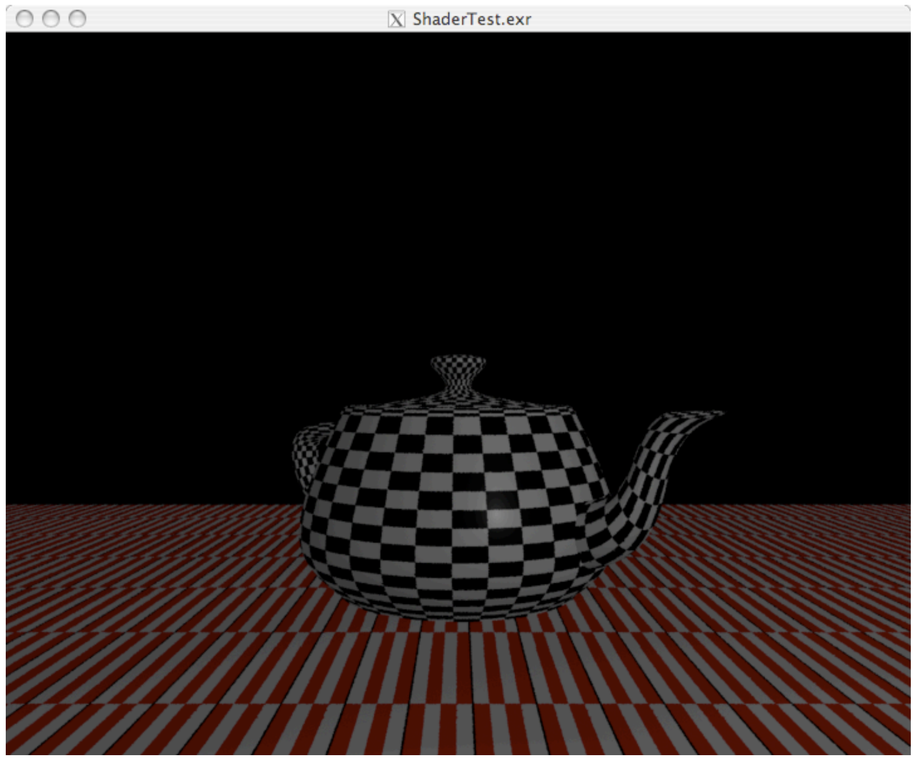
Renderman Shaders 2

More Patterns, Noise Displacement Textures

Identifying Tiles

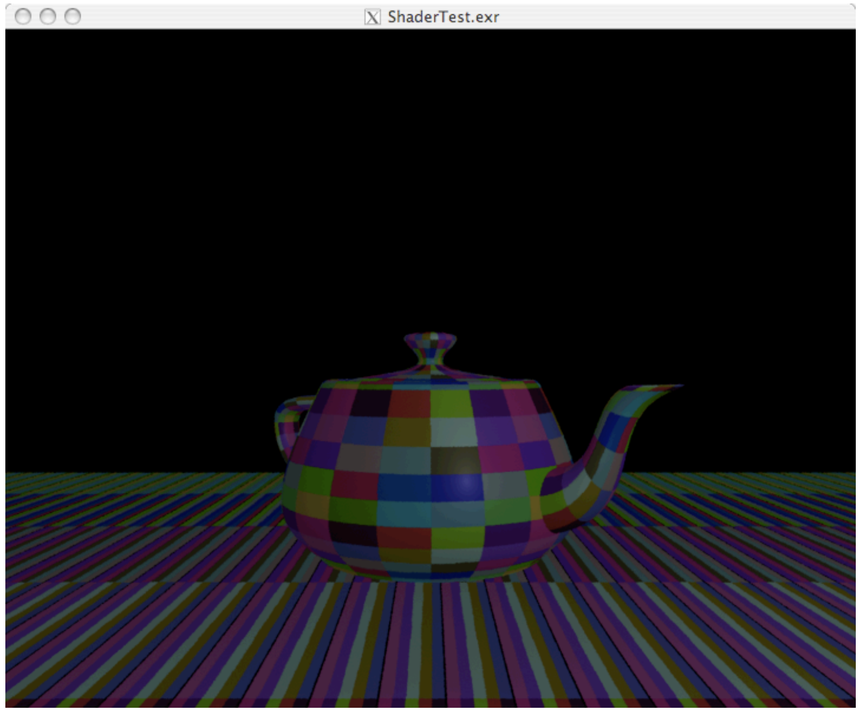
- To make a pattern more interesting we probably want to modify the basic motif so that it is slightly different in each cell.
- To do this you need to know which cell we are in as well as the position in the cell.
- Previously we used the mod() function to obtain the fractional part of a number we can use floor() to obtain the integer part and use this to signify if we are in a particular region.
- If we apply this to both the S and T directions we can generate basic check patterns




```
surface Check
(
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 color CheckColour = color "rgb" (0,0,0);
 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 sTile=floor(s*RepeatS);
float tTile=floor(t*RepeatT);
float inCheck=mod(sTile+tTile,2);
Ct=mix(Ct,CheckColour,inCheck);
// now calculate the shading values

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

}
```

Cell Noise

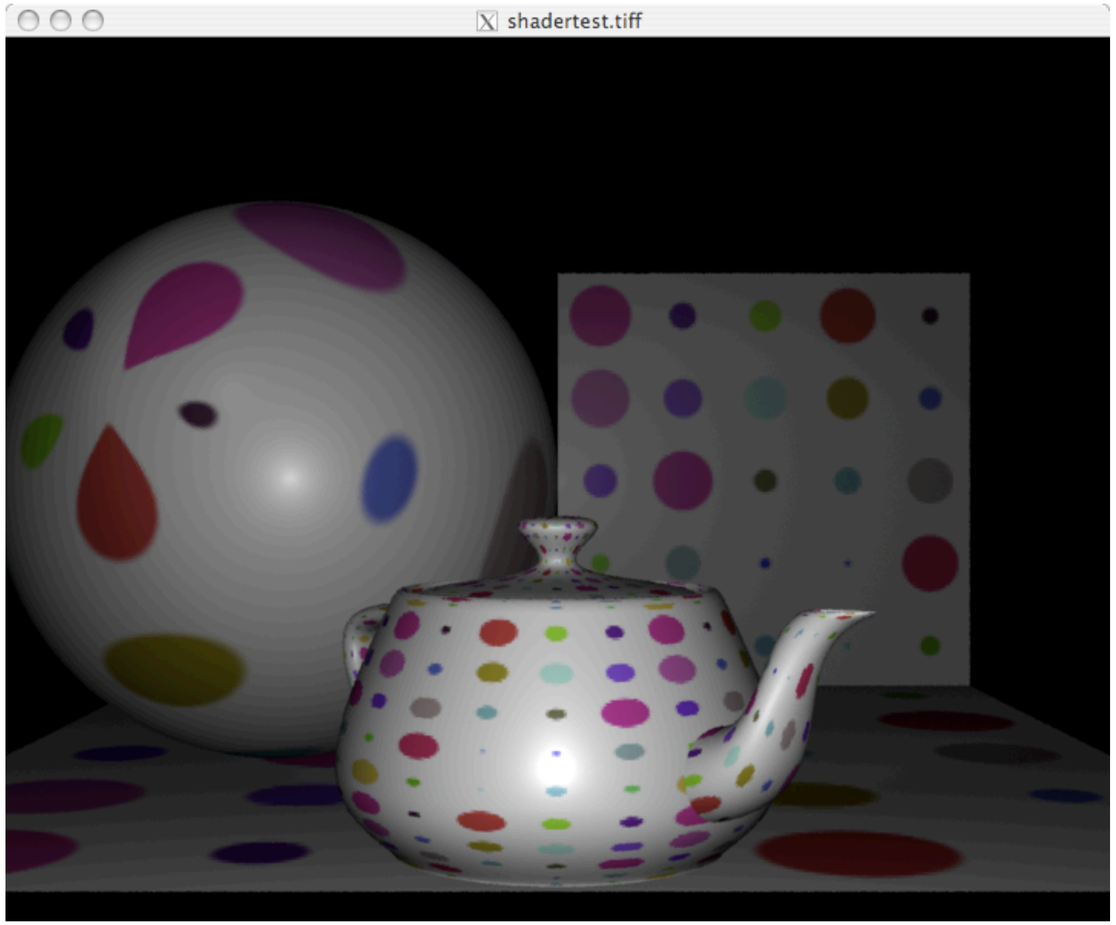
- cellnoise returns a value that is a pseudorandom function of its arguments.
- Its return value is uniformly distributed between 0 and 1, has constant value between integer lattice points, and is discontinuous at integer locations.
- This is useful if you are dividing space into regions ("cells") and want a different (repeatable) random number for each region.
- It is considerably cheaper than calling noise, and thus is preferable if you have been using noise simply to generate repeatable random sequences. The type desired is indicated by casting the function to the type desired




```
surface CellNoise
(
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 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=s*RepeatS;
float tt=t*RepeatT;
Ct=color cellnoise(ss,tt);
// now calculate the shading values

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

}
```

Cell Noise

- Cell noise is an overloaded function and can return :-
  - single float (0 -1)
  - a colour value
  - vector
- This is shown in the following example




```
surface RandomSpots
(
 float Ka=1;
 float Kd=0.5;
 float Ks=0.5;
 float roughness = 0.1;
 color specularcolor = 1;
 float RepeatS=5;
 float RepeatT=5;
 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;
point Center=point "shader" (0.5,0.5,0);
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 Radius = float cellnoise(RepeatS*s,RepeatT*t)*0.4;
color DiskColour=color cellnoise(RepeatS*s,RepeatT*t);
float inDisk=1-smoothstep(Radius-fuzz,Radius+fuzz,dist);
Ct=mix(Cs,DiskColour,inDisk);
// now calculate the shading values

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

}
```

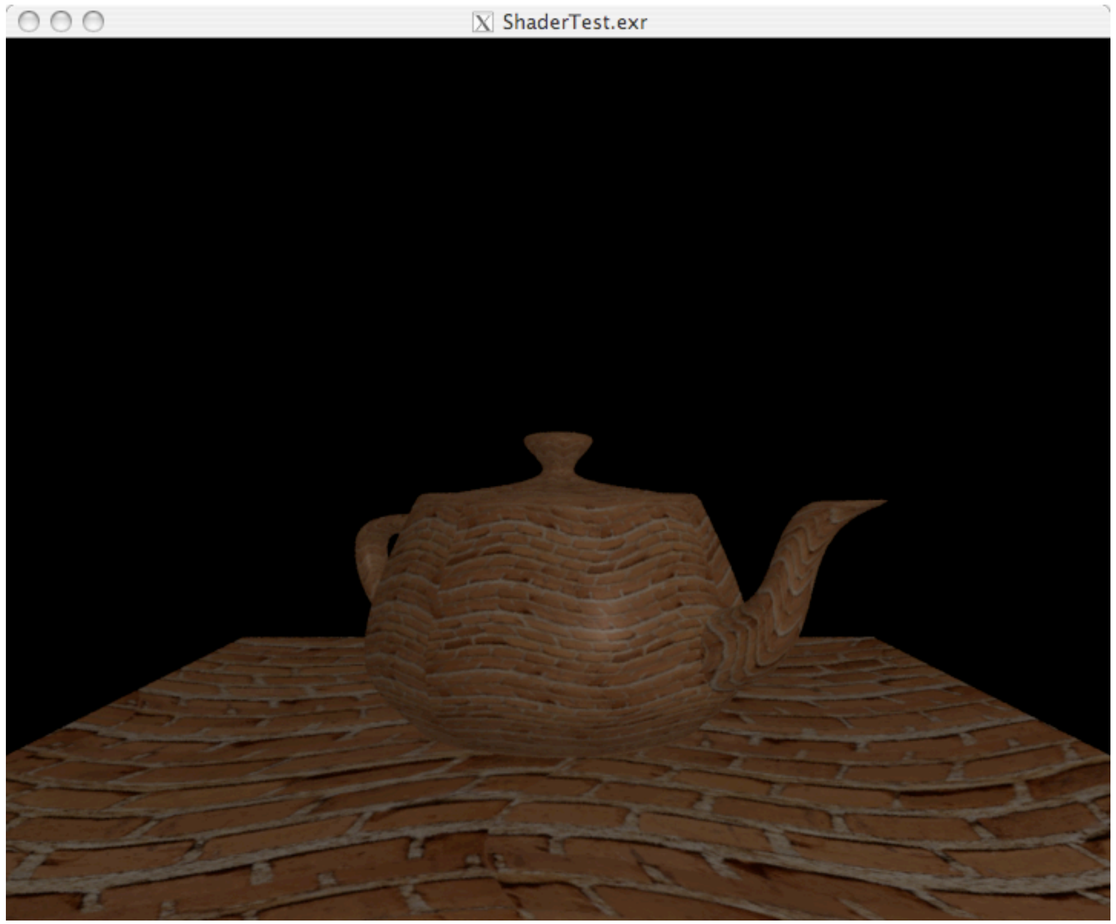
Accessing Textures

- The simplest way of accessing image textures is with the use of the texture function

```
1  Ct=colour texture("filename");
```

