

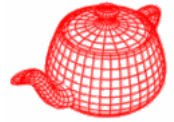
Textures and Materials

Digital Image Synthesis

Yung-Yu Chuang

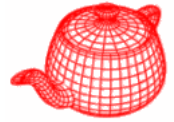
with slides by Pat Hanrahan, Mario Costa Sousa, Robin Chen

Textures



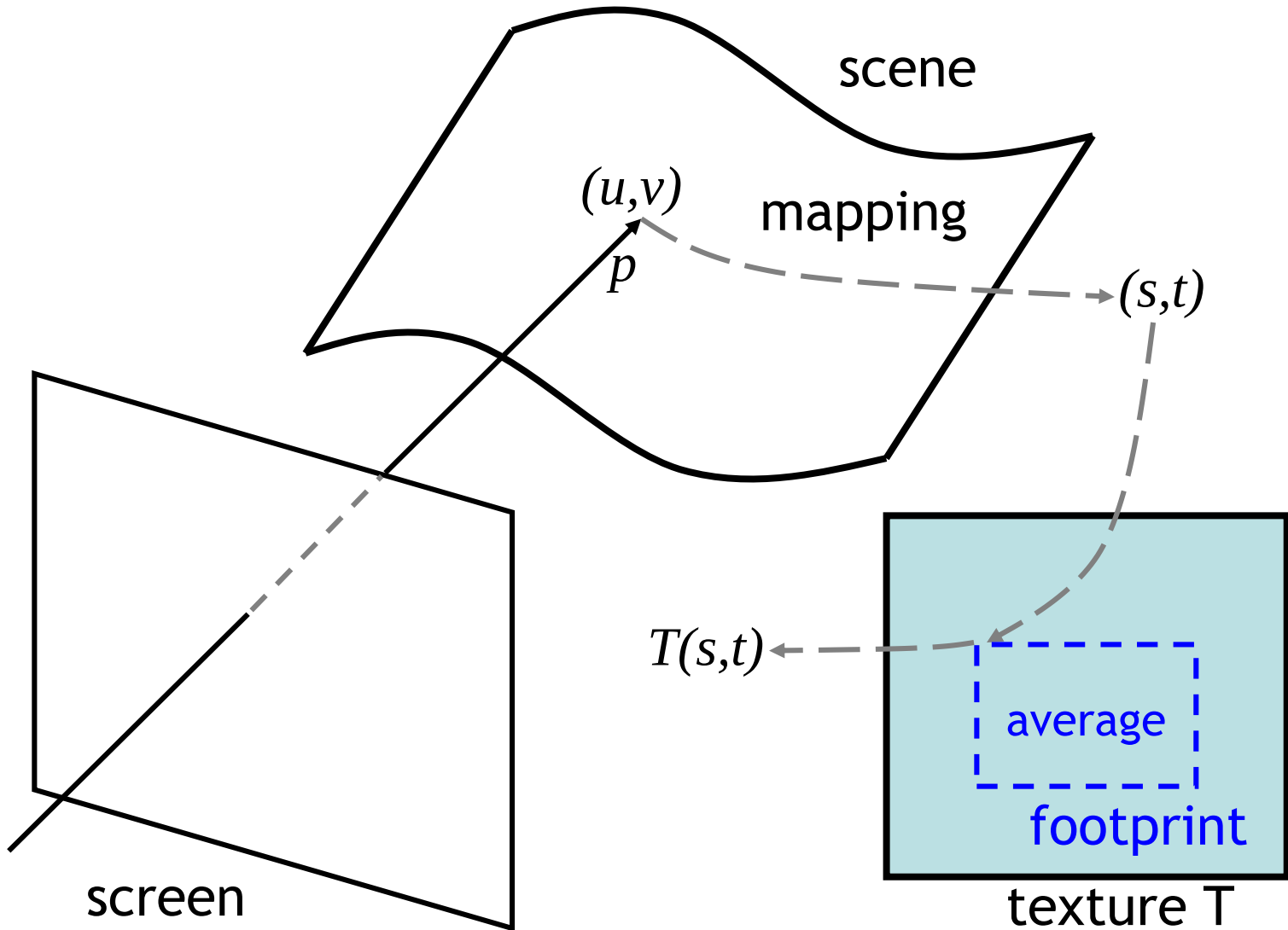
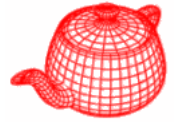
- Recall that materials require parameters to describe their characteristics. These parameters are often spatially varying and textures are used to model such spatial variations.
- A texture is a function that maps points in some domain to values in some other domain.
- In pbrt, pattern generation is separated from material implementation so that it is more flexible.

Texture maps

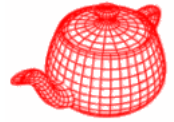


- How is a texture mapped to the surface?
 - Dimensionality: 1D, 2D, 3D
 - Texture coordinates (s,t)
 - Surface parameters (u,v)
 - Projection: spherical, cylindrical, planar
 - Reparameterization
- What can a texture control?
 - Surface color and transparency
 - Illumination: environment maps, shadow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

Texture mapping



Texture

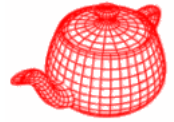


2D
mapping



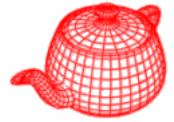
3D
mapping

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)

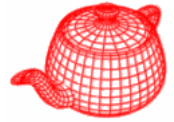
Texture maps



Tom Porter's Bowling Pin



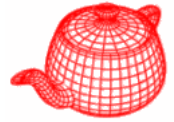
Reflection maps



Blinn and Newell, 1976



Environment maps

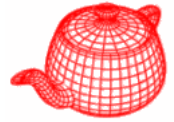


Ray Traced



Environment Map

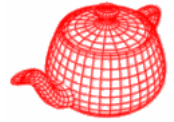
Bump/displacement maps



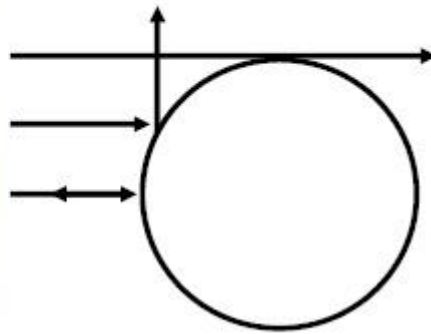
From Blinn 1976



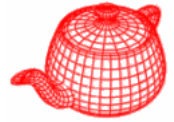
Illumination maps



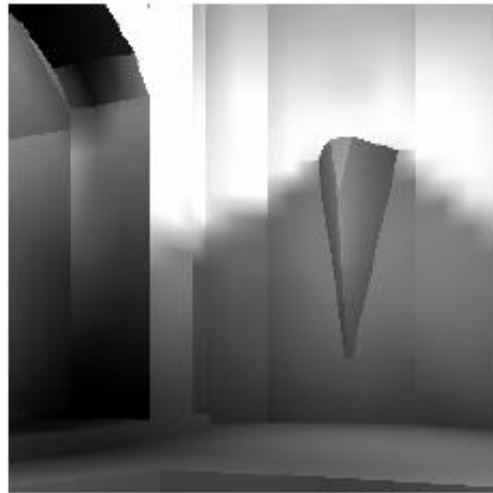
Miller and Hoffman, 1984



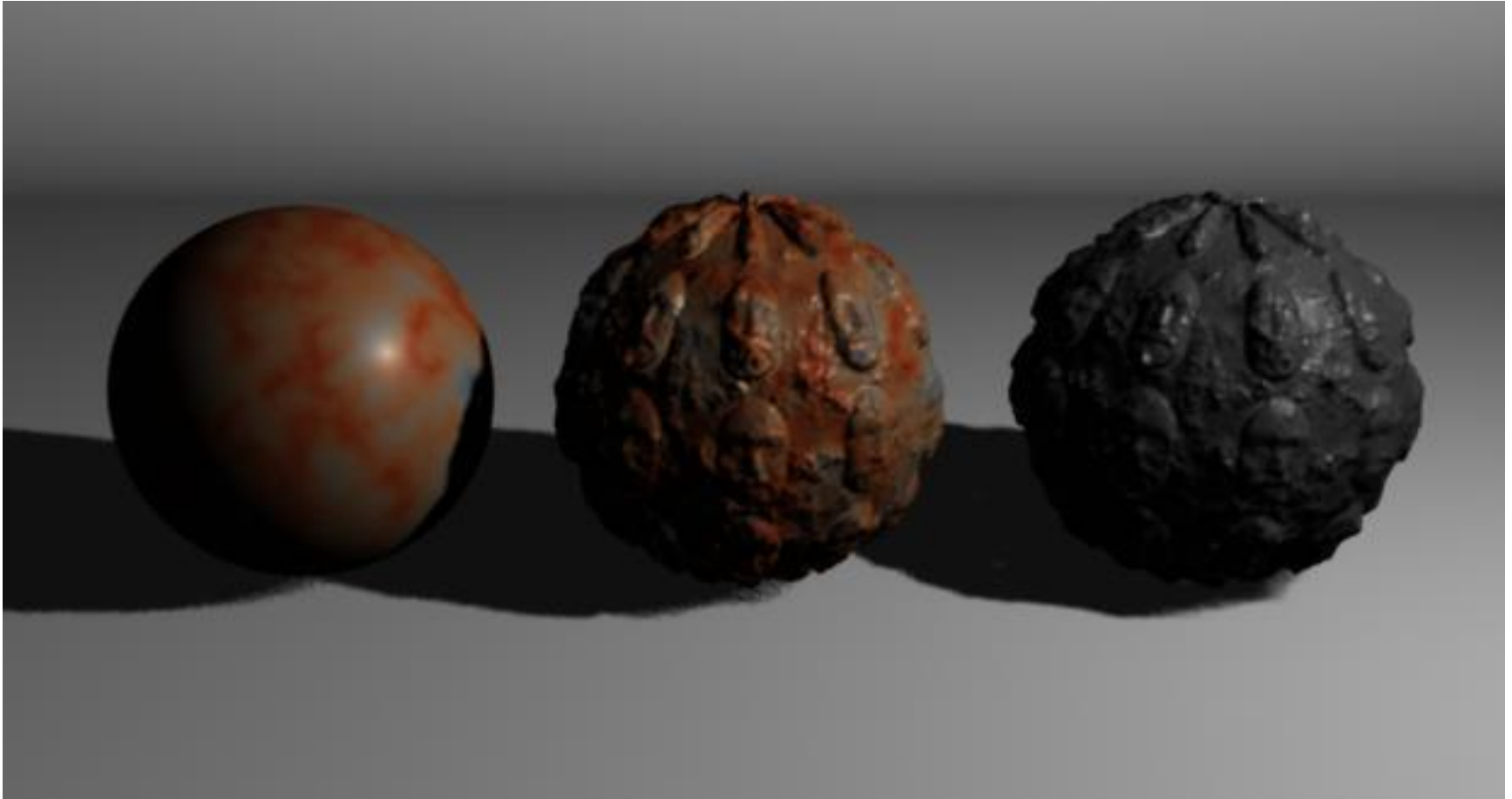
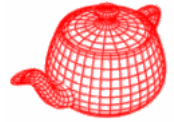
Illumination maps



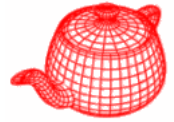
*



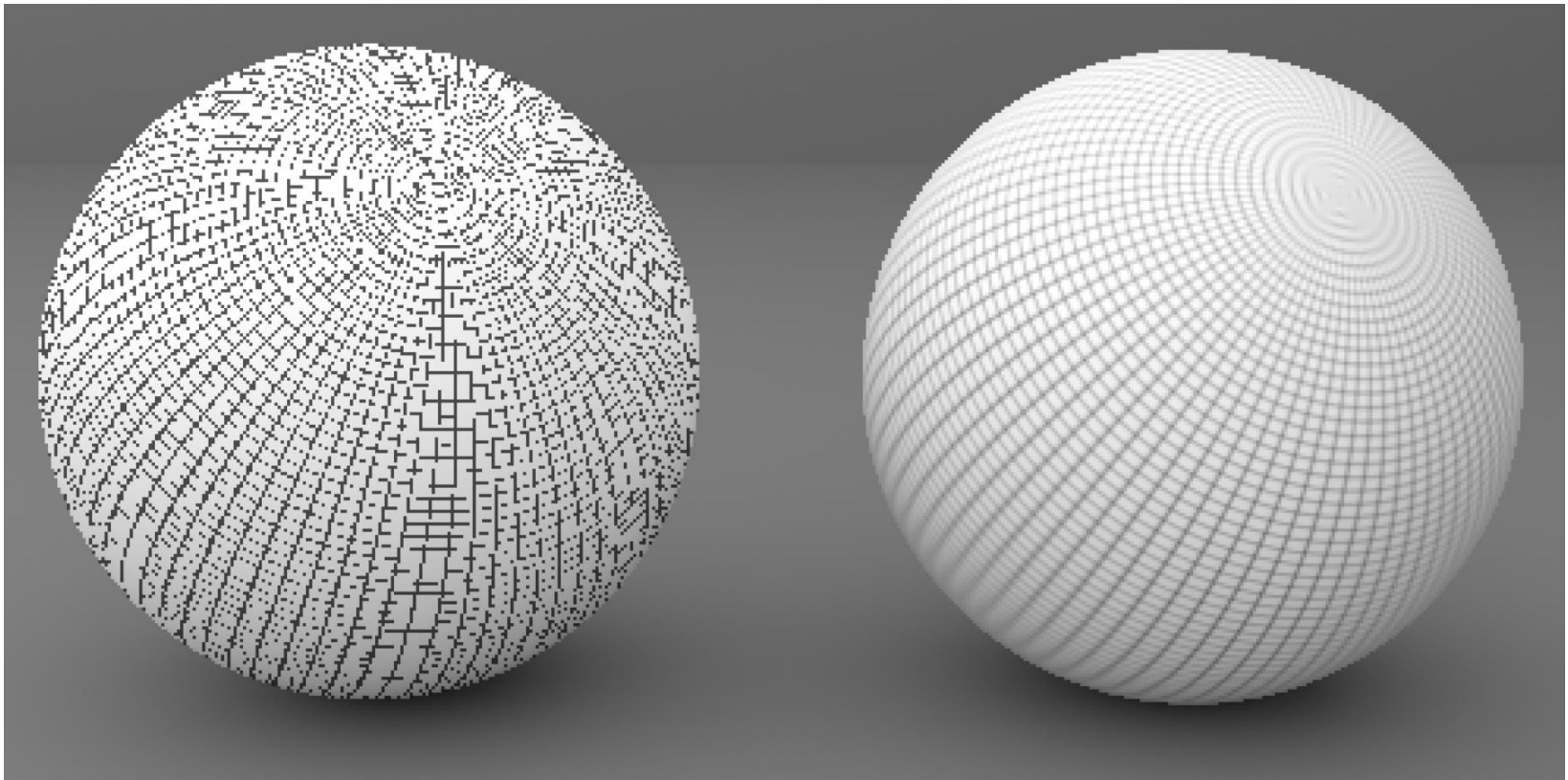
Solid textures



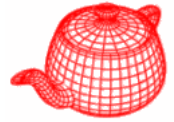
Sampling and antialiasing



- 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.

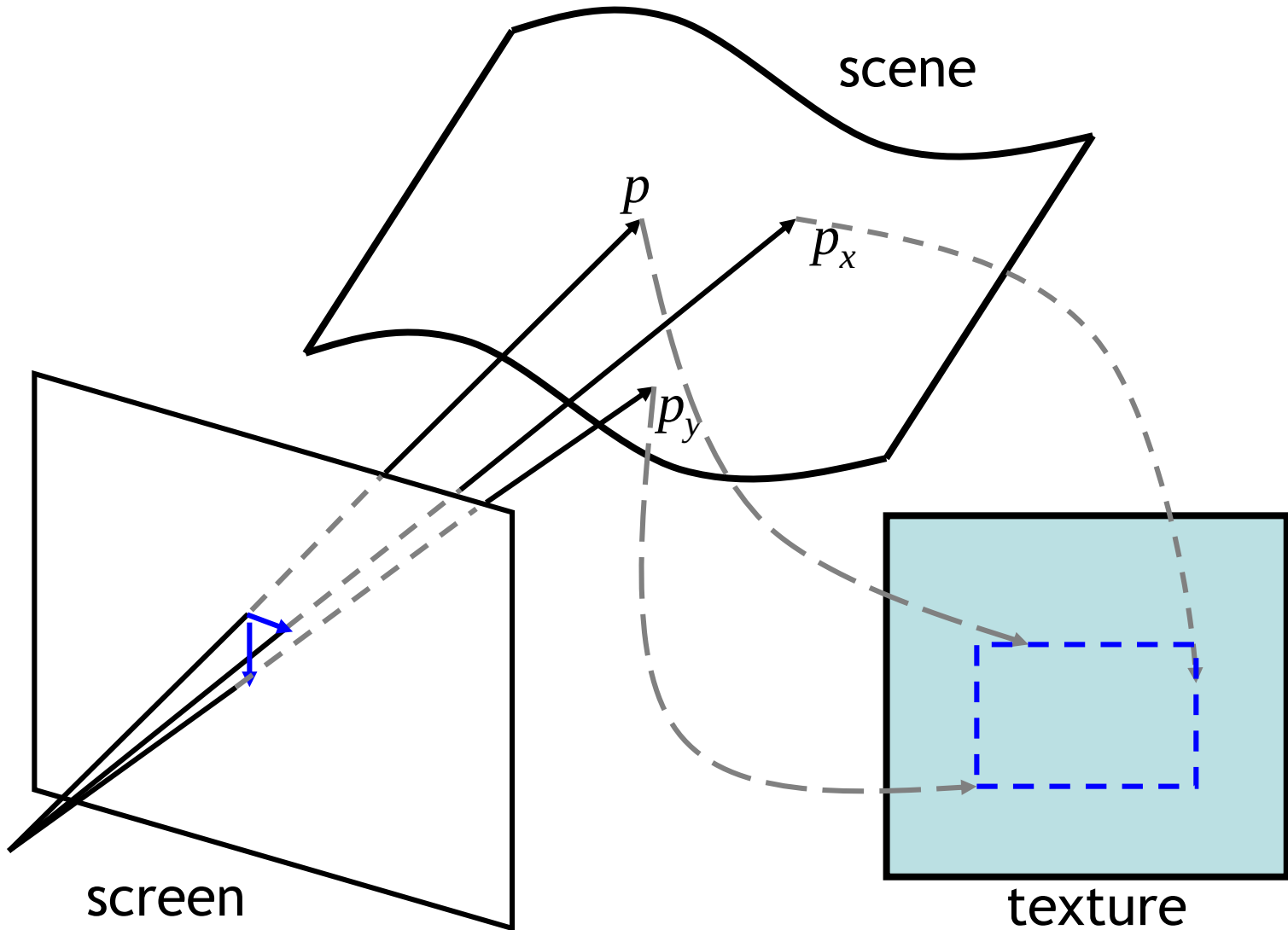
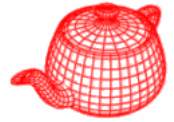


Sampling and antialiasing

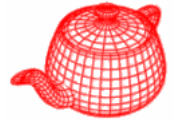


- Antialiasing for textures is not as hopeless as we have seen in the sampling chapter because there are either analytical forms or ways to avoid adding high-frequency
- Two problems that must be addressed
 1. Compute the sampling rate in the texture space
 2. Apply sampling theory to guide texture computation so that no higher frequency variation than can be represented by the sampling rate

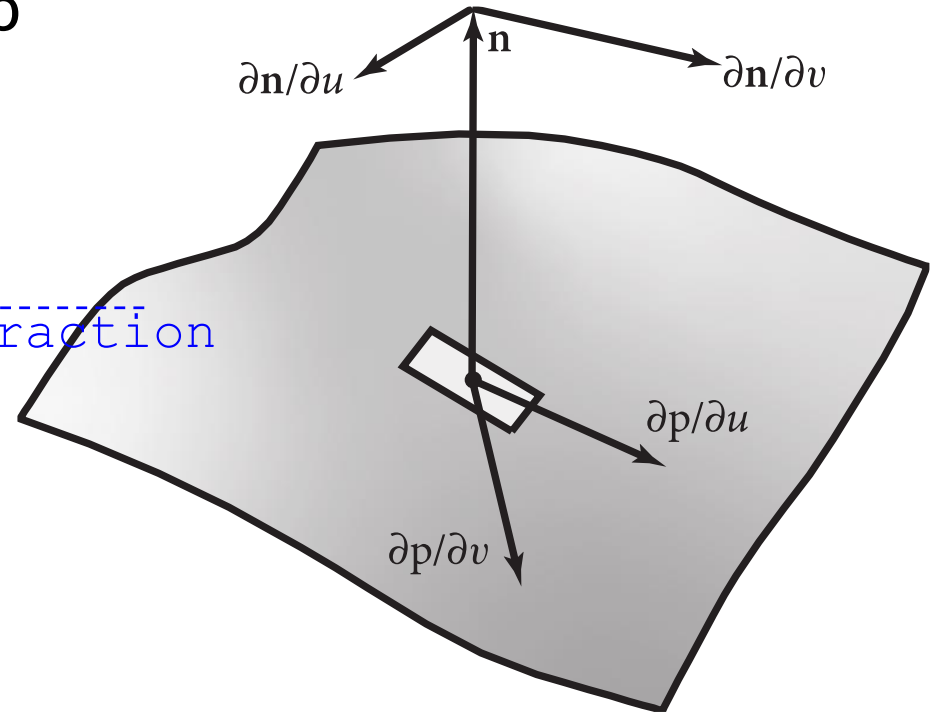
Finding the texture sampling rate



SurfaceInteraction



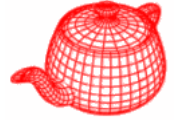
- Position: p (actually an interval)
- Negative ray direction: w_o
- Time t
- Normal: n
- Parameterization (u, v)
- Parametric derivatives
($dp/du, dp/dv$)
- Derivatives of normal
($dn/du, dn/dv$)
- Material, light...



```
Vector3f dpdx, dpdy;  
Float dudx = 0, dvdx = 0, dudy = 0, dvdy = 0;
```

Computed by ComputeDifferentials() called by SurfaceInteraction::GetBSDF()

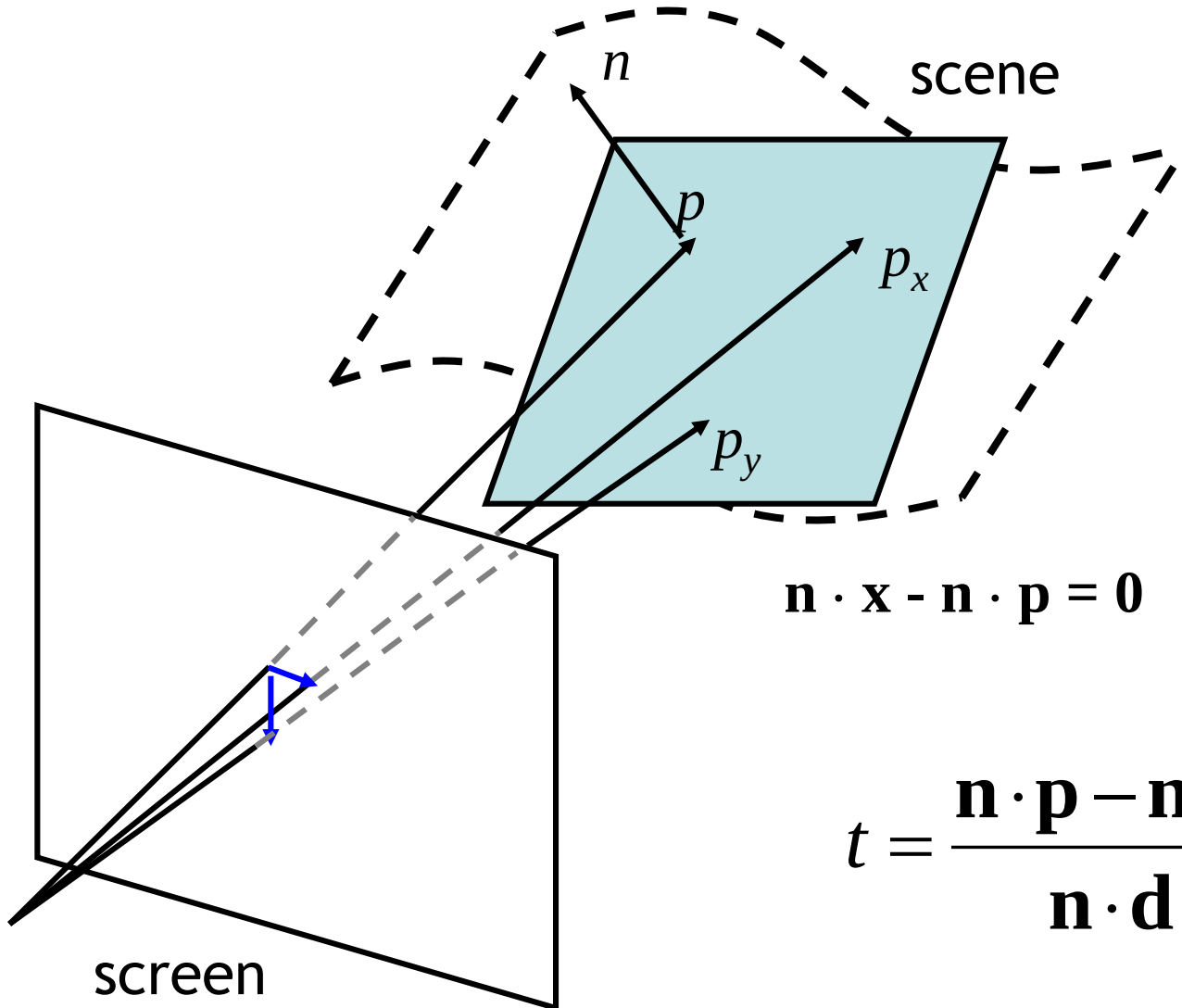
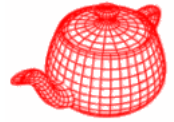
ComputeDifferentials



```
void SurfaceInteraction::ComputeDifferentials(const RayDifferential &ray,
    Camera camera, int samplesPerPixel) {
    if (ray.hasDifferentials && Dot(n, ray.rxDirection) != 0 &&
        Dot(n, ray.ryDirection) != 0) {
        <<Estimate screen-space change in p using ray differentials>> ☒
    } else {
        <<Approximate screen-space change in p based on camera projection>> ☒
        camera.Approximate_dp_dxy(p(), n, time, samplesPerPixel, &dpdx, &dpy);

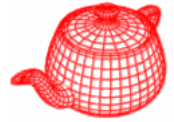
    }
    <<Estimate screen-space change in (u, v)>> ☒
}
```

