CHAPTER THREE

03 SHAPES

In this chapter, we will present pbrt's abstraction for geometric primitives such as spheres and triangles. Careful abstraction of geometric shapes in a ray tracer is a key component of a clean system design, and shapes are the ideal candidate for an object-oriented approach. All geometric primitives implement a common interface, and the rest of the renderer can use this interface without needing any details about the underlying shape. This makes it possible to isolate the geometric and shading subsystems of pbrt. Without this isolation, adding new shapes to the system would be unnecessarily difficult and error prone.

pbrt hides details about its primitives behind a two-level abstraction. The Shape class provides access to the raw geometric properties of the primitive, such as its surface area and bounding box, and provides a ray intersection routine. The Primitive class provides additional nongeometric information about the primitive, such as its material properties. The rest of the renderer then deals only with the abstract Primitive interface. This chapter will focus on the geometry-only Shape class; the Primitive interface is the topic of Chapter 4.

The interface for Shapes is in the source file core/shape.h, and definitions of common Shape methods can be found in core/shape.cpp.

3.1 BASIC SHAPE INTERFACE

The Shape class in pbrt is *reference counted*—pbrt keeps track of the number of outstanding pointers to a particular shape and automatically deletes the shape when that reference count goes to zero. Although not completely foolproof, this is a form of *garbage collection* that eliminates the need to have to worry about freeing shape memory at the wrong

time. The ReferenceCounted class handles all of the underlying mechanisms; its implementation is in Section A.5.2 in Appendix A. Reference counted objects, like Shape, must inherit from the ReferenceCounted base class.

⟨*Shape Declarations*⟩ ≡
```
class Shape : public ReferenceCounted {
public:
    ⟨Shape Interface 109⟩
    ⟨Shape Public Data 108⟩
};
```

All shapes are defined in object coordinate space; for example, all spheres are defined in a coordinate system where the center of the sphere is at the origin. In order to place a sphere at another position in the scene, a transformation that describes the mapping from object space to world space must be provided. The Shape class stores both this transformation and its inverse. Shapes also take a Boolean parameter, ReverseOrientation, that indicates whether their surface normal directions should be reversed from the default. This capability is useful because the orientation of the surface normal is used to determine which side of a shape is "outside." For example, shapes that emit illumination are emissive only on the side the surface normal lies on. The value of this parameter is set via the ReverseOrientation statement in pbrt input files.

Shapes also store the result of the Transform::SwapsHandedness() call for their object-to-world transformation. This value is needed by the DifferentialGeometry constructor that will be called each time a ray intersection is found, so the Shape constructor computes it once and stores it.

⟨*Shape Method Definitions*⟩ ≡
```
Shape::Shape(const Transform *o2w, const Transform *w2o, bool ro)
    : ObjectToWorld(o2w), WorldToObject(w2o), ReverseOrientation(ro),
      TransformSwapsHandedness(o2w->SwapsHandedness()),
      shapeId(nextshapeId++) {
}
```

An important detail is that shapes store pointers to their transformations rather than Transform objects directly. Recall from Section 2.7 that Transform objects require 32 floats, requiring 128 bytes of memory; because multiple shapes in the scene will frequently have the same transformation applied to them, pbrt keeps a pool of Transforms so that they can be re-used and passes pointers to the shared Transforms to the shapes. As such, the Shape destructor does not delete its Transform pointers, leaving the Transform management code to manage that memory instead.

⟨*Shape Public Data*⟩ ≡ 108
```
const Transform *ObjectToWorld, *WorldToObject;
const bool ReverseOrientation, TransformSwapsHandedness;
```

All Shapes in the system are given a unique 32-bit numeric id, stored here in the shapeId member variable. This identifier has a variety of uses, among them the adaptive image sampling routines that take additional samples in the areas of pixels that have multiple shapes overlapping them—pixels with more geometric complexity are likely to require

ReferenceCounted 1010
Shape 108
Transform 76
Transform::
SwapsHandedness() 89

higher sampling rates than those with less for a high-quality final image. Having this identifier available makes it easy for those routines to detect this situation.

⟨*Shape Public Data*⟩ +≡ 108
```
const uint32_t shapeId;
static uint32_t nextshapeId;
```

The first value assigned for a shape id is one, so that the value of zero for a shape id can be used to indicate "no shape" (for example, as the default value in the `Intersection` structure, which stores the id of the shape that a ray hits).

⟨*Shape Method Definitions*⟩ +≡
```
uint32_t Shape::nextshapeId = 1;
```

3.1.1 BOUNDING

Each `Shape` subclass must be capable of bounding itself with a `BBox`. There are two different bounding methods. The first, `ObjectBound()`, returns a bounding box in the shape's object space, and the second, `WorldBound()`, returns a bounding box in world space. The implementation of the first method is left up to each individual shape, but there is a default implementation of the second method that transforms the object bound to world space. Shapes that can easily compute a tighter world space bound should override this method, however. An example of such a shape is a triangle (Figure 3.1).

⟨*Shape Interface*⟩ ≡ 108
```
virtual BBox ObjectBound() const = 0;
```

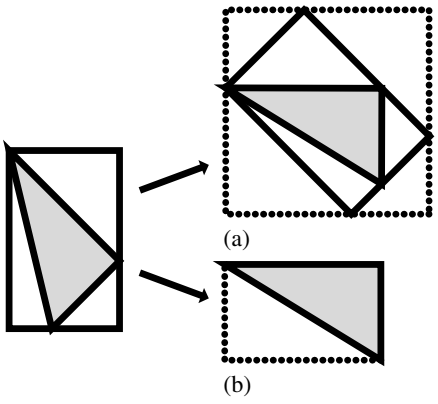

Figure 3.1: A world space bounding box of a triangle is computed by transforming its object space bounding box to world space and then finding the bounding box that encloses the resulting bounding box; a sloppy bound may result. (a) However, if the triangle's vertices are first transformed from object space to world space and then bounded (b), the fit of the bounding box can be much better.

`BBox` 70
`Intersection` 186
`Shape` 108

⟨*Shape Method Definitions*⟩ +≡
```
BBox Shape::WorldBound() const {
    return (*ObjectToWorld)(ObjectBound());
}
```

3.1.2 REFINEMENT

Not every shape needs to be capable of determining whether a ray intersects it. For example, a complex surface might first be tessellated into triangles, which can then be intersected directly. Another possibility is a shape that is a placeholder for a large amount of geometry stored on disk. We could store just the filename of the geometry file and the bounding box of the geometry in memory, and read the geometry in from disk only if a ray pierces the bounding box.

The default implementation of the `Shape::CanIntersect()` function indicates that a shape *can* compute ray intersections, so only shapes that are nonintersectable need to override this method.

⟨*Shape Method Definitions*⟩ +≡
```
bool Shape::CanIntersect() const {
    return true;
}
```

If the shape cannot be intersected directly, it must provide a `Shape::Refine()` method that splits the shape into a group of new shapes, some of which may be intersectable and some of which may need further refinement. The default implementation of the `Shape::Refine()` method issues an error message; thus, shapes that are intersectable (which is the common case) do not have to provide an empty instance of this method. `pbrt` will never call `Shape::Refine()` if `Shape::CanIntersect()` returns `true`.

⟨*Shape Method Definitions*⟩ +≡
```
void Shape::Refine(vector<Reference<Shape> > &refined) const {
    Severe("Unimplemented Shape::Refine() method called");
}
```

3.1.3 INTERSECTION

The `Shape` class provides two intersection routines. The first, `Shape::Intersect()`, returns geometric information about a single ray-shape intersection corresponding to the first intersection, if any, in the `[mint, maxt]` parametric range along the ray. The other, `Shape::IntersectP()`, is a predicate function that determines whether or not an intersection occurs, without returning any details about the intersection itself. Most shape implementations can provide a more efficient implementation for `IntersectP()` that can determine whether an intersection exists without computing all of its details.

There are a few important things to keep in mind when reading (and writing) intersection routines:

- The `Ray` structure contains `Ray::mint` and `Ray::maxt` variables that define a ray *segment*. Intersection routines must ignore any intersections that do not occur along this segment.

BBox 70
Ray 66
Ray::maxt 67
Ray::mint 67
Reference 1011
Severe() 1005
Shape 108
Shape::CanIntersect() 110
Shape::Intersect() 111
Shape::IntersectP() 111
Shape::ObjectToWorld 108
Shape::Refine() 110

- If an intersection is found, its parametric distance along the ray should be stored in the pointer `tHit` that is passed into the intersection routine. If there are multiple intersections along the ray, the closest one should be reported.
- If an intersection is found, the value pointed to by the `rayEpsilon` parameter must be initialized with a value that describes the maximum numeric error in the intersection calculation. This parameter is discussed further in Section 3.1.4.
- Information about an intersection is stored in the `DifferentialGeometry` structure, which completely captures the local geometric properties of a surface. This class is used heavily throughout `pbrt`, and it serves to cleanly isolate the geometric portion of the ray tracer from the shading and illumination portions. The `DifferentialGeometry` class was defined in Section 2.10.[1]
- The rays passed into intersection routines are in world space, so shapes are responsible for transforming them to object space if needed for intersection tests. The differential geometry returned should be in world space.

Rather than making the intersection routines pure virtual functions, the `Shape` class provides default implementations of the intersect routines that print an error message if they are called. All `Shapes` that return `true` from `Shape::CanIntersect()` must provide implementations of these functions; those that return `false` can depend on `pbrt` not to call these routines on nonintersectable shapes. If these were pure virtual functions, then each nonintersectable shape would have to implement a similar default function.

⟨*Shape Method Definitions*⟩ +≡
```
bool Shape::Intersect(const Ray &ray, float *tHit, float *rayEpsilon,
                      DifferentialGeometry *dg) const {
    Severe("Unimplemented Shape::Intersect() method called");
    return false;
}
```

⟨*Shape Method Definitions*⟩ +≡
```
bool Shape::IntersectP(const Ray &ray) const {
    Severe("Unimplemented Shape::IntersectP() method called");
    return false;
}
```

3.1.4 AVOIDING INCORRECT SELF-INTERSECTIONS

A classic issue in ray tracing is the self-intersection problem, where floating-point errors can lead to incorrect self-intersections when rays are traced starting from a previously computed ray-surface intersection point (e.g., shadow rays or rays for specular reflection and refraction.). Figure 3.2 illustrates the problem: Represented in finite-precision

`DifferentialGeometry` 102
`Ray` 66
`Severe()` 1005
`Shape` 108
`Shape::CanIntersect()` 110

1 Almost all ray tracers use this general idiom for returning geometric information about intersections with shapes. As an optimization, many will only partially initialize the intersection information when an intersection is found, storing just enough information so that the rest of the values can be computed later if actually needed. This approach saves work in the case where a closer intersection is later found with another shape. In our experience, the extra work to compute all the information isn't substantial, and for renderers that have complex scene data management algorithms (e.g., discarding geometry from main memory when too much memory is being used and writing it to disk), the deferred approach may fail because the shape is no longer in memory.

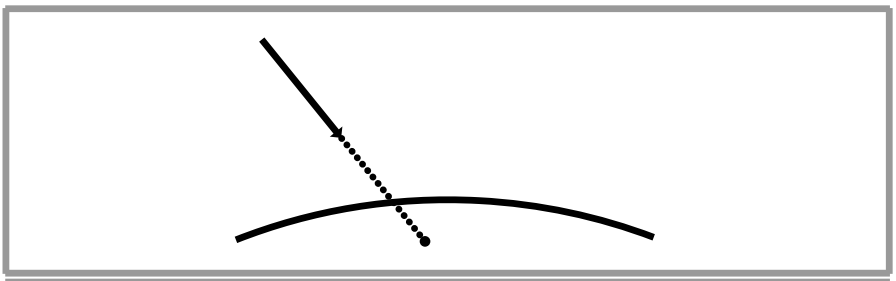

Figure 3.2: Due to finite floating-point precision and round-off error, when the intersection of a ray is found with a shape, the computed intersection point may lie slightly above or slightly below the true intersection point. This can lead to rendering errors when reflected and shadow rays are traced starting from the computed intersection point, as incorrect self-intersections with the surface may be detected.

floating-point numbers, the computed intersection point may lie slightly below the actual surface. If new rays are traced with this point as their origin, it's possible that they may intersect the surface even though a ray actually starting from the surface would not.

`pbrt` addresses this issue by requiring shape `Intersection()` method implementations to return a value through the `rayEpsilon` parameter that gives the maximum floating-point error in the parametric value computed for the intersection point. The integrators in Chapters 15 through 17 adjust the origins of rays spawned from intersection points based on this error term in order to ensure that they do not incorrectly reintersect the surface.

In general, computing this floating-point error bound requires careful numerical analysis of the specific computations performed in the particular ray–shape intersection algorithm used.

3.1.5 SHADING GEOMETRY

Some shapes (notably triangle meshes) support the idea of having two types of differential geometry at a point on the surface: the true geometry, which accurately reflects the local properties of the surface, and the *shading geometry*, which may have normals and tangents that are different than those in the true differential geometry. For triangle meshes, the user can provide normal vectors and primary tangents at the vertices of the mesh that are interpolated to give normals and tangents at points across the faces of triangles. Shading geometry with interpolated normals can make otherwise faceted triangle meshes appear to be more smooth than they geometrically are.

The `Shape::GetShadingGeometry()` method returns the shading geometry corresponding to the `DifferentialGeometry` returned by the `Shape::Intersect()` routine. By default, the shading geometry matches the true geometry, so the default implementation just copies the true geometry. One subtlety is that an object-to-world transformation is passed to this routine; if the routine needs to transform data from object space to world

DifferentialGeometry 102

space to compute the shading geometry, it must use this transformation rather than the Shape::ObjectToWorld transformation. This allows object instancing to be implemented in pbrt (see Section 4.1.2).

⟨*Shape Interface*⟩ +≡ 108
```
virtual void GetShadingGeometry(const Transform &obj2world,
        const DifferentialGeometry &dg,
        DifferentialGeometry *dgShading) const {
    *dgShading = dg;
}
```

3.1.6 SURFACE AREA

In order to properly use Shapes as area lights, it is necessary to be able to compute the surface area of a shape in object space. As with the intersection methods, this method will only be called for intersectable shapes.

⟨*Shape Method Definitions*⟩ +≡
```
float Shape::Area() const {
    Severe("Unimplemented Shape::Area() method called");
    return 0.;
}
```

3.1.7 SIDEDNESS

Many rendering systems, particularly those based on scan line or z-buffer algorithms, support the concept of shapes being "one-sided"—the shape is visible if seen from the front but disappears when viewed from behind. In particular, if a geometric object is closed and always viewed from the outside, then the back-facing parts of it can be discarded without changing the resulting image. This optimization can substantially improve the speed of these types of hidden surface removal algorithms. The potential for improved performance is reduced when using this technique with ray tracing, however, since it is often necessary to perform the ray-object intersection before determining the surface normal to do the back-facing test. Furthermore, this feature can lead to a physically inconsistent scene description if one-sided objects are not in fact closed. For example, a surface might block light when a shadow ray is traced from a light source to a point on another surface, but not if the shadow ray is traced in the other direction. For all of these reasons, pbrt doesn't support this feature.

3.2 SPHERES

Spheres are a special case of a general type of surface called *quadrics*—surfaces described by quadratic polynomials in x, y, and z. They are the simplest type of curved surface that is useful to a ray tracer and are a good starting point for general ray intersection routines. pbrt supports six types of quadrics: spheres, cones, disks (a special case of a cone), cylinders, hyperboloids, and paraboloids.

DifferentialGeometry 102
Severe() 1005
Shape::ObjectToWorld 108
Transform 76

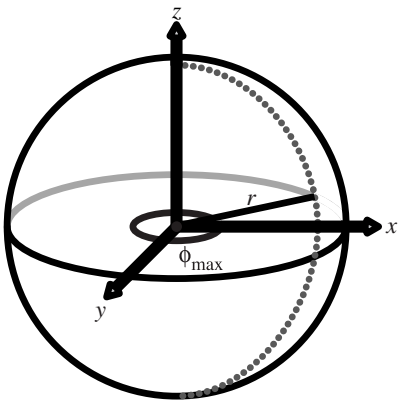

Figure 3.3: Basic Setting for the Sphere Shape. It has a radius of r and is centered at the object space origin. A partial sphere may be described by specifying a maximum ϕ value.

Many surfaces can be described in one of two main ways: in *implicit form* and in *parametric form*. An implicit function describes a 3D surface as

$$f(x, y, z) = 0.$$

The set of all points (x, y, z) that fulfill this condition defines the surface. For a unit sphere at the origin, the familiar implicit equation is $x^2 + y^2 + z^2 - 1 = 0$. Only the set of points one unit from the origin satisfies this constraint, giving the unit sphere's surface.

Many surfaces can also be described parametrically using a function to map 2D points to 3D points on the surface. For example, a sphere of radius r can be described as a function of 2D spherical coordinates (θ, ϕ), where θ ranges from 0 to π and ϕ ranges from 0 to 2π (Figure 3.3):

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta. \end{aligned}$$

We can transform this function $f(\theta, \phi)$ into a function $f(u, v)$ over $[0, 1]^2$ and also generalize it slightly to allow partial spheres that only sweep out $\theta \in [\theta_{\min}, \theta_{\max}]$ and $\phi \in [0, \phi_{\max}]$ with the substitution

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ \theta &= \theta_{\min} + v(\theta_{\max} - \theta_{\min}). \end{aligned}$$

This form is particularly useful for texture mapping, where it can be directly used to map a texture defined over $[0, 1]^2$ to the sphere. Figure 3.4 shows an image of two spheres; a grid image map has been used to show the (u, v) parameterization.

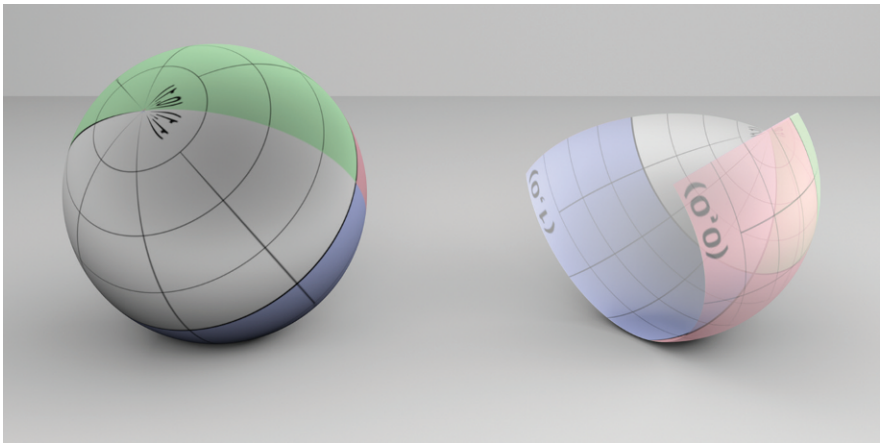

Figure 3.4: Two Spheres. On the left is a complete sphere, and on the right is a partial sphere (with $z_{\max} < r$ and $\phi_{\max} < 2\pi$). Note that the texture map used shows the (u, v) parameterization of the shape; the singularity at one of the poles is visible in the complete sphere.

As we describe the implementation of the sphere shape, we will make use of both the implicit and parametric descriptions of the shape, depending on which is a more natural way to approach the particular problem we're facing.

3.2.1 CONSTRUCTION

⟨*Sphere Declarations*⟩ ≡
```
class Sphere : public Shape {
public:
    ⟨Sphere Public Methods⟩
private:
    ⟨Sphere Private Data 116⟩
};
```

The `Sphere` class represents a sphere that is centered at the origin in object space. To place it elsewhere in the scene, the user must apply an appropriate transformation when specifying the sphere in the input file. It takes both the object-to-world and world-to-object transformations as parameters to the constructor, passing them along to the parent `Shape` constructor. Its implementation is in the files `shapes/sphere.h` and `shapes/sphere.cpp`.

The radius of the sphere can have an arbitrary positive value, and the sphere's extent can be truncated in two different ways. First, minimum and maximum z values may be set; the parts of the sphere below and above these, respectively, are cut off. Second, considering the parameterization of the sphere in spherical coordinates, a maximum ϕ value can be set. The sphere sweeps out ϕ values from 0 to the given $\phi_{\max}$ such that the section of the sphere with spherical ϕ values above $\phi_{\max}$ is also removed.