- This will then grab the current texture element from the file passed in and set the current colour from the file
- The texture function will be automatically wrapped using the s,t texture co-ordinates
- If we wish to use the model s,t values we need to pass these to the shader


```
surface SimpleTexture
(
 float Ka=1;float Kd=0.5;
 float Ks=0.5; float roughness = 0.1;
 color specularcolor = 1;
 string TextureName="";
 float RepeatS=1; float RepeatT=1;
)

{

// 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,1.0);
float tt=mod(t*RepeatT,1.0);
// set Ct to be the base colour if we have a texture file
// then change the value
Ct=Cs;
if(TextureName !="")
        Ct=texture(TextureName,ss,tt);
/* now calculate the shading values */
Oi=Os;

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



Modulating a texture

```
1  float ss=mod(s*2+0.1*sin(t*RepeatS),1);
2  float tt=mod(t*2+0.1*sin(s*RepeatT),1);
```

- By changing the s,t values we can warp the texture

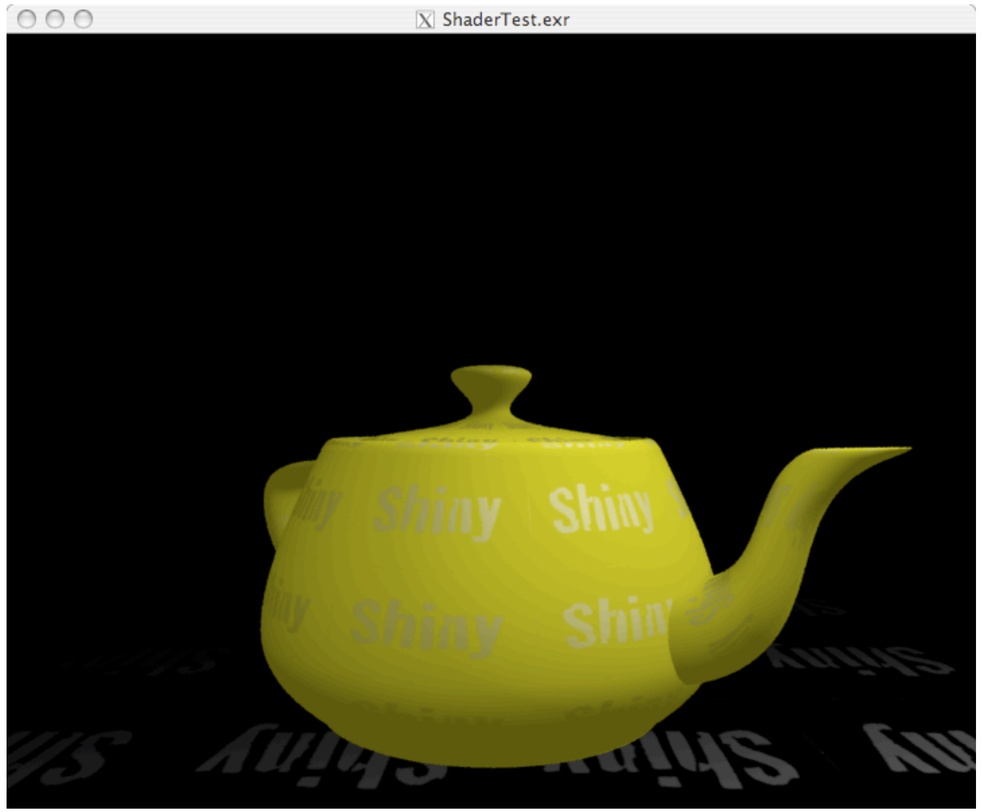
Mixing Textures

- We can use cellnoise to decide which texture to load
- This allows use to create image based textures which should never repeat.

```
surface MixTexture
(
 float Ka=1;float Kd=0.5;
 float Ks=0.5; float roughness = 0.1;
 color specularcolor = 1;
 string tex1="";
 string tex2="";
 string tex3="";
 float RepeatS=1; float RepeatT=1;
 float range1=0.7;
 float range2=0.8;
)
{
// 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,1.0);
float tt=mod(t*RepeatT,1.0);
Ct=Cs;
float which= float cellnoise(s*RepeatS,t*RepeatT);

if (which <=range1)
  Ct=texture(tex1,ss,tt);
else if(which <range2)  Ct=texture(tex2,ss,tt);
else Ct =texture(tex3,ss,tt);

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

Maps to control procedural textures

- We can use textures as maps to control certain elements of a surface.
- For example we can use a painted texture to specify which elements of the surface have specular highlights
- Whilst other elements are rendered as a matte surface
- This is shown in the following shader




```
1  surface SpecularImageTexture
2  (
3   float Ka=1;
4   float roughness = 0.1;
5   string TextureName="";
6   float RepeatS=1; float RepeatT=1;
7   color specularColour=1;
8  )
9  {
10 // init the shader values
11 normal Nf = faceforward(normalize(N),I);
12 vector V = -normalize(I);
13
14 // here we do the texturing
15
16 color Ct=Cs;
17
18 float ss=mod(s*RepeatS,1.0);
19 float tt=mod(t*RepeatT,1.0);
20 // set Ct to be the base colour if we have a texture file
21 // then change the value
22 Ct=Cs;
23 float SurfaceType;
24 if(TextureName !="")
25         SurfaceType = float texture(TextureName,1-ss,tt);
26 // now calculate the shading values
27 Oi=Os;
28
29 Ci= Oi * (Ct * (Ka * ambient() + SurfaceType *diffuse(Nf)) +
30     (1-SurfaceType) *specularColour * specular(Nf,V,roughness));
31 }
```

Environment Maps

- Environment maps are used to simulate the environment an object is placed in.
- They can be generated in CG using different views of the scene
- Or alternatively using photographs of a real life scene in which the object is to be placed

Environment Maps

- Environment maps are usually generated using a simple cube projection from the environment they are created in
- In this case I didn't have a wide enough angle lens to get the whole environment but it will do for illustration

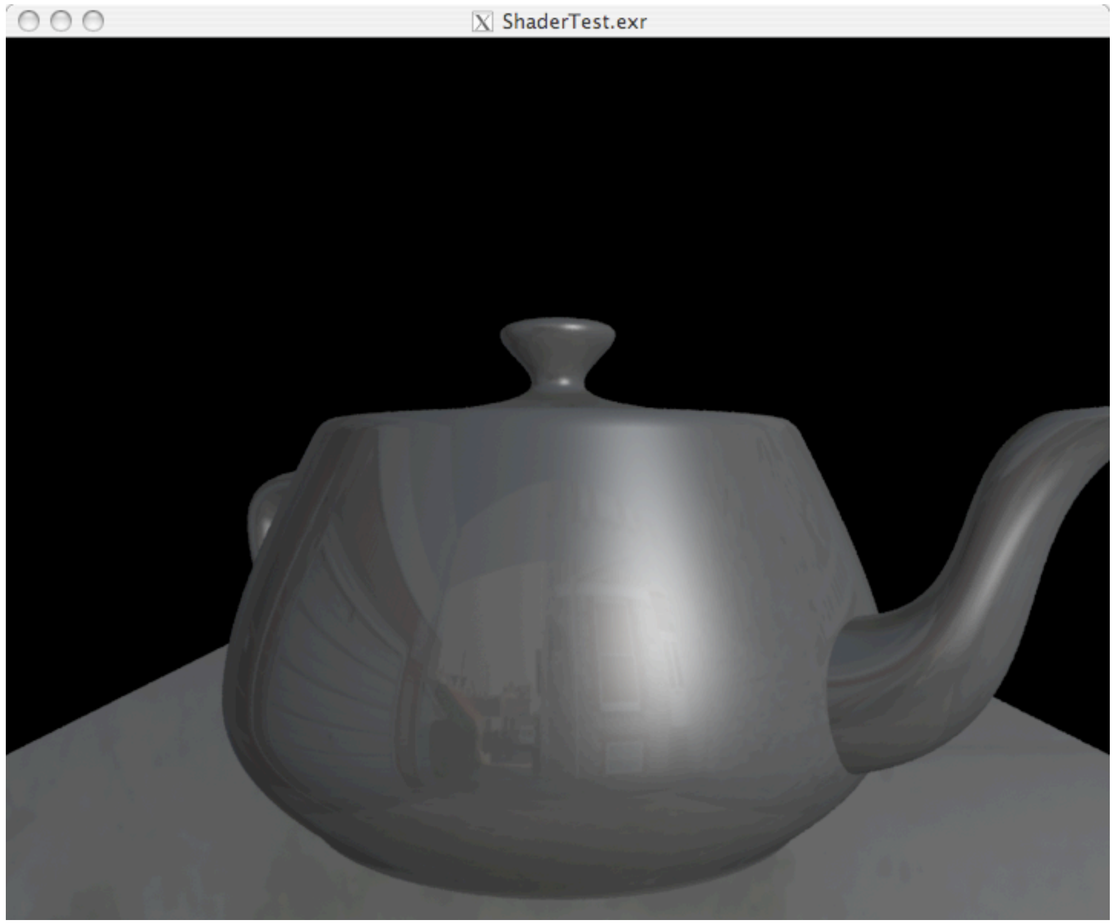
Making Cube Maps

```
1 txmake -envcube -fov 93 -filter gaussian -sfilterwidth 2
2 -tfilterwidth 2 -resize  down  px.tiff nx.tiff py.tiff ny.tiff
      pz.tiff nz.tiff map.tx
```

- In the cube face case, the -fov angle option can be used to specify a field-of-view angle in degrees, if the default field of view of 90 degrees was not used. The field of view may vary from 90 to 179 degrees.
- In the case of a cube-face environment map, selected with the -envcube option, the input consists of a sequence of six image names corresponding to six cube face views in the order: px, nx, py, ny, pz, and nz.

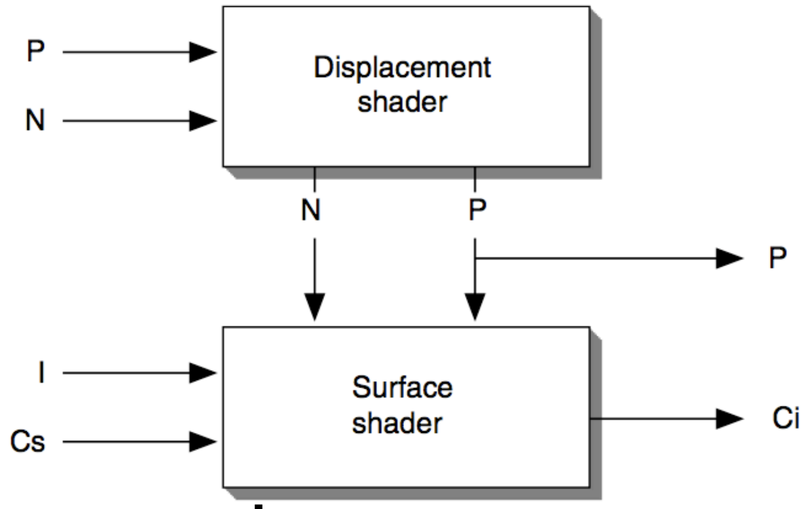
Using the Map

- To use the map we need to first calculate the direction of reflection, and then find the colour of the map in that direction.
- While it is quite simple to calculate the direction of reflection from the position of the viewer and the orientation of the surface, RenderMan can to do the work for us.
- By simply calling the reflect() function and converting the result into the correct co-ordinate space (world)




```
surface EnvMap
(
 float Ka=1; float Kd=.5;
 float Ks=.4; float Kr=.3;
 float roughness = 0.1;
 color specular=1;
 string MapName="";
float xroughness=0.5;float yroughness=0.5;
)
{

// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
// use Ward anistropic to give more metal like apperance
color spec = LocIllumWardAnisotropic (Nf, V, normalize(dPdu),
    xroughness, yroughness);
color Ct;
vector Rcurrent=reflect(I,Nf);
vector Rworld=vtransform("world",Rcurrent);
color Cr=color environment(MapName,Rworld);
// now calculate the shading values
Ct=Cs;
Oi=Os;
Ci= Oi * (Ct * (Ka * ambient() + Kd *diffuse(Nf)) +
  specular * (Ks*spec+Kr*Cr));
}
```

Displacement Shaders



- Displacement shaders are used to alter the smoothness of a surface
- Unlike Bump mapping which alters the surface normals a displacement shader actually alters the geometry and moves the points P
- By convention only a Displacement shader should alter the value of P but in most renderers this can also be done in the Surface shader.

