# Renderman Shaders

## Shading Language and Basic Shaders

# Renderman Shaders

- Renderman provides a number of different shader types all of which can be written in the Shading Language (SL)
- These are as follows
    - Surface Shaders describe the appearance of surfaces and how they react to the lights that shine on them.
    - Displacement Shaders describe how surfaces wrinkle or bump.
    - Light Shaders describe the directions, amounts and colours of illuminations distributed by a light source in the scene
    - Volume Shaders describe how light is affected as it passes through a participating medium such as smoke or haze.
- Other Renderer (such as BMRT) also support Image shaders

# Shading Language 2.0

- As of Version 13.5 the shading language has changed to add the following features
  - Shader Objects and Co-Shaders
  - Shaders can now have member variables and multiple methods. For example, a surface shader can define displacement and opacity methods, and share state with them via member variables.
  - Shaders can call methods and read member variables of other shaders.
  - Scene descriptions can now include co-shaders that allow custom shading pipelines to be expressed in the shading language itself.
- These new features will be explored later in the course for now we will concentrate on the basics.

# What is a Shader

- A shader essentially asks the Question "What is going on a this spot"?
- The execution model of the shader is that the programmers are only concerned with a single point on the surface and are supplying information about that point
    - This is know as an implicit model
- The job of a surface shader is to calculate the colour and opacity at a particular point on some surface.
- To do this it may calculate any function, do texture map look-ups, gather light and so on.
- The shader starts with a variety of information about the point being shaded but cannot find out about any other points (although from Version 11 there is now the ability to do this using the trace function)

# The RM Shading Language

- The Renderman Shading Language is a C like language you can use to program the behaviour of lights and surfaces. SL gives you :-
  - basic types useful for manipulating points, vectors and colours
  - mathematical, geometric and string functions
  - access to the geometric state at the point being shaded, including the position, normal, surface parameters, and amount of incoming light
  - parameters supplied to the shader, as specified in the declaration of the shader or alternatively attached to the geometry itself.
- With this information, the goal of the surface shader is to compute the resulting colour, opacity and possibly the surface normal and / or position at a particular point.

# A simple Shader

```
1  surface plastic( float Ks=.5, Kd=.5, Ka=1, roughness=.1;
2                   color specularcolor=1 )
3  {
4      normal Nf;
5      vector V;
6      Nf = faceforward( normalize(N), I );
7      V = -normalize(I);
8      Oi = Os;
9      Ci = Os * ( Cs * (Ka*ambient() + Kd*diffuse(Nf)) +
10                 specularcolor * Ks * specular(Nf,V,roughness) );
11 }
```

# Data Types

| Data type | Description |
|---|---|
| float | scalar floating point data (numbers) |
| point | Three dimensional positions |
| vector | directions |
| normal | surface orientations |
| color | Spectral reflective and light energy values |
| matrix | 4 x 4 transformation matrix |
| string | Character strings (usually for filenames) |

# Colours

- Colours are represented internally by 3 floating point components
- They are referred to by a colour space as follows

| Colour Space | Description |
|---|---|
| “rgb” | The co-ordinate system that all colours start out in and in which the renderer expects to find colours that are set by your shader |
| “hsv” | hue, saturation and value |
| “hsl” | hue, saturation and lightness |
| “YIQ” | The colour space used for the NTSC television standard |
| “xyz” | CIE XYZ coordinates |
| “xyY” | CIE xyY coordinates |

# Colours

```
1  color black = color (0,0,0);
2
3  color green = color "rgb" (0,0,1);
4
5  color hsvcol = color "hsv" (0.2,0.5,0.63);
6
7  color white = 1; // set all components to 1
```

- The examples show how colours may be set in various ways
- All the example return a value in rgb space, even the hsv version.
- To modify components of the colour variable we use the following

```
1  setcomp(white,0,0.3);
```

# Points, Vectors and Normals

- Points, Vectors and Normals are similar data types with identical structures but different semantics
- A Point is a position in 3D space
- A vector has a length and direction but does not exist in a particular location
- A normal is a special type of vector that is perpendicular to a surface and thus describes the surface's orientation.

```
1 point p1=point(0,2.3,1);
2 vector dir= vector(1,0,0);
3 normal NF = normal(0,1,0);
```

# Co-ordinate Systems

- There are many different coordinate systems that the renderer knows about.
- A Point in 3D can be represented by many different sets of 3D numbers one for each co-ordinate system
- Typically all data passed to the shader are in the “current” co-ordinate system however we can transform them to be in different spaces by specifying the following in the construction of the data type

# Names of pre-declared geometric space

- **"current"** The co-ordinate system that all points start out in and the one in which lighting calculations are carried out. Note that the choice of current space may be different on each renderer.
- **"object"** The local co-ordinate system of the graphics primitives (sphere, plane, patch etc) that we are shading.
- **"shader"** The co-ordinate system active at the time that the shader was declared (by the Surface, Displacement or Lightsource statement)
- **"world"** The coordinate system active at WorldBegin

# Names of pre-declared geometric space

- **“camera”** The co-ordinate system with its origin at the centre of the camera lens, x-axis pointing right, y-axis pointing up and the z-axis pointing into the screen
- **“screen”** The perspective-corrected coordinate system of the camera's image plane. Co-ordinate (0,0) in “screen” space is looking along the z-axis of “camera” space
- **“raster”** The 2D projected space of the final output image, with units of pixels. Co-ordinate (0,0) in “screen” space is the upper-left corner of the image with x and y increasing to the right and down respectively
- **“NDC”** Normalised device coordinates- like raster space but normalised so that both x and y range from 0 -1 .

# Matrices

```
1  matrix zero =0; // a zero matrix
2
3  matrix ident = 1; set an identity matrix
```

- SL has a matrix type that represents the transformation matrix. Internally it is represented by 16 floats.
- Matrices are defined in row major order
- They can be tested for equality using == and !=
- * is used to multiply two matrices together
- m1/ m2 denotes multiplying the matrix by the inverse of matrix m2 thus a matrix can be inverted by writing 1/m

# Strings

```
1 string TextureName = "image1.tx"
```

- Strings are typically used to hold file names for textures, filenames etc
- strings can be manipulated using the format() and concat() functions

# Shading Language Variables

- These variable called either “global variables” or “graphics state variables” contain the basic information that the renderer know about the point being shaded.
- These include position, surface orientation, the default surface colour.
- These variables do not have to be declared the are simply available by default in the shader.
- Think of these as the communication between the shaders and the renderer

# Global variables available inside surface and displacement shaders

| Variable | Description |
|---|---|
| point P | Position of the point you are shading, Changing this variable displaces the surface |
| normal N | The surface shading normal (orientation) at P. Changing N yields bump mapping |
| normal Ng | The true surface normal at P. This can differ from N; N can be overridden in various ways including bump-mapping and use-provided normals, but Ng is always the true surface normal of the facet your are shading |
| vector I | The incident vector, pointing from the viewing position to the shading position |
| color Cs Os | The default surface colour and opacity, respectively |
| float u, v | The 2D parametric co-ordinates of P (on the particular geometric primitive you are shading) |
| float s, t | The 2D texturing co-ordinates of P. These values can default to u, v but a number of mechanisms can override these values |
| vector dPdu<br>vector dPdv | The partial derivatives (tangents) of the surface at P |
| time | The time of the current shading sample |
| float du dv | An estimate of the amount that the surface parameters u and v change from sample to sample |
| vector L<br>color Cl | These variables contain the information coming from the lights and may be accessed from inside illuminance loops only |
| color Ci, Oi | The final surface colour and Opacity of the surface at P. Setting these two variables is the primary goal of a surface shader |

# Local Variables

- Local variables are the same as those used in C/ C++ and their scope rules are also the same

```
[class] type variablename [= initialiser]
```

- The optional class specifies either uniform or varying. If this is not used it defaults to varying within the shader.