Shape 108
Sphere 115

⟨*Sphere Method Definitions*⟩ ≡
```
Sphere::Sphere(const Transform *o2w, const Transform *w2o, bool ro,
               float rad, float z0, float z1, float pm)
    : Shape(o2w, w2o, ro) {
    radius = rad;
    zmin = Clamp(min(z0, z1), -radius, radius);
    zmax = Clamp(max(z0, z1), -radius, radius);
    thetaMin = acosf(Clamp(zmin/radius, -1.f, 1.f));
    thetaMax = acosf(Clamp(zmax/radius, -1.f, 1.f));
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

⟨*Sphere Private Data*⟩ ≡ 115
```
float radius;
float phiMax;
float zmin, zmax;
float thetaMin, thetaMax;
```

3.2.2 BOUNDING

Computing a bounding box for a sphere is straightforward. The implementation here uses the values of $z_{\min}$ and $z_{\max}$ provided by the user to tighten up the bound when less than an entire sphere is being rendered. However, it doesn't do the extra work to look at $\phi_{\max}$ and see if it can compute a tighter bounding box when $\phi_{\max}$ is less than 2π. This improvement is left as an exercise.

⟨*Sphere Method Definitions*⟩ +≡
```
BBox Sphere::ObjectBound() const {
    return BBox(Point(-radius, -radius, zmin),
                Point( radius,  radius, zmax));
}
```

3.2.3 INTERSECTION

The task of deriving an intersection test is simplified by the fact that the sphere is centered at the origin. However, if the sphere has been transformed to another position in world space, then it is necessary to transform rays to object space before intersecting them with the sphere, using the world-to-object transformation. Given a ray in object space, the intersection computation can be performed in object space instead.[2]

The following fragment shows the entire intersection method:

BBox 70
Clamp() 1000
Point 63
Radians() 1001
Shape 108
Sphere 115
Sphere::phiMax 116
Sphere::radius 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Sphere::zmax 116
Sphere::zmin 116
Transform 76

2 This is something of a classic theme in computer graphics. By transforming the problem to a particular restricted case, it is possible to more easily and efficiently do an intersection test: that is, many terms of the equations cancel out since the sphere is always at $(0, 0, 0)$. No overall generality is lost, since an appropriate translation can be applied to the ray for spheres at other positions.

⟨*Sphere Method Definitions*⟩ +≡
```
bool Sphere::Intersect(const Ray &r, float *tHit, float *rayEpsilon,
                       DifferentialGeometry *dg) const {
    float phi;
    Point phit;
    ⟨Transform Ray to object space 117⟩
    ⟨Compute quadratic sphere coefficients 118⟩
    ⟨Solve quadratic equation for t values 118⟩
    ⟨Compute sphere hit position and φ 119⟩
    ⟨Test sphere intersection against clipping parameters 120⟩
    ⟨Find parametric representation of sphere hit 120⟩
    ⟨Initialize DifferentialGeometry from parametric information 122⟩
    ⟨Update tHit for quadric intersection 123⟩
    ⟨Compute rayEpsilon for quadric intersection 123⟩
    return true;
}
```

First, the given world space ray is transformed to the sphere's object space. The remainder of the intersection test will take place in that coordinate system.

⟨*Transform* Ray *to object space*⟩ ≡ 117, 123, 127, 131
```
Ray ray;
(*WorldToObject)(r, &ray);
```

If a sphere is centered at the origin with radius r, its implicit representation is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

By substituting the parametric representation of the ray, given in Equation (2.3), into the implicit sphere equation, we have

$$(\mathrm{o}_x + t\mathbf{d}_x)^2 + (\mathrm{o}_y + t\mathbf{d}_y)^2 + (\mathrm{o}_z + t\mathbf{d}_z)^2 = r^2.$$

Note that all elements of this equation besides t are known values. The t values where the equation holds give the parametric positions along the ray where the implicit sphere equation holds and thus the points along the ray where it intersects the sphere. We can expand this equation and gather the coefficients for a general quadratic equation in t,

$$At^2 + Bt + C = 0,$$

where[3]

$$A = \mathbf{d}_x^2 + \mathbf{d}_y^2 + \mathbf{d}_z^2$$
$$B = 2(\mathbf{d}_x\mathrm{o}_x + \mathbf{d}_y\mathrm{o}_y + \mathbf{d}_z\mathrm{o}_z)$$
$$C = \mathrm{o}_x^2 + \mathrm{o}_y^2 + \mathrm{o}_z^2 - r^2.$$

DifferentialGeometry 102
Point 63
Ray 66
Shape::WorldToObject 108
Sphere 115

3 Some ray tracers require that the direction vector of a ray be normalized, meaning $A = 1$. This can lead to subtle errors, however, if the caller forgets to normalize the ray direction. Of course, these errors can be avoided by normalizing the direction in the ray constructor, but this wastes effort when the provided direction is *already* normalized. To avoid this needless complexity, pbrt never insists on vector normalization in intersection routines. This is particularly helpful since it reduces the amount of computation needed to transform rays to object space, because no normalization is necessary there.

This directly translates to this fragment of source code:

⟨*Compute quadratic sphere coefficients*⟩ ≡ 117, 123
```
float A = ray.d.x*ray.d.x + ray.d.y*ray.d.y + ray.d.z*ray.d.z;
float B = 2 * (ray.d.x*ray.o.x + ray.d.y*ray.o.y + ray.d.z*ray.o.z);
float C = ray.o.x*ray.o.x + ray.o.y*ray.o.y +
          ray.o.z*ray.o.z - radius*radius;
```

There are two possible solutions to this quadratic equation, giving zero, one, or two nonimaginary t values where the ray intersects the sphere:

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$
$$t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}.$$

The `Quadratic()` utility function solves a quadratic equation, returning `false` if there are no real solutions and returning `true` and setting `t0` and `t1` appropriately if there are solutions:

⟨*Solve quadratic equation for* `t` *values*⟩ ≡ 117, 123, 127
```
float t0, t1;
if (!Quadratic(A, B, C, &t0, &t1))
    return false;
⟨Compute intersection distance along ray 119⟩
```

⟨*Global Inline Functions*⟩ ≡
```
inline bool Quadratic(float A, float B, float C, float *t0, float *t1) {
    ⟨Find quadratic discriminant 118⟩
    ⟨Compute quadratic t values 119⟩
}
```

If the discriminant ($B^2 - 4AC$) is negative, then there are no real roots and the ray must miss the sphere:

⟨*Find quadratic discriminant*⟩ ≡ 118
```
float discrim = B * B - 4.f * A * C;
if (discrim <= 0.) return false;
float rootDiscrim = sqrtf(discrim);
```

The usual version of the quadratic equation can give poor numeric precision when $B \approx \pm\sqrt{B^2 - 4AC}$ due to cancellation error. It can be rewritten algebraically to a more stable form:

$$t_0 = \frac{q}{A}$$
$$t_1 = \frac{C}{q},$$

where

$$q = \begin{cases} -.5(B - \sqrt{B^2 - 4AC}) & B < 0 \\ -.5(B + \sqrt{B^2 - 4AC}) & \text{otherwise.} \end{cases}$$

Quadratic() 118
Sphere::radius 116

⟨*Compute quadratic* t *values*⟩ ≡ 118
```
float q;
if (B < 0) q = -.5f * (B - rootDiscrim);
else       q = -.5f * (B + rootDiscrim);
*t0 = q / A;
*t1 = C / q;
if (*t0 > *t1) swap(*t0, *t1);
return true;
```

Given the two intersection t values, the intersection method checks them against the ray segment from `mint` to `maxt`. Since t_0 is guaranteed to be less than t_1 (and `mint` less than `maxt`), if t_0 is greater than `maxt` or t_1 is less than `mint`, then it is certain that both hits are out of the range of interest. Otherwise, t_0 is the tentative hit distance. It may be less than `mint`, however, in which case we ignore it and try t_1. If that is also out of range, we have no valid intersection. If there is an intersection, `thit` holds the distance to the hit.

⟨*Compute intersection distance along ray*⟩ ≡ 118
```
if (t0 > ray.maxt || t1 < ray.mint)
    return false;
float thit = t0;
if (t0 < ray.mint) {
    thit = t1;
    if (thit > ray.maxt) return false;
}
```

3.2.4 PARTIAL SPHERES

Given the parametric distance along the ray to the intersection with a full sphere, it is necessary to handle partial spheres with clipped z or ϕ ranges. Intersections that are in clipped areas need to be ignored. The implementation starts by computing the object space position of the intersection, `phit`, and the ϕ value for the hit point. Using the parametric representation of the sphere,

$$\frac{y}{x} = \frac{r \sin\theta \sin\phi}{r \sin\theta \cos\phi} = \tan\phi,$$

so $\phi = \arctan y/x$. It is necessary to remap the result of the C standard library's `atan2f` function to a value between 0 and 2π, to match the sphere's original definition.

⟨*Compute sphere hit position and* ϕ⟩ ≡ 117, 120, 123
```
phit = ray(thit);
if (phit.x == 0.f && phit.y == 0.f) phit.x = 1e-5f * radius;
phi = atan2f(phit.y, phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

The hit point can now be tested against the specified minima and maxima for z and ϕ. One subtlety is that it's important to skip the z tests if the z range includes the entire sphere; the computed `phit.z` value may be slightly out of the z range due to floating-point roundoff, so we should only perform this test when the user expects the sphere to be partially incomplete. If the t_0 intersection wasn't actually valid, the routine tries again with t_1.

M_PI 1002

⟨*Test sphere intersection against clipping parameters*⟩ ≡ 117, 123
```
if ((zmin > -radius && phit.z < zmin) ||
    (zmax <  radius && phit.z > zmax) || phi > phiMax) {
    if (thit == t1) return false;
    if (t1 > ray.maxt) return false;
    thit = t1;
    ⟨Compute sphere hit position and φ 119⟩
    if ((zmin > -radius && phit.z < zmin) ||
        (zmax <  radius && phit.z > zmax) || phi > phiMax)
        return false;
}
```

At this point in the routine, it is certain that the ray hits the sphere, and the `DifferentialGeometry` structure can be initialized. The method computes u and v values by scaling the previously computed ϕ value for the hit to lie between 0 and 1 and by computing a θ value between 0 and 1 for the hit point, based on the range of θ values for the given sphere. Then, it finds the parametric partial derivatives of position $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ and surface normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$.

⟨*Find parametric representation of sphere hit*⟩ ≡ 117
```
float u = phi / phiMax;
float theta = acosf(Clamp(phit.z / radius, -1.f, 1.f));
float v = (theta - thetaMin) / (thetaMax - thetaMin);
⟨Compute sphere ∂p/∂u and ∂p/∂v 121⟩
⟨Compute sphere ∂n/∂u and ∂n/∂v 122⟩
```

Computing the partial derivatives of a point on the sphere is a short exercise in algebra. Here we will show how the x component of $\partial \mathrm{p}/\partial u$, $\partial \mathrm{p}_x/\partial u$, is calculated; the other components are found similarly. Using the parametric definition of the sphere, we have

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ \frac{\partial \mathrm{p}_x}{\partial u} &= \frac{\partial}{\partial u}(r \sin\theta \cos\phi) \\ &= r \sin\theta \frac{\partial}{\partial u}(\cos\phi) \\ &= r \sin\theta(-\phi_{\max} \sin\phi). \end{aligned}$$

Using a substitution based on the parametric definition of the sphere's y coordinate, this simplifies to

$$\frac{\partial \mathrm{p}_x}{\partial u} = -\phi_{\max} y.$$

Similarly,

$$\frac{\partial \mathrm{p}_y}{\partial u} = \phi_{\max} x,$$

and

$$\frac{\partial \mathrm{p}_z}{\partial u} = 0.$$

Clamp() 1000
DifferentialGeometry 102
Sphere::phiMax 116
Sphere::radius 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Sphere::zmax 116
Sphere::zmin 116

A similar process gives $\partial \mathrm{p}/\partial v$. The complete result is

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y, \phi_{\max} x, 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = (\theta_{\max} - \theta_{\min})(z \cos\phi, z \sin\phi, -r \sin\theta).$$

⟨*Compute sphere* $\partial \mathrm{p}/\partial u$ *and* $\partial \mathrm{p}/\partial v$⟩ ≡ 120

```
float zradius = sqrtf(phit.x*phit.x + phit.y*phit.y);
float invzradius = 1.f / zradius;
float cosphi = phit.x * invzradius;
float sinphi = phit.y * invzradius;
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0);
Vector dpdv = (thetaMax-thetaMin) *
    Vector(phit.z * cosphi, phit.z * sinphi,
           -radius * sinf(theta));
```

★ 3.2.5 PARTIAL DERIVATIVES OF NORMAL VECTORS

It is also useful to determine how the normal changes as we move along the surface in the u and v directions. For example, the antialiasing techniques in Chapter 10 are dependent on this information to antialias textures on objects that are seen reflected in curved surfaces. The differential changes in normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ are given by the *Weingarten equations* from differential geometry:

$$\frac{\partial \mathbf{n}}{\partial u} = \frac{fF - eG}{EG - F^2}\frac{\partial \mathrm{p}}{\partial u} + \frac{eF - fE}{EG - F^2}\frac{\partial \mathrm{p}}{\partial v}$$
$$\frac{\partial \mathbf{n}}{\partial v} = \frac{gF - fG}{EG - F^2}\frac{\partial \mathrm{p}}{\partial u} + \frac{fF - gE}{EG - F^2}\frac{\partial \mathrm{p}}{\partial v},$$

where E, F, and G are coefficients of the *first fundamental form* and are given by

$$E = \left|\frac{\partial \mathrm{p}}{\partial u}\right|^2$$
$$F = \left(\frac{\partial \mathrm{p}}{\partial u} \cdot \frac{\partial \mathrm{p}}{\partial v}\right)$$
$$G = \left|\frac{\partial \mathrm{p}}{\partial v}\right|^2.$$

These are easily computed with the $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ values found earlier. The e, f, and g are coefficients of the *second fundamental form*,

$$e = \left(\mathbf{n} \cdot \frac{\partial^2 \mathrm{p}}{\partial u^2}\right)$$
$$f = \left(\mathbf{n} \cdot \frac{\partial^2 \mathrm{p}}{\partial u \partial v}\right)$$
$$g = \left(\mathbf{n} \cdot \frac{\partial^2 \mathrm{p}}{\partial v^2}\right).$$

Sphere::phiMax 116
Sphere::radius 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Vector 57

The two fundamental forms have basic connections with the local curvature of a surface; see a differential geometry textbook such as Gray (1993) for details. To find e, f, and g, it is necessary to compute the second-order partial derivatives $\partial^2\mathrm{p}/\partial u^2$ and so on.

For spheres, a little more algebra gives the second derivatives:

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = (\theta_{\max} - \theta_{\min})\, z\, \phi_{\max}(-\sin\phi, \cos\phi, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = -(\theta_{\max} - \theta_{\min})^2(x, y, z).$$

⟨*Compute sphere* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$⟩ ≡ 120
```
Vector d2Pduu = -phiMax * phiMax * Vector(phit.x, phit.y, 0);
Vector d2Pduv = (thetaMax - thetaMin) * phit.z * phiMax *
                Vector(-sinphi, cosphi, 0.);
Vector d2Pdvv = -(thetaMax - thetaMin) * (thetaMax - thetaMin) *
                Vector(phit.x, phit.y, phit.z);
⟨Compute coefficients for fundamental forms 122⟩
⟨Compute ∂n/∂u and ∂n/∂v from fundamental form coefficients 122⟩
```

⟨*Compute coefficients for fundamental forms*⟩ ≡ 122, 128
```
float E = Dot(dpdu, dpdu);
float F = Dot(dpdu, dpdv);
float G = Dot(dpdv, dpdv);
Vector N = Normalize(Cross(dpdu, dpdv));
float e = Dot(N, d2Pduu);
float f = Dot(N, d2Pduv);
float g = Dot(N, d2Pdvv);
```

⟨*Compute* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$ *from fundamental form coefficients*⟩ ≡ 122, 128
```
float invEGF2 = 1.f / (E*G - F*F);
Normal dndu = Normal((f*F - e*G) * invEGF2 * dpdu +
                     (e*F - f*E) * invEGF2 * dpdv);
Normal dndv = Normal((g*F - f*G) * invEGF2 * dpdu +
                     (f*F - g*E) * invEGF2 * dpdv);
```

3.2.6 `DifferentialGeometry` INITIALIZATION

Having computed the surface parameterization and all the relevant partial derivatives, the `DifferentialGeometry` structure can be initialized with the geometric information for this intersection:

⟨*Initialize* `DifferentialGeometry` *from parametric information*⟩ ≡ 117, 127, 131
```
const Transform &o2w = *ObjectToWorld;
*dg = DifferentialGeometry(o2w(phit), o2w(dpdu), o2w(dpdv),
                           o2w(dndu), o2w(dndv), u, v, this);
```

DifferentialGeometry 102
Dot() 60
Normal 65
Shape::ObjectToWorld 108
Sphere::phiMax 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Sphere::zmax 116
Sphere::zmin 116
Transform 76
Vector 57

Since there is an intersection, the tHit parameter is updated with the parametric hit distance along the ray, which was stored in thit. This will allow subsequent intersection tests to terminate early if the potential hit would be farther away than the existing intersection.

A natural question to ask at this point is, "What effect does the world-to-object transformation have on the correct parametric distance to return?" Indeed, the intersection method has found a parametric distance to the intersection for the object space ray, which may have been translated, rotated, scaled, or worse when it was transformed from world space. However, it can be shown that the parametric distance to an intersection in object space is exactly the same as it would have been if the ray was left in world space and the intersection had been done there and, thus, tHit can be initialized directly. Note that if the object space ray's direction had been normalized after the transformation, then this would no longer be the case and a correction factor related to the unnormalized ray's length would be needed. This is another motivation for not normalizing the object space ray's direction vector after transformation.

⟨*Update* tHit *for quadric intersection*⟩ ≡ 117, 127, 131
```
*tHit = thit;
```

We have not rigorously derived floating-point error bounds for the sphere ray–object intersection algorithm implemented here, leaving that for an exercise at the end of the chapter. However, we have found that this expression generally works well.

⟨*Compute* rayEpsilon *for quadric intersection*⟩ ≡ 117, 127, 131
```
*rayEpsilon = 5e-4f * *tHit;
```

The Sphere::IntersectP() routine is almost identical to Sphere::Intersect(), but it does not fill in the DifferentialGeometry structure. Because the Intersect() and IntersectP() methods are always so closely related, we will not show implementations of IntersectP() for the remaining shapes.

⟨*Sphere Method Definitions*⟩ +≡
```
bool Sphere::IntersectP(const Ray &r) const {
    float phi;
    Point phit;
    ⟨Transform Ray to object space 117⟩
    ⟨Compute quadratic sphere coefficients 118⟩
    ⟨Solve quadratic equation for t values 118⟩
    ⟨Compute sphere hit position and φ 119⟩
    ⟨Test sphere intersection against clipping parameters 120⟩
    return true;
}
```

3.2.7 SURFACE AREA

Point 63
Ray 66
Sphere 115
Sphere::Intersect() 117

To compute the surface area of quadrics, we use a standard formula from integral calculus. If a curve $y = f(x)$ from $y = a$ to $y = b$ is revolved around the x axis, the surface area of the resulting swept surface is

$$2\pi \int_a^b f(x)\sqrt{1 + \left(f'(x)\right)^2}\,\mathrm{d}x,$$

where $f'(x)$ denotes the derivative $\mathrm{d}f/\mathrm{d}x$.[4] Since most of our surfaces of revolution are only partially swept around the axis, we will instead use the formula

$$\phi_{\max} \int_a^b f(x)\sqrt{1 + \left(f'(x)\right)^2}\,\mathrm{d}x.$$

The sphere is a surface of revolution of a circular arc. The function that defines the profile curve of the sphere is

$$f(x) = \sqrt{r^2 - x^2},$$

and its derivative is

$$f'(x) = -\frac{x}{\sqrt{r^2 - x^2}}.$$

Recall that the sphere is clipped at $z_{\min}$ and $z_{\max}$. The surface area is therefore

$$\begin{aligned} A &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} \sqrt{r^2 - x^2}\sqrt{1 + \frac{x^2}{r^2 - x^2}}\,\mathrm{d}x \\ &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} \sqrt{r^2 - x^2 + x^2}\,\mathrm{d}x \\ &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} r\,\mathrm{d}x \\ &= \phi_{\max}\, r\, (z_{\max} - z_{\min}). \end{aligned}$$

For the full sphere $\phi_{\max} = 2\pi$, $z_{\min} = -r$, and $z_{\max} = r$, so we have the standard formula $A = 4\pi r^2$, confirming that the formula makes sense.

⟨*Sphere Method Definitions*⟩ +≡

```
float Sphere::Area() const {
    return phiMax * radius * (zmax-zmin);
}
```

3.3 CYLINDERS

⟨*Cylinder Declarations*⟩ ≡

```
class Cylinder : public Shape {
public:
    ⟨Cylinder Public Methods⟩
protected:
    ⟨Cylinder Private Data 126⟩
};
```

Shape 108
Sphere 115
Sphere::phiMax 116
Sphere::radius 116
Sphere::zmax 116
Sphere::zmin 116

4 See Anton, Bivens, and Davis (2001) for a derivation.

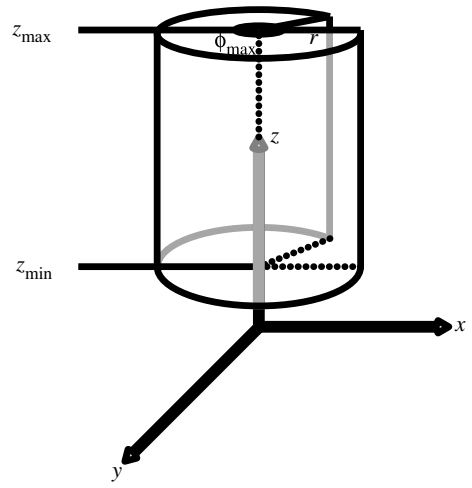

Figure 3.5: Basic Setting for the Cylinder Shape. It has a radius of r and covers a range along the z axis. A partial cylinder may be swept by specifying a maximum ϕ value.

3.3.1 CONSTRUCTION

Another useful quadric is the cylinder; `pbrt` provides cylinder `Shapes` that are centered around the z axis. The implementation is in the files `shapes/cylinder.h` and `shapes/cylinder.cpp`. The user supplies a minimum and maximum z value for the cylinder, as well as a radius and maximum ϕ sweep value (Figure 3.5). In parametric form, a cylinder is described by the following equations:

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ x &= r\cos\phi \\ y &= r\sin\phi \\ z &= z_{\min} + v(z_{\max} - z_{\min}). \end{aligned}$$

Figure 3.6 shows a rendered image of two cylinders. Like the sphere image, the left cylinder is a complete cylinder, while the right one is a partial cylinder due to having a $\phi_{\max}$ value less than 2π.

