

# Writing RenderMan® Shaders

**Siggraph 1992 Course 21**

Monday, July 27, 1992

**Tony Apodaca and Darwyn Peachey**

Course Co-Chairs

## ABSTRACT

The RenderMan Shading Language is a special-purpose programming language that performs shading calculations in RenderMan rendering programs. Shading Language programs, called shaders, can be used to model materials and effects in a physically realistic or in an “unrealistic” artistic style. Attendees learn how to write shaders that create a rich visual world in applications such as animation, CAD and presentation graphics. Examples of successful RenderMan images and animations will be presented as detailed case studies to give attendees an injection of practical experience with the Shading Language.

(This digital version, created for the 1995 Siggraph Course Note CD-ROM, includes minor text corrections as of May 1, 1995.)

Copyright © 1992–1995 Pixar. All rights reserved. RenderMan® is a registered trademark of Pixar.

## Course Objectives

The course covers the theory and practice of writing sophisticated RenderMan shaders to simulate natural and artificial objects and effects, such as: bricks, plants, fruit, fire, water and special light sources, in rendering styles ranging from photorealistic to cartoon style. No one is going to become an instant expert, but the attendees should learn some useful techniques of how to approach shader writing problems, and should leave with the confidence that they too can write visually complex shaders to simulate interesting surfaces.

## Level/Desired Background

We called it Advanced, but don't let that scare you. The course does require a solid background in both 3-D computer graphics and programming, but you don't have to be a rocket scientist. Some familiarity with the RenderMan Shading Language is strongly encouraged (e.g., took the previous Siggraph course or have read the "RenderMan Companion", or have written at least one shader). We will be talking a lot about Shading Language programs, however, so fluency in the C programming language is expected.

## Suggested Reading Material

"The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics," Steve Upstill, Addison-Wesley, ISBN 0-201-50868-0

This is the basic textbook for RenderMan, and is a must read. Chapters 11 through 16 cover shading and the relationship between geometry and shaders. The cover image and the color plates are the kind of images that attendees will be able to make after attending this course.

"The RenderMan Interface Specification, Version 3.1," Pixar

The second half of this handbook is the Technical Reference for the RenderMan Shading Language. Unfortunately, it is not as approachable as the Companion.

"The RenderMan Interface and Shading Language," Siggraph 1991 Course #21 Course Notes (or alternatively, Siggraph 1990 Course #18 Course Notes)

These are the notes to the previous course on RenderMan. The second half of the course notes concentrate on Shading Language basics, and fulfill the prerequisite for this course. If you can find one of these, you will find them very useful.

## Lecturers

### Tony Apodaca

Tony Apodaca is a Senior Engineer and Project Manager in the RenderMan Division at Pixar. Tony is co-developer and reigning Chief Architect of the RenderMan Interface Specification. He is also one of the Unknown Implementers of Pixar's image synthesis products. Tony received his Master's degree in Computer and Systems Engineering at Rensselaer Polytechnic Institute in 1986. His screen credits include *Tin Toy* and *knickknack*, but not *Terminator 2*.

### Darwyn Peachey

Darwyn Peachey is an Animation Scientist at Pixar. Since 1988 he has worked on the design and implementation of the RenderMan renderers and in-house animation software. Darwyn received M.Sc. and B.Sc. degrees in computer science from the University of Saskatchewan, Canada. Prior to joining Pixar he spent three years developing UNIX kernel software and four years as a member of the computer science research staff at the University of Saskatchewan.

### Tom Porter

Tom Porter is Director of Effects Animation at Pixar. Tom has been working in 3-D graphics since 1976. He wrote the software for the first commercial paint program (the Ampex AVA), developed fundamental algorithms for digital compositing (which led to the Pixar Image Computer) and for motion-blurred rendering. Tom has taken time out over the years to create still frame images (*1984*, *Textbook Strike*) to advance the state of the art in computer generated photorealism, his main research interest.

### Rick Sayre

Rick Sayre joined Pixar in 1987. He received his B.S.E.E. from the University of California at Berkeley and studied film at the University of California at Davis. At Pixar, he built a wide variety of warping and image processing tools and constructed shaders before joining the Animation Production Group, where he is a technical director and production software developer. Rick has outside interests which could be used to label him Art Damaged, and occasionally blows up performance artists.

### Eliot Smyrl

Eliot Smyrl joined Pixar in 1989 after receiving his M.S. and B.S. degrees in Computer Science from the University of California at Berkeley. At Pixar, Eliot worked in the RenderMan Group developing Looks software before joining the Animation Production Group in January, 1991. He currently serves as a technical director and production software developer. Eliot plays the trumpet and enjoys wandering around uninhabited mountainous areas.

## Schedule

|   |          |
|---|----------|
| Welcome and Introduction<br>Tony Apodaca<br>Page 1          | 8:45 AM  |
| Math and Graphics Review<br>Tony Apodaca<br>Page 6          | 9:00     |
| Basic Approach to Shader Writing<br>Tony Apodaca<br>Page 18 | 9:30     |
| <i>Break</i>  | 10:00    |
| Texture Generation<br>Darwyn Peachey<br>Page 32             | 10:15    |
| Writing Surface Shaders<br>Tom Porter<br>Page 57            | 11:00    |
| <i>Lunch</i>  | 12:00 PM |
| Lights and Shadows<br>Eliot Smyrl<br>Page 78                | 1:30     |
| Aliasing in Shaders<br>Darwyn Peachey<br>Page 96            | 2:30     |
| <i>Break</i>  | 3:00     |
| Antialiasing Techniques<br>Rick Sayre<br>Page 109           | 3:15     |
| Additional Case Studies                                     | 4:15     |

## Course Notes Table of Contents

|  |          |
|--|----------|
| Writing RenderMan Shaders.....                     | Page 1   |
| Math and Graphics Relevant to Shader Writers ..... | Page 6   |
| Basic Approach to Shader Writing .....             | Page 18  |
| Texture Generation .....                           | Page 32  |
| Writing Surface Shaders .....                      | Page 57  |
| Lights and Shadows .....                           | Page 78  |
| Aliasing in Shaders .....                          | Page 96  |
| Antialiasing Techniques.....                       | Page 109 |



# Writing RenderMan Shaders

Tony Apodaca

Welcome to *Writing RenderMan Shaders*. This is the second Siggraph course about RenderMan, and this course ought to really demonstrate why RenderMan is uniquely powerful, and should help you take advantage of that power. This introductory section will describe the goals of the course.

One of the great advantages of using a RenderMan renderer is the fact that you can describe the appearance characteristics of objects with as much detail and subtlety as you typically describe the shapes and positions of those objects. The RenderMan Shading Language is a special-purpose programming language for describing appearance characteristics. Shading Language programs, called *shaders*, can be used to model materials and effects in a physically realistic or in an “unrealistic” artistic style.

To describe appearance, you attach to each object a set of RenderMan shaders, which are loaded when the model is loaded, and executed by the renderer when it comes time to compute the colors of the objects in the image.

The power of the Shading Language comes from the fact that it is a fully functional programming language. You can program just about anything. This means that you are not limited to specific mathematical equations, or the old standard shading formulations published in Siggraph papers years ago.

The difficulty with the Shading Language comes from the fact that it is a fully functional programming language. You can program just about anything. This means you have to specify in great detail exactly what you want done, and making these decisions seems confusing and daunting.

So, welcome to *Writing RenderMan Shaders*. This course covers the theory and practice of writing sophisticated RenderMan shaders to simulate natural and artificial objects and effects, such as: bricks, plants, fruit, fire, water and special light sources, in rendering styles ranging from photorealistic to cartoon style. The course will help you design your shaders by thinking beyond those old-fashioned shading models and texture mapping tricks. You will learn how to invent your own tricks.

## Goals of This Course

---

- **Expand your mind**
  - think beyond standard lighting models
  - think beyond standard texture mapping
- **Expose the warts**
  - let you know what it really takes
- **Motivate you to write cool shaders**



We will go into some detail about exactly how difficult it is to get just the effect you want, avoid nasty side effects and artifacts, and get around some of the limitations of the language. This is not always easy, but we think it is important to be honest about the effort it takes.

Most importantly, we want to motivate you to get out there and write cool shaders. We want some of you to go into business writing shaders, and others of you to fill comp.graphics with shader postings. No one is going to become an instant expert, but you should learn some useful techniques of how to approach shader writing problems, and should leave with the confidence that you too can write visually complex shaders to simulate interesting surfaces.

The first step in writing cool shaders is to master Jim Blinn's *Ancient Art of Chi' Ting*. Very roughly, the fundamental tenet of this philosophy is: "If it looks good enough, it *is* good enough."



## Ancient Art of Chi' Ting

- **Learn when simulation is appropriate**  
“photo-realism”
- **Learn when hacking is appropriate**  
“photo-surrealism”
- **smoke and mirrors**



Sometimes, it is really necessary to do a full simulation of the surface in order for it to look right. People are very good at noticing when something looks wrong, particularly if it is mostly right. At other times, something that is not correct, but is somehow similar to it, is even better than correct. Such things are often called *art*. But most of the time, something that is close to right is good enough. Maybe the object is small, or moving fast, or the audience doesn't really know what it should look like anyway.

In fact, the real goal is not to make a perfect shader, but rather to make a *successful* shader.

## Criteria for a Successful Shader

- **Convincing in context of whole image**
- **Reasonably efficient in terms of CPU usage**
- **Very efficient in terms of programming effort**
- **Usability and reusability**



The shader must be convincing to the audience, but as a part of the image as a whole. Shading Language programmers tend to examine shaders in isolation, on simple objects against a black background, and this usually makes us work too hard to get our shaders to be “just right”.

For pride’s sake, the shader should be reasonably efficient in terms of compute time. Useless calls to normalize unit vectors are embarrassing. But of far more importance is efficiency in programming effort. A shader has to be run a few billion times before the CPU time it uses accumulates to match the hours and days you spend programming it, and the audience has to see the animation many times before they notice the tiny glitch that you see in still frames.

On the flip side, a well written shader can almost certainly be reused, the algorithm tuned slightly to give a wide variety of effects. The better engineered it is the first time, the more likely you’ll understand it when you try to adapt it next time.

Okay, enough proselytizing. Two administrative notes, and then on to the meat of the course. First, in the development of this course, we have made certain assumptions about the skill level of the students.

## Expected Background

- **C programmer**
- **Have read the *RenderMan Companion***
- **Have programmed at least one shader**
- **Not a rocket scientist**




We called it Advanced, but don’t let that scare you. The course does require a solid background in both 3-D computer graphics and programming, but you don’t have to be a rocket scientist. Some familiarity with the RenderMan Shading Language is strongly encouraged (e.g., took the previous Siggraph course or have read the *RenderMan Companion*, have written at least one shader). We will be talking a lot about Shading Language programs, however, so fluency in the C programming language is expected.

The second bit is a note about products. It is a somewhat disappointing, but true, fact that there is really only one commercially available RenderMan-compatible renderer which supports the full RenderMan Shading Language. Pixar’s PhotoRealistic RenderMan can be obtained in many packages on most platforms from several vendors, including Pixar, NeXT, Paracomp, Autodesk and others. All of these renderers are completely compatible. There used to be a product from Sun

Microsystems which was largely compatible, but they recently stopped selling it. We hope to see more renderers available in the future.

### Product Disclaimer

- **Pixar's PhotoRealistic RenderMan**  
MacRenderMan  
Autodesk RenderMan  
NeXT 3D Kit/Interactive RenderMan
- **Sun Microsystems SunART**



Given the current situation, we decided that the most helpful thing we could do for the students was gear this course towards success with this renderer, since it is the only one out there. So, certain algorithmic limitations, etc., which are specific to this renderer will be included in our discussion even though they don't apply to the Shading Language in the abstract.

## Math and Graphics Relevant to Shader Writers

Tony Apodaca

The first step to writing shaders with interesting patterns and unique reflectance functions is to become familiar with the primitive functions which you have to work with in the Shading Language. The Shading Language is very C-like, both in syntax and in math library functionality, so most people pick those details up very quickly. The hard part is engineering complex effects from these primitive ones. In this section, we will review a little of the relevant mathematics and basic graphics which you can leverage to make more interesting things.

### Math

Everyone remembers trigonometry. Sine equals opposite over hypotenuse. Cosine equals adjacent over hypotenuse. Tangent equals opposite over adjacent. There will always be places where a few mathematical identities like the Law of Cosines or the Double Angle Equations will help you solve for a certain value. However, what is much more likely to come up in graphical programming is simple vector algebra.

#### Trigonometry and Analytical Geometry

- **dot product of two vectors**

scalar

$$A \cdot B = |A| |B| \cos \theta$$

- **cross product of two vectors**

perpendicular to both vectors

$$A \wedge B = |A| |B| \sin \theta$$

- **normalized vectors**




Naturally, we can add and subtract vectors. The Shading Language correctly handles simple operations on the vector types `point` and `color`, so we don't need to write boring loops for each operation. In addition, the Language provides dot and cross product of `points`. Recall that the dot product of two vectors is a scalar which gives you the angle between the vectors. Vectors which point in roughly the same direction have a positive dot product, vectors which point in roughly opposite directions have a negative dot product, and we take advantage of this relation a lot. The cross product of two vectors is another vector which is perpendicular to both vectors. The magnitude of this vector is also related to the angle between.

One very important thing to remember is that in both of these relations, the length of the vectors themselves get into the equations. If we could guarantee that the vectors always had length exactly 1.0, we wouldn't have to worry too much about that. Unfortunately, the Shading Language does not automatically normalize any vectors for you, so you *must* remember to do that. Many is the time I've spent several hours trying to debug a shader which "can't possibly give me that picture!", only to find that I forgot to `normalize` some vector.

Quite often (as we will see in the next talk), we require a simple repeating pattern of some variable. There are three very simple ones which you should know without blinking.

### Simple Harmonic Patterns

- **sin, cos**
- **sawtooth waves**
- **square waves**



The first is the most obvious, the sine and cosine functions. They make great smooth patterns, are easy to modulate both in frequency and amplitude, and only require one function call. The second pattern is the sawtooth (aka triangle) wave. It is very easy to make this wave using the floating point modulo function provided by the language. The third pattern is the square wave. This is also easy to manufacture using the modulo function, and you can also control the duty cycle by choosing the appropriate cutoff.

## Using modulo to Make Waves



$$y = \text{mod}(x, 1.0)$$



$$y = \text{step}(0.5, \text{mod}(x, 1.0))$$

r

Sometimes you need a curve that rises more quickly or more slowly than linear. This is quite easy to do with exponentiation. Intuitively we know that if you raise a number to a power greater than one, it gets bigger very fast, and if you raise it to a power less than one, it's like taking a root, so it gets smaller. When writing shaders, we are usually exponentiating values between zero and one, and those values work backwards from the intuition.

## Exponentiation

$$x^y \text{ for } y < 1$$

gamma

$$x^y \text{ for } y > 1$$

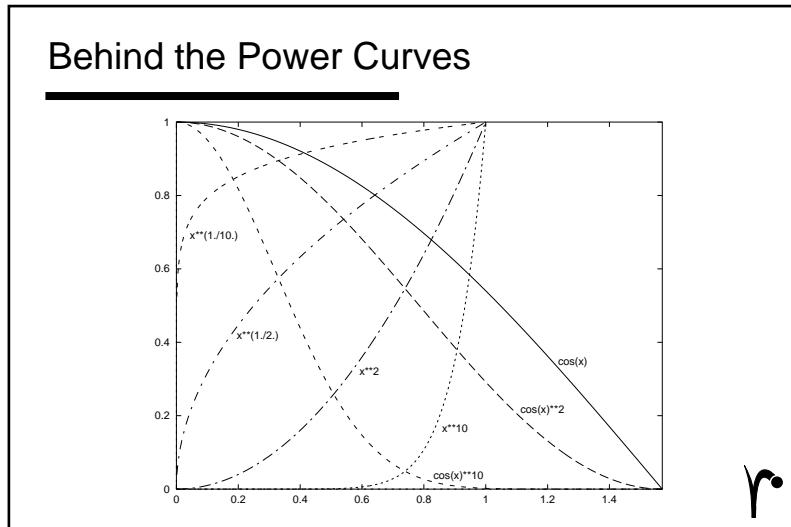
$$\cos(2\pi x)^y \text{ for } y > 1$$

specular highlights

r

We can easily see how we can generate very sharp slopes by using exponentiation on our earlier triangle waves. This is great if you want to have certain emphasized features like a sharp color band at

a certain value. One of the most common uses for exponentiation is to raise the cosine function to a low power, for example, when using Phong specular highlights. I think few people have ever graphed that function to try to understand just how sharp it really is.



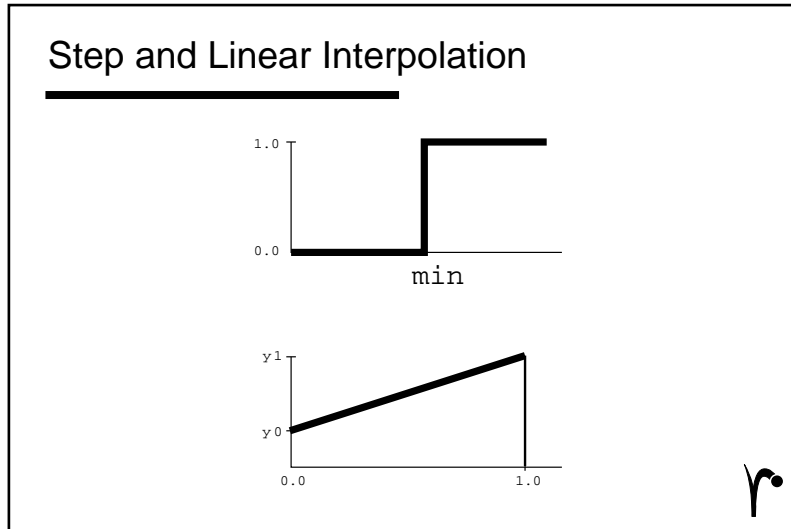
Quite often we find ourselves needing to interpolate some parameter between some values. There are several ways to do this, with increasing complexity and smoothness.

### Interpolation

- **step**
- **linear interpolation**  
“lerp”
- **smoothstep**
- **spline**

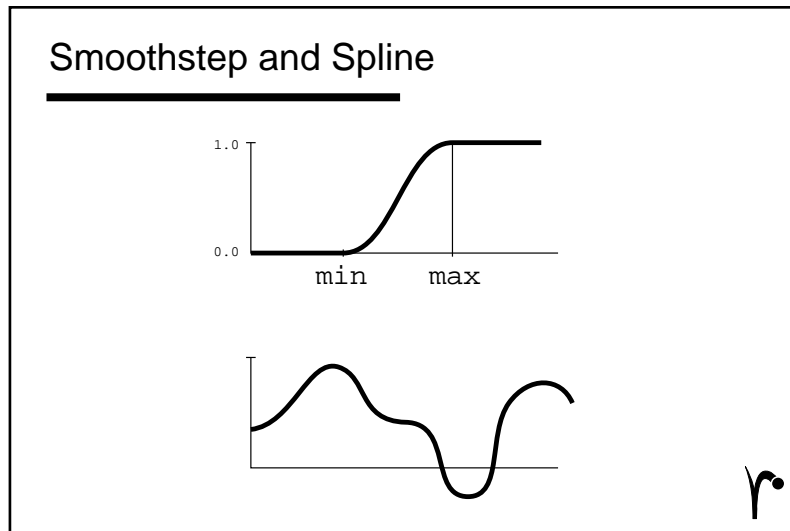
A small 'r' logo is visible in the bottom right corner of the slide.

The simplest is a step interpolation. When the controlling parameter exceeds some threshold, we switch over to the higher value. In the Shading Language, this can be done with an `if` statement, or with the `step` function, depending on the situation. A slightly better way is to linearly interpolate between two values. This is often called *lerping*. If you do this in two dimensions, it is called a *bilerp*, and in three dimensions, it is (not surprisingly) called a *trilerp*. In the Shading Language, lerps are done using the standard equation  $y = y0 + (y1 - y0)t$ . Since this is often done for colors, there is a presupplied shortcut function called `mix`.



If you want some more smoothness, such as derivative continuity on the ends, we suggest the `smoothstep` function. This is the standard cubic interpolation sometimes called *ease-in/ease-out*. We use this quite often to prevent staircasing artifacts, as you will see later in the day. Finally, if you need to interpolate smoothly between many values, there is a general-purpose Catmull-Rom interpolatory spline function `spline`, which can interpolate between floats, points or colors. Color splines are very interesting and useful for many effects effects, such as those associated with one-dimensional textures.





Before we leave the math review, I should mention the problems of transforming points and vectors. It is quite common for people to know that computer graphics uses 4-by-4 matrices to describe transformations. It is somewhat less common for people to know that these matrices are used because they allow us to take advantage of homogeneous coordinates. It is unfortunately very uncommon for people to understand what homogeneous coordinates are, and why we want to use them. Here is why: if you have a 4-by-4 transformation matrix  $M$ , you can describe all translations, rotations, scales, shears and perspective transformations within the same matrix. You can then use the same matrix to transform points, direction vectors and (with a little work) normal vectors.

### Transformations

- **points**  

$$p = (x, y, z, 1) M$$
- **direction vectors**  

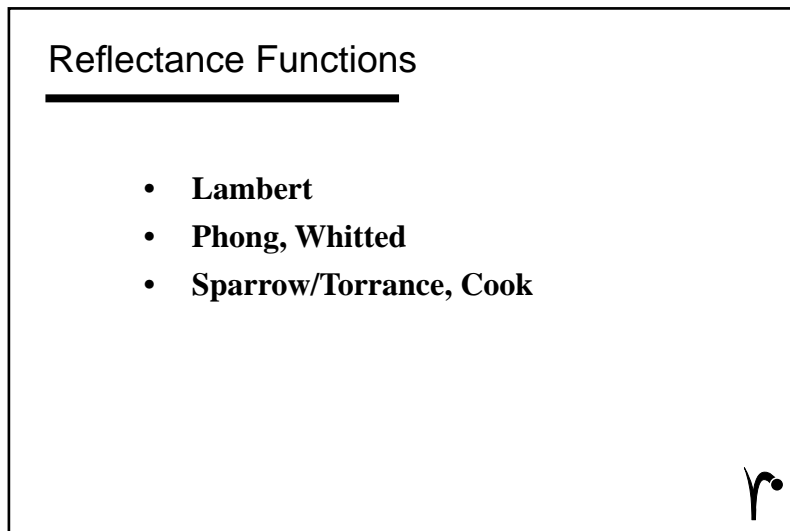
$$v = (x, y, z, 0) M$$
- **normal vectors**  

$$n = (x, y, z, 0) M^{-I T}$$

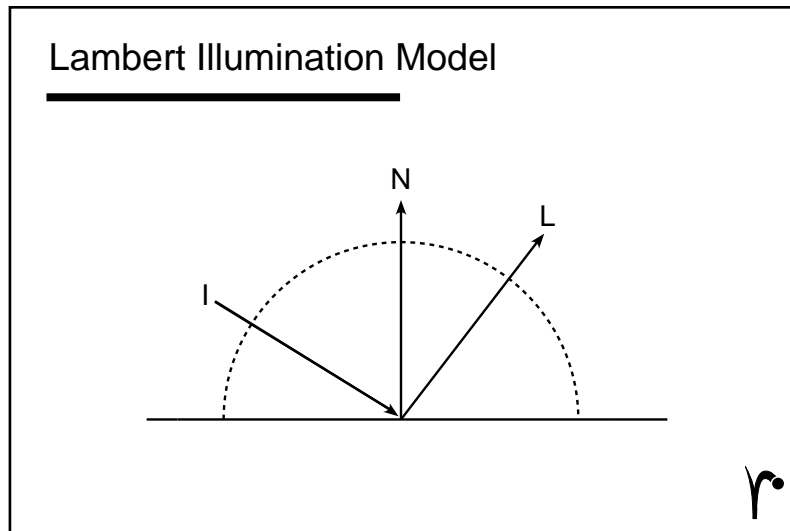
3-D points are transformed by sticking a 1.0 on the end of the point to make a 4-D homogeneous point. Multiply that by the matrix, divide out the  $w$ , and you have the transformed point. 3-D direction vectors are transformed by sticking a 0.0 on the end of the vector to make a 4-D homogeneous direction vector. Multiply that by the matrix, drop the 0.0, and you have the new transformed direction vector (note: not quite this simple for perspective matrices). 3-D normal vectors are transformed like direction vectors, but use the inverse transpose of the matrix. Many people try to transform normal vectors through the standard matrix, and sometimes works, but only if the standard matrix equals its own inverse transpose. This happens surprisingly often (translation, rotation and uniform scales), which is why most people think it always works.

## Graphics

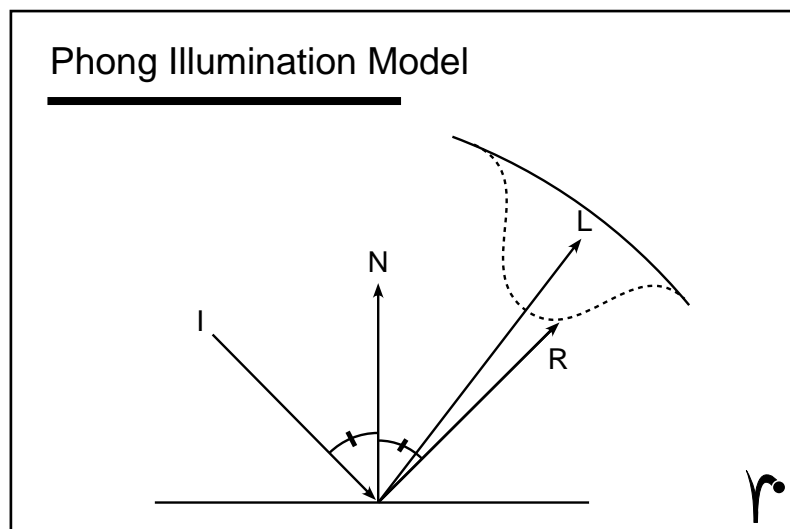
Most computer graphics programmers are familiar with the standard reflectance functions developed in the 1960's and 1970's. Here is a quick review.



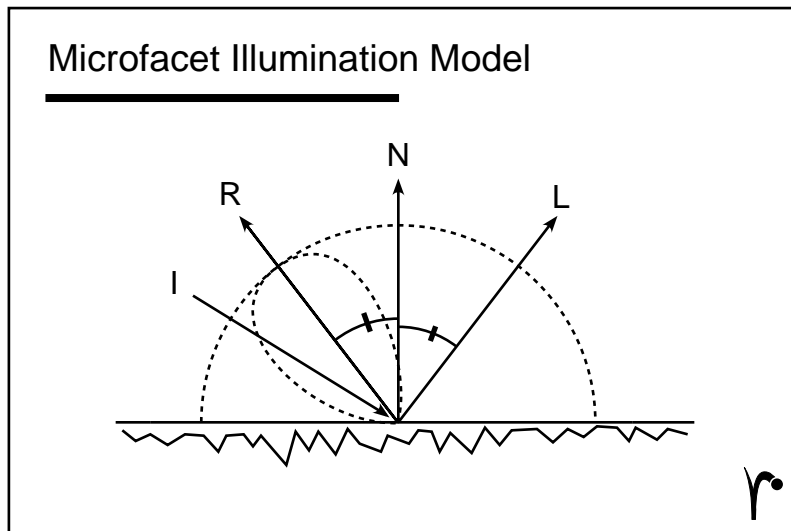
Diffuse illumination assumes that light is scattered in all directions when it hits a surface, according to Lambert's Law. The direction of view does not matter, only the cross-sectional area visible to the camera. This is your standard  $N \cdot I$  equation, and can be accessed in the Shading Language by the `diffuse` function.