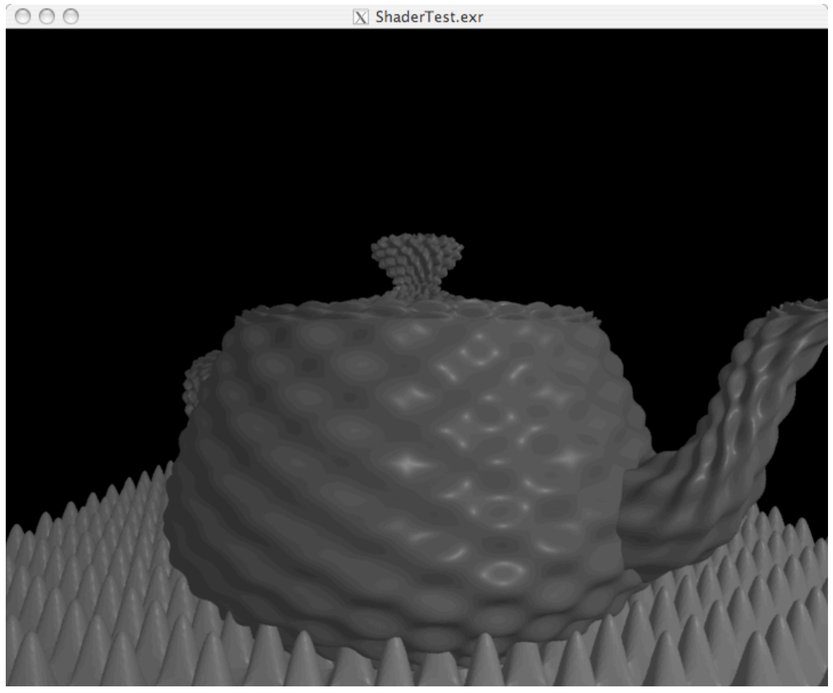
How Displacement works

- In prman each object in a 3D scene is sub-divided into a fine mesh of micro-polygons
- if a displacement shader has been assigned to an object, each micro-polygon is "pushed" or "pulled" in a direction that is parallel to the original surface normal of the micro-polygon.
- After figuring out how much it should move a micro-polygon, a displacement shader concludes its part of the rendering process by recalculating the orientation of the local surface normal(N).

Basic Displacement process

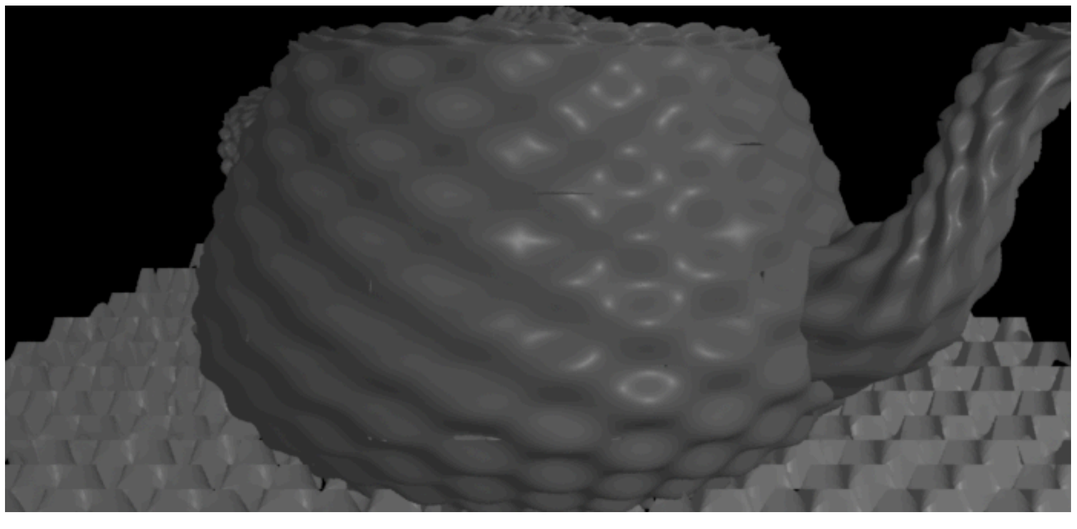
- The process of writing a displacement shader is as follows

1. Make a copy of the surface normal and normalize it
2. Calculate the value of the displacement required
3. Calculate a new position of the surface point "P" by moving it along the surface normal (typically this also requires some scaling)
4. Recalculate the surface normal (N)




```
displacement SimpleDisplacement(
    float dispScale=0.04;
    float RepeatS=4;
    float RepeatT=4;
    float trueDisp=1;

)
{
// make a copy of the normal
normal NN = normalize(N);
point PP;

// now calculate the new disp value for P
float ss=mod(s*RepeatS,1);
float tt=mod(t*RepeatT,1);

float disp=sin(ss*2*PI)*sin(tt*2*PI);
PP=P-NN*disp*dispScale;
N=calculatenormal(PP);
if(trueDisp == 1)
  P=PP;

}
```

displacement bounds



- This image is rendered with the default displacement bounds and some elements are cropped or missing
- To overcome this problem use the displacementbound attribute

```
1 ri.Attribute("displacementbound",{ri.COORDINATESYSTEM:["object"],"uniform␣float␣sphere":[0.5]})
```

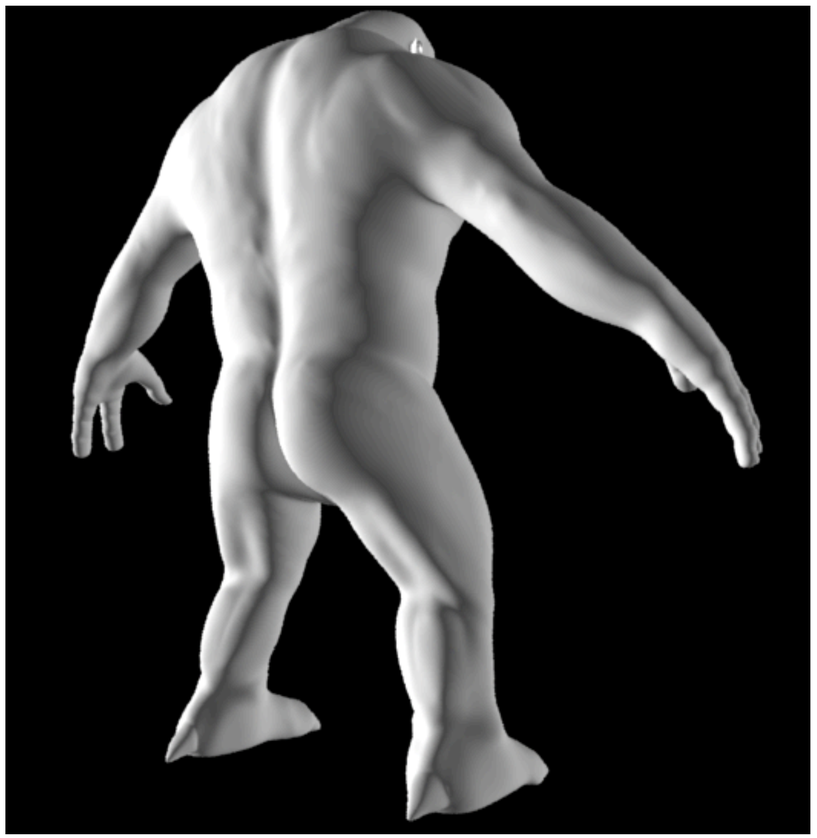
Displacement bounds

- This is a control that increases the sizes of calculated bounding boxes on primitives in order to account for the effects of displacement mapping.
- The size is specified by identifying a single floating-point value which is the radius of a sphere which is guaranteed to contain the maximum possible displacement, and the name of the coordinate system in which this sphere resides.
- This value should be as tight as possible. It is extremely inefficient, both in terms of memory usage and calculation time, to specify a bounding sphere which is larger than the actual displacement.
- Therefore, this sphere should be as small as possible without permitting points on the object to displace farther than the sphere's radius.

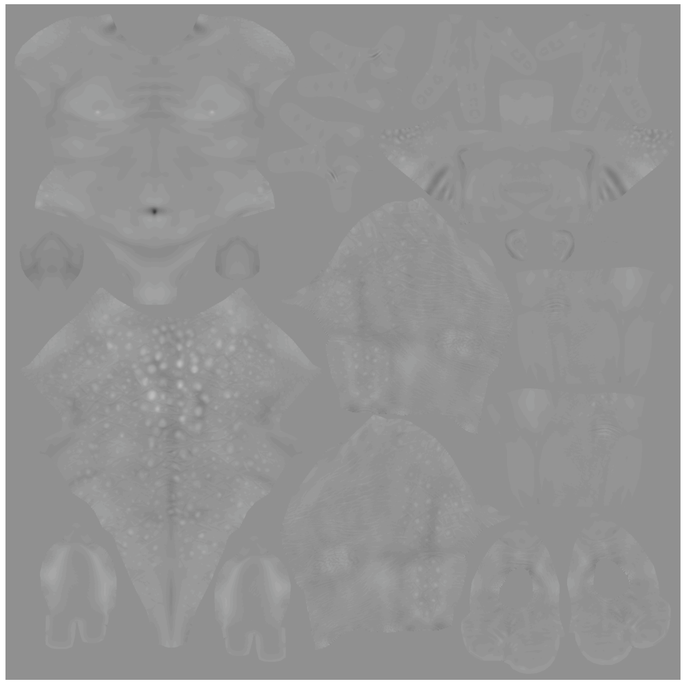
Bump Mapping

- Instead of moving the Point P we can instead re-orientate the point, so it is light as though it has been moved
- To do this we still calculate the new surface position but we don't assign it back to P.
- however we do calculate the surface normal N based on the displacement point
- This is controlled by the trueDisp parameter of the previous shader

