Profiling 1

Quote

Measure what is measurable, and make measurable what is not so.
—Galileo Galilei, as reported by Kurt Guntheroth.

Motivation

- We have a beautiful raytracing program
- But we'd like it to finish its work more quickly
- Is our only solution to spend more money on a faster computer?

Tools

- We have three tools we can use
  - Profiler: Compiler will tell you where the “hot spots” are
  - Instrumentation: We can add code to measure the hot spots
  - Notes: Important to keep track of our observations
    - Either pencil and paper or else in a text file

Optimizing

- Optimization is much like scientific experiments:
  - Prediction: Predict where slow spots are
    - Important to commit it to writing
    - Memory is unreliable; writing it down helps to focus, keep records
    - So we don't go in circles!

Optimizing

- Optimization is much like scientific experiments:
  - Prediction: Predict where slow spots are
  - Record changes that have been made to code
    - Either use version control comments
    - Or else write down on paper or in text file

Optimizing

- Optimization is much like scientific experiments:
  - Prediction: Predict where slow spots are
  - Record changes that have been made to code
  - Measure

Optimizing

- Optimization is much like scientific experiments:
  - Prediction: Predict where slow spots are
  - Record changes that have been made to code
  - Measure
    - Record results (in writing!)

Rule of Thumb

- Pareto’s law: 80/20 rule
  - Sometimes quoted as 90/10 rule

Rule

- Amdahl's law: Speedup = $\frac{T_{slow}}{T_{fast}}$
- Normally, we think of this in context of parallelization
- But it also applies for optimization

Example

- Suppose one piece of code accounts for 80% of runtime
- And we can make it take three-quarters the time it previously took
- Overall program speedup: $\frac{1}{0.8*0.75+0.2*1} = 1.25$

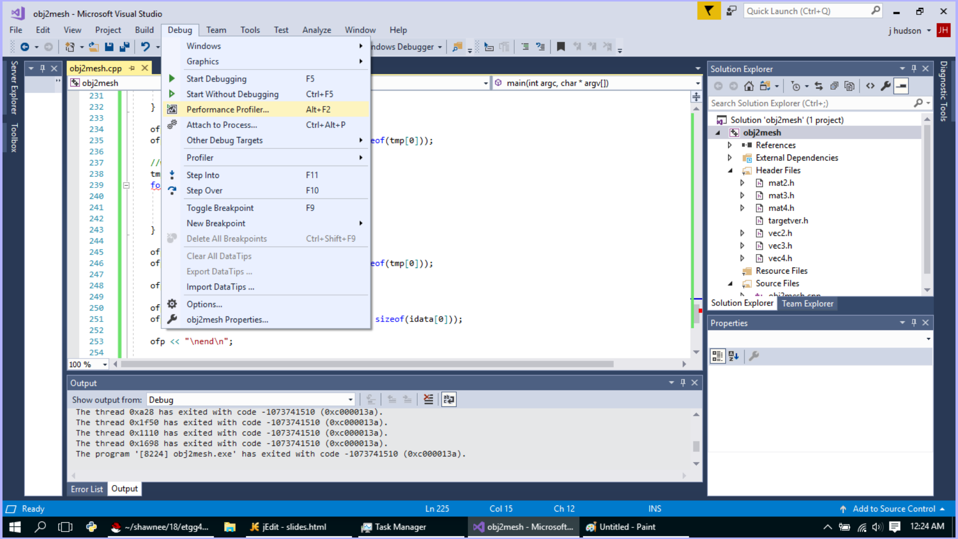
Conversely

- Suppose a piece of code accounts for only 30% of runtime
- We make it take 75% of the time it previously took
- Overall program speedup: $\frac{1}{0.3 * 0.75 + 0.7 * 1}$ = 1.08
  - Overall program is only 8% faster
- So we want to concentrate our efforts where they will have larger payoff