- Arrays in SL are only constant length and can only be 1 dimension

```
float a; // declare a variable
uniform float b;  // declare a variable as a uniform
float c=1; // declare and  assign
float d= b*a;
float e[10];
```

# Shader Parameters

- shader parameters allow the artist using the shader to have greater control of the final image.
- If the elements within the shader are hard coded this means that the shader will not be very flexible and we need redeveloping
- To overcome this problem we parametrise as much as possible the internal values passed to the shader as shown in the example.
- We also try to set default "good" values for each parameter so the user does not have to pass them but gets the option to override them.

# Shader Parameters

```
class Citrus(
      float veining = 5;
      float Ambient= 1;
      float Kd = .65;
      float Ks =0.5;
      float Kx =200;
      float Ky =300 ;
      float roughness = .7;
      float mix = .2;
      float turbfreq = .02;
      float SScale=100;
      float TScale=100;
      uniform color SpecColour=color "rgb" (0.600 ,0.200, 0.000);
      float Km = 0.31;
      varying float frequency = 0.5;
      varying float amp = 0.5;
      uniform color C1 = color "rgb"(0.957 ,0.612, 0.157);
      uniform color C2= color "rgb" (0.588 ,0.345, 0.090);
      uniform color C3 = color "rgb" (0.820 ,0.514, 0.000);
      uniform color C4 = color "rgb" (1.000 ,0.902, 0.000);
      float txtscale = 1;
      float octaves = 6, omega = 0.35, lambda = 2;
      float threshold = 2;
      string BrickMap="";
      color albedo = color(0.369 ,0.200, 0.000);   // marble
      color dmfp = color(0.084 ,0.165, 0.000);   // marble
      float ior = 1.3;   // marble
      float unitlength = 1.0;   // modeling scale
      float smooth = 1;   // NEW for PRMan 14.0 !! (see sec. 4)
      float maxsolidangle = 0.2;   // quality knob: lower is better
      )
```

# Renderman Shaders

- Shading in it's strictest term refers to calculating the interactions of light with the surface
- This doesn't actually take into account the variations of properties such as colour and roughness across the surface
- This is the actual texturing part of the shader and this part allows the creation of the visual interest needed to create a realistic surface
- The simplest way to approach the texturing elements of shaders is to use the basic plastic model and then calculate an extra texturing element to be modulated with the plastic surface calculation

# templateShader.sl

```
1  surface standard
2  (
3   float Ka=1;
4   float Kd=0.5;
5   float Ks=0.5;
6   float roughness = 0.1;
7   color specularcolor = 1;
8  )
9
10 {
11 // init the shader values
12 normal Nf = faceforward(normalize(N),I);
13
14 vector V = -normalize(I);
15 .....
16 color Ct;
17
18 // here we do the texturing
19 Ct=Cs;
20 ......
21 // now calculate the shading values
22
23 Oi=Os;
24 Ci= Oi * (Ct * (Ka * ambient() + Kd *diffuse(Nf)) +
25     specularcolor * Ks  * specular(Nf,V,roughness));
26 }
```

Initial Shader values

Calculate Normal and View Vectors

Generate texture colours at this point

Shade the surface

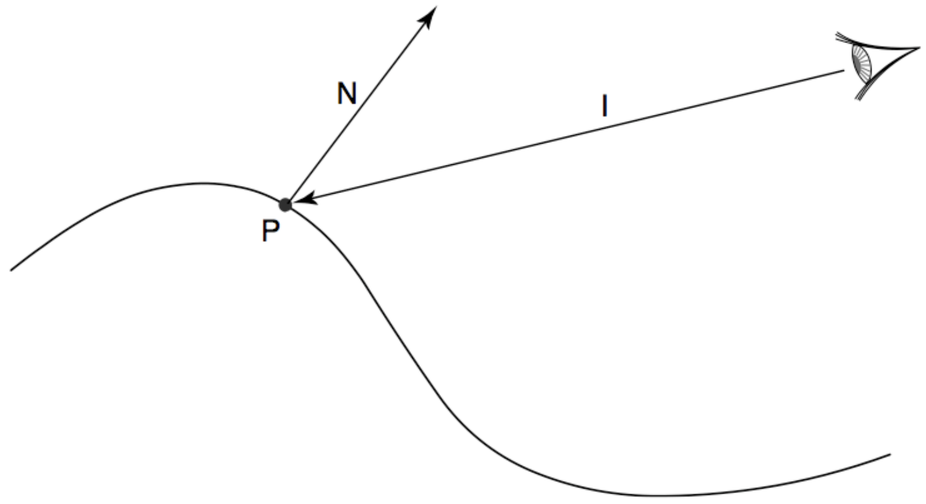
# templateShader.sl

- **Ka** The ambient reflection coefficient
- **Kd** The diffuse reflection coefficient
- **Ks** The specular reflection coefficient
- **roughness** The surface roughness used by the specular function
- **specularcolor** The colour of the specular highlight
- The shader is split into three section, the first calculates the forward facing normal Nf and the Viewing Vector V

# templateShader.sl

- The next section is configured for the texturing element of the shader, a Colour Ct is created and the calculation of this value will give us the surface texture at the point.
- The final section calculates the Opacity and the Colour for the current pixel.
- At present this uses the standard plastic model but elements may be changed to generate a different shading models based on any BRDF

# surface orientation




- The orientation of a surface at a point **P** is defined by the surface normal **N**
- The direction the surface is being viewed from is defined in the variable **I**
- The variable **N** tells us which way the surface is facing but in most cases when shading we don't really care so we can ensure the normal as to always be facing towards us.

