

# Texture

Digital Image Synthesis  
Yung-Yu Chuang

*with slides by Pat Hanrahan and Mario Costa Sousa*



# Texture

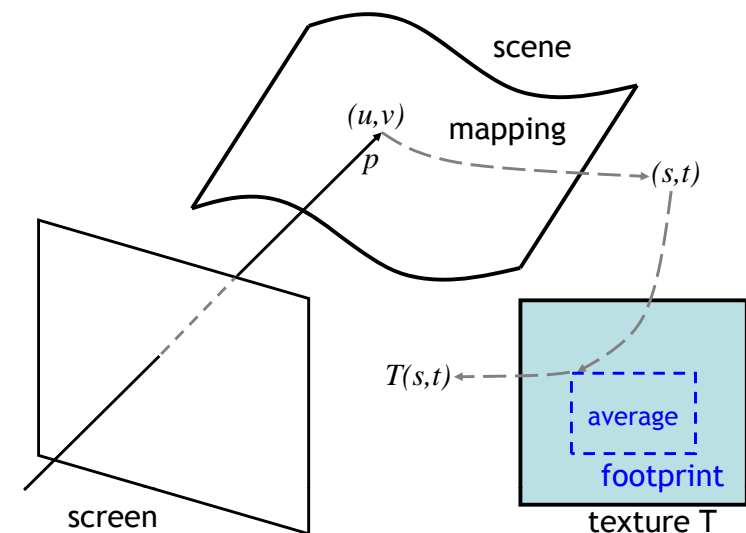
- Recall that materials requires parameters to describe their characteristics. These parameters are often spatially varying and textures are used to model such spatial variations.
- Textures is a function that maps points in some domain to values in some other domain.
- `core/texture.* mipmap.h texture/*` (currently 12 classes in total)
- In pbrt, pattern generation is separated from material implementation so that it is more flexible.

# Texture maps

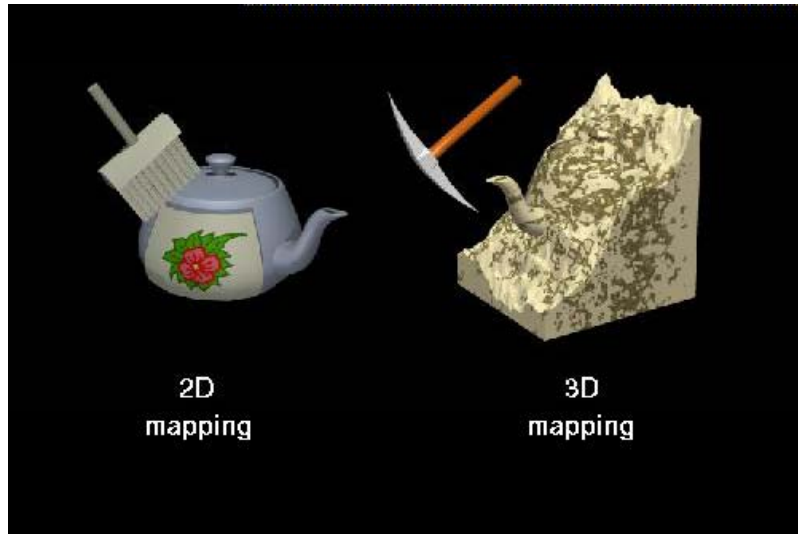


- How is a texture mapped to the surface?
  - Dimensionality: 1D, 2D, 3D
  - Texture coordinates (s,t)
    - Surface parameters (u,v)
    - Projection: spherical, cylindrical, planar
    - Reparameterization
- What can a texture control?
  - Surface color and transparency
  - Illumination: environment maps, shadow maps
  - Reflection function: reflectance maps
  - Geometry: displacement and bump maps

# Texture mapping



## Texture



## History



- 1974 basic idea (Catmull/Williams)
- 1976 reflection maps (Blinn/Newell)
- 1978 bump mapping (Blinn)
- 1983 mipmap (Williams)
- 1984 illumination (Miller/Hoffman)
- 1985 solid texture (Perlin)

## Texture maps



### Tom Porter's Bowling Pin



## Reflection maps



## Environment maps



Ray Traced



Environment Map

## Bump/displacement maps



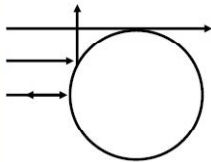
From Blinn 1976



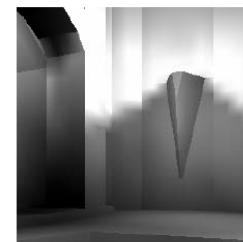
## Illumination maps



Miller and Hoffman, 1984



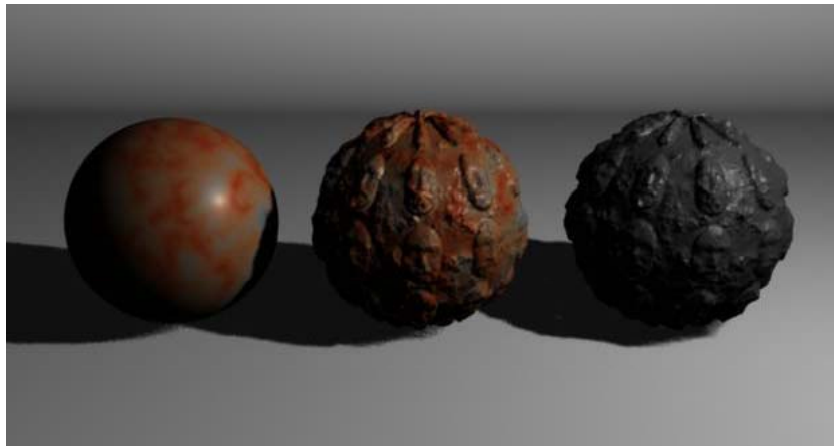
## Illumination maps



\*



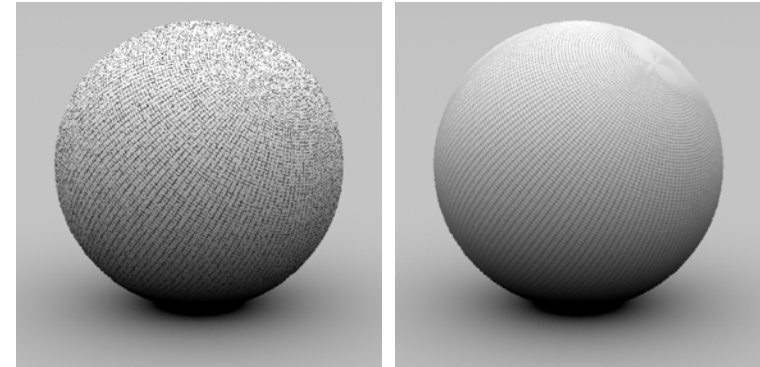
## Solid textures



## Sampling and antialiasing



- A source of high-frequency variation in the final image. Aliasing could be reduced by sampling, but it is more efficient if a good approximation can be provided.

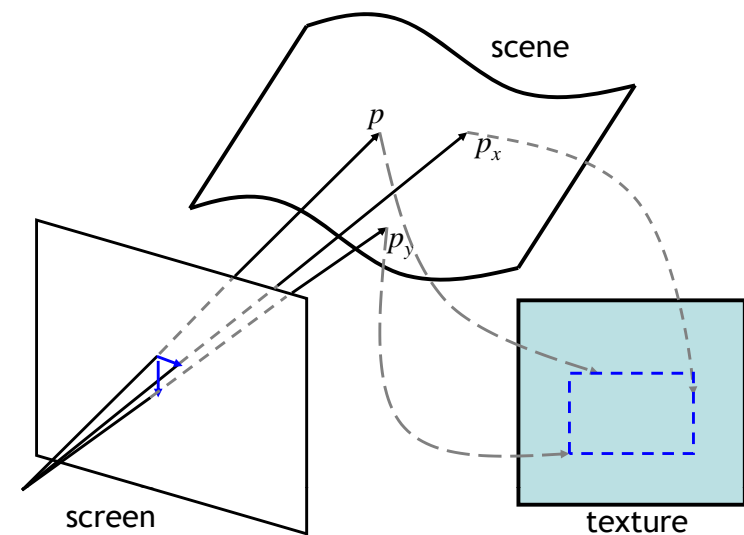


## Sampling and antialiasing



- Antialiasing for textures is not as hopeless as we have seen in chapter 7 because there are either analytical forms or ways to avoid adding high frequency
- Two problems that must be addressed
  1. Compute the sampling rate in the texture space
  2. Apply sampling theory to guide texture computation

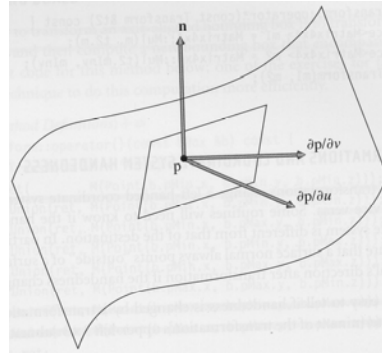
## Finding the texture sampling rate



## Differential geometry



- **DifferentialGeometry**: a self-contained representation for a particular point on a surface so that all the other operations in pbrt can be executed without referring to the original shape. It contains
- Position
- Surface normal
- Parameterization
- Parametric derivatives
- Derivatives of normals
- Pointer to shape