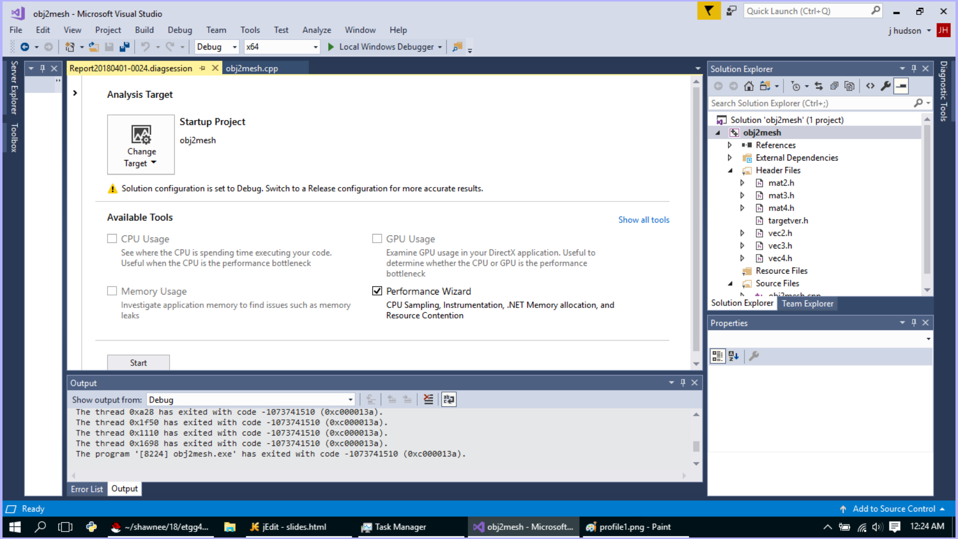
Profiling

- Most compilers/runtime systems today have profiling systems built-in
- We'll look at C/C++ (Visual Studio & Valgrind)

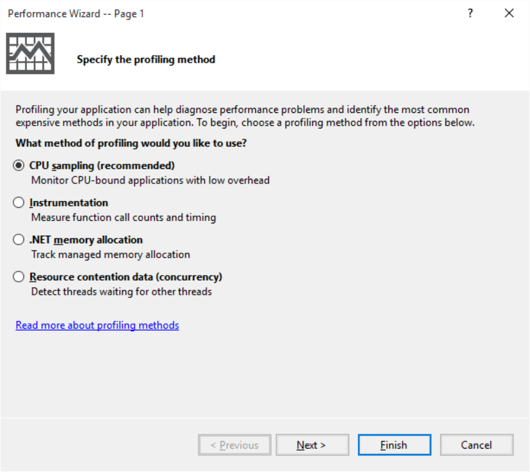
VS


obj2mesh - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Performance Profiler... Alt+F2
Attach to Process... Ctrl+Alt+P
Other Debug Targets
Profiler
Step Into F11
Step Over F10
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Clear All DataTips
Export DataTips ...
Import DataTips ...
Options...
obj2mesh Properties...
Solution Explorer
Solution 'obj2mesh' (1 project)
obj2mesh
References
External Dependencies
Header Files
mat2.h
mat3.h
mat4.h
targetver.h
vec2.h
vec3.h
vec4.h
Resource Files
Source Files
Properties
Output
Show output from: Debug
The thread 0xa28 has exited with code -1073741510 (0xc000013a).
The thread 0x1f50 has exited with code -1073741510 (0xc000013a).
The thread 0x1110 has exited with code -1073741510 (0xc000013a).
The thread 0x1698 has exited with code -1073741510 (0xc000013a).
The program '[8224] obj2mesh.exe' has exited with code -1073741510 (0xc000013a).
Error List Output
Ready Ln 225 Col 15 Ch 12 INS Add to Source Control