# faceforward

```
1  normal faceforward( vector N, I [, Nref = Ng] )
2  {
3    return sign(-I.Nref) * N;
4  }
5
6
7  normal Nf = faceforward(normalize(N),I);
```

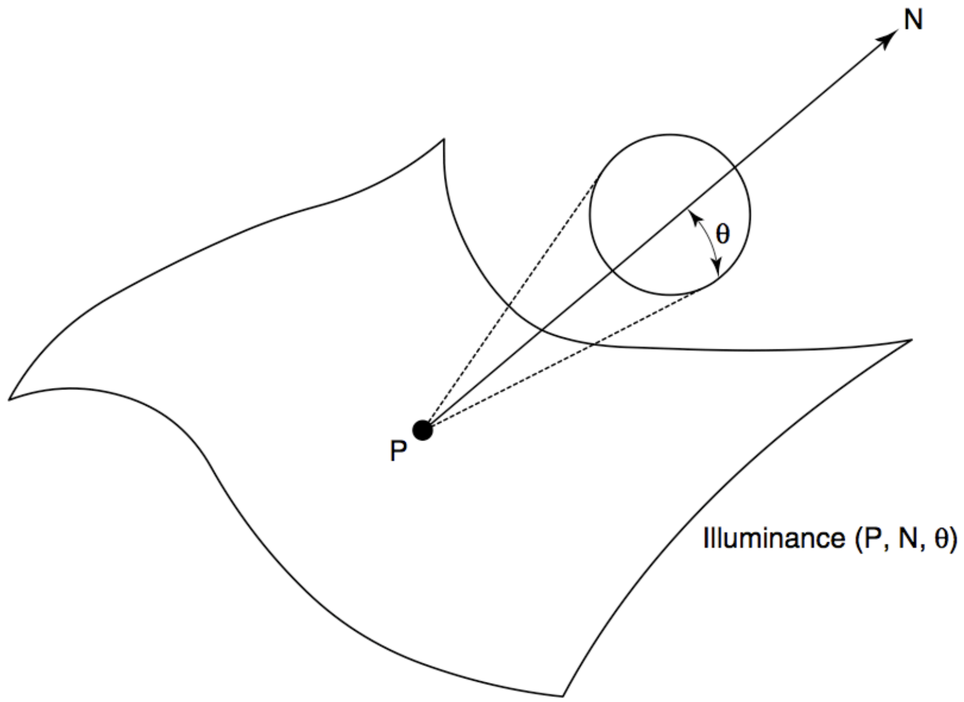
- Flip N so that it faces in the direction opposite to I, from the point of view of the current surface element.
- The surface element's point of view is the geometric normal Ng, unless Nref is supplied, in which case it is used instead.

# View Vector

- The direction towards the viewer is typically stored in a variable called V.
- As we have the vector I which gives the direction from the eye to the point on the surface we can simply calculate it by reversing and normalising this vector

```
1 vector V = -normalize(I);
```

- This vector is typically used in specular surface calculations




- Illuminance is a looping construct which loops over all the lights in the scene that are visible from the point being shaded.
- The 3 variables define a cone; any lights outside this cone are excluded.
- Within this loop the variables Cl and L are available to gather information about the light direction and colour

# diffuse surface calculations

- The diffuse(normal N) function returns the diffuse component of the lighting model. (Lambertian reflectance)
- N is a unit-length surface normal.
- Typically we multiply this value by Kd the diffuse co-efficient to scale the diffuse contribution of the surface.

```
1  color diffuse( normal N )
2  {
3    color C = 0;
4    illuminance( P, N, PI/2 )
5        C += Cl * normalize(L).N;
6    return C;
7  }
8
9  Ci =Kd *diffuse(Nf);
```

# ambient light

- ambient returns the total amount of ambient light incident upon the surface.
- An ambient light source is one in which there is no directional component, that is, a light that does not have an illuminate or a solar statement.
- Again we typically scale this value with an ambient coefficient Ka

```
1  Ci=Ka * ambient();
```

# specular

- Specular highlights can be generated using the specular function
- Renderman has its own built in specular function as well as access to a phong based model.
- Typically we scale the specular value using the coefficient Ks and supply an additional specular colour.

```
1 Ci=specularcolor * Ks   * specular(Nf,V,roughness);
```

```
1  color specular( normal N; vector V; float roughness )
2  {
3    color C = 0;
4    illuminance( P, N, PI/2 )
5        C += Cl * specularbrdf(normalize(L), N, V, roughness);
6    return C;
7  }
8
9  color specularbrdf(vector L, N, V; float roughness)
10 {
11     vector H = normalize(L+V);
12     return pow(max(0, N.H), 1/roughness);
13 }
```

- Returns the specular attenuation of light coming from the direction L, reflecting toward direction V, with surface normal N and roughness, roughness. All of L, V and N are assumed to be of unit length.

# Other BRDF models

- The previous model is very good for plastic and some metal type materials
- however other material properties are not well suited to this model.
- To model different surface materials we need to develop different BRDF models.

# Oren Nayar

- This model assumes rough surfaces to have microscopic grooves and hills.
- These are modelled mathematically as a collection of micro-facets having a statistical distribution of relative directions.
- This allows us to model materials such as Clay, stone etc.

# Oren Nayar

$$L_r(\ \ r,\ \ i, \blacksquare - \blacksquare, \blacksquare) = \frac{\rho}{\pi} E_0 \cos\ \ _i (A + B \max(0, \cos(\blacksquare - \blacksquare)) \sin\ \ \ \tan\ \ \ )$$

- where

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = \max(\theta_i, \theta_r)$$

$$\beta = \min(\theta_i, \theta_r)$$

$\rho$ is the reflectivity of the surface (Kd*Cs)

$E_o$ is the energy arriving from the light CI

$\theta_i$ is the angle between the surface normal and the direction of the light source

$\theta_r$ is the angle between the surface normal and the vector in the direction of the viewer

$\phi_r - \phi_i$ is the angle (about the normal) between incoming and reflected light

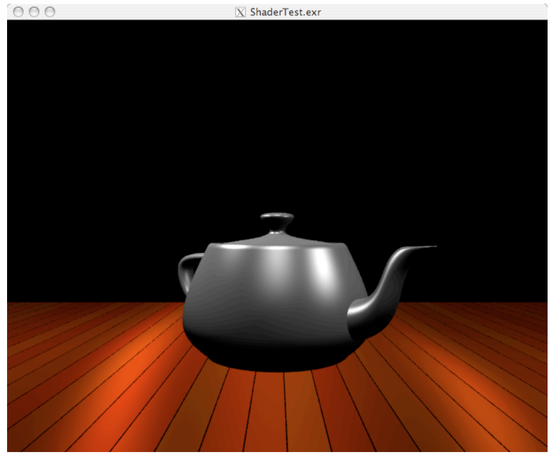
# Oren Nayar

$\sigma$ is the standard deviation of the angle distribution of the microfacets (in radians). Larger values represent more rough surfaces; smaller values represent smoother surfaces.

If $\sigma = 0$ ,the surface is perfectly smooth, and this function reduces to a simple Lambertian reflectance model.