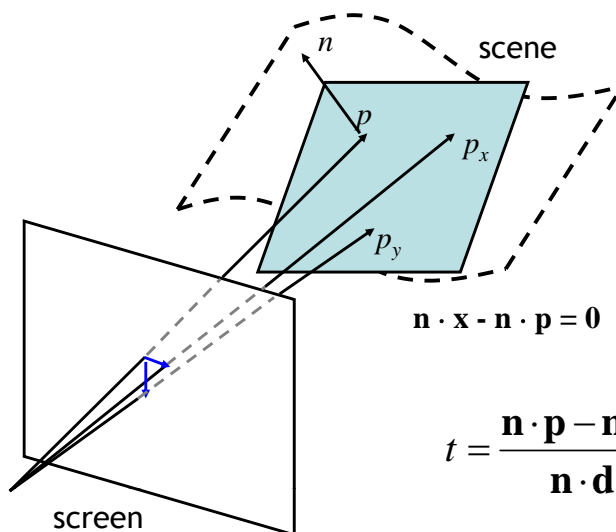


## DifferentialGeometry



```
struct DifferentialGeometry {  
    ...  
    // DifferentialGeometry Public Data  
    Point p;  
    Normal nn;  
    float u, v;  
    const Shape *shape;  
    Vector dpdu, dpdv;  
    Normal dndu, dndv;  
    mutable Vector dpdx, dpdy;  
    mutable float dudx, dvdx, dudy, dvdy;  
};  
Intersection::GetBSDF() calls  
DifferentialGeometry::ComputeDifferentials()  
to calculate these values.
```

## ComputeDifferentials



$$t = \frac{\mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{o}}{\mathbf{n} \cdot \mathbf{d}}$$

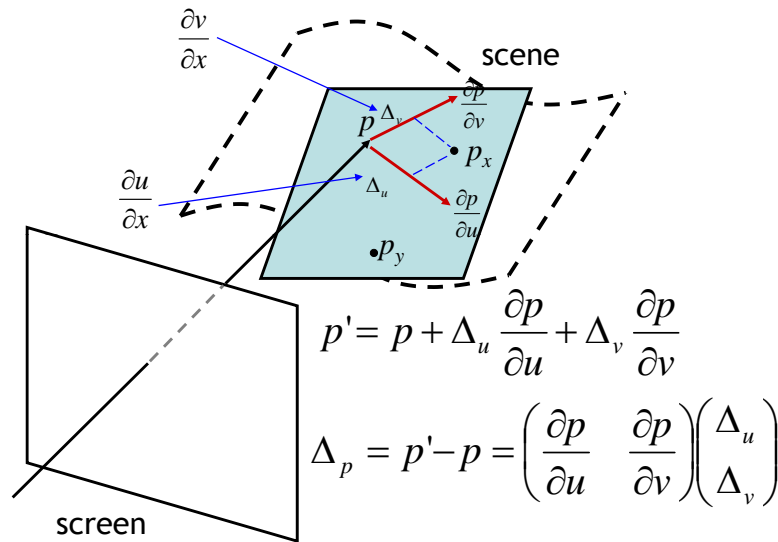
## ComputeDifferentials(RayDifferential&)



```
float d = -Dot(nn, Vector(p.x, p.y, p.z));  
Vector rxv(ray.rx.o.x, ray.rx.o.y, ray.rx.o.z);  
float tx = -(Dot(nn, rxv) + d)  
           / Dot(nn, ray.rx.d);  
Point px = ray.rx.o + tx * ray.rx.d;  
  
Vector ryv(ray.ry.o.x, ray.ry.o.y, ray.ry.o.z);  
float ty = -(Dot(nn, ryv) + d)  
           / Dot(nn, ray.ry.d);  
Point py = ray.ry.o + ty * ray.ry.d;  
  
dpdx = px - p;  
dpdy = py - p;
```

$$t = \frac{\mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{o}}{\mathbf{n} \cdot \mathbf{d}}$$

## ComputeDifferentials



## ComputeDifferentials



```
float A[2][2], Bx[2], By[2], x[2];

// we only need 2 equations; select two axes
int axes[2];
if (fabsf(nn.x) > fabsf(nn.y)
    && fabsf(nn.x) > fabsf(nn.z)) {
    axes[0] = 1; axes[1] = 2;
}
else if (fabsf(nn.y) > fabsf(nn.z)) {
    axes[0] = 0; axes[1] = 2;
}
else {
    axes[0] = 0; axes[1] = 1;
}
```

## ComputeDifferentials



```
// matrices for chosen projection plane
A[0][0] = dpdu[axes[0]];
A[0][1] = dpdv[axes[0]];
A[1][0] = dpdu[axes[1]];
A[1][1] = dpdv[axes[1]];
Bx[0] = px[axes[0]] - p[axes[0]];
Bx[1] = px[axes[1]] - p[axes[1]];
By[0] = py[axes[0]] - p[axes[0]];
By[1] = py[axes[1]] - p[axes[1]];
```

## ComputeDifferentials



```
if (SolveLinearSystem2x2(A, Bx, x)) {
    dudx = x[0]; dvdx = x[1];
}
else {
    dudx = 1.; dvdx = 0.;
}

if (SolveLinearSystem2x2(A, By, x)) {
    dudy = x[0]; dvdy = x[1];
}
else {
    dudy = 0.; dvdy = 1.;
}
```



## Texture coordinate generation

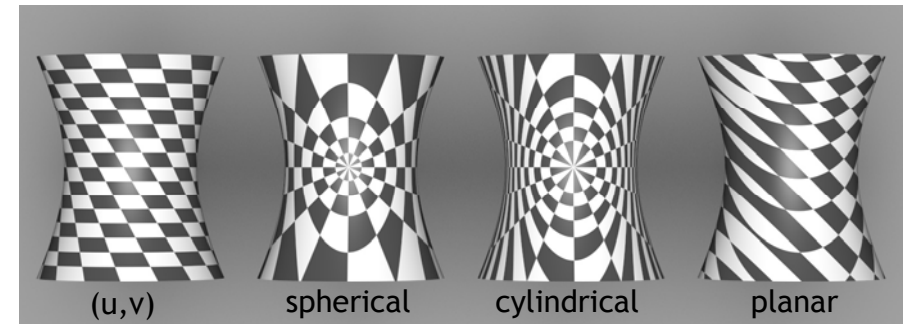


- We use  $(s,t)$  for texture coordinate and  $(u,v)$  for parametric coordinate.
- Creating smooth parameterization of complex meshes with low distortion is still an active research area in graphics.
- Two classes, `TextureMapping2D` and `TextureMapping3D`, provide an interface for computing these 2D or 3D texture coordinates.

## Texture coordinate generation



```
class TextureMapping2D {  
    virtual ~TextureMapping2D() { }  
    virtual void Map(const DifferentialGeometry &dg,  
                    float *s, float *t, float *dsdx, float *dtdx,  
                    float *dsdy, float *dtdy) const = 0;  
};
```

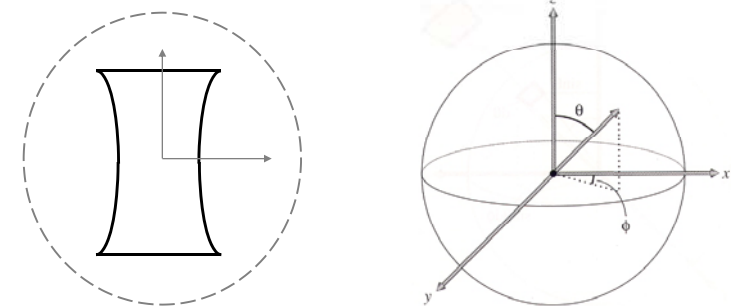


## (u.v)-mapping



```
class UVMapping2D : public TextureMapping2D {  
    UVMapping2D(float su = 1, float sv = 1, scale  
                float du = 0, float dv = 0); offset  
  
    ...     $s = s_u u + d_u$      $t = s_v v + d_v$   
}  
void UVMapping2D::Map(const DifferentialGeometry &dg,  
                     float *s, float *t, float *dsdx, float *dtdx,  
                     float *dsdy, float *dtdy) const  
{  
    *s = su * dg.u + du;    *t = sv * dg.v + dv;  
     $\frac{\partial s}{\partial x} = \frac{\partial s}{\partial u} \frac{\partial u}{\partial x} + \frac{\partial s}{\partial v} \frac{\partial v}{\partial x} = s_u \frac{\partial u}{\partial x}$      $x$  is a function of  $u,v$   
    *dsdx = su * dg.dudx;    *dtdx = sv * dg.dvdx;  
    *dsdy = su * dg.dudy;    *dtdy = sv * dg.dvdy;  
}
```

## Spherical mapping



```
class SphericalMapping2D : public TextureMapping2D {  
    ...  
    SphericalMapping2D(const Transform &toSph)  
        : WorldToTexture(toSph) { }  
};
```

## Spherical mapping



```
void sphere(Point &p, float *s, float *t) {  
    Vector v=Normalize(WorldToTexture(p)-Point(0,0,0));  
    float theta = SphericalTheta(v);  
    float phi = SphericalPhi(v);  
    *s = theta * INV_PI;  
    *t = phi * INV_TWOPHI;  
}
```

