`). The simplest is the *axis-aligned planar projection*. In that projection, you simply drop one of the spatial coordinates. All points that differ in only that one coordinate get the same bit of texture. Slightly more complex is the *general planar* or *orthographic projection*, where each point is characterized by its position on a some plane that is parallel to the projection plane.

I visualize planar projections by imagining an image rectangle moving through space, dropping bits of texture as it is dragged along. Using that analogy, you could easily imagine other projections where the image rectangle moves not along a straight line, but on some curved path. As long as it doesn't rock back and forth (the plane stays parallel to itself as it moves), there remains a unique mapping of every point in 3D to a point in 2D, and you still have a valid (if harder to compute) planar projection.

Obviously if you can do orthographic projections of texture, you can similarly do *perspective projections* based on any field-of-view. These are computed by dividing two of the components by the third, just as our perspective camera projections divide `x` and `y` by `z`.

A small step up in complexity are *quadric projections*, in particular the *spherical* and *cylindrical projections*. In these projections, every point is characterized by its position on some sphere/cylinder that is concentric to the projection sphere/cylinder. Another way of looking at it is that the projection sphere grows from a dot (cylinder from a line) dropping bits of texture as it expands. The point on the texture map is then equal to the parametric coordinates of the corresponding point on the sphere. Generally, the spherical projection calls upon the polar coordinates, which are sometimes called *latitude* and *longitude*. Hence, texture maps that are accessed with spherical polar coordinates are sometimes called *latitude-longitude* or *lat-long* maps. Again, using the expanding balloon analogy, it is easy to see that you could project with any convex shape that grows and covers all of space, as long as every point in space is touched only once.

Of course, if it is not obvious, the 3D point that you use as the input to these projections can be in any coordinate system that is appropriate. For example, it is pretty common to do a perspective projection in the coordinate system of a spot light in order to access a shadow map or a cookie. Simply transform the point or vector into the most useful coordinate system right before projection.

1.2 RenderMan Terminology

Okay, so the title of this course supposedly had something to do with RenderMan™ so maybe we can hear something about that, now, please? What is it, how do I use it, why should I use it?

1.2.1 Standards and Implementations

RenderMan is a name which is greatly overloaded. It can refer, at times, to a document, a specification, a language, a class of rendering programs, a specific rendering program, or a way of life. Just to clear the air, let's set all these things straight.

The *RenderMan Interface Specification* is a document, first published by Pixar in 1989 and updated a couple times since then. It was the work of a number of bright individuals, the brightest

of whom was Pat Hanrahan, Chief Architect. The document described a standardized API (applications programming interface), by which computer graphics modeling programs would transmit scene descriptions to rendering programs. This interface was originally described in C, although we knew that over time, support for multiple programming languages would eventually be needed. The document was created because the bright individuals at Pixar and various allied companies, thought that it was nigh time for the burgeoning pleathora of modeling programs to have a standard way to communicate with the burgeoning pleathora of rendering programs, in a way similar to the way that such programs had a standard way (well, several standard ways, as it turns out) to talk to interactive 3D graphics hardware. The difference was that RenderMan was positioned as an interface for photorealistic image generation, while all the existing standard interfaces were aimed clearly at interactive image generation on then-current hardware platforms. The RenderMan Interface was originally intended to be a multi-platform, multi-vendor, multi-language high-end solution to proactively stabilize the balkanization which had not yet happened in photorealistic rendering, but which clearly was about to.

Well, things didn't work out that way.

The first rendering program to support the RenderMan Interface Specification was released by Pixar in 1990, and for marketing reasons it was called *PhotoRealistic RenderMan*, or *PRMan* for short. Unfortunately, no modelers and no other renderer vendors followed suit, and for years, *PRMan* was the only RenderMan-compliant renderer. As a result, many people to this day call *PRMan* "RenderMan" in conversation.

In 2002, however, things have finally changed. There are now several renderers which advertise that they are "RenderMan-compatible", and several modeling programs which converse with these renderers through direct or third-party interface modules. It is important to note, however, that when these programs advertise their compatibility, they are making claims of utility, not proven scientific facts. Neither Pixar nor any other organization attempts to verify that programs (even *PRMan*) rigorously implement the current version of the RenderMan Interface Specification. Rather, "RenderMan-compatible" now colloquially means "generally does what you expect from reading the *RendeMan Companion* and *Advanced RenderMan* books, and generally understands the commands that the current version of Pixar's *PRMan* understands." It is up to the purchaser and user to decide whether the programs they are using are "compatible enough".

1.2.2 Tenets

So, what makes RenderMan a good and useful specification? Why is it so much better than the alternatives? What are the basic philosophical maxims that I need to understand in order to grok the RenderMan religion?

The fundamental, overarching, guiding concept in the design of the RenderMan Interface is *embrace photorealism*. Every choice that was made was held up to this one guiding principle — is this idea/concept/description going to make it possible for renderers to create ever-more realistic images as techniques and algorithms grow beyond the imaginings of the original designers? Or does this idea limit the choices a renderer has, and force it to make images less real or more noticably imperfect than it is capable of? 12 years later, with hindsight, we know that we were not 100% perfect judges of which ideas were propelling and which were hindering. But generally were were successful.

From the principle of photorealism came three cornerstone tenets that fundamentally defined the appearance and structure of the Interface: modeling-level description; immediate-mode; and extensibility.

Modeling-Level Description

The RenderMan Interface allows the modeler to describe the features of a scene. The scene description says *what* is in the scene, what objects, what materials, what animation, and does not say *how* it is to be rendered.

For example, all of RenderMan's geometric primitives are high-level modeling primitives, not drawing primitives. That is to say, the RenderMan Interface allows modelers to describe their models as they store them internally. We outfitted RenderMan with high-order curved surfaces of every variety we could think of, so that modelers could pass them directly to the renderer with no loss of fidelity.

In contrast, drawing primitives are simple or low-level picture elements such as lines, 2-D shapes or 3-D polygons described as vertex and color lists. These primitives are designed to be easy for the renderer (often hardware renderer) to draw directly, but require the modeler to convert their internal curved surface primitives into renderer-specific approximations. These approximations sacrifice fidelity, and limit the eventual image quality. In RenderMan, the onus is on the renderer to convert high-level primitives into whatever renderer-specific drawing primitives the renderer's algorithms can best handle, making the modelers easier to write and the scene descriptions more general.

Of course, there may be interesting modeling primitives that some modelers have which are not part of RenderMan's built-in set. For example, a modeler might have a surface of revolution primitive, or a tree primitive, or an articulated cowboy pull-string doll with or without hat primitive. Should RenderMan be extended to include every such primitive? Well, no, obviously it can't support everything in the universe. The set RenderMan has is supposed to be general enough that anything that a modeler uses as fundamental primitives can be converted into RenderMan modeling primitives *without loss of precision*. Moreover, the set was built to give renderers primitives that might be easier or more efficient to process. For example, some renderers can handle spheres more efficiently than they can quadratic NURB patch meshes, so even though it is possible to describe a sphere in the form of a NURB, we still let RenderMan have a separate sphere call — for the benefit of that renderer.

Immediate-Mode

RenderMan Interface is a strictly hierarchical, immediate-mode interface. This means several things. First, and most importantly, there are no display lists in the RenderMan Interface. The scene description given to a RenderMan renderer is a complete description of the model to be rendered for a single frame. Unlike current versions of OpenGL, it does not have provisions for rendering the same thing again, but with this one thing changed. Why? Because display list interfaces assume that essentially the same image will be rerendered multiple times, with minor (or no) changes between frames, and that the time to describe the scene dwarfs the time it takes to render the scene. Moreover, it assumes that the memory it takes to describe a scene is small enough that storing it for future reuse is easy. Neither of these things are true in a truly photorealistic scene, where scene descriptions are huge, rendering time is (unfortunately still) long, and where most of the database changes in some way from one frame to the next.

With this choice comes the corrollary that no forward references are allowed in the scene description. That is, everything that you need to know to render an object, its position relative to the camera, shape and geometric attributes, color and material properties, interactions with lights, etc. are *all* known when the object is handed to the renderer. The renderer does not need to wait for some unknown amount of time for the object description to be finalized by later information. For example, the camera description is the first thing in the scene description, so that all objects already know their place as soon as they are added to the scene.

Extensibility

In 1989, we knew that we had no idea how fast or in what direction rendering technology would progress in the near, let alone distant, future. But we wanted an interface that would last, at least a few years. Fundamentally, it had to be extensible. This is a common buzzword for software design, of course, so we had two specific goals in mind: it needed to be extensible by renderer writers, so that new techniques could be described by the interface without disrupting the basic structure of the interface; it needed to be extensible by the renderer users, so that users could invent their own techniques without requiring a rewrite of the renderer every time they wanted one little feature.

There are two extremely powerful ways that the RenderMan Interface is extensible by the user. The most obvious is the programmable RenderMan Shading Language. With the introduction of the Shading Language, appearance and material modeling became first class citizens, with a special language as powerful as that provided for shape and animation modeling. The vast majority of the expertise that makes a RenderMan-savvy TD so valuable is intimate knowledge of the Shading Language: both its syntax and its subtle but powerful programming paradigms. As a result, much of the rest of this course will concentrate on exploiting the Shading Language in a variety of unique but compatible rendering systems.

The second nod to extensibility is specific support for an open-ended set of attributes applied to geometric primitives. Most graphics APIs have a fixed set of 5 or 17 or 29 interesting bits of data that you can pass to the renderer (e.g., color, opacity, Phong normal, 4 sets of texture coordinates for each of the texturing pipeline stages), with specific API calls to pass each of these bits of data. RenderMan, on the other hand, has extensible *parameter lists*. Parameter lists are arbitrary length lists of *token-value pairs*. The token of each pair identifies the name and type of the data being supplied. The value of each pair is an array of data of the correct type. Certain common pieces of data have predefined token names, like `"P"` for vertex positions and `"Ci"` for vertex colors. But there is a syntax for defining new tokens, and with them, new and unique bits of data that allow the user to extend and customize his rendering environment.

Thanks to these two modes of extensibility, the RenderMan modeling and shading APIs are nearly the same now as they were 14 years ago, despite great improvements in the capabilities and breadth of application of the RenderMan renderers. This can hardly be said of any alternative graphics system.

1.2.3 Language Bindings

The RenderMan Interface Specification was originally written to be linked directly to modeling programs and make pictures in their windows. For this reason, the first description of the API was described as a C subroutine library. In computer-science-speak, this is known as the *C language binding*. Not long afterwards, it was realized that the most useful way to use a renderer that took hours to create a picture was as an offline batch process, driven by a scene description stored in a file. At that point, *RIB*, the RenderMan Interface Bytestream was developed. RIB is a *metafile*, a transcription of calls to the C API. The most common way to create a RIB file (other than using emacs, of course) is to call the normal C API from a modeling program, but the parameters and data given to each of the calls are just stored in a file.

Because the RIB version of RI and the C version of RI are nearly identical (by design), it is common to give examples in RIB, which is compact and doesn't suffer some of C's inconvenient syntactic idiosyncrasies. The *Advanced RenderMan* book does this, as do these course notes. Take it as read that anything you do in RIB can be done equally well in C. And although none have been officially sanctioned by Pixar, there are a variety of unofficial language bindings to other programming languages and file formats have been publicly proposed. Java, Python and Tcl come to mind immediately. Note to self: while the C binding currently serves as a C++ binding, a true C++ binding would be a welcome improvement to the official document.

1.2.4 Coordinate Systems

In every hierarchical scene description such as the RenderMan Interface, there is a naturally a hierarchical transformation stack, which allows the modeler to place objects relative to parent objects. Each node in this stack defines a *local coordinate system*, which is different from the parent's by the transformations that have been applied to that node. It is common for objects need to refer to various globally-useful coordinate systems, such as the coordinate system that the camera was defined in. These coordinate systems are defined by name, and there are transformation functions which can transform points, vectors and surface normals (differently, remember?) between any two coordinate systems by their names.

The RenderMan Interface predefines several key coordinate systems, so that they are always available for use. These include:

- `world` - the world coordinate system;
- `camera` - the camera is at the origin, looking out over the z axis;
- `object` - the local coordinate system that a primitive was defined in;
- `shader` - the local coordinate system that a shader was defined in;
- `raster` - the coordinate system of pixels on the display.

In a photorealistic scene description, though, it is relatively common for objects to have relationships with arbitrary objects in the scene, such as with lights attached to other objects, multiple cameras in the scene, or objects that reflect them or reflect in them. Distances between objects, relative scale of different objects, or angles of separation from important directions might need to be computed. The RenderMan Interface provides a general system for specifying and querying such outside-the-hierarchy transformation relationships by allowing the user to give a unique name to any local coordinate system at any point in the hierarchical scene description. These are known as *named coordinate systems*.

Once a named coordinate system is established, other objects can reference it by transforming points and vectors into or out of it. A common example is to calculate the distance to a light source by referring to the origin of the named coordinate system where a light was defined. Another example would be to transform a unit vector from one coordinate system into another coordinate system, in order to compare lengths (and therefore scales). Any object in any part of the hierarchy can access a named coordinate system equally well, no parent, sibling or common ancestor relationship is required.

1.2.5 Shaders

Perhaps the most important contribution that RenderMan has made to computer graphics is the RenderMan Shading Language. While the idea of a shading language was not new, the RenderMan Interface was the first to insist that a well-designed language for material description was available as a fundamental feature of a photorealistic renderer.

Shading Paradigm

A photorealistic scene description must includes not only the locations and shapes of all of the geometric primitives in the scene, but must also include some definition of the material properties of those primitives, as well as the locations and attributes of the lights in the scene. The RenderMan Interface describes both of these appearance properties as small programs written in the RenderMan Shading Language.

The RenderMan Shading Language is a C-like language which is specially designed to make shading and geometric calculations easier to write. The basic data types of the language are not the general purpose `int`, `float` and `char`, but rather `point`, `vector`, `color` and `matrix`. The language

has built-in operators to add, multiply, interpolate, and transform these data types, to access texture maps, and to evaluate light sources, since those are the types of operations that are common in a program that computes texture patterns, BRDFs and ultimately object colors.

Programs written in the Shading Language are called *shaders*. Four basic types of shaders can be written in Shading Language, and then used in the scene description:

- `surface` – describes the material properties of an object;
- `displacement` – describes the bumpiness of an object's surface;
- `atmosphere` – describes how light is attenuated as it travels through space;
- `light source` – describes the intensity of light emitted by light sources.

Each of these is written in the Shading Language. The differences between them are based entirely on their intended function in the appearance description, and the effect that that has on which geometric parameters are needed to accomodate that function. A program wouldn't be very powerful if it didn't have parameters, and shaders are no different. Shader parameters can be set in the scene description, to control their behavior on an object-by-object basis.

Since shaders are written in a programming language, there must be some process by which the programs are compiled and loaded into the render for eventual execution. According to the RenderMan Interface Specification, the method that a renderer uses to accomplish this is entirely up to the implementation of the renderer. In practice, essentially all RenderMan renderers have used essentially the same technique. The shader is compiled by a custom Shading Language compiler into *Shading Language object code*, a set of assembly language instructions for a mythical SL CPU. The renderer contains an interpretter for this assembly language, which is able to simulate the SL CPU one instruction at a time. Naturally the compiled object code for one renderer is unique to that renderer (or at least, unique to the compatible products of a single company).

Shader Evaluation

Shading Language programs are evaluated at points on the surface of geometric primitives. The renderer is responsible for deciding which points on the primitive need to be shaded, depending on a wide variety of scene description parameters. Although the user has some control over the renderer by manipulating its global scene description parameters, such as image resolution or fidelity, the user does not directly choose which or how many points on surface will be shaded, nor in which order.

Once the renderer has determined a point on the primitive which it wants to be shaded, it computes a set of "global" geometric values to describe that point to the shader. Obviously the position of the point is quite valuable, as well as the surface normal, the surface tangents, parametric and texture coordinates, etc. These are global in the sense that they are determined from the mathematical description of the geometric primitive itself. Also, any predefined or user-defined vertex data which was attached to the geometric primitive in the scene description is interpolated according to the rules of its class, in order to determine a value specific to the point in question.

Once all of this data is computed, the shaders are executed. Each geometric primitive in the scene description has exactly one surface, one displacement and one atmosphere shader bound to it, in order to describe its visual appearance. In addition, the scene description has any number of light sources, each of which is described by a light source shader. Individual lights can be turned on or turned off for any given primitive, so the number of light source shaders that are bound to an individual primitive can vary. Each shader uses the global data, vertex data and the object-specific values of their parameters to compute their contribution to the appearance of the object.

First to execute is the displacement shader. The displacement shader calculates bump mapping or displacement mapping. The output of this shader is a new position and surface normal for the point. In some renderers, or in some circumstances, displacement maps are not possible (generally

due to “shading after hiding” issues), in which case only the modified surface normal affects future calculations.

Second to execute is the surface shader, which is often coloquially referred to as *The* shader, since everyone knows it is the main shader. The surface shader takes all the precomputed geometric input, plus the potentially modified position and surface normal output by the displacement shaders. Its function is to compute the actual color of the object. The surface shader generally spends most of its time evaluating parameters of the local illumination function, such as diffuse surface color, specular maps, reflection vectors, etc. This step, calculating the material characteristics and patterns visible on the surface of the primitive, is by far the most time consuming part of the entire shading calculation (both for the renderer and for the Shading Language programmers), many days could be spent lecturing on the hows, whys and wherefores of doing this well. In the interest of time, I won’t get do that now.

The surface shader runs the local illumination function on these parameters to determine the outgoing color and object opacity in the direction of view. There are a variety of built-in simple local illumination functions, such as Lambertian diffuse and Phong specular, but most realistic surface shaders nowadays have custom functions.

In order to evaluate local illumination, the intensity of the light arriving at the point must be known. The surface shader’s invocation of illumination functions causes the evaluation of all of the light source shaders that are turned on for the object. Light source shaders use their parameters and global inputs to determine the color, intensity, and incoming direction of light that arrives at the point that the surface shader is shading.

Once the surface shader has completed, there is one more stage. The atmosphere shader’s job is to compute the affect of any atmospheric or volumetric effects that occur between the object and the camera, and it modifies both the color and opacity of the object to take into account any changes that those effects might cause. Often those affects vary with depth.

Having completed the shading of a particular point on a primitive, the renderer may have many more points to shade on the same primitive or on other primitives. Depending on the specific algorithm of the renderer, it might shade each point on the primitive one at a time, or it might shade many points simultaneously in parallel. In either case, the shader is written as though it is processing one point at a time. In Shading Language there is very little access to any information about characteristics of other places on the same primitive or on other primitives. The language supports a very powerful set of derivative operators, so it is possible to compute how global variables, or the results of complex calculations, change in the immediate neighborhood of the point being shaded. However, the language does not have operations for arbitrary queries about nearby or distant surfaces.

That being said, some rendering algorithms do exist that can provide a variety of information about nearby or distant neighbors. Such capabilities are often available in global illumination algorithms like ray tracers, so the language has facilities for tracing rays and probing the environment with them. Rendering algorithms that cannot provide this information simply do nothing if asked to trace a ray.

Once shaders have been run on all the relevant points on the geometric primitives, the colors that have been computed eventually make it into the pixels of the output image. As such, shader writers have complete control over the colors in the output image, and this control comes in very handy as a TD, as many of the speakers throughout the day will attest.

Why a Special Purpose Language?

Nearly 15 years after the design of the RenderMan Interface, a common question is, why have a Shading Language at all? Why not just program shaders in C? There is no definitive answer to this question, because we know that C-based shading extensions to photorealistic renderers can and do exist. However, the RenderMan choice still appears the superior choice to its devotees, for a number of reasons.

The RenderMan Shading Language is a "small language," in that it is a special purpose language with data types and semantic structure designed to solve a particular problem: calculating the color of objects in a photorealistic rendering. Despite its obvious syntactic similarities to C, it is not a general-purpose programming language like C or C++. Operations that are common in shading calculations can be described succinctly, while operations that have nothing to do with shading can't be described at all. This is a conscious choice. While it doesn't have many of the powerful data structures or programming flexibility of a modern general-purpose language, it also doesn't have the semantic complexity of one.

The existence of the Shading Language is also a recognition of the fact that the implementors of rendering systems are computer graphics specialists, not compiler theory professionals, and that the graphics hardware that the compiled code will be running on is similarly not a general-purpose CPU. As mentioned above, in most software RenderMan implementations, the graphics hardware that runs Shading Language programs is not a real CPU at all, but a virtual machine which is implemented as a simulator in the renderer. The basic assembly language of this shading virtual machine is not add, register move, branch if not equal, like a modern CPU. Rather it is transform a point, grab a texture, trace a ray. The power of having these as the lowest-level operators in your assembly language far exceeds the minor programming hassles of not having user-defined data structures or pass-by-value subroutine calls.

Of course, in the 21st century, it may be common to expect that a modern uberlanguage like C++ could subsume the semantics of a Shading Language, providing the same descriptive power without the syntactic limitations. This may be true, from the point of view of the Shading Language programmer. However, in the end, the reason that a small private language is better for shading is the same reason that JavaScript or Tcl or SQL exist. Software systems which contain embedded languages always need to exercise some control over the data structures, breadth of atomic operations, and security features of the language in order to provide a stable, predictable and bug-free environment within which they can run. It is very difficult to crash a renderer by providing a buggy Shading Language program. The same cannot be said for renderers whose shaders have dynamically allocated memory, reference arbitrary memory locations through unchecked pointers, and can make system calls (fire a ray into the Internet, anyone?)

1.3 Scene Modeling

In order to provide a scene description to the RenderMan renderer, one has to model the scene. There are many good modeling packages out there, and most have some form of model translator than can get their data into a RenderMan renderer. But when TDs are chosing the best (the most efficient, best looking, most artifact-free) way of modeling particular objects in the scene, there are many choices and many tradeoffs that they must keep in mind.

1.3.1 Polygonal Meshes

The most common primitive in use in computer graphics, 20 years ago or today, is the polygon. Polygons are easy to understand, flexible in use, and common to a variety of modeling and rendering systems. RenderMan breaks polygonal models down into 4 categories, based on the complexity of the topology of the model.

First, RenderMan classifies polygons as being either *convex* or *concave*. Convex polygons are, loosely, ones which have no indentations of any kind in their boundary. More specifically, a line drawn between any two points on the edges of the polygon stays inside the polygon and never crosses an edge. Concave polygons are anything else: polygons with indentations or protrusions in their boundary or polygons with holes in their interior. RenderMan calls concave polygons *general polygons*. The distinction between convex and concave is done only for efficiency considerations

inside the renderer, as many rendering algorithms are faster if it is known ahead of time that the polygons being processed are convex.

The second division of polygon types is a packaging consideration. RenderMan allows polygons to be described one at a time, or in groups with shared vertices (aka a *polyhedron* or *polygon mesh*). A polygon mesh is defined as a set of polygon *faces*, each of which has any number of vertices. The vertices are identified by indices into a common table of vertex positions, so that faces that share a vertex can duplicate the index and do not have to duplicate the vertex data itself. This is for compactness and efficiency of description, since the faces of a polygon mesh typically reuse a vertex many times, so it is more efficient in space and processing time to specify it once and share it.

So, in summary, RenderMan supports four types of polygonal primitives, lone convex polygons, lone general polygons, polyhedra made of convex polygons, and polyhedra made of general polygons.

Polygons have certain advantages for interactive hardware renderers, such as their straightforward description and simple equations that make hardware implementation compact and fast. However, for photorealistic renderings, they are very problematic — there are a number of unhappy *polygonal artifacts* that degrade our images.

To begin with, there is the problem of discontinuity. At the edge where two polygons meet, the surface has a sharp bend. This bend is known as a C^1 discontinuity, which means that the slope of the surface changes abruptly. This leads to two problems. First, the silhouette has corners where it abruptly changes direction. Second, the surface normals of the primitive are all the same on one facet, and then abruptly change on the next facet. This means there are shading discontinuities. The solution to these surface normal direction discontinuities is, of course, *Phong interpolation*, which smooths out the obvious shading discontinuities. But generally this is only a partial fix, since the interpolated normals themselves often don't quite look right, behave unexpectedly at silhouette boundaries, and cause problems for bump and displacement mapping.

The next problem is one of fixed resolution. The set of vertices that describe the polygon mesh are generally appropriate for looking at the object at a particular scale, or size on screen. If the object is much smaller than that size, the polygons are individually very very small, and cannot be rendered efficiently. The amount of data that it takes to describe the object, and the amount of time that it takes to render the object, is not proportional to the importance of such a small object.

If the object is much bigger than the appropriate size, then the individual polygons are very large on screen, which accentuates the polygonal discontinuities and robs the image of its believability. As we will see, there are other primitives that can fill in with curves between vertices, and make it look smoother and more continuous as the object gets larger on screen. You simply can't fill in a polygonal mesh.

Finally, getting good smooth shading on a polygonal mesh is difficult. On a polygonal mesh there is no smooth, continuous 2-D coordinate system that covers the entire primitive. There might be small 2-D coordinate systems that cover pieces of the mesh, such as one of the triangular facets, but generally speaking there is no single equation that covers the whole thing. Therefore, applying texture maps or other shading features that span many facets in a smooth and unkinked way is not straightforward. Moreover, every geometric quantity on the primitive, not merely the surface normals, suffer from the discontinuities at the polygonal edges. So for example, if your shading equation relies on tangents, or just the scale of the local 2-D coordinate system (for texture filtering), these quantities will also abruptly change at polygon edges, leading to more artifacts.

The advantage of polygonal meshes, of course, is that it is extremely easy to create arbitrary shapes with them. Digitization technology is very advanced, making it easy to acquire and mesh together large numbers of vertices. They can handle complicated topologies, such as protrusions, holes or even Klein bottles just as easily as they can handle simple topologies like planes and spheres. Moreover, when it comes to animation, it is easy to move and deform polygonal objects into desired positions because you are working with the vertices directly. The animation software doesn't have to concern itself with the odd mathematical equations of the parametric surface, and guess how to

modify the parameters to get the right shape.

1.3.2 Parametric Primitives

Parametric geometric primitives are primitives which can be described by mathematical equations with *parameters*. That is, an equation of the form $P = f(a, b, c)$ In 3-D computer graphics, we are generally interested in parametric primitives whose equations have 2 parameters, which we conventionally denote `u` and `v`. Typically we are interested in a subset of the range of the equation, for example, all the points where `u` and `v` range between 0.0 and 1.0.

The primitive generated by these equations are two-dimensional (it forms a surface, not a line or a volume), and a wide variety of useful and important geometric primitives are parametric surfaces. For example, a plane has a simple parametric equation:

$$P = u \cdot \bar{V_0} + v \cdot \bar{V_1} + P_{origin}$$

A sphere (that is, the outside shell of a sphere) has a very simple parametric equation:

$$\begin{aligned} P_x &= \sin u \cdot \sin v \\ P_y &= \cos u \cdot \sin v \\ P_z &= -\cos v \end{aligned}$$

In RenderMan, all of the quadric surfaces, patches and patch meshes, and NURBS are parametric surfaces. Parametric surfaces are good surfaces for rendering for three reasons. First, the parametric equation makes it very easy to locate and draw all of the points on the surface. We simply use a pair of loops over the two parameters to generate as many points as we want. If our algorithm needs to find additional points on the surface that lie between two points we already have (for example, finding the midpoint of an arc), we just generate the point in the middle of the parametric range. Such points are always correctly on the surface, and we can generate as many as we need no matter how close we get to the object. Recall that on a polygonal primitive, we could only interpolate vertices with straight lines.

Second, the parametric equation makes it very easy to compute other valuable geometric quantities about the surface that we need for shading. For example, we generally need to know the surface normal for our local illumination equation, and probably need the surface tangents for antialiasing purposes. The derivatives of the parametric equation of the surface give us the tangents directly and exactly. The cross product of the tangents gives us the normal. Nothing could be simpler.

Third, parametric surfaces have a natural 2-D coordinate system drawn on them, thanks to their two parameter equations. You can grid the surface with lines that have equal values of either `u` or `v`, and these lines are called *isoparams*. This 2-D coordinate system is very handy for shading purposes, most obviously for texture mapping. You just put the pixel (`s`, `t`) of the texture map onto point (`u`, `v`) of the parametric surface. Even if your texture is not being laminated directly onto the parametric primitive, the fact that the smooth, continuous, unkinked 2-D coordinate system exists makes most shading operations easier.

The big problem with parametric surfaces is that they are always rectangles. If you want to create more complicated shapes, you must stitch together a bunch of parametric primitives into a *mesh*, a "quilt" of rectangles that wrap around non-rectangular features of the object. This usually means breaking up our handy continuous 2-D coordinate system into a less convenient piecewise continuous coordinate system. Moreover, for a lot of shapes, such as objects with protrusions (like fingers) or with holes, covering the surface with this mesh of rectangles is not always easy. It is often the case that these meshes suffer from discontinuities or cracks at the edges, or that the rectangles end up being vastly different sizes in order to accomodate the shape of the object.

1.3.3 Subdivision Meshes

Recently, the computer graphics community has been looking to a hybrid technology to solve the problems of polygonal meshes without introducing the topological restrictions of parametric surfaces. Interestingly, the technology we were looking for was under our noses for nearly 20 years, but it was ignored and forgotten until it was rediscovered in the late 1990s by researchers at the University of Washington. This magical new geometric primitive is known as the *subdivision surface*. (In RenderMan, we prefer the term *subdivision mesh* in order to avoid confusing the word "surface" with the shader concept of a "surface".)

The easiest way to think about subdivision meshes is that they are polygon meshes with rounded corners. Imagine a polygon mesh that you dropped into a rock tumbler, and it came out all smoothed out and shiny. Because of this, they have a lot of the advantages of polygon meshes, but with many of the defects of polygon meshes removed. First, structurally, they are very similar to concave polygon meshes. You define a subdivision mesh as a group of faces, each of which has any number of vertices, and the vertices are shared between adjacent faces. Because of this, you can model any topology with subdivision meshes that you can with polygon meshes. They are as easy to digitize and animate as polygon meshes. And they can be easily converted to and from polygon meshes in modeling packages that are behind the times and don't handle them directly.

Second, due to their automatic smoothing properties, they don't suffer from polygonal artifacts the way that normal polygon meshes do. There is no discontinuity of tangents or surface normal at edge or corners, and there is no straight lines on silhouettes. Phong interpolation of surface normals is not necessary, since they have beautifully smooth normals already. They don't have discontinuous texture coordinate sizes across edges, either.

They do still suffer from the problem that their texture coordinates themselves change across the edge between two faces. This problem, as with polygon meshes, is a direct result of their general topology, since they can model objects which cannot be "unfolded" into a rectangular 2-D plane like a parametric patch can. However, like polygon meshes, user-defined `s` and `t` coordinates are often attached to the vertices to solve this problem.

RenderMan's subdivision meshes have an additional feature over basic subdivision meshes, known as *creasing*. Every edge or corner point can be given a parameter which describes just how smooth it should be. With this control, it is possible to add back the hard edges or corners that polygons have, or to go somewhere in-between (like a tightly beveled edge on a wooden table). Recent releases of Maya and other modeling packages have similar creasing or pinching controls which are similar to those in RenderMan.

1.3.4 Procedural Primitives

Another powerful modeling concept that is fully supported by RenderMan is that of *procedural primitives*. The idea behind a procedural primitive is to break away from the use of predefined, modeled geometry to describe the objects in a scene, and to let a program create geometry on the fly as the scene is being rendered. For example, rather than have a person individually model the thousands of trees in a forest, a program could be written that creates tree models, and that program could be run by the renderer to get as many trees as necessary. There are two big advantages of letting a program generate complex models. First, the program can take parameters which alter the model generation, so that each individual in the group looks slightly different. Such parameterized models can create an infinite variety of similar but different individuals, which looks much more natural and photorealistic than endless carbon copies of the same hand-carved model. Second, because the renderer asks for the model data, rather than having the data force fed to it, it can delay asking until it has determined that the model is relevant to the scene. In modeled sets and environments with large numbers of objects, it is not uncommon for many of these to be off-screen or otherwise irrelevant to making a particular image. If the renderer knows that, it needn't ask for the data, saving time in

both model generation and rendering.