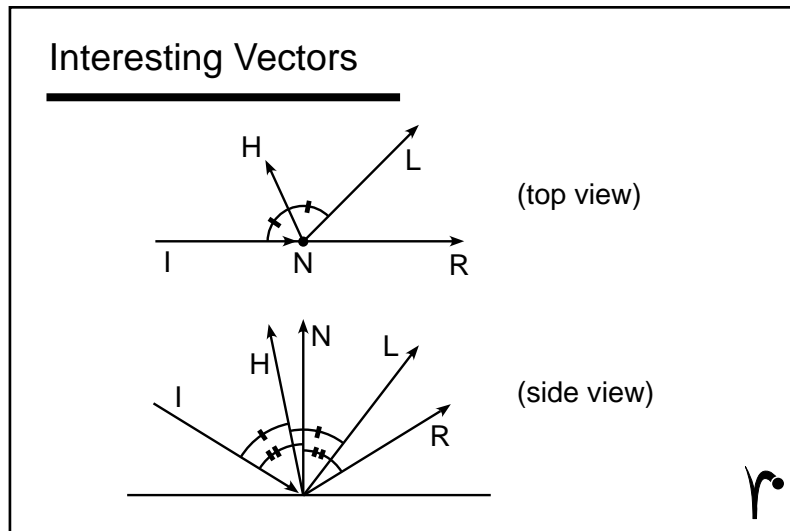
Specular highlights are caused by mirror-like reflection of the light sources. The first attempt was, of course, Phong lighting, which was a purely empirical equation based on  $N \cdot L$  to a (random) power. In other words, a codified cheat! This equation is available in the Shading Language as `phong`. Later, Whitted introduced a slightly modified version of the equation based on  $N \cdot H$  which produced better falloff in 3-D. In fact, more recent analysis of these equations suggests that they are a better model of a mirror reflection of a spherical area light source than they are of near-specular reflection. Phong's specular reflection almost always is calculated to reflect the color of the light source, independent of the color of the object. Cook later explained why this function looks good for plastic, and why metal should reflect specularly using the color of the object, instead.



Sparrow and Torrance looked at the physics of rough surfaces, and modeled specular reflections as mirror reflections off of billions and billions of tiny, randomly-oriented mirrors called *microfacets*. On smooth surfaces, most of the facets are pointed roughly up, and so the specular highlight is constrained to be near the mirror vector. On rough surfaces, the facets are pointed all over, and the specular highlight is broad and can be seen from many directions. They suggested several equations for modeling the statistical distribution of facet directions, and Blinn (who publicized their work) chose one as being easiest to calculate. While at Pixar, Cook used a slightly different formulation which was also easy compute and “looked nicer” (cheat!!). Cook’s version of the Sparrow/Torrance illumination function is what you get when you use `specular` in the Shading Language.



The mathematical equations for all of those illumination models involve copious use of certain vectors at the surface of the object. The following diagram illustrates the relationships among those vectors.

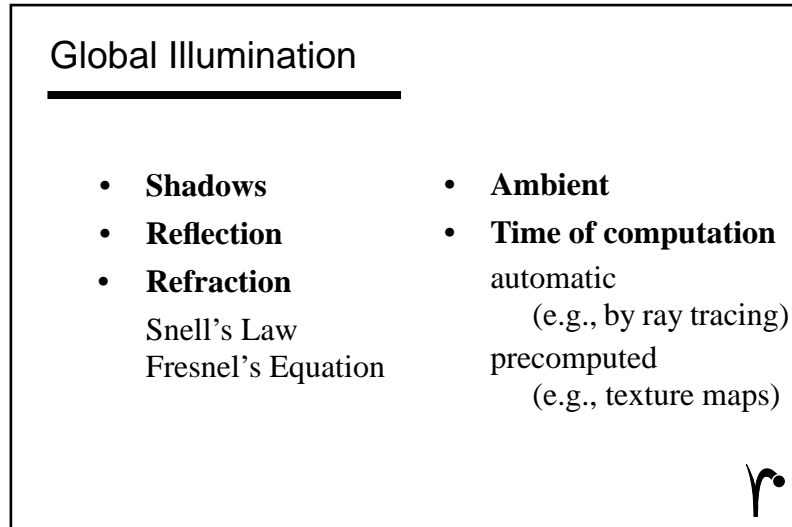


In addition, the equations that involve the normal vector always assume that the normal vector points towards the camera, that is, that the surface is a *front-facing*. Back-facing surfaces are generally not visible for solid objects, and if you use the backwards facing surface normal, you get negative values for dot products and the equations fail. For this reason, we always try to reverse the direction of backwards facing surface normals, on the assumption that any object that we see the back of must really be a two-sided thin sheet (like a piece of paper). This procedure is called *faceforward*.

### Faceforward

- **Reverse direction of back facing normals**
- **Avoids negative values in Lambert Equation**
- **Assumes two-sidedness of all thin sheets**
- **Special processing at silhouettes**

All of the illumination models described above are called *local illumination models*, because they rely only on local surface information (the position of the surface, surface normal and roughness, and the direction towards the light source) to calculate the light leaving the surface. They do not take into account any interaction between surfaces, or even that their might be other surfaces. Renderers which do take that into account are said to deal with *global illumination models*.



Without oversimplifying too much, global illumination falls into two rough categories. The first is global illumination which controls how much light source illumination arrives at various places in the 3-D scene. Shadows prevent light from going directly from the light source to the object. Shadows have umbrae and penumbrae, which makes them harder to compute correctly than most people care to admit. The simplest, or hardest, global illumination to account for is ambient light — the light which has bounced around the room so much you can't tell which light source it originally came from. The simple way is to assume that it is roughly equal everywhere, and add a constant term (cheat!). The hard way is to compute radiosity, but there are no RenderMan radiosity renderers in the world yet, so we'll ignore this for now.

The second category is images of other objects in the mirror reflections of the surface, and in the refractions of clear surfaces. Reflection is governed by the familiar “angle of incidence equals the angle of reflection” law. There are two equations which influence refraction. Snell's Law tells us which way the refracted vector travels, based on the index of refraction. Index of refraction is really frequency-dependent, and so there should be a different refracted ray for each frequency of light (this is what causes prisms and rainbows), but this is very expensive, so we usually cheat and just send one refracted ray down the middle and call it a day. Fresnel's Equation tells us what percentage of the light is refracted and what percentage is reflected, based on the index of refraction and the incident angle. Few people bother to use this equation, since fixed percentages seem so much easier, but the effect is very important for realism and is not that expensive. One interesting bit about Fresnel's Equation is that it can tell you that the reflected and refracted percentages total to more than 100%. This is *not* an error, so don't be confused. It is a result of the lens effect which focuses the beam and

therefore increases the intensity without increasing the energy or power. This arises because intensity is a measure of power per unit solid angle (watts per steradian), not just power.

There are a large number of algorithms which can be used to calculate global illumination, and they fall into two categories: automatic and preprocessing. Automatic global illumination can be computed at rendering time, and therefore the user doesn't need to do anything other than turn them on. For example, ray traced reflections, or BSP-tree shadows. Preprocessed global illumination must be computed before the main image and stored. For example, in reflection or shadow maps. This initial step must be handled by the user, or handled for him by a smart application program. The existing RenderMan renderers all require preprocessed global illumination maps, and the Shading Language provides hooks for these maps in the form of the `environment` and `shadow` function calls. Even if there was a RenderMan renderer that provided automatic global illumination, it would still support precomputed maps as well, because they are generally quite a bit more efficient, and so can be time savers.

## Basic Approach to Shader Writing

Tony Apodaca

First, we will review the basic RenderMan paradigm and introduce the Shading Language as a tool to write interesting shaders beyond the bounds of the standard models. In the second part, we will look at some simple tricks that will get you into the “shader hacking” mindset. At the end, we will take a look at a few simple programming tricks that we’ve picked up over the years which should make your life a little simpler.

### Why?

Before we get too far along in details of writing shaders, I want to cover a very important concept: “Why?” In this case, I mean, “What purpose is this shader going to serve?” Just like everything else in my talks, there are two categories: general purpose and special purpose

#### General Purpose versus Special Purpose

- **Plastic**
- **Shaders for sale**
- **Custom shaders for individual images**
- **Custom shaders for custom geometry**



Sometimes, we are called upon to write shaders which are truly general purpose. For example, we want to write “plastic” or something equally universal. This type of shader must be carefully engineered to work on all types of geometry, at all shapes and scales. A particularly interesting case of this is when you wish to write shaders which you plan to sell. There are several companies which are now in the business of selling RenderMan shaders, and I hope some people taking this course will eventually do so. Such shaders must be robust, easily controlled by novice users, and general enough to keep customer support to a minimum.

Sometimes, we are called upon to write shaders which are very special purpose. For example, there’s a weird little widget in the back of the picture that “needs a certain something” that we can provide with a special shader. Quite often, though not always, we even have the luxury of manipulating the model so that it is just the right shape to make the shader a little easier to write. These special purpose



shaders occur far more often than you might think. In fact, the vast majority of the shaders Pixar has written fall into this category. (After all, just how many people want to buy a shader for making a half-blackened carrot for a snowman's nose?)

Why bother? Mostly, because we can. Someone went to the trouble to make a special model of that bent nose, and didn't question whether they should bother. One of the basic tenets of this course is that you can and should spend as much time modeling the appearance of an object as you do modeling the shape and position. Perhaps more. I think that realizing that it is perfectly acceptable to write very special purpose shaders for very special purpose tasks frees you to try really interesting tricks and get interesting effects that no renderer except RenderMan can do. *This* is how you get at the real power of RenderMan.


## Tools

Okay, on with the show. The RenderMan Interface provides a very rich shading paradigm for describing the objects in a 3-D scene. The paradigm has four facets which give the user powerful tools to describe appearance.

### RenderMan's Shading Paradigm

---

- **Available geometry**
- **Shading “white boxes”**
- **C-like programming language**
- **Shader global environment**



The first facet is the geometry. RenderMan supports a wide variety of geometric primitive types, including curved surfaces as well as polyhedra. There is a powerful transformation stack, which includes user-named coordinate systems. Both of these can be motion-blurred and subjected to depth-of-field. In addition, the interface supports a user-extensible set of surface attributes. The most obvious among these are color and texture coordinates, which are familiar to users of other renderers. These can actually be described in multiple ways. However, the RenderMan Interface allows the user to invent new parameters and attach them arbitrarily to the surfaces of objects. For example, one might do a physical simulation where the temperature of each polygonal vertex might be valuable information. This temperature would then be visible to the shader, which could make images based on it.

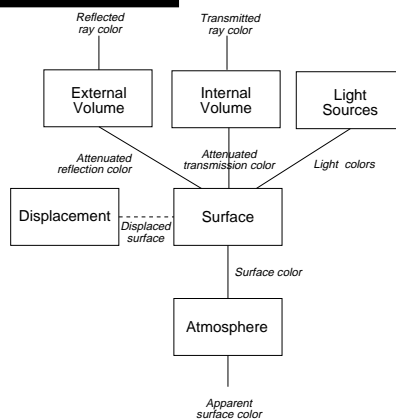
## RenderMan Geometry

- **Parametric curved surfaces**
- **Polyhedra**
- **Texture parameters**
- **User vertex variables**



The second facet is the shader *white box* concept. In a black box, you give the box chosen inputs, and the black box gives you results without you knowing how it computed it. A shader white box is just the opposite. You describe the inside of the box, and the system gives you a unified set of inputs without you knowing where they came from. RenderMan describes several places in the rendering pipeline where users may influence the calculations going on within the renderer. This distinct modularity makes it much easier to isolate concepts, for portability and robustness.

## Shader Evaluation Pipeline



The third facet is the C-like Shading Language which allows the user to write complex functions to fit into each white box. The Shading Language has certain features of C left out and certain new features added in order to fit better as a special-purpose programming language designed specifically for doing the geometric and color calculations necessary for describing appearances.

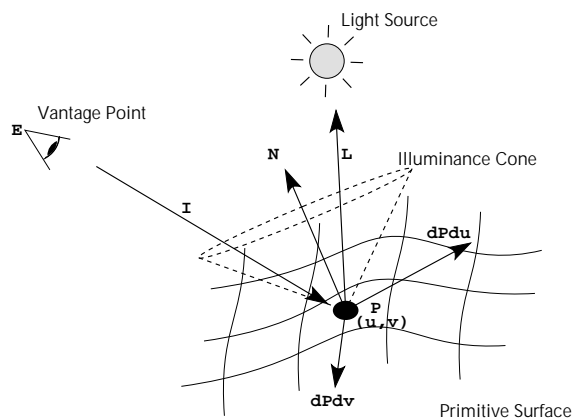
The fourth facet is the rich global environment provided by the renderer to the Shading Language shaders. This environment provides a wealth of information to the shader about the local environment of the point to be shaded. There is a special global state for each type of shader, providing the information relevant to that type. The information available to surface shaders is the most general.

## Surface Graphics State Variables

- |                                |                                      |
|--------------------------------|--------------------------------------|
| • <b>P</b> Surface position    | • <b>Os</b> Surface opacity          |
| • <b>N</b> Shading normal      | • <b>s,t</b> Texture coordinates     |
| • <b>Ng</b> Geometric normal   | • <b>L,Cl</b> Light vector and color |
| • <b>I</b> Incident vector     | • <b>u,v</b> Parametric coordinates  |
| • <b>E</b> Vantage (eye) point | • <b>du,dv</b> Change in coordinates |
| • <b>Cs</b> Surface color      | • <b>dPdu,dPdv</b> Surface tangents  |

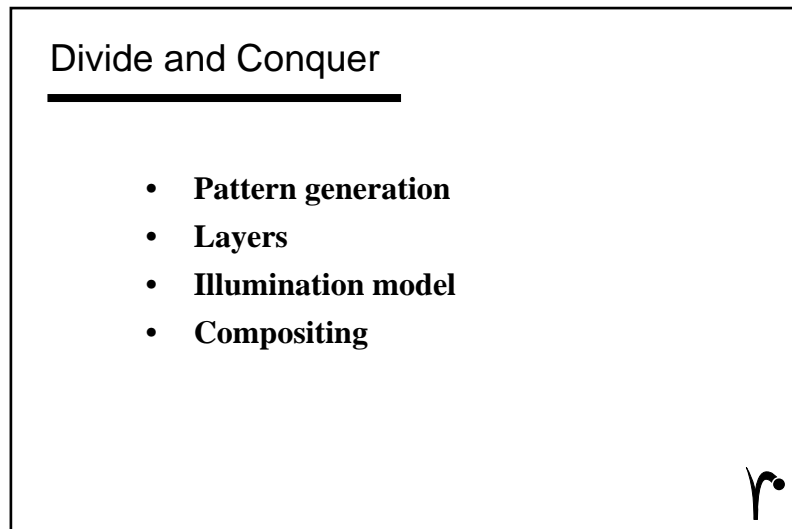


## Geometry at the Surface



## Tricks

The most basic and nearly universal trick to writing shaders, as with any program or subroutine, is *Divide and Conquer*. For shader writing, this means dividing the shading calculation up into four phases.



The first phase is pattern generation. We will spend lots of time on pattern generation through the rest of this course, because for most shaders, this is the most interesting part of the shader. The second phase is layering. Most interesting patterns cannot be described by a single function. Quite often we discover that there are layers of patterns, subtle patterns on top of grosser patterns, or tiny patterns inside larger patterns. It is usually quite helpful to handle each layer separately, and merge them together into the final pattern.

The third phase is the illumination model. For many shaders, we just copy the plastic or metal illumination model, letting the color be derived from the pattern rather than a solid color. For other shaders, the illumination model itself is patterned, or perhaps it is totally non-standard because the particular surface reflects light in a way quite unlike plastic. The fourth phase is compositing. Some shaders have physical layers of totally different characteristics, and the shaders might calculate the patterns and illumination of each physical layer separately, and then composite them together using the standard compositing equations. This divide and conquer methodology lets you create much more complex shaders without losing your mind.

Let's review the standard uses of texture patterns. I use "standard" to describe any use of texturing described in published literature before 1985. The fact that most renderers in the world only let you do these (or usually some subset of these) is only a secondary consideration (wink!).

## Standard Uses of Texture Patterns

---

- **Surface color**
- **Transparency**
- **Bumps and displacements**
- **Reflection and environment maps**
- **Parameter modification**
- **Illumination functions**
- **Projections**



The most familiar use of texture patterns is to set the surface color to an interesting pattern. This was first done by Catmull in 1974. Modulating opacity with texture patterns was introduced by Gardner in 1983. Bump mapping is the perturbing of surface normals based on a texture map. This was introduced by Blinn in 1979. The more obvious but also more difficult version of this technique is displacement mapping, which moves the surface rather than just tweaking the normal vectors. Cook produced the first displacement mapped image in 1984. Reflection maps are simple texture maps which contain the view from a virtual camera located behind a flat mirror. Environment maps are either rendered or drawn images of the entire *world sphere* — a texture which contains the global environment surrounding, but at a large distance from, the objects in the scene. These techniques were also first described by Blinn in 1979, and were used quite extensively by Williams and others at NYIT in the early 1980s.

Parameter modification is the use of a texture map to provide the value for a shading equation variable, such as  $K_s$ , rather than using a single constant value. Illumination mapping is the use of a texture map to provide the light intensity coming from various directions rather than using real light sources. This trick was used at NYIT for years to cut down rendering times before it became generally known in the mid-1980s.

Projections are a technique for choosing the texture map coordinates based on the value of some mathematical equation, rather than being limited to the predefined texture coordinates or parametric coordinates at the vertices of the model. Typically the equations used are sweeps of planes, spheres or cylinders through space, which “projects” the image onto the object in a easily calculated fashion. Barr first described it in the literature in 1984, although the technique was used extensively before that time. This technique spawned Perlin’s and Peachey’s solid textures in 1985.

In the mid-1980’s, it was not uncommon for commercial renderers to provide some or all of these features because they were simple extensions to the single shading equation that was compiled into the program. But adding such features was limited to renderer writers, a very small priesthood. The introduction of the Shading Language removed the barriers to users tweaking the shading equations

in whatever way they wanted to, and incidently made it much more difficult to get a Siggraph paper published which described a new neat feature to add to your shading equation.

So, let's consider some obvious tricks that came to my mind as I was writing these course notes. Several of them are hinted at above. Not that I invented any of these techniques. I call them "obvious" not because they were trivial to come up with the very first time, or that you should have thought of them in the last 5 minutes, but because they give you the flavor of the type of things that start to become simple once you remove those barriers.

### Obvious Tricks

---

- **Splitting effects based on a test**
- **Composited texture layers**
- **Texture map containing texture coordinates**
- **Variables controlling texture in unique ways**
- **Multiple textures**
- **Using geometric info for texturing purposes**
- **Nonstandard illumination functions**



Some surfaces have different parts which are made of distinctly different materials. Consider, for example, a gold inlay on a wooden box. With the Shading Language it is quite easy to use a boolean test at every point on the surface to decide whether to compute a wood shader or a gold shader for that point.

## Splitting Effects

---

```

    if ( length(P - center) < radius ) {
        /* gold */
    } else {
        /* wood */
    }

```

r

This concept can be extended to creating layers of partially transparent colors, which are composited on the fly by the shader until it reaches the innermost or opaque layer. For example, the appearance of a planet might be programmed as several cloud layers over water over land.

## Composited Layers

---

```

float trans = (1 - texture("topcloud" [3]));
Ci = texture("topcloud");
Ci += trans * texture("bottomcloud");
trans *= (1 - texture("bottomcloud" [3]));
Ci += trans * texture("water");
trans *= (1 - texture("water" [3]));
Ci += trans * color(.5, .3, .2); /* brown */
Oi = 1.0;

```

r

One interesting effect is to use a texture map to hold 2- or 3-D coordinates for another texture map access. You could paint some interesting color ramps with odd swirls to create a warped mapping, and then texture any image onto a simple polygon using the warp.

## Mapping of a Mapping

```
float new_s, new_t;
new_s = texture("warp"[0], s, t);
new_t = texture("warp"[1], s, t);
Ci = Cs * texture(texturename, new_s, new_t);
```

r

One feature which many renderers are forced to handle is pseudocoloring of some sort. Typically, this involves the user computing some function of his data and applying these color values to polygonal vertices, and the renderer has little to do but copy these colors onto the screen. In RenderMan, the user could put his *data* on the vertices of his geometry, and write a shader which uses that data either directly or indirectly to influence the color of the object.

## Special Surface Variables

```
surface hotstuff (varying float
    temperature = 273;) {
    color red = color(1,0,0),
    blue = color(0,1,0),
    white = color(1,1,1);
    Ci = color spline(Cs, Cs, red, blue,
        white, white, (temperature - 273)/500);
}
```

r

Another really obvious trick is to add a special texture map to catch a particular feature which can not be handled by the equations running the rest of the appearance. For example, on the bowling pin



image, there are five separate textures controlling the color of the pin. There is a texture for the underlying surface color relative to the height of the pin (which also handles the red crowned section), three texture maps for the three decals applied to the surface in their particular places on the pin, and a texture which has dirt and grit which overlays the rest. There is a sixth texture which controls the pits and gouges which are part of the displacement map.

Sometimes the special texturing effect we desire can be created entirely by taking advantage of information provided in the geometric description of an object. For example, when we needed a glow around a very bright object, we thought about it for a while and realized that we needed something that was bright and opaque near the center, fading to black and transparent near the edges of a circle. Fading out in a roughly cosine or cosine squared way. Can you see this coming? The dot product of  $N$  and  $I$  on a sphere is exactly perfect!

## Glow

---

```
color yellow = color(1,1,0);
Ci = yellow *
    pow(normalize(I).normalize(N), 2.0);
```

*r.*

It's quite fun to play with non-standard illumination functions. Let's consider a LP vinyl record (anyone remember them?). The surface of an LP is not really flat with lots of surface microfacets. It is much more nearly a sawtooth wave, perhaps with some plateaus between the teeth. When light hits this, it reflects very specularly in three preferred directions, rather than one, giving three distinct specular highlights and a very unique surface appearance.

## Nonisotropic Specular

```
point Nf = normalize(faceforward(N, I));
point delta = normalize(dPdu);
color spec = specular(Nf, I, roughness) +
             specular(Nf + delta, I, roughness) +
             specular(Nf - delta, I, roughness);
Ci = Cs * (Ka * ambient() + Kd * diffuse(Nf)) +
        Ks * specularcolor * spec;
```



## Programming Tricks

There are a few simple programming constructs and limitations that you should be aware of before you start off writing your own cool shaders. These were discovered over the years by our shader writers, and knowing them will save you many frustrating hours debugging shaders.

### Programming Tricks

- **RenderMan coordinate systems**  
world, camera  
object, shader, current  
user-defined
- **Vector Transformations**
- **Area operators inside conditionals**
- **Geometry versus shaders**



First, you should understand RenderMan coordinate systems. You can transform points and vectors to any space you need to in the Shading Language using the `transform` function, and it is quite often the case that certain patterns are much easier to generate in one coordinate system than in an-

other. There are four built-in coordinate systems that are very useful. *world* coordinates are obvious, the coordinate system at `RiWorldBegin` from which everything else flows. *camera* coordinates is the space centered around the camera, with the positive *z*-axis directly ahead, *x* left and *y* up.

*object* coordinates is the space centered around the object. That is, the coordinate system which was active at the time the particular geometric primitive was defined. This is slightly different from *shader* coordinates, which is the coordinate system which was active at the time the shader was defined (i.e., when `RiSurface` was called). Sometimes these two are the same, but quite often the model includes transformations between the `RiSurface` and the `RiSphere`, for example.

The Shading Language also includes the concept of *current* space. This abstract space is defined to be whatever the renderer finds most convenient to do its shading in. Some renderers will do shading in *world* space. Other renderers will do shading in *camera* space. In any case, it is the default space for all points and vectors, and it is not a good idea to assume that it is any particular space.

The user can also define his own *named coordinate systems* by using `RiCoordinateSystem` in the model. The shader can transform to and from any of these spaces, as well, and this can be a powerful tool. For example, you may wish to do some of the calculation in the local coordinate system of one of the lights. Simply name the coordinate system when the light is defined, and then use that name in the shader.

The Shading Language has a function to transform points, namely `transform`. However, as we have discussed earlier, direction vectors do not transform quite the same, and if you try to transform a direction with `transform`, you will get screwy shading results. The only correct method is to treat the direction vector as two endpoints, transform those endpoints, and subtract them again to get a direction in the new space. We have developed a trick which is slightly ugly but makes this a little faster. The trick is based on the fact that (perspective aside), the first endpoint of the direction vector doesn't really matter, so you can choose any point which is convenient. The most convenient point is either the origin of the destination space (in which case you don't need to subtract it at the end (it will be (0,0,0)!)), or else the origin of the original space (in which case you don't need to add it at the beginning (it's (0,0,0)!)). In either case, the value is a constant, so you don't have to transform two points every time. The final functions are:

## Vector Transformation

```
v2 = transform("current", "to", v +
               point "to" (0,0,0));
v2 = transform("from", "current", v) -
       point "from" (0,0,0);
```

One of the limitations that the Shading Language has which applies to Pixar's PhotoRealistic RenderMan in particular concerns area operators. Area operators are those Shading Language functions which relate or compare values on adjacent parts of the surface of an object. These functions are: `area`, `Deriv`, `Du`, `Dv` and `calculatenormal`. Although it is not obvious, this list also includes all of the texture mapping functions, because the shader filters all texture map lookups by examining the area of the texture map request in texture space. These functions are: `texture`, `environment`, `bump`, and `shadow`. Because of the algorithm which PhotoRealistic RenderMan uses to compute these area operators, they must not be used inside of conditional statements controlled by varying expressions. That is, when you use an `if` statement, and the controlling boolean expression is *varying* (may have different results on different parts of the surface of some primitive), you should not use any of those 9 area operators inside either the `if`-part or the `then`-part of the conditional. Boolean expressions based on constant or *uniform* expressions (which always have the same result everywhere on the surface of an individual primitive) do not suffer from this restriction.

### Area Ops in Conditionals

```
if (xcomp(P) > 0.5) foo = Deriv(P, s); /* fails! */
if (Ks > 0.5) foo = area(P);          /* okay! */
```



Yes, this is really a bug, but it is one that is not likely to be fixed in the near future, so you should write shaders with this in mind.