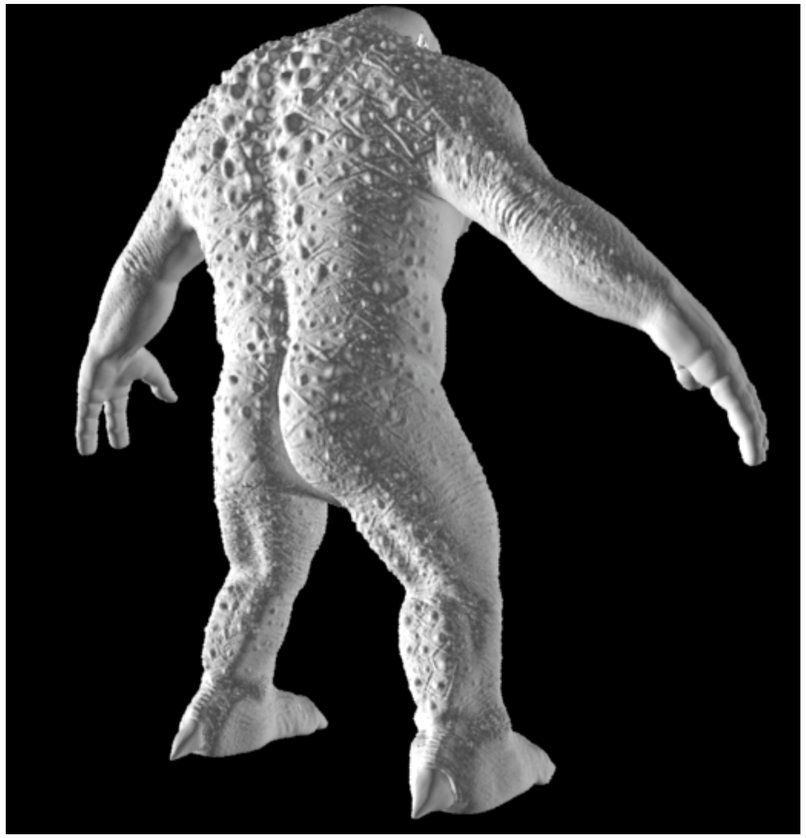
Texture Displacement



HierarchicalSubdivisionMesh



ZBrush Displacement Map



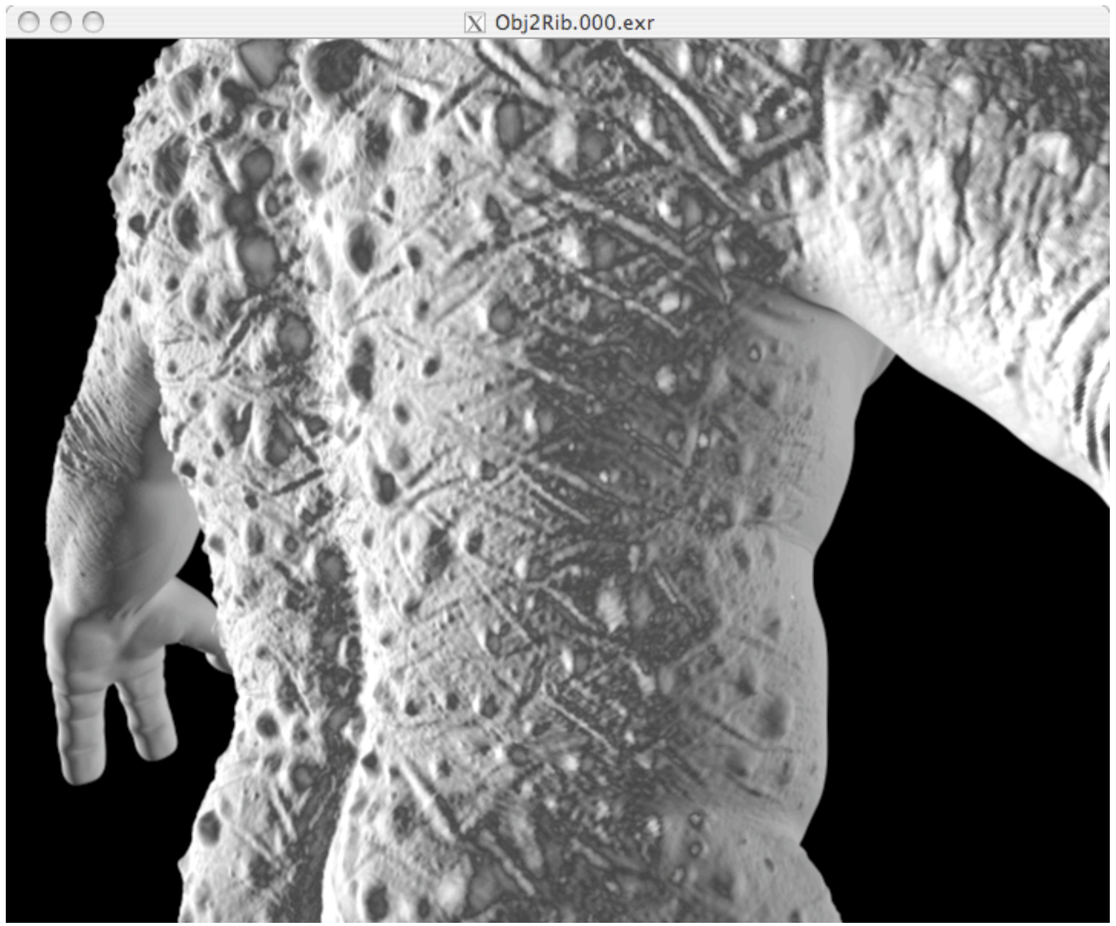
Displacement Shader Plastic Surface

cave troll model courtesy of Ritche Moore MA3D

Texture Map Displacement

- Z Brush displacement maps work best on Sub Division models
- These can be exported from Maya (or converted from an OBJ)
- The displacement will not work properly if a non subdiv model is used.
- It is also advisable to set the following and adjust to suit the model

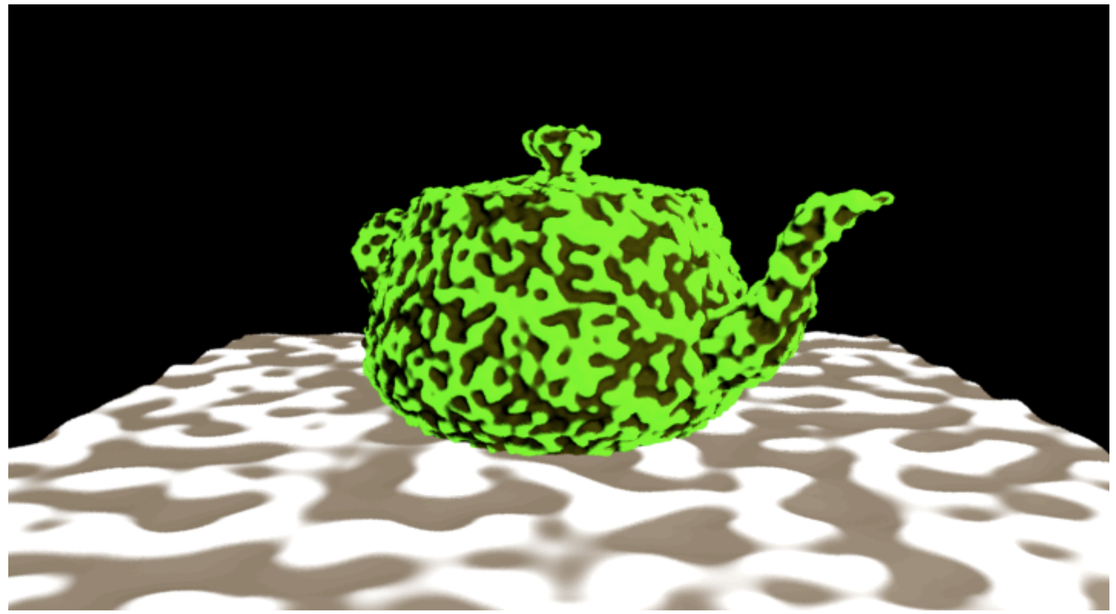
```
1 ri.ShadingInterpolation("smooth")
2 ri.ShadingRate( [0.1])
3 ri.PixelSamples( 8, 8)
4 ri.PixelFilter( ri.BOX ,1.0, 1.0)
```




```
ri.Displacement(  "ZBrushDisplacement" ,
    {"float␣Km": [0.35],"string␣
    displace_map" : "CaveTrollDisp.tx",
    "float␣swidth": [0.0001],"float␣
    twidth": [0.0001],"float␣samples": [
    400.000], })
```

```
displacement ZBrushDisplacement(
  float    Km       = 1.0;
    string displace_map="";
  float swidth=0.0001;
  float twidth=0.0001;
  float samples=200;
  )
{
  float    magnitude = 0;
  float level = 0;

if( displace_map != "")
  {
  level = float texture(displace_map,s,t,
              "swidth", swidth,
              "twidth",twidth,
              "samples", samples);
  // calculate the displacement
  magnitude = ((level * 2) - 1) * Km;
  }
  // no map so don't displace
  else
    magnitude = 0.0;
// calculate the new position
P -= normalize(N) * magnitude;
// re-calc the normal
N=calculatenormal(P);
}
```

Value Passing



- In the following example the values calculated in the Displacement shader are passed to the surface shader
- This reduces the amount of time required to shade the scene as the calculations only need to be done once per surface point
- In newer versions of renderman this is improved with the introduction of shader objects and co-shaders

Message Passing and Information Functions

```
1  float atmosphere( string paramname, output type variable )
2  float displacement( string paramname, output type variable )
3  float lightsource( string paramname, output type variable )
4  float surface( string paramname, output type variable )
```

- These functions access the value of the parameter named paramname of one of the shaders attached to the geometric primitive that is currently being shaded.
- If the appropriate shader exists, and a parameter name paramname exists in that shader, and the parameter is the same type as variable, then the value of that parameter is stored in variable and the function returns 1.0;
- otherwise, variable is unchanged and the function returns 0.0.

```
displacement SurfaceBump(float  Km = 0.1,freq = 4;
                output varying float  hump = 0;)
{
vector n = normalize(N);
point PP=  point "object" (P);
hump = noise(PP * freq);

hump = hump - 0.5;
P = P - n * hump * Km;
N = calculatenormal(P);
}
```

hump calculated and passed here

```
surface SurfaceShader(float Kd = 0.5,
              snow_ht = 0.0;
              color snow_color = 1,
              ground_color = color(0.466,0.388,0.309)
     )

{

vector  n = normalize(N),
nf = faceforward(n, I);

float   height = 0;
color surfcolor;
Oi = Os;

// Add snow according to the bump height

if(displacement("hump", height) == 1)
     {
     float blend =smoothstep(snow_ht-0.05,snow_ht+0.05,height);
     surfcolor = mix(ground_color,snow_color,blend);
     }

color  diffusecolor = Kd * diffuse(nf)+surfcolor;
Ci = Oi * Cs * diffusecolor;

}
```

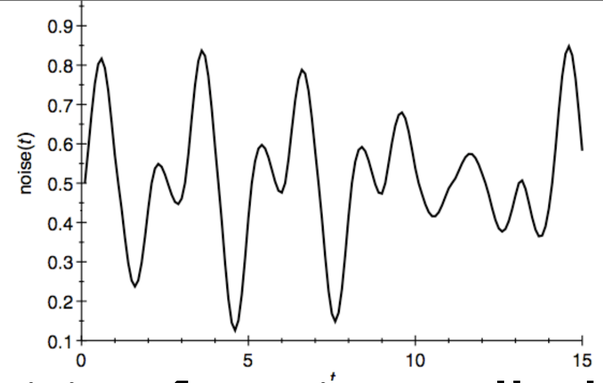
Noise

- To add visual interest to a surface we need to add some randomness
- However normal random functions are no good as the random function will change from frame to frame
- Instead we need a controlled randomness
- These problems are solved by a function known as noise().
- This generates a value which can be used to provide randomness in our textures, but it changes smoothly.
- Small changes in the input produce small changes in the output, making it tolerant to small movements, and numerical errors.

Noise Functions

- A noise function is essentially a seeded random number generator.
- It takes an integer as a parameter, and returns a random number based on that parameter.
- If you pass it the same parameter twice, it produces the same number twice.

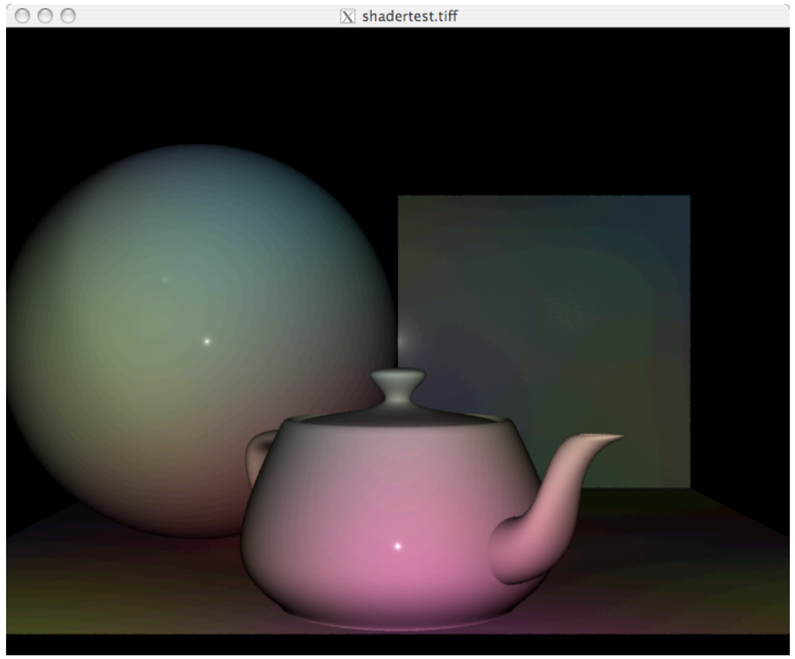
Renderman Noise



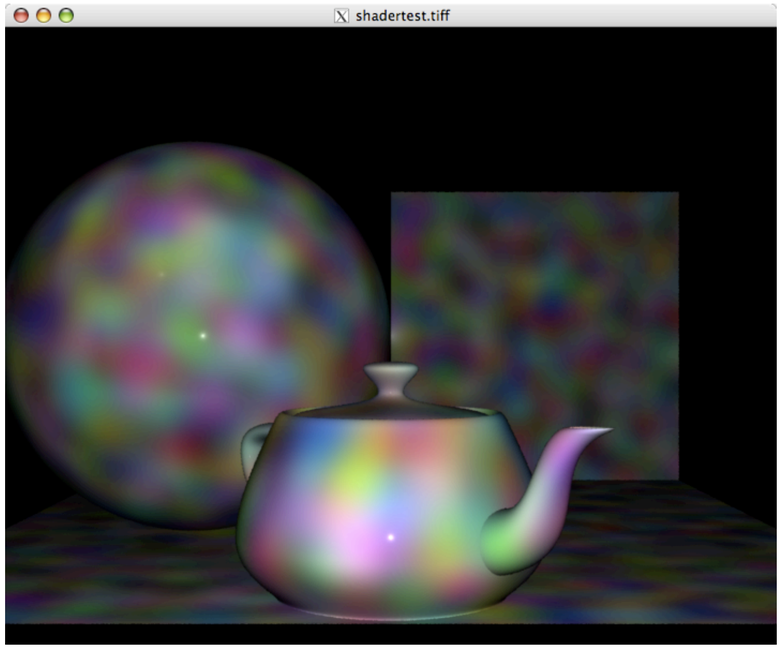
- The RenderMan Shading Language contains a primitive function called noise, which is a kind of repeatable, continuous, somewhat-random function with known frequency response.
- Noise is useful in generating stochastic patterns. Almost all interesting shaders have some calls to noise.
- The noise function can take different inputs, which determines the dimension of the domain of the noise function
- The values returned are guaranteed to be between 0 and 1, and average 0.5.
- The value returned is also guaranteed to be 0.5 if the input is a whole number, and will change smoothly between these "lattice points."

```
surface Noise
(
 float Ka=1; float roughness = 0.01;
 float RepeatS=1; float RepeatT=1; float Frequency=1;
)
{

// init the shader values

normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);
// create repeats
float ss=mod(s,RepeatS);
float tt=mod(t,RepeatT);

// transform the current point into shader space
// this creates a solid texture

point Np= transform("shader",P);
// now set the colour based on the noise function

color Ct= noise(Np*Frequency);
// now calculate the shading values

Oi=Os;
Ci= Oi * (Ct * (Ka * ambient() + diffuse(Nf)) +
    specular(Nf,V,roughness));
}
```

