⟨*Compute tangents of boundary face*⟩ ?≡ ???

```
S = pRing[valence - 1] - pRing[0];
if (valence == 2)
    T = Vector3f(pRing[0] + pRing[1] - 2 * vertex->p);
else if (valence == 3)
    T = pRing[1] - vertex->p;
else if (valence == 4) // regular
    T = Vector3f(-1 * pRing[0] +  2 * pRing[1] + 2 * pRing[2] +
                 -1 * pRing[3] + -2 * vertex->p);
else {
    Float theta = Pi / float(valence-1);
    T = Vector3f(std::sin(theta) * (pRing[0] + pRing[valence - 1]));
    for (int k = 1; k < valence-1; ++k) {
        Float wt = (2 * std::cos(theta) - 2) * std::sin((k) * theta);
        T += Vector3f(wt * pRing[k]);
    }
    T = -T;
}
```

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

* 3.9 MANAGING ROUNDING ERROR

Thus far, we've been discussing ray–shape intersection algorithms purely with respect to the mathematics of their operation with the real numbers. This approach has gotten us far, although the fact that computers can only represent finite quantities and therefore can't actually represent all of the the real numbers is important. In place of real numbers, computers use floating-point numbers, which have fixed storage requirements. However, error may be introduced each time a floating-point operation is performed, since the result may not be representable in the floating-point numbers.

The accumulation of this error means that computed ray–shape intersection points may be above or below the actual surface. This leads to a problem: when new rays are traced starting from computed intersection points for shadow rays and reflection rays, if the ray origin is below the actual surface, we may find an incorrect re-intersection with the surface. Conversely, if the origin is too far above the surface, shadows and reflections may appear detached. (See Figure 3.31.)

Typical practice to address this issue in ray tracing is to offset spawned rays by a fixed "epsilon" value, ignoring any intersections along the ray $\mathrm{p} + t\mathbf{d}$ closer than some $t_{\min}$ value. Figure 3.32 shows why this approach requires fairly high $t_{\min}$ values to work effectively: if the spawned ray is fairly oblique to the surface, incorrect ray intersections may occur quite some distance from the ray origin. Unfortunately, large $t_{\min}$ values cause ray origins to be relatively far from the

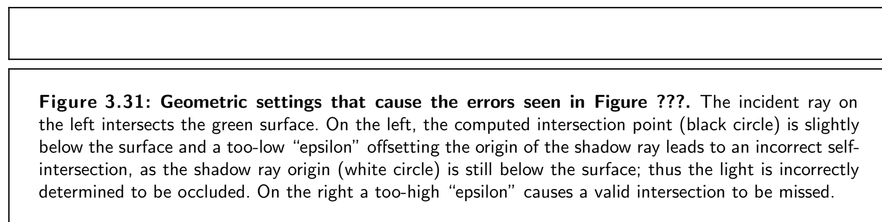

Figure 3.31: Geometric settings that cause the errors seen in Figure ???. The incident ray on the left intersects the green surface. On the left, the computed intersection point (black circle) is slightly below the surface and a too-low "epsilon" offsetting the origin of the shadow ray leads to an incorrect self-intersection, as the shadow ray origin (white circle) is still below the surface; thus the light is incorrectly determined to be occluded. On the right a too-high "epsilon" causes a valid intersection to be missed.

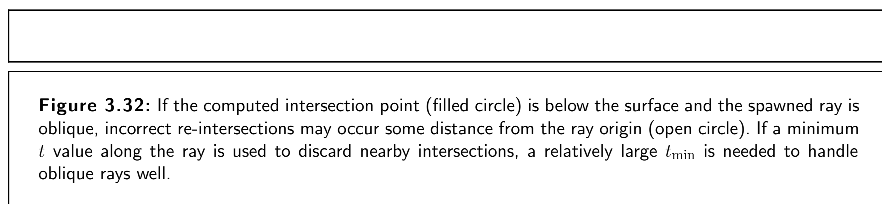

Figure 3.32: If the computed intersection point (filled circle) is below the surface and the spawned ray is oblique, incorrect re-intersections may occur some distance from the ray origin (open circle). If a minimum t value along the ray is used to discard nearby intersections, a relatively large $t_{\min}$ is needed to handle oblique rays well.

original intersection points, which in turn can cause loss of fine detail in shadows and reflections.

In this section, we'll introduce the ideas underlying floating-point arithmetic and describe techniques for analyzing the error in floating-point computations. We'll then apply these methods to the ray–shape algorithms introduced earlier in this chapter and show how to compute ray intersection points with bounded error, which in turn allows us to conservatively position ray origins so that incorrect self-intersections are never found, while keeping ray origins extremely close to the actual intersection point. In turn, no "epsilon" values are needed.

3.9.1 FLOATING-POINT ARITHMETIC

Computation must be performed on a finite representation of numbers that fits in a finite amount of memory; the infinite set of real numbers just can't be represented on a computer. One such finite representation is *fixed point*, where given a 16-bit integer, for example, one might say that the first 8 bits are used to represent the whole numbers from 0 to 255, and that the second 8 bits are used to represent fractions with equal spacing 1/256. With this representation, the pair of 8-bit numbers $(5, 64)$ would represent the value $5 + 64/256 = 5.25$. Fixed-point numbers can be implemented efficiently using integer arithmetic operations (a property that made them popular on early PCs that didn't support floating-point computation), but they suffer from a number of shortcomings; among them, the maximum number they can represent is limited, and they aren't able to accurately represent very small numbers near zero.

An alternative representation for real numbers on computers is *floating-point* numbers. These are based on representing numbers with a sign, a significand[10],

10 The word "mantissa" is often used in place of significand, though floating-point purists note that "mantissa" has a different meaning in the context of logarithms, and thus prefer "significand". We follow this usage here.

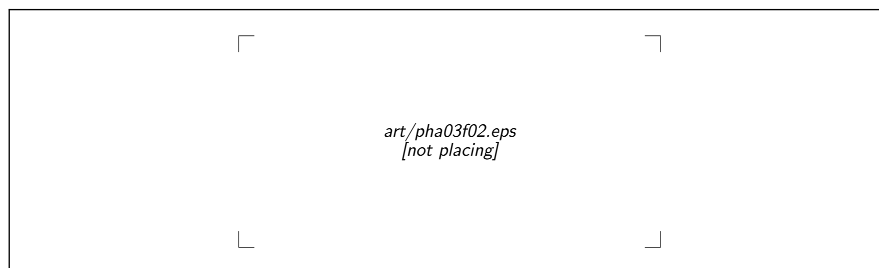

Figure 3.33: Due to finite floating-point precision and round-off error, when the intersection of a ray is found with a shape, the computed intersection point may lie slightly above or slightly below the true intersection point. This can lead to rendering errors when reflected and shadow rays are traced starting from the computed intersection point, as incorrect self-intersections with the surface may be detected.

and an exponent: essentially, the same representation as scientific notation, but with a fixed number of digits devoted to significand and exponent. (In the following, we will assume base-2 digits exclusively.) This representation makes it possible to represent and perform computations on numbers with a wide range of magnitudes while using a fixed amount of storage.

Programmers using floating-point arithmetic are generally aware that floating-point is imprecise; this understanding sometimes leads to a belief that floating-point arithmetic is unpredictable. In this section we'll see that floating-point arithmetic has a carefully-designed foundation that in turn makes it possible to compute conservative bounds on the error introduced in a particular computation. For ray tracing calculations, this error is often surprisingly small.

Modern CPUs and GPUs nearly ubiquitously implement a model of floating-point arithmetic based on a standard promulgated by the Institute of Electrical and Electronics Engineers (XXXX year cite XXX). (Henceforth when we refer to floats, we will specifically be referring to 32-bit floating-point numbers as specified by IEEE 754.) The IEEE 754 technical standard specifies the format of floating-point numbers in memory as well as specific rules for precision and rounding of floating-point computations; it is these rules that make it possible to reason rigorously about the error present in a given floating-point value.

Floating-Point Representation

The IEEE standard specifies that 32-bit floats are represented with a sign bit, 8 bits for the exponent, and 23 bits for the significand. With 8 bits, the exponent e_{b} ranges from 0 to 255; the actual exponent used, e_{b}, is computed by biasing e:

$$e_{\mathrm{b}} = e - 127.$$

The significand actually has 24 bits of precision when a *normalized* floating-point value is stored. When a number expressed with significand and exponent is normalized, there are no leading zeros in the significand. In binary, this means that the leading digit of the significand must be one; in turn, there's no need to

store this value explicitly. Thus, the implicit leading one digit with the 23 digits encoding the fractional part of the significand give a total of 24 bits of precision.

Thus, given a sign $s = \pm 1$, significand m, and exponent e, the corresponding floating-point value is

$$s \times 1.m \times 2^{e-127}.$$

For example, with a normalized significand, the floating-point number 6.5 is written as $1.101_2 \times 2^2$, where the 2 subscript denotes a base-2 value. (If binary decimals aren't immediately intuitive, note that the first number to the right of the decimal contributes $2^{-1} = 1/2$, and so forth.) Thus, we have

$$(1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^2 = 1.625 \times 2^2 = 6.5.$$

$e_\mathrm{b} = 2$, so $e = 129 = 1000001_2$ and $m = 10100000000000000000000_2$.

Floats are laid out in memory with the sign bit at the most significant bit of the 32-bit value (with negative signs encoded with a one bit), then the exponent, and the significand. Thus, for the value 6.5 the binary in-memory representation of the value is

$$0\ \ 10000001\ \ 10100000000000000000000 = \text{0x40d00000}.$$

Similarly the floating-point value 1.0 has $m = 0 \cdots 0^2$ and $e_\mathrm{b} = 0$, so $e = 127 = 01111111^2$ and so its binary representation is:

$$0\ \ 01111111\ \ 00000000000000000000000 = \text{0x3f800000}.$$

This hexadecimal number is a value worth remembering, as it often comes up in memory dumps when debugging.

