

Materials

Digital Image Synthesis

Yung-Yu Chuang

with slides by Robin Chen

Materials



- The renderer needs to determine which BSDFs to use at a particular point and their parameters.
- A surface shader, represented by **Material**, is bound to each primitive in the scene.
- **Material=BSDF+Texture** (canned materials)
- Material has a method that takes a point to be shaded and dynamically returns a BSDF object, a combination of several BxDFs with parameters from the texture. (it plays a role like shaders)
- **core/material.* materials/***

BSDFs



- **BSDF**=a collection of **BxDF** (BRDFs and BTDFs)
- A real material is likely a mixture of several specular, diffuse and glossy components.

```
class BSDF {
    ...
    const DifferentialGeometry dgShading;
    const float eta;
private:
    Normal nn, ng; // shading normal, geometry normal
    Vector sn, tn; // shading tangents
    int nBxDFs;
    #define MAX_BxDFs 8
    BxDF * bxdfs[MAX_BxDFs];
};
```

BSDF



```
BSDF::BSDF(const DifferentialGeometry &dg,
           const Normal &ngeom, float e)
    : dgShading(dg), eta(e) { refraction index of the medium surrounding the shading point.
    ng = ngeom;
    nn = dgShading.nn;
    sn = Normalize(dgShading.dpdu);
    tn = Cross(nn, sn);
    nBxDFs = 0;
}

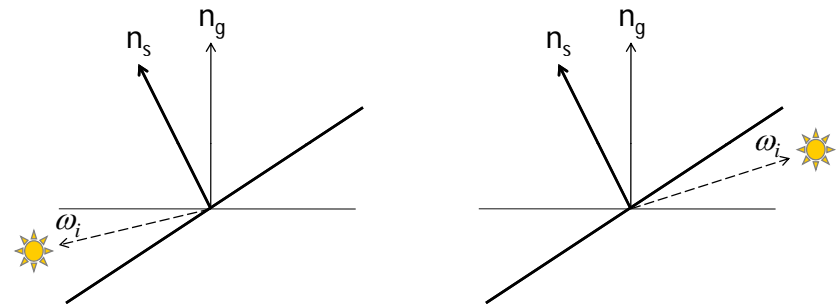
inline void BSDF::Add(BxDF *b) {
    Assert(nBxDFs < MAX_BxDFs);
    bxdfs[nBxDFs++] = b;
}
```

BSDF



```
Spectrum BSDF::f(const Vector &woW, const Vector &wiW,
                BxDFType flags) {
    Vector wi=WorldToLocal(wiW), wo=WorldToLocal(woW);
    if (Dot(wiW, ng) * Dot(woW, ng) > 0)
        // ignore BTDFs Use geometry normal not shading normal
        // to decide the side to avoid light leak
        flags = BxDFType(flags & ~BSDF_TRANSMISSION);
    else
        // ignore BRDFs
        flags = BxDFType(flags & ~BSDF_REFLECTION);
    Spectrum f = 0.;
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(flags))
            f += bxdfs[i]->f(wo, wi);
    return f;
}
```

Light leak



Material



- `Material::GetBSDF()` determines the reflective properties for a given point on the surface.

```
class Material : public ReferenceCounted {
public:
    // real geometry around intersection
    virtual BSDF *GetBSDF(DifferentialGeometry &dgGeom,
                          DifferentialGeometry &dgShading) const = 0;
    // shading geometry around intersection
    virtual BSDF *GetBSSRDF(...);
    static void Bump(Reference<Texture<float> > d,
                    const DifferentialGeometry &dgGeom,
                    const DifferentialGeometry &dgShading,
                    DifferentialGeometry *dgBump);
};
```

Calculate the normal according to the bump map

Matte



- Purely diffuse surface (only one component which is either Lambertian or Oren-Nayar)
- ```
class MatteMaterial : public Material {
public:
 Matte(Reference<Texture<Spectrum> > kd,
 Reference<Texture<float> > sig,
 Reference<Texture<float> > bump)
 { Kd = kd; sigma = sig; bumpMap = bump; }
 BSDF *GetBSDF(const DifferentialGeometry &dgGeom,
 const DifferentialGeometry &dgShading) const;
private:
 Reference<Texture<Spectrum> > Kd;
 Reference<Texture<float> > sigma, bumpMap;
};
```
- essentially a height map