ComputeDifferentials



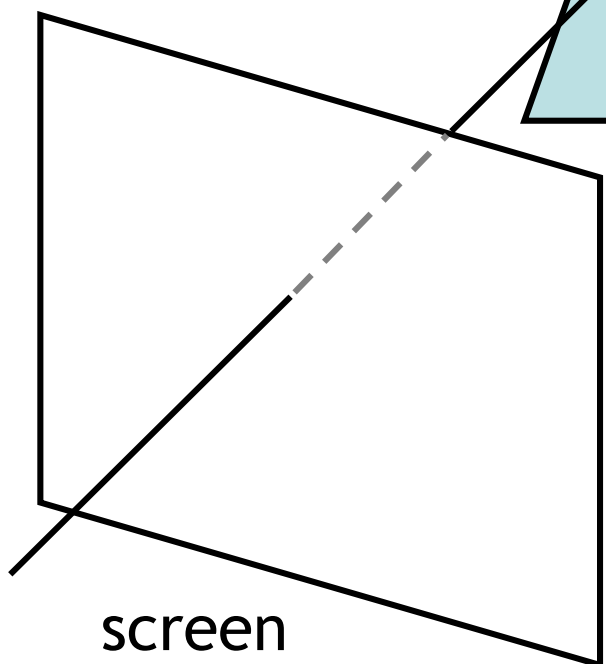
$$t = \frac{\mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{o}}{\mathbf{n} \cdot \mathbf{d}}$$

ComputeDifferentials

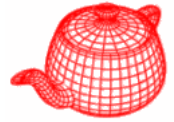


```
Float d = -Dot(n, Vector3f(p()));  
Float tx = (-Dot(n, Vector3f(ray.rxOrigin)) - d) /  
            Dot(n, ray.rxDirection);  
Point3f px = ray.rxOrigin + tx * ray.rxDirection;  
Float ty = (-Dot(n, Vector3f(ray.ryOrigin)) - d) /  
            Dot(n, ray.ryDirection);  
Point3f py = ray.ryOrigin + ty * ray.ryDirection;  
  
dpdx = px - p();  
dpdy = py - p();
```

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

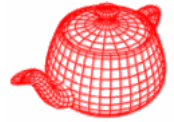


Texture coordinate generation

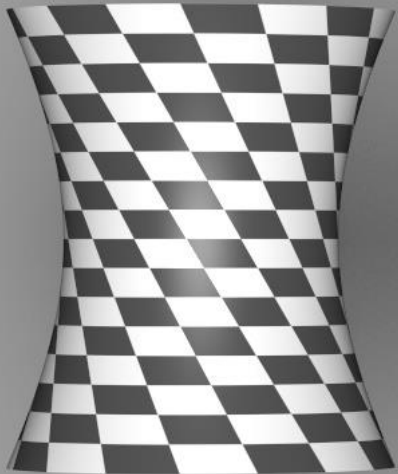


- We use (s,t) for the texture coordinate and (u,v) for the parametric coordinate.
- Creating smooth parameterization of complex meshes with low distortion is still an active research area in graphics.
- Two classes, **TextureMapping2D** and **TextureMapping3D**, provide an interface for computing these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

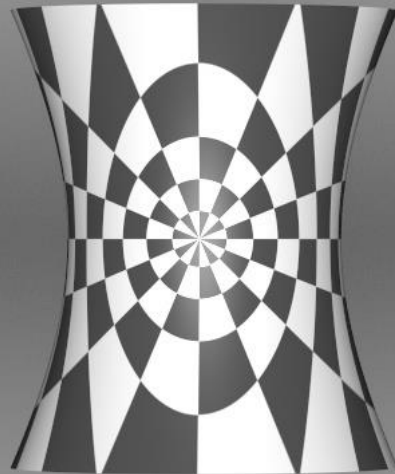
Texture coordinate generation



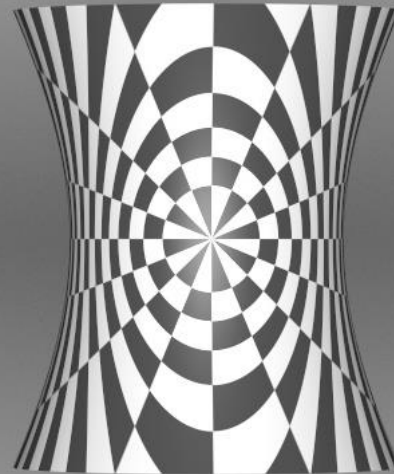
```
class TextureMapping2D
    : public TaggedPointer<UVMapping, SphericalMapping,
                          CylindricalMapping, PlanarMapping> {
public:
    <<TextureMapping2D Interface>> ☒
};
```



(u,v)



spherical

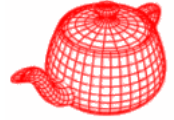


cylindrical



planar

Texture coordinate generation



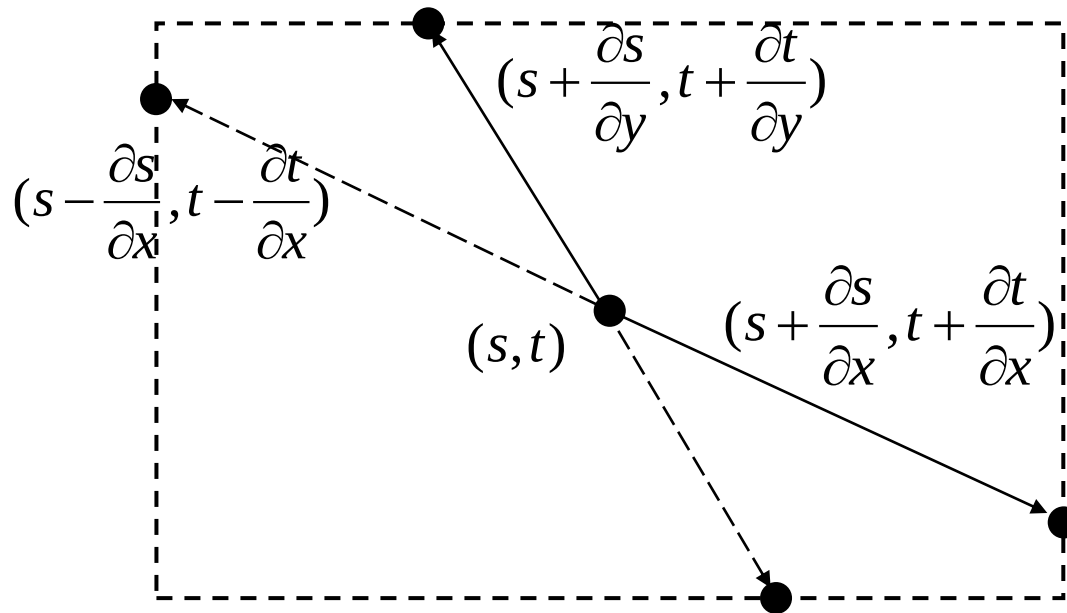
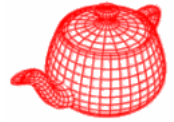
```
struct TexCoord2D {  
    Point2f st;  
    Float dsdx, dsdy, dtdx, dtdy;  
};
```

```
TexCoord2D Map(TextureEvalContext ctx) const;
```

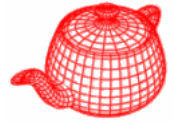
```
struct TextureEvalContext {  
    <<TextureEvalContext Public Methods>>   
    Point3f p;  
    Vector3f dpdx, dpdy;  
    Normal3f n;  
    Point2f uv;  
    Float dudx = 0, dudy = 0, dvdx = 0, dvdy = 0;  
};
```

The subset of the SurfaceInteraction that is necessary for texture mapping. Faster for GPUs because of less memory traffic.

Estimate footprint



(u.v)-mapping



```
class UVMapping {  
    public:  
        <<UVMapping Public Methods>>  
    private:  
        Float su, sv, du, dv;  
};          scale    offset
```

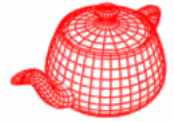
$$s = s_u u + d_u$$

$$t = s_v v + d_v$$

$$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial u} \frac{\partial u}{\partial x} + \frac{\partial s}{\partial v} \frac{\partial v}{\partial x} = s_u \frac{\partial u}{\partial x}$$

```
TexCoord2D Map(TextureEvalContext ctx) const {  
    <<Compute texture differentials for 2D (u, v) mapping>>   
        Float dsdx = su * ctx.dudx, dsdy = su * ctx.dudy;  
        Float dtdx = sv * ctx.dvdx, dtdy = sv * ctx.dvdy;  
  
    Point2f st(su * ctx.uv[0] + du, sv * ctx.uv[1] + dv);  
    return TexCoord2D{st, dsdx, dsdy, dtdx, dtdy};  
}
```

(u,v)-mapping



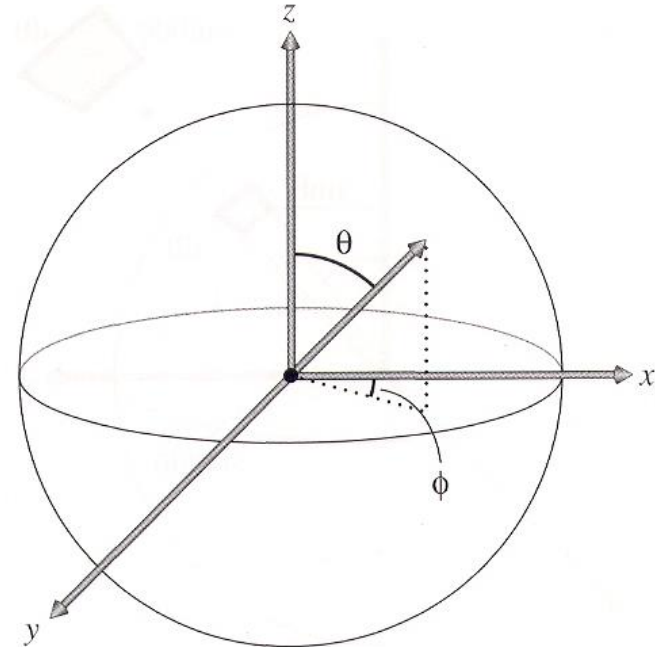
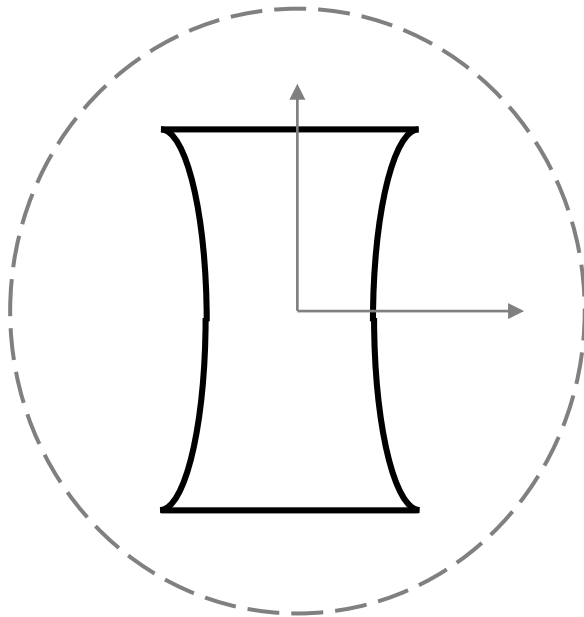
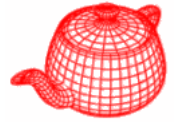
The general form:

$$\frac{\partial(s, t)}{\partial(x, y)} = \begin{pmatrix} \frac{ds}{dx} & \frac{ds}{dy} \\ \frac{dt}{dx} & \frac{dt}{dy} \end{pmatrix} = \begin{pmatrix} \frac{\partial s}{\partial(u,v)} \cdot \frac{\partial(u,v)}{\partial x} & \frac{\partial s}{\partial(u,v)} \cdot \frac{\partial(u,v)}{\partial y} \\ \frac{\partial t}{\partial(u,v)} \cdot \frac{\partial(u,v)}{\partial x} & \frac{\partial t}{\partial(u,v)} \cdot \frac{\partial(u,v)}{\partial y} \end{pmatrix}$$

For (u,v)-mapping: $f(u, v) = (s_u u + d_u, s_v v + d_v)$

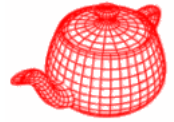
$$\frac{\partial(s, t)}{\partial(x, y)} = \begin{pmatrix} (s_u, 0) \cdot \frac{\partial(u,v)}{\partial x} & (s_u, 0) \cdot \frac{\partial(u,v)}{\partial y} \\ (0, s_v) \cdot \frac{\partial(u,v)}{\partial x} & (0, s_v) \cdot \frac{\partial(u,v)}{\partial y} \end{pmatrix} = \begin{pmatrix} s_u \frac{du}{dx} & s_u \frac{du}{dy} \\ s_v \frac{dv}{dx} & s_v \frac{dv}{dy} \end{pmatrix}$$

Spherical mapping



$$f(p) = \left(\frac{1}{2\pi} \left(\pi + \arctan \frac{p_y}{p_x} \right), \frac{1}{\pi} \arccos \frac{p_z}{\|p_x^2 + p_y^2 + p_z^2\|} \right)$$

Spherical mapping



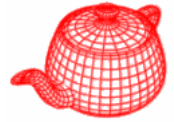
```
class SphericalMapping {  
    public:  
        <<SphericalMapping Public Methods>>   
    private:  
        <<SphericalMapping Private Members>>   
        Transform textureFromRender;  
};
```

$$\frac{\partial(s, t)}{\partial(x, y)} = \begin{pmatrix} \frac{\partial s}{\partial p} \cdot \frac{\partial p}{\partial x} & \frac{\partial s}{\partial p} \cdot \frac{\partial p}{\partial y} \\ \frac{\partial t}{\partial p} \cdot \frac{\partial p}{\partial x} & \frac{\partial t}{\partial p} \cdot \frac{\partial p}{\partial y} \end{pmatrix} = \begin{pmatrix} \frac{ds}{dx} & \frac{ds}{dy} \\ \frac{dt}{dx} & \frac{dt}{dy} \end{pmatrix}$$

$$\frac{\partial s}{\partial p} = \frac{1}{2\pi(x^2 + y^2)}(-y, x, 0)$$

$$\frac{\partial t}{\partial p} = \frac{1}{\pi(x^2 + y^2 + z^2)} \left(\frac{xz}{\sqrt{x^2 + y^2}}, \frac{yz}{\sqrt{x^2 + y^2}}, -\sqrt{x^2 + y^2} \right)$$

Spherical mapping

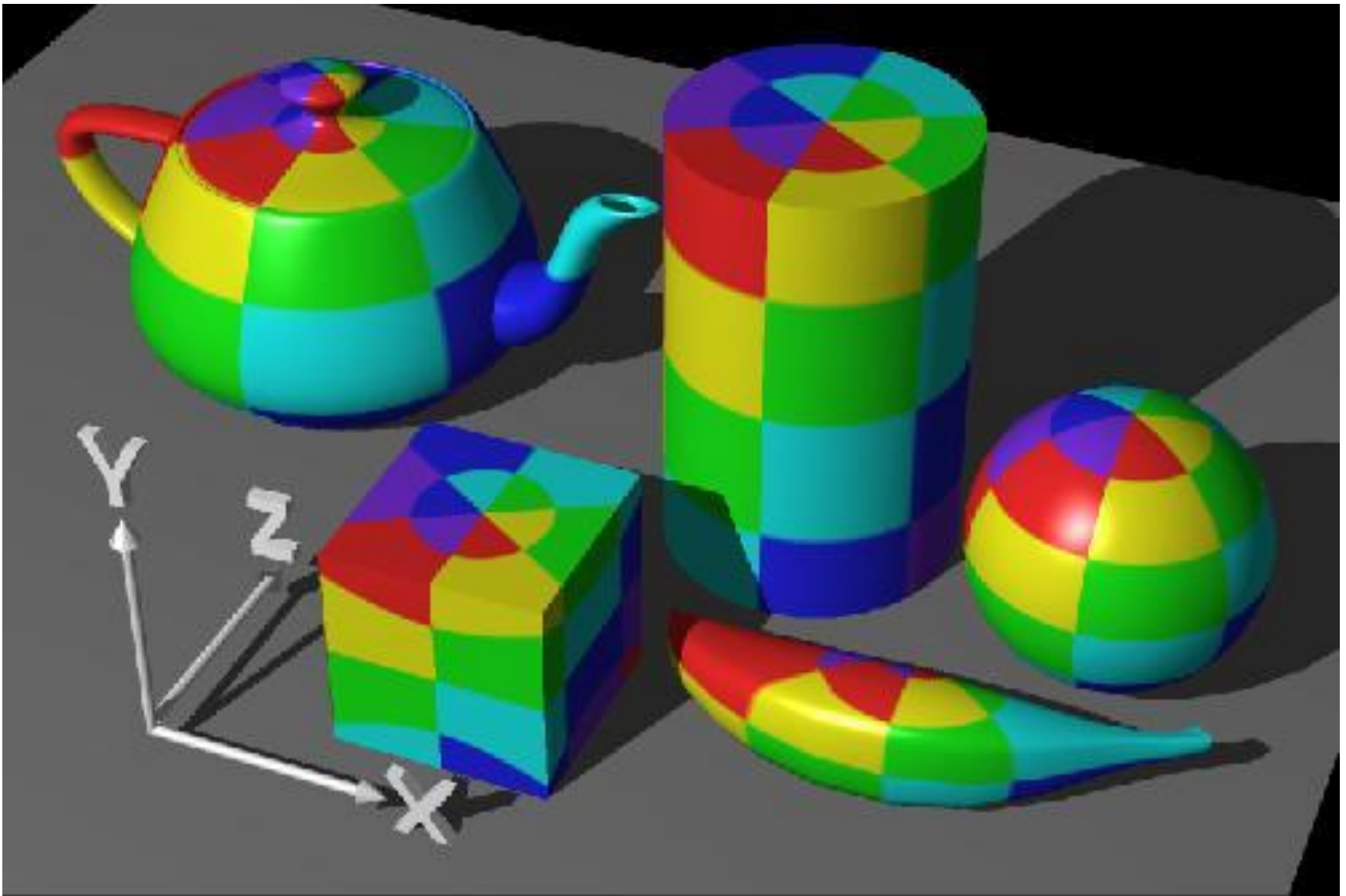
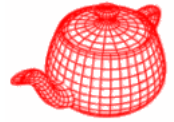


```
TexCoord2D Map(TextureEvalContext ctx) const {
    Point3f pt = textureFromRender(ctx.p);
    <<Compute  $\partial s / \partial p$  and  $\partial t / \partial p$  for spherical mapping>> ⊕
    <<Compute texture coordinate differentials for spherical mapping>> ⊕
    <<Return  $(s, t)$  texture coordinates and differentials based on spherical mapping>>
}
    <<Compute  $\partial s / \partial p$  and  $\partial t / \partial p$  for spherical mapping>>=
    Float x2y2 = Sqr(pt.x) + Sqr(pt.y);
    Float sqrtx2y2 = std::sqrt(x2y2);
    Vector3f dsdp = Vector3f(-pt.y, pt.x, 0) / (2 * Pi * x2y2);
    Vector3f dtdp = 1 / (Pi * (x2y2 + Sqr(pt.z))) *
        Vector3f(pt.x * pt.z / sqrtx2y2, pt.y * pt.z / sqrtx2y2, -sqrtx2y2);

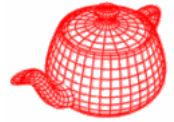
    <<Compute texture coordinate differentials for spherical mapping>>=
    Vector3f dpdx = textureFromRender(ctx.dpdx);
    Vector3f dpdy = textureFromRender(ctx.dpdy);
    Float dsdx = Dot(dsdp, dpdx), dsdy = Dot(dsdp, dpdy);
    Float dtdx = Dot(dtdp, dpdx), dtdy = Dot(dtdp, dpdy);

    <<Return  $(s, t)$  texture coordinates and differentials based on spherical mapping>>=
    Vector3f vec = Normalize(pt - Point3f(0,0,0));
    Point2f st(SphericalTheta(vec) * InvPi, SphericalPhi(vec) * Inv2Pi);
    return TexCoord2D{st, dsdx, dsdy, dtdx, dtdy};
```

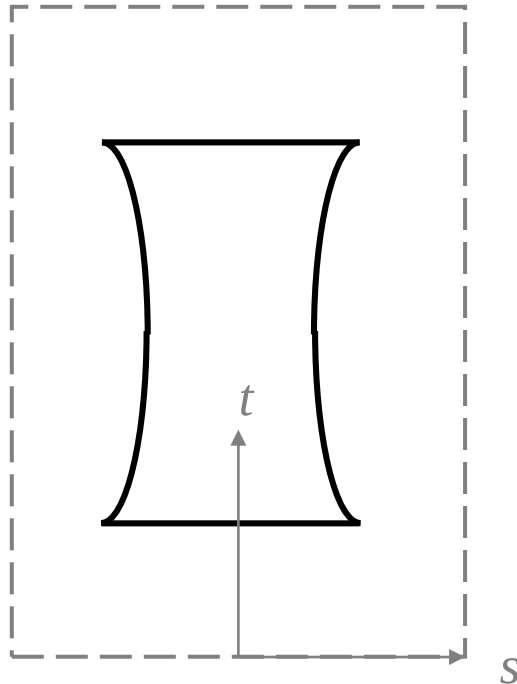
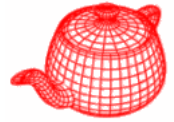
Spherical mapping



Spherical mapping



Cylindrical mapping



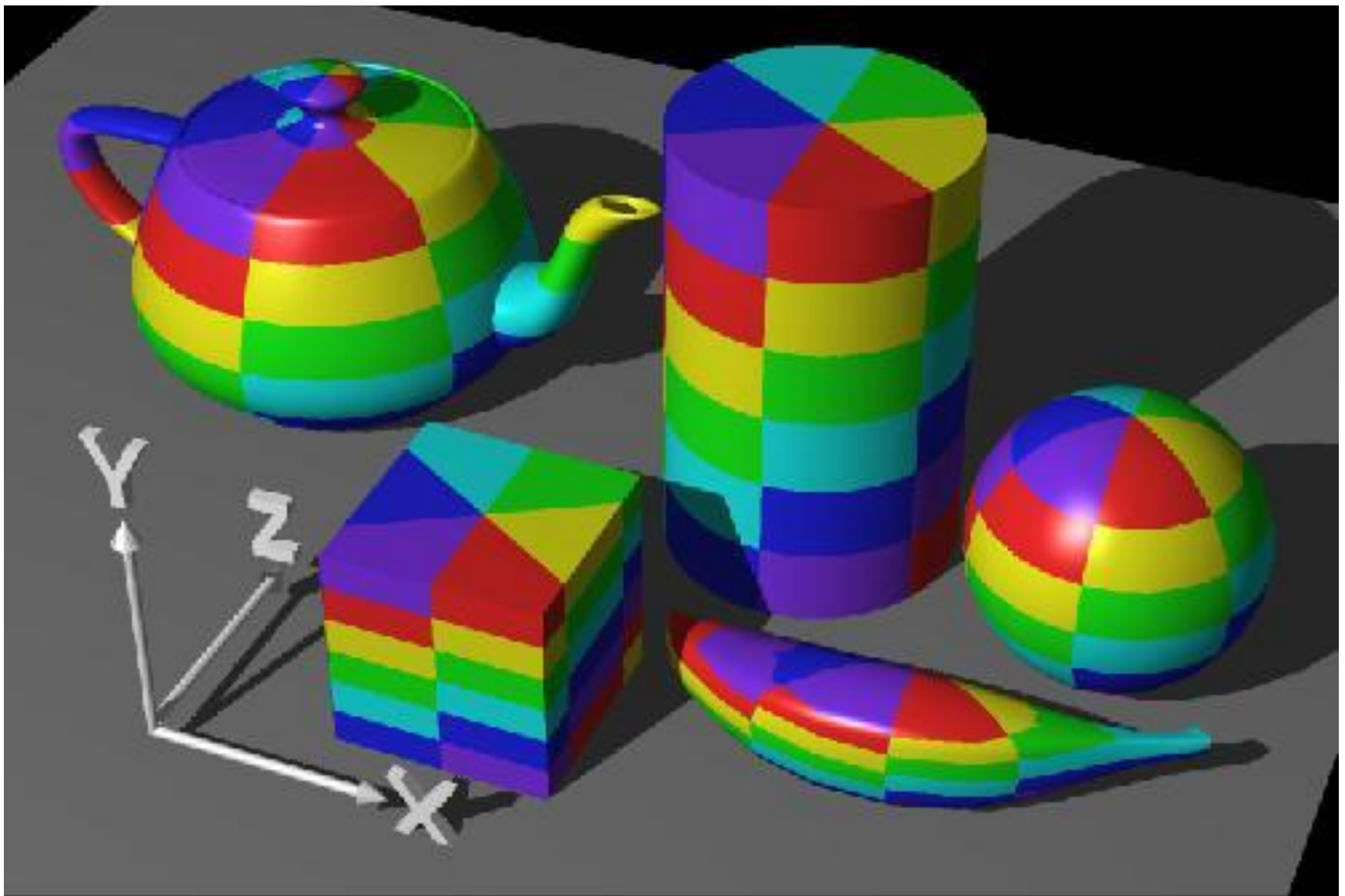
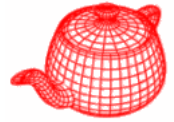
$$f(p) = \left(\frac{1}{2\pi} \left(\pi + \arctan \frac{p_y}{p_x} \right), p_z \right)$$

t is not within $[0, 1]$;
should be scaled in z or the texture
should return results outside $[0, 1]$

