Appendix C

More on Real-World Computer Number Representation

C.1 Introduction

In Chapter 1, we discussed representation of real numbers on a computer, using the IEEE floating point format. In this chapter we'll continue by reviewing some of the issues surrounding the representation of whole numbers and integers on a computer. This review mainly serves to introduce concepts that will carry over to a discussion of an alternate representation for real numbers: fixed point numbers . The remainder of the chapter will cover this representation, along with its bitwise formats, basic operations, features, and limitations. As before, we will transition from general mathematical discussions of number representation toward implementation-related topics of specific relevance to 3D graphics programmers.

C.2 Representing Integral Types on a Computer

C.2.1 Finiteness of Representation

The sets of whole numbers ($[0, 1, 2 \ldots]$, known as $\mathbb{W}$), integers ($[\ldots -2, -1, 0, 1, 2 \ldots]$, known as $\mathbb{Z}$), and real numbers (e.g., 1.5, 1/3, $\sqrt{2}$, known as $\mathbb{R}$) share one trait in common — they each have

infinitely many elements. Computers, on the other hand, by their very physical nature can only represent a finite number of different values. As a result computers cannot represent *any* of the aforementioned number sets exactly and completely. We will have to settle for some finite subset of each. The sizes of these finite sets are determined by the number of distinct values that can be represented by the given number of storage systems.

Modern computers store their numbers as binary codings: finite, fixed-length strings of bits. (For a basic discussion of binary number representation, we refer the reader to a basic computer architecture text, such as Stallings's *Computer Organization and Architecture*.) As such, any N-bit computer number representation can only represent 2^N distinct values. For our purposes, we will generally assume 32-bit words, which can represent about 4 billion different values. Each of the distinct values in a given representation can represent at most one element of the number set exactly. While 4 billion may seem an enormous number of possible values, we shall see that it becomes painfully finite when used to cover the set of real numbers.

C.2.2 Range

The range of a number representation system is described by two values: the minimum representable value and the maximum representable value, often written as the interval [*minimum, maximum*]. Values outside of this interval are assumed to be unrepresentable (although, as shown in the text, there are some cases in floating point where values outside of the proper range of a representation can still be represented indirectly). We shall review the common computer representations of integers and whole numbers here, mainly to discuss the properties of these numbers and to give examples of range.

The whole numbers ($\mathbb{W}$) have an inherent, finite minimum: 0. In order to represent the whole numbers with a finite computer representation, we use the fact that for any finite whole number W_{max}, there will be a finite number of elements (possibly zero) $w \in \mathbb{W}$ such that $w \leq W_{max}$. In fact, the size of such a set is $(W_{max} + 1)$. Based on this observation, we can represent a useful K-element subset of the whole numbers by simply selecting a maximum representable value of $W_{max} = K - 1$. All whole numbers less than K can be represented exactly by such a system.

The type `unsigned int` is the most commonly used C/C++ representation of $\mathbb{W}$. For smaller numbers, `unsigned short` and `unsigned char` are also used. For larger numbers, there is `unsigned long long`. We will discuss only `unsigned int` in this section; the analysis of the other representations is analogous. On a most modern computers, the representation of `unsigned int` is simply an unsigned 32-bit binary number, capable of representing 2^{32} distinct values, more specifically represented as `uint32_t`. As a result, all whole numbers in the range $[0, 2^{32} - 1]$ can be represented by `unsigned int`. For a basic discussion of the binary representation of unsigned numbers, again see Stallings or the equivalent. Note that the C++ specification does not require `unsigned int` to be a 32-bit type; however, for the course of this discussion, we will assume the common case, which is a 32-bit `unsigned int`.

