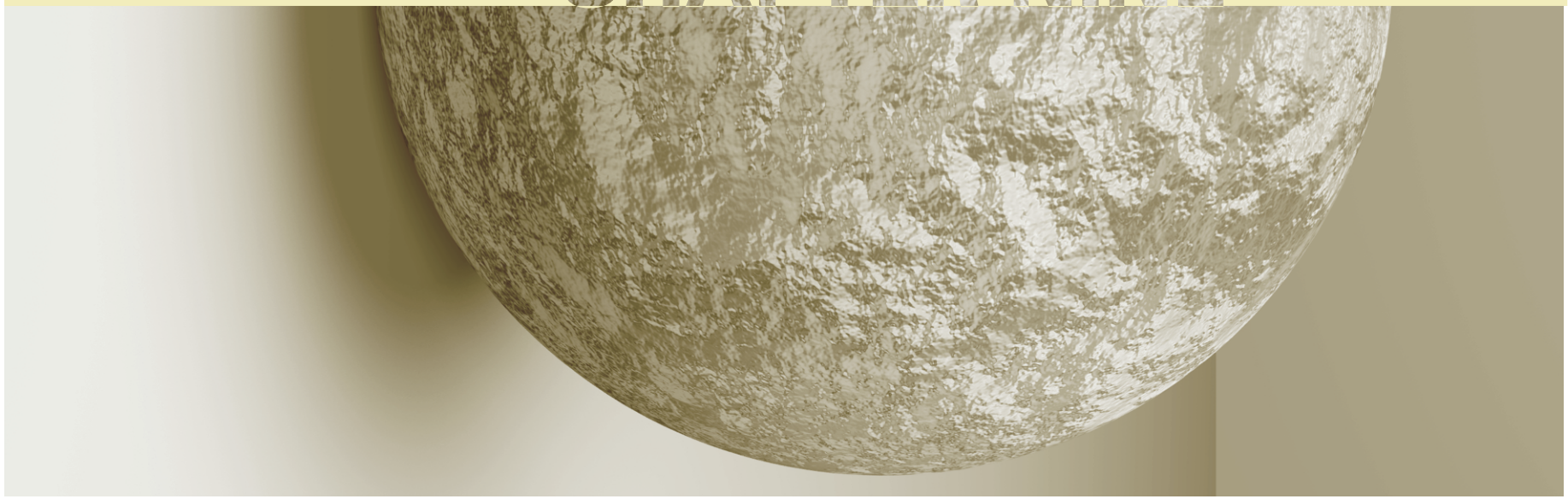
CHAPTER NINE



MATERIALS

The low-level BRDFs and BTDFs introduced in the previous chapter address only part of the problem of describing how a surface scatters light. Although they describe how light is scattered at a particular point on a surface, the renderer needs to determine *which* BRDFs and BTDFs to use at that point and how to set their parameters. In this chapter, we describe a procedural shading mechanism that determines the BRDFs and BTDFs to use at points on surfaces.

The basic idea is that a *surface shader* is bound to each primitive in the scene. The surface shader is represented by an instance of the `Material` interface class, which has a method that takes a point to be shaded and returns a `BSDF` object. The `BSDF` class holds a set of `BxDF`s that collectively describe scattering at a point. `Material`s, in turn, use instances of the `Texture` class (to be defined in the next chapter) to determine the material properties at particular points on surfaces. For example, an `ImageTexture` might be used to modulate the color of diffuse reflection across a surface. This is a somewhat different shading paradigm than many rendering systems use; it is common practice to combine the function of the surface shader and the lighting integrator (see Chapter 15) into a single module and have the shader return the color of reflected light at the point. However, by separating these two components and having the `Material` return a `BSDF`, `pbrt` is better able to handle a variety of light transport algorithms.

9.1 BSDFs

The `BSDF` class represents a collection of BRDFs and BTDFs. Grouping them in this manner allows the rest of the system to work with composite `BSDF`s directly, rather than having to consider all of the components they may have been built from. Equally important, the `BSDF` class hides some of the details of shading normals from the rest of the system. Shading normals, either from per-vertex normals in triangle meshes or from bump mapping, can substantially improve the visual richness of rendered scenes, but because they are

an *ad hoc* construct, they are tricky to incorporate into a physically based renderer. The issues that they introduce are handled in the BSDF implementation.

⟨*BSDF Declarations*⟩ +≡
```
class BSDF {
public:
    ⟨BSDF Public Methods 479⟩
    ⟨BSDF Public Data 479⟩
private:
    ⟨BSDF Private Methods 483⟩
    ⟨BSDF Private Data 479⟩
};
```

The BSDF constructor takes a DifferentialGeometry object that represents the shading differential geometry, the true surface normal ngeom, and, optionally, the index of refraction of the medium enclosed by the surface. It stores these values in member variables and constructs an orthonormal coordinate system with the shading normal as one of the axes; this will be useful for transforming directions to and from the BxDF coordinate system described in Section 8.1. Throughout this section, we will use the convention that $\mathbf{n}_s$ denotes the shading normal and $\mathbf{n}_g$ the geometric normal (Figure 9.1).

⟨*BSDF Method Definitions*⟩ ≡
```
BSDF::BSDF(const DifferentialGeometry &dg, const Normal &ngeom,
           float e)
    : dgShading(dg), eta(e) {
    ng = ngeom;
```

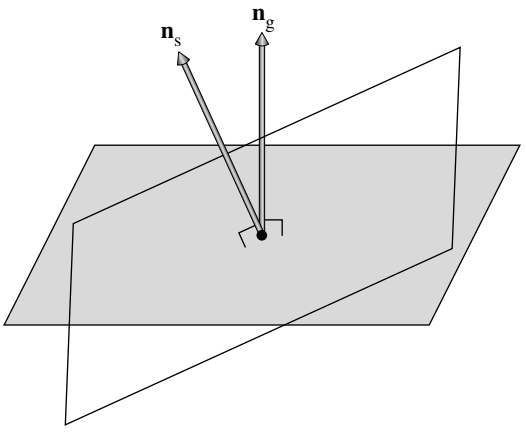


Figure 9.1: The geometric normal, $\mathbf{n}_g$, defined by the surface geometry, and the shading normal, $\mathbf{n}_s$, given by per-vertex normals and/or bump mapping, will generally define different hemispheres for integrating incident illumination to compute surface reflection. This inconsistency is important to handle carefully since it can otherwise lead to artifacts in images.

BSDF 478
BSDF::nBxDFs 479
BxDF 428
Cross() 62
DifferentialGeometry 102
Normal 65
Vector::Normalize() 63

```
    nn = dgShading.nn;
    sn = Normalize(dgShading.dpdu);
    tn = Cross(nn, sn);
    nBxDFs = 0;
}
```

⟨*BSDF Public Data*⟩ ≡ 478
```
const DifferentialGeometry dgShading;
const float eta;
```

⟨*BSDF Private Data*⟩ ≡ 478
```
Normal nn, ng;
Vector sn, tn;
```

The BSDF implementation stores only a fixed limited number of individual BxDF components. It could easily be extended to allocate more space if more components were given to it, although this isn't necessary for any of the Material implementations in pbrt thus far, and the current limit of eight is plenty for almost all practical applications.

⟨*BSDF Inline Method Definitions*⟩ ≡
```
inline void BSDF::Add(BxDF *b) {
    Assert(nBxDFs < MAX_BxDFS);
    bxdfs[nBxDFs++] = b;
}
```

⟨*BSDF Private Data*⟩ +≡ 478
```
int nBxDFs;
#define MAX_BxDFS 8
BxDF *bxdfs[MAX_BxDFS];
```

For other parts of the system that need additional information about the particular BRDFs and BTDFs present, two different methods return the number of BxDFs stored by the BSDF; the second only returns the number that matches a particular set of BxDFType flags. Note that we could use a single function with a default parameter value of BSDF_ALL, but the no-argument variant of BSDF::NumComponents() is called frequently, so we would like to avoid the overhead of checking the flags of the BxDFs individually in that case.