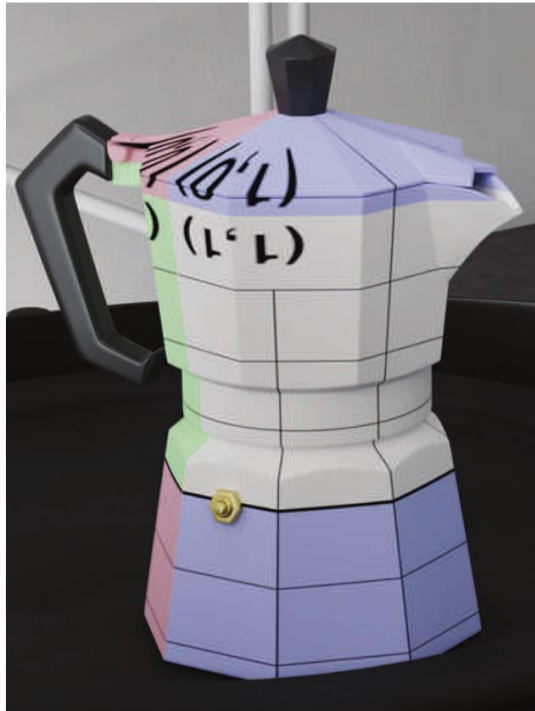
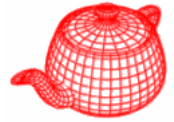
$$\frac{\partial s}{\partial p} = \frac{1}{2\pi(x^2 + y^2)} (-y, x, 0)$$

$$\partial t / \partial p = (0, 0, 1)$$

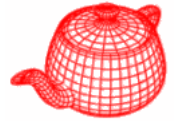
Cylindrical mapping



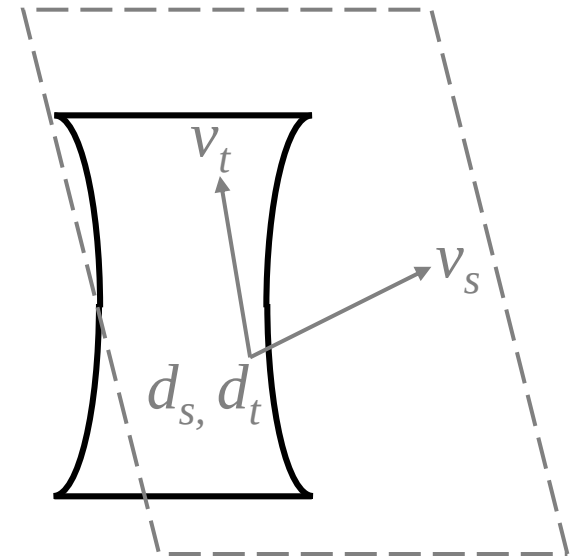
Cylindrical mapping



Planar mapping



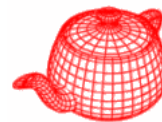
```
class PlanarMapping {  
    public:  
        <<PlanarMapping Public Methods>>   
    private:  
        <<PlanarMapping Private Members>>   
        Transform textureFromRender;  
        Vector3f vs, vt;  
        Float ds, dt;  
};
```



$$f(p) = ((p - (0, 0, 0)) \cdot \mathbf{v}_s + d_s, (p - (0, 0, 0)) \cdot \mathbf{v}_t + d_t)$$

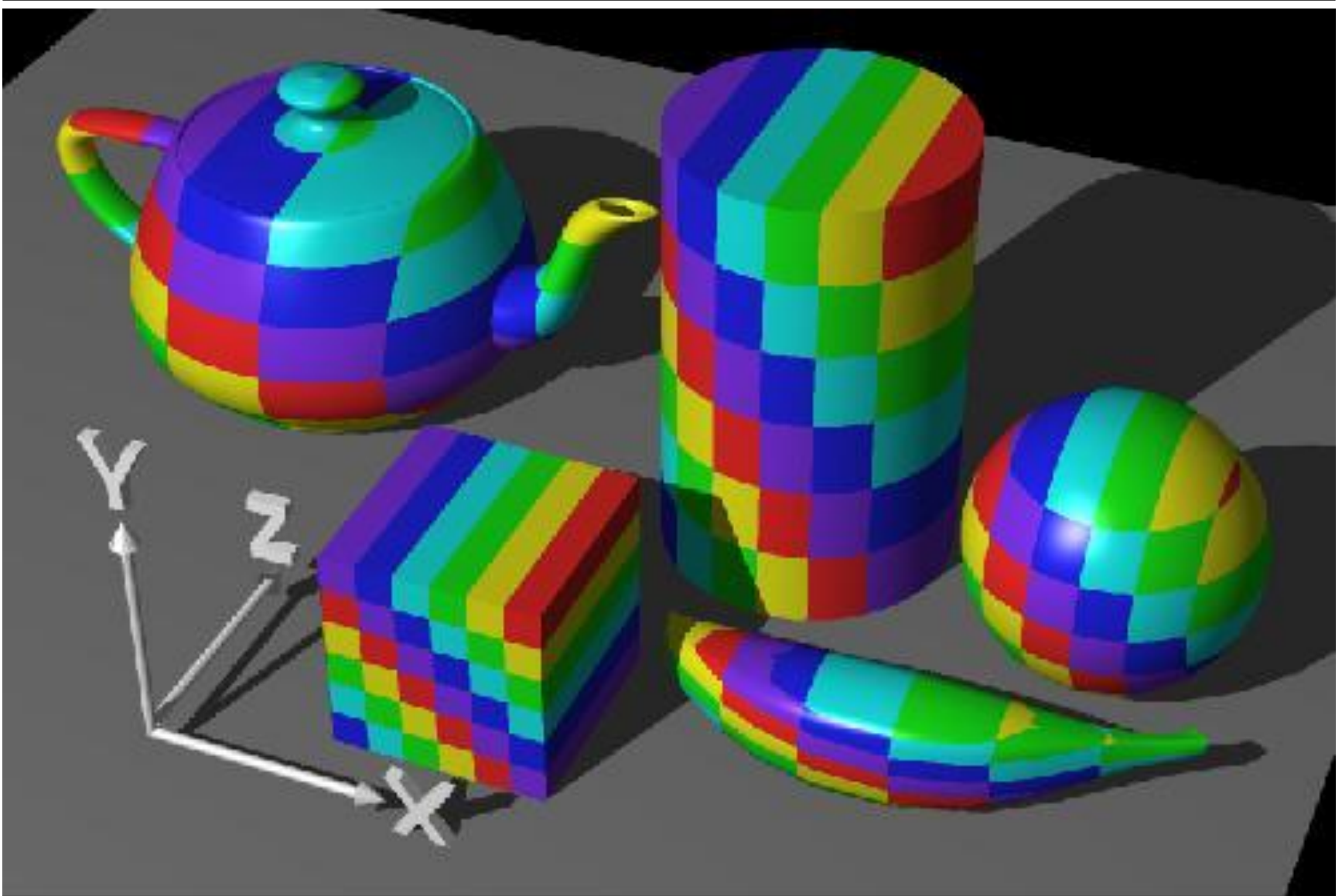
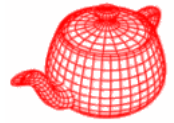
Example, for $z=0$ plane,
 $\mathbf{v}_s=(1,0,0)$, $\mathbf{v}_t=(0,1,0)$, $d_s=d_t=0$

Planar mapping

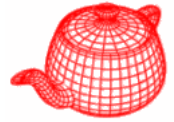


```
TexCoord2D Map(TextureEvalContext ctx) const {  
    Vector3f vec(textureFromRender(ctx.p));  
    <<Initialize partial derivatives of planar mapping( $s, t$ ) coordinates>>  
    Vector3f dpdx = textureFromRender(ctx.dpdx);  
    Vector3f dpdy = textureFromRender(ctx.dpdy);  
    Float dsdx = Dot(vs, dpdx), dsdy = Dot(vs, dpdy);  
    Float dtdx = Dot(vt, dpdx), dtdy = Dot(vt, dpdy);  
  
    Point2f st(ds + Dot(vec, vs), dt + Dot(vec, vt));  
    return TexCoord2D{st, dsdx, dsdy, dtdx, dtdy};  
}
```

Planar mapping



3D mapping

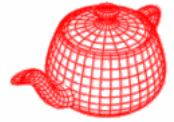


```
class TextureMapping3D : public TaggedPointer<PointTransformMapping> {  
    public:  
        <<TextureMapping3D Interface>>   
};  
  
TexCoord3D Map(TextureEvalContext ctx) const;  
  
struct TexCoord3D {  
    Point3f p;  
    Vector3f dpdx, dpdy;  
};
```



```
TexCoord3D Map(TextureEvalContext ctx) const {
    return TexCoord3D{textureFromRender(ctx.p), textureFromRender(ctx.dpx),
                     textureFromRender(ctx.dpy)};
}
```


ConstantTexture



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

Simple but useful, in particular for spatially-invariant cases

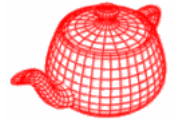
```
class FloatConstantTexture {  
    public:  
        FloatConstantTexture(Float value) : value(value) {}  
        Float Evaluate(TextureEvalContext ctx) const { return value; }  
    private:  
        Float value;  
};
```

SpectrumConstantTexture is defined similarly.

<<SpectrumConstantTexture Public Methods>>=

```
SampledSpectrum Evaluate(TextureEvalContext ctx,  
                        SampledWavelengths lambda) const {  
    return value.Sample(lambda);  
}
```

ScaleTexture

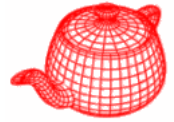


Product of two textures. Leave the anti-aliasing work to the underlining textures without making an effort to anti-alias their product.

```
class FloatScaledTexture {
    public:
        <<FloatScaledTexture Public Methods>>
    private:
        FloatTexture tex, scale;
};

Float Evaluate(TextureEvalContext ctx) const {
    Float sc = scale.Evaluate(ctx);
    if (sc == 0) return 0;
    return tex.Evaluate(ctx) * sc;
}
```

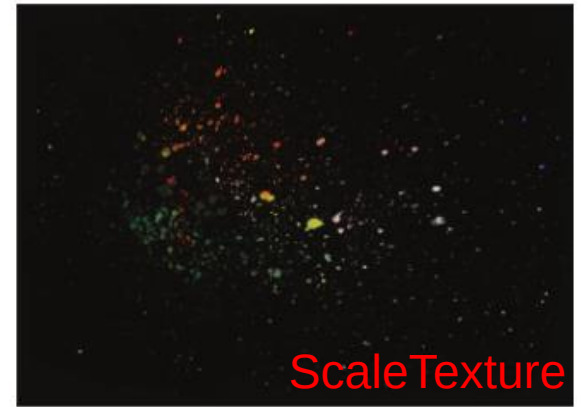
ScaleTexture



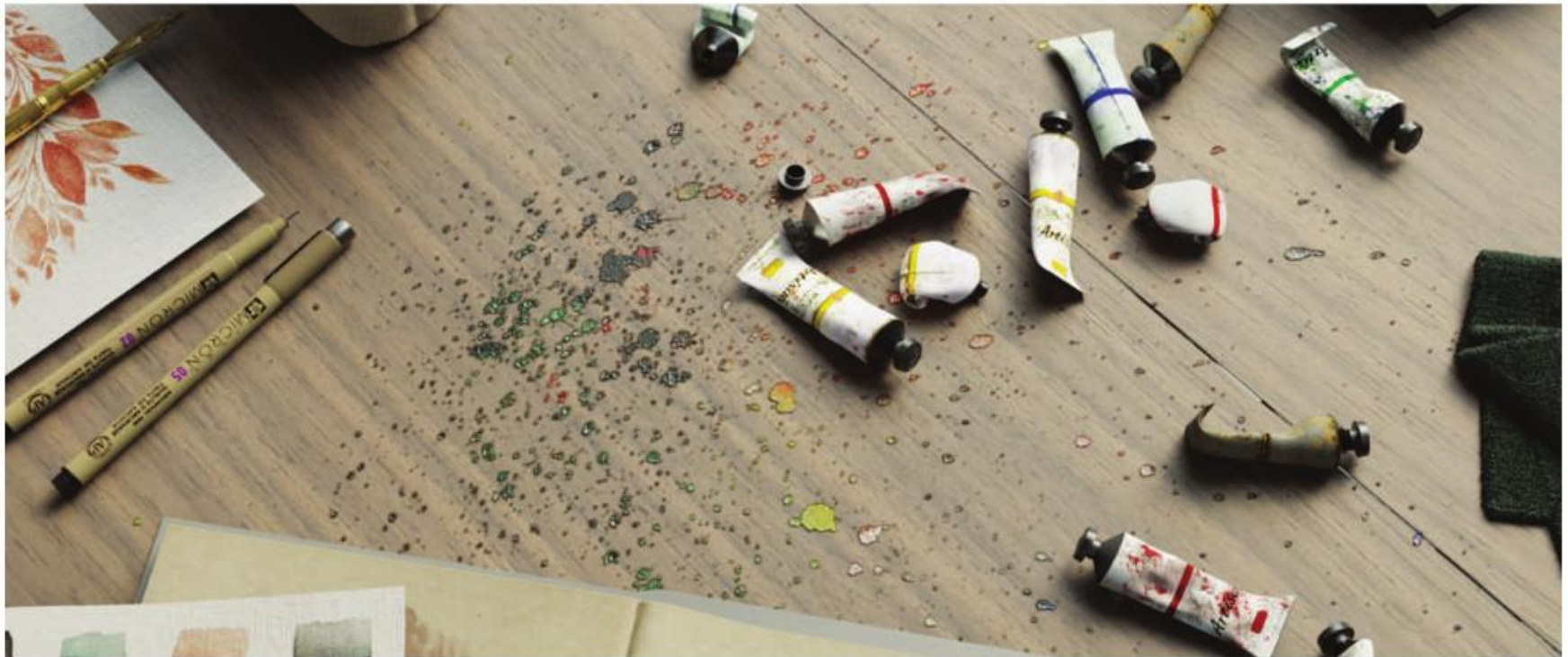
tex



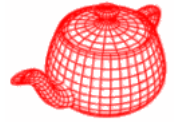
scale



ScaleTexture



MixTexture



Blend two textures of the same type based on a scalar blending factor.

```
class FloatMixTexture {  
    public:  
        <<FloatMixTexture Public Methods>>  
    private:  
        FloatTexture tex1, tex2;  
        FloatTexture amount;  
};
```

Usually assume it is within [0, 1], but not enforced.
Thus, the values could be extrapolated.

```
Float Evaluate(TextureEvalContext ctx) const {  
    Float amt = amount.Evaluate(ctx);  
    Float t1 = 0, t2 = 0;  
    if (amt != 1) t1 = tex1.Evaluate(ctx);  
    if (amt != 0) t2 = tex2.Evaluate(ctx);  
    return (1 - amt) * t1 + amt * t2;  
}
```

again, ignore antialiasing issue here

MixTexture

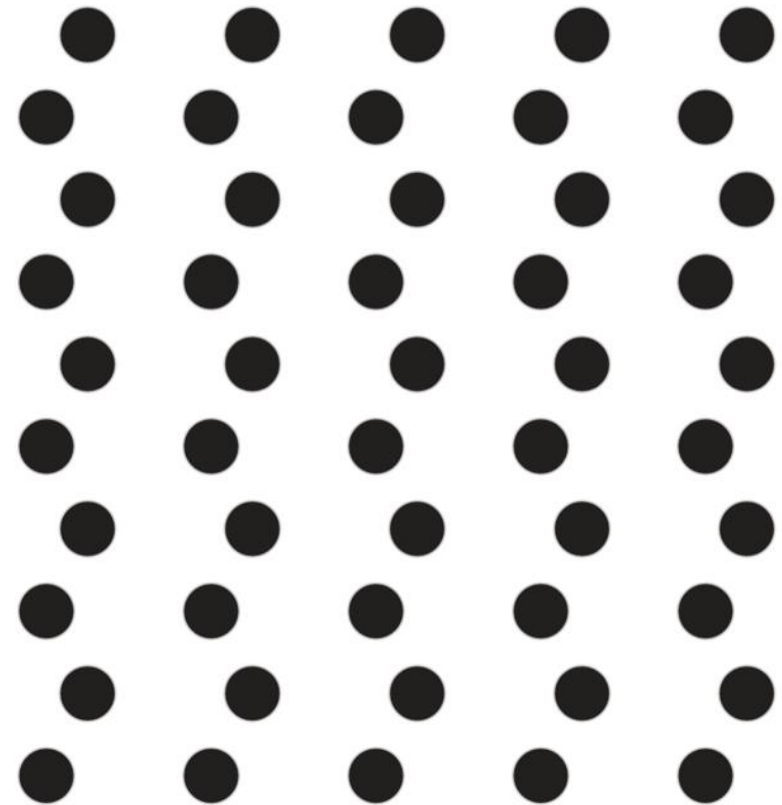
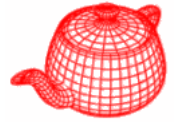
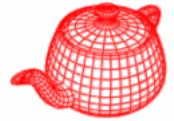


image texture

The two cups use a MixTexture formed by modulating two fixed RGB colors using the image texture.

DirectionMixTexture



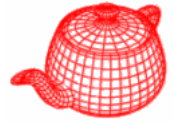
Control the amount by the inner product of the normal with a pre-defined vector.

```
class SpectrumDirectionMixTexture {
    public:
        <<SpectrumDirectionMixTexture Public Methods>> ⊕
    private:
        <<SpectrumDirectionMixTexture Private Members>> ⊕
        SpectrumTexture tex1, tex2;
        Vector3f dir;

};

SampledSpectrum Evaluate(TextureEvalContext ctx,
                        SampledWavelengths lambda) const {
    Float amt = AbsDot(ctx.n, dir);
    SampledSpectrum t1, t2;
    if (amt != 0) t1 = tex1.Evaluate(ctx, lambda);
    if (amt != 1) t2 = tex2.Evaluate(ctx, lambda);
    return amt * t1 + (1 - amt) * t2;
}
```

DirectionMixTexture



Tex1 for the book cover and tex2 for the book spine.

Image texture

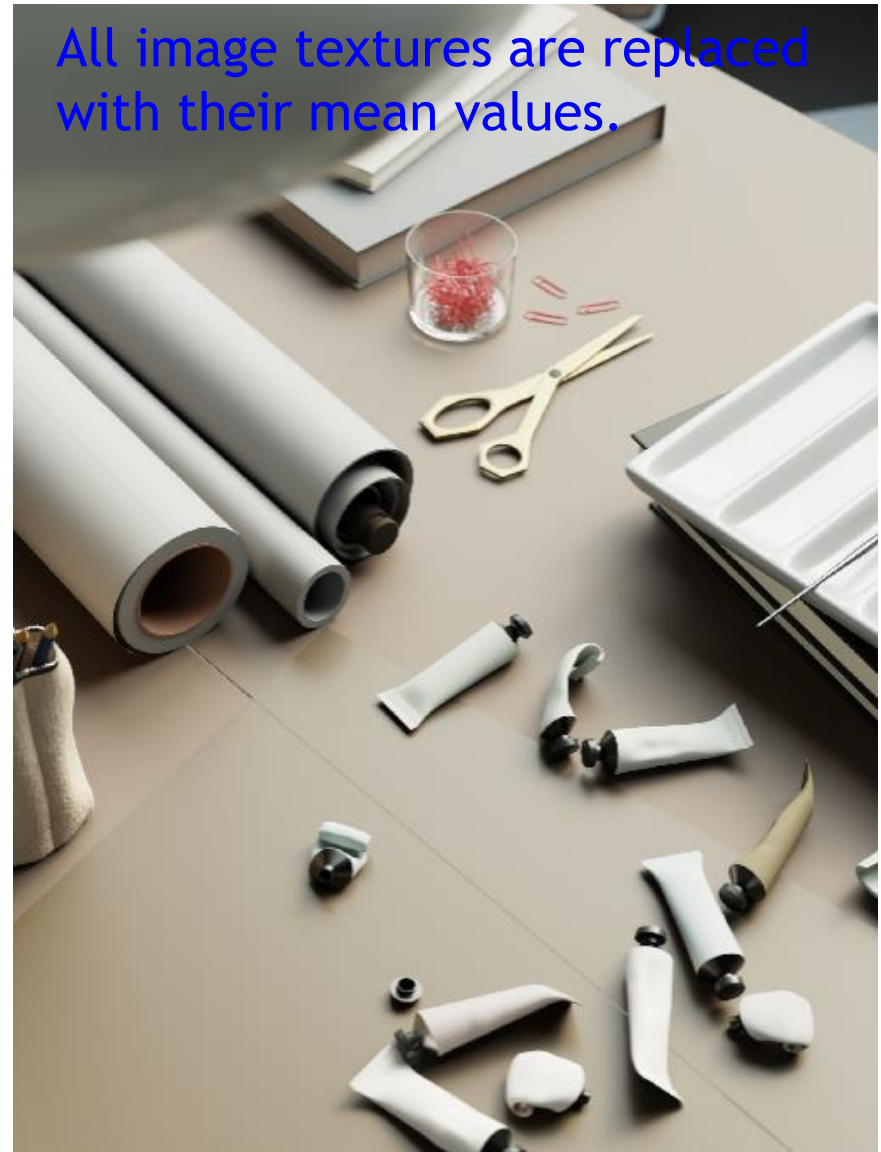
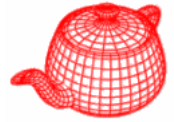


Image texture

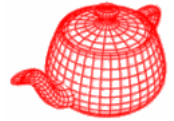


Image textures store 2D arrays of point-sampled values (texels) of a texture function.

Pbrt provides both Float and spectral variants, both inherited from ImageTextureBase.

The most widely used texture maps in graphics.

```
class ImageTextureBase
```

```
class SpectrumImageTexture : public ImageTextureBase {
```

```
    public:
```

```
        <<SpectrumImageTexture Public Methods>>  ⊞
```

```
    private:
```

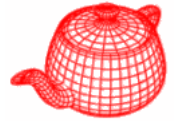
```
        <<SpectrumImageTexture Private Members>>  ⊞
```

```
        SpectrumType spectrumType;
```

```
};    Recall that
```

```
    enum class SpectrumType { Illuminant, Albedo, Unbounded };
```

Image texture



```
SpectrumImageTexture(TextureMapping2D mapping, std::string filename,
    MIPMapFilterOptions filterOptions, WrapMode wrapMode, Float scale,
    bool invert, ColorEncoding encoding, SpectrumType spectrumType,
    Allocator alloc)
: ImageTextureBase(mapping, filename, filterOptions, wrapMode,
    scale, invert, encoding, alloc),
    spectrumType(spectrumType) {}
```

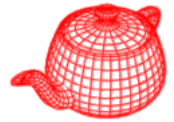
```
ImageTextureBase(TextureMapping2D mapping, std::string filename,
    MIPMapFilterOptions filterOptions, WrapMode wrapMode, Float scale,
    bool invert, ColorEncoding encoding, Allocator alloc)
: mapping(mapping), filename(filename), scale(scale), invert(invert) {
    <<Get MIPMap from texture cache if present>> ☒
    <<Create MIPMap for filename and add to texture cache>> ☒
}
```

<<ImageTextureBase Protected Members>>=

```
TextureMapping2D mapping;
std::string filename;
Float scale;
bool invert;
MIPMap *mipmap;
```

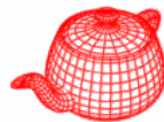
An on-disk image could be used multiple times. It is only necessary to store one copy in memory.