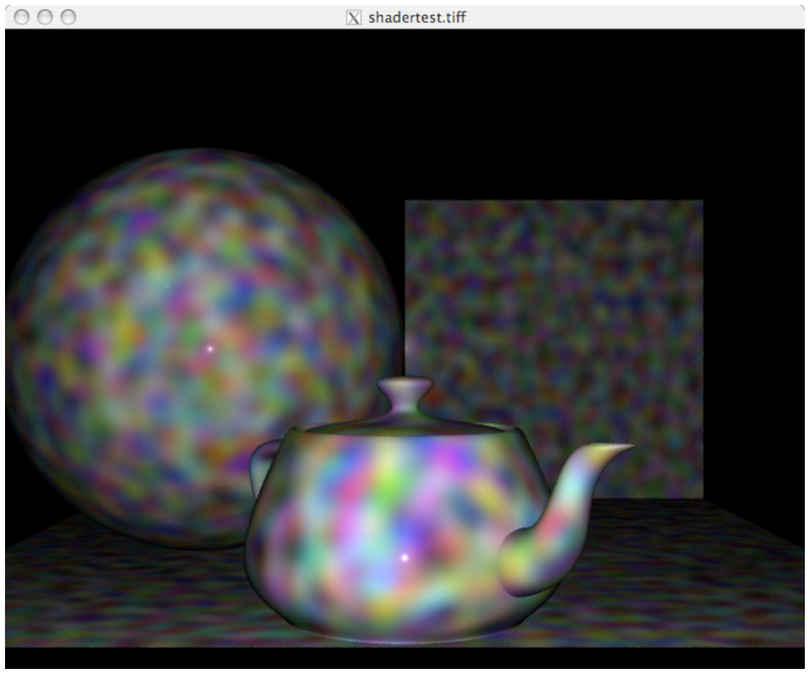
Using Noise



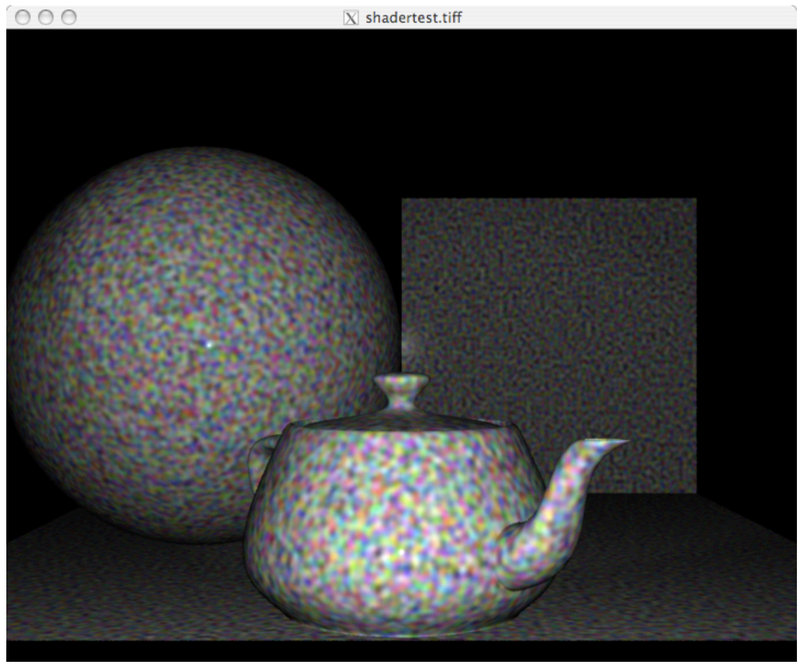
freq 0.5



freq 2



freq 16



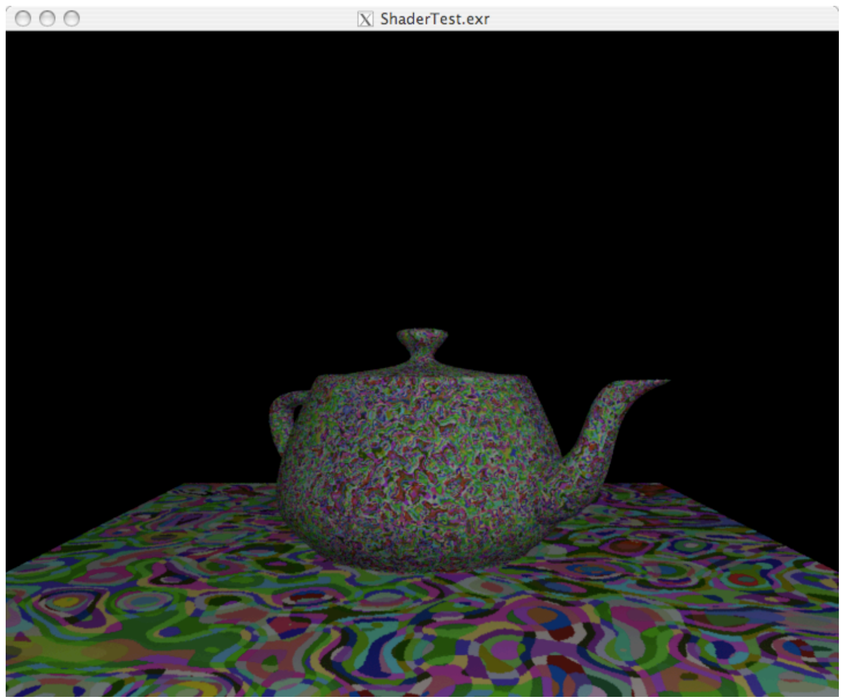
freq 32

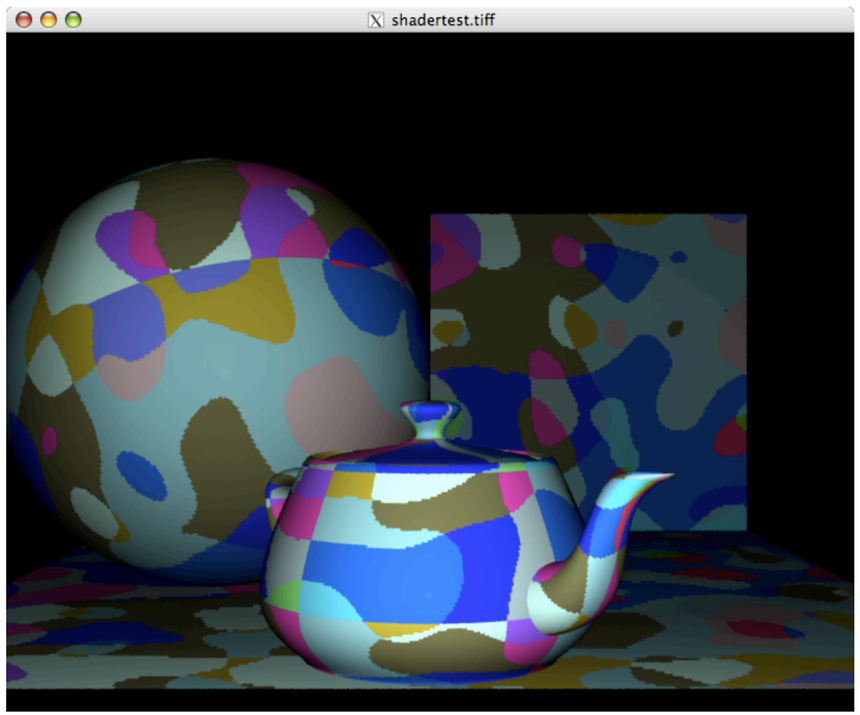
Noise Shader

- The previous shader transforms the current point P being rendered into shader space
- This is then used to determine the noise value at that point
- • This has the effect of ensuring that the object being rendered is “carved” out of the solid noise
- By adjusting the frequency value we can modify the amount of the noise being generated

Distorting Texture Cords

- Rather than assigning noise directly to a surface, or mixing it into another colour, you can also achieve useful results by using noise to distort the texture co-ordinates.
- This will break up the straight lines that are generated by simple pattern code.