⟨*BSDF Public Methods*⟩ ≡ 478
```
int NumComponents() const { return nBxDFs; }
int NumComponents(BxDFType flags) const;
```

The BSDF also has methods that perform transformations to and from the local coordinate system expected by BxDFs. Recall that, in this coordinate system, the surface normal is along the z axis (0, 0, 1), the primary tangent is (1, 0, 0), and the secondary tangent is (0, 1, 0). The transformation of directions into "shading space" simplifies many of the BxDF implementations in Chapter 8. Given three orthonormal vectors $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{n}$ in world space, the matrix $\mathbf{M}$ that transforms vectors in world space to the local reflection space is

$$\mathbf{M} = \begin{pmatrix} \mathbf{s}_x & \mathbf{s}_y & \mathbf{s}_z \\ \mathbf{t}_x & \mathbf{t}_y & \mathbf{t}_z \\ \mathbf{n}_x & \mathbf{n}_y & \mathbf{n}_z \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{t} \\ \mathbf{n} \end{pmatrix}.$$

Assert() 1005
BSDF 478
BSDF::bxdfs 479
BSDF::nBxDFs 479
BSDF::NumComponents() 479
BSDF_ALL 428
BxDF 428
BxDFType 428
DifferentialGeometry 102
Material 483
Normal 65
Vector 57

To confirm this yourself, consider, for example, the value of $\mathbf{M}$ times the surface normal $\mathbf{n}$, $\mathbf{Mn} = (\mathbf{s} \cdot \mathbf{n}, \mathbf{t} \cdot \mathbf{n}, \mathbf{n} \cdot \mathbf{n})$. Since $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{n}$ are all orthonormal, the x and y components of $\mathbf{Mn}$ are zero. Since $\mathbf{n}$ is normalized, $\mathbf{n} \cdot \mathbf{n} = 1$. Thus, $\mathbf{Mn} = (0, 0, 1)$, as expected.

In this case, we don't need to compute the inverse transpose of $\mathbf{M}$ to transform normals (recall the discussion of transforming normals in Section 2.8.3). Because $\mathbf{M}$ is an orthogonal matrix (its rows and columns are mutually orthogonal), its inverse is equal to its transpose, so it is its own inverse transpose already.

⟨*BSDF Public Methods*⟩ +≡ 478

```
Vector WorldToLocal(const Vector &v) const {
    return Vector(Dot(v, sn), Dot(v, tn), Dot(v, nn));
}
```

The method that takes vectors back from local space to world space uses the transpose to invert $\mathbf{M}$ before doing the appropriate dot products.

⟨*BSDF Public Methods*⟩ +≡ 478

```
Vector LocalToWorld(const Vector &v) const {
    return Vector(sn.x * v.x + tn.x * v.y + nn.x * v.z,
                  sn.y * v.x + tn.y * v.y + nn.y * v.z,
                  sn.z * v.x + tn.z * v.y + nn.z * v.z);
}
```

Shading normals can cause a variety of undesirable artifacts in practice (Figure 9.2). Figure 9.2(a) shows a *light leak*: the geometric normal indicates that ω_i and ω_o lie on opposite sides of the surface, so if the surface is not transmissive, the light should have no contribution. However, if we directly evaluate the scattering equation, Equation (5.8), about the hemisphere centered around the shading normal, we will incorrectly incorporate the light from ω_i. This case demonstrates that $\mathbf{n}_s$ can't just be used as a direct replacement for $\mathbf{n}_g$ in rendering computations.

Figure 9.2(b) shows a similar tricky situation: the shading normal indicates that no light should be reflected to the viewer, since it is not in the same hemisphere as the illumination, while the geometric normal indicates that they are in the same hemisphere. Direct use of $\mathbf{n}_s$ would cause ugly black spots on the surface where this situation happens.

Fortunately, there is an elegant solution to these problems. When evaluating the BSDF, we use the geometric normal to decide if we should be evaluating reflection or transmission: if ω_i and ω_o lie in the same hemisphere with respect to $\mathbf{n}_g$, we evaluate the BRDFs, and otherwise we evaluate the BTDFs. In evaluating the scattering equation, however, the dot product of the normal and the incident direction is still taken with the shading normal, rather than the geometric normal.

BSDF::nn 479
BSDF::sn 479
BSDF::tn 479
BxDF 428
Dot() 60
Vector 57

Now it should be clear why `pbrt` requires `BxDF`s to evaluate their values without regard to whether ω_i and ω_o are in the same or different hemispheres. Thus, light leaks are avoided, since we will only evaluate the BTDFs for the situation in Figure 9.2(a), giving no reflection for a purely reflective surface. Similarly, black spots are avoided since we will evaluate the BRDFs for the situation in Figure 9.2(b), even though the shading normal would suggest that the directions are in different hemispheres.

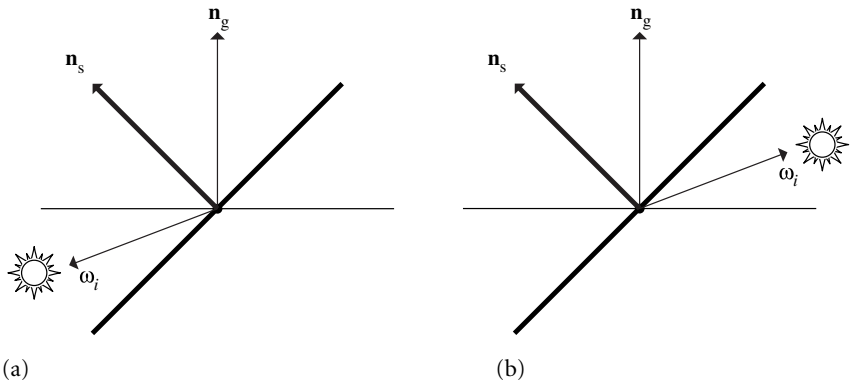


Figure 9.2: The Two Types of Errors That Result from Using Shading Normals. (a) A light leak: the geometric normal indicates that the light is on the back side of the surface, but the shading normal indicates the light is visible (assuming a reflective and not transmissive surface). (b) A dark spot: the geometric normal indicates that the surface is illuminated, but the shading normal indicates that the viewer is behind the lit side of the surface.

Given these conventions, the method that evaluates the BSDF for a given pair of directions is straightforward. It starts by transforming the world space direction vectors to local BSDF space and then determines whether it should use the BRDFs or the BTDFs. It then loops over the appropriate set and evaluates the sum of their contributions.

⟨*BSDF Method Definitions*⟩ +≡

```
Spectrum BSDF::f(const Vector &woW, const Vector &wiW,
                 BxDFType flags) const {
    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;
}
```

BSDF 478
BSDF::bxdfs 479
BSDF::nBxDFs 479
BSDF::WorldToLocal() 480
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDF 428
BxDF::f() 429
BxDF::MatchesFlags() 429
BxDF::rho() 430
BxDFType 428
Dot() 60
RNG 1003
Spectrum 263
Vector 57

pbrt also provides BSDF methods that return the summed reflectance values of their individual BxDFs. These methods just loop over the BxDFs and call the appropriate BxDF::rho() methods and therefore won't be shown here. A RNG must be passed to these methods so that they can generate random samples for BxDF implementations for use in Monte Carlo sampling algorithms if needed—recall the BxDF::rho() interface defined in Section 8.1.1.

⟨*BSDF Public Methods*⟩ +≡ 478
```
Spectrum rho(RNG &rng, BxDFType flags = BSDF_ALL,
             int sqrtSamples = 6) const;
Spectrum rho(const Vector &wo, RNG &rng, BxDFType flags = BSDF_ALL,
             int sqrtSamples = 6) const;
```

9.1.1 BSDF MEMORY MANAGEMENT

For each camera ray that intersects geometry in the scene, one or more BSDF objects will be created by the SurfaceIntegrator in the process of computing the reflected radiance from the intersection point. (Integrators that account for multiple interreflections of light will generally create a number of BSDFs along the way.) Each of these BSDFs in turn has a number of BxDFs stored inside it, as returned by the Materials at the intersection points. A naïve implementation of this process would use new and delete to dynamically allocate storage for both the BSDF as well as each of the BxDFs that it holds.