The final programming trick is to do less programming. We have already seen that special vertex variables, or convenient geometric properties of objects, or specially designed coordinate systems can help us generate interesting effects. Therefore, it should not be at all surprising that the enterprising shader writer attempts to make his programming job easier by manipulating the geometry to present him with opportunities to use these tricks. For example, glints, glows, hairs, and other surfaces effects which occur “just off the edge” of the surface are usually done with partially-transparent geometry placed conveniently adjacent or surrounding the objects. Placing a special coordinate system conveniently around the object at some angle makes it easy to apply projections to objects at some rotation not perfectly aligned with object space.

Placing texture coordinates with a special purpose program is sometimes the only way to get exactly the effect you want on a surface. For example, objects which morph will travel move thorough tex-

ture space under most solid textures and texture projections. If instead you write a program which attaches the “unmorphed” texture coordinates onto the coordinates of the morphed object, the texture will “stick” to the object.

## Texture Generation

Darwyn Peachey

### What is Texture Generation?

---

- **vary shading characteristics across surface**
- **color, transparency, shininess, bumps, ...**
- **write shaders to make textures**

this is procedural generation



Much of our effort and ingenuity in writing shaders goes into the process we call *texture generation*. Broadly speaking, “texture” is any variation in shading characteristics across a surface. A “plastic” surface that has the same color, specular and diffuse coefficients, and roughness at all points doesn’t have a texture. But most often we write shaders for surfaces that *do* have a texture, simply because those are the interesting surfaces. After all, the “plastic” shader already exists; there’s only so much you can do without texture!

You can vary any shading characteristic as some function of surface parameters. This is a consequence of the arbitrary programmability of shading calculations in the shading language.

## Why is this hard?

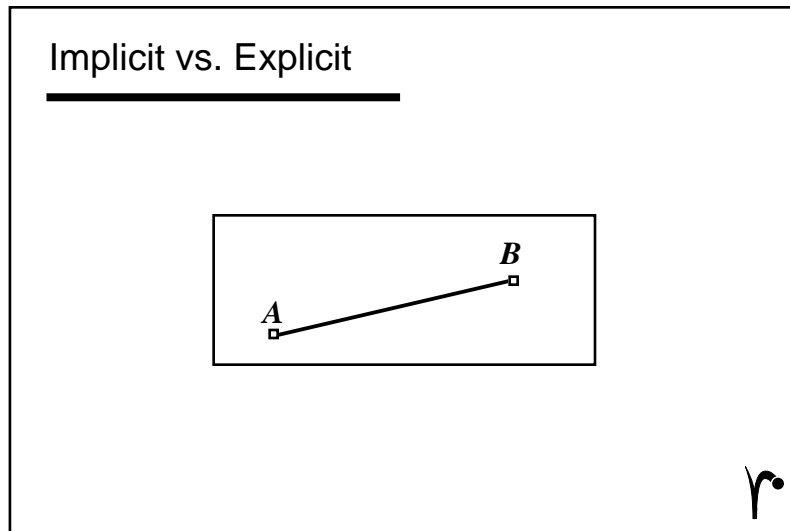
---

- **programming is hard**
- **many degrees of freedom**
- **minimal tools, delayed feedback**
- **implicit vs. explicit imaging model**

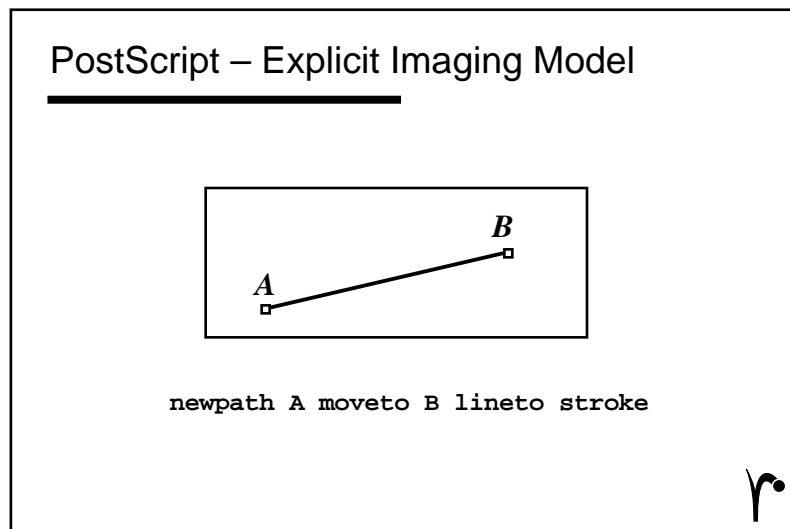


Texture generation by writing a shading language program can be quite hard. That's why we're spending much of the day giving you examples and attempting to identify general principles and techniques of shader construction.

One of the problems in writing and debugging shaders is that you usually have to render a small image using the shader before you can really figure out how close you are to your goal. Rendering is fairly slow. It would be a major step forward in shader writing if we had a tool that gave instant feedback on shader appearance as soon as the code was changed. Ideally, you would be able to manipulate sliders for shader parameters and immediately see the effects of the change. In writing a shader, you are searching an incredibly vast state space of possible textures attempting to find the texture you originally imagined. Often you find another texture that works well enough, or even one that you like better than your original concept. I recommend Karl Sims' paper "Artificial Evolution for Computer Graphics" on pages 319–328 of the *SIGGRAPH '91 Conference Proceedings* for an interesting view of "evolutionary" approaches to exploring this state space.



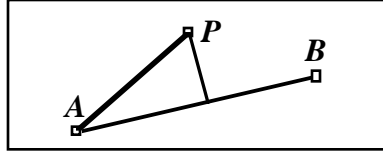
There is another factor that makes the task of shading language texture generation even harder. Many patterns, especially lines, curves, and polygons, are most easily described in an *explicit* fashion.



PostScript is an example of a system which uses the explicit model. Drawing a line from point A to point B is simply a matter of moving the virtual pen to point A and then drawing to point B.



## Shading Language – Implicit Model



```

AB = B-A; len_AB = length(AB);
D = AB/len_AB; AP = P-A;
len_APD = AT . D; APD = len_APD * D;
AP_perpendicular = AP - APD;
dist = length(AP_perpendicular);
if (dist <= linewidth &&
    len_APD >= 0 && len_APD <= len_AB)
    Ct = linecolor;
else
    Ct = bgcolor;

```

Drawing a line from point A to point B is much harder in the shading language, because shapes must be described in an *implicit* fashion. Each time the shader is invoked, it must answer the question: “What are the shading attributes at point P on the surface?” To draw a line, the given point P must be tested to see if it lies on the line from A to B. If so, point P should be assigned the line color; otherwise point P should be assigned the background color.

The code fragment on the slide shows one way to do this. The vector from point A to point P is broken into two components: APD which is the component in the direction AB (also called D), and AP\_perpendicular which is the component in the direction perpendicular to AB. If the length of AP\_perpendicular (distance of P from the line) is less than or equal to the desired line width, then P is on the infinite line through A and B. If the signed length of APD is between 0 and the length of AB, then P lies between A and B.

This rather complex procedure will draw a line from A to B, but it will be a jaggy line: the code, though complex, doesn’t contain any antialiasing features (as will be discussed later today).

## Texture Spaces

---

- **textures defined on some coordinates**
- **2D surface textures, 3D solid textures**
- **2D: (u,v) or (s,t) or arbitrary ones**
- **3D: world space**
- **3D: shader space**
- **3D: arbitrary calculations**



All texture generators are functions of their inputs. The shading language makes several pieces of information available for every surface point. Textures are usually defined in terms of the surface parametric coordinates  $(u, v)$  or the texture coordinates  $(s, t)$ . By default, the texture coordinates  $(s, t)$  are the same as  $(u, v)$ , but the modeler can modify the texture coordinates using the `RiTextureCoordinates` call or by attaching texture coordinate values to the vertices of the surface geometric primitives.

Another type of texture is *solid texture* which is defined on the 3D coordinates of a point in world space or shader space, rather than the 2D surface coordinates. If a solid texture is defined in terms of the “current space” point  $P$ , the texture will be locked to the camera (in PhotoRealistic RenderMan) and will change as the camera pans or approaches the objects in the scene. If the texture is defined in terms of world space, the texture will be tied to the world and an object’s texture will change as it moves through the world. Sometimes this is just what is needed, but most often solid textures will be defined in terms of the shader space location, which will move with the object.

Of course, arbitrary calculations and transformations can be applied to generate texture coordinates in the shader. In most cases you should be careful to keep the texture space continuous and smooth.

## Types of Patterns

- **stored images (texture files)**
- **regular patterns**
- **pure stochastic patterns**
- **perturbed regular patterns**
- **random placement patterns**
- **perturbed access to texture files**



For the remainder of this presentation, we'll look at several types of patterns that have proven useful for texture generation, and give examples of each.

## Texture Files

```
Ct = texture("name.tx");  
Ct = texture("name.tx", ss, tt);  
Ct = texture("name.tx", ss, tt,  
            "swidth", 2, "twidth", 4);  
Ct = environment("name.env", D);
```



Of course, simply accessing a stored texture image (texture file) is the easiest and most obvious thing to do. This is a very powerful way to get realistic or natural textures, by taking photographs of the real materials and then scanning them into your computer. Texture file access is very easy in the shading language, using either the standard texture coordinates ( $s, t$ ) or arbitrarily computed ones ( $ss, tt$ ). The "swidth" and "twidth" parameters can be used to blur the texture somewhat by

widening the texture filter. Environment textures are indexed by a direction instead of  $(s, t)$  coordinates, and can be used to simulate reflection and refraction.

### Disadvantages of Scanned Textures

- **environment of original texture**
- **limited in size**  
seams, unnatural periodicity when repeated
- **limited in resolution**  
pixel size of scan may be visible



It would be nice if implementing a texture consisted only of finding a sample of the desired material, photographing it, and digitizing the photograph. But this approach is rarely adequate by itself. If the material is not completely flat and smooth the photograph will capture information about the lighting direction. Each bump in the material will be shaded based on its slope, and in the worst case, the bumps will cast shadows which obscure other features. Even if the material is flat and smooth, the photograph often will record uneven lighting conditions, light source color, reflections of the environment surrounding the material, highlights from the light sources, and so on. This information generates incorrect visual cues when the photograph is texture mapped on to a scene with simulated lighting and environmental characteristics that differ from those in the photograph. A beautiful photograph often looks out-of-place when texture mapped onto a computer graphics model.

Another problem with a scanned texture image is that it has a finite size and resolution. Because its size is limited, it will be necessary to repeat the texture in order to cover a very large surface. Most scanned textures can't be repeated or "tiled" across a surface without ugly visible seams between tiles. The opposite edges of the texture don't match unless the image is carefully retouched with a paint program. Even after seams are eliminated, the texture can exhibit unnatural regularity or repetition. Prominent features in the texture image are replicated over and over and make the tiled nature of the texture obvious and objectionable. On the other hand, the texture is limited in resolution, so zooming in on the texture may reveal pixel artifacts.

So we embark on the task of learning how to generate textures procedurally, without ruling out the use of texture files where appropriate, or where we can combine texture files and procedural texture to gain the advantages of both.

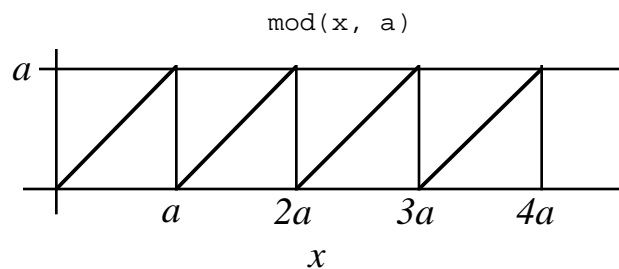
## Regular Patterns

- **have no stochastic component**
- **lines, grids, checkerboards, polygons**
- **built up using standard geometric programming tricks**

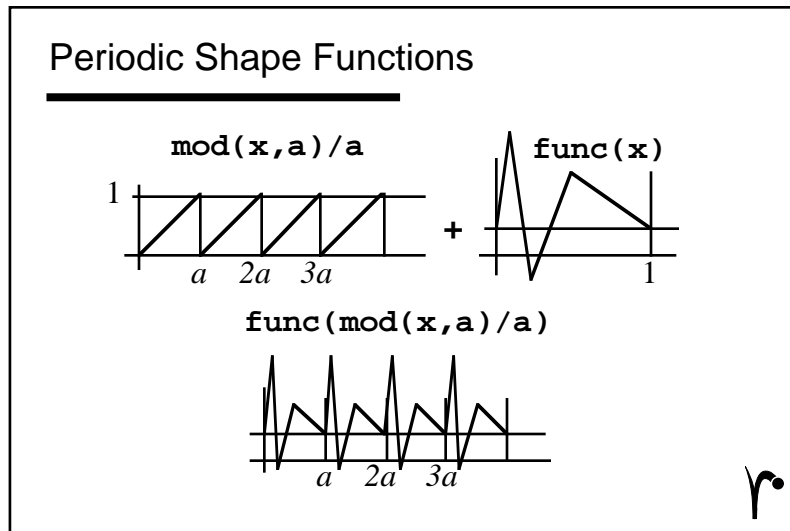
Regular patterns are those without stochastic components. They are created by the application of geometric reasoning and programming cleverness. Here are some of my favorite tricks.

## Favorite Built-In Functions: mod

- **basic building block for periodic patterns**



The mod function is a basic building block for periodic functions.

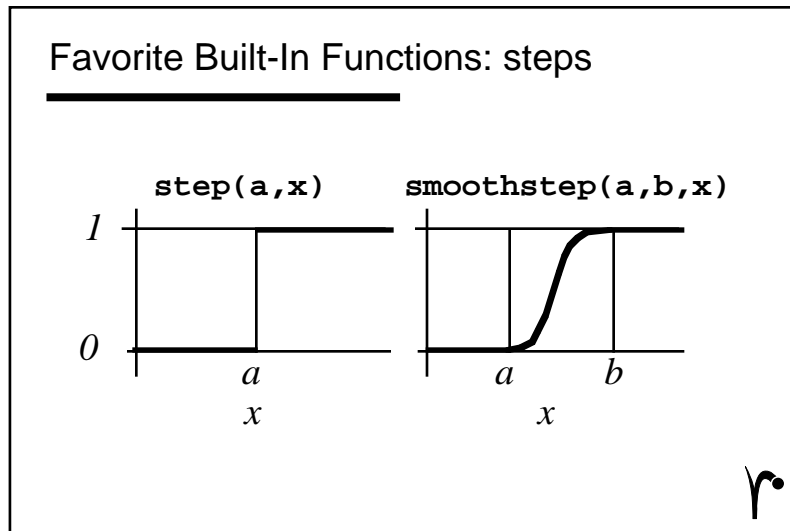


Often we take a texture coordinate and use mod to reduce it to a periodic zero-to-one sawtooth function:  $\text{mod}(x, a)/a$ . Then we define our overall function as a composition of the sawtooth and a basic shape defined on the half-open interval  $[0, 1)$ . The resulting function is a periodic repetition of the basic shape function.

### Favorite Built-In Functions: sin

- **fundamental smooth periodic function**
- **basis of Fourier synthesis**
- **used effectively by Gardner**  
complex regular patterns can look irregular!

A small 'r' logo is in the bottom right corner.



Later today I'll describe an intermediate type of step function called `boxstep` that is somewhere between `step` and `smoothstep`. `Smoothstep` is heavily used to produce gradual transitions rather than sharp ones, often to avoid aliasing artifacts. We'll discuss aliasing in detail this afternoon.

### Steps as Conditionals

---

```

if (a < b)
    Ct = C0;
else
    Ct = C1;

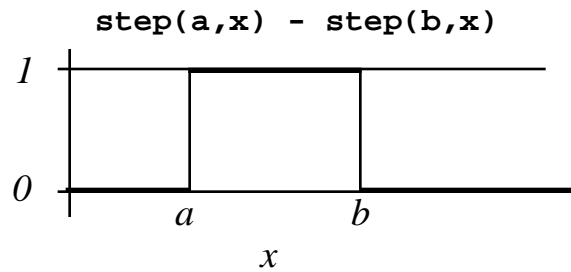
    ...can be replaced by...

Ct = mix(C0, C1, step(b, a));
  
```

r

One advantage of using a `step` construct instead of an `if` statement is that there are easy ways to smooth it out (i.e., `smoothstep`) in order to antialias it. Another advantage is that it avoids the pitfalls sometimes associated with the `if` statement and area operators.

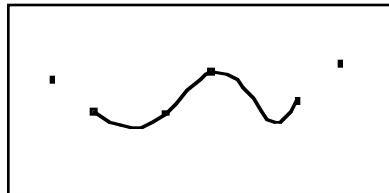
## Two Steps Make a Pulse



r

The pulse is another important functional building block. A rounded pulse can easily be built from `smoothstep` in a very similar way.

## Favorite Built-In Functions: spline

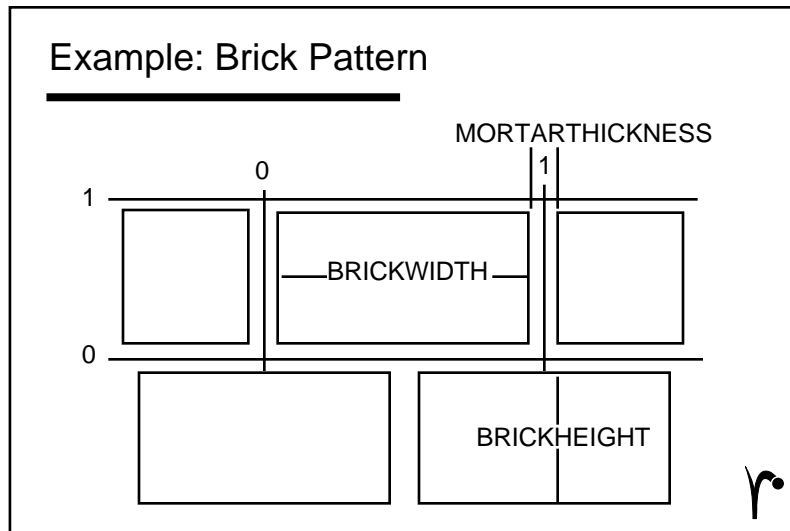


- **Catmull-Rom spline through float or color values**
- **need an extra point at each end**

r

The `spline` function provides a good way to map a continuous float parameter into a collection of colors which change more sharply and help distinguish the different values of the parameter. This is the pseudo-color technique well-known to scientific visualizers.





### Example: Brick Pattern

```

ss = scoord / BMWIDTH;
tt = tcoord / BMHEIGHT;
if (mod(tt*0.5,1) > 0.5)
    ss += 0.5;      /* shift alternate rows */

tbrick = floor(tt); /* which brick? */
sbrick = floor(ss); /* which brick? */
ss -= sbrick;
tt -= tbrick;
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MHF,tt) - step(1-MHF,tt);
Ct = mix(Cmortar, Cbrick, w*h);

```

A small 'r' logo is in the bottom right corner.

Here's our first serious example of a regular pattern, namely, a simple brick pattern. `BMWIDTH` is the width of the brick plus the mortar, and `BMHEIGHT` is the height of the brick plus the mortar. `MWF` is the "mortar width fraction" and `MHF` is the "mortar height fraction." `tbrick` and `sbrick` are unique coordinate values for each brick. They will be useful in the "improved" versions of this shader shown later. The key to the version of the brick shader on this slide is in the two pulses `w` and `h` which control the horizontal and vertical color selection between mortar color and brick color.

Here's the full shader listing (too long to put on a slide!):

```

#define BRICKWIDTH      0.25
#define BRICKHEIGHT     0.08
#define MORTARTHICKNESS 0.01

#define BMWIDTH          (BRICKWIDTH+MORTARTHICKNESS)
#define BMHEIGHT         (BRICKHEIGHT+MORTARTHICKNESS)
#define MWF              (MORTARTHICKNESS*0.5/BMWIDTH)
#define MHF              (MORTARTHICKNESS*0.5/BMHEIGHT)

surface
brick(
    uniform float Ka = 1;
    uniform float Kd = 1;
    uniform color Cbrick = color (0.5, 0.15, 0.14);
    uniform color Cmortar = color (0.5, 0.5, 0.5);
)
{
    color Ct;
    point NN;
    float ss, tt, sbrick, tbrick, w, h;
    float scoord = s;
    float tcoord = t;

    NN = normalize(faceforward(N,I));

    ss = scoord / BMWIDTH;
    tt = tcoord / BMHEIGHT;

    if (mod(tt*0.5,1) > 0.5)
        ss += 0.5;          /* shift alternate rows */

    tbrick = floor(tt); /* which brick? */
    sbrick = floor(ss); /* which brick? */
    ss -= sbrick;
    tt -= tbrick;

    w = step(MWF,ss) - step(1-MWF,ss);
    h = step(MHF,tt) - step(1-MHF,tt);

    Ct = mix(Cmortar, Cbrick, w*h);

    /* "matte" reflection model */
    Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(NN));
}

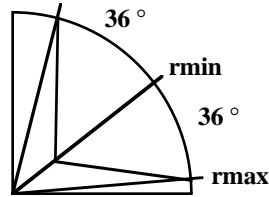
```

### Example: Star

```

sa = 2*PI/npoints;
p0 = rmax*(cos(0),sin(0),0);
p1 = rmin*
    (cos(sa/2),sin(sa/2),0);
ss = s - sctr; tt = t - tctr;
angle = atan(ss,tt) + PI;
r = sqrt(ss*ss + tt*tt);
angle /= sa;
angle -= floor(angle);
if (angle < 0.5)
    angle = 1 - angle;
d0 = p1 - p0;
d1 = r*(cos(angle),sin(angle),0) - p0;
Ct = mix(Cs,starcolor,step(0,zcomp(d0^d1)));

```



*r*

The star is another regular pattern, one that looks quite hard until you think about it in polar coordinates. The diagram shows that each point of a five-pointed star is 72 degrees wide. Each half-point (36 degrees) is described by a single edge. The shader converts the parameters ( $s$ ,  $t$ ) into polar coordinates with respect to the center of the star, and then determines which half-point it might be in from the angle. In that half-point, it finds the vectors  $d0$  from the tip of the star point to the  $rmin$  vertex and  $d1$  from the tip of the star point to the point ( $s$ ,  $t$ ). The sign of the cross product of the two vectors determines whether ( $s$ ,  $t$ ) is inside the star or outside of it.

### Stochastic Patterns

- **irregular patterns have stochastic component**
- **special meaning of “stochastic”**
- **not random! repeatable functions of inputs**
- **no apparent patterns; pseudo-random**
- **noise() function is our stochastic building block**

*r*

The remaining types of patterns are all “irregular,” that is, they have a stochastic component. However, the term *stochastic*, which usually implies “random,” is being used in a special sense here. Our stochastic functions are not random at all. In fact, our stochastic functions take inputs and are repeatable, deterministic functions of those inputs. This is essential to our purpose, because we need functions which remain the same from frame to frame of an animated sequence. On the other hand, we don’t want to see a visible, regular pattern in the stochastic function — it must be pseudo-random.

## Noise Function

---

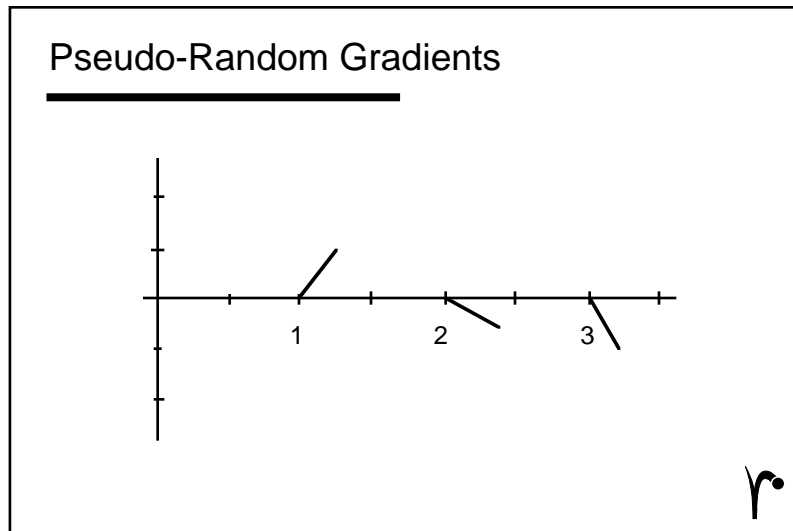
- **repeatable pseudo-random function**
- **1D, 2D, or 3D input**
- **float, color, or point output**
- **value ranges from 0 to 1**
- **value is 0.5 at integer locations**
- **oscillates between integer locations**
- **snoise = 2 \* noise - 1**



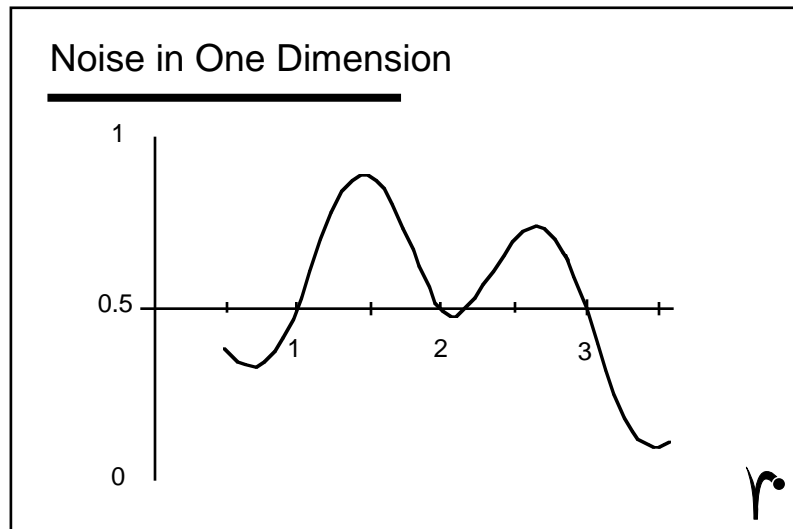
The `noise()` function is our basic stochastic primitive. The `noise` function was introduced by Ken Perlin in his Siggraph ‘85 paper “An Image Synthesizer.” Perlin and Hoffert give a more precise definition of the function on pages 255 and 256 of the Siggraph ‘89 paper “Hypertexture.” J.P. Lewis analyzes the deficiencies of this `noise` function in the Siggraph ‘89 paper “Algorithms for Solid Noise Synthesis.”

Note that the RenderMan `noise` function ranges from 0 to 1 with a value of 0.5 at integer points, while Ken Perlin’s `noise` function ranges from -1 to 1 with a value of 0 at integer points. It is sometimes convenient to define a signed noise function `snoise` which behaves like Perlin’s function:

```
#define snoise(x) (2 * noise(x) - 1)
```



A pseudo-random function of the input parameters is used to generate a gradient vector at each integer point.



The pseudo-random gradient vectors determine the shape of a smooth function that passes through 0.5 at each integer point.

## Properties of Noise

- **continuous and smooth**
- **dominant frequency of 0.5 to 1**
- **approximately band-limited**  
little high-frequency content



Although the noise function is not, strictly speaking, band-limited, it does have a dominant frequency which can be used to build up more elaborate stochastic functions by summing noises of different frequencies. Aliasing artifacts can be reduced by simply not including higher frequency noise components when the sampling rate is too low.