VS

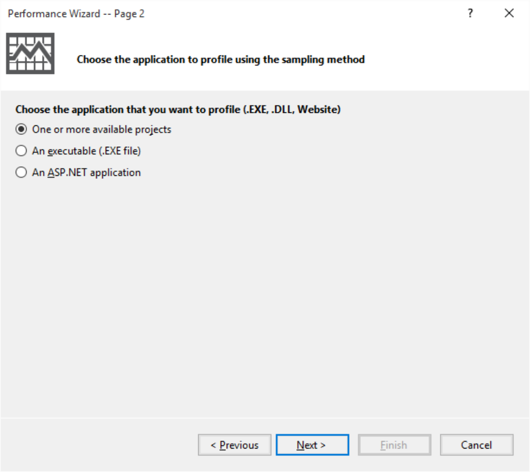

obj2mesh - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
j hudson
JH
Debug
x64
Local Windows Debugger
Server Explorer
Toolbox
Report20180401-0024.diagsession
obj2mesh.cpp
Analysis Target
Change Target
Startup Project
obj2mesh
Solution configuration is set to Debug. Switch to a Release configuration for more accurate results.
Available Tools
Show all tools
CPU Usage
See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck
GPU Usage
Examine GPU usage in your DirectX application. Useful to determine whether the CPU or GPU is the performance bottleneck
Memory Usage
Investigate application memory to find issues such as memory leaks
Performance Wizard
CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention
Start
Output
Show output from: Debug
The thread 0xa28 has exited with code -1073741510 (0xc000013a).
The thread 0x1f50 has exited with code -1073741510 (0xc000013a).
The thread 0x1110 has exited with code -1073741510 (0xc000013a).
The thread 0x1698 has exited with code -1073741510 (0xc000013a).
The program '[8224] obj2mesh.exe' has exited with code -1073741510 (0xc000013a).
Error List
Output
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'obj2mesh' (1 project)
obj2mesh
References
External Dependencies
Header Files
mat2.h
mat3.h
mat4.h
targetver.h
vec2.h
vec3.h
vec4.h
Resource Files
Source Files
Solution Explorer
Team Explorer
Properties
Diagnostic Tools
Ready
Add to Source Control
~/shawnee/18/etgg4...
jEdit - slides.html
Task Manager
obj2mesh - Microsoft...
profile1.png - Paint
12:24 AM

VS

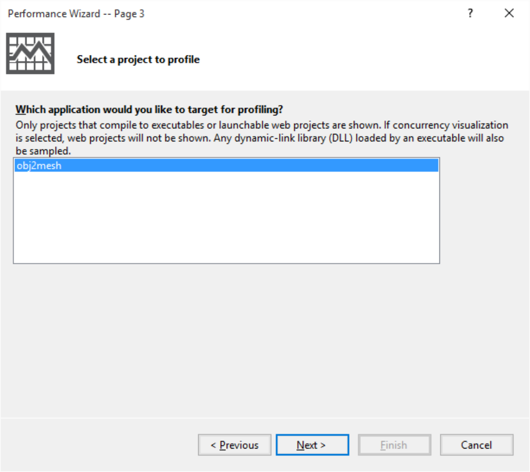


Performance Wizard -- Page 1
Specify the profiling method
Profiling your application can help diagnose performance problems and identify the most common expensive methods in your application. To begin, choose a profiling method from the options below.
What method of profiling would you like to use?
CPU sampling (recommended)
Monitor CPU-bound applications with low overhead
Instrumentation
Measure function call counts and timing
.NET memory allocation
Track managed memory allocation
Resource contention data (concurrency)
Detect threads waiting for other threads
Read more about profiling methods
< Previous
Next >
Finish
Cancel