Unfortunately, such an approach is inefficient—too much time is spent in the dynamic memory management routines for a series of small memory allocations. Instead, the implementation here uses a specialized allocation scheme based on the MemoryArena class described in Section A.5.4 in Appendix A. MemoryArena allocates a large block of memory and responds to allocation requests via the MemoryArena::Alloc() call by returning successive sections of that block. It does not support freeing individual allocations, but instead frees all of them simultaneously when the MemoryArena::FreeAll() method is called. The result of this approach is that both allocation and freeing of memory are extremely efficient.

For the convenience of code that allocates BSDFs and BxDFs (e.g., the Materials in this chapter), there is a macro that hides some of the messiness of using the memory arena approach. Instead of using the new operator to allocate those objects like this:

```
BSDF *b = new BSDF;
BxDF *lam = new Lambertian(Spectrum(1.0));
```

code should instead be written with the BSDF_ALLOC() macro, like this:

```
BSDF *b = BSDF_ALLOC(arena, BSDF);
BxDF *lam = BSDF_ALLOC(arena, Lambertian)(Spectrum(1.0));
```

where arena is a MemoryArena. A MemoryArena object is generally passed down to the methods that need to allocate memory for BSDFs. For example, the SamplerRendererTask::Run() method allocates a MemoryArena for each rendering task and passes it to the integrators.

This macro calls the MemoryArena::Alloc() routine to allocate the appropriate amount of memory for the object and then uses the placement operator new to run the constructor for the object at the returned memory location.

⟨*BSDF Declarations*⟩ +≡
```
#define BSDF_ALLOC(arena, Type) new (arena.Alloc(sizeof(Type))) Type
```

The BSDF destructor is a private method, in order to ensure that it isn't inadvertently called, for example, due to an attempt to delete a BSDF. Making the destructor private

BSDF 478
BSDF_ALL 428
BSDF_ALLOC() 482
BxDF 428
BxDFType 428
Integrator 740
Material 483
MemoryArena 1015
MemoryArena::Alloc() 1016
MemoryArena::FreeAll() 1017
RNG 1003
SamplerRendererTask::Run() 30
Spectrum 263
SurfaceIntegrator 740
Vector 57

ensures a compile time error if it is called. Trying to `delete` memory allocated by the `MemoryArena` could lead to errors or crashes, since a pointer to the middle of memory managed by the `MemoryArena` would be passed to the system's dynamic memory freeing routine. Thus, a further implication of the allocation scheme here is that `BSDF` and `BxDF` destructors are never executed. This isn't a problem for the ones currently implemented in the system.

⟨*BSDF Private Methods*⟩ ≡ 478
```
~BSDF() { }
```

9.2 MATERIAL INTERFACE AND IMPLEMENTATIONS

The abstract `Material` class defines two methods that material implementations must provide, `Material::GetBSDF()` and `Material::GetBSSRDF()`. Implementations of these methods are responsible for determining the reflective properties at the given point on the surface and returning an instance of the `BSDF` class that describes them. The `Material` class is defined in the files `core/material.h` and `core/material.cpp`.

⟨*Material Declarations*⟩ ≡
```
class Material : public ReferenceCounted {
public:
    ⟨Material Interface 483⟩
};
```

The `Material::GetBSDF()` method is given two differential geometry objects. The first, `dgGeom`, represents the actual differential geometry at the ray intersection point, and the second, `dgShading`, represents possibly perturbed shading geometry, such as that from per-vertex normals in a triangle mesh. The material implementation may further perturb the shading geometry with bump mapping in the `GetBSDF()` method; the returned `BSDF` holds information about the final shading geometry at the point as well as the BRDF and BTDF components for the point.

⟨*Material Interface*⟩ ≡ 483
```
virtual BSDF *GetBSDF(const DifferentialGeometry &dgGeom,
                      const DifferentialGeometry &dgShading,
                      MemoryArena &arena) const = 0;
```

For materials that represent translucent materials that exhibit subsurface scattering, the `GetBSSRDF()` method returns a `BSSRDF` object that represents the object's subsurface scattering properties. The `BSSRDF` is defined later, in Section 11.6, after the foundations of volumetric scattering have been introduced. Because most materials don't have meaningful amounts of subsurface scattering, this method has a default implementation that just returns `NULL`.

BSDF 478
BSSRDF 598
DifferentialGeometry 102
Material 483
Material::GetBSDF() 483
Material::GetBSSRDF() 484
MemoryArena 1015
ReferenceCounted 1010

⟨*Material Interface*⟩ +≡ 483
```
virtual BSSRDF *GetBSSRDF(const DifferentialGeometry &dgGeom,
                          const DifferentialGeometry &dgShading,
                          MemoryArena &arena) const {
    return NULL;
}
```

Since the usual interface to the hit point used by `Integrators` is through an instance of the `Intersection` class, we will add two convenience methods to `Intersection`, `GetBSDF()` and `GetBSSRDF()`, that respectively return the `BSDF` or `BSSRDF` at the hit point. These methods call the `Differentialgeometry::ComputeDifferentials()` method to compute information about the projected size of the surface area around the intersection on the image plane for use in texture antialiasing and then forward the request to the `Primitive`, which in turn will call the corresponding `GetBSDF()` or `GetBSSRDF()` method of its `Material`. (See, for example, the `GeometricPrimitive::GetBSDF()` implementation.)

⟨*Intersection Method Definitions*⟩ ≡
```
BSDF *Intersection::GetBSDF(const RayDifferential &ray,
                            MemoryArena &arena) const {
    dg.ComputeDifferentials(ray);
    BSDF *bsdf = primitive->GetBSDF(dg, ObjectToWorld, arena);
    return bsdf;
}
```

The implementation of the `GetBSSRDF()` method is essentially the same, other than the fact that it calls the `Primitive::GetBSSRDF()` method.

⟨*Intersection Public Methods*⟩ ≡ 186
```
BSSRDF *GetBSSRDF(const RayDifferential &ray, MemoryArena &arena) const;
```

9.2.1 MATTEMATERIAL

The `MatteMaterial` material is defined in `materials/matte.h` and `materials/matte.cpp`. It is the simplest material in `pbrt` and describes a purely diffuse surface. It is parameterized by a spectral diffuse reflection value, `Kd`, and a scalar roughness value, `sigma`. If `sigma` has the value zero at the point on a surface, `MatteMaterial` returns a `Lambertian` BRDF; otherwise, the `OrenNayar` model is used. Like all of the other `Material` implementations in this chapter, it also takes an optional scalar texture that defines an offset function over the surface. If non-`NULL`, this texture is used to compute a shading normal at each point based on the function it defines. Figure 8.14 in the previous chapter shows the `MatteMaterial` material with the Killeroo model.

⟨*MatteMaterial Declarations*⟩ ≡
```
class MatteMaterial : public Material {
public:
    ⟨MatteMaterial Public Methods 485⟩
private:
    ⟨MatteMaterial Private Data 485⟩
};
```

BSDF 478
BSSRDF 598
DifferentialGeometry 102
DifferentialGeometry::ComputeDifferentials() 505
GeometricPrimitive::GetBSDF() 189
Integrator 740
Intersection 186
Intersection::dg 186
Intersection::ObjectToWorld 186
Intersection::primitive 186
Lambertian 447
Material 483
MatteMaterial 484
MemoryArena 1015
OrenNayar 449
Primitive 185
Primitive::GetBSDF() 187
Primitive::GetBSSRDF() 187
RayDifferential 69

⟨*MatteMaterial Public Methods*⟩ ≡ 484
```
MatteMaterial(Reference<Texture<Spectrum> > kd,
              Reference<Texture<float> > sig,
              Reference<Texture<float> > bump)
    : Kd(kd), sigma(sig), bumpMap(bump) {
}
```

⟨*MatteMaterial Private Data*⟩ ≡ 484
```
Reference<Texture<Spectrum> > Kd;
Reference<Texture<float> > sigma, bumpMap;
```

The `GetBSDF()` method puts the pieces together, determining the bump map's effect on the shading geometry, evaluating the textures, and allocating and returning the `BSDF`.