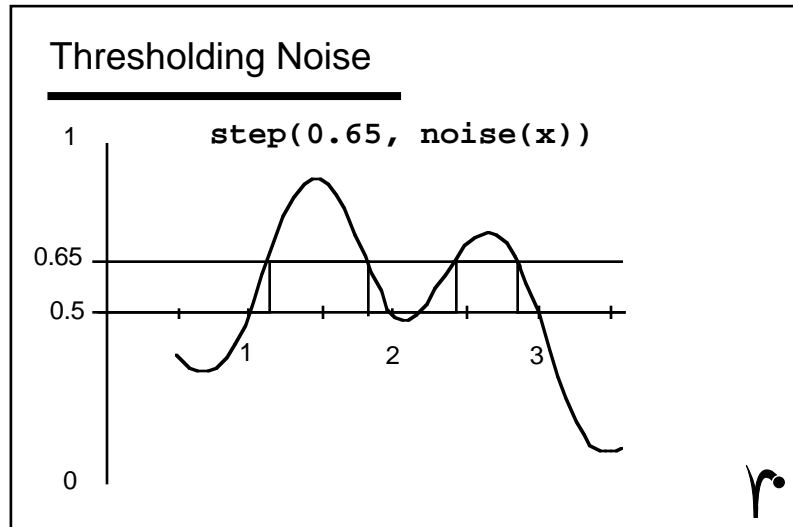
## Building Stochastic Functions

- **transform noise inputs:**  
offset:  $\text{noise}(x + k)$   
frequency:  $\text{noise}(f * x + k)$
- **sum multiple components**  
spectral synthesis



There are many ways of using noise. Different frequencies can be weighted and summed to produce noises with desired power spectra. Offsets and rotations of the noise parameter provide different noise values and avoid directional artifacts which sometimes appear in noise. Raw noise has its own

characteristic appearance; other patterns can be created by perturbing regular patterns with a little noise. Threshold functions like smoothstep can be set up to change rapidly if noise exceeds a certain value; this creates more sharply defined edges within the otherwise smooth noise function.



Pure stochastic patterns consist of sums of noise components of various frequencies. They tend to lack any regular structure and can be called “amorphous.”

### Pure Stochastic Patterns

- “amorphous” patterns
- noise with specified frequency spectrum
- thresholding, color mapping to emphasize pattern
- fluid flow phenomena: vapor clouds, stone materials formed by mixing

Several calls to `noise` can be combined to build up a stochastic function with a particular frequency/power spectrum. Repeated calls of the form

```
value += amplitude * noise(Q * f)
```

with amplitude varying as a function of frequency  $f$  will build up a value with any desired spectrum.

The *turbulence* function described by Ken Perlin [9] is essentially a stochastic function with a “fractal” power spectrum, that is, a power spectrum in which amplitude is proportional to  $1/f$ .

```
value += abs(snoise(f*s))/f;
```

Derivative discontinuities are added to the turbulence function by using the absolute value of the `snoise` function. Taking the absolute value folds the function at each zero crossing, making the function undifferentiable at these points. The number of peaks in the function is doubled since the troughs become peaks, and the overall frequency is therefore doubled as well.

As an example, we’ll go through a marble shader which uses a “spectral function” synthesized from four noise components with a  $1/f$  power spectrum. This “fractal noise” is reminiscent of *turbulence*, but it doesn’t have the absolute value feature.

### A simple marble shader

```
float texturescale = 1;
point PP;
float i, f, marble;
#define NNOISE 4
marble = 0;
f = 1;          /* starting frequency */
for (i = 0; i < NNOISE; i+= 1) {
    marble += noise(PP * f) * 1/f;
    f *= 2;
}
marble = clamp(4*marble - 3, 0, 1);
CT = marble_color(marble);
```



CT is set to a color `marble_color` which is a function of `marble`, the value of a stochastic function with a “fractal” power spectrum.



## A very simple color selection function

```
color
marble_color(float m)
{
    return m * PALE_BLUE;
}
```

Slightly more sophisticated:

```
color
marble_color(float m)
{
    return mix(PALE_BLUE, DARKER_BLUE, m);
}
```

r

`marble_color` can select a color just by scaling the pale blue, or by blending between a pale blue and a darker blue.

## A better color selection function

```
#define PALE_BLUE      color (0.25, 0.25, 0.35)
#define MEDIUM_BLUE   color (0.20, 0.20, 0.30)
#define DARK_BLUE      color (0.15, 0.15, 0.26)
#define DARKER_BLUE    color (0.10, 0.10, 0.20)
color
marble_color(float m)
{
    return color spline (m,
        PALE_BLUE, PALE_BLUE,
        MEDIUM_BLUE, MEDIUM_BLUE, MEDIUM_BLUE,
        PALE_BLUE, PALE_BLUE,
        DARK_BLUE, DARK_BLUE,
        DARKER_BLUE, DARKER_BLUE,
        PALE_BLUE, PALE_BLUE);
}
```

r

The best results are achieved by selecting a color from a spline function which passes through a number of shades of blue.

### Example: Stucco Wall

```

PS = transform("shader", P) * scale;
PQ = (xcomp(PS)+ycomp(PS), xcomp(PS)-flycomp(PS),
      zcomp(PS));
f = noise(PQ) + noise(PS);

bh = smoothstep(1.1, 1.2, f); /* mesas */
NN = normalize(faceforward(N, I));
PP = P - NN * depth * (1 - bh);
NN = normalize(calculatenormal(P));

```

...and shade with "matte".



This wall shader was used in a commercial we produced for "Cellular One." The version on the slide is somewhat simplified for presentation. Here is the complete shader:

```

surface
wall(float Ka = 1, Kd = 1;
    float scale = 0.4, depth = 0.02)
{
    point PS, PQ, NN, PP;
    float f, bh;

    PS = transform("shader", P) * scale;
    setxcomp(PQ, xcomp(PS) + ycomp(PS));
    setycomp(PQ, xcomp(PS) - ycomp(PS));
    setzcomp(PQ, zcomp(PS));
    f = noise(PQ) + noise(PS);

    /* add some higher frequencies to roughen edges */
    f += 0.3 * (noise(PS * 3.713)-0.5);
    f += 0.2 * (noise(PS * 7.436)-0.5);

    /* threshold the function to get "mesas" in 0:1 range */
    bh = smoothstep(1.1, 1.2, f);

    /* add higher frequency bumps, only where the low
       * freq ones aren't */
    bh += (1 - bh) * smoothstep(0.7, 0.8, noise(PQ * 2.5));
    bh += (1 - bh) * smoothstep(0.7, 0.8, noise(PS * 2.9));
}

```

```

bh += (1 - bh) * smoothstep(0.7, 0.8, noise(PS * 4.3));
bh += (1 - bh) * smoothstep(0.7, 0.8, noise(PQ * 5.7));

/* bh is between 0 and 1 */
PP = P - normalize(N) * depth * (1 - bh);
NN = calculatenormal(P);

Ci = Os * Cs * (Ka * ambient() + Kd * diffuse(NN));
Oi = Os;
}

```

### Perturbed Regular Patterns

- **begin with a regular pattern**
- **add noise to calculation to perturb pattern**
- **makes pattern irregular, more interesting or “natural”**



The most useful type of pattern is the “perturbed regular pattern,” which combines a basic regular structure with elements of irregularity to keep it interesting. Most natural materials are somewhat irregular and non-uniform. Even man-made materials are irregular due to poor quality control, shipping damage, and weathering.

### Example: Improved Brick Shader

```
ss = scoord / BMWIDTH;
tt = tcoord / BMHEIGHT;
if (mod(tt*0.5) > 0.5)
    ss += 0.5;      /* shift alternate rows */
tbrick = floor(tt); /* which brick? */
ss += 0.2 * (noise(tbrick+0.5) - 0.5);
sbrick = floor(ss); /* which brick? */
ss -= sbrick;
tt -= tbrick;
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MHF,tt) - step(1-MHF,tt);
Ct = mix(Cmortar, Cbrick, min(w,h));
```



In the brick shader, we can add a line of code which perturbs the horizontal location of each row of bricks a little to suggest that the bricklayer was human and therefore error-prone.

### Random Placement Patterns

- **regular features or subpatterns**
- **dropped at random positions, orientations in texture space**
- **sometimes called “bombing”**




“Bombing” or placing subpatterns at random positions and orientations in the texture space is a useful trick. In the following wallpaper example, we break up the texture space into a grid of cells. Each cell has a star in it, and the location of the center of the star varies depending on noise. To make the texture more irregular, we use a noise value to decide whether to place a star in a given cell or leave it empty.

## Example: Wallpaper Shader

---

```
#define CELLSIZE (1/NCELLS)
ss = s * NCELLS; tt = t * NCELLS;
scell = floor(ss); tcell = floor(tt);
sctr = CELLSIZE * (scell + 0.5 +
    0.6 * snoise(scell+0.5, tcell+0.5));
tctr = CELLSIZE * (tcell + 0.5 +
    0.6 * snoise(scell+3.5, tcell+8.5));
ss = ss * CELLSIZE - sctr;
tt = tt * CELLSIZE - tctr;
```

...and now just use the star example, omitting the first 1 

## Perturbed Access to Texture Files

---

- **access texture file adding noise to calculation of texture coordinates**
- **humorous distortions**
- **adds variety to natural textures such as wood**
- **hides obvious repetition in repeated texture tile**



Texture files can be made more interesting by accessing them using perturbed coordinates. This introduces additional irregularity in the texture and can hide some of the artifacts that result when the same texture file is repeated many times across a surface.

## Example: Perturbed Access

Access texture file with “noisy” (s,t) coordinates

```
Psh = transform("shader", P);  
ss = s + 0.2 * noise(Psh) - 0.1;  
tt = t + 0.2 * noise(Psh+(1.5,6.7,3.4)) - 0.1;  
Ct = texture("name.tx", ss, tt);
```

*r*

# Writing Surface Shaders

Tom Porter

## Opening comments

Let's leave the textbook examples behind and consider shader writing in everyday life. I want to give you a candid view of a couple of days of shader writing in the animation group at Pixar. I hope that you'll find that the process is straightforward, that there are a few simple principles to follow in putting shaders together, and that it is fairly easy to achieve *some* success. I will let Rick and Eliot talk later about how hard it may be in achieving *complete* success.

### Overview

---

- **Let's get beyond the textbook**
- **Consider some real objects**
- **What do you need to know?**
- **What do you need to do?**
- **What should you watch out for?**



I am going to talk only about surface shaders. *Surface* shaders comprise at least 90% of the shaders that get written. Production teams that we spend on a 30-second TV spot. We expect to write perhaps 1500 shaders for the movie we are undertaking.

I am going to talk only about simple shaders. *Simple* shaders comprise at least 90% of the shaders that get written. I will leave it to Rick and Eliot to show some really clever stuff; my point is to give you a glimpse of the process.

What do you need to know before writing a shader? First, read the section on the shading language in the RenderMan book. Second, I found the notes to last year's SIGGRAPH course to be terrific. Third, consult all example shaders in the book, and in the notes, and those written by your colleagues. Finally, a high school math education helps, especially all that stuff about sines and cosines and other smoothly varying functions.

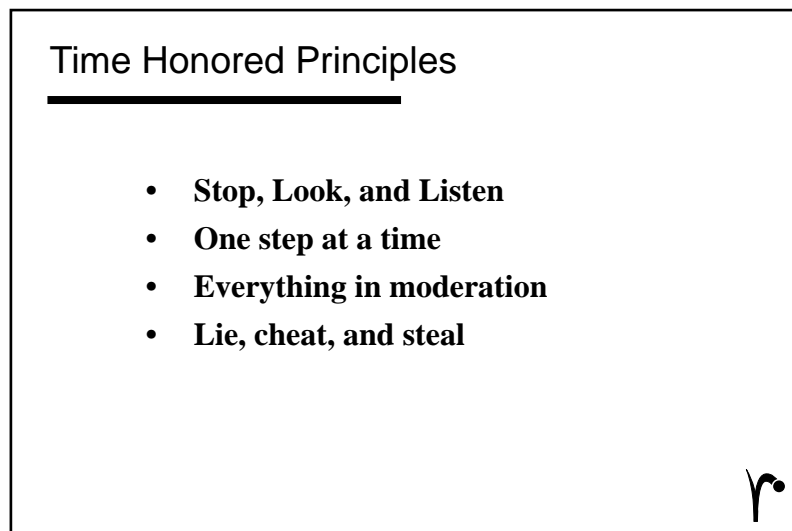
What do you need to do in writing a shader? That's the major portion of my talk. The quick and obvious answer is:

- Make sure you have good geometry.
- Know what you're trying to achieve.
- Start with a shader closest to what you are aiming for.
- Work on various aspects of the look one layer at a time.

What should you watch out for? Slow rendering times, aliased images, wasted effort. I will speak to these points throughout the talk. The overall principle here is to create a shader only as good as it needs to be.

## Axioms of shader writing

I am not going to tell you much that your mother didn't tell you:



- *Stop, Look, and Listen.* Don't set foot until the street is clear and you have an idea how to get to the other side.
- *One Step at a Time.* Build up your shader one layer at a time, mimicking the manner in which the simulated object achieves its real surface characteristics, whether grown or manufactured.
- *Everything in Moderation.* There are (Nyquist) limits beyond which your shader should not go. Take care.
- *Lie, Cheat, and Steal.* This is only computer graphics, a science of simulation. The shader needs to be accurate only within a certain range of conditions. Steal code from your previous shaders and patch together something that works.



**Know what you're after!****Pattern Generation: Divide & Conquer**

---

- **Bowling pins have color, scratches, labels**
- **Soccer balls have color, bumps, dirt, decals**
- **Leaves have color, blemished, striation**



As with many programming tasks, you should adopt a strategy of divide and conquer in planning the shader. Bowling pins have color, scratches, and imprinted labels. Soccer balls have a color pattern, bumps and dirt. Leaves have a color pattern, blemishes, and striations.

You will need such a plan of action in creating the shader. Break down the surface features into distinct characteristics and concentrate on the most significant ones. Failure to do this at the outset will hamper you throughout the process. Resist the urge to hack a shader from the very start -- there will be excellent opportunities for hacks as the shader progresses!

**Know What You're After**

---

- **Get physical models, photos, renditions**
- **Understand how the real object is made**
- **Decide on the accuracy needed**



You must understand what you are trying to achieve with the shader. Get as many physical examples of comparable surfaces as you can. Rip pages out of picture books and post them on your wall.

Consider the mango. We accepted a job to animate fruit for multiple Tropicana juice ads. I had the task of putting together shaders for several of the fruit. I am going to walk you through the process of writing such shaders. This will be a fairly complete look at the process, showing the shader code and the test pictures at several stages of the process.

The first order of business is to buy mangoes. The ads called for a realistic depiction of the fruit, so I purchased some realistic fruit. Of course, television realism differs from grocery store realism, but I use nature as a starting point nonetheless. In fact, this point about realism needs to be made again. Very often, a realistic simulation of a physical object evolves into a realistic simulation of the shared and glorified human expectation of the look of an object. The man in the street expects a mango to look *different* than it really does, but we *still* use nature as a starting point.

## Consider The Mango

---

- **Buy a few mangoes**
- **Notice a base color, little spots, and larger marks**
- **Cylindrical growth; singularities at stem and base**



Next, contemplate your mango. Stare at it. Understand how nature grows it. Theorize on what causes variations among mangoes. What is its color? What is its reflectivity? What is its bumpiness? Take as much time as you need to understand the 4 or 5 principal features of the surface.

An hour with a mango will convince you that it has significant low frequency color variation, lots of tiny dark spots, a minor amount of wrinkling, and a stem at the top. Some people can deduce this sort of information in a matter of minutes, but the serious shader writer will walk the hallways with mango in hand for at least an hour.

Now you need to find out the range over which your mango will be seen in the animation. Allocate your time for writing this shader based on the maximum number of pixels covered in any frame. If your mango is just making a cameo appearance in the back of a large crowd scene, stop now and use the plastic shader with an orange/green tone. If your fruit is to be rendered full screen, get out the magnifying glass and spend another hour meditating about your mango.

## Pattern & illumination

### Shaders Involve Pattern and Illumination

- **Start with plastic.sl**
- **Copy the illumination function**
- **Invent the pattern**
- **Your job: create some clever function for Cs**
- **...or maybe Os, Ks, Kd**



A shader involves pattern generation and illumination. Look at any shader in the RenderMan book, and you will find a fairly standard illumination function at the end, with contributions for ambient, diffuse, and specular light. Everything else is a pattern that you devise, usually from scratch. Your shader will probably have a standard illumination function. In fact, a good starting point is to copy the code from `plastic.sl` and start from there.

Render a picture with `plastic.sl`.

#### But first, a digression on test rendering

I have the luxury of fast workstations, and you may conclude that I employ lots of test renderings because I can afford to abuse CPU time. This is true. But no matter what machines you render on, you will want to render tests quickly. To that end, you should clearly understand the effect of the `RiFormat`, `RiShadingRate` and `RiPixelSamples` calls to RenderMan. These are the best to adjust for the purpose of fast test renderings.

The total rendering time is proportional to the cost of computing each shader at various points over the surface of each possibly-visible object on the screen. Use `RiFormat` to limit the resolution of the target image. Use `RiShadingRate(4)` to ask the renderer not to compute the shader more than once every 2x2 pixel area. Use `RiPixelSamples(2,2)` (or (1,1)) to limit the number of hidden surface determinations to only 4 (or 1) per pixel.

The plastic mango will confirm two things: the geometry and the lighting. Both geometry and lighting should be adequate before proceeding. Do not expect to fix either of these in the shader.

## Plastic Mango

---

- **Plastic illumination function will do**

```
Ci=cs * (ka*ambient() +
         kd*diffuse(Nf)) +
      ks*specular(Nf,V,roughness);
```

How shall we set cs, ka, kd, ks, Nf?



By copying the illumination function from `plastic.sl`, you have completed half the shader. Good work. Now you must invent the pattern. You will notice several coefficients in the plastic illumination function, such as `cs`, `os`, `ks`, `kd`. These are often locked down as parameters to the shader and written as `CS`, `OS`, `KS`, `KD`. I write them in lower case to stress that each of these can be smoothly varying function. Your job in inventing the pattern for the surface is usually to design some sufficiently clever function for `cs`. This is a classic procedural texture map, one that varies the color over the surface. By setting patterns for other variables, you can create opacity maps and bump maps and lots of other effects.

### Procedural vs. photographic texture patterns

## Patterns: Photographic vs. Procedural

---

- **Procedures scale, can be tuned up**
- **Photographs scan, can be touched up**
- **Both need “wrapping”**
- **Guiding Principle:**  
Use photos for artwork, procedures otherwise



You will often face the decision of writing a procedure or scanning a photograph to describe a pattern. Both are viable alternatives, with pros and cons. The fact that RenderMan offers a shading language doesn't mean that you should *not* use a photograph; it just tempts you with greater flexibility in writing procedures. Quite often, you will find that the problem of wrapping the pattern onto the surface is as hard as detailing the pattern itself.

Procedures can scale nicely as the object moves. Procedures are easily adjusted, just a minor matter of reprogramming. Photographs can easily be scanned and touched up, just a minor matter of re-painting. The guiding principle that I live by is to use scanned photographs for artwork, such as product packaging, and to try my hand at programming everything else.

Thus, for bowling pins, scan the Brunswick decal and program the color variations over the surface. For the scratches and dirt, try writing a procedure. In 1989, I painted the scratches and bumps. Today, I might be more tempted to program them. Mango skins should be programmed -- they are too messy on a scanner anyway.

### Mango: base color

#### Mango: The First Attempts

---

- **Start with `plastic.sl` to confirm geometry**
- **Use `show_st.sl` to understand orientation**
- **Set up lighting for test shot**
- **Try library shaders to gauge color, bumpiness**



After starting with the plastic shader, I tried two shaders from the shader library that we keep. The first was a peach shader, just to look at the nature of color variation over the surface. It is clear that the mango has a far lower frequency variation. The second is a simple bump-mapped shader used for a wall in *Tin Toy*. I was interested in comparing the results with the natural bumpiness of mangoes. I set it up with my first attempt at an orange-green color, and produced a picture vaguely reminiscent of a chocolate leather. I quickly squirreled *that* shader away for some future use.

It was clear that neither of these would lead anywhere, so I returned to `plastic.sl` and set out to design a pattern ranging over the  $s$ - $t$  of the mango geometry. I thus ran a test with `show_st.sl` to understand how  $s$  and  $t$  varied over the surface.

Next, I set out to simulate the distinctive color variation of the mango. I needed a variable which would vary slowly over the surface. Note that I needed the variable  $i$  to “wrap” in  $s$ , showing no seam as  $s$  wraps around from 1 back to 0. Thus, I use  $(\sin(2\pi s), t)$  instead of  $(s, t)$  as arguments to the noise function. Notice that the  $(s, t)$  really is appropriate here; mango skin does expand in a cylindrical fashion, creating singularities at the top and bottom.

### Mango: Base Color

---

- **Compute a smoothly varying index variable**

```
i = noise(.5*sin(2*PI*s), 2*t);
```

- **Index into a spectrum of color**

```
spotcolor = spline(i, grn, ylw,  
                  org, red);
```



Now I use the spline function to turn the variable  $i$  into a smoothly varying color variable  $cs$ . That is what I plug into in the illumination function, to good effect.

### Mango: spots and marks

The color of the spots of the mango vary across the surface. The reddish parts of mangoes I buy in Marin County grocery stores have yellow spots, and the yellowish parts have white spots. It is a simple matter to run the spline function through a new color table to compute the spot color. The spots themselves are positioned by computing a new variable  $darken$ . We take the noise function with parameters  $(150*s, 150*t)$  to create a smoothly varying function with at least 150 “cycles” in it. We then square the function as a way of sharpening the peaks of the function and pushing most of the values less than 0.25. We then use this variable in mixing the spot color with  $cs$ .

## Mango: Spot Color

---

- **Use high frequency noise for position of spots**

```
darken = float noise(150*s,150*t);
```

- **Sharpen the peaks of the noise function**

```
darken = pow(darken,2);
```

- **Mix the spots in with the color**

```
cs = mix(c,spotcolor,darken);
```



This has been the standard use of the `noise` function, to create a random and smoothly varying function with some guaranteed frequencies to it. As a further refinement, I decided to darken the tips of the spots. If the `darken` variable is greater than 0.33, I mix in the darker *mark color*.

### Stop! A digression on transitions and aliasing

This last statement should set off alarms in the head of any self-respecting shader writer. The use of an `if`-statement in a shader is dangerous! Up until now, we have been dealing with smoothly varying functions. The `if`-statement allows us to go beyond the safety of these functions. Transitions are evil, and `if`-statements cause transitions.

Under no circumstances should you create a discontinuity in the the color of the surface. This will almost certainly create visible aliasing. Similarly, you should avoid discontinuities in the reflectivity or any other variable contributing to the illumination function, though the visual effect of such discontinuities may be less pronounced. In fact, we do keep the color function continuous in this code, but we do create a discontinuity in the first derivative of color. This can be the cause of Mach banding, and we should watch for that in looking at the rendered pictures.

Darwyn will be discussing tools such as `smoothstep()` used to keep functions smooth and combat various forms of aliasing. Rick will show off some elegant examples. For our purposes here, notice that I keep my functions continuous and don't put too many wobbles in them.

## Mango: Mark Color

---

- **Use lower frequency noise for position**

```
darken = float noise(30*s,30*t);
```

- **Leave very few peaks of the function**

```
darken = pow(darken,5);
```

- **Use spline again to create spectrum**

```
markcolor = spline(i,ylw,bej,brn,  
                  bej,ylw);
```



Every mango I looked at had black marks, skin blemishes caused by growth, picking, or handling. I noticed perhaps 5 to 10 marks per mango, the size of your smallest fingernail. These are computed by taking a medium-frequency noise function in  $s$  and  $t$ . We then sharpen it by putting to the 5th power, so that very little of it is greater than 0.05. We then run the function through a sin function to round it out a little bit. We then use this variable to control the mixing of a dark mark color.

### Mango: generalize, wrinkle, and finalize

## Mango: Generalize

---

- **Introduce “label” argument**

```
darken = noise(150*(s+label),  
              150*(t+label));
```

```
darken = noise(30*(s+label),  
              30*(t+label));
```





We now introduce the `label` argument in the shader, to generalize this code for use on multiple mangoes. We want these characteristic color, spot and mark computations to occur for each mango, without producing clones. We therefore give each mango a label from 0 to 1, and use `label` to access each noise function that we call. This provides the variability we want.

### Mango: Bump Map For Wrinkle

- **Define `p2 = (sin(2*PI*s),2*t,cos(2*PI*s));`**  

```
#define AMPLITUDE1/40
#define FREQUENCY 25
turb = noise(FREQUENCY*p2)*AMPLITUDE;
newP = calculatenormal
        (P+turb+normalize(N));
Nf = faceforward(normalize(newP,I));
```



Until now, we have carried along this computation of `Nf` to be a forward facing normal to the surface of the mango. Let's adjust that slightly to give the impression of wrinkles on the surface. This is bump-mapping. We compute a point `p2` in space and run that through a noise function, subject to a defined frequency and amplitude. This value is used move the surface point to some other spot along the surface normal.

### Mango: Align Wrinkle With Spots

- **A different bump map**  

```
p2 = transform("shader",P);
turb = noise(FREQUENCY*p2);
if (turb<.8) turb = 0;
else turb = (turb-.8)/.2;
turb = turb*AMPLITUDE;
newP = calculatenormal
        (P+turb+normalize(N));
Nf = faceforward(normalize(newP,I));
darken = turb;
```



In the course of playing around with bump maps, I did take some wrong turns. I had this great idea of aligning the color spots with bumps. I thus ran through the bump mapping code, use the computed turbulence function as the variable `darken`, used for the spot color. I tried it two different ways, with different bump frequencies and amplitudes. As the pictures show, the dimples and pimples are not too pleasing.

### Mango: Final Shader

- **plastic illumination; spots and bruises**
- **bump map for nice wrinkle**
- **noise() and spline() and mix() are you friends**
- **fast rendering helps tremendously**



I went back to the wrinkled bump-mapping, adjusted a few numbers and created the final picture. The final mango shader is listed in the first appendix.

Here are few items to keep in mind about the process:

- `noise()` and `spline()` and `mix()` are your friends
- use `plastic.sl` and `show_st.sl` to check geometry and lighting
- fast rendering of lots of test images helps tremendously

As a minor point, I keep open two text windows and a bunch of image windows on the workstation screen. One text window is for editing; when you get rolling you are able to edit the next shader change as the last shader is being rendered. The other text window is merely for initiating the shader compiler and renderer. The rest of the screen is commonly littered with versions of the rendered picture.

### **Consider the banana**

(The lecture will include a quick discussion of a similar process for developing a banana shader.) The final shader is listed in the second appendix.

### **Conclusions about fruit**

We see a number of similarities between bananas and mangoes. It seems that nature has a surprisingly small repertoire of tricks that it uses to differentiate fruit! Get the basic color right. Use the noise function and some high school math to create smoothly varying functions across the surface.

Proceed one layer at a time, and create a small number of adjustable parameters to allow for tuning the images. Use a label parameter to create largely similar functions for different pieces of fruit.