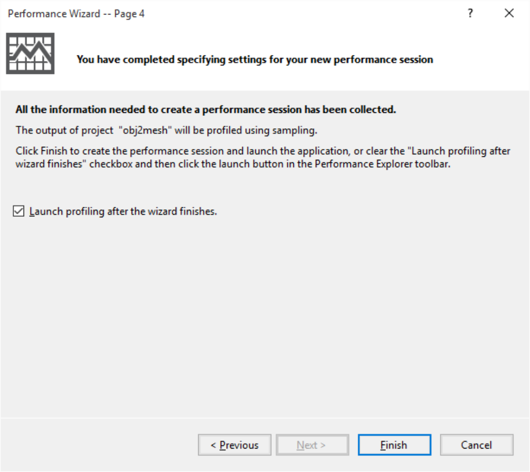
VS


Performance Wizard -- Page 2
?
Choose the application to profile using the sampling method
Choose the application that you want to profile (.EXE, .DLL, Website)
One or more available projects
An executable (.EXE file)
An ASP.NET application
< Previous
Next >
Finish
Cancel

VS


Performance Wizard -- Page 3
Select a project to profile
Which application would you like to target for profiling?
Only projects that compile to executables or launchable web projects are shown. If concurrency visualization is selected, web projects will not be shown. Any dynamic-link library (DLL) loaded by an executable will also be sampled.
obj2mesh
< Previous
Next >
Finish
Cancel

VS


Performance Wizard -- Page 4
You have completed specifying settings for your new performance session
All the information needed to create a performance session has been collected.
The output of project "obj2mesh" will be profiled using sampling.
Click Finish to create the performance session and launch the application, or clear the "Launch profiling after wizard finishes" checkbox and then click the launch button in the Performance Explorer toolbar.
Launch profiling after the wizard finishes.
< Previous
Next >
Finish
Cancel

Methods

- Two methods for C code:
  - Sampling
  - Instrumentation

Sampling

- Sampling = CPU is periodically interrupted by profiler
  - Profiler examines eip/rip register, sees which function is executing
  - Records this information
  - Default: MS uses once per 10M cycles
- Advantage: Less intrusive: Executing code is not modified at all
- Disadvantage: Less exact
  - The more samples, the more exact
  - But more overhead too
- Disadvantage: Sometimes it gives no data!

Instrumentation

- Every function executes extra code in prologue/epilogue
- Advantage: More exact: Every function call is caught
- Disadvantage: More overhead

Example

- (Do in class: Example executions)
- Sampling: Exclusive vs. Inclusive counts
  - On each interrupt, we have one function ip is inside of and other function(s) on call stack
    - Function we're inside of: Gets *exclusive* count incremented
    - Other functions: Get *inclusive* count incremented

Linux

- gcc -g -O3 main.cpp -o main
- valgrind --tool=callgrind ./main scene1.txt
- callgrind_annotate --show-percs=yes --auto=yes callgrind.out.*

Example

- I ran my raytracing code under valgrind on Linux on a scene with 3 spheres and one esh and found that ~92% of the time was in traceTriangles

```
12,199,146,303 (69.07%)  math3d.h:traceTriangles(Scene&, vec3Base<float>&, vec3Base<float>&, vec3Base<float>&,
      vec3Base<float>&, vec3Base<f
loat>&, float&, bool&)
 4,044,274,321 (22.90%)  traceRay.h:traceTriangles(Scene&, vec3Base<float>&, vec3Base<float>&, vec3Base<float
      >&, vec3Base<float>&, vec3Base<float>&, float&, bool&) [main-o3]
 1,208,483,840 ( 6.84%)  /usr/include/c++/9/bits/stl_vector.h:traceTriangles(Scene&, vec3Base<float>&,
      vec3Base<float>&, vec3Base<float>&, vec3Base<float>&, vec3Base<float>&, float&, bool&)
    63,006,219 ( 0.36%)  ???:0x0000000000027e90 [/usr/lib64/libpng16.so.16.36.0]
```

- Notice that (due to optimization) profiler thinks traceTriangles is spread out between several source code files