Image texture



```
SampledSpectrum SpectrumImageTexture::Evaluate(  
    TextureEvalContext ctx, SampledWavelengths lambda) const {  
    <<Apply texture mapping and flip  $t$  coordinate for image texture lookup>> ⊕  
    <<Lookup filtered RGB value in MIPMap>> ⊕  
    <<Return SampledSpectrum for RGB image texture value>> ⊕  
}
```

Image texture



<<Apply texture mapping and flip t coordinate for image texture lookup>>=

```
TexCoord2D c = mapping.Map(ctx);  
c.st[1] = 1 - c.st[1];
```

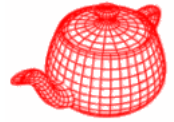
<<Lookup filtered RGB value in MIPMap>>=

```
RGB rgb = scale * mipmap->Filter<RGB>(c.st, {c.dsdx, c.dtdx},  
                                         {c.dsdy, c.dtdy});  
rgb = ClampZero(invert ? (RGB(1, 1, 1) - rgb) : rgb);
```

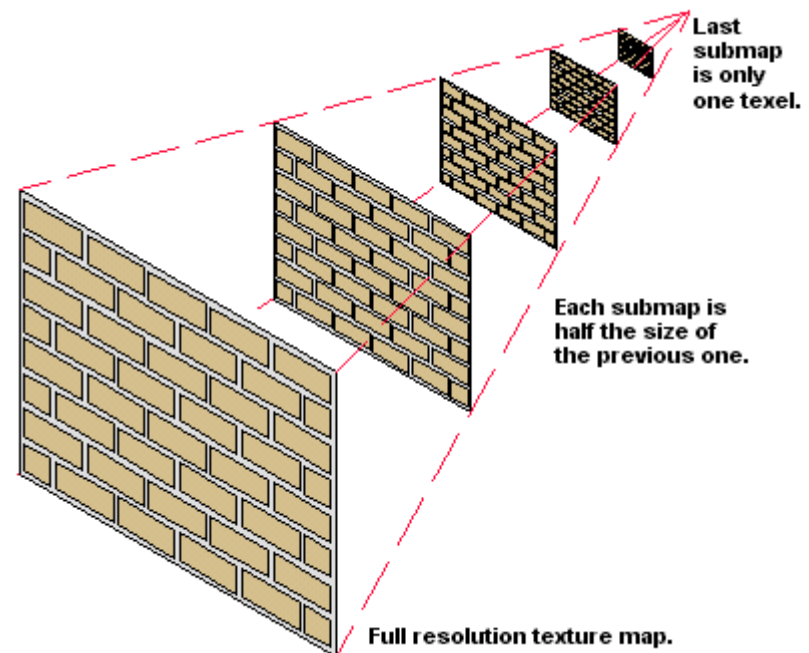
<<Return SampledSpectrum for RGB image texture value>>=

```
if (const RGBColorSpace *cs = mipmap->GetRGBColorSpace(); cs) {  
    if (spectrumType == SpectrumType::Unbounded)  
        return RGBUnboundedSpectrum(*cs, rgb).Sample(lambda);  
    else if (spectrumType == SpectrumType::Albedo)  
        return RGBAlbedoSpectrum(*cs, Clamp(rgb, 0, 1)).Sample(lambda);  
    else  
        return RGBIlluminantSpectrum(*cs, rgb).Sample(lambda);  
}  
DCHECK(rgb[0] == rgb[1] && rgb[1] == rgb[2]);  
return SampledSpectrum(rgb[0]);
```

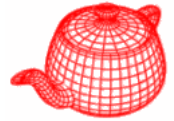
Mip maps



- It is more efficient to do antialiasing for textures because
 1. Less expensive to get samples
 2. 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



Mip Maps



```
MIPMap::MIPMap(
    Image image, const RGBColorSpace *colorSpace, WrapMode wrapMode,
    Allocator alloc, const MIPMapFilterOptions &options)
    : colorSpace(colorSpace), wrapMode(wrapMode), options(options) {
    pyramid = Image::GeneratePyramid(std::move(image), wrapMode, alloc);
}
```

<<MIPMap Private Members>>=

```
pstd::vector<Image> pyramid;
```

```
const RGBColorSpace *colorSpace;
```

```
WrapMode wrapMode;
```

```
MIPMapFilterOptions options;
```

```
enum class WrapMode { Black, Clamp, Repeat, OctahedralSphere };
```

```
struct MIPMapFilterOptions {
```

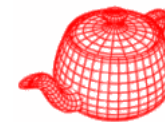
```
    FilterFunction filter = FilterFunction::EWA;
```

```
    Float maxAnisotropy = 8.f;
```

```
};
```

```
enum class FilterFunction { Point, Bilinear, Trilinear, EWA };
```

Mip Maps



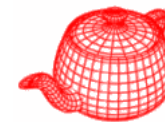
<<MIPMap Public Methods>>=

```
Point2i LevelResolution(int level) const {  
    return pyramid[level].Resolution();  
}  
int Levels() const { return int(pyramid.size()); }  
const RGBColorSpace *GetRGBColorSpace() const { return colorSpace; }  
const Image &GetLevel(int level) const { return pyramid[level]; }
```

<<MIPMap Method Definitions>>+= ▲ ▼

```
template <>  
RGB MIPMap::Texel(int level, Point2i st) const {  
    if (int nc = pyramid[level].NChannels(); nc == 3 || nc == 4)  
        return RGB(pyramid[level].GetChannel(st, 0, wrapMode),  
                    pyramid[level].GetChannel(st, 1, wrapMode),  
                    pyramid[level].GetChannel(st, 2, wrapMode));  
    else {  
        Float v = pyramid[level].GetChannel(st, 0, wrapMode);  
        return RGB(v, v, v);  
    }  
}
```

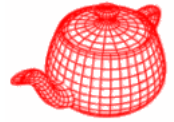
Image Map Filter



- Return the texel value for a given (s, t)
- Two categories:
 - Isotropic
 - Anisotropic (EWA filter)

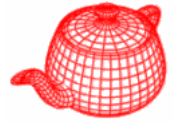
```
template <typename T>
T MIPMap::Filter(Point2f st, Vector2f dst0, Vector2f dst1) const {
    if (options.filter != FilterFunction::EWA) {
        <<Handle non-EWA MIP Map filter>> ⊕
    }
    <<Compute EWA ellipse axes>> ⊕
    <<Clamp ellipse vector ratio if too large>> ⊕
    <<Choose level of detail for EWA lookup and perform EWA filtering>> ⊕
}
```

Mipmap lookup (isotropic)



```
Float width = 2 * std::max({std::abs(dst0[0]), std::abs(dst0[1]),  
                             std::abs(dst1[0]), std::abs(dst1[1])});  
<<Compute MIP Map level for width and handle very wide filter>> ⊕  
if (options.filter == FilterFunction::Point) {  
    <<Return point-sampled value at selected MIP level>> ⊕  
} else if (options.filter == FilterFunction::Bilinear) {  
    <<Return bilinear-filtered value at selected MIP level>> ⊕  
} else {  
    <<Return trilinear-filtered value at selected MIP level>> ⊕  
}
```

Mipmap lookup (isotropic)



- Trilinear filtering:

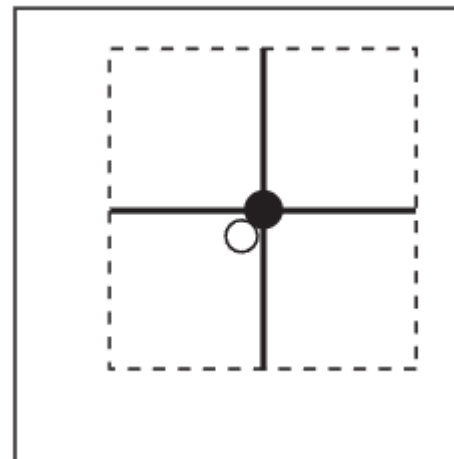
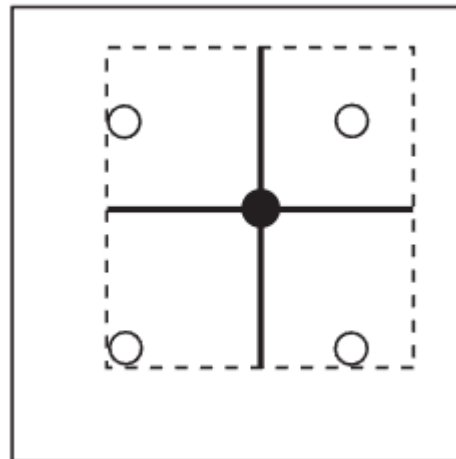
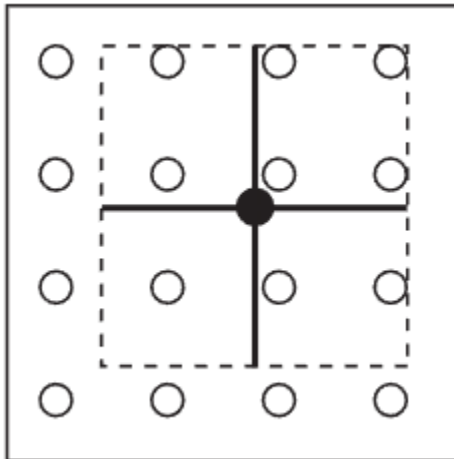
`Lookup(float s, float t, float width)`

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

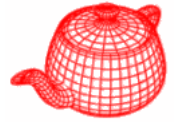
$$l = n - 1 + \log w$$

choose the level to
cover four texels

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



Mipmap lookup (point)



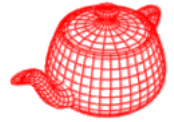
<<Compute MIP Map level for width and handle very wide filter>>=

```
int nLevels = Levels();  
Float level = nLevels - 1 + Log2(std::max<Float>(width, 1e-8));  
if (level >= Levels() - 1)  
    return Texel<T>(Levels() - 1, Point2i(0, 0));  
int iLevel = std::max(0, int(pstd::floor(level)));
```

<<Return point-sampled value at selected MIP level>>=

```
Point2i resolution = LevelResolution(iLevel);  
Point2i sti(pstd::round(st[0] * resolution[0] - 0.5f),  
            pstd::round(st[1] * resolution[1] - 0.5f));  
return Texel<T>(iLevel, sti);
```

Mipmap lookup (bilinear)



<<Return bilinear-filtered value at selected MIP level>>=

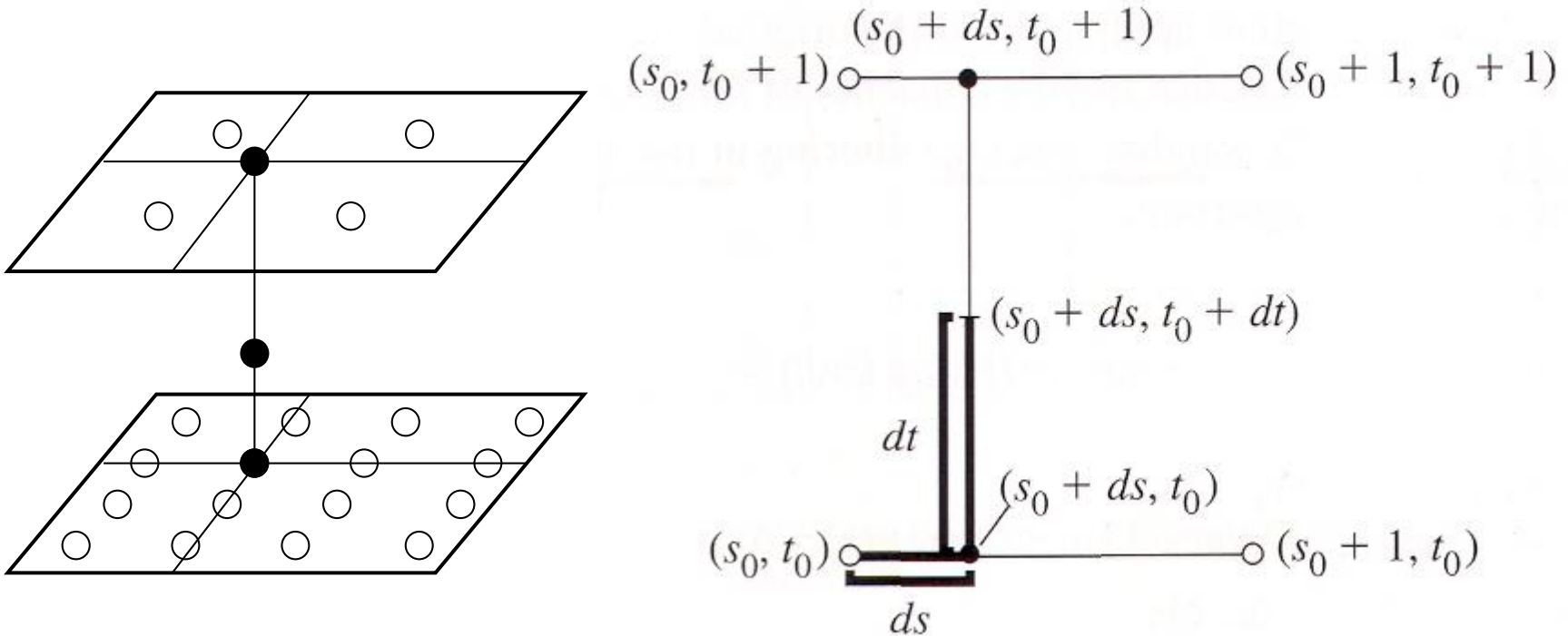
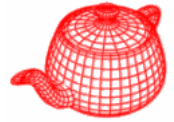
```
return Bilerp<T>(iLevel, st);
```

<<MIPMap Method Definitions>>+= ▲ ▼

```
template <>
```

```
RGB MIPMap::Bilerp(int level, Point2f st) const {  
    if (int nc = pyramid[level].NChannels(); nc == 3 || nc == 4)  
        return RGB(pyramid[level].BilerpChannel(st, 0, wrapMode),  
                    pyramid[level].BilerpChannel(st, 1, wrapMode),  
                    pyramid[level].BilerpChannel(st, 2, wrapMode));  
    else {  
        Float v = pyramid[level].BilerpChannel(st, 0, wrapMode);  
        return RGB(v, v, v);  
    }  
}
```

Mipmap lookup (trilinear)



<<Return trilinear-filtered value at selected MIP level>>=

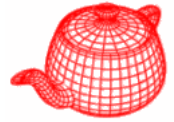
```
if (iLevel == 0)
```

```
    return Bilerp<T>(0, st);
```

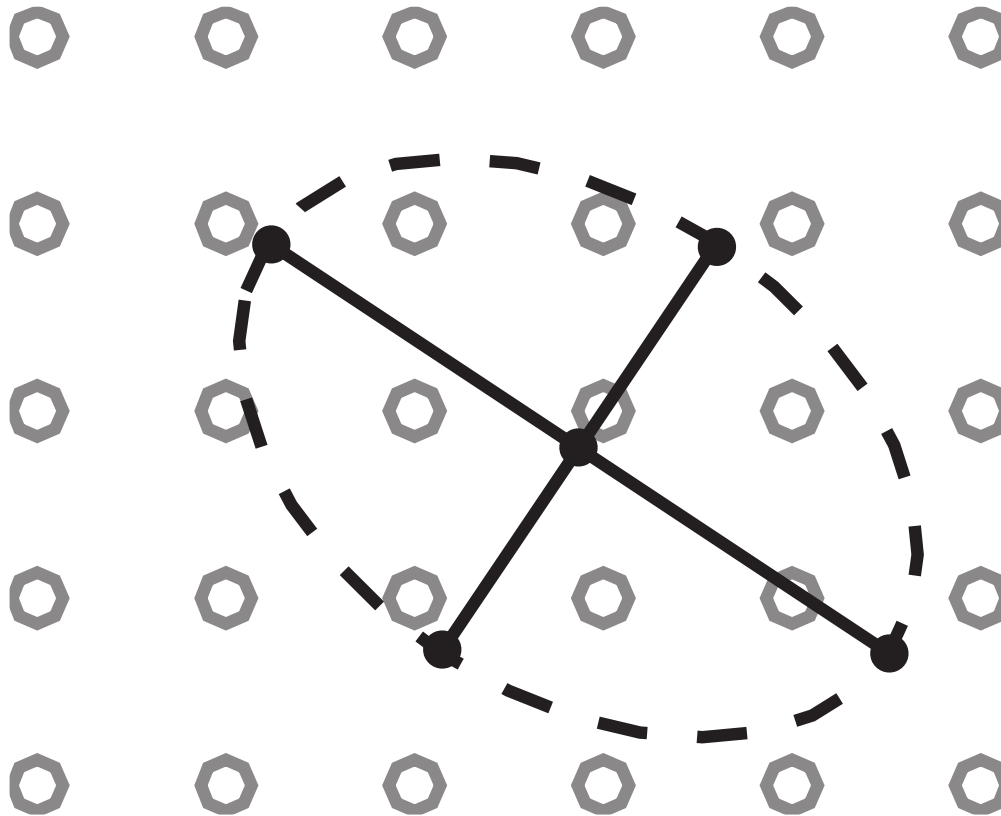
```
else
```

```
    return Lerp(level - iLevel, Bilerp<T>(iLevel, st),  
                Bilerp<T>(iLevel + 1, st));
```

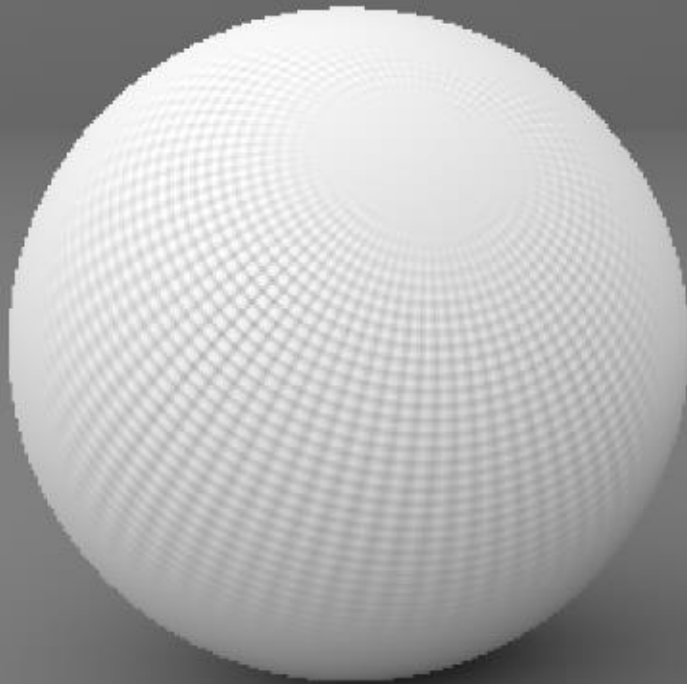
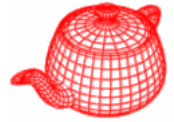
Mipmap lookup (EWA)



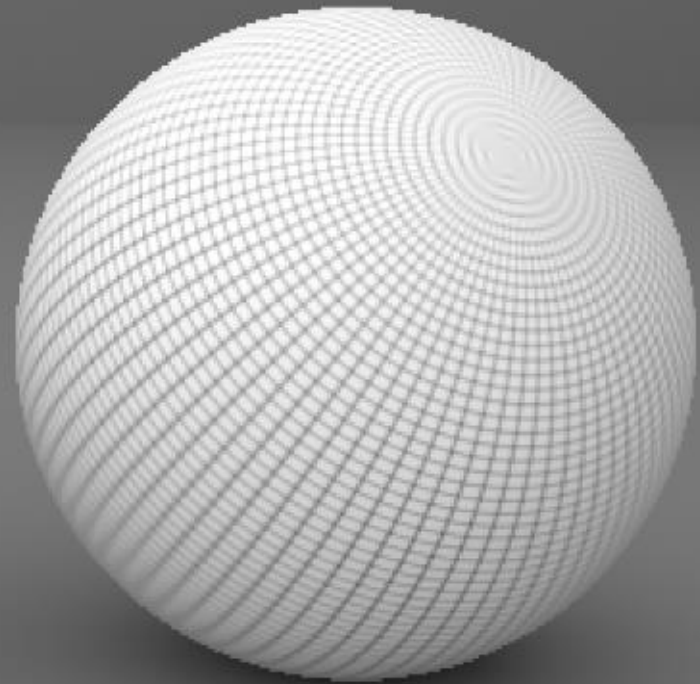
- Elliptically weighted average filtering



Mip Maps

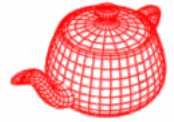


trilinear



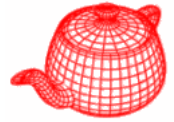
EWA

Solid and procedural texturing

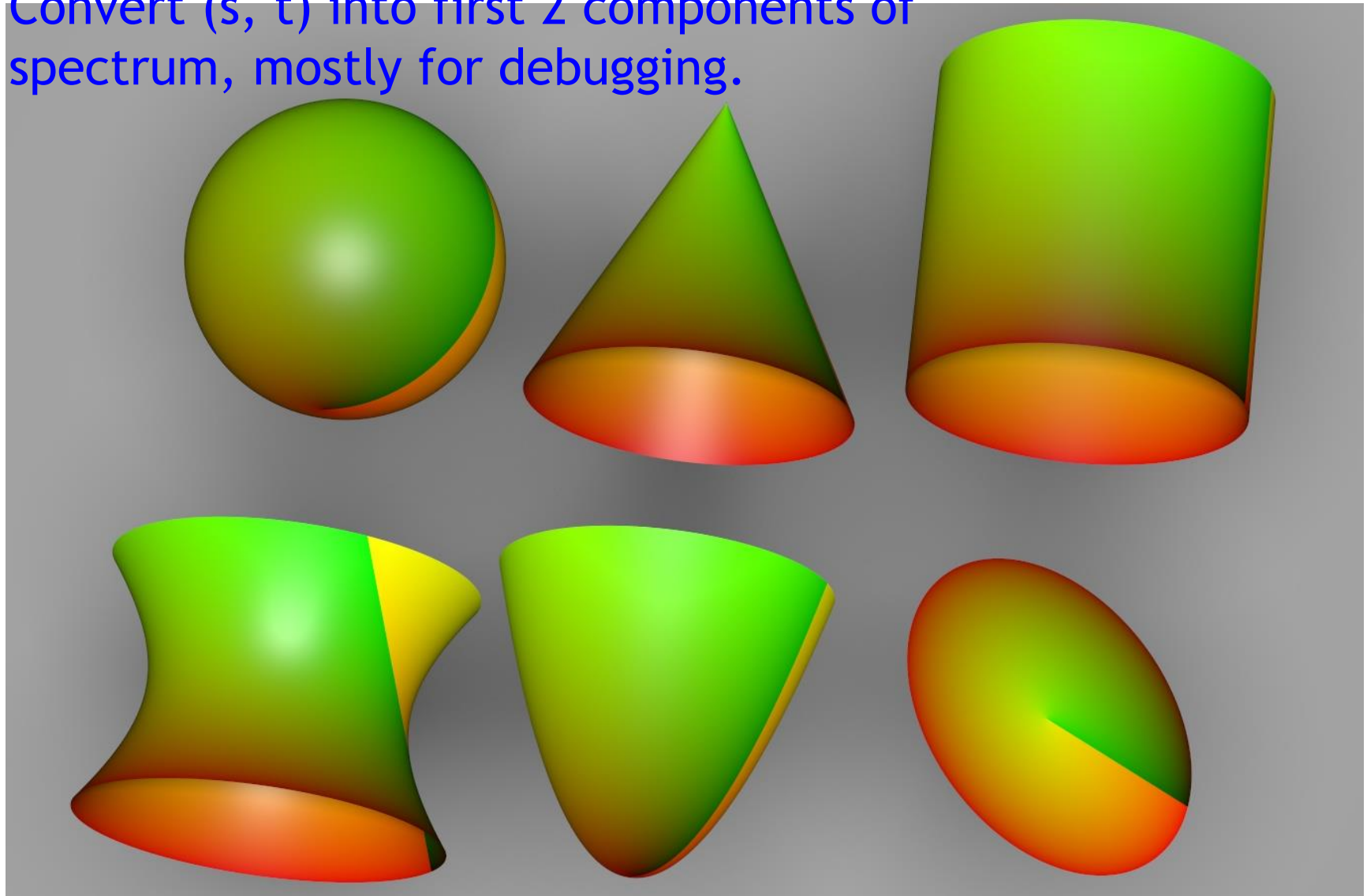


- Solid texture: textures defined over 3D domains; often requiring a large amount of storage
- Procedural texture: short programs are used to generate texture values (e.g. sine waves)
 - Less storage
 - Better details (evaluate accurate values when needed; resolution independent)
 - Harder to control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

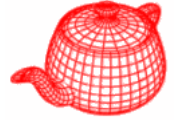
UV texture (v2)



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

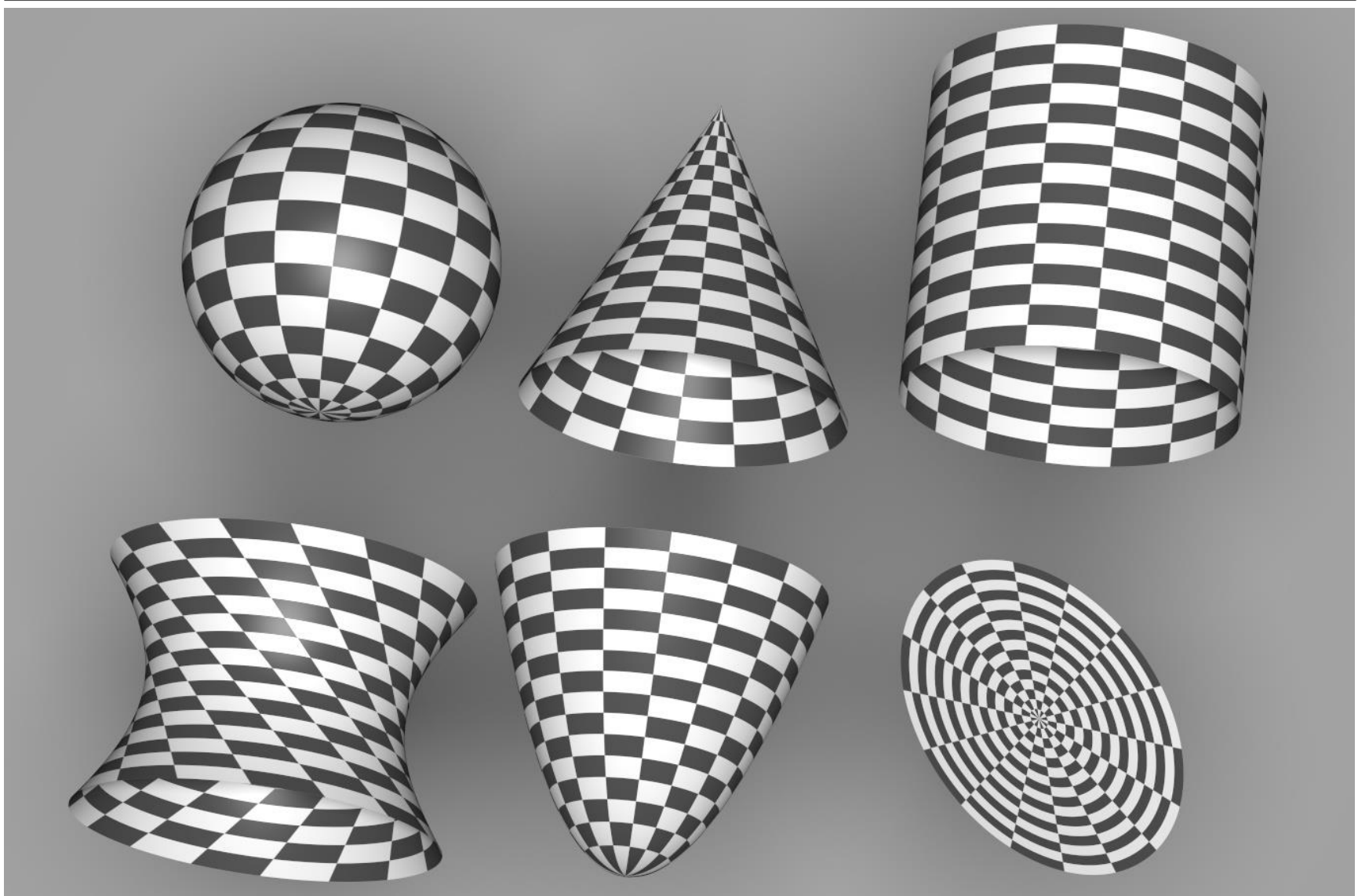
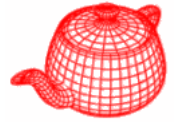


UV texture

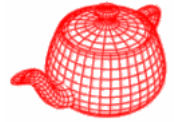


```
Spectrum Evaluate(const DifferentialGeometry &dg) {  
    float s, t, dsdx, dtdx, dsdy, dtdy;  
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);  
    float rgb[3] = {s-Floor2Int(s), t-Floor2Int(t), 0};  
    return Spectrum::FromRGB(rgb);  
}
```

Checkboard (v2)

