CHAPTER TWO

02 GEOMETRY AND TRANSFORMATIONS

Almost all nontrivial graphics programs are built on a foundation of geometric classes. These classes represent mathematical constructs like points, vectors, and rays. Because these classes are ubiquitous throughout the system, good abstractions and efficient implementations are critical. This chapter presents the interface to and implementation of `pbrt`'s geometric foundation. Note that these are not the classes that represent the actual scene geometry (triangles, spheres, etc.); those classes are the topic of Chapter 3.

The geometric classes in this chapter are defined in the files `core/geometry.h` and `core/geometry.cpp` in the `pbrt` distribution and the implementation of transformation matrices (Section 2.7) are in the files `core/transform.h` and `core/transform.cpp`.

2.1 COORDINATE SYSTEMS

As is typical in computer graphics, `pbrt` represents three-dimensional points, vectors, and normal vectors with three floating-point coordinate values: x, y, and z. Of course, these values are meaningless without a *coordinate system* that defines the origin of the space and gives three linearly independent vectors that define the x, y, and z axes of the space. Together, the origin and three vectors are called the *frame* that defines the coordinate system. Given an arbitrary point or direction in 3D, its (x, y, z) coordinate values depend on its relationship to the frame. Figure 2.1 shows an example that illustrates this idea in 2D.

In the general n-dimensional case, a frame's origin $\mathrm{p_o}$ and its n linearly independent basis vectors define an n-dimensional *affine space*. All vectors $\mathbf{v}$ in the space can be expressed as a linear combination of the basis vectors. Given a vector $\mathbf{v}$ and the basis vectors $\mathbf{v}_i$,

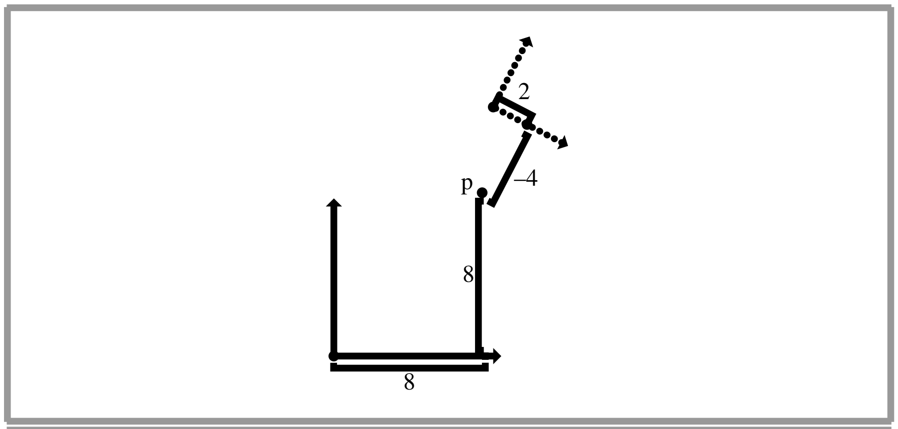


Figure 2.1: In 2D, the (x, y) coordinates of a point p are defined by the relationship of the point to a particular 2D coordinate system. Here, two coordinate systems are shown; the point might have coordinates $(8, 8)$ with respect to the coordinate system with its coordinate axes drawn in solid lines, but have coordinates $(2, -4)$ with respect to the coordinate system with dashed axes. In either case, the 2D point p is at the same absolute position in space.

there is a unique set of scalar values s_i such that

$$\mathbf{v} = s_1\mathbf{v}_1 + \cdots + s_n\mathbf{v}_n.$$

The scalars s_i are the *representation* of $\mathbf{v}$ with respect to the basis $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ and are the coordinate values that we store with the vector. Similarly, for all points p, there are unique scalars s_i such that the point can be expressed in terms of the origin p_o and the basis vectors

$$\mathrm{p} = \mathrm{p}_\mathrm{o} + s_1\mathbf{v}_1 + \cdots + s_n\mathbf{v}_n.$$

Thus, although points and vectors are both represented by x, y, and z coordinates in 3D, they are distinct mathematical entities and are not freely interchangeable.

This definition of points and vectors in terms of coordinate systems reveals a paradox: to define a frame we need a point and a set of vectors, but we can only meaningfully talk about points and vectors with respect to a particular frame. Therefore, in three dimensions we need a *standard frame* with origin (0, 0, 0) and basis vectors (1, 0, 0), (0, 1, 0), and (0, 0, 1). All other frames will be defined with respect to this canonical coordinate system which we call *world space*.

2.1.1 COORDINATE SYSTEM HANDEDNESS

There are two different ways that the three coordinate axes can be arranged, as shown in Figure 2.2. Given perpendicular x and y coordinate axes, the z axis can point in one of two directions. These two choices are called *left-handed* and *right-handed*. The choice between the two is arbitrary, but has a number of implications for how some of the geometric operations throughout the chapter are defined. `pbrt` uses a left-handed coordinate system.

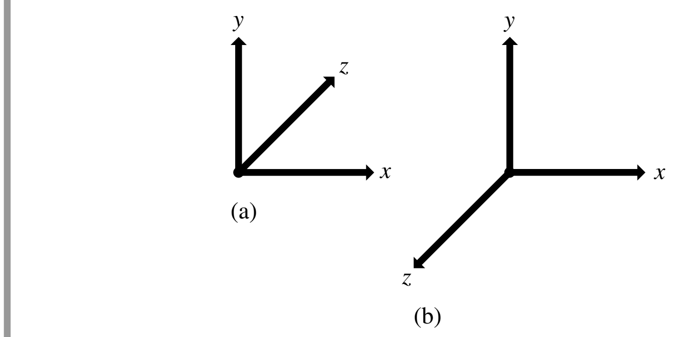


Figure 2.2: (a) In a left-handed coordinate system, the z axis points into the page when the x and y axes are oriented with x pointing to the right and y pointing up. (b) In a right-handed system, the z axis points out of the page.

2.2 VECTORS

```
⟨Geometry Declarations⟩ ≡
    class Vector {
    public:
        ⟨Vector Public Methods 58⟩
        ⟨Vector Public Data 57⟩
    };
```

A `Vector` in `pbrt` represents a direction in 3D space. As described earlier, vectors are represented with a three-tuple of components that give its representation in terms of the x, y, and z axes of the space it is defined in. The individual components of a vector $\mathbf{v}$ will be written $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$.

```
⟨Vector Public Data⟩ ≡                                                  57
    float x, y, z;
```

Readers who are experienced in object-oriented design might object to our decision to make the `Vector` data publicly accessible. Typically, data members are only accessible inside the class, and external code that wishes to access or modify the contents of a class must do so through a well-defined API of selector and mutator functions. Although we generally agree with this design principle, it is not appropriate here. The purpose of selector and mutator functions is to hide the class's internal implementation details. In the case of `Vector`s, hiding this basic part of their design gains nothing and adds bulk to the class usage.

`Assert()` 1005
`Vector` 57

By default, the (x, y, z) values are set to zero, although the user of the class can optionally supply values for each of the components. If the user does supply values, we check that none of them has the floating-point not-a-number (NaN) value using the `Assert()` macro. When compiled in optimized mode, this macro disappears from the compiled code, saving the expense of verifying this case. `Assert()` is defined in Section A.3.1.

NaNs almost certainly indicate a bug in the system; if NaNs are generated by some computation, we'd like to catch them as soon as possible in order to make isolating their source easier.

⟨*Vector Public Methods*⟩ ≡ **57**

```
Vector() { x = y = z = 0.f; }
Vector(float xx, float yy, float zz)
    : x(xx), y(yy), z(zz) {
    Assert(!HasNaNs());
}
```

The code to check for NaNs just calls the standard math library `isnan()` function on each of x, y, and z.

⟨*Vector Public Methods*⟩ +≡ **57**

```
bool HasNaNs() const { return isnan(x) || isnan(y) || isnan(z); }
```

2.2.1 ARITHMETIC

Addition and subtraction of vectors are done component-wise. The usual geometric interpretation of vector addition and subtraction is shown in Figures 2.3 and 2.4.

⟨*Vector Public Methods*⟩ +≡ **57**

```
Vector operator+(const Vector &v) const {
    return Vector(x + v.x, y + v.y, z + v.z);
}

Vector& operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

The code for subtracting two vectors is similar, and therefore not shown here.

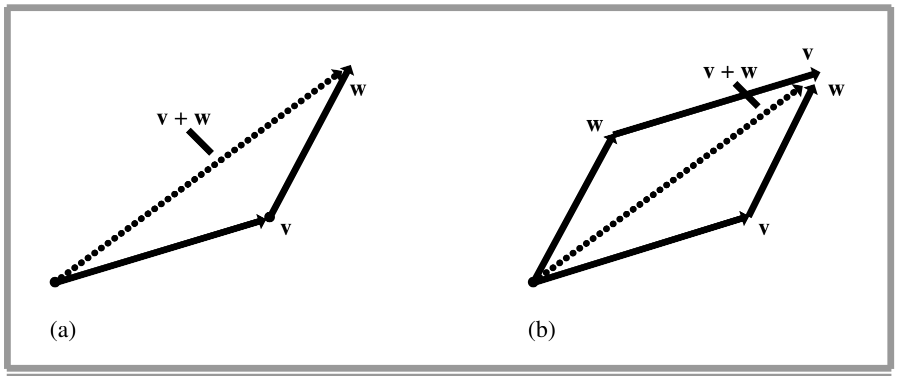


Figure 2.3: (a) Vector addition: $\mathbf{v}+\mathbf{w}$. (b) Notice that the sum $\mathbf{v}+\mathbf{w}$ forms the diagonal of the parallelogram formed by $\mathbf{v}$ and $\mathbf{w}$, which shows the commutativity of vector addition: $\mathbf{v}+\mathbf{w}=\mathbf{w}+\mathbf{v}$.

`Assert()` 1005
`Vector` 57
`Vector::HasNaNs()` 58

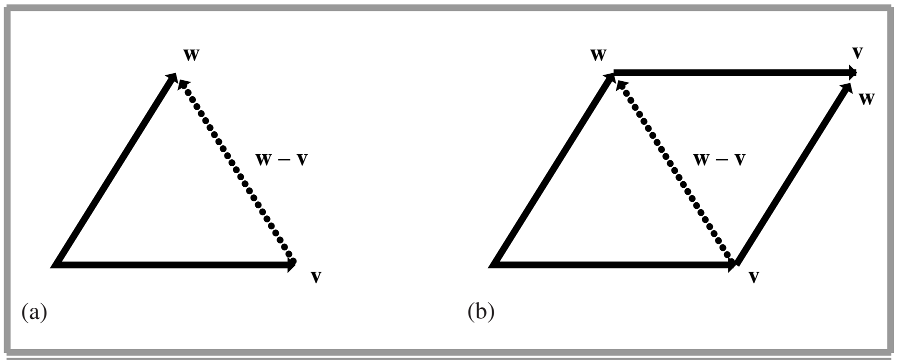


Figure 2.4: (a) Vector subtraction. (b) The difference −**v** − **w** is the other diagonal of the parallelogram formed by **v** and **w**.

2.2.2 SCALING

A vector can be multiplied component-wise by a scalar, thereby changing its length. Three functions are needed in order to cover all of the different ways that this operation may be written in source code (i.e., `v*s`, `s*v`, and `v *= s`):

⟨*Vector Public Methods*⟩ +≡ 57

```
Vector operator*(float f) const { return Vector(f*x, f*y, f*z); }

Vector &operator*=(float f) {
    x *= f; y *= f; z *= f;
    return *this;
}
```

⟨*Geometry Inline Functions*⟩ ≡

```
inline Vector operator*(float f, const Vector &v) { return v*f; }
```

Similarly, a vector can be divided component-wise by a scalar. The code for scalar division is similar to scalar multiplication, although division of a scalar by a vector is not well defined and so is not permitted.

In the implementation of these methods, we use a single division to compute the scalar's reciprocal, then perform three component-wise multiplications. This is a useful trick for avoiding division operations, which are generally much slower than multiplies on modern CPUs.[1]

Vector 57

1 It is a common misconception that these sorts of optimizations are unnecessary because the compiler will perform the necessary analysis. Compilers are frequently restricted from performing many optimizations of this type. For example, given two floating-point numbers, the quantities $a + b$ and $b + a$ are not candidates for common subexpression elimination because the IEEE floating-point standard (which is standard on modern architectures) does not guarantee that the two sums will be identical. In fact, compilers are limited in their ability to legally make these substitutions because expressions may have been carefully ordered in order to minimize round-off error. For division, IEEE requires that $x/x = 1$ for all x, but if we compute $1/x$ and store it in a variable and then multiply x by that value, it is not guaranteed that 1 will be the result. In this case, we are willing to lose that guarantee in exchange for higher performance.

We use the `Assert()` macro to make sure that the given scale is not zero; this should never happen and would indicate a bug elsewhere in the system.

⟨*Vector Public Methods*⟩ +≡ 57

```
Vector operator/(float f) const {
    Assert(f != 0);
    float inv = 1.f / f;
    return Vector(x * inv, y * inv, z * inv);
}

Vector &operator/=(float f) {
    Assert(f != 0);
    float inv = 1.f / f;
    x *= inv; y *= inv; z *= inv;
    return *this;
}
```

The `Vector` class also provides a unary negation operator that returns a new vector pointing in the opposite direction of the original one:

⟨*Vector Public Methods*⟩ +≡ 57

```
Vector operator-() const { return Vector(-x, -y, -z); }
```

Some routines will find it useful to be able to easily loop over the components of a `Vector`; the `Vector` class also provides a C++ operator so that, given a vector `v`, `v[0] == v.x` and so forth.

⟨*Vector Public Methods*⟩ +≡ 57

```
float operator[](int i) const {
    Assert(i >= 0 && i <= 2);
    return (&x)[i];
}

float &operator[](int i) {
    Assert(i >= 0 && i <= 2);
    return (&x)[i];
}
```

2.2.3 DOT AND CROSS PRODUCT

Two useful operations on vectors are the dot product (also known as the scalar or inner product) and the cross product. For two vectors **v** and **w**, their *dot product* $(\mathbf{v} \cdot \mathbf{w})$ is defined as:

$$\mathbf{v}_x\mathbf{w}_x + \mathbf{v}_y\mathbf{w}_y + \mathbf{v}_z\mathbf{w}_z.$$

⟨*Geometry Inline Functions*⟩ +≡

```
inline float Dot(const Vector &v1, const Vector &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

`Assert()` 1005
`Vector` 57

The dot product has a simple relationship to the angle between the two vectors:

$$(\mathbf{v} \cdot \mathbf{w}) = \|\mathbf{v}\| \, \|\mathbf{w}\| \cos\theta, \tag{2.1}$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$, and $\|\mathbf{v}\|$ denotes the length of the vector $\mathbf{v}$. It follows from this that $(\mathbf{v} \cdot \mathbf{w})$ is zero if and only if $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, provided that neither $\mathbf{v}$ nor $\mathbf{w}$ is *degenerate*—equal to (0, 0, 0). A set of two or more mutually perpendicular vectors is said to be *orthogonal*. An orthogonal set of unit vectors is called *orthonormal*.

It immediately follows from Equation (2.1) that if $\mathbf{v}$ and $\mathbf{w}$ are unit vectors, their dot product is exactly the cosine of the angle between them. As the cosine of the angle between two vectors often needs to be computed in computer graphics, we will frequently make use of this property.