Profiling

- Let’s drill down into the functions...

```
          .           void traceSpheres(Scene& scene, vec3& s, vec3& v, vec3& ip, vec3& N, vec3& color, float& closestT, bool& inter)
1,572,864 ( 0.01%)    {
  262,144 ( 0.00%)        unsigned ci=(unsigned)-1;
5,242,880 ( 0.03%)        for(unsigned i=0;i<scene.spheres.size();++i){
          .                   auto q = s - scene.spheres[i].c;
          .                   auto A = dot(v,v);
  786,432 ( 0.00%)            auto B = 2*dot(q,v);
1,572,864 ( 0.01%)            auto C = dot(q,q)-scene.spheres[i].r*scene.spheres[i].r;
2,359,296 ( 0.01%)            auto disc = B*B-4*A*C;
1,572,864 ( 0.01%)                if( disc < 0 )
          .                       continue;
   60,024 ( 0.00%)            disc = sqrt(disc);
   80,032 ( 0.00%)            auto denom = 1.0 / (2.0*A);
   60,024 ( 0.00%)            auto t1 = (-B + disc) * denom;
  604,320 ( 0.00%)            auto t2 = (-B - disc) * denom;
          .                   float t;
   60,024 ( 0.00%)            if( t1 < 0 && t2 < 0 ){
          .                       continue;
          .                   } else if( t1 < 0 )
          .                       t = t2;
   40,016 ( 0.00%)            else if( t2 < 0 )
          .                       t = t1;
          .                   else
   40,016 ( 0.00%)                t = ( (t1<t2) ? t1:t2 );
          .
   60,024 ( 0.00%)            if( t < closestT ){
   19,980 ( 0.00%)                closestT = t;
   19,980 ( 0.00%)                ci = i;
          .                   }
          .               }
  524,288 ( 0.00%)        if( ci == (unsigned)-1 ){
  484,328 ( 0.00%)            inter=false;
          .                   return;
          .               }
   79,920 ( 0.00%)        ip = s + closestT * v;
   59,940 ( 0.00%)        N = normalize(ip - scene.spheres[ci].c);
   79,920 ( 0.00%)        color = scene.spheres[ci].color;
   39,960 ( 0.00%)        inter=true;
```

Analysis

- We see that traceSpheres doesn't account for much overall runtime
- But: This input file had three spheres and 768 triangles
- We should run program on more input files to get better sense of where bottlenecks are

Profile

- What about traceTriangles?

```
            .          void traceTriangles(Scene& scene, vec3& s, vec3& v, vec3& ipC, vec3& N,
            .                      vec3& color, float& closestT, bool& inter)
    2,621,440 ( 0.01%)  {
    1,572,864 ( 0.01%)      for(auto& M : scene.meshes ){
  807,141,376 ( 4.57%)          for(unsigned i=0;i<M.triangles.size();++i){
            .                       Triangle& T = M.triangles[i];
            .                       float denom = dot(T.N,v);
  805,306,368 ( 4.56%)              if( denom == 0.0 )
            .                           continue;
  402,915,328 ( 2.28%)              float numer = -(T.D + dot(T.N,s) );
  201,326,592 ( 1.14%)              float t = numer/denom;
  402,653,184 ( 2.28%)              if( t < 0 )
            .                           continue;
            .                       auto ip = s + t * v;
            .                       float A=0.0;
            .                       for(unsigned j=0;j<3;++j)
  472,522,149 ( 2.68%)                  A += length( cross(T.e[j], ip-T.p[j] ) );
  472,522,149 ( 2.68%)              A *= T.oneOverTwiceArea;
  473,046,437 ( 2.68%)              if( A > 1.001 )
            .                           continue;
      163,260 ( 0.00%)              if( t < closestT ){
       82,719 ( 0.00%)                  ipC = ip;
      110,292 ( 0.00%)                  N = T.N;
      110,292 ( 0.00%)                  color = M.color;
       27,573 ( 0.00%)                  closestT = t;
       55,146 ( 0.00%)                  inter = true;
            .                       }
            .                   }
            .               }
    2,097,152 ( 0.01%)  }
```

Analysis

- Notice that this function doesn't seem to be very significant time-wise either!
- But...