⟨*MatteMaterial Method Definitions*⟩ ≡
```
BSDF *MatteMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                             const DifferentialGeometry &dgShading,
                             MemoryArena &arena) const {
    ⟨Allocate BSDF, possibly doing bump mapping with bumpMap 485⟩
    ⟨Evaluate textures for MatteMaterial material and allocate BRDF 486⟩
    return bsdf;
}
```

If a bump map was provided to the `MatteMaterial` constructor, the `Material::Bump()` method is called to calculate the shading normal at the point. This method will be defined in the next section.

⟨*Allocate* `BSDF`, *possibly doing bump mapping with* `bumpMap`⟩ ≡ 485, 487, 489
```
DifferentialGeometry dgs;
if (bumpMap)
    Bump(bumpMap, dgGeom, dgShading, &dgs);
else
    dgs = dgShading;
BSDF *bsdf = BSDF_ALLOC(arena, BSDF)(dgs, dgGeom.nn);
```

Next, the `Textures` that give the values of the diffuse reflection coefficient and the roughness are evaluated; these may return constant values, look up values from image maps, or do complex procedural shading calculations to compute these values (the texture evaluation process is the subject of Chapter 10). Given these values, all that needs to be done is to allocate a `BSDF` and the appropriate BRDF component using the `BSDF` memory allocation macros and return the result. Because `Textures` may return negative values or values otherwise outside of the expected range, these values are clamped before they are passed to the BRDF constructor.

BSDF 478
DifferentialGeometry 102
Material::Bump() 495
MatteMaterial 484
MatteMaterial::bumpMap 485
MatteMaterial::Kd 485
MatteMaterial::sigma 485
MemoryArena 1015
Reference 1011
Spectrum 263
Texture 519

⟨*Evaluate textures for* `MatteMaterial` *material and allocate BRDF*⟩ ≡ 485

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

9.2.2 PLASTICMATERIAL

Plastic can be modeled as a mixture of a diffuse and glossy scattering function with parameters controlling the particular colors and specular highlight size. The parameters to `PlasticMaterial` are two reflectivities, `Kd` and `Ks`, which respectively control the amounts of diffuse reflection and glossy specular reflection. Next is a roughness parameter (which ranges from zero to one) that determines the size of the specular highlight; the higher the roughness value, the larger the highlight. Figure 9.3 shows a plastic Killeroo. `PlasticMaterial` is defined in `materials/plastic.h` and `materials/plastic.cpp`.

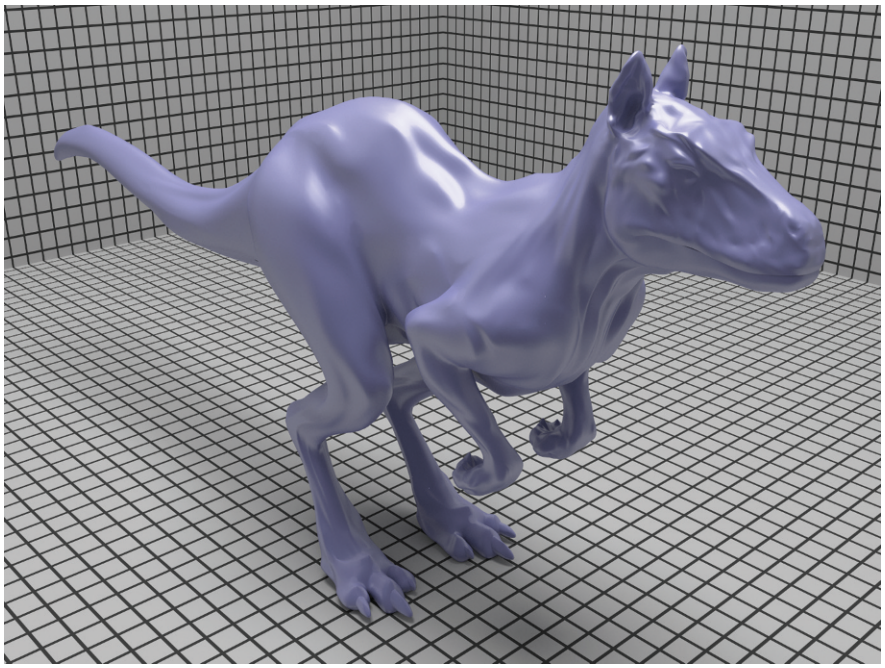


Figure 9.3: Killeroo Rendered with a Plastic Material. Note the combination of diffuse and glossy specular reflection. *(Killeroo courtesy of headus/Rezard.)*

BSDF::Add() 479
BSDF_ALLOC() 482
Lambertian 447
MatteMaterial::Kd 485
MatteMaterial::sigma 485
OrenNayar 449
PlasticMaterial 487
Spectrum 263
Spectrum::Clamp() 265
Texture::Evaluate() 520

⟨*PlasticMaterial Declarations*⟩ ≡
```
class PlasticMaterial : public Material {
public:
    ⟨PlasticMaterial Public Methods 487⟩
private:
    ⟨PlasticMaterial Private Data 487⟩
};
```

⟨*PlasticMaterial Public Methods*⟩ ≡ 487
```
PlasticMaterial(Reference<Texture<Spectrum> > kd,
                Reference<Texture<Spectrum> > ks,
                Reference<Texture<float> > rough,
                Reference<Texture<float> > bump)
    : Kd(kd), Ks(ks), roughness(rough), bumpMap(bump) {
}
```

⟨*PlasticMaterial Private Data*⟩ ≡ 487
```
Reference<Texture<Spectrum> > Kd, Ks;
Reference<Texture<float> > roughness, bumpMap;
```

The `PlasticMaterial::GetBSDF()` method follows the same basic structure as `MatteMaterial::GetBSDF()`: it evaluates textures, calls the bump-mapping function, allocates `BxDF`s, and initializes the `BSDF`.

⟨*PlasticMaterial Method Definitions*⟩ ≡
```
BSDF *PlasticMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                               const DifferentialGeometry &dgShading,
                               MemoryArena &arena) const {
    ⟨Allocate BSDF, possibly doing bump mapping with bumpMap 485⟩
    Spectrum kd = Kd->Evaluate(dgs).Clamp();
    BxDF *diff = BSDF_ALLOC(arena, Lambertian)(kd);
    Fresnel *fresnel = BSDF_ALLOC(arena, FresnelDielectric)(1.5f, 1.f);
    Spectrum ks = Ks->Evaluate(dgs).Clamp();
    float rough = roughness->Evaluate(dgs);
    BxDF *spec = BSDF_ALLOC(arena, Microfacet)
                 (ks, fresnel, BSDF_ALLOC(arena, Blinn)(1.f / rough));
    bsdf->Add(diff);
    bsdf->Add(spec);
    return bsdf;
}
```

Blinn 456
BSDF 478
BSDF::Add() 479
BxDF 428
DifferentialGeometry 102
Fresnel 436
FresnelDielectric 437
Lambertian 447
Material 483
MatteMaterial::GetBSDF() 485
MemoryArena 1015
Microfacet 454
MixMaterial 488
PlasticMaterial 487
PlasticMaterial::bumpMap 487
PlasticMaterial::GetBSDF() 487
PlasticMaterial::Kd 487
PlasticMaterial::Ks 487
PlasticMaterial::roughness 487
Reference 1011
Spectrum 263
Spectrum::Clamp() 265
Texture 519
Texture::Evaluate() 520

9.2.3 MIX MATERIAL

It's useful to be able to combine two `Materials` with a varying weight. The `MixMaterial` takes two other `Materials` and a `Spectrum`-valued texture; it uses the `Spectrum` returned by the texture to blend between the two materials at the point being shaded. It is defined in the files `materials/mixmat.h` and `materials/mixmat.cpp`.

