Tips for Optimizing C/C++ Code

1. *Remember Ahmdal's Law:* $Speedup = \frac{time_{old}}{time_{new}} = \frac{1}{(1-func_{cost})+func_{cost}/func_{speedup}}$
   - Where $func_{cost}$ is the percentage of the program runtime used by the function $func$, and $func_{speedup}$ is the factor by which you speedup the function.
   - Thus, if you optimize the function $TriangleIntersect()$, which is 40% of the runtime, so that it runs twice as fast, your program will run 25% faster ($\frac{1}{(1-0.4)+0.4/2} = \frac{1}{0.8} = 1.25$).
   - This means infrequently used code (e.g., the scene loader) probably should be optimized little (if at all).
   - This is often phrased as: "make the common case fast and the rare case correct."
2. *Code for correctness first, then optimize!*
   - This does not mean write a fully functional ray tracer for 8 weeks, then optimize for 8 weeks!
   - Perform optimizations on your ray tracer in multiple steps.
   - Write for correctness, then if you know the function will be called frequently, perform obvious optimizations.
   - Then profile to find bottlenecks, and remove the bottlenecks (by optimization or by improving the algorithm). Often improving the algorithm drastically changes the bottleneck – perhaps to a function you might not expect. This is a good reason to perform obvious optimizations on all functions you know will be frequently used.
3. *People I know who write very efficient code say they spend at least twice as long optimizing code as they spend writing code.*
4. *Jumps/branches are expensive. Minimize their use whenever possible.*
   - Function calls require *two* jumps, in addition to stack memory manipulation.
   - Prefer iteration over recursion.
   - Use inline functions for short functions to eliminate function overhead.
   - Move loops inside function calls (e.g., change *for(i=0;i<100;i++) DoSomething();* into *DoSomething() { for(i=0;i<100;i++) { ... } }*).
   - Long if...else if...else if...else if... chains require *lots* of jumps for cases near the end of the chain (in addition to testing each condition). If possible, convert to a *switch* statement, which the compiler sometimes optimizes into a table lookup with a single jump. If a *switch* statement is not possible, put the most common clauses at the beginning of the if chain.
5. *Think about the order of array indices.*
   - Two and higher dimensional arrays are still stored in one dimensional memory. This means (for C/C++ arrays) *array[i][j]* and *array[i][j+1]* are adjacent to each other, whereas *array[i][j]* and *array[i+1][j]* may be arbitrarily far apart.
   - Accessing data in a more-or-less sequential fashion, as stored in physical memory, can *dramatically* speed up your code (sometimes by an order of magnitude, or more)!
   - When modern CPUs load data from main memory into processor cache, they fetch more than a single value. Instead they fetch a block of memory containing the requested data and adjacent data (a *cache line*). This means after *array[i][j]* is in the CPU cache, *array[i][j+1]* has a good chance of already being in cache, whereas *array[i+1][j]* is likely to still be in main memory.

6. *Think about instruction-level-parallelism.*
    - Even though many applications still rely on single threaded execution, modern CPUs already have a significant amount of parallelism inside a single core. This means a single CPU might be simultaneously executing 4 floating point multiplies, waiting for 4 memory requests, and performing a comparison for an upcoming branch.
    - To make the most of this parallelism, blocks of code (i.e., between jumps) need to have enough independent instructions to allow the CPU to be fully utilized.
    - Think about unrolling loops to improve this.
    - This is also a good reason to use inline functions.
7. *Avoid/reduce the number of local variables.*
    - Local variables are normally stored on the stack. However if there are few enough, they can instead be stored in registers. In this case, the function not only gets the benefit of the faster memory access of data stored in registers, but the function avoids the overhead of setting up a stack frame.
    - (Do not, however, switch wholesale to global variables!)
8. *Reduce the number of function parameters.*
    - For the same reason as reducing local variables – they are also stored on the stack.
9. *Pass structures by reference, not by value.*
    - I know of no case in a ray tracer where structures should be passed by value (even simple ones like Vectors, Points, and Colors).
10. *If you do not need a return value from a function, do not define one.*
11. *Try to avoid casting where possible.*
    - Integer and floating point instructions often operate on different registers, so a cast requires a copy.
    - Shorter integer types (char and short) still require the use of a full-sized register, and they need to be padded to 32/64-bits and then converted back to the smaller size before storing back in memory. *(However, this cost must be weighed against the additional memory cost of a larger data type.)*
12. *Be very careful when declaring C++ object variables.*
    - Use initialization instead of assignment (*Color c(black);* is faster than *Color c; c = black;*).
13. *Make default class constructors as lightweight as possible.*
    - Particularly for simple, frequently used classes (e.g., color, vector, point, etc.) that are manipulated frequently.
    - These default constructors are often called behind your back, where you are not expecting it.
    - Use constructor initializer lists. (Use *Color::Color() : r(0), g(0), b(0)* {} rather than *Color::Color()* { *r = g = b = 0;* } .)
14. *Use shift operations >> and << instead of integer multiplication and division, where possible.*
15. *Be careful using table-lookup functions.*
    - Many people encourage using tables of precomputed values for complex functions (e.g., trigonometric functions). For ray tracing, this is often unnecessary. Memory lookups are exceedingly (and increasingly) expensive, and it is often as fast to recompute a trigonometric function as it is to retrieve the value from memory (especially when you consider the trig lookup pollutes the CPU cache).