The integers have no inherent minimum or maximum. In order to represent the integers on a computer, we must select a finite *pair* of values Z_{min} and Z_{max}. There will be a finite number of elements (possibly zero) $i \in \mathbb{Z}$, such that $Z_{min} \leq i \leq Z_{max}$. The size of such a set is $max(0, Z_{max} - Z_{min} + 1)$. Unlike the whole numbers, there are infinitely many K-element sets, one for each chosen minimum value ($[Z_{min}, Z_{min} + K - 1]$). Historically, most computer representations of integers select Z_{min} such that the K-element set is as evenly distributed around 0 as possible:

$$Z_{min} \approx -Z_{max}, or Z_{min} \approx -\frac{K}{2}$$

This is done to ensure that for as many elements as possible:

$$i \in [Z_{min}, Z_{max}] \implies -i \in [Z_{min}, Z_{max}]$$

The type `int` is the most common C/C++ representation of $\mathbb{Z}$ (as before, we will not discuss the similar representations `short`, `char` and `long long`). On a modern computer the representation of `int` is generally a signed 32-bit binary representation (`int32_t`) using so-called '2's complement' to represent both positive and negative numbers, where the negative numbers are represented as unsigned values beyond Z_{max} (to get the 2's complement version of a negative integer, you represent the corresponding positive integer in binary, flip the bits, and add 1). Being a 32-bit

representation, it is capable of representing 2^{32} distinct values. As this is an even number of elements, there is no way to represent 2^{32} distinct integers *and* fulfill the requirement to have the range of the representation exactly center about 0. The standard 2's complement format of `int` represents all integers in the range $[-(2^{31}), 2^{31} - 1]$, meaning that while $-(2^{31})$ can be represented, its negative $-(-(2^{31})) = 2^{31}$ is out of range, since $2^{31} > 2^{31} - 1$.

Overflow

Overflow is a term used to describe what occurs when a computation generates a result with a value outside the range of the representation in use. As we shall see, different representation systems have different ways of dealing with this situation, but in all cases the result cannot be represented exactly, and in most cases the represented result is *very* different from the correct result. In general, the best method with any number system is to avoid overflow entirely. Such a strategy requires that the programmer understand the exact range of the representation(s) that they are using. The following sections will discuss the range of the number representations, along with the likely results of overflow. In many cases, standardization has set the overflow behavior of a given representation across platforms.

As mentioned, the range of the common type `uint32_t` is $[0, 2^{32} - 1]$. Application code must take care to ensure that the results of all operations involving `unsigned int` values are in range. Positive overflow of 32-bit unsigned integer values can be controlled, depending on usage. For example, if a counter is incremented once per frame on a game that is running at 100 frames per second as a 32-bit unsigned integer, this counter will overflow only after

$$\frac{2^{32} frames}{100\frac{frames}{sec} \times 60\frac{secs}{min} \times 60\frac{mins}{hour} \times 24\frac{hours}{day} \times 365\frac{days}{yr}} \approx \texttt{1.4 years!}$$

However, if we were to update this counter per actor, with 1000 actors in a game, it would overflow in about half a day. In this case, a 64-bit counter would probably be wise.

Negative overflow of `unsigned int` values is rather easy to generate, especially in code with simple logic errors. Commonly, such code will subtract a larger number from a smaller one, leading

to a negative result (which cannot be represented as a whole number). The C/C++ standard requires that 32-bit unsigned operations always return a result that is equal to the correct value *modulo* 2^{32}.

For most common applications, this result is not particularly useful as it leads to the following examples:

$$(2^{32} - 1) + 1 \rightarrow 0$$
$$0 - 1 \rightarrow (2^{32} - 1)$$

However, it does make sense when considered as a part of the larger picture. The result of any unsigned integer operation is the least significant 32 bits of the correct result. Using assembly language (where the overflow flag, or "carry bit," is accessible), it is possible to chain together 32-bit addition operations to add 64-bit (or larger) numbers. The carry bit "carries" into the low-order bit of the next 32-bit operation.

For most applications, the best way to handle negative overflow of unsigned integers is to avoid the situation by ensuring that the result of the subtraction will be nonnegative prior to computation and reworking code that can generate negative overflows.