⟨*MixMaterial Declarations*⟩ ≡
```
class MixMaterial : public Material {
public:
    ⟨MixMaterial Public Methods 488⟩
private:
    ⟨MixMaterial Private Data 488⟩
};
```

⟨*MixMaterial Public Methods*⟩ ≡ 488
```
MixMaterial(Reference<Material> mat1, Reference<Material> mat2,
            Reference<Texture<Spectrum> > sc)
    : m1(mat1), m2(mat2), scale(sc) {
}
```

⟨*MixMaterial Private Data*⟩ ≡ 488
```
Reference<Material> m1, m2;
Reference<Texture<Spectrum> > scale;
```

`MixMaterial::GetBSDF()` gets `BSDF *`s from the two materials. It scales `BxDF`s in the `BSDF` from the first material, `b1`, using the `ScaledBxDF` adapter class, and then scales the `BxDF`s from the second `BSDF`, adding them to `b1`. It may appear that there's a lurking memory leak in this code, in that the `BxDF *`s in `b1->bxdfs` are clobbered by newly allocated `ScaledBxDF`s. However, recall that those `BxDF`s, like the new ones here, were allocated through a `MemoryArena` and will thus be freed when the `MemoryArena` frees its entire block of memory.

⟨*MixMaterial Method Definitions*⟩ ≡
```
BSDF *MixMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                           const DifferentialGeometry &dgShading,
                           MemoryArena &arena) const {
    BSDF *b1 = m1->GetBSDF(dgGeom, dgShading, arena);
    BSDF *b2 = m2->GetBSDF(dgGeom, dgShading, arena);
    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(arena, ScaledBxDF)(b1->bxdfs[i], s1);
    for (int i = 0; i < n2; ++i)
        b1->Add(BSDF_ALLOC(arena, ScaledBxDF)(b2->bxdfs[i], s2));
    return b1;
}
```

The implementation of `MixMaterial::GetBSDF()` needs direct access to the `bxdfs` member variables of the `BSDF` class. Because this is the only class that needs this access, we've just made `MixMaterial` a friend of `BSDF` rather than adding a number of accessor and setting methods.

⟨*BSDF Private Data*⟩ +≡ 478
```
friend class MixMaterial;
```

BSDF 478
BSDF::bxdfs 479
BSDF::NumComponents() 479
BxDF 428
DifferentialGeometry 102
Material 483
Material::GetBSDF() 483
MemoryArena 1015
MixMaterial 488
MixMaterial::m1 488
MixMaterial::m2 488
MixMaterial::scale 488
Reference 1011
ScaledBxDF 431
Spectrum 263
Spectrum::Clamp() 265
Texture 519
Texture::Evaluate() 520

9.2.4 MEASURED MATERIAL

The MeasuredMaterial class supports two forms of measured isotropic BRDF data: irregular and regular, corresponding to the BxDFs implemented in Sections 8.6.1 and 8.6.2. It is defined in the files materials/measured.h and materials/measured.cpp.

⟨*MeasuredMaterial Declarations*⟩ ≡
```
class MeasuredMaterial : public Material {
public:
    ⟨MeasuredMaterial Public Methods⟩
private:
    ⟨MeasuredMaterial Private Data 489⟩
};
```

For irregularly sampled BRDFs, the MeasuredMaterial constructor creates a kd-tree of the BRDF samples, using the BRDFRemap() function to map 4D pairs of directions to the 3D space that the IrregIsotropicBRDF uses for interpolation. For regularly sampled data, the table and its sampling resolution in the θ_h, θ_d, and ϕ_d dimensions are stored in corresponding member variables. (Only one of thetaPhiData or regularHalfangleData will be non-NULL.)

We won't include the code that reads this data from files here. The file formats for these two types of data are described in comments in the core/materials.cpp file.

⟨*MeasuredMaterial Private Data*⟩ ≡ 489
```
KdTree<IrregIsotropicBRDFSample> *thetaPhiData;
float *regularHalfangleData;
uint32_t nThetaH, nThetaD, nPhiD;
Reference<Texture<float> > bumpMap;
```

Once the data is in memory, the GetBSDF() method's task is straightforward. After the usual bump-mapping computation, it just has to allocate the appropriate BxDF implementation based on whether this instance of the MeasuredMaterial has regularly or irregularly sampled data.

⟨*MeasuredMaterial Method Definitions*⟩ ≡
```
BSDF *MeasuredMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                                const DifferentialGeometry &dgShading,
                                MemoryArena &arena) const {
    ⟨Allocate BSDF, possibly doing bump mapping with bumpMap 485⟩
    if (regularHalfangleData)
        bsdf->Add(BSDF_ALLOC(arena, RegularHalfangleBRDF)
            (regularHalfangleData, nThetaH, nThetaD, nPhiD));
    else if (thetaPhiData)
        bsdf->Add(BSDF_ALLOC(arena, IrregIsotropicBRDF)(thetaPhiData));
    return bsdf;
}
```

BRDFRemap() 465
BSDF 478
BSDF::Add() 479
BxDF 428
DifferentialGeometry 102
IrregIsotropicBRDF 464
IrregIsotropicBRDFSample 465
KdTree 1029
Material 483
MeasuredMaterial::nPhiD 489
MeasuredMaterial::nThetaD 489
MeasuredMaterial::nThetaH 489
MeasuredMaterial::regularHalfangleData 489
MeasuredMaterial::thetaPhiData 489
MemoryArena 1015
Reference 1011
RegularHalfangleBRDF 467
Texture 519

9.2.5 ADDITIONAL MATERIALS

Beyond these basic materials, there are eight more `Material` implementations available in `pbrt`, in the `materials/` directory. We will not show all of their implementations here, since they are all just variations on the basic themes introduced in the material implementations above. All take `Textures` that define scattering parameters, these textures are evaluated in their `GetBSDF()` methods, and appropriate `BxDFs` are created and returned in a `BSDF`. See the documentation on `pbrt`'s file format in the file `docs/fileformat.pdf` for a summary of the parameters that these materials take.

These materials include:

- `GlassMaterial`: Specular reflection and transmission, weighted by Fresnel terms for accurate angular-dependent variation.
- `MetalMaterial`: Metal, using the Fresnel equations for conductors. See the files in the directory `scenes/spds/metals/` for measured spectral data for the indices of refraction η and absorption coefficients k for a variety of metals.
- `MirrorMaterial`: A simple mirror, modeled with perfect specular reflection.
- `SubstrateMaterial`: A layered model that varies between glossy specular and diffuse reflection depending on the viewing angle (based on the `FresnelBlend` BRDF).
- `SubsurfaceMaterial` and `KdSubsurfaceMaterial`: Materials that return `BSSRDFs` that describe materials that exhibit subsurface scattering.
- `TranslucentMaterial`: A material that describes diffuse and glossy specular reflection and transmission through the surface.
- `UberMaterial`: A "kitchen sink" material representing the union of many of the preceding materials. This is a highly parameterized material that is particularly useful when converting scenes from other file formats into `pbrt`'s.

Figure 8.9 in the previous chapter shows the Killeroo model rendered with the glass material, and Figure 9.4 shows it with the `MetalMaterial`. Figure 9.5 demonstrates the `KdSubsurfaceMaterial`.