A few basic properties directly follow from the definition. For example, if $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are vectors and s is a scalar value, then:

$$\begin{aligned}(\mathbf{u} \cdot \mathbf{v}) &= (\mathbf{v} \cdot \mathbf{u}) \\ (s\mathbf{u} \cdot \mathbf{v}) &= s(\mathbf{v} \cdot \mathbf{u}) \\ (\mathbf{u} \cdot (\mathbf{v} + \mathbf{w})) &= (\mathbf{u} \cdot \mathbf{v}) + (\mathbf{u} \cdot \mathbf{w}).\end{aligned}$$

We will frequently need to compute the absolute value of the dot product as well. The `AbsDot()` function does this for us so that we don't need a separate call to `fabsf()`:

⟨*Geometry Inline Functions*⟩ +≡
```
inline float AbsDot(const Vector &v1, const Vector &v2) {
    return fabsf(Dot(v1, v2));
}
```

The *cross product* is another useful vector operation. Given two vectors in 3D, the cross product $\mathbf{v}\times\mathbf{w}$ is a vector that is perpendicular to both of them. Note that this new vector can point in one of two directions; the coordinate system's handedness determines which is appropriate. Given orthogonal vectors $\mathbf{v}$ and $\mathbf{w}$, then $\mathbf{v}\times\mathbf{w}$ is defined to be a vector such that $(\mathbf{v}, \mathbf{w}, \mathbf{v}\times\mathbf{w})$ form a coordinate system of the appropriate handedness.

In a left-handed coordinate system, the cross product is defined as:

$$\begin{aligned}(\mathbf{v}\times\mathbf{w})_x &= \mathbf{v}_y\mathbf{w}_z - \mathbf{v}_z\mathbf{w}_y \\ (\mathbf{v}\times\mathbf{w})_y &= \mathbf{v}_z\mathbf{w}_x - \mathbf{v}_x\mathbf{w}_z \\ (\mathbf{v}\times\mathbf{w})_z &= \mathbf{v}_x\mathbf{w}_y - \mathbf{v}_y\mathbf{w}_x.\end{aligned}$$

An easy way to remember this is to compute the determinant of the matrix:

$$\mathbf{v}\times\mathbf{w} = \begin{vmatrix} i & j & k \\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z \\ \mathbf{w}_x & \mathbf{w}_y & \mathbf{w}_z \end{vmatrix},$$

AbsDot() 61
Dot() 60
Vector 57

where i, j, and k represent the axes (1, 0, 0), (0, 1, 0), and (0, 0, 1), respectively. Note that this equation is merely a memory aid and not a rigorous mathematical construction, since the matrix entries are a mix of scalars and vectors.

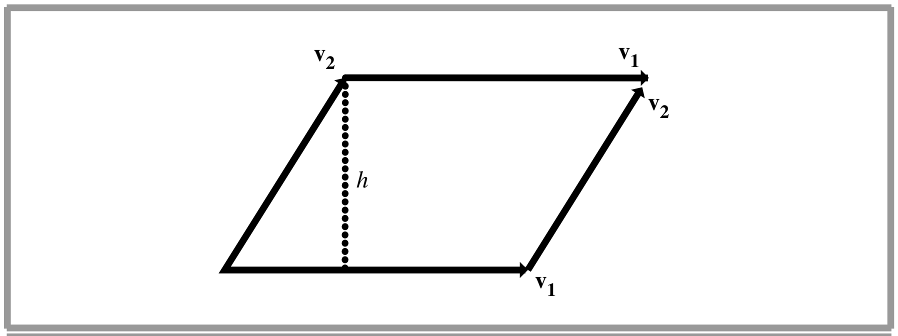


Figure 2.5: The area of a parallelogram with edges given by vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is equal to $\mathbf{v}_1 h$. From Equation (2.2), the length of the cross product of $\mathbf{v}_1$ and $\mathbf{v}_2$ is equal to the product of the two vector lengths times the sine of the angle between them—the parallelogram area.

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector Cross(const Vector &v1, const Vector &v2) {
    return Vector((v1.y * v2.z) - (v1.z * v2.y),
                  (v1.z * v2.x) - (v1.x * v2.z),
                  (v1.x * v2.y) - (v1.y * v2.x));
}
```

From the definition of the cross product, we can derive

$$\|\mathbf{v}\times\mathbf{w}\| = \|\mathbf{v}\|\ \|\mathbf{w}\| \sin\theta, \tag{2.2}$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$. An important implication of this is that the cross product of two perpendicular unit vectors is itself a unit vector. Note also that the result of the cross product is a degenerate vector if $\mathbf{v}$ and $\mathbf{w}$ are parallel.

This definition also shows a convenient way to compute the area of a parallelogram (Figure 2.5). If the two edges of the parallelogram are given by vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, and it has height h, the area is given by $\|\mathbf{v}_1\| h$. Since $h = \sin\theta \|\mathbf{v}_2\|$, we can use Equation (2.2) to see that the area is $\|\mathbf{v}_1 \times \mathbf{v}_2\|$.

2.2.4 NORMALIZATION

It is often necessary to *normalize* a vector—that is, to compute a new vector pointing in the same direction but with unit length. A normalized vector is often called a *unit vector*. The function to do this is called `Vector::Normalize()`. The notation used in this book for normalized vectors is that $\hat{\mathbf{v}}$ is the normalized version of $\mathbf{v}$. `Normalize()` divides each component by the length of the vector, $\|\mathbf{v}\|$.

⟨*Vector Public Methods*⟩ +≡ 57
```
float LengthSquared() const { return x*x + y*y + z*z; }
float Length() const { return sqrtf(LengthSquared()); }
```

Vector 57
Vector::Normalize() 63

`Normalize()` returns a new vector; it does *not* normalize the vector in place:

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector Normalize(const Vector &v) { return v / v.Length(); }
```

2.2.5 COORDINATE SYSTEM FROM A VECTOR

We will frequently want to construct a local coordinate system given only a single vector. Because the cross product of two vectors is orthogonal to both, we can apply the cross product two times to get a set of three orthogonal vectors for the coordinate system. Note that the two vectors generated by this technique are unique only up to a rotation about the given vector.

The implementation of this function assumes that the vector passed in, `v1`, has already been normalized. It first constructs a perpendicular vector by zeroing one of the components of the original vector, swapping the remaining two, and negating one of them. Inspection of the two cases should make clear that `v2` will be normalized and that the dot product $(\mathbf{v}_1 \cdot \mathbf{v}_2)$ must be equal to zero. Given these two perpendicular vectors, a single cross product gives the third, which by definition will be perpendicular to the first two.

⟨*Geometry Inline Functions*⟩ +≡
```
inline void CoordinateSystem(const Vector &v1, Vector *v2, Vector *v3) {
    if (fabsf(v1.x) > fabsf(v1.y)) {
        float invLen = 1.f / sqrtf(v1.x*v1.x + v1.z*v1.z);
        *v2 = Vector(-v1.z * invLen, 0.f, v1.x * invLen);
    }
    else {
        float invLen = 1.f / sqrtf(v1.y*v1.y + v1.z*v1.z);
        *v2 = Vector(0.f, v1.z * invLen, -v1.y * invLen);
    }
    *v3 = Cross(v1, *v2);
}
```

2.3 POINTS

⟨*Geometry Declarations*⟩ +≡
```
class Point {
public:
    ⟨Point Public Methods 64⟩
    ⟨Point Public Data 63⟩
};
```

A point is a zero-dimensional location in 3D space. The `Point` class in `pbrt` represents points in the obvious way: using x, y, and z coordinates with respect to their coordinate system. Although the same (x, y, z) representation is used for vectors, the fact that a point represents a position, whereas a vector represents a direction, leads to a number of important differences in how they are treated. Points are denoted in text by p.

Cross() 62
Point 63
Vector 57

⟨*Point Public Data*⟩ ≡ 63
```
float x, y, z;
```

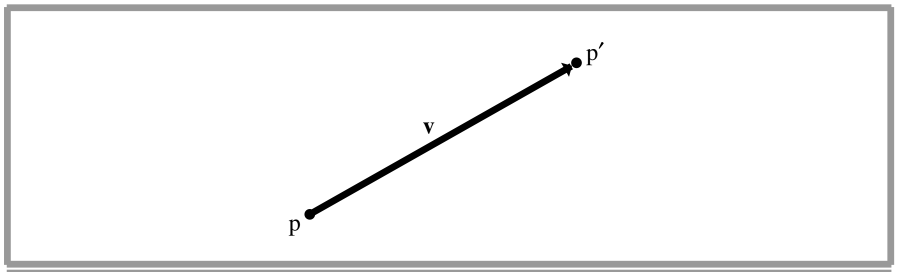


Figure 2.6: Obtaining the Vector between Two Points. The vector $p' - p$ is given by the component-wise subtraction of the points p' and p.

Like the `Vector` constructor, the `Point` constructor takes optional parameters to set the `x`, `y`, and `z` coordinate values.

⟨*Point Public Methods*⟩ ≡ 63

```
Point() { x = y = z = 0.f; }
Point(float xx, float yy, float zz)
    : x(xx), y(yy), z(zz) {
}
```

There are certain `Point` methods that either return or take a `Vector`. For instance, one can add a vector to a point, offsetting it in the given direction and obtaining a new point. Alternately, one can subtract one point from another, obtaining the vector between them, as shown in Figure 2.6.

⟨*Point Public Methods*⟩ +≡ 63

```
Point operator+(const Vector &v) const {
    return Point(x + v.x, y + v.y, z + v.z);
}

Point &operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

⟨*Point Public Methods*⟩ +≡ 63

```
Vector operator-(const Point &p) const {
    return Vector(x - p.x, y - p.y, z - p.z);
}

Point operator-(const Vector &v) const {
    return Point(x - v.x, y - v.y, z - v.z);
}
```

`Point` 63
`Vector` 57

```
Point &operator-=(const Vector &v) {
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

The distance between two points is easily computed by subtracting them to compute the vector between them and then finding the length of that vector:

⟨*Geometry Inline Functions*⟩ +≡

```
inline float Distance(const Point &p1, const Point &p2) {
    return (p1 - p2).Length();
}
inline float DistanceSquared(const Point &p1, const Point &p2) {
    return (p1 - p2).LengthSquared();
}
```

Although in general it doesn't make sense mathematically to weight points by a scalar or add two points together, the `Point` class still allows these operations in order to be able to compute weighted sums of points, which is mathematically meaningful as long as the weights used all sum to one. The code for scalar multiplication and addition with `Point`s is identical to `Vector`s, so it is not shown here.

2.4 NORMALS

⟨*Geometry Declarations*⟩ +≡

```
class Normal {
public:
    ⟨Normal Public Methods 66⟩
    ⟨Normal Public Data⟩
};
```

A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position. It can be defined as the cross product of any two nonparallel vectors that are tangent to the surface at a point. Although normals are superficially similar to vectors, it is important to distinguish between the two of them: because normals are defined in terms of their relationship to a particular surface, they behave differently than vectors in some situations, particularly when applying transformations. This difference is discussed in Section 2.8.

The implementations of `Normal`s and `Vector`s are very similar. Like vectors, normals are represented by three `float`s `x`, `y`, and `z`; they can be added and subtracted to compute new normals; and they can be scaled and normalized. However, a normal cannot be added to a point, and one cannot take the cross product of two normals. Note that, in an unfortunate turn of terminology, normals are *not* necessarily normalized.

Normal 65
Point 63
Vector 57

The `Normal` provides an extra constructor that initializes a `Normal` from a `Vector`. Because `Normal`s and `Vector`s are different in subtle ways, we want to make sure that this conversion doesn't happen when we don't intend it to, but we'd like it to be possible where it is appropriate. Fortunately, the C++ `explicit` keyword ensures that conversion between two compatible types only happens when the programmer explicitly requests such a conversion. `Vector` also provides a constructor that converts the other way. Thus, given the declarations `Vector v; Normal n;`, the assignment `n = v` is illegal, so it is necessary to explicitly convert the vector, as in `n = Normal(v)`.

⟨*Normal Public Methods*⟩ ≡ **65**
```
explicit Normal(const Vector &v)
  : x(v.x), y(v.y), z(v.z) {
}
```

⟨*Vector Public Methods*⟩ +≡ **57**
```
explicit Vector(const Normal &n);
```

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector::Vector(const Normal &n)
  : x(n.x), y(n.y), z(n.z) {
}
```

The `Dot()` and `AbsDot()` functions are also overloaded to compute dot products between the various possible combinations of normals and vectors. This code won't be included in the text here. We also won't include implementations of all of the various other `Normal` methods here, since they are similar to those for vectors.

It's often necessary to flip a surface normal so that it lies in the same hemisphere as a given vector—for example, the surface normal that lies in the same hemisphere as an outgoing ray is frequently needed. The `Faceforward()` utility function encapsulates this small computation. `pbrt` also provides variants of this function for the other three combinations of `Vector`s and `Normal`s as parameters. Be careful when using the other instances, though: when using the version that takes two `Vector`s, for example, ensure that the first parameter is the one that should be returned (possibly flipped) and the second is the one to test against. Reversing the two parameters will give unexpected results.

⟨*Geometry Inline Functions*⟩ +≡
```
inline Normal Faceforward(const Normal &n, const Vector &v) {
    return (Dot(n, v) < 0.f) ? -n : n;
}
```

2.5 RAYS

⟨*Geometry Declarations*⟩ +≡
```
class Ray {
public:
    ⟨Ray Public Methods 68⟩
    ⟨Ray Public Data 67⟩
};
```

A *ray* is a semi-infinite line specified by its origin and direction. `pbrt` represents a `Ray` with a `Point` for the origin and a `Vector` for the direction. A ray is denoted by r; it has origin o and direction **d**, as shown in Figure 2.7. Because we will be referring to these variables

AbsDot() 61
Dot() 60
Normal 65
Point 63
Ray 66
Vector 57

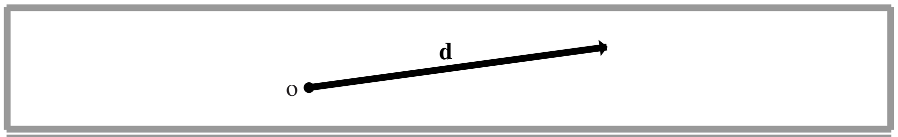


Figure 2.7: A ray is a semi-infinite line defined by its origin o and its direction vector **d**.

often throughout the code, the origin and direction members of a `Ray` are named `o` and `d`. Note that we again choose to make the data publicly available for convenience.

⟨*Ray Public Data*⟩ ≡ 66
```
Point o;
Vector d;
```

The *parametric form* of a ray expresses it as a function of a scalar value t, giving the set of points that the ray passes through:

$$\mathrm{r}(t) = \mathrm{o} + t\mathbf{d} \qquad 0 \le t \le \infty. \tag{2.3}$$

The `Ray` also includes fields to limit the ray to a particular segment along its infinite extent. These fields, called `mint` and `maxt`, allow us to restrict the ray to a segment of points [r(mint), r(maxt)]. Notice that these fields are declared as `mutable`, meaning that they can be changed even if the `Ray` structure that contains them is `const`.

Thus, when a ray is passed to a method that takes a `const Ray &`, that method is not allowed to modify its origin or direction but can modify its parametric range. This convention fits one of the most common uses of rays in the system, as parameters to ray–object intersection testing routines, which will record the offsets to the closest intersection in `maxt`.

⟨*Ray Public Data*⟩ +≡ 66
```
mutable float mint, maxt;
```

For simulating motion blur in scenes with animated objects, each ray may have a time value associated with it, so a data member to store time is available as well. The rendering system is responsible for constructing a representation of the scene at the appropriate time for each ray.

⟨*Ray Public Data*⟩ +≡ 66
```
float time;
```

In many light transport algorithms, a path of multiple rays is followed through the scene. A light path might start at a light source and scatter off of a few surfaces before it reached the camera, for example. The ray `depth` member variable allows light transport algorithms to track how many times light has bounced along the current path so that they can terminate the path after a particular number of bounces, etc.

Point 63
Ray 66
Vector 57

⟨*Ray Public Data*⟩ +≡ 66
```
int depth;
```

Constructing Rays is straightforward. The default constructor relies on the Point and Vector constructors to set the origin and direction to (0, 0, 0). Alternately, a particular point and direction can be provided. If an origin and direction are provided, the constructor requires that a value be given for mint as well (but maxt is initialized to infinity if not provided). The reason that mint is required is that it is generally set to a small greater-than-zero (but scene-specific) epsilon value, large enough so that a ray starting at the computed point of intersection between another ray and a surface doesn't incorrectly intersect the surface a very short distance along its extent. More details on this topic are in Section 3.1.4.

⟨*Ray Public Methods*⟩ ≡ 66