### Random walks through n-D space

I started by insisting on Divide-and-Conquer as an approach to shader writing. The whole point of this strategy is to tease apart the different layers of surface characteristics and set up adjustable variables for tuning each layer independently. This is the optimal strategy.

#### Random Walks Through n-D Space

- **Avoid the random walk with Divide & Conquer**
- **But events may conspire against you**
- **Guidelines**

Make sure lighting is right and monitor is good

Wedge it (sparsely sample the n-D space)

Start with most significant parameter



There will be times, however, when the strategy breaks down. We have had occasions where the shader writer can make an object brighter or bumpier or browner, but the Art Director wants it “better”. You will find yourself a few hours from a deadline and without a clue. Here are some quick thoughts to keep in mind when you find yourself in such a predicament.

Make sure once again that the lighting for your shader is similar to the lighting used for the entire scene. For the same reasons, use a monitor that you trust. All too often, I have attempted to optimize a shader for lighting and monitor settings other than those to be used in the real scene.

Determine the most significant feature of the surface and start by getting that right. Usually, this is color variation, but there will be times when it is really the quality of the highlight or reflectivity which turns out to be the most important feature.

When all else fails, you will find yourself with a shader with  $n$  adjustable parameters and no idea how to set them. You are now wandering around in an  $n$ -dimensional space, a space skewed because of interdependencies of some of the dimensions. I suggest that you take a scattershot approach. Sample this  $n$ -D space very sparsely by choosing random values for each variable. Compute a picture for each selection and take a look at the results. Visually interpolate the images and average together one final setting for each the variables.

## Fast shaders

### Shaders That Execute Quickly

- **There is no magic; `sqrt()` and `noise()` take time**
- **Avoid calling noise more than a few times**
- **Avoid large-scale hit-testing in the shader**



There is no magic here. If you include lots of floating point computations in your shaders, your rendering will run more slowly. Certain functions take longer to compute than others on your computer. Square roots might be optimized on some machines, but you will find that matrix inversions almost always take time. I find the `noise` function indispensable, but it also takes some time to compute. I try to limit myself to four well chosen calls to the `noise` function per shader.

The efficiency consideration which I face most often is *hit* testing. The surface pattern is generated implicitly rather than explicitly. A shader answers the renderer's question: What color is the surface at point P? The easiest shaders to compute are those which create an analytic function to describe the surface. When the renderer asks for the color at a point, the shader samples the function.

Explicit descriptions of surfaces are much harder to deal with. Consider a PostScript file, an explicit list of drawing primitives, as a surface description. The only way to compute the color at point P is to test whether each and every drawn primitive hits the point P. If the list is long, the rendering will take forever. The only reasonable way I know to handle a PostScript surface is to create a texture map at a sufficiently high resolution and map it on.

## Hit-Testing in the Shader

---

- **Surface pattern is generated implicitly**
- **Shader's job: What color is surface at point P?**
- **Can afford to create function and sample it**
- **Cannot afford to hit-test entire PostScript file**
- **Basketball example**



So this is the *hit testing* problem: What is the most efficient scheme for computing the color at point P when you have a list of primitives which might hit point P? As you can imagine, if the list is long or if the primitives are unwieldy, this process can involve quite a bit of computation. *Avoid this problem at every opportunity!* Rick will talk later about a polka dot shader which faces up to the problem; here is an example of one which avoids it.

A basketball has little circular nibs covering its surface. The circles are not in any regular pattern; they are densely packed but generally random. After much thought about clever ways to decide whether a point on the surface was inside one of the nibs, I gave up and used a texture map. I felt that I had a much better chance of producing a pleasing distribution of nibs by using a C program to create a texture than with a shader. In fact, I used a dart throwing algorithm suggested by Don Mitchell in a SIGGRAPH 1991 paper to space the nibs. The final basketball shader merely wraps the texture map around the patches making up the sphere.

### Concluding comments about our axioms

So what have we learned from all this?

- Make sure the input geometry is adequate.
- Know what you're trying to compute.
- Steal code from previous shaders.
- Divide and Conquer.

*But these axioms are no different from those for any programming effort!* Edit, compile, run, debug. Just be thankful it's graphics and not accounting. Laugh at the buggy intermediate results and enjoy the final pictures.

**Appendix 1: Mango Shader**

```

surface
mango(
    float label = 0.0;
    float Ka = 1.0;
    float Kd = .6;
    float Ks = 0.1;
    float roughness = .1;)
{
    point Nf,V;
    color cs ;
    float darken;
    varying float i;
    color spotcolor,markcolor;
    point turb,p2;
    point newP;

    color whitey = color (1.0,0.70,0.1);
    color yellow = color (0.9,0.50,0.0);
    color yelora = color (0.9,0.40,0.0);
    color orange = color (0.9,0.30,0.0);
    color redora = color (0.9,0.20,0.0);
    color red = color (1.0,0.10,0.0);
    color green = color (0.25,0.50,0.0);
    color greora = color (0.47,0.43,0.0);
    color oragre = color (0.69,0.37,0.0);
    color brown = color (0.1,0.05,0.0);
    color lbrown = color (0.5,0.25,0.05);

    /* Nf = faceforward(normalize(N),I);*/

    setxcomp(p2,sin(2*PI*s));
    setycomp(p2,2*t);
    setzcomp(p2,cos(2*PI*s));
#define BUMP_AMPLITUDE (1/40)
#define BUMP_FREQUENCY (25)
    turb = noise(BUMP_FREQUENCY * p2)*BUMP_AMPLITUDE ;
    newP = calculatenormal(P + turb * normalize(N));
    Nf = faceforward(normalize(newP), I);

    V = -normalize(I);

    /* Now let's get the basic color of the mango. */
#define SFACTOR .5
#define TFACTOR 2

```

```

i = float noise(SFACTOR*sin(2*PI*s)+PI+
               label,TFACTOR*t+1.414+label);
cs = spline(i,green,green,greora,
            oragre,redora,red,red );
spotcolor=spline(i,green,yellow,whitey,
                yellow,whitey,yellow,yelora);
markcolor=spline(i,yellow,lbrown,brown ,
                brown,brown,lbrown,yellow);

/*
Now let's look at the spots on the mango by looking up a noise
value.
We will be setting the darkening coefficient "darken".
*/
    darken = float noise(150*(s+label),150*(t+label));
    darken = pow(darken,2);
    cs = mix(cs,spotcolor,darken);
    if (darken > .33)
        cs = mix(cs,markcolor,(darken-.33)*1.5);

/*
Now let's put some bruises on the mango by looking up a noise
value. We will be setting the darkening coefficient "darken".
*/
    darken = float noise(30*(s+label),30*(t+label));
    darken = pow(darken,5); /* looks ok between 4 and 6 */
    darken = .5+.5*sin(PI*(darken-1/2)); /* to sharpen the func-
tion */
    cs = mix(cs,markcolor,darken);

    Oi = 1.0;
    Ci = cs * (Ka*ambient() + Kd*diffuse(Nf))
        + Ks*specular(Nf,V,roughness);
}

```

## Appendix 2: Banana Shader

```

surface
banana(
    float label = 0.0; /* for each banana, choose random in [0,1) */
    float spotted = 0.5; /* 0 for perfect yellow; 1 for bruised */
    float lined = 0.7; /* 0 for perfect yellow; 1 for lined */
    float Ksmix = 0.8; /* 0 for white specular color; 1 for yellow
    */
    float Ka = 1.0;
    float Kd = .7;

```

```

    float Ks = 0.2;
    float roughness = .1;)
{
    float Scale,lineScale;
    float level,level2,level3;
    float angle,dangle;
    point Nf,V;
    point Nfd,Nfs;
    point p1;
    float x,y,z;
    color cs ;
    float labels;
    float darken;
    varying point turb;
    color specularcolor;
    point p2,newP;
    float ks;

    color yellow = color (1.0,0.55,.0);
    color white = color (1.0,1.0,1.0);
    color green = color (0.3,0.25,.0);
    color grelow = color (0.50,0.35,.0);
    color lbrown = color (.5,.25,.0);
    color dbrown = color (.07,.05,.025);

    color bruiseicolor = color (0.2,0.05,.025);

    Scale = 8-5*spotted;
    lineScale = 8-7*lined;
    ks = Ks;

    setxcomp(p2,sin(2*PI*s));
    setycomp(p2,3*t);
    setzcomp(p2,cos(2*PI*s));
    Nf = faceforward(normalize(N),I);
    Nfd = Nf;

    /* We'll use a bump map just to disperse the specular highlight */
    #define BUMP_AMPLITUDE (1/40)
    #define BUMP_FREQUENCY (25)
    turb = noise(BUMP_FREQUENCY * p2)*BUMP_AMPLITUDE ;
    newP = calculatenormal(P + turb * normalize(N));
    Nfs = faceforward(normalize(newP), I);

    V = -normalize(I);

```



```

/*
Now let's get the basic color of the banana, by looking primarily
at t. We will also affect the specular coefficient.
*/

    labels = s + label;
    level = t+.02*sin(2*PI*labels) +.01*cos(2*PI*labels)+
        .03*noise(s);
#define L1 .13
#define L2 .16
#define L3 .45
#define L4 .70
#define L5 .85
    if (level <= L1) {
        cs = dbrown;
        ks = 0;
    } else
    if (level <= L2) {
        cs = mix(dbrown, yellow, (level-L1)/(L2-L1));
        ks *= (level-L1)/(L2-L1);
    } else
    if (level <= L3) cs = yellow;
    else {
    if (level <= L4) {
        cs = mix(yellow, green, (level-L3)/(L4-L3));
        ks *= (level-L4)/(L3-L4);
    } else {
        cs = spline((level-L4)/(1-L4),
            grelow,green ,green ,green ,grelow,
            dbrown,lbrown,lbrown,lbrown,lbrown);
        ks = 0;
    }
    }

/*
Now let's look at the bruises on the banana by looking up a noise
value. We will be setting the darkening coefficient "darken".
There is one calculation for the bruises along the seams; there
is another for the bumps on the side.
*/
#define AA .2
    angle = mod(s, AA);

#define B0 .015
#define B1 .025

```

```

#define B4 .18
#define B5 .19

    if ((level < L5) && (level > L1)) {
#define SF 1
#define tF 5
        dangle = angle-AA/2;
        if (dangle < 0) dangle += AA;
        if (angle <= B0)
            darken = float noise(SF*sin(2*PI*(s+.1234))+
                                SF*dangle/AA,tF*t);
        else
            if (angle <= B1)
                darken = float noise(SF*sin(2*PI*(s+.1234))+
                                    SF*dangle/AA,tF*t)*(B1-angle)/(B1-B0);
            else
                if (angle >= B5)
                    darken = float noise(SF*sin(2*PI*(s+.1234))+
                                        SF*dangle/AA,tF*t);
                else
                    if (angle >= B4)
                        darken = float noise(SF*sin(2*PI*(s+.1234))+
                                            SF*dangle/AA,tF*t)*(angle-B4)/(B5-B4);
                    else
                        darken = 0;

            if (darken > 0) {
                darken = sin((PI/2)*darken);
                darken = darken*lineScale;
                darken = darken-lineScale+1;
                if (darken < 0) darken = 0;
                cs = mix(cs,bruiseColor,darken);
            }
    }

#define A0 .005
#define A1 .08
#define A2 .09
#define A3 .11
#define A4 .12
#define A5 .185

#define SF 10
#define TF 50
    if (angle <= A0)
        darken = float noise(SF*(s+angle/AA),TF*t)*(angle/A0);
    else
        if (angle <= A1)

```

```

        darken = float noise(SF*(s+angle/AA),TF*t);
    else
    if (angle <= A2)
        darken = float noise(SF*(s+angle/AA),TF*t)*(A2-angle)/
            (A2-A1);
    else
    if (angle >= A5)
        darken = float noise(SF*(s+angle/AA),TF*t)*(AA-angle)/
            (AA-A5);
    else
    if (angle >= A4)
        darken = float noise(SF*(s+angle/AA),TF*t);
    else
    if (angle >= A3)
        darken = float noise(SF*(s+angle/AA),TF*t)*(angle-A3)/
            (A4-A3);
    else
        darken = 0;

/*
Now we have the basic darkening coefficient.
The following is a little math to scale the coefficient
appropriately.
*/
    if (darken > 0) {
        darken = sin((PI/2)*darken);
        darken = darken*darken;
        darken = darken*Scale;
        darken = darken-Scale+1;
        if (darken < 0) darken = 0;
        cs = mix(cs,bruise,color,darken);
    }
}

/*
One of our parameters is the extent to which the banana reflects
specular highlights as white or yellow.
*/
    specularcolor = mix(white,yellow,Ksmix);

    Oi = 1.0;
    Ci = cs * (Ka*ambient() + Kd*diffuse(Nfd))
        + ks*specularcolor*specular(Nfs,V,roughness);
}

```

## Lights and Shadows

Eliot Smyrl

### Basic Light Sources

Light source shaders calculate the amount of light that leaves a light source and arrives at the surface of an object. The graphics state provides the information about which point is of interest, etc. and the shader then calculates the color of the light ray  $C_l$ .

#### Light Source Shaders

- **Calculate amount of light leaving light source and arriving at surface**
- **Input: graphics state and  $L$**
- **Output:  $C_l$**

#### Light Graphics State Variables

- **$L$     Outgoing light ray direction**
- **$P$     Light position**
- **$P_s$    Illuminated surface position**
- **$E$     Position of eye**

One important thing to note is that the vector  $L$  points out of the light toward the surface when programming a light shader. When programming a surface shader,  $L$  points out of the surface toward the light.

A light shader controls where its light goes by using one of two cone-oriented control flow statements. The most common of these is `illuminate`.

## Illuminate

- **Light out in all directions from source position**
- **Asks renderer for points that need light**
- **For each such point, set  $L$  from light to point**
- **Execute code block**

r

The `illuminate` block restricts itself by specifying a cone within which it will calculate illumination. Points outside of the cone will always be unilluminated.

## Illuminate Cones

```
illuminate ( position ) {
    /* 360 degree omnidirectional light */
}

illuminate (position, axis, angle ) {
    /* standard light cone */
}
```

r

## pointlight.sl

```
light pointlight(
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
{
    illuminate( from )
        Cl = intensity * lightcolor / L.L;
}
```



Here is the standard “pointlight” local light source. Light goes out in all directions from the center point `from`, and its intensity falls off with the square of the distance from the light source.

The other statement controlling illumination is `solar`. The difference is that light in a `solar` statement doesn’t come from any particular position.

## Solar

- **Light in every direction from infinity**
- **Asks renderer for points that need light**
- **For each such point, “integrates” over light directions**



Also like the `illuminate` block, the `solar` block can be specified with a cone that restricts the directions in which light rays will travel. The most common use is to specify a 0-angle cone, which means all rays are travelling in exactly the specified direction.

## Solar Cones

```
solar ( axis, angle ) {
    /* light only comes from these directions */
}
solar () {
    /* light from every direction */
    /* (e.g., an illumination map) */
}
```

r

Here is the standard “distantlight” light source. Light travels in parallel rays along the specified axis, but is not attenuated by any distance.

## distantlight.sl

```
light distantlight(
    float intensity=1 ;
    color lightcolor=1 ;
    point from = point "shader" (0,0,0) ;
    point to = point "shader" (0,0,1) ; )
{
    solar( to - from, 0.0 )
    Cl = intensity * lightcolor;
}
```

r

And here’s a more complex light, a spotlight. I’ve just included the body of the shader; if you want to see the parameter definitions you can look in the *RenderMan Interface Specification* or the *RenderMan Companion*.

**spotlight.sl**

```

float atten, cosangle;
uniform point A = (to - from) /
                  length(to - from);
uniform float cosoutside= cos(coneangle),
              cosinside = cos(coneangle-
                              conedeltaangle);

illuminate( from, A, coneangle ) {
    cosangle = L.A / length(L);
    atten = pow(cosangle, beamdistribution) /
            (L.L);
    atten *= smoothstep( cosoutside, cosinside,
                        cosangle );
    Cl = atten * intensity * lightcolor;
}

```

r

For every illuminated point, the cosine of the angle from the central axis is computed. A cosine dropoff is used to make the center of the beam brighter than the periphery. Between the angles of `coneangle-deltaangle` and `coneangle`, the light is attenuated quickly to zero. This gives the appearance of a soft penumbra around the edges of the spotlight. There is also a one-over-r-squared distance falloff.

**Shadows From Shadow Maps**

Some rendering algorithms, such as ray-tracing and radiosity, attempt to calculate global illumination effects, and therefore provide shadows (almost) automatically. However, RenderMan renderers that do not simulate global illumination can still produce shadows, often with less cost and more flexibility than global illumination renderers.

The primary mechanism for rendering shadows without global illumination is the *shadow map*.



## Shadow Maps

---

- **Depth value stored in texture pixels**
- **Z-buffer computed from light source**
- **Camera at point in scene**
- **Value in map is closest surface**



## Making a Shadow Map

---

- **Render a zfile from light position**
- **RiMakeShadow() turns depth image into shadow map**
- **txmake utility does the same**
- **Access from light shader with shadow() function**



One thing to consider is that if you plan to use the shadow map in a light source at infinity (such as distantlight), you should use an orthographic projection to render the zfile. If you plan to use the map in a light source such as a spotlight, use a perspective projection.

This is how you would call the shadow map in a distantlight.

### shadowdistant.sl

```
solar( to - from, 0.0 ) {
    Cl = intensity * lightcolor;
    if (shadowname != "") {
        Cl *= 1 - shadow(shadowname,Ps);
    }
}
```



At each surface point, if the value stored in the shadow map is closer to the light than the surface, the point is in the shadow of that light.

### Camera Transformation

- **Transformation to image camera position stored in depth image**
- **Used in shadow() function**
- **Flexibility**
  - other lights can use map
  - camera does not have to be at light position
  - can even call shadow() from surface shader



### **Shadows From Alpha Images**

A shadow map is accessed at every shaded point inside the illuminate cone of the appropriate light source, and this slows down rendering somewhat even in areas which may not need any shadows. At the cost of some realism, you may be able to speed up shadow rendering in some circumstances using an alpha image instead of a shadow map.

Imagine that you have a bunch of characters on a simple floor plane. You can cast shadows of the characters onto the floor using just an alpha image.

### Making Alpha Shadows

- **Render image (without floor) from light source**
- **Either:**
  - use alpha channel (1 where light hits objects), or
  - render all “constant”, color [1 1 1], use RGB
- **Make single-channel texture map**



### Change the Surface Shader

- **Change coordinates to light’s “NDC”**
  - transform to light position and orientation
  - scale image range to [0–1] horizontal and vertical
- **Call texture map**



### floor.sl

```

surface floor( float Ka=1, Kd=1 )
{
    point Nf;
    float tx,ty;
    float shade;
    /* other stuff needed for shadow.i */
    Nf = faceforward(normalize(N),I);
#include "shadow.i"
    shade = 1 - texture("shadow.tex",tx,ty);
    Oi = Os;
    Ci = Os * Cs * ( Ka*ambient()
        + shade*Kd*diffuse(Nf) ) ;
}

```



This shader is basically *matte* with some extra stuff. The file `shadow.i` contains code to do the coordinate transformation, then the next line uses the new coordinates to index the texture map. Anywhere the texture map is 1 there is a shadow, so the shader darkens its result appropriately.

The actual code to do the coordinate transformation can get pretty messy, because you need to use more than just the position of the light source. You also need to use the correct orientation and field-of-view of the camera used for the texture. We've set our animation system up to generate this code automatically, so we don't have to worry about it. Anywhere the texture map is 1 there is a shadow, so the shader darkens its result appropriately.

Here's an example of how `shadow.i` looks. Don't bother to try to understand it; this is just to give you an idea of the kind of thing you can do by making shader-generator programs.

```

tI = transform("world",I);
tx = (0.033588,-0.0187386,-2.46976e-08) . tI + -0.600342;
ty = (0.00697126,0.0124956,0.0357009) . tI + -0.0340346;
tw = (4.52237e-05,8.10614e-05,-3.72029e-05) . tI + 0.00648925;
tx = ((tx / tw)+1)/2;
ty = (((-ty) / tw)+1)/2;

```

This procedure can often be much faster than rendering with a shadow map. However, it has some drawbacks.

## Drawbacks

- **Must change each shadowed surface shader**
- **Hard to hand-generate**
- **Objects can't shadow themselves**



## **Shadows From Painted or Scanned Textures**

### Shadows of fake objects

- **Might want shadows of objects that don't exist in scene**
- **Can't generate a shadow or alpha map**  
so
- **Paint or scan it!**



When you have a shadow image, modify the light source shader to access it using the camera image coordinates, similar to using an alpha image.

### distprojlight.sl

```
solar( to - from, 0.0 ) {  
    /* calculate tx and ty */  
    #include "shadowproj.i"  
  
    shade = 1 - texture(texname,tx,ty);  
    Cl = shade * intensity * lightcolor;  
}
```



This will project the shadow onto your scene like a slide projector. If you use a separate painted shadow for each frame of an animation, you will get animated shadows.

This technique is very effective in producing various effects, from menacing shadowy monsters to the canopy of a forest.

### **Shadows From Procedural Textures**

#### Procedural Shadows

- **Similar to texture generation in surface shaders**
- **Don't need texture or shadow maps**
- **Much faster**



Here are some examples. The first shader just generates a square lighted region, simulating barndoor stage lights.

```
light barndoor ( float left = -1; right = 1;
                 float bottom = -1, top = 1;
                 float edge = 0.1;
                 float intensity = 1; color lightcolor = 1;)
{
    float tau, x, y;
    point P2, L2;
    P2 = transform("shader", P);
    axis = point "shader" (0,0,0) - point "shader" (0,0,1);
    illuminate(P, axis, PI/2) {
        L2 = transform("shader", Ps) - P2;
        tau = (1 - zcomp(P2)) / zcomp(L2);
        x = xcomp(P2) + tau * xcomp(L2);
        y = ycomp(P2) + tau * ycomp(L2);

        C1 = intensity * lightcolor *
            smoothstep(left, left+edge, x) *
            (1 - smoothstep(right-edge, right, x)) *
            smoothstep(bottom, bottom+edge, y) *
            (1 - smoothstep(top-edge, top, y));
    }
}
```

The first lines inside the `illuminate` statement calculate the intersection of the illumination ray with a plane in front of the light source (just like projecting the scene onto a camera's film plane). The long final statement evaluates the light intensity to full on when the point is within the box, full off if it is outside the box, and a soft penumbra when near the edge.

You can get the effect of a light through a paned window by just adding crossbars to the same shader.

**window.sl**

```

Cl = smoothstep(left, left+edge, x) *
    (1 - smoothstep(right-edge, right, x)) *
    smoothstep(bottom, bottom+edge, y) *
    (1 - smoothstep(top-edge, top, y));
Cl *= intensity * lightcolor *
    (1 -
        smoothstep(-edge, 0, x) *
        (1 - smoothstep(0, edge, x))
    ) * (1 -
        smoothstep(-edge, 0, y) *
        (1 - smoothstep(0, edge, y))
    );

```



Here the first line does the box evaluation, as in the previous shader. The second line superimposes crossbars on the result of the first calculation.

If you want venetian blinds instead, just add a bunch of horizontal bars to the barn doors.

Procedural techniques such as these can often be used for special-purpose lights, when you know exactly what the geometry will look like.

## Procedural Light Sources

Now that you've generated lots of shadows, you might want to change the characteristics of the light itself. There are nearly as many ways to do this as there are to write surface shaders, but here are a couple of examples that we've used.

This is a very simple shader that produces a nice fiery effect.



**firelight.sl**

```

light
    firelight(
        float intensity = 1;
        color lightcolor = 1;
        point from = point "shader" (0,0,0);
        float time = 0;
        float freq = 1; )
    {
        uniform float flicker = noise(freq*time +
                                         0.25);

        illuminate( from )
            Cl = intensity * lightcolor / L.L;
    }

```

r

The noise function has a 0.25 offset because noise with an integer parameter always returns 0.5.

You can imagine lots of other ways to generate fiery or ripply effects. The classic stage fire cylinder, for example, could just be done with a rotating texture map, or noise() “streamers” moving up.

The next example is more complex.

```

light
mirrorball(
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
    point up = point "shader" (0,0,1);
    point front = point "shader" (0,-1,0);
    float numspots = 36;
    float spotsize = .25;)
{
    float sp, tp, ns;
    point upvec = normalize(up - from);
    point frontvec = normalize(front - from);
    point pp, stcenter;
    float spotrad = numspots/36*spotsize;
    float dfc, atten, lr;
    float jittamp = 1;

    illuminate( from ) {
        /* get spherical coordinates sp and tp */
        tp = acos(normalize(L) . upvec)*numspots/PI;
    }
}

```

```

    ns = floor(sin(floor(tp)/numspots*PI)*numspots+0.5);
    pp = normalize(L - upvec*(upvec . L));
    lr = (pp ^ frontvec) . upvec;
    if (lr <= 0) lr = -1; else lr = 1;
    sp = lr * ns * acos(pp . frontvec)/PI;

    /* jitter spot centers */
    stcenter = .5+jittamp *
               (point
                noise(floor(sp)+234.5,floor(tp)+501.5)-.5);

    /* attenuate by distance from spot center */
    tp = mod(tp, 1)-xcomp(stcenter);
    sp = mod(sp, 1)-ycomp(stcenter);
    dfc = sqrt(sp * sp + tp * tp);
    atten = 1-smoothstep(0, spotrad, dfc);

    Cl = intensity * lightcolor * atten;
}
}

```

The first section of code inside the `illuminate` statement calculates spherical coordinates based on the three direction vectors determined by the points passed in to the shader. The next section uses `noise()` to jitter light circles on a grid established on these spherical coordinates. The light is animated by changing the directional vectors.

## Miscellaneous Light Tricks

Just as with surface shaders, there are a lot of tricks that you can pull with light shaders. Here are a few examples.

## Colored Shadows

Colored shadows can give you some really nice effects, and they're very easy to do.

## Colored shadows

---

- **Change light color when in shadow**
- **Multiple colors use multiple shadow maps**

r

## dist2col.sl

---

```
solar( to - from, 0.0 ) {
    Cl = intensity * lightcolor;

    if (shad1name != "")
        Cl = mix(Cl,shad1color,
            shadow(shad1name,Ps));
    if (shad2name != "")
        Cl = mix(Cl,shad2color,
            shadow(shad2name,Ps));
}
```

r

You can do the same thing with alpha maps or texture shadows: just access two textures in the shader instead of one.

### Shdw Surface

If you want to generate only the shadows on an object (perhaps to composite over live action), you could use the following shader.

## shdw.sl

```
surface shdw(string mapname = "")
{
    float shade=0;

    if (mapname != "")
        shade = shadow(mapname, P);

    Oi = 1;
    Ci = shade;
}
```



Notice that, as mentioned earlier, this is a surface shader that uses the `shadow` function.

## shdw

- **Object white where shadowed, black elsewhere**
- **Alpha channel like normal (object opaque)**  
don't want to see shadows on "back" of object
- **Use rgb as shadow "mattes" to modify other images**



## Textured Lights

You might want to modify the color of a light with a texture map.