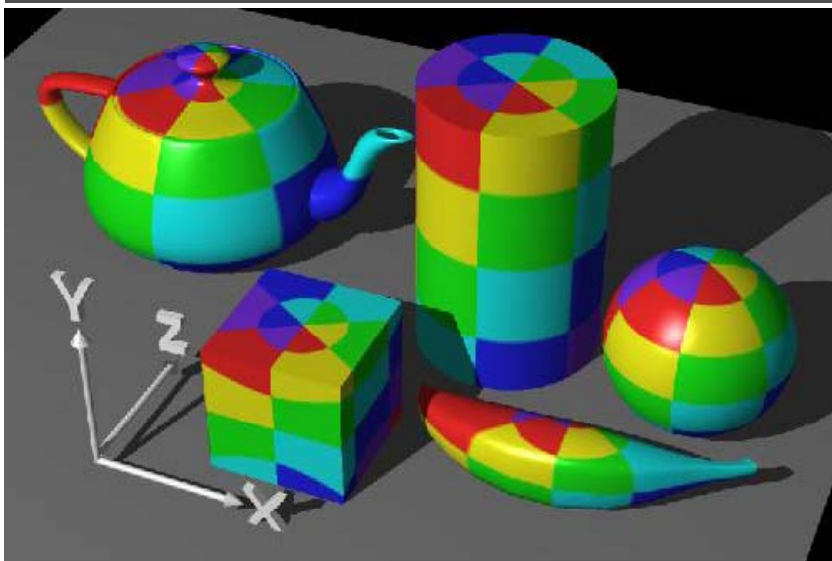
$$\frac{\partial s}{\partial x} \approx \frac{f_s(p + \Delta \partial p / \partial x) - f_s(p)}{\Delta}$$

## Spherical mapping

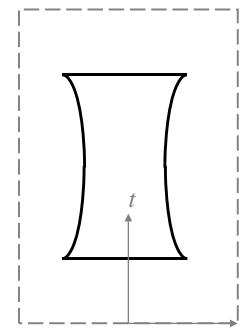


```
void SphericalMapping2D::Map(...)  
{  
    sphere(dg.p, s, t);  
    float sx, tx, sy, ty;  
    const float delta = .1f;  
    sphere(dg.p + delta * dg.dpdx, &sx, &tx);  
    *dsdx = (sx - *s) / delta;  
    *dtdx = (tx - *t) / delta;    avoid seam at t=1  
    if (*dtdx > .5) *dtdx = 1.f - *dtdx;  
    else if (*dtdx < -.5f) *dtdx = -(*dtdx + 1);  
    sphere(dg.p + delta * dg.dpdy, &sy, &ty);  
    *dsdy = (sy - *s) / delta;  
    *dtdy = (ty - *t) / delta;  
    if (*dtdy > .5) *dtdy = 1.f - *dtdy;  
    else if (*dtdy < -.5f) *dtdy = -(*dtdy + 1);  
}
```

## Spherical mapping



## Cylindrical mapping

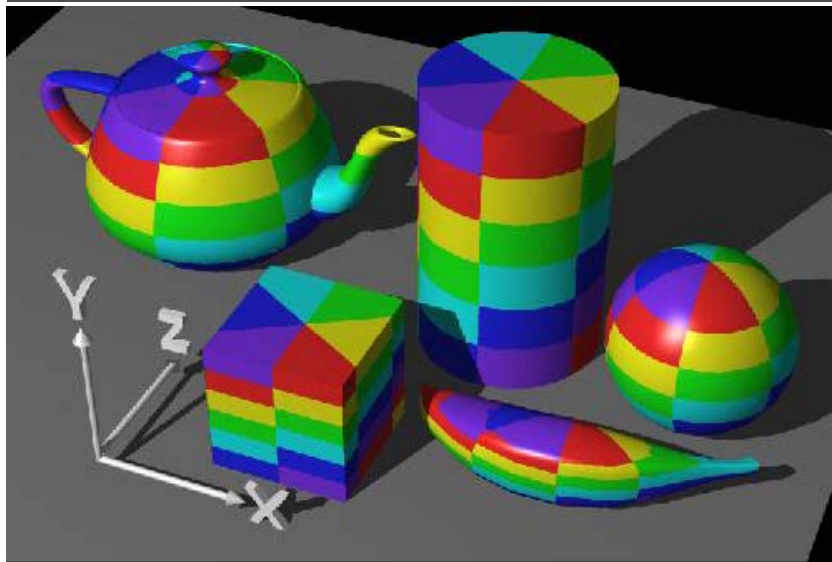


```
void cylinder(const Point &p, float *s, float *t) {  
    Vector v=Normalize(WorldToTexture(p)-Point(0,0,0));  
    *s = (M_PI + atan2f(v.y, v.x)) / (2.f * M_PI);  
    *t = v.z;  
}
```

How to calculate differentials?



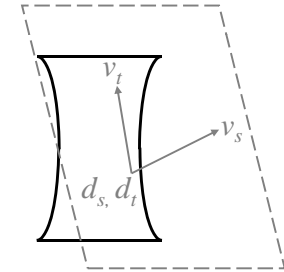
## Cylindrical mapping



## Planar mapping

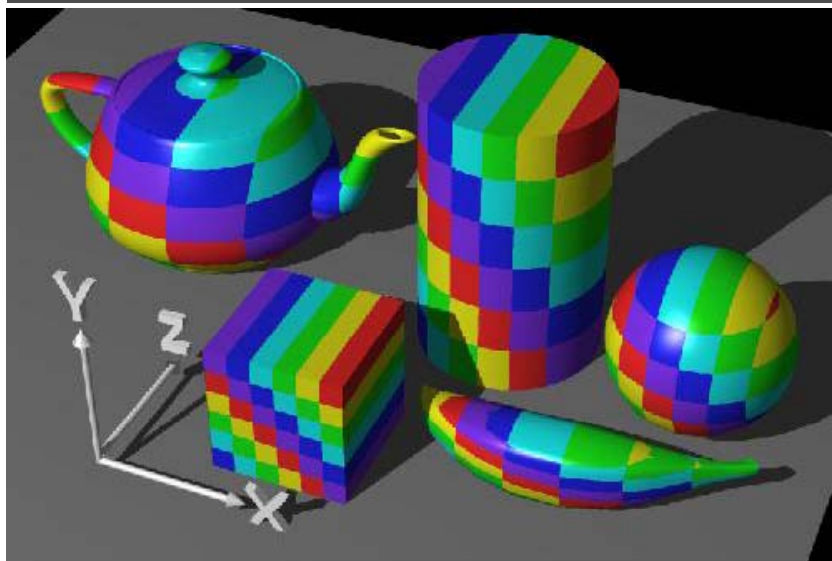


```
void PlanarMapping2D::Map(DifferentialGeometry &dg,
    float *s, float *t, float *dsdx, float *dtdx,
    float *dsdy, float *dtdy) const {
    Vector vec = dg.p - Point(0,0,0);
    *s = ds + Dot(vec, vs);
    *t = dt + Dot(vec, vt);
    *dsdx = Dot(dg.dpdx, vs);
    *dtdx = Dot(dg.dpdx, vt);
    *dsdy = Dot(dg.dpdy, vs);
    *dtdy = Dot(dg.dpdy, vt);
}
```



Example, for  $z=0$  plane,  
 $v_s=(1,0,0)$ ,  $v_t=(0,1,0)$ ,  $d_s=d_t=0$

## Planar mapping



## 3D mapping



```
class TextureMapping3D {
public:
    virtual ~TextureMapping3D() { }
    virtual Point Map(const DifferentialGeometry &dg,
        Vector *dpdx, Vector *dpdy) const = 0;
};

Point IdentityMapping3D::Map(
    const DifferentialGeometry &dg,
    Vector *dpdx, Vector *dpdy) const
{
    *dpdx = WorldToTexture(dg.dpdx);
    *dpdy = WorldToTexture(dg.dpdy);
    return WorldToTexture(dg.p);
}
```

## Texture



```
template <class T> class Texture {
public:
    virtual T Evaluate(DifferentialGeometry &) = 0;
    virtual ~Texture() { }
};
```

pbrr currently uses only float and Spectrum textures.

## ConstantTexture



```
template <class T>
class ConstantTexture : public Texture<T> {
public:
    ConstantTexture(const T &v) { value = v; }
    T Evaluate(const DifferentialGeometry &) const {
        return value;
    }
private:
    T value;
};
```

It can be accurately reconstructed from any sampling rate and therefore needs no antialiasing.

## ScaleTexture



```
template <class T1, class T2>
class ScaleTexture : public Texture<T2> {
public:
    // ScaleTexture Public Methods
    ScaleTexture(Reference<Texture<T1> > t1,
                Reference<Texture<T2> > t2)
    {
        tex1 = t1; tex2 = t2;
    }
    T2 Evaluate(const DifferentialGeometry &dg) const
    {
        return tex1->Evaluate(dg) * tex2->Evaluate(dg);
    }
private:
    Reference<Texture<T1> > tex1;
    Reference<Texture<T2> > tex2;
};
```

Leave antialiasing to tex1 and tex2; ignore antialiasing of the product, it is generally fine

## MixTexture



```
template <class T>
class MixTexture : public Texture<T> {
public:
    MixTexture(Reference<Texture<T> > t1,
                Reference<Texture<T> > t2,
                Reference<Texture<float> > amt)
    {
        tex1 = t1; tex2 = t2; amount = amt;
    }
    T Evaluate(const DifferentialGeometry &dg) const {
        T t1=tex1->Evaluate(dg), t2=tex2->Evaluate(dg);
        float amt = amount->Evaluate(dg);
        return (1.f - amt) * t1 + amt * t2;
    }
private:
    Reference<Texture<T> > tex1, tex2;
    Reference<Texture<float> > amount;
};
```

again, ignore antialiasing issue here

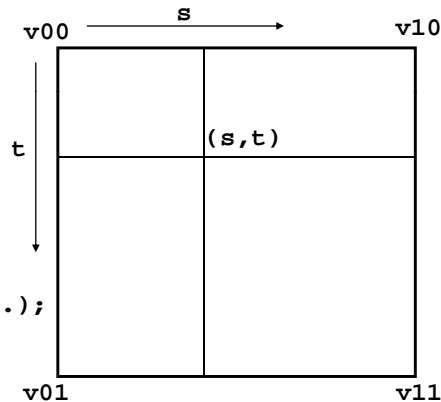
## BilerpTexture



```
BilerpTexture::BilerpTexture(TextureMapping2D *m,
                             const T &t00, const T &t01,
                             const T &t10, const T &t11)
```

```
{
    mapping = m;
    v00 = t00;
    v01 = t01;
    v10 = t10;
    v11 = t11;
}

T Evaluate(... &dg)
{
    float s, t, ...;
    mapping->Map(dg, &s, &t,...);
    ...
}
```



