Optimizing Fixed Point (FP) Math with J2ME

Although hard-core code optimization on modern fast desktop and server machines is an arcane and disappearing art, there's still a need for such optimization when targeting the less powerful processors used in the handheld and mobile devices, particularly for game programming.

by André de Leiradella

Once upon a time, game programmers had to write code in assembler and make many other optimizations to deliver games that ran well even with the slow processors and small memory address spaces available then. Today, processors running at gigahertz speed and hundreds of megabytes of RAM help programmers climb out of the Black Programming Art Tower—a place where only a few could survive.

In fact, the power unleashed by modern PCs is so huge, even compared to computers only ten years ago, that programmers began ignoring the optimizations and creating high-quality games that require increasing amounts of processor power, memory, and disk space. These less-optimized games have proven at least as addictive as the earlier games. In other words, the massive increase in computing power simplified game programming by removing the need for the very knowledge and optimizations that once made game programming an arcane art. For a while it seemed as if the Black Programming Art Tower was closed for good. Only a small number of game companies still push our desktop computers to the edge.

But history repeats itself. The handhelds and mobile devices gaining devotees today have characteristics very similar to the less-powerful desktop computers of the past. Small-footprint devices are back—but if you want great games-to-go, they bring with them the need to master obscure optimization techniques. One such optimization involves fixed-point math.

Fixed-point Math

FP math is a great optimization technique, particularly for math-intensive applications such as 3-D games; but instead of showing you how to perform arithmetic operations on FP numbers, a topic already covered in many other places, this article focuses on how to optimize FP operations to achieve a greater performance.

FP Libraries

One of the easiest ways to implement FP math in your program is to use one of the free FP libraries available. You substitute standard arithmetic operations with calls to the FP library. While this works well in most cases and lets you use fractions in your program, making large numbers of calls to methods in another class, the FP class, eats precious CPU cycles.

This is a direct consequence of the way the Java Virtual Machine works. For every method call, the JVM builds a new stack frame and copies method parameters to that newly created stack frame. Upon completion, it must copy the method's return value to the previous stack frame. Finally, it must destroy the current stack frame. That's a lot of work!

To reduce that workload, you have to remove the calls to the FP class methods in your code.

Eliminating FP Method Calls

You might think that one way to remove the overhead of calls to the FP class is to make a copy of its methods inside your class; but unfortunately, that doesn't work. The calls will still be there, only the methods would have moved from one class to another. Instead, you must insert the code that performs the arithmetic operation in place of the call, substituting inline code for the method calls themselves.

As an example, here's an expression that computes x*a+y*b+z*c using the Beartronics FP library available on SourceForge.

```
FP.Mul(x,a)+FP.Mul(y,b)+FP.Mul(z,c)
```

The above expression is common in 3-D game code to scale, rotate and translate points according to a transformation matrix. If you inline the FP arithmetic code in the expression to get rid of the FP calls, you'd end up with code like this (all FP numbers are considered to be 16.16 bits):

```
((((int)((((long)x) * ((long)a)) >> 16))+
    ((int)((((long)y) * ((long)b)) >> 16)))+
    ((int)((((long)z) * ((long)c)) >> 16)))
```

The multiplications in the preceding code cast the operands to longs (because a 16.16 number multiplied by another 16.16 number yields a 32.32 number that has 64 bits), perform the multiplication and shift the 32.32 result back to 16.16 (the 16 most and the 16 least significant bits are lost).

A Performance Comparison

Now some speed test results. The numbers below show the time spent, in milliseconds, to compute the example expression 1,000,000 times, with the FP library and with inlined FP code. The tests were run five times each and the results presented here are the mean. The time spent to run an empty loop 1,000,000 times is also presented:

```
FP library      135,536
Inline          16,728
Empty loop       2,516
```

> Author's Note: The programs were run with the J2ME Wireless Toolkit version 2.1. I measured the empty loop time to be able to compare the time spent just in the expression, removing the time spent on the loop increment, compare, and jump.

Discounting the time to run the empty loop, you can see that the inlined code runs in about one tenth of the time required by the code with FP library calls! From a pure performance perspective, the inlining was definitely worth the effort. The only problem is that translating such expressions to inlined FP arithmetic is a complicated process. Any error in that process can lead to bugs that are very difficult to catch.

Wouldn't it be nice if you could *automatically* convert an expression to use inlined FP arithmetic? Fortunately, now you can.

Converting Expressions Automatically

You can convert any FP expression to use inlined FP. To do that you need a lexical analyzer, a parser, and a code generator that spits out inlined FP arithmetic.

The code that accompanies this article contains a command-line expression parser called "FPComp." FPComp reads an expression supplied as an argument and outputs Java code with the FP operations inlined. It assumes that all numbers and identifiers are FP numbers in the 16.16 format and that all function calls take FP expressions as arguments and return an FP value, unless you use the -i option. Figure 1 shows two usage examples.

To use FPComp in interactive mode, you can simply copy the returned inlined code and paste it in place of your original expression.

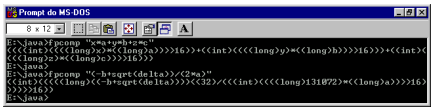

Figure 1. FPComp In Interactive Mode: When you pass FPComp an expression, it returns the expression written in inlined code, which you would then insert in place of the original expression in your code.

> Warning: Don't forget to save a copy of the original expression inside a comment; the expressions generated by FPComp can get very cryptic and you won't be able to easily recall what it is supposed to do.

Inlining FP Arithmetic Automatically

If you use Ant and Antenna to compile your J2ME projects, you may be interested in installing a patch to Antenna version 0.9.12 (released under the same license as Antenna itself, and included in the downloadable code for this article) that adds a new directive to the Antenna pre-processor: exec. The exec directive will execute any external program and replace itself with the output of the executed program, so you can use it to easily inline FP code in your J2ME application:

```
int compute(int x,y,z) {
  return
    //#exec fpcomp.bat "x*a+y*b+z*c"
  ;
}
```

When processed by the patched Antenna, the preceding code becomes:

```
int compute(int x,y,z) {
  return
    // Exec directive running external program:
    // fpcomp.bat "x*a+y*b+z*c"
    ((((int)((((long)x)*((long)a))>>16))+
    ((int)((((long)y)*((long)b))>>16)))+
    ((int)((((long)z)*((long)c))>>16)))
  ;
}
```

Note that it's not just method calls to FP libraries that can have a huge performance impact in your program; *all* methods that are called repeatedly have the same problem and you should avoid such repeated calls whenever possible.

In this article you've seen evidence that you can make FP expressions run about ten times faster by replacing the FP calls with inlined FP arithmetic. You've also seen FPComp, a small Java program that can take an arbitrary expression and transform it to a version with the FP operations inlined, ready to be inserted in your

Java source code. Finally, you've seen an automated way to inline FP calls using Ant and a patched version of Antenna able to call external programs such as FPComp.

Of course inlining of basic arithmetic methods doesn't completely free you from your FP library. Such libraries usually have other useful methods to compute square roots, trigonometric functions, and other code that's much harder to inline.

André de Leiradella *is from Rio de Janeiro. He started programming in 1981 using a ZX-81 compatible with a built-in BASIC interpreter, and became fascinated with the ability to "teach" the computer many different tasks. He's worked with Pascal, assembly language, C, Java, and other programming languages. André has a CS degree and is working on a Master's in Artificial Intelligence. He works for Xerox as a system analyst.*