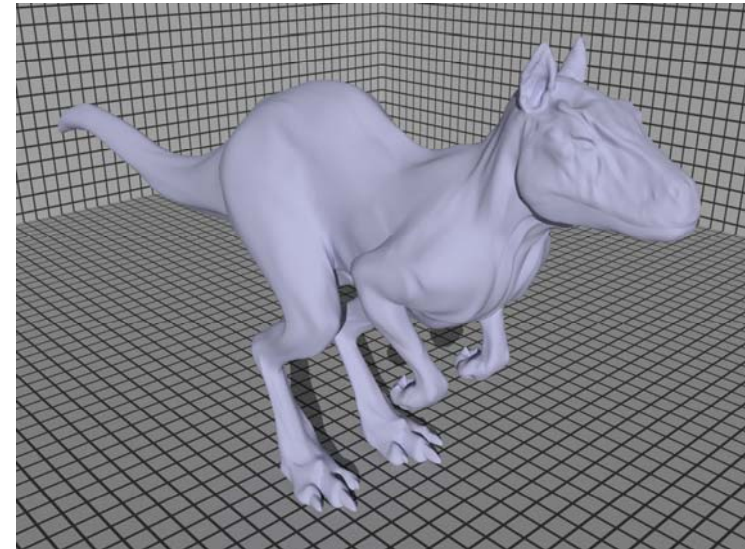
## Matte



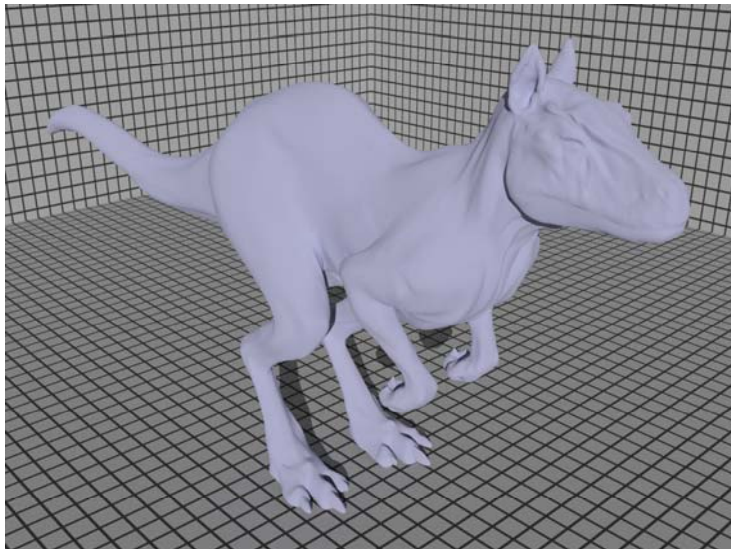
```
BSDF *Matte::GetBSDF(DifferentialGeometry &dgGeom,
 DifferentialGeometry &dgShading) {
 DifferentialGeometry dgs;
 if (bumpMap) Bump(bumpMap, dgGeom, dgShading, &dgs);
 else dgs = dgShading;
 BSDF *bsdf = BSDF_ALLOC(BSDF)(dgs, dgGeom.nn);

 Spectrum r = Kd->Evaluate(dgs).Clamp();
 float sig = Clamp(sigma->Evaluate(dgs), 0.f, 90.f);
 if (sig == 0.)
 bsdf->Add(BSDF_ALLOC(Lambertian)(r));
 else
 bsdf->Add(BSDF_ALLOC(OrenNayar)(r, sig));
 return bsdf;
}
```

