Materials

Digital Image Synthesis

Yung-Yu Chuang

11/19/2008

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 returns a BSDF object, a combination of several BxDFs with parameters from the texture.
- **`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];
  static MemoryArena arena;
};
```

BSDF

```
BSDF::BSDF(const DifferentialGeometry &dg,
           const Normal &ngeom, float e)
  : dgShading(dg), eta(e) {
  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;
}
```

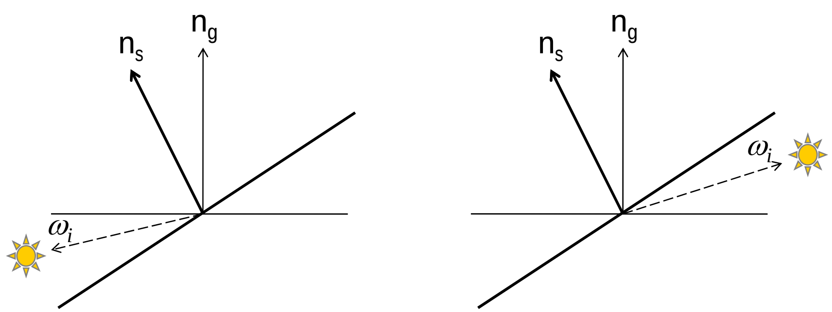
refraction index of the medium surrounding the shading point.

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
    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;
}
```

Use geometry normal not shading normal to decide the side to avoid light leak

Light leak



Material

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

```
class Material : public ReferenceCounted {
public:
  virtual BSDF *GetBSDF(DifferentialGeometry &dgGeom,
          DifferentialGeometry &dgShading) const = 0;
  virtual ~Material();
  static void Bump(Reference<Texture<float> > d,
              const DifferentialGeometry &dgGeom,
              const DifferentialGeometry &dgShading,
              DifferentialGeometry *dgBump);
};
```

real geometry around intersection

shading geometry around intersection

Calculate the normal according to the bump map

Matte

- Purely diffuse surface

```
class Matte : 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;
};
```