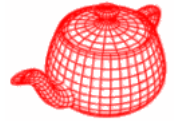


Checkerboard2D



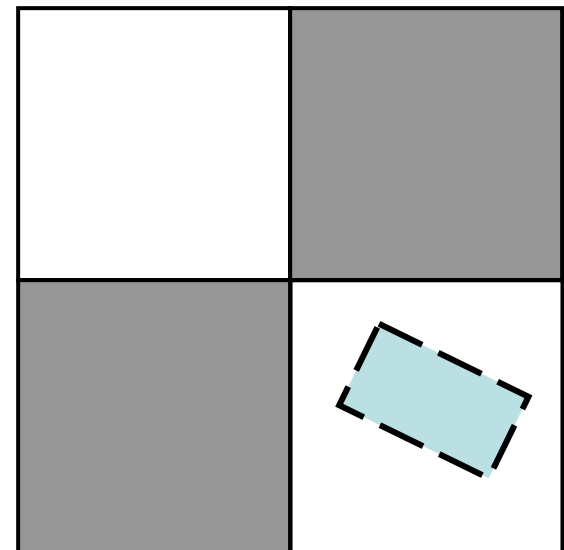
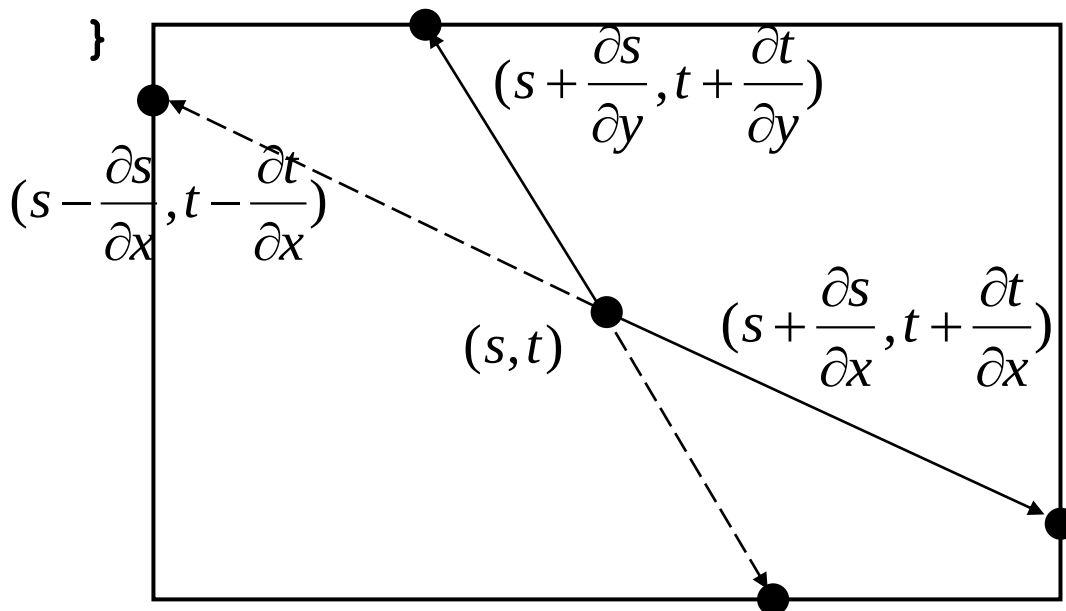
```
class Checkerboard2D : public Texture<T> {
    Checkerboard2D(TextureMapping2D *m,
        Reference<Texture<T>> c1, Reference<Texture<T>> c2,
        const string &aa);
    ...
    T Evaluate(const DifferentialGeometry &dg) const {
        float s, t, dsdx, dtdx, dsdy, dtdy;
        mapping->Map(dg, &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

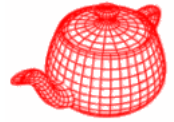


```
float ds = max(fabsf(dsdx), fabsf(dsdy));  
float dt = max(fabsf(dtdx), fabsf(dtdy));  
float s0 = s - ds, s1 = s + ds;  
float t0 = t - dt, t1 = t + dt;  
if (Floor2Int(s0) == Floor2Int(s1) &&  
    Floor2Int(t0) == Floor2Int(t1)) {  
    if ((Floor2Int(s) + Floor2Int(t)) % 2 == 0)  
        return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

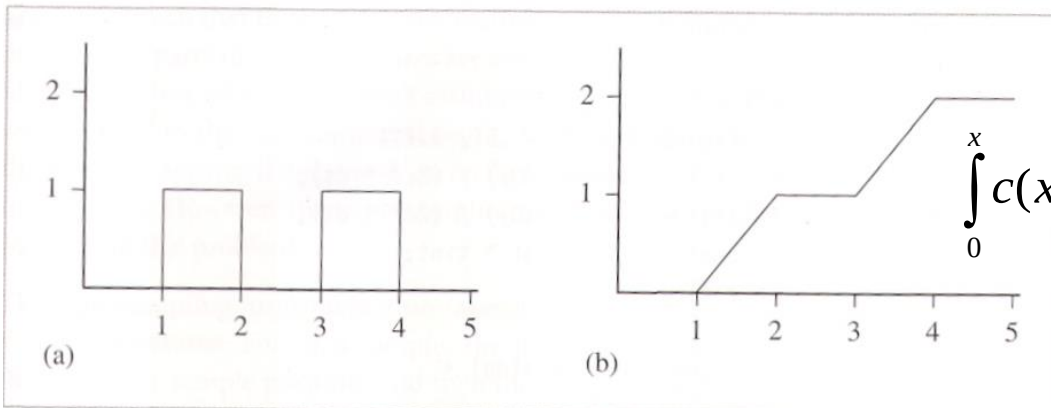
point sampling



Close-form

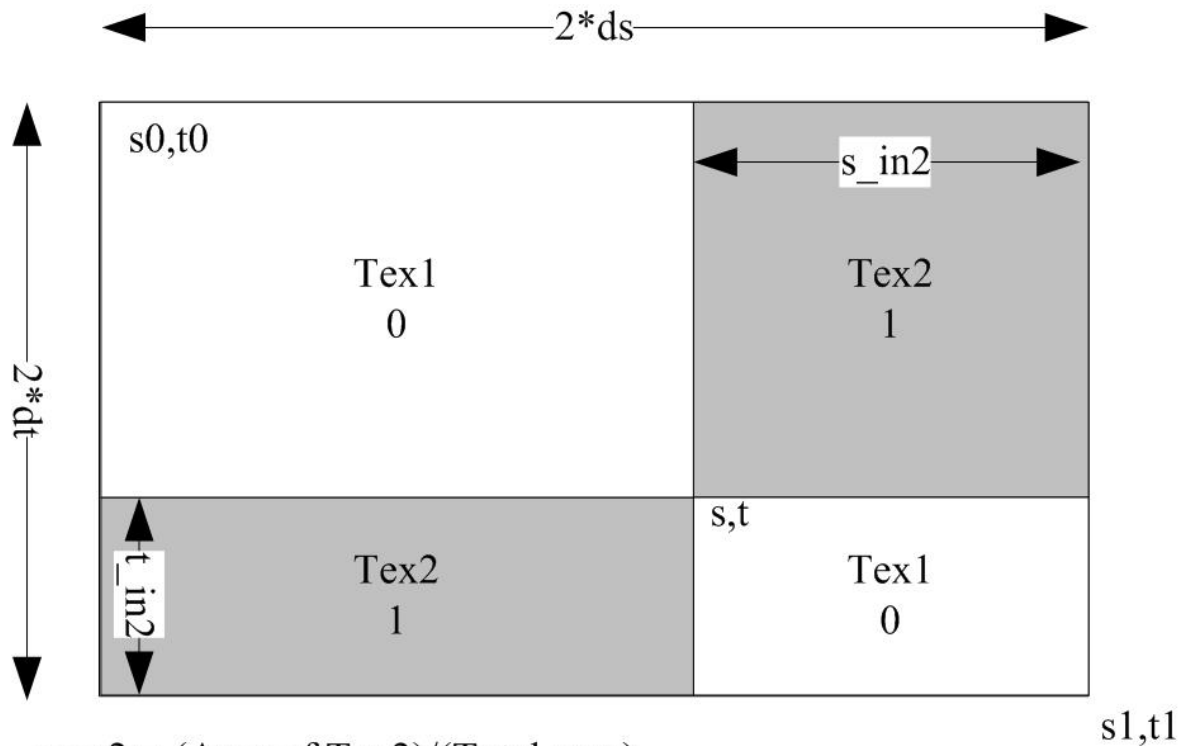
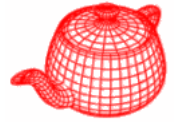


```
#define BUMPINT(x) \  
    (Floor2Int((x)/2) + \  
     2.f * max((x/2)-Floor2Int(x/2) - .5f, 0.f))  
float sint = (BUMPINT(s1) - BUMPINT(s0)) / (2.f*ds);  
float tint = (BUMPINT(t1) - BUMPINT(t0)) / (2.f*dt);  
float area2 = sint + tint - 2.f * sint * tint;  
if (ds > 1.f || dt > 1.f) area2 = .5f;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



$$\int_0^x c(x) dx = \left\lfloor \frac{x}{2} \right\rfloor + 2 \max\left(\frac{x}{2} - \left\lfloor \frac{x}{2} \right\rfloor - \frac{1}{2}, 0\right)$$

Close-form



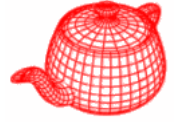
$$\begin{aligned}
 \text{area2} &= (\text{Area of Tex2}) / (\text{Total area}) \\
 &= (s_in2 * (2*dt - t_in2) + (2*ds - s_in2) * t_in2) / (2*ds * 2*dt) \\
 &= \underline{(s_in2 / (2*ds)) * (1 - t_in2 / (2*dt))} + \underline{(1 - s_in2 / (2*ds)) * (t_in2 / (2*dt))}
 \end{aligned}$$

$$s_in2 / (2*ds) = (\text{BUMPINT}(s_1) - \text{BUMPINT}(s_0)) / (2*ds) = \text{sint}$$

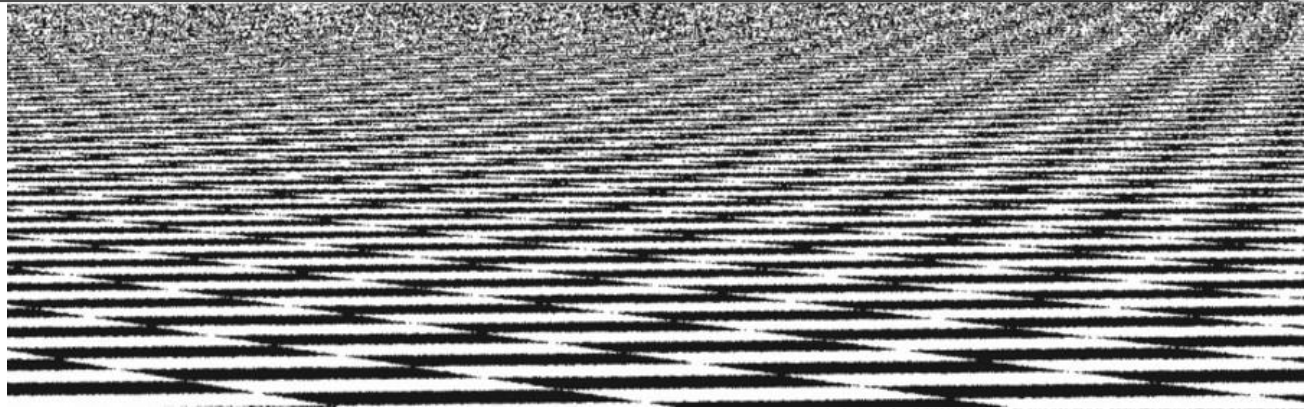
$$t_in2 / (2*dt) = (\text{BUMPINT}(t_1) - \text{BUMPINT}(t_0)) / (2*dt) = \text{tint}$$

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

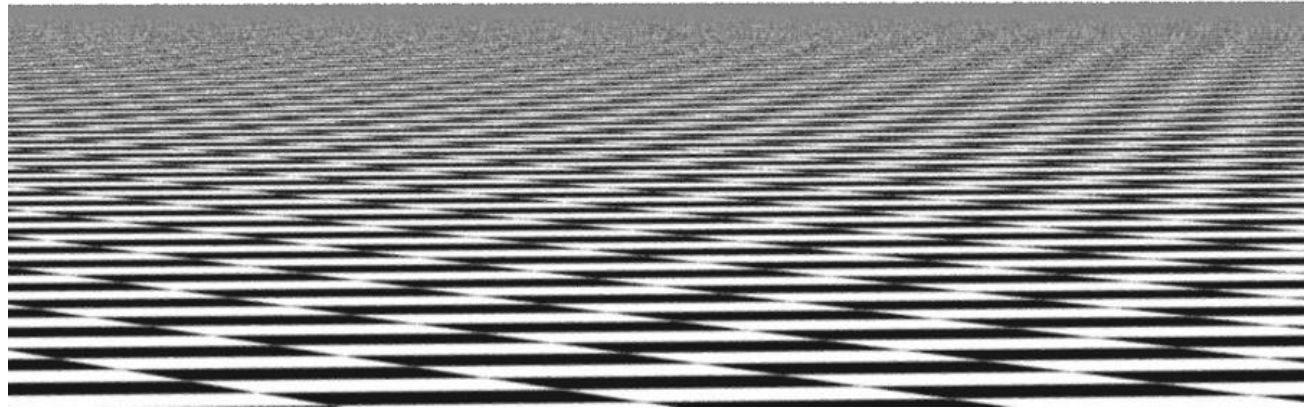
Comparisons



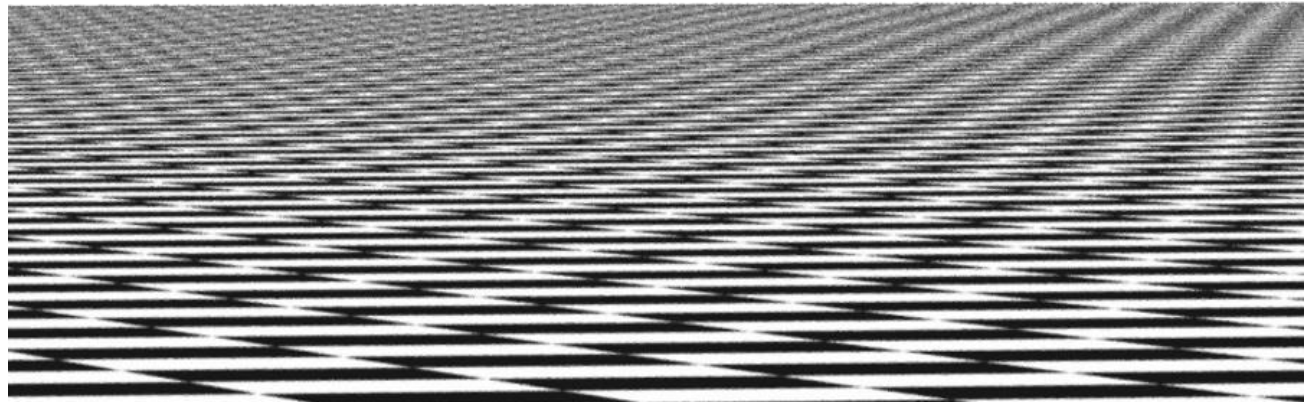
no antialiasing



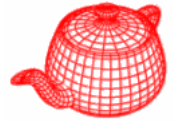
closed-form
less aliasing
but blurring



supersampling
less blurring
susceptible to
aliasing



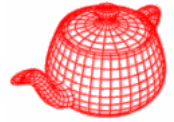
Solid checkboard



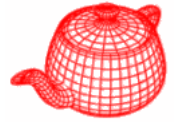
```
Checkerboard3D(TextureMapping3D *m,  
    Reference<Texture<T>> c1, Reference<Texture<T>> c2)  
{  
    mapping = m;  
    tex1 = c1;  
    tex2 = c2;  
}
```