This parameter is called “roughness”

```
color
LocIllumOrenNayar (normal N;  vector V;  float roughness;)
{
    // Surface roughness coefficients for Oren/Nayar's formula
    float sigma2 = roughness * roughness;
    float A = 1 - 0.5 * sigma2 / (sigma2 + 0.33);
    float B = 0.45 * sigma2 / (sigma2 + 0.09);
    // Useful precomputed quantities
    float  theta_r = acos (V . N);          // Angle between V and N
    vector V_perp_N = normalize(V-N*(V.N)); // Part of V perpendicular to N

    // Accumulate incoming radiance from lights in C
    color  C = 0;
    extern point P;
    illuminance (P, N, PI/2)
    {
    vector LN = normalize(L);
    float cos_theta_i = LN . N;
    float cos_phi_diff = V_perp_N . normalize(LN - N*cos_theta_i);
    float theta_i = acos (cos_theta_i);
    float alpha = max (theta_i, theta_r);
    float beta = min (theta_i, theta_r);
    C += 1 * Cl * cos_theta_i * (A + B * max(0,cos_phi_diff) * sin(alpha) *
        tan(beta));
    }
    return C;
}
```




# Ward Anistropic

The Ward anistropic model describes a surface similar to brushed metal where the machined grooves face in a particular direction.

$$\frac{1}{\sqrt{\cos\theta_i \cos\theta_r}} \frac{1}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{\hat{h}\cdot\hat{x}}{\alpha_x}\right)^2 + \left(\frac{\hat{h}\cdot\hat{x}}{\alpha_y}\right)^2}{1+\hat{h}\cdot\hat{n}}\right]$$

- $\theta_i$ is the angle between the surface normal and the direction of the light source
- $\theta_r$ is the angle between the surface normal and the direction of the viewer

# Ward Anistropic

$\hat{x}$ and $\hat{y}$ are the two perpendicular tangent directions of the surface

$\alpha_x$ and $\alpha_y$ are the standard deviations of the slope in the x and y directions called x and y roughness

$\hat{n}$ is the unit surface normal normalize(N)

$\hat{h}$ is the half angle between incident and reflection rays

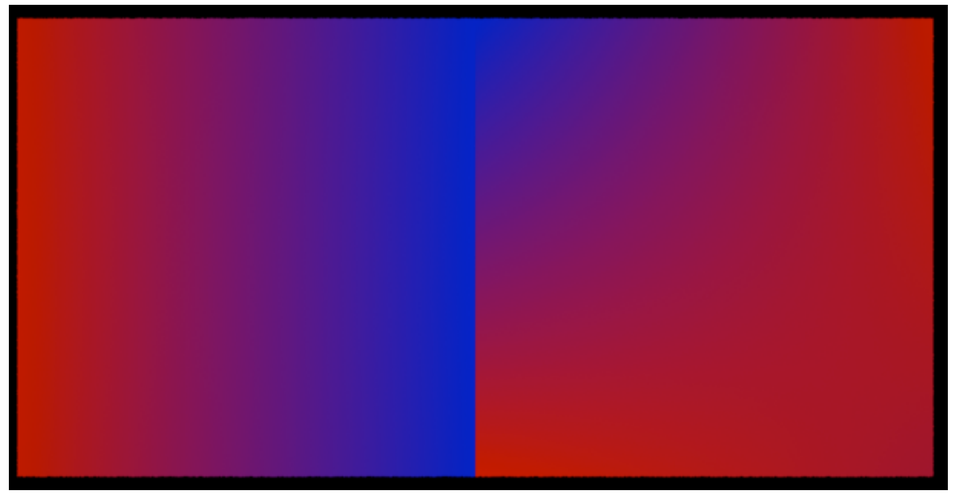
H=normalize(normalize(-I)+normalize(L))

```
float sqr (float x) { return x*x; }

color LocIllumWardAnisotropic (
        normal N;  vector V;
                vector xdir;
                float xroughness;
                float yroughness;
        )
{

float cos_theta_r = clamp (N.V, 0.0001, 1);
vector X = xdir / xroughness;
vector Y = (N ^ xdir) / yroughness;

color C = 0;
illuminance (P, N, PI/2)
  {
  vector LN = normalize (L);
  float cos_theta_i = LN . N;
  if (cos_theta_i > 0.0)
    {
    vector H = normalize (V + LN);
    float rho = exp (-2 * (sqr(X.H) + sqr(Y.H)) / (1 + H.N))
    / sqrt (cos_theta_i * cos_theta_r);
    C += Cl * (  cos_theta_i * rho);
    }
  }
return C / (4 * xroughness * yroughness);
}
```

# Creating a texture

- To calculate the texture information at each point on the surface we need to know where we are on the surface
- This information is provided by two variables u and v which tell how far across and up the surface the current point is
- • These values range from 0 to 1 in both directions (0 bottom left) (1 top right)
- This works on most Renderman geometry types as they are based on patches (4 corners)

# s,t and u,v



- To access the texture co-ordinates in a shader the programmer has a choice of either s,t or u,v
- by default these are identical, however these may be overridden by the modeller as follows

```
1  Patch "bilinear"
2        "P" [-1 1 0  1 1 0  -1 -1 0  1 -1 0  ]
3        "st" [1 0.5 0 1 0 .5 .2 0]
```

- Within the shader if the variables u,v are used this will refer to the underlying geometry (Renderman)
- if s,t are used these will be the user supplied (if any) values

# Using s,t to create a Colour ramp

- A simple ramp shader is a simple as using the following expression

```
1  Ct = s; or Ct = t;
```

- or for Colour

```
1  Ct =s*Cs; or Ct=t*Cs;
```

- for different colour space we could do the following

```
1  Ct = color "hsv" (s,t,1);
```

# Colour Blends

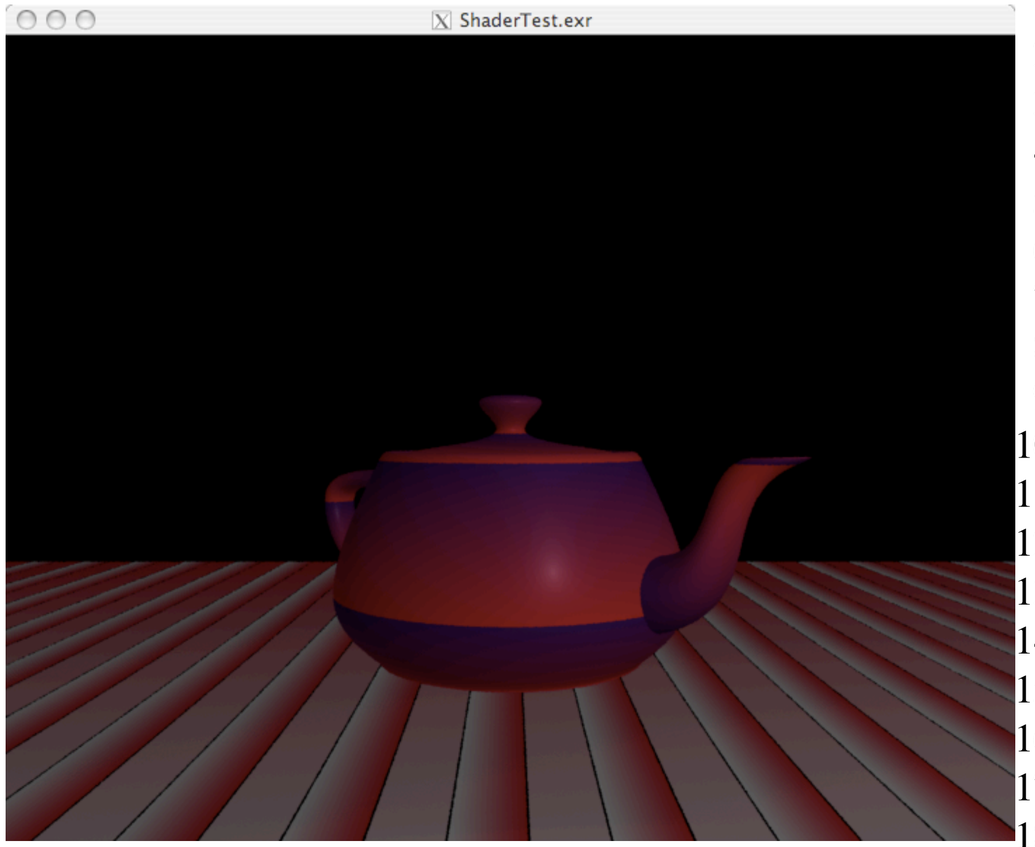
- If we wish to blend two colours we can use simple linear interpolation between the current colour (Cs) and another user defined colour BlendColour
- This can be done in two ways

```
1 color BlendColour = color "rgb" (0,1,0);
2 Ct = (1-s)*Cs+s*BlendColour;
```