Interestingly, the renderer rarely needs as much data as procedural primtives are capable of churning out. A program can create a whole forest, but we only see a few of the trees up close, and a lot more trees really far away. Fortunately, procedural primitives can take that into account, too. Since the renderer asks for a model when it is relevant to the image being rendered, one of the parameters to the model generation can be its eventual size on screen. In RenderMan, we call this the *detail size*. Trees with large detail sizes will be scrutinized by the audience, so the program makes a very, er, detailed, version of the tree. Trees in the distance have small detail sizes, and the program can know that a simple approximation is perfectly acceptable. The request from the renderer to the procedural primitive program includes the detail size as one of the parameters, so that the program can tailor the output to the particular use.

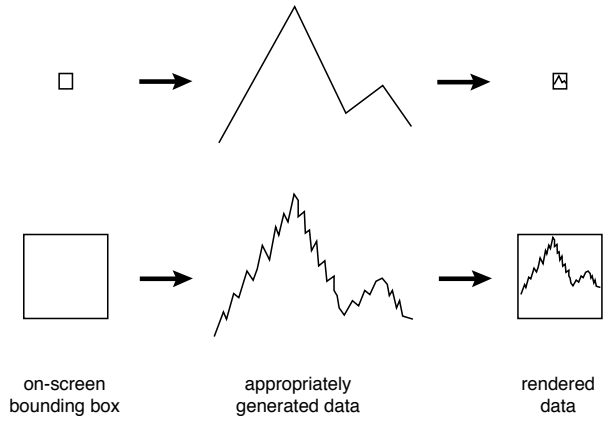


Figure 1.6: A procedural primitive at differing detail sizes.

RenderMan does not provide a Procedural Modeling Language with which to write procedural primitives, in the way it provides a Shading Language to write materials. Instead, RenderMan renderers rely on connecting to external programs written in normal programming languages. Different renderers will create these connections in different ways, such as statically linking the procedural primitive libraries into the renderer, dynamically linking those libraries into the renderer executable at runtime, or using interprocess communication to transmit data over channels.

1.3.5 Vertex Data

All of the modeling primitives available in RenderMan, whether they are polygonal meshes, parametric meshes, or something more exotic, are fundamentally described by a set of vertices — points in 3-D.

Historically in computer graphics, renderers have found it useful to have more information at each vertex than just the position. For example, when shading a polygon mesh, it is generally valuable to have a Phong normal at every vertex which can be interpolated across the surface of the facets for better shading. Similarly, it is quite common to attach *texture coordinates* `s` and `t` to each vertex, or apply a color to each vertex.

One of RenderMan's powerful extension mechanisms, the parameter list, allows us to break free of a restrictive, predefined set of such data, and permit us arbitrary data attached to the primitives. In

RenderMan, it is trivially easy to define a new piece of data and attach it to the vertices of a primitive. This data is carried along with the primitive throughout its travails through the rendering pipeline, and in particular, is available to the shaders that are attached to the object. It is quite common for those shaders to compute shading based not merely on the standard, built-in, geometric quantities like the surface normal and tangents, but also upon these custom-designed per-vertex parameters. We call these parameters *vertex data*.

RenderMan's mechanism for creating vertex data is very general. Syntactically, there are eight *types* of data which can be defined:

- `float` – a single floating point number;
- `color` – an RGB triple;
- `point` – a 3-D position;
- `vector` – a 3-D direction vector;
- `normal` – a 3-D surface normal vector;
- `string` – a quoted character string;
- `matrix` – a 4×4 transformation matrix;
- `mpoint` – a position represented as a coordinate system (used only on implicit surfaces).

Fixed length arrays of any of these types can also be defined. There are five levels of granularity, or *storage classes*, which allow the data to be applied and interpolated differently across the surface of the primitive:

- `constant` – exactly one value, identical over the whole primitive;
- `uniform` – one value per facet, constant over the facet;
- `varying` – one value per vertex, interpolated parametrically over the facets;
- `facevarying` - one value per vertex per facet, interpolated parametrically over the facet, but discontinuous between facets (newly invented for *PRMan* Release 10);
- `vertex` – one value per vertex, interpolated identically to position.

1.3.6 Level of Detail

In realistically modeled scenes, the geometric and shading complexity is enormous. Photorealistic images requires the extremely fine detail that is found in nature, or the illusion will be broken. Dirt, smudges, tiny paint chips, fabric wearing patterns, the list is endless. However, not every object will be rendered in closeup, so not every object in the scene needs to have the same high level of detail. Objects seen in the background may be so small, or poorly lit, or even out of focus, that they only need cursory treatment to be "good enough". Doing so will have great advantages in rendering time, since giving the renderer large amounts of geometry and shading information to process when just a little does the trick is, well, wasteful, of time and memory.

However, sadly, it is not always easy to tell ahead of time which objects will be in the foreground of a scene and which will be in the background. Camera angles change with the whimsy of the photographer, objects move in animation, lighting may change, etc. It is easily possible that objects that are close in one part of an animation are in the distance in another part of the animation. It seems clear that for such objects to render efficiently in all cases, multiple versions of the object would be useful.

One approach to deploying these varied versions is completely manual. When the scene is built and layed out for the camera, highly detailed versions of the foreground objects are used, while rough or simple versions of unimportant background objects are used. As the objects and the camera move through the scene, object versions can be manually swapped out at the appropriate time to make sure that the right version is in front of the camera at all times.

Of course, this is tedious and somewhat error prone. It is hard to guess exactly the right version without seeing the rendered result, and managing the swapping must be done with great care.

However, the renderer "knows" which objects are close and which are far, so in theory it is quite capable of knowing which versions of each object to use. If the renderer could swap them in and out automatically, it would be a lot easier for the photographer.

The RenderMan Interface supports this requirement with the concept of *level of detail*. For each object in the scene description, the modeler can specify a variety of different representations of that object. With each representation of the object comes a metric which identifies when that representation should be used. The renderer examines the metrics attached to each representation, and chooses from them the one that is best to use in that particular rendering. As the camera or objects move, the renderer may chose a different representation in the next frame. The metric is a measure of importance we've seen before — the size on screen, a.k.a. detail size.

It all works like this: for each object in the scene, the modeler specifies a box in the world to use as a ruler. The scene description then contains any number of different representations of the object, and with each one specifies a range of detail sizes of the ruler for which this description is relevant. When the renderer encounters this object in the scene description, it examines the ruler and computes how large it actually is on screen. This provides a number for the actual detail size of the ruler. Then, the renderer compares that to the various detail size ranges of the descriptions, and renders the relevant one. All of the irrelevant descriptions are merely thrown away.

As it turns out, if you always pick exactly one description for every object, you get artifacts. In particular, if you watch an object shrink as it slowly drifts away, you can see a sudden visual change at the point in the animation where the renderer decides to change representations. This is called a "pop", and is very distracting and unhappy all around. The solution for this is for the renderer to fade in and out representations, so that there is never a pop. For this to work, the detail ranges of representations must overlap, so that the renderer has transition ranges where two representations are relevant, and the renderer can blend or fade between them.

In the RenderMan Interface scene description, you are allowed to use different geometry and different shaders in your level of detail representations. A high detail representation might be a complex subdivision mesh with a primo shader that uses displacement maps and subtle texturing. A midrange representation might be a very blocky subdivision mesh with a plastic shader that texture maps on a color and specular coefficient. A low detail representation might be a square with a photograph of the object textured on. Even though the object entirely changes both geometric topology and shader types, a fully-featured RenderMan renderer will make a smooth transition between the different representations.

Building sucessful level of detail representations and managing their transition ranges for "invisible fades" is a bit of an art, which will be discussed in greater detail in the afternoon sessions.

1.3.7 Modeling vs. Shading

In many animation and special effects studios, the team or individual who is responsible for creating the models in the scene is responsible for both their shape and their appearance. There is a natural tension between modeling shape and modeling appearance for photorealistic scene descriptions. There are two reasons. First, the appearance modeler generally needs some sort of easily parameterized, discontinuity free, smooth and elegant primitives onto which they can apply their shaders. Attractive shading is heavily dependent on procedural or image-based texture maps, which are difficult to apply if the primitive has unusual parameterization or discontinuous texture coordinates. These requirements place restrictions on the choice of geometric primitives and modeling techniques that can be used to create the shape, and it is not uncommon for the simplest or most elegant way to make a shape in modern modeling packages to create serious problems for shading.

Second, there are many situations where a particular feature of the intended object can be realized either by modifying shape or by enhancing the shader, and deciding which way is "best" is often a difficult choice. A hole can be put into an object by rebuilding the mesh and creating a geometric hole, or by modifying the shader to make a part of the existing geometry transparent. Bumps and

pits on an object's surface can be made by undulating the geometry, or by writing a displacement shader. I have been in meetings where technical directors jokingly suggested that every prop in the show should just be spheres or cubes, and the shader writers would be responsible for making the spheres look like cars, trees, buildings, tables, doorknobs and people. It seemed laughable at the time, but as you'll hear this afternoon, it turns out this is not always such a bad idea.

In fact, every large studio has modeling studs who can easily create holes and bumps, and spiraled screws with Philips heads and rock strewn landscapes and amazingly detailed mechanisms that'll look completely realistic with even the simplest metal shader attached. They also have shading studs who can create the same holes and bumps, and screws and landscapes and mechanisms, from spheres and planes by using amazingly subtle displacements, procedural shading tricks and custom illumination models. There is rarely only one solution that will work. The question that art directors and modeling supervisors must answer is what's the cheapest way to get a look that is "good enough", given the size, importance and screen time of the prop, and the relative complexity of one side's requirements on the other side.

1.4 Life in CGI Production

The rest of this course is going to get beyond the background material, terminology review and history lessons, and deal with the real reason that everyone is here today — to learn the ropes. To understand the practices, techniques, methodologies, tricks that working animation and special effects TDs apply every day in their pursuit of the perfect on-screen image. We haven't written the story, we didn't design the look of the universe, we don't do the animation or lend voice to the characters. We just make it all real. We make the images – 24 per second, 86400 per hour – that make the audience believe that what they see on-screen is all really happening.

Before the other speakers get started on the technical details of various tasks, techniques and operating paradigms, I want to provide a little background on how TDs work — the very foundations of the practice of the Lore of the TDs.

1.4.1 Photosurrealism

For more than 20 years, one of the stated goals of the computer graphics research community has been to solve the problem of making truly *photorealistic* images. We have strived to make the CG image, as much as possible, look exactly like a photograph of the scene, were that virtual scene to actually exist. Much of this work was done very specifically so that CGI could be used in films. Solving such problems as accurate light reflection models, motion blur, depth of field, and the handling of massive visual complexity was motivated by the very demanding requirements of the film industry.

Movies, however, are illusions. They show us a world that does not exist, and use that world to tell a story. The goal of the filmmaker is to draw the audience into this world and to get the audience to *suspend its disbelief* and watch the story unfold. Every element of the movie — the dialog, costumes and makeup, sets, lighting, sound and visual effects, music, and so on — is there to support the story and must help lead the viewer from one story point to the next. In order for the audience to understand and believe the movie's world, it clearly must be realistic. It cannot have arbitrary, nonsensical rules, or else it will jar and confuse the viewers and make them drop out of the experience. However, movie realism and physical realism are two different things. In a movie, it is realistic for a 300-foot-tall radioactive hermaphroditic lizard to destroy New York City, as long as its skin looks like lizard skin we are familiar with. Some things we are willing to suspend our disbelief for, and some things we are not!

There are two people who are primarily responsible for determining what the audience sees when they watch a movie. The director is responsible for telling the story. He determines what message

(or "story point") the audience should receive from every scene and every moment of the film. The cinematographer (cinema photographer) is responsible for ensuring that the photography of the film clearly portrays that message.

Over time, filmmakers have developed a visual language that allows them to express their stories unambiguously. This language helps the filmmaker to manipulate the imagery of the film in subtle but important ways to focus the audience's attention on the story points. Distracting or confusing details, no matter how realistic, are summarily removed. Accenting and focusing details, even if physically unrealistic, are added in order to subtly but powerfully keep the audience's attention focused on what the director wants them to watch. The job and the art of the cinematographer is to arrange that this is true in every shot of the film. The result is that live-action footage does not look like real-life photography. It is distilled, focused, accented, bolder and *larger than life*. In short, film images manipulate reality so that it better serves the story.

Our computer graphics images must, therefore, also do so. Perhaps even more than other parts of the filmmaking process, CGI special effects are there specifically to make a story point. The action is staged, lit, and timed so that the audience is guaranteed to see exactly what the director wants them to see. When a CG special effect is added to a shot, the perceived realism of the effect is more influenced by how well it blends with the existing live-action footage than by the photorealism of the element itself. What the director really wants are images that match as closely as possible the other not-quite-real images in the film. They are based in realism, but they bend and shape reality to the will of the director. Computer graphics imagery must be *photosurrealistic*.

In a CG production studio, the role of the cinematographer is filled by the TDs, whose jobs include geometric modeling, camera placement, material modeling, lighting, and renderer control. Computer graphics gives the TDs a whole host of new tools, removes restrictions and provides an interesting and growing bag of tricks with which to manipulate reality. Jim Blinn once called these tricks "The Ancient Chinese Art of Chi-Ting" Technical directors now call them "getting work done."

Altering Reality

One of the best cheats is to alter the physics of light. This technique is based on the observation that the movie viewer generally does not know where the lights are located in a scene, and even if he does, his limited 2D view does not allow him to reason accurately about light paths. Therefore, there is no requirement that the light paths behave realistically in order to ensure that an image is believable.

This situation is exploited by the live-action cinematographer in his placement of special-purpose lights, filters, bounce cards, and other tricks to illuminate objects unevenly, kill shadows, create additional specular highlights, and otherwise fudge the lighting for artistic purposes. Everyone who has ever seen a movie set knows that even in the middle of the day, every shoot has a large array of artificial lights that are used for a variety of such purposes.

However, the live-action cinematographer, despite his best tricks, cannot alter the physics of the light sources he is using. Computer graphics can do better. Creating objects that cast no shadow, having lights illuminate some objects and not others, and putting invisible light sources into the scene near objects are all de rigueur parlor tricks. Things get more interesting when even more liberties are taken. Consider, for example,

- a light that has negative intensity, and thus dims the scene
- a light that casts light brightly into its illuminated regions, but into its "shadowed" regions casts a dim, blue-tinted light
- a spotlight that has at its source not simply 2D barndoors to control its shape, but instead has full 3D volumetric control over the areas that it illuminates and doesn't illuminate, and at what intensities

- a light that contributes only to the diffuse component of a surface's bidirectional reflection distribution function (BRDF) or only to the specular component

- a light that has independent control over the direction of the outgoing beam, the direction toward which the specular highlight occurs, and the direction that shadows cast from the light should fall.

In fact, such modifications to light physics in CGI are so common that the production community does not think of them as particularly special or noteworthy tricks. They are standard, required features of their production renderers. Other, even more nonphysical cheats, such as curved light rays, photons that change color based on what they hit, lights that alter the physical characteristics of the objects they illuminate, or even lights that alter the propagation of other lights' photons, are all useful tricks in the production setting.

Another interesting cheat is to alter the physics of optics, or more generally all reflection and refraction, to create images with the appropriate emphasis and impact. Much of the trickery involving manipulating optical paths is descended, at least in part, from limitations of early rendering systems and the techniques that were invented to overcome these limitations. For example, for over 20 years, most CGI was created without the benefit of ray tracing to determine the specular interreflection of mirrored and transparent objects.

As before, the limited knowledge of the viewer comes to our rescue. Because the viewer cannot estimate accurately the true paths of interreflection in the scene, approximations generally suffice. In place of exact reflection calculations, texture maps are used. Use of environment maps, planar reflection maps and refraction maps ratyer than true ray tracing lead to slight imperfections. But, experience has shown that viewers have so little intuition about the reflection situation that almost any approximation will do. Only glaring artifacts, such as constant colored reflections, inconsistency from frame to frame, or the failure of mirrors to reflect objects that touch them, will generally be noticed. In fact, experience has shown that viewers have so very, very little intuition about the reflection situation that wildly and even comically inaccurate reflections are perfectly acceptable. For example, it is common for production studios to have a small portfolio of "standard reflection images" (such as someone's house or a distorted photo of someone's face) that they apply to all objects that are not perfect mirrors.

Even more nonphysically, viewers are generally unaware of the difference between a convex and a concave mirror, to the director's delight. Convex mirrors are generally more common in the world, such as automobile windshields, computer monitors, metal cans, pens, and so on. However, mirrors that magnify are generally more interesting objects cinematographically. They help the director focus attention on some small object, or small detail, onto which it would otherwise be difficult to direct the viewers' attention. Unfortunately for the live-action cinematographer, magnifying mirrors are concave and are extremely difficult to focus. But this is no problem in CGI. Because the reflection is generally a texture map anyway (and even if it is not, the reflected rays are under the control of the technical director), it is quite simple (and even quite common) for a convex (minifying) mirror to have a scaling factor that is not merely arbitrarily independent of its curvature, but actually *magnifying* — completely inverted from reality.

Similarly, refractions are extremely difficult for viewers to intuit, and they generally accept almost anything as long as it is vaguely similar to the background, distorted, and more so near the silhouette of the refracting object. They typically are blissfully unaware of when a real refraction would generate an inverted image and are certainly unable to discern if the refraction direction is precisely in the right direction.

The goal of the CGI image, and therefore the goal of the TD, is to accentuate the action and clearly and unambigously present the story point. Only secondarily is it to create a beautiful artistic setting within which to stage the action. Film blasts by at 24 frames per second, and the audience doesn't have time to look for tiny flaws, inconsistencies with known physics, or minor continuity

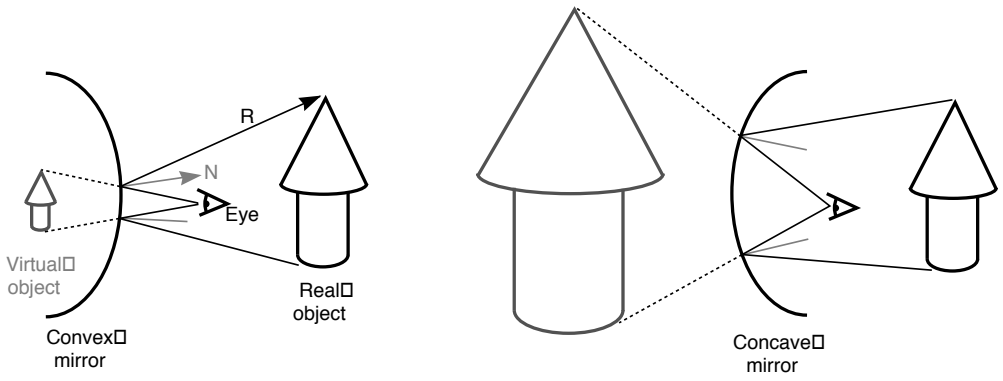


Figure 1.7: Looking at a convex and concave mirror.

problems. If the image looks real, in the Hollywood sense of the word, if it is appropriately photo-surrealistic, then the TD has won.

Long live the Art of Chi-Ting!

1.4.2 Special Effects Shaders on Custom Geometry

A special effect is any trick that adds something visual to a scene that wasn't there when it was shot normally on film. It might be as simple as filming a miniature version of a vehicle, or as complex as adding a computer generated actor to a shot. In an all-CGI movie, you could say that the entire film is one large special effect, but the Academy of Motion Picture Arts and Sciences has decided that this is not considered to be the case. Well, that's fine with me, because I consider a special effect in an all-CGI film to be a trick that adds something visual to a scene that wouldn't be there if it was rendered with all the normal sets, props and characters using their standard shaders.

One very common case for computer generated special effects is rendering the whispy and etherial, things that add visually to the image, but don't have a hard surface made of a conventional material. RenderMan renderers require, however, a geometric primitive with a shader attached, for anything they are going to compute and render. So, TDs create custom geometry with effects shaders. These pieces of geometry will not, in the final image, be identifiable as themselves, in the sense that you will see a sphere sitting there with a texture map of fire on it. Rather, the geometry is merely a mechanism for the effects shader writer to get his desired colors into the specific pixels that need to be covered by the effect.

For example, a bright light source might be surrounded by an aura or glow, caused by the light illuminating particles in the air near the light, or perhaps caused by diffraction effects in the eye of the viewer or lens of the camera. In either case, these effects cause colors to appear in the image in places where there is no geometry to be shaded. So, we add some geometry to hold the shading effect. A sphere around the light source which is not lit in the conventional way (that is, has no local illumination model), but instead merely puts color directly into the pixels, works nicely. Conventionally, it would be mostly transparent, and would fade to obscurity with on-screen distance from the light source.

Another example are clouds, smoke and explosion effects. Geometrically such objects would be impossibly complex to model. Instead, they rely on shader tricks. Typically the models for

these phenomena are extremely simple – some spheres or planes – which are animated by a particle system or some other dynamic animation system. The shaders again rely on transparency and special illumination models to make them look less like surface primitives and more like volumes, and to take direct control of their contributions to the pixels on the screen.

1.4.3 Layers

The first rule of comping is *separate everything into layers*. The first rule of writing shaders is *do everything in layers*. The first rule of building a believable universe is, well, you get the idea.

The complexity of nature is a very difficult thing to simulate with mathematical equations and little programs. The reason is because there is far too much going on simultaneously to grok all at once. Nature uses a wide range of disperate processes to create and modify the shapes and appearances of everything we see. It should not be surprising, then, that well written shaders and special effects rely on the same diversity of process to simulate those shapes and appearances.

When developing a look, think about things from the outside in, starting from the largest basic features and working your way down towards the most subtle. Don't try to get subtleties worked in in the early stages of development, because fixing bugs in the larger features will probably break the underpinnings for the subtle work, and you'll have to do it over again from scratch. Yes, it may seem boring and slow to do it that way, because success is measured by getting a cool picture, and working on subtle effects in a vacuum gets cool pictures fast. But sadly, those cool pictures rarely look right outside of their vacuum. Getting the "right" cool picture will go faster if you have a strong foundation to build subtlety upon.

Consider if we are actually looking at multiple separate things that each can be handled independently and then combined. Nature does this a lot. An asphault road has the asphault itself as the main visual element, but it is also covered with a variety of other substances which make it believable as a road. There's paint and Botz Dots dividing the lanes, rain water collecting in pools, spilled gasoline and oil floating on the pools reflecting iridescently, dirt and grass clippings which blew in from the median strip, litter and trash thrown out by reckless teenagers, tire marks from recent skids, bits of broken headlight glass from the accident, a little pile of dust from the burned out warning flare, and of course the occassional dead armadillo has left his mark. These are all layered upon each other in the real world, and so should be layered upon each other in the CG world as well.

But each of these layers itself is a temporal layering of a variety of processes that affect their appearance. The asphault was slick black pebbles of tar when laid, but over time it has sagged and heaved, cracked and pitted, worn smooth in the tire lanes and accumulated tire stains there, but weathered to a faded grey between them. A smart TD will find ways to build up his appearance by layering these effects as well. Having independent control of these layers makes it easier to satisfy the whim of the director when he wants more tire stains but less smoothing.

And every layer itself will have a variety of characteristics which define them, and a variety of effects on the overall appearance they create. The two tire lanes in each traffic lane are not straight lines, but seem to weave a little. They are not uniformly colored, but are generally darker in their centers (although the actual appearance maybe is of overlapping stripes). The tire stains obviously affect the color of the road surface, but also affect the diffuse and specularity of the underlying tar. They also clearly affect the bumpiness of the tar, and also seem to be correlated to the long thin cracks as well. And those armadillos tend to be found just outside the stripes. Why is that?

Consider the layers, and the layers within layers. Even if you don't understand how nature made something look as it does, at least try to see what features correlate with what other features, and what features are independent of each other. Don't make the mistake of thinking that example you're looking at is the prototypical object to be emulated perfectly, because the director wants lots of them and they all have to look different.

1.4.4 Bang for the Buck

As a TD working in feature film production, television production, video game production, or CD/ROM production (oops, I guess not), one single factor will rule your life more than any other. You are behind schedule. You just *are*. You were behind schedule when before you even started, the deadline has been moved up for marketing reasons, even if your supervisor thinks you can't finish in time you still cannot get more time because another department is waiting for you to finish the model, the director needs to buy off on it tomorrow morning before the big meeting, and, oh, by the way, can you get this other high priority fix done before the end of the day?

For this reason, the seasoned and skillful TD understands the art of always getting the most bang for the buck. The basic rules are: don't do anything from scratch; don't twiddle too much; don't do it all yourself.

First of all, very few objects in the universe are totally unique and unlike anything else in the world. Whenever you are starting a new project, see if there is something that already works which is a good starting point. Gee, that leather couch I need to do has a lot in common with the shrivelled orange that I did last week. Since you're doing things in layers, it might be that just one of the layers is helpful. In normal software engineering, code reuse is pretty common, but in Shading Language programming it seems to be less so. Fight that. One of the most productive shading TDs I know was able to shade dozens of objects "in record time" because she had a private library of half a dozen modular, generic material shaders lying around that could be tweaked to get a variety of really neat results. In the afternoon session, speakers will discuss the needs and techniques for making modular shader libraries. Soak that stuff up.

Second, don't get too detailed and finicky. Our shaders are extremely complex beasts, and they have lots of knobs. There is a natural tendency to sit hunched in front of your workstation and create hundreds of pictures with tiny changes in the knobs, comparing them at the pixel level to decide which is "best". Don't succumb to the siren's call of perfection. It is a waste of precious time for two reasons: (A), the director doesn't want what you think is best – he wants what *he* thinks is best; and (B), if you can barely see the differences staring carefully at your monitor, and are even diffing images to compare them, then there is no way that the differences will survive the blurring, color mangling and other image destruction which film out and conversion to NTSC has in store. Take a stab at reasonable values for your knobs, and be prepared to make adjustments. Give a reasonable number of layers, and be prepared to add more controls later. Iterate until time runs out.

Third, don't carry the weight of the world on your shoulders. Don't do it all yourself. There's lots of code to steal, from previous productions, books, Web sites and course notes. Use it! There are many times when giving a texture layer to a digital painter and having them create something will be vastly better than anything you can reasonably code up procedurally. Take advantage of their skills! Let the director make the hard decisions on which knob settings look best. Give the director 4 variations of grey on the wing tip, and 3 different bumpinesses around the nostril hole on the beak. This is called *bracketing*. The more pedestrian work that someone else does for you, the less mindless and unsatisfying work you have to do yourself, and the more time is left for you to plus it.

1.4.5 Plussing and Leaving your Mark

One of the most ridiculous verbified adjectives in the movie business is *to plus*. Plussing is taking something that exists, and making it better. If there is a joke, make it funnier. If there is an emotional story point, make it more heartwrenching. If there is a budget, make it bigger.

As TDs, one of the best ways to get job satisfaction, high fives all around, and recognition of your potential as a future Effects Supervisor is to plus a prop, shader or effect. Sure, the art director told you what he thought it should look like, but he doesn't know nearly as much about what is and what is not possible in CGI as you do. Besides, he probably told you in very specific terms like

"I want it scary, with lots of ooze, a real Alien feel, only with more heart. A kind of Incredible Dissolving Man meets ET kind of look . . . but pink."

Now, really, you cannot just go off on your own artistic self-expression and make an Incredible Shrinking Woman meets Pokemon, in pink, because you are *not* the Director, and you *will* get in trouble. However, you can plus it by going the extra mile, making the ooze slightly more refractive and slimey, making the blood vessles throb gently under the skin, and giving the director four different saturations of pink so that he can feel like he's making decisions.

One important thing to remember, however, is that very likely you will never see these tweaks on the screen. They are almost always subtle enough that editors and color timers and bad conversion to NTSC will hide them. *Do not be disappointed* – they weren't for the audience anyway. They were for you and your friends and for showing at Users Group Meetings at Siggraph, and they look great on your reel.

Another TD passtime is *leaving your mark*. This is not the same as *pissing on everything*, which is a sure way of getting your peers to hate you. Leaving your mark is finding a way to subtly, inobtrusively, putting your name into an image. For example, a racing car you've modeled has a bunch of decals with fake company names plastered all over it. Gee, one of the companies just happens to be named after you! Imagine that!

Again, you have to be subtle. You can't go change the name of the hotel from "Hotel Bates" to "Hotel Jonathan Christiansen" without someone noticing who is in a position to fire you. If you're in good with the art director for giving him lots of different pinks to choose from, you might actually get permission to stick your name as the author of one of the books on the shelf. Its all about finding a place where it doesn't matter. Be aware, though, that some directors have been known to take even innocent fun with a very ill humor. Know your limits. If you get fired because you scrawled your name in the specularity map of the left engine pod of the space ship, and it caught a perfect highlight and blazed your name across the screen for one frame in dailies and you got your ass fired, well, don't come running to me.

1.4.6 Thinking Out of the Box

The final advice I have for you, as nascent superstar CGI TDs, is quite simply, forget everything you've ever been told cannot be done. Tool many times I've senior TDs insist something couldn't be done until some junior TD who doesn't know any better simply does it. Think out of the box.

Most importantly, don't be constrained by the solutions to problems that have been proposed in those books and Web sites whose code you are grabbing. If the illumination models you have are not working very well, don't be afraid to write a new one. If you can't figure out how to get a texture to stick onto the right parts of an object you're shading, consider whether you have the right model for that object. Perhaps you need a totally different kind of geometry, or some type of custom vertex data attached to the geometry, or a brand new form of texture mapping projection. Animated texture maps which contain the texture coordinates for another texture projection? Generating a hold-out matte for comping from an animated piece of geometry, with a shader that makes of the object opaque only near their silhouettes? Two depth maps, one from the light and one from the camera, to calculate the length of the optical path of illumination through the interior of the object?

The Art of Chi-Ting means just that. Cheat. Whenever possible. Users groups sponsor "Stupid Tricks" presentations in order to inspire you to think of even crazier solutions to intractable problems. The only thing that matters is the color of the pixels in the final image, so whatever sideways, upsidedown, chewing gum and bailing wire solution you can find that gets those right pixels is legal.

Another way to think out of the box is to think about using the renderer for processes other than computing the final image. Now, it is not uncommon for people to render in multiple passes, such as rendering reflection maps and shadows before the beauty pass, or rendering the final image in layers for compositing. But there could be so much more! The renderer is an extremely powerful piece of software that has matrix manipulation, spline evaluation, convolution and filtering operators,

bounding box evaluation, ray intersection, and a hundred other useful graphics algorithms coded up, debugged, working and available for your use. There's also a very powerful interpretted language which efficiently parallel processes numerical calculations. Seems like a lot of cool stuff just waiting to be taken advantage of.

What if you used RIB and Shading Language to write models and shaders whose which would compute interesting information, for use not just in rendering, but in modeling, animation or dynamics? Consider, could you use the renderer:

- to evaluate the curvature of a piece of cloth to see if it is folding more than the thickness permits?
- for collision detection between two subdivision surfaces?
- to drive a particle system based on noisy forces?
- to draw schematic diagrams of motion paths and their projections onto an uneven ground plane?
- to call your spouse and tell them you'll be late again tonight?

The list seems endless, and is only limited by your imagination.

1.4.7 Conclusion

If you've made it this far, you truly have the stuff that TDs are made of. You are standing on the shoulders of all the TDs who came before you (and of the CGI researchers who toiled before TDs existed), many of whose work has been referenced in my spiel. The Lore is a dynamic, growing body of knowledge, and those of us who have drawn deeply of it in the past expect that you will use it, spread it and add to it as we all have. Your destiny, it is!