```
Ray() : mint(0.f), maxt(INFINITY), time(0.f), depth(0) { }
Ray(const Point &origin, const Vector &direction,
    float start, float end = INFINITY, float t = 0.f, int d = 0)
    : o(origin), d(direction), mint(start), maxt(end), time(t), depth(d) { }
```

For convenience we also provide another ray constructor that takes a "parent" ray; the idea is that when additional rays are being spawned from an intersection, it's useful to be able to copy the time value and set the depth value for the new ray based on the corresponding values from the previous ray.

⟨*Ray Public Methods*⟩ +≡ 66

```
Ray(const Point &origin, const Vector &direction, const Ray &parent,
    float start, float end = INFINITY)
    : o(origin), d(direction), mint(start), maxt(end),
      time(parent.time), depth(parent.depth+1) { }
```

Because a ray can be thought of as a function of a single parameter t, the Ray class overloads the function application operator for rays. This way, when we need to find the point at a particular position along a ray, we can write code like:

```
    Ray r(Point(0,0,0), Vector(1,2,3));
    Point p = r(1.7);
```

⟨*Ray Public Methods*⟩ +≡ 66

```
Point operator()(float t) const { return o + d * t; }
```

2.5.1 RAY DIFFERENTIALS

In order to be able to perform better antialiasing with the texture functions defined in Chapter 10, pbrt keeps track of some additional information with each ray that is traced. In Section 10.1, this information will be used to compute information so that the Texture class can estimate the projected area on the image plane of a small part of the scene. From this, the Texture can compute the texture's average value over that area, leading to a better final image.

RayDifferential is a subclass of Ray that contains additional information about two auxiliary rays. These extra rays represent camera rays offset one sample in the x and y direction from the main ray on the film plane. By determining the area that these three rays project to on an object being shaded, the Texture can estimate an area to average over for proper antialiasing.

INFINITY 1002
Point 63
Ray 66
RayDifferential 69
Texture 519
Vector 57

Because the `RayDifferential` class inherits from `Ray`, geometric interfaces in the system are written to take `const Ray &` parameters, so that either a `Ray` or `RayDifferential` can be passed to them. Only the routines related to antialiasing and texturing require `RayDifferential` parameters.

⟨*Geometry Declarations*⟩ +≡

```
class RayDifferential : public Ray {
public:
    ⟨RayDifferential Public Methods 69⟩
    ⟨RayDifferential Public Data 69⟩
};
```

The `RayDifferential` constructors mirror the `Ray`'s constructors.

⟨*RayDifferential Public Methods*⟩ ≡ **69**

```
RayDifferential() { hasDifferentials = false; }
RayDifferential(const Point &org, const Vector &dir, float start,
    float end = INFINITY, float t = 0.f, int d = 0)
        : Ray(org, dir, start, end, t, d) {
    hasDifferentials = false;
}
RayDifferential(const Point &org, const Vector &dir, const Ray &parent,
    float start, float end = INFINITY)
        : Ray(org, dir, start, end, parent.time, parent.depth+1) {
    hasDifferentials = false;
}
```

We provide a constructor to create `RayDifferentials` from `Rays`, but use the `explicit` keyword to prevent `Rays` from accidentally being converted to `RayDifferentials`. The constructor sets `hasDifferentials` to false initially because the neighboring rays, if any, are not known.

⟨*RayDifferential Public Methods*⟩ +≡ **69**

```
explicit RayDifferential(const Ray &ray) : Ray(ray) {
    hasDifferentials = false;
}
```

⟨*RayDifferential Public Data*⟩ ≡ **69**

```
bool hasDifferentials;
Point rxOrigin, ryOrigin;
Vector rxDirection, ryDirection;
```

`Camera` implementations in `pbrt` compute differentials for rays leaving the camera under the assumption that camera rays are spaced one pixel apart. The `ScaleDifferentials()` method below updates the differential rays for an actual sample spacing `s`.

Camera 302
INFINITY 1002
Point 63
Ray 66
Ray::depth 67
Ray::time 67
RayDifferential 69
RayDifferential::hasDifferentials 69
Vector 57

⟨*RayDifferential Public Methods*⟩ +≡ 69

```
void ScaleDifferentials(float s) {
    rxOrigin = o + (rxOrigin - o) * s;
    ryOrigin = o + (ryOrigin - o) * s;
    rxDirection = d + (rxDirection - d) * s;
    ryDirection = d + (ryDirection - d) * s;
}
```

2.6 THREE-DIMENSIONAL BOUNDING BOXES

⟨*Geometry Declarations*⟩ +≡

```
class BBox {
public:
    ⟨BBox Public Methods 70⟩
    ⟨BBox Public Data 71⟩
};
```

The scenes that pbrt will render will often contain objects that are computationally expensive to process. For many operations, it is often useful to have a three-dimensional *bounding volume* that encloses an object. For example, if a ray does not pass through a particular bounding volume, pbrt can avoid processing all of the objects inside of it for that ray.

The measurable benefit of this technique is related to two factors: the expense of processing the bounding volume compared to the expense of processing the objects inside of it, and the tightness of the fit of the bounding box. If we have a very loose bound around an object, we will often incorrectly determine that its contents need to be examined further. However, in order to make the bounding volume a closer fit, it may be necessary to make the volume a complex object itself, thus increasing the expense of processing it.

There are many choices for bounding volumes; pbrt uses *axis-aligned bounding boxes* (AABBs). Other popular choices are spheres and *oriented bounding boxes* (OBBs). An AABB can be described by one of its vertices and three lengths, each representing the distance spanned along the x, y, and z coordinate axes. Alternatively, two opposite vertices of the box can describe it. We chose the two-point representation for pbrt's BBox class; it stores the positions of the vertex with minimum x, y, and z values and of the one with maximum x, y, and z. A 2D illustration of a bounding box and its representation is shown in Figure 2.8.

The default BBox constructor sets the extent to be degenerate; by violating the invariant that pMin.x <= pMax.x and similarly for the other dimensions, this convention ensures that any operations done with this box (e.g., Union()) will have the correct result for a completely empty box.

⟨*BBox Public Methods*⟩ ≡ 70

```
BBox() {
    pMin = Point( INFINITY,  INFINITY,  INFINITY);
    pMax = Point(-INFINITY, -INFINITY, -INFINITY);
}
```

BBox	70
BBox::pMax	71
BBox::pMin	71
INFINITY	1002
Point	63
Ray::d	67
Ray::o	67
RayDifferential::rxDirection	69
RayDifferential::rxOrigin	69
RayDifferential::ryDirection	69
RayDifferential::ryOrigin	69

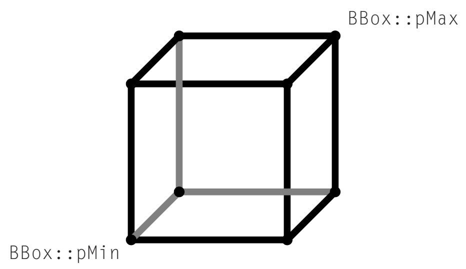


Figure 2.8: An Example Axis-Aligned Bounding Box. The `BBox` class stores only the coordinates of the minimum and maximum points of the box. All other box corners are implicit in this representation.

⟨*BBox Public Data*⟩ ≡ 70

```
Point pMin, pMax;
```

It is also useful to be able to initialize a `BBox` to enclose a single point:

⟨*BBox Public Methods*⟩ +≡ 70

```
BBox(const Point &p) : pMin(p), pMax(p) { }
```

If the caller passes two corner points (`p1` and `p2`) to define the box, since `p1` and `p2` are not necessarily chosen so that `p1.x <= p2.x`, and so on, the constructor needs to find their component-wise minimum and maximum values.

⟨*BBox Public Methods*⟩ +≡ 70

```
BBox(const Point &p1, const Point &p2) {
    pMin = Point(min(p1.x, p2.x), min(p1.y, p2.y), min(p1.z, p2.z));
    pMax = Point(max(p1.x, p2.x), max(p1.y, p2.y), max(p1.z, p2.z));
}
```

Given a bounding box and a point, the `BBox::Union()` function computes and returns a new bounding box that encompasses that point as well as the space that the original box encompassed:

⟨*BBox Method Definitions*⟩ ≡

```
BBox Union(const BBox &b, const Point &p) {
    BBox ret = b;
    ret.pMin.x = min(b.pMin.x, p.x);
    ret.pMin.y = min(b.pMin.y, p.y);
    ret.pMin.z = min(b.pMin.z, p.z);
    ret.pMax.x = max(b.pMax.x, p.x);
    ret.pMax.y = max(b.pMax.y, p.y);
    ret.pMax.z = max(b.pMax.z, p.z);
    return ret;
}
```

`BBox` 70
`BBox::pMax` 71
`BBox::pMin` 71
`BBox::Union()` 71
`Point` 63

It is similarly possible to construct a new box that bounds the space encompassed by two other bounding boxes. The definition of this function is similar to the earlier `Union()` method that takes a `Point`; the difference is that the `pMin` and `pMax` of the second box are used for the `min()` and `max()` tests, respectively.

⟨*BBox Public Methods*⟩ +≡ **70**

```
friend BBox Union(const BBox &b, const BBox &b2);
```

It is easy to determine if two `BBox`es overlap by seeing if their extents overlap in all of x, y, and z:

⟨*BBox Public Methods*⟩ +≡ **70**

```
bool Overlaps(const BBox &b) const {
    bool x = (pMax.x >= b.pMin.x) && (pMin.x <= b.pMax.x);
    bool y = (pMax.y >= b.pMin.y) && (pMin.y <= b.pMax.y);
    bool z = (pMax.z >= b.pMin.z) && (pMin.z <= b.pMax.z);
    return (x && y && z);
}
```

Three simple 1D containment tests determine if a given point is inside the bounding box:

⟨*BBox Public Methods*⟩ +≡ **70**

```
bool Inside(const Point &pt) const {
    return (pt.x >= pMin.x && pt.x <= pMax.x &&
            pt.y >= pMin.y && pt.y <= pMax.y &&
            pt.z >= pMin.z && pt.z <= pMax.z);
}
```

The `BBox::Expand()` method pads the bounding box by a constant factor.

⟨*BBox Public Methods*⟩ +≡ **70**

```
void Expand(float delta) {
    pMin -= Vector(delta, delta, delta);
    pMax += Vector(delta, delta, delta);
}
```

Methods for computing the surface area of the six faces of the box and the volume inside of it are also frequently useful.

⟨*BBox Public Methods*⟩ +≡ **70**

```
float SurfaceArea() const {
    Vector d = pMax - pMin;
    return 2.f * (d.x * d.y + d.x * d.z + d.y * d.z);
}
```

⟨*BBox Public Methods*⟩ +≡ **70**

```
float Volume() const {
    Vector d = pMax - pMin;
    return d.x * d.y * d.z;
}
```

`BBox` 70
`BBox::Expand()` 72
`BBox::pMax` 71
`BBox::pMin` 71
`Point` 63
`Vector` 57

The BBox::MaximumExtent() method tells the caller which of the three axes is longest. This is useful, for example, when deciding along which axis to subdivide when building a kd-tree (see Section A.8).

⟨*BBox Public Methods*⟩ +≡ 70
```
int MaximumExtent() const {
    Vector diag = pMax - pMin;
    if (diag.x > diag.y && diag.x > diag.z)
        return 0;
    else if (diag.y > diag.z)
        return 1;
    else
        return 2;
}
```

In some cases, it's also useful to use array indexing to select between the two points at the corners of the box.

⟨*BBox Public Methods*⟩ +≡ 70
```
const Point &operator[](int i) const;
Point &operator[](int i);
```

The Lerp() method linearly interpolates between the corners of the box by the given amount, and Offset() returns the position of a point relative to the corners of the box, where a point at the minimum corner has offset (0, 0, 0), a point at the maximum corner has offset (1, 1, 1), and so forth.

⟨*BBox Public Methods*⟩ +≡ 70
```
Point Lerp(float tx, float ty, float tz) const {
    return Point(::Lerp(tx, pMin.x, pMax.x), ::Lerp(ty, pMin.y, pMax.y),
                 ::Lerp(tz, pMin.z, pMax.z));
}
```

⟨*BBox Public Methods*⟩ +≡ 70
```
Vector Offset(const Point &p) const {
    return Vector((p.x - pMin.x) / (pMax.x - pMin.x),
                  (p.y - pMin.y) / (pMax.y - pMin.y),
                  (p.z - pMin.z) / (pMax.z - pMin.z));
}
```

BBox 70
BBox::MaximumExtent() 73
BBox::pMax 71
BBox::pMin 71
Lerp() 1000
Point 63
Vector 57

Finally, the BBox provides a method that returns the center and radius of a sphere that bounds the bounding box. In general, this may give a far looser fit than a sphere that bounded the original contents of the BBox directly, although it is a useful method to have available. For example, in Chapter 14, we use this method to get a sphere that completely bounds the scene in order to generate a random ray that is likely to intersect the scene geometry.

⟨*BBox Method Definitions*⟩ +≡
```
void BBox::BoundingSphere(Point *c, float *rad) const {
    *c = .5f * pMin + .5f * pMax;
    *rad = Inside(*c) ? Distance(*c, pMax) : 0.f;
}
```

2.7 TRANSFORMATIONS

In general, a *transformation* **T** is a mapping from points to points and from vectors to vectors:

$$\mathrm{p}' = \mathbf{T}(\mathrm{p}) \qquad \mathbf{v}' = \mathbf{T}(\mathbf{v}).$$

The transformation **T** may be an arbitrary procedure. However, we will consider a subset of all possible transformations in this chapter. In particular, they will be

- *Linear:* If **T** is an arbitrary linear transformation and s is an arbitrary scalar, then $\mathbf{T}(s\mathbf{v}) = s\mathbf{T}(\mathbf{v})$ and $\mathbf{T}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{T}(\mathbf{v}_1) + \mathbf{T}(\mathbf{v}_2)$. These two properties can greatly simplify reasoning about transformations.
- *Continuous:* Roughly speaking, **T** maps the neighborhoods around p and **v** to neighborhoods around p′ and **v**′.
- *One-to-one and invertible:* For each p, **T** maps p to a single unique p′. Furthermore, there exists an inverse transform $\mathbf{T}^{-1}$ that maps p′ back to p.

We will often want to take a point, vector, or normal defined with respect to one coordinate frame and find its coordinate values with respect to another frame. Using basic properties of linear algebra, a 4 × 4 matrix can be shown to express the linear transformation of a point or vector from one frame to another. Furthermore, such a 4 × 4 matrix suffices to express all linear transformations of points and vectors within a fixed frame, such as translation in space or rotation around a point. Therefore, there are different (and incompatible!) ways that a matrix can be interpreted:

- *Transformation of the frame:* Given a point, the matrix could express how to compute a *new* point in the same frame that represents the transformation of the original point (e.g., by translating it in some direction).
- *Transformation from one frame to another:* A matrix can express the coordinates of a point or vector in a new frame in terms of the coordinates in the original frame.

Most uses of transformations in `pbrt` are for transforming points from one frame to another.

In general, transformations make it possible to work in the most convenient coordinate space. For example, we can write routines that define a virtual camera assuming that the camera is located at the origin, looks down the z axis, and has the y axis pointing up and the x axis pointing right. These assumptions greatly simplify the camera implementation. Then, to place the camera at any point in the scene looking in any direction, we just construct a transformation that maps points in the scene's coordinate system to the camera's coordinate system. (See Section 6.1.1 for more information about camera coordinate spaces in `pbrt`.)

`BBox` 70
`BBox::Inside()` 72
`BBox::pMax` 71
`BBox::pMin` 71
`Distance()` 65
`Point` 63

2.7.1 HOMOGENEOUS COORDINATES

Given a frame defined by $(\mathrm{p}, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$, there is ambiguity between the representation of a point $(\mathrm{p}_x, \mathrm{p}_y, \mathrm{p}_z)$ and a vector $(\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z)$ with the same (x, y, z) coordinates. Using the representations of points and vectors introduced at the start of the chapter, we can write the point as the inner product $[s_1\, s_2\, s_3\, 1][\mathbf{v}_1\, \mathbf{v}_2\, \mathbf{v}_3\, \mathrm{p_o}]^T$ and the vector as the inner product $[s'_1\, s'_2\, s'_3\, 0][\mathbf{v}_1\, \mathbf{v}_2\, \mathbf{v}_3\, \mathrm{p_o}]^T$. These four vectors of three s_i values and a zero or one are called the *homogeneous* representations of the point and the vector. The fourth coordinate of the homogeneous representation is sometimes called the *weight*. For a point, its value can be any scalar other than zero: the homogeneous points $[1, 3, -2, 1]$ and $[-2, -6, 4, -2]$ describe the same Cartesian point $(1, 3, -2)$. In general, homogeneous points obey the identity

$$(x, y, z, w) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right).$$

We will use these facts to see how a transformation matrix can describe how points and vectors in one frame can be mapped to another frame. Consider a matrix $\mathbf{M}$ that describes the transformation from one coordinate system to another:

$$\mathbf{M} = \begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{pmatrix}.$$

(In this book, we define matrix element indices starting from zero, so that equations and source code correspond more directly.) Then if the transformation represented by $\mathbf{M}$ is applied to the x axis vector $(1, 0, 0)$, we have

$$\mathbf{Mx} = \mathbf{M}[1\,0\,0\,0]^T = [m_{0,0}\, m_{1,0}\, m_{2,0}\, m_{3,0}]^T.$$

Thus, directly reading the columns of the matrix shows how the basis vectors and the origin of the current coordinate system are transformed by the matrix:

$$\mathbf{My} = [m_{0,1}\, m_{1,1}\, m_{2,1}\, m_{3,1}]^T$$
$$\mathbf{Mz} = [m_{0,2}\, m_{1,2}\, m_{2,2}\, m_{3,2}]^T$$
$$\mathbf{M}\mathrm{p} = [m_{0,3}\, m_{1,3}\, m_{2,3}\, m_{3,3}]^T.$$

In general, by characterizing how the basis is transformed, we know how any point or vector specified in terms of that basis is transformed. Because points and vectors in the current coordinate system are expressed in terms of the current coordinate system's frame, applying the transformation to them directly is equivalent to applying the transformation to the current coordinate system's basis and finding their coordinates in terms of the transformed basis.

We will not use homogeneous coordinates explicitly in our code; there is no `Homogeneous` class. However, the various transformation routines in the next section will implicitly convert points, vectors, and normals to homogeneous form, transform the homogeneous points, and then convert them back before returning the result. This isolates the details of homogeneous coordinates in one place (namely, the implementation of transformations).