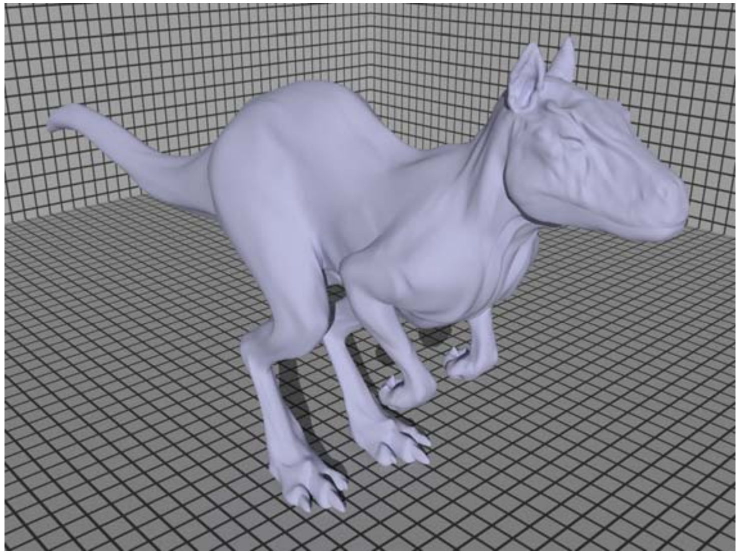
essentially a height map

Matte

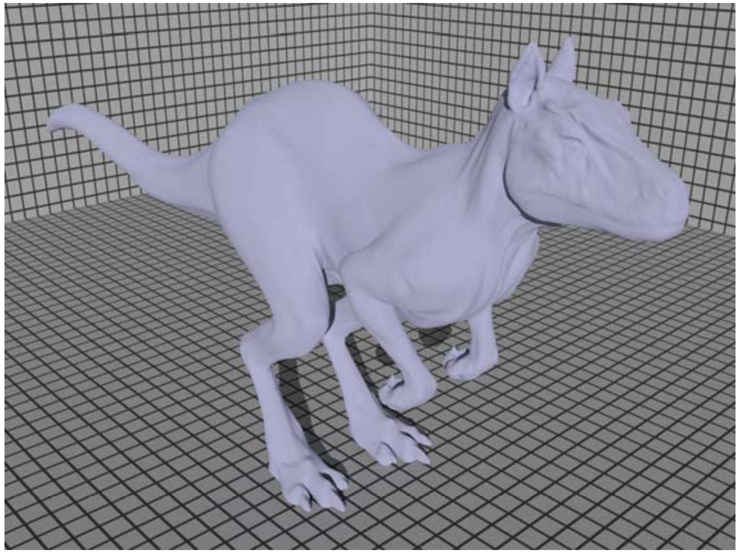
```
BSDF *Matte::GetBSDF(DifferentialGeometry &dgGeom,
                     DifferentialGeometry &dgShading) {
  DifferentialGeometry dgs;
  if (bumpMap) Bump(bumpMap, dgGeom, dgShading, &dgs);
  else dgs = dgShading;
  BSDF *bsdf = BSDF_ALLOC(BSDF)(dgs, dgGeom.nn);

  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-Nayer model



Plastic

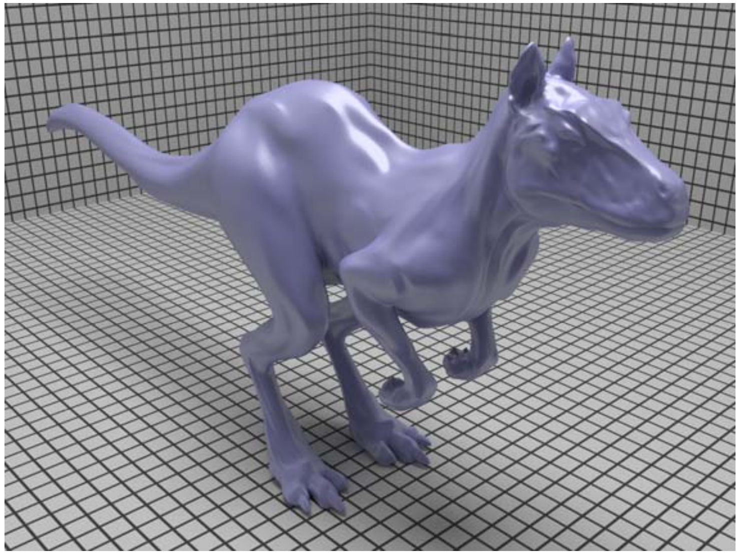
- A mixture of a diffuse and glossy scattering function with parameters, **kd**, **ks** and **rough**