- or using the built in mix function

```
1 Ct=mix(Cs,BlendColour,s);
```

# Putting it all together




```
surface RampShader(
 color BlendColor = color "rgb" (0,1,0);
 float orientation = 0;
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);


color Ct;

// here we do the texturing
if(orientation ==0)
        Ct=mix(Cs,BlendColor,s);
else
        Ct=mix(Cs,BlendColor,t);
// now calculate the shading values
Oi=Os;
Ci= Oi * (Ct * (Ka * ambient() + Kd *diffuse(Nf)) +
    specularcolor * Ks  * specular(Nf,V,roughness));
}
```

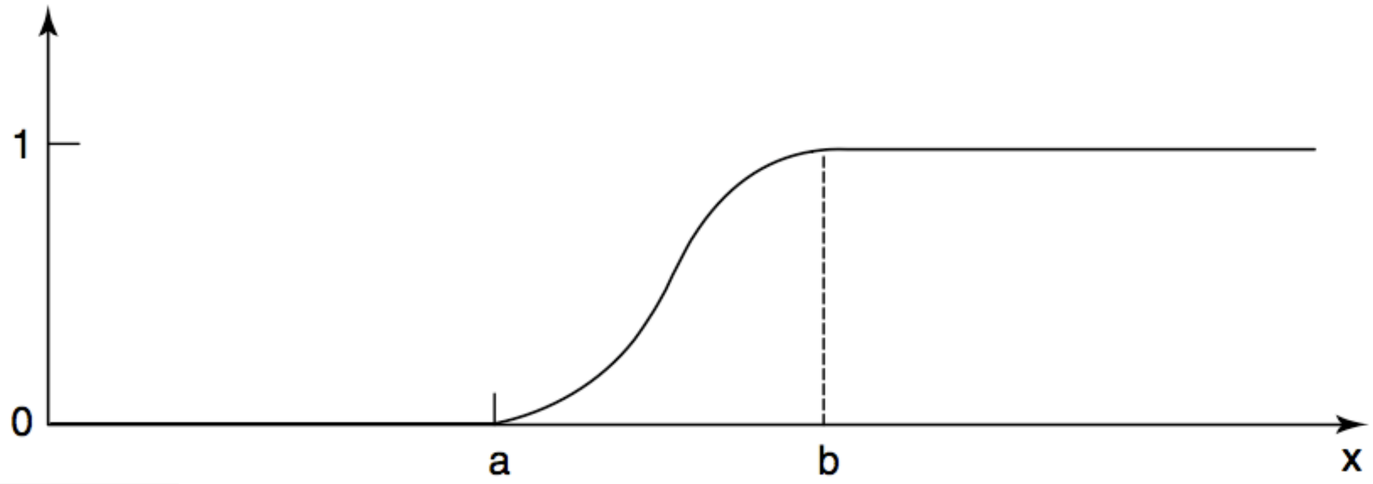
# Bands

- The simplest pattern to create in a shader is a simple Band between two colours

```
1  color red=color "rgb" (1,0,0);
2  color blue=color "rgb" (0,1,0);
3  if(t<0.5)
4    Ct=red;
5  else
6    Ct=blue;
```

- This works but can cause problems as the line is very sharp and could cause aliasing
- The easiest way is not to use sharp transitions but use a smooth blend between colour values

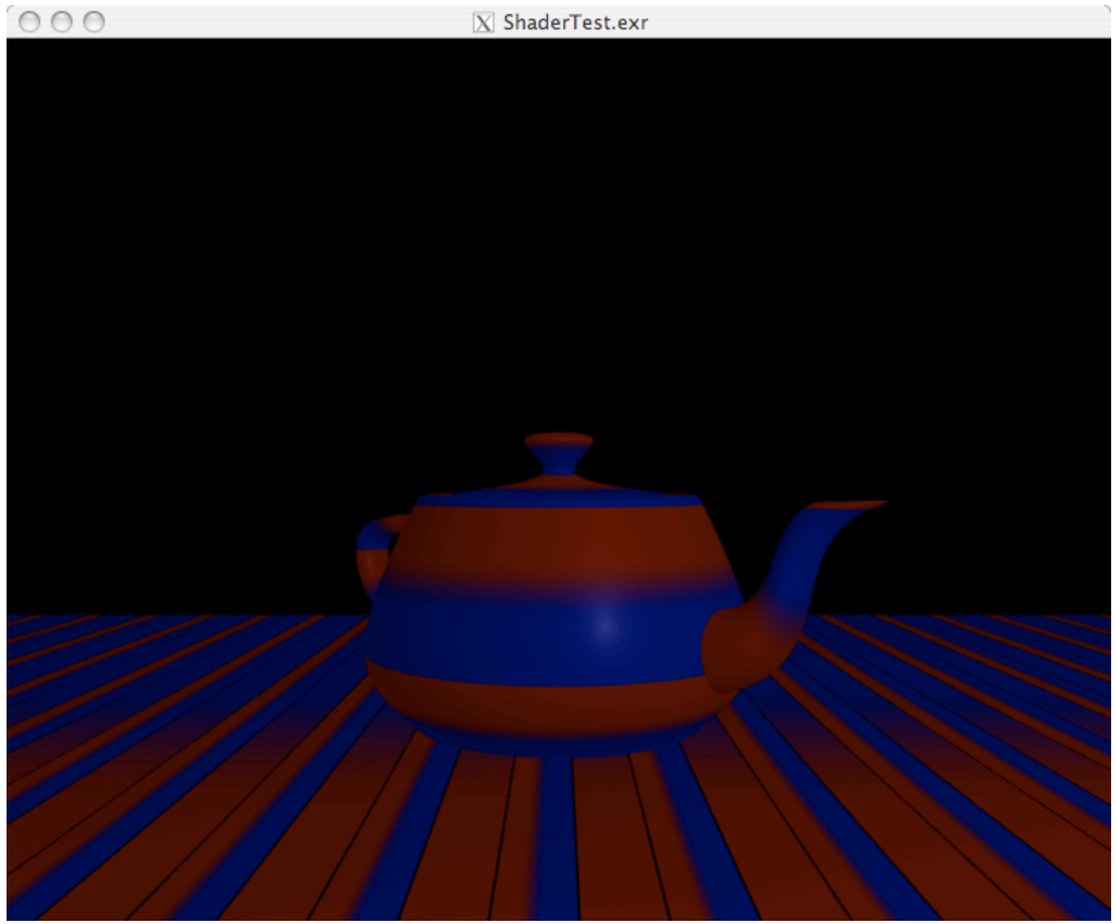
# Smoothstep




```
1 float smoothstep( float min, max, value )
2 float smoothstep( color min, max, value )
```

