The neglected art of Fixed Point arithmetic

Jetro Lauha

Seminar Presentation

Assembly 2006, 3rd - 6th August 2006
(Revised: September 13, 2006)

2019-08, small note:
Many of the statements in this presentation do not hold true for “today’s hardware”.
(floating point support is now common in mobile and even IoT CPUs).

Contents

- Motivation
- Introduction
- Typically needed functions
- Caveats and tricks
- Tips for making a fixed point library

Motivation

- *Man sent himself to moon, and space probes even beyond that*. Do you think the hardware used to accomplish those feats had fancy FPU to do all the calculations?
- They used RCA 1802.
  - Processing power equals roughly 6502 or 6510, used in Apple II and Commodore 64.

Motivation

- But it's a lot of work.
  - 30% of the Apollo software development effort was spent on scaling. [KrL64]
- So they eventually switched to floating point when hardware got better.

Motivation

- So why am I talking about this?
  - Well, at least it's COOL, in retro-way:
    This is how demo & game coders did their 3D stuff 15 years ago and made some pretty cool stuff even with the minuscule CPU power.
- But does that matter anymore – except if you are going to take part in the old school demo competition with some retro stuff?

Motivation

- There's still plenty of platforms where using only fixed point (integer) calculations is still very relevant.
  - Mobile devices (Typical: ARM CPU, no FPU)
    - Almost all mobile phones (J2ME or native code)
    - Handheld consoles (Gameboy, Nintendo DS)
  - DSP Programming
    - There's both fixed & floating point DSPs

Motivation

- ...continued...
  - OpenGL ES is the standard for embedded 3D.
    - Profiles for both fixed point and floating point, but often only Common-Lite profile is provided (no floating point).
  - Fixed point is often still a bit faster on desktop than floating point.
  - Stable calculations across platforms
    - Floating point calculations are prone to slight differences based on compiler, CPU and other dependencies.

Introduction

- Basics
- Notation
- Range and precision
- Conversion
- Basic operations: + - * /

Introduction: Basics

- What are the fixed point numbers in “layman's” terms?
  - Scale all real numbers by a constant factor, such as 65536, round to nearest integer and and store the numbers as integers.
  - This allows you to represent an evenly distributed subset of real numbers roughly from -32768 to 32767 (with 32-bit signed integers and factor of 65536).

Introduction:
Basics

- More exactly, you are dividing your range of values to two parts – the integer part and fractional part.

8-bit example:

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

2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1	.	1/2	1/4	1/8	1/16

That's the "*fixed point!*"

Introduction: Notation

- Notations:
  - $M.N$, e.g. 16.16
  - QN (Q factor), e.g. Q16
- M is number of integer bits and N is number of fractional bits.

Introduction: Range and precision

- Range: defined by the integer (upper) part.
  - 16.16 (signed): range is [-32768, 32767]
- Precision: smallest difference between two successive numbers is $1/2^N$.
  - 16.16: 1/65536 (~0.000015258789)

8-bit example:

2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1	.	1/2	1/4	1/8	1/16

4.4
Range: [-8, 7] (if signed)
Precision: 1/16 (0.0625)

Introduction: Conversion

- Conversion from real to fixed point number
  - Multiply by 2^N and round to nearest integer
    - `(int)(R * (1<<N) + (R>=0 ? 0.5 : -0.5))`
- Conversion from fixed point to real number
  - Cast to real and divide by 2^N
    - `(float)F / (1<<N)`
- Conversion from/to integers (lossless)
  - Shift N bits up or down (scaling by 2^N)
    - `F = I<<N ,  I = F>>N`

Introduction: Basic operations

- Addition (+) and subtraction (-)
  - Same as adding and subtracting integers
- Multiplication (a * b)
  - Multiply as integers and divide result by 2^N.
    - ~~`((a * b) >> N)`~~
  - That overflows very easily, as both a and b are fixed point numbers!
    - If both a and b are 2.0 (131072) as 16.16 fixed point `(a * b) == 17179869184` - 32 bits isn't enough!