Clamp() 1000
Cylinder 124
Cylinder::phiMax 126
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Radians() 1001
Shape 108
Transform 76

⟨*Cylinder Method Definitions*⟩ ≡

```
Cylinder::Cylinder(const Transform *o2w, const Transform *w2o, bool ro,
                   float rad, float z0, float z1, float pm)
    : Shape(o2w, w2o, ro) {
    radius = rad;
    zmin = min(z0, z1);
    zmax = max(z0, z1);
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

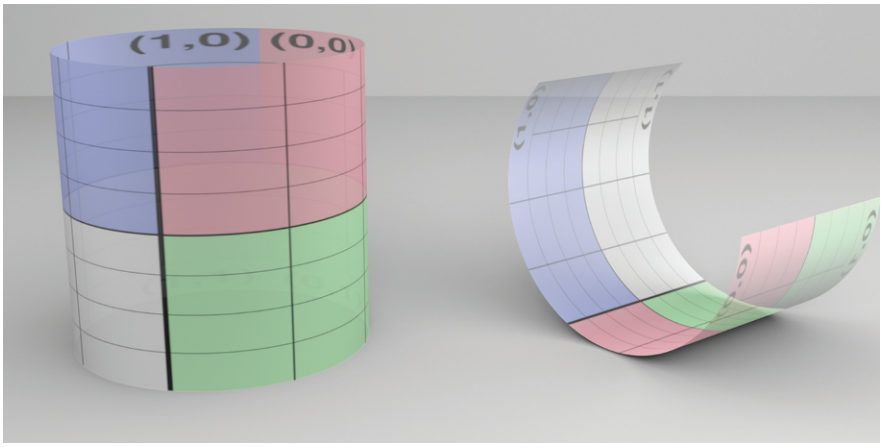

Figure 3.6: Two Cylinders. A complete cylinder is on the left, and a partial cylinder is on the right.

⟨*Cylinder Private Data*⟩ ≡ 124
```
float radius, zmin, zmax, phiMax;
```

3.3.2 BOUNDING

As was done with the sphere, the bounding method computes a conservative bounding box for the cylinder using the z range but without taking into account the maximum ϕ.

⟨*Cylinder Method Definitions*⟩ +≡
```
BBox Cylinder::ObjectBound() const {
    Point p1 = Point(-radius, -radius, zmin);
    Point p2 = Point( radius,  radius, zmax);
    return BBox(p1, p2);
}
```

3.3.3 INTERSECTION

The ray–cylinder intersection formula can be found by substituting the ray equation into the cylinder's implicit equation, similarly to the sphere case. The implicit equation for an infinitely long cylinder centered on the z axis with radius r is

$$x^2 + y^2 - r^2 = 0.$$

Substituting the ray equation, Equation (2.3), we have

$$(o_x + td_x)^2 + (o_y + td_y)^2 = r^2.$$

When we expand this and find the coefficients of the quadratic equation $At^2 + Bt + C$, we have

BBox 70
Cylinder 124
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Point 63

$$A = \mathbf{d}_x^2 + \mathbf{d}_y^2$$
$$B = 2(\mathbf{d}_x \mathrm{o}_x + \mathbf{d}_y \mathrm{o}_y)$$
$$C = \mathrm{o}_x^2 + \mathrm{o}_y^2 - r^2.$$

⟨*Compute quadratic cylinder coefficients*⟩ ≡ 127
```
float A = ray.d.x*ray.d.x + ray.d.y*ray.d.y;
float B = 2 * (ray.d.x*ray.o.x + ray.d.y*ray.o.y);
float C = ray.o.x*ray.o.x + ray.o.y*ray.o.y - radius*radius;
```

The solution process for the quadratic equation is similar for all quadric shapes, so some fragments from the Sphere intersection method will be reused in the following.

⟨*Cylinder Method Definitions*⟩ +≡
```
bool Cylinder::Intersect(const Ray &r, float *tHit, float *rayEpsilon,
                         DifferentialGeometry *dg) const {
    float phi;
    Point phit;
    ⟨Transform Ray to object space 117⟩
    ⟨Compute quadratic cylinder coefficients 127⟩
    ⟨Solve quadratic equation for t values 118⟩
    ⟨Compute cylinder hit point and φ 127⟩
    ⟨Test cylinder intersection against clipping parameters 128⟩
    ⟨Find parametric representation of cylinder hit 128⟩
    ⟨Initialize DifferentialGeometry from parametric information 122⟩
    ⟨Update tHit for quadric intersection 123⟩
    ⟨Compute rayEpsilon for quadric intersection 123⟩
    return true;
}
```

3.3.4 PARTIAL CYLINDERS

As with the sphere, we can invert the parametric description of the cylinder to compute a ϕ value by inverting the x and y parametric equations to solve for ϕ. In fact, the result is the same as for the sphere.

⟨*Compute cylinder hit point and φ*⟩ ≡ 127, 128
```
phit = ray(thit);
phi = atan2f(phit.y, phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

Cylinder 124
DifferentialGeometry 102
M_PI 1002
Point 63
Ray 66
Sphere 115

The intersection routine now makes sure that the hit is in the specified z range, and that the angle ϕ is acceptable. If not, it rejects the hit and tries with t_1, if it hasn't already, just like the sphere.

⟨*Test cylinder intersection against clipping parameters*⟩ ≡ 127
```
if (phit.z < zmin || phit.z > zmax || phi > phiMax) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > ray.maxt) return false;
    ⟨Compute cylinder hit point and φ 127⟩
    if (phit.z < zmin || phit.z > zmax || phi > phiMax)
        return false;
}
```

Again the u value is computed by scaling ϕ to lie between 0 and 1. Straightforward inversion of the parametric equation for the cylinder's z value gives the v parametric coordinate.

⟨*Find parametric representation of cylinder hit*⟩ ≡ 127
```
float u = phi / phiMax;
float v = (phit.z - zmin) / (zmax - zmin);
⟨Compute cylinder ∂p/∂u and ∂p/∂v 128⟩
⟨Compute cylinder ∂n/∂u and ∂n/∂v 128⟩
```

The partial derivatives for a cylinder are quite easy to derive:

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y,\ \phi_{\max} x,\ 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = (0,\ 0,\ z_{\max} - z_{\min}).$$

⟨*Compute cylinder ∂p/∂u and ∂p/∂v*⟩ ≡ 128
```
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0);
Vector dpdv(0, 0, zmax - zmin);
```

We again use the Weingarten equations to compute the parametric change in the cylinder normal. The relevant partial derivatives are

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2 (x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = (0, 0, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = (0, 0, 0).$$

⟨*Compute cylinder ∂n/∂u and ∂n/∂v*⟩ ≡ 128
```
Vector d2Pduu = -phiMax * phiMax * Vector(phit.x, phit.y, 0);
Vector d2Pduv(0, 0, 0), d2Pdvv(0, 0, 0);
⟨Compute coefficients for fundamental forms 122⟩
⟨Compute ∂n/∂u and ∂n/∂v from fundamental form coefficients 122⟩
```

Cylinder::phiMax 126
Cylinder::zmax 126
Cylinder::zmin 126
Vector 57

3.3.5 SURFACE AREA

A cylinder is just a rolled-up rectangle. If you unroll the rectangle, its height is $z_{\max} - z_{\min}$, and its width is $r\phi_{\max}$:

⟨*Cylinder Method Definitions*⟩ +≡
```
float Cylinder::Area() const {
    return (zmax-zmin) * phiMax * radius;
}
```

3.4 DISKS

⟨*Disk Declarations*⟩ ≡
```
class Disk : public Shape {
public:
    ⟨Disk Public Methods⟩
private:
    ⟨Disk Private Data 130⟩
};
```

The disk is an interesting quadric since it has a particularly straightforward intersection routine that avoids solving the quadratic equation. In `pbrt`, a `Disk` is a circular disk of radius r at height h along the z axis. It is implemented in the files `shapes/disk.h` and `shapes/disk.cpp`. In order to make partial disks, the user may specify a maximum ϕ value beyond which the disk is cut off (Figure 3.7). The disk can also be generalized to an

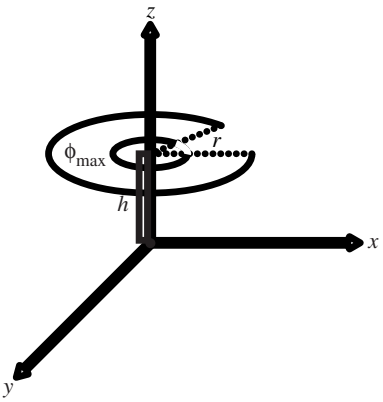

Figure 3.7: Basic Setting for the Disk Shape. The disk has radius r and is located at height h along the z axis. A partial disk may be swept by specifying a maximum ϕ value and an inner radius r_i.

Cylinder 124
Cylinder::phiMax 126
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Disk 129
Shape 108

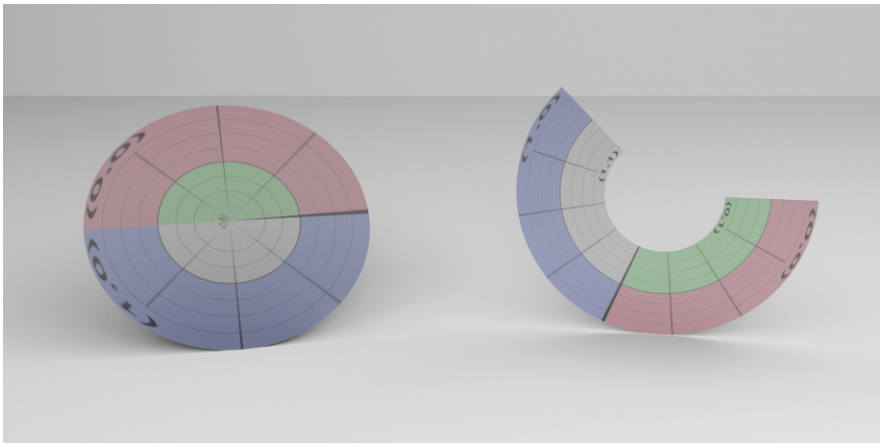

Figure 3.8: Two Disks. A complete disk is on the left, and a partial disk is on the right.

annulus by specifying an inner radius, r_i. In parametric form, it is described by

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ x &= ((1-v)r_i + vr)\cos\phi \\ y &= ((1-v)r_i + vr)\sin\phi \\ z &= h. \end{aligned}$$

Figure 3.8 is a rendered image of two disks.

3.4.1 CONSTRUCTION

⟨*Disk Method Definitions*⟩ ≡

```
Disk::Disk(const Transform *o2w, const Transform *w2o, bool ro,
           float ht, float r, float ri, float tmax)
    : Shape(o2w, w2o, ro) {
    height = ht;
    radius = r;
    innerRadius = ri;
    phiMax = Radians(Clamp(tmax, 0.0f, 360.0f));
}
```

⟨*Disk Private Data*⟩ ≡ 129

```
float height, radius, innerRadius, phiMax;
```

Clamp() 1000
Disk 129
Disk::phiMax 130
Radians() 1001
Shape 108
Transform 76

3.4.2 BOUNDING

The bounding method is quite straightforward; it computes a bounding box centered at the height of the disk along z, with extent of `radius` in both the x and y directions.

⟨*Disk Method Definitions*⟩ +≡
```
BBox Disk::ObjectBound() const {
    return BBox(Point(-radius, -radius, height),
                Point( radius,  radius, height));
}
```

3.4.3 INTERSECTION

Intersecting a ray with a disk is also quite easy. The intersection of the ray with the $z = h$ plane that the disk lies in is found and the intersection point is checked to see if it lies inside the disk.

⟨*Disk Method Definitions*⟩ +≡
```
bool Disk::Intersect(const Ray &r, float *tHit, float *rayEpsilon,
                     DifferentialGeometry *dg) const {
    ⟨Transform Ray to object space 117⟩
    ⟨Compute plane intersection for disk 131⟩
    ⟨See if hit point is inside disk radii and φmax 132⟩
    ⟨Find parametric representation of disk hit 132⟩
    ⟨Initialize DifferentialGeometry from parametric information 122⟩
    ⟨Update tHit for quadric intersection 123⟩
    ⟨Compute rayEpsilon for quadric intersection 123⟩
    return true;
}
```

The first step is to compute the parametric t value where the ray intersects the plane that the disk lies in. We want to find t such that the z component of the ray's position is equal to the height of the disk. Thus,

$$h = \mathrm{o}_z + t\mathbf{d}_z$$

and

$$t = \frac{h - \mathrm{o}_z}{\mathbf{d}_z}.$$

If the ray is parallel to the disk's plane (i.e., the z component of its direction is zero), there is no intersection. Otherwise, the intersection method then checks if t is inside the legal range of values [`mint`, `maxt`]. If not, the routine can return `false`.

⟨*Compute plane intersection for disk*⟩ ≡ 131
```
if (fabsf(ray.d.z) < 1e-7) return false;
float thit = (height - ray.o.z) / ray.d.z;
if (thit < ray.mint || thit > ray.maxt)
    return false;
```

Now the intersection method can compute the point `phit` where the ray intersects the plane. Once the plane intersection is known, `false` is returned if the distance from the hit to the center of the disk is more than `Disk::radius` or less than `Disk::innerRadius`. This process can be optimized by actually computing the squared distance to the center,

BBox 70
DifferentialGeometry 102
Disk 129
Disk::height 130
Disk::innerRadius 130
Disk::radius 130
Point 63
Ray 66

taking advantage of the fact that the x and y coordinates of the center point (0, 0, `height`) are zero, and the z coordinate of `phit` is equal to `height`.

⟨*See if hit point is inside disk radii and $\phi_{\max}$*⟩ ≡ 131
```
Point phit = ray(thit);
float dist2 = phit.x * phit.x + phit.y * phit.y;
if (dist2 > radius * radius || dist2 < innerRadius * innerRadius)
    return false;
⟨Test disk φ value against φmax 132⟩
```

If the distance check passes, a final test makes sure that the ϕ value of the hit point is between zero and $\phi_{\max}$, specified by the caller. Inverting the disk's parameterization gives the same expression for ϕ as the other quadric shapes.

⟨*Test disk ϕ value against $\phi_{\max}$*⟩ ≡ 132
```
float phi = atan2f(phit.y, phit.x);
if (phi < 0) phi += 2. * M_PI;
if (phi > phiMax)
    return false;
```

If we've gotten this far, there is an intersection with the disk. The parameter `u` is scaled to reflect the partial disk specified by $\phi_{\max}$, and `v` is computed by inverting the parametric equation. The equations for the partial derivatives at the hit point can be derived with a process similar to that used for the previous quadrics. Because the normal of a disk is the same everywhere, the partial derivatives $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ are both trivially (0, 0, 0).

⟨*Find parametric representation of disk hit*⟩ ≡ 131
```
float u = phi / phiMax;
float v = 1.f - ((sqrtf(dist2)-innerRadius) /
                 (radius-innerRadius));
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0.);
Vector dpdv(-phit.x / (1-v), -phit.y / (1-v), 0.);
dpdu *= phiMax * INV_TWOPI;
dpdv *= (radius - innerRadius) / radius;
Normal dndu(0,0,0), dndv(0,0,0);
```

3.4.4 SURFACE AREA

Disks have trivially computed surface area, since they're just portions of an annulus:

$$A = \frac{\phi_{\max}}{2}(r^2 - r_{\mathrm{i}}^2).$$

⟨*Disk Method Definitions*⟩ +≡
```
float Disk::Area() const {
    return phiMax * 0.5f *
       (radius * radius - innerRadius * innerRadius);
}
```

Disk 129
Disk::innerRadius 130
Disk::phiMax 130
Disk::radius 130
INV_TWOPI 1002
M_PI 1002
Normal 65
Point 63
Vector 57

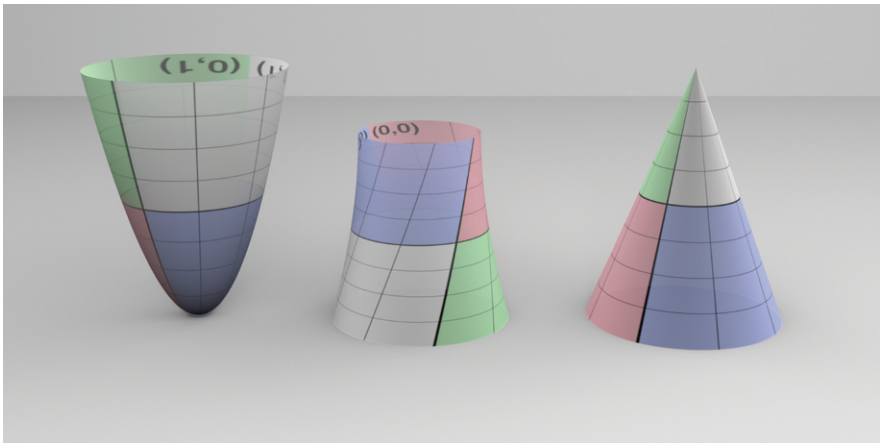

Figure 3.9: The Remaining Quadric Shapes. From left to right: the paraboloid, the hyperboloid, and the cone.

3.5 OTHER QUADRICS

`pbrt` supports three more quadrics: cones, paraboloids, and hyperboloids. They are implemented in the source files `shapes/cone.h`, `shapes/cone.cpp`, `shapes/paraboloid.h`, `shapes/paraboloid.cpp`, `shapes/hyperboloid.h`, and `shapes/hyperboloid.cpp`. We won't include their full implementations here, since the techniques used to derive their quadratic intersection coefficients, parametric coordinates, and partial derivatives should now be familiar. However, we will briefly summarize the implicit and parametric forms of these shapes. A rendered image of the three of them is in Figure 3.9.

3.5.1 CONES

The implicit equation of a cone centered on the z axis with radius r and height h is

$$\left(\frac{hx}{r}\right)^2 + \left(\frac{hy}{r}\right)^2 - (z-h)^2 = 0.$$

Cones are also described parametrically:

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ x &= r(1-v)\cos\phi \\ y &= r(1-v)\sin\phi \\ z &= vh. \end{aligned}$$

The partial derivatives at a point on a cone are

$$\begin{aligned} \frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0) \\ \frac{\partial \mathrm{p}}{\partial v} &= \left(-\frac{x}{1-v}, \frac{y}{1-v}, h\right), \end{aligned}$$

and the second partial derivatives are

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = \frac{\phi_{\max}}{1-v}(y, -x, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = (0, 0, 0).$$

3.5.2 PARABOLOIDS

The implicit equation of a paraboloid centered on the z axis with radius r and height h is

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0,$$

and its parametric form is

$$\phi = u\ \phi_{\max}$$
$$z = v(z_{\max} - z_{\min})$$
$$r = r_{\max}\sqrt{\frac{z}{z_{\max}}}$$
$$x = r\ \cos\phi$$
$$y = r\ \sin\phi.$$

The partial derivatives are

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y, \phi_{\max} x, 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = (z_{\max} - z_{\min})\left(\frac{x}{2z}, \frac{y}{2z}, 1\right),$$

and

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = \phi_{\max}(z_{\max} - z_{\min})\left(-\frac{y}{2z}, \frac{x}{2z}, 0\right)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = -(z_{\max} - z_{\min})^2\left(\frac{x}{4z^2}, \frac{y}{4z^2}, 0\right).$$

3.5.3 HYPERBOLOIDS

Finally, the implicit form of the hyperboloid is

$$x^2 + y^2 - z^2 = -1,$$

and the parametric form is

$$\begin{aligned}
\phi &= u\,\phi_{\max} \\
x_r &= (1-v)x_1 + v\,x_2 \\
y_r &= (1-v)y_1 + v\,y_2 \\
x &= x_r \cos\phi - y_r \sin\phi \\
y &= x_r \sin\phi + y_r \cos\phi \\
z &= (1-v)z_1 + v\,z_2.
\end{aligned}$$

The partial derivatives are

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y, \phi_{\max} x, 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = ((x_2 - x_1)\cos\phi - (y_2 - y_1)\sin\phi, (x_2 - x_1)\sin\phi + (y_2 - y_1)\cos\phi, z_2 - z_1),$$

and

$$\begin{aligned}
\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0) \\
\frac{\partial^2 \mathrm{p}}{\partial u \partial v} &= \phi_{\max} \left(-\frac{\partial \mathrm{p}_y}{\partial v}, \frac{\partial \mathrm{p}_x}{\partial v}, 0 \right) \\
\frac{\partial^2 \mathrm{p}}{\partial v^2} &= (0, 0, 0).
\end{aligned}$$

3.6 TRIANGLES AND MESHES

⟨*TriangleMesh Declarations*⟩ ≡
```
class TriangleMesh : public Shape {
public:
    ⟨TriangleMesh Public Methods 138⟩
protected:
    ⟨TriangleMesh Protected Data 137⟩
};
```

The triangle is one of the most commonly used shapes in computer graphics. `pbrt` supports *triangle meshes*, where a number of triangles are stored together so that their per-vertex data (position, shading normal, etc.) can be shared among multiple triangles. Single triangles are simply treated as degenerate meshes. Figure 3.10 shows an image of a complex triangle mesh of over one million triangles.