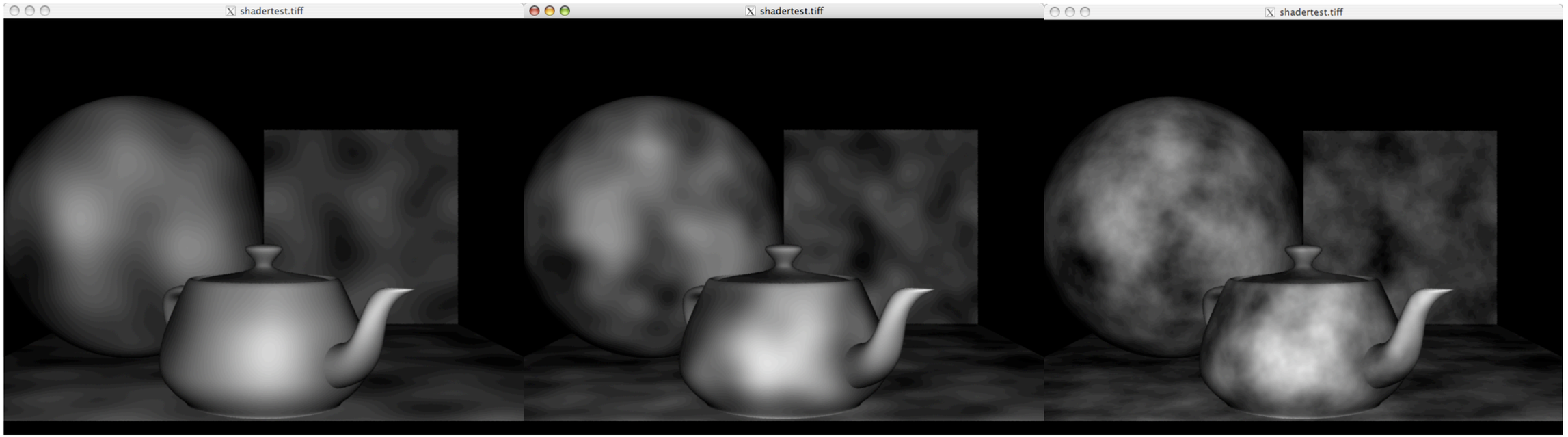



```
surface ModNoise

(
 float Ka=1;float roughness = 0.01;
 float RepeatS=1;  float RepeatT=1;
 float Frequency=1;
 float Ks=0.1;
 float Kd=1;
 vector NoiseVec=vector (1,0,0);
)

{
// init the shader values

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

// here we do the texturing

// create repeats

point PP = transform("shader",P);
normalize(NoiseVec);

float ss=s+ float noise(PP*RepeatS)*Frequency;
float tt=t+ float noise((PP*RepeatT) + NoiseVec)*Frequency;

// now set the colour based on the noise function

color Ct= cellnoise(ss*Frequency,tt*Frequency);

// now calculate the shading values

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

Layering Noise

- To allow several layers of noise detail to be combined we can use a loop to add different octaves of noise
- The Renderman noise function returns a value between 0 and 1 with an average of 0.5
- by subtracting 0.5 and multiplying by two we can scale the noise to be in a range ±1
- This results in an average of 0 so the average of mag is not changed as more layers are added
- At each layer the noise value is divided by the frequency so that the higher frequency layers have a smaller amplitude.

Fractional Brownian Motion (fBM) shader

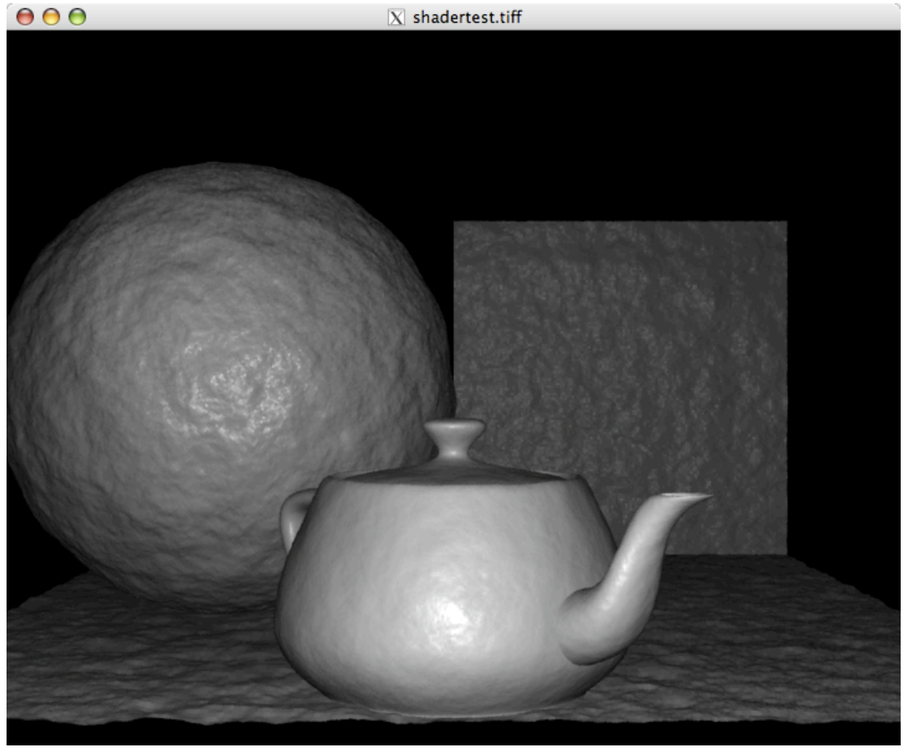


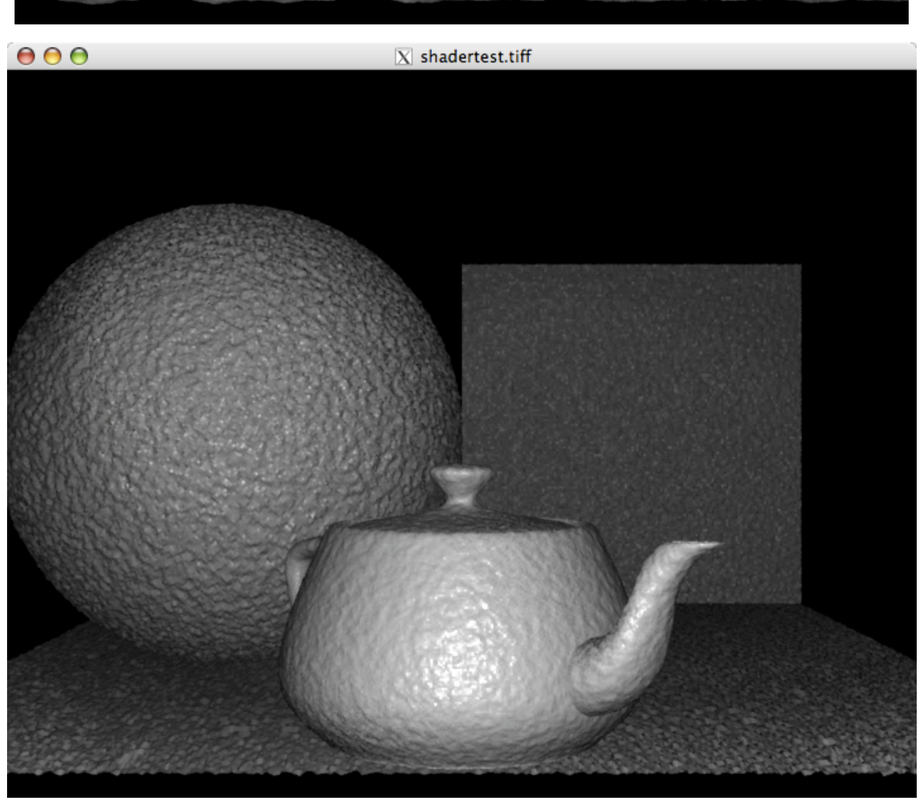
1 Layer

2 Layers

7 Layers

```
surface Layer
(
 float Ka=1;
 float Kd=1;
 float Scale=0.1;
 float Layers=6;
 output varying float Freq=1;
)
{

// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
float i;
float mag=0;
point Pt=transform("shader",P);
// Layer Noise
for(i=0; i<Layers; i+=1)
  {
  mag+=(float noise(Pt*Freq)-0.5)*2/Freq;
  Freq*=2;
  }
// now calculate the shading values */
color Ct=mag+0.5;
Oi=Os;
Oi = Os;
Ci = Os * (Ct * ( Ka*ambient() + Kd*diffuse(Nf) ) );
}
```





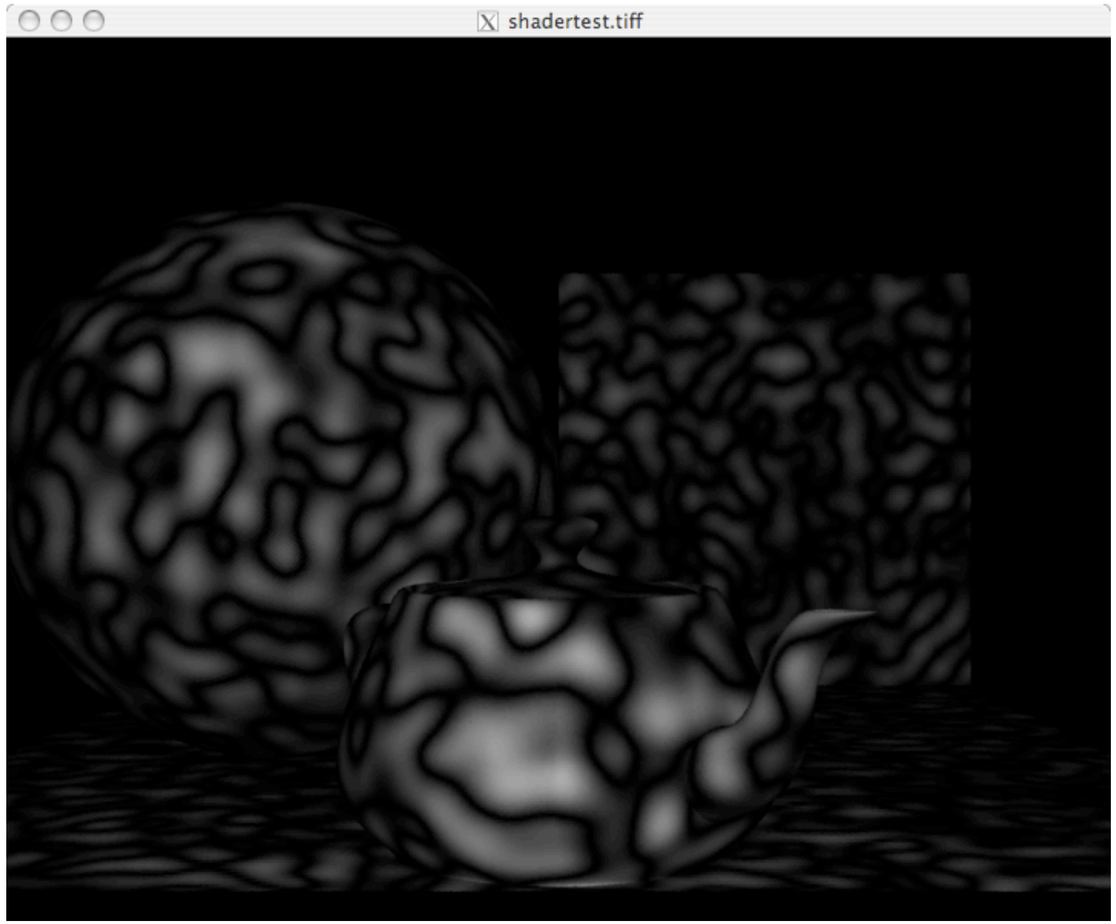

```
displacement fbmDisp
(
 float Km=0.1;
 float Layers=6;
 output varying float Freq=1;
)
{

// init the shader values
vector NN=normalize(N);
float i;
float mag=0;
point Pt=transform("shader",P);
for(i=0; i<Layers; i+=1)
   {
   mag+=(float noise(Pt*Freq)-0.5)*2/Freq;
   Freq*=2;
   }
mag /=length(vtransform("object",NN));
P=P+mag*NN*Km;
N=calculatenormal(P);
}
```

Turbulence

- turbulence may be implemented using the previous shader but taking the abs value of the noise

```
1 mag+=abs(float noise(P*Freq)-0.5)*2/Freq;
```

- It is possible to control the exact nature of the turbulence and fBM by reducing the number of layers and changing the frequency between layers
- This is know as lacunarity
- The use of the abs function “folds” the noise creating a discontinuity that makes turbulence appear subtly different to fBm.



Note we will get the following warning :-

Warning: potential space problem unwise to get noise in "current" space

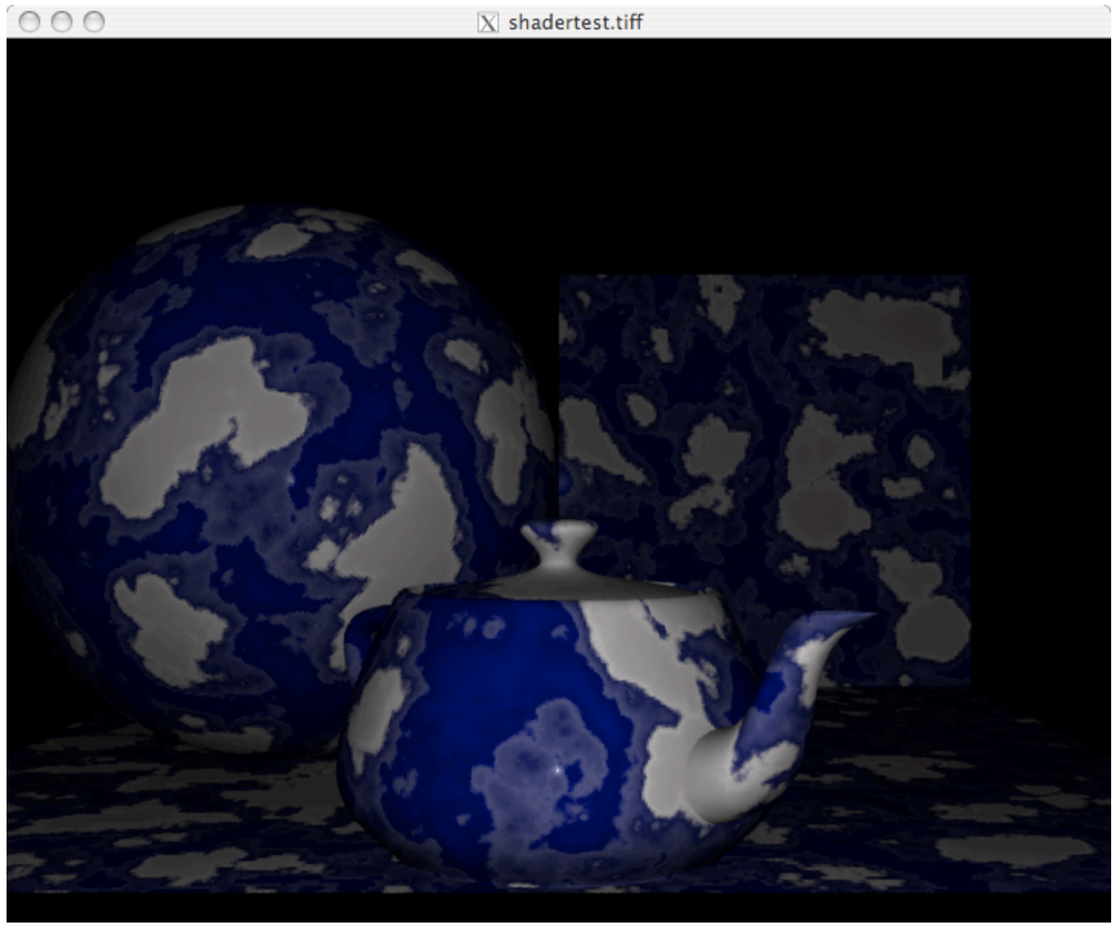
This is due to the fact we pass the space as a parameter and the compiler ignores the string type. This is not a problem with the shader

```
surface Turbulence
(
 float Ka=1;
 float Kd=1;
 float roughness = 0.01;
 float Scale=0.1;
 float Layers=4;
 output varying float StartFreq=4;
 float gain=4;
 float lacunarity=1.9132;
 string noiseSpace="shader";
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing
float i;
float mag=0;
float freq=1;
point PP=transform(noiseSpace,P);
PP*=StartFreq;

for(i=0; i<Layers; i+=1)
  {
  mag+=abs(float noise(PP*freq)-0.5)*2/pow(freq,gain);
  freq*=lacunarity;
  }
// now calculate the shading values
color Ct=mag;
Oi=Os;
Ci = Os * (Ct * ( Ka*ambient() + Kd*diffuse(Nf) ) );