And always remember,

The Audience Is Watching!

Chapter 2

A Recipe for Texture Baking

Larry Gritz,
Exluna, Inc.
lg@exluna.com

2.1 Introduction and Problem Statement

Baking is common parlance in production circles to mean taking complex, expensive, and data-heavy operations that would ordinarily be done at render-time, and pre-computing the operations and stashing the results in texture maps or other data structures that can be quickly and easily referenced at render time.

Many surface and displacement shaders, upon inspection, reveal themselves to be easily partitioned into (a) fairly expensive, usually view-independent, almost always lighting-independent, code, and (b) simpler, usually view-dependent, generally lighting-dependent code. Without loss of generality, let's call (a) the "pattern" section of the shader, and (b) the "lighting" section of the shader. Given the way people think about and develop shaders, it's often quite simple to spot a single line in a shader that has the property that all pattern code precedes that point, while all lighting code follows the point.

If the pattern computations are extremely expensive, it is very tempting to want to pre-compute the operations once, store them in a texture map, and replace the entire pattern section of the shader with a simple `texture()` lookup (or a small number of `texture()` lookups) on all subsequent executions of the shader.

There are several excellent reasons why you might want to bake complex computations into texture maps:

- Baking can speed up render-time shader execution by replacing a section of expensive view-independent and lighting-independent computations with a small number of `texture()` calls. For patterns that are truly dependent on only parametric (or reference-space) position, and have no time-dependence at all, a single baking can be reused over many shots. This means that it's okay for the original computations to be absurdly expensive, because that expense will be thinly spread across all frames, all shots, for an entire production.

- RenderMan-compatible renderers do an excellent job of automatically antialiasing texture lookups. High-res baking can therefore effectively antialias patterns that have resisted all the usual attempts to analytically or phenomenologically antialias (see *Advanced RenderMan*, chapter 11, for an overview of techniques.)

- Modern graphics hardware can do filtered texture map lookups of huge numbers of polygons in real time. Although the programmability of GPU's in rapidly increasing, they are still years away from offering the full flexibility offered by RenderMan Shading Language. But by pre-baking complex procedural shaders with an offline renderer, those shaders can be reduced to texture maps that can be drawn in real-time environments. This revitalizes renderers such as Entropy or PRMan as powerful tools for game and other realtime graphics content development, even though those renderers aren't used in the realtime app itself.

  As a case in point, consider that you could use *Entropy* (or another renderer with GI capabilities) to bake out the contribution of indirect illumination into "light maps." Those light maps can be used to speed up GI appearance in offline rendering, or to give a GI lighting look to realtime-rendered environments.

2.2 Approach and Implementation

2.2.1 Theory of Operation

The basic approach I will use for texture baking involves three steps:

1. The scene is pre-rendered, possibly from one or more views unrelated to the final camera position. During this pass, the shaders on the baking object compute the "pattern" and save (s, t, val) tuples to a temporary file. The (s, t) coordinates may be any way of referencing the values by 2D parameterization, and need not be the surface `u`,`v` or `s`,`t` parameters (though they usually are).

2. The pile of (s, t, val) tuples is then used to derive a 2D texture map at a user-specified resolution.

3. During the "beauty pass," the shader recognizes that it should simply look up the data from the texture map, rather than recomputing.

Furthermore, to make the system robust and easy to use, we also assume the following additional design constraints:

- This scheme should work with `float`, `point`, `vector`, `normal`, or `color` data.
- The 2D texture map should be of high fidelity and should easily handle data that are oddly-spaced in (s, t).
- It should be very easy to understand and modify a shader to utilize the baking scheme.

2.2.2 Saving baked data to a sample file

The first component we need is a way to efficiently write the (s, t, val) tuples to a given file. We accomplish this with a "DSO Shadeop." A DSO (DLL on Windows) is a compiled code module that may be linked at runtime. With *Entropy*, *PRMan*, and other renderers, you DSO Shadeops allow you to write functions in C or C++ that may be called from your shaders. See the user manual for your renderer of choice for more information on writing DSO shadeops. It is necessary to write the baking function in C only because Shading Language does not have a convenient way to write to arbitrary files.

The shadeop we will implement is called `bake()`, and takes four arguments: the name of the filename to write to, the s and t coordinates, and the value to save. The value may be a `float`, `color`, `point`, `vector`, or `normal`.

In our implementation, it is completely up to the user to choose the filename in such a way as to allow easy lookup on the same patch when rendering the beauty pass. It's also up to the user to be sure that different values (such as diffuse reflectivity, bump amount, or whatever) are stored in different files without mixing up the data.

The implementation of the `bake()` function is straightforward: it writes the tuples to the given filename. To minimize disk I/O, tuples are batched and written to the file only when several thousand have been accumulated (or when all renderer operations are completed and the DSO's cleanup routine is called). There's a separate buffer for each file, which are distinguished by the filename passed.

In order to keep multiple simultaneous threads from overwriting each other, a `pthread mutex` is utilized to ensure that This may only be necessary only for Entropy; I'm not aware of any other compatible renderers that support multithreading. It's still up to the user to ensure that there aren't multiple simultaneous renders all trying to write to the same bake files (be sure to run the baking pass in isolation).

Listing 2.1 gives the complete code for the `bake` DSO shadeop. It was tested on *Entropy*, but should work without modification with any of several compatible renderers.

Listing 2.1: C++ code for the bake DSO shadeop.

```
// bake.C -- code for an Entropy DSO shadeop implementing the "bake" function.
//
// The function prototype for bake is:
//    void bake (string file, float s, float t, TYPE val)
// where TYPE may be any of {float, color, point, vector, normal}.
// The bake function writes (s,t,val) tuples to the given filename,
// which may later be processed in a variety of ways.
//
// "bake" is (c) Copyright 2002 by Larry Gritz.  It was developed
// for the SIGGRAPH 2002, course #16, "RenderMan in Production."
// This code may be modified and/or redistributed, but only if it
// contains the original attribution above and a reference to the
// SIGGRAPH 2002 course notes in which it appears.


#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <map>
#include <string>
#include <pthread.h>


////////////////////////////////////////////////////////////////////////
// Declarations -- this is all boilerplate and is necessary for the
// shader compiler and renderer to understand what functions are
// implemented in this DSO and what arguments they take.
////////////////////////////////////////////////////////////////////////

#include "dsoshadeop.h"

extern "C" {

EXPORT int protocol_version = DS_PROTOCOL_VERSION;
```

```
EXPORT DS_DispatchTableEntry bake_shadeops[] = {
    { "void bake_f (string, float, float, float)", "bake_init", "bake_done" },
    { "void bake_3 (string, float, float, color)", "bake_init", "bake_done" },
    { "void bake_3 (string, float, float, point)", "bake_init", "bake_done" },
    { "void bake_3 (string, float, float, vector)", "bake_init", "bake_done" },
    { "void bake_3 (string, float, float, normal)", "bake_init", "bake_done" },
    { "", "", "" }
};

EXPORT void *bake_init (int, void *);
EXPORT void bake_done (void *data);
EXPORT int bake_f (void *data, int nargs, void **args);
EXPORT int bake_3 (void *data, int nargs, void **args);

};  /* extern "C" */



////////////////////////////////////////////////////////////////////////
// Implementation
////////////////////////////////////////////////////////////////////////

const int batchsize = 10240;  // elements to buffer before writing

// Make sure we're thread-safe on those file writes
static pthread_mutex_t mutex;
static pthread_once_t once_block = PTHREAD_ONCE_INIT;

static void ptinitonce (void)
{
    pthread_mutex_init (&mutex, NULL);
}



class BakingChannel {
    // A "BakingChannel" is the buffer for a single baking output file.
    // We buffer up samples until "batchsize" has been accepted, then
    // write them all at once.  This keeps us from constantly accessing
    // the disk.  Note that we are careful to use a mutex to keep
    // simultaneous multithreaded writes from clobbering each other.
public:
    // Constructors
    BakingChannel (void) : filename(NULL), data(NULL), buffered(0) { }
    BakingChannel (const char *_filename, int _elsize) {
        init (_filename, _elsize);
    }

    // Initialize - allocate memory, etc.
    void init (const char *_filename, int _elsize) {
        elsize = _elsize+2;
        buffered = 0;
        data = new float [elsize*batchsize];
        filename = strdup (_filename);
```

```
        pthread_once (&once_block, ptinitonce);
    }

    // Destructor: write buffered output, close file, deallocate
    ~BakingChannel () {
        writedata();
        free (filename);
        delete [] data;
    }

    // Add one more data item
    void moredata (float s, float t, float *newdata) {
        if (buffered >= batchsize)
            writedata();
        float *f = data + elsize*buffered;
        f[0] = s;
        f[1] = t;
        for (int j = 2;  j < elsize;  ++j)
            f[j] = newdata[j-2];
        ++buffered;
    }

private:
    int elsize;     // element size (e.g., 3 for colors)
    int buffered;   // how many elements are currently buffered
    float *data;    // pointer to the allocated buffer (new'ed)
    char *filename; // pointer to filename (strdup'ed)

    // Write any buffered data to the file
    void writedata (void) {
        if (buffered > 0 && filename != NULL) {
            pthread_mutex_lock (&mutex);
            FILE *file = fopen (filename, "a");
            float *f = data;
            for (int i = 0;  i < buffered;  ++i, f += elsize) {
                for (int j = 0;  j < elsize;  ++j)
                    fprintf (file, "%g ", f[j]);
                fprintf (file, "\n");
            }
            fclose (file);
            pthread_mutex_unlock (&mutex);
        }
        buffered = 0;
    }
};



typedef std::map<std::string, BakingChannel> BakingData;
// We keep a mapping of strings (filenames) to bake data (BakingChannel).


// DSO shadeop Initilizer -- allocate and return a new BakingData
EXPORT void *bake_init (int, void *)
{
```

```
    BakingData *bd = new BakingData;
    return (void *)bd;
}



// DSO shadeop cleanup -- destroy the BakingData
EXPORT void bake_done (void *data)
{
    BakingData *bd = (BakingData *) data;
    delete bd;  // Will destroy bd, and in turn all its BakingChannel's
}



// Workhorse routine -- look up the channel name, add a new BakingChannel
// if it doesn't exist, add one point's data to the channel.
void
bake (BakingData *bd, const std::string &name,
      float s, float t, int elsize, float *data)
{
    BakingData::iterator found = bd->find (name);
    if (found == bd->end()) {
        // This named map doesn't yet exist
        (*bd)[name] = BakingChannel();
        found = bd->find (name);
        BakingChannel &bc (found->second);
        bc.init (name.c_str(), elsize);
        bc.moredata (s, t, data);
    } else {
        BakingChannel &bc (found->second);
        bc.moredata (s, t, data);
    }
}



// DSO shadeop for baking a float -- just call bake with appropriate args
EXPORT int bake_f (void *data, int nargs, void **args)
{
    BakingData *bd = (BakingData *) data;
    std::string name (*((char **) args[1]));
    float *s = (float *) args[2];
    float *t = (float *) args[3];
    float *bakedata = (float *) args[4];
    bake (bd, name, *s, *t, 1, bakedata);
    return 0;
}


// DSO shadeop for baking a triple -- just call bake with appropriate args
EXPORT int bake_3 (void *data, int nargs, void **args)
{
    BakingData *bd = (BakingData *) data;
    char **name = (char **) args[1];
```

```
    float *s = (float *) args[2];
    float *t = (float *) args[3];
    float *bakedata = (float *) args[4];
    bake (bd, *name, *s, *t, 3, bakedata);
    return 0;
}
```

2.2.3 Processing the bake file into a texture map

After running the shaders that call `bake()`, you will end up with a huge file containing data like this:

```
0.0625 0.0625 0.658721 0.520288 0.446798
0.0669643 0.0625 0.634662 0.523097 0.457728
0.0714286 0.0625 0.609961 0.52505 0.466174
...
```

Each line consists of an s value, a t value, and data for that point (in the above example, 3-component data such as a color). The next task is to process all that data and get it into a texture map. The implementation of this step is straightforward, but can be long and very dependent on your exact software libraries, so I will present only the pseudocode below:

Listing 2.2: Pseudocode for converting the bake samples to a texture map.

xres,*yres* = desired resolution of the texture map
$image[0..yres-1, 0..xres-1] = 0$; /* clear the image */
$F =$ filter function

for $y = 0$ to $yres - 1$ do:
  for $x = 0$ to $xres - 1$ do:
    $r = 2$; /* filter radius */
again:
    S = all samples with $(s \cdot xres) \in (x - r, x + r)$ and $(t \cdot yres) \in (y - r, y + r)$
    $totalweight = 0$;
    v = 0;
    for all samples $i \in S$ do:
      $w = F(s_i \cdot xres - x, t_i \cdot yres - y)$;
      $totalweight = totalweight + w$;
      $v = v + w \cdot val_i$
    if $totalweight = 0$ then
      $r = 2r$; /* try again with larger filter */
      goto again;
    else
      $image[y, x] = v/totalweight$;

write *image* as a texture file;

The pseudocode probably needs very little explanation. Please note that the pixels (x, y) range from 0 to the resolution, but the (s, t) values in the tuples ranged from 0 to 1. Also note that the step that must find all samples underneath the filter region for each pixel can be tremendously sped up by use of an appropriate spatial data structure to store the tuples and retrieve only those close to the pixel. Even simple binning can be considerably helpful in reducing the search.

2.3 Using Texture Baking

Listing 2.3 Example shader using texture baking.

```
surface expensive (float Ka = 1, Kd = 0.5, Ks = 0.5, roughness = 0.1;
                   color specularcolor = 1;
                   string bakename = "bake")
{
    string objname = "";
    attribute ("identifier:name", objname);
    string passname = "";
    option ("user:pass", passname);
    float bakingpass = match ("bake", passname);
    color foo;
    if (bakingpass != 0) {
        foo = color noise(s*10,t*10);
        string bakefilename = concat (objname, ".", bakename, ".bake");
        bake (bakefilename, s, t, foo);
    } else {
        string filename = concat (objname, ".", bakename, ".tx");
        foo = color texture (filename, s, t);
    }

    color Ct = Cs * foo;

    normal Nf = faceforward (normalize(N),I);
    Ci = Ct * (Ka*ambient() + Kd*diffuse(Nf)) +
         specularcolor * Ks*specular(Nf,-normalize(I),roughness);
    Oi = Os;  Ci *= Oi;
}
```

Listing 2.3 shows a sample shader that uses the `bake()` function. The shader takes the name of the value to bake as a parameter, `bakename`. It also asks the renderer for the name of the object (set by `Attribute "identifier" "name"`) and the name of the *pass* (set by `Option "user" "string pass"`).[1]

If the `"user:pass"` option is set to `"bake"`, the data (in this case, just `noise()` for demonstration purposes) is written via the `bake` DSO shadeop to the file "*objectname.bakename*`.bake`". For the beauty pass (when `"user:pass"` is not set to `"bake"`), the data is looked up from the texture file named "*objectname.bakename*`.tx`". (It's assumed that the `.bake` file has been converted to a texture file in the interim between the two passes.)

The `bakepass` shader parameter allows for multiple "layers" or "passes" for the baking scheme. Also, if you are properly naming your objects (`Attribute "identifier" "name"`), this will make a separate bake/texture for each object. It should be clear how to extend this technique to name files differently for each patch, if so desired.

[1] Both *Entropy* 3.1 and *PRMan* 10, and probably other renderers by now, support user options and attributes in this manner.

Listing 2.4 The BAKE macro for even easier texture baking.

```
/* bake.h - helper macro for texture baking */

#define BAKE(name,s,t,func,dest)                                          \
{                                                                         \
    string objname = "";                                                  \
    attribute ("identifier:name", objname);                               \
    string passname = "";                                                 \
    option ("user:pass", passname);                                       \
    float bakingpass = match ("bake", passname);                          \
    color foo;                                                            \
    if (bakingpass != 0) {                                                \
        dest = func(s,t);                                                 \
        string bakefilename = concat (objname, ".", bakename, ".bake"); \
        bake (bakefilename, s, t, dest);                                  \
    } else {                                                              \
        string filename = concat (objname, ".", bakename, ".tx");       \
        dest = texture (filename, s, t);                                  \
    }
}
```

As a further simplification to minimize impact on shader coding, consider the BAKE macro in Listing 2.4. The BAKE macro encapsulates all of the functionality and mechanism used for Listing 2.3, but computes the actual baked value as the result of a function whose name is passed to the macro. The function is assumed to have only two parameters: s and t. If other data (such as shader params, locals, etc.) are needed, they may be referenced from the outer scope with the `extern` keyword, and simply using the lexical scoping rules of locally-defined functions in SL (as per *R*enderMan Interface Specification 3.2). Listing 2.5 shows how simple a baking shader can be when using this handy macro.

Listing 2.5 Example shader using texture baking with the BAKE macro.

```
#include "bake.h"

surface expensive2 (float Ka = 1, Kd = 0.5, Ks = 0.5, roughness = 0.1;
                    color specularcolor = 1;
                    string bakename = "bake")
{
    color myfunc (float s, t) {
        return color noise (s, t);
    }

    color Ct;
    BAKE (bakename, s, t, myfunc, Ct);

    normal Nf = faceforward (normalize(N),I);
    Ci = Ct * (Ka*ambient() + Kd*diffuse(Nf)) +
         specularcolor * Ks*specular(Nf,-normalize(I),roughness);
    Oi = Os;  Ci *= Oi;
}
```

2.4 Helpful Renderer Enhancements

The code and methods presented in this chapter should work just fine for any of the largely compatible renderers discussed in this course. In *Entropy* 3.2, we have added a few minor helpful features to make this scheme even easier to use.

First, an attribute has been added to force a particular fixed subdivision rate (expressed as the number of grid steps per unit of u and v):

```
Attribute "dice" "float[2] fixed" [1024 1024]
```

Although our method of recombining the samples is quite robust with respect to uneven distribution of samples (which would naturally happen with the adaptive grid dicing found in most renderers), at times locking down the exact dicing resolution for the baking pass is helpful.

Second, we address the problem of needing to ensure that all parts of the surface are shaded. This can sometimes be tricky since the renderer tries to cull occluded or off-screen objects before shading, and is usually solved by doing several baking passes with different camera views in order to get full object coverage. So we added an attribute that forces the entire object to be shaded, regardless of occlusion or other factors that would ordinarily cull it:

```
Attribute "render" "forceshade" [1]
```

Finally, the `mkmip` program has been modified to accept a `-bake` command (followed by *x* and *y* resolutions) of the texture map) to create texture files from the bake samples. It implements a method similar to the one outlined in Listing 2.2, For example,

```
mkmip -bake 1024 1024 pattern.bake pattern.tex
```

2.5 Conclusion

The vast majority of shaders are not worth turning into baking shaders. The Shading Language interpreters in most of the RenderMan-compatible renderers tend to be efficient enough to make it convenient to simple code all computations into the shader. It's rarely worth modifying the workflow, or juggling the bake and texture files, just to speed up the shaders a bit. Furthermore, there are many cases when "patterns" are view-, lighting-, or time-dependent, and thus are not amenable to baking.

We're assuming that you generally *don't* want to bake textures, but when you do, it sure is handy to have a simple, reliable method. This chapter presented a method of baking data into textures which is easy to use, minimally invasive of your existing shader code, inexpensive, and should work with several compatible renderers with minimal modification. We also sketched out a number of circumstances when texture baking is very useful and can save large amounts of computation and aggravation.

We look forward to hearing your baking success stories, seeing your modifications and improvements to our method, and learning about new and unique uses for this technology. Enjoy the code!

Chapter 3

Light/Surface Interactions

Matt Pharr
Exluna, Inc.
mmp@exluna.com

3.1 Introduction

This section of the course notes will focus on the interaction between lights and surfaces in RenderMan, specifically how light and surface shaders work together to compute the final color values of surfaces. General familiarity with RI and the shading language is assumed, but expert knowledge is not necessary.

The first half of the notes will cover how surface reflection is defined in surface shaders. Surface shaders typically have two main phases: they first compute the values of patterns over the surface, making liberal use of shading language calls such as `texture()`, `noise()`, etc. The second phase then describes light reflection from the surface, taking the information about the pattern values at particular points on the surface, combining it with information about the illumination arriving at the surface, and computing the final color value at that point. We will focus on this second phase in these notes.

The second half of these notes will then discuss light shaders in detail. It will start with a review of the semantics of lights in RI and the shading language syntax and constructs for describing lights. We will describe a number of applications of these constructs. The remainder of the section will wrap up by considering the role of lights in affecting the behavior of surfaces in more ways than just computing how much light is shining on them.

3.2 Surface Reflection Background

When one first starts writing surface shaders, one is usually using either the built-in functions (like `diffuse()`, `specular()`, etc.), or pre-written functions (like the ones in the ARM book: `LocIllumOrenNayar()`, etc.) to describe the way light reflects from the surface. Under the covers, these functions are all constructed with `illuminance` loops, which allow a shader to compute how the surface reacts to light.

When a surface shader hits an `illuminance` statement (or its equivalent via a `diffuse()` call, etc.), the renderer loops over the light sources bound to the surface, executing each in turn. After the first light has finished running, the surface shader's `illuminance` loop runs with the color variable `Cl` and light direction vector variable `L` set appropriately for the incident intensity and its

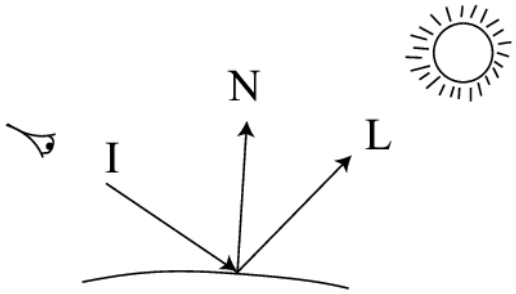


Figure 3.1: General setting for `illuminance` loops. A point on a surface is being shaded; the surface shader needs to compute the color `Ci` at the point as seen along the viewing direction `I`. The surface normal is `N` and the direction of incident illumination is `L`.

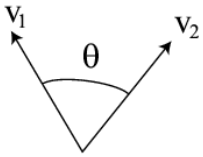


Figure 3.2: Two vectors v_1 and v_2, and the angle between them, θ.

direction for the light. The surface shader uses this information along with the viewing direction `I` and the surface normal `N` to compute the additional light reflected from the surface due to the light's illumination; see Figure 3.1. Then the next light is executed and the `illuminance` loop runs again, until all lights have been processed.

The single most important piece of math to know for writing and understanding illuminance loops is how the dot product works. For two vectors v_1 and v_2,

$$(v_1 \cdot v_2) = |v_1|\,|v_2| \cos\theta$$

where $\cdot$ signifies the dot product, $|v|$ signifies the length of a vector, and θ is the angle between the two vectors. In short, given two normalized vectors (i.e. $|v| = 1$), the dot product gives you the cosine of the angle between them; see Figure 3.2.

The dot product is computed by accumulating the sum of products of the components of the vectors:

$$(v_1 \cdot v_2) = x(v_1)x(v_2) + y(v_1)y(v_2) + z(v_1)z(v_2)$$

where $x(v)$ maps the vector to its floating-point x component value.

Recall that $\cos 0 = 1$ and $\cos \pi/2 = 0$ (with angles measured in radians). In other words, given two normalized vectors pointing in the same direction, their dot product is one; two perpendicular vectors have a dot product of zero, and vectors pointing in directly opposite directions have a dot product of -1. Thus, what the dot product gives you is a smoothly varying sense of how closely aligned two normalized vectors are, ranging from -1 to 1.

The dot product is a sufficiently basic operation that the shading language provides a `.` operator for it:

```
vector v1, v2;
float dot = v1 . v2;
```

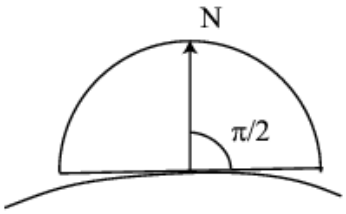


Figure 3.3: Specifying the cone of directions of illumination to consider in an `illuminance` loop: for most surfaces the hemisphere centered around the surface normal is all that is necessary.

Armed with this knowledge, we can understand how the `diffuse()` function would be implemented with an illuminance loop–see Listing 3.1. `diffuse()` describes a surface that equally reflects incoming light in all directions–a Lambertian reflector. The color at each point on such a surface doesn't change as you change your viewing position–this is in contrast to glossy surfaces, for example, where the specular highlight moves around as you look at it from different directions.

Listing 3.1 The `illuminance` loop corresponding to the built-in `diffuse()` function.

```
normal Nf = normalize(faceforward(N, I));
illuminance (P, Nf, PI/2) {
    Ci += Cl * (Nf . normalize(L));
}
```

The code starts out with the standard preamble to `illuminance` loops; the normal `N` is normalized and flipped if need be, so that it is facing outward toward the viewer. The `illuminance` loop then specifies the position to gather light at (here just `P`, the point being shaded), and a cone of directions to gather light over–here the hemisphere centered around the surface normal, given by the direction `Nf` and the angle in radians that the cone spreads out around `Nf`–see Figure 3.3.

The contribution for each light, `Cl`, is accumulated into `Ci`, the final output color of the shader. We multiply `Cl` by the cosine of the angle between the incoming light's direction and the surface normal–this is the expression of *Lambert's Law*, which says that the amount of light arriving at a point on a surface from a light source of given power is proportional to the cosine of the angle of incidence of the light. See Figure 3.4 for an intuitive sense of why this is so. Fortunately, the dot product operator gives us exactly the cosine we're looking for.

The dot product also plays a key role in glossy reflection models. The heart of an illuminance loop implementing the Phong specular model is given in Listing 3.2.

Listing 3.2 Illuminance loop implementing the Phong reflection model. The cosine of the angle between the light direction and the reflected direction `R` is used to determine the size of the specular highlight.

```
normal Nf = normalize(faceforward(N, I));
vector R = normalize(reflect(I, Nf));
illuminance (P, Nf, PI/2) {
    float cosr = max(0, R . normalize(L));
    Ci += Cl * pow(cosr, 1/roughness);
}
```

The Phong model starts by computing the reflection vector `R`, which gives the mirror reflection direction for the viewing direction `I`; see Figure 3.5. It then computes the cosine of the angle between

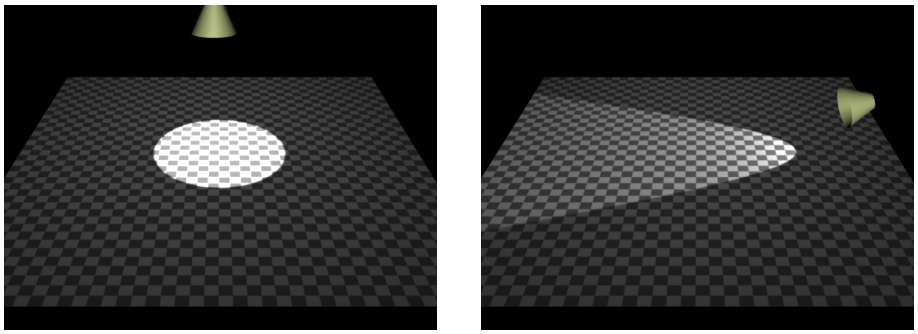


Figure 3.4: The cosine law for light arriving at a surface: as a light moves from being directly above the surface toward the horizon, the energy arriving at a point is given by the cosine of the angle between the light vector and the surface normal. This can be understood intuitively by seeing how the light inside the cone of a spotlight is spread out over a larger area of the surface as it heads toward grazing angles; at any particular point in the cone, less light energy is arriving.

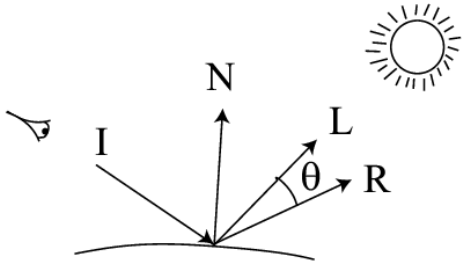


Figure 3.5: Phong model setting. The incident direction `I` is reflected about the normal to compute a reflected direction `R`. The angle between this and the light direction determines the highlight size.

that direction and the direction the light is coming from. Recalling the intuition of the dot product, it is computing a value that reaches a maximum value of 1 when the light is arriving exactly along the mirror direction but then decreases as the light direction deviates from this.

This cosine value is arbitrarily raised to a exponent determined from the user-supplied `roughness` value (which should be in the range $(0, 1]$). The higher the exponent, the smaller the highlight will be–this is just because the more times you multiply `cosr` with itself, the more quickly it falls off to zero along the edges (recall that `cosr` is always between 0 and 1 in the above code.)

3.3 Illuminance Loop Trickery

In `illuminance` loops, a few dot products will get you a long way. We will now explore a number of wacky things to do in `illuminance` loops to get unusual shading effects.

3.3.1 Diffuse Falloff

It's sometimes handy to be able to control the falloff of the diffuse shading model, varying the effect of the `(N . L)` term of Lambert's Law. This is a completely non-physical thing to do, but it's

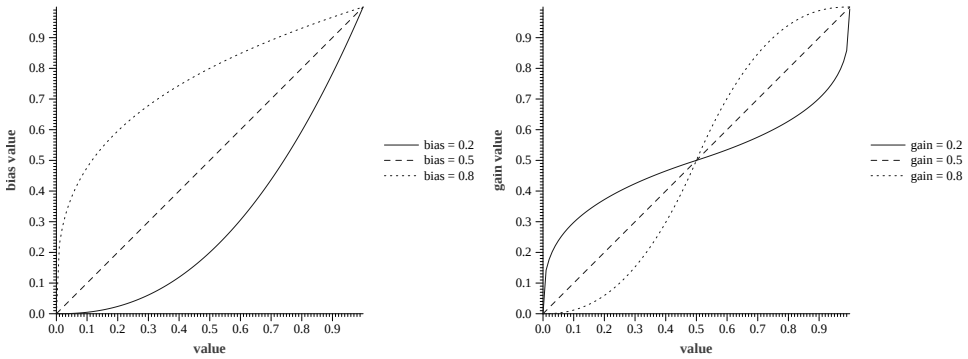


Figure 3.6: Graphs of the bias and gain functions. These are two handy functions that can be used to remap values between 0 and 1 to new values between 0 and 1. Providing reasonably intuitive handles to vary the behavior of the remapping, these functions have a lot of use in shaders.

a useful control to make available so that the falloff of light on a surface can be tightened up or expanded to happen over a shorter or larger area of the surface.

In order to put an intuitive handle on this control, we'll first look at two handy shader functions developed by Ken Perlin–`gain()` and `bias()`. These both remap values in the range zero to one to new values in the range. Both smoothly vary over that range, are monotonic, and always map the value zero to zero and the value one to one–all desirable properties for such functions. Source code for the functions is given in Listing 3.3.

Listing 3.3 Implementation of the `bias()` and `gain()` functions

```
float bias(varying float value, b) {
    return (b > 0) ? pow(value, log(b) / log(0.5)) : 0;
}

float gain(float value, g) {
    return .5 * ((value < .5) ? bias(2*value, 1-g) :
                                (2 - bias(2-2*value, 1-g)));
}
```

The behavior of the functions is best understood by looking at their graphs–see Figure 3.6. `bias()` pushes the input values higher or lower–bias values less than 0.5 push the input values down from where they were, and bias values over 0.5 push them higher. Rather than pushing all values in a particular direction `gain()` can be used to push the input values toward the middle of the range, around 0.5 (with a low gain), or can push the input values toward the extremes–0 and 1–with a gain greater than 0.5.