Range and Type Conversion

Mathematically, whole numbers are a proper subset of the set of integers. However, on a computer our representations of integers (`int`) and whole numbers (`unsigned int`) have the same size (generally 32 bits), so the set of `int`s and `unsigned int`s each have the same number of elements. This leads to the (sometimes problematic) fact that on a computer, `unsigned int` $\not\subseteq$ `int` and `int` $\not\subseteq$ `unsigned int`. Each set contains values that cannot be represented by the other set. Programmers must be very careful when converting between `int` and `unsigned int` to avoid problems.

Given that the range of a 32-bit `int` is $[-(2^{31}), 2^{31} - 1]$ and the range of a 32-bit `unsigned int` is $[0, 2^{32} - 1]$, the safe range for conversion is thus

$$[-(2^{31}), 2^{31} - 1] \cap [0, 2^{32} - 1] = [0, 2^{31} - 1]$$

Applications should check `int` values to make sure they are not negative and `unsigned int` values to ensure that they will not overflow $2^{31} - 1$ prior to converting (casting) them. Most C/C++ compilers will generate a warning (at some warning levels) unless a signed/unsigned cast is made explicit.

C.3 Fixed Point

C.3.1 Introduction

Much is made of the performance of hardware floating point units (FPUs) in modern desktop processors and full-sized game consoles. The basic use of floating point numbers is familiar to even the novice programmer. However, floating point is not the only way that real numbers are approximated on computers. In fact, for decades another representation, fixed point, was far more popular owing to its high performance and accuracy when used correctly, even on low-powered computers.

Over the past decade, fixed point numbers have become somewhat of a lost art to all but the most hardcore, experienced 3D programmers. For example, 3D PC games written in the late 1990s and beyond tended to use floating point heavily (if not exclusively). However, the popularity of powerful handheld computers and cellular telephones brought fixed-point arithmetic back to the forefront of 3D game development. While recently these platforms also have introduced powerful floating point processors, being aware of fixed point can still provide certain options for optimization.

C.3.2 Basic Representation

Fixed point numbers are a method of representing a subset of the real numbers on a computer. Fixed point numbers are based upon the computer representation of integers. In fact, as we shall see, integers can be thought of as a special case of fixed point. Like the computer representations of integers upon which they are built, fixed point numbers are finite. As such, they cannot represent the entire set of real numbers. However, the range and precision limitations of fixed point numbers are very simple, making them easy to describe and analyze.

Fixed point numbers are based on a very simple observation with respect to computer representation of integers. In the standard binary representation, each bit represents twice the value of the bit to its right, with the least significant bit representing 1. The following diagram shows these powers of two for a standard 8-bit unsigned value:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Just as a decimal number can have a decimal point, which represents the break between integral and fractional values, a binary value can have a binary point, or more generally a radix point (a decimal number is referred to as radix 10, a binary number as radix 2). In the previous number layout, we can imagine the radix point being to the right of the last digit. However, it does not have to be placed there. For example, let us revisit the previous case, this time placing the radix point in the middle of the number (between the fourth and fifth bits). The diagram would then look like this:

2^3	2^2	2^1	2^0 . 2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1 . $\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Now, the least significant bit represents 1/16. The basic idea behind fixed point is one of scaling. A fixed point value is related to an integer with the same bit pattern by an implicit scaling factor. This scaling factor is fixed for a given fixed point format and is the value of the least significant bit in the representation. In the case of the preceding format, the scaling factor is 1/16.

The standard nomenclature for a fixed point format is "M-dot-N," where M is the number of integral bits (to the left of the radix point) and N is the number of fractional bits (to the right of the radix point). For example, the 8-bit format in our example would be referred to as "4-dot-4." As a further example, regular 32-bit integers would be referred to as "32-dot-0" because they have no fractional bits. More generally, the scaling factor for an M-dot-N format is simply 2^{-N}. Note that, as expected, the scaling factor for a 32-dot-0 format (integers) is $2^0 = 1$. No matter what the format, the radix point is "fixed" (or locked) at N bits from the least significant bit; thus the name "fixed point."