Introduction: Basic operations

- For multiplication, the intermediate result from `(a * b)` is in *2M*:*2N* (Q2*N*) format
  - Store intermediate value in double sized integer format. That is, for 32-bit 16.16 fixed point numbers, you need a 64-bit integer to store the 32.32 (Q32) intermediate result.
    - ```
      (int)(((INT64)a *
             (INT64)b) >> N)
      ```

INT64

MSVC: `__int64`
GCC: `signed long long`
Java: `long`
      ```

Introduction: Basic operations

- Division (a / b)
  - Multiply `a` by 2^N and divide by `b` (as integers).
    - ~~`((a << N) / b)`~~
  - Again, intermediate result is prone to overflowing, so the correct way for 16.16 is:
    - `(((INT64)a << N) / b)`
- See references for more detailed introductory texts to fixed points. [VVB04, Str04, WikF]

Typically Needed Functions

- Sine and cosine: `sin(x), cos(x)`
- Arcus tangent: `atan2(y, x)`
- Square root: `sqrt(x)`
- Try CORDIC

Typically Needed Functions: Sine and cosine

- Typical approach is to use a look-up table.
  - Requires memory proportional to desired accuracy
  - Requires some storage space to load table from or time for pre-calculating table on startup
  - Can interpolate between sampled values to gain some more accuracy
- Note that it's enough to calculate $\pi/4$ entries to table, rest of the samples can be mirrored and transformed from those.

Typically Needed Functions: Sine and cosine

- It's possible to find or construct less accurate approximations for functions if you need smaller code, memory usage or more speed.
  - DSP coders have some quite nice tricks. [Ben06]
- See also [Str04] for code example of how to calculate sin, cos and tan algorithmically using only a small arctan table.

Typically Needed Functions: Square root

- Several fairly good iterative algorithms exist, so I don't recommend using a look-up table.
- Can be as simple as trying out to multiply integers by themselves until you find out the closest one
  - Or binary search version of the above
- Ken Turkowski's implementation is probably the most often used one. [Tur94]
  - For your convenience, code on the next slide.

Typically Needed Functions: Square root

```
/* The definitions below yield 2 integer bits, 30 fractional bits */
#define FRACBITS 30    /* Must be even! */
#define ITERS    (15 + (FRACBITS >> 1))
typedef long TFract;

TFract
FFracSqrt(TFract x)
{
    register unsigned long root, remHi, remLo, testDiv, count;

    root = 0;         /* Clear root */
    remHi = 0;        /* Clear high part of partial remainder */
    remLo = x;        /* Get argument into low part of partial remainder */
    count = ITERS;    /* Load loop counter */

    do {
        remHi = (remHi << 2) | (remLo >> 30); remLo <<= 2;  /* get 2 bits of arg */
        root <<= 1;   /* Get ready for the next bit in the root */
        testDiv = (root << 1) + 1;    /* Test radical */
        if (remHi >= testDiv) {
            remHi -= testDiv;
            root += 1;
        }
    } while (count-- != 0);

    return(root);
}
```

[Tur94]

Typically Needed Functions: Arcus tangent

- You can try some look-up table tricks, again.
- If fast and rough approximation is enough, implementation can be very simple. [Cap91]
- For accurate results, try using CORDIC (covered next).
- For my favorite approximation (for the time being), check Jim Shima's DSP Trick: Fixed-Point Atan2 With Self Normalization. [Shi99]

Typically Needed Functions: Try CORDIC

- “COordinate Rotation DIgital Computer”, an algorithm to calculate hyperbolic and trigonometric functions, from 1959. [WikC]
  - Only small look-up tables, bitshifts and additions.
- Use it run-time or to pre-calculate look-up tables. (sin, cos, atan, ...)
- Accurate results
- Not the fastest solution

Caveats And Tricks

- Back to range and precision
- Watch out for division by zero
- Exact results
- Dealing with problems

Caveats And Tricks: Back to range and precision

- When storing result of `a*b` to normal sized fixed point (integer) value
  - Possible range & precision for the original values is much more limited than the normal to prevent overflow & underflow.
  - For storing `a*a`:
    - abs(a)<=~181 -- 181*181 = 32761, barely fits in signed 16.16 fixed point number.
    - abs(a)>=~0.004 -- 0.004*0.004 = 0.000016, truncated down to 1/65536.

Caveats And Tricks: Back to range and precision

- Similarly, make sure that a/b will stay in range
  - When |b| > 1.0
    - Check ranges so that result doesn't end up being 0.
  - When |b| < 1.0
    - $b>1/(2^{M-1}/a)$
      - If max value for a is 32, b must be at least 0.000991821 (65/65536) so that a/b fits in 16.16 fixed point number: 32/0.000991821=~32263.
      - If b would be one less (64/65536), then a/b will be 32768, not fitting in [-32768, 32767] 16.16 fixed point value range.

Caveats And Tricks: Watch out for division by zero

- Floating points have “Infinity Arithmetic”
  - Even result of division by zero is defined, so you simply get Inf as a result
    - Easier to go unnoticed by mistake
- Fixed point (integer) division by zero leads to interrupt or an exception is thrown
  - Typically programs just crash at this

Caveats And Tricks: Exact results

- Possible in some cases: modify division involving formulas to keep numerator and denumerator separate, and try to find out final (exact) result by examining those, without doing the division. See [Eri05] for example.
- Generally speaking, it's rare and hard to take advantage of this.