### slideprojector.sl

```

P2 = transform("shader", P);
axis = point "shader" (0,0,0) -
      point "shader" (0,0,1);
illuminate(P, axis, PI/2) {
    L2 = transform("shader", Ps) - P2;
    tau = (1 - zcomp(P2)) / zcomp(L2);
    x = xcomp(P2) + tau * xcomp(L2);
    y = ycomp(P2) + tau * ycomp(L2);
    x = (x+1)/2;
    y = (y+1)/2;
    C1 = intensity * lightcolor *
        color texture(texname,1-x,y);
    /* use 1-x to make 0 left of 1 on surface
}

```

This shader acts like a slide projector, shining the texture map across the scene. The plane intersection is calculated the same way as in the barn-doors shader we looked at earlier, and then the coordinates on this plane are used for the texture map.

## Aliasing in Shaders

Darwyn Peachey

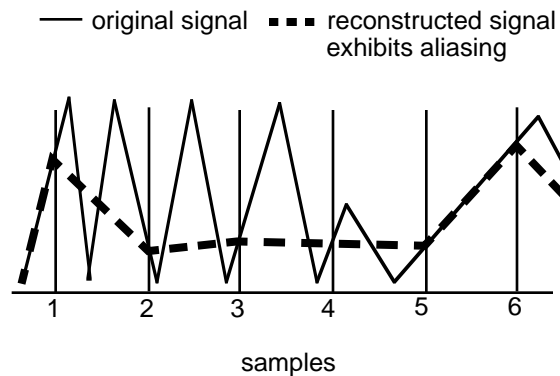
As if writing realistic shaders isn't challenging enough, we have to deal with aliasing artifacts as well. It would be nice if the RenderMan renderer could solve shader aliasing problems as well as it solves geometric aliasing (jaggy edges on geometry). As we will see, however, this isn't the case, and we have to do quite a lot of work to suppress aliasing artifacts in our shaders.

### What is aliasing?

- **classic examples**
- **jaggy edge or line**
  - checkerboard perspective
  - signal frequencies too high for sampling rate
- **high freq energy reconstructed as low freq artifact**

r.

### Aliasing



r.

Aliasing occurs when a regular sampling process is used to sample and reconstruct a signal whose highest frequencies are greater than one-half of the sampling rate. The high frequency energy shows up as a low-frequency “alias” in the reconstructed signal.

### How to Antialias

---

- **low-pass filtering of signal before sampling**
- **stochastic sampling**
  - high freq show up as noise
  - LOA (less objectionable artifact)



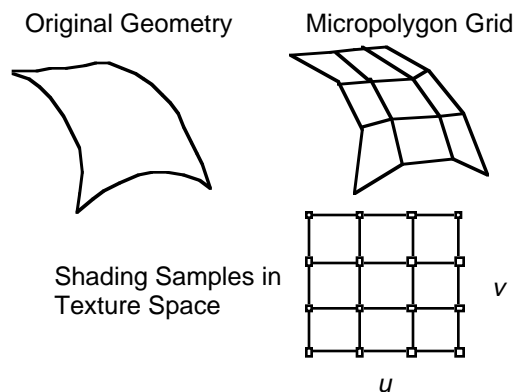
We can prevent aliasing in two ways. The best way is to filter out the high frequencies before sampling. Since the high frequencies can't be reproduced properly, we're better off without them. An alternative is to use stochastic sampling of the signal. The high frequency energy is still present in this method, but it appears as noise in the reconstructed signal, and noise is generally less objectionable to the human eye than are low frequency aliases.

## Why Do Shaders Alias?

- **prman uses Reyes algorithm**
  - geometry converted to grid of micropolygons
  - shading done at vertices of micropolygons
  - point sampling of shading function on regular  $(u,v)$  grid
  - aliases if freq higher than half sampling rate



## The Reyes Algorithm

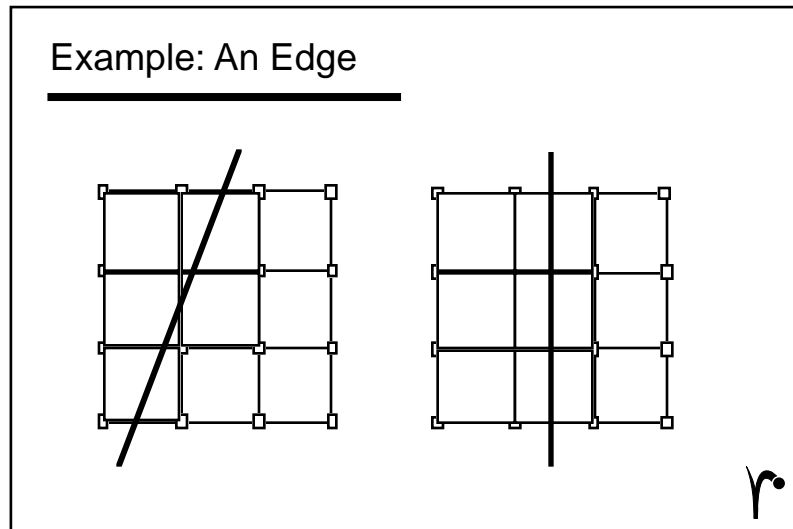


PhotoRealistic RenderMan performs antialiasing by stochastically sampling the scene geometry in micropolygon form, and filtering the results of the sampling process. This form of antialiasing is quite effective for geometric edges.

Unfortunately, there is another sampling process in PhotoRealistic RenderMan which is very prone to aliasing. A shader may be thought of as a signal-generating function defined on  $u, v$ . The function is evaluated at a regularly-spaced grid of shading sample points in  $u, v$  (the micropolygon vertices). The sampling rate is the reciprocal of the size of the micropolygons in the texture space where the function is defined. If the function has frequencies higher than one-half the shader sampling rate, the



reconstruction of the function from the shading samples will exhibit aliases. The reconstruction of the function is the flat-shaded or Gouraud-shaded micropolygon mesh which is resampled by the pixel sampling/filtering process. Once aliases have been introduced in the shader sampling process, they can never be removed by pixel sampling and filtering.



This example shows why a sharp edge in the shading function appears jaggy in the micropolygon mesh. Micropolygon colors are set to one of two distinct values based on which side of the edge the shading sample is on.

### **Why Doesn't Renderer Antialias?**

- **can't solve shader aliasing after shading**
- **stochastic shader sampling is hard in Reyes algorithm**
- **ray tracer could stochastically sample shader**
- **prefiltering is a cleaner solution**
- **automatic prefiltering of texture file is easy**
- **automatic prefiltering of procedural texture is hard**

There are two ways that the shader aliasing can be eliminated:

- prefilter the shading function to eliminate high frequency energy.
- stochastically sample the shading function so that high frequency energy is reproduced as noise rather than as aliases.

The stochastic sampling solution seems to be the easiest to apply in the renderer, since filtering arbitrary procedural shading functions requires programmer ingenuity. Unfortunately, regular shader sampling is closely tied to the method of geometric approximation in PhotoRealistic RenderMan and would be hard to change. Jittering the parametric positions of micropolygon vertices to eliminate regular shader sampling creates weirdly irregular micropolygons. This is itself an unacceptable artifact unless the micropolygons are smaller than a pixel. Since splits must occur along isoparametric lines, jittering only can be done at the interiors of grids; the remaining regularity along grid edges (needed to prevent cracking) might create odd sampling artifacts.

A RenderMan ray tracer using stochastic sampling is in a better position to apply this method of shader antialiasing. In fact, it would be hard *not* to do this. A ray tracer stochastically generates camera rays based on the pixel sampling process. These rays strike the geometric model at irregularly spaced locations in parameter space and in 3D space. Consequently the shading function is evaluated at a stochastic set of points. The same antialiasing method will automatically be applied to geometric edges and shading functions, and the same pixel filter will be used to combine the samples. But even if we had such a renderer, the best image quality would be obtained by prefiltering the shading functions to eliminate both aliases and the noise generated by stochastic sampling of high frequency signals.

So how can we write shaders that prefilter the shading function to reduce objectionable high frequencies?

### Writing Antialiased Shaders

- **limit frequency content of shader**
  - clamping
  - filtering
- **filter width from sampling rate**
- **simplest filter is box**



There are two cases: functions that are synthesized as a sum of components of various frequencies (spectral functions), and other functions. The “clamping” technique applies to the spectral functions.

For other functions, we have to apply analysis and ingenuity to figure out how to build filtering into the shader. For the purposes of this presentation, we'll tackle the easiest case, namely, applying a box filter to simple edges such as those in the brick shader.

## Clamping a Spectral Function

- **function consists of sum of components**
- **attenuate all components with frequency near or above one-half of sampling rate**
- **sudden amplitude change will pop or alias**



The term *clamping* comes from the SIGGRAPH '82 paper "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space," by Alan Norton, Alyn Rockwood, and Philip Skolmoski.

## Clamped Turbulence Function

### sudden clamping

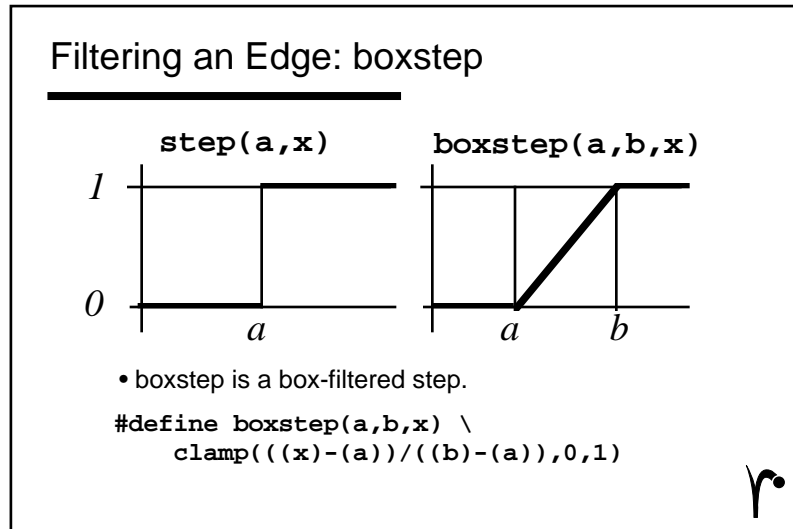
```
value = 0
cutoff = clamp(0.5 / swidth, 0, MAXFREQ);
for (f=MINFREQ; f < cutoff; f *= 2)
    value += abx(noise(f*s))/f;
```

### gradual clamping

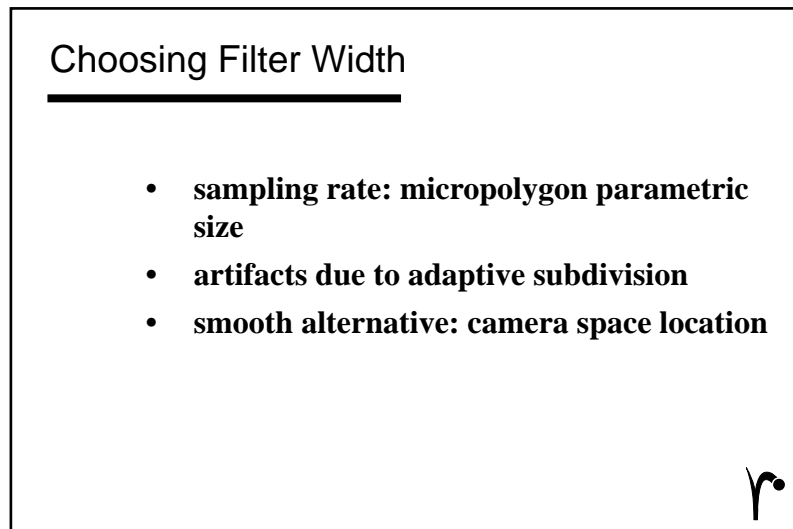
```
value = 0
cutoff = clamp(0.5 / swidth, 0, MAXFREQ);
for (f=MINFREQ; f < cutoff/2; f *= 2)
    value += abx(noise(f*s))/f;
value += 2*(cutoff-f)/cutoff *
    abs (snoise(f*s))/f;
```



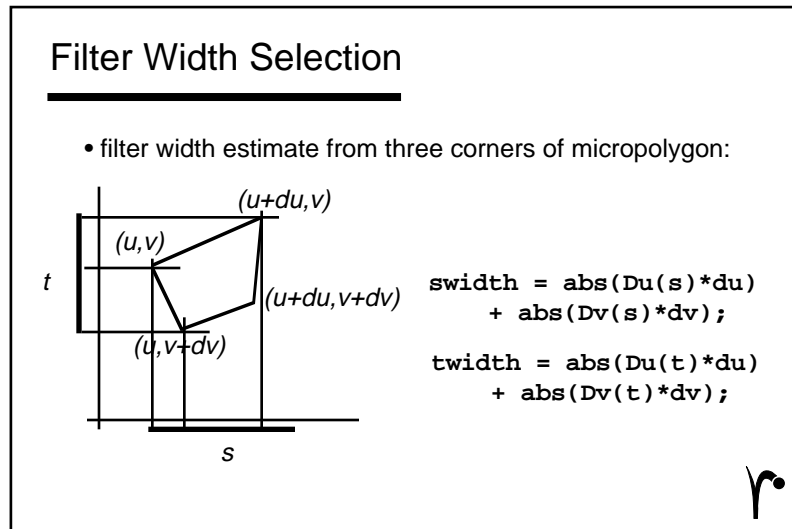
The “sudden clamping” version of the *turbulence* function comes from the SIGGRAPH ‘85 paper “An Image Synthesizer,” by Ken Perlin.



If the procedural texture function contains steps or conditionals of some kind, you will have to know how to prefilter an edge. The `boxstep` function (shown here as a preprocessor macro) is the result of the convolution of a box filter with a step edge. The width of the box filter is  $(b - a)$  and the slope of the ramp is  $1/\text{width}$ .

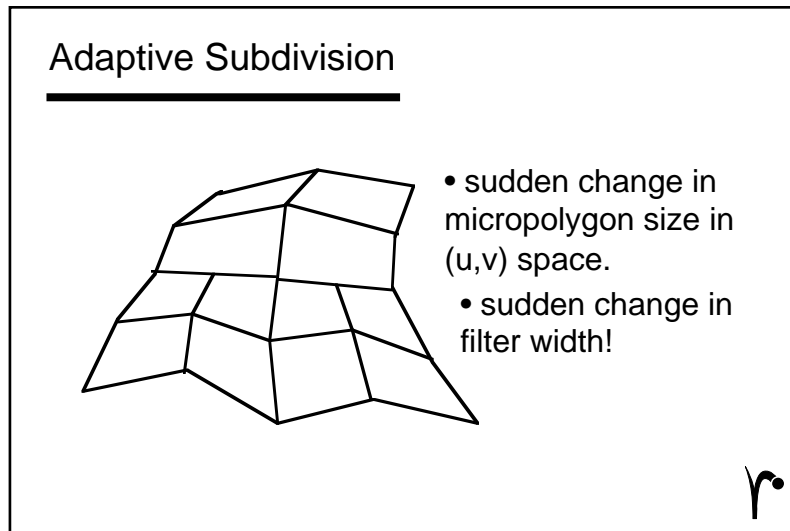


Usually PhotoRealistic RenderMan is operated in the `RiShadingInterpolation` "constant" mode. In this mode, shading calculations done at the corner  $(u, v)$  of a micropolygon are used to color the entire micropolygon. To box filter (area sample) the shading function, we want the filter box to exactly cover the micropolygon.



This trick for determining filter width works pretty well most of the time. Note that the filter will fit the micropolygon exactly if  $(s, t) = (u, v)$ .

An alternative filter width estimate is to use the cross product of the two edges of the micropolygon to compute the area of the micropolygon. This estimate will be wrong if the micropolygon is not a parallelogram, and it gives the same filter width (square root of the area) for the  $s$  and  $t$  directions, so the two directions must be filtered equally.

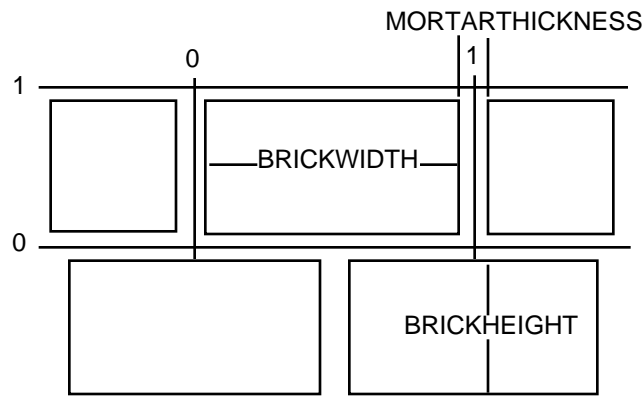


Using a filter width based on the parametric size of the micropolygon can cause problems. PhotoRealistic RenderMan uses adaptive subdivision to divide surfaces into an appropriate number of micropolygons so that each micropolygon is no larger than the limit specified by `RiShadingRate`. This adaptive subdivision leads to sharp changes in the size of micropolygons across a surface, as shown in the diagram. Tapered surfaces or surfaces in perspective can have large differences in screen space size of a given parametric step. As a result, more micropolygons are needed on the closer parts of the surface, and fewer on the more distant parts of the surface. The size of the steps  $du$  and  $dv$  across a micropolygon may differ by a factor of two or more between the two cases. The sharp change in the micropolygon parametric size leads to a sharp change in the filter size, which might be visible in the image. If the micropolygons are small enough (`RiShadingRate` small enough) the transition should be hidden, unless your filtering or clamping is sudden rather than gradual.

Note that the binary subdivision option in PhotoRealistic RenderMan, which is needed to eliminate patch cracking, forces the renderer to change micropolygon size by a factor of two and might increase the adaptive subdivision artifacts. Increasing the maximum grid size parameter will give the renderer more subdivision opportunities, and might reduce the artifacts.

A smooth estimate of filter width can be calculated by considering the distance of the surface from the camera (ie, camera space position  $\mathbb{P}$ ). Filter width varies as distance from the camera varies. Remember that you are trying to guess the parametric size of the micropolygons based on their distance from the camera, and a known shading rate, field-of-view, and resolution. Sophisticated estimates could make use of the surface normal to adjust for orientation, and could project a known set of camera space points into raster space to discover the projection and resolution.

### Example: Brick Shader



### Example: Brick Shader

```
ss = scoord / BMWIDTH
tt = tcoord / BMHEIGHT
if (mod(tt*0.5) > 0.5)
    ss += 0.5;          /* shift alternate rows */

tbrick = floor(tt); /* which brick? */
ss += 0.2 * (noise(tbrick+0.5) - 0.5);
sbrick = floor(ss); /* which brick? */
ss -= sbrick;
tt -= tbrick;
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MWF,tt) - step(1-MWF,tt);
Ct = mix(Cmortar, Cbrick, w*h);
```

This is just the brick shader example from my earlier presentation on texture generation.

### Example: Antialiased Brick Shader

---

```
w = step(MWF,ss) - step(1-MWF,ss);
```

...is replaced with...

```
if (swidth >=1)
    w = 1 - 2*MWF
else
    w = boxstep(MWF-swidth,MWF,ss)
      - boxstep(1-MWF-swidth,1-MWF,ss);
```



In order to antialias the edges of the brick/mortar transition, we can replace the `step`-based pulse with a similar construction based on the `boxstep`. If the filter is very wide, we simply supply the value `1 - 2*MWF` which is the fraction of the total width covered by the brick.

Unfortunately, since the brick function is periodic, the `boxstep` pulse function really doesn't give us quite the right answer; it handles the edges of a single brick in an infinite wall of mortar, without including the effect of the next or previous brick. The two are indistinguishable as long as the filter is no wider than twice `MWF` (or `MHF`). We've already handled the case of filters wider than one. To fix the periodic brick case, we add additional clamping to the `boxsteps` so that the amount of mortar cannot be overestimated.

### More Antialiased Brick Shader

---

```
w = clamp(boxstep(MWF-swidth,MWF,ss),
          max(1-MWF/swidth,0),1) -
      clamp(boxstep(1-MWF-swidth,MWF,ss),
          0,2*MWF/swidth);
```





Here is a listing of the complete brick shader with antialiasing:

```
#define BRICKWIDTH      0.25
#define BRICKHEIGHT     0.08
#define MORTARTHICKNESS 0.01

#define BMWIDTH          (BRICKWIDTH+MORTARTHICKNESS)
#define BMHEIGHT         (BRICKHEIGHT+MORTARTHICKNESS)
#define MWF              (MORTARTHICKNESS*0.5/BMWIDTH)
#define MHF              (MORTARTHICKNESS*0.5/BMHEIGHT)

surface
brick(
    uniform float Ka = 1;
    uniform float Kd = 1;
    uniform color Cbrick = color (0.5, 0.15, 0.14);
    uniform color Cmortar = color (0.5, 0.5, 0.5);
)
{
    color Ct;
    point NN;
    float ss, sbrick, swidth, scoord = s;
    float tt, tbrick, twidth, tcoord = t;
    float w, h;

    NN = normalize(faceforward(N, I));
    ss = scoord / BMWIDTH;
    tt = tcoord / BMHEIGHT;

    if (mod(tt*0.5,1) > 0.5)
        ss += 0.5;                                /* shift alternate rows */

    tbrick = floor(tt);                            /* which brick? */
    ss += 0.2 * (noise(tbrick+0.5) - 0.5);
    sbrick = floor(ss);                            /* which brick? */
    swidth = abs(Du(ss)*du) + abs(Dv(ss)*dv);
    twidth = abs(Du(tt)*du) + abs(Dv(tt)*dv);
    ss -= sbrick;
    tt -= tbrick;

    if (swidth >= 1)
        w = 1 - 2 * MWF;
    else
        w = clamp(boxstep(MWF-swidth,MWF,ss),
            max(1-MWF/swidth,0),1) -
```

```

        clamp(boxstep(1-MWF-swidth,1-MWF,ss),0,
        2*MWF/swidth);

    if (twidth >= 1)
        h = 1 - 2 * MHF;
    else
        h = clamp(boxstep(MHF-twidth,MHF,tt),
        max(1-MHF/twidth,0),1) -
        clamp(boxstep(1-MHF-twidth,1-MHF,tt),0,
        2*MHF/twidth);

    Ct = mix(Cmortar, Cbrick, w*h);

    /* "matte" reflection model */
    Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(NN));
}

```

# Antialiasing Techniques

Rick Sayre

## Introduction

With wistful gazes back to that simpler, more innocent time of `step` and `if`, we turn our weary minds towards the task of antialiasing. Rather than send you screaming for the security of texture maps, I hope to provide you with a framework for attacking aliasing shaders in a coherent manner. There are several tricks and techniques we can use, depending upon the shader at hand. Some are plug 'n play, some require a bit of head-scratching (or a good symbolic math program), but all are intended to give you ideas about what to try when the jaggy gets in your way.

These different techniques will all attempt to exploit certain characteristics of a shader to make the job more tractable. You may need to combine these methods, and it makes sense to do so. Consider a checkered floor with dirt smudges. The checkers lend themselves well to analytic integration, while the dirt may be more simply and tractably handled by adjusting its frequency content directly.

### Antialiasing Techniques

---

- **You have some measure of allowable feature size.**

$ds, dt, Du, Dv, \text{area}(), \text{etc.}$

- **Shader shouldn't produce patterns containing frequencies higher than this limit.**

Our basic goal is to limit shader frequencies based on some measure of allowable feature size. These will most commonly be based on estimates of how much a given feature-space changes over a micro-polygon.

## Useful Sample Rate Metrics

---

```

/* compute width of anti-aliasing filter */
#define MINFILTERWIDTY      1e-7
/* should be sqrt(MINFILTERWIDTH) */
#define MINDERIV            .0003
#define FILTERWIDTH(x)\
    (max(abs(Du(x)*du) + abs(Dv(x)*dv),\
         MINFILTERWIDTH))
#define FILTERWIDTH_POINT(P)\
    (max(sqrt(area(p)), MINFILTERWIDTH)

```

r

These macros provide a useful way to estimate feature-space changes for 1 and 3-dimensional feature spaces. The result can be used to fade out frequencies or adjust filter widths. We pick a minimum allowable filter width in the interest of avoiding numerical problems when the changes get particularly teeny.

## Texture Maps

### Texture Maps

---

- **Use them when you can**  
Filtering for free
- **Don't forget 1-D texture maps**

r


It is easy to become blinded by the power of algorithmic shaders. Don't let every problem look like a nail; there are many cases where texture maps will work just fine. The filtering comes for free.

A handy use of texture maps is to generate filtered versions of 1D functions. If the function is of finite domain and reasonable range, you can simply sample it to create a one-dimensional image, and make a texture map out of it. Your shader can then call `texture` with the desired domain mapped into `[0,1]` to get a low-pass filtered version of the function. Ramps, slats and so on can be easily generated this way. You do, of course, lose the ability to quickly and easily change any parameters of this expression. For that you must forge on, and actually antialias the expression in question.

## Direct Frequency Control

### Direct Frequency Control

- **Pattern function generated by additive synthesis.**  
Only include allowable frequencies.



There are many cases in which our pattern function is expressible directly in the frequency domain, in terms of component frequencies. In these cases, low-pass filtering becomes nice and simple; just don't include any unwanted frequencies.

## Octaves of Noise

### Octaves of Noise

- **Shader sums several octaves of noise together to create fractal features or “turbulence”.**

Don’t include octaves beyond the desired cutoff frequency.

When only one octave left, attenuate it as necessary.



Many shaders produce their patterns by combining octaves of noise to create fractal features or “turbulence”. To low-pass filter, we never include octaves above our cutoff frequency, and attenuate those near the cutoff.