The arguments to the `TriangleMesh` constructor are as follows:

- `nt`: Number of triangles.
- `nv`: Number of vertices.
- `vi`: Pointer to an array of vertex indices. For the `i`th triangle, its three vertex positions are `P[vi[3*i]]`, `P[vi[3*i+1]]`, and `P[vi[3*i+2]]`.
- `P`: Array of `nv` vertex positions.
- `N`: An optional array of normal vectors, one per vertex in the mesh. If present, these are interpolated across triangle faces to compute shading differential geometry.

Shape 108
TriangleMesh 135

Figure 3.10: Happy Buddha Model. This triangle mesh contains over one million individual triangles. It was created from a real statue using a 3D laser scanner. *(Model courtesy of Stanford Computer Graphics Laboratory Scanning Repository.)*

- `S`: An optional array of tangent vectors, one per vertex in the mesh. These are also used to compute shading geometry.
- `uv`: An optional array of parametric (u, v) values, one for each vertex.
- `atex`: An optional *alpha mask* texture, which can be used to cut away part of the triangle's surface.

The constructor copies the relevant information and stores it in the `TriangleMesh` object. In particular, it must make its own copies of `vi`, `P`, `N`, and `S`, since the caller retains ownership of the data being passed in.

`TriangleMesh` 135

Triangles have a dual role among the primitives in `pbrt`: not only are they a user-supplied primitive, but other primitives often tessellate themselves into triangle meshes. For exam-

ple, subdivision surfaces end up creating a mesh of triangles to approximate the smooth limit surface. Ray intersections are performed against these triangles, rather than directly against the subdivision surface. (See Section 3.7.3.)

Due to this second role, it's important that a routine that is creating a triangle mesh be able to specify the parameterization of the triangles. If a triangle was created by evaluating the position of a parametric surface at three particular (u, v) coordinate values, for example, those (u, v) values should be interpolated to compute the (u, v) value at ray intersection points inside the triangle; hence the `uv` parameter. The (u, v) values are also useful for texture mapping, where an external program that created a triangle mesh may want to assign (u, v) coordinates to the mesh so that a texture map assigns color to the mesh surface in the desired way.

⟨*TriangleMesh Method Definitions*⟩ ≡
```
TriangleMesh::TriangleMesh(const Transform *o2w, const Transform *w2o,
        bool ro, int nt, int nv, const int *vi, const Point *P,
        const Normal *N, const Vector *S, const float *uv,
        const Reference<Texture<float> > &atex)
    : Shape(o2w, w2o, ro), alphaTexture(atex) {
    ntris = nt;
    nverts = nv;
    vertexIndex = new int[3 * ntris];
    memcpy(vertexIndex, vi, 3 * ntris * sizeof(int));
    ⟨Copy uv, N, and S vertex data, if present⟩
    ⟨Transform mesh vertices to world space 138⟩
}
```

The ⟨*Copy* `uv`, `N`, *and* `S` *vertex data, if present*⟩ fragment just allocates the appropriate amount of space and copies any data. Its implementation isn't included here.

⟨*TriangleMesh Protected Data*⟩ ≡ 135
```
int ntris, nverts;
int *vertexIndex;
Point *p;
Normal *n;
Vector *s;
float *uvs;
Reference<Texture<float> > alphaTexture;
```

Normal 65
Point 63
Reference 1011
Shape 108
Texture 519
Transform 76
TriangleMesh 135
Vector 57

Unlike the other shapes that leave the primitive description in object space and then transform incoming rays from world space to object space, triangle meshes transform the shape into world space and thus save the work of transforming incoming rays into object space and the work of transforming the intersection's differential geometry out to world space. This is a good idea because this operation can be performed once at start-up, avoiding transforming rays many times during rendering. Doing this with quadrics is more complicated, although possible—see Exercise 3.1 at the end of the chapter for more information. Normal and tangent vectors for shading geometry are left in object space, since the `GetShadingGeometry()` method must transform them to world space with the

transformation matrix supplied to that method, which may not necessarily be the one stored by the Shape.

⟨*Transform mesh vertices to world space*⟩ ≡ 137
```
for (int i = 0; i < nverts; ++i)
    p[i] = (*ObjectToWorld)(P[i]);
```

The object space bound of a triangle mesh is easily found by computing a bounding box that encompasses all of the vertices of the mesh. Because the vertex positions p are transformed to world space in the constructor, the implementation here has to transform them back to object space before computing their bound. Note that these bounds are fairly expensive to compute for large triangle meshes. The implementation assumes that the caller will cache the results of these computations if they will be reused, which is the case elsewhere in pbrt where the bounding methods are called.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox TriangleMesh::ObjectBound() const {
    BBox objectBounds;
    for (int i = 0; i < nverts; i++)
        objectBounds = Union(objectBounds, (*WorldToObject)(p[i]));
    return objectBounds;
}
```

The TriangleMesh shape is one of the shapes that can compute a better world space bound than can be found by transforming its object space bounding box to world space. Its world space bound can be directly computed from the world space vertices.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox TriangleMesh::WorldBound() const {
    BBox worldBounds;
    for (int i = 0; i < nverts; i++)
        worldBounds = Union(worldBounds, p[i]);
    return worldBounds;
}
```

The TriangleMesh shape does not directly compute intersections. Instead, it splits itself into many separate Triangles, each representing a single triangle. All of the individual triangles reference the shared set of vertices in p, avoiding per-triangle replication of the shared data. The TriangleMesh class therefore overrides the Shape::CanIntersect() method to indicate that TriangleMeshes cannot be intersected directly.

⟨*TriangleMesh Public Methods*⟩ ≡ 135
```
bool CanIntersect() const { return false; }
```

When pbrt encounters a shape that cannot be intersected directly, it calls its Refine() method. Shape::Refine() is expected to produce a list of simpler shapes in the vector passed in, refined. The implementation here is simple; it creates a new Triangle for each of the triangles in the mesh.

BBox 70
Shape::CanIntersect() 110
Shape::ObjectToWorld 108
Shape::Refine() 110
Shape::WorldToObject 108
Triangle 139
TriangleMesh 135
TriangleMesh::nverts 137
TriangleMesh::p 137
Union() 72

⟨*TriangleMesh Method Definitions*⟩ +≡
```
void TriangleMesh::Refine(vector<Reference<Shape> > &refined) const {
    for (int i = 0; i < ntris; ++i)
        refined.push_back(new Triangle(ObjectToWorld,
                          WorldToObject, ReverseOrientation,
                          (TriangleMesh *)this, i));
}
```

3.6.1 TRIANGLE

⟨*TriangleMesh Declarations*⟩ +≡
```
class Triangle : public Shape {
public:
    ⟨Triangle Public Methods 139⟩
private:
    ⟨Triangle Private Data 139⟩
};
```

The `Triangle` doesn't store much data—just a pointer to the parent `TriangleMesh` that it came from and a pointer to its three vertex indices in the mesh:

⟨*Triangle Public Methods*⟩ ≡ 139
```
Triangle(const Transform *o2w, const Transform *w2o, bool ro,
         TriangleMesh *m, int n)
    : Shape(o2w, w2o, ro) {
    mesh = m;
    v = &mesh->vertexIndex[3*n];
}
```

Note that the implementation stores a pointer to the first vertex *index*, instead of storing three pointers to the vertices themselves. This reduces the amount of storage required for each `Triangle` at a cost of another level of indirection.

⟨*Triangle Private Data*⟩ ≡ 139
```
Reference<TriangleMesh> mesh;
int *v;
```

As with `TriangleMeshes`, it is possible to compute better world space bounding boxes for individual triangles by bounding the world space vertices directly.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox Triangle::ObjectBound() const {
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    return Union(BBox((*WorldToObject)(p1), (*WorldToObject)(p2)),
                 (*WorldToObject)(p3));
}
```

BBox 70
Reference 1011
Shape 108
Shape::ObjectToWorld 108
Shape::ReverseOrientation 108
Shape::WorldToObject 108
Transform 76
Triangle 139
TriangleMesh 135
TriangleMesh::ntris 137
TriangleMesh::vertexIndex 137

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox Triangle::WorldBound() const {
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    return Union(BBox(p1, p2), p3);
}
```

⟨*Get triangle vertices in* p1, p2, *and* p3⟩ ≡ 139, 140, 141, 145, 719
```
const Point &p1 = mesh->p[v[0]];
const Point &p2 = mesh->p[v[1]];
const Point &p3 = mesh->p[v[2]];
```

3.6.2 TRIANGLE INTERSECTION

An efficient algorithm for ray–triangle intersection can be derived using *barycentric coordinates*. Barycentric coordinates provide a way to parameterize a triangle in terms of two variables, b_1 and b_2:

$$\mathrm{p}(b_1, b_2) = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2.$$

The conditions on b_1 and b_2 are that $b_1 \geq 0$, $b_2 \geq 0$, and $b_1 + b_2 \leq 1$. The barycentric coordinates are also a natural way to interpolate across the surface of the triangle. Given values a_0, a_1, and a_2 defined at the vertices and given the barycentric coordinates for a point on the triangle, we can compute an interpolated value of a at that point as $(1 - b_1 - b_2)a_0 + b_1a_1 + b_2a_2$.

To derive an algorithm for intersecting a ray with a triangle, we insert the parametric ray equation into the triangle equation:

$$\mathrm{o} + t\mathbf{d} = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2. \tag{3.1}$$

Following the technique described by Möller and Trumbore (1997), we use the shorthand notation $\mathbf{e}_1 = \mathrm{p}_1 - \mathrm{p}_0$, $\mathbf{e}_2 = \mathrm{p}_2 - \mathrm{p}_0$, and $\mathbf{s} = \mathrm{o} - \mathrm{p}_0$. We can now rearrange the terms of Equation (3.1) to obtain the matrix equation

$$\begin{pmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \mathbf{s}. \tag{3.2}$$

Solving this linear system will give us both the barycentric coordinates of the intersection point as well as the parametric distance along the ray. Geometrically, we can interpret this system as a translation of the triangle to the origin and a transformation of the triangle to a unit right triangle in y and z.

We can easily solve Equation (3.2) using Cramer's rule. Note that we are introducing a bit of notation for brevity here; we write $|\ \mathbf{a} \quad \mathbf{b} \quad \mathbf{c}\ |$ to mean the determinant of the matrix having $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ as its columns. Cramer's rule gives

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{|\ -\mathbf{d} \quad \mathbf{e}_1 \quad \mathbf{e}_2\ |} \begin{bmatrix} |\ \mathbf{s} \quad \mathbf{e}_1 \quad \mathbf{e}_2| \\ |-\mathbf{d} \quad \mathbf{s} \quad \mathbf{e}_2| \\ |-\mathbf{d} \quad \mathbf{e}_1 \quad \mathbf{s}| \end{bmatrix}. \tag{3.3}$$

BBox 70
Point 63
Triangle 139
TriangleMesh::p 137

This can be rewritten using the identity $|\,\mathbf{a}\quad\mathbf{b}\quad\mathbf{c}\,| = -(\mathbf{a}\times\mathbf{c})\cdot\mathbf{b} = -(\mathbf{c}\times\mathbf{b})\cdot\mathbf{a}$. We then obtain

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{(\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{e}_1} \begin{bmatrix} (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{e}_2 \\ (\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{d} \end{bmatrix}. \tag{3.4}$$

If we use the substitution $\mathbf{s}_1 = \mathbf{d}\times\mathbf{e}_2$ and $\mathbf{s}_2 = \mathbf{s}\times\mathbf{e}_1$, the common subexpressions are made more explicit:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\mathbf{s}_1\cdot\mathbf{e}_1} \begin{bmatrix} \mathbf{s}_2\cdot\mathbf{e}_2 \\ \mathbf{s}_1\cdot\mathbf{s} \\ \mathbf{s}_2\cdot\mathbf{d} \end{bmatrix}. \tag{3.5}$$

In order to compute $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{s}$ we need 9 subtractions. To compute $\mathbf{s}_1$ and $\mathbf{s}_2$, we need 2 cross products, which is a total of 12 multiplications and 6 subtractions. Finally, to compute t, b_1, and b_2, we need 4 dot products (12 multiplications and 8 additions), 1 reciprocal, and 3 multiplications. Thus, the total cost of ray-triangle intersection is 1 divide, 27 multiplies, and 17 adds (counting adds and subtracts together). Note that some of these operations can be avoided if it is determined midcalculation that the ray does not intersect the triangle.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
bool Triangle::Intersect(const Ray &ray, float *tHit, float *rayEpsilon,
                         DifferentialGeometry *dg) const {
    ⟨Compute s1 141⟩
    ⟨Compute first barycentric coordinate 142⟩
    ⟨Compute second barycentric coordinate 142⟩
    ⟨Compute t to intersection point 142⟩
    ⟨Compute triangle partial derivatives 143⟩
    ⟨Interpolate (u, v) triangle parametric coordinates 143⟩
    ⟨Test intersection against alpha texture, if present 144⟩
    ⟨Fill in DifferentialGeometry from triangle hit 145⟩
    *tHit = t;
    *rayEpsilon = 1e-3f * *tHit;
    return true;
}
```

The implementation first computes the denominator from Equation (3.5). It finds the three mesh vertices that make up this particular `Triangle` and then computes the edge vectors and denominator. Note that if the denominator is zero, this triangle is degenerate and therefore cannot intersect a ray.

Cross() 62
DifferentialGeometry 102
Dot() 60
Ray 66
Triangle 139
Vector 57

⟨*Compute* $\mathbf{s}_1$⟩ ≡ 141
```
⟨Get triangle vertices in p1, p2, and p3 140⟩
Vector e1 = p2 - p1;
Vector e2 = p3 - p1;
Vector s1 = Cross(ray.d, e2);
float divisor = Dot(s1, e1);
```

```
if (divisor == 0.)
    return false;
float invDivisor = 1.f / divisor;
```

Now the routine can compute the barycentric coordinate b_1. Recall that barycentric coordinates that are less than zero or greater than one represent points outside the triangle, indicating no intersection.

⟨*Compute first barycentric coordinate*⟩ ≡ 141

```
Vector d = ray.o - p1;
float b1 = Dot(d, s1) * invDivisor;
if (b1 < 0. || b1 > 1.)
    return false;
```

The second barycentric coordinate, b_2, is computed in a similar way:

⟨*Compute second barycentric coordinate*⟩ ≡ 141

```
Vector s2 = Cross(d, e1);
float b2 = Dot(ray.d, s2) * invDivisor;
if (b2 < 0. || b1 + b2 > 1.)
    return false;
```

Now that we know the ray intersects the triangle, we compute the distance along the ray at which the intersection occurs. This gives a last opportunity to exit the procedure early, in the case where the t value falls outside the `Ray::mint` and `Ray::maxt` bounds.

⟨*Compute* `t` *to intersection point*⟩ ≡ 141

```
float t = Dot(e2, s2) * invDivisor;
if (t < ray.mint || t > ray.maxt)
    return false;
```

In order to generate consistent tangent vectors over triangle meshes, it is necessary to compute the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ using the parametric (u, v) values at the triangle vertices, if provided. Although the partial derivatives are the same at all points on the triangle, the implementation here recomputes them each time an intersection is found. Although this results in redundant computation, the storage savings for large triangle meshes can be significant.

The triangle is the set of points

$$\mathrm{p}_\mathrm{o} + u\frac{\partial \mathrm{p}}{\partial u} + v\frac{\partial \mathrm{p}}{\partial v},$$

for some p_o, where u and v range over the parametric coordinates of the triangle. We also know the three vertex positions p_i, $i = 1,\ 2,\ 3$, and the texture coordinates (u_i, v_i) at each vertex. From this it follows that the partial derivatives of p must satisfy

$$\mathrm{p}_i = \mathrm{p}_\mathrm{o} + u_i\frac{\partial \mathrm{p}}{\partial u} + v_i\frac{\partial \mathrm{p}}{\partial v}.$$

In other words, there is a unique affine mapping from the two-dimensional (u, v) space to points on the triangle (such a mapping exists even though the triangle is specified in 3D space because the triangle is planar). To compute expressions for $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$,

Cross() 62
Dot() 60
Ray::maxt 67
Ray::mint 67
Ray::o 67
Vector 57

we start by computing the differences $p_1 - p_3$ and $p_2 - p_3$, giving the matrix equation

$$\begin{pmatrix} u_1 - u_3 & v_1 - v_3 \\ u_2 - u_3 & v_2 - v_3 \end{pmatrix} \begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} p_1 - p_3 \\ p_2 - p_3 \end{pmatrix}.$$

Thus,

$$\begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} u_1 - u_3 & v_1 - v_3 \\ u_2 - u_3 & v_2 - v_3 \end{pmatrix}^{-1} \begin{pmatrix} p_1 - p_3 \\ p_2 - p_3 \end{pmatrix}.$$

Inverting a 2×2 matrix is straightforward. The inverse of the (u, v) differences matrix is

$$\frac{1}{(u_1 - u_3)(v_2 - v_3) - (v_1 - v_3)(u_2 - u_3)} \begin{pmatrix} v_2 - v_3 & -(v_1 - v_3) \\ -(u_2 - u_3) & u_1 - u_3 \end{pmatrix}.$$

⟨*Compute triangle partial derivatives*⟩ ≡ 141

```
Vector dpdu, dpdv;
float uvs[3][2];
GetUVs(uvs);
⟨Compute deltas for triangle partial derivatives 143⟩
float determinant = du1 * dv2 - dv1 * du2;
if (determinant == 0.f) {
    ⟨Handle zero determinant for triangle partial derivative matrix 143⟩
}
else {
    float invdet = 1.f / determinant;
    dpdu = ( dv2 * dp1 - dv1 * dp2) * invdet;
    dpdv = (-du2 * dp1 + du1 * dp2) * invdet;
}
```

⟨*Compute deltas for triangle partial derivatives*⟩ ≡ 143

```
float du1 = uvs[0][0] - uvs[2][0];
float du2 = uvs[1][0] - uvs[2][0];
float dv1 = uvs[0][1] - uvs[2][1];
float dv2 = uvs[1][1] - uvs[2][1];
Vector dp1 = p1 - p3, dp2 = p2 - p3;
```

Finally, it is necessary to handle the case when the matrix is singular and therefore cannot be inverted. Note that this only happens when the user-supplied per-vertex parameterization values are degenerate. In this case, the `Triangle` just chooses an arbitrary coordinate system about the triangle's surface normal, making sure that it is orthonormal:

⟨*Handle zero determinant for triangle partial derivative matrix*⟩ ≡ 143

```
CoordinateSystem(Normalize(Cross(e2, e1)), &dpdu, &dpdv);
```

CoordinateSystem() 63
Cross() 62
Triangle 139
Triangle::GetUVs() 144
Vector 57
Vector::Normalize() 63

To compute the (u, v) parametric coordinates at the hit point, the barycentric interpolation formula is applied to the (u, v) parametric coordinates at the vertices.

⟨*Interpolate* (u, v) *triangle parametric coordinates*⟩ ≡ 141

```
float b0 = 1 - b1 - b2;
float tu = b0*uvs[0][0] + b1*uvs[1][0] + b2*uvs[2][0];
float tv = b0*uvs[0][1] + b1*uvs[1][1] + b2*uvs[2][1];
```

The utility routine `GetUVs()` returns the (u, v) coordinates for the three vertices of the triangle, either from the `TriangleMesh`, if it has them, or returning default values if explicit (u, v) coordinates were not specified with the mesh.

⟨*Triangle Public Methods*⟩ +≡ 139

```
void GetUVs(float uv[3][2]) const {
    if (mesh->uvs) {
        uv[0][0] = mesh->uvs[2*v[0]];
        uv[0][1] = mesh->uvs[2*v[0]+1];
        uv[1][0] = mesh->uvs[2*v[1]];
        uv[1][1] = mesh->uvs[2*v[1]+1];
        uv[2][0] = mesh->uvs[2*v[2]];
        uv[2][1] = mesh->uvs[2*v[2]+1];
    }
    else {
        uv[0][0] = 0.; uv[0][1] = 0.;
        uv[1][0] = 1.; uv[1][1] = 0.;
        uv[2][0] = 1.; uv[2][1] = 1.;
    }
}
```

Before a successful intersection is reported, the intersection point is tested against an alpha mask texture, if one has been assigned to the shape. This texture can be thought of as a one-dimensional function over the triangle's surface, where at any point where its value is zero, the intersection is ignored, effectively treating that point on the triangle as not being present. (Chapter 10 defines the texture interface and implementations in more detail.) Alpha masks can be helpful for representing objects like leaves: a leaf can be modeled as a single triangle, with an alpha mask "cutting out" the edges so that a leaf shape remains. This functionality is much less useful for other primitives, so `pbrt` only supports it for triangles.

⟨*Test intersection against alpha texture, if present*⟩ ≡ 141