- smoothstep returns 0 if value is less than min, 1 if value is greater than or equal to max, and performs a smooth Hermite interpolation between 0 and 1 in the interval min to max.
- When using versions of these operators that work on colour arguments, the operation is applied on a component by component basis.

# Using smoothstep to create fuzzy lines



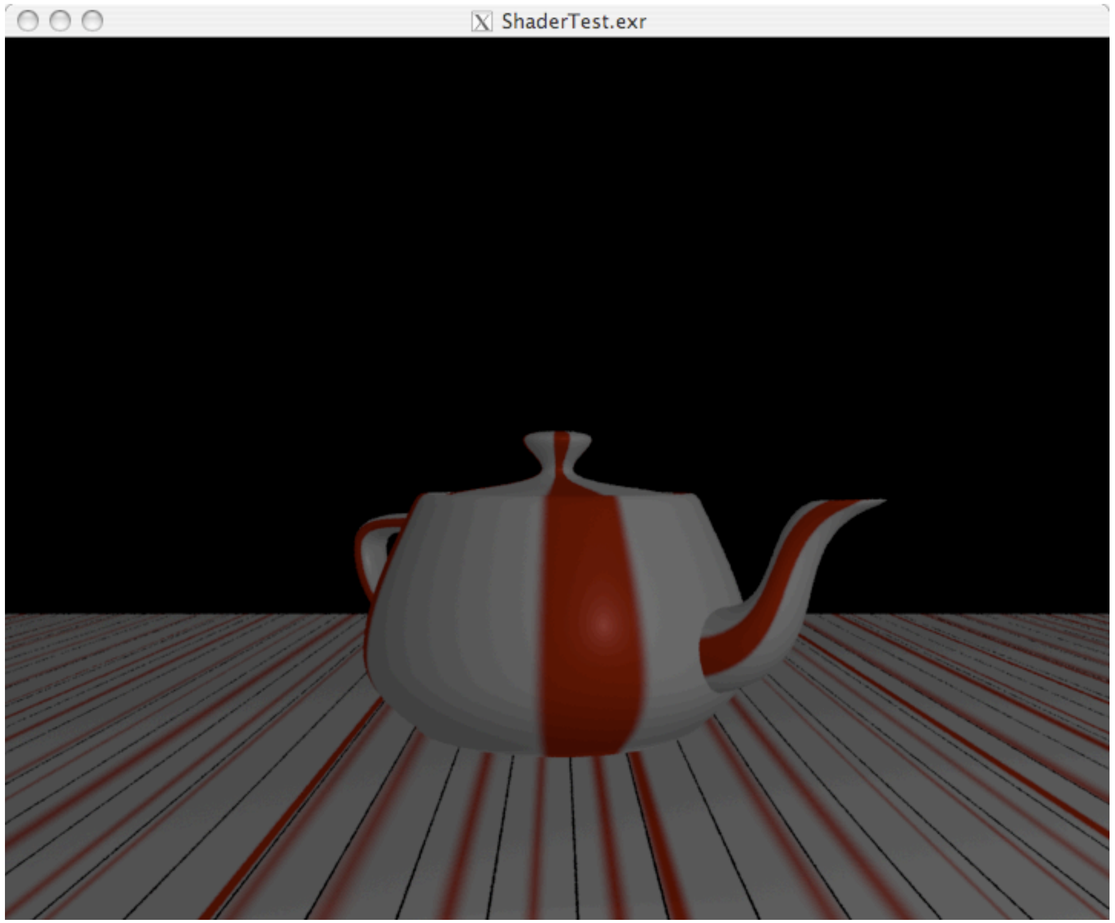

```
surface Band(
 color C1 = color "rgb" (1,0,0);
 color C2 = color "rgb" (0,0,1);
 float begin=0.3;
 float end=0.6;
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 float Orient=0;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
color Ct;
float inTop;
if (Orient==0)
  inTop = smoothstep(begin,end,s);
else
  inTop = smoothstep(begin,end,t);

Ct=mix(C1,C2,inTop);
// now calculate the shading values
Oi=Os;
Ci= Oi * (Ct * (Ka * ambient() + Kd *diffuse(Nf)) +
    specularcolor * Ks  * specular(Nf,V,roughness));
}
```

# Lines

- if you want to produce a vertical line in the centre of the object you first need to find the distance of the point from the centre

```
1 float dist = abs(s-0.5)
```

- We use the function abs to throw away the +/- sign as we don't care which side of the line were on
- If we wanted the line to be 0.2 wide then if dist is less than 0.1 we are considering a point on the line
- The following shader allows for the line to be of varying thickness and orientation




```
surface Lines
(
 color LineColor = color "rgb" (1,0,0);
 color MixColor = color "rgb" (1,1,1);
 float fuzz = 0.025;
 float LineSize=0.1;
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 float Orient=0;
 float offset=0.5;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);
color Ct;

// here we do the texturing
float inTop;
float dist;
if(Orient==0)
  dist=abs(t-offset);
else
  dist=abs(s-offset);
float inLine;

inTop=1-smoothstep(LineSize/2.0-fuzz,LineSize/2.0+fuzz,dist);
Ct=mix(MixColor,LineColor,inTop);


// now calculate the shading values

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

}
```