### 1/f Noise

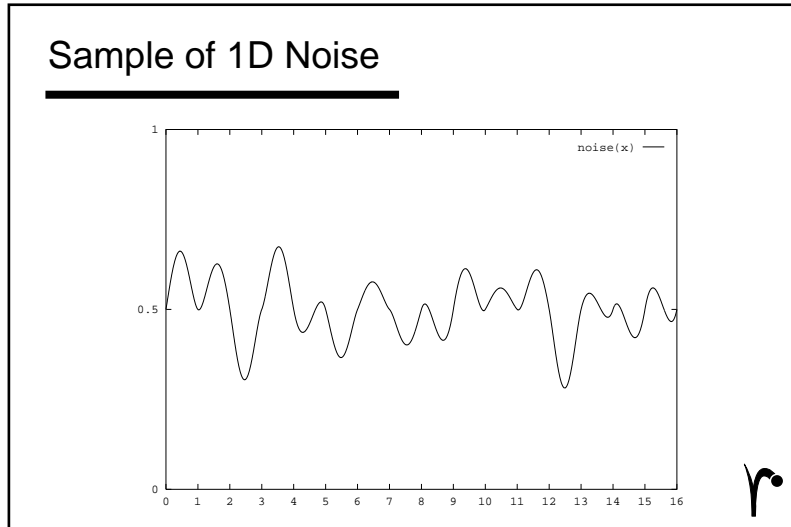
- **Generate approximation to 1/f noise, up to a maximum frequency of MAXFREQ.**

```
cutoff = clamp(1/(2*FILTERWIDTH_POINT(Q)), 0,
               MAXFREQ);
for (f = swirlfreq; f < cutoff/2; f *= 2)
    dd += swirl * noise(Q * f) / (f/swirlfreq);
if (f < cutoff)
    dd += swirl *
        (1 - smoothstep(cutoff/2, cutoff, f)) *
        noise(Q * f) / (f/swirlfreq);
```

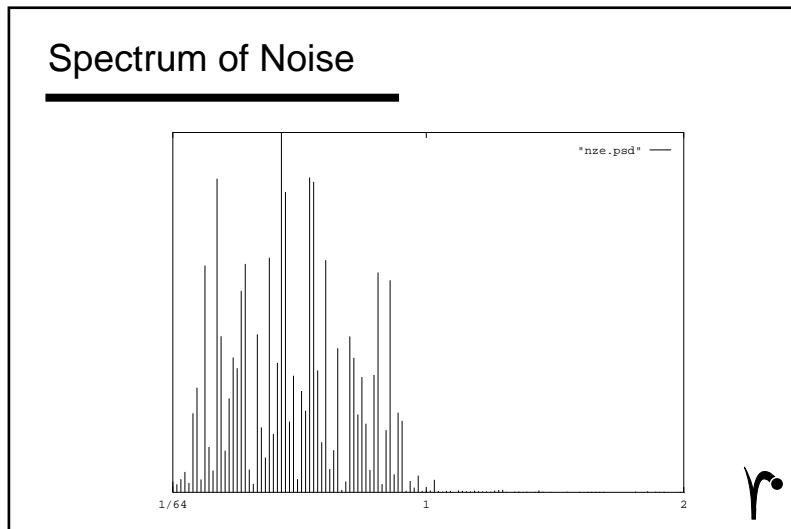


Here’s a shader fragment to do this for “1/f noise.” `swirl` controls the overall magnitude of the noise, while `swirlfreq` is the requested maximum frequency of the noise. `MAXFREQ` serves as an upper-limit to prevent us from wasting time computing eleventy-jillion octaves of noise when we

get really close. Note that the range of this function is  $[0, 2 * \text{swirlfreq}]$ , but because of clustering, the values will strongly tend to lie around  $.75 * \text{swirlfreq}$  to  $\text{swirlfreq}$ .



Actually looking at a sample of `noise` will confirm the unequal distribution of the output range. Notice that `noise` very rarely reaches its extrema.



We can see here that, although there is some leakage, `noise` is reasonably well band-limited. Its spectrum is a bit spiky, and it is definitely *not* white within its pass-band. It is clear, though, that it does not really have a predominant frequency.

There are some problems to beware of in that innocent-looking chunk of code. By attenuating the noise, we are not continuously low-pass filtering it; as the cut-off point of the filter is continuously lowered, rather than producing a signal with continuously decreasing maximum frequency, we are lowering the amplitude of a cluster of frequencies simultaneously. The severity of this approximation depends of course on the size of these clusters. This is an issue independent of the discrepancies between our desired synthesized function and our approximated additively synthesized function. These discrepancies are an inherent problem in this additive method of synthesis.

When there is only one octave left and we begin attenuating it, things get particularly dicey. At this point, there is nothing left but DC. Often we will notice that desirable low frequencies are being attenuated along with the higher frequencies.

We can alleviate these problems by using smaller frequency clusters of noise.

### Fourier Synthesis

#### Fourier Synthesis

- **Desired function constructed from sum of sines.**
- **Similar computation to Octaves of Noise**

Coefficients available analytically

Coefficients from FFT

*1-D texture map*

*spline() evaluated at knots*



We can use a similar technique for regular pattern synthesis. If we decompose our signal into its frequency components, then we can adjust the weighting of those components to achieve our low-pass filtering.

If the coefficients are available analytically, our task is straightforward. We can also take samples of arbitrary (periodic) functions, run them through an FFT program to determine the appropriate coefficients, and store them in a table. Since there are no arrays in the shading language, we need to be crafty. We can store coefficients in a 1D texture map, and evaluate the texture map at the sample points with no filtering. Bear in mind that we'll end up with only 8-bit resolution for the entire coefficient range. Alternatively, we can stuff all of the coefficients into a *spline*, and evaluate it at the knots to pull them back out.

With FFT-decomposed functions, however, we can no longer alter the parameters of the function without recomputing the FFT.



## Fourier Synthesis Stripes

```
ss = s*pulses;
dss = FILTERWIDTH(ss);
sum = 0;
k = floor(1/(2*dss)-.5);
for (i=0; i<k; i=i+1)
    sum = sum + sin(ss*2*PI*(2*i+1))/(2*i+1);
sum = sum * 4 / PI;
/* rescale and shift [-1,1] to [0,1] */
sum = (sum+1/2);
Oi = Os;
Ci = Oi * Cs * sum;
```



Here's a simple shader fragment to generate antialiased stripes. `sum` can of course be used to control other surface features. We keep adding sine waves, with the appropriate weighting, of higher and higher frequency until we reach the cutoff. Since we know that square waves contain only odd harmonics, we save time by never bothering with the even ones. In this particular case, we don't worry about attenuation of the last frequency. If we did, we would have none of the concerns we had in *Octaves of Noise*, since there would be only one frequency present.

## Fourier Synthesis - Reality

- **Works quite well, and can be used as a no-brainer.**  
Fourier synthesis shaders are easy to generate automatically
- **May be extremely slow**  
The closer you get, the longer it takes



This technique works quite well, and has the advantage that it can be used to automatically generate a large class of shaders. Given functions can be thrown into frequency space through appropriate analytic or discrete means, and tables or equations in template shaders filled in.

However, these shaders can be extremely slow. Since we are computing all frequencies from scratch, the sampling rate is telling us how much work we need to do, not how much we can throw away. If we didn't compute all the frequencies for the stripe, for example, we'd end up with ripples. Thus, the more magnified the pattern, the longer the shader is going to take. If you can low-pass filter the function any other way, you're probably better off doing it.

There *is* a better way, and we'll look at it later on.

### Global Mixing

#### Global Mixing

- **Some functions are just too difficult to analytically filter in reasonable time. We can compute the average (global) color and fade to it as we near the (approximate) Nyquist limit.**

Success limited by bandwidth of pattern function.

Often ok as long as Nyquist limit changes only as a function of time.



There will be times where we'll be really hard-pressed to come up with a "correct" method for low-pass filtering. This is particularly true for randomly-arrayed complex features seen from far away, such as polkadots or geometric wallpaper.

We can "fade" from the computed surface features to a pre-determined "average" feature. This is, of course, entirely bogus unless the function in question contains only one frequency. Rather than filtering, we are equally attenuating all frequencies, resulting in the loss of necessary lower frequencies. For animation which involves global changes in feature size, as in a patterned box receding away from the viewer, this approach is not terribly objectionable. However, when the projected feature size changes over an object, such as a plane in perspective, visible artifacts arise. These artifacts are made *much* more noticeable by the discontinuities in micropolygon size previously discussed.

The success of this hack is based on the frequency content of a shader; low bandwidth shaders survive comparatively better.

## Global Mixing - Template

- **featureWidth** is average size of feature
- **fMag** express micropolygon size in same space

```
mix(Ci, avgColor,
    smoothstep(featureWidth,
                featureWidth*2, fMag));
```



This dangerous chunk of code blends from a computed fancy color to a pre-computed “average” color.

## Feature-Space Filtering

### Feature-Space Filtering

$$g'(t) = \int_{-w/2}^{w/2} f(t) g(t + m(t - 1/2)) dt$$

- **g(t)** is pattern-generation function
- **f(t)** is filter function
- **w** is filter extent
- **m** is minimum feature size in t

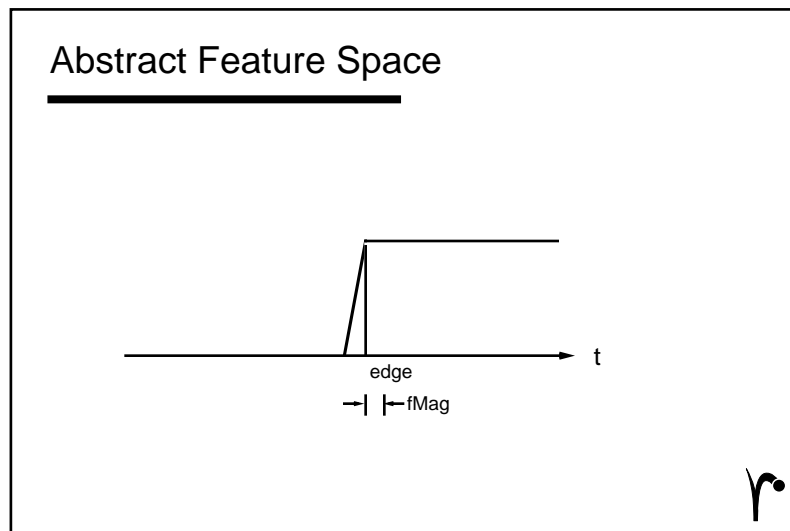


With the miserable performance of the Fourier stripe to prompt us, we turn to filtering in signal space rather than frequency space. A large class of useful patterns are generated as regular functions in some feature space. The value of the function at a particular point is used to control a visible feature such as opacity or color. A feature space might be linear distance from the center of a circle, the func-

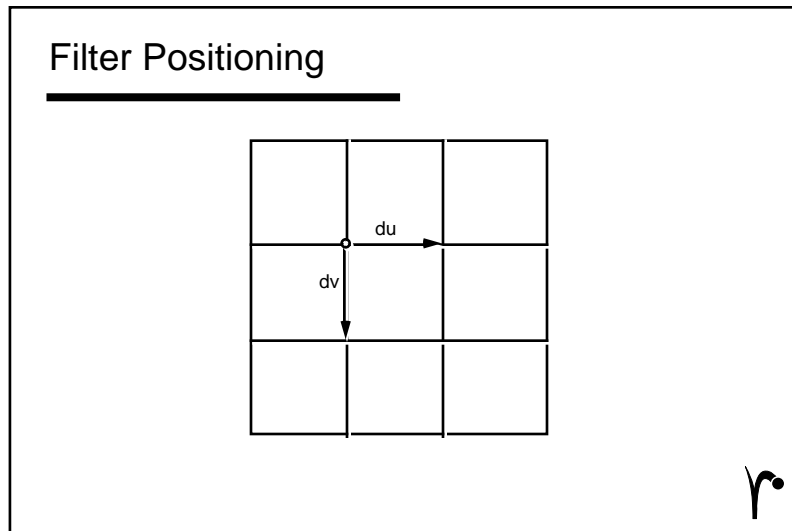
tion a step. This would produce filled circles. Checkers, eyelashes and ramps are other examples of such patterns.

These patterns can be filtered by convolving the feature with a low-pass filtering function. Since we have the analytic form of the feature, we can do the convolution analytically, rather than discretely. Given a feature space traversed by  $t$ , a pattern-generation function  $g(t)$ , and an appropriate, normalized, filtering function  $f(t)$ , we can use the equation on the slide to generate the low-pass filtered function  $g'(t)$ .  $w$  is the filter extent and  $m$  is the width *in feature space* of the micropolygon over which we are integrating.  $f(t)$  is zero outside  $\left[-\frac{w}{2}, \frac{w}{2}\right]$ .

While the one-dimensional case is illustrated for simplicity, extension to two-dimensions is straightforward.



In 1D, we see a simple edge in the feature space  $t$  (note that this is an abstract signal space, not necessarily the texture "t" coordinate).  $fMag$  represents the width of the minimum feature size: the micropolygon extent in  $t$  at the edge.



Note that the filter is *not* centered over the micropolygon. In *PRMan*, corners are used for position variables such as  $\mathbf{P}$ ,  $\mathbf{s}$ , and  $\mathbf{t}$ , while the micropolygon to the lower right in the grid is used to determine derivatives (actually differences). Thus, our best bet is to use shading interpolation “constant”, and to center the filter at the upper left of the micropolygon. In practice, this means adding a negative offset of half the micropolygon size in feature space.

### Simple Pattern Function

To compute the filtering integral efficiently, we can make tradeoffs in the design of the filter function to exploit our knowledge of the pattern generator. There are many shaders for which the pattern function is quite simple. Polkadots, spirals, diamonds and punched metal, for example, simply use an edge (*step*). If we consider only one edge transition, then  $g(t)$  is to one side always zero, to the other always one. The answer when  $g(t)$  is zero is the same for all  $f(t)$ , so that gives us part of our solution. The only other value of  $g(t)$  is one.

## Simple Pattern Function

$$g'(t) = \begin{cases} 1 & t \geq \text{edge} + m(w/2 - 1/2) \\ 2 \left( (x - \text{edge}) / m - 1/2 \right) & \text{edge} - m(w/2 - 1/2) < t < \text{edge} + m(w/2 - 1/2) \\ \int_{-w/2}^0 f(t) dt & \\ 0 & t \leq \text{edge} - m(w/2 - 1/2) \end{cases}$$

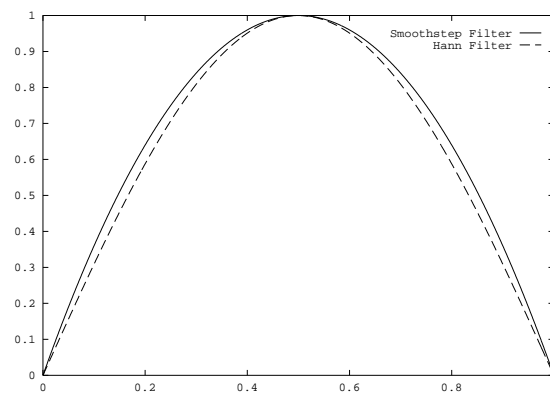
r

In this equation, `edge` is the location in  $t$  at which the desired edge is located.

For these types of edges, then, our only concern in computation is the filter function.

Interestingly enough, this is exactly what we're doing when we `smoothstep` over an edge instead of simply calling `step`. The addition here is a more rigorous understanding of what the “blur” parameters that often get thrown in there are actually doing.

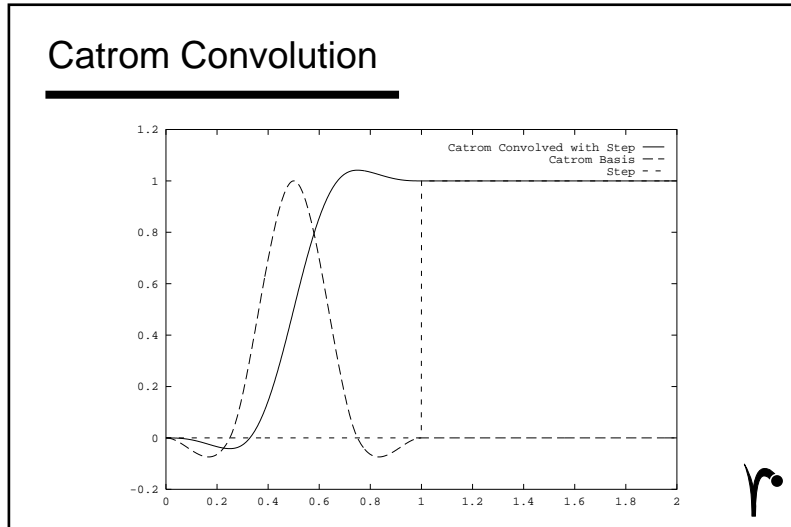
## Implicit Smoothstep Filter



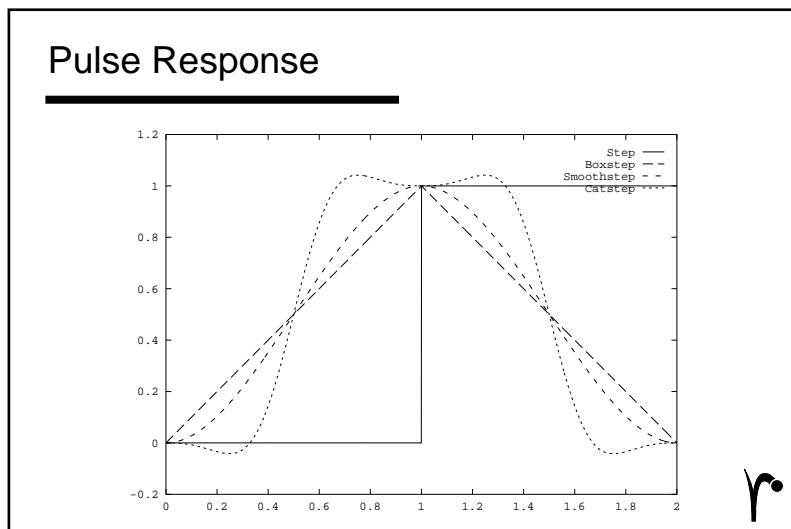
r

With this framework in place, let's look at exactly what filter we've been using all along when we've `smoothstep`'ed things. This polynomial filter is very similar to a Hann filter with  $\alpha=1$ , a class of

cosine filters. Depending on our needs, there are some much better filters out there. From this superficial look in the spatial domain, this doesn't look like a terribly exciting filter.

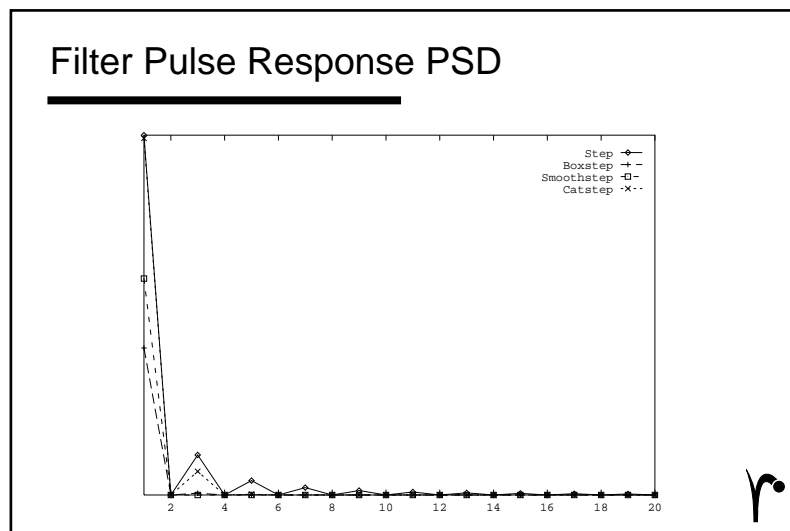


We'll now try a Catmull-Rom basis function because it is a relatively simple function which gives very good results for FIR image resampling. Here's a look at what's going on behind the scenes. Our big bunch o' math represents a convolution of our filter function with a step function. Imagine sliding the Catrom filter slowly to the right, computing the product of the two functions as they cross, and tracking the sum of all of those results. The result is the over/under-shooting edge you see. Note here that we're following our convention of not centering the filter at the edge.



It's useful to take a look at what happens when we filter a unit pulse. Note well that here we're setting the extent of all of these filters to be equal. This isn't strictly correct; the ideal extent will differ from filter to filter. But by setting them all equal we can get a better idea of their shapes. `boxstep` uses a box filter, the result being a ramp.

Notice particularly that the `catstep` function appears to be generating illegal values, at least when used to mix colors. Firstly, `mix` will just go ahead and extrapolate the colors for us. Secondly, this "negative energy" is usually absorbed by the ambient term in most shading models. The "extra bright" values usually are in range when shaded, and will be clamped at the final stage of imaging if need be. The negative lobes of this filter will tend to accentuate edges, making them appear crisp without aliasing. Sometimes this is just what you want. If not, go ahead and use `smoothstep`; it's built-in, and therefore a bit faster.



Finally, let's take a look at the spectrum of these filtered pulses to see what sort of advantages we can expect from this filter. `boxstep` and `smoothstep` seem to be cutting off frequencies near the fundamental excessively; that is, they are "blurry". `step`, of course, has way too many high frequencies. `catstep` here, in keeping with the unit extent we used for visual comparison, is in fact too narrow. Its proper extent is 4. We can thus expect it to be too leaky here. Nonetheless, it still manages to chop the third harmonic down by almost  $1/2$ , and effectively nukes everything else from then on.

If we "properly" set all of the filter's extents, `catstep` looks even better. However, it is worth remembering that the extent is one more parameter you should feel free to fiddle, as you can use it to control the blurriness or sharpness of your edges if you wish.



## Filteredge

```
/* This macro filters a given edge transition using a Catmull-Rom basis
 * function FIR filter. 'edge' is the location of the requested transition,
 * 'fMag' is the filter magnification factor (usually the change in 't'
 * across
 * a micropolygon), and 't' is the current location in feature space.
 */
```

```
#define FILTEREDGE(edge, fMag, t)
    catstep((edge)-((fMag)*5/2),
            (edge)+((fMag)*3/2), (t))
```

- **Same technique can be used with any desired filter function.**



This macro gives you a low-pass filtered edge. You can make fMag “too big” to blur the edge if you want. catstep is a shading language function you would need to include in any shader which uses *FILTEREDGE*.

```
/*
 * This function computes the result of convolving a catrom
 * basis of width (s1-s0) with a step function beginning at s0.
 * The basis fn. BEGINS at s0; it is not centered there.
 */
float
catstep(float s0, s1, s)
{
    float a;
    a = (s-s0)/(s1-s0);
    if (a <= 0)
        a = 0;
    else if (a >= 1)
        a = 1;
    else if (a < 1/4)
        a = 32*a*a*a*(-1/3 + a);
    else if (a < 1/2)
        a = -5/6 + a*(12 + a*(-64 + a*(416/3 - 96*a)));
    else if (a < 3/4)
        a = 67/6 + a*(-84 + a*(224 + a*(-736/3 + 96*a)));
    else
        a = -61/3 + a*(96 + a*(-160 + a*(352/3 - 32*a)));
    return a;
}
```

## Polkadots

---

```
fMag = length(dsVec + dtVec);
edge = FILTEREDGE(jitRad, fMag, length(rVec));
```



The actual filtered edge for polkadots is quite easy to compute. `jitRad` contains the desired radius of a dot (the location of an edge), while `rVec` points towards the origin of the dot. One call to `FILTEREDGE` and we've got a nicely antialiased edge.

There's a bit of a trick in this shader which is worth noting. Since we are randomizing *both* dot radius and dot location, we no longer are guaranteed that our dots will lie within one of the "cells" into which we have divided our *st* space. Thus, we need to examine the three-by-three neighborhood around our current cell to see if there are dots from adjacent cells contributing to this one.

When we're all done with the color calculation, we realize that we've only filtered *one* edge. When seen from really far away, multiple edges will fall into a given sample. So, we break out the *Global Mixing* trick. We can calculate the "average color" since we know what the average dot density is. We are aided in this mixing by a side effect of the neighborhood contribution accumulation we were forced to do; multiple edges *will* manage to find their way into a sample. They may not do so entirely correctly. But this helps to reduce the bandwidth of the function we are fading, and thus enables us to pull off a global mix.

Here's the code:

```
/*
 * This shader produces randomly sized and positioned Polka Dots
 * of a specified color.
 *
 * The polkadots have a radius of 1 inch. dotcolor specifies the
 * polkadot color; Cs is used for the background. dotdensity
 * requests a specific average dot density, [0.,1.]. posvariance
 * determines the average amount by which the result will differ
 * from regularly placed dots [0.,1.]. Finally, sizevariance
 * request an average amount by which the result will differ
```

```

    * from dots of the same size, with a range of [0.,1.].
    */
#include "filter.h"
#define DOTDENSITY      clamp(dotdensity, 0, 1)
#define POSVARIANCE     clamp(posvariance, 0, 1)
#define SIZEVARIANCE    clamp(sizevariance,0,1)
#define DOTRAD          (1/12)/* one inch radius */

/* center of 3x3 neighborhood */
#define SPOS floor(ss + .5)
#define TPOS floor(tt + .5)

#define TMP    sPosJit

surface
polkadots (
    uniform color  dotcolor = color (1, .03, .737); /* hot pink */
    uniform float  dotdensity = 0.5;
    uniform float  posvariance = 1;
    uniform float  sizevariance = 1;
    uniform float  Ks=.1;
    uniform float  Kd=.7;
    uniform float  Ka=1;
    uniform float  roughness=.1;
    uniform color  specularcolor=1;
)
{
    /* local vars */
    varying float  dotFrac;
    varying float  ss, tt;
    varying float  sPosJit, tPosJit;
    varying float  x, y;
    varying float  fMag, jitRad;
    varying point  rVec;
    uniform float  dScale;          /* # dots in one linear unit */
    varying point  dsVec;
    varying point  dtVec;
    varying point  V, Nf;

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

    setzcomp(rVec, 0);
    setzcomp(dsVec, 0);
    setzcomp(dtVec, 0);

    dScale = DOTDENSITY / (2*DOTRAD);

```

```

ss = s * dScale; /* map into lattice; one lattice pt per dot */
tt = t * dScale;

/*
 * distance is discontinuous at dot-grid boundaries,
 * use radial surface approximation instead.
 */
setxcomp(dsVec, Du(ss)*du);
setycomp(dsVec, Dv(ss)*dv);
setxcomp(dtVec, Du(tt)*du);
setycomp(dtVec, Dv(tt)*dv);
fMag = length(dsVec + dtVec);

/* There are nine dots which might contribute; sum the
 * contributions.
 * NB variance must be <= 1 ; bigger implies probability of
 * shaved dots and is useless anyway.
 */
dotFrac = 0; /* assume in background to start */

for (y = TPOS-1; y<TPOS+2; y+=1) {
    for (x=SPOS-1; x<SPOS+2; x+=1) {
        /* Perturb lattice location */
        sPosJit = x + POSVARIANCE*snoise(x+111.5,y+111.5);
        tPosJit = y + POSVARIANCE*snoise(x+222.5,y+222.5);

        setxcomp(rVec, ss - sPosJit);
        setycomp(rVec, tt - tPosJit);

        jitRad=dScale*DOTRAD*
        (1+SIZEVARIANCE*snoise(sPosJit+333.5,tPosJit+333.5));

        /*
         * Zap high frequencies with catrom
         */
        TMP = FILTEREDGE(jitRad, fMag, length(rVec));
        if (abs(1-TMP) > abs(dotFrac))
            dotFrac = 1-TMP; /* dots are opaque */
    }
}

/* fade to average color between Nyquist/4 and Nyquist */
/* we can't cut it more closely because grid boundaries
 * will show
 */

```

```

Ci = mix(mix(Cs, dotcolor, dotFrac), mix(Cs, dotcolor,
                                          DOTDENSITY),
          smoothstep(dScale*DOTRAD/2,dScale*DOTRAD*2,fMag));

Oi = Os;
Ci = Oi * (Ci * (Ka*ambient() + Kd*diffuse(Nf)) +
           specularcolor * Ks * specular(Nf,V,roughness));
}

```

### Simple Filter Function

#### Simple Filter Function

Many edges on a given piece of geometry  
Simplify filter function — box filter

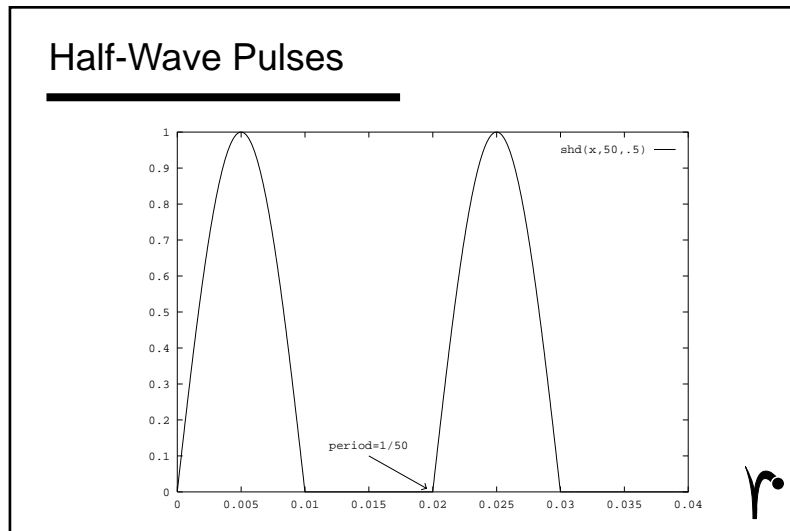
$$g'(t) = \frac{1}{w} \int_{t-m(w/2-1/2)}^{t+m(w/2-1/2)} g(t) dt$$

✓

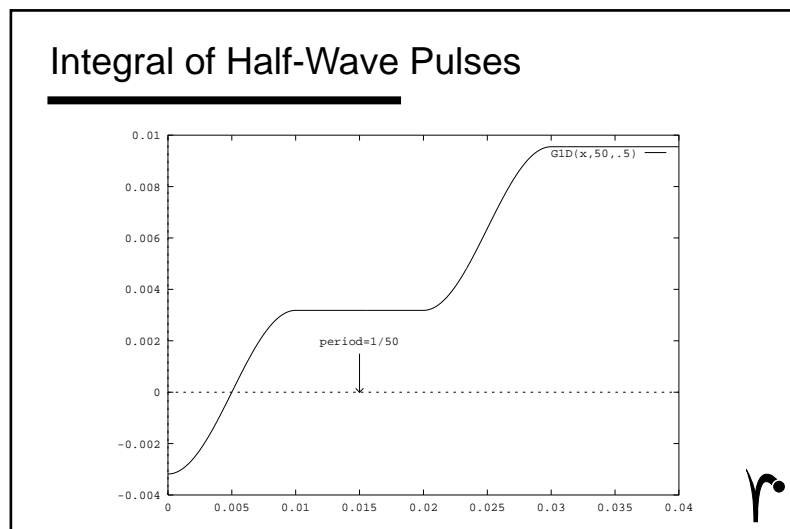
When edges are relatively isolated, as in a piece of geometry with a diagonal stripe on it, the previous approach works well. However, when there are many edges on a given piece of geometry, our assumptions break down. We assumed that  $f(t)$  only makes one transition within the filter extent. With repetitive features this may no longer be true, and we can not simply filter each edge in isolation. Checkers in perspective will alias, although the individual edges may be filtered. The solution here is that the entire pattern-generation function must be filtered.