An implication of this representation is that the spacing between representable floats between two powers of two is uniform throughout the range. (It corresponds to increments of the significand bits by one). In a range $[2^e, 2^{e+1}]$, the spacing is

$$2^{e-23}. \tag{3.5}$$

Thus, for floating-point numbers between 1 and 2, $e = 0$, and the spacing between floating-point values is $2^{-23} \approx 1.19209 \ldots \times 10^{-7}$. This spacing is also referred to as the magnitude of a *unit in last place* ("ulp"); note that the magnitude of an ulp is determined by the floating-point value that it is with respect to—ulps are relatively larger at numbers with bigger magnitudes than they are at numbers with smaller magnitudes.

As we've described the representation so far, it's impossible to exactly represent zero as a floating-point number. This is obviously an unacceptable state of affairs, so the minimum exponent $e = 0$, or $e_\mathrm{b} = -127$ is set aside for special treatment. With this exponent, the floating-point value is interpreted as not having the implicit leading one bit in the significand, which means that a significand of all zero bits results in

$$s \times 0.0 \cdots 0_2 \times 2^{-127} = 0.$$

Eliminating the leading one significand bit also makes it possible to represent *denormalized* numbers: if the leading one was present, then the smallest 32-bit float would be

$$1.0 \cdots 0_2 \times 2^{-127} \approx 5.8774718 \times 10^{-39}.$$

Without the leading one bit, the minimum value is

$$0.00 \cdots 1_2 \times 2^{-126} = 2^{-126} \times 2^{-23} \approx 1.4012985 \times 10^{-45}.$$

Providing some capability to represent these small values can make it possible to avoid needing to round very small values to zero.

Note that there is both a "positive" and "negative" zero value with this representation. This is mostly transparent to the programmer. For example, the standard guarantees that the comparison `-0.0 == 0.0` evaluates to true, even though the in-memory representations of these two values are different.

The maximum exponent, $e = 255$, is also reserved for special treatment. Therefore, the largest regular floating-point value that can be represented with $e = 254$ or $e_b = 127$ is approximately

$$3.402823 \ldots \times 10^{38}.$$

With $e_b = 255$, if the significand bits are all zero, the value corresponds to positive or negative infinity, according to the sign bit. Infinite values result when performing computations like $1/0$ in floating-point, for example. No arithmetic operations with infinity are valid, but in comparisons, positive infinity is larger than any non-infinite value and similarly for negative infinity.

The `MaxFloat` and `Infinity` constants are initialized to be the largest representable and "infinity" floating-point values, respectively. We make them available in a separate constant so that code that uses these values doesn't need to use the wordy C++ standard library call to get their value.

⟨*Global Constants*⟩ ?≡ ???

```
static constexpr Float MaxFloat = std::numeric_limits<Float>::max();
static constexpr Float Infinity = std::numeric_limits<Float>::infinity();
```

With $e_b = 255$, non-zero significand bits correspond to special "not a number" (NaN) values, which result from operations like taking the square root of a negative number, trying to compute $0/0$, or performing an operation with infinity as an operand. NaNs propagate through computations: any arithmetic operation where one of the operands is a NaN itself always returns NaN. Thus, if a NaN emerges from a long chain of computations, we know that something went awry somewhere along the way. In debug builds, `pbrt` has many `Assert()` statements that check for NaN values, as we almost never expect them to come up in the

regular course of events. Any comparison with a NaN value returns false; thus, checking for `x != x` serves to check if a value is not a number.[11]

Utility Routines

The C++ standard library provides a `std::isnan()` function to check for not-a-number for `float` and `double` types, but because template classes like `Bounds2` are sometimes instantiated with an integer type for their indices, if those functions were used directly, their assertions to check for NaNs would sometimes try to check whether an integer value was not-a-number. Though doing so should be innocuous, it runs afoul of C++ template overloading rules, since it's unclear whether the `float` or `double` `isnan()` variant should be used.

Therefore, `pbrt` provides a custom `IsNaN()` function that dispatches to `std::isnan()` for `float` and `double` and returns `false` otherwise. Fairly arcane C++-isms are required to do this; here we use functionality from the `type_traits` header in the standard library to define two versions of the function, one for integral (i.e. not floating-point), and one for non-integral (floating-point) types.

⟨*Global Inline Functions*⟩ ?≡ ???

```
template <typename T>
typename std::enable_if<std::is_integral<T>::value, bool>::type
IsNaN(T val) {
    return false;
}

template <typename T>
typename std::enable_if<!std::is_integral<T>::value, bool>::type
IsNaN(T val) {
    return std::isnan(val);
}
```

For certain low-level operations, it can be useful to be able to interpret a floating-point value in terms of its constituent bits and to convert the bits representing a floating-point value to an actual `float` or `double`.

One natural approach to this would be to take a pointer to a value to be converted and cast it to a pointer to the other type:

```
float f = ...;
uint32_t bits = *((uint32_t *)&f);
```

However, modern versions of C++ specify that it's illegal to cast a pointer of one type, `float`, to a different type, `uint32_t`. (This restriction allows the compiler to optimize more aggressively in its analysis of whether two pointers may point to the same memory location, which can inhibit storing values in registers.)

---

11 This is one of a few places where compilers must not perform seemingly obvious and safe algebraic simplifications with expressions that include floating-point values—such comparisons must not be simplified to `false`.

Another common approach is to use an `enum` with elements of both types, assigning to one type and reading from the other:

```
enum FloatBits {
    float f;
    uint32_t ui;
};
FloatBits fb;
fb.f = ...;
uint32_t bits = fb.ui;
```

This, too, is illegal: the C++ standard says that reading from a different element of a union than the one last one assigned to is undefined behavior.

These conversions can be properly made using `memcpy()` to copy from a pointer to the source type to a pointer to the destination type:

⟨*Global Inline Functions*⟩ ?≡ ???

```
inline uint32_t FloatToBits(float f) {
    uint32_t ui;
    memcpy(&ui, &f, sizeof(float));
    return ui;
}
```

⟨*Global Inline Functions*⟩ ?≡ ???

```
inline float BitsToFloat(uint32_t ui) {
    float f;
    memcpy(&f, &ui, sizeof(uint32_t));
    return f;
}
```

While a call to the `memcpy()` function may seem gratuitously expensive to avoid these issues, in practice good compilers turn this into a no-op and just reinterpret the contents of the register or memory as the other type. (Versions of these functions that convert between `double` and `uint64_t` are also available in `pbrt`, but are similar and are therefore not included here.)

http://randomascii.wordpress.com/2012/01/11/tricks-with-the-floating-point-format/ has some nice discussion decomposing floats in memory

These conversions can be used to implement functions that bump a floating-point value up or down to the next greater or next smaller representable floating-point value. These functions are useful for some conservative rounding operations that we'll need in code to follow. Thanks to the specifics of the in-memory representation of floats, these operations are quite efficient.

⟨*Global Inline Functions*⟩ ?≡ ???

```
inline float NextFloatUp(float v) {
    ⟨Handle infinity and negative zero for NextFloatUp()⟩
    ⟨Advance v to next higher float⟩
}
```

There are two important special cases: if `v` is positive infinity, then this function just returns `v` unchanged. Negative zero is skipped forward to positive zero before continuing on to the code that advances the significand. This step must be handled explicitly, since the bit patterns for -0.0 and 0.0 aren't adjacent.

⟨*Handle infinity and negative zero for* `NextFloatUp()`⟩ ?≡ ???

```
if (std::isinf(v) && v > 0.)
    return v;
if (v == -0.f)
    v = 0.f;
```

Conceptually, given a floating-point value we want to increase the significand by one, where if the result overflows, the significand is reset to zero and the exponent is increased by one. Fortuitously, adding one to the in-memory representation of a float achieves this: because the exponent lies at the high bits above the significand, adding one to the low bit of the significand will cause a one to be carried all the way up into the exponent if the significand is all ones and otherwise will advance to the next higher significand for the current exponent.[12] For negative values, subtracting one from the bit representation advances to the next value.

⟨*Advance* `v` *to next higher float*⟩ ?≡ ???

```
uint32_t ui = FloatToBits(v);
if (v >= 0.) ++ui;
else         --ui;
return BitsToFloat(ui);
```

The `NextFloatDown()` function, not included here, follows the same logic, but effectively in reverse. `pbrt` also provides versions of these functions for `double`s.

Arithmetic Operations

IEEE 754 provides important guarantees about the properties of floating-point arithmetic: specifically, it guarantees that addition, subtraction, multiplication, division, and square root give the same results given the same inputs and that these results are the floating-point number that is closest to the result of the underlying computation if it had been performed in infinite-precision arithmetic.[13] It is remarkable that this is possible on finite-precision digital computers at all; one of the achievements in IEEE 754 was the demonstration that this level of accuracy is possible and can be implemented fairly efficiently in hardware.

Using circled operators to denote floating-point arithmetic operations and `sqrt` for floating-point square root, these precision guarantees can be written as:

---

12 These functions are equivalent to `std::nextafter(v, Infinity)` and `std::nextafter(v, -Infinity)`, but are more efficient since they don't try to handle NaN values or deal with signaling floating-point exceptions.

13 IEEE float allows the user to select one of a number of rounding modes, but we will assume the default—round to nearest even—here.

$$
\begin{aligned}
a \oplus b &= \text{round}(a + b) \\
a \ominus b &= \text{round}(a - b) \\
a \otimes b &= \text{round}(a * b) \\
a \oslash b &= \text{round}(a / b) \\
\texttt{sqrt}(\texttt{a}) &= \text{round}(\sqrt{a})
\end{aligned}
\tag{3.6}
$$

where round(x) indicates the result of rounding a real number to the closest floating-point value.

This bound on the rounding error can also be represented with an interval of real numbers: for example, for addition, we can say that the rounded result is within an interval

$$
\begin{aligned}
a \oplus b = \text{round}(a + b) &\subset (a + b)(1 \pm \epsilon) \\
&= [(a + b)(1 - \epsilon), (a + b)(1 + \epsilon)]
\end{aligned}
\tag{3.7}
$$

for some ϵ. The amount of error introduced from this rounding can be no more than half the floating-point spacing at $a + b$—if it was more than half the floating-point spacing, then it would be possible to round to a different floating-point number with less error (Figure 3.34).

For 32-bit floats, we can bound the floating-point spacing at $a + b$ from above using Equation (3.5) (i.e. an ulp at that value) by $(a + b)2^{-23}$, so half the spacing is bounded from above by $(a + b)2^{-24}$ and so $|\epsilon| \leq 2^{-24}$. This bound is the *machine epsilon*[14]. For 32-bit floats, $\epsilon_{\text{m}} = 2^{-24} \approx 5.960464 \ldots \times 10^{-8}$.