```
⟨Transform Declarations⟩ ≡
    class Transform {
    public:
        ⟨Transform Public Methods 77⟩
    private:
        ⟨Transform Private Data 76⟩
    };
```

A transformation is represented by the elements of the matrix `m[4][4]`, a `Matrix4x4` object. The low-level `Matrix4x4` class is defined in Section A.6.2. `m` is stored in *row-major* form, so element `m[i][j]` corresponds to $m_{i,j}$, where i is the row number and j is the column number. For convenience, the `Transform` also stores the inverse of the matrix `m` in the `Transform::mInv` member; for `pbrt`'s needs, it is better to have the inverse easily available than to repeatedly compute it as needed.

This representation of transformations is relatively memory hungry: assuming 4 bytes of storage for a `float` value, a `Transform` requires 128 bytes of storage. Used naïvely, this approach can be wasteful; if a scene has millions of shapes but only a few thousand unique transformations, there's no reason to redundantly store the same transform many times in memory. Therefore, `Shape`s in `pbrt` store a pointer to a `Transform`, and the scene specification code defined in Section B.3.5 uses a `TransformCache` to ensure that all shapes that share the same transformation point to a single instance of that transformation in memory.

This decision to share transformations implies a loss of flexibility, however: the elements of a `Transform` shouldn't be modified after it is created if the `Transform` is shared by multiple objects in the scene (and those objects don't expect it to be changing.) This limitation isn't a problem in practice, since the transformations in a scene are typically created when `pbrt` parses the scene description file and don't need to change later at rendering time.

```
⟨Transform Private Data⟩ ≡                                              76
    Matrix4x4 m, mInv;
```

2.7.2 BASIC OPERATIONS

When a new `Transform` is created, it defaults to the *identity transformation*—the transformation that maps each point and each vector to itself. This transformation is represented by the *identity matrix*:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The implementation here relies on the default `Matrix4x4` constructor to fill in the identity matrix for `m` and `mInv`.

`Matrix4x4`	1021
`Shape`	108
`Transform`	76
`Transform::mInv`	76
`TransformCache`	1066

⟨*Transform Public Methods*⟩ ≡ 76
```
Transform() { }
```

A `Transform` can also be created from a given matrix. In this case, the given matrix must be explicitly inverted.

⟨*Transform Public Methods*⟩ +≡ 76
```
Transform(const float mat[4][4]) {
    m = Matrix4x4(mat[0][0], mat[0][1], mat[0][2], mat[0][3],
                  mat[1][0], mat[1][1], mat[1][2], mat[1][3],
                  mat[2][0], mat[2][1], mat[2][2], mat[2][3],
                  mat[3][0], mat[3][1], mat[3][2], mat[3][3]);
    mInv = Inverse(m);
}
```

⟨*Transform Public Methods*⟩ +≡ 76
```
Transform(const Matrix4x4 &mat)
    : m(mat), mInv(Inverse(mat)) {
}
```

The most commonly used constructor takes a reference to the transformation matrix along with an explicitly provided inverse. This is a superior approach to computing the inverse in the constructor because many geometric transformations have very simple inverses and we can avoid the expense and potential loss of numeric precision from computing a general 4×4 matrix inverse. Of course, this places the burden on the caller to make sure that the supplied inverse is correct.

⟨*Transform Public Methods*⟩ +≡ 76
```
Transform(const Matrix4x4 &mat, const Matrix4x4 &minv)
   : m(mat), mInv(minv) {
}
```

A `Transform` representing the inverse of a `Transform` can be returned by just swapping the roles of `mInv` and `m`.

⟨*Transform Public Methods*⟩ +≡ 76
```
friend Transform Inverse(const Transform &t) {
    return Transform(t.mInv, t.m);
}
```

We provide `Transform` equality (and inequality) testing methods; their implementation is straightforward and not included here. `Transform` also provides an `IsIdentity()` method that checks to see if the transformation is the identity.

`Inverse()` 1021
`Matrix4x4` 1021
`Transform` 76
`Transform::m` 76
`Transform::mInv` 76

2.7.3 TRANSLATIONS

One of the simplest transformations is the *translation transformation*, $\mathrm{T}(\Delta x, \Delta y, \Delta z)$. When applied to a point p, it translates p's coordinates by Δx, Δy, and Δz, as shown in Figure 2.9. As an example, $\mathrm{T}(2, 2, 1)(x, y, z) = (x + 2, y + 2, z + 1)$.

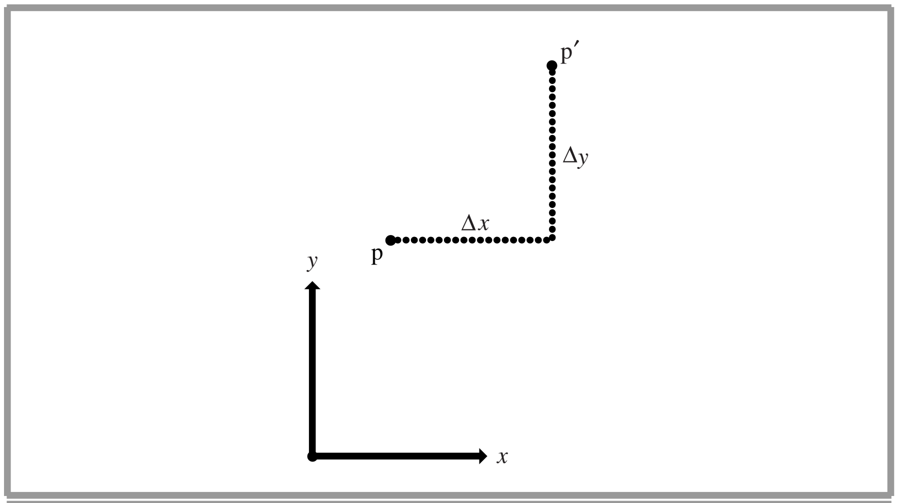


Figure 2.9: Translation in 2D. Adding offsets Δx and Δy to a point's coordinates correspondingly changes its position in space.

Translation has some simple properties:

$$\begin{aligned}
\mathrm{T}(0, 0, 0) &= \mathrm{I} \\
\mathrm{T}(x_1, y_1, z_1)\mathrm{T}(x_2, y_2, z_2) &= \mathrm{T}(x_1 + x_2, y_1 + y_2, z_1 + z_2) \\
\mathrm{T}(x_1, y_1, z_1)\mathrm{T}(x_2, y_2, z_2) &= \mathrm{T}(x_2, y_2, z_2)\mathrm{T}(x_1, y_1, z_1) \\
\mathrm{T}^{-1}(x, y, z) &= \mathrm{T}(-x, -y, -z).
\end{aligned}$$

Translation only affects points, leaving vectors unchanged. In matrix form, the translation transformation is

$$\mathrm{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

When we consider the operation of a translation matrix on a point, we see the value of homogeneous coordinates. Consider the product of the matrix for $\mathrm{T}(\Delta x, \Delta y, \Delta z)$ with a point p in homogeneous coordinates $[x\ y\ z\ 1]$:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix}.$$

As expected, we have computed a new point with its coordinates offset by $(\Delta x, \Delta y, \Delta z)$. However, if we apply T to a vector **v**, we have

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}.$$

The result is the same vector **v**. This makes sense because vectors represent directions, so translation leaves them unchanged.

We will define a routine that creates a new `Transform` matrix to represent a given translation—it is a straightforward application of the translation matrix equation. This routine fully initializes the `Transform` that is returned, also initializing the matrix that represents the inverse of the translation.

⟨*Transform Method Definitions*⟩ ≡

```
Transform Translate(const Vector &delta) {
    Matrix4x4 m(1, 0, 0, delta.x,
                0, 1, 0, delta.y,
                0, 0, 1, delta.z,
                0, 0, 0,       1);
    Matrix4x4 minv(1, 0, 0, -delta.x,
                   0, 1, 0, -delta.y,
                   0, 0, 1, -delta.z,
                   0, 0, 0,        1);
    return Transform(m, minv);
}
```

2.7.4 SCALING

Another basic transformation is the *scale transformation*, $\mathrm{S}(s_x, s_y, s_z)$. It has the effect of taking a point or vector and multiplying its components by scale factors in x, y, and z: $\mathrm{S}(2, 2, 1)(x, y, z) = (2x, 2y, z)$. It has the following basic properties:

$$\mathrm{S}(1, 1, 1) = \mathbf{I}$$
$$\mathrm{S}(x_1, y_1, z_1)\mathrm{S}(x_2, y_2, z_2) = \mathrm{S}(x_1x_2, y_1y_2, z_1z_2)$$
$$\mathrm{S}^{-1}(x, y, z) = \mathrm{S}\left(\frac{1}{x}, \frac{1}{y}, \frac{1}{z}\right).$$

We can differentiate between *uniform scaling*, where all three scale factors have the same value, and *nonuniform scaling*, where they may have different values. The general scale matrix is

$$\mathrm{S}(x, y, z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Matrix4x4 1021
Transform 76
Vector 57

⟨*Transform Method Definitions*⟩ +≡
```
Transform Scale(float x, float y, float z) {
    Matrix4x4 m(x, 0, 0, 0,
                0, y, 0, 0,
                0, 0, z, 0,
                0, 0, 0, 1);
    Matrix4x4 minv(1.f/x,    0,     0,    0,
                   0,     1.f/y,    0,    0,
                   0,        0,     1.f/z, 0,
                   0,        0,     0,    1);
    return Transform(m, minv);
}
```

It's useful to be able to test if a transformation has a scaling term in it; an easy way to do this is to transform the three coordinate axes and see if any of their lengths are appreciably different from one.

⟨*Transform Public Methods*⟩ +≡ 76
```
bool HasScale() const {
    float la2 = (*this)(Vector(1,0,0)).LengthSquared();
    float lb2 = (*this)(Vector(0,1,0)).LengthSquared();
    float lc2 = (*this)(Vector(0,0,1)).LengthSquared();
#define NOT_ONE(x) ((x) < .999f || (x) > 1.001f)
    return (NOT_ONE(la2) || NOT_ONE(lb2) || NOT_ONE(lc2));
#undef NOT_ONE
}
```

2.7.5 x, y, AND z AXIS ROTATIONS

Another useful type of transformation is the *rotation transformation*, **R**. In general, we can define an arbitrary axis from the origin in any direction and then rotate around that axis by a given angle. The most common rotations of this type are around the x, y, and z coordinate axes. We will write these rotations as $\mathbf{R}_x(\theta)$, $\mathbf{R}_y(\theta)$, and so on. The rotation around an arbitrary axis (x, y, z) is denoted by $\mathbf{R}_{(x,y,z)}(\theta)$.

Rotations also have some basic properties:

$$\begin{aligned}\mathbf{R}_a(0) &= \mathbf{I}\\ \mathbf{R}_a(\theta_1)\mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_1+\theta_2)\\ \mathbf{R}_a(\theta_1)\mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_2)\mathbf{R}_a(\theta_1)\\ \mathbf{R}_a^{-1}(\theta) = \mathbf{R}_a(-\theta) &= \mathbf{R}_a^T(\theta),\end{aligned}$$

where $\mathbf{R}^T$ is the matrix transpose of **R**. This last property, that the inverse of **R** is equal to its transpose, stems from the fact that **R** is an *orthogonal matrix*; its upper 3×3 components are all orthogonal to each other. Fortunately, the transpose is much easier to compute than a full matrix inverse.

For a left-handed coordinate system, the matrix for rotation around the x axis is

Matrix4x4 1021
Transform 76
Vector 57
Vector::LengthSquared() 62

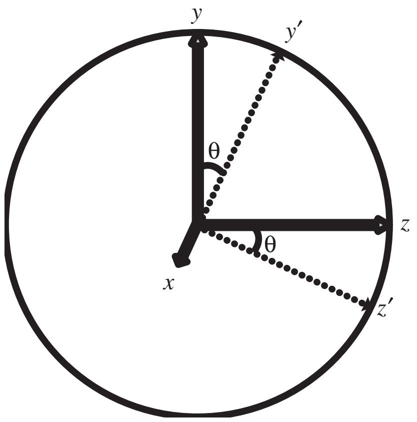


Figure 2.10: Rotation by an angle θ about the x axis leaves the x coordinate unchanged. The y and z axes are mapped to the vectors given by the dashed lines; y and z coordinates move accordingly.

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Figure 2.10 gives an intuition for how this matrix works. It's easy to see that it leaves the x axis unchanged:

$$\mathbf{R}_x(\theta)[1\,0\,0\,0]^T = [1\,0\,0\,0]^T.$$

It maps the y axis $(0, 1, 0)$ to $(0, \cos\theta, \sin\theta)$ and the z axis to $(0, -\sin\theta, \cos\theta)$. The y and z axes remain in the same plane, perpendicular to the x axis, but are rotated by the given angle. An arbitrary point in space is similarly rotated about the x axis by this transformation while staying in the same yz plane as it was originally.

The implementation of the `RotateX()` function is straightforward.

⟨*Transform Method Definitions*⟩ +≡