```
Point p = mapping->Map(dg, &dpdx, &dpdy);  
if ((Floor2Int(p.x)+Floor2Int(p.y)+ Floor2Int(p.z))  
    % 2 == 0)  
    return tex1->Evaluate(dgs);  
else  
    return tex2->Evaluate(dgs);
```

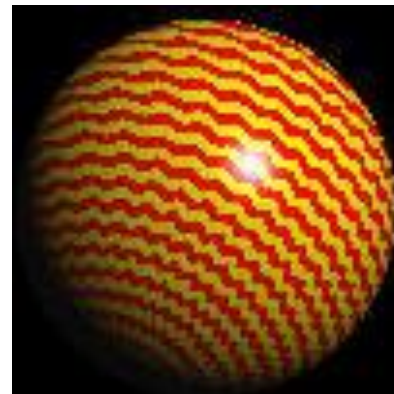
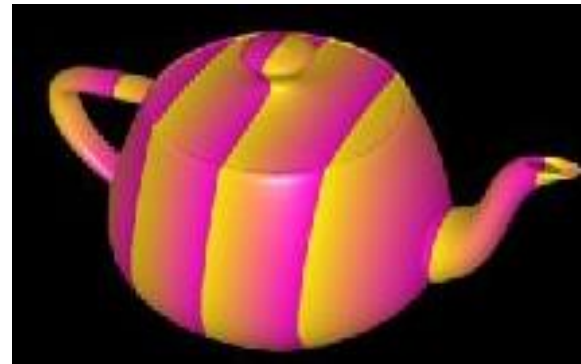
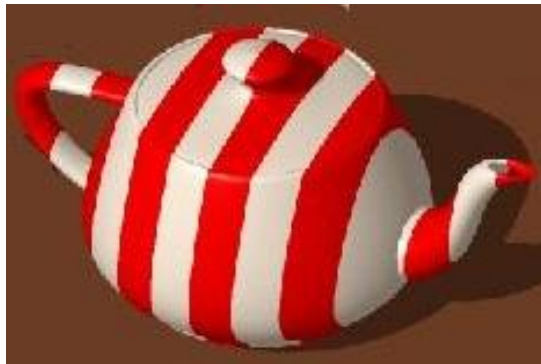
Solid checkboard



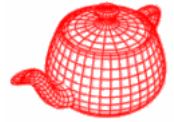
Some other procedure textures



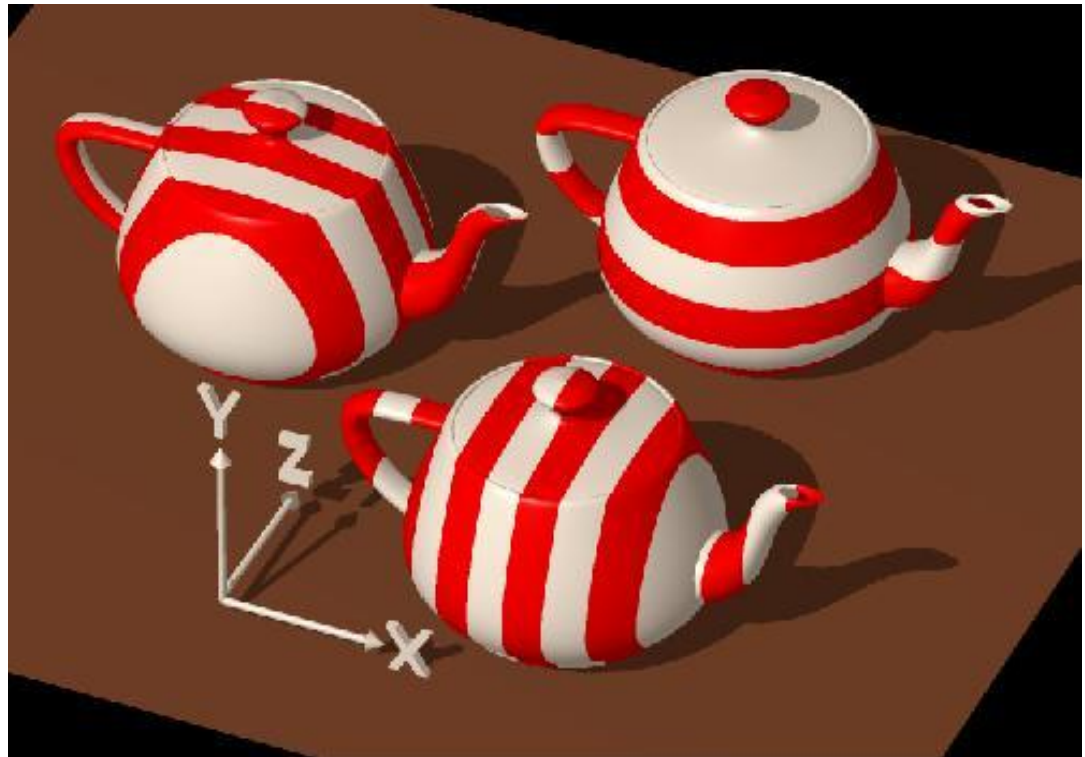
- Stripe
- Ramp and sine
- Rings
- Wood



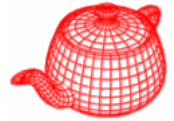
Stripes on teapot



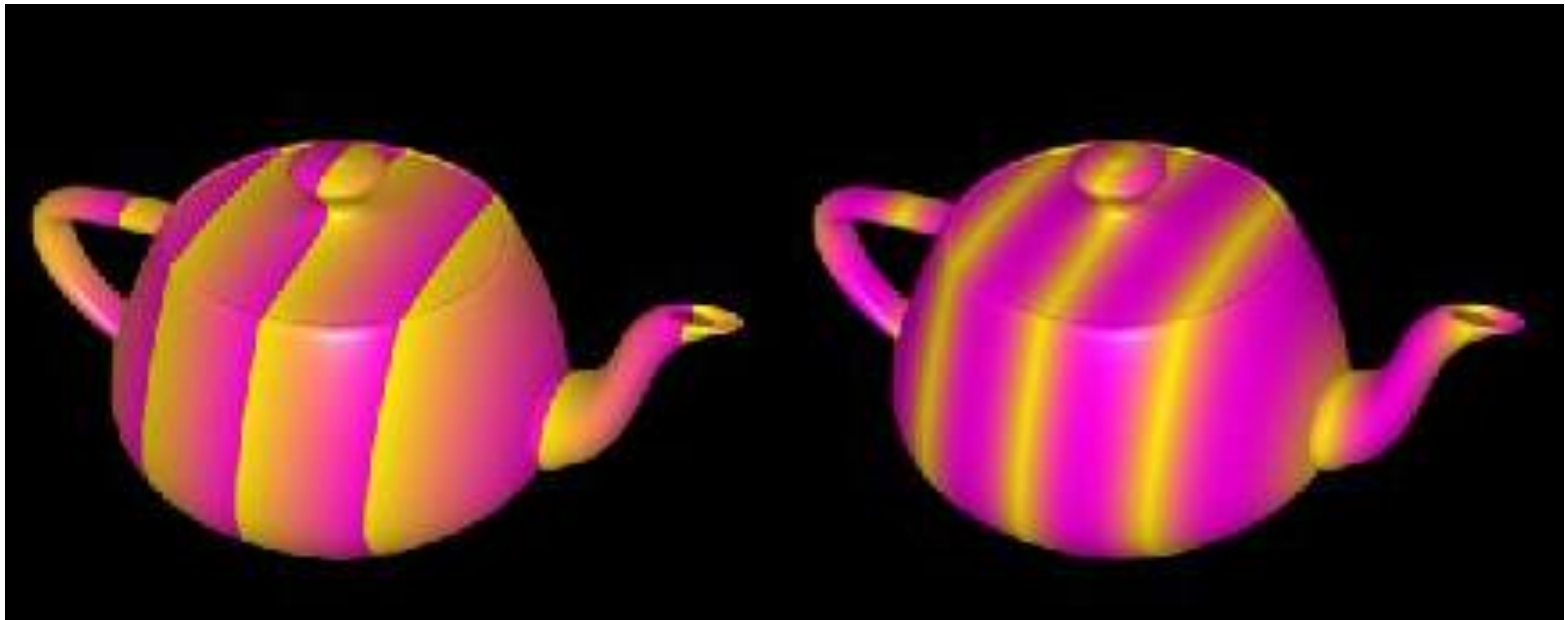
- Find the integer part of the x-, y-, or z-value of each point of the object.
- If resulting value is even,
 - then choose red;
 - else choose white.



Ramp and Sine Functions

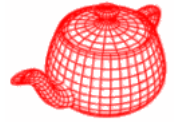


- A nice ramp: $\text{mod}(x,a)/a$
- This ramp function has a range of zero to one, as does $(\sin(x)+1)/2$.

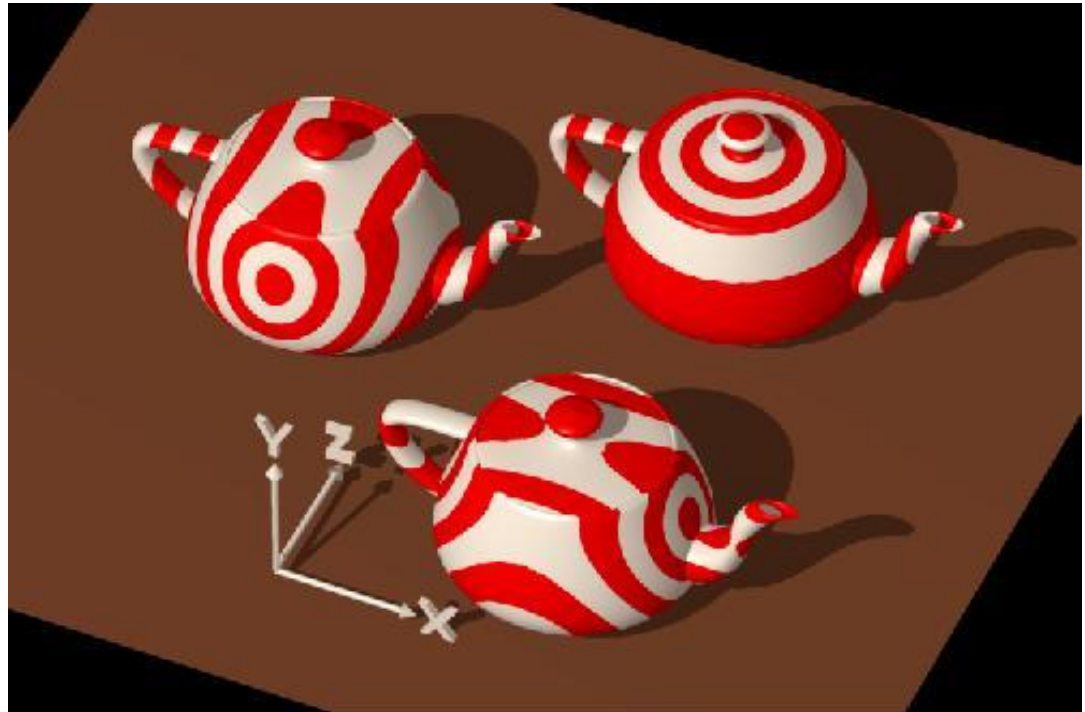


Magenta to value of zero; and yellow to the value one

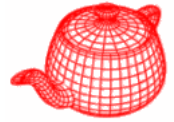
Rings on teapot



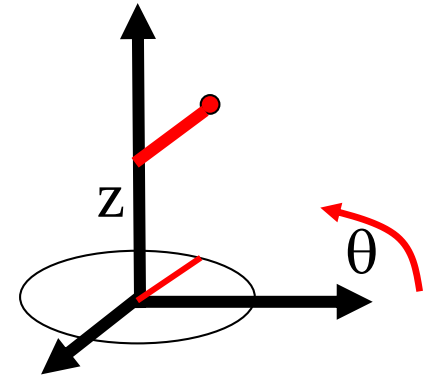
- Use the x - and y -components to compute the distance of a point from the object's center, and truncate the result.
- If the resulting value is even,
 - then we choose red;
 - else we choose white.



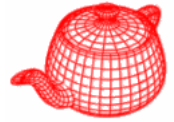
Wood grain



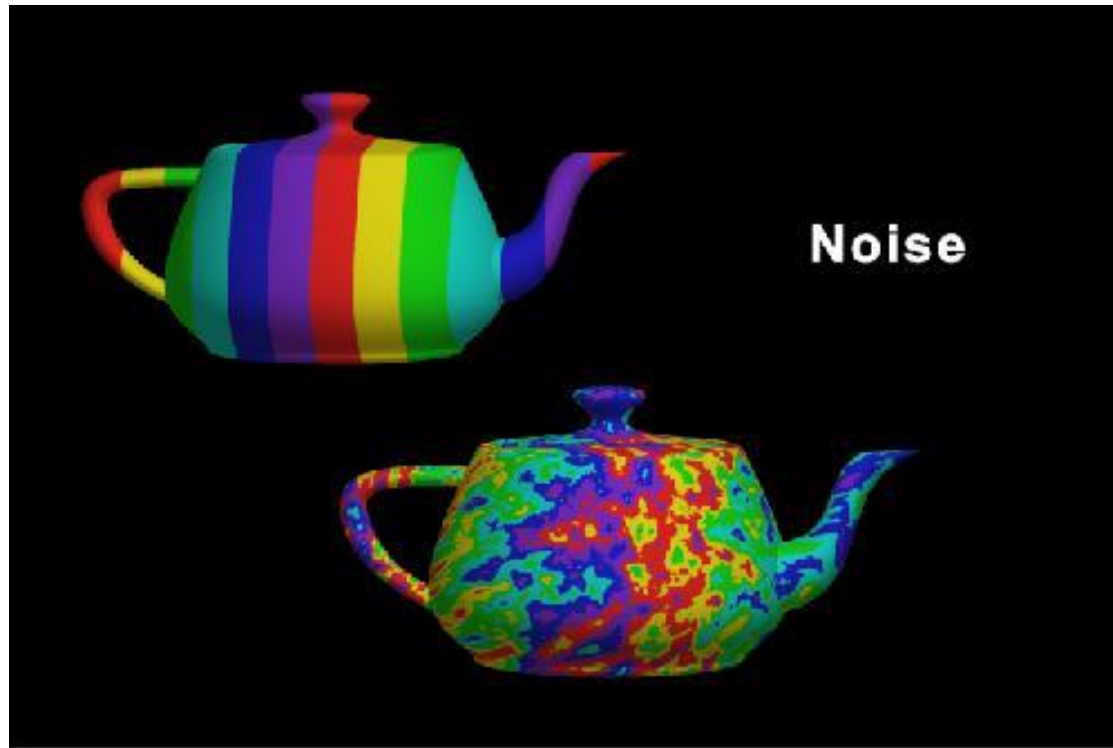
- $r = x^2 + y^2$
- $\text{ring}(r) = (\text{int}(r)) \% 2$
- $\text{Wobble}(r) = \text{ring}(r/M + k \cdot \sin(\theta/N))$



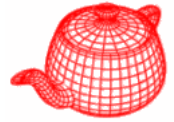
Noise



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

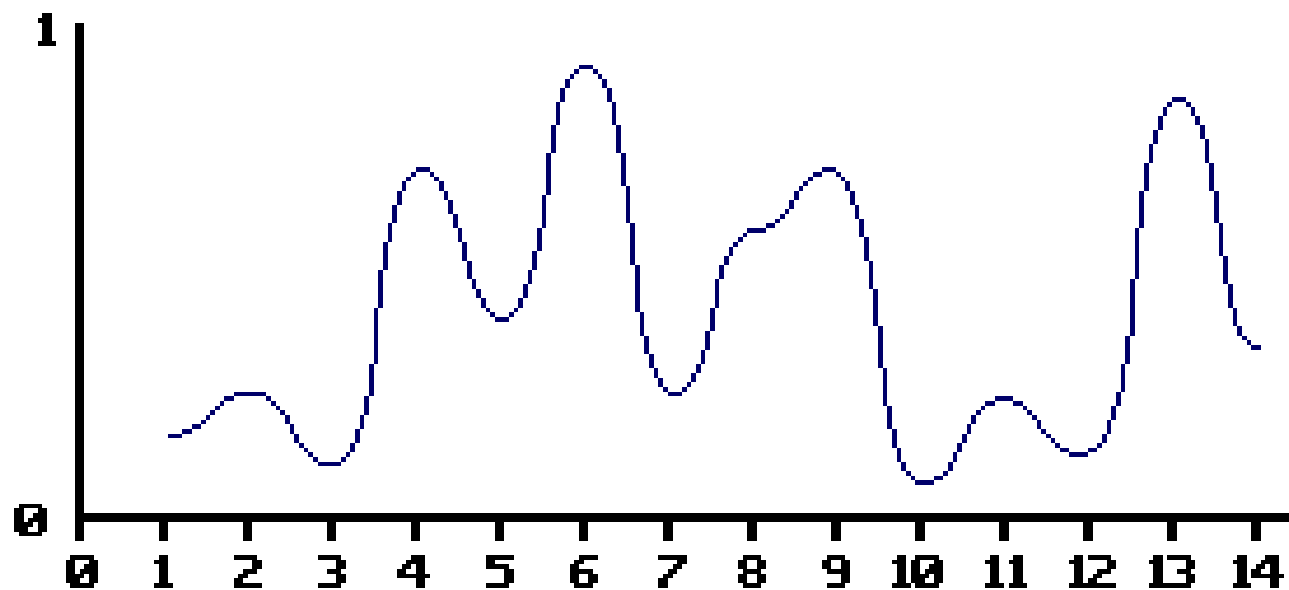
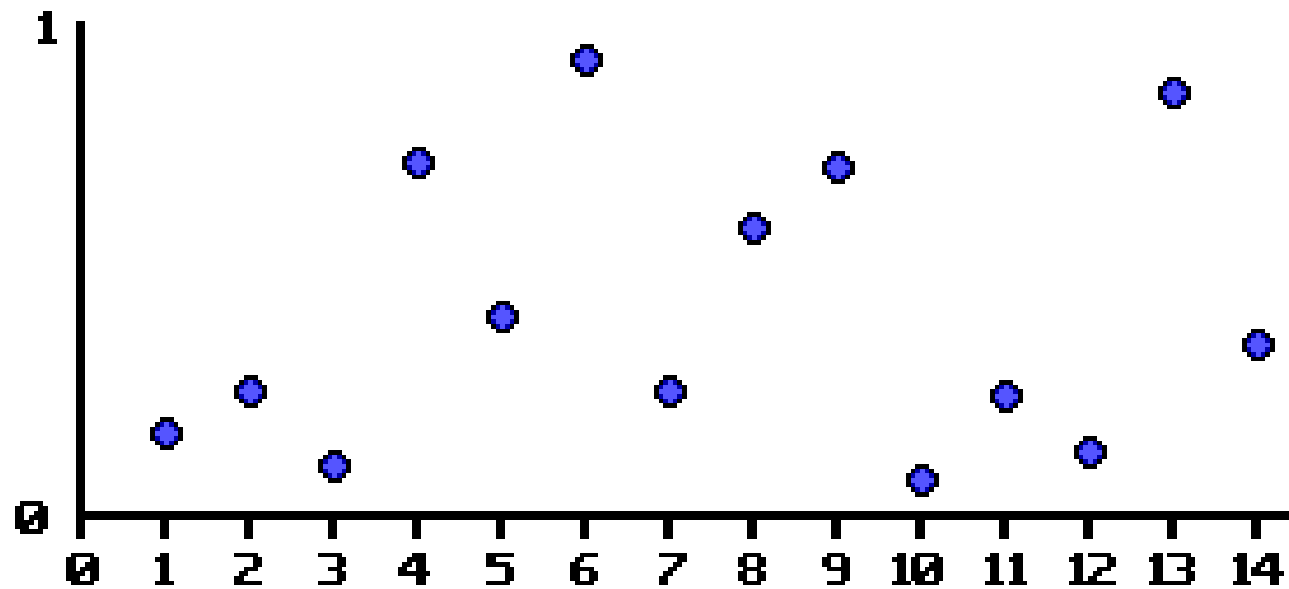
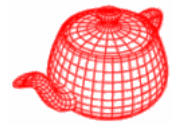


Perlin noise

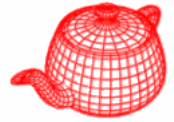


- For example, we might want to add “noise” to the stripe function. One possibility would be to add random numbers to the stripe function. This won’t look good because it looks just like white noise in TV.
- We want to make it smoother without losing the random quality.
- Another option is to take a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

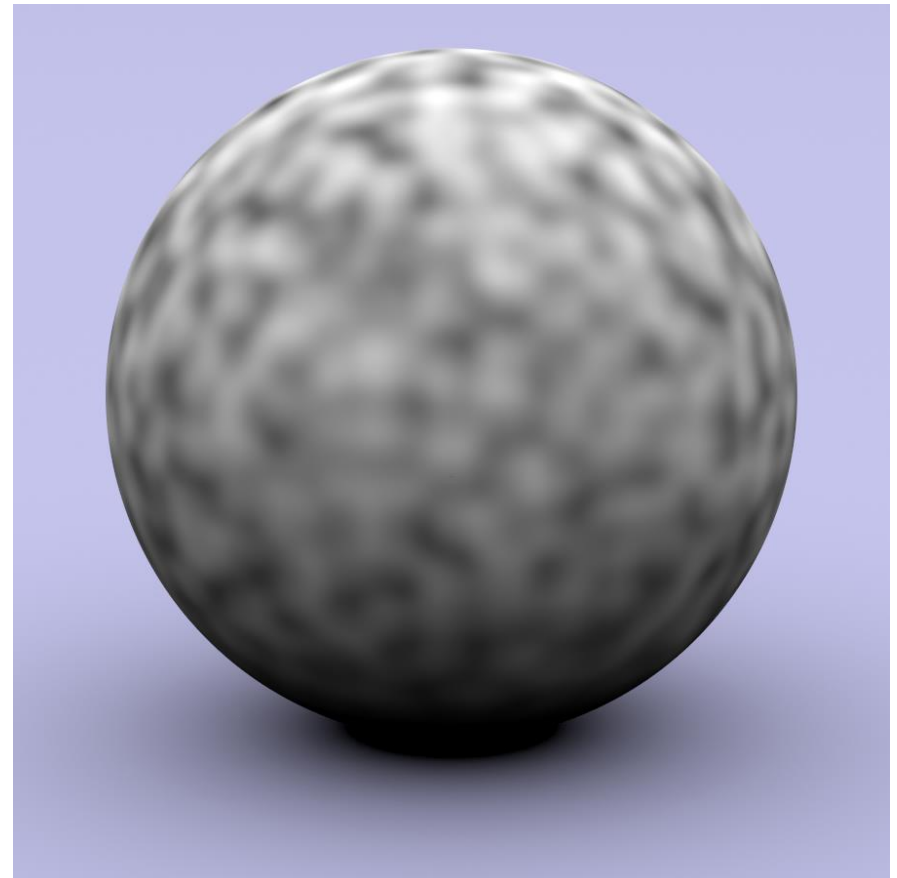
Smooth out randomness (1D)



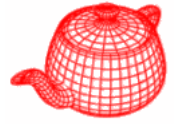
Perlin noise (v2)



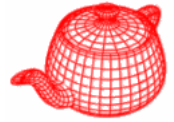
- Developed by Ken Perlin for TRON (1982)
- TRON is the first 3D shaded graphics in a Hollywood film
- Perlin used some tricks to improve the basic lattice approach to achieve “**determined and coherent** randomness”
- `float Noise(float x, float y, float z) in pbrt`



TRON



Perlin noise



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

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

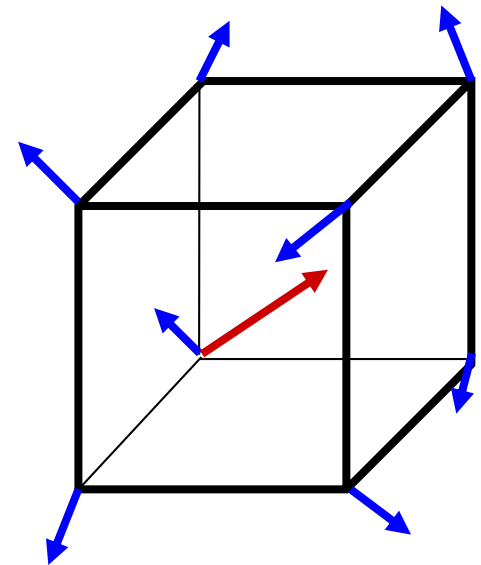
$$\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u, v, w))$$

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

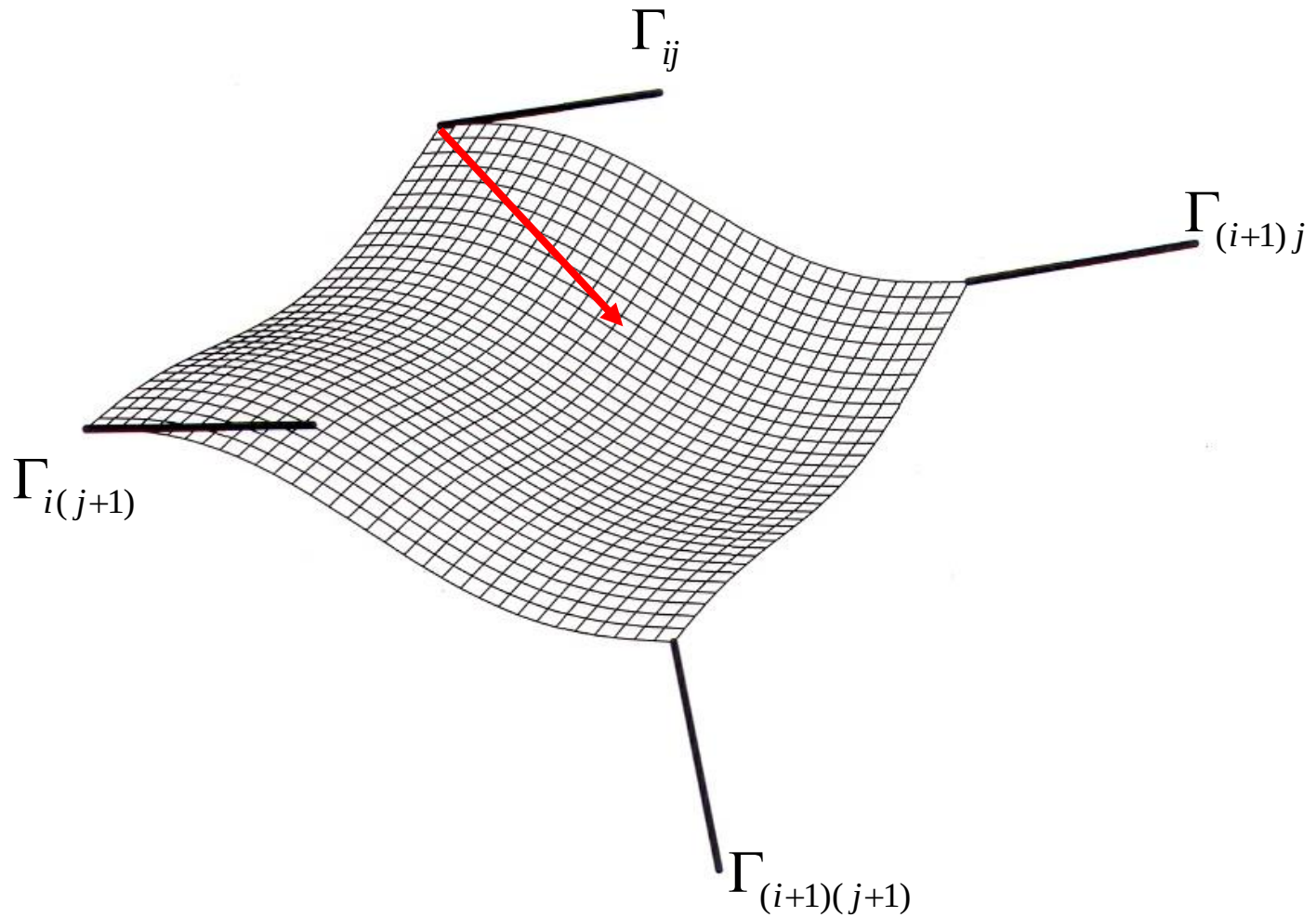
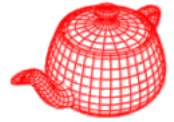
$$\Gamma_{ijk} = G(\phi(i + \phi(j + \phi(k))))$$

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

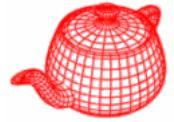
G is a precomputed array of n random unit vectors



Perlin noise (2D illustration)



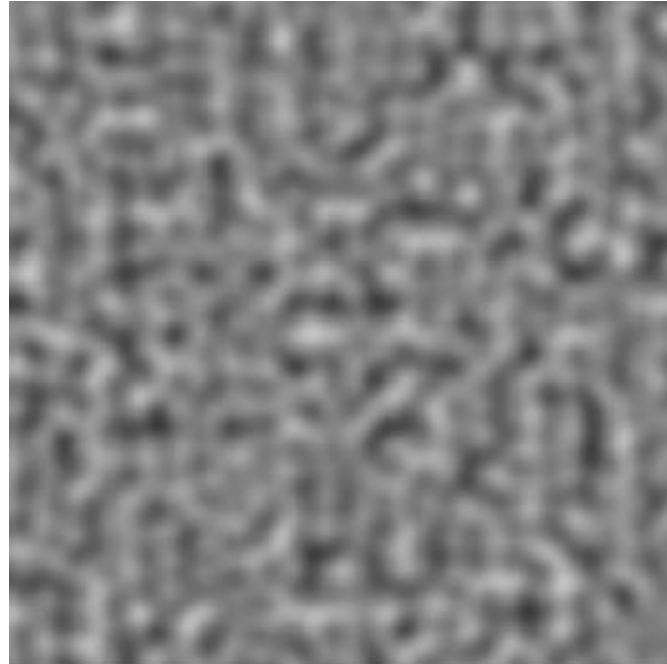
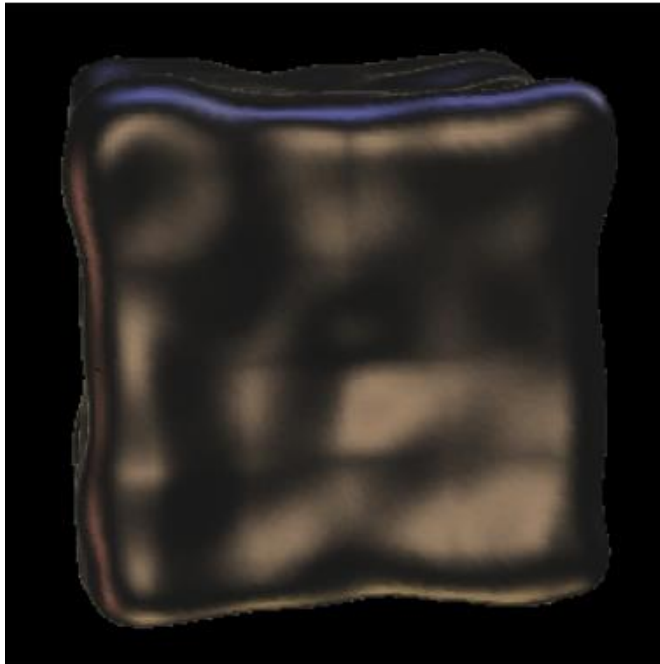
Improving Perlin noise (Perlin 2002)



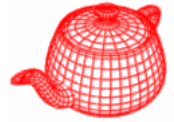
- Two deficiencies:

$12t-6$

- 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

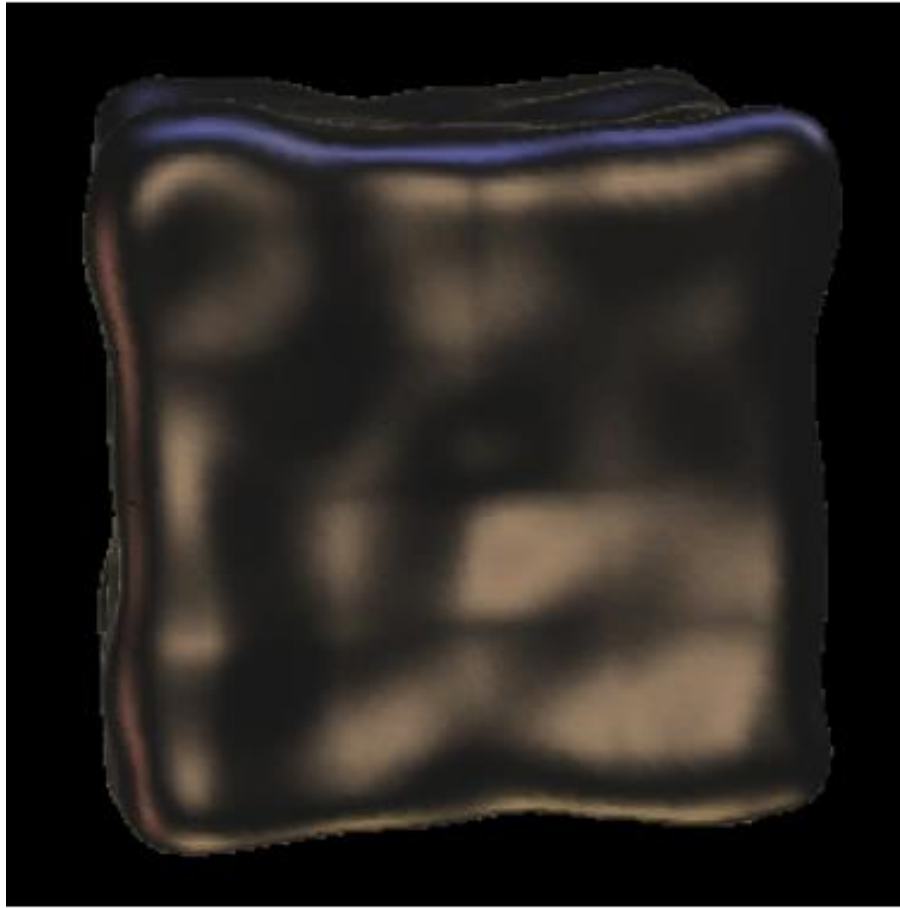
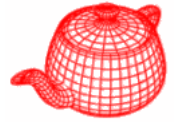


Improving Perlin noise (Perlin 2002)



- $\omega(t) = 6t^5 - 15t^4 + 10t^3 \quad 0 < t < 1$
 $30t^4 - 60t^3 + 30t^2$
 $120t^3 - 180t^2 + 60t$
- 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)

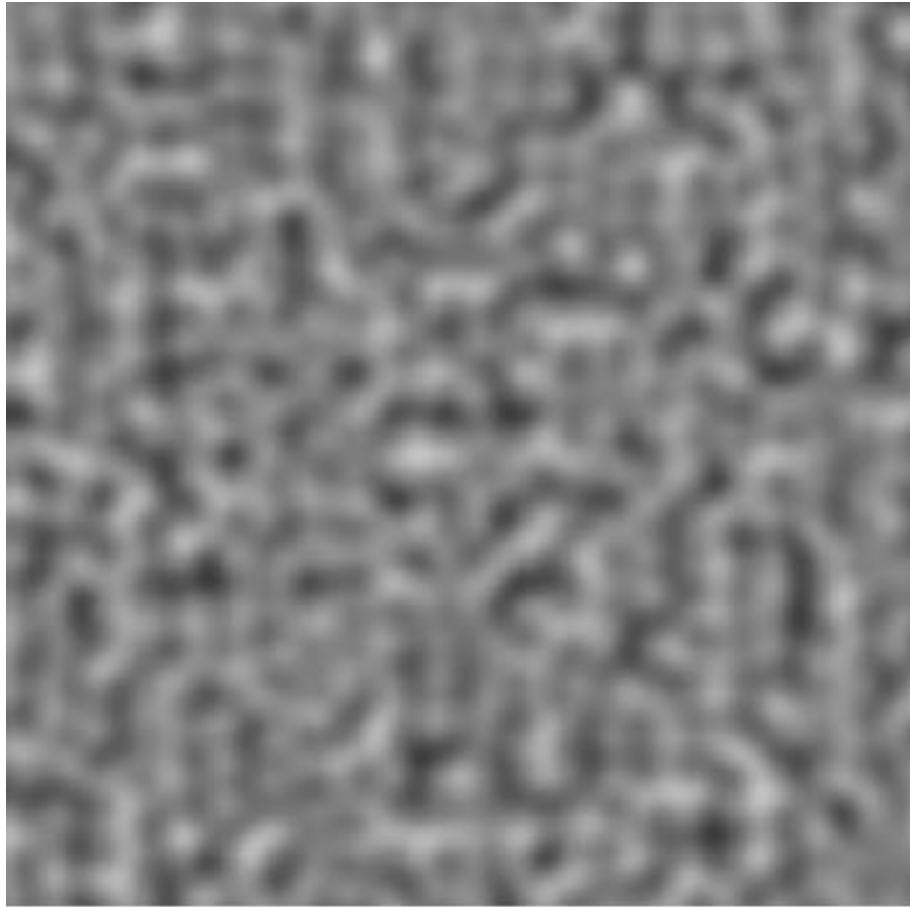
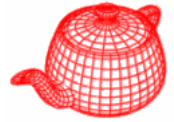


$$\omega(t) = 2t^3 - 3t^2 + 1$$

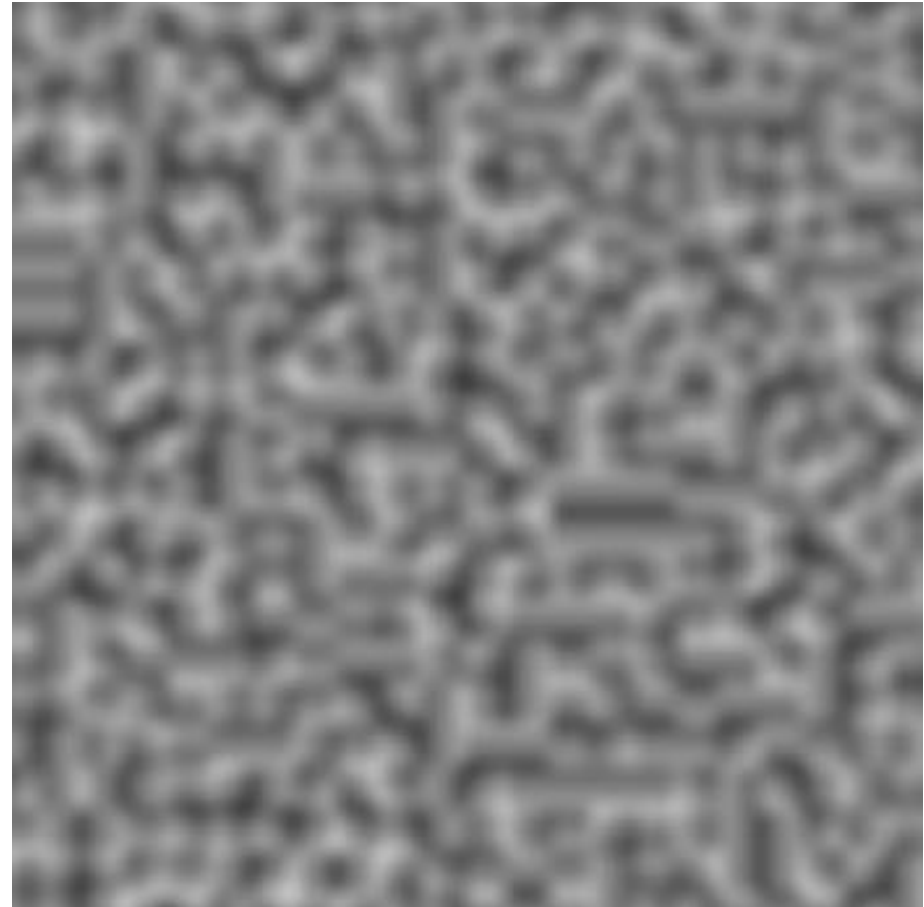


$$\omega(t) = 6t^5 - 15t^4 + 10t^3$$

Improving Perlin noise (Perlin 2002)

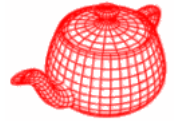


uniformly sampled from a sphere



use only 12 vectors;
also faster

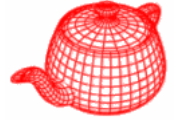
Random Polka dots (v2)



- 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;  
}
```