⟨*Global Constants*⟩ ?≡ ???

```
static constexpr Float MachineEpsilon =
        std::numeric_limits<Float>::epsilon() * 0.5;
```

Thus, we have

$$
\begin{aligned}
a \oplus b = \text{round}(a + b) &\subset (a + b)(1 \pm \epsilon_{\text{m}}) \\
&= [(a + b)(1 - \epsilon_{\text{m}}), (a + b)(1 + \epsilon_{\text{m}})].
\end{aligned}
$$

Analogous relations hold for the other arithmetic operators and the square root operator.[15]

A number of useful properties follow directly from Equation (3.6). For a floating-point number x,

- $1 \otimes x = x$.
- $x \oslash x = 1$.
- $x \oplus 0 = x$.
- $x \ominus x = 0$.

14 The C and C++ standards unfortunately define the machine epsilon in their own special way, which is that it is the magnitude of one ulp above the number 1. For 32-bit float, this value is 2^{-23}, which is twice as large as the machine epsilon as the term is used in numerical analysis.

15 This bound assumes that there's no overflow or underflow in the computation; these possibilities can be easily handled (Higham 2002, p. 56), but aren't generally important for our application here.

Figure 3.34: The IEEE standard specifies that floating-point calculations must be implemented as if the calculation was performed with infinite-precision real numbers and then rounded to the nearest representable float. Here, an infinite precision result in the real numbers is denoted by a filled dot, with the representable floats around it denoted by ticks in a number line. We can see that the error introduced by rounding to the nearest float, δ, can be no more than half the spacing between floats.

- $2 \otimes x$ and $x \oslash 2$ are exact; no rounding is performed to compute the final result. More generally, any multiplication by or division by a power of two gives an exact result (assuming there's no overflow or underflow).
- $x \oslash 2^i = x \otimes 2^{-i}$ for all integer i, assuming 2^i doesn't overflow.

All of these properties follow from the principle that the result must be the nearest floating-point value to the actual result; when the result can be represented exactly, the exact result must be computed.

Error Propagation

Using the guarantees of IEEE floating-point arithmetic, it is possible to develop methods to analyze and bound the error in a given floating-point computation. For more details on this topic, see the excellent book by Higham (2002), as well as Wilkinson's earlier classic (1963).[16]

Two measurements of error are useful in this effort: absolute and relative. If we perform some floating point computation and get a rounded result $\tilde{a}$, we say that the magnitude of the difference between $\tilde{a}$ and the result of doing that computation in the real numbers is the *absolute error*, δ_{a}:

$$\delta_{\mathrm{a}} = |\tilde{a} - a|.$$

Relative error, δ_{r}, is the ratio of the absolute error to the precise result:

$$\delta_{\mathrm{r}} = \left| \frac{\tilde{a} - a}{a} \right| = \left| \frac{\delta_{\mathrm{a}}}{a} \right|, \tag{3.8}$$

as long as $a \neq 0$. Using the definition of relative error, we can thus write the computed value $\tilde{a}$ as a perturbation of the exact result a:

$$\tilde{a} = a \pm \delta_{\mathrm{a}} = a(1 \pm \delta_{\mathrm{r}}).$$

As a first application of these ideas, consider computing the sum of four numbers, a, b, c, and d, represented as floats. If we compute this sum as `r = (((a + b) + c) + d)`, Equation (3.7) gives us

$$\begin{aligned}(((a \oplus b) \oplus c) \oplus d) &\subset ((((a + b)(1 \pm \epsilon_{\mathrm{m}})) + c)(1 \pm \epsilon_{\mathrm{m}}) + d)(1 \pm \epsilon_{\mathrm{m}}) \\ &= (a + b)(1 \pm \epsilon_{\mathrm{m}})^3 + c(1 \pm \epsilon_{\mathrm{m}})^2 + d(1 \pm \epsilon_{\mathrm{m}}).\end{aligned}$$

16 Handling denormalized floats in this sort of analysis requires special treatment. We will ignore them in our analysis here, though extending the analysis to account for them is fairly straightforward (Higham 2002).

Because ϵ_m is small, higher-order powers of ϵ_m can be bounded by an additional ϵ_m term, and so we can bound the $(1 \pm \epsilon_m)^n$ terms with

$$(1 \pm \epsilon_m)^n \le (1 \pm (n+1)\epsilon_m).$$

(As a practical matter, $(1 \pm n\epsilon_m)$ almost bounds these terms, since higher powers of ϵ_m get very small very quickly, but the above is a fully conservative bound.)

This bound lets us simplify the result of the addition to:

$$(a+b)(1 \pm 4\epsilon_m) + c(1 \pm 3\epsilon_m) + d(1 \pm 2\epsilon_m) = \\ a+b+c+d+[\pm 4\epsilon_m(a+b) \pm 3\epsilon_m c \pm 2\epsilon_m d].$$

The term in square brackets gives the absolute error: its magnitude is bounded by

$$4\epsilon_m|a+b| + 3\epsilon_m|c| + 2\epsilon_m|d|. \tag{3.9}$$

Thus, if we add four floating-point numbers together with the above parenthesization, we can be certain that the difference between the final rounded result and the result we would get if we added them with infinite-precision real numbers is bounded by Equation (3.9); this error bound is easily computed given specific values of a, b, c, and d.

This is a fairly interesting result; we see that the magnitude of $a + b$ makes a relatively large contribution to the error bound, especially compared to d. (This result gives a sense for why, if adding a large number of floating-point numbers together, sorting them from small to large magnitudes generally gives a result with a lower final error than an arbitrary ordering.)

Our analysis here has implicitly assumed that the compiler would generate instructions according to the expression used to define the sum. Compilers are required to follow the form of the given floating-point expressions in order to not break carefully crafted computations that may have been designed to minimize round-off error. Here again is a case where certain transformations that would be valid on expressions with integers can not be applied when floats are involved.

What happens if we change the expression to the algebraically equivalent `float r = (a + b) + (c + d)`? This corresponds to the floating-point computation

$$((a \oplus b) \oplus (c \oplus d)).$$

If we apply the same process of applying Equation (3.7), expanding out terms, converting higher-order $(1 \pm \epsilon_m)^n$ terms to $(1 \pm (n+1)\epsilon_m)$, we get absolute error bounds of

$$3\epsilon_m|a+b| + 3\epsilon_m|c+d|,$$

which are lower than the first formulation if $|a + b|$ is relatively large, but higher if $|d|$ is relatively large.

This approach to computing error is known as *forward error analysis*; given inputs to a computation, we can apply a fairly mechanical process that provides conservative bounds on the error in the result. The derived bounds in the result may overstate the actual error—in practice, the signs of the error terms are

often mixed, so that there is cancellation when they are added.[17] An alternative approach is *backward error analysis*, which treats the computed result as exact and provides bounds on perturbations on the inputs that give the same result. This approach can be more useful then analyzing the stability of a numerical algorithm, but is less applicable to deriving conservative error bounds on the geometric computations we're interested in here.

The conservative bounding of $(1 \pm \epsilon_m)^n$ by $(1 \pm (n+1)\epsilon_m)$ is somewhat unsatisfying since it adds a whole ϵ_m term purely to conservatively bound the sum of various higher powers of ϵ_m. Higham (2002, Section 3.1) gives an approach to more tightly bound products of $(1 \pm \epsilon_m)$ error terms. If we have $(1 \pm \epsilon_m)^n$, it can be shown that this value is bounded by $1 + \theta_n$, where

$$|\theta_n| \le \frac{n\,\epsilon_m}{1 - n\,\epsilon_m}, \tag{3.10}$$

as long as $n\,\epsilon_m < 1$ (which will certainly be the case for the calculations we're considering.) Note that the denominator of this expression will be just less than one for reasonable n values, so just barely increases $n\epsilon_m$ to achieve a conservative bound.

We will denote this bound by γ_n:

$$\gamma_n = \frac{n\,\epsilon_m}{1 - n\,\epsilon_m}.$$

⟨*Global Inline Functions*⟩ ?≡ ???

```
inline Float gamma(int n) {
    return (n * MachineEpsilon) / (1  - n * MachineEpsilon);
}
```

Even better, quotients of $(1 \pm \epsilon_m)^n$ terms can be bounded with the γ function. Given

$$\frac{(1 \pm \epsilon_m)^m}{(1 \pm \epsilon_m)^n},$$

the interval is bounded by $(1 \pm \gamma_{m+n})$. Thus, γ can be used to collect ϵ_m terms from both sides of an equality over to one side by dividing them through; this will be useful in some of the following derivations.

When working with these error intervals, it's important to remember that because $(1 \pm \epsilon_m)$ terms represent intervals, canceling them incorrect:

$$\frac{(1 \pm \epsilon_m)^m}{(1 \pm \epsilon_m)^n} \ne (1 \pm \epsilon_m)^{m-n}.$$

Using the γ notation, our bound on the error of the sum of the four values is

$$|a+b|\gamma_3 + |c|\gamma_2 + |d|\gamma_1.$$

---

17 Some numerical analysts use a rule of thumb that the error in practice is often close to the square root of the forward error bounds, thanks to the cancellation of error in intermediate results.

Given inputs to some computation that themselves carry some amount of error, it's instructive to see how this error is carried through various elementary arithmetic operations. Given two values, $a(1 \pm \gamma_i)$ and $b(1 \pm \gamma_j)$ that each carry some accumulated error from earlier operations, consider their product. Using the definition of $\otimes$, the result is in the interval:

$$a(1 \pm \gamma_i) \otimes b(1 \pm \gamma_j) \subset ab(1 \pm \gamma_{i+j+1}),$$

where we've used the relationship $(1 \pm \gamma_i)(1 \pm \gamma_j) \subset (1 \pm \gamma_{i+j})$, which follows directly from Equation (3.10).

The relative error in this result is bounded by:

$$\left| \frac{ab\, \gamma_{i+j+1}}{ab} \right| = \gamma_{i+j+1},$$

and so the final error is thus just roughly $(i+j+1)/2$ ulps at the value of the product—about as good as we might hope for given the error going into the multiplication. (The situation for division is similarly good.)