```
if (mesh->alphaTexture) {
    DifferentialGeometry dgLocal(ray(t), dpdu, dpdv,
                                 Normal(0,0,0), Normal(0,0,0),
                                 tu, tv, this);
    if (mesh->alphaTexture->Evaluate(dgLocal) == 0.f)
        return false;
}
```

Now we certainly have a hit and can update the values pointed to by the pointers passed to the intersection routine. We now have all the information we need to initialize the `DifferentialGeometry` structure for this intersection. In contrast to previous shapes, it's not necessary to transform the partial derivatives to world space, since the triangle's vertices were already transformed to world space. Like the disk, the partial derivatives of the triangle's normal are also both (0, 0, 0), since it is flat.

DifferentialGeometry 102
Normal 65
Texture::Evaluate() 520
TriangleMesh 135
TriangleMesh::
alphaTexture 137
TriangleMesh::uvs 137

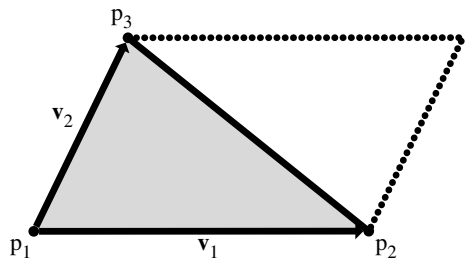

Figure 3.11: The area of a triangle with two edges given by vectors v_1 and v_2 is one-half of the area of the parallelogram. The parallelogram area is given by the length of the cross product of $\mathbf{v}_1$ and $\mathbf{v}_2$.

⟨*Fill in* `DifferentialGeometry` *from triangle hit*⟩ ≡ 141
```
*dg = DifferentialGeometry(ray(t), dpdu, dpdv,
                           Normal(0,0,0), Normal(0,0,0),
                           tu, tv, this);
```

3.6.3 SURFACE AREA

Recall from Section 2.2 that the area of a parallelogram is given by the length of the cross product of the two vectors along its sides. From this, it's easy to see that given the vectors for two edges of a triangle, its area is half of the area of the parallelogram given by those two vectors (Figure 3.11).

⟨*TriangleMesh Method Definitions*⟩ +≡
```
float Triangle::Area() const {
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    return 0.5f * Cross(p2-p1, p3-p1).Length();
}
```

3.6.4 SHADING GEOMETRY

If the `TriangleMesh` has per-vertex normals or tangent vectors, the `GetShadingGeometry()` method uses them to initialize the shading geometry appropriately:

⟨*TriangleMesh Method Definitions*⟩ +≡
```
void Triangle::GetShadingGeometry(const Transform &obj2world,
        const DifferentialGeometry &dg,
        DifferentialGeometry *dgShading) const {
    if (!mesh->n && !mesh->s) {
        *dgShading = dg;
        return;
    }
    ⟨Initialize Triangle shading geometry with n and s 146⟩
}
```

DifferentialGeometry 102
Normal 65
Transform 76
Triangle 139
Triangle::mesh 139
TriangleMesh 135
TriangleMesh::n 137
TriangleMesh::s 137

Because the DifferentialGeometry from a ray–triangle intersection stores parametric (u, v) coordinates but doesn't have the barycentric coordinates of the intersection point, the first task is to determine what the barycentric coordinates are.[5] These are then used to compute an interpolated normal vector and **s** vector, which are converted to two tangent vectors that give the same normal, since the DifferentialGeometry constructor takes two tangent vectors rather than the normal. Finally, the partial derivatives of the shading normal are found and the shading geometry structure can be initialized.

⟨*Initialize* Triangle *shading geometry with* n *and* s⟩ ≡ 145
```
    ⟨Compute barycentric coordinates for point 147⟩
    ⟨Use n and s to compute shading tangents for triangle, ss and ts 147⟩
    Normal dndu, dndv;
    ⟨Compute ∂n/∂u and ∂n/∂v for triangle shading geometry⟩
    *dgShading = DifferentialGeometry(dg.p, ss, ts,
        (*ObjectToWorld)(dndu), (*ObjectToWorld)(dndv),
        dg.u, dg.v, dg.shape);
```

Recall that the (u, v) parametric coordinates in the DifferentialGeometry for a triangle are computed with barycentric interpolation of parametric coordinates at the triangle vertices:

$$u = b_0u_0 + b_1u_1 + b_2u_2$$
$$v = b_0v_0 + b_1v_1 + b_2v_2,$$

where $b_0 = 1 - b_1 - b_2$. Here, the values of u, v, u_i, and v_i are all known, u and v from the DifferentialGeometry and u_i and v_i from the Triangle. We can substitute for the b_0 term and rewrite the above equations, giving a linear system in two unknowns b_1 and b_2:

$$\begin{pmatrix} u_1 - u_0 & u_2 - u_1 \\ v_1 - v_0 & v_2 - v_1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} u - u_0 \\ v - v_0 \end{pmatrix}.$$

This is a linear system of the basic form $\mathbf{AB} = \mathbf{C}$. We can solve for $\mathbf{B}$ by inverting $\mathbf{A}$, giving the two barycentric coordinates

$$\mathbf{B} = \mathbf{A}^{-1}\mathbf{C}.$$

The closed-form solution for linear systems of this form is implemented in the utility routine SolveLinearSystem2x2().

DifferentialGeometry 102
DifferentialGeometry::p 102
DifferentialGeometry::shape 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Normal 65
Shape::ObjectToWorld 108
SolveLinearSystem2x2() 1020
Triangle 139

5 An alternative system design could recognize the importance of triangle meshes as a rendering primitive and provide space for the barycentric coordinates in the DifferentialGeometry structure, with other shapes not setting those fields. More generally, a handful of floats that could be used for shape-specific information like this could be provided in DifferentialGeometry. Other parts of the system wouldn't access these fields, not knowing their semantics, but they could be accessed by the shape in the GetShadingGeometry() call when DifferentialGeometry that it had initialized was passed back to it.

⟨*Compute barycentric coordinates for point*⟩ ≡ 146
```
float b[3];
⟨Initialize A and C matrices for barycentrics 147⟩
if (!SolveLinearSystem2x2(A, C, &b[1], &b[2])) {
    ⟨Handle degenerate parametric mapping 147⟩
}
else
    b[0] = 1.f - b[1] - b[2];
```

⟨*Initialize* A *and* C *matrices for barycentrics*⟩ ≡ 147
```
float uv[3][2];
GetUVs(uv);
float A[2][2] =
    { { uv[1][0] - uv[0][0], uv[2][0] - uv[0][0] },
      { uv[1][1] - uv[0][1], uv[2][1] - uv[0][1] } };
float C[2] = { dg.u - uv[0][0], dg.v - uv[0][1] };
```

If the determinant of **A** is zero, the solution is undefined and `SolveLinearSystem2x2()` returns `false`. This case happens if all three triangle vertices had the same texture coordinates, for example. In this case, the barycentric coordinates are all set arbitrarily to $\frac{1}{3}$:

⟨*Handle degenerate parametric mapping*⟩ ≡ 147
```
b[0] = b[1] = b[2] = 1.f/3.f;
```

Given the barycentrics, it's straightforward to compute the shading normal and shading **s** tangent vector by interpolating among the appropriate vertex normals and tangents. First, the vector `ts` for the differential geometry constructor is found using the cross product of `ss` and `ns`, giving an orthogonal vector to the two of them. Next, `ss` is overwritten with the cross product of `ns` and `ts`; this ensures that the cross product of `ss` and `ts` gives `ns`.

Thus, if per-vertex **n** and **s** values are provided and if the interpolated **n** and **s** values aren't perfectly orthogonal, **n** will be preserved and **s** will be modified so that the coordinate system is orthogonal.

⟨*Use* n *and* s *to compute shading tangents for triangle,* ss *and* ts⟩ ≡ 146
```
Normal ns;
Vector ss, ts;
if (mesh->n) ns = Normalize(obj2world(b[0] * mesh->n[v[0]] +
                                      b[1] * mesh->n[v[1]] +
                                      b[2] * mesh->n[v[2]]));
else   ns = dg.nn;
if (mesh->s) ss = Normalize(obj2world(b[0] * mesh->s[v[0]] +
                                      b[1] * mesh->s[v[1]] +
                                      b[2] * mesh->s[v[2]]));
else   ss = Normalize(dg.dpdu);
```

Cross() 62
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Normal 65
SolveLinearSystem2x2() 1020
Vector 57
Vector::Normalize() 63

```
ts = Cross(ss, ns);
if (ts.LengthSquared() > 0.f) {
    ts = Normalize(ts);
    ss = Cross(ts, ns);
}
else
    CoordinateSystem((Vector)ns, &ss, &ts);
```

The code to compute the partial derivatives $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ is almost identical to the code to compute the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ on a triangle. Therefore, it has been elided from the text here.

⋆ 3.7 SUBDIVISION SURFACES

We will wrap up this chapter by defining a `Shape` that implements *subdivision surfaces*, a representation that is particularly well suited to describing complex smooth shapes. The subdivision surface for a particular mesh is defined by repeatedly subdividing the faces of the mesh into smaller faces and then finding the new vertex locations using weighted combinations of the old vertex positions.

For appropriately chosen subdivision rules, this process converges to give a smooth *limit surface* as the number of subdivision steps goes to infinity. In practice, just a few levels of subdivision typically suffice to give a good approximation of the limit surface. Figure 3.12 shows a simple example of a subdivision, where a tetrahedron has been subdivided zero, one, two, and six times. Figure 3.13 shows the effect of applying subdivision to the Killeroo model; on the top is the original control mesh, and below is the subdivision surface that the control mesh represents.

Although originally developed in the 1970s, subdivision surfaces have recently received a fair amount of attention in computer graphics thanks to some important advantages over

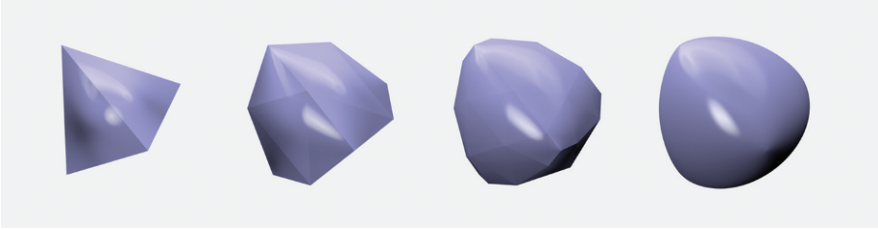

Figure 3.12: Subdivision of a Tetrahedron. From left to right, zero, one, two, and six subdivision steps have been used. (At zero levels, the vertices are just moved to lie on the limit surface.) As more subdivision is done, the mesh approaches the limit surface, the smooth surface described by the original mesh. Notice how the specular highlights become progressively more accurate and the silhouette edges appear smoother as more levels of subdivision are performed.

Shape 108

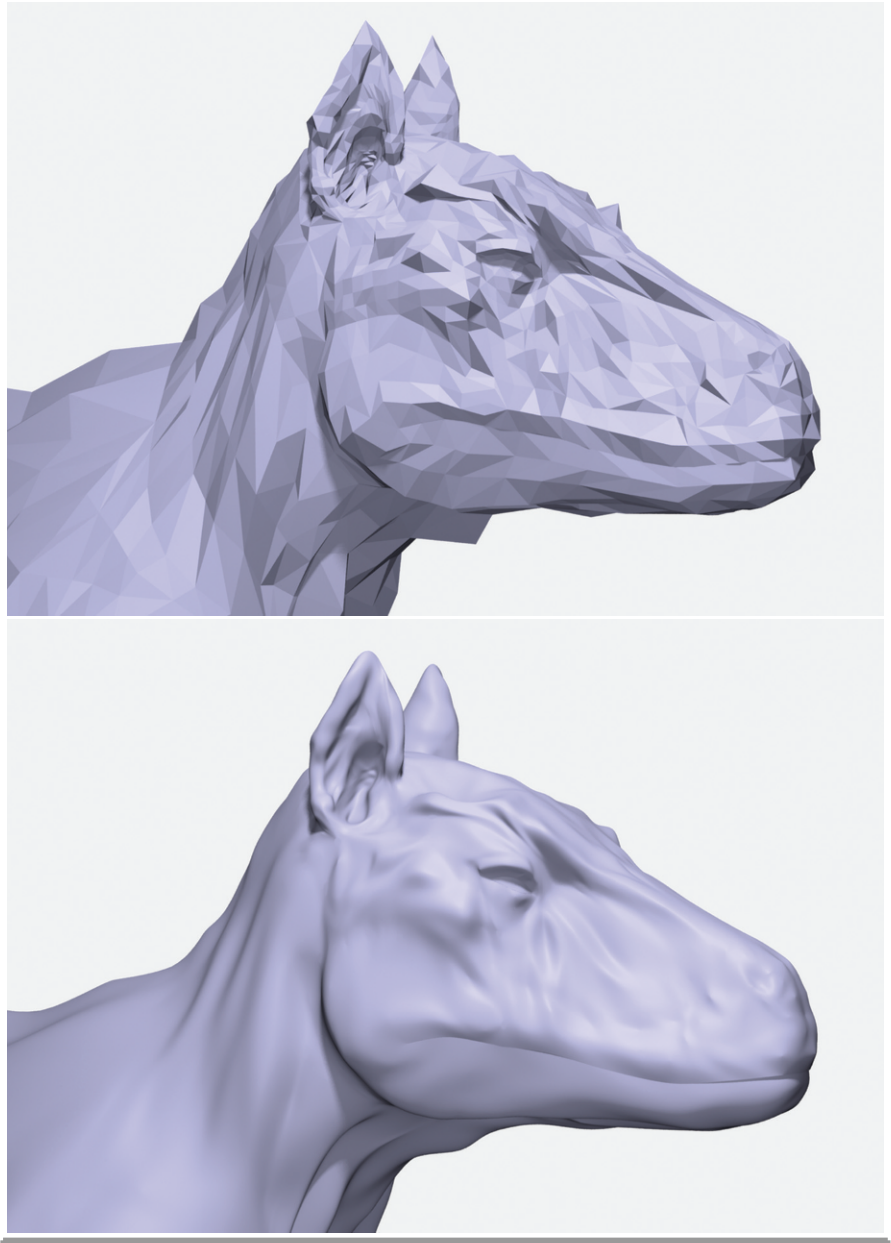

Figure 3.13: Subdivision Applied to the Killeroo Model. The control mesh (top) describes the subdivision surface shown below it. Subdivision is well suited to modeling shapes like this one, since it's easy to add detail locally by refining the control mesh, and there are no limitations on the topology of the final surface. *(Model courtesy of headus/Rezard.)*

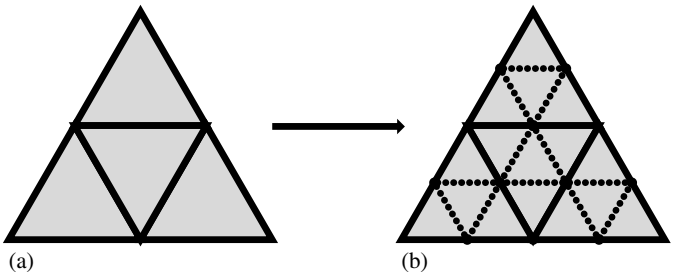

Figure 3.14: Basic Refinement Process for Loop Subdivision. (a) The control mesh before subdivision. (b) The new mesh after one subdivision step. Each triangular face of the mesh has been subdivided into four new faces by splitting each of the edges and connecting the new vertices with new edges.

polygonal and spline-based representations of surfaces. The advantages of subdivision include the following:

- Subdivision surfaces are smooth, as opposed to polygon meshes, which appear faceted when viewed close up, regardless of how finely they are modeled.
- A lot of existing infrastructure in modeling systems can be retargeted to subdivision. The classic toolbox of techniques for modeling polygon meshes can be applied to modeling subdivision control meshes.
- Subdivision surfaces are well suited to describing objects with complex topology, since they start with a control mesh of arbitrary (manifold) topology. Parametric surface models generally don't handle complex topology well.
- Subdivision methods are often generalizations of spline-based surface representations, so spline surfaces can often just be run through general subdivision surface renderers.
- It is easy to add detail to a localized region of a subdivision surface simply by adding faces to appropriate parts of the control mesh. This is much harder with spline representations.

Here, we will describe an implementation of *Loop subdivision surfaces*.[6] The Loop subdivision rules are based on triangular faces in the control mesh; faces with more than three vertices are triangulated at the start. At each subdivision step, all faces split into four child faces (Figure 3.14). New vertices are added along all of the edges of the original mesh, with positions computed using weighted averages of nearby vertices. Furthermore, the position of each original vertex is updated with a weighted average of its position and its new neighbors' positions. The implementation here uses weights based on improvements to Loop's method developed by Hoppe et al. (1994). We will not include discussion here about how these weights are derived. They must be chosen carefully to ensure that the limit surface actually has particular desirable smoothness properties, although subtle mathematics are necessary to prove that they indeed do this.

6 Named after the inventor of the subdivision rules used, Charles Loop.

3.7.1 MESH REPRESENTATION

⟨*LoopSubdiv Declarations*⟩ ≡
```
class LoopSubdiv : public Shape {
public:
    ⟨LoopSubdiv Public Methods⟩
private:
    ⟨LoopSubdiv Private Methods 163⟩
    ⟨LoopSubdiv Private Data 152⟩
};
```

We will start by describing the data structures used to represent the subdivision mesh. These data structures need to be carefully designed in order to support all of the operations necessary to cleanly implement the subdivision algorithm. The parameters to the `LoopSubdiv` constructor specify a triangle mesh in exactly the same format used in the `TriangleMesh` constructor (Section 3.6): each face is described by three integer vertex indices, giving offsets into the vertex array `P` for the face's three vertices. We will need to process this data to determine which faces are adjacent to each other, which faces are adjacent to which vertices, and so on.

⟨*LoopSubdiv Method Definitions*⟩ ≡
```
LoopSubdiv::LoopSubdiv(const Transform *o2w, const Transform *w2o,
                       bool ro, int nfaces, int nvertices,
                       const int *vertexIndices, const Point *P, int nl)
    : Shape(o2w, w2o, ro) {
    nLevels = nl;
    ⟨Allocate LoopSubdiv vertices and faces 151⟩
    ⟨Set face to vertex pointers 154⟩
    ⟨Set neighbor pointers in faces 156⟩
    ⟨Finish vertex initialization 158⟩
}
```

We will shortly define `SDVertex` and `SDFace` structures, which hold data for vertices and faces in the subdivision mesh. The constructor starts by allocating one instance of the `SDVertex` class for each vertex in the mesh and an `SDFace` for each face. For now, these are mostly uninitialized.

⟨*Allocate* `LoopSubdiv` *vertices and faces*⟩ ≡ 151
```
int i;
SDVertex *verts = new SDVertex[nvertices];
for (i = 0; i < nvertices; ++i) {
    verts[i] = SDVertex(P[i]);
    vertices.push_back(&verts[i]);
}
SDFace *fs = new SDFace[nfaces];
for (i = 0; i < nfaces; ++i)
    faces.push_back(&fs[i]);
```

LoopSubdiv 151
LoopSubdiv::faces 152
LoopSubdiv::vertices 152
Point 63
SDFace 153
SDVertex 152
Shape 108
Transform 76
TriangleMesh 135

The LoopSubdiv destructor, which we won't include here, deletes all of the faces and vertices allocated above.

⟨*LoopSubdiv Private Data*⟩ ≡ 151
```
int nLevels;
vector<SDVertex *> vertices;
vector<SDFace *> faces;
```

The Loop subdivision scheme, like most other subdivision schemes, assumes that the control mesh is *manifold*—no more than two faces share any given edge. Such a mesh may be closed or open: A *closed mesh* has no boundary, and all faces have adjacent faces across each of their edges. An *open mesh* has some faces that do not have all three neighbors. The LoopSubdiv implementation here supports both closed and open meshes.

In the interior of a triangle mesh, most vertices are adjacent to six faces and have six neighbor vertices directly connected to them with edges. On the boundaries of an open mesh, most vertices are adjacent to three faces and four vertices. The number of vertices directly adjacent to a vertex is called the vertex's *valence*. Interior vertices with valence other than six, or boundary vertices with valence other than four, are called *extraordinary vertices*; otherwise, they are called *regular*.[7] Loop subdivision surfaces are smooth everywhere except at their extraordinary vertices.

Each SDVertex stores its position P, a Boolean that indicates whether it is a regular or extraordinary vertex, and a Boolean that records if it lies on the boundary of the mesh. It also holds a pointer to an arbitrary face adjacent to it; this pointer gives a starting point for finding all of the adjacent faces. Finally, there is a pointer to store the corresponding SDVertex for the next level of subdivision, if any.

⟨*LoopSubdiv Local Structures*⟩ ≡
```
struct SDVertex {
    ⟨SDVertex Constructor 152⟩
    ⟨SDVertex Methods⟩
    Point P;
    SDFace *startFace;
    SDVertex *child;
    bool regular, boundary;
};
```

The constructor for SDVertex does the obvious initialization. Note that the value of SDVertex::startFace is initially set to NULL.

⟨*SDVertex Constructor*⟩ ≡ 152
```
SDVertex(Point pt = Point(0,0,0))
    : P(pt), startFace(NULL), child(NULL),
      regular(false), boundary(false) { }
```

LoopSubdiv 151
Point 63
SDFace 153
SDVertex 152
SDVertex::startFace 152

7 These terms are commonly used in the modeling literature, although "irregular" versus "regular" or "extraordinary" versus "ordinary" might be more intuitive.

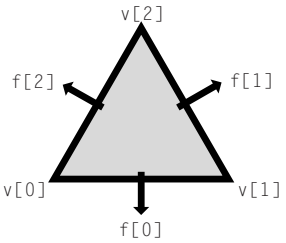

Figure 3.15: Each triangular face stores three pointers to `SDVertex` objects `v[i]` and three pointers to neighboring faces `f[i]`. Neighboring faces are indexed using the convention that the `i`th edge is the edge from `v[i]` to `v[(i+1)%3]`, and the neighbor across the `i`th edge is in `f[i]`.

The `SDFace` structure is where most of the topological information about the mesh is maintained. Because all faces are triangular, faces always store pointers to their three vertices and pointers to the adjacent faces across its three edges. The corresponding face neighbor pointers will be `NULL` if the face is on the boundary of an open mesh.

The face neighbor pointers are indexed such that if we label the edge from `v[i]` to `v[(i+1)%3]` as the `i`th edge, then the neighbor face across that edge is stored in `f[i]` (Figure 3.15). This labeling convention is important to keep in mind. Later when we are updating the topology of a newly subdivided mesh, we will make extensive use of it to navigate around the mesh. Similarly to the `SDVertex` class, the `SDFace` also stores pointers to child faces at the next level of subdivision.

⟨*LoopSubdiv Local Structures*⟩ +≡
```
struct SDFace {
    ⟨SDFace Constructor⟩
    ⟨SDFace Methods 159⟩
    SDVertex *v[3];
    SDFace *f[3];
    SDFace *children[4];
};
```

The `SDFace` constructor is straightforward—it simply sets these various pointers to `NULL`—so it is not shown here.

In order to simplify navigation of the `SDFace` data structure, we'll provide macros that make it easy to determine the vertex and face indices before or after a particular index. These macros add appropriate offsets and compute the result modulus three to handle cycling around. To compute the previous index, they add 2 instead of subtracting 1, which avoids possibly taking the modulus of a negative number, the result of which is implementation dependent in C++.

SDFace 153
SDVertex 152

⟨*LoopSubdiv Macros*⟩ ≡
```
#define NEXT(i) (((i)+1)%3)
#define PREV(i) (((i)+2)%3)
```

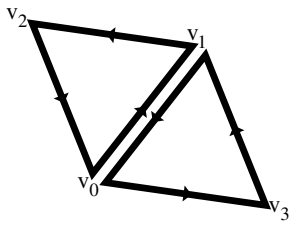

Figure 3.16: All of the faces in the input mesh must be specified so that each shared edge is given no more than once in each direction. Here, the edge from v_0 to v_1 is traversed from v_0 to v_1 by one face and from v_1 to v_0 by the other. Another way to think of this is in terms of face orientation: all faces' vertices should be given consistently in either clockwise or counterclockwise order, as seen from outside the mesh.

In addition to requiring a manifold mesh, the `LoopSubdiv` class expects that the control mesh specified by the user will be *consistently ordered*—each *directed edge* in the mesh can be present only once. An edge that is shared by two faces should be specified in a different direction by each face. Consider two vertices, v_0 and v_1, with an edge between them. We expect that one of the triangular faces that has this edge will specify its three vertices so that v_0 is before v_1, and that the other face will specify its vertices so that v_1 is before v_0 (Figure 3.16). A Möbius strip is one example of a surface that cannot be consistently ordered, but such surfaces come up rarely in rendering so in practice this restriction is not a problem. Poorly formed mesh data from other programs that don't create consistently ordered meshes can be troublesome, however.

Given this assumption about the input data, the constructor can now initialize this mesh's topological data structures. It first loops over all of the faces and sets their `v` pointers to point to their three vertices. It also sets each vertex's `SDVertex::startFace` pointer to point to one of the vertex's neighboring faces. It doesn't matter which of its adjacent faces is used, so the implementation just keeps resetting it each time it comes across another face that the vertex is incident to, thus ensuring that all vertices have some non-NULL face pointer by the time the loop is complete.

⟨*Set face to vertex pointers*⟩ ≡ 151