```
Transform RotateX(float angle) {
    float sin_t = sinf(Radians(angle));
    float cos_t = cosf(Radians(angle));
    Matrix4x4 m(1,     0,      0, 0,
                0, cos_t, -sin_t, 0,
                0, sin_t,  cos_t, 0,
                0,     0,      0, 1);
    return Transform(m, Transpose(m));
}
```

Matrix4x4 1021
RotateX() 81
Transform 76
Transpose() 1021

Similarly, for rotation around y and z, we have

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The implementations of `RotateY()` and `RotateZ()` follow directly and are not included here.

2.7.6 ROTATION AROUND AN ARBITRARY AXIS

We also provide a routine to compute the transformation that represents rotation around an arbitrary axis. The usual derivation of this matrix is based on computing rotations that map the given axis to a fixed axis (e.g., z), performing the rotation there, and then rotating the fixed axis back to the original axis. A more elegant derivation can be constructed with vector algebra.

Consider a normalized direction vector $\mathbf{a}$ that gives the axis to rotate around by angle θ, and a vector $\mathbf{v}$ to be rotated (Figure 2.11). First, we can compute the vector $\mathbf{v_c}$ along the axis $\mathbf{a}$ that is in the plane through the end point of $\mathbf{v}$ and is parallel to $\mathbf{a}$. Assuming $\mathbf{v}$ and $\mathbf{a}$ form an angle α, we have

$$\mathbf{v_c} = \mathbf{a}\,\|\mathbf{v}\|\,\cos\alpha = \mathbf{a}(\mathbf{v}\cdot\mathbf{a}).$$

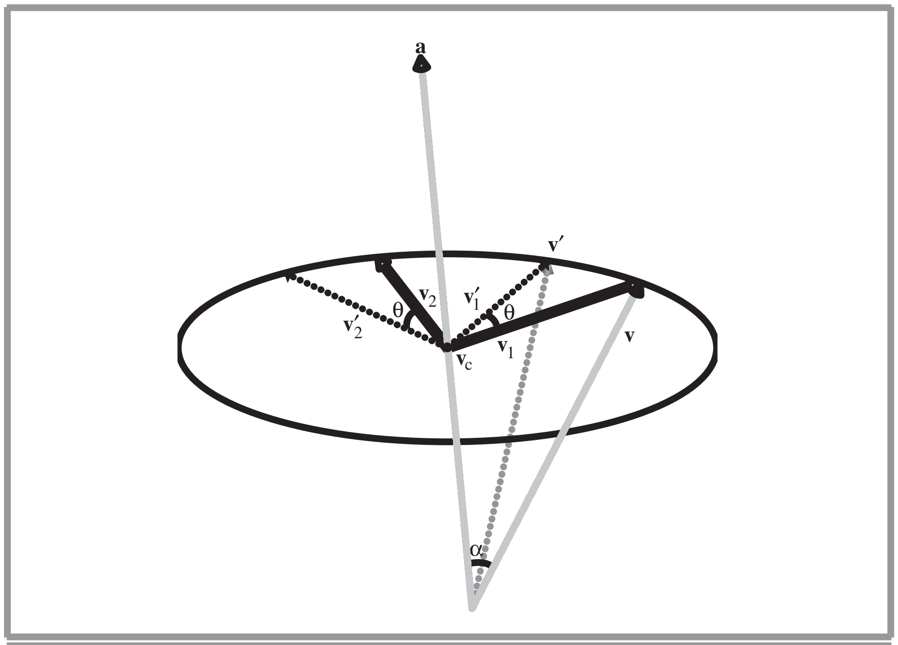


Figure 2.11: A vector $\mathbf{v}$ can be rotated around an arbitrary axis $\mathbf{a}$ by constructing a coordinate system $(\mathrm{p}, \mathbf{v}_1, \mathbf{v}_2)$ in the plane perpendicular to the axis that passes through $\mathbf{v}$'s end point and rotating the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ about p. Applying this rotation to the axes of the coordinate system $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ gives the general rotation matrix for this rotation.

We now compute a pair of basis vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ in this plane. Trivially, one of them is

$$\mathbf{v}_1 = \mathbf{v} - \mathbf{v}_c,$$

and the other can be computed with a cross product

$$\mathbf{v}_2 = (\mathbf{v}_1 \times \mathbf{a}).$$

Because $\mathbf{a}$ is normalized, $\mathbf{v}_1$ and $\mathbf{v}_2$ have the same length, equal to the length of the vector between $\mathbf{v}$ and $\mathbf{v}_c$. To now compute the rotation by an angle θ about $\mathbf{v}_c$ in the plane of rotation, the rotation formulas earlier give us

$$\mathbf{v}' = \mathbf{v}_c + \mathbf{v}_1 \cos\theta + \mathbf{v}_2 \sin\theta.$$

To convert this to a rotation matrix, we apply this formula to the basis vectors (1, 0, 0), (0, 1, 0), and (0, 0, 1) to get the values of the rows of the matrix. The result of all this is encapsulated in the following function. As with the other rotation matrices, the inverse is equal to the transpose.

⟨*Transform Method Definitions*⟩ +≡
```
Transform Rotate(float angle, const Vector &axis) {
    Vector a = Normalize(axis);
    float s = sinf(Radians(angle));
    float c = cosf(Radians(angle));
    float m[4][4];

    m[0][0] = a.x * a.x + (1.f - a.x * a.x) * c;
    m[0][1] = a.x * a.y * (1.f - c) - a.z * s;
    m[0][2] = a.x * a.z * (1.f - c) + a.y * s;
    m[0][3] = 0;

    m[1][0] = a.x * a.y * (1.f - c) + a.z * s;
    m[1][1] = a.y * a.y + (1.f - a.y * a.y) * c;
    m[1][2] = a.y * a.z * (1.f - c) - a.x * s;
    m[1][3] = 0;

    m[2][0] = a.x * a.z * (1.f - c) - a.y * s;
    m[2][1] = a.y * a.z * (1.f - c) + a.x * s;
    m[2][2] = a.z * a.z + (1.f - a.z * a.z) * c;
    m[2][3] = 0;

    m[3][0] = 0;
    m[3][1] = 0;
    m[3][2] = 0;
    m[3][3] = 1;

    Matrix4x4 mat(m);
    return Transform(mat, Transpose(mat));
}
```

Matrix4x4 1021
Transform 76
Transpose() 1021
Vector 57
Vector::Normalize() 63

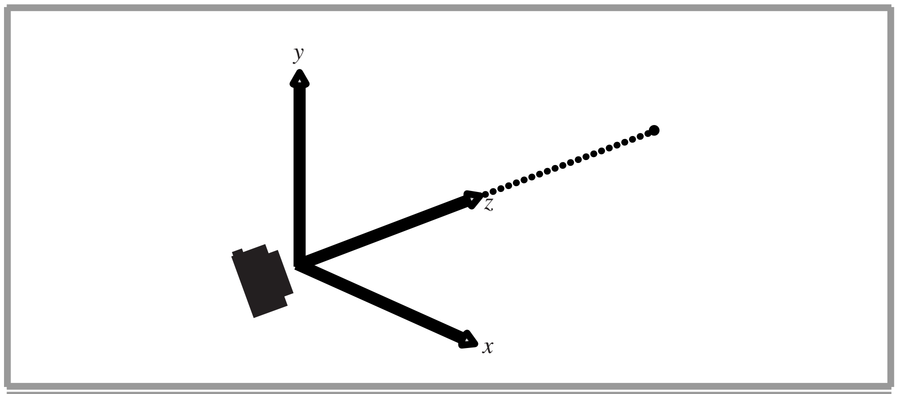


Figure 2.12: Given a camera position, the position being looked at from the camera, and an "up" direction, the look-at transformation describes a transformation from a viewing coordinate system where the camera is at the origin looking down the $+z$ axis, and the $+y$ axis is along the up direction.

2.7.7 THE LOOK-AT TRANSFORMATION

The *look-at transformation* is particularly useful for placing a camera in the scene. The caller specifies the desired position of the camera, a point the camera is looking at, and an "up" vector that orients the camera along the viewing direction implied by the first two parameters. All of these values are given in world space coordinates. The look-at construction then gives a transformation between camera space and world space (Figure 2.12).

In order to find the entries of the look-at transformation matrix, we use principles described earlier in this section: the columns of a transformation matrix give the effect of the transformation on the basis of a coordinate system.

⟨*Transform Method Definitions*⟩ +≡
```
Transform LookAt(const Point &pos, const Point &look, const Vector &up) {
    float m[4][4];
    ⟨Initialize fourth column of viewing matrix 84⟩
    ⟨Initialize first three columns of viewing matrix 85⟩
    Matrix4x4 camToWorld(m);
    return Transform(Inverse(camToWorld), camToWorld);
}
```

The easiest column is the fourth one, which gives the point that the camera space origin, $[0\ 0\ 0\ 1]^T$, maps to in world space. This is clearly just the camera position, supplied by the user.

⟨*Initialize fourth column of viewing matrix*⟩ ≡ 84
```
m[0][3] = pos.x;
m[1][3] = pos.y;
m[2][3] = pos.z;
m[3][3] = 1;
```

Inverse()	1021
Matrix4x4	1021
Point	63
Transform	76
Vector	57

The other three columns aren't much more difficult. First, `LookAt()` computes the normalized direction vector from the camera location to the look-at point; this gives the vector coordinates that the z axis should map to and, thus, the third column of the matrix. (Camera space is defined with the viewing direction down the $+z$ axis.) The first column, giving the world space direction that the $+x$ axis in camera space maps to, is found by taking the cross product of the user-supplied "up" vector with the recently computed viewing direction vector. Finally, the "up" vector is recomputed by taking the cross product of the viewing direction vector with the transformed x axis vector, thus ensuring that the y and z axes are perpendicular and we have an orthonormal viewing coordinate system.

⟨*Initialize first three columns of viewing matrix*⟩ ≡ 84

```
Vector dir = Normalize(look - pos);
Vector left = Normalize(Cross(Normalize(up), dir));
Vector newUp = Cross(dir, left);
m[0][0] = left.x;
m[1][0] = left.y;
m[2][0] = left.z;
m[3][0] = 0.;
m[0][1] = newUp.x;
m[1][1] = newUp.y;
m[2][1] = newUp.z;
m[3][1] = 0.;
m[0][2] = dir.x;
m[1][2] = dir.y;
m[2][2] = dir.z;
m[3][2] = 0.;
```

2.8 APPLYING TRANSFORMATIONS

We can now define routines that perform the appropriate matrix multiplications to transform points and vectors. We will overload the function application operator to describe these transformations; this lets us write code like:

```
Point P = ...;
Transform T = ...;
Point newP = T(P);
```

2.8.1 POINTS

The point transformation routine takes a point (x, y, z) and implicitly represents it as the homogeneous column vector $[x\ y\ z\ 1]^T$. It then transforms the point by premultiplying this vector with the transformation matrix. Finally, it divides by w to convert back to a nonhomogeneous point representation. For efficiency, this method skips the divide by the homogeneous weight, w, when $w = 1$, which is common for most of the transformations that will be used in `pbrt`—only the projective transformations defined in Chapter 6 will require this divide.

`Cross()` 62
`LookAt()` 84
`Vector` 57
`Vector::Normalize()` 63

⟨*Transform Inline Functions*⟩ ≡

```
inline Point Transform::operator()(const Point &pt) const {
    float x = pt.x, y = pt.y, z = pt.z;
    float xp = m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z + m.m[0][3];
    float yp = m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z + m.m[1][3];
    float zp = m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z + m.m[2][3];
    float wp = m.m[3][0]*x + m.m[3][1]*y + m.m[3][2]*z + m.m[3][3];
    if (wp == 1.) return Point(xp, yp, zp);
    else          return Point(xp, yp, zp)/wp;
}
```

We also provide transformation methods that let the caller pass a pointer to an object for the result. This saves the expense of returning structures by value on the stack. Note that we copy the original (x, y, z) coordinates to local variables in case the result pointer points to the same location as `pt`. This way, these routines can be used even if a point is being transformed in place.

⟨*Transform Inline Functions*⟩ +≡

```
inline void Transform::operator()(const Point &pt,
                                  Point *ptrans) const {
    float x = pt.x, y = pt.y, z = pt.z;
    ptrans->x = m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z + m.m[0][3];
    ptrans->y = m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z + m.m[1][3];
    ptrans->z = m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z + m.m[2][3];
    float w   = m.m[3][0]*x + m.m[3][1]*y + m.m[3][2]*z + m.m[3][3];
    if (w != 1.) *ptrans /= w;
}
```

2.8.2 VECTORS

The transformations of vectors can be computed in a similar fashion. However, the multiplication of the matrix and the row vector is simplified since the implicit homogeneous w coordinate is zero.

⟨*Transform Inline Functions*⟩ +≡

```
inline Vector Transform::operator()(const Vector &v) const {
  float x = v.x, y = v.y, z = v.z;
  return Vector(m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z,
                m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z,
                m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z);
}
```

There is also a method allowing the caller to pass a pointer to the result vector. The code to do this has a similar design to the `Point` transformation code and is not shown here. This code will also be omitted for subsequent transformation methods.

`Point` 63
`Transform` 76
`Vector` 57

2.8.3 NORMALS

Normals do not transform in the same way that vectors do, as shown in Figure 2.13. Although tangent vectors transform in the straightforward way, normals require special

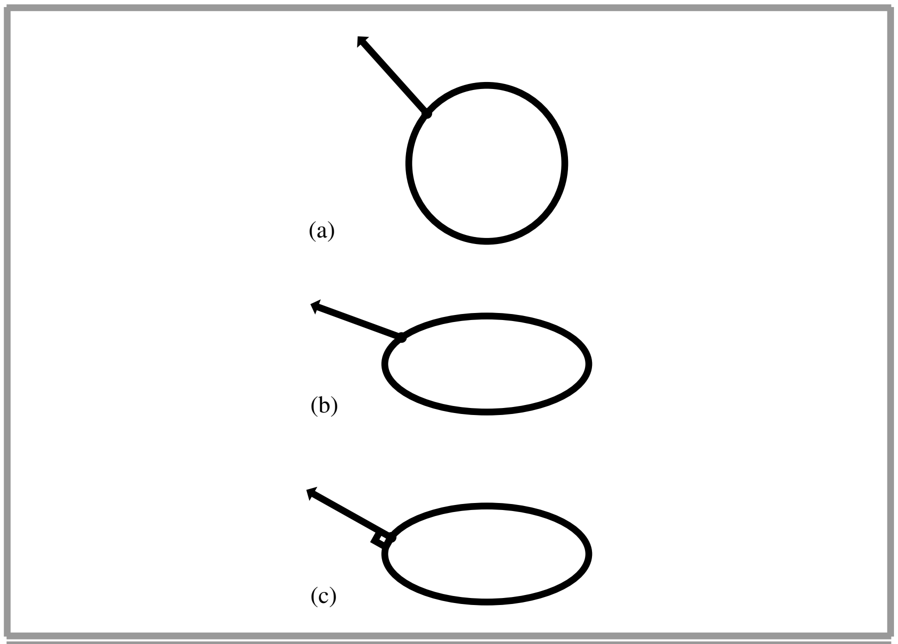


Figure 2.13: Transforming Surface Normals. (a) Original circle. (b) When scaling the circle to be half as tall in the y direction, simply treating the normal as a direction and scaling it in the same manner will lead to incorrect results. (c) A properly transformed normal.

treatment. Because the normal vector $\mathbf{n}$ and any tangent vector $\mathbf{t}$ on the surface are orthogonal by construction, we know that

$$\mathbf{n} \cdot \mathbf{t} = \mathbf{n}^T \mathbf{t} = 0.$$

When we transform a point on the surface by some matrix $\mathbf{M}$, the new tangent vector $\mathbf{t}'$ at the transformed point is $\mathbf{Mt}$. The transformed normal $\mathbf{n}'$ should be equal to $\mathbf{Sn}$ for some 4×4 matrix $\mathbf{S}$. To maintain the orthogonality requirement, we must have

$$\begin{aligned} 0 &= (\mathbf{n}')^T \mathbf{t}' \\ &= (\mathbf{Sn})^T \mathbf{Mt} \\ &= (\mathbf{n})^T \mathbf{S}^T \mathbf{Mt}. \end{aligned}$$

This condition holds if $\mathbf{S}^T \mathbf{M} = \mathbf{I}$, the identity matrix. Therefore, $\mathbf{S}^T = \mathbf{M}^{-1}$, and so $\mathbf{S} = \mathbf{M}^{-1^T}$, and we see that normals must be transformed by the inverse transpose of the transformation matrix. This detail is one of the main reasons why `Transforms` maintain their inverses.

Transform 76
Vector 57

Note that this method does not explicitly compute the transpose of the inverse when transforming normals. It just indexes into the inverse matrix in a different order (compare to the code for transforming `Vectors`).

⟨*Transform Inline Functions*⟩ +≡
```
inline Normal Transform::operator()(const Normal &n) const {
    float x = n.x, y = n.y, z = n.z;
    return Normal(mInv.m[0][0]*x + mInv.m[1][0]*y + mInv.m[2][0]*z,
                  mInv.m[0][1]*x + mInv.m[1][1]*y + mInv.m[2][1]*z,
                  mInv.m[0][2]*x + mInv.m[1][2]*y + mInv.m[2][2]*z);
}
```

2.8.4 RAYS

Transforming rays is straightforward: the method to do this transforms the constituent origin and direction and copies the other data members. pbrt also provides similar methods for transforming RayDifferentials.

⟨*Transform Inline Functions*⟩ +≡
```
inline Ray Transform::operator()(const Ray &r) const {
    Ray ret = r;
    (*this)(ret.o, &ret.o);
    (*this)(ret.d, &ret.d);
    return ret;
}
```

2.8.5 BOUNDING BOXES

The easiest way to transform an axis-aligned bounding box is to transform all eight of its corner vertices and then compute a new bounding box that encompasses those points. We will present code for this method below; one of the exercises for this chapter is to implement a technique to do this computation more efficiently.

⟨*Transform Method Definitions*⟩ +≡
```
BBox Transform::operator()(const BBox &b) const {
    const Transform &M = *this;
    BBox ret(        M(Point(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

BBox 70
Normal 65
Point 63
Ray 66
RayDifferential 69
Transform 76

2.8.6 COMPOSITION OF TRANSFORMATIONS

Having defined how the matrices representing individual types of transformations are constructed, we can now consider an aggregate transformation resulting from a series of individual transformations. Finally, we will see the real value of representing transformations with matrices.