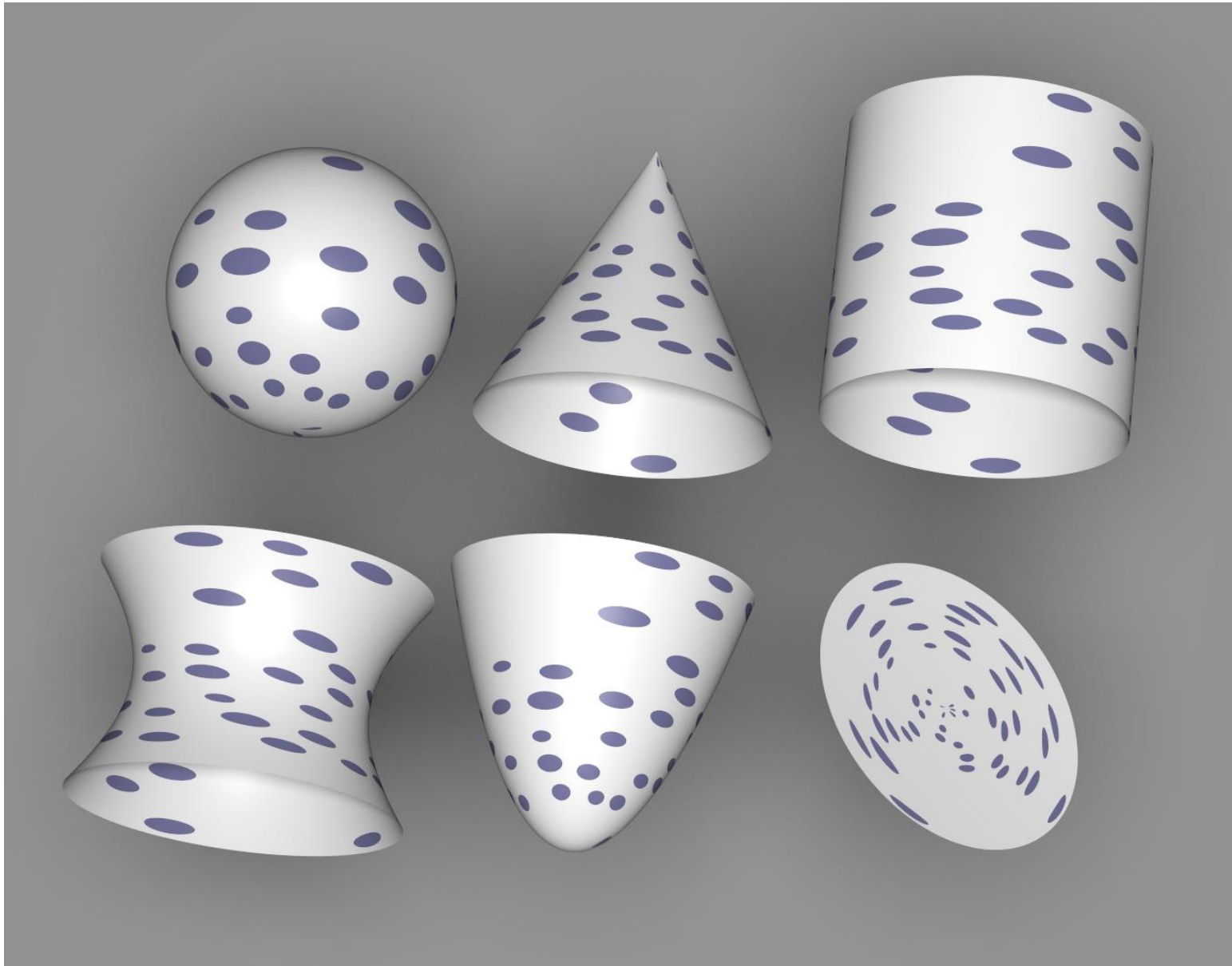
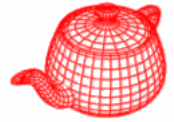
Random Polka dots



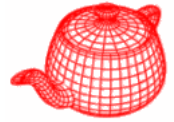
```
T Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg,&s,&t,&dsdx,&dtdx,&dsdy,&dtdy);
    int sCell = Floor2Int(s + .5f),
        tCell = Floor2Int(t + .5f);
    if (Noise(sCell+.5f, tCell+.5f) > 0) {
        float radius = .35f;
        float maxShift = 0.5f - radius;
        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);
}
```

It is deterministic so that it makes consistent decision for pixels.

Random Polka dots



Multiple-scale Perlin noise



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

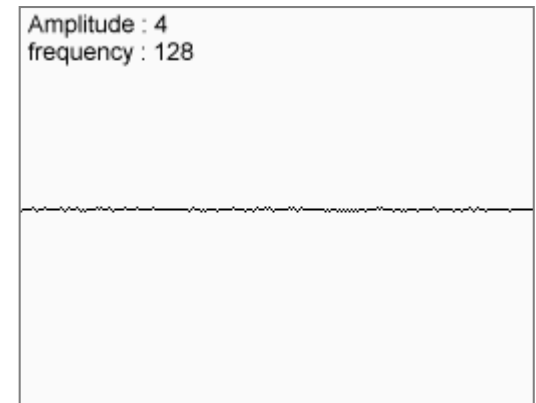
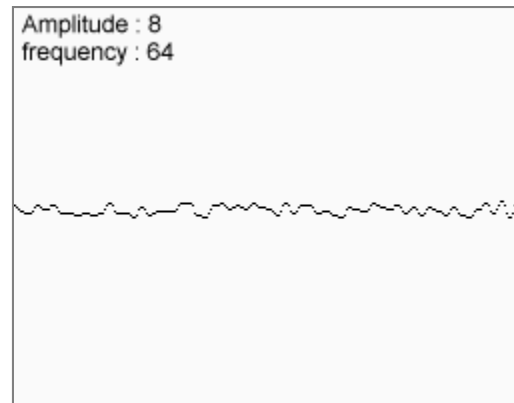
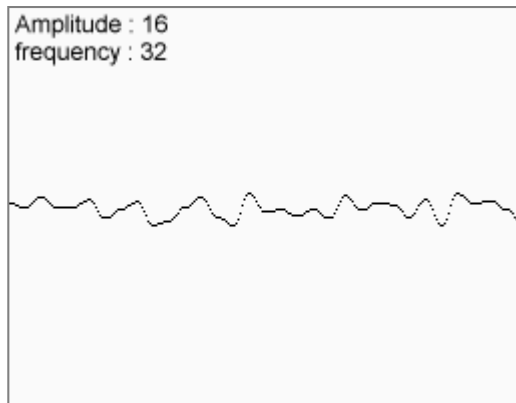
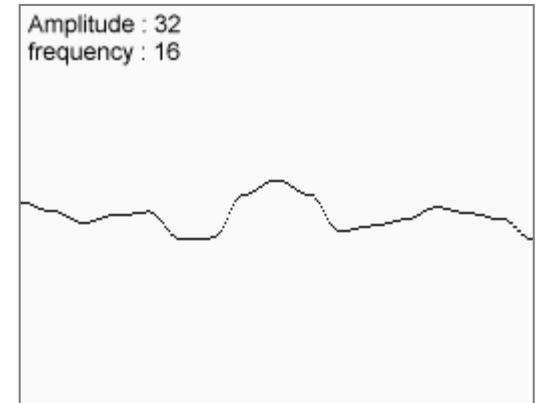
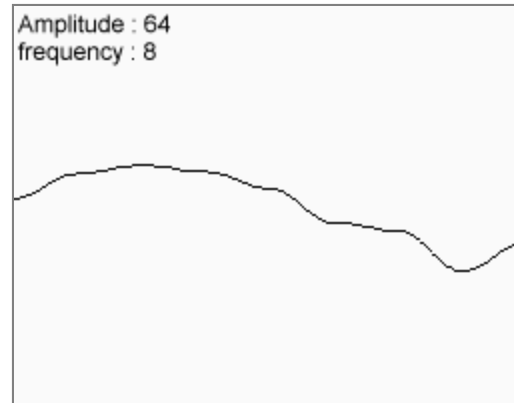
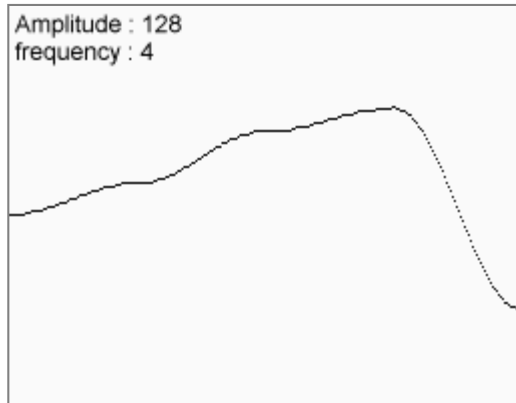
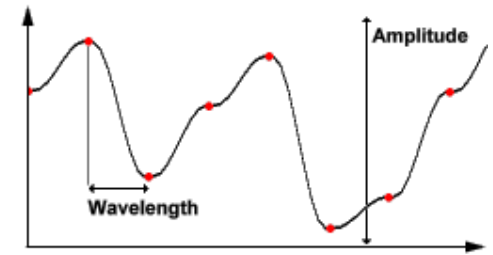
$$f_s(x) = \sum_i w_i f(s_i x)$$
$$w_i = \frac{1}{2} w_{i-1} \quad s_i = 2s_{i-1}$$

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

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

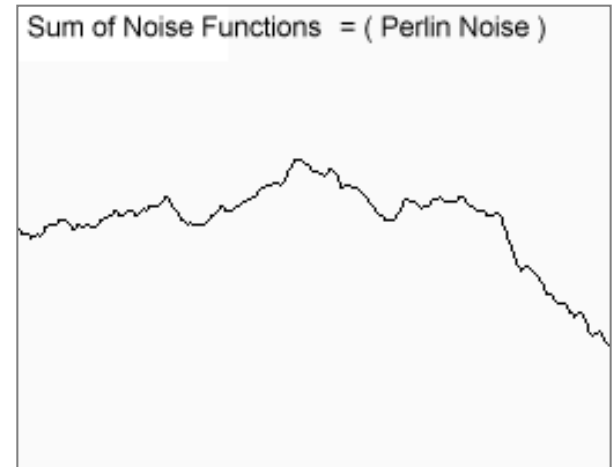
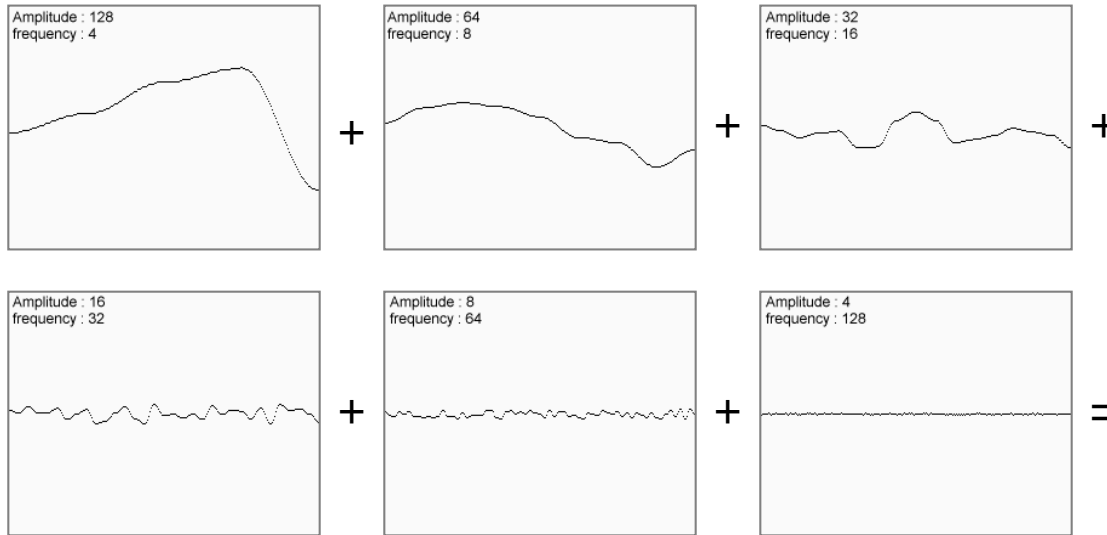
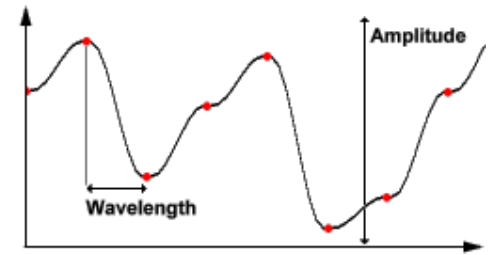
(1-D) Perlin Noise Function :

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

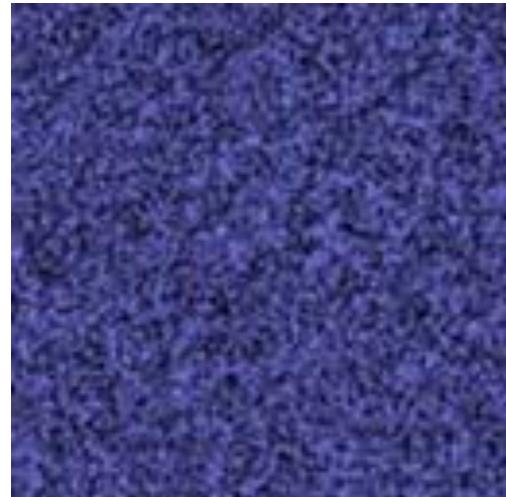
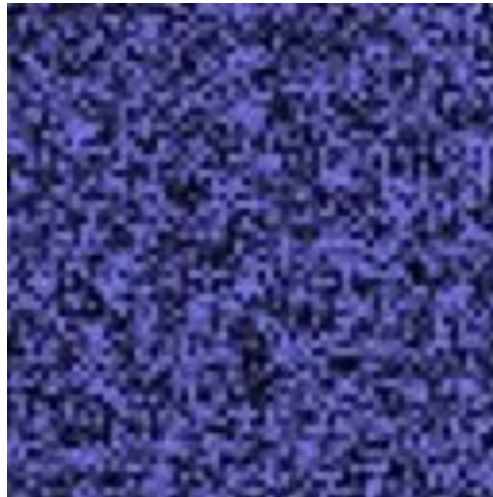
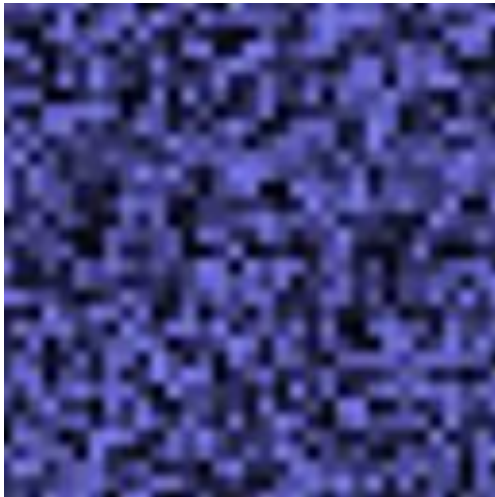
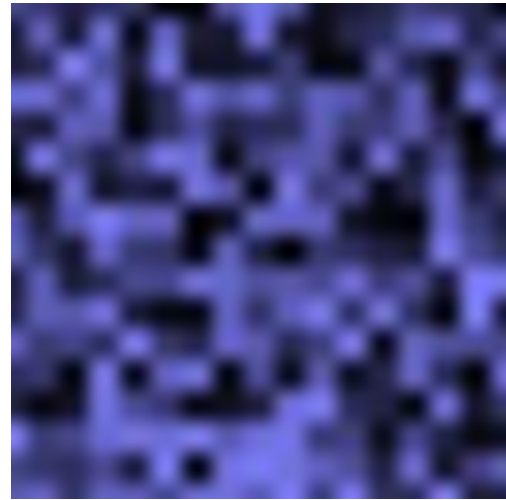
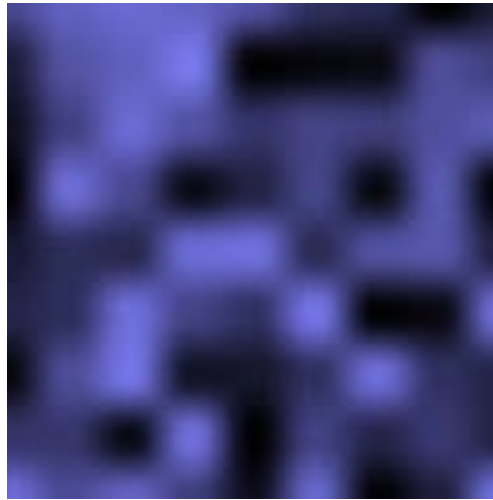
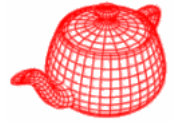


(1-D) Perlin Noise Function :

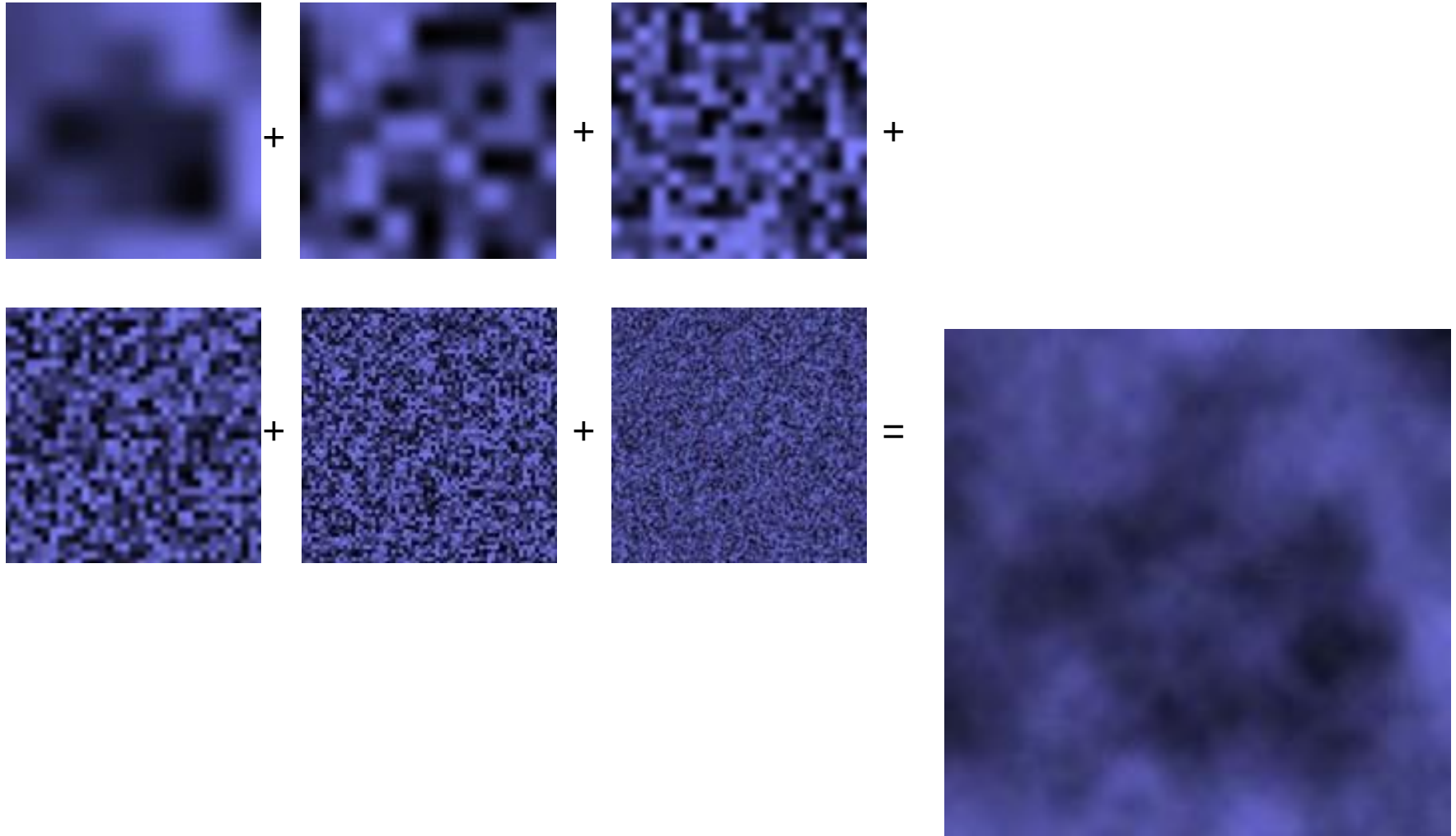
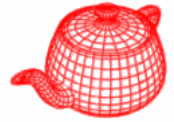
- Get lots of such smooth functions, with various frequencies and amplitudes
- Add them all together to create a nice noisy function