Profile

- Take a look at traceRay:

```
          .            bool traceRay(Scene& scene, vec3& s, vec3& v, vec3& ip, vec3& N, vec3&
                  color)
          .            {
    262,144 ( 0.00%)       float closestT = 1E99;
          .                bool inter;
  2,359,296 ( 0.01%)       traceSpheres(scene,s,v,ip,N,color,closestT,inter);
 35,782,500 ( 0.20%)  => traceRay.h:traceSpheres(Scene&, vec3Base<float>&, vec3Base<float>&,
     vec3Base<float>&, vec3Base<float>&, vec3Base<float>&, float&, bool&) (262,144x)
  2,359,296 ( 0.01%)       traceTriangles(scene,s,v,ip,N,color,closestT,inter);
17,452,166,608 (98.82%)  => traceRay.h:traceTriangles(Scene&, vec3Base<float>&, vec3Base<float
     >&, vec3Base<float>&, vec3Base<float>&, vec3Base<float>&, float&, bool&) (262,144x)
          .                return inter;
          .            }
```

Analysis

- Notice that this says that traceTriangles *is* responsible for >98% of the program's execution time but traceTriangles doesn't seem to be anywhere near that costly
- What's going on????

Explanation

- This report is showing only *exclusive* counts
- We need to look through the rest of the report for high-cost operations
- When I did so, this is what I found...

Profile

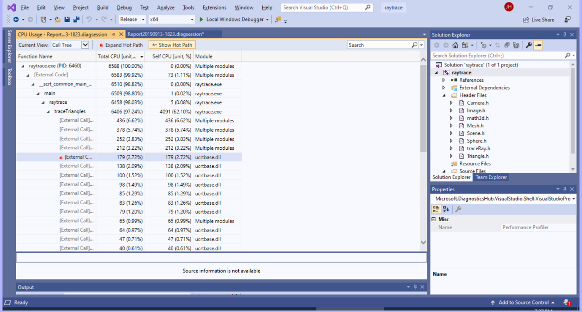
```
            .          float dot(const vec3& v, const vec3& w){
3,231,002,556 (18.29%)     return v.x*w.x + v.y*w.y + v.z*w.z;
            .          }

1,422,912,839 ( 8.06%)     MyType operator-(const MyType& other) const {      op3(-);     }

            .          vec3 cross(const vec3& v, const vec3& w){
1,417,568,757 ( 8.03%)     return vec3(
  472,522,919 ( 2.68%)         v.y*w.z - w.y*v.z,
  945,045,068 ( 5.35%)         w.x*v.z - v.x*w.z,
1,417,567,217 ( 8.03%)         v.x*w.y - w.x*v.y
            .              );
            .          }

            .              float magnitudeSq() const {
            .                  float total;
            .                  total  = this->x*this->x;
  945,230,236 ( 5.35%)         total += this->y*this->y;
  945,230,236 ( 5.35%)         total += this->z*this->z;
            .                  return total;
            .              }
```

VS

- Let’s see what Visual Studio has to say...

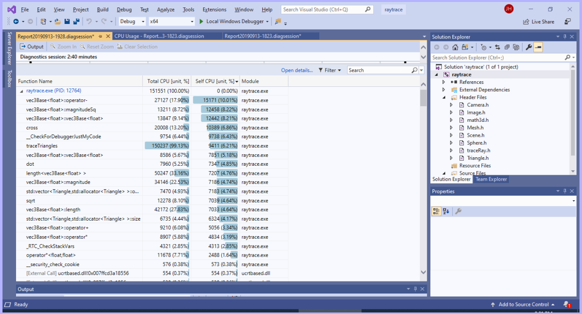
VS

- Using Release mode, we may not get very useful results



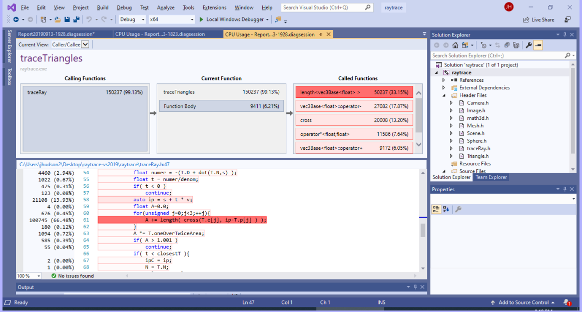
VS

- VS warns against using Debug mode, but it gives us more information



VS

- We can zoom in on a single function (double click its name)