## Lambertian



## Oren-Nayar model



## Plastic



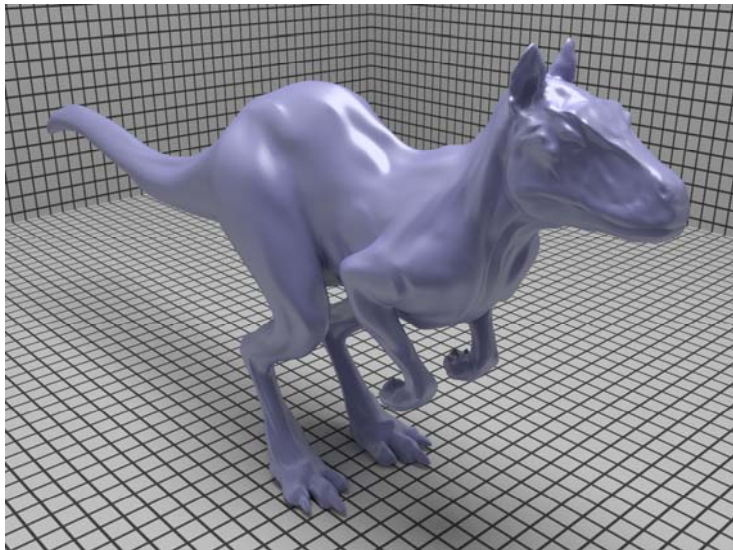
- A mixture of a diffuse and glossy scattering function with parameters, **kd**, **ks** and **rough**

```
BSDF *PlasticMaterial::GetBSDF(...) {
 <Allocate BSDF, possibly doing bump-mapping>
 Spectrum kd = Kd->Evaluate(dgs).Clamp();
 BxDF *diff = BSDF_ALLOC(Lambertian)(kd);

 Fresnel *f=BSDF_ALLOC(FresnelDielectric)(1.5f,1.f);
 Spectrum ks = Ks->Evaluate(dgs).Clamp();
 float rough = roughness->Evaluate(dgs);
 BxDF *spec = BSDF_ALLOC(Microfacet)(ks, f,
 BSDF_ALLOC(Blinn)(1.f / rough));

 bsdf->Add(diff); bsdf->Add(spec); return bsdf;
}
```

## Plastic



## Mix material



- Combine two materials with a varying weight.

```
class MixMaterial : public Material {
public:
 MixMaterial(Reference<Material> mat1,
 Reference<Material> mat2,
 Reference<Texture<Spectrum> > sc)
 : m1(mat1), m2(mat2), scale(sc)
 { }
 ...
private:
 Reference<Material> m1, m2;
 Reference<Texture<Spectrum> > scale;
};
```

## Mix material



```
BSDF *MixMaterial::GetBSDF(...) const
{
 BSDF *b1 = m1->GetBSDF(dgGeom, dgShading);
 BSDF *b2 = m2->GetBSDF(dgGeom, dgShading);
 Spectrum s1 = scale->Evaluate(dgShading).Clamp();
 Spectrum s2 = (Spectrum(1.f) - s1).Clamp();
 int n1 = b1->NumComponents(),
 n2 = b2->NumComponents();
 for (int i = 0; i < n1; ++i)
 b1->bxdfs[i]
 = BSDF_ALLOC(ScaledBxDF)(b1->bxdfs[i], s1);
 for (int i = 0; i < n2; ++i)
 b1->Add(BSDF_ALLOC(ScaledBxDF)(b2->bxdfs[i], s2));
 return b1;
}
```

## Measured material



```
class MeasuredMaterial : public Material {
public:
 MeasuredMaterial(const string &filename,
 Reference<Texture<float> > bump);
 ...
private:
 KdTree<IrregIsotropicBRDFSample> *thetaPhiData;
 float *regularHalfangleData;
 uint32_t nThetaH, nThetaD, nPhiD;
 Reference<Texture<float> > bumpMap;
};
```



## Measured material



```
BSDF *MeasuredMaterial::GetBSDF(...) const
{
 <bump mapping>
 BSDF *bsdf = BSDF_ALLOC(BSDF)(dgs, dgGeom.nn);
 if (regularHalfangleData)
 bsdf->Add(BSDF_ALLOC(RegularHalfangleBRDF)
 (regularHalfangleData, nThetaH, nThetaD, nPhiD));
 else if (thetaPhiData)
 bsdf->
 Add(BSDF_ALLOC(IrregIsotropicBRDF)(thetaPhiData));

 return bsdf;
}
```