C.3.3 Range and Precision

Computing the range and precision for a given fixed point format is very easy and can be computed solely by knowing the "M-dot-N" format name. This simple analysis is made possible by the previously stated fact about the relationship between fixed point numbers and integers with the same bitwise representation. For any fixed point number viewed directly as an integer, we compute the fixed point number's value in M-dot-N format by multiplying the integer by a scaling factor equal to 2^{-N}.

To compute the range of an M-dot-N format, we recall the earlier discussion regarding 2's complement integers. We know that a 2's complement integer with B bits has range $[-(2^{B-1}), 2^{B-1}-1]$. Since the total number of bits in a fixed-point representation is $M + N$, this leads to a fixed-pointrange of

$$\begin{gathered}\left[-(2^{M+N-1}) \times 2^{-N}, (2^{M+N-1}-1) \times 2^{-N}\right] \\ \left[\frac{-(2^{M+N-1})}{2^N}, \frac{(2^{M+N-1}-1)}{2^N}\right] \\ \left[-(2^{M-1}), \left(2^{M-1} - \frac{1}{2^N}\right)\right]\end{gathered}$$

The precision of the representation can be computed just as simply. When dealing with integers, the spacing between each integer and its nearest neighbor is simply 1.0. Multiplying by the fixed point format's scaling factor, we find that the difference between any M-dot-N number and its closest neighbor is

$$1.0 \times 2^{-N} = \frac{1}{2^N}$$

Given this fixed distance between any dot-N fixed point value and its closest representable neighbor, we know that any real number A within the valid range for the preceding M-dot-N format is, at worst, different from its representation by half the distance between the values directly above and

below A. So, A can be represented with an absolute error of at most

$$AbsError_A = |Rep(A) - A| \leq \frac{1}{2^N} \times \frac{1}{2} = \frac{1}{2^{N+1}}$$

This absolute error bound is constant across the range of the format. On the other hand, the relative error bound is

$$RelError_A = \left| \frac{Rep(A) - A}{A} \right| \leq \frac{AbsError_A}{|A|} = \frac{1}{|A| \times 2^{N+1}}$$

which rises sharply as A tends toward zero. In other words, with a fixed-point system the relative error falls as magnitudes increase. This leads to the basic guideline that an application must determine how small its smallest values can become and set the fractional precision based on this quantum.

Converting between Real and Fixed-Point

Converting a real number R to an M-dot-N fixed-point number F can be accomplished via the following method:

$$F = round(R \times 2^N)$$

where *round* is the function used to round a real number to the nearest integer. Basically, this method scales the real number to the correct scaling value for the fixed point format, and then rounds away any precision beyond what can be represented in the given format. For example, to convert the value 4.5 to our 4-dot-4 format, we do the following:

$$\begin{aligned} F &= round(4.5 \times 2^4) \\ &= round(4.5 \times 16) \\ &= round(72) \\ &= 72 \end{aligned}$$

$72 =$ | 0 | 1 | 0 | 0 . 1 | 0 | 0 | 0 |

which represents 4.5 exactly. Note that in this case, the *round* operation had no effect. If the *round* operation had changed the value, then this would indicate that the M-dot-N formula could not represent the given value exactly. An example of such a case is 3.7 represented in 4-dot-4 format:

$$\begin{aligned} F &= round(3.7 \times 2^4) \\ &= round(59.2) \\ &= 59 \end{aligned}$$

$59 =$ | 0 | 0 | 1 | 1 . 1 | 0 | 1 | 1 |

which represents the rational value 3.6875. The absolute error of representation in this case is $3.7 - 3.6875 = 0.0125$. This is much less than the maximum possible absolute error in 4-dot-4 format, which is $1/2^5 = 0.03125$.

It is very important that the rounding step be done after the scaling, or else the real number will be rounded to an integer and *all* of the fractional precision will be lost (the N least-significant bits of the resulting fixed point value will be zeros). Note that real numbers outside of the range of the fixed point format will overflow during the conversion to the integer format and must be considered invalid.