Consider a series of transformations **ABC**. We'd like to compute a new transformation **T** such that applying **T** gives the same result as applying each of **A**, **B**, and **C** in reverse order; that is, $\mathbf{A}(\mathbf{B}(\mathbf{C}(\mathrm{p}))) = \mathbf{T}(\mathrm{p})$. Such a transformation **T** can be computed by multiplying the matrices of the transformations **A**, **B**, and **C** together. In pbrt, we can write:

```
Transform T = A * B * C;
```

Then we can apply T to Points p as usual, Point pp = T(p), instead of applying each transformation in turn: Point pp = A(B(C(p))).

We use the C++ * operator to compute the new transformation that results from post-multiplying a transformation with another transformation t2. In matrix multiplication, the (i, j)th element of the resulting matrix ret is the inner product of the ith row of the first matrix with the jth column of the second.

The inverse of the resulting transformation is equal to the product of t2.mInv * mInv. This is a result of the matrix identity

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}.$$

⟨*Transform Method Definitions*⟩ +≡

```
Transform Transform::operator*(const Transform &t2) const {
    Matrix4x4 m1 = Matrix4x4::Mul(m, t2.m);
    Matrix4x4 m2 = Matrix4x4::Mul(t2.mInv, mInv);
    return Transform(m1, m2);
}
```

2.8.7 TRANSFORMATIONS AND COORDINATE SYSTEM HANDEDNESS

Certain types of transformations change a left-handed coordinate system into a right-handed one, or vice versa. Some routines will need to know if the handedness of the source coordinate system is different from that of the destination. In particular, routines that want to ensure that a surface normal always points "outside" of a surface might need to flip the normal's direction after transformation if the handedness changes.

Fortunately, it is easy to tell if handedness is changed by a transformation: it happens only when the determinant of the transformation's upper-left 3×3 submatrix is negative.

⟨*Transform Method Definitions*⟩ +≡

```
bool Transform::SwapsHandedness() const {
    float det = ((m.m[0][0] *
                  (m.m[1][1] * m.m[2][2] -
                   m.m[1][2] * m.m[2][1])) -
                 (m.m[0][1] *
                  (m.m[1][0] * m.m[2][2] -
                   m.m[1][2] * m.m[2][0])) +
                 (m.m[0][2] *
                  (m.m[1][0] * m.m[2][1] -
                   m.m[1][1] * m.m[2][0])));
    return det < 0.f;
}
```

Matrix4x4 1021
Matrix4x4::Mul() 1021
Point 63
Transform 76

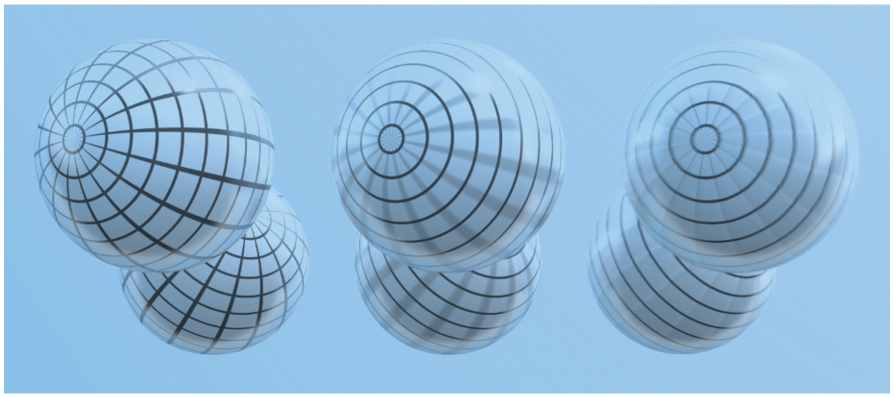


Figure 2.14: Spinning Spheres. Three spheres, spinning at different rates using the transformation animation code implemented in this section. Note that the reflections of the spheres are blurry as well as the spheres themselves.

★ 2.9 ANIMATING TRANSFORMATIONS

`pbrt` supports *keyframe matrix animation* for cameras and geometric primitives in the scene. Rather than just supplying a single transformation to place the corresponding object in the scene, the user may supply a number of *keyframe* transformations, each one associated with a particular point in time. This makes it possible for the camera to move, and for objects in the scene to be moving during the time the simulated camera's shutter is open. Figure 2.14 shows three spheres animated using keyframe matrix animation in `pbrt`.

In general, the problem of interpolating between keyframe matrices is under-defined. As one example, if we have a rotation about the x axis of 179 degrees followed by another of 181 degrees, does this represent a small rotation of 2 degrees, or a large rotation of -358 degrees? For another example, consider two matrices where one is the identity and the other is a 180-degree rotation about the z axis. There are an infinite number of ways to go from one orientation to the other.

Keyframe matrix interpolation is an important problem in computer animation, where a number of different approaches have been developed. Fortunately, the problem of matrix interpolation in a renderer is generally less challenging than it is for animation systems for two reasons.

First, in a renderer like `pbrt`, we generally have a keyframe matrix at the camera shutter open time and another at the shutter close time; we only need to interpolate between the two of them across the time of a single image. In animation systems, the matrices are generally available at a lower time frequency, so that there are many frames between

★ This section covers advanced topics and may be skipped on a first reading.

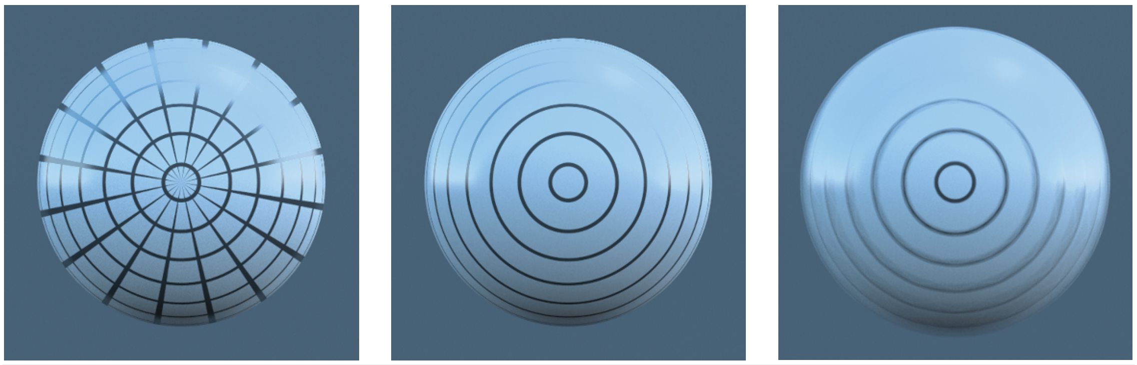


Figure 2.15: (a) Sphere with a grid of lines as a texture, not rotating. (b) Sphere rotating 90 degrees during the course of the frame, using the technique for interpolating transformations implemented in this section. (c) Sphere rotating 90 degrees using direct interpolation of matrix components to interpolate transformations. In this case, the animated sphere incorrectly grows larger. Furthermore, the lines toward the outside of the sphere, which should remain sharp, incorrectly become blurry.

pairs of keyframe matrices; as such, there's more opportunity to notice shortcomings in the interpolation.

Second, in a physically based renderer, the longer the period of time over which we need to interpolate the pair of matrices, the longer the virtual camera shutter is open and the more motion blur there will be in the final image; in general, the increased amount of motion blur will hide sins of the interpolation.

Here we will implement a method that interpolates between the transformations defined by the keyframe matrices. The most straightforward approach—directly interpolating the individual components of the two matrices—is not a good one, as it will generally lead to unexpected and undesirable results. For example, if the transformations apply different rotations, then even if we have a rigid-body motion, the intermediate matrices may scale the object, which is clearly undesirable. (If they have a full 180-degree rotation between them, the object may be scaled down to nothing at the middle of the interpolation!)

Figure 2.15 shows a sphere that rotates 90 degrees over the course of the frame; direct interpolation of matrix elements (on the right) gives a less accurate result than the approach implemented in this section (in the middle).

The approach used for transformation interpolation in pbrt is based on *matrix decomposition*—given an arbitrary transformation matrix $\mathbf{M}$, we decompose it into a concatenation of scale ($\mathbf{S}$), rotation ($\mathbf{R}$), and translation ($\mathbf{T}$) transformations,

$$\mathbf{M} = \mathbf{S}\mathbf{R}\mathbf{T},$$

where each of those components is independently interpolated and then the composite interpolated matrix is found by multiplying the three interpolated matrices together.

Interpolation of translation and scale can be performed easily and accurately with linear interpolation of the components of their matrices; interpolating rotations is more difficult. Before describing the matrix decomposition implementation in pbrt, we will first introduce *quaternions*, an elegant representation of rotations that leads to elegant methods for interpolating rotations.

2.9.1 QUATERNIONS

Quaternions were originally invented by Sir William Rowan Hamilton in 1843 as a generalization of imaginary numbers. He determined that just as in two dimensions (x, y), where imaginary numbers could be defined as $x + y\mathrm{i}$, with $\mathrm{i}^2 = -1$, a generalization could be made to four dimensions, giving quaternions.

A quaternion is a four-tuple,

$$\mathbf{q} = (x, y, z, w) = w + x\mathrm{i} + y\mathrm{j} + z\mathrm{k}, \tag{2.4}$$

where i, j, and k are defined so that $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$.[2] Other important relationships between the components are that $\mathrm{ij} = \mathrm{k}$, and $\mathrm{ji} = -\mathrm{k}$. (Note that quaternion multiplication is not commutative.)

A quaternion can be represented as a quadruple $\mathbf{q} = (\mathbf{q}_x, \mathbf{q}_y, \mathbf{q}_z, \mathbf{q}_w)$, or as $\mathbf{q} = (\mathbf{q}_{xyz}, \mathbf{q}_w)$, where $\mathbf{q}_{xyz}$ is a 3-vector. We will use both representations interchangeably in this section.

Quaternion multiplication is defined by expanding out the product of two quaternions:

$$\mathbf{q}\mathbf{q}' = (\mathbf{q}_w + \mathbf{q}_x\mathrm{i} + \mathbf{q}_y\mathrm{j} + \mathbf{q}_z\mathrm{k})(\mathbf{q}'_w + \mathbf{q}'_x\mathrm{i} + \mathbf{q}'_y\mathrm{j} + \mathbf{q}'_z\mathrm{k}).$$

Collecting terms and using identities among the components like those listed above (e.g., $\mathrm{i}^2 = -1$), the result can be expressed concisely using vector cross and dot products:

$$\begin{aligned}(\mathbf{q}\mathbf{q}')_{xyz} &= \mathbf{q}_{xyz} \times \mathbf{q}'_{xyz} + \mathbf{q}_w\mathbf{q}'_{xyz} + \mathbf{q}'_w\mathbf{q}_{xyz} \\ (\mathbf{q}\mathbf{q}')_w &= \mathbf{q}_w\mathbf{q}'_w - (\mathbf{q}_{xyz} \cdot \mathbf{q}'_{xyz}).\end{aligned} \tag{2.5}$$

A unit quaternion (where $x^2 + y^2 + z^2 + w^2 = 1$) represents a rotation $(\mathbf{q}_{xyz} \sin\theta, \cos\theta)$ of angle 2θ around the unit axis $\hat{\mathbf{q}}_{xyz}$. Given a point p in homogeneous coordinates, the result of the rotation of the point by the quaternion is given by the quaternion product

$$\mathrm{p}' = \mathbf{q}\mathrm{p}\mathbf{q}^{-1}.$$

The implementation of the `Quaternion` class in `pbrt` is in the files `core/quaternion.h` and `core/quaternion.cpp`. The default constructor initializes a unit quaternion.

⟨*Quaternion Public Methods*⟩ ≡
```
Quaternion() { v = Vector(0., 0., 0.); w = 1.f; }
```

We use a `Vector` to represent the xyz components of the quaternion; this lets us take advantage of the already implemented methods of `Vector` in the implementation of some of the methods below.

`Quaternion` 92
`Quaternion::v` 93
`Quaternion::w` 93
`Vector` 57

2 Hamilton found the discovery of this relationship among the components compelling enough that he used a knife to carve the formula on the bridge he was crossing when it came to him.

⟨*Quaternion Public Data*⟩ ≡
```
Vector v;
float w;
```

Addition and subtraction of quaternions is performed component-wise. This follows directly from the definition in Equation (2.4). For example,

$$\begin{aligned}\mathbf{q} + \mathbf{q}' &= w + x\mathrm{i} + y\mathrm{j} + z\mathrm{k} + w' + x'\mathrm{i} + y'\mathrm{j} + z'\mathrm{k} \\ &= (w + w') + (x + x')\mathrm{i} + (y + y')\mathrm{j} + (z + z')\mathrm{k}.\end{aligned}$$

Other arithmetic methods (subtraction, multiplication, and division by a scalar) are defined and implemented similarly and won't be included here.

⟨*Quaternion Public Methods*⟩ +≡
```
Quaternion &operator+=(const Quaternion &q) {
    v += q.v;
    w += q.w;
    return *this;
}
```

The inner product of two quaternions is implemented by its `Dot()` method, and a quaternion can be normalized by dividing by its length.

⟨*Quaternion Inline Functions*⟩ ≡
```
inline float Dot(const Quaternion &q1, const Quaternion &q2) {
    return Dot(q1.v, q2.v) + q1.w * q2.w;
}
```

⟨*Quaternion Inline Functions*⟩ +≡
```
inline Quaternion Normalize(const Quaternion &q) {
    return q / sqrtf(Dot(q, q));
}
```

It's useful to be able to compute the transformation matrix that represents the same rotation as a quaternion. In particular, after interpolating rotations with quaternions in the `AnimatedTransform` class, we'll need to convert the interpolated rotation back to a transformation matrix to compute the final composite interpolated transformation.