To control the diffuse falloff of an object, we can just run the cosine term from Lambert's law through the bias function–see Listing 3.4. Figure 3.7 shows a few spheres shaded with this model. With a low `falloff` value of 0.2 (left), there is a quick transition from the illuminated region to the unilluminated region–the transition region has been tightened up substantially, leading to an almost cartoon-like look. The center image has a falloff of 0.5, which means that the `gain()` function is no effect–the result is the standard `diffuse()` model. On the right, the falloff is 0.8, which gives a more spread-out and diffused transition between them.

Listing 3.4 Shader implementing a modified diffuse model with control over the falloff between the lit and un-lit regions of the surface.

```
surface matteFalloff(float Ka = 1, Kd = 1, falloff = 0.5) {
    normal Nf = normalize(faceforward(N, I));
    illuminance(P, Nf, PI/2)
        Ci += Kd * Cl * gain(Nf . normalize(L), clamp(falloff, 0, 1));
    Ci += Ka * ambient();
    Ci *= Cs;
}
```

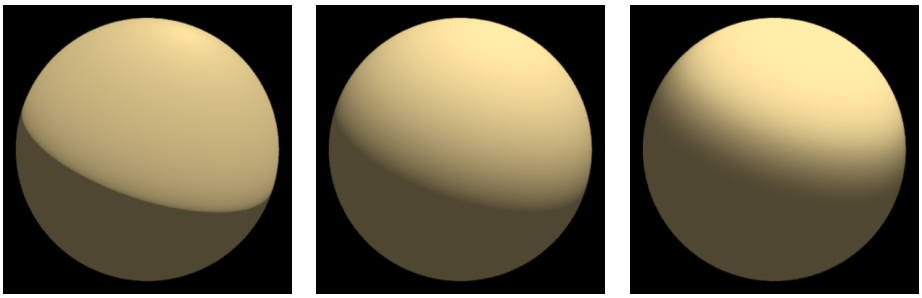


Figure 3.7: The effect of varying the falloff control in the matteFalloff shader. From left, falloff values of 0.2, 0.5, and 0.8. The falloff value of 0.5 has no effect on the falloff zone, and gives the same effect as the builtin `diffuse()` function.

3.3.2 Specular Highlight Control

Similarly, it can be handy to have more precise control about how specular highlights fall off with glossy models like `specular()`. The ARM book includes a glossy reflection model that adds a `sharpness` parameter [**?**, p. 231]; as with in the built-in model, a `roughness` value determines the size of specular highlights, while `sharpness` describes how the highlight fades in and out; see Listing 3.5.

This code starts with a model based on Jim Blinn's half-angle specular formulation. The main idea is that a normalized half-angle vector `H` is computed that represents the average of the light and viewing directions. The cosine of the angle between this and the surface normal is then raised to a power, similar to the Phong model above. Like the Phong model, this cosine reaches a maximum value of 1 when the light direction is equal to the reflection of the incident direction around the surface normal. The falloff beyond this is slightly different, however.

The result of the exponentiation is then run through `smoothstep()`. The idea is that if the cosine term is below a threshold value, it goes to zero, above another threshold value it just goes to one, with a smooth transition in between–very different behavior than the standard model where the contribution smoothly varies from zero to one, only reaching one for a single incident lighting direction (rather than for a number of them, as in the new model.) The result is in Figure 3.8; by sharpening up the highlight transition zone, we have given the surface a substantially different look.

3.3.3 Rim Lighting

It is often desirable to use lighting to visually separate an object from the background [**?**, Chapter 13]; by having lights behind objects that illuminate their edges as seen by the viewer, the objects stand out better. Rather than placing lots of lights just so, a little help from the shader can make this task easier.

Listing 3.5 Sharpened specular highlight model. Rather than running the cosine of the angle between the `H` vector and the normal through `pow()`, it is thresholded with `smoothstep()`, giving a more abbreviated transition from no highlight to full highlight, and increasing the area of maximum highlight value.

```
surface sharpSpecular(float Ka=1, Kd=1, Ks=1, sharpness = .1,
                      roughness = .1; color specularcolor = 1) {
    normal Nf = normalize(faceforward(N, I));
    vector In = normalize(I);
    float w = .18 * (1-sharpness);
    illuminance(P, Nf, PI/2) {
        vector Ln = normalize(L);
        vector H = normalize(Ln + -In);
        Ci += Cl * specularcolor * Ks *
            smoothstep(.72-w, .72+w, pow(max(H . Nf, 0), 1/roughness));
    }
    Ci += Ka * ambient() + Kd * diffuse(Nf);
    Ci *= Cs;
}
```

The rim lighting shader shown in Listing 3.6 is a simple example of such a shader. This shader extends the standard diffuse model by computing an additional `edgeScale` term. Define the viewing vector `V` to be `-I`, the vector back toward the viewer from the point being shaded. The dot product of `V` and the normal `Nf` tends toward zero as we approach the edges of the surface as seen by the viewer. Take one minus this value to give us something that is larger around the edges and then spiff it up a bit to give an edge scale factor. This edge scale is then just used to scale to the standard diffuse shading model up or down a bit; the result is shown in Figure 3.9.

Listing 3.6 Rim lighting shader. An extra kick is added to brighten glancing edges, as detected by the `1 - (V . Nf)` term. This is boosted to enhance its effect and used to scale the brightness in the standard diffuse model.

```
surface rim(float Ka=1, Kd=1, edgeWidth = .2) {
    normal Nf = normalize(faceforward(N, I));
    vector V = -normalize(I);

    illuminance(P, Nf, PI/2) {
        vector Ln = normalize(L);
        float edgeScale = bias(1 - (V . Nf), edgeWidth);
        edgeScale = max(.7, 4*edgeScale);
        Ci += Cl * Kd * (Ln . Nf) * edgeScale;
    }
    Ci += Ka * ambient();
    Ci *= Cs;
}
```

Note that not all of the edges of the object are brightened up in the image–the ones at the very bottom right of it, for example. One might want to try to add rim lighting all around, by removing the `(Ln . Nf)` term from the shader entirely–this is the part that is making those edges go black. This just replaces a slightly undesirable effect with an actual artifact, however–the problem is that that part of the model is in a transition zone between being lit and in a shadow. Because `Cl` goes to zero in the shadowed region, there's nothing we can do with a scale term that will brighten it up.

There are a few ways this problem could be worked-around. For one, we could make the extra

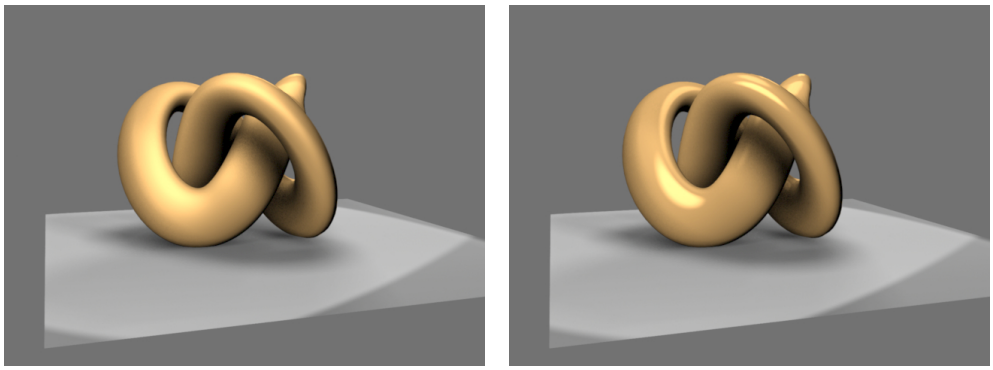


Figure 3.8: Glossy control with the `sharpSpecular` shader. On the left is an implementation of the standard Blinn model, while on the right it has been augmented with a sharpness control as in Listing 3.5. This has made it possible to tighten up the broad transition zones of the specular highlights for a sharper effect.

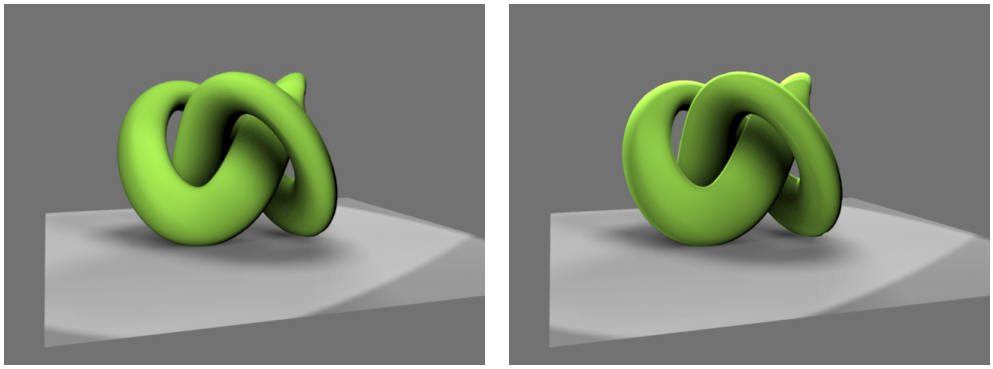


Figure 3.9: Rim lighting example. On the left, a standard `matte` shader has been applied, while on the right the rim lighting model was used, emphasizing the lighting along the glancing edges.

edge effect an additive term, rather than a multiplicative scale factor. This way, the light color `Cl` could just be ignored. This makes us unable to control the brightness of the edge effects by turning up or down the light brightness, however. Another option would be to have a separate set of lights that don't cast shadows and that are only used for rim lighting; light categories, which are described in the next section, give a convenient way have special sets of lights like this.

3.3.4 Retro-Reflection

Steve Westin has written a shader that describes *retro-reflective* surfaces; we will describe the mechanics of its operation with the dot-product-intuition in mind. Most surfaces–plastics, metals, etc.–preferentially scatter light along the reflected direction of the incident ray. Some surfaces preferentially scatter light back toward the direction it came from; the moon and some fabrics are two examples. Westin's shader, shown in Listing 3.7, is an empirical model of retro-reflection, based on actual research results.

This shader computes two terms. The first is a Phong-style glossy lobe, but where the cosine of the angle between the `V` and `L` vectors is raised to a power–see Figure 3.10. The closer these two are

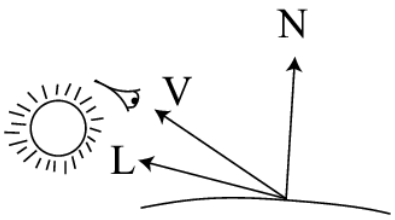


Figure 3.10: Retro model setting

together, the more retro-reflection there is. Next, the sine of the angle between `N` and `V` is computed; when these two are perpendicular, the since is one, thus emphasizing the edges. This also is raised to a power to adjust the rate of its falloff. Results are shown in Figure 3.11.

Listing 3.7 Velvet shader source code. A glossy retro-reflective lobe is combined with extra edge-scattering effects to give the appearance of a velvet-like sheen.

```
surface
velvet(float Ka = 0.05, Kd = 0.1, Ks = 0.1; float backscatter = 0.1,
       edginess = 10; color sheen = .25; float roughness = .1) {
  float sqr(float f) { return f*f; }

  normal Nf = faceforward (normalize(N), I);
  vector V = -normalize (I);

  color shiny = 0;
  illuminance (P, Nf, PI/2) {
    vector Ln = normalize ( L );
    float cosine = max ( Ln.V, 0 );
    shiny += Cl * sheen * pow (cosine, 1.0/roughness) * backscatter;

    cosine = max ( Nf.V, 0 );
    float sine = sqrt (1.0-sqr(cosine));
    shiny += Cl * sheen * pow (sine, edginess) * (Ln . Nf);
  }

  Oi = Os;
  Ci = Os * (Ka*ambient() + Kd*diffuse(Nf)) * Cs + shiny;
}
```

3.3.5 Anisotropic Reflection

We're not yet done finding things to take dot products with. We can develop a basic anisotropic shader by using this machinery as well. Anisotropic surfaces are surfaces that have glossy specular reflection, but where the shape and brightness of the highlights vary as the orientation of the surface varies. Brushed metal and phonograph records are classic examples of anisotropic surfaces.

Anisotropy in these surfaces is generally due to their having many small cylindrical grooves or bumps on them; these features all share the same (or very similar) orientation. This is in contrast to

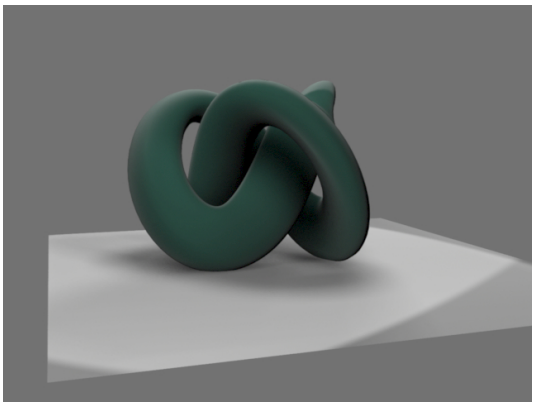


Figure 3.11: Velvet shaded object. Note emphasized edge effects and retro-specular highlights.

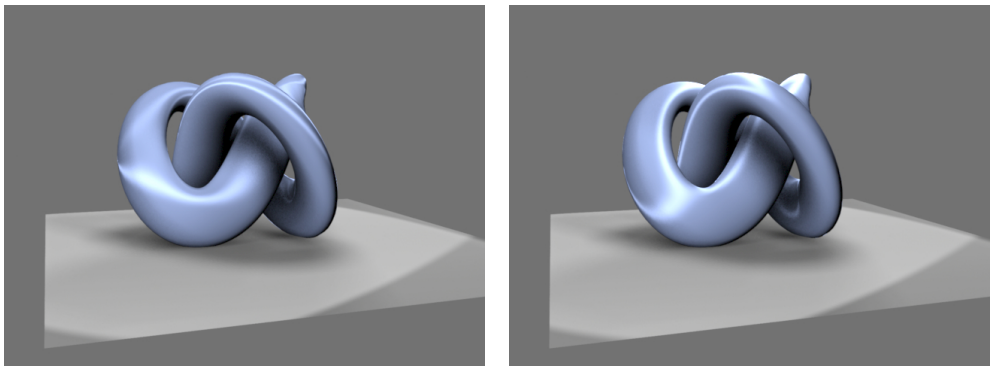


Figure 3.12: Anisotropic model with varying rotations of the direction of anisotropy

most specular surfaces, which are modeled as bumpy surfaces but with no dominant orientation of the bumps.

The shader in Listing 3.8 shows a basic anisotropic model. We start with a orientation `dir`, which gives the direction of the grooves in the surface. In case the user has given a direction that is not perpendicular to the surface normal, we take the cross product of the given direction with the normal, giving a direction that is guaranteed to be perpendicular to both the normal and the given direction, and then take another cross product, to give us a direction perpendicular to that. If the original direction was in fact perpendicular to the normal, this should be a no-op; otherwise it gives us a direction close to the original one but that lies on the tangent plane at the point being shaded.

We then compute a half-angle vector `H` (similar to the Blinn model) and look at the angle between that and the groove direction. When they are perpendicular, the model should be maximally bright, while as they become closer to parallel, it darkens. The images in Figure 3.12 shows this shader in action. The same object has been rendered twice, with two different values for the `angle` parameter to the shader.

3.3.6 Parting thoughts

There's a lot of fun to be had fooling around with `illuminance` loops; the key is having the right building blocks and developing an intuition about how the various vectors involved can be mixed to-

Listing 3.8 Anisotropic surface shader. The user supplies an orientation direction for the grooves in the surface as well as an option angle to rotate it by. The shader computes the effects of anisotropic reflection using the deviation between a half-angle vector and the groove direction.

```
surface aniso(float Ka=1, Kd=1, Ks=1, roughness=.02;
              uniform float angle = 0; color specularcolor = 1;
              varying vector dir = dPdu) {
    float sqr(float f) { return f*f; }

    normal Nf = normalize(faceforward(N, I));
    vector V = -normalize(I);

    vector anisoDir = dir;
    if (angle != 0) {
        matrix rot = rotate(matrix 1, radians(angle), Nf);
        anisoDir = vtransform(rot, anisoDir);
    }
    anisoDir = normalize((anisoDir ^ Nf) ^ Nf);

    illuminance(P, Nf, PI/2) {
        vector Ln = normalize(L);
        vector H = normalize(Ln + -In);
        float aniso = pow(1-sqr(H . anisoDir), 1/roughness);
        Ci += Cl * specularcolor * (Nf . Ln) * Ks * aniso / max(.1, (V . Nf));
    }
    Ci += Ka * ambient() + Kd * diffuse(Nf);
    Ci *= Cs;
}
```

gether in ways that give the effect you're looking for. As the examples in this section have shown, the dot product, `gain()`, `bias()`, `pow()` and `smoothstep()` functions are among the most important for shaping the results of computations with the `illuminance` vectors.

One last function that we haven't yet used is `fresnel()`; this function describes the effect of *Fresnel reflection* from surfaces. Fresnel reflection is a physical phenomenon that describes what happens to light at the boundaries between different media (such as air and a surface.) Some of the arriving light enters the surface, is scattered inside it, and then exits in new directions, and some just bounces off the surface in the reflected direction and keeps going. The effect can be seen on many surfaces–for example, glass, where the parts that are facing the viewer head-on mostly let light transmit through them, while the edges reflect more light than they transmit. Or look at a varnished wood table from above and then notice the mirror-like reflection at glancing angles–both of these are Fresnel reflection in action.

3.4 Lights and Surfaces, Working Together

A fancy `illuminance` loop is no good without something illuminating it, so each geometric primitive in RI has a set of lights bound to it. In this section, we will look at how lights and surfaces interact in RI, covering both basic operation and the more unusual ways they can work together.

When the surface shader for the primitive is executed to compute its final shaded color, the bound light sources are executed to compute the amount of illumination on the surface. The lights are responsible for doing shadow checks (if needed), in addition to whatever other computation is appropriate. A number of basic light shaders are provided, including "pointlight", "spotlight", and "distantlight". Ronen Barzel's well-known "uberlight" shader is a particularly sophisticated light

shader, with many tuning controls for tweaking the light's effect.

Most light shaders use the `illuminate` or `solar` constructs to describe how they illuminate the scene. `illuminate` is used for light coming from a particular point in the scene (e.g. the position where a point light has been placed, or a point on an area light source), while `solar` is used for "lights at infinity"–distant lights that are defined as casting illumination from a particular direction, rather than a particular location.

`illuminate` and `solar` both implicitly set the variable `L` that holds the vector along which illumination is arriving at the point being shaded. The main remaining responsibility of the light shader, then, is to compute `Cl`, the color of light arriving at the shading point.

3.4.1 Illuminate

`illuminate` is similar syntactically to `illuminance` in that it takes a position and optional direction and angle arguments. Light is cast from the given position, so the light's `L` variable is implicitly set to `Ps-P`, where `Ps` is the surface point being shaded. (The direction of the `L` variable is flipped after the light shader finishes, before the surface gets a hold of it.) The optional direction and angle can be used to specify a cone that defines a limit to the volume of space that the light illuminates.

```
illuminate(P) { ... }
illuminate(P, direction, angle) { ... }
```

3.4.2 Solar

`solar` has two variants as well. When called with no arguments, it describes a light source that illuminates from all directions. More commonly, it's called with a direction vector that describes where the light is coming from and an angle (possibly zero) that specifies a cone of illumination directions around the axis.

```
solar() { ... }
solar(axis, angle) { ... }
```

3.4.3 Scope and Binding

The RenderMan interface allows quite a bit of flexibility in terms of how lights are bound to surfaces. When a geometric primitive is defined in a RIB file, any light source that has previously been defined in the RIB stream can be bound to the surface.

Light sources behave differently than most other entities in RIB files in terms of how attribute blocks affect them. After a light has been defined, it illuminates all of the primitives that follow it, up until it is explicitly turned off or an `AttributeEnd` statement is reached–see Listing 3.9.

Listing 3.9 RIB snippet that demonstrates light source binding rules.

```
AttributeBegin
  LightSource "spotlight" 1234 "intensity" [100]
  Sphere 1 -1 1 360   # lit by the light
  Illuminate 1234 0   # turn it off
  Sphere 1 -1 1 360   # not lit
  Illuminate 1234 1   # turn it back on
AttributeEnd
Sphere 1 -1 1 360   # not lit by the light because of attribute end
Illuminate 1234 1   # turn it on; we can still refer to it
Sphere 1 -1 1 360   # lit because the light was turned on
```

This behavior in attribute blocks is different than almost everything else in RI; for example, if a surface shader is defined inside an attribute block, it is completely forgotten when `AttributeEnd` is reached. After that point, there is no way to refer back to that shader and bind it to a new object without having a new `Surface` call in the RIB stream. The sections below will describe a number of tricks that are possible thanks to the fact that lights have this greater flexibility.

3.4.4 Light Categories

Light categories add a new dimension to the light binding mechanism. Each light in the scene can have one or more string category names associated with it. In an `illuminance` loop, surfaces can then limit the set of lights that are executed for that loop.

To associate a light source with one or more category names, the light needs to be declared to have a string parameter named `__category` (that's with two underscores).

```
light mylight(...; string __category = "") {
   ...
}
```

When this parameter is set, it associates the light with the given categories. To associate a light with multiple categories, the names can just be separated with a comma.

An optional string parameter can be given to `illuminance` calls in surface shaders to limit the light categories to be considered for that illuminance loop.

```
  illuminance("specialLights", ...) {
     ...
  }
```

All lights that have a category that matches the given string will be executed for that loop. Alternatively, if the category string passed starts with a minus sign, "-", all lights *except* those matching the given category name will be executed.

Light categories could be used, for example, to specify a special set of lights that only contribute to rim lighting. The main `illuminance` loops would exclude the lights in the "rimLights" category, while the one for the rim term would only use the "rimLights".

3.4.5 Message passing

In addition to selecting a set of lights with categories, surfaces and lights can also pass general data back and forth between them. This allows the surface shader to pass information to the light shader, the light shader to make use of it, and then the light shader to pass additional data back to the surface shader, beyond the `L` and `Cl` variables that are set by the light.

A classic use for message passing is for the light to pass additional shading parameters back to the surface. For example, light sources in OpenGL have three colors associated with them: one affects the object's ambient component, one affects diffuse, and the last one affects specular. While this can't be emulated directly with the `Cl` variable in the shading language, it can easily be done with some message passing.

```
light openglPointLight(point from = point "shader" (0,0,0);
      float intensity = 1; color amb = 1, diff = 1, spec = 1;
      string shadowname = "";
      output color ClAmbient = 0, ClDiffuse = 0, ClSpecular = 0) {
    illuminate (from) {
        float visibility = 1;
        if (shadowname != "")
```

```
            visibility = 1 - shadow(shadowname, Ps);
        ClAmbient = intensity * amb * visibility;
        ClDiffuse = intensity * diff * visibility;
        ClSpecular = intensity * spec * visibility;
    }
}
```

A corresponding `openglSurface` then just ignores `Cl` in its `illuminance` loop but instead can use the appropriate light color value from the light shader, depending on whether it's doing ambient, diffuse, or specular reflection. Inside the `illuminance` loop, the surface calls the `lightsource()` function:

```
illuminance ( ... ) {
    color ClAmb = 0, ClDiff = 0, ClSpec = 0;
    lightsource("CLAmbient", ClAmb);
    lightsource("CLDiffuse", ClDiff);
    lightsource("CLSpecular", ClSpec;
    ...
}
```

`lightsource()` returns non-zero and sets the given variable if the named variable exists as an output variable in the light. If there is no such output variable, zero is returned and the supplied variable is left unchanged.

Light-controlled diffuse falloff

For a more interesting example, consider the diffuse falloff control described in Section 3.3.1; rather than making this a parameter of the surface shader, we might want to be able to have it be a property of the lights in the scene. This way, we could have two lights shining on an object, where the diffuse falloff from the first light's contribution was quite sharp, but the falloff for the second one was smoother. Without the ability to pass information between the lights and the surfaces, this would be impossible to do, since the surface wouldn't be able to differentiate between the two lights.[1]

What we need to do is to define another protocol for the surfaces and the lights. If the lights in the scene optionally provide a `float diffuseFalloff` value, and if the surfaces look for it and make use of it in their `illuminance` loops, then we can reach this goal.

First, the light shaders need to have an appropriate `output float` parameter in their declarations:

```
light mylight(...; output float diffuseFalloff = 0.5)
```

`diffuseFalloff` can be bound to a particular value in the `LightSource` call in the RIB file, or it can be set by the light shader as the result of some computation. The surface shader, then, can ask to see if the light source has passed a value called "diffuseFalloff"

```
illuminance (P, Nf, PI/2) {
    uniform float falloff = 0.5;  // default value
    if (lightsource("diffuseFalloff", falloff) != 0) {
        // the light passed us a value; it's stored in falloff
    }
    Ci += Cl * gain(Nf . normalize(L), falloff) * Kd * Cs;
}
```

Thus, we can easily associate different falloff values with different lights and achieve the effect we were interested in. As with light categories, this approach requires coordination between the surface and light shaders in the scene; however, the flexibility that it offers is substantial.

[1] One might imagine a scheme based on light categories, where the first light belonged to a "sharpDiffuseFalloff" category and the second belonged to a "smoothDiffuseFalloff" category. However, this isn't a very flexible solution.

Disabling shadows on a per-surface basis

This message passing can be bidirectional; the surface shader can also declare some of its parameters to be `output` variables, and then the light source can access them with the `surface()` message passing function. For example, a surface might want to request that lights not compute shadows for it. It could just declare an `output float noShadows` parameter. As long as lights look for this parameter and act appropriately, then the desired result is achieved.

```
light mylight() {
     float noShad = 0;
     surface("noShadows", noShad);
     ...
}
```

Choosing a shadow method per-surface

Consider a surface with fine displacement-mapped geometry, where shadow maps were unable to generate accurate shadows. If shadow maps were adequate for the rest of the objects in the scene, we can use shadow maps for them and only do ray-traced shadows for the displaced object. The surface shader for that object just needs to pass a value to the light shaders that directed them to use ray-traced shadows for it instead of shadow maps. While this problem could be solved with other approaches–e.g. having a completely separate set of lights for that object that had identical parameters to the original set of lights but that traced shadow rays instead, but the message-passing approach makes it much easier to switch between behaviors, just by changing a value bound to the particular object.

Non-diffuse and non-specular lights

One widely-used instance of surface/light message passing is to flag lights as not contributing to the diffuse component of the surface's BRDF or not contributing to the specular component. For example, a light that was just intended to cause a specular highlight would be flagged as not contributing to diffuse lighting, or a light that was used to fake global illumination would be flagged to not cast a specular highlight.

Light shaders are declared to have two `output float` parameters named `__nondiffuse` and `__nonspecular` (again, with two underscores). Surface shaders then use these in their illuminance loop; for example, the actual equivalent to the built-in `diffuse()` function is:

```
illuminance (P, Nf, PI/2) {
    uniform float nondiff = 0;
    lightsource("__nondiffuse", nondiff);
    Ci += Cl * (Nf . normalize(L)) * Kd * Cs * (1-nondiffuse);
}
```

If the light source has no `__nondiffuse` output variable, then the `lightsource()` call leaves `nondiff` untouched. Otherwise it is initialized appropriately. By treating it as a scale value, rather than a binary on/off switch, we also have the ability to reduce (or increase!) a light's contribution to the diffuse channel in relation to the others.

Multiple lighting rigs

A more complex example shows how light message passing can be used to set up multiple lighting rigs for the scene when doing global illumination. Lights can be identified as contributing only to lighting for indirect lighting computations, only for direct lighting, or for both. One can imagine a couple of reasons why it might be worth setting up a second set of lights for this:

1. Efficiency: if there are many lights in the scene and their shaders are expensive to evaluate (e.g. due to lots of ray-traced shadows, complex procedural lighting computations, etc.), then setting up a second set of lights that light the scene in a generally similar way to the primary lights but are much less expensive to evaluate may speed up rendering substantially.

2. Artistic: you may want to increase or decrease the overall brightness or tone of the indirect lighting in the scene, shine more light on some objects so that they cast more indirect light on other objects in the scene, etc.

The shader in Listing 3.10 shows how this might work in practice: lights in the scene have `output float` variables associated with them called `directLight` and `indirectLight`. When computing direct illumination (i.e. `isindirectray()` is zero), we grab at the `directLight` output variable from the light. If there is no such output variable on the light, we use a default value of one. We then modulate the light's contribution by the value of `directLight`; for lights that are only for the indirect rig, this value should be zero. We handle lights for indirect lighting only analogously.

Listing 3.10 Surface shader for multiple lighting rigs for direct and indirect lights. Depending on whether the surface is being shaded for direct camera visibility or for indirect illumination, a different set of lights can be used.

```
surface foo(color Kd = .5, Ks = .1; roughness = .1) {
    normal Nf = faceforward(normalize(N), I);
    vector In = normalize(I);
    color brdf(normal Nf; vector In, Ln; color Kd, Ks; float roughness) {
        vector H = normalize(In + Ln);
        return Kd * (Nf . Ln) + Ks * pow(H.Nf, 1/roughness);
    }

    illuminance(P, Nf, PI/2) {
        float direct = 1, indirect = 1;
        lightsource("directLight", direct);
        lightsource("indirectLight", indirect);

        if (isindirectray() != 0)
            Ci += Cl * indirect * brdf(Nf, In, normalize(L), Kd,
                Ks, roughness);
        else
            Ci += Cl * direct * brdf(Nf, In, normalize(L), Kd,
                Ks, roughness);
    }

    Oi = Os;
    Ci += Ka * ambient();
    Ci *= Cs * Oi;
}
```

This approach does require coordination between surfaces and lights in the scene; conventions need to be set up and naming schemes for the message-passed variables agreed upon ahead of time. Unfortunately, this can mean that it isn't easy to directly re-use old shaders unchanged–for the above example with the indirect lighting rig, all of the surface shaders in the scene would need to be modified to pay attention to the light output variables in order for the scheme to work.

Example images showing the results of this approach are in Figure 3.13. On the left is a Cornell box scene illuminated with a standard single lighting setup for direct and indirect lights, and on the

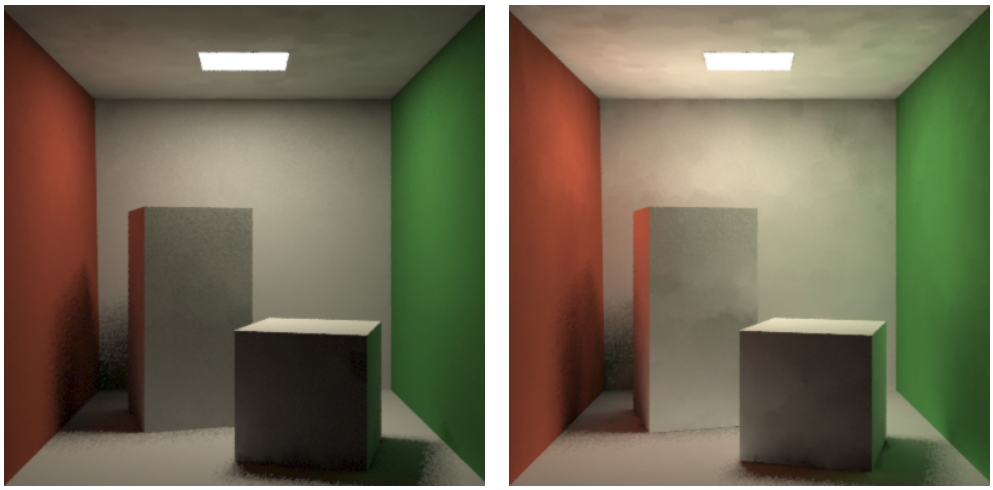


Figure 3.13: Indirect lighting with separate lighting rigs for direct and indirect lighting. On the left is the basic image, with the same rig for both. On the right, the light source for the indirect rig has been replaced with another light source that is 3 times as bright as the light used for direct lighting. This makes it possible to brighten up the indirect lighting without blowing out the entire image.

right, the brightness of the light in the indirect setup has been increased in order to brighten up the scene, particularly the shadow areas.