## Additional materials

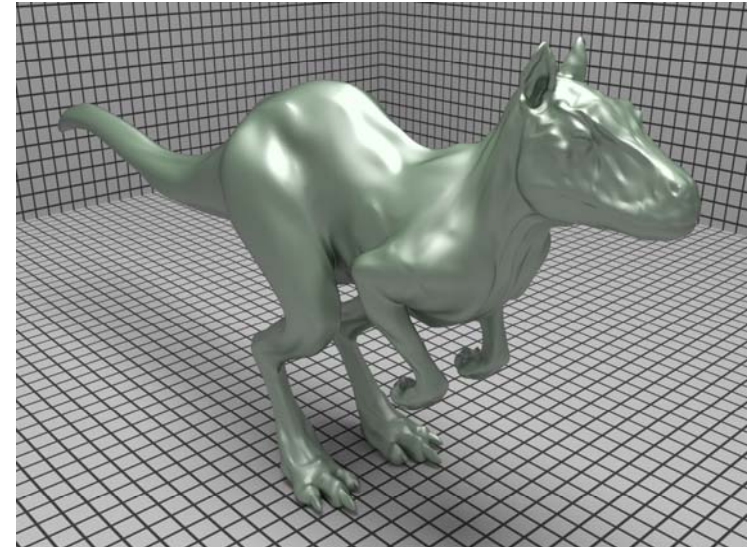


- There are totally 13 material classes available in pbrt.
- **Glass**: reflection and transmission, Fresnel weighted
- **Metal**: a metal surface with perfect specular reflection
- **Mirror**: perfect specular reflection
- **Substrate**: layered-model
- **Subsurface** and **KdSubsurface**: BSSRDF
- **Translucent**: glossy transmission
- **Uber**: a “union” of previous material; highly parameterized

## Metal



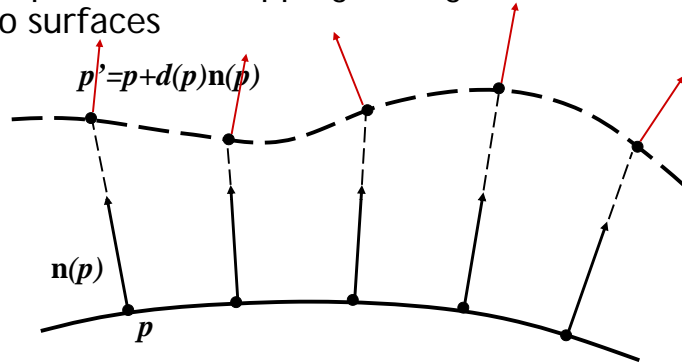
## Substrate



## Bump mapping



- Displacement mapping adds geometrical details to surfaces

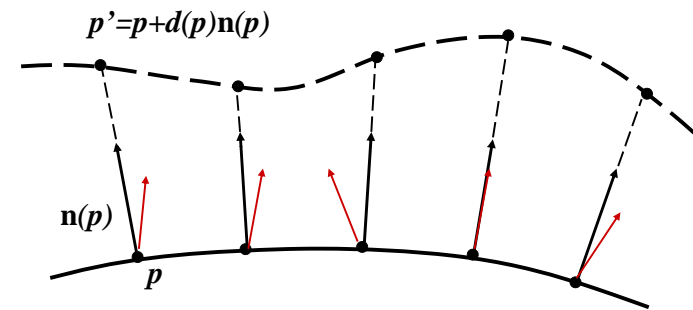


- Bump mapping: augments geometrical details to a surface without changing the surface itself
- It works well when the displacement is small