}
```

Use of turbulence

- in the previous example we use mag to assign the colour
- However a more useful method is to use mag to blend different colours.
- The easiest way to do this is using the mix function to mix between two colours
- However for a more realistic mix of colours we can use the spline function

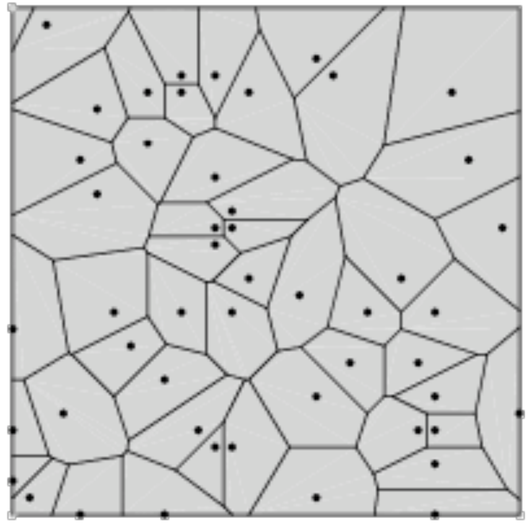
```
color Ct=spline(mag,
color "rgb" (0.25,0.35,0.25),
color "rgb" (0.25,0.35,0.25),
color "rgb" (0.20,0.30,0.20),
color "rgb" (0.20,0.30,0.20),
color "rgb" (0.20,0.30,0.20),
color "rgb" (0.25,0.35,0.35),
color "rgb" (0.25,0.35,0.35),
color "rgb" (0.15,0.25,0.15),
color "rgb" (0.15,0.25,0.10),
color "rgb" (0.15,0.25,0.10),
color "rgb" (0.10,0.20,0.10),
color "rgb" (0.10,0.20,0.10),
color "rgb" (0.25,0.35,0.25),
color "rgb" (0.10,0.10,0.20)
);
```



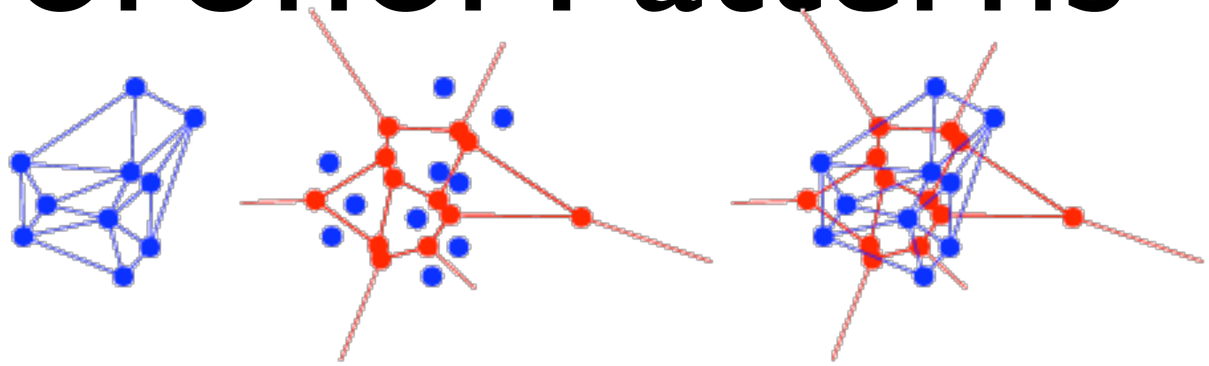

```
surface Marble
(
 float Ka=1; float Kd=0.5; float Ks=0.5; float roughness = 0.01;
 float Scale=0.76; float Layers=21;
 output varying float StartFreq=2; float gain=0.86;
 float lacunarity=1.74;
 float Repeat=1.0; string noiseSpace="shader";
 color specularcolor=1; float ColourLayers=4;
 color Colour1= color "rgb" (0.051 ,0.059, 0.596);
 color Colour2= color "rgb" (0.365 ,0.369, 0.635);
 color Colour3= color "rgb" (1.000 ,0.976, 0.988);
 float rDiff=0.05; float gDiff=0.05;  float bDiff=0.05;
)
{
// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);
// here we do the texturing
float i; float mag=0; float freq=StartFreq;
point PP=transform(noiseSpace,P);
PP*=Scale;
for(i=0; i<Layers; i+=1)
  {
  mag+=abs(float noise(P*Repeat*freq)-0.5)*2/freq;
  freq*=lacunarity;
  }
mag=smoothstep(0,0.4,mag);
color MColours[];
push(MColours,Colour1);
float rC=rDiff;  float gC=gDiff; float bC=bDiff;
// build up an array of colours for the spline function
color Ca;
Ca=color "rgb"(Colour1[0]-rC,Colour1[1]-gC,Colour1[2]-bC);
for (i=0; i<ColourLayers; i+=1)
  {
  push(MColours,Ca);
  Ca=color "rgb"(Ca[0]-rC,Ca[1]-gC,Ca[2]-bC);
  }
Ca=color "rgb"(Colour2[0]-rDiff,Colour2[1]-gDiff,Colour2[2]-bDiff);
for (i=0; i<ColourLayers; i+=1)
  {
  push(MColours,Ca);
  Ca=color "rgb"(Ca[0]-rC,Ca[1]-gC,Ca[2]-bC);
  }
Ca=color "rgb"(Colour3[0]-rDiff,Colour3[1]-gDiff,Colour3[2]-bDiff);
for (i=0; i<ColourLayers; i+=1)
  {
  push(MColours,Ca);
  Ca=color "rgb"(Ca[0]-rC,Ca[1]-gC,Ca[2]-bC);
  }
color Ct=spline(mag,MColours);
Oi=Os;
Ci= Oi * (Ct * (Ka * ambient() + Kd*diffuse(Nf)) +specularcolor *Ks *
    specular(Nf,V,roughness));
}
```

Dynamic Arrays

- The previous example uses the new renderman dynamic arrays to build up the colour table for the spline function
- We set an empty Colour array at the start of the shader then use the push function to add values to the end of the array



Voronoi Patterns



Delaunay triangulation

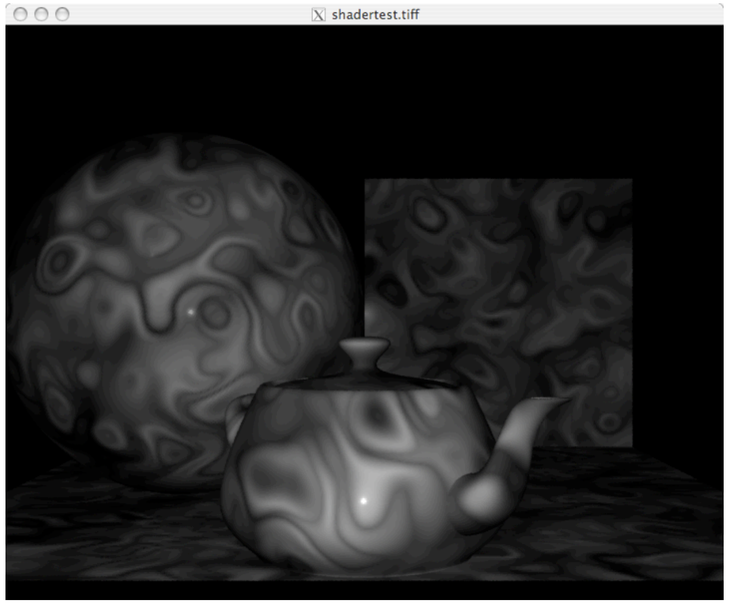
Voronoi diagram

Delaunay and Voronoi

- The partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other.
- A Voronoi diagram is sometimes also known as a Dirichlet tessellation. The cells are called Dirichlet regions, Thiessen polytopes, or Voronoi polygons.

Voronoi

- These were first suggested for use in Shading by Steven Worley
- In the advanced Renderman book the present a useful implementation of the function (with speedups)
- These functions allow for the 2d and 3d look up of cells and the generation of cellular type patterns
- The implementation of these functions can be found in the “noises.h” file included with the advanced renderman book
- They are included in the lecture code in the directory include.
- To use them we need to pass the path to the file to the shader compiler using -I [path to file] option



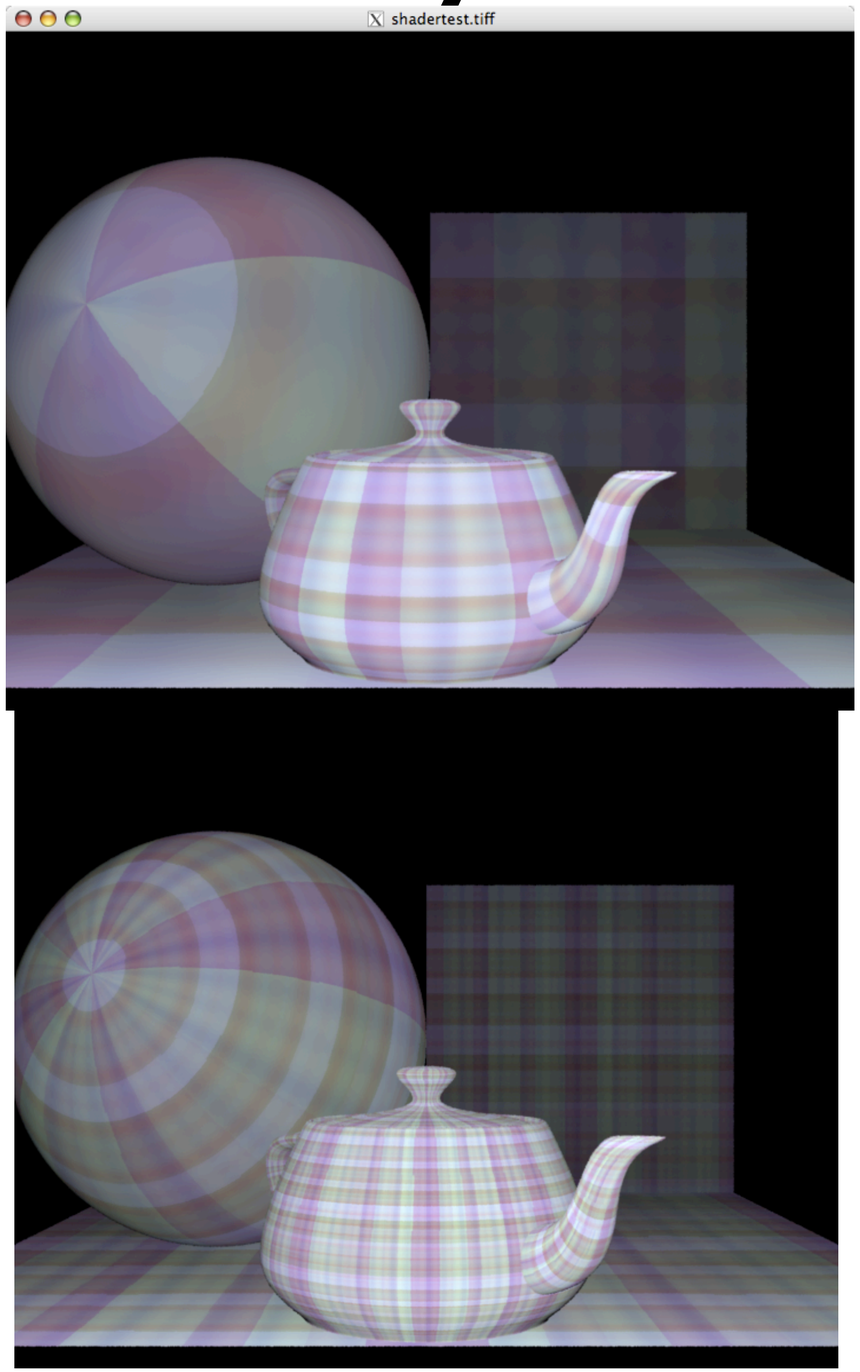

```
// Voronoi cell noise (a.k.a. Worley noise) -- 3-D, 1-feature version.
void
voronoi_f1_3d (point P;
          float jitter;
          output float f1;
          output point pos1;
     )
{
    point thiscell = point (floor(xcomp(P))+0.5, floor(ycomp(P))+0.5,
          floor(zcomp(P))+0.5);
    f1 = 1000;
    uniform float i, j, k;
    for (i = -1;  i <= 1;  i += 1) {
        for (j = -1;  j <= 1;  j += 1) {
            for (k = -1;  k <= 1;  k += 1) {
    point testcell = thiscell + vector(i,j,k);
                point pos = testcell +
                jitter * (vector cellnoise (testcell) - 0.5);
    vector offset = pos - P;
                float dist = offset . offset; // actually dist^2
                if (dist < f1) {
                    f1 = dist;  pos1 = pos;
                }
            }
  }
    }
    f1 = sqrt(f1);
}
```