Converting back to the desired real number is as simple as treating the fixed point number's integer representation as a real number (in C/C++, this is simply a typecast) and then scaling that real number by the $1/2^N$ scaling factor. In practice, this does assume that the integer value can be represented exactly as a floating point number. For example, the largest integer representable by single precision floating point is 2^{24}. Beyond that the lower order bits will be dropped because they won't fit in the mantissa. In this case, you'll have to use a double precision floating point number to avoid precision loss.

The conversion methods between integers and fixed point values are even simpler. This is due to

the fact that the scaling factors of the fixed point formats we have discussed are all powers of two. Multiplying an integer by 2^N is equivalent to shifting that integer to the left by N bits. Shifting is an operation that is supplied by the integer math units (arithmetic logic units, or ALUs) of all major CPUs and is extremely fast (free on some CPUs when done at the same time as another math operation). Dividing an integer by 2^N is equivalent to shifting the integer to the right by N bits.

We can use these fast special cases of multiplication and division in our integer/fixed point conversion. The conversion from integer to fixed point can never lose precision (although it will overflow if the integer is not in the range of the fixed point format) and is implemented by shifting the integer to the left by N bits. The conversion from fixed point to integer will never overflow but often loses precision and is implemented by shifting the integer to the right by N bits. Note that shifting to the right truncates the number. In 2's complement, this computes the floor of the number; it does not round it. In order to round during the conversion to integer, we must first add 2^{N-1} (which is equal to 1/2 in the M-dot-N format) and *then* shift the result to the right by N bits. The addition of 1/2 prior to the shift turns the truncation into a form of rounding.

C.3.4 Addition and Subtraction

Addition and subtraction of two fixed point numbers of the same format is extremely simple — they are merely the integer versions of addition and subtraction. This is possible because two M-dot-N fixed point numbers have radix points that line up (just like standard integers). A 4-dot-4 examplefollows:

	Bits							Integer	Real
	0	0	0	1 . 0	0	0	0	[16]	(1.0)
+	0	0	0	0 . 1	1	0	0	[12]	(0.75)
	0	0	0	1 . 1	1	0	0	[28]	(1.75)

C.3.5 Multiplication

The simplicity of addition and subtraction may lead one to hope that multiplication and division of fixed point numbers are equally simple. However, a quick example shows a problem with this method. For example, let us convert 0.5 and 0.25 into 4-dot-4 fixed point and multiply them together. We expect a result of 0.125. First, we convert the real numbers to 4-dot-4fixed point:

$0.5 \rightarrow 8 =$

0	0	0	0 . 1	0	0	0

$0.25 \rightarrow 4 =$

0	0	0	0 . 0	1	0	0

Next, we multiply them using the standard integer method:

								Bits	Integer	Real
	0	0	0	0 . 1	0	0	0		[8]	(0.5)
$\times$	0	0	0	0 . 0	1	0	0		[4]	(0.25)
Incorrect!	0	0	1	0 . 0	0	0	0		[32]	(2.0)

The result is clearly incorrect and just as clearly (given the magnitude of the error) not simple rounding error. However, there is a clear reason for this error. Fixed point numbers are not equivalent to their bitwise representation as integers, but rather as their integer representation times the $1/2^N$ scaling value. Thus, the integer bits represent the real number times 2^N. If we recompute the multiplication just done adding these scale values, we find the following:

$$(0.5 \times 2^4) \times (0.25 \times 2^4) = (0.125 \times 2^4) \times 2^4$$

The problem is that each of the two operands brings its own implicit scale value. Multiplying these together causes the result to be too large by exactly the scaling factor. Thus, to reestablish the correct fixed point format, we must divide the result by 2^4:

$$\frac{(0.5 \times 2^4) \times (0.25 \times 2^4)}{2^4} = (0.125 \times 2^4)$$

This method generalizes as follows: to multiply two M-dot-N fixed point numbers, we multiply

their representations using the integer multiplication method and then divide the result by 2^N. Using the same observation as we did for integer/fixed-point conversion, we replace the division by 2^N with an N-bit right shift (written as $\gg N$) of the result of the integer multiplication. This gives a method for multiplying two M-dot-N numbers $A_{M.N}$ and $B_{M.N}$ of

$$(A_{M.N} \times B_{M.N}) \gg N$$

Next, we show this multiplication method graphically, computing$1.0 \times 0.375 = 0.375$:

	Bits	Integer	Real
	0 0 0 1 . 0 0 0	[16]	(1.0)
×	0 0 0 0 . 0 1 1 0	[6]	(0.375)
	0 1 1 0 . 0 0 0 0	[96]	
≫	4 Bits		
	0 0 0 0 . 0 1 1 0	[6]	(0.375)

C.3.6 Division

Fixed point division suffers from another issue of scale correction but in the inverse manner to what happened with multiplication. Rather than ending up with two scale values multiplying together and requiring correction, in the case of division, the scale values cancel out, and the result is represented as an integer with no fractional precision. We'll demonstrate with the same numbers used in our multiplication example:

$0.5 \rightarrow 8 =$ 0 0 0 0 . 1 0 0 0

$0.25 \rightarrow 4 =$ 0 0 0 0 . 0 1 0 0

We attempt to divide one by the other using the standard integer method:

	Bits							Integer	Real
	0	0	0	0 . 1	0	0	0	[8]	(0.5)
/	0	0	0	0 . 0	1	0	0	[4]	(0.25)
Incorrect!	0	0	0	0 . 0	0	1	0	[2]	(0.125)

In this case, we can see that the scale values have canceled:

$$\frac{0.5 \times 2^4}{0.25 \times 2^4} = \frac{0.5}{0.25} = 2 = (0.125 \times 2^4)$$

To reestablish the correct fixed point format, we could multiply the result by 2^4. However, the problem with such a method is that the result would have no fractional precision (a 4-dot-4 number times 2^4 is always an integer)! The precision was lost in the division. To avoid this loss of precision, we must multiply the dividend by 2^4:

$$\frac{0.5 \times 2^4 \times 2^4}{0.25 \times 2^4} = \frac{0.5 \times 2^4}{0.25} = 2 \times 2^4 = (2.0 \times 2^4)$$

This division method generalizes as follows: to divide one M-dot-N fixed-point number by another, we multiply the dividend by 2^N and then divide their representations using the integer division method. Using the same observation as we did for integer/fixed-point conversion, we replace the multiplication by 2^N by shifting the dividend to the left (written $\ll$) by N bits. This gives a method for dividing two M-dot-N numbers $A_{M.N}$ and $B_{M.N}$ of

$$\frac{(A_{M.N} \ll N)}{(B_{M.N})}$$

We show this method graphically, computing $0.25/2.0 = 0.125$:

	Bits	Integer	Real
	0 0 0 . 0 1 0 0	[4]	(0.25)
≪	4 Bits		
	0 1 0 . 0 0 0 0	[64]	
/	0 0 1 . 0 0 0 0	[32]	(2.0)
	0 0 0 . 0 0 1 0	[2]	(0.125)

C.3.7 Real-World Fixed Point

The astute reader will note that the preceding examples of fixed point operations were rather contrived. In fact, it would have been very easy to generate examples of the algorithms discussed thus far that caused overflow and incorrect results. The very nature of fixed point numbers, the fact that even small real values are represented by large integers (e.g., the real value 1.0 being represented by 65,536 in 16-dot-16) can lead to overflow in surprisingly low-magnitude situations. Additionally, while fixed point has been presented as a fast alternative to floating point on integer-only platforms, it is apparent that there is likely to be some performance penalty for the additional shifting that is required in many fixed point operations.

In fact, several modern processors take both of these issues into account, offering features that can assist the programmer. In one case, such a feature makes the extra shifting operations computationally inexpensive (or even free). In another, a set of extra instructions makes overflow far less of a problem. Details of these situations, as well as both platform-dependent and platform-independent methods of dealing with them, are covered in the following section.