## Bump mapping

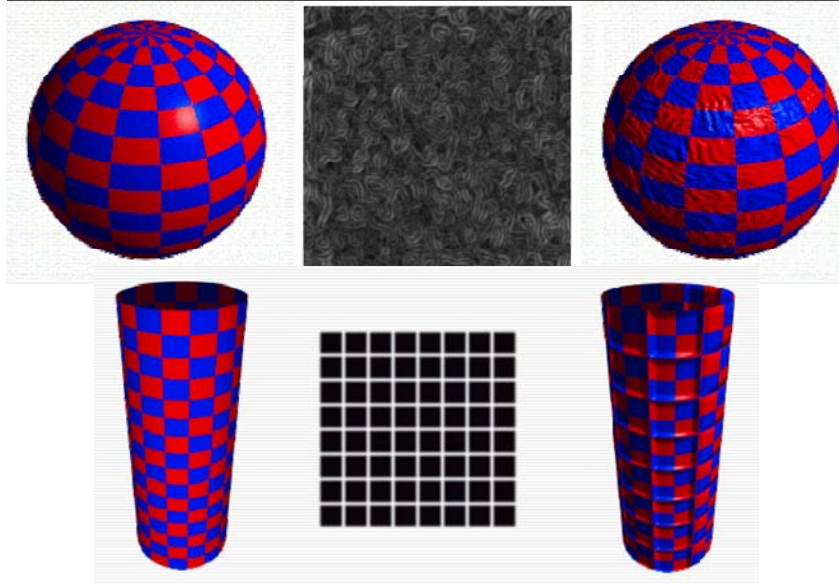


- Displacement mapping adds geometrical details to surfaces

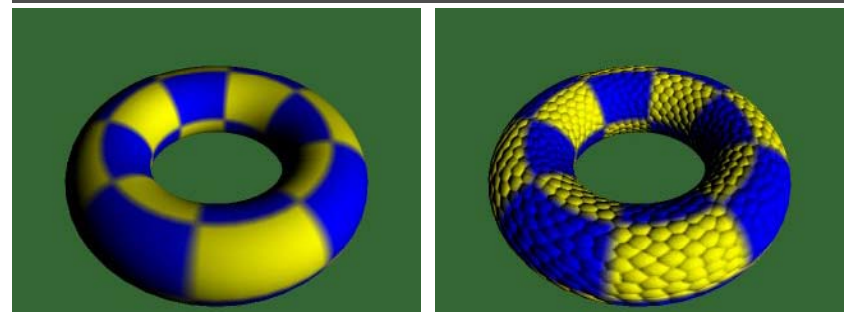


- Bump mapping: augments geometrical details to a surface without changing the surface itself
- It works well when the displacement is small

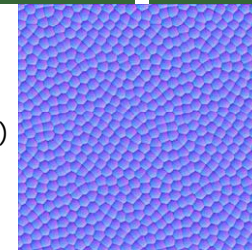
## Bump mapping



## 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

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

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

often ignored because  $d(u, v)$  is small. But, adding constant to  $d$  won't change appearance then. Pbrt adds this term.

`Material::Bump(...)` does the above calculation

## Material::Bump



```
DifferentialGeometry dgEval = dgs;
// Shift _dgEval_ in the u direction
float du =
 0.5*(fabsf(dgs.dudx)+fabsf(dgs.dudy));
if (du == 0.f) du = .01f;
dgEval.p = dgs.p + du * dgs.dpdu;
dgEval.u = dgs.u + du;
dgEval.nn =
 Normalize((Normal)Cross(dgs.dpdu, dgs.dpdv)
 + du*dgs.dndu);
float uDisplace = d->Evaluate(dgEval);
// do similarly for v
float displace = d->Evaluate(dgs);
```

## Material::Bump



```
*dgBump = dgs;
dgBump->dpdu = dgs.dpdu
 + (uDisplace-displace)/du * Vector(dgs.nn)
 + displace * Vector(dgs.dndu);
dgBump->dpdv = ...
dgBump->nn = Normal(Normalize(
 Cross(dgBump->dpdu, dgBump->dpdv)));
...
// Orient shading normal to match geometric normal
dgBump->nn
 = Faceforward(dgBump->nn, dgGeom.nn);
```