Caveats And Tricks: Dealing with problems

- When troubled by overflows, underflows or accuracy problems
  - Try keeping the intermediate result(s) in the bigger (64 bit) format and work out the final result directly from there.
  - Use asserts and do other verification checks rigorously, especially in debug builds.
  - Compare to results of same calculations done in floating points.

Tips For Making A Fixed Point Library

- There's built-in support... if you code in Ada.
- C/C++ alternatives:
  - Code it all in-line, using normal integers
  - Use helper macros (conversions, operations)
  - Create a real number class with overloaded operators
    - Allows to switch easily between floats and fixed points

Tips For Making A Fixed Point Library

- Create debug version of the real number class
  - Perform both fixed point and floating point calculations in parallel
    - Detect overflow & underflow conditions
    - Detect drifting
    - Error/warning asserts and checks can be made run-time togglable
- If you work on J2ME, it's best to inline all calculations yourself for performance.

Other Tidbits

- Nobody noticed that I changed the underlying physics engine from floating point to fixed point in latest version of *Pogo Sticker*.
- You can do fixed point (integer) abs() without branches. [And05, War02]
  - For 32-bit ints:
    - `result = (v ^ (v >> 31)) - (v >> 31)`
  - Ridiculously that's patented. But that's not the only way, check the references.

Other Tidbits

- 32-bit signed 0x80000000 (highest bit) is special
  - `int x; if (x < 0) x = -x;`
    Doesn't work as expected if `x==0x80000000`!
    X will still be 0x80000000 (-2147483648).
  - For the above example, solution is to cast result to unsigned int as you know it will not be negative.

References

KrL64 http://www.hq.nasa.gov/office/pao/History/computers/Ch4-2.html – Kreide, H., Lambert, D.W., *Computation: Aerospace Computers in Aircraft, Missiles and Spacecraft*, Space/Aeronaut., 42, 78 (1964); see also N.H. Herman and U.S. Lingon, *Mariner 4 Timing and Sequencing*, Astronaut. Aeronaut., 43 (October 1965).

VVB04 Van Verth, J. M., Bishop, L. M., *Essential Mathematics for Games & Interactive Applications – A Programmer's Guide*, Morgan Kaufmann, 2004.

Str04 Street, M., *A Fixed Point Math Primer,* OpenGL® ES Game Development, Course Technology PTR, 2004.

WikF http://en.wikipedia.org/wiki/Fixed-point_arithmetic – Fixed-point arithmetic article in Wikipedia.

Tur94 Turkowski, K., *Fixed Point Square Root*, Apple Technical Report No. 96, 1994. Also appears in *Graphics Gems V*, Paeth, A. W. (editor), Academic Press, 1995. See http://www.graphicsgems.org/.

Cap91 Capelli, R., *Fast Approximation to the Arctangent*, Graphics Gems II, Academic Press, 1991. See http://www.graphicsgems.org/.

Shi99 http://www.dspguru.com/comp.dsp/tricks/alg/fxdatan2.htm – Shima, J., *DSP Trick: Fixed-Point Atan2 With Self Normalization*, post in comp.dsp newsgroup, Apr 23, 1999.

Ben06 http://www.audiomulch.com/~rossb/code/sinusoids/ – Bencina, R., *Fun with Sinusoids*, 2006.

WikC http://en.wikipedia.org/wiki/CORDIC – CORDIC article in Wikipedia, see especially the referenced "CORDIC Bibliography Site" and the C implementation by Peter Knoppers (http://people.csail.mit.edu/hqm/imode/fplib/cordic_code.html).

Eri05 Ericson, C., *Numerical Robustness for Geometric Calculations (aka EPSILON is NOT 0.00001!)*, GDC Proceedings, 2005. Also available from http://realtimecollisiondetection.net/pubs/.

And05 http://graphics.stanford.edu/~seander/bithacks.html – Anderson, S. E., *Bit Twiddling Hacks*.

War02 Warren, H. S., *Hacker's Delight*, Addison-Wesley, 2002.

Thank You!

URL for these slides:

https://iki.fi/jetro/2006/08/07/neglected-art-of-fixed-point-arithmetic/

Fill out this form if you’re interested in more information about Fixed Point Math:

https://docs.google.com/forms/d/e/1FAIpQLScZ56aEt7oJED-kDFFlaUHJZ6FLy3AZ520P9gHYMv8OAtIsVg/viewform

- Short URL: http://j.mp/morefixedpoint