3.4.6 And beyond...

Once you get used to the idea that lights and surfaces can have rich inter-communication between them, all sorts of new possibilities come to mind. One common use is for lights to tell surfaces the filenames for environment maps to use. Special light sources that don't cast any illumination at all just pass a string parameter to the surface:

```
light bindEnvironment(output string envName = "";
                      string __category = "envMaps") {
    illuminate (point(0,0,0)) {
        Cl = 0;
    }
}
```

(We need a no-op `illuminate` loop in the shader so that the renderer doesn't flag this as an ambient light–ambient lights don't show up in `illuminance` loops, so message passing to them isn't possible.)

In the RIB file, `envName` is bound to an appropriate environment map file. Then, the surface shader has an additional illuminance loop:

```
vector R = reflect(In, Nf);
R = normalize(vtransform("world", R));
illuminance ("envMaps", P) {
    string envname;
    if (lightsource("envName", envname) != 0)
        Ci += environment(envname, R);
}
```

There are a number of reasons that this is a nice way to specify the environment maps for an object. First, it's an easy way to be able to specify more than one environment map to be applied to an object. While the surface shaders could be modified to take an array of strings to specify multiple environment maps, arrays in the shading language must be of fixed length, so an upper limit would need to be set ahead of time.

More generally, it's more intuitive and more flexible for a "light" to specify this information. Consider a complex model stored in a RIB file to be reused over multiple frames, with a different lighting environment in each frame (e.g. a real-world environment map changing in time, or a model that's re-used in many different scenes) If the environment map is specified in the `Surface` line of the RIB file, then a separate RIB file needs to be generated each time the environment map name changes. With this approach of binding the map via a light source, all of the flexibility of light source binding is available.

Having made this leap beyond the usual `illuminate` and `solar` work of the light source, all manner of bizarre things can be done. For example, consider lights in a special "scale" category where surfaces loop over them after the other lights have been processed; their `Cl` values could instead be interpreted as scale factors to be applied to the final color computed by the rest of the shader. Though completely non-physical, this is an easy way to make individual objects darker or brighter.

Acknowledgements

In a series of conversations and e-mails, Dan Goldman helped me understand how the unique flexibility in light-binding semantics in RI gives it many unexpected uses.

Mike King got me thinking about how a separate indirect lighting rig might be done. Also, his rim lighting shader in the RenderMan repository was a useful resource.

Steve Westin kindly allowed me to include his velvet shader.

Chapter 4

Preproduction Planning for Rendering Complex Scenes

Dan B Goldman,
Industrial Light + Magic
dgoldman@ilm.com

Abstract

The Renderman interface offers many features for handling large scenes. But harnessing these features, especially in the context of a large production with lots of division of labor, requires a great deal of coordination. This section of the Advanced Renderman course will discuss some of the considerations that need to be made, both technical and logistical, in the planning process for rendering scenes with large crowds, complex sets, or both.

4.1 Introduction

I should start by admitting that very little of the material presented here is particularly new. A lot of it has been said or written down before, especially in previous years of this course. My aim is to collect and summarize tips and tricks which are specifically oriented towards issues that arise when working with really complex scenes.

4.1.1 What do you mean, complex?

We'll be talking about scenes that don't fit in memory all at once, in any modeller or renderer or animation package or what-have-you. We're talking multi-gigabytes here.

It's been observed that by the time you get to about 1 to 10 million control vertices or polygons, you're not adding any visual detail to your film-resolution scene, which only has on the order of a few million pixels. (In fact, the REYES rendering engine was originally designed with the goal of rendering a million primitives in mind, based on this very reasoning.)

But while it's theoretically possible to hand-customize each frame of your animation so that only a million or so primitives are on screen, it's just not practical. You may, for instance, have a set composed of many small pieces, each of which is built in great detail for closeup work, but all of which appear together in an establishing shot. Or you may have a creature which is designed for

hero animation, which then must appear in a crowd of thousands. Either way, you can't just throw all of your scene geometry and textures and shaders together and hope for the best: Your producer will hunt you down and kill you before a single scanline of your render is completed...

Because these types of scenes just don't fit in memory all at once, you must carefully consider the best way to work with them. You want the artists designing the models have the control they need, but you also need to squeeze out a few thousand rendered frames before your show wraps.

This section of the course will discuss some of the strategies in use at Industrial Light and Magic for handling these types of megascenes. There are certainly other approaches that will work; consider this a sampling of philosophies. Once upon a time it was common to have lots of shots with just one dinosaur or spaceship or whatever, but those days are long gone: Todays epic directors demand epic *scope*! And we lowly droids must serve their dark will...

4.2 Stating the Obvious

The best way to plan for complexity is to *start early*! This is painfully obvious, but widely ignored... Still, it can't be restated often enough: The secret to smooth production is to think through the processes as early as possible.

A typical production workflow might look like this:

Artist A: Artwork/Designs

Artist B: Modelling/Skeletons/Skinning

Artist C: Texturing/Shading

Artist D: Animation

Artist E: Lighting/Rendering/Compositing

Or, if you work in a large production environment, it might look like this!

Artist A,B,C: Designs

Artist D,E,F: Artwork

Artist G: Modelling

Artist H: Skeletons

Artist I: Skinning

Artist J: Texturing

Artist K: Shaders

Artist L,M,N,O,P: Animation

Artist Q: Setup Lighting

Artist R: Shot Rendering

Artist S: Compositing

In other words, by the time you even get to render your scene, it's probably been through dozens of hands, all of whom are answering to a number of people who are very interested in adding realistic details but not very interested in how difficult it's going to be for you to get the movie/commercial/game/whatever finished.

So how can you convince everyone to listen? Just tell your producer it'll cost a lot of money otherwise... and remind the other artists you are only helping them make sure *their* vision is acheived on screen! I'm not going to give you any advice to use less detail for closeups, or change the models, or do anything that would sacrifice their efforts... just advice on how to use it to best effect, without bringing your renderfarm to its knees. (OK, you might do that anyway, but I guarantee it'll happen sooner otherwise...)

OK, so hopefully you've convinced your producer to start thinking about rendering now, even while you're in the design process. Now what?

There are two main concerns with rendering complex scenes:

1. They take a long time, and...
2. They use a lot of memory.

Correspondingly, there are really just three main ideas behind every technique I'm presenting:

1. Avoid loading data
2. Avoid doing computations
3. Postpone loading data

Obviously, it's best if you only load the data you need and only do the computations you need. But if you must load data, it's better to postpone doing this until you really really need it.

(Why didn't I mention postponing computation? Well, mainly because it doesn't help much... you have to do it eventually. Whereas postponing loading data helps by spreading out the memory needs of your render over the entire frame, so you don't get a big spike at the beginning that just hogs your machine – or repeatedly swaps in and out. This is probably not as important now for your average render as it was 5 years ago, but it still matters a great deal for big, complex scenes.)

4.3 Choose Your Weapons: Geometry and Textures

The first thing you can do is to make sure that your models are being built with an appropriate level of geometric detail. The goal here is not to use the whizziest newfangled geometric representation, but rather to have the most *compact* representation for your model. For instance, it's often said that Renderman handles patches better than polygons. While this is (somewhat) true, it would be more accurate to say "Renderman does a better job with smooth objects represented as patches instead of polygons." If you are building the creatures from the planet Polyhedra, for heavens' sake go ahead and use polygons!

Having said that, however, it is generally true that most organic models can be represented more compactly as patches or nurbs instead of poly meshes, and more compactly as subdivision-surfaces than patches or nurbs. But don't ever forget that your goal is compactness: converting a triangle mesh to patches by making every triangle into a bicubic patch isn't going to help any.

Don't forget about Renderman's lightweight primitives, too. `Points` and `Curves` aren't just for physically tiny things dust, hair, or grass: Any geometry which is very small on screen due to either size or distance is a candidate for one of these primitives. For example, if you have a very distant crowd you might consider using `Points` or `Curves` to represent its members. `Points` and `Curves` are especially useful when included in a level-of-detail hierarchy... but we'll get there later.

4.4 Procedural Primitives

Procedural primitives are a technique for delaying processing and data until it's needed. Rather than including a complete description of a model in your scene, instead you leave a note that says: ask me later. All procedurals come with a bounding box. This bounding box is a contract between the renderer and the procedural that says: "I promise not to make any geometry outside this box." Procedurals come in three forms:

- `DelayedReadArchive` means "look in this file." When the renderer gets to the part of the screen that might be occupied by the procedural (based on the bounding box), a given RIB file is loaded.
- `RunProgram` means (oddly enough) "run this program." When the bounding box is crossed onscreen, the program is run, generating more valid scene data.
- `DynamicLoad` actually loads a dynamic module (a dso or dll) into the program to generate the RIB describing what's in the box. (This variant is especially powerful, but also pretty dangerous: You can crash your entire render very easily with this, and it may be hard to make it work on other platforms.)

Procedurals are absolutely essential for scenes with lots of geometry. For example, the frames from *Star Wars Episode II: Attack of the Clones* shown in Figures 4.1 and 4.2 have tens of thousands of primitives in them.

Figure 4.1: Industrial Might and Logic. ©2002 Lucasfilm Ltd.

If we tried to stuff all this data into one giant RIB file for every frame, we'd spend a lot more time writing rib files than actually rendering! Instead, we make rib files that just point to a bunch of other prewritten rib files, and we use programs which manipulate preprocessed data to generate ribs quickly from prefabricated parts.

Note that the more procedurals you have in a given scene, the more important it becomes that the procedurals don't take too long to compute. Even if it takes just 10 seconds to run a `RunProgram`, if you have 1000 of those objects in your scene, that will add up to almost 3 hours per frame, just running your procedurals!

At ILM, we use a combination of procedurals for different situations. The ones which get run a lot are written in C and are very very fast. (Some of them have been rewritten many times over to improve their speed!) Some which get run only a few times per frame are written in python for

Figure 4.2: A Rebel, Adrift in a Sea of Conformity. ©2002 Lucasfilm Ltd.

more flexibility, though when working with lots of data it still sometimes makes sense to use a native compiled language like C for efficiency.

- For objects like set pieces which don't move and don't change their appearance often, we use `DelayedReadArchive`. The scene data that our animators work with consists mostly of lores proxy geometry, which is replaced by the `DelayedReadArchive` call when the RIB file is generated.
- For rigid objects which might appear multiple times with different appearances, (think cars, asteroids, etc.) we use a `RunProgram` which merges a geometry rib with a material file describing the shaders to apply to each piece of geometry within the object. That way we don't have to have a separate file including all the same geometry for every possible appearance.
- For objects which have internal motion (ie. having different geometry on each frame) such as walking or running creatures, we pregenerate a geometry rib file and an auxiliary data file for each frame, and use particle systems to propel the objects around the scene. Then a `RunProgram` instances that geometry according to the particle data. Generally each instance of a creature has a separate material, so this `RunProgram` creates instances of the material-replacement `RunProgram` mentioned above: there are actually two tiers of `RunProgram`.

More details about these `RunPrograms` were disclosed at this course in Siggraph 2001, so I won't go into detail on their operation here. You can find the relevant information in last year's course notes, chapter 4. Working python code is presented: the particle replacement script is called `dataread.py`, and the material replacement script is called `DJA.py`. Of course, for production use, you would probably want to rewrite both of these scripts in C or C++ for speed.

4.5 Level of Detail

Level of detail (LOD) is one of those buzzwords which means all things to all people. It refers both to some specific features of modelling and rendering systems, but also to a larger conceptual notion of building CG objects according to how they will be viewed on screen. While Renderman has a "level of detail" feature, I'll be focussing more on the latter definition, because it is a general way of thinking about handling large data sets of any type. It's useful to think about level of detail throughout your production pipeline.

The idea of level of detail is to use multiple versions of a model, choosing one depending on their size on screen. Renderman supports this notion with the `RiDetailRange` call, which allows you to specify multiple models in a single RIB. At render time, the renderer selects one of the models — or dissolves between two of them — according to the size of the model onscreen, in pixels. (Actually you can use any subpart of the model as your "pixel ruler", but I'll ignore that for this discussion... you can find complete details in Tony and Larry's book *Advanced Renderman*.)

Before starting work on any model which will appear in lots of scenes at lots of different scales, it's best to gather to discuss how and where it will appear, and decide how many and what kind of levels of detail you are going to build. Anyone who might be working on this model should attend this gathering, because everyone needs to be on the same page. You'll probably hear some questions like this:

"Won't it take a lot of extra effort?"

Some, but probably not as much as you think. We generally proceed by building our highest level-of-detail models first, then simplifying them to make lower levels of detail. Thus, building multiple levels of detail becomes a process of removing detail rather than adding it, so the hard work has (mostly) been done already. Depending on what representation you use for your model, it may also be possible to partially automate making lower levels of detail. (Subd-surfaces, for example, can often be derezzed by omitting smaller subdivisions.) Nonetheless, it does take some time, so be sure to budget time for building the lower levels of detail; don't wait until you need it in a shot to build it.

"OK, I'm game. How?"

Here are some suggestions for how to create levels of detail in all aspects of your model. It's important that all disciplines have a shared understanding of how the models will be used in the final film. Once you've agreed on how many levels of detail to use and how large they will be onscreen, disseminate this information among the team! If you start by creating the highest res first, as suggested above, one easy way to share this information is to render the high res creature at the desired resolution, then shrink down the image to the desired scales for the other models. By distributing this sort of image, all the artists have a shared understanding of the goals for each resolution.

4.5.1 Models

The easiest thing to do to reduce the resolution of a model is to just delete pieces! In particular, architectural models or vehicles often have a lot of "greeblies" which can just be removed for the lower resolutions. It's usually not as easy to do for creature models, but sometimes separate pieces like fingernails and clothing elements are good candidates for removal. Don't worry about their coloration; this can be fixed in paint.

Secondly, reduce the span or polygon count. Your hires model may have lots of folds and wrinkles that you won't see at lores... or maybe you can put that detail in displacement maps. Most modelling tools have a function to reduce span count; in Maya it's *Rebuild Surface*. If you're working with polys you'll want to remove edges and vertices, particularly subtleties like bevelled or smoothed edges. Some ILM modellers have been known to put bevels and threads on their bolts...

Finally, combine entire pieces. Especially on the lowest resolutions, you may be able to combine fingers together into "mittens" or replace entire limbs with polygonal (or subdivision-surface) box shapes.

4.5.2 Paint

The first thing that might come to mind when creating paint for lower resolution models is to reduce the size of the texture maps. While this won't hurt anything, it also won't help much! That's because the major implementations of Renderman already use a technique called mip-mapping which automatically creates levels of detail for texture maps, and uses the appropriate level depending on the size of the object being textured. So in many cases, you can and should use the exact same texture maps as your high resolution model, even if you've rebuilt the surfaces to have fewer spans.

A bigger factor to consider is the *number* of separate texture maps you have. You may want to consider combining textures for your lower resolution models. For example, while your highest res model may have a separate set of maps for each part of the body, you may want to combine parts using projected maps. One quick way of creating these new projections is to render your creature from orthogonal views using a strong ambient light. While this will create some texture stretching in some areas, it may not be noticeable at lower resolutions.

4.5.3 Skeletons and Skinning

The one thing you probably don't want to change between levels of detail is your skeleton, since you'd like to be able to map the same animations onto all the resolutions of the model. But you might be able to omit some smaller joints like fingers and toe joints as you switch to lower resolutions.

Also, at the smallest resolutions you may be able to eliminate skinning altogether and just use pieces which intersect at the joints. This can save a lot of animation computation for big crowds, though it won't make your renders themselves any faster.

4.5.4 Shaders and Materials

Just as the geometry and textures should be adjusted for lower-resolution models, there are lots of ways you can improve the efficiency of your shaders for lower-resolutions models so they will render quickly. Displacement can be replaced with bump, or eliminated entirely. Color maps can be replaced with noise or even a constant color. Complex lighting models can be replaced with simpler ones, or even no lighting at all!

For example, the terrain shader used in the Pod Race sequence of *Star Wars Episode I: The Phantom Menace* used a complex blend of warped tiled textures for the foreground, but off at the horizon where it was almost entirely covered up with haze and dust and heat-ripple, it dissolved to a constant color with no texture or lighting at all!

Another way you can simplify your shaders is by baking a lot of procedural shading into texture maps. This can be done using the "Renderman shader baking" technique presented by Jon Litt and myself at the Siggraph 2001 Renderman User's Group meeting. Although the details of this technique aren't available in published form yet, we hope to make them available before Siggraph 2002.

4.5.5 How many levels are enough?

The more shots you have with many creatures, the more benefit you will get from building multiple levels of detail. As a high-resolution model gets smaller and smaller on screen, you start to pay a higher and higher price per pixel, and at some point you want to switch completely away from that hires model. I call this the "pain threshhold"!

Your pain threshhold may be different from mine, but my general rule of thumb is *factors of 4*. Once my hires model is reduced in size to one quarter the number of pixels it was designed to be seen at, I'm spending about 16 times as much cpu time per pixel as I need! So I want to have a different level of detail for every factor of 4 in pixel size.

Here's a breakdown of resolutions for a fictitious hero critter, along with the type of representations that might be needed at each resolution. Note that the names actually include an indication of the intended screen-size; I've found that if you just use abbreviations like "hi", "med", and "lo", there can be a lot of confusion about how much detail should be included:

critter.1k - full hires model, paint and shaders

critter.2c - reduced resolution, paint, removed detail maps

critter.50 - merged geometry, removed/combined paint maps, replaced some shaders

critter.10 - block models? sprites?

critter.2 - dots?

4.5.6 How do I determine the exact detail ranges?

Even if you are targeting a specific resolution for a given model, it will either hold up better or worse than you expected. Finding the exact detail ranges to switch between models is a delicate art. But you don't have to do a lot of expensive render tests to figure out where the boundary should lie: Just render one zoom-in sequence of each model with paint and shaders, etc., then use a compositing system to dissolve between levels. You can find out the detail ranges corresponding to each frame of your zoom sequence by using a dummy `Procedural` which just prints out the detail for each frame. You can find a code snippet in last year's Advanced Renderman course notes (listing 4.10) which does exactly that.

4.5.7 Summary

Figures 4.3 and 4.4 show models using some techniques from all of the above. The high resolution models on the left are used in closeups, and may be seen at resolutions over 1000 pixels high. The medium resolution models in the middle have reduced span counts, and some geometry has been omitted and/or merged with other pieces. The low resolution model on the right have omitted texture maps, a simplified shader, and grossly merged geometry: note the "mittens" instead of hands for the clone trooper. A fourth resolution, not shown here, was composed solely of 6-sided polyhedra for each limb, with no texture maps or procedural noise.

4.6 Rendering Tips and Tricks

4.6.1 Rendering in layers

One obvious way to simplify a large and complex scene is to split it up into sections. This can be done in a variety of ways. However, all of them usually improve memory usage at the expense of some performance penalty.

Geometric groups

This is the simplest approach, as your scene is probably organized into separate groups of geometry for animation purposes anyway. But it can also lead to inefficiencies, since you may have back layers which are almost completely obscured by front layers. Do you really want to render that whole layer if only a few pixels show up in the final image? One solution is to use holdout objects for the foreground; in *PRMan* and Entropy this will cause the background objects to be culled earlier, avoiding lots of geometry processing, texture access, and shader calls. But this adds another kind of overhead: Every time you render your background layer you're also rendering your foreground objects.

Figure 4.3: Class Clones. ©2002 Lucasfilm Ltd.

Hopefully you're using a simpler shader for holdout objects (usually all you need is displacement and opacity), but you still have to load all that geometry!

One trick you can use to avoid too much rerendering is to render your foreground layer first, then use its alpha matte as a holdout object for the background layer! You can do this using a shader on a clipping plane object (see Advanced Renderman, section 12.7), and the fractional matte capability in recent versions of *PRMan*. Where the foreground matte is solid, the renderer will try to cull objects behind the matte before they are shaded (or if they're procedurals, before they're even loaded!) resulting in a big potential savings for rendering the background layers.

Just to get you started implementing this on your own, the shader body would do something like this...

```
point Pndc = transform("NDC", P);
Oi = texture(map,xcomp(Pndc),ycomp(Pndc));
```

... and the RIB fragment would look something like this:

```
AttributeBegin
CoordSysTransform "screen"
Surface "alphamatte" "map" "fgalpha.txt"
Matte 1
Patch "bilinear" "P" [ -2 2 0  2 2 0  -2 -2 0  2 -2 0 ]
AttributeEnd
```

There is a big caveat to this technique... if you do exactly what I've described you'll end up with a thin line between your foreground and background elements when you composite them together. So in practice you'll want to shrink your foreground matte before using it in this way. This is easily accomplished in most compositing systems by repeated applications of min-filters.

Figure 4.4: Battle Droids. ©2002 Lucasfilm Ltd.

Split into tiles

You might choose to take a large scene and split it horizontally or vertically, rendering each tile on a separate cpu or just in sequence. While rendering tiles in sequence doesn't reduce the time it takes to render a frame, it may be possible to render frames with more geometry using the same amount of memory, since each tile only needs to load what's needed for its subframe.

This isn't hard to do; in fact there's a RIB request called `CropWindow` which causes only a given section of a frame to be rendered. If you're using procedurals this is an especially efficient way to render a scene with limited memory, since each subframe's RIB file isn't all that big, and each tile will only load the geometry it needs when it needs it. The only drawback is there will be some pieces of geometry that are effectively rendered two or more times, if they cross the boundary between tiles.

By the way, you may have heard a suggestion to split a scene in depth as well as horizontally and vertically, using clipping planes. I don't advise this unless you can be certain that no primitive will be split in half by one of these clipping planes: Because each portion of the object is independently pixel-filtered, it can sometimes be quite difficult to recombine renders split in this fashion.

Rendering sideways

Another approach to reducing memory usage for large shots is pretty simple (and pretty bizarre). Render them sideways (ie. rotated 90 degrees)! This doesn't *always* work, and when it does it isn't dramatically different or better than splitting into multiple crop windows, but we have found this useful for rendering crowd scenes at ILM.

Why does rendering sideways make any difference at all? Notice that *PRMan* renders its image in buckets, top to bottom and left to right. Each time it reaches a bucket with a procedural bounding box that intersects that bucket, it has to open up the procedural and load the geometry. But cinematic

scenes are usually much wider than they are tall: sometimes as much as 2.35 times wider! So as *PRMan* marches across the screen it must load lots and lots of geometry before it needs most of it: maybe even the entire scene. By rendering sideways you are attempting to postpone the loading of geometry until close to when it's really needed.

Usually this makes the biggest difference when rendering crowds. Crowds often consist of lots of procedurals arranged mostly horizontally across the screen. Rendering the scene in the "usual" orientation may cause every one of those procedurals to be loaded after the first row of buckets! By rotating the render sideways, the creatures are loaded as needed, and the geometry is disposed of after all its pieces have been rendered.

(Note: Entropy uses a different strategy for rendering buckets, where it may choose to render them out of order. Although we haven't done any tests to find out, it seems likely that rendering sideways using Entropy might not make as big a difference in memory usage as it does in *PRMan*.)

4.6.2 Motionfactor

Although motion blur has a reputation for being expensive, there's an option you can insert inro your rib file that can sometimes make your motion-blurred objects render faster than everything else! This underadvertised "magic bullet" is called "motionfactor":

```
Attribute "dice" "motionfactor" [1]
```

(Both entropy and *PRMan* also support an older form using the `GeometricApproximation` RIB request. But the newer `Attribute` format is recommended.)

So what's the magic? This attribute causes objects to automatically adjust their shading rate depending on how fast they are moving across the screen! This tends to work because the object is getting blurry anyway, so you probably won't notice if it gets a little blurrier than you expected. Of course, you could do this manually if you wanted, but it's much easier to set the shading rate so that it looks good when not in motion, and let the renderer adjust it for fast-moving pieces.

The motionfactor value is an arbitrary positive value: you can increase or decrease it to enhance or diminish this effect. The exact formula that's used to adjust the shading rate isn't published, but a motionfactor value of 1 tends to work fine for most scenes.

The only exception may be when you have very tiny spots or streaks or other high-contrast features in your textures. In the speeder chase sequence of *Star Wars Episode II: Attack of the Clones*, we found that the tiny windows in the texture maps of the giant Coruscant buildings were getting blurred out of existence when the camera moved too fast! Since the cameras in this sequence were always shaking around a lot, this resulted in a stroby appearance, where the lights would turn on and off seemingly in sync with the speed of the camera motion. It took a while to track this down but Jon Litt eventually discovered that motionfactor was the culprit. So we had to disable motionfactor for most of the backgrounds in that sequence.

In spite of the occasional difficulty, motionfactor is *so* helpful at keeping render times under control that we recently made it our default for every object in all our renders. It's my #1 favorite timesaving trick!

4.6.3 Diagnosing texture thrashing

Sometimes you'll get a scene which takes forever to render, and for no apparent reason. Sure, there's lots of geometry and textures, but the shaders are pretty simple, and the shadow buffers render in about 10 minutes. Why does my beauty render take 5 hours?!?

Well, there could be a number of reasons, but one that I always look into first is texture thrashing.

Both Entropy and *PRMan* use a texture cache to avoid loading all the textures into memory simultaneously. The way this works is when you request a texture pixel, the renderer looks first in the cache to see if it's already loaded. If not, it loads that pixel (along with a bunch of nearby pixels)

into the cache. But here's the problem: If the texture cache is full when this happens, the renderer has to throw away some pixels already loaded into the cache. It tries to throw away the "oldest" pixels first in the hopes they're no longer needed, but there's no guarantee they won't be needed again soon! If they are, they will have to be loaded from disk again.

What this means is that even if you have 500M of texture data in your scene, you can end up reading the same parts over and over again into memory, resulting in gigabytes and gigabytes worth of disk access! This can really slow down a render, especially if the texture data is on a file server and not on the machine you're rendering with.

If your textures are almost entirely procedural this probably won't affect you. But ILM has always relied very heavily on painted and sampled textures, and the availability of good 3d painting tools has made this practice even more widespread throughout the industry. (We hear rumours that even Pixar sometimes paints textures nowadays!) The default texture cache sizes in *PRMan* and Entropy are just too small for heavily texture-mapped scenes: 16M and 10M respectively. You can increase these values using

```
Option "limits" "texturememory" [64000]
```

This sets the texture cache to 64M (it's measured in Kb), which is a much more reasonable size for modern computers with 512M-1G of real memory. Note that this specifies a *maximum* cache size, so it might be okay to increase this number even for relatively small scenes: the renderers won't use the memory unless they really need it.

4.7 Unsolved problems with Renderman's level of detail

This section is labelled "unsolved problems" for a reason: I don't have any good answers for them! I've included them here as a way to cast the gauntlet for the big brains at Pixar and Exluna to come up with brilliant ways to address them.

Level of detail is your friend, but the Renderman implementation of level of detail can also be your enemy. If you've worked at all with big crowd scenes using level of detail models you've had the following problem:

You've got a camera right in the middle of a big crowd that's lit by direct sunlight. There are creatures both in the foreground and background, at all levels of detail. But in your sunlight shadow buffer, which is rendered from an orthographic camera, they're all the same size, and all the same level of detail! So your foreground hero creatures are using shadow buffers rendered with lores versions of those same creatures, and they don't line up! So you get weird self-shadowing artifacts on your hero creatures.

The workaround is to split your sunlight shadow buffer into several different buffers, isolating each to shadow just the creatures in a particular layer, at an appropriate resolution. Or, in worst cases, you don't use Renderman's level of detail functionality at all and choose a specific model resolution manually for each creature, so that it's the same across all your shadow buffers. These aren't really satisfactory answers, and they can turn really hard shots into excruciatingly difficult ones, just because of the number of lights and buffers and creatures you may have to keep track of.

(In fact this problem extends beyond just levels of detail; the pixels in your shadow buffer aren't dense enough in some places, and are too dense in others. So even without levels of detail you may have difficulty getting your sunlight shadow buffer resolution high enough to work for your foreground creatures. In Entropy you can always ray-trace your shadows, but this can be expensive, especially for very large scenes.)

Another difficulty comes up when you start ray-tracing with different levels of detail using Entropy. You would probably like to use a lower level of detail for your ray-traced reflections than for the camera-visible geometry, especially if your reflections are very blurry. But the Renderman spec doesn't provide a good mechanism for this. In this case, you have to insert both levels of detail into

your rib, making the lower resolution visible to reflections and the higher resolution visible to the camera:

```
AttributeBegin
Attribute "visibility" "camera" [1]
Attribute "visibility" "reflection" [0]
...insert hires model...
AttributeEnd

AttributeBegin
Attribute "visibility" "camera" [0]
Attribute "visibility" "reflection" [1]
...insert lores model...
AttributeEnd
```

But this has some of the same problems as with shadow buffers! Because your reflections are seeing a different model than the camera, you may get grossly incorrect self-reflections. Ideally you'd like to tell the renderer something like this: "for self-reflections use the hires model, but for reflections of other objects use their lores model." There's no way at present to do that.

4.8 Conclusion

I hope this has been a useful summary of techniques for working with massively complex scenes. The Renderman specification is one of the few scene description formats which explicitly handles scenes which don't fit in memory, but it takes some careful forethought to take the best advantage of it and the renderers which support it.

Acknowledgements

Thanks to Christophe Hery and Doug Sutton for pioneering most of the methods described here during the production of *Star Wars Episode I*. Also thanks to Craig Hammack, Tommy Burnette, Jon Litt and others, for extending them to handle the added demands of *Episode II*.

All images are courtesy of Industrial Light and Magic.

Chapter 5

Production-Ready Global Illumination

Hayden Landis,
Industrial Light + Magic

5.1 Introduction

Global illumination can provide great visual benefits, but we're not always willing to pay the price. In production we often have constraints of time and resources that can make traditional global illumination approaches unrealistic. "Reflection Occlusion" and "Ambient Environments" have provided us with several reasonably efficient and flexible methods for achieving a similar look but with a minimum of expense.

This chapter will cover how Industrial Light and Magic (ILM) uses Reflection Occlusion and Ambient Environment techniques to integrate aspects of global illumination into our standard RenderMan pipeline. Both techniques use a ray-traced occlusion pass that is independent of the final lighting. The information they contain is passed on to our RenderMan shaders where they are used in the final lighting calculations. This allows lighting and materials too be altered and re-rendered without having to recalculate the occlusion passes themselves.

We will show that when used together these two techniques have given us an integrated solution for realistically lighting scenes. They allow us to decrease setup time, lighting complexity, and computational expense while at the same time increasing the overall visual impact of our images.

5.2 Environment Maps

Reflection Occlusion and Ambient Environments are most effective when used with an accurate environment map. We will also explain how standard light sources benefit from using these techniques, but environment maps still remain the easiest and most accurate way of lighting with either technique. So let's take a moment to discuss how we go about creating these environments.

Whenever possible we have traditionally shot a reflective "chrome" sphere and a diffuse "gray" sphere on location as lighting reference (see Figure 5.1). They provide a way of calibrating the CG lighting back at ILM with the lighting environment that existed on location.

Figure 5.1: Chrome sphere, gray sphere, and Randy Jonsson on location in Hawaii.

While the gray sphere is most often used simply as visual reference, the chrome sphere can be applied directly to our CG scene as a representation of the surrounding environment. As shown in Figure5.2, we take the image of the chrome sphere and unwrap it into a spherical environment map. This allows us to access the maps with a standard RenderMan environment call. At times unwanted elements, like the camera crew for example, can be found hanging out in the reflection. These are easily removed with a little paint work (see Figure 5.2, right).

Figure 5.2: Unwrapped chrome sphere environment texture and final painted version.

More than just a reference for reflections, the chrome sphere and it's resulting environment map give us a reasonably complete representation of incoming light on the location it was shot. If no chrome sphere exists then it is up to the artist to construct their own environment maps from the background plate or other photographed reference. It is also possible to use running footage in environment maps to give you interactive lighting based on events taking place in the shot.