# User Specified Lines

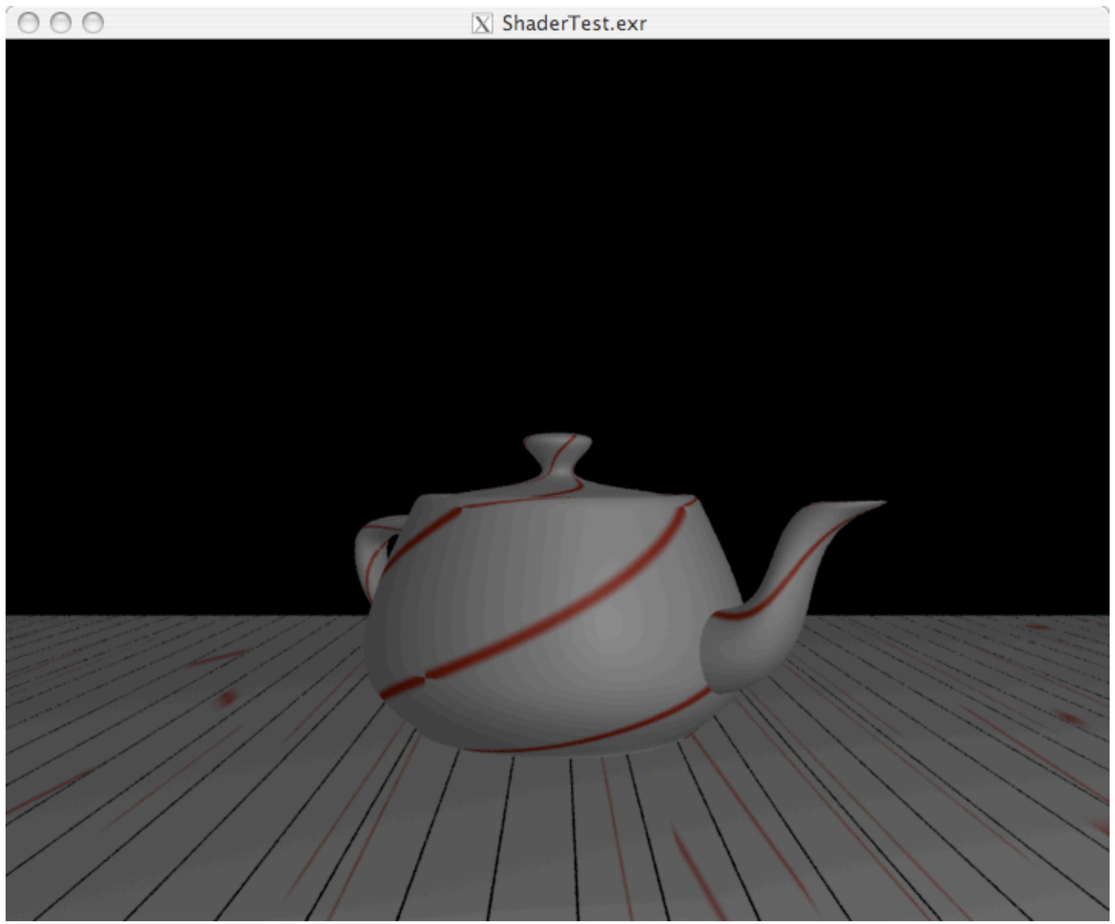
- To define an arbitrary line we need to specify the start and end positions

```
1  float ptlined( point  P1, P2, Q )
```

- To calculate this we use the following

- Returns the minimum perpendicular distance between the point Q and the line segment that passes from the point P1 to the point P2 (not the infinite line which passes through P1 and P2)

```
1  point Here = point(s,t,0);
2
3  float dist=ptlined(StartPos,EndPos,Here);
4  float inLine = smoothstep(0.1-fuzz,0.1+fuzz,dist);
5
6  Ct=mix(Cs,LineColor,inLine);
```




```
surface UserLines
(
 color LineColor = color "rgb" (1,0,0);
 float LineSize=0.01;
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.5;
 color specularcolor = 1;
 point P1= point "shader" (0.1,0.7,0);
 point P2= point "shader" (0.7,0.7,0);
 float fuzz = 0.025;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
color Ct=Cs;
// we need this point in shader space so it is on the surface of
    the
// object we are shading
point Here = point "shader" (s,t,0);
float dist=ptlined(P1,P2,Here);
float inLine = 1-smoothstep(LineSize/2.0-fuzz,LineSize/2.0+fuzz,
   dist);
Ct=mix(Ct,LineColor,inLine);

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

}
```

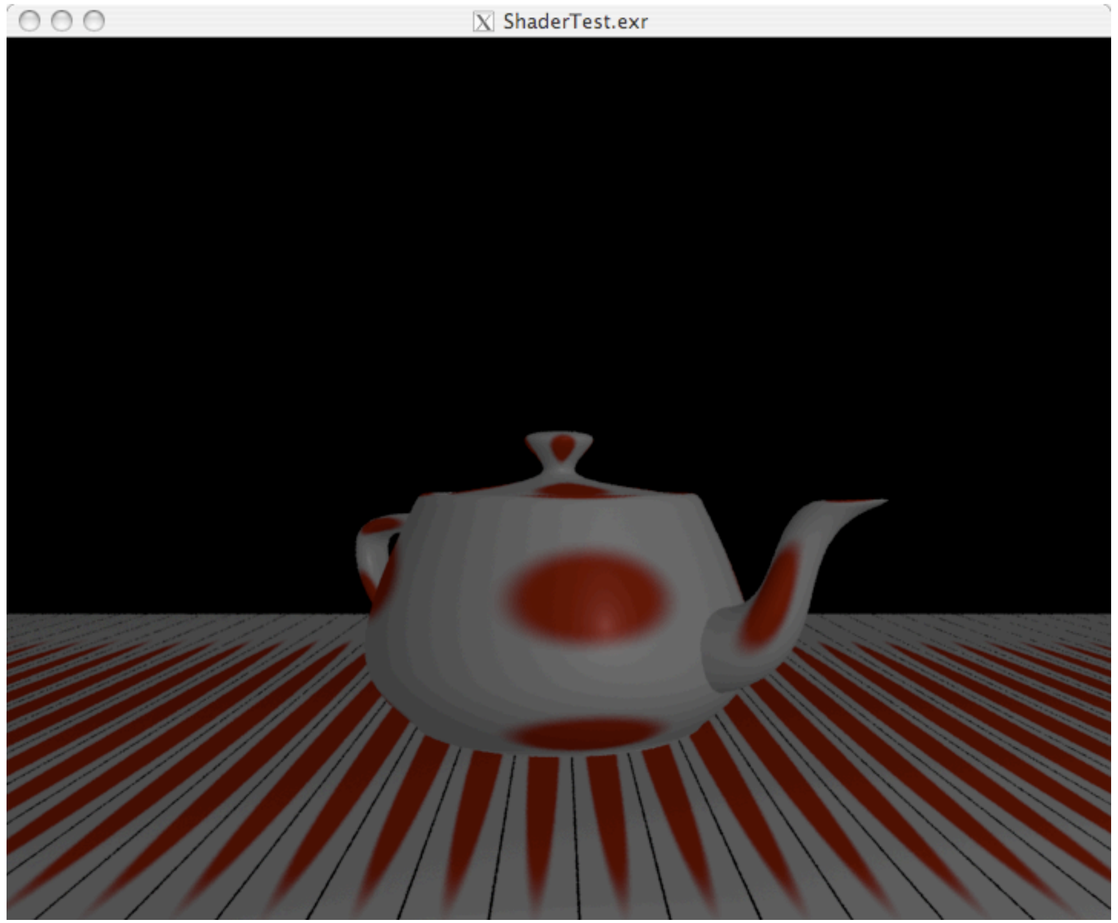
# Disks

- To create a coloured disk we can use decide if the point is in or outside the disk by using Pythagorus

```
1 float dist = sqrt((s-0.5)*(s-0,5) *(t-0.5)*(t-0.5));
```