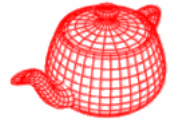
(2-D) Perlin Noise Function



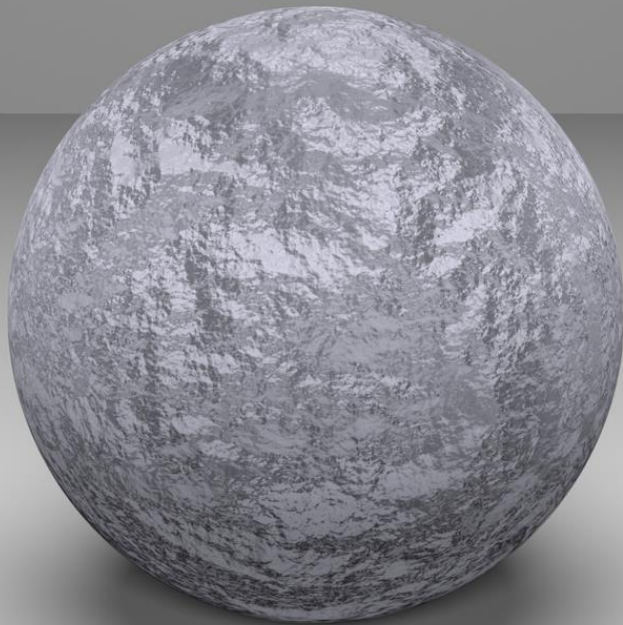
(2-D) Perlin Noise Function



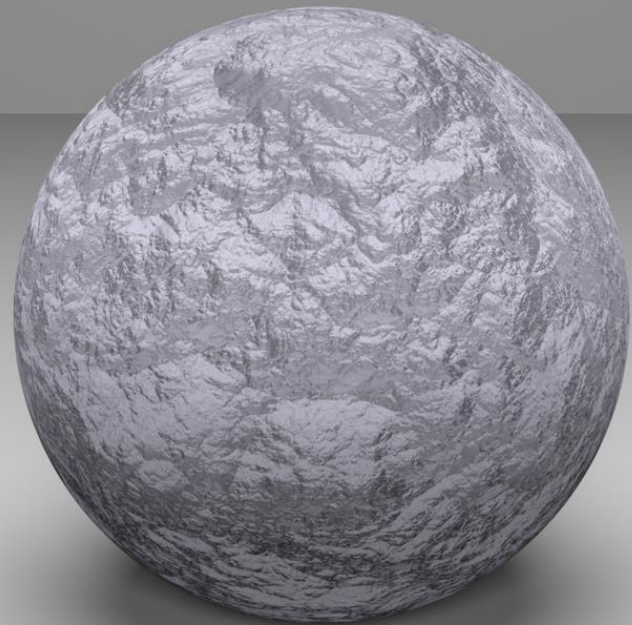
Bumpy and wrinkled textures



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

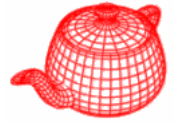


FBmTexture

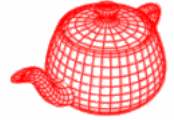


WrinkledTexture

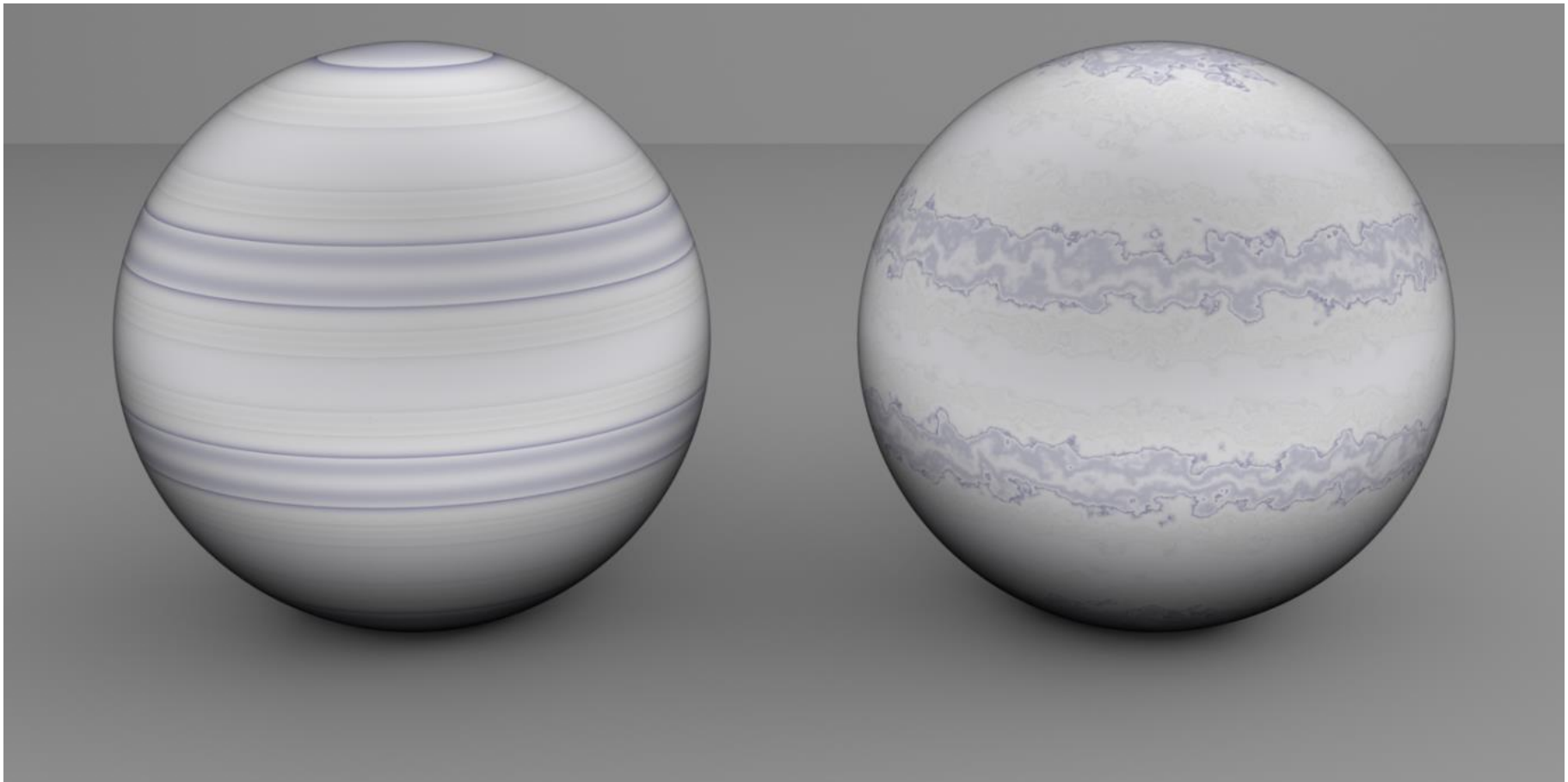
Windy waves



Marble



- Perturb texture coordinates before using another texture or lookup table

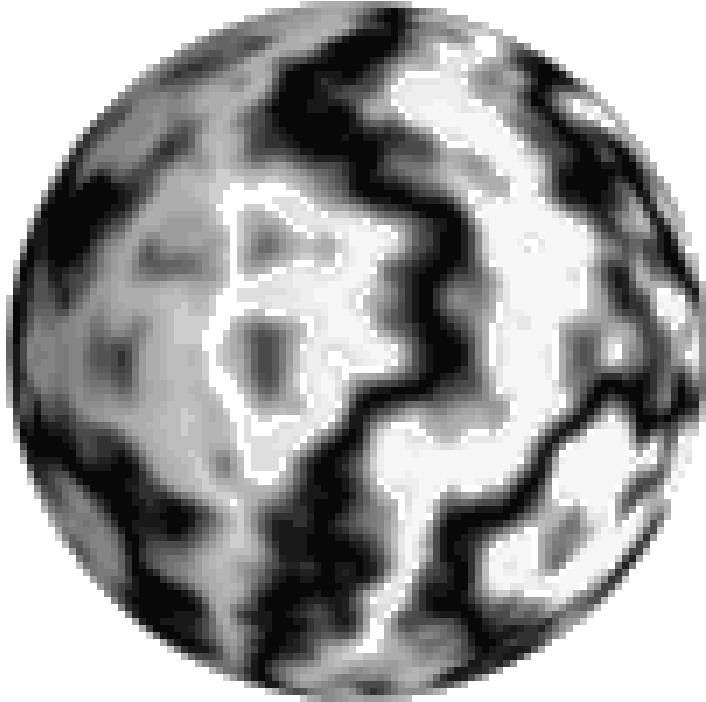


Texture Generation using 3D Perlin Noise



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)
```

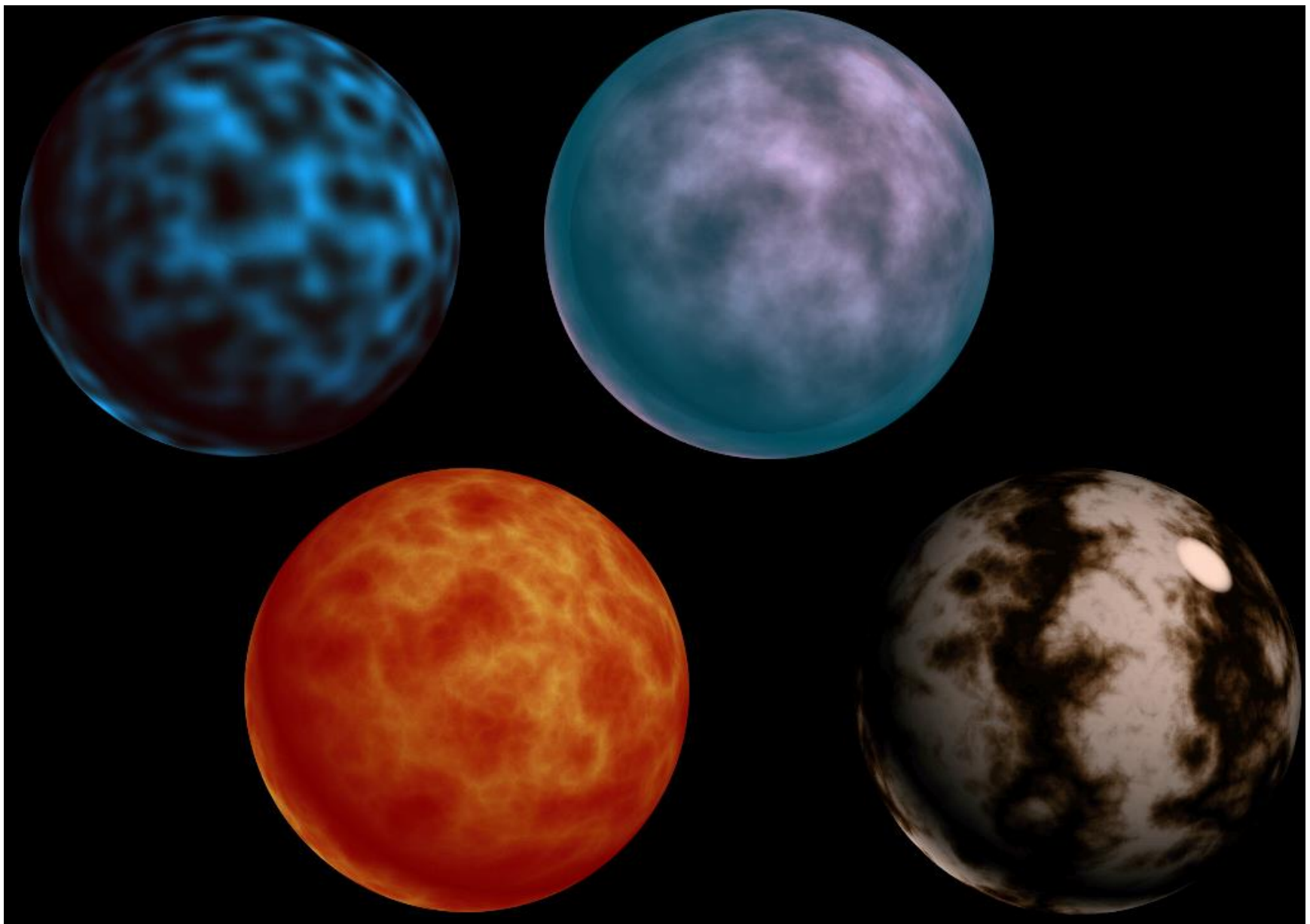
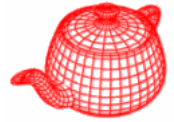


- The very fine bumps you can see on the wood are high frequency noise that has been stretched in one dimension.

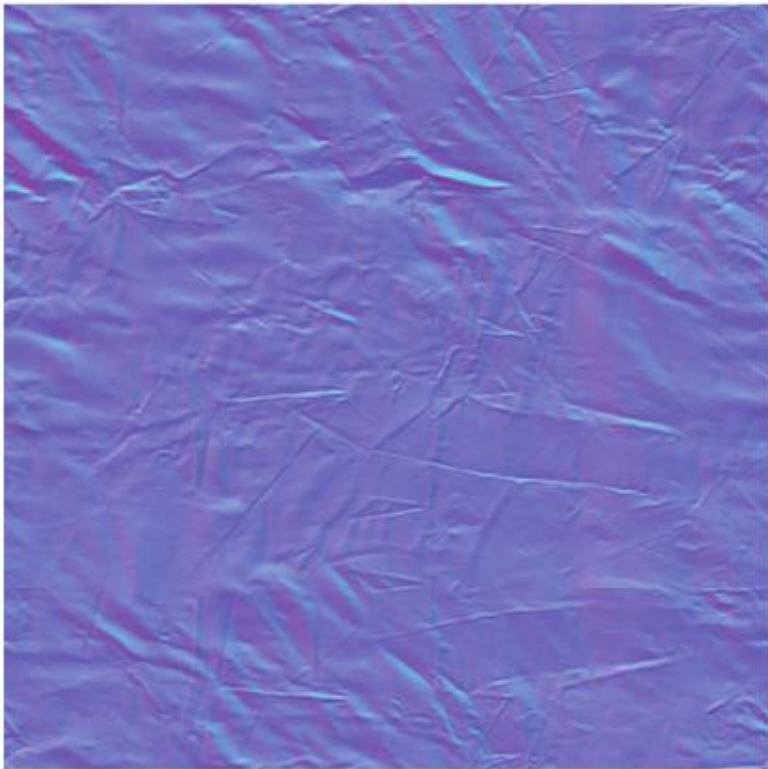
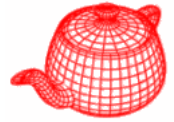
```
bumps = perlin(x*50,y*50,z*20)  
if bumps < .5  
then bumps = 0  
else bumps = 1
```



Creative use of Perlin noise



Normal mapping



Normal map

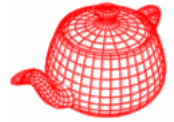


without

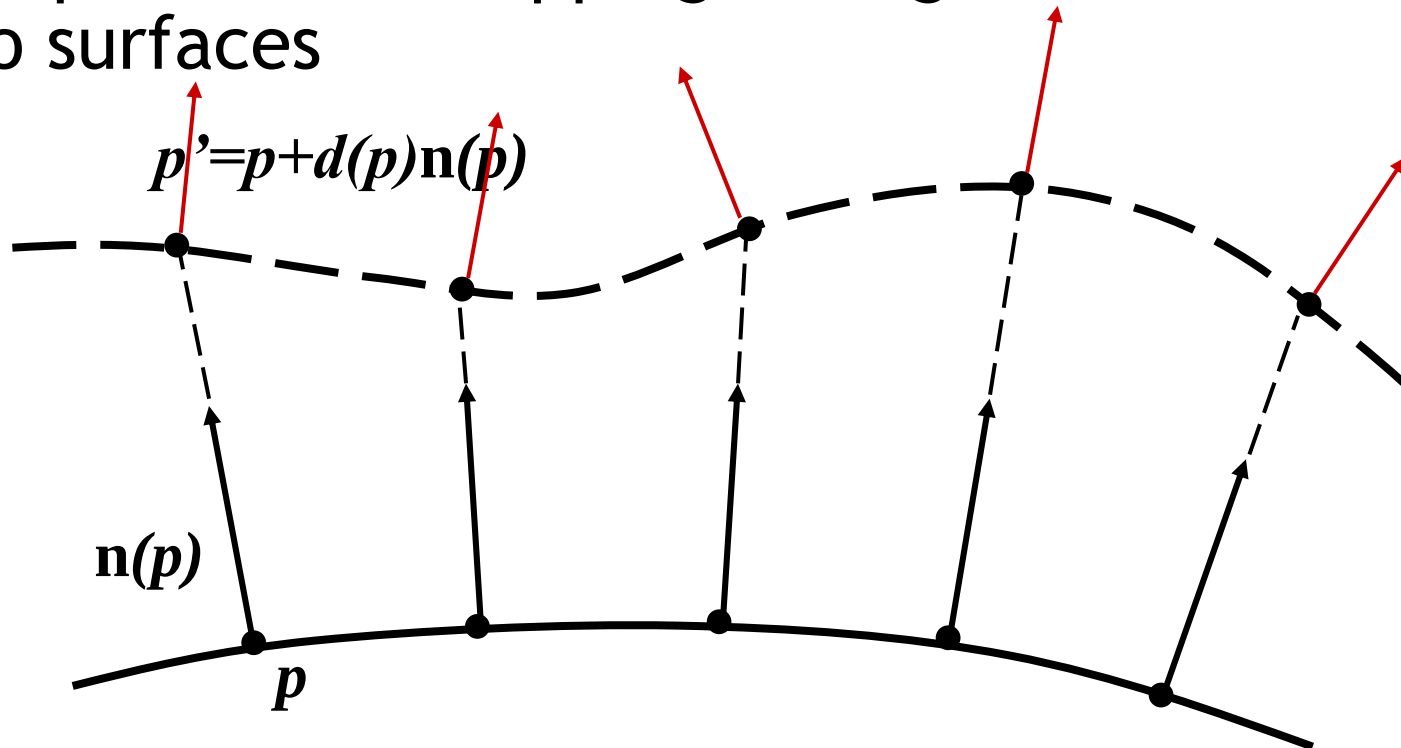


with

Bump mapping

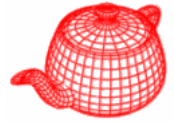


- Displacement mapping adds geometrical details to surfaces

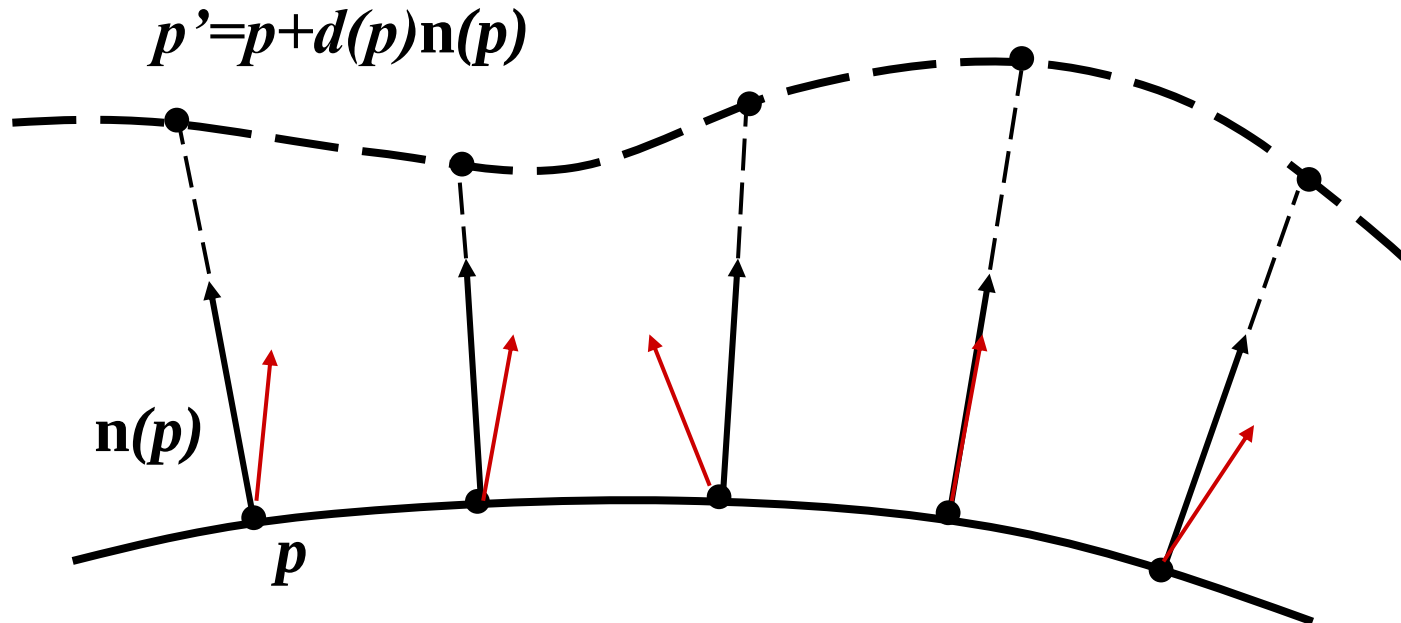


- Bump mapping: augments geometrical details to a surface without changing the surface itself
- It works well when the displacement is small

Bump mapping

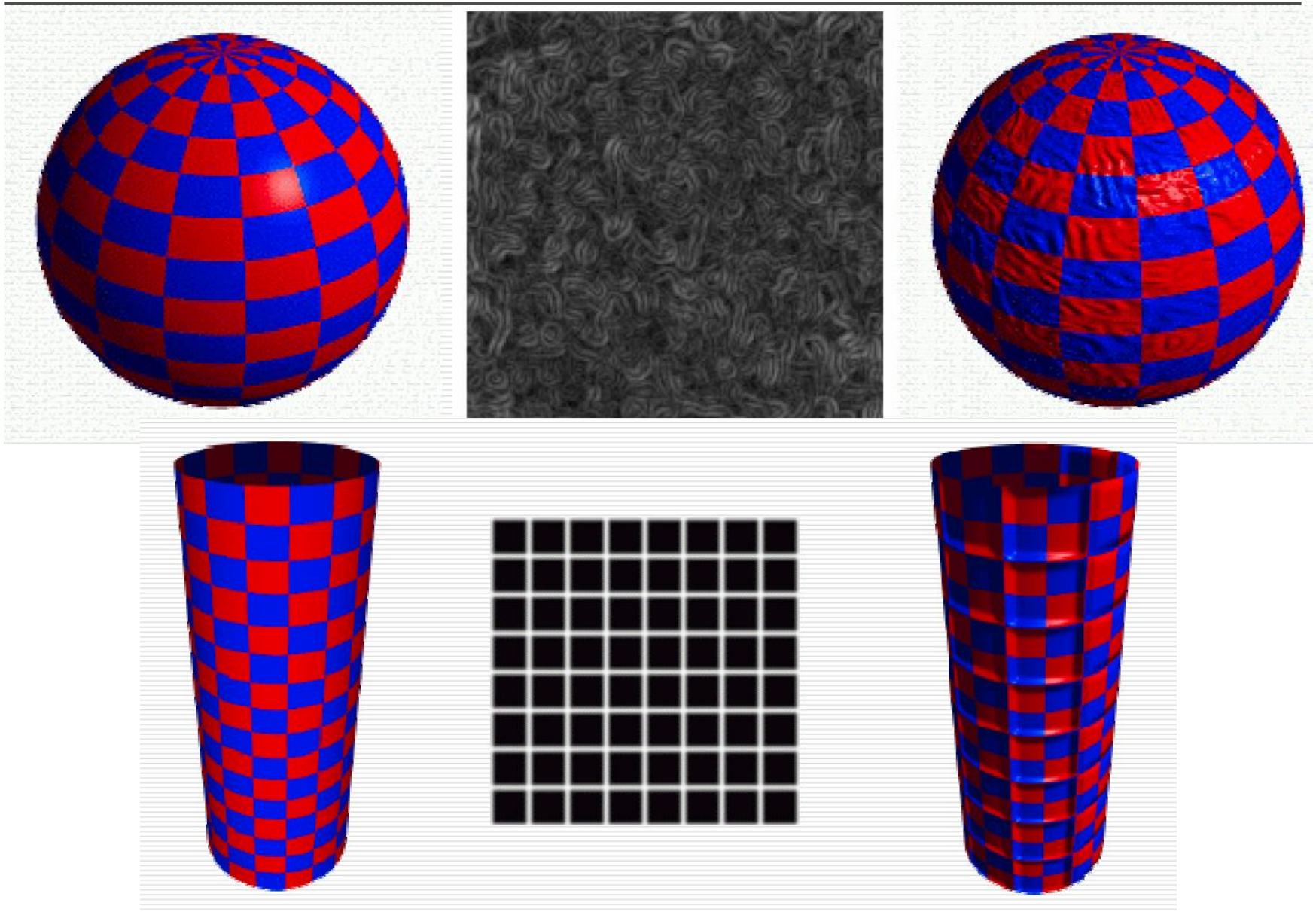
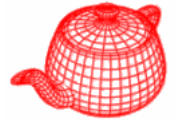


- Displacement mapping adds geometrical details to surfaces

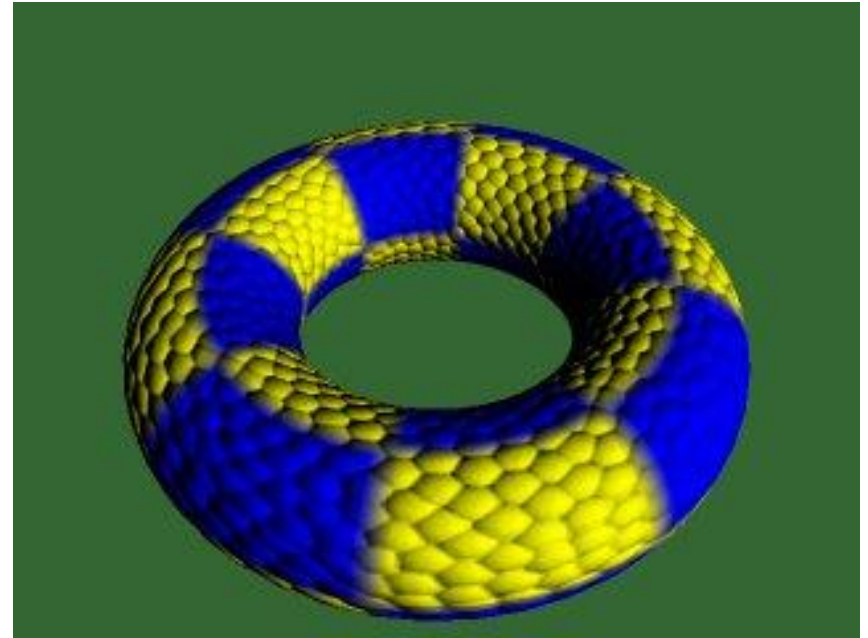
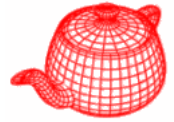


- Bump mapping: augments geometrical details to a surface without changing the surface itself
- It works well when the displacement is small

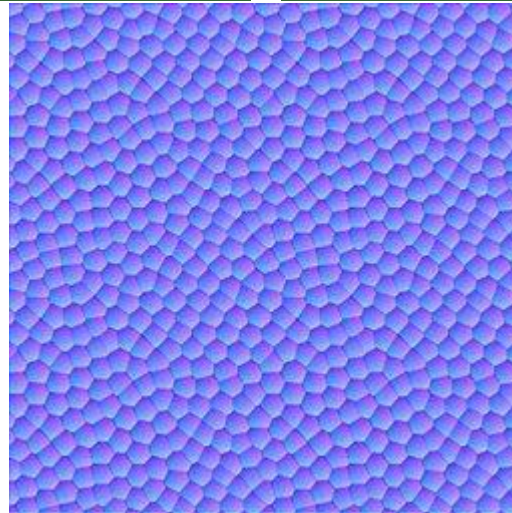
Bump mapping



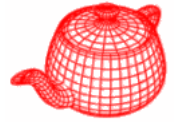
Bump mapping



To use bump mapping, a displacement map is often converted into a bump map (normal map)



Bump mapping



$$\mathbf{q}(u, v) = \mathbf{p}(u, v) + d(u, v)\mathbf{n}(u, v)$$

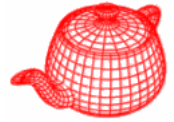
$$\mathbf{n}' = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}$$

the only unknown term

$$\frac{\partial \mathbf{q}}{\partial u} = \frac{\partial \mathbf{p}(u, v)}{\partial u} + \boxed{\frac{\partial d(u, v)}{\partial u}} \mathbf{n}(u, v) + d(u, v) \frac{\partial \mathbf{n}(u, v)}{\partial u}$$

$$\frac{\partial \mathbf{q}}{\partial u} \approx \frac{\partial \mathbf{p}}{\partial u} + \frac{d(u + \Delta_u, v) - d(u, v)}{\Delta_u} \mathbf{n} + d(u, v) \frac{\partial \mathbf{n}}{\partial u}$$

Without bump map



With bump map