## Image texture



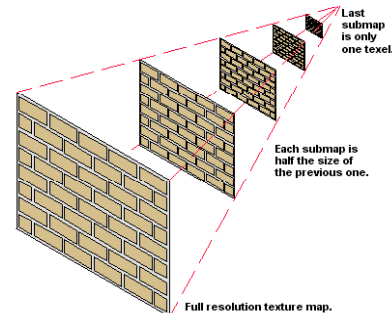
```
class ImageTexture : public Texture<T> {
public:
    ImageTexture(TextureMapping2D *m, string &filename,
                bool doTri, float maxAniso, ImageWrap wm,
                float scale, float gamma);
    T Evaluate(const DifferentialGeometry &);
    ~ImageTexture();
private:
    static MIPMap<T>
    *GetTexture(
        string &filename,
        bool doTrilinear,
        float maxAniso,
        ImageWrap wm, ...);
    ...
    MIPMap<T> *mipmap;
    TextureMapping2D *mapping;
};
```



## Mip maps



- It is more efficient to do antialiasing for textures because
  - Less expensive to get samples
  - We can apply pre-filtering since the texture function is fully defined
- However, the sampling rate is spatially varying, need to **filter over arbitrary regions efficiently**
- A mipmap takes at most 1/3 more memory



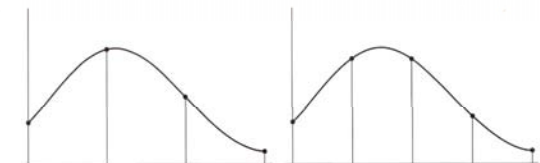
## Mipmaps



- Resize by resampling if the original resolution is not power of 2. Apply a box filter to do downsampling.

```
typedef enum { TEXTURE_REPEAT, TEXTURE_BLACK,
              TEXTURE_CLAMP } ImageWrap;
```

- Use 4 points and Lanczos reconstruction filter
- Implemented as a **BlockedArray** to improve performance



## Access texel at a level



```
const T &MIPMap<T>::texel(int level, int s, int t) {
    const BlockedArray<T> &l = *pyramid[level];
    switch (wrapMode) {
        case TEXTURE_REPEAT:
            s = Mod(s, l.uSize());
            t = Mod(t, l.vSize()); break;
        case TEXTURE_CLAMP:
            s = Clamp(s, 0, l.uSize() - 1);
            t = Clamp(t, 0, l.vSize() - 1); break;
        case TEXTURE_BLACK: {
            static const T black = 0.f;
            if (s < 0 || s >= l.uSize() ||
                t < 0 || t >= l.vSize())
                return black; break;
        }
    }
    return l(s, t);
}
```

## Mipmap lookup (MIPMap<T>::Lookup)



- Return the texel value for a given (s, t)
- Two methods:
  - Triangle filter
  - EWA filter

## Mipmap lookup



- Trilinear filtering:

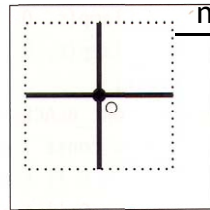
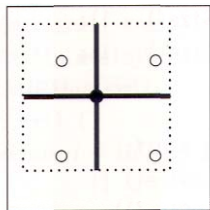
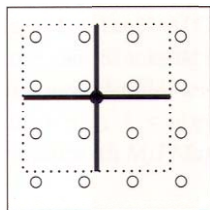
Lookup(float s, float t, float width)

$$\frac{1}{w} = 2^{n-1-l}$$

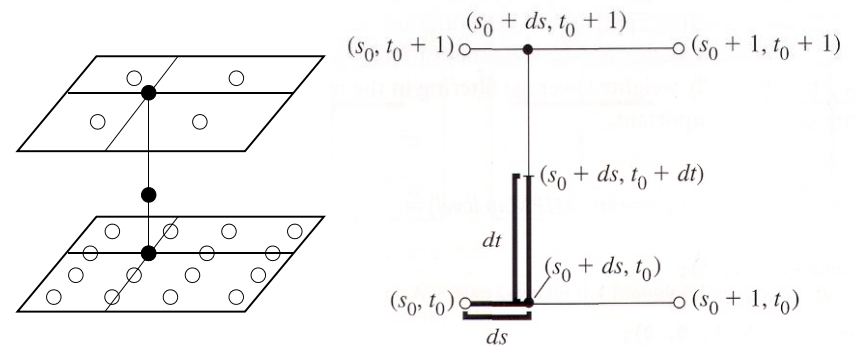
$$l = n-1 + \log w$$

level	Res.
0	$2^{n-1}$
1	
:	:
l	$2^{n-1-l}$
:	:
n-1	1

choose the level to cover four texels



## Trilinear filtering



## Trilinear filtering



```
T Lookup(float s, float t, float width){
    float level = nLevels - 1 + Log2(max(width,
    1e-8f));

    if (level < 0)
        return triangle(0, s, t);
    else if (level >= nLevels - 1)
        return texel(nLevels-1, 0, 0);
    else {
        int iLevel = Floor2Int(level);
        float delta = level - iLevel;
        return (1-delta)*triangle(iLevel, s, t)
            + delta * triangle(iLevel+1, s, t);
    }
}
```

## Trilinear filtering

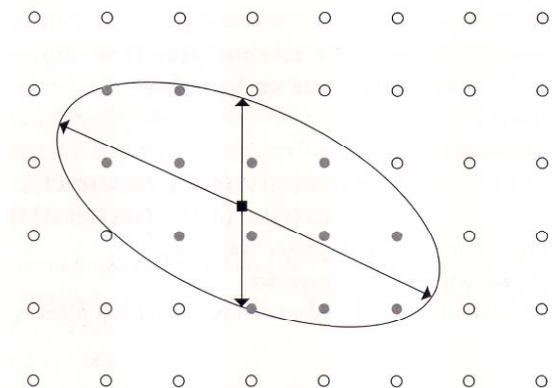


```
T triangle(int level, float s, float t)
{
    level = Clamp(level, 0, nLevels-1);
    s = s * pyramid[level]->uSize() - 0.5f;
    t = t * pyramid[level]->vSize() - 0.5f;
    int s0 = Floor2Int(s), t0 = Floor2Int(t);
    float ds = s - s0, dt = t - t0;
    return (1-ds)*(1-dt)*texel(level,s0,t0)
        +(1-ds)*dt * texel(level, s0, t0+1)
        +ds*(1-dt) * texel(level, s0+1, t0)
        +ds*dt * texel(level, s0+1, t0+1);
}
```

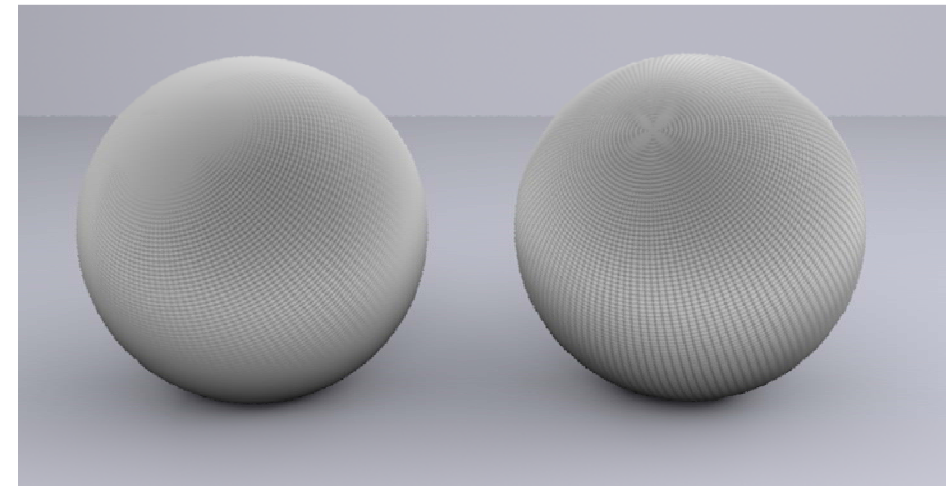
## Mipmap lookup



- Elliptically weighted average filtering



## Mipmaps



trilinear

EWA

## Solid and procedural texturing

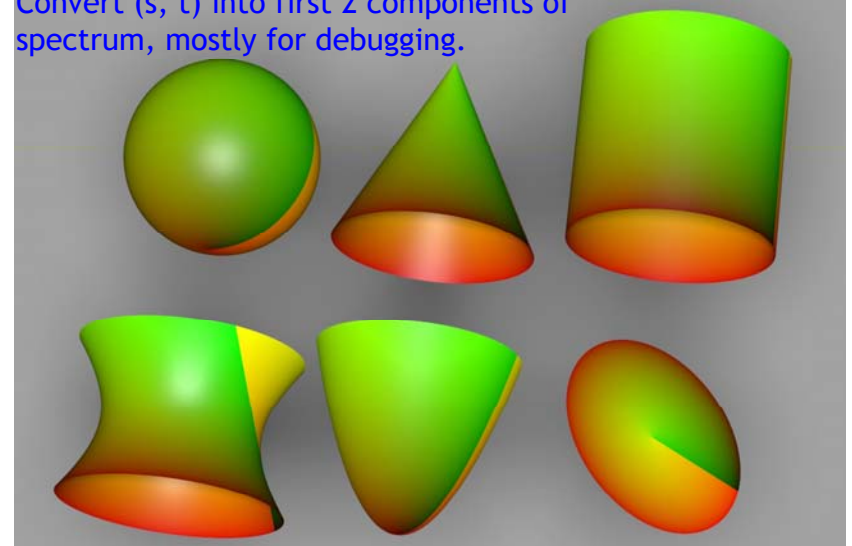


- Solid texture: textures defined over 3D domains; often requiring a large amount of storage
- Procedural texture: short programs are used to generate texture values (e.g. sine waves)
  - Less storage
  - Better details (evaluate accurate values when needed; resolution independent)
  - Harder to control
  - More difficult to do antialiasing

## UV texture



Convert (s, t) into first 2 components of spectrum, mostly for debugging.