Unfortunately, with addition and subtraction, it's possible for the relative error to increase substantially. Using the same definitions of the values being operated on, consider

$$a(1 \pm \gamma_i) \oplus b(1 \pm \gamma_j),$$

which is in the interval

$$a(1 \pm \gamma_{i+1}) + b(1 \pm \gamma_{j+1}),$$

and so the absolute error is bounded by $|a|\gamma_{i+1} + |b|\gamma_{j+1}$.

If the signs of a and b are the same, then the absolute error is bounded by $|a+b|\gamma_{i+j+1}$ and the relative error is around $(i+j+1)/2$ ulps around the computed value.

However, if the signs of a and b differ (or, equivalently, they are the same but subtraction is performed), then the relative error can be quite high. Consider the case where $a \approx -b$: the relative error is

$$\frac{|a|\gamma_{i+1} + |b|\gamma_{j+1}}{a+b} \approx \frac{2|a|\gamma_{i+j+1}}{a+b}.$$

The numerator's magnitude is proportional to the original value $|a|$, yet is divided by a very small number, and thus the relative error is quite high. This substantial increase in relative error is called *catastrophic cancellation*. Equivalently, we can have a sense of the issue from the fact that the absolute error is in terms of the magnitude of $|a|$, though it's now in relation to a value much smaller than a.

Running Error Analysis

In addition to working out error bounds algebraically, we can also have the computer do this work for us as some computation is being performed. This approach is known as *running error analysis*. The idea behind it is simple: each time a floating-point operation is performed, we also have it compute terms that compute intervals based on Equation (3.6) to compute a running bound on the

error that has been accumulated so far. While this approach can have higher runtime overhead than deriving expressions that give error bound ahead of time, it can be convenient when derivations become unwieldy.

`pbrt` provides a simple `EFloat` class, which mostly acts like a regular `float` but uses operator overloading to provide all of the regular arithmetic operations on floats while computing these error bounds.

⟨*EFloat Public Methods*⟩ ?≡ ???

```
EFloat() { }
EFloat(float v, float err = 0.f) : v(v), err(err) {
#ifndef NDEBUG
    ld = v;
    Check();
#endif
}
```

`EFloat` maintains a computed value `v` and the absolute error bound, `err`.

⟨*EFloat Private Data*⟩ ?≡ ???

```
float v;
float err;
```

In debug builds, `EFloat` also maintains a highly-precise version of `v` that can be used as a reference value to compute an accurate approximation of the relative error. In optimized builds, we'd generally rather not pay the overhead for computing this additional value.

⟨*EFloat Private Data*⟩ ?≡ ???

```
#ifndef NDEBUG
long double ld;
#endif // DEBUG
```

The implementation of the addition operation for this class is essentially an implementation of the relevant definitions. We have:

$$\begin{aligned}(a \pm \delta_a) \oplus (b \pm \delta_b) &= ((a \pm \delta_a) + (b \pm \delta_b))(1 \pm \gamma_1) \\ &= a + b + [\pm\delta_a \pm \delta_b \pm (a+b)\gamma_1 \pm \gamma_1\delta_a \pm \gamma_1\delta_b].\end{aligned}$$

And so the absolute error (in brackets) is bounded by

$$\delta_a + \delta_b + \gamma_1(|a+b| + \delta_a + \delta_b).$$

⟨*EFloat Public Methods*⟩ ?≡ ???

```
EFloat operator+(EFloat f) const {
    EFloat r;
    r.v = v + f.v;
#ifndef NDEBUG
    r.ld = ld + f.ld;
#endif  // DEBUG
    r.err = err + f.err +
        MachineEpsilon * (std::abs(v + f.v) + err + f.err);
    return r;
}
```

The implementations for the other arithmetic operations for `EFloat` are analogous.

The `float` value in a `EFloat` is available via a type conversion operator; it has an `explicit` qualifier to require the caller to have an explicit `(float)` cast to extract the floating-point value. The requirement to use an explicit cast reduces the risk of an unintended round-trip from `EFloat` to `Float` and back, thus losing the accumulated error bounds.

⟨*EFloat Public Methods*⟩ ?≡ ???

```
explicit operator float() const { return v; }
```

If a series of computations is performed using `EFloat` rather than `float`-typed variables, then at any point in the computation, the `GetAbsoluteError()` method can be called to find a bound on the absolute error of the computed value.

⟨*EFloat Public Methods*⟩ ?≡ ???

```
float GetAbsoluteError() const { return err; }
```

The bounds of the error interval are available via the `UpperBound()` and `LowerBound()` methods. Their implementations use `NextFloatUp()` and `NextFloatDown()` to ensure that the returned values are rounded up and down respectively, ensuring that the interval is conservative.

⟨*EFloat Public Methods*⟩ ?≡ ???

```
float UpperBound() const { return NextFloatUp(v + err); }
float LowerBound() const { return NextFloatDown(v - err); }
```

In debug builds, method are available to get both the relative error as well as the precise value maintained in `ld`.

⟨*EFloat Public Methods*⟩ ?≡ ???

```
#ifndef NDEBUG
float GetRelativeError() const { return std::abs((ld - v)/ld); }
long double PreciseValue() const { return ld; }
#endif
```

`pbrt` also provides a variant of the `Quadratic()` function that operates on coefficients that may have error and returns error bounds with the `t0` and `t1` values. The implementation is the same as the regular `Quadratic()` function, just using `EFloat`.

⟨*EFloat Inline Functions*⟩ ?≡ ???

```
inline bool Quadratic(EFloat A, EFloat B, EFloat C,
                      EFloat *t0, EFloat *t1);
```

3.9.2 CONSERVATIVE RAY–BOUNDS INTERSECTIONS

Floating-point round-off error can cause the ray–bounding box intersection test to miss cases where a ray actually does intersect the box. While it's acceptable to have occasional false positives from ray–box intersection tests, we'd like to never miss an actual intersection—getting this right is important for the correctness of the `BVHAccel` in Section 4.3 so that ray–shape intersections aren't missed.

The ray–bounding box test introduced in Section 3.1.2 is based on computing a series of ray–slab intersections to find the parametric t_{min} along the ray where the ray enters the bounding box and the t_{max} where it exits. If $t_{\text{min}} < t_{\text{max}}$, the ray passes through the box; otherwise it misses it. With floating-point arithmetic, there may be error in the computed t values—if the computed t_{min} value is greater than t_{max} purely due to round-off error, the intersection test will incorrectly return a false result.

XXX check notation vs sec 3.1.2 in the below.

Recall that the computation to find the t value for a ray intersection with a plane perpendicular to the x axis at a point x is $t = (x - \mathrm{o}_x)/\mathbf{d}_x$. Expressed as a floating-point computation and applying Equation (3.6), we have

$$\begin{aligned} t &= (x \ominus \mathrm{o}_x) \oslash \mathbf{d}_x \\ &\subset \frac{(x - \mathrm{o}_x)}{\mathbf{d}_x}(1 \pm \epsilon)^2, \end{aligned}$$

and so

$$t(1 \pm \gamma_2) = \frac{(x - \mathrm{o}_x)}{\mathbf{d}_x}.$$

The difference between the computed result t and the precise result is bounded by $\gamma_2|t|$.

If we consider the intervals around the computed t values that bound the fully-precise value of t, then the case we're concerned with is when the intervals overlap; if they don't then the comparison of computed values will give the correct result (Figure XXX). If the intervals do overlap, it's impossible to know the actual ordering of the t values. In this case, increasing t_{max} by twice the error bound, $2\gamma_3 t_{\text{max}}$, before performing the comparison ensures that we conservatively return true in this case.

Caption: If the error bounds of the computed t_{min} and t_{max} values overlap, the comparison $t_{\text{min}} < t_{\text{max}}$ may not actually indicate if a ray hit a bounding box. It's better to conservatively return true in this case than to miss an actual intersection. Extending t_{max} by twice its error bound ensures that the comparison is conservative.

We can now define the fragment for the ray–bounding box test in Section 3.1.2 that makes this adjustment.

⟨*Update* `tFar` *to ensure robust ray–bounds intersection*⟩ ?≡ ???

```
tFar *= 1.f + 2 * gamma(2);
```

The fragments for the `Bounds3::IntersectP()` method that also takes one over the ray direction, ⟨*Update* `tMax` *and* `tyMax` *to ensure robust bounds intersection*⟩ and ⟨*Update* `tzMax` *to ensure robust bounds intersection*⟩ are similar, though they have a factor of γ_3 rather than γ_2 since they multiply by the reciprocal of the ray direction rather than dividing by it; an analysis similar to the one above shows that this approach introduces one more γ_1 error factor in the result.

3.9.3 ROBUST TRIANGLE INTERSECTIONS

The details of the ray–triangle intersection algorithm in Section 3.6.2 were carefully designed so that no valid intersection would ever be missed due to floating-point roundoff error. Recall that the algorithm is based on transforming triangle vertices into a coordinate system with the ray's origin at its origin and with the ray direction aligned along the $+z$ axis. Although roundoff error may be introduced by transforming the vertex positions to this coordinate system, this error doesn't affect the watertightness of the intersection test. (Further, this error is quite small, so it doesn't significantly impact the accuracy of the computed intersection points.)

Given vertices in this coordinate system, the three edge functions defined in Equation (3.1) are evaluated at the point $(0, 0)$; the corresponding expressions, Equation (3.2), are quite straightforward.

The key to the robustness of the algorithm is that with floating-point arithmetic, the edge function evaluations are guaranteed to have the correct sign. In general, we have

$$(a \otimes b) \ominus (c \otimes d). \tag{3.11}$$

First, note that if $ab = cd$, then Equation (3.11) evaluates to exactly zero, even in floating point. We therefore just need to show that if $ab > cd$, then $(a \otimes b) \ominus (c \otimes d)$ is never negative. If $ab > cd$, then $(a \otimes b)$ must be greater than or equal to $(c \otimes d)$. In turn, their difference must be greater than or equal to zero. (These properties all follow from the fact that floating-point arithmetic operations are all rounded to the nearest representable floating-point value.)

If the value of the edge function is zero, then it's impossible to tell whether it is exactly zero or whether a small positive negative value has rounded to zero. In this case, the fragment ⟨*Fall back to double precision test at triangle edges*⟩ reevaluates the edge function with double precision; it can be shown that doubling the precision suffices to distinguish these cases.