9.3 BUMP MAPPING

All of the `Materials` defined in the previous section take an optional `float` texture map that defines a displacement at each point on the surface: each point p has a displaced point p′ associated with it, defined by $\mathrm{p}' = \mathrm{p} + d(\mathrm{p})\mathbf{n}(\mathrm{p})$, where $d(\mathrm{p})$ is the offset returned by the displacement texture at p and $\mathbf{n}(\mathrm{p})$ is the surface normal at p (Figure 9.6). We would like to use this texture to compute shading normals so that the surface appears as if it actually had been offset by the displacement function, without modifying its geometry. This process is called *bump mapping*. For relatively small displacement functions, the visual effect of bump mapping can be quite convincing. This idea and the specific technique to compute these shading normals in a way that gives a plausible appearance of the actual displaced surface were developed by Blinn (1978).

Figure 9.7 shows the effect of applying bump mapping defined by an image map of a grid of lines to a sphere. A more complex example is shown in Figure 9.8, which shows a scene rendered with and without bump mapping. There, the bump map gives the appearance of a substantial amount of detail in the walls and floors that isn't actually present in

BSDF 478
BxDF 428
Material 483
Texture 519

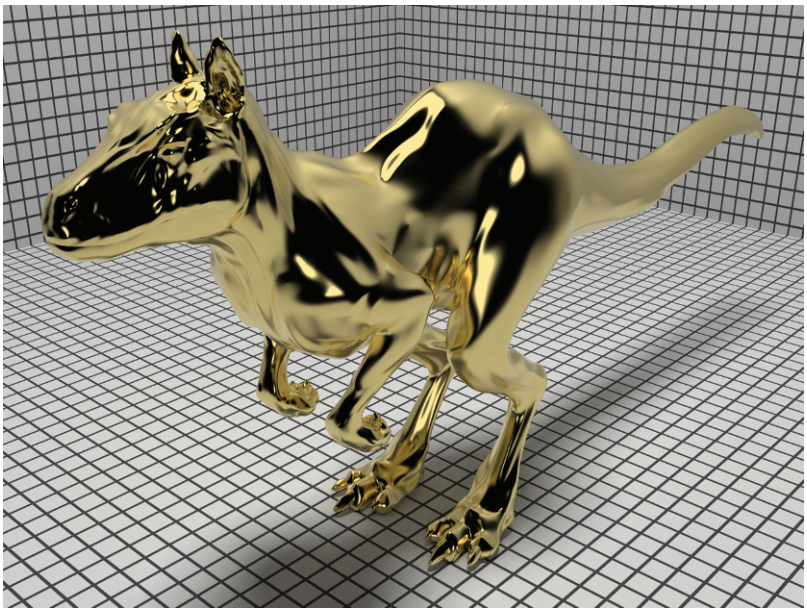


Figure 9.4: Killeroo rendered with the `MetalMaterial`, based on realistic measured gold scattering data. *(Killeroo courtesy of headus/Rezard.)*

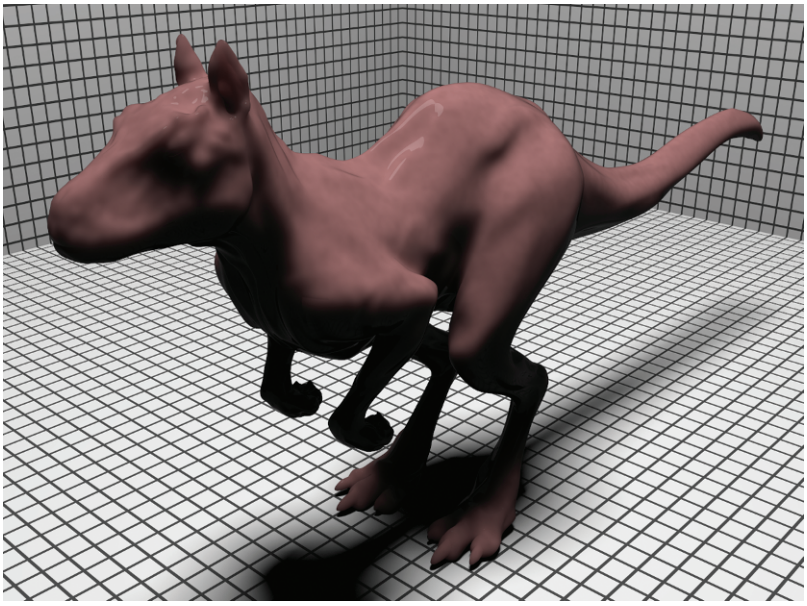


Figure 9.5: Killeroo rendered with the `KdSubsurfaceMaterial`, which models subsurface scattering (in conjunction with the subsurface scattering light transport techniques from Section 16.5). *(Killeroo courtesy of headus/Rezard.)*

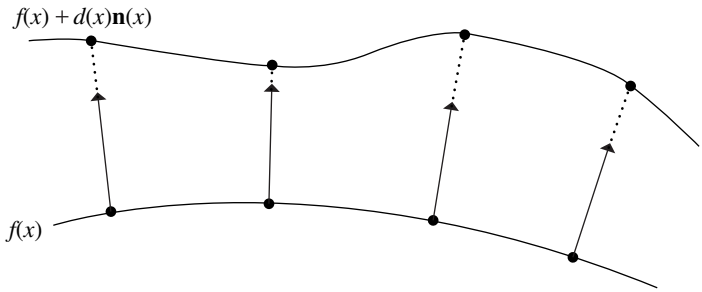


Figure 9.6: A displacement function associated with a material defines a new surface based on the old one, offset by the displacement amount along the normal at each point. `pbrt` doesn't compute a geometric representation of this displaced surface, but instead uses it to compute shading normals for bump mapping.

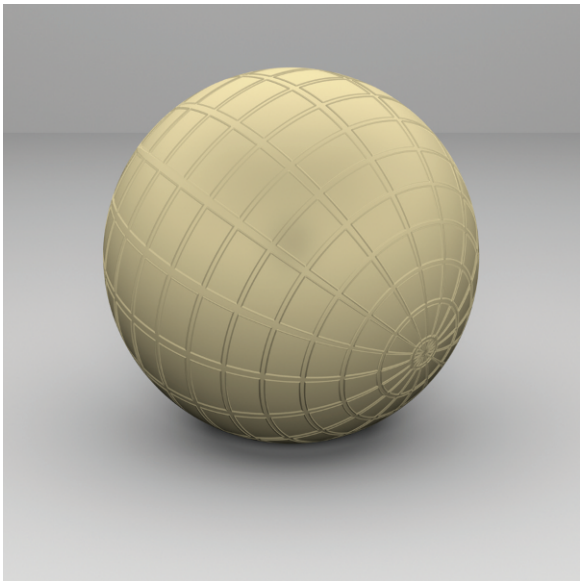


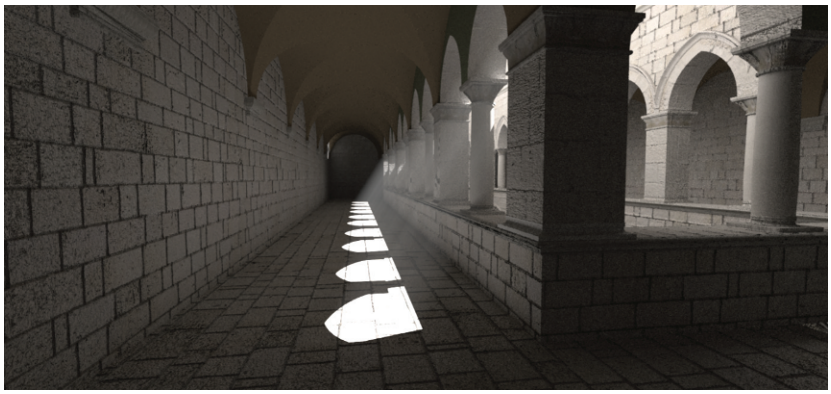
Figure 9.7: Using bump mapping to compute the shading normals for a sphere gives it the appearance of having much more geometric detail than is actually present.

the geometric model. Figure 9.9 shows one of the image maps used to define the bump function in Figure 9.8.