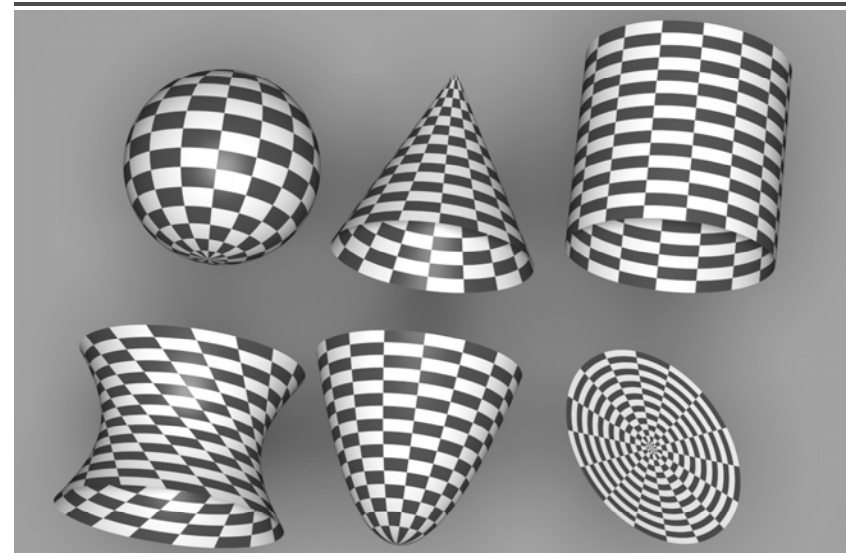


## UV texture



```
Spectrum Evaluate(const DifferentialGeometry &dg) {  
    float s, t, dsdx, dtdx, dsdy, dtdy;  
    mapping->Map(dg, &s, &t, &dsdx,&dtdx,&dsdy,&dtdy);  
    float rgb[3] = {s-Floor2Int(s), t-Floor2Int(t), 0};  
    return Spectrum::FromRGB(rgb);  
}
```

## Checkboard





## Checkerboard2D



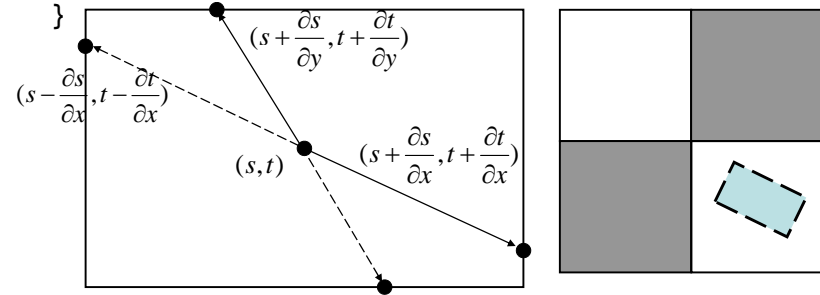
```
class Checkerboard2D : public Texture<T> {
    Checkerboard2D(TextureMapping2D *m,
        Reference<Texture<T>> c1, Reference<Texture<T>> c2,
        const string &aa);
    ...
    T Evaluate(const DifferentialGeometry &dg) const {
        float s, t, dsdx, dtdx, dsdy, dtdy;
        mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
        if (aaMethod==NONE) {
            // no antialiasing
            if ((Floor2Int(s) + Floor2Int(t)) % 2 == 0)
                return tex1->Evaluate(dg);
            return tex2->Evaluate(dg);
        } else
        { // closed-form box-filtered }
    }
}
```

## Close-form



```
float ds = max(fabsf(dsdx), fabsf(dsdy));
float dt = max(fabsf(dtdx), fabsf(dtdy));
float s0 = s - ds, s1 = s + ds;
float t0 = t - dt, t1 = t + dt;
if (Floor2Int(s0) == Floor2Int(s1) &&
    Floor2Int(t0) == Floor2Int(t1)) {
    if ((Floor2Int(s) + Floor2Int(t)) % 2 == 0)
        return tex1->Evaluate(dg);
    return tex2->Evaluate(dg);
}
```

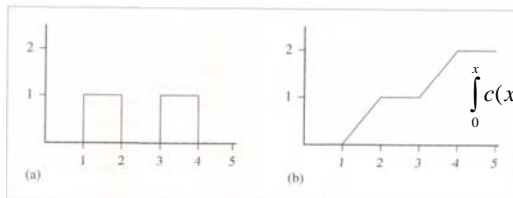
point sampling



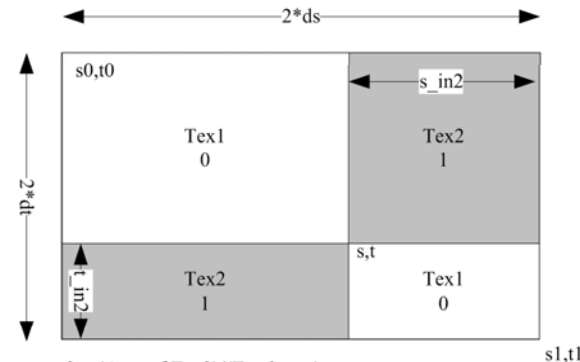
## Close-form



```
#define BUMPINT(x) \
    (Floor2Int((x)/2) + \
    2.f * max((x/2)-Floor2Int(x/2) - .5f, 0.f))
float sint = (BUMPINT(s1) - BUMPINT(s0)) / (2.f*ds);
float tint = (BUMPINT(t1) - BUMPINT(t0)) / (2.f*dt);
float area2 = sint + tint - 2.f * sint * tint;
if (ds > 1.f || dt > 1.f) area2 = .5f;
return (1.f - area2) * tex1->Evaluate(dg) +
    area2 * tex2->Evaluate(dg);
```



## Close-form



$$\begin{aligned} \text{area2} &= (\text{Area of Tex2}) / (\text{Total area}) \\ &= (s\_in2 * (2*dt - t\_in2) + (2*ds - s\_in2) * t\_in2) / (2*ds * 2*dt) \\ &= \frac{s\_in2 / (2*ds)}{2} * (1 - \frac{t\_in2}{2*dt}) + (1 - \frac{s\_in2}{2*ds}) * \frac{t\_in2}{2*dt} \end{aligned}$$

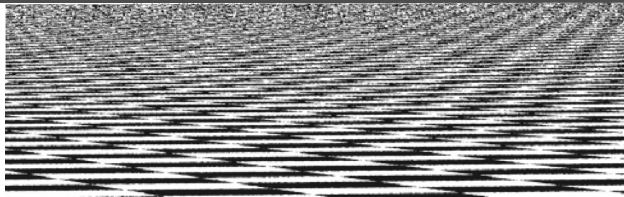
$$\begin{aligned} s\_in2 / (2*ds) &= (\text{BUMPINT}(s1) - \text{BUMPINT}(s0)) / (2*ds) = \text{sint} \\ t\_in2 / (2*dt) &= (\text{BUMPINT}(t1) - \text{BUMPINT}(t0)) / (2*dt) = \text{tint} \end{aligned}$$

$$\text{area2} = \text{sint} * (1 - \text{tint}) + (1 - \text{sint}) * \text{tint} = \text{sint} + \text{tint} - 2 * \text{sint} * \text{tint}$$

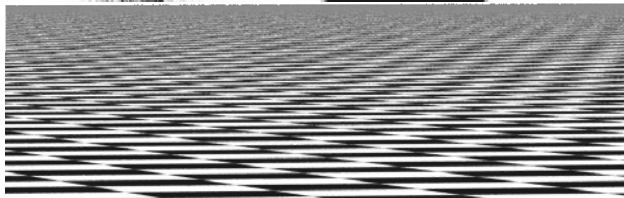
## Comparisons



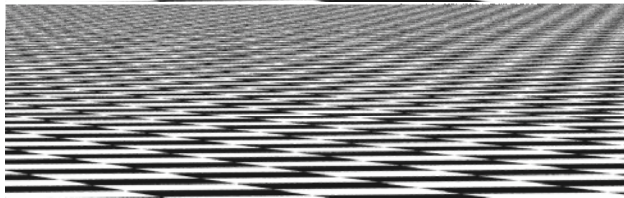
no antialiasing



closed-form  
*less aliasing  
but blurring*



supersampling  
*less blurring  
susceptible to  
aliasing*



## Solid checkboard



```
Checkerboard3D(TextureMapping3D *m,  
    Reference<Texture<T>> c1, Reference<Texture<T>> c2)  
{  
    mapping = m;  
    tex1 = c1;  
    tex2 = c2;  
}  
  
Point p = mapping->Map(dg, &dpx, &dpy);  
if ((Floor2Int(p.x)+Floor2Int(p.y)+ Floor2Int(p.z))  
    % 2 == 0)  
    return tex1->Evaluate(dgs);  
else  
    return tex2->Evaluate(dgs);
```