HDR Images

I'm sure someone out there is asking: "What about High Dynamic Range Images?" There is no reason we can't use HDR images with either of these techniques. However, in practice we are usually lucky to get just a single chrome sphere image from location let alone a series of calibrated exposures. I'm sure that sometime in the future we will start to use HDR images more often but they are not always necessary. We have found that the single chrome sphere image can work just fine in many cases.

There is the issue of how to represent very bright areas and highlights in these environments. Once you adjust reflection levels properly for an environment you often notice that you've lost the brightest highlights and reflective areas of the environment. They become dull and washed out. We have come up with a useful trick, shown in Listing 5.1, that allows us to retain these intense reflection highlights. This works by taking the brightest parts of the image and expanding them to a set intensity. While not as necessary for Ambient Environments, this technique comes in very handy for reflection environments.

Listing 5.1 Example of expanding select ranges of an environment map.

```
float expandDynRange = 1.00; /* Expand specified range of the map to this max value.*/
float dynRangeStartLum = 0.50; /* Starting luminance for expansion to begin. */
float dynRangeExponent = 2.00; /* Exponent for falloff */
color Cenv = color environment (envMap, R, "blur", envBlur, "filter", "gaussian");
if (expandDynRange > 1) {
    /* Luminance calculation, 0.3*Red + 0.59*Green + 0.11*Blue.*/
    float lum = 0.3*comp(Cenv,0)+0.59*comp(Cenv,1)+0.11*comp(Cenv,2);
    if (lum > dynRangeStartLum) {
        /* remap lum values 0 - 1*/
        lum = (lum - dynRangeStartLum)/(1.0 - dynRangeStartLum);
        float dynMix = pow(lum,dynRangeExponent);
        Cenv = mix(Cenv, Cenv*expandDynRange, dynMix);
    }
}
```

5.3 Reflection Occlusion

First developed during *Speed II* and enlisted into full time service on *Star Wars: Episode 1*, Reflection Occlusion has become an important tool for creating realistic looking reflections.

When you use an all encompassing reflection environment you have the problem of occluding inappropriate reflections. Single channel Reflection Occlusion maps, like those shown in Figure 5.3, allow us to attenuate reflections in areas that are either self occluding or blocked by other objects in the scene. As illustrated in Figure 5.4, our surface shaders read these occlusion maps and then attenuate the environment to provide us with more realistic reflections.

Figure 5.3: Example of Reflection Occlusion passes: B25 and Spinosaurus.

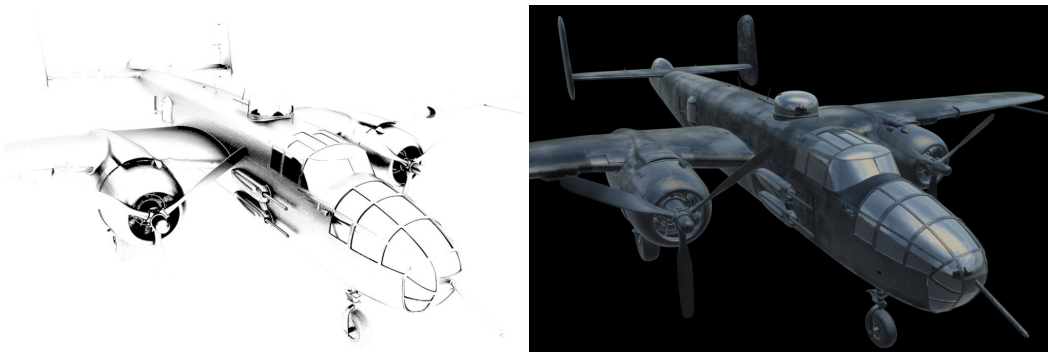


Figure 5.4: B25 rendered with reflections (left), and with the addition of reflection occlusion (right). ©Lucas Digital Ltd. LLC. All rights reserved.

Reflection Blur

Reflection blur is another important component of Reflection Occlusion and is used to help simulate various surface textures in the model. It is achieved by jittering the secondary rays around the main reflection vector. Mirror surfaces get little or no blur while matte surfaces receive a more diffused occlusion. As shown in Figure 5.5, glass surfaces, receive an almost mirror reflection while a rubber surface has a very diffused occlusion. For a proper occlusion, transparent surfaces should be made visible to primary rays but not secondary rays.

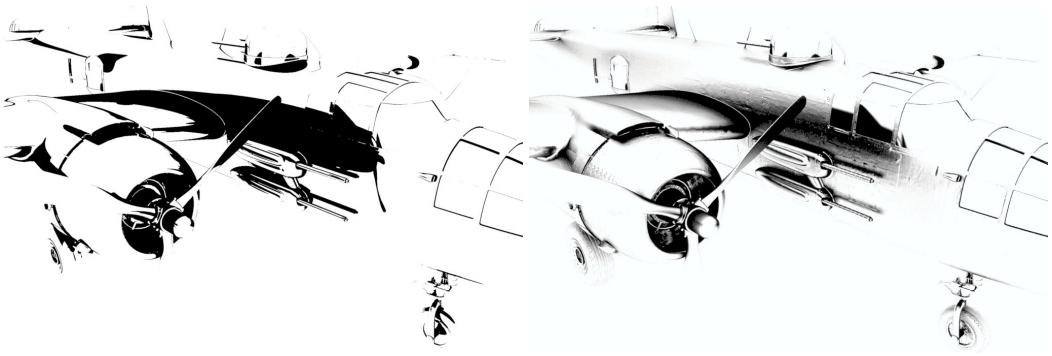


Figure 5.5: The image to the left shows a perfectly mirrored reflection occlusion. The image on the right shows differing blur amounts for reflective glass areas, diffuse paint, and more diffuse rubber tires. ©Lucas Digital Ltd. LLC. All rights reserved.

Reflection occlusion gives us some of the advantages of doing a full ray-traced reflection pass without all of the expense. As long as our animation doesn't change we can keep reusing the same occlusion pass for subsequent iterations of the final render. It allows us the convenience of using standard RenderMan environments and reflections but gives them the illusion of a more complex ray-traced scene. For reflective objects this solution allows us to bypass the expense and hassle of a full ray-traced render unless it's absolutely necessary. An example reflection occlusion shader is shown in Listing5.2.

Listing 5.2 `reflOccl.sl`: Example of an Entropy shader that produces a reflection occlusion image.

```
#include "entropy.h"

surface reflOccl_srf (float rflBlurPercent = 0.00;
                      float rflBlurSamples = 4.00;)
{
    Oi = Os;
    if (raylevel() > 0) {
        Ci = Oi;
    } else {
        float rflBlur = rflBlurPercent/100;
        normal NN = normalize(N);
        vector IN = normalize(I);
        vector R = reflect(IN,NN);
        float occ = environment ("reflection", R, "blur", rflBlur,
                                 "samples", rflBlurSamples);
        Ci = (1-occ)*Oi;
    }
}
```

5.4 Ambient Environments

Lighting with fill lights involves guesswork and can take up a great deal of time if done correctly. Ambient Environments is a technique that was developed to try and free us from the necessity of wrangling lots of fill lights.

There are two components to an Ambient Environment. “Ambient Environment Lights” provide illumination while “Ambient Occlusion” provides shadowing and important directional information for looking up into the Ambient Environment map. Similar to Reflection Occlusion, Ambient Environments also use a pre-rendered occlusion map accessed at render time to give our scene realistic shadowing. We can conveniently use the same environment map for both our ambient environment and our reflection environment.

Traditionally ambient lights have never been too popular in production because they simply add a single overall color, a less than spectacular lighting effect. By naming this technique “Ambient Environments,” we hope to help restore the good name of the much maligned ambient light.

Developed initially during *Pearl Harbor* this technique quickly spread to other shows and has since become an important lighting tool for most productions at ILM.

5.4.1 Ambient environments defined

ambient (’am-b{e-}-*nt)
Etymology: L i[ambient-], i[ambiens], prp of i[ambire] to go around, fr. i[ambi-] + i[ire] to go – more at ISSUE aj, surrounding on all sides: ENCOMPASSING

An “ambient environment” represents the diffuse fill light that surrounds an object. It’s not intended to represent direct light sources. These are left to standard RenderMan lights. The Ambient Environment’s job is to give us indirect “bounce” or “fill” light from the environment. Rather than setting up multiple fill lights and guessing their color, intensity and direction, an ambient environment light provides all of this relatively free. It’s also necessary to provide shadowing from the surrounding lighting environment. Points not fully exposed to the environment need to be attenuated properly. This process is known as “Ambient Occlusion.” An example of the Ambient Environment process is shown in Figure 5.6.

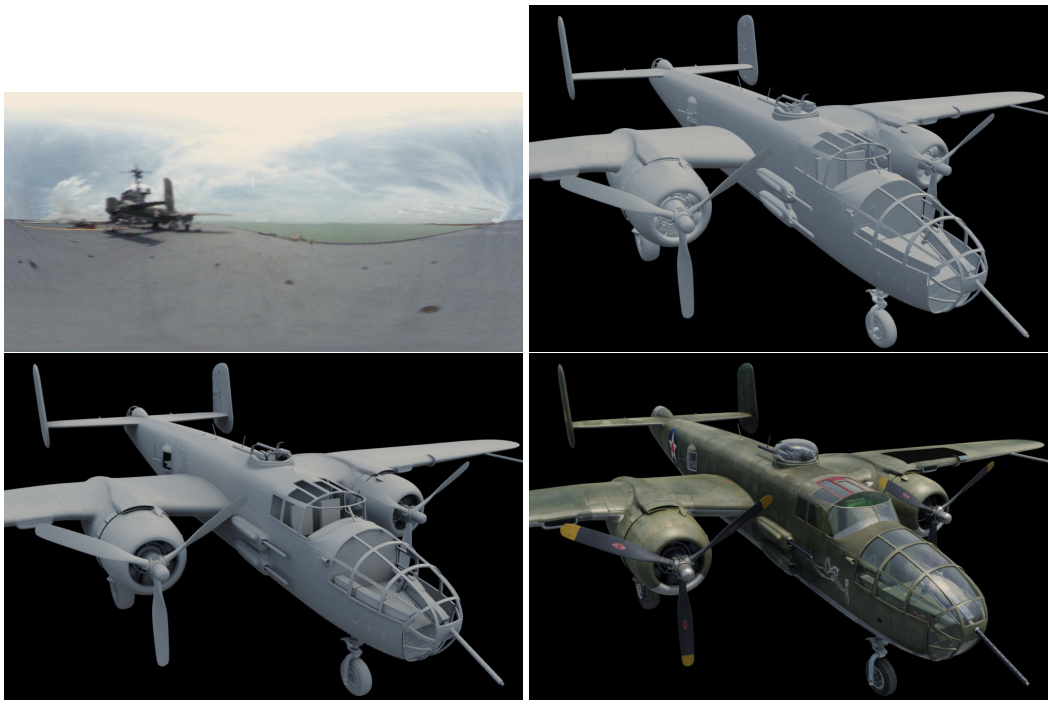


Figure 5.6: Simple example of Ambient Environment process. Top: environment map and plastic B25 illuminated with only an Ambient Environment Light. Bottom left: Ambient Environment Light with occlusion and ”bent normals”. Bottom right: The B25’s final beauty render.

5.4.2 Why ambient environments?

There are several advantages of using this method over traditional fill lighting techniques.

- Using a chrome sphere gives you a more accurate representation of the environment than placing fill lights by hand. There is little guess work involved and you get the exact environment as reflected in the chrome sphere.
- The light completely surrounds an object. No dark patches or areas of missing illumination.
- It is very efficient to set up and adjust - one light, one map.
- Fast! One ambient environment light replaces 3 or more fill lights. There are no multiple shadow passes to render, only a single Ambient Occlusion pass is required. You save the time it takes to compute the additional lights and shadow passes.
- View independent. If the environment’s orientation changes there is no need to re-render the occlusion pass. If “baked” occlusion maps exist for a model, no shadow or occlusion renders are necessary except for your key light and any other direct ”hard shadowed” light sources. Baked maps also free you from any dependence on camera or object orientation.

5.4.3 Ambient environment lights

An Ambient Environment Light is simply a modified environment reflection. Rather than using the reflection vector

```
R = reflect(IN,NN);
```

that we normaly use with an environment map, an Ambient Environment uses the surface normal

```
R = normalize( faceforward(NN,I) );
```

which is the direction of the greatest diffuse contribution, to gather illumination from the environment. This is illustrated by the top two images of Figure 5.7.

Since the RenderMan environment call conveniently blurs across texture seams, we can apply a large map blur value (25%-30%) rather than sampling the environment multiple times as you might do with a ray-traced approach. The blurred lookup into the environment map represents the diffuse contribution of the environment for any point on the surface (see bottom image, Figure 5.7). This has the speed advantage of sampling our environment only once rather than multiple times.

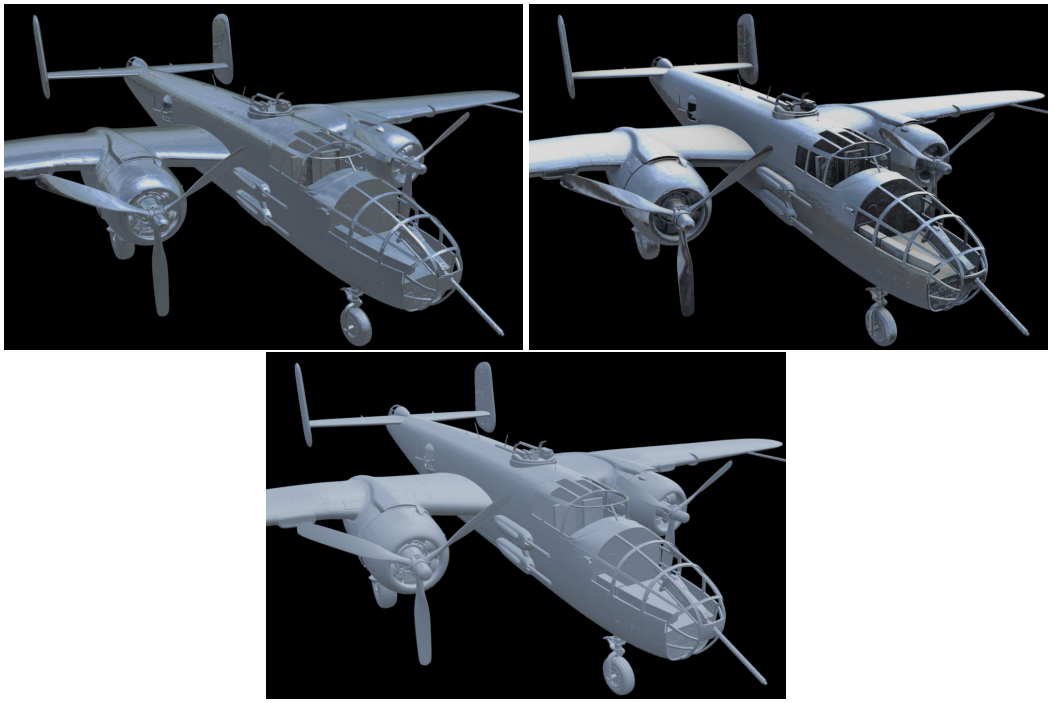


Figure 5.7: Top Left: Regular reflection environment lookup. Top right: Environment lookup using surface normal. Bottom: Environment lookup using surface normal with 30% blur.

5.4.4 Ambient occlusion

Ambient occlusion is a crucial element in creating a realistic ambient environment. It provides the soft shadowing that we have come to expect from global illumination and other more complex indirect lighting techniques. Surfaces not fully exposed to the environment need to be attenuated properly so that they do not receive the full contribution of the ambient environment light. This is one of the main attractions of using the Ambient Environment technique. In Figure 5.8 you can begin to see some of the subtle visual cues that will eventually help convince us that the lighting is real.

Figure 5.8: Example Ambient Occlusion images. B25 bomber, Spinosaurus and Tyrannosaurus. ©Lucas Digital Ltd. LLC. All rights reserved.

In order to get this effect, it is necessary to have an ambient occlusion render or "baked" ambient occlusion maps that represent this attenuation of outside light. Ambient occlusion is achieved through the following process: For every surface point, rays are cast in a hemisphere around the surface normal. The final occlusion amount is dependent on the number of rays that hit other surfaces or objects in the scene.

Figure 5.9: Simple illustration of surface sending out rays, some of which are blocked. Perhaps from the B25 fuselage under the wing. Showing blocked rays from wing and engine nacelle. ©Lucas Digital Ltd. LLC. All rights reserved.

Since the strongest diffuse contribution comes from the general direction of the surface normal, the result is weighted to favor samples that are cast in that direction. If there is an object directly parallel to the surface it will be occluded more than if the same object were placed to the side. Transparent or glass materials should be excluded from the Ambient Occlusion render. If you have opacity maps you want to make sure that your ambient occlusion shader takes this into account. This pass can be rendered each frame for objects with internal animation. For solid objects with few moving parts it can be rendered once and baked into texture maps. Baking the occlusion maps gives you a huge advantage since they only need to be rendered once per object. This works because unlike Reflection Occlusion, Ambient Occlusion is not dependent on orientation of the object or environment. You can share the same maps amongst multiple instances of an object and in any scene.

Bent normals

Another important component of Ambient Occlusion is the addition of an "average light direction vector." This represents the average direction of the available light arriving at any point on the surface. The unoccluded rays from the initial occlusion calculation are averaged together to find the difference between this "average light direction vector" and the original surface normal. This offset is stored in the G, B and A channels of the Ambient Occlusion map (see Figure 5.10, right). This vector is used to redirect the lookup into the ambient environment so that the color is gathered from the appropriate direction. The surface normal originally used to lookup into the environment texture will now be bent at render time to point in this new direction (see Figure 5.11). We use the term "bent normals" to refer to this effect since it is difficult to say "average light direction vector" ten times fast.

These two components of the Ambient Occlusion render combine to give us realistic shadowing as well as an accurate lookup into the Ambient Environment texture.

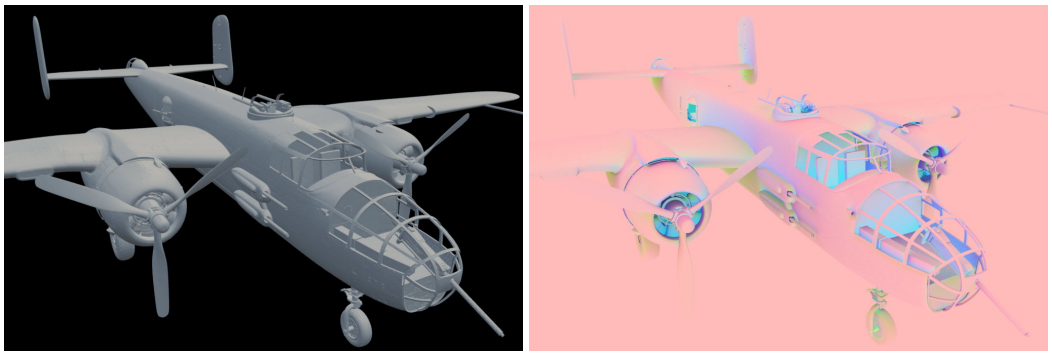


Figure 5.10: Raw materials: Ambient environment light render and an Ambient Occlusion render, representing the stored occlusion and bent normal data. ©Lucas Digital Ltd. LLC. All rights reserved.

Figure 5.11: Final product: Plastic render of the B25 with an occluded ambient environment light. The image on the right shows the final step of integrating the “bent normals.” ©Lucas Digital Ltd. LLC. All rights reserved.

Listing 5.3 is an example of an Ambient Occlusion shader. Listing shad:hayden:bendnorms contains a pseudocode example of how we convert the bent normals stored in the Ambient Occlusion maps back into a normal that the surface shader can use.

5.4.5 Other Ambient Environment light types

We can use the occlusion and directional information contained in the Ambient Occlusion map and apply it to other light sources as well. If you take a standard point or spot light, pass it the ”bent normal” rather than the original surface normal and then attenuate it with the occlusion channel, you will get a nice soft fill light source with no additional shadowing necessary (see Figure 5.12). This makes it fairly cheap to add lights to an object already using Ambient Occlusion. These additional lights allow you to add or subtract light from the base ambient environment.

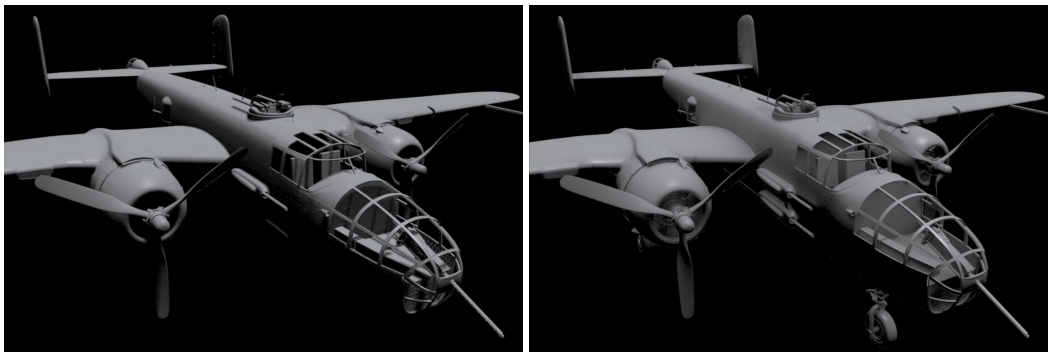


Figure 5.12: Example of a spot light using standard shadows and a spot light using occlusion and bent normals for shadowing.

5.4.6 Other uses for Ambient Occlusion

We've found several other useful applications for ambient occlusion. One of these is for creating contact shadows (see Figure 5.13). Hiding shadow casting objects from primary rays but not secondary rays allows us to create contact shadows for objects that can then be applied in the render or composite.

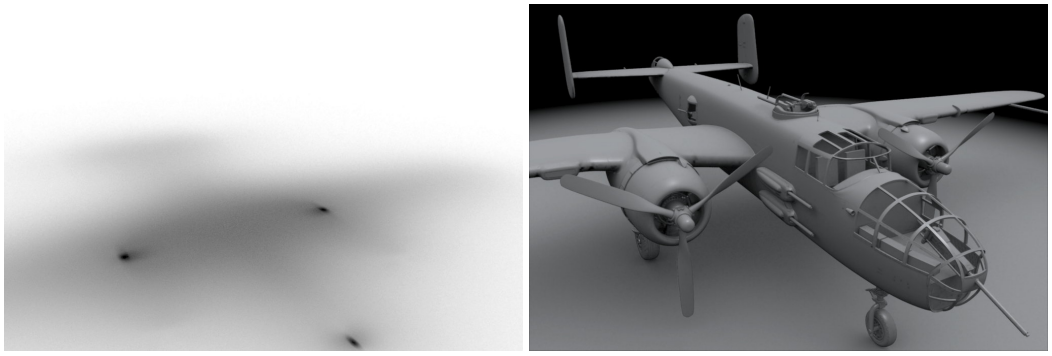


Figure 5.13: Example of Ambient Occlusion contact shadow.

5.5 Application

Combining Reflection Occlusion and Ambient Environments has allowed us to realistically light complex scenes with a minimum of effort. The lighting of many final scenes has been accomplished by using only three lights: Key, reflection, and Ambient Environment lights.

Figure 5.14 shows an example from *Pearl Harbor*. This shot required us to place 14 computer generated B25 bombers next to four real B25 bombers on the deck of an aircraft carrier in Texas. Then we had to place this landlocked carrier it in the middle of the Pacific Ocean.

Figure 5.14: From *Pearl Harbor*, two frames from the establishing shot of Doolittle's raid. ©Lucas Digital Ltd. LLC. All rights reserved.

One key to the success of this shot was the development of materials that were created in a calibrated lighting environment. This environment was built using the gray and chrome sphere references as well as photos of the real B25 bombers. If this is done correctly you can drop the model and its' materials into any other environment and soon have it looking right at home.

Armed with an environment map, reflection occlusion pass, baked ambient occlusion maps and a good set of materials, it took only part of a day to tweak the final lighting for this shot. Only three lights: Key, reflection environment, and ambient environment were used. Not every shot goes this smoothly but it is a testament to the ease of using this simple but effective lighting setup.

On *Jurassic Park III* we had the task of creating realistic dinosaurs and used Ambient Environments to create several interesting effects in addition to their regular lighting tasks. By adding running footage of flames to an environment, a realistic and interactive lighting effect was created for the Spinosaurus in this shot (see Figure 5.16).

5.6 Conclusion

I am sure that at some point we will be ray tracing complex scenes on our palm pilots. Until then we'll continue to create efficient cheats and tricks to get the visual advantages of global illumination

Listing 5.3 `occlusion.sl`: Entropy Ambient Occlusion example shader.

```
#define BIG 1e20

color vector2color(vector v) {
    return ((color v) + 1) / 2;
}


surface occlusion (float samples = 16;
                   float doBendNormal = 0;)
{
    normal NN = normalize(N);
    vector up = vector(0,1,0);
    float sum = 0;
    float i;
    vector dirsum = 0;
    vector side = up ^ NN;
    for (i = 0; i<samples; i = i+1) {
        float phi = random()*2*PI;
        float theta = acos(sqrt(random()));
        vector dir = rotate(NN,theta,point(0),side);
        dir = rotate(dir,phi,point(0),NN);
        point Ph;
        normal Nh;
        if (rayhittest(P,dir,Ph,Nh) > BIG) {
            sum = sum + 1;
            dirsum = dirsum + dir;
        }
    }
    sum /= samples;
    dirsum = normalize(dirsum);
    if (doBendNormal != 0) {
        vector bend = dirsum - NN;
        Ci = vector2color(bend);
    } else {
        Ci = sum;
    }
}
```

Figure 5.15: Frames of the B25 in its look development environment composited over reference photos of the real B25.

Figure 5.16: We used running footage from this background in our ambient environment map to create interactive fire light on the Spinosaurus. ©Lucas Digital Ltd. LLC. All rights reserved.

without the time and expense.

We continue to expand on the basic concepts of Ambient Environments. Some colleagues at ILM have already been busy adding features that allow for self illumination and other more advanced lighting effects. At some point I am sure this will eventually migrate to full blown global illumination.

For some shots it is true that you can get away with only the minimum of these lighting tools alone. However, in production we unfortunately know that "real" is never quite good enough. For other shots these techniques are not the "end all, be all" but a solid foundation on which to build your final lighting. We hope you find these techniques useful and as much fun to work with as we have had in developing them.

Acknowledgements

I would like to thank the following people:

Dan Goldman and Christophe Hery for their assistance in preparing this chapter; Thanks to Dan as well for the Entropy shader examples; Dawn Yamada and the ILM PR department; Simon Cheung for the great B25 model and Tony Sommers for his wonderfull paint work.

Reflection Occlusion has a long list of contributors. It was originally developed by Stefen Fangmeier on *Speed II*. Reflection blur implemented by Barry Armour. Further work by Dan Goldman on *Star Wars: Episode I* provided speed and quality optimizations. Other contributors include: Ken McGaugh and Will Anielewicz.

Ambient Environments would never have gotten off the ground without the hard work and inspiration of Ken McGaugh and Hilmar Koch. Thanks guys! Further refinements have been contributed by Kevin Sprout, Dan Goldman, and Doug Sutton.

Thanks to everyone at ILM for your hard work and dedication. Without your imagery I'd be using tea pots and mandrills. Not that there is anything wrong with that...

Listing 5.4 This is an example of how we convert the information stored in the Ambient Occlusion images back into occlusion and ”bent normal” data that can be used by the surface shader.

```
uniform float ambOcclDensity; // Amount of the original occlusion you want
                              // to mix in (usually 1)
vector IN = normalize(I);
normal NN = normalize(N);
normal bentNormal = normalize( ntransform("object", faceforward(NN,IN)) );

color ambBendNormColor = 0;
Pndc = transform("NDC",P);
ambS = xcomp(Pndc);
ambT = ycomp(Pndc);

ambOccAmt = float texture(ambOcclMap[0], ambS, ambT);
ambOccAmt = mix(1, ambOccAmt, ambOcclDensity);

ambBendNormColor = color texture(ambOcclMap[1], ambS, ambT);
/*This line actually reads the "bent normal" channels 1-3 of the map,
  not just channel 1 as it might appear. */

normal addNormal = -1 + 2 * (normal ambBendNormColor);
bentNormal = normalize(bentNormal + addNormal);

R = vtransform("object",environmentSpace,bentNormal);
R = normalize(R);
Cenv = color environment (envMap, R, "blur", blurAmount, "filter", "gaussian");


/* Note: We have come across a problem with the way RenderMan blurs
 * across texture seams.  Sometimes this shows up as a pixel wide line
 * that appears in the results of your environment() calls. If you do run
 * into this problem the following is a workaround.
 */
R = vtransform("object",environmentSpace,R);
vector R1 = R + Du(R)*du;
vector R2 = R + Dv(R)*dv;
vector R3 = R1 + R2 - R;
color Cenv = color environment (envMap, R,R1,R2,R3, "blur", blurAmount,
                                "filter", "gaussian");
```

Renderman on Film

Combining CG & Live Action using Renderman
with examples from *Stuart Little 2*

Rob Bredow
Sony Pictures Imageworks
rob@185vfx.com

Abstract

We present a complete (albeit brief) summary of the digital production process used for creating computer generated images in the context of a live action motion picture citing examples from the films *Stuart Little* and *Stuart Little 2*. Special attention is paid to issues relating specifically to Renderman including considerations for *shading*, *lighting* and *rendering* for use in feature film effects. We also touch on several compositing techniques required to complete the production process.

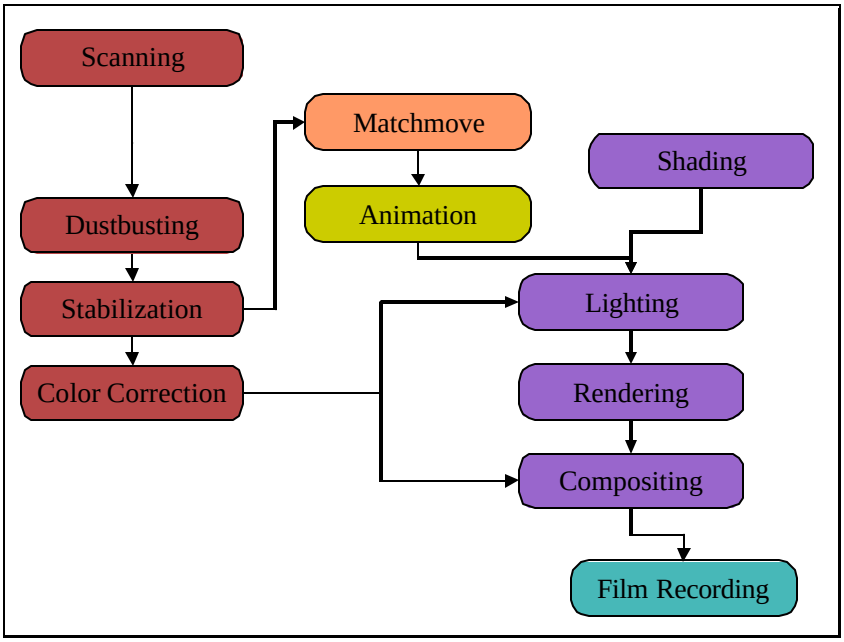


Figure 1 -Summary of the digital production pipeline

1.1 Pre-Renderman

There are several steps when working on a live action film that must take place before any rendering can begin. For the purposes of this course we will skip over the many aspects of live-action photography and editing and start with the digital production process.

1.1.1 Scanning

Once a shot has been in the editorial process as the "take" that will be in the movie and requires an effect, it can be scanned. This is generally done based on a series of keycode in and out points designated by the editorial department. Each frame is scanned individually and stored in the computer as a sequence of numbered images.