To derive the rotation matrix for a quaternion, recall that the transformation of a point by a quaternion is given by $\mathrm{p}' = \mathbf{q}\mathrm{p}\mathbf{q}^{-1}$. We want a matrix $\mathbf{M}$ that performs the same transformation, so that $\mathrm{p}' = \mathbf{M}\mathrm{p}$. If we expand out the quaternion multiplication $\mathbf{q}\mathrm{p}\mathbf{q}^{-1}$ using Equation (2.5), simplify with the quaternion basis identities, collect terms, and represent the result in a matrix, we can determine that the following 3×3 matrix represents the same transformation:

$$\mathbf{M} = \begin{pmatrix} 1 - 2(\mathbf{q}_y^2 + \mathbf{q}_z^2) & 2(\mathbf{q}_x\mathbf{q}_y + \mathbf{q}_z\mathbf{q}_w) & 2\mathbf{q}_x\mathbf{q}_z - \mathbf{q}_y\mathbf{q}_w) \\ 2(\mathbf{q}_x\mathbf{q}_y - \mathbf{q}_z\mathbf{q}_w) & 1 - 2(\mathbf{q}_x^2 + \mathbf{q}_z^2) & 2(\mathbf{q}_y\mathbf{q}_z + \mathbf{q}_x\mathbf{q}_w) \\ 2(\mathbf{q}_x\mathbf{q}_z + \mathbf{q}_y\mathbf{q}_w) & 2(\mathbf{q}_y\mathbf{q}_z - \mathbf{q}_x\mathbf{q}_w) & 1 - 2(\mathbf{q}_x^2 + \mathbf{q}_y^2) \end{pmatrix}. \qquad [2.6]$$

This computation is implemented in the method `Quaternion::ToTransform()`. We won't include its implementation here since it's a direct implementation of Equation (2.6).

`AnimatedTransform` 96
`Dot()` 60
`Quaternion` 92
`Quaternion::Dot()` 93
`Quaternion::v` 93
`Quaternion::w` 93
`Vector` 57

⟨*Quaternion Public Methods*⟩ +≡
```
    Transform ToTransform() const;
```

Note that we could alternatively use the fact that a unit quaternion represents a rotation $(\mathbf{q}_{xyz} \sin\theta, \cos\theta)$ of angle 2θ around the unit axis $\hat{\mathbf{q}}_{xyz}$ to compute a rotation matrix. First we would compute the angle of rotation θ as $\theta = 2 \arccos \mathbf{q}_w$ and then we'd use the previously defined `Rotate()` function, passing it the axis $\hat{\mathbf{q}}_{xyz}$ and the rotation angle θ. However, this alternative would be substantially less efficient, requiring multiple calls to trigonometric functions, while the approach here only uses floating-point addition, subtraction, and multiplication.

It is also useful to be able to create a quaternion from a rotation matrix. For this purpose, `Quaternion` provides a constructor that takes a `Transform`. The appropriate quaternion can be computed by making use of relationships between elements of the rotation matrix in Equation (2.6) and quaternion components. For example, if we subtract the transpose of this matrix from itself, then the (0, 1) component of the resulting matrix has the value $-4\mathbf{q}_w\mathbf{q}_z$. Thus, given a particular instance of a rotation matrix with known values, it's possible to use a number of relationships like this between the matrix values and the quaternion components to generate a system of equations that can be solved for the quaternion components.

We won't include the details of the derivation or the actual implementation here in the text; for more information about how to derive this technique, including handling numerical robustness, see Shoemake (1991).

⟨*Quaternion Public Methods*⟩ +≡
```
    Quaternion(const Transform &t);
```

2.9.2 QUATERNION INTERPOLATION

The last quaternion function we will define, `Slerp()`, interpolates between two quaternions using spherical linear interpolation. Spherical linear interpolation gives constant speed motion along great circle arcs on the surface of a sphere and consequently has two desirable properties for interpolating rotations:

- The interpolated rotation path exhibits *torque minimization*: the path to get between two rotations is the shortest possible path in rotation space.
- The interpolation has *constant angular velocity*: the relationship between change in the animation parameter t and the change in the resulting rotation is constant over the course of interpolation (in other words, the speed of interpolation is constant across the interpolation range).

See the "Further Reading" section at the end of the chapter for references that discuss more rigorously what characteristics a good interpolated rotation should have.

Spherical linear interpolation for quaternions was originally presented by Shoemake (1985) as follows, where two quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$ are given and $t \in [0, 1]$ is the parameter value to interpolate between them:

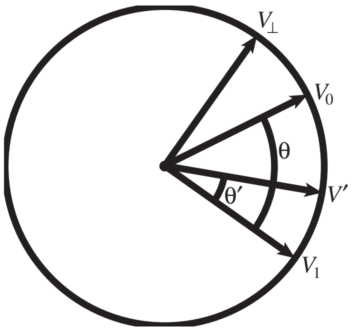


Figure 2.16: To understand quaternion spherical linear interpolation, consider in 2D two vectors on the unit sphere, $\mathbf{v}_0$ and $\mathbf{v}_1$, with angle θ between them. We'd like to be able to compute the interpolated vector at some angle θ' between the two of them. To do so, we can find a vector that's orthogonal to $\mathbf{v}_1$, $\mathbf{v}_\perp$ and then apply the trigonometric identity $\mathbf{v} = \mathbf{v}_1 \cos\theta' + \mathbf{v}_\perp \sin\theta'$.

$$slerp(\mathbf{q}_1, \mathbf{q}_2, t) = \frac{\mathbf{q}_1 \sin((1-t)\theta) + \mathbf{q}_2 \sin(t\theta)}{\sin\theta}.$$

An intuitive way to understand `Slerp()` was presented by Blow (2004). As context, given the quaternions to interpolate between, $\mathbf{q}_1$ and $\mathbf{q}_2$, denote by θ the angle between them. Then, given a parameter value $t \in [0, 1]$, we'd like to find the intermediate quaternion $\mathbf{q}'$ that makes angle $\theta' = \theta t$ between it and $\mathbf{q}_1$, along the path from $\mathbf{q}_1$ to $\mathbf{q}_2$.

An easy way to compute $\mathbf{q}'$ is to first compute an orthogonal coordinate system in the space of quaternions where one axis is $\mathbf{q}_1$ and the other is a quaternion orthogonal to $\mathbf{q}_1$ such that the two axes form a basis that spans $\mathbf{q}_1$ and $\mathbf{q}_2$. Given such a coordinate system, we can compute rotations with respect to $\mathbf{q}_1$. (See Figure 2.16, which illustrates the concept in the 2D setting.) An orthogonal vector $\mathbf{q}_\perp$ can be found by projecting $\mathbf{q}_1$ onto $\mathbf{q}_2$ and then subtracting the orthogonal projection from $\mathbf{q}_2$; the remainder is guaranteed to be orthogonal to $\mathbf{q}_1$:

$$\mathbf{q}_\perp = \mathbf{q}_2 - (\mathbf{q}_1 \cdot \mathbf{q}_2)\mathbf{q}_1. \quad [2.7]$$

Given the coordinate system, quaternions along the animation path are given by

$$\mathbf{q}' = \mathbf{q}_1 \cos(\theta t) + \mathbf{q}_\perp \sin(\theta t). \quad [2.8]$$

The implementation of the `Slerp()` function checks to see if the two quaternions are nearly parallel, in which case it uses regular linear interpolation of quaternion components in order to avoid numerical instability. Otherwise, it computes an orthogonal quaternion `qperp` using Equation (2.7) and then computes the interpolated quaternion with Equation (2.8).

`Slerp()` 96

⟨*Quaternion Method Definitions*⟩ ≡
```
Quaternion Slerp(float t, const Quaternion &q1,
                 const Quaternion &q2) {
    float cosTheta = Dot(q1, q2);
    if (cosTheta > .9995f)
        return Normalize((1.f - t) * q1 + t * q2);
    else {
        float theta = acosf(Clamp(cosTheta, -1.f, 1.f));
        float thetap = theta * t;
        Quaternion qperp = Normalize(q2 - q1 * cosTheta);
        return q1 * cosf(thetap) + qperp * sinf(thetap);
    }
}
```

2.9.3 `AnimatedTransform` IMPLEMENTATION

Given the foundations of the quaternion infrastructure, we can now implement the `AnimatedTransform` class, which implements keyframe transformation interpolation in `pbrt`. Its constructor takes two transformations and the time values they are associated with.

Following an approach introduced by Shoemake and Duff (1992), `AnimatedTransform` decomposes the given composite transformation matrices into scaling, rotation, and translation components. The decomposition is performed by the `AnimatedTransform::Decompose()` method.

⟨*AnimatedTransform Public Methods*⟩ ≡
```
AnimatedTransform(const Transform *transform1, float time1,
                  const Transform *transform2, float time2)
    : startTime(time1), endTime(time2),
      startTransform(transform1), endTransform(transform2),
      actuallyAnimated(*startTransform != *endTransform) {
    Decompose(startTransform->m, &T[0], &R[0], &S[0]);
    Decompose(endTransform->m, &T[1], &R[1], &S[1]);
}
```

⟨*AnimatedTransform Private Data*⟩ ≡
```
const float startTime, endTime;
const Transform *startTransform, *endTransform;
const bool actuallyAnimated;
Vector T[2];
Quaternion R[2];
Matrix4x4 S[2];
```

Given the composite matrix for a transformation, information has been lost about any individual transformations that were composed to compute it. For example, given the matrix for the product of a translation and then a scale, an equal matrix could also be computed by first scaling and then translating (by different amounts). Thus, we need to choose a canonical sequence of transformations for the decomposition. For our needs

AnimatedTransform 96
AnimatedTransform::Decompose() 97
Clamp() 1000
Matrix4x4 1021
Quaternion 92
Quaternion::Dot() 93
Quaternion::Normalize() 93
Transform 76
Vector 57

here, this decision isn't significant. (It would be more important in an animation system that was decomposing composite transformations in order to make them editable by changing individual components, for example.)

Further, we will handle only affine transformations here, which is what is needed for animating cameras and geometric primitives in a rendering system; perspective transformations aren't generally relevant to animation of objects like these.

The transformation decomposition we will use is the following:

$$\mathbf{M} = \mathbf{TRS},$$

where **M** is the given transformation, **T** is a translation, **R** is a rotation, and **S** is a scale. **S** is actually a generalized scale (Shoemake and Duff call it *stretch*) that represents a scale in *some* coordinate system, just not necessarily the current one. In any case, it can still be properly interpolated with linear interpolation of its components. The `Decompose()` method computes the decomposition given a `Matrix4x4`.

```
⟨AnimatedTransform Method Definitions⟩ ≡
  void AnimatedTransform::Decompose(const Matrix4x4 &m, Vector *T,
                                    Quaternion *Rquat, Matrix4x4 *S) {
      ⟨Extract translation T from transformation matrix 97⟩
      ⟨Compute new transformation matrix M without translation 97⟩
      ⟨Extract rotation R from transformation matrix 98⟩
      ⟨Compute scale S using rotation and original matrix 98⟩
  }
```

Extracting the translation **T** is easy; it can be found directly from the appropriate elements of the 4×4 transformation matrix.

```
⟨Extract translation T from transformation matrix⟩ ≡                97
  T->x = m.m[0][3];
  T->y = m.m[1][3];
  T->z = m.m[2][3];
```

Since we are assuming an affine transformation (no projective components), after we remove the translation, what is left is the upper 3×3 matrix that represents scaling and rotation together. This matrix is copied into a new matrix `M` for further processing.

```
⟨Compute new transformation matrix M without translation⟩ ≡         97
  Matrix4x4 M = m;
  for (int i = 0; i < 3; ++i)
      M.m[i][3] = M.m[3][i] = 0.f;
  M.m[3][3] = 1.f;
```

Matrix4x4 1021
Matrix4x4::m 1021
Quaternion 92
Vector 57

Next we'd like to extract the pure rotation component of `M`. We'll use a technique called *polar decomposition* to do this. It can be shown that the polar decomposition of a matrix **M** into rotation **R** and scale **S** can be computed by successively averaging **M** to its inverse transpose

$$\mathbf{M}_{i+1} = \frac{1}{2}\left(\mathbf{M}_i + (\mathbf{M}_i^T)^{-1}\right) \qquad [2.9]$$

until convergence. (It's easy to see that if **M** is a pure rotation, then averaging it with its inverse transpose will leave it unchanged, since its inverse is equal to its transpose. The "Further Reading" section has more references that discuss why this series converges to the rotation component of the original transformation.) Shoemake and Duff (1992) proved that the resulting matrix is the closest orthogonal matrix to **M**—a desirable property.

To compute this series, we iteratively apply Equation (2.9) until either the difference between successive terms is small or a fixed number of iterations have been performed. In practice, this series generally converges quickly.

⟨*Extract rotation* `R` *from transformation matrix*⟩ ≡ **97**

```
float norm;
int count = 0;
Matrix4x4 R = M;
do {
    ⟨Compute next matrix Rnext in series 98⟩
    ⟨Compute norm of difference between R and Rnext 98⟩
    R = Rnext;
} while (++count < 100 && norm > .0001f);
*Rquat = Quaternion(R);
```

⟨*Compute next matrix* `Rnext` *in series*⟩ ≡ **98**

```
Matrix4x4 Rnext;
Matrix4x4 Rit = Inverse(Transpose(R));
for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 4; ++j)
        Rnext.m[i][j] = 0.5f * (R.m[i][j] + Rit.m[i][j]);
```

⟨*Compute norm of difference between* `R` *and* `Rnext`⟩ ≡ **98**

```
norm = 0.f;
for (int i = 0; i < 3; ++i) {
    float n = fabsf(R.m[i][0] - Rnext.m[i][0]) +
              fabsf(R.m[i][1] - Rnext.m[i][1]) +
              fabsf(R.m[i][2] - Rnext.m[i][2]);
    norm = max(norm, n);
}
```

Once we've extracted the rotation from **M**, the scale is all that's left. We would like to find the matrix **S** that satisfies $\mathbf{M} = \mathbf{RS}$. Now that we know both **R** and **M**, we just solve for $\mathbf{S} = \mathbf{R}^{-1}\mathbf{M}$.

⟨*Compute scale* `S` *using rotation and original matrix*⟩ ≡ **97**

```
*S = Matrix4x4::Mul(Inverse(R), M);
```

The `Interpolate()` method computes the interpolated transformation matrix at a given time. The matrix is found by interpolating the previously extracted translation, rotation, and scale and then multiplying them together to get a composite matrix that represents the effect of the three transformations together.

`Inverse()`	1021
`Matrix4x4`	1021
`Matrix4x4::m`	1021
`Matrix4x4::Mul()`	1021
`Quaternion`	92
`Transpose()`	1021

⟨*AnimatedTransform Method Definitions*⟩ +≡
```
void AnimatedTransform::Interpolate(float time, Transform *t) const {
    ⟨Handle boundary conditions for matrix interpolation 99⟩
    float dt = (time - startTime) / (endTime - startTime);
    ⟨Interpolate translation at dt 99⟩
    ⟨Interpolate rotation at dt 99⟩
    ⟨Interpolate scale at dt 99⟩
    ⟨Compute interpolated matrix as product of interpolated components 100⟩
}
```

If the given time value is outside the time range of the two transformations stored in the `AnimatedTransform`, then the transformation at the start time or end time is returned, as appropriate. The `AnimatedTransform` constructor also checks whether the two `Transforms` stored are the same; if so, then no interpolation is necessary either. Most of the classes in `pbrt` that support animation always store an `AnimatedTransform` for their transformation, rather than storing either a `Transform` or `AnimatedTransform` as appropriate. This simplifies their implementations, though it does make it worthwhile to check for this case here and not unnecessarily do the work to interpolate between two equal transformations.

⟨*Handle boundary conditions for matrix interpolation*⟩ ≡ **99**
```
if (!actuallyAnimated || time <= startTime) {
    *t = *startTransform;
    return;
}
if (time >= endTime) {
    *t = *endTransform;
    return;
}
```

The `dt` variable stores the offset in the range from `startTime` to `endTime`; it is zero at `startTime` and one at `endTime`. Given `dt`, interpolation of the translation is trivial.

⟨*Interpolate translation at* `dt`⟩ ≡ **99**
```
Vector trans = (1.f - dt) * T[0] + dt * T[1];
```

The rotation is interpolated between the start and end rotations using the `Slerp()` routine (Section 2.9.2).

⟨*Interpolate rotation at* `dt`⟩ ≡ **99**
```
Quaternion rotate = Slerp(dt, R[0], R[1]);
```

Finally, the interpolated scale matrix is computed by just interpolating the individual elements of the start and end scale matrices. Because the `Matrix4x4` constructor sets the matrix to the identity matrix, we don't need to initialize any of the other elements.