3.9.4 BOUNDING INTERSECTION POINT ERROR

We'll now apply this machinery for analyzing rounding error to derive conservative bounds on the absolute error in computed ray-shape intersection points,

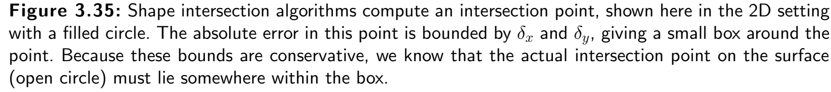

Figure 3.35: Shape intersection algorithms compute an intersection point, shown here in the 2D setting with a filled circle. The absolute error in this point is bounded by δ_x and δ_y, giving a small box around the point. Because these bounds are conservative, we know that the actual intersection point on the surface (open circle) must lie somewhere within the box.

which allows us to construct bounding boxes that are guaranteed to include an intersection point on the actual surface (Figure 3.35). These bounding boxes provide the basis of the algorithm for generating spawned ray origins that will be introduced in Section 3.9.5.

It's useful to start by looking at the sources of error in conventional approaches to computing intersection points. It is common practice in ray tracing to compute 3D intersection points by first solving the parametric ray equation $\mathrm{o} + t\mathbf{d}$ for a value t_{hit} where a ray intersects a surface and then computing the hit point p with $\mathrm{p} = \mathrm{o} + t_{\mathrm{hit}}\mathbf{d}$. If t_{hit} carries some error δ_t, then we can bound the error in the computed intersection point. Considering the x coordinate, for example, we have

$$\begin{aligned} x &= \mathrm{o}_x \oplus (t_{\mathrm{hit}} + \delta_t) \otimes \mathbf{d}_x \\ &\subset \mathrm{o}_x \oplus (t_{\mathrm{hit}} + \delta_t)\mathbf{d}_x(1 \pm \gamma_1) \\ &\subset \mathrm{o}_x(1 \pm \gamma_1) + (t_{\mathrm{hit}} + \delta_t)\mathbf{d}_x(1 \pm \gamma_2) \\ &= \mathrm{o}_x + t_{\mathrm{hit}}\mathbf{d}_x + [\pm \mathrm{o}_x\gamma_1 + \delta_t\mathbf{d}_x + t_{\mathrm{hit}}\mathbf{d}_x\gamma_2 + \delta_t\mathbf{d}_x\gamma_2]. \end{aligned}$$

The error term (in square brackets), is bounded by

$$\gamma_1|\mathrm{o}_x| + \delta_t(1 \pm \delta_2)|\mathbf{d}_x| + \gamma_2|t_{\mathrm{hit}}\mathbf{d}_x|. \tag{3.12}$$

There are two things to see from Equation (3.12): first, the magnitudes of the terms that contribute to the error in the computed intersection point (o_x, $\mathbf{d}_x$, and $t\mathbf{d}_x$) may be quite different than the magnitude of the intersection point. Thus, there is a danger of catastrophic cancellation in computing the intersection point's value. Second, ray intersection algorithms generally perform tens of floating-point operations to compute t values, which in turn means that we can expect δ_t to be at least of magnitude $\gamma_n t$, with n in the tens. (And possibly much more, due to catastrophic cancellation.) Each of these may be much larger than the computed point x.

Together, these factors can lead to relatively large error in the computed intersection point. Therefore, we'll introduce better approaches shortly.

Reprojection: Quadrics

We'd like to reliably compute intersection points on surfaces with just a few ulps of error rather than the hundreds of ulps of error that intersection points computed with the parametric ray equation may have. Previously, Woo et al. suggested using the first intersection point computed as a starting point for a second ray–plane intersection, for ray–polygon intersections (Woo et al. 1986).

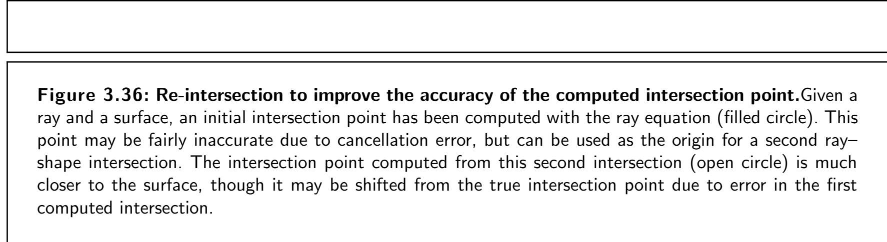

Figure 3.36: Re-intersection to improve the accuracy of the computed intersection point. Given a ray and a surface, an initial intersection point has been computed with the ray equation (filled circle). This point may be fairly inaccurate due to cancellation error, but can be used as the origin for a second ray–shape intersection. The intersection point computed from this second intersection (open circle) is much closer to the surface, though it may be shifted from the true intersection point due to error in the first computed intersection.

From the bounds in Equation (3.12), we can see why the second intersection point will be much closer to the surface than the first: the t_{hit} value along the second ray will be quite close to zero, so that the magnitude of the absolute error in t_{hit} will be quite small and thus using this value in the parametric ray equation will give a point quite close to the surface (Figure 3.36). Further, the ray origin will have similar magnitude to the intersection point, so the $\gamma_1|\mathrm{o}_x|$ term won't introduce much additional relative error.

Although the second intersection point with this approach is much closer to the plane of the surface, it still suffers from error by being offset due to error in the first computed intersection. The farther away the ray origin from the intersection point (and thus, the larger the absolute error in t_{hit}), the larger this error will be. In spite of this error, the approach has merit: we're generally better off with a computed intersection point that is quite close to the actual surface, even if offset from the most accurate possible intersection point than we are with a point that is some distance above or below the surface (and likely also far from the most accurate intersection point).

Rather than doing a full re-intersection computation, which may not only be computationally costly but also will still have error in the computed t value, an effective approach is to refine computed intersection points by reprojecting them to the surface. The error bounds in these reprojected points are often remarkably small.

Consider a ray–sphere intersection: given a computed intersection point (for example, from the ray equation) p with a sphere at the origin with radius r, we can reproject the point onto the surface of the sphere by scaling it with the ratio of the sphere's radius to the computed point's distance to the origin, computing a new point $\mathrm{p}' = (x', y', z')$ with

$$x' = x\frac{r}{\sqrt{x^2+y^2+z^2}},$$

and so forth. The floating-point computation is

$$\begin{aligned}
x' &= x \otimes r \oslash \mathtt{sqrt}(\mathtt{x} \otimes \mathtt{x} \oplus \mathtt{y} \otimes \mathtt{y} \oplus \mathtt{z} \otimes \mathtt{z}) \\
&\subset \frac{xr(1\pm\epsilon_{\mathrm{m}})^2}{\sqrt{x^2(1\pm\epsilon_{\mathrm{m}})^3 + y^2(1\pm\epsilon_{\mathrm{m}})^3 + z^2(1\pm\epsilon_{\mathrm{m}})^2}(1\pm\epsilon_{\mathrm{m}})} \\
&\subset \frac{xr(1\pm\gamma_2)}{\sqrt{x^2(1\pm\gamma_3) + y^2(1\pm\gamma_3) + z^2(1\pm\gamma_2)}(1\pm\gamma_1)}
\end{aligned}$$

Because x^2, y^2, and z^2 are all positive, the terms in the square root can share the same γ term, and we have

$$\begin{aligned} x' &\subset \frac{xr(1 \pm \gamma_2)}{\sqrt{(x^2+y^2+z^2)(1\pm\gamma_4)}(1\pm\gamma_1)} \\ &= \frac{xr(1 \pm \gamma_2)}{\sqrt{(x^2+y^2+z^2)}\sqrt{(1\pm\gamma_4)}(1\pm\gamma_1)} \\ &\subset \frac{xr}{\sqrt{(x^2+y^2+z^2)}}(1\pm\gamma_5) \\ &= x'(1\pm\gamma_5). \end{aligned} \tag{3.13}$$

Thus, the absolute error of the reprojected x coordinate is bounded by $\gamma_5|x'|$ (and similarly for y' and z'), and is no more than 2.5 ulps from a point on the surface of the sphere.

Here is the fragment that reprojects the intersection point for the `Sphere` shape.

⟨*Refine sphere intersection point*⟩ ?≡ ???

```
pHit *= radius / Distance(pHit, Point3f(0, 0, 0));
```

The error bounds follow from Equation (3.13).

⟨*Compute error bounds for sphere intersection*⟩ ?≡ ???

```
Vector3f pError = gamma(5) * Abs((Vector3f)pHit);
```

Reprojection algorithms and error bounds for other quadrics can be defined similarly: for example, for a cylinder along the z axis, only the x and y coordinates need to be reprojected, and the error bounds in x and y turn out to be only γ_3 times their magnitudes.

⟨*Refine cylinder intersection point*⟩ ?≡ ???

```
Float hitRad = std::sqrt(pHit.x * pHit.x + pHit.y * pHit.y);
pHit.x *= radius / hitRad;
pHit.y *= radius / hitRad;
```

⟨*Compute error bounds for cylinder intersection*⟩ ?≡ ???

```
Vector3f pError = gamma(3) * Abs(Vector3f(pHit.x, pHit.y, 0.f));
```

The disk primitive is particularly easy; we just need to set the z coordinate of the point to lie on the plane of the disk.

⟨*Refine disk intersection point*⟩ ?≡ ???

```
pHit.z = height;
```

In turn, we have a point with zero error; it lies exactly on the surface on the disk.