The digital film scanning process is designed to record as much of the information stored on the film negative as possible while considering the downstream ramifications of creating extremely large files for each frame. There are two aspects to consider when determining how much detail you are interested in preserving from the original negative: *Image resolution* and *color resolution*.

Image resolution is simply the number of pixels you want to generate in the scanning process. An image resolution of 4096 wide by 3112 tall is commonly referred to as a "4k" image and is often considered the highest useful resolution scan for today's 35mm motion picture negative.

Color resolution is the number of bits of data that you use to store each pixel. The most common standard of 8 bits for each of the red, green, and blue channel (a 24-bit image) yields only 255 shades of gray which is not sufficient to record all of the values that can be represented with film. A more complete representation can be stored by doubling the storage to 16 bits for each of the channels resulting in over 65,000 shades of gray which is more adequate to describe film's contrast range and subtlety. It is most common however to use the standard detailed by Kodak's Cineon file format (for more information see the Kodak paper “Conversion of 10-bit Log Film Data To 8-bit Linear or Video Data for The Cineon Digital Film System”). This standard is specified as scanning 10 bits of data per channel and storing the data in a logarithmic space to preserve as much detail in the areas of the curve that correspond to subtle gradations that the eye can clearly see.

Many visual effects films today will choose to work at a resolution of 2k and 10 bits per pixel. Often this level of detail is high enough that the differences between the work print generated off of the original camera negative and a print generated from the "digital" negative are not perceptible.

1.1.2 Dustbusting

Once the film has been scanned, it invariable has imperfections that need to be removed. These issues can manifest themselves as scratches on the negative which show up as white lines or dots, lens imperfections that need to be removed, or dark specs and lines that correspond to dust and hair. All of these are removed in the *dustbusting* process that consists of a series of both automated tools and careful frame-by-frame paintwork to prepare the plates for the downstream process.

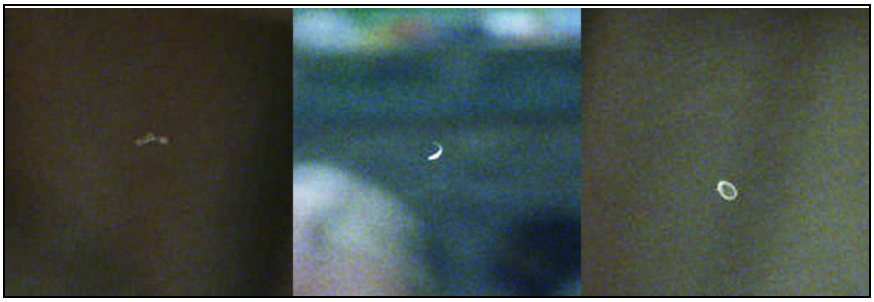


Figure 2 – Three samples of dust and scratches from negative damage

1.1.3 Stabilization

Even with advances in steadying both the in-camera and scanning technology, most every plate requires stabilization before it is ready to be enhanced with visual effects. Most frequently, the stabilization process is simply required to take out a small amount of weave inherent to the film shooting and scanning process. This can be done with an automated 2-d tracking system and some simple math to smooth the resulting curves that will "unshake" the footage on a frame-by-frame basis.

In some more complicated cases, there are bumps in the camera move or a shake introduce by a long boom arm or some other on-set tool that needs to be stabilized out or locked-down to improve the look of a shot. In this case, an entire suite of tools may be needed to stabilize and perspective correct the photography.

1.1.4 Color Correction

Scanning negative and viewing the resulting image directly does not generally result in aesthetically pleasing image. The director of photography (D.P.) shoots a movie with the process of printing in mind and being able to control the exposure level and color balance during that process. In order to be able to, later in the pipeline, match colors and exposures of computer-generated objects to the plate photography, the plate must be "timed" or color corrected to match the DP's requirements.

This is accomplished with a careful process of adjusting the exposure level and color balance with digital tools that emulate their real-world counterparts in the "print timing" process. These color corrections are then filmed out and approved by the director of photography before any lighting work can begin.

Figure 3 - Original scan (left) and color corrected plate (right)

1.1.5 Matchmove

The matchmove process is a fundamental step when working on a live action effects film. It involves duplicating, in the digital environment, the complete shooting environment that was used on the set including, most importantly, the camera.

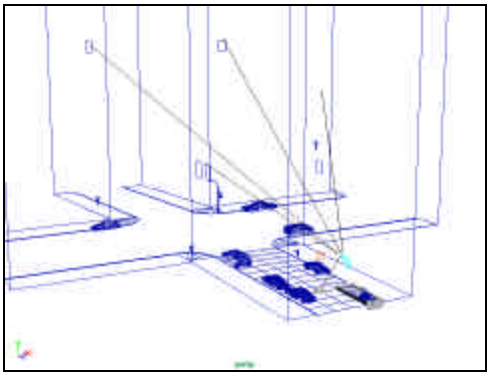


Figure 4 - 3d model of the live action set

First, a relatively simple model of the set is generated in the computer whose proportions match as closely as possible to the live action stage. This model will be used later in the process to help the animators know where to place the characters and, in the rendering process, it will catch shadows and reflections as needed.

The second major part is the tracking of the camera in relation to this model. This is a very precise and challenging task which consists of matching the lens and camera properties to the camera on set as well as the camera's position and orientation over time for each frame of the plate.

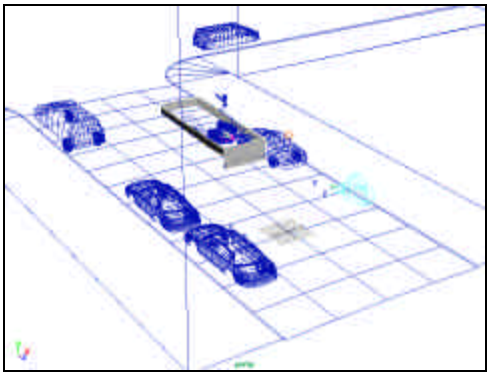


Figure 5 - Camera placed in the virtual set

There are many tools (some semi-automated) that help in both the reconstruction of the digital set and the tracking of the digital camera to this set. The specifics of which are too lengthy to detail here.

Figure 6 - Single finished matchmove frame

1.1.6 Animation

It would be naïve to attempt to summarize in any meaningful was what takes place in the animation step of the production process in our limited space. For our purposes, the design and movement of all the props and characters can now be accomplished and approved before any rendering can take place.

1.2 Renderman

It's at this point that the Renderman related processes can begin. The "look development" process gets things started by designing the shaders and setting the look for each of our characters and props. Once the look has been established, the lighting can begin and the CG will be lit to fit into the scene and then be enhanced for dramatic effect as needed. Then the process of rendering the various passes can be undertaken.

1.2.1 Shading

There is one key concept to keep in mind when writing shaders for visual effects: control. Most of the lighting models that are mo st useful in production are based more on the fact that a lighter can predict

what the shader will do in a given case and less on physics or mathematical accuracy. This methodology has lead to a class of lighting models that could be categorized as "Pseudo-realistic lighting models." We will detail a couple of useful lighting models that would fall into such a category here.

1.2.1.1. Diffuse Controllability: Falloff start, end and rate.

Diffuse shading calculations are the most common and simple way to shade an object. Creating a flexible diffuse shading model greatly increases controllability.

The figure below illustrates a standard Lambertian diffuse shading model.

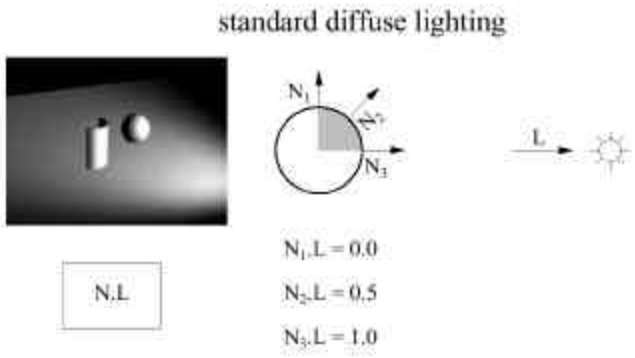


Figure 7 - Standard diffuse lighting model

1.2.1.1.1 Falloff End (Wrap)

The first step in generating soft and nicely wrapped lighting is to give the light the ability to reach beyond the 90 degree point on the objects. This has the effect of softening the effect of the light on the surface simulating an area light. This can be done with controls added to the lights which specify the end-wrapping-point in terms of degrees where a wrap of 90 degrees corresponded to the standard Lambertian shading model and higher values indicated more wrap. This control is natural for the lighting TD's to work with and yields predictable results.

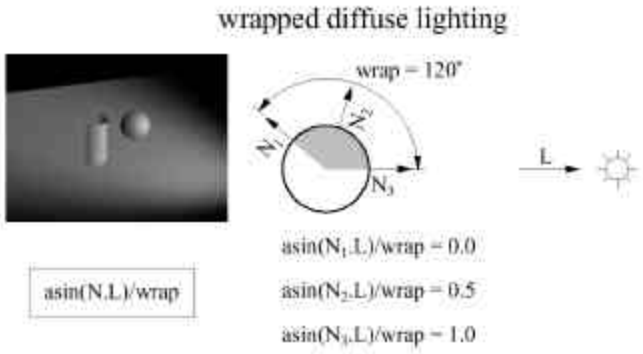


Figure 8 - "Wrapped" diffuse lighting

For wrapped lights to calculate correctly, the third argument to the illuminance() statement must be set to at least the same degree as the wrap value for the highest light. Otherwise lights will be culled out from the lighting calculations on the back of the surface and the light will not correctly wrap. In our implementation, the wrap parameter was set in each light and then passed into the shader (using message passing) where we used the customized diffuse calculation to take into account the light wrapping.

As an aside, wrapping the light "more" actually makes the light contribute more energy to the scene. If the desired effect is to keep the overall illumination constant, it will be necessary to reduce the intensity light control while increasing the wrap.

1.2.1.1.2 Falloff Start

The natural opposite of the "Falloff End" parameter which controls wrap is the "Falloff Start" parameter that controls the area that is illuminated with 100% intensity from a light. The "Falloff Start" parameter is also specified in degrees and is defaults to a value of 0. By increasing this value, the light's 100% intensity will be spread across a larger area of the object and the gradation from the lit to the unlit are of the object will be reduced.

The "Falloff Start" parameter has no logical counterpart in real life and has fewer uses than the "Falloff End" parameter previously discussed. The most common use that we found in production was in the case of a backlight or a rim around a character. When increasing the "Falloff End" or wrap to a high value, sometimes the rim would not look strong enough or have a hard enough falloff (since it roughly simulates the effect of an area light). By increasing the "Falloff Start", you get a sharper falloff around the terminator of the object and effectively increase the sharpness of the rim light.

1.2.1.1.3 Falloff Rate

Gamma is one of the most useful and convenient functions in computer graphics. When applied to lighting, a gamma function leaves the white and the black points unchanged while modifying the rate of the falloff of the light. This has the effect of modifying the value of the sphere at the “N2” point in the diagram above.

The apparent softness or sharpness of a light can be dialed in as needed using these three controls:

"Falloff End", "Falloff Start",and "Falloff Rate".

1.2.1.1.4 Shadows

The first problem we encountered when putting these controllable lights to practical use was shadows. When you have shadow maps for an object, naturally the backside of the object will be made dark because of the shadow call. This makes it impossible to wrap light onto the backside of the object.

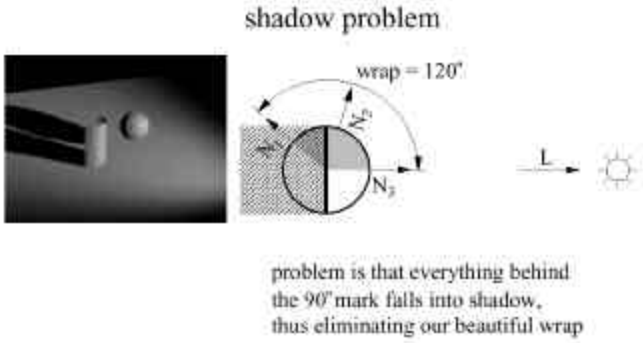


Figure 9 - The shadow problem with wrapped lights

Fortunately Renderman affords a few nice abilities to get around this problem. One way to solve this problem is by reducing the size of your geometry for the shadow map calculations. This way, the object will continue to cast a shadow (albeit a slightly smaller shadow) and the areas that need to catch the wrapped lighting are not occluded by the shadow map.

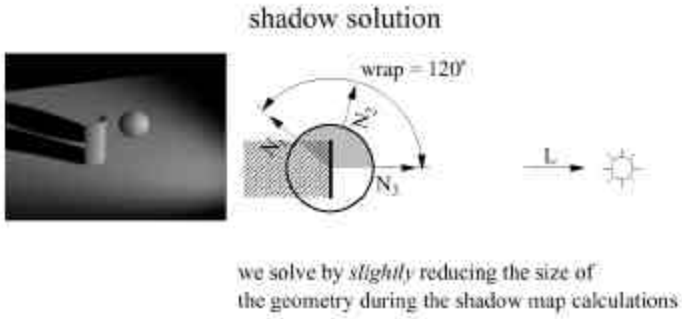


Figure 10 - The solution to the shadow problem

As a practical matter, shrinking an arbitrary object is not always an easy task. It can be done by displacing the object inwards during the shadow map calculations but this can be very expensive.

In the case of our hair, we wrote opacity controls into the hair shader that were used to drop the opacity of the hair to 0.0 a certain percentage of the way down their length. It is important to note that the surface shader for an object is respected during shadow map calculation and if you set the opacity of a shading point to 0.0, it will not register in the shadow map. The value that is considered

"transparent" can be set with the following rib command:

```
Option "limits" "zthreshold" [0.5 0.5 0.5]
```

Lastly and most simply, you can blur the shadow map when making the "shadow" call. If your requirements are for soft shadows anyway this is probably the best solution of all. It allows for the falloff of the light to "reach around" the object and if there is any occlusion caused by the shadow, it will be occluded softly which will be less objectionable.

For both *Stuart Little* 1 and 2, we used a combination of shortening the hair for the shadow map calculations and blurring our shadows to be able to read the appropriate amount of wrap.

1.2.1.2. Hair controllability

Solving the problem of how to properly shade hair and fur in a way that is both realistic and easy to control is a challenging project. In our work on *Stuart Little* we experimented with conventional lighting models before deciding to create a model that was easier to control.

When using a more conventional lighting model for the fur (essentially a Lambert shading model applied to a very small cylinder) we came across a few complications. First, a diffuse model that integrates the contribution of the light as if the normal pointed in all directions around the cylinder leaves fur dark when oriented towards the light. This may be accurate but does not produce images that are very appealing. In our testing, we felt that the reason that this model didn't work as well was because a lot of the light that fur receives is a results of the bounce of the light off other fur in the character which could be modeled with some sort of global illumination solution, but would be very computationally expensive.

The other major reason that we chose to develop a new diffuse lighting model for our fur was that it was not intuitive for a TD to light. Most lighting TD's have become very good at lighting surfaces and the rules by which those surfaces behave. So, our lighting model was designed to be lit in a similar way that you would light a surface while retaining variation over the length of the fur which is essential to get a realistic look.

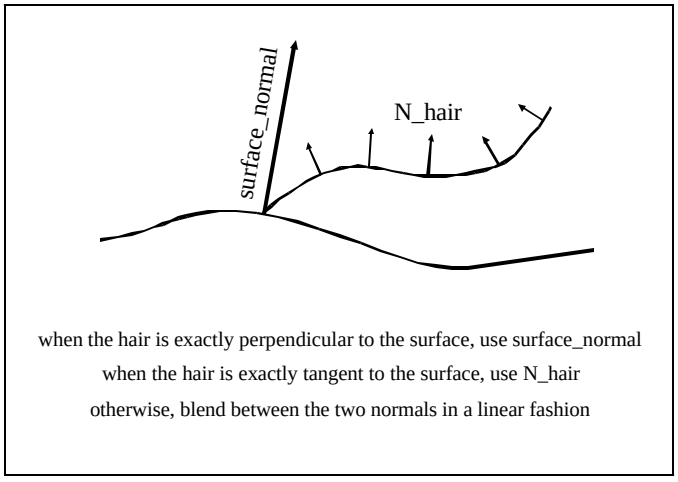


Figure 11 - Obtaining a shading normal

In order to obtain a shading normal at the current point on the hair, we mix the surface normal vector at the base of the hair with the normal vector at the current point on the hair. The amount with which each of these vectors contributes to the mix is based on the angle between the tangent vector at the current point on the hair, and the surface normal vector at the base of the hair. The smaller this angle, the more the surface normal contributes to the shading normal. We then use a Lambertian model to calculate the intensity of the hair at that point using this shading normal. This has the benefit of allowing the user to light the underlying skin surface and then get very predictable results when fur is turned on. It also accounts for shading differences between individual hairs and along the length of each hair.

Figure 12 - Finished *Stuart Little 2* rendering with fur shading

1.2.1.3. "Ambient Occlusion" technique

In the past, most computer graphics lighting techniques have relied on a very simple model to approximate the contribution from the ambient light which does not come from a specific light source but rather bounces around a set and lights an object from all directions. Because modeling all of the actual light bouncing throughout a set is very computationally expensive, many shaders substitute a constant for this ambient value (commonly referred to as the "Ka"). This constant is generally dialed in by the lighting artist and is a rough approximation of the light from the bouncing from nearby objects.

The disadvantages of using such a simple ambient lighting model are obvious. All the different sides of an object will get the same ambient contribution no matter which direction it is facing or what shape it is. That can lead to objects looking flat and uninteresting if the "Ka" is turned up too high.

One workaround that has been used for years in this area is to leave the "Ka" value set to 0.0 (or nearly 0.0) and use more lights to "fill" in the object from different directions. It is common when lighting for visual effects to have your primary lights that are the key, rim and fill and then complement those lights with a set of bounce lights from the ground and any nearby objects. This can produce very convincing results in the hands of a skilled lighting artist.

But in the case of rendering large flat objects with some small details (like a building), none of the previously mentioned techniques give desirable results. The subtleties that we are used to seeing in real life which include the darkening of surfaces when they are near convex corners and the soft blocking of light from nearby objects are missing and the visual miscues are obvious.

Figure 13 - Building with texture, key light, and constant ambient

The "Ambient Occlusion" technique that we used for *Stuart Little 2* gave us an accurate and controllable simulation of the ambient contributions from the sky and the ground on our computer generated objects. The technique, in concept, consists of placing two large area lights, one representing the ground and the other for the sky, into the set and accurately modeling the contribution of these lights on the object of interest.

These calculations are most convenient to implement with ray-traced shadows and area lights so we chose Exluna's Entropy renderer to generate our "Ambient Occlusion" information. The setup simply consists of two area lights (implemented as solar light shaders) and a very simple surface shader that

samples those lights appropriately, taking into account the objects self-shadowing with ray-tracing.

```
Surface srf_diffuse_pref (
  Varying Point Pref = (0,0,0);
  float DiffuseFalloffAngleEnd = 90;
)
{
  point PP = 0;


  vector NN, NF;

  /*  Initialize  */

  PP = Pref;
  NN = normalize(Du(PP)^Dv(PP));
  NF = NN;

  /* Light Calculations */

  Ci = 0.0;

  illuminance (PP, NF, radians(DiffuseFalloffAngleEnd))  {
      LN = normalize(L);
      Ci += diffuse(NF);

  }
  Oi = 1.0;

}
```

Given enough samples into the area lights, this process generates very smooth soft shadows and a natural falloff of light into the corners of the object.

Figure 14 - Test rendering of Pishkin with "Ambient Occlusion" lighting only

In the case of the Pishkin Building for *Stuart Little 2* these lighting calculations can be made once and then stored for all future uses of the building since the building would not be moving or deforming over time. This was accomplished by pre-calculating the "Ambient Occlusion" pass and storing it as a series of textures – one texture for each patch or group of polygons on the building.

The Renderman implementation of this pipeline consisted of creating a rib for each patch on the building. This rib consisted one patch and a shadow object. In the rib generation, the patch's "P" coordinates were moved to be located directly in front of the camera with it's coordinates normalized to the screen space of the camera so that the patch completely filled the view. The unmoved vertices of the patch were stored as "Pref" data for shading purposes. The entire building object was stored in the rib as a shadow object.

The shader than ran the shading calculations on the "Pref" geometry which took into account the shadow object of the building and returned the results to "Ci" which effectively generated a texture map for that patch.

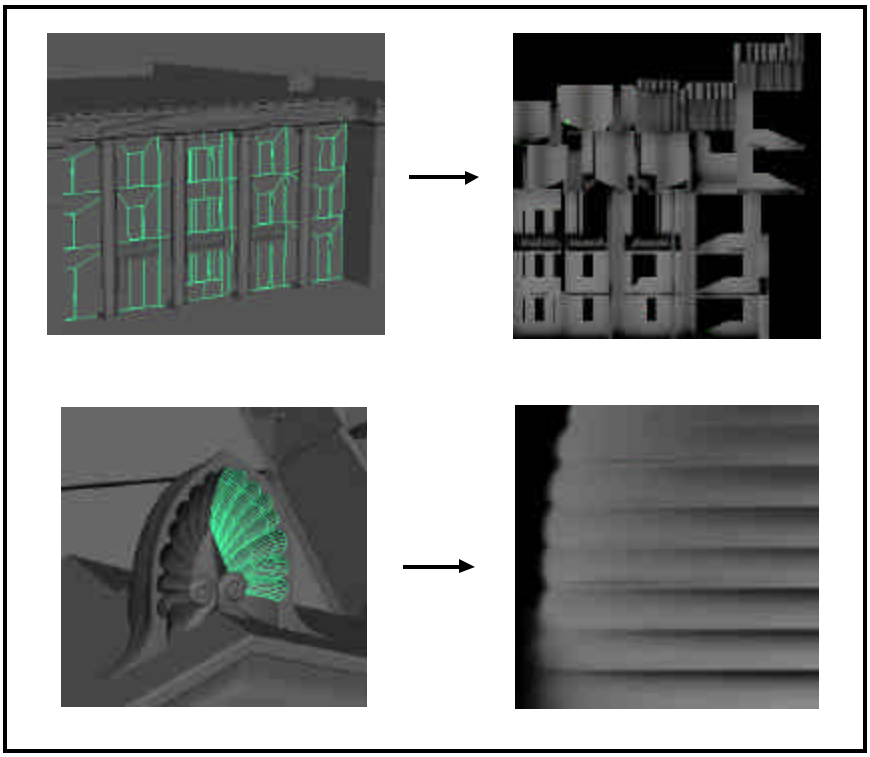


Figure 15 - "Ambient Occlusion" texture maps for 2 sections of the Pishkin

In our implementation we generated a separate set of texture maps for the sky dome contribution and the ground bounce contribution and then dialed the two new ambient levels in the shading for the building at the final rendering stage. If the sky was more blue or more overcast for a shot, the sky dome contribution color could be changed on the fly at the final rendering stage without performing any expensive calculations.

Figure 16 - Final version of the Pishkin as seen in *Stuart Little 2*

The end result was a more physically accurate shading model for ambient lighting which was both controllable and efficient since the more expensive calculations could be executed once and saved as texture maps.

1.2.2 Lighting

There are several things that provide input and reference to the lighting process including both on set reference and the creative process that is controlled by the director and the visual effects supervisors. In addition there are the technical aspects that are worth mentioning here including the lighting color space and keeping objects from clipping when they get too bright.

1.2.2.1. Reference balls on the set

One of the first things we inspect when starting a new visual effects shot are the reference balls that are shot on set. For each setup, the on set visual effects team clicks off a few frames of a 90% white sphere, a 50% gray sphere and a chrome sphere. These spheres can be visually inspected to tell you a few things about the photography and the lighting setup on stage.

Figure 17 - Reference orbs photographed on set

The 90% white sphere and 50% gray sphere are very useful to see the general lighting directions of the keys and fills and the relative color temperatures. Very soft lights and bounce cards will occlude more softly and point lights will have sharper falloffs. The relative warms and cools of the various lights can also be seen on these objects.

The chrome sphere is perhaps the most interesting of the three because it actually contains in it's reflections a map of the entire shooting environment (missing only what is directly behind the sphere). From this reference you can pinpoint with some degree of accuracy the direction from which the lights are illuminating the set and even relative sizes of bounce cards and color temperatures. If a C.G. sphere is matchmoved to the on set sphere, software can unwrap the sphere and give you an accurate reflection map for the set.

1.2.2.2. Lighting cues from the photography

The next items to get serious attention when the lighting begins on a shot are the cues from the plates themselves. Generally, the plates are carefully inspected by the lighting artist for any cues that help to determining how the characters or objects should be lit to fit into the plate. Shadow density and color, location and orientation are all things that are checked for. Highlight and specular features are also used to provide both spacial and color reference for the lighter. The plates are also examined for any

shiny objects that should catch a reflection of our character or bright objects that should bounce some light onto the subjects.

Figure 18 - Reference material shot on set with Stuart "stand-in" model

Basically, common sense combined with careful observation are used to dissect the photography and setup a computer lighting environment that matches the set as closely as possible.

1.2.2.3. Creative input

Once the character and objects are basically integrated into the live action photography, the creative process can really begin. This is probably most important aspect of the lighting process and it can be provided by a number of people on a film including the director, visual effects supervisor, or computer graphics supervisor and the lighting artist themselves. The goal is to enhance the mood and the realism of the film by either reinforcing the set lighting or breaking the "rules" a little and adding a light that wouldn't have been possible on the set.

Figure 19 - Margalo in a can illustrating the use of "creatively" driven lighting

For Stuart, the creative decision was made that he should have characteristic rim light with him at all times. In some scenes, the lighter needed to be resourceful to find an excuse in the plate to rim light the mouse to fit his heroic character. Since we had plenty of control over the computer generated lighting, we could also do tricks like rim light only Stuart's head and top half of his clothes and let his legs and feet more closely match the photograph of the plate to accomplish both the creative tasks and the technical requirements at hand.

Figure 20 - Rim lighting in all environments for Stuart

Backlighting on the falcon was another opportunity to use lighting to an artistic advantage. Because the feathers on the falcon had separate backlighting controls in his wings and various areas of his body, the lighting could be used to accentuate a performance.

Figure 21 - Backlighting on the Falcon's wings

1.2.2.4. Color space issues

The topic of lighting color space is one of much debate and concern on each show that I have been a part of over the years and for good reason. The color space in which you choose to light controls the way that the light will fall from bright to dark and every value in between. It is also a complicated subject that could take many pages to cover in a thorough manner so the attempt here is to discuss the topic in a brief and pragmatic way that will perhaps encourage you to investigate further if your interest is piqued.

There are two significantly different color spaces that we will introduce briefly and then discuss how these color spaces work in production.

1.2.2.4.1 Linear Color Space : Gamma 2.2

Gamma 2.2 is also referred to as "linear color space" because, presuming a correctly calibrated display device, doubling the intensity of a pixel value actually doubles the amount of energy coming out of your monitor or bouncing off the screen at the front of a theater. This makes computer graphics operations like blur and anti-aliasing correctly preserve energy.

For instance, if you take 1 pixel with a value of 1.0 and spread it out over 2 pixels, both pixels will contain values of 0.5. If you sample the energy coming off of a calibrated display device in both the before and after case, the energy should be the same. This is because you are working in a linear color space.

The confusing thing about working in a linear color space is that our eyes are not linear in the way they perceive changes in intensity. For example, if you look at a 50-watt light bulb and compare it to 100-watt bulb, the 100-watt bulb will appear brighter. But when you compare a 100-watt bulb to a 200-watt bulb the relative difference in brightness will not be as great as the difference between the 50 and 100-watt bulbs. This is because our eyes are more sensitive to changes in dark values than in bright values.

So, when you are working in a linear color space, you find that in order to get a pixel value that looks visually to be about 50% gray you need to use a value of approximately 22%. This results in heavier use of the lower range of color values and necessitates rendering at least 16 bit images to preserve color detail in the lower range.

It is interesting to note that a point light illuminating a sphere from a long distance when viewed at gamma 2.2 looks just about right when compared to a real world experiment. This is because a simple lighting model correctly models the falloff of a light in linear color space.

All of these are good reasons to light computer graphics in a linear color space.

1.2.2.4.2 Logarithmic Color Space: Cineon

The color space defined by the Cineon format is another very useful color space. Rather than attempting to produce a space that is mathematically linear, this space is designed around the response of film negative to an exposure. Each time the amount of exposure is doubled, the Cineon code value is increased by 90 points. Since Cineon files are stored with 10 bits of data per channel, there is enough room to store the entire useful range of the negative in the Cineon format file.

This is a very useful format for color correction and other techniques that rely on a direct correspondence to the response of film. For instance, if you wanted to preview what a shot would

look like printed a stop brighter (effectively twice as bright), if your image is in logarithmic color space you simply add 90 Cineon code points to the image.

1.2.2.4.3 Workflow

How does this work out in production? In our production environment we scan and color-time our images in the Logarithmic color space using Cineon files. All of our computer-generated objects are lit in linear color space with a gamma of 2.2 and stored as 16 bit linear images. In the compositing stage, the background plates are converted to linear color space, the composite is done in linear color space, and then the images are converted back to Cineon files to be recorded back to film.

1.2.2.5. De-clipping

Now that we have introduced the issues of color space and color space conversion, we can do something truly useful with it. A problem that is often encountered in computer graphics rendering is making colorful objects bright without letting them clip. Particularly when you are lighting in a linear color space, in order to double the brightness intensity of an object, you have to multiply it's color by a factor of 2. If your object already has a red value of 0.8 and the green and blue channel are each 0.4, the red channel is going to clip when the brightness is doubled to a value of 1.6. When you have deeply saturated objects like a bright yellow bird, these issues show up with great frequency.

Figure 22 - Margalo's bright yellow color would tend to clip under high intensity lights.

The solution is to use a logarithmic space to handle the brightening of the objects that are likely to clip. Our implementation was written into the Renderman shaders. Since shaders store colors as floats they can exist outside of the 0.0 - 1.0 range. Our de-clip shade op was fed floating point color and returned a "de-clipped" version of that color by darkening the color by an automatically determined multiplier, converting that color to logarithmic space, and then adding the appropriate number of Cineon code points back to the color to preserve the original intensity. This results in a color that smoothly desaturates as it gets over-exposed and has a very filmic look.

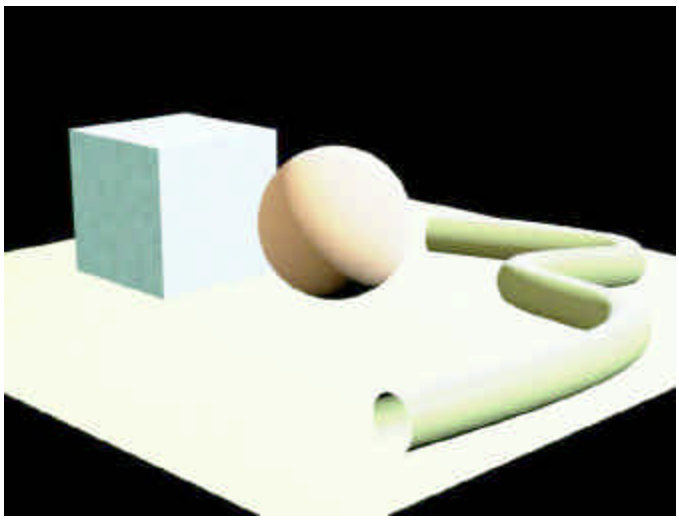


Figure 23 - "Declip" test which shows colors naturally desaturating with intensity

1.2.2.6. Multiple Passes

In order to have more control over both the rendering and the compositing process, it is common to render out many passes for a single character and each of the props in the scene.