- In other instances, lookup tables may be quite useful. For GPU programming, table lookups are often preferred for complex functions.

16. *For most classes, use the operators += , -= , *= , and /= , instead of the operators + , - , * , and / .*

    - The simple operations need to create an unnamed, temporary intermediate object.
    - For instance: *Vector v = Vector(1,0,0) + Vector(0,1,0) + Vector(0,0,1);* creates five unnamed, temporary Vectors: *Vector(1,0,0)*, *Vector(0,1,0)*, *Vector(0,0,1)*, *Vector(1,0,0) + Vector(0,1,0)*, and *Vector(1,0,0) + Vector(0,1,0) + Vector(0,0,1)*.
    - The slightly more verbose code: *Vector v(1,0,0); v+= Vector(0,1,0); v+= Vector(0,0,1);* only creates two temporary Vectors: *Vector(0,1,0)* and *Vector(0,0,1)*. This saves 6 functions calls (3 constructors and 3 destructors).

17. *For basic data types, use the operators + , - , * , and / instead of the operators += , -= , *= , and /= .*

18. *Delay declaring local variables.*

    - Declaring object variable always involves a function call (to the constructor).
    - If a variable is only needed sometimes (e.g., inside an if statement) only declare when necessary, so the constructor is only called if the variable will be used.

19. *For objects, use the prefix operator* (++obj) *instead of the postfix operator* (obj++).

    - This probably will not be an issue in your ray tracer.
    - A copy of the object must be made with the postfix operator (which thus involves an extra call the the constructor and destructor), whereas the prefix operator does not need a temporary copy.

20. *Be careful using templates.*

    - Optimizations for various instantiations may need to be different!
    - The standard template library is reasonably well optimized, but I would avoid using it if you plan to implement an interactive ray tracer.
    - Why? By implementing it yourself, you'll know the algorithms it uses, so you will know the most efficient way to use the code.
    - More importantly, my experience is that debug compiles of STL libraries are slow. Normally this isn't a problem, except you will be using debug versions for profiling. You'll find STL constructors, iterators, etc. use 15+% of your run time, which can make reading the profile output more confusing.

21. *Avoid dynamic memory allocation during computation.*

    - Dynamic memory is great for storing the scene and other data that does not change during computation.
    - However, on many (most) systems dynamic memory allocation requires the use of locks to control a access to the allocator. For multi-threaded applications that use dynamic memory, you may actually get a slowdown by adding additional processors, due to the wait to allocate and free memory!
    - Even for single threaded applications, allocating memory on the heap is more expensive than adding it on the stack. The operating system needs to perform some computation to find a memory block of the requisite size.

22. *Find and utilize information about your system's memory cache.*

    - If a data structure fits in a single cache line, only a single fetch from main memory is required to process the entire class.

- Make sure all data structures are aligned to cache line boundaries. (If both your data structure and a cache line is 128 bytes, you will still have poor performance if 1 byte of your structure is in once cache line and the other 127 bytes are in a second cache line).

23. *Avoid unnecessary data initialization.*

    - If you must initialize a large chunk of memory, consider using *memset()*.

24. *Try to early loop termination and early function returns.*

    - Consider intersecting a ray and a triangle. The "common case" is that the ray will miss the triangle. Thus, this should be optimized for.
    - If you decide to intersect the ray with the triangle plane, you can immediately return if the *t* value ray-plane intersection is negative. This allows you to skip the barycentric coordinate computation in roughly half of the ray-triangle intersections. A big win! As soon as you know no intersection occurs, the intersection function should quit.
    - Similarly, some loops can be terminated early. For instance, when shooting shadow rays, the location of the nearest intersection is unnecessary. As soon as *any* occluding intersection is found, the intersection routine can return.

25. *Simplify your equations* on paper*!*

    - In many equations, terms cancel out... either always or in some special cases.
    - The compiler cannot find these simplifications, but you can. Eliminating a few expensive operations inside an inner loop can speed your program more than days working on other parts.

26. *The difference between math on integers, fixed points, 32-bit floats, and 64-bit doubles is not as big as you might think.*

    - On modern CPUs, floating-point operations have essentially the same throughput as integer operations. In compute-intensive programs like ray tracing, this leads to a negligable difference between integer and floating-point costs. This means, you should not go out of your way to use integer operations.
    - Double precision floating-point operations may not be slower than single precision floats, particularly on 64-bit machines. I have seen ray tracers run *faster* using all doubles than all floats on the same machine. I have also seen the reverse.

27. *Consider ways of rephrasing your math to eliminate expensive operations.*

    - *sqrt()* can often be avoided, especially in comparisons where comparing the value squared gives the same result.
    - If you repeatedly divide by x, consider computing $\frac{1}{x}$ and multiplying by the result. This used to be a big win for vector normalizations (3 divides), but I've recently found it's now a toss-up. However, it should still be beneficial if you do more than 3 divides.
    - If you perform a loop, make sure computations that do not change between iterations are pulled out of the loop.
    - Consider if you can compute values in a loop incrementally (instead of computing from scratch each iteration).