⟨*Compute error bounds for disk intersection*⟩ ?≡ ???

```
Vector3f pError(0., 0., 0.);
```

Parametric Evaluation: Triangles

XXX fixme numbering from zero... e_0, etc

Another effective approach to computing precise intersection points is to use the parametric representation of a shape to compute accurate intersection points. For example, the triangle intersection algorithm in Section 3.6.2 computes three edge function values e_1, e_2 and e_3 and reports an intersection if all three have the same sign. Their values can be used to find barycentric coordinates

$$b_i = \frac{e_i}{e_1 + e_2 + e_3}.$$

Attributes v_i at the triangle vertices (including the vertex positions) can be interpolated across the face of the triangle by

$$v' = b_1 v_1 + b_2 v_2 + b_3 v_3.$$

We can show that interpolating the positions of the vertices in this manner gives a point very close to the surface of the triangle. First consider precomputing the inverse sum of e_i:

$$\begin{aligned} d &= 1 \oslash (e_1 \oplus e_2 \oplus e_2) \\ &\subset \frac{1}{(e_1 + e_2)(1 \pm \epsilon)^2 + e_3(1 \pm \epsilon)}(1 \pm \epsilon). \end{aligned}$$

Because all e_i have the same sign, we can collect the e_i terms and conservatively bound d:

$$\begin{aligned} d &\subset \frac{1}{(e_1 + e_2 + e_3)(1 \pm \epsilon)^2}(1 \pm \epsilon) \\ &\subset \frac{1}{e_1 + e_2 + e_3}(1 \pm \gamma_3). \end{aligned}$$

Now, considering interpolation of the x coordinate of the position in the triangle corresponding to the edge function values, we have

$$\begin{aligned} x' &= ((e_1 \otimes x_1) \oplus (e_2 \otimes x_2) \oplus (e_3 \otimes x_3)) \otimes d \\ &\subset (e_1 x_1 (1 \pm \epsilon)^3 + e_2 x_2 (1 \pm \epsilon)^3 + e_3 x_3 (1 \pm \epsilon)^2) d (1 \pm \epsilon) \\ &\subset (e_1 x_1 (1 \pm \gamma_4) + e_2 x_2 (1 \pm \gamma_4) + e_3 x_3 (1 \pm \gamma_3)) d. \end{aligned}$$

Using the bounds on d,

$$\begin{aligned} x &\subset \frac{e_1 x_1 (1 \pm \gamma_7) + e_2 x_2 (1 \pm \gamma_7) + e_3 x_3 (1 \pm \gamma_6)}{e_1 + e_2 + e_3} \\ &= b_1 x_1 (1 \pm \gamma_7) + b_2 x_2 (1 \pm \gamma_7) + b_3 x_3 (1 \pm \gamma_6). \end{aligned}$$

Thus, we can finally see that the absolute error in the computed x' value is in the interval

$$\pm b_1 x_1 \gamma_7 \pm b_2 x_2 \gamma_7 \pm b_3 x_3 \gamma_7,$$

which is bounded by

$$\gamma_7(|b_1 x_1| + |b_2 x_2| + |b_3 x_3|). \tag{3.14}$$

(Note that the $b_3 x_3$ term could have a γ_6 factor instead of γ_7, but the difference between the two is very small, we choose a slightly simpler final expression.) Equivalent bounds hold for y' and z'.

TODO: reconcile: code indexes barycentrics from 0, equations here from 1.

Equation (3.14) lets us bound the error in the interpolated point computed in `Triangle::Intersect()`.

⟨*Compute error bounds for triangle intersection*⟩ ?≡ ???

```
Float xAbsSum = std::abs(b0 * p0.x) + std::abs(b1 * p1.x) +
    std::abs(b2 * p2.x);
Float yAbsSum = std::abs(b0 * p0.y) + std::abs(b1 * p1.y) +
    std::abs(b2 * p2.y);
Float zAbsSum = std::abs(b0 * p0.z) + std::abs(b1 * p1.z) +
    std::abs(b2 * p2.z);
Vector3f pError = gamma(7) * Vector3f(xAbsSum, yAbsSum, zAbsSum);
```

Other Shapes

For shapes where we may not want to derive reprojection methods and tight error bounds, running error analysis can be quite useful: we implement all of the intersection calculations using `EFloat` instead of `Float`, compute a t_{hit} value, and use the parametric ray equation to compute a hit point. We can then find conservative bounds on the error in the computed intersection point via the `EFloat` `GetAbsoluteError()` method.

TODO: is this the source of e.g. cone noise? Test with long double–are the error bounds legit??

⟨*Compute error bounds for intersection computed with ray equation*⟩ ?≡ ???

```
EFloat px = ox + tShapeHit * dx;
EFloat py = oy + tShapeHit * dy;
EFloat pz = oz + tShapeHit * dz;
Vector3f pError = Vector3f(px.GetAbsoluteError(), py.GetAbsoluteError(),
                           pz.GetAbsoluteError());
```

This approach is used for cones, paraboloids, and hyperboloids in `pbrt`.

⟨*Compute error bounds for cone intersection*⟩ ?≡ ???
 ⟨*Compute error bounds for intersection computed with ray equation*⟩

XXX text about curves

⟨*Compute error bounds for curve intersection*⟩ ?≡ ???

```
Vector3f pError(2 * hitWidth, 2 * hitWidth, 2 * hitWidth);
```

Effect of Transformations

The last detail to attend to in order to bound the error in computed intersection points is the effect of transformations, which introduce additional rounding error when they are applied to computed intersection points.

The quadric `Shapes` in `pbrt` transform world-space rays into object-space before performing ray–shape intersections and then transform computed intersection points back to world space. Both of these transformation steps introduce rounding error that needs to be accounted for in order to maintain robust bounds around intersection points.

If possible, it's best to try to avoid coordinate-system transformations of rays and intersection points. For example, it's better to transform triangle vertices to world space and intersect world-space rays with them than to transform rays to object space and then transform intersection points to world space.[18] Transformations are still useful, for example for the quadrics object instancing, so we'll show how to bound the error that they introduce.

We'll start by considering the error introduced by transforming a point (x, y, z) that is exact—i.e., without any accumulated error. Given a 4×4 non-projective transformation matrix with elements denoted by $m_{i,j}$, the transformed point x' is

$$\begin{aligned} x' &= ((m_{0,0} \otimes x) \oplus (m_{0,1} \otimes y)) \oplus ((m_{0,2} \otimes z) \oplus m_{0,3}) \\ &\subset m_{0,0}x(1 \pm \epsilon_{\mathrm{m}})^3 + m_{0,1}y(1 \pm \epsilon_{\mathrm{m}})^3 + m_{0,2}z(1 \pm \epsilon_{\mathrm{m}})^3 + m_{0,3}(1 \pm \epsilon_{\mathrm{m}})^2 \\ &\subset (m_{0,0}x + m_{0,1}y + m_{0,2}z + m_{0,3}) + \gamma_3(\pm m_{0,0}x \pm m_{0,1}y \pm m_{0,2}z \pm m_{0,3}) \\ &\subset (m_{0,0}x + m_{0,1}y + m_{0,2}z + m_{0,3}) \pm \gamma_3(|m_{0,0}x| + |m_{0,1}y| + |m_{0,2}z| + |m_{0,3}|). \end{aligned}$$

Thus, the absolute error in the result is bounded by

$$\gamma_3(|m_{0,0}x| + |m_{0,1}y| + |m_{0,2}z| + |m_{0,3}|). \tag{3.15}$$

Similar bounds follow for the transformed y' and z' coordinates.

We'll use this result to add a method to the `Transform` class that also returns the absolute error in the transformed point due to applying the transformation.

⟨*Transform Inline Functions*⟩ ?≡ ???

```
template <typename T> inline Point3<T>
Transform::operator()(const Point3<T> &p, Vector3<T> *pError) const {
    T x = p.x, y = p.y, z = p.z;
    ⟨Compute transformed coordinates from point pt⟩
    ⟨Compute absolute error for transformed point⟩
    if (wp == 1.) return Point3<T>(xp, yp, zp);
    else          return Point3<T>(xp, yp, zp)/wp;

}
```

The fragment ⟨*Compute transformed coordinates from point* `pt`⟩ isn't included here; it implements the same matrix/point multiplication as in Section 2.8.

Note that this code is buggy if the matrix is projective and the homogeneous w coordinate of the projected point is not one; Exercise 3.XXX at the end of the chapter has you fix this nit.

---

18 Although rounding-error is introduced when transforming triangle vertices to world space (for example), this error doesn't add error that needs to be handled in computing intersection points. In other words, the transformed vertices may represent a perturbed representation of the scene, but they are the most precise representation available given the transformation.

⟨*Compute absolute error for transformed point*⟩ ?≡ ???

```
T xAbsSum = (std::abs(m.m[0][0] * x) + std::abs(m.m[0][1] * y) +
             std::abs(m.m[0][2] * z) + std::abs(m.m[0][3]));
T yAbsSum = (std::abs(m.m[1][0] * x) + std::abs(m.m[1][1] * y) +
             std::abs(m.m[1][2] * z) + std::abs(m.m[1][3]));
T zAbsSum = (std::abs(m.m[2][0] * x) + std::abs(m.m[2][1] * y) +
             std::abs(m.m[2][2] * z) + std::abs(m.m[2][3]));
*pError = gamma(3) * Vector3<T>(xAbsSum, yAbsSum, zAbsSum);
```

The result in Equation (3.15) assumes that the point being transformed is exact. If the point itself has error bounded by δ_x, δ_y, and δ_z, then the transformed x coordinate is given by:

$$x' = (m_{0,0} \otimes (x \pm \delta_x) \oplus m_{0,1} \otimes (y \pm \delta_y)) \oplus (m_{0,2} \otimes (z \pm \delta_z) \oplus m_{0,3}).$$

Applying the definition of floating-point addition and multiplication's error bounds, we have:

$$\begin{aligned} x' = m_{0,0}(x \pm \delta_x)(1 \pm \epsilon)^3 + m_{0,1}(y \pm \delta_y)(1 \pm \epsilon)^3 + \\ m_{0,2}(z \pm \delta_z)(1 \pm \epsilon)^3 + m_{0,3}(1 \pm \epsilon)^2. \end{aligned}$$

Transforming to use γ, we can find the absolute error term to be bounded by

$$\begin{aligned} (\gamma_3 + 1)(|m_{0,0}|\delta_x + |m_{0,1}|\delta_y + |m_{0,2}|\delta_z) + \\ \gamma_3(|m_{0,0}x| + |m_{0,1}y| + |m_{0,2}z| + |m_{0,3}|). \end{aligned} \tag{3.16}$$

The `Transform` class also provides an `operator()` that takes a point and its own absolute error and returns the absolute error in the result, applying Equation (3.16). The definition is straightforward, so isn't be included in the text here.