Figure 24 - From left to right, hands, jacket, head, eyes, pants, whiskers and tail

1.2.2.6.1 Separate shadow maps for each "type" of object

Not only do we render separate color passes for each type of object, but we also break out our shadow maps into lots of passes. For instance, having one shadow map for Stuart's head and a separate map for his clothes gives the artist more control over setting the blur and bias controls for each. Because the fur is not a hard surface, the blur and bias controls need to be pushed slightly higher than for the clothes shadows to avoid objectionable and noisy self-shadows.

1.2.2.6.2 Cloth beauty pass

The cloth is usually rendered on it's own. Usually the different layers of cloth can be rendered together but sometimes they are also broken out into layers.

1.2.2.6.3 Character skin/fur beauty pass

The character's skin, fur and feathers are rendered together without any other elements. Since this is generally the most expensive of the render passes, the goal is to only render these elements once or twice if possible. The skin and fur is rendered with the eyes and costume as a hold out so that everything else can simply be layered behind in the composite stage.

1.2.2.6.4 Eye beauty pass

The eyes are rendered by themselves and placed behind the skin pass.

1.2.2.6.5 Cast and contact shadow passes

For each character we have several different types of shadow passes that are rendered out. Each character has near and far contact shadows which are generated from a depth map rendered from below the character and then projected onto the surface on which the characters walk. Each character also has various cast shadow elements that are rendered from the key lights and projected from those lights onto the set geometry.

1.2.2.6.6 Reflection passes

If there are shiny objects in the scene, a reflection camera is set up to render the characters from the perspective of the reflective object.

1.2.2.7. Tile-rendering when needed

Even with improved processing power and increases in memory configurations (our rendering computers have 1 gigabyte of ram per processor) there are always some frames that just won't render. In the cases where it is related to memory usage, we break the rendering problem down into tiles so that Prman can work on just a section of the image at a time.

In some cases where the character or prop is motion blurring right by the camera, we may break a single frame into more than 200 tiles to split up the job on many processors and reduce the memory consumption. Once all of the individual tiles have been completed on the render farm, the complementing scripts then re-assemble the tiles back into a complete image and the frame is finally complete.

1.3 Post-Renderman

In visual effects for live action, about half of the work is done in the lighting stage, and the other half takes place in compositing. The various passes need to be combined and massaged into the plate. We will detail here several of those 2d techniques which are widely used to integrate rendered images onto live action photography.

1.3.1 Compositing

1.3.1.1. Black and white levels/Color balancing

One of the first things that is done at the compositing stage is to make sure that the computer rendered images conform to the plate in the area of contrast and color temperature. Because the compositing tools are both powerful and provide quicker feedback than rendering the complicated geometry, the compositing stage is often the best place for these minor adjustments.

When composting a shot, one quick check is to see that the computer generated portion of the image do not get any darker or brighter than any of the cues in the live action photography without good reason. There can be exceptions but in general, the CG elements will pop if they don't match the contrast range very closely.

In the case that a particular light begins to feel out of balance with the others, it may require going to back to the rendering stage to make an adjustment but in general, with a number of compositing tools the color balancing can be accomplished efficiently.

1.3.1.2. Edge treatments

Almost nothing in real life is as crisp as what can be generated in the computer. Lines are generally smooth and nicely anti-aliased and geometry tends to be a little crisper than what we see in the real world.

In order to match the CG to the plate we treat the edges of the geometry carefully. Before composting we extract a matte of all of the edges of all of the CG elements.

Figure 25 - Stuart's "Edge Treatment" area as generated in the composite

Then, once the CG elements are composited over the live action background, then we use that "edge matte" to soften the image inside that matte only to blend the artificial elements into the photography.

Figure 26 - Stuart as seen in the final composite with edge softening

1.3.1.3. Film grain

Every film stock has it's own characteristic grain structure that introduces some noise into the scan. This noise actually has it's own response curve and shows up at different intensities depending on the level of exposure. Generally a film is shot on one or two different types of stocks so for each show we dial in our grain matching parameters to add grain to our synthetic elements that should closely match the grain in the plate. Each compositor also adjusts the grain parameters on a shot by shot and sometimes an element-by-element basis as needed to match the individual plate.

1.3.1.4. Lens warp

Even with advances in lens technology, the wide lenses used for today's feature films are not perfectly aspherical. They slightly warp the image near the corners of the frame and since the cameras used in computer graphics are perfect perspective projections the images will not register perfectly in the corners of the frame.

This is most noticeable in the case where you have long straight lines that are represented both in the live action plate and in the CG elements. In this case, a 2d image warp which mimics the behavior of the live action lens can help register the two more precisely to each other and solve the problem.

1.3.1.5. 2d contact shadows

One of the most useful tricks in the composting steps is the ability to add little contact shadows between elements. Often, because of the blur or the bias settings on a shadow from a light, or simply

because of the positioning of the lights in the scene, you don't get a little dark shadow from one layer of the cloth to another or from the cloth around the neck to the neck's skin and fur.

Fortunately, since the objects are already rendered as separate elements, generating a contact shadow between two elements is straightforward. The matte of the "top" element can be offset and used to darken the color channels of the "below" element. If needed, the offset can be animated and the color correction for the shadow can then be dialed in to taste.

1.3.1.6. Holdout "gap" filling

All of these separate elements give lots of control at the compositing stage but do come with one drawback. Because the "expensive" renders of the skin, feather and fur are rendered with 3d holdout objects or matte objects and the other renders are not, when you composite those images "over" each other and they move significantly, the motion blur can cause gaps between the elements.

These gaps come from the fact that what we're doing is a cheat. To be correct, each of the elements should be rendered with all of the overlapping elements as a matte object and then the various passes should be added together to form a solid matte and perfectly colored RGB channels.

Figure 27 - Before (left) and after (right) the "gaps" have been filled between the shirt and head

This is possible but not very cost-effective when you are talking about increasing the processing resources by a factor or 4 to accomplish such a workaround. In most cases it was adequate to find the gap areas by pulling a matte of the gray areas of the matte and finding where that overlaps the background objects. Then the fringing (whether it's showing up as light or dark fringing in a particular shot) can be color corrected to not be noticeable.

1.3.2 Film Recording

Once the final composite is completed, the shot is recorded back onto film with a laser recorder and the negative is processed, a work print is generated and the work print can be screened generally the next day to evaluate the work and hopefully final a shot!

1.4 Conclusion

We have discussed the use of Renderman in the context of a live action feature film with examples from the film *Stuart Little 2*. It is hoped that this introduction will serve as a jumping off point for of the topics discussed.

It should also be noted that a team of very talented artists are behind the examples and images presented and much credit is due them for the look and content of these notes.

The Making of Monsters, Inc.

Furry Monsters in a Foggy World.

PIXAR

ANIMATION STUDIOS

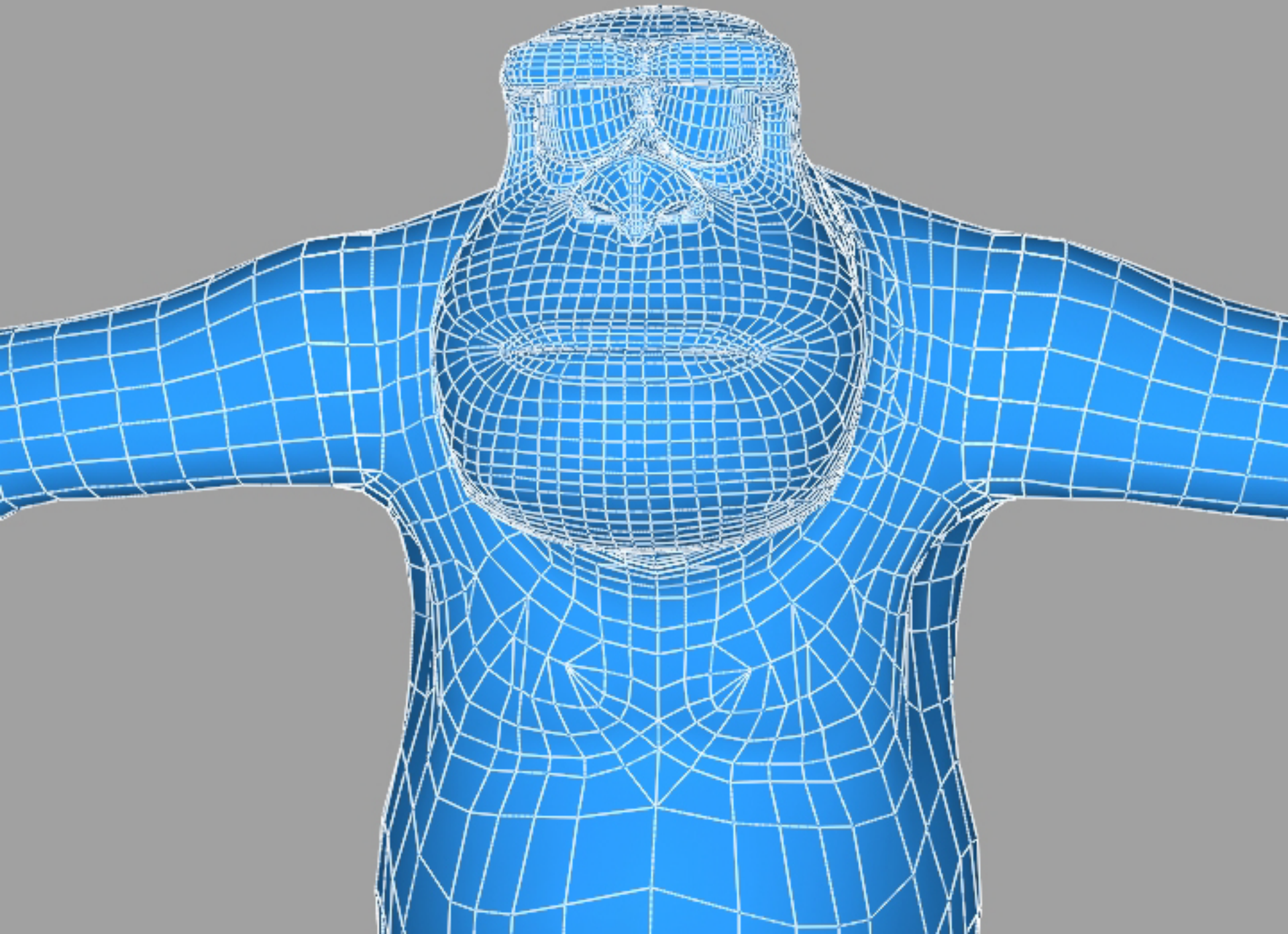
James P. Sullivan (Sulley)

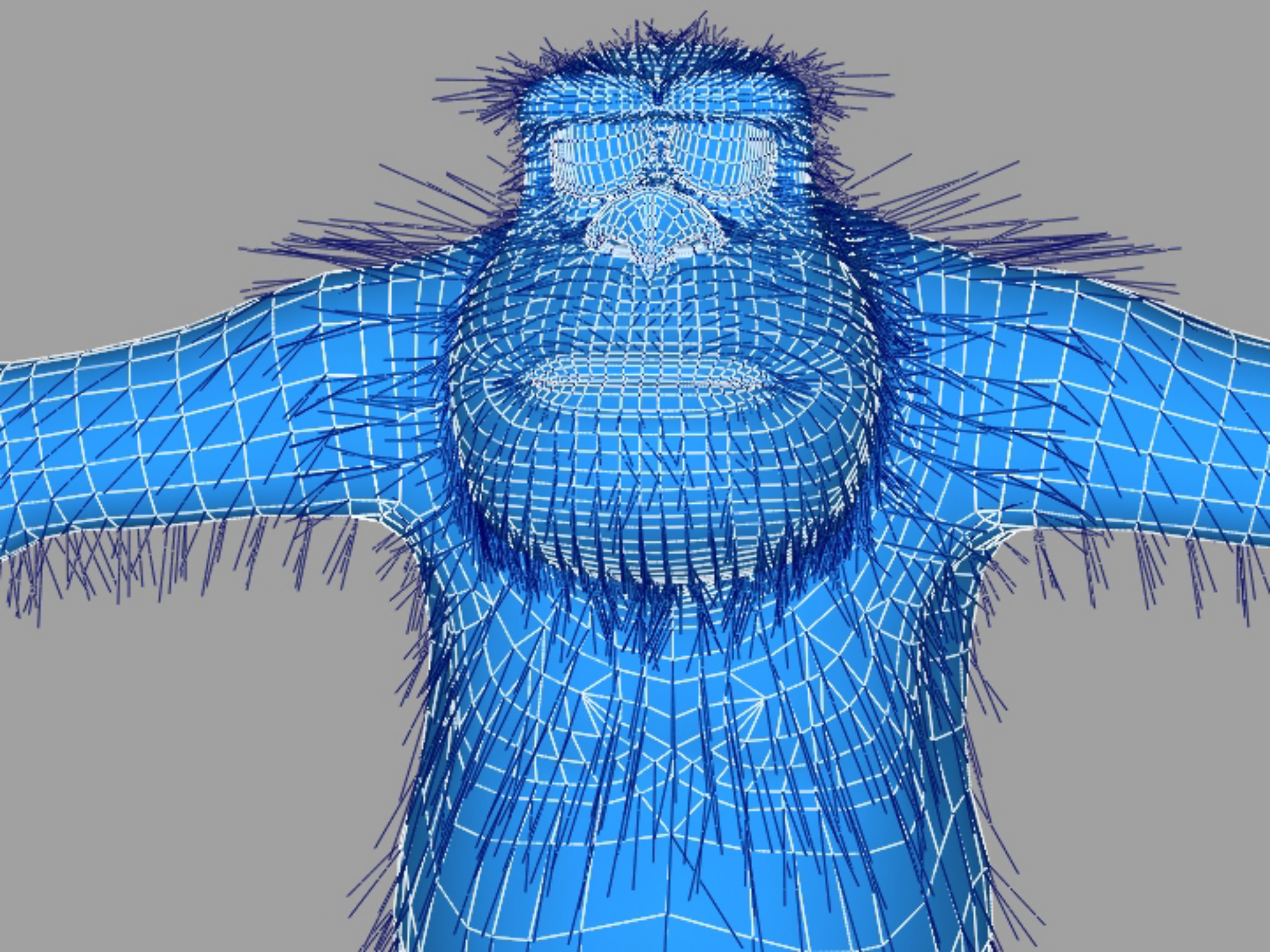
- *The top Scarer at Monsters, Inc.*
- *Covered in long fur*
- *Appears in about 800 shots*
- *Has over 2.3 million hairs*
- *Fur motion is dynamically simulated*
- *May have "things" in fur (e.g. snow)*

PIXAR
ANIMATION STUDIOS

Modeling Fur

- *Key hairs*
  - One per vertex of subdivision mesh
  - Specify overall length, shape, and motion
  - Not rendered
  - 10 CVs per hair (uniform B-spline)
- *Grooming done with "hairbrush" tool*
  - Sulley has about 25K key hairs
  - Use virtual hair brushes to comb hair





Modeling Fur

- *Inbetween hairs*
  - Comprise the final fur coat
  - Overall shape derived from key hairs
  - Sulley has about 2.3 million
  - Grown new each frame
- *Fur look specified procedurally using builders*

Modeling Fur

- ***Builders***

  Analogous to shaders

  Grow geometry on surfaces

  For fur, they specify color, density, tapering, opacity, clumping, scraggle, special animation, etc. of each hair

- ***Motion (dynamics) is decoupled from look***

Animating Fur

- ***Physical simulation***

  Don't interfere with animation

  Minimize user (TD) intervention

- ***Process***

  Sulley animated without fur

  "Shots" department adds collision objects,

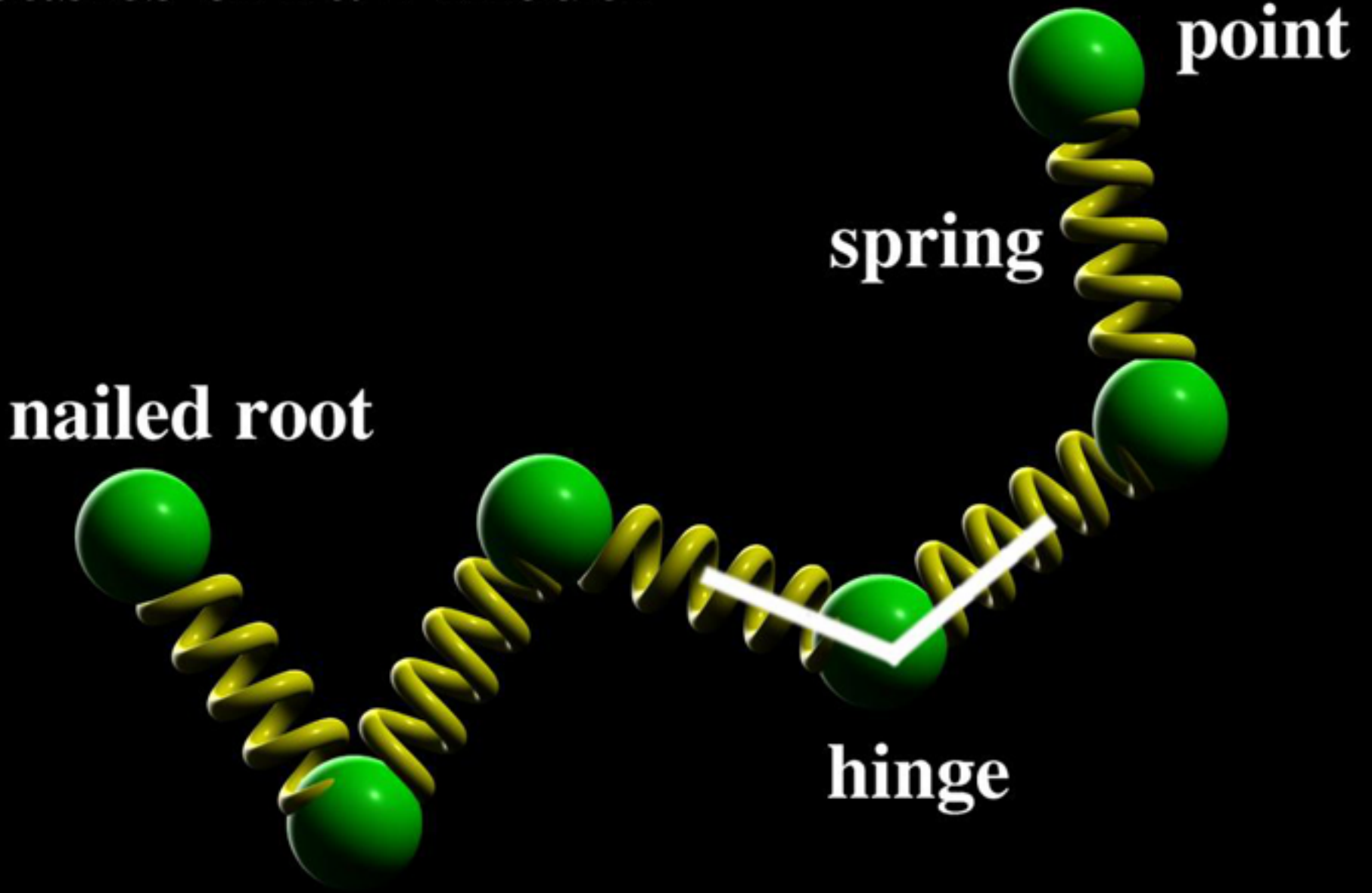
  runs simulation, fixes defects

  Simulator parameters can be adjusted per shot

  Handed off to lighting and effects

PIXAR
ANIMATION STUDIOS

basics of hair model



Animating Fur

- *Simulation challenges*

  Tangles, stretching, and explosions

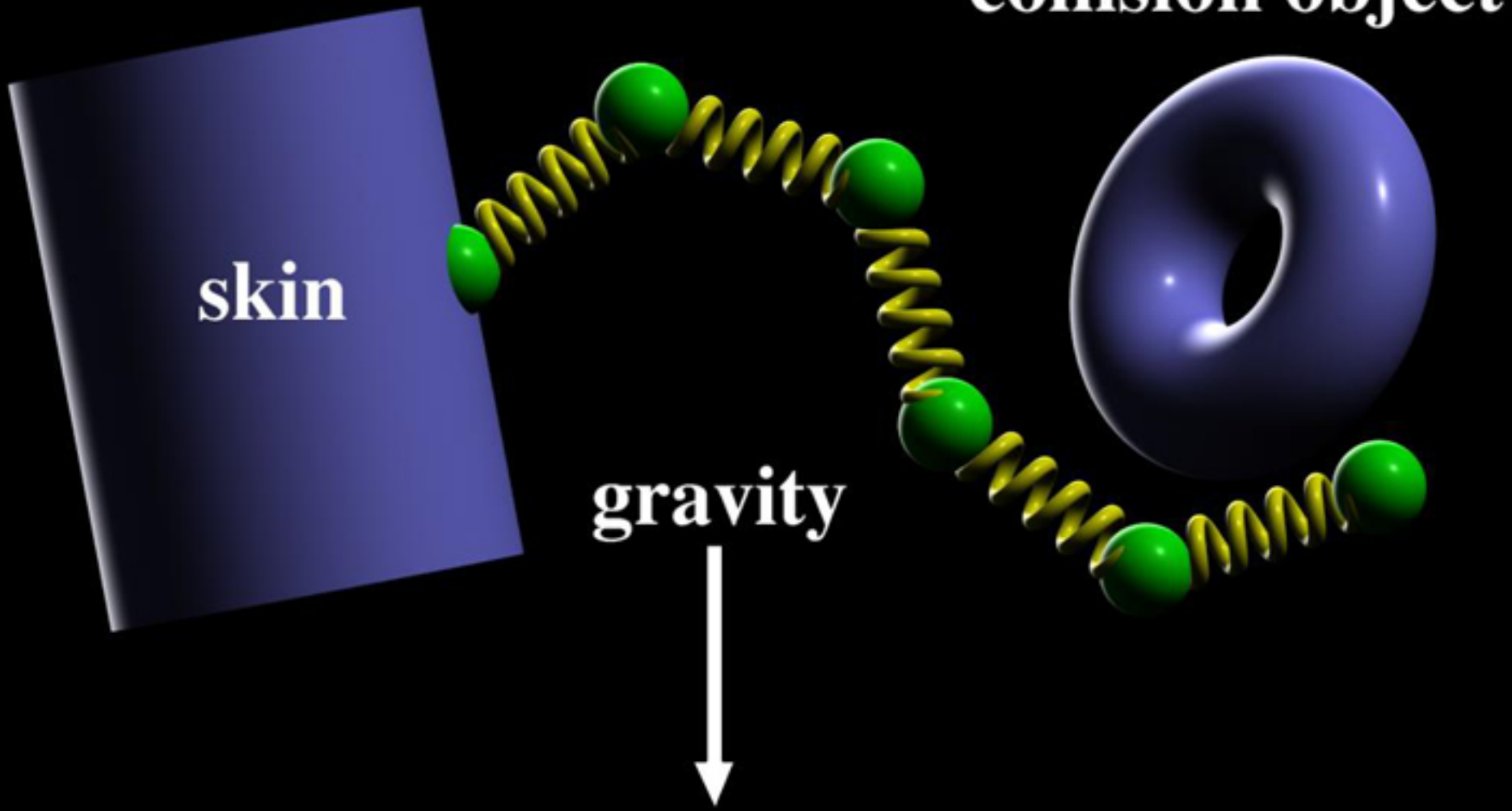
  Cartoon physics

  Animation problems: bad knots, off screen horrors, intentional intersections

  No pre-roll

  Obstacle courses

major causes of movement



Other Hairy Characters

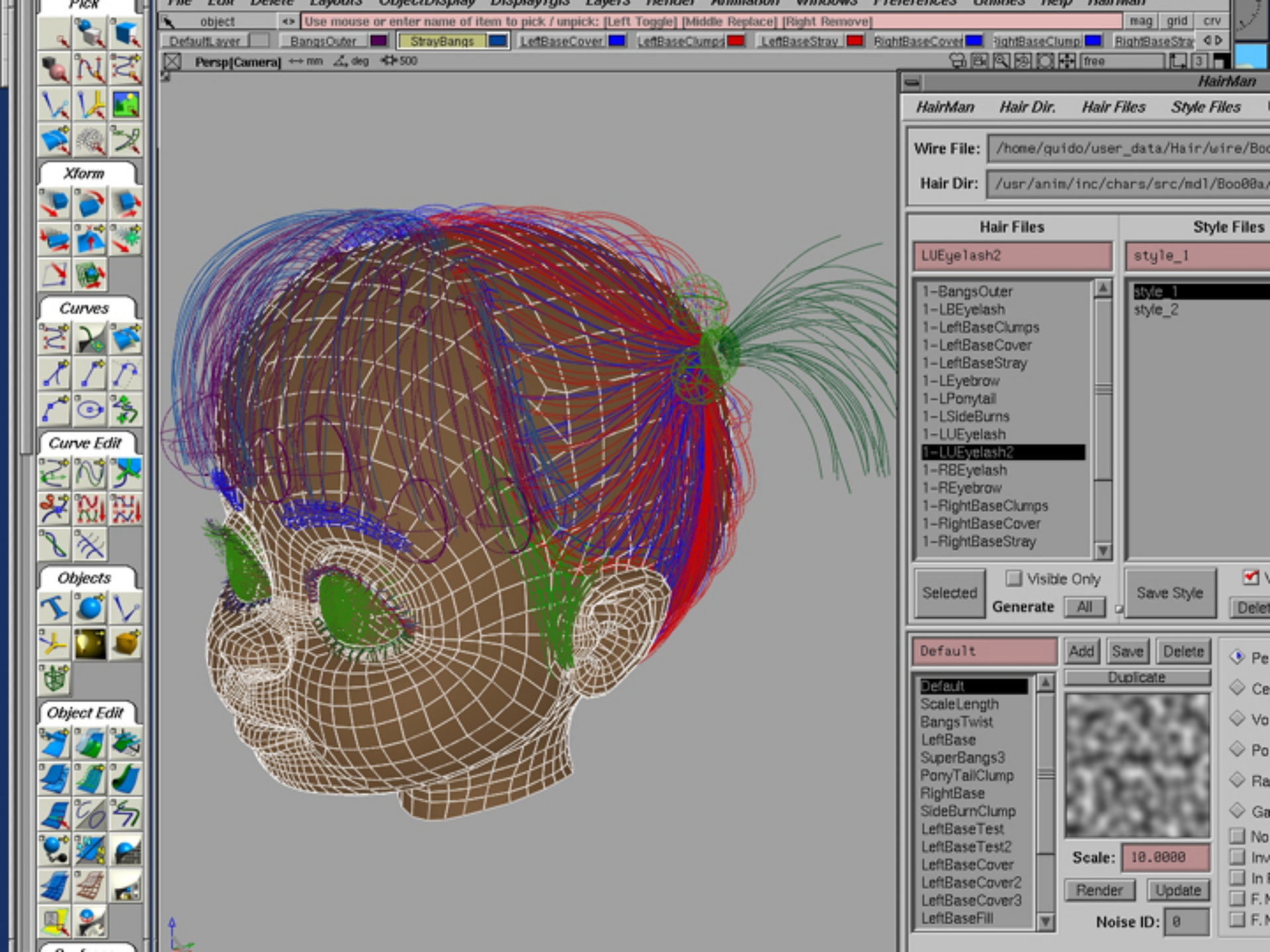
- ***Boo***

  Hair sculpted in Alias Studio

  Simulated ponytail
- ***Yeti***

  Fully procedural hair
- ***George***
- ***Hairscare***

PIXAR
ANIMATION STUDIOS


File Edit Delete Layouts ObjectDisplay DisplayFlgs Layers Render Animation Windows Preferences Utilities Help HairMan
object
Use mouse or enter name of item to pick / unpick: [Left Toggle] [Middle Replace] [Right Remove]
mag grid crv
DefaultLayer
BangsOuter
StrayBangs
LeftBaseCover
LeftBaseClumps
LeftBaseStray
RightBaseCover
RightBaseClump
RightBaseStra
Persp|Camera|
free
Pick
Xform
Curves
Curve Edit
Objects
Object Edit
HairMan
HairMan
Hair Dir.
Hair Files
Style Files
Wire File:
/home/quido/user_data/Hair/wire/Boo
Hair Dir:
/usr/anim/inc/chars/src/mdl/Boo00a/
Hair Files
Style Files
LUEyelash2
style_1
1-BangsOuter
1-LBEyelash
1-LeftBaseClumps
1-LeftBaseCover
1-LeftBaseStray
1-LEyebrow
1-LPonytail
1-LSideBurns
1-LUEyelash
1-LUEyelash2
1-RBEyelash
1-REyebrow
1-RightBaseClumps
1-RightBaseCover
1-RightBaseStray
style_1
style_2
Selected
Visible Only
Generate
All
Save Style
Default
Add
Save
Delete
Duplicate
Default
ScaleLength
BangsTwist
LeftBase
SuperBangs3
PonyTailClump
RightBase
SideBurnClump
LeftBaseTest
LeftBaseTest2
LeftBaseCover
LeftBaseCover2
LeftBaseCover3
LeftBaseFill
Scale:
10.0000
Render
Update
Noise ID:
0

The Himalayas

- *Wind and falling snow*
- *Impressions in snow*
- *Particle effects*
- *Snow in fur*

Snow in Fur

- *About 1 million snowflakes (peak)*
- *Clusters of snowflakes created as hairs are grown and animated*
- *Distribution controlled by*

  Wind direction

  Time (accumulation)

  3D paint

Atmospheric Effects

- *80% of rendered shots*
- *Complex enviroment*
- *Interaction with hair*
- *Self shadowing*
- *Steam jets*
- *Snow storm*

PIXAR
ANIMATION STUDIOS

Existing "fog" models

- *Cone lights*
  - Geometry with surface shader
  - Not volumetric
- *Cheap fog*
  - Atmosphere shader mixes in fog color
  - No illumination
  - Analytic and fast

Existing "fog" models

- *Foglights*
  - Fog attached to a single light
  - Combine by addition
  - Limited volumetric effects
  - Expensive
  - Used for special effects (i.e. death ray)

The atmosphere shader

- *Step through space from the camera to the shading sample (mp)*
- *Sample the fog density and illumination*
- *Accumulate the total fog contribution*

A new "fog" model

- *Support for complex volume density functions*
- *Multiple fog objects can be illuminated by multiple lights*
- *The contribution of all fog objects and all lights is considered in a single pass*
- *The fog is not recomputed for every sample*
- *Caching allows fewer computations*

PIXAR
ANIMATION STUDIOS

Fog Widgets

- *Defines a density field over space.*
- *Several properties determine the appearance:*

  color, opacity, attenuation

  4D fractal noise
- *Each widget has a stepSize parameter*

PIXAR
ANIMATION STUDIOS

Fog Widgets

- ***Atmospheric Fog***

  Camera-relative atmospheric fog
- ***Blobby Fog***

  Smooth ramp falloff

  Can import particles from Maya
- ***Puffy Sticker***

  Height field fog

  Smooth ramp falloff

Illuminating Fog

- ***Problem***

  In RenderMan, lights bind to surfaces, not fog widgets

Illuminating Fog

- *Solution:*

  The light category string specifies the fog widgets to be illuminated

  Lights that illuminate any fog widgets must be illuminated onto all surfaces at the same level

  Deep shadow map support

PIXAR
ANIMATION STUDIOS

Fog-Only Lights

- *Lights that illuminate fog are called fog-only lights*
- *By default, a fog-only light illuminates all fog widgets.*
- *Illuminaton of specific fog widgets can be set via the lighting tool*
- *Each fog-only light has a fogStepSize control*

Fog Shadows

- *Self-shadowing fog can add a lot of details*

- *This is done by rendering a fog deep shadow*

PIXAR
ANIMATION STUDIOS