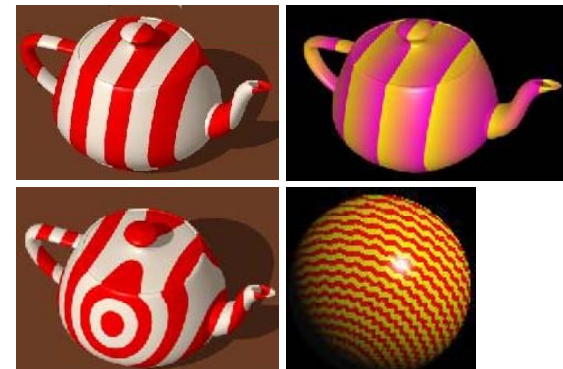
## Solid checkboard



## Some other procedure textures



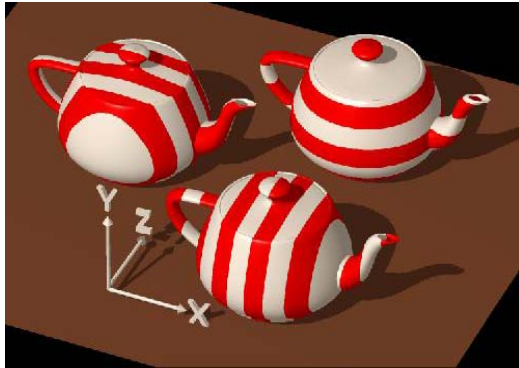
- Stripe
- Ramp and sine
- Rings
- Wood



## Stripes on teapot



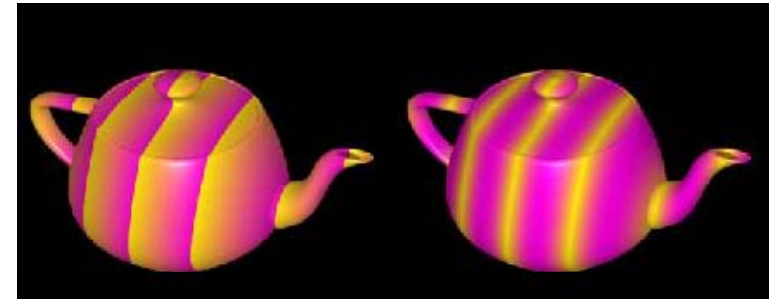
- Find the integer part of the  $x$ -,  $y$ -, or  $z$ -value of each point of the object.
- If resulting value is even,
  - then choose red;
  - else choose white.



## Ramp and Sine Functions



- A nice ramp:  $\text{mod}(x, a)/a$
- This ramp function has a range of zero to one, as does  $(\sin(x)+1)/2$ .

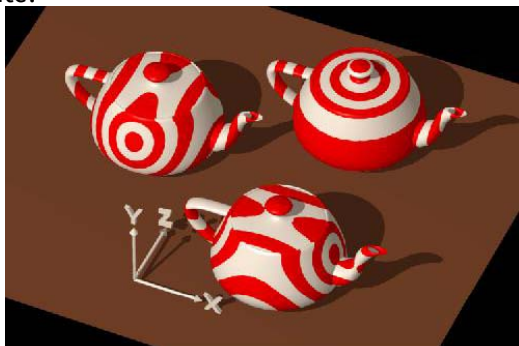


Magenta to value of zero; and yellow to the value one

## Rings on teapot



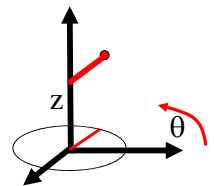
- Use the  $x$ - and  $y$ -components to compute the distance of a point from the object's center, and truncate the result.
- If the resulting value is even,
  - then we choose red;
  - else we choose white.



## Wood grain



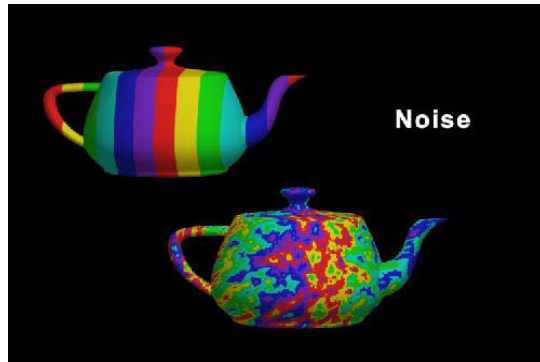
- $r = x^2 + y^2$
- $\text{ring}(r) = (\text{int}(r)) \% 2$
- $\text{Wobble}(r) = \text{ring}(r/M + k \cdot \sin(\theta/N))$



## Noise



- Real-world objects have both regularity and irregularity in nature.

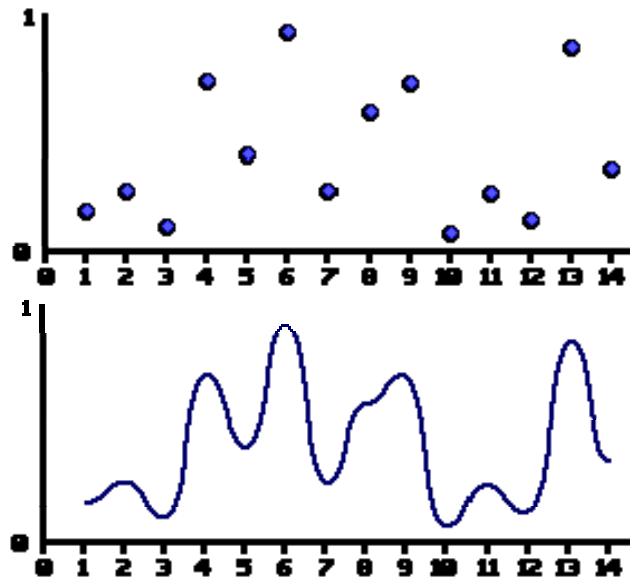


## Perlin noise



- For example, we might want to add “noise” to the stripe function. One possibility would be to add random numbers to the stripe function. This won’t look good because it looks just like white noise in TV.
- We want to make it smoother without losing the random quality.
- Another option is to take a random value at every lattice point and interpolate them. This often makes the lattice too obvious.

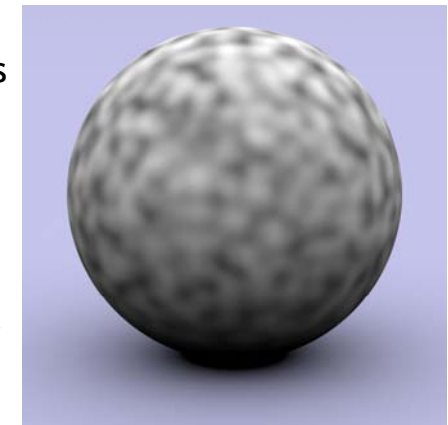
## Smooth out randomness (1D)



## Perlin noise



- Developed by Ken Perlin for TRON (1982)
- [TRON](#) is the first 3D shaded graphics in a Hollywood film
- Perlin used some tricks to improve the basic lattice approach to achieve “**determined and coherent** randomness”
- `float Noise(float x, float y, float z) in pbrt`



## Perlin noise



- Hermite interpolation
- Random vectors instead of random values; values are derived using inner product
- Hashing for randomness

$$n(x, y, z) = \sum_{i=-\lfloor x \rfloor}^{\lfloor x \rfloor+1} \sum_{j=-\lfloor y \rfloor}^{\lfloor y \rfloor+1} \sum_{k=-\lfloor z \rfloor}^{\lfloor z \rfloor+1} \Omega_{ijk}(x-i, y-j, z-k)$$

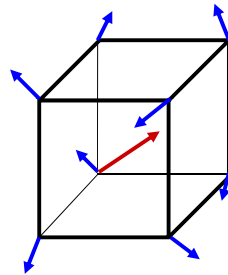
$$\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u, v, w))$$

$$\omega(t) = 2t^3 - 3t^2 + 1 \quad 0 \leq t \leq 1$$

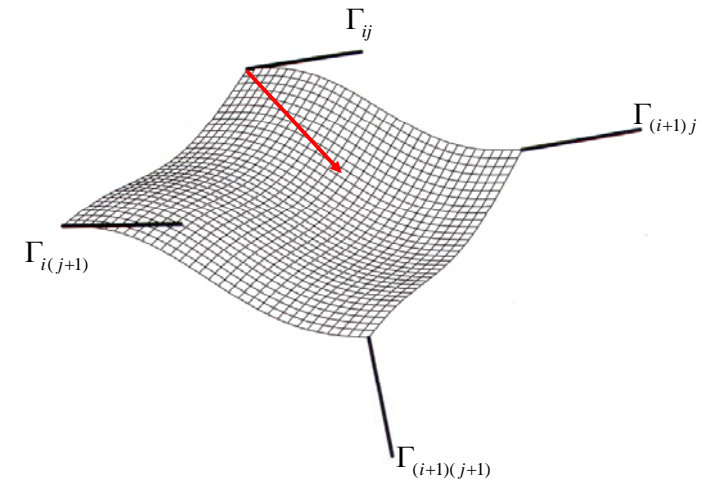
$$\Gamma_{ijk} = G(\phi(i + \phi(j + \phi(k))))$$

$\phi(i) = P[i \% n]$   $P$  is an array of length  $n$  containing a permutation of  $0..n-1$ ;  $n=256$  in practice