⟨*Transform Public Methods*⟩ ?≡ ???

```
template <typename T> inline Point3<T>
operator()(const Point3<T> &p, const Vector3<T> &pError,
           Vector3<T> *pTransError) const;
```

The `Transform()` class also provides methods to transform vectors and rays, returning the resulting error. The vector error bound derivations (and thence, implementations) are very similar to those for points, and so also aren't included here.

⟨*Transform Public Methods*⟩ ?≡ ???

```
template <typename T> inline Vector3<T>
operator()(const Vector3<T> &v, Vector3<T> *vTransError) const;
template <typename T> inline Vector3<T>
operator()(const Vector3<T> &v, const Vector3<T> &vError,
           Vector3<T> *vTransError) const;
```

YES Exercise: how much error does our bound claim is introduced by multiplying by an identity matrix? How much is actually introduced? How might you modify `pbrt` so that the actual error bound is used instead? Is this likely to be worthwhile?

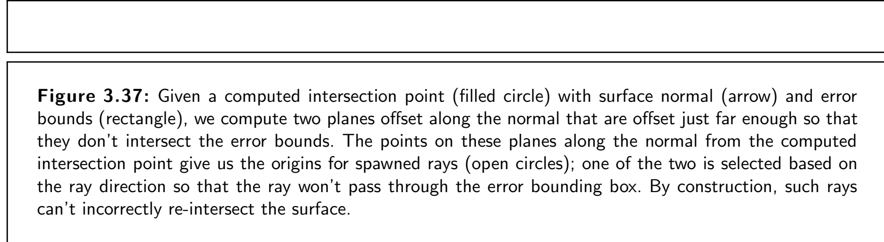

Figure 3.37: Given a computed intersection point (filled circle) with surface normal (arrow) and error bounds (rectangle), we compute two planes offset along the normal that are offset just far enough so that they don't intersect the error bounds. The points on these planes along the normal from the computed intersection point give us the origins for spawned rays (open circles); one of the two is selected based on the ray direction so that the ray won't pass through the error bounding box. By construction, such rays can't incorrectly re-intersect the surface.

This method is use to transform the intersection point and its error bounds in the `Transform::operator()` method for `SurfaceInteraction`s.

⟨*Transform* `p` *and* `pError` *in* `SurfaceInteraction`⟩ ?≡ ???

```
ret.p = (*this)(is.p, is.pError, &ret.pError);
```

3.9.5 ROBUST SPAWNED RAY ORIGINS

Having error bounds around a computed intersection point makes it possible to position the origins of rays leaving the surface so that they are always on the right side of the surface so that they don't incorrectly reintersect it. When tracing spawned rays leaving the intersection point p, we offset their origins enough to ensure that they are on the boundary of the error cube and thus won't incorrectly re-intersect the surface.

Computed intersection points and their error bounds give us a small 3D box that bounds a region of space. We know that the precise intersection point must be somewhere inside this box and that thus the surface must pass through the box (at least enough to present the point where the intersection is). (Recall Figure 3.35.)

In order to ensure that the spawned ray origin is definitely on the right side of the surface, we move far enough along the normal so that the plane perpendicular to the normal is outside the error bounding box. For a computed intersection point at the origin, the plane equation for the plane going through the intersection point is just

$$f(x, y, z) = \mathbf{n}_x x + \mathbf{n}_y y + \mathbf{n}_z z,$$

where the plane is implicitly defined by $f(x, y, z) = 0$ and the surface normal is $(\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$.

For a point not on the plane, the value of the plane equation $f(x, y, z)$ gives the offset along the normal that gives a plane that goes through the point. We'd like to find the maximum value of $f(x, y, z)$ for the eight corners of the error bounding box; if we offset the plane plus and minus this offset, we have two planes that don't intersect the error box that should be (locally) on opposite sides of the surface, at least at the computed intersection point offset along the normal (Figure 3.37.)

If the eight corners of the error bounding box are given by $(\pm\delta_x, \pm\delta_y, \pm\delta_z)$, then the maximum value of $f(x, y, z)$ is easily computed:

$$d = |\mathbf{n}_x|\delta_x + |\mathbf{n}_y|\delta_y + |\mathbf{n}_z|\delta_z.$$

Computing spawned ray origins by offsetting along the surface normal has a few advantages: assuming that the surface is locally planar (a reasonable assumption, especially at the very small scale of the intersection point error bounds), moving along the normal allows us to get from one side of the surface to the other while moving the shortest distance. In general, minimizing the distance that ray origins are offset is desirable for maintaining shadow and reflection detail.

⟨*Geometry Inline Functions*⟩ ?≡ ???

```
inline Point3f OffsetRayOrigin(const Point3f &p, const Vector3f &pError,
                               const Normal3f &n, const Vector3f &w) {
    Normal3f nAbs = Abs(n);
    Float d = Dot(nAbs, pError);
    Vector3f offset = d * Vector3f(n);
    if (Dot(w, n) < 0)
        offset = -offset;
    Point3f po = p + offset;
    ⟨Round offset point po away from p⟩
    return po;
}
```

We also must handle round-off error when computing the offset point: when `offset` is added to `p`, the result will in general need to be rounded to the nearest floating-point value. In turn, it may be rounded down towards `p` such that the resulting point is in the interior of the error box rather than in its boundary (Figure XXXX). Therefore, the offset point is rounded away from `p` here to ensure that it's not inside the box.[19]

Caption: The rounded value of the offset point `p+offset` computed in `OffsetRayOrigin()` may end up in the interior of the error box rather than on its boundary, which in turn introduces the risk of incorrect self intersections if the rounded point is on the wrong side of the surface. Advancing each coordinate of the computed point away from `p` ensures that it is outside of the error box.

Alternatively, the floating-point rounding mode could have been set to round towards plus or minus infinity (based on the sign of the value). Changing the rounding mode is generally fairly expensive, so we just shift by one floating-point value here. This will sometimes cause a value already outside of the error box to go slightly further outside it, but because the floating-point spacing is so small, this isn't a problem in practice.

19 The observant reader may now wonder about the effect of rounding error when computing the error bounds that are passed into this function. Indeed, these bounds should also be computed with rounding toward positive infinity. We ignore that issue under the expectation that the additional offset of one ulp here will more than cover that error.

⟨*Round offset point* `po` *away from* `p`⟩ ?≡ ???

```
for (int i = 0; i < 3; ++i) {
    if (offset[i] > 0)      po[i] = NextFloatUp(po[i]);
    else if (offset[i] < 0) po[i] = NextFloatDown(po[i]);
}
```

Given the `OffsetRayOrigin()` function, we can now implement the `Interaction` methods that generate rays leaving intersection points.

⟨*Interaction Public Methods*⟩ ?≡ ???

```
Ray SpawnRay(const Vector3f &d, int depth = 0) const {
    Point3f o = OffsetRayOrigin(p, pError, n, d);
    return Ray(o, d, Infinity, time, depth, GetMedium(d));
}
```

TODO: explain the issue when the receiving point is sampled from a surface...

⟨*Interaction Public Methods*⟩ ?≡ ???

```
Ray SpawnRayTo(const Point3f &p2, int depth = 0) const {
    Point3f origin = OffsetRayOrigin(p, pError, n, p2 - p);
    Vector3f d = p2 - origin;
    return Ray(origin, d, 1 - ShadowEpsilon, time, depth, GetMedium(d));
}
```

⟨*Global Constants*⟩ ?≡ ???

```
const Float ShadowEpsilon = 0.0001f;
```

The other variant of `SpawnRayTo()`, which takes a `Interaction` is analogous.

Finally, XXX....

⟨*Offset ray origin to edge of error bounds and compute* `tMax`⟩ ?≡ ???

```
Float lengthSquared = d.LengthSquared();
Float tMax = r.tMax;
if (lengthSquared > 0) {
    Float dt = Dot(Abs(d), oError) / lengthSquared;
    o += d * dt;
//    tMax -= dt;
}
```

3.9.6 AVOIDING INTERSECTIONS BEHIND RAY ORIGINS

Bounding the error in computed intersection points allows us to compute ray origins that are guaranteed to be on the right side of the surface so that they ray doesn't incorrectly intersect the surface it's leaving. However, a second source of rounding error most also be addressed for robust intersections: the error in parametric t values computed for ray–shape intersections. Rounding error can lead to an intersection algorithm computing a value $t > 0$ for the intersection point even though the t value for the actual intersection is negative (and thus should be ignored).

It's possible to show that some intersection test algorithms always return a t value with the correct sign; this is the best case, as no further computation is needed to bound the actual error in the computed t value. For example, consider the ray–axis-aligned slab computation: $t = (x \ominus \mathrm{o}_x) \oslash \mathbf{d}_x$. IEEE guarantees that if $a > b$, then $a \ominus b \geq 0$ (and if $b < a$, then $a \ominus b \leq 0$). To see why this is so, note that if $a > b$, then the real number $a - b$ must be greater than zero. When rounded to a floating-point number, the result must be either zero or a positive float; there's no a way a negative float could be the closest floating point number. Second, floating-point division returns the correct sign; these together guarantee that the sign of the computed t value is correct. (Or, that $t = 0$, but this case is fine, since our test for an intersection is $t > 0$).)

For shape intersection routines that use `EFloat`, the computed t value in the end has an error bound associated with it and no further computation is necessary. See the definition of the fragment ⟨*Check quadric shape* `t0` *and* `t1` *for nearest intersection*⟩ in Section 3.2.2.

Triangles

XXX barycentrics edge indexed from 0 now...