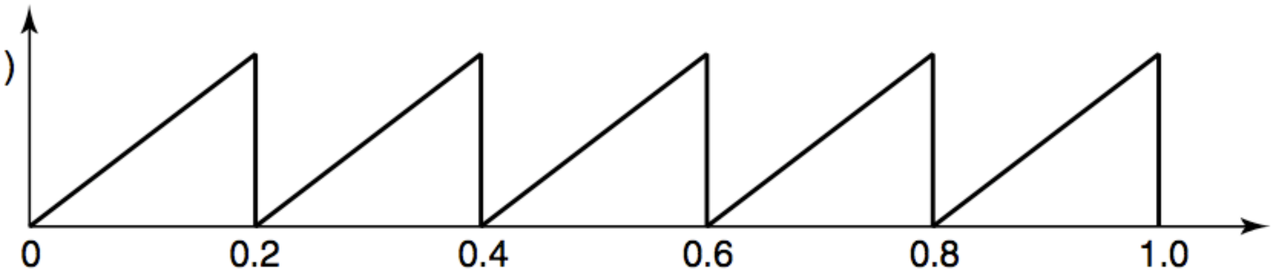
- an alternative to using this is the distance function as shown in the following example

```
1 color Ct=Cs;
2 point here = point(s,t,0);
3
4 float dist=distance(center,here);
5 float inDisk=1-smoothstep(0.3-fuzz,0.3+fuzz,dist);
6
7 Ct=mix(Ct,DiskColour,inDisk);
```




```
surface Disk
(
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 color DiskColour = color "rgb" (1,1,1);
 point center = point "shader" (0.5,0.5,0.0);
 float fuzz=0.025;
 float Radius = 0.5;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
color Ct=Cs;
point here = point "shader" (s,t,0);
float dist=distance(center,here);
float inDisk=1-smoothstep(Radius/2.0-fuzz,Radius/2.0+fuzz,dist);
Ct=mix(Ct,DiskColour,inDisk);
// now calculate the shading values

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

}
```

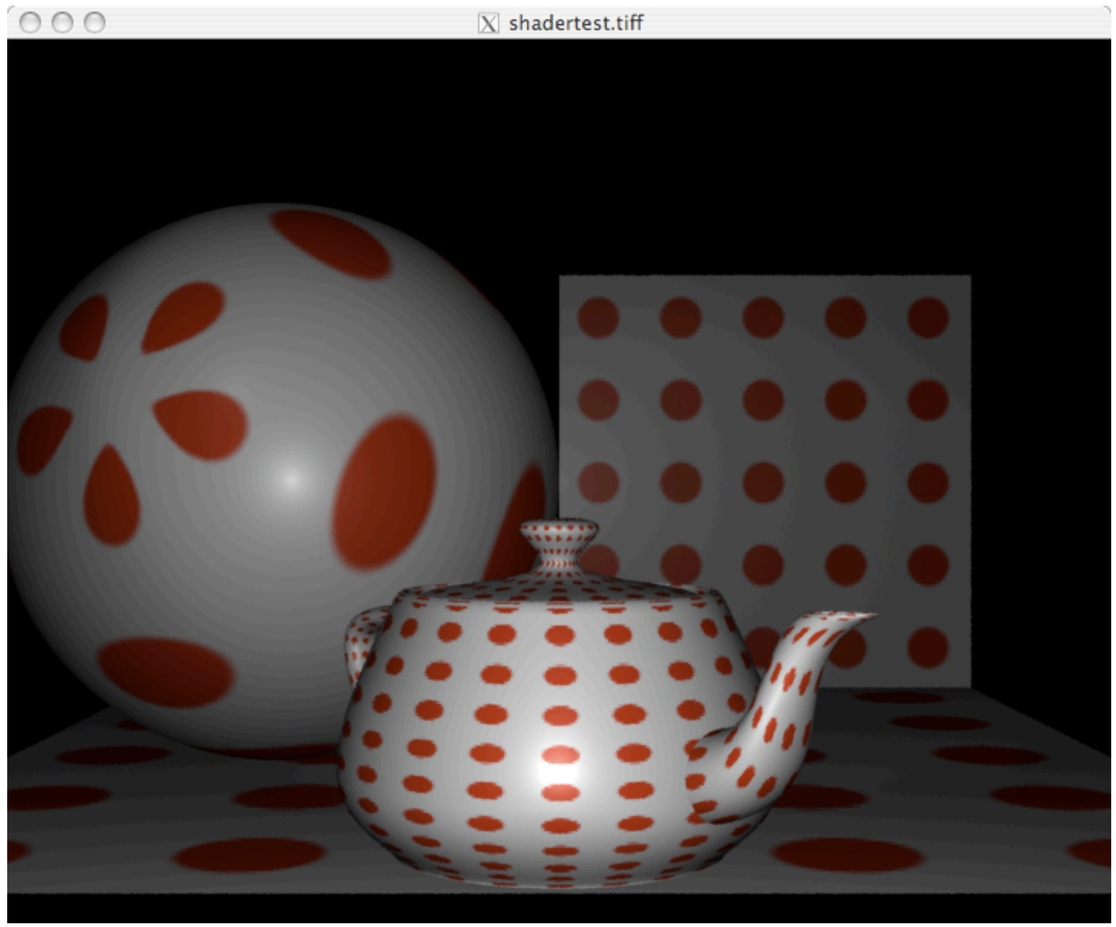
# Repeating things




- A very simple way to repeat patterns is by dividing up the s and t values of the section being textured.
- To do this we can use the mod function as follows

```
1  float ss=mod(s*RepeatS,1);
2
3  float tt=mod(t*RepeatT,1);
```

- Then any calculation which used the values of s or t now use the ss or tt versions




```
surface Disk
(
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 color DiskColour = color "rgb" (1,1,1);
 point center = point "shader" (0.5,0.5,0.0);
 float fuzz=0.025;
 float Radius = 0.5;
 float RepeatS=5;
 float RepeatT=5;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
color Ct=Cs;
float ss=mod(s*RepeatS,1);
float tt=mod(t*RepeatT,1);

point here = point "shader" (ss,tt,0);
float dist=distance(center,here);
float inDisk=1-smoothstep(Radius/2.0-fuzz,Radius/2.0+fuzz,dist);
Ct=mix(Ct,DiskColour,inDisk);
// now calculate the shading values

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

}
```

# References


- [1] Ian Stephenson. Essential Renderman Fast. 2nd Edition. Springer-Verlag, 2007.
- [2] Larry Gritz Anthony A Apodaca. Advanced Renderman (Creating CGI for Motion Pictures). Morgan Kaufmann, 2000.
- S.K. Nayar and M. Oren, "Generalization of the Lambertian Model and Implications for Machine Vision". International Journal on Computer Vision, Vol. 14, No. 3, pp. 227-251, Apr, 1995

# Further reading

- http://en.wikipedia.org/wiki/Oren%E2%80%93Nayar_diffuse_model
- RSL Function in the renderman documentation
- Application Note #17 Converting Shaders to use new Shading Language Features
- New Shading Language Features (2008) Renderman Documents