⟨*Interpolate scale at* `dt`⟩ ≡ **99**
```
Matrix4x4 scale;
for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 3; ++j)
        scale.m[i][j] = Lerp(dt, S[0].m[i][j], S[1].m[i][j]);
```

AnimatedTransform 96
AnimatedTransform::actuallyAnimated 96
AnimatedTransform::endTime 96
AnimatedTransform::endTransform 96
AnimatedTransform::R 96
AnimatedTransform::S 96
AnimatedTransform::startTime 96
AnimatedTransform::startTransform 96
AnimatedTransform::T 96
Matrix4x4 1021
Quaternion 92
Slerp() 96
Transform 76
Vector 57

Given the three interpolated parts, the product of their three transformation matrices gives us the final result.

⟨*Compute interpolated matrix as product of interpolated components*⟩ ≡ 99

```
*t = Translate(trans) * rotate.ToTransform() * Transform(scale);
```

AnimatedTransform also provides a number of methods that apply interpolated transformations directly, using the provided time for Points and Vectors and Ray::time for Rays. These methods are more efficient than calling AnimatedTransform::Interpolate() and then using the returned matrix when there is no actual animation, since a copy of the transformation matrix doesn't need to be made.

⟨*AnimatedTransform Public Methods*⟩ +≡

```
void operator()(const Ray &r, Ray *tr) const;
void operator()(const RayDifferential &r, RayDifferential *tr) const;
Point operator()(float time, const Point &p) const;
Vector operator()(float time, const Vector &v) const;
Ray operator()(const Ray &r) const;
```

Given a geometric primitive with an animated transformation, it's necessary to be able to compute a bounding box that encompasses the primitive's motion over the animation time period. The AnimatedTransform::MotionBounds() method takes a bounding box and computes a bounding box of its animated motion over the AnimatedTransform's time range. Its useInverse parameter indicates whether the forward or inverse transform should be used to transform the bounding box.

This method, which is based on computing the animated transformation at a large number of time samples, may compute an insufficiently conservative bounding box as implemented here (though in practice it works well). An exercise at the end of the chapter discusses its limitations in more detail and suggests a direction for a more robust approach.

⟨*AnimatedTransform Method Definitions*⟩ +≡

```
BBox AnimatedTransform::MotionBounds(const BBox &b,
                                     bool useInverse) const {
    if (!actuallyAnimated) return Inverse(*startTransform)(b);
    BBox ret;
    const int nSteps = 128;
    for (int i = 0; i < nSteps; ++i) {
        Transform t;
        float time = Lerp(float(i)/float(nSteps-1), startTime, endTime);
        Interpolate(time, &t);
        if (useInverse) t = Inverse(t);
        ret = Union(ret, t(b));
    }
    return ret;
}
```

2.10 DIFFERENTIAL GEOMETRY

We will wrap up this chapter by developing a self-contained representation for the geometry of a particular point on a surface (typically the point of a ray intersection). This abstraction needs to hide the particular type of geometric shape the point lies on, supplying enough information about the surface point to allow the shading and geometric operations in the rest of `pbrt` to be implemented generically, without the need to distinguish between different shape types such as spheres and triangles. The `DifferentialGeometry` class implements just such an abstraction; its implementation is in the files `core/diffgeom.h` and `core/diffgeom.cpp`.

The information needed by the rest of the system includes:

- The position of the 3D point p
- The surface normal **n** at the point
- (u, v) coordinates from the parameterization of the surface
- The parametric partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$
- The partial derivatives of the change in surface normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$
- A pointer to the `Shape` that the differential geometry lies on (the `Shape` class will be introduced in the next chapter)

See Figure 2.17 for a depiction of these values.

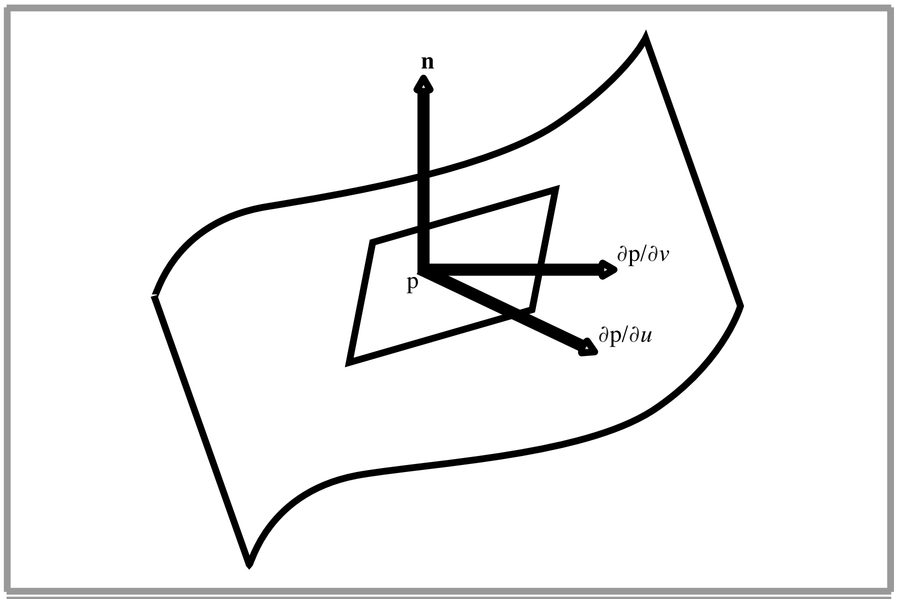

n
p
$\partial \mathrm{p}/\partial v$
$\partial \mathrm{p}/\partial u$

Figure 2.17: The Local Differential Geometry around a Point p. The parametric partial derivatives of the surface, $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$, lie in the tangent plane but are not necessarily orthogonal. The surface normal **n** is given by the cross product of $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$. The vectors $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ (not shown here) record the differential change in surface normal as we move u and v along the surface.

`DifferentialGeometry` 102
`Shape` 108

This representation implicitly assumes that shapes have a parametric description—that for some range of (u, v) values, points on the surface are given by some function f such that $\mathrm{p} = f(u, v)$. Although this isn't true for all shapes, all of the shapes that `pbrt` supports do have at least a local parametric description, so we will stick with the parametric representation since this assumption is helpful elsewhere (e.g., for antialiasing of textures in Chapter 10).

⟨*DifferentialGeometry Declarations*⟩ ≡
```
struct DifferentialGeometry {
    DifferentialGeometry() { u = v = 0.; shape = NULL; }
    ⟨DifferentialGeometry Public Methods⟩
    ⟨DifferentialGeometry Public Data 102⟩
};
```

The names of the member variables directly correspond to the quantities they represent. Here we have named the `Normal` member variable `nn` with a second "n" to reflect the fact that it is a normalized version of the surface normal.

⟨*DifferentialGeometry Public Data*⟩ ≡ **102**
```
Point p;
Normal nn;
float u, v;
const Shape *shape;
```

We also need to store the partial derivatives of the surface position and the surface normal:

⟨*DifferentialGeometry Public Data*⟩ +≡ **102**
```
Vector dpdu, dpdv;
Normal dndu, dndv;
```

The `DifferentialGeometry` constructor only needs a few parameters—the point of interest, the partial derivatives of position and normal, and the (u, v) coordinates. It computes the normal as the cross product of the partial derivatives.

⟨*DifferentialGeometry Method Definitions*⟩ ≡
```
DifferentialGeometry::DifferentialGeometry(const Point &P,
        const Vector &DPDU, const Vector &DPDV,
        const Normal &DNDU, const Normal &DNDV,
        float uu, float vv, const Shape *sh)
    : p(P), dpdu(DPDU), dpdv(DPDV), dndu(DNDU), dndv(DNDV) {
    ⟨Initialize DifferentialGeometry from parameters 102⟩
    ⟨Adjust normal based on orientation and handedness 103⟩
}
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩ ≡ **102**
```
nn = Normal(Normalize(Cross(dpdu, dpdv)));
u = uu;
v = vv;
shape = sh;
```

`Cross()` 62
`DifferentialGeometry` 102
`DifferentialGeometry::nn` 102
`DifferentialGeometry::shape` 102
`DifferentialGeometry::u` 102
`DifferentialGeometry::v` 102
`Normal` 65
`Point` 63
`Shape` 108
`Vector` 57
`Vector::Normalize()` 63

The surface normal has special meaning to pbrt, which assumes that, for closed shapes, the normal is oriented such that it points to the "outside" of the shape. For example, this assumption will be used later when we need to decide if a ray is entering or leaving the volume enclosed by a shape. Furthermore, for geometry used as an area light source, light is emitted from only the side of the surface that the normal points toward; the other side is black.

Because normals have this special meaning, pbrt provides a mechansim for the user to reverse the orientation of the normal, flipping it to point in the opposite direction. The ReverseOrientation directive in pbrt's input file flips the normal to point in the opposite, nondefault direction. Therefore, it will be necessary to check if the given Shape has this flag set and, if so, switch the normal's direction.

However, one other factor plays into the orientation of the normal and must be accounted for here as well. If the Shape's transformation matrix has switched the handedness of the object coordinate system from pbrt's default left-handed coordinate system to a right-handed one, we need to switch the orientation of the normal as well. To see why this is so, consider a scale matrix $\mathbf{S}(1, 1, -1)$. We would naturally expect this scale to switch the direction of the normal, although because we compute the normal by $\mathbf{n} = \partial \mathrm{p}/\partial u \times \partial \mathrm{p}/\partial v$ it can be shown that:

$$\begin{aligned}\mathbf{S}(1, 1, -1)\frac{\partial \mathrm{p}}{\partial u} \times \mathbf{S}(1, 1, -1)\frac{\partial \mathrm{p}}{\partial v} &= \mathbf{S}(-1, -1, 1)\frac{\partial \mathrm{p}}{\partial u} \times \frac{\partial \mathrm{p}}{\partial v} \\ &= \mathbf{S}(-1, -1, 1)\mathbf{n} \neq \mathbf{S}(1, 1, -1)\mathbf{n}.\end{aligned}$$

Therefore, it is also necessary to manually flip the normal's direction if the transformation switches the handedness of the coordinate system, since the flip won't be accounted for by the computation of the normal's direction using the cross product.

The normal's direction is swapped if one but not both of these two conditions is met; if both were met, their effect would cancel out. The exclusive-OR operation tests this condition.

⟨*Adjust normal based on orientation and handedness*⟩ ≡ 102

```
if (shape && (shape->ReverseOrientation ^ shape->TransformSwapsHandedness))
    nn *= -1.f;
```

FURTHER READING

DeRose, Goldman, and their collaborators have argued for an elegant "coordinate-free" approach to describing vector geometry for graphics, where the fact that positions and directions happen to be represented by (x, y, z) coordinates with respect to a particular coordinate system is deemphasized and where points and vectors themselves record which coordinate system they are expressed in terms of (Goldman 1985; DeRose 1989; Mann, Litke, and DeRose 1997). This makes it possible for a software layer to ensure that common errors like adding a vector in one coordinate system to a point in another coordinate system are transparently handled by transforming them to a common coordinate system first. We have not followed this approach in pbrt, although the principles behind

DifferentialGeometry::nn 102

DifferentialGeometry::shape 102

Shape::ReverseOrientation 108

Shape::TransformSwapsHandedness 108

this approach are well worth understanding and keeping in mind when working with coordinate systems in computer graphics.

Schneider and Eberly's *Geometric Tools for Computer Graphics* is influenced by the coordinate-free approach and covers the topics of this chapter in much greater depth (Schneider and Eberly 2003). It is also full of useful geometric algorithms for graphics. A classic and more traditional introduction to the topics of this chapter is *Mathematical Elements for Computer Graphics* by Rogers and Adams (1990). Note that their book uses a row-vector representation of points and vectors, however, which means that our matrices would be transposed when expressed in their framework, and that they multiply points and vectors by matrices to transform them ($p\mathbf{M}$), rather than multiplying matrices by points as we do ($\mathbf{M}p$). Homogeneous coordinates were only briefly mentioned in this chapter, although they are the basis of projective geometry, where they are the foundation of many elegant algorithms. Stolfi's book is an excellent introduction to this topic (Stolfi 1991).

There are many good books on linear algebra and vector geometry. We have found Lang (1986) and Buck (1978) to be good references on these respective topics. See also Akenine-Möller et al.'s *Real-Time Rendering* book (2008) for a solid graphics-based introduction to linear algebra.

The subtleties of how normal vectors are transformed were first widely understood in the graphics community after articles by Wallis (1990) and Turkowski (1990c).

Shoemake (1985) introduced quaternions to graphics and showed their utility for animating rotations. Using polar matrix decomposition for animating transformations was described by Shoemake and Duff (1992); Higham (1986) developed the algorithm for extracting the rotation from a composite rotation and scale matrix by successively adding the matrix to its inverse transpose. Shoemake's chapters in *Graphics Gems* (1991, 1994a) respectively give more details on the derivation of the conversion from matrices to quaternions and the implementation of polar matrix decomposition.

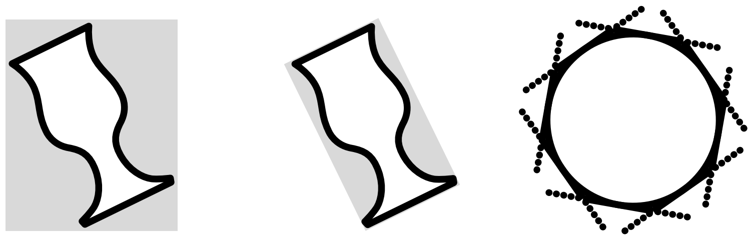
We followed Blow's derivation of spherical linear interpolation (2004) in our exposition in this chapter. Bloom et al. (2004) discuss desirable properties of interpolation of rotations for animation in computer graphics and which approaches deliver which of these properties. For more sophisticated approaches to rotation interpolation, see Ramamoorthi and Barr (1997) and Buss and Fillmore (2001).

EXERCISES

❶ 2.1 Find a more efficient way to transform axis-aligned bounding boxes by taking advantage of the symmetries of the problem: because the eight corner points are linear combinations of three axis-aligned basis vectors and a single corner point, their transformed bounding box can be found much more efficiently than by the method we presented (Arvo 1990).

❷ 2.2 Reimplement the core geometry classes using a vector instruction set such as Intel's SSE. Evaluate the performance implications on scenes with a variety of characteristics. What can you conclude about the effectiveness of this approach?

❷ 2.3 Instead of boxes, tighter bounds around objects could be computed by using the intersections of many nonorthogonal slabs. Extend the bounding box class in `pbrt` to allow the user to specify a bound comprised of arbitrary slabs.



❶ 2.4 Change `pbrt` so that it transforms `Normal`s just like `Vector`s and create a scene that gives a clearly incorrect image due to this bug. (Don't forget to eliminate this change from your copy of the source code when you're done!)

❷ 2.5 If only the translation components of a transformation are time varying, for example, then the `AnimatedTransform` implementation does unnecessary computation in interpolating between two rotations that are the same. Modify the `AnimatededTransform` impementation so that it avoids this work in cases where the full generality of its current implementation isn't necessary. How much of a performance difference do you observe for scenes where your optimizations are applicable?

❸ 2.6 The `AnimatedTransform::MotionBounds()` method is not guaranteed to be robust: because it samples the animated transform at a number of discrete times, it's not guaranteed that it will locate the extrema of motion of the bounding box and thus the computed bound may not actually bound the full motion of the object. Can you derive a more robust method that gives *guaranteed* conservative bounds?

A reasonable approach might be to write out formulas that give the (x, y, z) components of points as they are animated as a function of time, in terms of the values of decomposed matrix components. The extrema of each of these will come when their derivative with respect to time is zero; given the coordinates of the corners of the bounding box, is it possible to solve for the time values where these positions reach extrema? If it's not possible to solve for them in closed form, can you guide sampling of the parameter space based on a search for zero derivatives?

`AnimatedTransform` 96

`AnimatedTransform::MotionBounds()` 100

`Normal` 65

`Vector` 57