`EFloat` introduces computational overhead that we'd prefer to avoid for more commonly used shapes where efficient intersection code is more important. For these shapes, we can derive efficient-to-evaluate conservative bounds on the error in computed t values. We can apply our floating-point error analysis tools to the computation of t values for more complex intersection routines. The ray–triangle intersection algorithm in Section 3.6.2 computes a final t value by computing three edge function values e_i and using them to compute a barycentric-weighted sum of transformed vertex z coordinates, z_i:

$$t = \frac{e_1 z_1 + e_2 z_2 + e_3 z_3}{e_1 + e_2 + e_3} \tag{3.17}$$

⟨*Ensure that computed triangle t is conservatively greater than zero*⟩ ?≡ ???

```
⟨Compute δz term for triangle t error bounds⟩
⟨Compute δx and δy terms for triangle t error bounds⟩
⟨Compute δe term for triangle t error bounds⟩
⟨Compute δt term for triangle t error bounds⟩
if (t <= deltaT)
    return false;
```

Given a ray r with origin o, direction $\mathbf{d}$ and a triangle vertex p, the projected z coordinate is

$$z = (1 \oslash \mathbf{d}_z) \otimes (\mathrm{p}_z \ominus \mathrm{o}_z)$$

Applying the usual approach, we can find that the maximum error in z_i for each of three vertices of the triangle $\mathrm{p_i}$ is bounded by $\gamma_3 |z_i|$, and we can thus find a conservative upper bound for the error in *any* of the z positions by taking the maximum of these errors:

$$\delta_z = \gamma_3 \max_i |z_i|.$$

⟨*Compute δ_z term for triangle t error bounds*⟩ ?≡ ???

```
Float maxZt = MaxComponent(Abs(Vector3f(p0t.z, p1t.z, p2t.z)));
Float deltaZ = gamma(3) * maxZt;
```

The edge function values are computed as the difference of two products of transformed x and y vertex positions:

$$\begin{aligned} e_0 &= (x_3 \otimes y_2) \ominus (y_3 \otimes x_2) \\ e_1 &= (x_1 \otimes y_3) \ominus (y_1 \otimes x_3) \\ e_2 &= (x_2 \otimes y_1) \ominus (y_2 \otimes x_1) \end{aligned}$$

Bounds for the error in the transformed positions x_i and y_i are

$$\begin{aligned} \delta_x &= \gamma_5(|\mathrm{p}_x - \mathrm{o}_x| + |\mathrm{p}_z - \mathrm{o}_x|) \\ \delta_y &= \gamma_5(|\mathrm{p}_y - \mathrm{o}_y| + |\mathrm{p}_z - \mathrm{o}_z|). \end{aligned}$$

XXX we're again taking the maximums right? Do code and equation match?

⟨*Compute δ_x and δ_y terms for triangle t error bounds*⟩ ?≡ ???

```
Float maxX = MaxComponent(Abs(Vector3f(p0t.x, p1t.x, p2t.x)));
Float maxY = MaxComponent(Abs(Vector3f(p0t.y, p1t.y, p2t.y)));
Float maxZ = MaxComponent(Abs(Vector3f(p0t.z, p1t.z, p2t.z)));
Float deltaX = gamma(5) * (maxX + maxZ);
Float deltaY = gamma(5) * (maxY + maxZ);
```

Taking the maximum error over all three of the vertices, the $x_i \otimes y_j$ products in the edge functions are bounded by

$$(\max_i |x_i| + \delta_x)(\max_i |y_i| + \delta_y)(1 \pm \epsilon),$$

which have a bound on absolute error of

$$\delta_{xy} = \gamma_2 \max_i |x_i| \max_i |y_i| + \delta_y \max_i |x_i| + \delta_x \max_i |y_i| + \cdots.$$

Dropping the (negligible) higher-order terms of products of γ and δ terms, the error bound on the difference of two x and y terms for the edge function is

$$\delta_e = 2(\gamma_2 \max_i |x_i| \max_i |y_i| + \delta_y \max_i |x_i| + \delta_x \max_i |y_i|).$$

XXX double check that, in particular for the error from rounding after the subtraction of terms XXX.

⟨*Compute δ_e term for triangle t error bounds*⟩ ?≡ ???

```
Float maxXt = MaxComponent(Abs(Vector3f(p0t.x, p1t.x, p2t.x)));
Float maxYt = MaxComponent(Abs(Vector3f(p0t.y, p1t.y, p2t.y)));
Float deltaE = 2 * (gamma(2) * maxXt * maxYt + deltaY * maxXt +
                    deltaX * maxYt);
```

Again bounding error by taking the maximum of e_i terms, the error bound for the computed value of the numerator of t in Equation (3.17) is

$$\delta_t = 3(\gamma_3 \max_i |e_i| \max_i |z_i| + \delta_e \max_i |z_i| + \delta_z \max_i |e_i|).$$

A computed t value (before normalization by the sum of e_i) must be greater than this value for it to be accepted as a valid intersection.

⟨*Compute* δ_t *term for triangle* t *error bounds*⟩ ?≡ ???

```
Float maxE = MaxComponent(Abs(Vector3f(e0, e1, e2)));
Float deltaT = 3.f * (gamma(3) * maxE * maxZt + deltaE * maxZt +
                      deltaZ * maxE) * std::abs(invDet);
```

Although it may seem that we have made a number of choices to compute looser bounds than we could, in the interests of efficiency, in practice the bounds on error in t are extremely small. For a regular scene that fills a bounding box roughly ± 10 in each dimension, our t error bounds near ray origins are generally around 10^{-7}.

3.9.7 DISCUSSION

Minimizing and bounding numeric error in other geometric computations (e.g. partial derivatives of surface positions, interpolated texture coordinates, etc.) is much less important than it is for the positions of ray intersections. In a similar vein, the computations involving color and light in physically-based rendering generally don't present trouble with respect to round-off error; they generally involve sums of products of positive numbers (of generally the same magnitude), so there aren't generally problems from catastrophic cancellation. Furthermore, these sums are generally of few enough terms that accumulated error is small; the variance that is inherent in the Monte Carlo algorithms used for them generally dwarfs any floating-point error in computing them.

Interestingly enough, we saw an increase of roughly 20% in overall ray-tracing execution time after replacing `pbrt`'s old ad-hoc method to avoid incorrect self-intersections with the method described in this section. (In comparison, rendering with double-precision floating point causes an increase in rendering time of roughly 30%.) Profiling shows that very little of the additional time is due to the additional computation to find error bounds; this is not surprising, as the incremental computation our method requires is limited—most of the error bounds are just scaled sums of absolute values of terms that have already been computed.

The majority of this slowdown is actually due to an increase in ray–object intersection tests. The reason for this increase in intersection tests was first identified by XXXX Wächter, p. 30; when ray origins are very close to shape surfaces, more nodes of intersection acceleration hierarchies must be visited when tracing spawned rays than if overly loose offsets are used and more intersection tests are performed near the ray origin. Thus, while the reduction in performance is unfortunate, it is actually a direct result of the greater accuracy of the method; it is the price to be paid for more accurate resolution of valid nearby intersections.

FURTHER READING

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

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

Kajiya (1982) developed the first algorithm for computing intersections with parametric patches. Recent work on more efficient techniques for direct ray intersection with patches includes papers by Stürzlinger (1998), Martin et al. (2000), and Roth et al. (2001). Benthin et al. (2004) presented more recent results and include additional references to previous work. Ramsey et al. (2004) describe an efficient algorithm for computing intersections with bilinear patches, and Ogaki and Tokuyoshi (2011) introduce a technique for directly intersecting smooth surfaces generated from triangle meshes with per-vertex normals.

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

The ray–triangle intersection test in Section 3.6 was developed by Woop et al. (2013). See Möller and Trumbore (1997) for another widely-used ray–triangle intersection algorithm. A ray–quadrilateral intersection routine was developed by Lagae and Dutré (2005). Shevtsov et al. (2007a) described a highly optimized ray–triangle intersection routine for modern CPU architectures and included a number of references to other recent approaches. An interesting approach for developing a fast ray–triangle intersection routine was introduced by Kensler and Shirley (2006): they implemented a program that performed a search across the space of mathematically equivalent ray–triangle tests, automatically generating software implementations of variations and then benchmarking them. In the end, they found an more efficient ray–triangle routine than had been in use previously.

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

The layout of triangle meshes in memory can have a measurable impact on performance in many situations. In general, if triangles that are close together in 3D space are close together in memory, cache hit rates will be higher, and overall system performance will benefit. See Yoon et al. (2005) and Yoon and Lindstrom (2006) for algorithms for creating cache-friendly mesh layouts in memory.

The curve primitive intersection algorithm in Section 3.7 is based on the approach developed by Nakamaru and Ohno (2002). Earlier methods for computing ray intersections with generalized cylinders are also applicable to rendering curves, though they are much less efficient (???; ???). The book by Farin (2001) provides an excellent general introduction to splines, and the blossoming approach to them was introduced by Ramshaw (1987).

One challenge with rendering thin geometry like hair and fur is that thin geometry may require many pixel samples to be accurately resolved, which in turn increases rendering time. van Swaaij (2006) described a system that precomputed voxel grids to represent hair and fur, storing aggregate information about multiple hairs in a small region of space for more efficient rendering. More recently, Qin et al. (2014) described an approach based on cone tracing for rendering fur, where narrow cones are traced instead of rays. In turn, all of the curves that intersect a cone can be considered in computing the cone's contribution, allowing rendering with a small number of cones per pixel.

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

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

Higham's (2002) book on floating-point computation is excellent; it also developes the γ_n notation that we have used in Section 3.9. Other good references to this topic are Wilkinson (1994) and Goldberg (1991).

The incorrect self-intersection problem has been a known problem for ray-tracing practitioners for quite some time ???. In addition to offsetting rays by an "epsilon" at their origin, approaches that have been suggested include ignoring intersections with the object that was first intersected, "root polishing" ???, where the computed intersection point is refined to become more numerically accurate; and using higher-precision floating point representations (e.g. `double` instead of `float`).

Kalra and Barr (1989) and Dammertz and Keller (2006) developed algorithms for numerically-robust intersections based on recursively subdividing object bounding boxes, discarding boxes that don't encompass the object's surface, and discarding boxes missed by the ray. Both of these approaches are much less efficient than traditional ray–object intersection algorithms as well as the techniques introduced in Section 3.9.

Salesin et al. (1989) introduced techniques to derive robust primitive operations for computational geometry that accounted for floating-point round-off error, and Ize showed how to perform numerically-robust ray–bounding box intersections (Ize 2013); his approach is implemented in Section 3.9.2. Wächter (2008) discussed self-intersection issues in his thesis; he suggested recomputing the intersection point starting from the initial intersection (root polishing) and offsetting spawned rays along the normal by a fixed small fraction of the intersection point's magnitude. The approach implemented in this chapter uses his approach of offsetting along the normal, but is based on conservative bounds on the offsets based on the numeric error present in computed intersection points. (As it turns out, our bounds are generally tighter than Wächter's while also being provably conservative.)

EXERCISES

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

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

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

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

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