C.3.8 Intermediate Value Overflow and Underflow

The basic method for fixed point multiplication discussed thus far requires that the intermediate multiplied value be shifted downward in magnitude to reestablish the correct position of the radix point. A side effect of this method is that the intermediate value can overflow, even if the final result should be well within the range of the fixed point format. As an introductory example, we

demonstrate 1.0×1.0 in 4-dot-4 fixed point. Clearly, the result should be 1.0, obviously within range:

	Bits	Integer	Real
	0 0 0 1.0 0 0 0	[16]	(1.0)
×	0 0 0 1.0 0 0 0	[16]	(1.0)
Overflow!	0 0 0 0.0 0 0 0	[256 = 0]	(0.0)
≫	4 Bits		
Incorrect!	0 0 0 0.0 0 0 0	[0]	(0.0)

Even a simple multiplication can result in overflow in the intermediate value. Seeing this situation, one might be tempted to simply move the shift operation up in the process, shifting first and then multiplying the preshifted result. For example, in our 4-dot-4 case imagine a method in which each operand was preshifted by two bits prior to the multiplication. Multiplying $1.0 \times 1.0 = 1.0$:

	Bits	Integer	Real
	0 0 0 1.0 0 0 0	[16]	(1.0)
≫	2 Bits		
	0 0 0 0.0 1 0 0	[4]	(0.25)
×	0 0 0 1.0 0 0 0	[16]	(0.25)
≫	2 Bits		
	0 0 0 0.0 1 0 0	[4]	(0.25)
	0 0 0 1.0 0 0 0	[16]	(1.0)

Success! However, blindly preshifting operands can result in problems as well. This same method nets much less satisfying results with the following case of $7.0 \times 0.125 = 0.875$. Note what happens to the second operand when preshifted:

							Bits	Integer	Real
0	0	0	0 . 0	0	1	0		[2]	(0.125)
$\gg$							2 Bits		
0	0	0	0 . 0	0	0	0		[0]	(0.0)

With one of the operands being truncated to zero, the result of the multiplication will be zero! This is quite a sizable error. Clearly, no single shifting method can satisfactorily deal with all precision and overflow cases.

Extended Precision Hardware Assistance

The method that is perhaps the best at dealing with overflow and underflow requires some outside assistance on the part of the hardware platform. This assistance comes in the form of extended-precision mathematical operations. Such instructions are based on the fact that two N-bit numbers when multiplied together can require up to $2N$ bits to avoid overflow. Numerous modern processors (especially those without floating point units, such as older versions of the ARM architecture) include either 16-bit multiplication operations with 32-bit results or 32-bit multiplication operations with 64-bit results. Such operations are practically (if not specifically) tailor-made for fixed point implementations.

For example, imagine our 4-dot-4 (8-bit) fixed point system. As we demonstrated, if we multiply 1.0×1.0 using the direct method, the operation's intermediate value will overflow, leaving an incorrect result. However, with an 8-bit $\times$ 8-bit $\implies$ 16-bit instruction, even the larger-magnitude operation 2.0×2.0 can be completed using the direct method. Note that the 16-bit intermediate result is actually the correct answer as well, but in 8-dot-8 format:

							0	0	1	0 . 0	0	0	0	(2.0)
						×	0	0	1	0 . 0	0	0	0	(2.0)
0	0	0	0	0	1	0	0 . 0	0	0	0	0	0	0	0	

≫ 4 Bits

0	1	0	0 . 0	0	0	0	(4.0)

It is important to note that these extended-precision operations do not actually extend the values that can be represented in a given fixed point format. If the result cannot be represented in the final format, the only option would be to change the format used or else reformulate the problem. In the following example, the intermediate result can be represented in the 16-dot-16 intermediate format, but the final result is truncated incorrectly and ends up overflowing:

							0	1	1	0 . 0	0	0	0	(6.0)
						×	0	1	1	1 . 0	0	0	0	(7.0)
0	0	1	0	1	0	1	0 . 0	0	0	0	0	0	0	0	(42.0)

≫ 4 Bits

0	0	0	0	0	0	1	0 . 1	0	1	0	0	0	0	0	(42.0)
						Overflow – Incorrect!	1	0	1	0 . 0	0	0	0	(−6.0)	

In this case the extended precision was not the answer, as the final result still had to be converted to the shorter format. What extended precision does avoid is overflow in intermediate results. This, in turn, avoids the need to preshift precision from the operands, avoiding unnecessary underflow. With careful programming, these instructions can be used for strings of operations, with the conversion from the long format to the original format happening once at the end. A common instruction form that is given on extended-precision hardware is an extended-precision multiply-accumulate, as follows:

$$\text{32-bit} \times \text{32-bit} + \text{64-bit} \implies \text{64-bit}$$

An extended-precision multiply-accumulate instruction is useful for quickly computing a safe, precise 16-dot-16 fixed point dot product. We assume that we begin with the pair of 3-vectors (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2):

1. $X_1 \times X_2 \implies \mathit{Accumulator}_{64}$
2. $Y_1 \times Y_2 + \mathit{Accumulator}_{64} \implies \mathit{Accumulator}_{64}$
3. $Z_1 \times Z_2 + \mathit{Accumulator}_{64} \implies \mathit{Accumulator}_{64}$
4. $\mathit{Accumulator}_{64} \gg 16 \implies \mathit{Result}_{32}$

C.3.9 Limits of Fixed Point

To better understand the significant limitations of fixed point, even with hardware-assisted extended-precision, we shall consider one of the most popular fixed point formats in general fixed point libraries, the 32-bit signed 16-dot-16 format. Although applications can pick a wide range of formats, and can even use multiple formats in the same application, 16-dot-16 is representative. We can summarize the minimum and maximum representable values in this format, as well as the value of epsilon (ϵ), the distance between adjacent representable values as follows:

$$\begin{aligned} \text{Maximum representable value:} \quad & \mathit{Max}_{16.16} \approx 32767 \\ \text{Minimum representable value:} \quad & \mathit{Min}_{16.16} \approx -32768 \\ \text{Smallest positive value:} \quad & \mathit{Eps}_{16.16} \approx 1.5 \times 10^{-5} \end{aligned}$$

While these may seem like large amounts of range and precision (and indeed they are), it should be noted that $\sqrt{\mathit{Max}_{16.16}} \approx 181$, and $\sqrt{\mathit{Eps}_{16.16}} \approx 0.004$. In other words, if the application needs to store the value $\mathbf{a} \bullet \mathbf{b}$, then for safety $\|\mathbf{a}\| < 181$ and $\|\mathbf{b}\| < 181$ to avoid overflow. Similarly, to avoid underflow, $\|\mathbf{a}\| > 0.004$ and $\|\mathbf{b}\| > 0.004$.

These suddenly begin to seem like significantly tighter limitations. While issues such as these can be overcome by careful scaling of the data throughout the application, the simple fact is that fixed point requires that programmers keep a very close bound over the required range and precision values of their data throughout the entire application. If the application happens to be a complete and general 3D pipeline, this can be a rather daunting task.

C.3.10 Fixed Point Summary

In today's applications the decision of whether or not to use fixed point is more often than not based entirely on the application's target platform capabilities. Few modern software 3D pipelines choose to use fixed point unless their target platform has no floating point hardware. However, on platforms with fast integer ALUs and no FPUs, fixed point can make the difference between a high-performance, compelling 3D experience and a non-interactive, low frame rate "slide show."

C.4 Chapter Summary

In this chapter we have discussed how computers represent integer values, and real numbers in the fixed point format. We have covered basic operations and some error metrics for these operations. We've also noted that these representations have inherent limitations that any serious programmer must understand in order to use them efficiently and correctly. For further reading, back issues of *Game Developer* magazine provide frequent discussion of number representations as they relate to computer games.