$G$  is a precomputed array of  $n$  random unit vectors



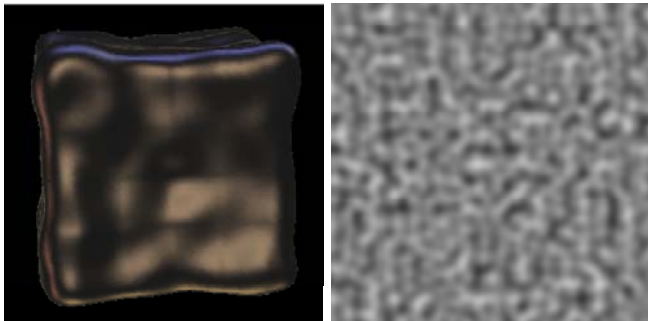
## Perlin noise (2D illustration)



## Improving Perlin noise (Perlin 2002)



- Two deficiencies: 12t-6
  - Interpolation function's 2nd derivative is not zero at either  $t=0$  or  $t=1$ , creating 2nd order discontinuities.
  - Gradient in  $G$  are distributed uniformly over a sphere; but, cubic grid has directional biases



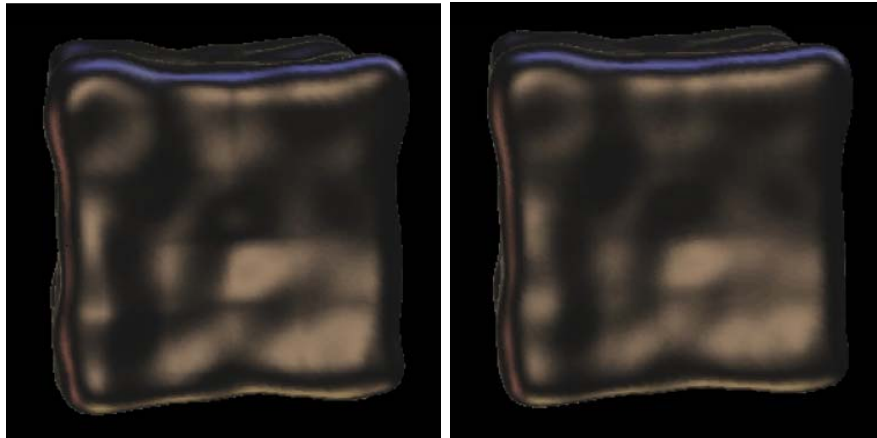
## Improving Perlin noise (Perlin 2002)



- $\omega(t) = 6t^5 - 15t^4 + 10t^3 \quad 0 < t < 1$   $\frac{30t^4 - 60t^3 + 30t^2}{120t^3 - 180t^2 + 60t}$
- Use only 12 vectors, defined by the directions from the center of a cube to its edges. It is not necessary for  $G$  to be random since  $P$  provides plenty of randomness.



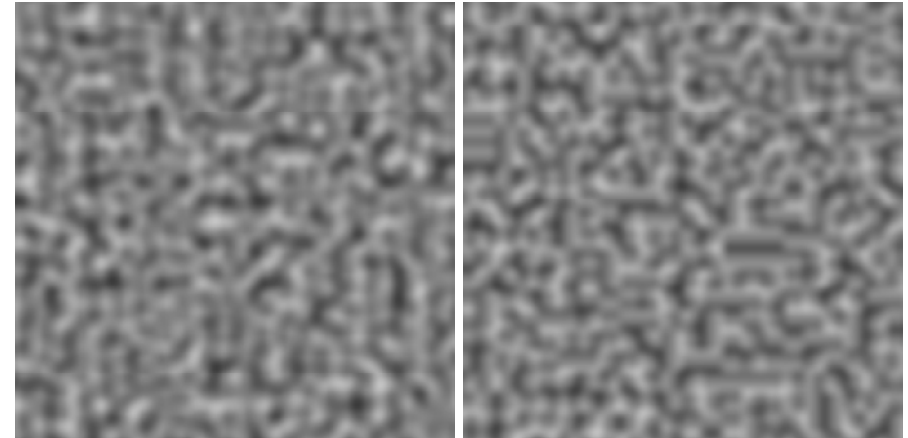
## Improving Perlin noise (Perlin 2002)



$$\omega(t) = 2t^3 - 3t^2 + 1$$

$$\omega(t) = 6t^5 - 15t^4 + 10t^3$$

## Improving Perlin noise (Perlin 2002)



uniformly sampled from a sphere

use only 12 vectors;  
also faster

## Random Polka dots

- Divide texture space into regular cells; each cell has a 50% chance of having a dot inside it.

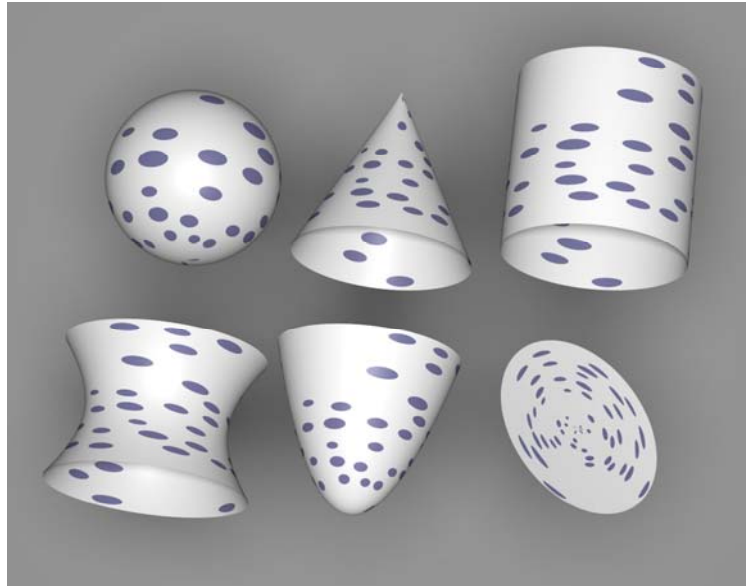
```
DotsTexture(TextureMapping2D *m,  
  Reference<Texture<T>> c1, Reference<Texture<T>> c2)  
{  
  mapping = m;  
  outsideDot = c1;  
  insideDot = c2;  
}
```

## Random Polka dots

```
T Evaluate(const DifferentialGeometry &dg) const {  
  float s, t, dsdx, dtdx, dsdy, dtdy;  
  mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);  
  int sCell = Floor2Int(s + .5f),  
      tCell = Floor2Int(t + .5f);  
  if (Noise(sCell+.5f, tCell+.5f) > 0) {  
    float radius = .35f; It is deterministic so that it makes  
    float maxShift = 0.5f - radius; consistent decision for pixels.  
    float sCenter = sCell + maxShift *  
        Noise(sCell + 1.5f, tCell + 2.8f);  
    float tCenter = tCell + maxShift *  
        Noise(sCell + 4.5f, tCell + 9.8f);  
    float ds = s - sCenter, dt = t - tCenter;  
    if (ds*ds + dt*dt < radius*radius)  
      return insideDot->Evaluate(dg);  
  }  
  return outsideDot->Evaluate(dg);  
}
```



# Random Polka dots



# Multiple-scale Perlin noise



- Many applications would like to have variation over multiple scales.

$$f_s(x) = \sum_i w_i f(s_i x)$$

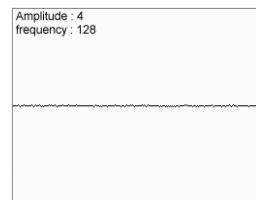
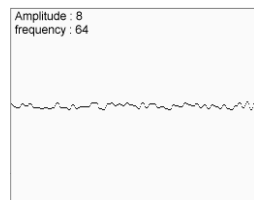
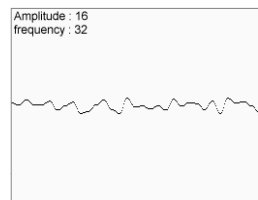
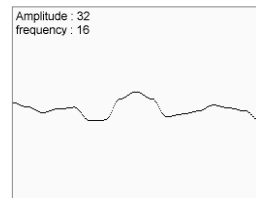
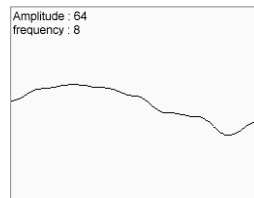
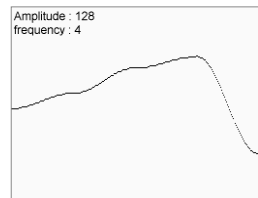
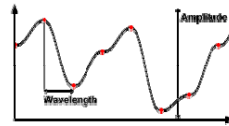
$$w_i = \frac{1}{2} w_{i-1} \quad s_i = 2s_{i-1}$$

- Application of this to Perlin noise leads to “Fractional Brownian motion” (FBm)
- Turblence

$$f_s(x) = \sum_i w_i |f(s_i x)|$$

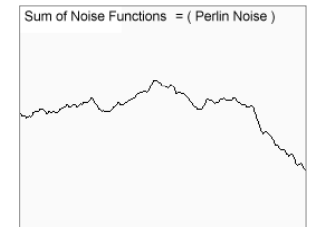
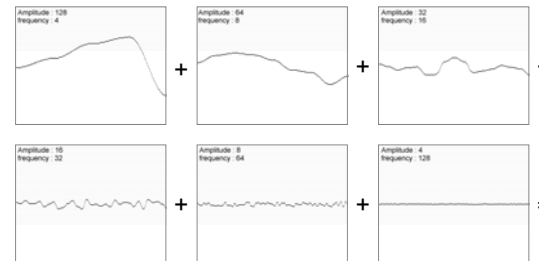
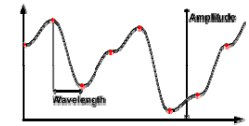
## (1-D) Perlin Noise Function :