```
BSDF *Plastic::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;
}
```

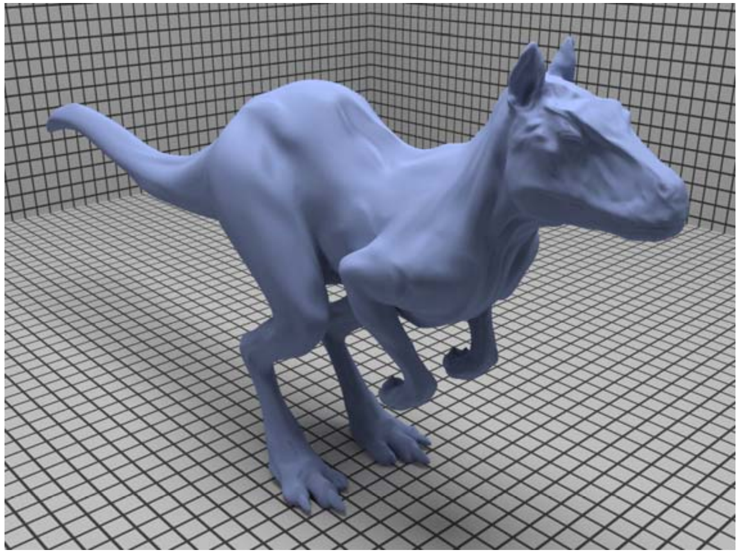
Plastic



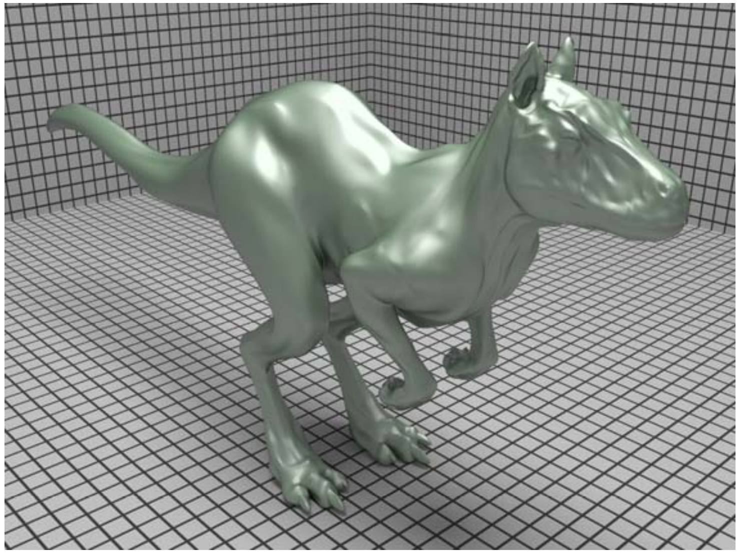
Additional materials

- There are totally 14 material plug-ins available in pbrt. Most of them are just variations of **`Matte`** and **`Plastic`**.
- **`Translucent`**: glossy transmission
- **`Mirror`**: perfect specular reflection
- **`Glass`**: reflection and transmission, Fresnel weighted
- **`ShinyMetal:`** a metal surface with perfect specular reflection
- **`Substrate:`** layered-model
- **`Clay, Felt, Primer, Skin, BluePaint, Brushed Metal`**: measured data fitted by Lafortune
- **`Uber:`** a "union" of previous material; highly parameterized

Blue paint (measured Lafortune)

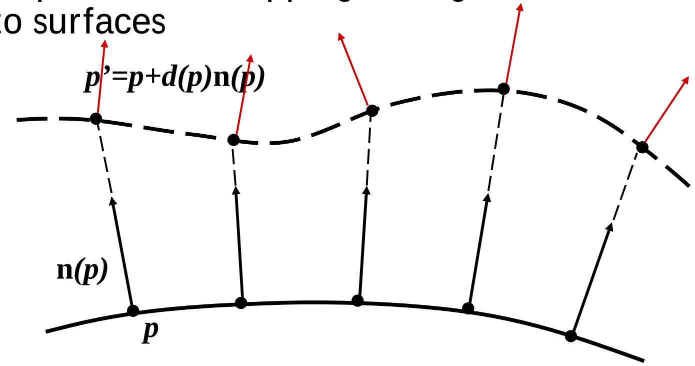


Substrate



Bump mapping

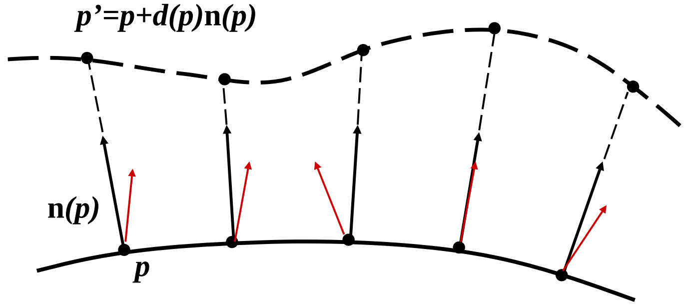
- Displacement mapping adds geometrical details to surfaces



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

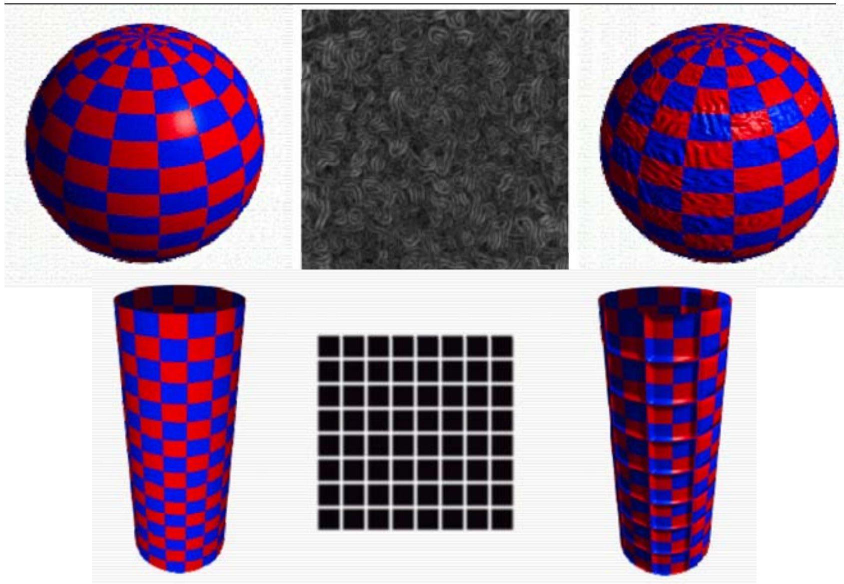
Bump mapping

- Displacement mapping adds geometrical details to surfaces

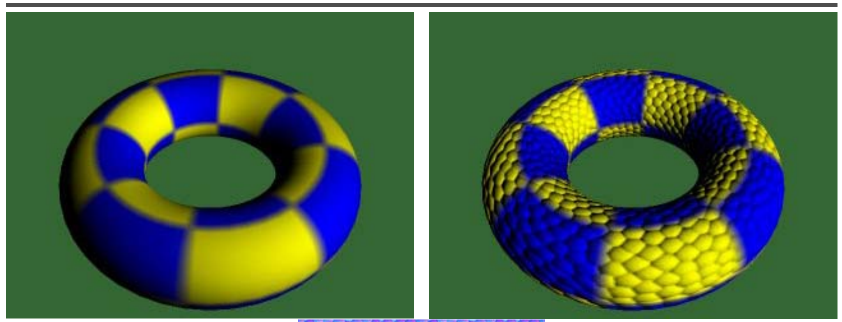


- Bump mapping: augments geometrical details to a surface without changing the surface itself
- It works well when the displacement is small

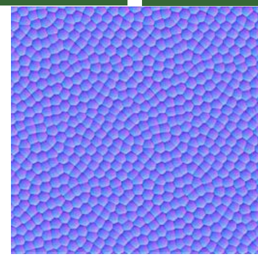
Bump mapping



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

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

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

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);
float displace = d->Evaluate(dgs);
// do similarly for v
```

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
if (Dot(dgGeom.nn, dgBump->nn) < 0.f)
  dgBump->nn *= -1.f;
```