```
const int *vp = vertexIndices;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int j = 0; j < 3; ++j) {
        SDVertex *v = vertices[vp[j]];
        f->v[j] = v;
        v->startFace = f;
    }
    vp += 3;
}
```

LoopSubdiv 151
SDFace 153
SDFace::v 153
SDVertex 152
SDVertex::startFace 152

Now it is necessary to set each face's `f` pointer to point to its neighboring faces. This is a bit trickier, since face adjacency information isn't directly specified by the user. The constructor loops over the faces and creates an `SDEdge` object for each of their three edges. When it comes to another face that shares the same edge, it can update both faces' neighbor pointers.

```
⟨LoopSubdiv Local Structures⟩ +≡
    struct SDEdge {
        ⟨SDEdge Constructor 155⟩
        ⟨SDEdge Comparison Function 155⟩
        SDVertex *v[2];
        SDFace *f[2];
        int f0edgeNum;
    };
```

The `SDEdge` constructor takes pointers to the two vertices at each end of the edge. It orders them so that `v[0]` holds the one that is first in memory. This code may seem strange, but it is simply relying on the fact that pointers in C++ are effectively numbers that can be manipulated like integers,[8] and that the ordering of vertices on an edge is arbitrary. Sorting vertices on the address of the pointer guarantees that the edge (v_a, v_b) is correctly recognized as the same as the edge (v_b, v_a), regardless of what order the vertices are given in.

```
⟨SDEdge Constructor⟩ ≡ 155
    SDEdge(SDVertex *v0 = NULL, SDVertex *v1 = NULL) {
        v[0] = min(v0, v1);
        v[1] = max(v0, v1);
        f[0] = f[1] = NULL;
        f0edgeNum = -1;
    }
```

The class also defines an ordering operation for `SDEdge` objects so that they can be stored in other data structures that rely on ordering being well defined.

```
⟨SDEdge Comparison Function⟩ ≡ 155
    bool operator<(const SDEdge &e2) const {
        if (v[0] == e2.v[0]) return v[1] < e2.v[1];
        return v[0] < e2.v[0];
    }
```

Now the constructor can get to work, looping over the edges in all of the faces and updating the neighbor pointers as it goes. It uses a `set` to store the edges that have only one adjacent face so far. The `set` makes it possible to search for a particular edge in $O(\log n)$ time.

SDEdge 155
SDEdge::v 155
SDFace 153
SDVertex 152

8 Segmented architectures notwithstanding.

⟨*Set neighbor pointers in* faces⟩ ≡ 151
```
set<SDEdge> edges;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int edgeNum = 0; edgeNum < 3; ++edgeNum) {
        ⟨Update neighbor pointer for edgeNum 156⟩
    }
}
```

For each edge in each face, the loop body creates an edge object and sees if the same edge has been seen previously. If so, it initializes both faces' neighbor pointers across the edge. If not, it adds the edge to the set of edges. The indices of the two vertices at the ends of the edge, `v0` and `v1`, are equal to the edge index and the edge index plus one.

⟨*Update neighbor pointer for* edgeNum⟩ ≡ 156
```
int v0 = edgeNum, v1 = NEXT(edgeNum);
SDEdge e(f->v[v0], f->v[v1]);
if (edges.find(e) == edges.end()) {
    ⟨Handle new edge 156⟩
}
else {
    ⟨Handle previously seen edge 156⟩
}
```

Given an edge that hasn't been encountered before, the current face's pointer is stored in the edge object's `f[0]` member. Because the input mesh is assumed to be manifold, there can be at most one other face that shares this edge. When such a face is discovered, it can be used to initialize the neighboring face field. Storing the edge number of this edge in the current face allows the neighboring face to initialize its corresponding edge neighbor pointer.

⟨*Handle new edge*⟩ ≡ 156
```
e.f[0] = f;
e.f0edgeNum = edgeNum;
edges.insert(e);
```

When the second face on an edge is found, the neighbor pointers for each of the two faces are set. The edge is then removed from the edge set, since no edge can be shared by more than two faces.

⟨*Handle previously seen edge*⟩ ≡ 156
```
e = *edges.find(e);
e.f[0]->f[e.f0edgeNum] = f;
f->f[edgeNum] = e.f[0];
edges.erase(e);
```

Now that all faces have proper neighbor pointers, the `boundary` and `regular` flags in each of the vertices can be set. In order to determine if a vertex is a boundary vertex, we'll

NEXT() 153
SDEdge 155
SDEdge::f 155
SDEdge::f0edgeNum 155
SDFace 153
SDFace::f 153

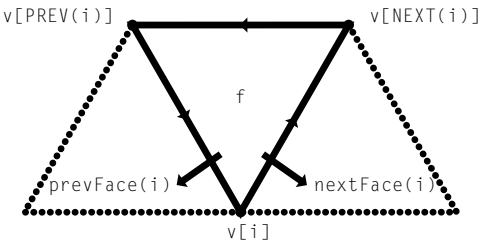

Figure 3.17: Given a vertex `v[i]` and a face that it is incident to, `f`, we define the *next face* as the face adjacent to `f` across the edge from `v[i]` to `v[NEXT(i)]`. The previous face is defined analogously.

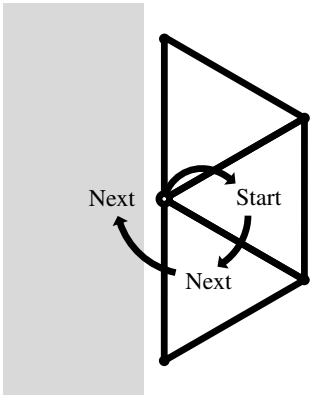

Figure 3.18: We can determine if a vertex is a boundary vertex by starting from the adjacent face `startFace` and following next face pointers around the vertex. If we come to a face that has no next neighbor face, then the vertex is on a boundary. If we return to `startFace`, it's an interior vertex.

define an ordering of faces around a vertex (Figure 3.17). For a vertex `v[i]` on a face `f`, we define the vertex's *next face* as the face across the edge from `v[i]` to `v[NEXT(i)]` and the *previous face* as the face across the edge from `v[PREV(i)]` to `v[i]`.

By successively going to the next face around `v`, we can iterate over the faces adjacent to it. If we eventually return to the face we started at, then we are at an interior vertex; if we come to an edge with a `NULL` neighbor pointer, then we're at a boundary vertex (Figure 3.18). Once the initialization routine has determined if this is a boundary vertex, it computes the valence of the vertex and sets the `regular` flag if the valence is 6 for an interior vertex or 4 for a boundary vertex; otherwise, it is an extraordinary vertex.

⟨*Finish vertex initialization*⟩ ≡ 151
```
for (i = 0; i < nvertices; ++i) {
    SDVertex *v = vertices[i];
    SDFace *f = v->startFace;
    do {
        f = f->nextFace(v);
    } while (f && f != v->startFace);
    v->boundary = (f == NULL);
    if (!v->boundary && v->valence() == 6)
        v->regular = true;
    else if (v->boundary && v->valence() == 4)
        v->regular = true;
    else
        v->regular = false;
}
```

Because the valence of a vertex is frequently needed, we provide the method `SDVertex::valence()`.

⟨*LoopSubdiv Inline Functions*⟩ ≡
```
inline int SDVertex::valence() {
    SDFace *f = startFace;
    if (!boundary) {
        ⟨Compute valence of interior vertex 158⟩
    }
    else {
        ⟨Compute valence of boundary vertex 159⟩
    }
}
```

To compute the valence of a nonboundary vertex, this method counts the number of the adjacent faces starting by following each face's neighbor pointers around the vertex until it reaches the starting face. The valence is equal to the number of faces visited.

⟨*Compute valence of interior vertex*⟩ ≡ 158
```
int nf = 1;
while ((f = f->nextFace(this)) != startFace)
    ++nf;
return nf;
```

For boundary vertices we can use the same approach, although in this case, the valence is one more than the number of adjacent faces. The loop over adjacent faces is slightly more complicated here: it follows pointers to the next face around the vertex until it reaches the boundary, counting the number of faces seen. It then starts again at `startFace` and follows previous face pointers until it encounters the boundary in the other direction.

SDFace 153
SDFace::nextFace() 159
SDVertex 152
SDVertex::boundary 152
SDVertex::regular 152
SDVertex::startFace 152
SDVertex::valence() 158

⟨*Compute valence of boundary vertex*⟩ ≡ 158
```
int nf = 1;
while ((f = f->nextFace(this)) != NULL)
    ++nf;
f = startFace;
while ((f = f->prevFace(this)) != NULL)
    ++nf;
return nf+1;
```

`SDFace::vnum()` is a utility function that finds the index of a given vertex pointer. It is a fatal error to pass a pointer to a vertex that isn't part of the current face—this case would represent a bug elsewhere in the subdivision code.

⟨*SDFace Methods*⟩ ≡ 153
```
int vnum(SDVertex *vert) const {
    for (int i = 0; i < 3; ++i)
        if (v[i] == vert) return i;
    Severe("Basic logic error in SDFace::vnum()");
    return -1;
}
```

Since the next face for a vertex `v[i]` on a face `f` is over the `i`th edge (recall the mapping of edge neighbor pointers from Figure 3.15), we can find the appropriate face neighbor pointer easily given the index `i` for the vertex, which the `vnum()` utility function provides. The previous face is across the edge from `PREV(i)` to `i`, so the method returns `f[PREV(i)]` for the previous face.

⟨*SDFace Methods*⟩ +≡ 153
```
SDFace *nextFace(SDVertex *vert) {
    return f[vnum(vert)];
}
```

⟨*SDFace Methods*⟩ +≡ 153
```
SDFace *prevFace(SDVertex *vert) {
    return f[PREV(vnum(vert))];
}
```

It is also useful to be able to get the next and previous vertices around a face starting at any vertex. The `SDFace::nextVert()` and `SDFace::prevVert()` methods do just that (Figure 3.19).

⟨*SDFace Methods*⟩ +≡ 153
```
SDVertex *nextVert(SDVertex *vert) {
    return v[NEXT(vnum(vert))];
}
```

⟨*SDFace Methods*⟩ +≡ 153
```
SDVertex *prevVert(SDVertex *vert) {
    return v[PREV(vnum(vert))];
}
```

NEXT() 153
PREV() 153
SDFace 153
SDFace::f 153
SDFace::nextFace() 159
SDFace::nextVert() 159
SDFace::prevFace() 159
SDFace::prevVert() 159
SDFace::v 153
SDFace::vnum() 159
SDVertex 152
Severe() 1005

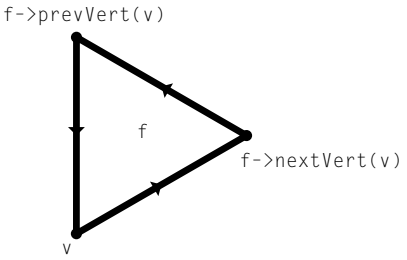

Figure 3.19: Given a vertex `v` on a face `f`, the method `f->prevVert(v)` returns the previous vertex around the face from `v`, and `f->nextVert(v)` returns the next vertex, where "next" and "previous" are defined by the original ordering of vertices when this face was defined.

3.7.2 BOUNDS

Loop subdivision surfaces have the *convex hull property*: the limit surface is guaranteed to be inside the convex hull of the original control mesh. Thus, for the bounding methods, we can just bound the original control vertices. The bounding methods are essentially equivalent to those in `TriangleMesh`, so we won't include them here.

3.7.3 SUBDIVISON

Now we can show how subdivision proceeds with the modified Loop rules. The `LoopSubdiv` shape doesn't support intersection directly, but applies subdivision a fixed number of times to generate a `TriangleMesh` for rendering. Exercise 3.10 at the end of the chapter discusses adaptive subdivision, where each original face is subdivided enough times so that the result looks smooth from a particular viewpoint rather than just using a fixed number of levels of subdivision, which may over-subdivide some areas while simultaneously under-subdividing others.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
bool LoopSubdiv::CanIntersect() const {
    return false;
}
```

The `Refine()` method handles all of the subdivision. It repeatedly applies the subdivision rules to the mesh, each time generating a new mesh to be used as the input to the next step. After each subdivision step, the `f` and `v` arrays in the `Refine()` method are updated to point to the faces and vertices from the level of subdivision just computed. When the `LoopSubdiv` is done subdividing, it creates a `TriangleMesh` representation of the surface that it returns to the caller. The `MemoryArena` class, defined in Section A.5.4 in Appendix A, provides a custom memory allocation method that quickly allocates memory, automatically freeing the memory when it goes out of scope.

LoopSubdiv 151
MemoryArena 1015
TriangleMesh 135

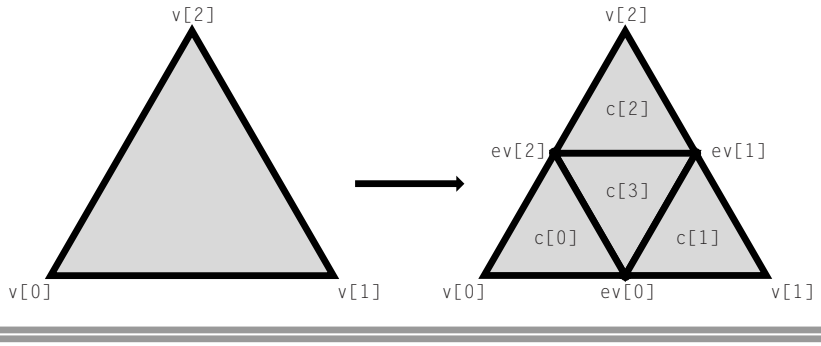

Figure 3.20: Basic Subdivision of a Single Triangular Face. Four child faces are created, ordered such that the `i`th child face is adjacent to the `i`th vertex of the original face and the fourth child face is in the center of the subdivided face. Three edge vertices need to be computed; they are numbered so that the `i`th edge vertex is along the `i`th edge of the original face.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
void LoopSubdiv::Refine(vector<Reference<Shape> > &refined) const {
    vector<SDFace *> f = faces;
    vector<SDVertex *> v = vertices;
    MemoryArena arena;
    for (int i = 0; i < nLevels; ++i) {
        ⟨Update f and v for next level of subdivision 161⟩
    }
    ⟨Push vertices to limit surface 171⟩
    ⟨Compute vertex tangents on limit surface 171⟩
    ⟨Create TriangleMesh from subdivision mesh⟩
}
```

The main loop of a subdivision step proceeds as follows: it creates `vectors` for all of the vertices and faces at the current level of subdivision and then proceeds to compute new vertex positions and update the topological representation for the refined mesh. Figure 3.20 shows the basic refinement rules for faces in the mesh. Each face is split into four child faces, such that the `i`th child face is next to the `i`th vertex of the input face and the final face is in the center. Three new vertices are then computed along the split edges of the original face.

LoopSubdiv 151
LoopSubdiv::faces 152
LoopSubdiv::nLevels 152
LoopSubdiv::vertices 152
MemoryArena 1015
Reference 1011
SDFace 153
SDVertex 152
Shape 108

⟨*Update* `f` *and* `v` *for next level of subdivision*⟩ ≡ 161
```
vector<SDFace *> newFaces;
vector<SDVertex *> newVertices;
⟨Allocate next level of children in mesh tree 162⟩
⟨Update vertex positions and create new edge vertices 162⟩
⟨Update new mesh topology 169⟩
⟨Prepare for next level of subdivision 170⟩
```

First, storage is allocated for the updated values of the vertices already present in the input mesh. The method also allocates storage for the child faces. It doesn't yet do any

initialization of the new vertices and faces other than setting the `regular` and `boundary` flags for the vertices since subdivision leaves boundary vertices on the boundary and interior vertices in the interior and it doesn't change the valence of vertices in the mesh.

⟨*Allocate next level of children in mesh tree*⟩ ≡ 161
```
for (uint32_t j = 0; j < v.size(); ++j) {
    v[j]->child = arena.Alloc<SDVertex>();
    v[j]->child->regular = v[j]->regular;
    v[j]->child->boundary = v[j]->boundary;
    newVertices.push_back(v[j]->child);
}
for (uint32_t j = 0; j < f.size(); ++j)
    for (int k = 0; k < 4; ++k) {
        f[j]->children[k] = arena.Alloc<SDFace>();
        newFaces.push_back(f[j]->children[k]);
    }
```

Computing New Vertex Positions

Before worrying about initializing the topology of the subdivided mesh, the refinement method computes positions for all of the vertices in the mesh. First, it considers the problem of computing updated positions for all of the vertices that were already present in the mesh; these vertices are called *even vertices*. It then computes the new vertices on the split edges. These are called *odd vertices.*

⟨*Update vertex positions and create new edge vertices*⟩ ≡ 161
```
⟨Update vertex positions for even vertices 162⟩
⟨Compute new odd edge vertices 166⟩
```

Different techniques are used to compute the updated positions for each of the different types of even vertices—regular and extraordinary, boundary and interior. This gives four cases to handle.

⟨*Update vertex positions for even vertices*⟩ ≡ 162
```
for (uint32_t j = 0; j < v.size(); ++j) {
    if (!v[j]->boundary) {
        ⟨Apply one-ring rule for even vertex 163⟩
    }
    else {
        ⟨Apply boundary rule for even vertex 165⟩
    }
}
```

For both types of interior vertices, we take the set of vertices adjacent to each vertex (called the *one-ring* around it, reflecting the fact that it's a ring of neighbors) and weight each of the neighbor vertices by a weight β (Figure 3.21). The vertex we are updating, in the center, is weighted by $1 - n\beta$, where n is the valence of the vertex. Thus, the new position v' for a vertex v is

$$\mathrm{v}' = (1 - n\beta)\mathrm{v} + \sum_{i=1}^{N} \beta \mathrm{v}_i.$$

MemoryArena::Alloc() 1016
SDFace 153
SDFace::children 153
SDVertex 152
SDVertex::boundary 152
SDVertex::regular 152

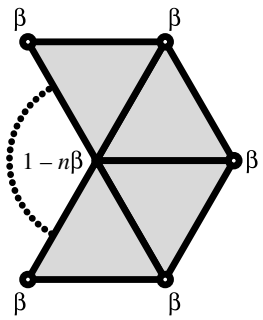

Figure 3.21: The new position v′ for a vertex v is computed by weighting the adjacent vertices v_i by a weight β and weighting v by $(1 - n\beta)$, where n is the valence of v. The adjacent vertices v_i are collectively referred to as the *one-ring* around v.

This formulation ensures that the sum of weights is one, which guarantees the convex hull property that was used earlier for bounding the surface. The position of the vertex being updated is only affected by vertices that are nearby; this is known as *local support*. Loop subdivision is particularly efficient because its subdivision rules all have this property.

The specific weight β used for this step is a key component of the subdivision method and must be chosen carefully in order to ensure smoothness of the limit surface, among other desirable properties.[9] The `LoopSubdiv::beta()` method that follows computes a β value based on the vertex's valence that ensures smoothness. For regular interior vertices, `LoopSubdiv::beta()` returns $\frac{1}{16}$. Since this is a common case, the implementation uses $\frac{1}{16}$ directly instead of calling `LoopSubdiv::beta()` every time.

⟨*Apply one-ring rule for even vertex*⟩ ≡ 162
```
if (v[j]->regular)
    v[j]->child->P = weightOneRing(v[j], 1.f/16.f);
else
    v[j]->child->P = weightOneRing(v[j], beta(v[j]->valence()));
```

⟨*LoopSubdiv Private Methods*⟩ ≡ 151
```
static float beta(int valence) {
    if (valence == 3) return 3.f/16.f;
    else return 3.f / (8.f * valence);
}
```

LoopSubdiv::beta() 171
LoopSubdiv::weightOneRing() 164
SDVertex::child 152
SDVertex::P 152
SDVertex::regular 152
SDVertex::valence() 158

9 Again, see the papers cited at the start of this section and in the "Further Reading" section for information about how values like β are derived.

The `LoopSubdiv::weightOneRing()` function loops over the one-ring of adjacent vertices and applies the given weight to compute a new vertex position. It uses the `SDVertex::oneRing()` function, defined in the following, which returns the positions of the vertices around the vertex `vert`.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
Point LoopSubdiv::weightOneRing(SDVertex *vert, float beta) {
    ⟨Put vert one-ring in Pring 164⟩
    Point P = (1 - valence * beta) * vert->P;
    for (int i = 0; i < valence; ++i)
        P += beta * Pring[i];
    return P;
}
```

Because a variable number of vertices are in the one-rings, we use the `ALLOCA()` macro to efficiently allocate space to store their positions.