The `Material::Bump()` method is a utility routine for use by `Material` implementations. It is responsible for computing the effect of bump mapping at the point being shaded given a particular displacement `Texture`. So that future `Material` implementations aren't required to support bump mapping with this particular mechanism (or at all), we've

Material 483
Material::Bump() 495
Texture 519

(a)



(b)

Figure 9.8: The Sponza atrium model, rendered (a) without bump mapping and (b) with bump mapping. Bump mapping substantially increases the apparent geometric complexity of the model, without the increased rendering time and memory use that would result from a geometric representation with the equivalent amount of small-scale detail.

Figure 9.9: One of the image maps used as a bump map for the Sponza atrium rendering in Figure 9.8.

placed this method outside of the hard-coded material evaluation pipeline and left it as a function that particular material implementations can optionally call.

The implementation of `Material::Bump()` is based on finding an approximation to the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ of the displaced surface and using them in place of the surface's actual partial derivatives to compute the shading normal. (Recall that the surface normal is given by the cross product of these vectors, $\mathbf{n} = \partial \mathrm{p}/\partial u \times \partial \mathrm{p}/\partial v$.) Assume that the original surface is defined by a parametric function $p(u, v)$, and the bump offset function is a scalar function $d(u, v)$. Then the displaced surface is given by

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

where $\mathbf{n}(u, v)$ is the surface normal at (u, v).

The partial derivatives of this function can be found using the chain rule. For example, the partial derivative in u is

$$\frac{\partial p'}{\partial u} = \frac{\partial p(u, v)}{\partial u} + \frac{\partial d(u, v)}{\partial u}\mathbf{n}(u, v) + d(u, v)\frac{\partial \mathbf{n}(u, v)}{\partial u}. \quad [9.1]$$

We already have computed the value of $\partial p(u, v)/\partial u$; it's $\partial \mathrm{p}/\partial u$ and is available in the `DifferentialGeometry` structure, which also stores the surface normal $\mathbf{n}(u, v)$ and the partial derivative $\partial \mathbf{n}(u, v)/\partial u = \partial \mathbf{n}/\partial u$. The displacement function $d(u, v)$ can be evaluated as needed, which leaves $\partial d(u, v)/\partial u$ as the only unknown term.

There are two possible approaches to finding the values of $\partial d(u, v)/\partial u$ and $\partial d(u, v)/\partial v$. One option would be to augment the `Texture` interface with a method to compute partial derivatives of the underlying texture function. For example, for image map textures mapped to the surface directly using its (u, v) parameterization, these partial derivatives can be computed by subtracting adjacent texels in the u and v directions. However, this approach is difficult to extend to complex procedural textures like some of the ones defined in Chapter 10. Therefore, `pbrt` directly computes these values with forward differencing in the `Material::Bump()` method, without modifying the `Texture` interface.

Recall the definition of the partial derivative:

$$\frac{\partial d(u, v)}{\partial u} = \lim_{\Delta_u \to 0} \frac{d(u + \Delta_u, v) - d(u, v)}{\Delta_u}.$$

Forward differencing approximates the value using a finite value of Δ_u and evaluating $d(u, v)$ at two positions. Thus, the final expression for $\partial p'/\partial u$ is the following (for simplicity, we have dropped the explicit dependence on (u, v) for some of the terms):

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

Interestingly enough, most bump-mapping implementations ignore the final term under the assumption that $d(u, v)$ is expected to be relatively small. (Since bump mapping is mostly useful for approximating small perturbations, this is a reasonable assumption.) The fact that many renderers do not compute the values $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ may also have something to do with this simplification. An implication of ignoring the last term is that the magnitude of the displacement function then does not affect the bump-mapped partial derivatives; adding a constant value to it globally doesn't affect the final result,

`DifferentialGeometry` 102
`Material::Bump()` 495
`Texture` 519

since only differences of the bump function affect it. pbrt computes all three terms since it has $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ readily available, although in practice this final term rarely makes a visually noticeable difference.

One important detail in the definition of this function is that the d parameter is declared to be of type const Reference<Texture<float> > &, rather than, for example, Reference<Texture<float> >. This is very important for performance, but for a subtle reason. If it was not a C++ reference type, then the Reference type would need to increment the Texture's reference count for the value passed on the stack, and the reference count would need to be decremented when the method returned. This isn't a problem for serial code, but with multiple threads of execution leads to a situation where multiple processing cores are modifying the same memory location whenever different rendering tasks have the same displacement texture passed to this method. This in turn leads to the expensive "read for ownership" operation described in Section A.9.1.

⟨*Material Method Definitions*⟩ ≡
```
void Material::Bump(const Reference<Texture<float> > &d,
                    const DifferentialGeometry &dgGeom,
                    const DifferentialGeometry &dgs,
                    DifferentialGeometry *dgBump) {
    ⟨Compute offset positions and evaluate displacement texture 495⟩
    ⟨Compute bump-mapped differential geometry 496⟩
    ⟨Orient shading normal to match geometric normal 496⟩
}
```

⟨*Compute offset positions and evaluate displacement texture*⟩ ≡ 495
```
DifferentialGeometry dgEval = dgs;
⟨Shift dgEval du in the u direction 495⟩
float uDisplace = d->Evaluate(dgEval);
⟨Shift dgEval dv in the v direction 496⟩
float vDisplace = d->Evaluate(dgEval);
float displace = d->Evaluate(dgs);
```

One remaining issue is how to choose the offsets Δ_u and Δ_v for the finite differencing computations. They should be small enough that fine changes in $d(u, v)$ are captured but large enough so that available floating-point precision is sufficient to give a good result. Here, we will choose Δ_u and Δ_v values that lead to an offset that is about half the image space pixel sample spacing and use them to update the appropriate member variables in the DifferentialGeometry to reflect a shift to the offset position. (See Section 10.1.1 for an explanation of how the image space distances are computed.)

⟨*Shift* dgEval du *in the u direction*⟩ ≡ 495
```
float du = .5f * (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);
```

DifferentialGeometry 102
DifferentialGeometry::dndu 102
DifferentialGeometry::dpdu 102
DifferentialGeometry::dudx 505
DifferentialGeometry::dudy 505
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DifferentialGeometry::u 102
Material 483
Normal 65
Normal::Normalize() 65
Reference 1011
Texture 519
Texture::Evaluate() 520

⟨*Shift* dgEval dv *in the v direction*⟩ ≡ 495
```
float dv = .5f * (fabsf(dgs.dvdx) + fabsf(dgs.dvdy));
if (dv == 0.f) dv = .01f;
dgEval.p = dgs.p + dv * dgs.dpdv;
dgEval.u = dgs.u;
dgEval.v = dgs.v + dv;
dgEval.nn = Normalize((Normal)Cross(dgs.dpdu, dgs.dpdv) +
                      dv * dgs.dndv);
```

Given the new positions and the displacement texture's values at them, the partial derivatives can be computed directly using Equation (9.1):

⟨*Compute bump-mapped differential geometry*⟩ ≡ 495
```
*dgBump = dgs;
dgBump->dpdu = dgs.dpdu + (uDisplace - displace) / du * Vector(dgs.nn) +
               displace * Vector(dgs.dndu);
dgBump->dpdv = dgs.dpdv + (vDisplace - displace) / dv * Vector(dgs.nn) +
               displace * Vector(dgs.dndv);
dgBump->nn = Normal(Normalize(Cross(dgBump->dpdu, dgBump->dpdv)));
if (dgs.shape->ReverseOrientation ^ dgs.shape->TransformSwapsHandedness)
    dgBump->nn *= -1.f;
```

Finally, this method flips the shading coordinate frame if needed, so that the shading normal lies in the hemisphere around the geometric normal. Since the shading normal represents a relatively small perturbation of the geometric normal, the two of them should always be in the same hemisphere.