- Get lots of such smooth functions, with various frequencies and amplitudes

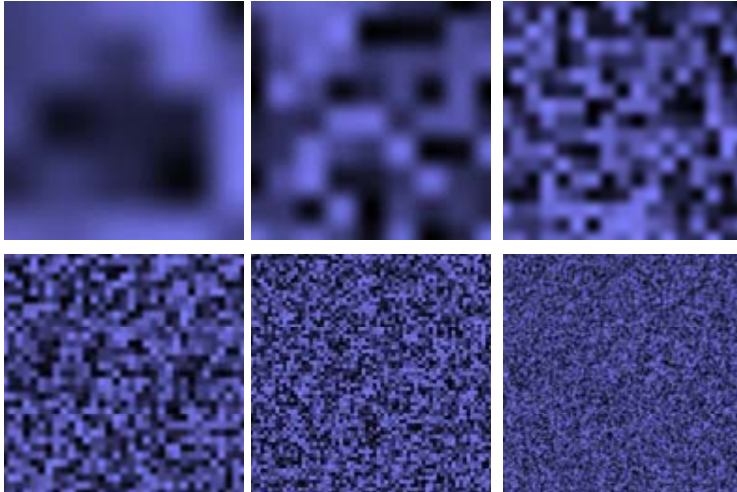


## (1-D) Perlin Noise Function :

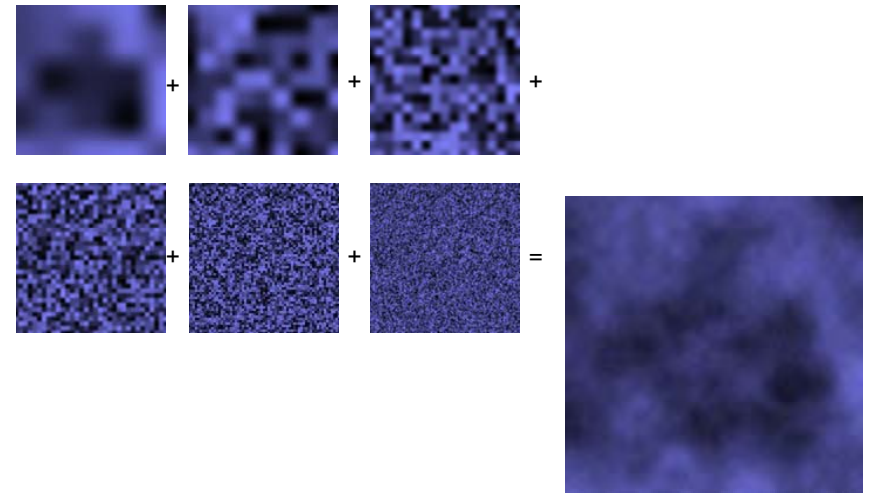
- Get lots of such smooth functions, with various frequencies and amplitudes
- Add them all together to create a nice noisy function



## (2-D) Perlin Noise Function

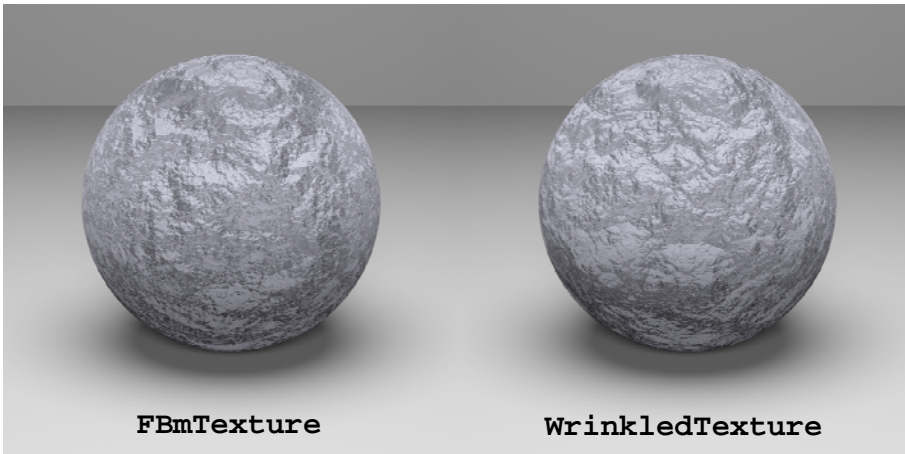


## (2-D) Perlin Noise Function



## Bumpy and wrinkled textures

- `FBmTexture` uses `FBm` to compute offset and `WrinkledTexture` uses `Turbulence` to do so.



## `FBmTexture`

```
FBmTexture(int oct, float roughness,  
           TextureMapping3D *map) {  
    omega = roughness;  
    octaves = oct;  
    mapping = map;  
}  
  
T Evaluate(const DifferentialGeometry &dg){  
    Vector dpdx, dpdy;  
    Point P = mapping->Map(dg, &dpdx, &dpdy);  
    return FBm(P, dpdx, dpdy, omega, octaves);  
}
```

## Windy waves



```
T WindyTexture:Evaluate(DifferentialGeometry &dg) {  
    Vector dpdx, dpdy;  
    Point P = mapping->Map(dg, &dpdx, &dpdy);  
    float windStrength = low frequency for local wind strength  
        Fbm(.1f * P, .1f * dpdx, .1f * dpdy, .5f, 3);  
    float waveHeight = amplitude of wave independent of wind  
        Fbm(P, dpdx, dpdy, .5f, 6);  
    return fabsf(windStrength) * waveHeight;  
}
```

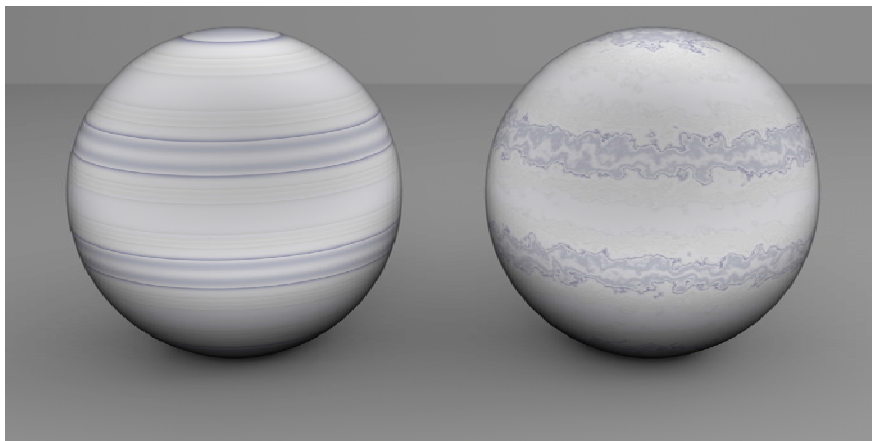
## Windy waves



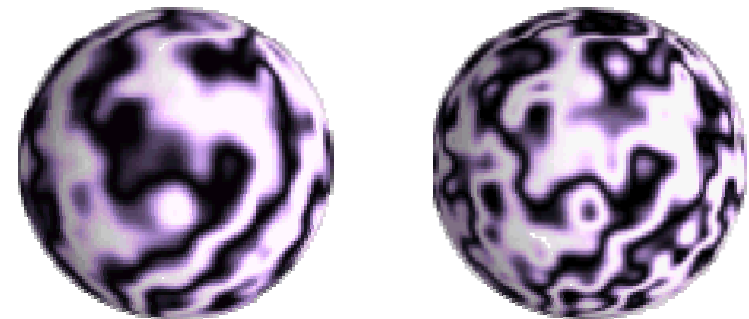
## Marble



- Perturb texture coordinates before using another texture or lookup table

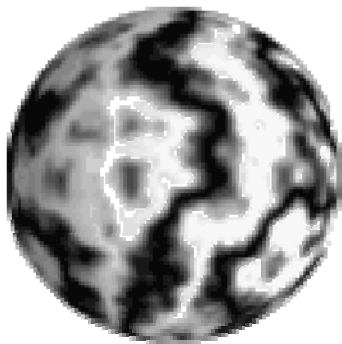


## Texture Generation using 3D Perlin Noise



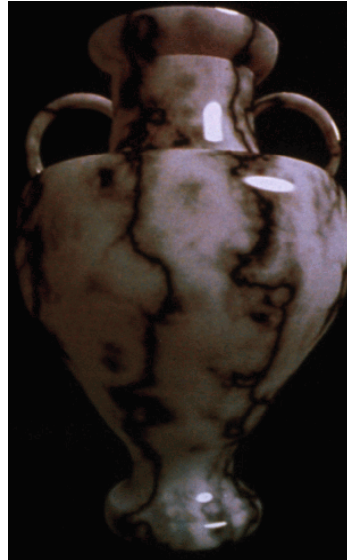
Standard 3 dimensional Perlin noise  
4 octaves, persistence 0.25 and 0.5

## Texture Generation using 3D Perlin Noise



A marble texture can be made by using a Perlin function as an offset to a cosine function.

**texture = cosine( x + perlin(x,y,z)**



## Texture Generation using 3D Perlin Noise

- Very nice wood textures can be defined.
- The grain is defined with a low persistence function like this:

```
g = perlin(x,y,z) * 20  
grain = g - int(g)
```



- The very fine bumps you can see on the wood are high frequency noise that has been stretched in one dimension.

```
bumps = perlin(x*50,y*50,z*20)  
if bumps < .5  
then bumps = 0  
else bumps = 1t
```



## Creative use of Perlin noise