⟨*Put* `vert` *one-ring in* `Pring`⟩ ≡ 164, 166
```
int valence = vert->valence();
Point *Pring = ALLOCA(Point, valence);
vert->oneRing(Pring);
```

The `oneRing()` method assumes that the pointer passed in points to an area of memory large enough to hold the one-ring around the vertex.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
void SDVertex::oneRing(Point *P) {
    if (!boundary) {
        ⟨Get one-ring vertices for interior vertex 164⟩
    }
    else {
        ⟨Get one-ring vertices for boundary vertex 165⟩
    }
}
```

It's relatively easy to get the one-ring around an interior vertex by looping over the faces adjacent to the vertex and for each face retaining the vertex after the center vertex. (Brief sketching with pencil and paper should convince you that this process returns all of the vertices in the one-ring.)

⟨*Get one-ring vertices for interior vertex*⟩ ≡ 164
```
SDFace *face = startFace;
do {
    *P++ = face->nextVert(this)->P;
    face = face->nextFace(this);
} while (face != startFace);
```

The one-ring around a boundary vertex is a bit more tricky. The implementation here carefully stores the one-ring in the given `Point` array so that the first and last entries in the array are the two adjacent vertices along the boundary. This ordering is important

ALLOCA() 1009
LoopSubdiv 151
LoopSubdiv::weightOneRing() 164
Point 63
SDFace 153
SDFace::nextFace() 159
SDFace::nextVert() 159
SDVertex 152
SDVertex::boundary 152
SDVertex::oneRing() 164
SDVertex::P 152
SDVertex::startFace 152
SDVertex::valence() 158

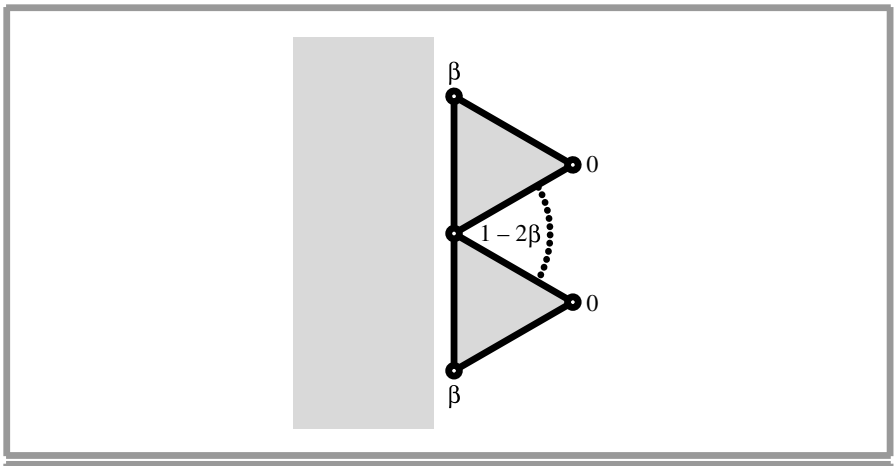

Figure 3.22: Subdivision on a Boundary Edge. The new position for the vertex in the center is computed by weighting it and its two neighbor vertices by the weights shown.

because the adjacent boundary vertices will often be weighted differently than the adjacent vertices that are in the interior of the mesh. Doing so requires that we first loop around neighbor faces until we reach a face on the boundary and then loop around the other way, storing vertices one by one.

⟨*Get one-ring vertices for boundary vertex*⟩ ≡ 164
```
SDFace *face = startFace, *f2;
while ((f2 = face->nextFace(this)) != NULL)
    face = f2;
*P++ = face->nextVert(this)->P;
do {
    *P++ = face->prevVert(this)->P;
    face = face->prevFace(this);
} while (face != NULL);
```

For vertices on the boundary, the new vertex's position is based only on the two neighboring boundary vertices (Figure 3.22). Not depending on interior vertices ensures that two abutting surfaces that share the same vertices on the boundary will have abutting limit surfaces. The `weightBoundary()` utility function applies the given weighting on the two neighbor vertices v_1 and v_2 to compute the new position v' as

$$v' = (1 - 2\beta)v + \beta v_1 + \beta v_2.$$

The same weight of $\frac{1}{8}$ is used for both regular and extraordinary vertices.

⟨*Apply boundary rule for even vertex*⟩ ≡ 162
```
v[j]->child->P = weightBoundary(v[j], 1.f/8.f);
```

The `weightBoundary()` utility function applies the given weights at a boundary vertex. Because the `SDVertex::oneRing()` function orders the boundary vertex's one-ring such

LoopSubdiv::
weightBoundary() 166
SDFace 153
SDFace::nextFace() 159
SDFace::nextVert() 159
SDFace::prevFace() 159
SDFace::prevVert() 159
SDVertex::child 152
SDVertex::oneRing() 164
SDVertex::P 152
SDVertex::startFace 152

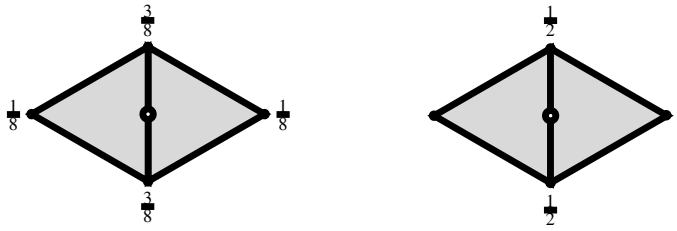

Figure 3.23: Subdivision Rule for Edge Split. The position of the new odd vertex, marked with an open circle, is found by weighting the two vertices at the ends of the edge and the two vertices opposite it on the adjacent triangles. (a) The weights for an interior vertex; (b) the weights for a boundary vertex.

that the first and last entries are the boundary neighbors, the implementation here is particularly straightforward.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
Point LoopSubdiv::weightBoundary(SDVertex *vert, float beta) {
    ⟨Put vert one-ring in Pring 164⟩
    Point P = (1-2*beta) * vert->P;
    P += beta * Pring[0];
    P += beta * Pring[valence-1];
    return P;
}
```

Now the refinement method computes the positions of the odd vertices—the new vertices along the split edges of the mesh. It loops over each edge of each face in the mesh, computing the new vertex that splits the edge (Figure 3.23). For interior edges, the new vertex is found by weighting the two vertices at the ends of the edge and the two vertices across from the edge on the adjacent faces. It loops through all three edges of each face, and each time it comes to an edge that hasn't been seen before it computes and stores the new odd vertex for the edge in the `edgeVerts` associative array.

⟨*Compute new odd edge vertices*⟩ ≡ 162
```
map<SDEdge, SDVertex *> edgeVerts;
for (uint32_t j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Compute odd vertex on kth edge 167⟩
    }
}
```

LoopSubdiv 151
Point 63
SDEdge 155
SDFace 153
SDVertex 152
SDVertex::P 152

As was done when setting the face neighbor pointers in the original mesh, an `SDEdge` object is created for the edge and checked to see if it is in the set of edges that have already

been visited. If it isn't, the new vertex on this edge is computed and added to the `map`, which is an associative array structure that performs efficient lookups.

⟨*Compute odd vertex on* `k`*th edge*⟩ ≡ 166
```
SDEdge edge(face->v[k], face->v[NEXT(k)]);
SDVertex *vert = edgeVerts[edge];
if (!vert) {
    ⟨Create and initialize new odd vertex 167⟩
    ⟨Apply edge rules to compute new vertex position 167⟩
    edgeVerts[edge] = vert;
}
```

In Loop subdivision, the new vertices added by subdivision are always regular. (This means that the proportion of extraordinary vertices with respect to regular vertices will decrease with each level of subdivision.) Therefore, the `regular` member of the new vertex can immediately be set to `true`. The `boundary` member can also be easily initialized, by checking to see if there is a neighbor face across the edge that is being split. Finally, the new vertex's `startFace` pointer can also be set here. For all odd vertices on the edges of a face, the center child (child face number three) is guaranteed to be adjacent to the new vertex.

⟨*Create and initialize new odd vertex*⟩ ≡ 167
```
vert = arena.Alloc<SDVertex>();
newVertices.push_back(vert);
vert->regular = true;
vert->boundary = (face->f[k] == NULL);
vert->startFace = face->children[3];
```

For odd boundary vertices, the new vertex is just the average of the two adjacent vertices. For odd interior vertices, the two vertices at the ends of the edge are given weight $\frac{3}{8}$, and the two vertices opposite the edge are given weight $\frac{1}{8}$ (Figure 3.23). These last two vertices can be found using the `SDFace::otherVert()` utility function, which returns the vertex opposite a given edge of a face.

⟨*Apply edge rules to compute new vertex position*⟩ ≡ 167
```
if (vert->boundary) {
    vert->P =  0.5f * edge.v[0]->P;
    vert->P += 0.5f * edge.v[1]->P;
}
else {
    vert->P =  3.f/8.f * edge.v[0]->P;
    vert->P += 3.f/8.f * edge.v[1]->P;
    vert->P += 1.f/8.f * face->otherVert(edge.v[0], edge.v[1])->P;
    vert->P += 1.f/8.f *
        face->f[k]->otherVert(edge.v[0], edge.v[1])->P;
}
```

MemoryArena::Alloc() 1016
NEXT() 153
SDEdge 155
SDEdge::v 155
SDFace::children 153
SDFace::f 153
SDFace::otherVert() 168
SDFace::v 153
SDVertex 152
SDVertex::boundary 152
SDVertex::P 152
SDVertex::regular 152
SDVertex::startFace 152

The SDFace::otherVert() method is self-explanatory:

⟨*SDFace Methods*⟩ +≡ 153

```
SDVertex *otherVert(SDVertex *v0, SDVertex *v1) {
    for (int i = 0; i < 3; ++i)
        if (v[i] != v0 && v[i] != v1)
            return v[i];
    Severe("Basic logic error in SDVertex::otherVert()");
    return NULL;
}
```

Updating Mesh Topology

In order to keep the details of the topology update as straightforward as possible, the numbering scheme for the subdivided faces and their vertices has been chosen carefully (Figure 3.24). Review that figure carefully; the conventions shown there are key to the next few pages.

There are four main tasks required to update the topological pointers of the refined mesh:

1. The odd vertices' SDVertex::startFace pointers need to store a pointer to one of their adjacent faces.
2. Similarly, the even vertices' SDVertex::startFace pointers must be set.
3. The new faces' neighbor f[i] pointers need to be set to point to the neighboring faces.
4. The new faces' v[i] pointers need to point to the appropriate vertices.

The startFace pointers of the odd vertices were already initialized when they were first created. We'll handle the other three tasks in order here.

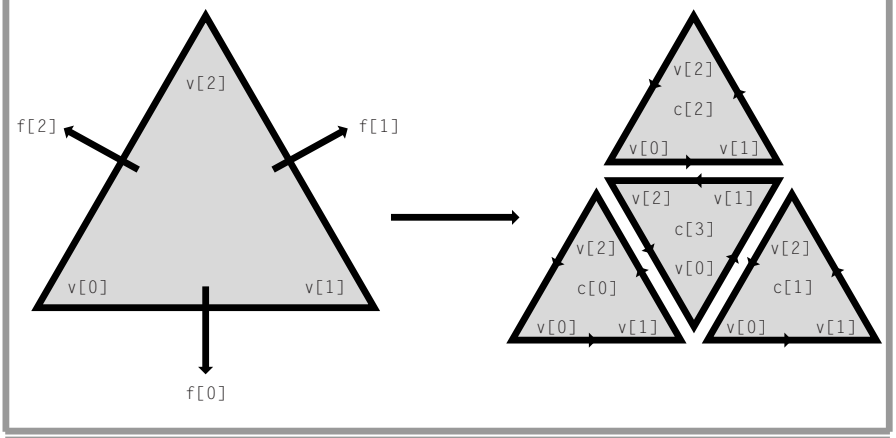

Figure 3.24: Each face is split into four child faces, such that the ith child is adjacent to the ith vertex of the original face, and such that the ith child face's ith vertex is the child of the ith vertex of the original face. The vertices of the center child are oriented such that the ith vertex is the odd vertex along the ith edge of the parent face.

SDFace::otherVert() 168
SDFace::v 153
SDVertex 152
SDVertex::startFace 152
Severe() 1005

⟨*Update new mesh topology*⟩ ≡ 161
⟨*Update even vertex face pointers* 169⟩
⟨*Update face neighbor pointers* 169⟩
⟨*Update face vertex pointers* 170⟩

If a vertex is the `i`th vertex of its `startFace`, then it is guaranteed that it will be adjacent to the `i`th child face of `startFace`. Therefore, it is just necessary to loop through all the parent vertices in the mesh, and for each one find its vertex index in its `startFace`. This index can then be used to find the child face adjacent to the new even vertex.

⟨*Update even vertex face pointers*⟩ ≡ 169
```
for (uint32_t j = 0; j < v.size(); ++j) {
    SDVertex *vert = v[j];
    int vertNum = vert->startFace->vnum(vert);
    vert->child->startFace =
        vert->startFace->children[vertNum];
}
```

Next, the face neighbor pointers for the newly created faces are updated. We break this into two steps: one to update neighbors among children of the same parent, and one to do neighbors across children of different parents. This involves some tricky pointer manipulation.

⟨*Update face neighbor pointers*⟩ ≡ 169
```
for (uint32_t j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update children f pointers for siblings 169⟩
        ⟨Update children f pointers for neighbor children 170⟩
    }
}
```

For the first step, recall that the interior child face is always stored in `children[3]`. Furthermore, the `k` + 1st child face (for `k` = 0, 1, 2) is across the `k`th edge of the interior face, and the interior face is across the `k` + 1st edge of the `k`th face.

⟨*Update children* `f` *pointers for siblings*⟩ ≡ 169
```
face->children[3]->f[k] = face->children[NEXT(k)];
face->children[k]->f[NEXT(k)] = face->children[3];
```

We'll now update the children's face neighbor pointers that point to children of other parents. Only the first three children need to be addressed here; the interior child's neighbor pointers have already been fully initialized. Inspection of Figure 3.24 reveals that the `k`th and `PREV(k)`th edges of the `i`th child need to be set. To set the `k`th edge of the `k`th child, we first find the `k`th edge of the parent face, then the neighbor parent `f2` across that edge. If `f2` exists (meaning we aren't on a boundary), the neighbor parent index for the vertex `v[k]` is found. That index is equal to the index of the neighbor child we are searching for. This process is then repeated to find the child across the `PREV(k)`th edge.

NEXT() 153
SDFace 153
SDFace::children 153
SDFace::f 153
SDFace::vnum() 159
SDVertex 152
SDVertex::startFace 152

⟨*Update children* f *pointers for neighbor children*⟩ ≡ 169
```
SDFace *f2 = face->f[k];
face->children[k]->f[k] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
f2 = face->f[PREV(k)];
face->children[k]->f[PREV(k)] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
```

Finally, we handle the fourth step in the topological updates: setting the children faces' vertex pointers.

⟨*Update face vertex pointers*⟩ ≡ 169
```
for (uint32_t j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update child vertex pointer to new even vertex 170⟩
        ⟨Update child vertex pointer to new odd vertex 170⟩
    }
}
```

For the kth child face (for k = 0, 1, 2), the kth vertex corresponds to the even vertex that is adjacent to the child face. For the noninterior child faces, there is one even vertex and two odd vertices; for the interior child face, there are three odd vertices. This vertex can be found by following the child pointer of the parent vertex, available from the parent face.

⟨*Update child vertex pointer to new even vertex*⟩ ≡ 170
```
face->children[k]->v[k] = face->v[k]->child;
```

To update the rest of the vertex pointers, the edgeVerts associative array is reused to find the odd vertex for each split edge of the parent face. Three child faces have that vertex as an incident vertex. The vertex indices for the three faces are easily found, again based on the numbering scheme established in Figure 3.24.

⟨*Update child vertex pointer to new odd vertex*⟩ ≡ 170
```
SDVertex *vert = edgeVerts[SDEdge(face->v[k], face->v[NEXT(k)])];
face->children[k]->v[NEXT(k)] = vert;
face->children[NEXT(k)]->v[k] = vert;
face->children[3]->v[k] = vert;
```

After the geometric and topological work has been done for a subdivision step, the newly created vertices and faces are moved into the v and f arrays:

⟨*Prepare for next level of subdivision*⟩ ≡ 161
```
f = newFaces;
v = newVertices;
```

To the Limit Surface and Output

One of the remarkable properties of subdivision surfaces is that there are special subdivision rules that give the positions that the vertices of the mesh would have if we continued subdividing forever. We apply these rules here to initialize an array of limit surface positions, Plimit. Note that it's important to temporarily store the limit surface positions

NEXT() 153
PREV() 153
SDEdge 155
SDFace 153
SDFace::children 153
SDFace::f 153
SDFace::v 153
SDFace::vnum() 159
SDVertex 152
SDVertex::child 152

somewhere other than in the vertices while the computation is taking place. Because the limit surface position of each vertex depends on the original positions of its surrounding vertices, the original positions of all vertices must remain unchanged until the computation is complete.

The limit rule for a boundary vertex weights the two neighbor vertices by $\frac{1}{5}$ and the center vertex by $\frac{3}{5}$. The rule for interior vertices is based on a function `gamma()`, which computes appropriate vertex weights based on the valence of the vertex.

⟨*Push vertices to limit surface*⟩ ≡ 161
```
Point *Plimit = new Point[v.size()];
for (uint32_t i = 0; i < v.size(); ++i) {
    if (v[i]->boundary)
        Plimit[i] =  weightBoundary(v[i], 1.f/5.f);
    else
        Plimit[i] =  weightOneRing(v[i], gamma(v[i]->valence()));
}
for (uint32_t i = 0; i < v.size(); ++i)
    v[i]->P = Plimit[i];
```

⟨*LoopSubdiv Private Methods*⟩ +≡ 151
```
static float gamma(int valence) {
    return 1.f / (valence + 3.f / (8.f * beta(valence)));
}
```

In order to generate a smooth-looking triangle mesh with per-vertex surface normals, a pair of nonparallel tangent vectors to the limit surface is computed at each vertex. As with the limit rule for positions, this is an analytic computation that gives the precise tangents on the actual limit surface.

⟨*Compute vertex tangents on limit surface*⟩ ≡ 161
```
vector<Normal> Ns;
Ns.reserve(v.size());
vector<Point> Pring(16, Point());
for (uint32_t i = 0; i < v.size(); ++i) {
    SDVertex *vert = v[i];
    Vector S(0,0,0), T(0,0,0);
    int valence = vert->valence();
    if (valence > (int)Pring.size())
        Pring.resize(valence);
    vert->oneRing(&Pring[0]);
    if (!vert->boundary) {
        ⟨Compute tangents of interior face 172⟩
    } else {
        ⟨Compute tangents of boundary face 173⟩
    }
    Ns.push_back(Normal(Cross(S, T)));
}
```

Cross() 62
LoopSubdiv::gamma() 171
LoopSubdiv::
  weightBoundary() 166
LoopSubdiv::
  weightOneRing() 164
Normal 65
Point 63
SDVertex 152
SDVertex::boundary 152
SDVertex::oneRing() 164
SDVertex::valence() 158
Vector 57

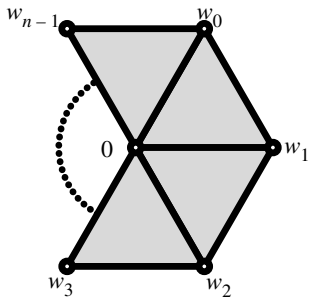

Figure 3.25: To compute tangents for interior vertices, the one-ring vertices are weighted with weights w_i. The center vertex, where the tangent is being computed, always has a weight of 0.