```
#include "noises.h"
surface Voronoi
(
 float Ka=1;
 float roughness = 0.01;
 float Scale=0.1;
 float Layers=6;
 output float Freq=1;
 float RepeatX=0.5;
 float RepeatY=0.5;
 float RepeatZ=0.5;
)
{

// init the shader values
normal Nf = faceforward(normalize(N),I);
vector V = -normalize(I);

// here we do the texturing

point Ps= point(xcomp(P)/RepeatX,ycomp(P)/RepeatY,zcomp(P)
    /RepeatZ);
point PP = transform("shader",Ps);

point pos1;
float f1;
PP+=turbulence(Ps,0.2,2,1.8,3.0);

voronoi_f1_3d (PP, 0.8, f1, pos1);
color C1 = color(1,1,1);
color C2= color(0,0,0);
color Ct=mix(C1,C2,f1);
Oi=Os;
Ci= Oi * (Ct * (Ka * ambient() + 0.5*diffuse(Nf)) +
    specular(Nf,V,roughness));
}
```

Worked Examples

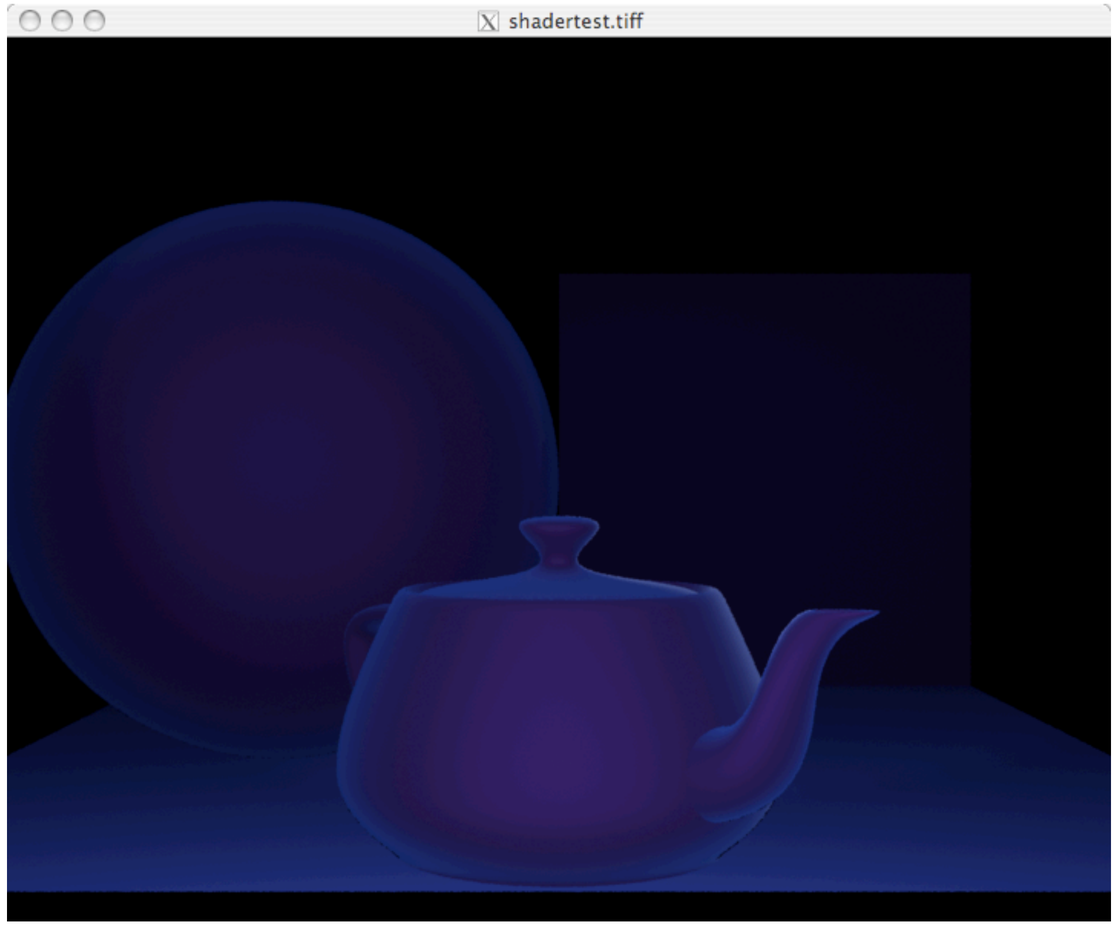
- The following example shaders are the work of Previous MSc student
- Most require the ARMAN include files to be compiled

Tartan By Elaine Kieran




```
#include "material.h"

surface tartan( float Ka= 1;
        float Ks = 0.1;
        float Kd = 1;
        float roughness = 1;
          float specularcolor = 0.1;
        float fuzz = 0.010;
        color ClothBack = color "rgb" (0.4, 0.4, 0.4);
        color Fleece1Colour = color "rgb" (0.533, 0.153, 0.133);
        color Fleece2Colour = color "rgb" (0.162, 0.140, 0.395);
        color Fleece3Colour = color "rgb" (0.022 ,0.395,0.078);
        color Fleece3BGColour = color "rgb" (0,0,0);
        color Fleece4Colour = color "rgb" (0,0,0);
        color Fleece4BGColour = color "rgb" (0.162, 0.140, 0.395);
        float freq1 = 10;
        float freq2 = 10;
        float RepeatS=1;
        float RepeatT=1;
        )

{
point PP = transform("shader", P);
normal Nf = faceforward (normalize(N), I);
vector V = -normalize(I);

color Ct;

color fleece1 = 0;
color fleece2 = 0;
color fleece3 = 0;
color fleece4 = 0;

// create different colour layers
fleece1 = mix(Fleece1Colour, ClothBack, pnoise( RepeatT*t*freq1, 3) );
fleece2 = mix(Fleece2Colour, ClothBack,pnoise(RepeatS* s*freq2, 5) );

fleece3 = mix( Fleece3Colour, Fleece3BGColour, cellnoise( RepeatS*s*5,
    fuzz) );
fleece4 = mix( Fleece4Colour, Fleece4BGColour, cellnoise( RepeatT*t*5,
    fuzz) );
// combine them
Ct = fleece3 + fleece1 + fleece2 + fleece4;


// use Oren Nayer Clay material from ARMAN notes
Ci = Oi * MaterialClay(Nf,Ct,Ka,Kd,roughness);
}
```

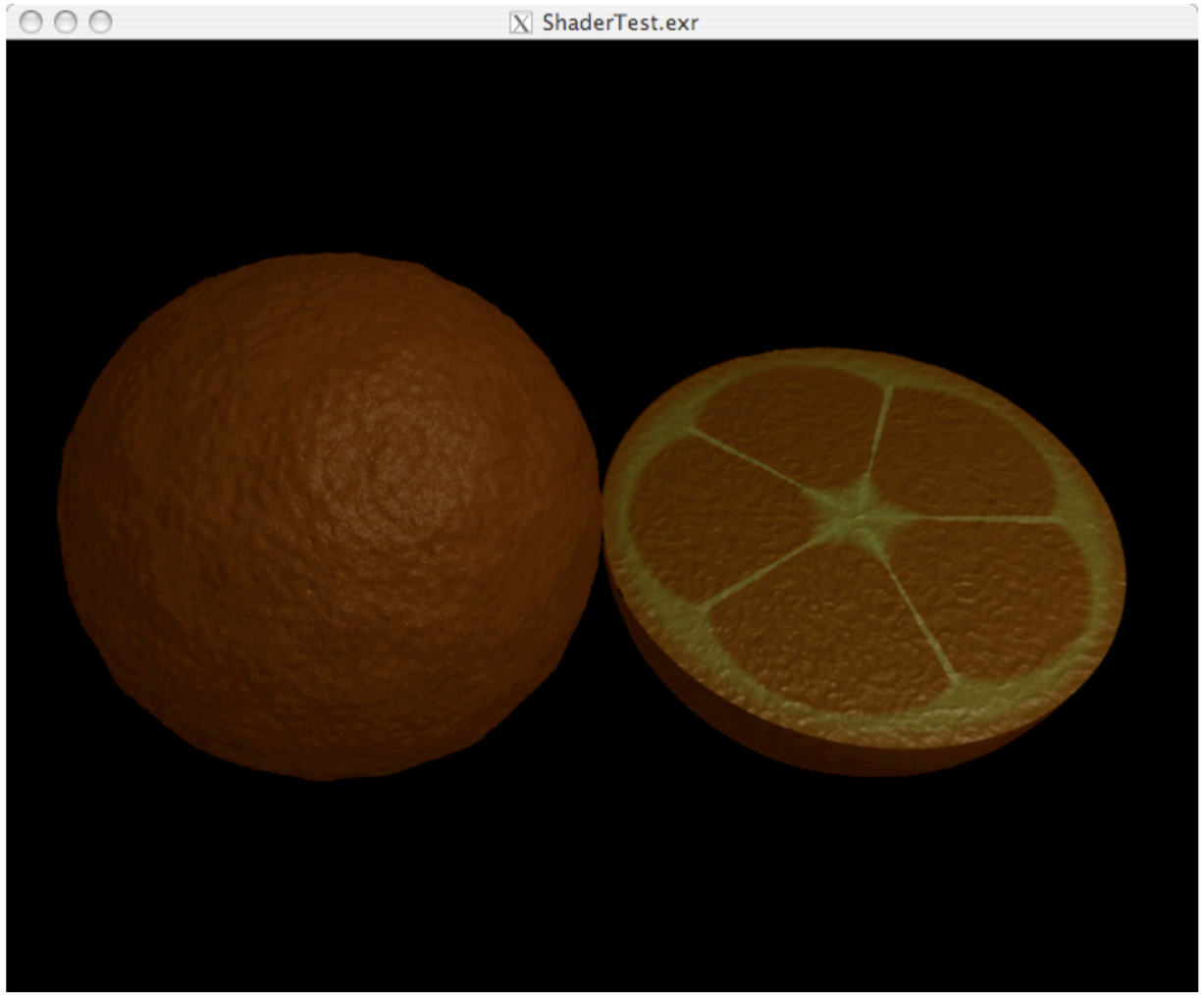
Velvet By Elaine Kieran



Adapted from Velvet shader by Stephen H. Westin

```
surface velvet( float Ka= 1; float Ks = 0.01;
        float Kd = 0.3; float Kr = 0.5;
        float roughness = 0.05; float specularcolor = 0.1;
        float fuzz = 0.010;
        color SheenColour= color "rgb" (0,0.648,0.438);
        uniform float octaves  = 6; uniform float lacunarity = 2;
        uniform float gain = 0.5;
        )
{
point PP = transform("shader", P);
float tt = t;
float ss = s;
normal Nf = faceforward (normalize(N), I);
vector V = -normalize(I);

float xroughness = 0.5;
float yroughness = 0.5;
vector xDir = normalize(dPdu);
vector aDir = normalize( (Dv(P) + 2 *Du(P) )/3);

vector Dir = normalize( Du(P) );
vector Dirv = normalize (Dv(P) );
float freq = 2; float depth = 2;

//the direction of the threads is alternated
vector MyDir= Dir / 0.5 ;

vector MyDir1 = Dir *random();
vector MyDir2 =aDir;

float d = noise(freq*s) + noise(freq*t) - 1; // from -1 to 1
vector MyDird = N *depth *d;

vector X = xDir / xroughness;
vector Y = (N * xDir) / yroughness;
color Ct;
color Caniso = 0;
float mag = 0; float magt = 0;
color Velvet = 0;
illuminance(P, Nf, PI/2)
  {
  color MyCl = VelvetColour;
  float MyCos, MySin;
  float reflect = 0.3;
  float highlight = 0.5;
  float strength = 8;
  vector Ln = normalize(L);
  vector Vn = normalize(V);
  MyCos = max(Dir.Vn, 0);
    Velvet += pow ( MyCos, 1.0/roughness ) * reflect *  Cl * highlight;
    MyCos = max ( Nf.Vn, 0 );
    MySin = sqrt (1.0-(MyCos * MyCos));
    Velvet += pow ( MySin, strength ) *  Cl * highlight *MyCl;
   }
Oi = Os;
Ci = Oi *  ( Velvet + Cs *(ambient() + Kd * diffuse(Nf)));

```

Orange Shaders By Colm Doherty



- See the Orange Shaders directory in the notes for the shaders

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
- http://mathworld.wolfram.com/VoronoiDiagram.html

Further Reading

- http://en.wikipedia.org/wiki/Voronoi_diagram