Combining this with a complicated filter function quickly makes the task daunting. We can simplify matters by using a simpler filtering function, namely a box:  $f(t) = \frac{1}{w}$  for  $\frac{-w}{2} < t < \frac{w}{2}$ . We can pull out the constant and change limits to get the equation on the slide, which tells us that we need to integrate the pattern-generator  $g(t)$  over a chosen interval.

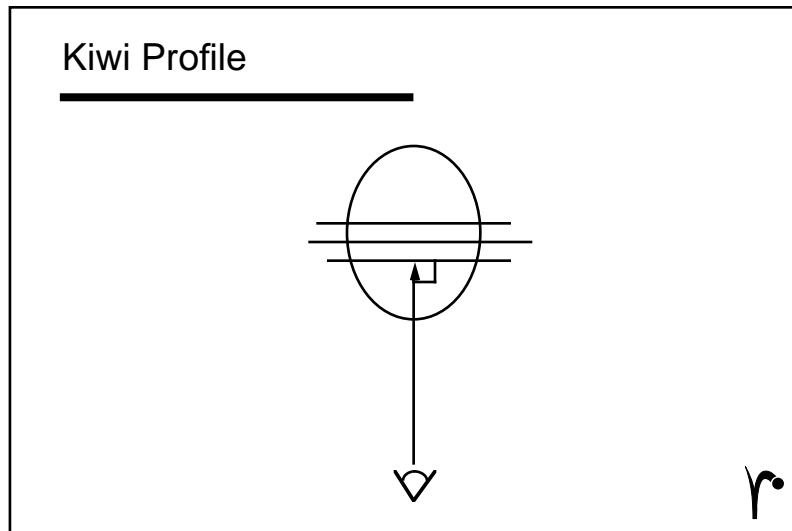
For simplicity, or lack of a better symbolic math program, we often solve for the indefinite integral and leave the rest to the shader compiler; our shader then need only evaluate it twice for the one-dimensional case or four times in the two-dimensional case to recover the definite integral.



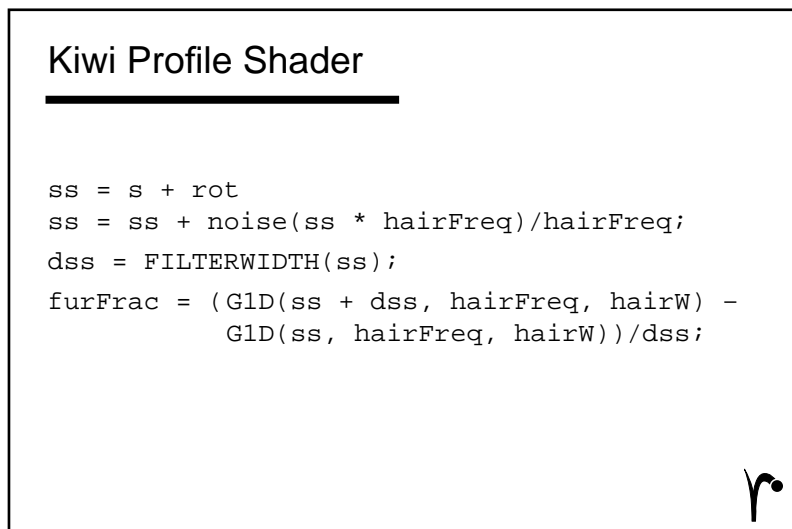
Let's look at a relatively simple, but very useful function which we have used in productions to generate rope, eyelashes, fur and so on. This is a half-wave sine pulse train, with variable frequency and variable pulse width. In this example, the pulse width is .5, so there is equal empty space between the pulses. The sinusoidal nature of this function is useful for generating “fake” cylinders.



If you don't have a symbolic math program which is comfortable with step functions, don't despair. These functions are often fairly reasonable to solve by inspection. Typically, you integrate the non-step portion, and then hook a bunch of them together with lines of the appropriate slope. Staring at plots of these functions will quickly improve your intuition.



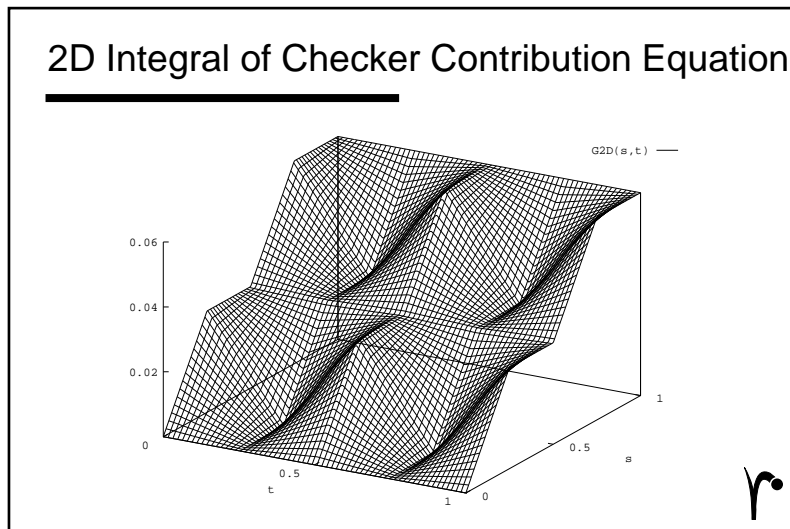
Here's an example of where such a shader was called for. We needed to create a fuzzy kiwi for a fruit juice commercial. The hairs on the surface of the kiwi were created by a shader on the fruit itself, which simulated the projection of hairs onto the surface. It could not, however, produce hairs which really poked outside the bounds of the surface, which was necessary to produce convincing profiles. Thus, several flat surfaces were created near the middle of the fruit, which roughly followed its outside boundaries. They were automatically kept perpendicular to the camera. Each of them used a jittered version of the sine half-waves to modulate their transparency around their outside edges, essentially creating fake cylinders.



This is the relevant chunk of shader code which generated the hairs. *furFrac* was then used to shade the fake cylinders. The actual functions used to produce the integral are given below. Note that they are not rewritten into their most compact form, but serve to illustrate the way in which the integrals are pieced together from simple components.

```
/* Common tessellation thing; mod(x,1) screws up across zero */
#define FRAC(x) ((x)-floor(x))

/* 1D integral of contribution equation
 * antialiased half-wave function with variable spacing
 * "freq" is the frequency of the sine-wave itself, "d" is a
 * scalar for the frequency of occurrence of the wavelets,
 * relative to "freq". Thus, "d" varies from epsilon (infinitely
 * far apart) to 2 (bunched up). At "1", the bumps and empty
 * spaces are equally separated.
 *
 */
/* DC component */
#define DC(t,freq,w) (floor((t)*(freq))/((freq)/(2*(w))*PI))
/* integral */
#define G1D(t,freq,w) \
  ((FRAC((t)*(freq))/(2*(w))<.5)?\
   -cos(FRAC((t)*(freq))/(w)*pi)/((freq)/(w)*pi)+DC((t),(w)):\
   DC((t),(w)) + 1/((freq)/(w)*pi))
```



So far I've waved my hands through the 2D case by saying it's "straightforward". We only need to evaluate a double integral instead of a single one. So let's try it with an old CG cliché, the checkerboard.



These functions are hard enough to visualize in 1D, let alone 2D. The slide shows a piece of the integral, with four “checks” on the unit  $st$  square.

## 2D Checkers Shader

```
dssDu = max(abs(Du(ss)*du, MINDERIV);
dttDv = max(abs(Dv(tt)*dv, MINDERIV);
frac = ((G2D(ss + dssDu, tt + dttDv, CHECKS) -
        G2D(ss - dssDu, tt + dttDv, CHECKS) -
        G2D(ss + dssDu, tt - dttDv, CHECKS) +
        G2D(ss - dssDu, tt - dttDv, CHECKS)) /
        (4 * dssDu * dttDv)) + .5;
```



Here’s the relevant chunk of shader code. `frac` contains the relative mix of the two checker features. The extra “+.5” is there because we selected a convenient integral which gives results in the range  $[-.5, .5]$ . Notice that we indeed call the indefinite integral four times, as we would expect. In this particular case, we are filtering over the region  $(ss-dss, tt-dtt)$  to  $(ss+dss, tt+dtt)$  rather than  $(ss, tt)$  to  $(ss+dss, tt+dtt)$ . This doubles the filter width, which we did in an attempt to hide visible grid boundaries. Checkers have very visible high-frequency content, and sharp differences in frequency content within the image are easily visible.

Here’s the actual integrals as used by the code. Again, they are in a hopefully easier-to-understand form.

```
/* 1D integral of contribution equation */
#define G1D(t,cycles)\
((max(0,FRAC((t)*(cycles))-.5) + floor((t)*(cycles))/2)/(cycles))

/* 2D integral of contribution equation - lerp between
 * two offset versions of the 1D function
 */
#define G2D(s,t,cycles)\
((G1D((s)+(1/((cycles)*2)),(cycles))*abs(FRAC((t)*(cycles))*2-1)\
+G1D((s),(cycles))*(1-abs(FRAC((t)*(cycles))*2-1)))/(4*(cycles))
```

## Grids and You

---

- **Micropolygon size changes only at grid boundaries in PRMan.**

Increase filter widths

Lower maxgrid size

Use a sampling metric which doesn't depend on "derivatives"



If derivatives and micropolygon sizes changed continuously over geometry, the above filtering would work nicely. Unfortunately for us, *PRMan* changes micropolygon sizes only at grid boundaries, and provides us with differentials rather than derivatives. Assumptions are also made about the behavior of these differentials at grid boundaries. Micropolygon size can thus change abruptly, and along a line or curve; here there will be a difference in frequency content between the two sides of this curve. If this difference is great enough, a trapezoidal or curved region (the grid with lower frequency content) will be plainly visible. Increasing the width of our  $f(t)$  filter to cause more blur can ameliorate the problem, but it can always be induced. Setting the `maxgridsize` option to a low value helps to reduce the difference between adjacent grids, and can tremendously reduce visible artifacts.

All of the antialiasing methods which we have discussed are affected by this discontinuous behavior.

## Pun'Ting

### Pun'Ting

- **Some patterns are too difficult to filter analytically and contain too many frequencies to be effectively “mixed”.**
- **Punt**
- **If the shader uses noise, this may work**

Aliased noise is still noise.

If it moves, you're hosed.

Turn up the shading rate and wait.



There are some cases in which a given look can not be analytically filtered, and contains too many frequencies to be “mixed”. The “granite” shader, which uses one octave of noise, is a good example. In this case, we can often punt. Aliased noise is still noise, and in many cases looks acceptable in still images. Such shaders will, however, contract a bad case of the creepy-crawlies when animated.

## Alternative Measures of Allowable Frequency Content

### Alternative Measures of Allowable Frequency Content

- **To avoid grid artifacts, we may need to use metrics which do not depend on derivatives. They may not even depend directly on sampling rate.**

Dependence on camera position.

Project features into raster space.



Visible grid derivative discontinuities can often be avoided by judicious geometric construction. There are still cases, most notably severe perspective, where these discontinuities are unavoidable. Lowering `maxgridsize` and increasing filter widths can reduce or eliminate the problem for particular shots. If the camera or geometry moves to induce a more severe perspective, we're hosed again.

Since our troubles stem from the use of functions which depend on micropolygon size, we can eliminate our troubles by eliminating or being more careful about our use of derivatives.

### Distance from Camera for Simple Geometry

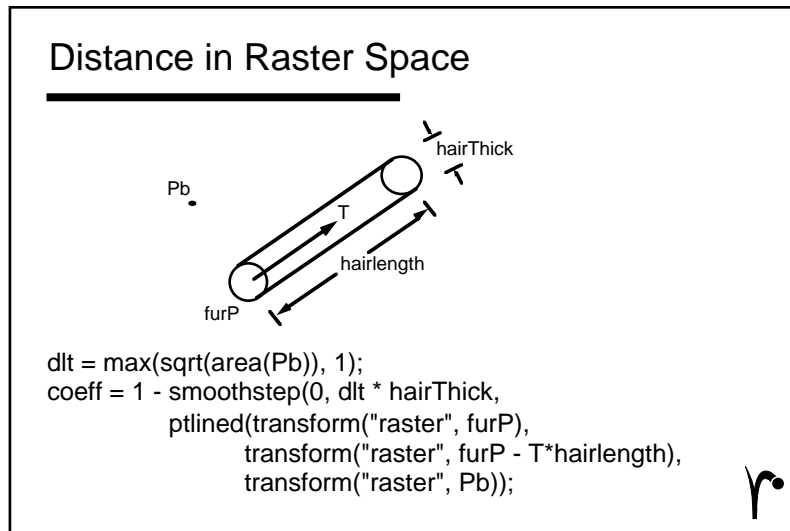
```
wP = transform("world", P);
wE = transform("world", E);
dist = length(wE - wP);
projVec = wE - wP;
setzcomp(projVec, 0); /* project into xy plane */
cosTheta = abs(normalize(wE - wP) .
               normalize(projVec));
fade = dist * cosTheta;
fade = smoothstep(fadeMin, fadeMax, fade);
```

- **Shader on ground plane. Fade based on distance and viewing angle.**
- **fadeMin and fadeMax are found through the Method of Exhaustion.**

Distance from the camera can often serve as a useful filtering metric. Since it does not depend on grid size, it will not be discontinuous. For simple geometry, we can also exploit our knowledge of the geometry to determine where projective effects will come into play.

A very common situation is a ground plane reaching off into the distance in perspective. What we're doing here is calculating a *fade* amount based on distance to the camera and angle with the ground. The further away or more obliquely we view the ground, the less detail we want.

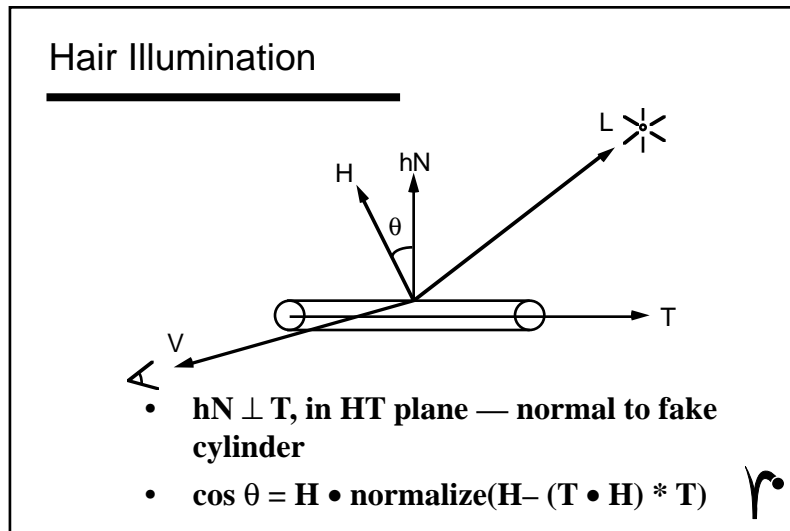
This approach usually involves setting the distance bounds through empirical search.



Finally, we may have a feature described directly by points in space. If so, we can try transforming those points into raster space, and taking the distance between them in this new space. This will give us a direct measure of the pixel size of the feature, without regard to micropolygon sizes. We can then apply our choice of filtering.

This example comes from the kiwi shader I mentioned before. We've already taken care of the hair on the silhouettes, now we need to handle the hairs over the surface itself. The shader generates a jittered grid of "follicles" for the hairs. Vectors of the appropriate length are then wiggled around from there to define the endpoint of these hairs. A neighborhood accumulation technique similar to what was done for polkadots ensures that all contributions to a cell are accounted for.

Given these endpoints for the hair, we then find the distance of the current surface point to the center of this hair, *in raster space*. This simultaneously lets us do filtering, and projects the hair into the appropriate place on the surface "underneath". Although we do in fact look at the micropolygon width, we only care when it's greater than one pixel. This greatly ameliorates discontinuities, since by that time we're already blending hairs together. Note that the neighborhood accumulation should really be a depth ordering. We don't have that kind of time to spare, so we just smush them all together when they overlap, and ignore their order. Since they're the same color in this case, that's not too bad.



Now that we have our points for this virtual hair, and know how much to weight it, we need to shade it. We don't have any geometry for the hair, so we need to make some up. We're most interested in specular effects, since the specularity of hair is what makes it so interesting.

We need to compute the angle between the virtual hair normal,  $hN$ , and the half-vector between  $V$  and  $L$ . The radius of the cylinder ends up being irrelevant. All we need to determine is  $hN$ . Knowing  $hN$  to be, by definition, perpendicular to  $T$  and in the  $HT$  plane, we simply apply a little linear algebra, and presto!

We control the specular function directly by choosing a range of specular angles over which we want a specular highlight, and smoothly interpolating. This gives us easy control over the shape of highlights.

Finally, the hairs are merged on-the-fly over the bump-mapped and color-fiddled surface of the fruit. The complete code for the shader follows below.

```
#define HAIRS
#define BODY

surface kiwi(
    float   furOpac=0.85; /* opacity of fur */
    /* fur reflection model */
    float   furKa=.2;
    float   furKd=.5;
    float   furKh=.15; /* highlight coeff. for fur */
    color    furC=color(0.1960,0.1013,0.0230); /* fur color */
    color    furCh=color(0.4422,0.2525,0.0956); /* fur highlight color */
    float   furHAMin=0; /* beginning of cutoff (in degrees of */
                        /* specular angle) for fur highlight */
```

```

float    furHACut=18; /* cutoff for fur highlight */
float    illumFudge=0; /* extra degrees to add to half-sphere */
                        /* illumination of hairs to fake */
                        /* scattering */

float    bodyKa= 1;
float    bodyKd=.6;
float    bodyKs=.1;
float    bodyRough=.4;
/* hair generators */
float    furSFreq=150; /* # follicles in u direction per */
                        /* parametric unit */
float    furTFreq=150; /* # follicles in v direction per */
                        /* parametric unit */
float    hairLength = .2; /* length of hairs relative to grid
*/
float    hairThick = 2; /* thickness of a hair in screen */
                        /* space */

/* Fur is generated by warping a grid of follicles by a bunch
 * of random vectors. The endpoints of these hairs are also
 * warped by a second set of vectors. These control the
 * direction of these vectors.
 */
float    swirlFollScale = 1;
float    swirlTipScale = 1;
float    rot = 0; /* make it different for different objs */
float    ridgeAmp = .025;
float    ridgeSFreq = 100;
float    ridgeTFreq = 20;
float    bodyCSFreq = 30;
float    bodyCTFreq = 27;
) {
    varying point    Nf;
    uniform float    i, j;
    varying float    pu, pv, iu, iv, wiu, wiv;
    varying point    furP, T, Ln;
    varying float    specDAng, dlt, coeff;
    varying color    hairCAcc = color(0,0,0); /* fur color */
                                                /* accumulator */
    varying float    opacAcc=0; /* fur opacity accumulator */
    varying float    theta; /* generic angular quantity */
    varying point    V;
    varying color    Cillum;
    varying point    Pb;
    varying color    Cbody;
    varying point    H;

```

```

color gr  = color (0.0981,0.1608,0.0196);
color brgr = color (0.1608,0.1205,0.0202);
color br   = color (0.1357,0.0547,0.0102);
Nf = normalize(faceforward(N, I));
V = normalize(-I);

/* vertical ridges */
Pb = pnoise(ridgeSFreq*s+rot, ridgeTFreq*t+rot,
            ridgeSFreq, ridgeTFreq*2);

Pb = P + ridgeAmp*Pb*Nf;

Nf = faceforward(normalize(calculatenormal(Pb)), I);

#ifdef HAIRS
/* get opacity and color contribution of the hairs */

/* regular grid for hairs */
iu = mod(floor(u*furSFreq+0.5), furSFreq);
iv = mod(floor(v*furTFreq+0.5), furTFreq);

/* accumulate over 3x3 follicle neighborhood in attempt to
 * account for overlap between lattices
 */

for (j = -1; j < 2; j += 1) {
    for (i = -1; i < 2; i += 1) {
        /* wrapped versions */
        wiu = mod(iu + i, furSFreq);
        wiv = mod(iv + j, furTFreq);
        if (wiu < 0)
            wiu += furSFreq;
        if (wiv < 0)
            wiv += furTFreq;
        pu = (wiu +
              snoise(wiu*(111+(1/7))+rot, wiv*(111+(1/7))+rot))/
              (furSFreq-1);
        pv = (wiv +
              snoise(wiu*(222+(1/7))+rot, wiv*(222+(1/7))+rot))/
              (furTFreq-1);

        /* furP is the location of the base of the follicle,
         * T is the axis of the hair cylinder
         */
        furP = Pb + (pu-u)*dPdu + (pv-v)*dPdv;
        T = normalize(Nf+(pu-u)*Du(Nf)+(pv-v)*Dv(Nf));
    }
}

```



```

setxcomp(T,xcomp(T) +
    snoise(iu*(333+(1/7))+rot, iv*(333+(1/7))+rot));
setycomp(T,ycomp(T) +
    snoise(iu*(444+(1/7))+rot, iv*(444+(1/7))+rot));
setzcomp(T,zcomp(T) +
    snoise(iu*(555+(1/7))+rot, iv*(555+(1/7))+rot));
T = normalize(T);

/* about the only safe filtering metric */
dlt = max(sqrt(area(Pb)), 1);
coeff = 1-smoothstep(0, dlt*hairThick,
    ptlined(transform("raster", furP),
        transform("raster", furP-T*hairLength),
        transform("raster", Pb)));

/*
 * We need to sum all contributions, but we don't want
 * to waste time doing an A-buffer ordering (since they
 * may have different lighting colors).
 * We could merge each one "over" the other, but since
 * the ordering isn't defined, we'll just accumulate
 * them and stop when we've got full coverage.
 */
if (opacAcc+coeff*furOpac > 1)
    coeff = (1-opacAcc)/furOpac;

if (coeff > 0) {
    /*
     * A lambertian diffuse falloff for the
     * hair color is used, and an additional
     * highlight color is thrown in through an
     * interpolated range of specular difference angles.
     *
     * What we want is the difference between the
     * half-angle vector formed by L and V [H], and the
     * perpendicular to T in the HT plane.
     */
    Cillum = 0;

    illuminance(P, normalize(faceforward(N,I)),
        PI/2+illumFudge) {
        if (L != 0) {
            /* NB - a bug in PRMan causes ambient lights to
             * show up in the illuminance loop, with L=0.
             * These are normally excluded automatically
             * by L.N shading, but here, with our wacky

```

```

        * tangent shading, we must exclude them
        * "by hand".
        */

    Ln = normalize(L);
    H = normalize(Ln+V);

    specDAng =
        acos(H . normalize(H - (T . H ) * T));

    Cillum = Cillum + furC*furKd*(Nf.Ln) +
        furKh*furCh*
        (1-smoothstep(radians(furHAMin),
            radians(furHACut),
            specDAng));
    }
}
Cillum = Cillum + furKa * ambient() * furC;

/* accumulate contributions */
coeff *= furOpac;
hairCAcc = hairCAcc +
    mix(color(0,0,0), Cillum, coeff);
opacAcc += coeff;
    }
}
}

#ifdef BODY
    Oi = opacAcc;
    Ci = hairCAcc;
#endif
#endif /* HAIRS */
#ifdef BODY
    /* now compute color of body */
    coeff = float pnoise(bodyCSFreq*s+rot, bodyCTFreq*t+rot,
        bodyCSFreq, bodyCTFreq*2);
    coeff += float pnoise(bodyCSFreq*s+rot+234.5,
        bodyCTFreq*t+rot+712.5,
        bodyCSFreq, bodyCTFreq*2);
    coeff /= 2;

    Cbody = spline(coeff,gr,gr,brgr,br,br);

#ifdef HAIRS

```

```

    Oi = Os;
    Ci = Os*(Cbody * (bodyKa*ambient() + bodyKd*diffuse(Nf)) +
              bodyKs * specular(Nf,V,bodyRough) );
#else

    /* merge fur over object */
    Oi = Os;
    Ci = hairCAcc +
        (1-opacAcc)*Oi*
        (Cbody * (bodyKa*ambient() +
                  bodyKd*diffuse(Nf)) + bodyKs *
          specular(Nf,V,bodyRough) );
#endif /* !HAIRS */
#endif /* BODY */
}

```