Figure 3.25 shows the setting for computing tangents in the mesh interior. The center vertex is given a weight of zero and the neighbors are given weights w_i. To compute the first tangent vector **s**, the weights are

$$w_i = \cos\left(\frac{2\pi i}{n}\right),$$

where n is the valence of the vertex. The second tangent **t** is computed with weights

$$w_i = \sin\left(\frac{2\pi i}{n}\right).$$

⟨*Compute tangents of interior face*⟩ ≡ 171

```
for (int k = 0; k < valence; ++k) {
    S += cosf(2.f*M_PI*k/valence) * Vector(Pring[k]);
    T += sinf(2.f*M_PI*k/valence) * Vector(Pring[k]);
}
```

Tangents on boundary vertices are a bit trickier. Figure 3.26 shows the ordering of vertices in the one-ring expected in the following discussion.

The first tangent, known as the *across tangent*, is given by the vector between the two neighboring boundary vertices:

$$\mathbf{s} = \mathrm{v}_{n-1} - \mathrm{v}_0.$$

The second tangent, known as the *transverse tangent*, is computed based on the vertex's valence. The center vertex is given a weight w_c and the one-ring vertices are given weights specified by a vector $(w_0, w_1, \ldots, w_{n-1})$. The transverse tangent rules we will use are

Valence	w_c	w_i
2	−2	(1, 1)
3	−1	(0, 1, 0)
4 (regular)	−2	(−1, 2, 2, −1)

M_PI 1002
Vector 57

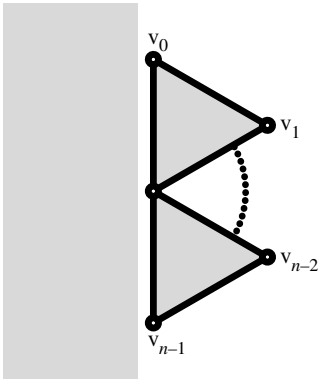

Figure 3.26: Tangents at boundary vertices are also computed as weighted averages of the adjacent vertices. However, some of the boundary tangent rules incorporate the value of the center vertex.

For valences of 5 and higher, $w_c = 0$ and

$$w_0 = w_{n-1} = \sin\theta$$
$$w_i = (2\cos\theta - 2)\ \sin(\theta i),$$

where

$$\theta = \frac{\pi}{n-1}.$$

Although we will not prove it here, these weights sum to zero for all values of i. This guarantees that the weighted sum is in fact a tangent vector.

⟨*Compute tangents of boundary face*⟩ ≡ 171

```
S = Pring[valence-1] - Pring[0];
if (valence == 2)
    T = Vector(Pring[0] + Pring[1] - 2 * vert->P);
else if (valence == 3)
    T = Pring[1] - vert->P;
else if (valence == 4) // regular
    T = Vector(-1*Pring[0] + 2*Pring[1] + 2*Pring[2] +
               -1*Pring[3] + -2*vert->P);
else {
    float theta = M_PI / float(valence-1);
    T = Vector(sinf(theta) * (Pring[0] + Pring[valence-1]));
    for (int k = 1; k < valence-1; ++k) {
        float wt = (2*cosf(theta) - 2) * sinf((k) * theta);
        T += Vector(wt * Pring[k]);
    }
    T = -T;
}
```

M_PI 1002
Vector 57

Finally, the fragment ⟨*Create* `TriangleMesh` *from subdivision mesh*⟩ creates the triangle mesh object and adds it to the `refined` vector passed to the `LoopSubdiv::Refine()` method. We won't include it here, since it's just a straightforward transformation of the subdivided mesh into an indexed triangle mesh.

FURTHER READING

An Introduction to Ray Tracing has an extensive survey of algorithms for ray-shape intersection (Glassner 1989a). Goldstein and Nagel (1971) discussed ray-quadric intersections, and Heckbert (1984) discussed the mathematics of quadrics for graphics applications in detail, with many citations to literature in mathematics and other fields. Hanrahan (1983) described a system that automates the process of deriving a ray intersection routine for surfaces defined by implicit polynomials; his system emits C source code to perform the intersection test and normal computation for a surface described by a given equation. Mitchell (1990) showed that interval arithmetic could be applied to develop algorithms for robustly computing intersections with implicit surfaces that cannot be described by polynomials and are thus more difficult to accurately compute intersections for; more recent work in this area was done by Knoll et al. (2009). See Moore's book (1966) for an introduction to interval arithmetic.

Other notable early papers related to ray–shape intersection include Kajiya's work on computing intersections with surfaces of revolution and procedurally generated fractal terrains (Kajiya 1983). Fournier et al.'s paper on rendering procedural stochastic models (Fournier, Fussel, and Carpenter 1982) and Hart et al.'s paper on finding intersections with fractals (Hart, Sandin, and Kauffman 1989) illustrate the broad range of shape representations that can be used with ray-tracing algorithms.

Kajiya (1982) developed the first algorithm for computing intersections with parametric patches. Recent work on more efficient techniques for direct ray intersection with patches includes papers by Stürzlinger (1998), Martin et al. (2000), and Roth et al. (2001). Benthin et al. (2004) presented more recent results and include additional references to previous work.

The ray–triangle intersection test in Section 3.6 was developed by Möller and Trumbore (1997). A ray–quadrilateral intersection routine was developed by Lagae and Dutré (2005). Shevtsov et al. (2007a) described a highly optimized ray–triangle intersection routine for modern CPU architectures and included a number of references to other recent approaches. An interesting approach for developing a fast ray–triangle intersection routine was introduced by Kensler and Shirley (2006): they implemented a program that performed a search across the space of mathematically equivalent ray–triangle tests, automatically generating software implementations of variations and then benchmarking them. In the end, they found an more efficient ray–triangle routine than had been in use previously.

The layout of triangle meshes in memory can have a measurable impact on performance in many situations. In general, if triangles that are close together in 3D space are close together in memory, cache hit rates will be higher, and overall system performance will

benefit. See Yoon et al. (2005) and Yoon and Lindstrom (2006) for algorithms for creating cache-friendly mesh layouts in memory.

Subdivision surfaces were invented by Doo and Sabin (1978) and Catmull and Clark (1978). The Loop subdivision method was originally developed by Charles Loop (1987), although the implementation in `pbrt` uses the improved rules for subdivision and tangents along boundary edges developed by Hoppe et al. (1994). There has been extensive work in subdivision surfaces recently. The SIGGRAPH course notes give a good summary of the state of the art and also have extensive references (Zorin et al. 2000). See also Warren's book on the topic (Warren 2002). Müller et al. (2003) described an approach that refines a subdivision surface on demand for the rays to be tested for intersection with it. (See also Benthin et al. 2007, for a related approach.)

An exciting development in subdivision surfaces is the ability to evaluate them at arbitrary points on the surface (Stam 1998). Subdivision surface implementations like the one in this chapter are often relatively inefficient, spending as much time dereferencing pointers as they do applying subdivision rules. Stam's approach also reduces the impact of this problem. Bolz and Schröder (2002) suggest a much more efficient implementation approach that precomputes a number of quantities that make it possible to compute the final mesh much more efficiently. More recently, Patney et al. (2009) have demonstrated a very efficient approach for tessellating subdivision surfaces on data-parallel throughput processors.

Phong and Crow (1975) first introduced the idea of interpolating per-vertex shading normals to give the appearance of smooth surfaces from polygonal meshes.

The notion of shapes that repeatedly refine themselves into collections of other shapes until ready for rendering was first introduced in the Reyes renderer (Cook, Carpenter, and Catmull 1987).

An excellent introduction to differential geometry is Gray (1993); Section 14.3 of that book presents the Weingarten equations. Turkowski (1990a) has expressions for first and second partial derivatives of a handful of parametric primitives.

EXERCISES

2 **3.1** One nice property of mesh-based shapes like triangle meshes and subdivision surfaces is that the shape's vertices can be transformed into world space, so that it isn't necessary to transform rays into object space before performing ray intersection tests. Interestingly enough, it is possible to do the same thing for ray–quadric intersections.

The implicit forms of the quadrics in this chapter were all of the form

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + G = 0,$$

where some of the constants $A \ldots G$ were zero. More generally, we can define quadric surfaces by the equation

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gz + 2Hy + 2Iz + J = 0,$$

where most of the parameters $A \ldots J$ don't directly correspond to the earlier $A \ldots G$. In this form, the quadric can be represented by a 4×4 symmetric matrix $\mathbf{Q}$:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{pmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & I \\ G & H & I & J \end{pmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathrm{p}^T \mathbf{Q} \mathrm{p} = 0.$$

Given this representation, first show that the matrix $\mathbf{Q}'$ representing a quadric transformed by the matrix $\mathbf{M}$ is

$$\mathbf{Q}' = (\mathbf{M}^T)^{-1} \mathbf{Q} \mathbf{M}^{-1}.$$

To do so, show that for any point p where $\mathrm{p}^T \mathbf{Q} \mathrm{p} = 0$, if we apply a transformation $\mathbf{M}$ to p and compute $\mathrm{p}' = \mathbf{M}\mathrm{p}$, we'd like to find $\mathbf{Q}'$ so that $(\mathrm{p}')^T \mathbf{Q}' \mathrm{p}' = 0$.

Next, substitute the ray equation into the earlier more general quadric equation to compute coefficients for the quadratic equation $at^2 + bt + c = 0$ in terms of entries of the matrix $\mathbf{Q}$ to pass to the `Quadratic()` function.

Now implement this approach in `pbrt` and use it instead of the original quadric intersection routines. Note that you will still need to transform the resulting world space hit points into object space to test against $\theta_{\max}$, if it is not 2π, and so on. How does performance compare to the original scheme?

❶ 3.2 Improve the object space bounding box routines for the quadrics to properly account for $\phi_{\max} < 2\pi$ and compute tighter bounding boxes when possible. How much does this improve performance when rendering scenes with partial quadric shapes?

❷ 3.3 There is room to optimize the implementations of the various quadric primitives in `pbrt` in a number of ways. For example, for complete spheres some of the tests in the intersection routine related to partial spheres are unnecessary. Furthermore, some of the quadrics have calls to trigonometric functions that could be turned into simpler expressions using insight about the geometry of the particular primitives. Investigate ways to speed up these methods. How much does doing so improve the overall run time of `pbrt`?

❶ 3.4 Currently `pbrt` recomputes the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ for triangles every time they are needed, even though they are constant for each triangle. Precompute these vectors and analyze the speed/storage trade-off, especially for large triangle meshes. How does the depth complexity of the scene and the size of triangles in the image affect this trade-off?

❷ 3.5 Implement a general polygon primitive that supports an arbitrary number of vertices and convex or concave polygons as a new `Shape` in `pbrt`. You can assume that a valid polygon has been provided and that all of the vertices of the polygon lie on the same plane, although you might want to issue a warning when this is not the case.

`Quadratic()` 118
`Shape` 108

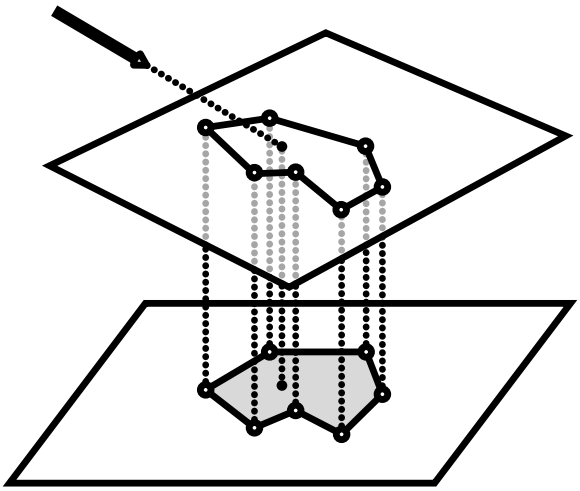

Figure 3.27: A ray–polygon intersection test can be performed by finding the point where the ray intersects the polygon's plane, projecting the hit point and polygon vertices onto an axis-aligned plane, and doing a 2D point-in-polygon test there.

An efficient technique for computing ray-polygon intersections is to find the plane equation for the polygon from its normal and a point on the plane. Then compute the intersection of the ray with that plane and project the intersection point and the polygon vertices to 2D. Then apply a 2D point-in-polygon test to determine if the point is inside the polygon. An easy way to do this is to effectively do a 2D ray-tracing computation and intersect the ray with each of the edge segments and count how many it goes through. If it goes through an odd number of them, the point is inside the polygon and there is an intersection. See Figure 3.27 for an illustration of this idea.

You may find it helpful to read the article by Haines (1994) that surveys a number of approaches for efficient point-in-polygon tests. Some of the techniques described there may be helpful for optimizing this test. Furthermore, Section 13.3.3 of Schneider and Eberly (2003) discusses strategies for getting all the corner cases right, for example, when the 2D ray is aligned precisely with an edge or passes through a vertex of the polygon.

❷ 3.6 Constructive solid geometry (CSG) is a classic solid modeling technique, where complex shapes are built up by considering the union, intersection, and differences of more primitive shapes. For example, a sphere could be used to create pits in a cylinder if a shape was modeled as the difference of a cylinder and set of spheres that partially overlapped it. See Hoffmann (1989) for further information about CSG.

Add support for CSG to pbrt and render images that demonstrate interesting shapes that can be rendered using CSG. You may want to read Roth (1982), which first described how ray tracing could be used to render models described by CSG, as well as Amanatides and Mitchell (1990), which discusses precision-related issues for CSG ray tracing.

● 3.7 Procedurally–described parametric surfaces: Write a Shape that takes a general mathematical expression of the form $f(u, v) \rightarrow (x, y, z)$ that describes a parametric surface as a function of (u, v). Evaluate the given function at a grid of (u, v) positions and create a TriangleMesh that approximates the given surface when the Shape::Refine() method is called.

● 3.8 Almost all methods for subdivision surfaces are based on either refining a mesh of triangles or a mesh of quadrilaterals. If a rendering system only supports one type of mesh, meshes of the other type are typically tessellated to make faces of the expected type in a preprocessing step. However, doing this can introduce artifacts in the final subdivision surface. Read Stam and Loop's paper on a hybrid subdivision scheme that supports meshes with both quadrilateral and triangular faces (Stam and Loop 2003) and implement a Shape based on their method. Demonstrate cases where the subdivision surface that your implementation creates does not have artifacts that are present in the output from LoopSubdiv.

● 3.9 The smoothness of subdivision surfaces isn't always desirable. Sometimes it is useful to be able to flag some edges of a subdivision control mesh as "creases" and apply different subdivision rules there to preserve a sharp edge. Extend the LoopSubdiv implementation so that some edges can be denoted as creases, and use the boundary subdivision rules to compute the positions of vertices along those edges. Render images showing the difference this makes.

● 3.10 Implement adaptive subdivision for the LoopSubdiv Shape. A weakness of the basic implementation in Section 3.7 is that each face is always refined a fixed number of times: this may mean that some faces are underrefined, leading to visible faceting in the triangle mesh, and some faces are overrefined, leading to excessive memory use and rendering time. With adaptive subdivision, individual faces are no longer subdivided once a particular error threshold has been reached.

An easy error threshold to implement computes the face normals of each face and its directly adjacent faces. If they are sufficiently close to each other (e.g., as tested via dot products), then the limit surface for that face will be reasonably flat and further refinement will likely make little difference to the final surface. Alternatively, you might want to approximate the area that a subdivided face covers on the image plane and continue subdividing until this area becomes sufficiently small. This approximation could be done using ray differentials; see Section 10.1.1 for an explanation of how to relate the ray differential to the screen space footprint.

LoopSubdiv 151
Shape 108
Shape::Refine() 110
TriangleMesh 135

The trickiest part of this exercise is that some faces that don't need subdivision due to the flatness test will still need to be subdivided in order to provide vertices so that neighboring faces that do need to subdivide can get their vertex one-rings. In particular, adjacent faces can differ by no more than one level of subdivision. You may find it useful to read recent papers by Patney et al. (2009) and Fisher et al. (2009) for discussion of how to avoid cracks in adaptively subdivided meshes.

3.11 Ray-tracing point-sampled geometry: Extending methods for rendering complex models represented as a collection of point samples (Levoy and Whitted 1985; Pfister et al. 2000; Rusinkiewicz and Levoy 2000), Schaufler and Jensen (2000) have described a method for intersecting rays with collections of oriented point samples in space. They probabilistically determined that an intersection has occurred when a ray approaches a sufficient local density of point samples and computes a surface normal with a weighted average of the nearby samples. Read their paper and extend `pbrt` to support a point-sampled geometry shape. Do any of `pbrt`'s basic interfaces need to be extended or generalized to support a shape like this?

3.12 Ray-tracing ribbons: Hair is often modeled as a collection of *generalized cylinders*, which are defined as the shape that results from sweeping a disk along a given curve. Because there are often a large number of individual hairs, an efficient method for intersecting rays with generalized cylinders is needed for ray-tracing hair. A number of methods have been developed to compute ray intersections with generalized cylinders (Bronsvoort and Klok 1985; de Voogt, van der Helm, and Bronsvoort 2000); investigate these algorithms and extend `pbrt` to support a fast hair primitive with one of them. Alternatively, investigate the generalization of Schaufler and Jensen's approach for probabilistic point intersection (Schaufler and Jensen 2000) to probabilistic line intersection and apply this to fast ray tracing of hair.

3.13 Deformation motion blur: The `TransformedPrimitive` in Section 4.1.2 of Chapter 4 supports animated shapes via transformations of primitives that vary over time. However, this type of animation isn't general enough to represent a triangle mesh where each vertex has a position given at the start time and another one at the end time. (For example, this type of animation description can be used to describe a running character model where different parts of the body are moving in different ways, such that the overall shape deforms.) Implement a more general `TriangleMesh` shape that supports specifying vertex positions at the start and end of frame and interpolates between them based on the ray time passed to the intersection methods. Be sure to update the bounding routines appropriately.

Meshes with very large amounts of motion may exhibit poor performance due to triangles sweeping out very large bounding boxes and thus many ray-triangle intersections being performed that don't hit the triangle. Can you come up with approaches that could be used to reduce the impact of this problem?

TransformedPrimitive 190

3.14 Implicit functions: Just as implicit definitions of the quadric shapes are a useful starting point for deriving ray intersection algorithms, more complex implicit functions can also be used to define interesting shapes. In particular, difficult-to-model organic shapes, water drops, and so on can be well-represented by implicit surfaces. Blinn (1982a) introduced the idea of directly rendering implicit surfaces, and Wyvill and Wyvill (1989) gave a basis function for implicit surfaces with a number of advantages compared to Blinn's.

Implement a method for finding ray intersections with general implicit surfaces and add it to `pbrt`. You may wish to read papers by Kalra and Barr (1989) and Hart (1996) for methods for ray-tracing them. Mitchell's algorithm for robust ray intersections with implicit surfaces gives another effective method for finding these intersections (Mitchell 1990), and more recently Knoll et al. (2009) described refinements to this idea. You may find an approach along these lines easier to implement than the others. See Moore's book on interval arithmetic as needed for reference (Moore 1966).

3.15 L-systems: A very successful technique for procedurally modeling plants was introduced to graphics by Alvy Ray Smith (1984), who applied *Lindenmayer systems* (L-systems) to model branching plant structures. Prusinkiewicz and collaborators have generalized this approach to encompass a much wider variety of types of plants and effects that determine their appearance (Prusinkiewicz 1986; Prusinkiewicz, James, and Mech 1994; Deussen et al. 1998; Prusinkiewicz et al. 2001). L-systems describe the branching structure of these types of shapes via a grammar. The grammar can be evaluated to form expressions that describe a topological representation of a plant, which can then be translated into a geometric representation. Add an L-system primitive to `pbrt` that takes a grammar as input and evaluates it on demand at rendering time to create the plant described by it.

3.16 The expressions used to initialize `rayEpsilon` values in the intersection routines in this chapter were derived in an *ad hoc* manner and aren't necessarily always sufficiently conservative. For either the ray–triangle intersection routine or some of the ray–quadric intersection routines, apply techniques from numerical analysis to derive conservative bounds on the floating-point error in the computed intersection parametertic t values. In general, these expressions should depend on the particular floating-point computations performed in the intersection routines and will be related to the magnitudes of the ray origin, direction, and the parameters that describe the shape (vertex position, radius, etc.).

3.17 As an alternative to Exercise 3.16, Kalra and Barr (1989) described an approach for robust ray–object intersections that is based on recursively subdividing the bounding box of the object, discarding boxes that don't encompass the object's surface, and discarding boxes that the ray misses. Dammertz and Keller (2006) proposed an application of this approach where once a box has been found where the ray intersects it and it is no longer possible to subdivide the box without it vanishing due to no remaining numerical precision, then an intersection

is reported. (They also described implementations for spline patches and triangles.) Implement this approach in `pbrt` and evaluate its speed and quality. You will probably need to make substantial modifications to the system, as the `Shape::Intersect()` method probably should return a pair of `Point`s, one on the side of the surface the ray came from and the other on the other side of the surface, rather than just a parametric t value. (These two points then should be be used for tracing spawned reflected and transmitted rays, respectively.)

Point 63
Shape::Intersect() 111