⟨*Orient shading normal to match geometric normal*⟩ ≡ 495
```
dgBump->nn = Faceforward(dgBump->nn, dgGeom.nn);
```

FURTHER READING

Blinn (1978) invented the bump-mapping technique. Kajiya (1985) generalized the idea of bump mapping the normal to *frame mapping*, which also perturbs the surface's primary tangent vector and is useful for controlling the appearance of anisotropic reflection models. Mikkelsen's thesis (2008) carefully investigates a number of the assumptions underlying bump mapping, proposes generalizations, and also addresses a number of subtleties with respect to its application to real-time rendering.

Snyder and Barr (1987) noted the light leak problem from per-vertex shading normals and proposed a number of work-arounds. The method we have used in this chapter is from Section 5.3 of Veach's thesis (Veach 1997); it is a more robust solution than those of Snyder and Barr.

Shading normals introduce a number of subtle problems for physically based light transport algorithms that we have not addressed here. For example, they can easily lead to surfaces that reflect more energy than was incident upon them, which can wreak havoc with light transport algorithms that are designed under the assumption of energy conservation. Veach (1996) investigated this issue in depth and developed a number of solutions.

Cross() 62
DifferentialGeometry::dndv 102
DifferentialGeometry::dpdu 102
DifferentialGeometry::dpdv 102
DifferentialGeometry::dvdx 505
DifferentialGeometry::dvdy 505
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Faceforward() 66
Normal 65
Normal::Normalize() 65
Shape::ReverseOrientation 108
Shape::TransformSwapsHandedness 108
Vector 57
Vector::Normalize() 63

One visual shortcoming of bump mapping is that it doesn't naturally account for self-shadowing, where bumps cast shadows on the surface and prevent light from reaching nearby points. These shadows can have a significant impact on the appearance of rough surfaces. Max (1988) developed the *horizon mapping* technique, which performs a preprocess on bump maps stored in image maps to compute a term to account for this effect. This approach isn't directly applicable to procedural textures, however. Dana et al. (1999) have measured spatially varying reflection properties from real-world surfaces, including these self-shadowing effects; they convincingly demonstrate this effect's importance for accurate image synthesis.

Another difficult issue related to bump mapping is that antialiasing bump maps that have higher-frequency detail than can be represented in the image is quite difficult. In particular, it is not enough to remove high-frequency detail from the bump map function, but in general the BSDF needs to be modified to account for this detail. Fournier (1992) applied *normal distribution functions* to this problem, where the surface normal was generalized to represent a distribution of normal directions. Becker and Max (1993) developed algorithms for blending between bump maps and BRDFs that represented higher-frequency details. Schilling (1997, 2001) investigated this issue particularly for application to graphics hardware. More recently, effective approaches have been developed by Toksvig (2005) and Han et al. (2007).

An alternative to bump mapping is displacement mapping, where the bump function is used to actually modify the surface geometry, rather than just perturbing the normal (Cook 1984; Cook, Carpenter, and Catmull 1987). Advantages of displacement mapping include geometric detail on object silhouettes and the possibility of accounting for self-shadowing. Patterson and collaborators have described an innovative algorithm for displacement mapping with ray tracing where the geometry is unperturbed but the ray's direction is modified such that the intersections that are found are the same as would be found with the displaced geometry (Patterson, Hoggar, and Logie 1991; Logie and Patterson 1994). Heidrich and Seidel (1998) developed a technique for computing direct intersections with procedurally defined displacement functions.

With the advent of increased memory on computers and caching algorithms, the option of finely tessellating geometry and displacing its vertices for ray tracing has become feasible. Pharr and Hanrahan (1996) described an approach to this problem based on geometry caching, and Wang et al. (2000) described an adaptive tessellation algorithm that reduces memory requirements. Smits, Shirley, and Stark (2000) lazily tessellate individual triangles, saving a substantial amount of memory.

The book by Dorsey et al. (2008) provides comprehensive coverage of material and reflection modeling for computer graphics.

EXERCISES

`Material` 483
`PlasticMaterial::Kd` 487
`PlasticMaterial::Ks` 487
`Texture` 519

● 9.1 If the same `Texture` is bound to more than one component of a `Material` (for example, to both `PlasticMaterial::Kd` and `PlasticMaterial::Ks`), the texture will be evaluated twice. This unnecessarily duplicated work may lead to a noticeable increase in rendering time if the `Texture` is itself computationally expensive. Modify the materials in `pbrt` to eliminate this problem. Measure the

change in the system's performance, both for standard scenes as well as for contrived scenes that exhibit this redundancy.

● 9.2 One form of aliasing that pbrt doesn't try to eliminate is specular highlight aliasing. Glossy specular surfaces with high specular exponents, particularly if they have high curvature, are susceptible to aliasing as small changes in incident direction or surface position (and thus surface normal) may cause the highlight's contribution to change substantially. Read Amanatides's paper on this topic (Amanatides 1992) and extend pbrt to reduce specular aliasing, either using his technique or by developing your own. Most of the quantities needed to do the appropriate computations are already available—$\partial \mathbf{n}/\partial x$ and $\partial \mathbf{n}/\partial y$ in the DifferentialGeometry, and so on—although it will probably be necessary to extend the BxDF interface to provide more information about the values of specular exponents for glossy specular reflection components.

● 9.3 Another approach to specular highlight aliasing is to supersample the BSDF, evaluating it multiple times around the point being shaded. After reading the discussion of supersampling texture functions in Section 10.1, modify the BSDF::f() method to shift around to a set of positions around the intersection point but within the pixel sampling rate around the intersection point and evaluate the BSDF at each one of them when the BSDF evaluation routines are called. (Be sure to account for the change in normal using its partial derivatives.) How well does this approach combat specular highlight aliasing?

● 9.4 pbrt doesn't directly represent interfaces between objects that transmit light; instead, it assumes that for transmitted rays entering or leaving an object, the index of refraction outside the object is one. This assumption breaks down for many common cases—for example, a glass of water where rays pass directly from the glass into the water. Furthermore, in pbrt, if such a scene is modeled with the edge of the water coincident with the interior edge of the glass, the transmission from glass to water may be missed completely, since only one of the two ray intersections would be detected. Modify the system to eliminate this problem and render images showing that your approach renders correct images where pbrt currently computes an incorrect result. See, for example, Schmidt and Budge (2002) for one approach to this issue.

● 9.5 Read some of the papers on filtering bump maps referenced in the "Further Reading" section of this chapter, choose one of the techniques described there, and implement it in pbrt. Show both the visual artifacts from bump map aliasing without the technique you implement as well as examples of how well your implementation addresses them.

● 9.6 Neyret (1996, 1998) developed algorithms that automatically take descriptions of complex shapes and their reflective properties and turn them into generalized reflection models at different resolutions, each with limited frequency content. The advantage of this representation is that it makes it easy to select an appropriate level of detail for an object based on its size on the screen, thus reducing aliasing. Read these papers and implement the algorithms described

BSDF::f() 481
BxDF 428
DifferentialGeometry 102

in them in pbrt. Show how they can be used to reduce geometric aliasing from detailed geometry and extend them to address bump map aliasing.

● 9.7 Use the triangular face refinement infrastructure from the LoopSubdiv shape to implement displacement mapping in pbrt. The usual approach to displacement mapping is to finely tessellate the geometric shape and then to evaluate the displacement function at its vertices, moving each vertex the given distance along its normal. Because displacement mapping may make the extent of the shape larger, the bounding box of the undisplaced shape will need to be expanded by the maximum displacement distance that a particular displacement function will ever generate.

Refine each face of the mesh until, when projected onto the image, it is roughly the size of the separation between pixels. To do this, you will need to be able to estimate the image pixel-based length of an edge in the scene when it is projected onto the screen. Use the texturing infrastructure in Chapter 10 to evaluate displacement functions. See Patney et al. (2009) and Fisher et al. (2009) for discussion of issues related to avoiding cracks in the mesh due to adaptive tessellation.