Analysis

- We can see that no single statement is responsible for a large chunk of the execution time
  - If it was, that would be an obvious place to start our optimization efforts

Analysis

- Much of our time is being spent in fundamental operations like dot, cross, subtraction
- Not really clear how we could speed these up (other than by doing fewer of them)

Speeding Up

- Can we make the program faster, though?
- Yes!
- A few strategies that we can apply in these cases

Technique

- Don’t recompute data in a loop if we don’t need to
- Look carefully at sphere intersection routines. Is there anything being repeatedly computed that need not be?
  - Go back a few slides, have a look...

Technique

- Look more closely:

```
auto q = s - scene.spheres[i].c;
auto A = dot(v,v);
auto B = 2*dot(q,v);
auto C = dot(q,q)-scene.spheres[i].r*scene.spheres[i].r;
```

- Observe: A depends only on v
  - Which is constant for all iterations of loop
- Observe: C depends on r^2 which is constant for each sphere

Code

- I rewrote the code like so:

```
auto A = dot(v,v);
auto fourA = 4.0*A;
auto denom = 2.0 / (fourA);
unsigned ci=(unsigned)-1;
for(unsigned i=0;i<scene.spheres.size();++i){
    auto q = s - scene.spheres[i].c;
    auto B = 2*dot(q,v);
    auto C = dot(q,q)-scene.spheres[i].r2;
    auto disc = B*B-fourA*C;
    if( disc < 0 )
        continue;
    disc = sqrt(disc);
    auto t1 = (-B + disc) * denom;
    auto t2 = (-B - disc) * denom;
    ...rest is the same...
}
```

Results

- I ran the code on an input file with 1000 spheres
  - Original code: 1.145 sec
  - New code: 1.152 sec
- Hmmm...

Explanation

- Modern compilers are very good at hoisting loop invariants
- So we don’t really see any significant benefit from doing the work ourselves

Technique

- Second technique: Examine the algorithm we're using and make it better
  - Advantage: This can give significant speedups
  - Disadvantage: No general way to approach this
    - You just need to study your code carefully and *reason* it through
- Let's see some examples

Example

- Recall our scheme to find ray-sphere intersections:
  - Ray start, direction are $\vec{s}$ & $\vec{v}$; sphere center,radius are $\vec{c}$ & r; $\vec{q} = \vec{s} - \vec{c}$
- $t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$
  - A=$\vec{v} \cdot \vec{v}$
  - B=2$(\vec{q} \cdot \vec{v})$
  - C= $\vec{q} \cdot \vec{q} - r^2$
  - If discriminant < 0: No intersection
  - Otherwise, take smallest value of t that is nonnegative, compare it to the closest t value found so far
- This can be made better!

Observe

- We defined B=2$(\vec{q} \cdot \vec{v})$
- But what if we actually plug B into the quadratic formula?
- $t = \frac{-(2(\vec{q} \cdot \vec{v})) \pm \sqrt{(2(\vec{q} \cdot \vec{v}))^2 - 4AC}}{2A}$
- $t = \frac{-2(\vec{q} \cdot \vec{v}) \pm \sqrt{4(\vec{q} \cdot \vec{v})^2 - 4AC}}{2A} = \frac{-2(\vec{q} \cdot \vec{v}) \pm \sqrt{4((\vec{q} \cdot \vec{v})^2 - AC)}}{2A}$
- $t = \frac{-2(\vec{q} \cdot \vec{v}) \pm 2\sqrt{(\vec{q} \cdot \vec{v})^2 - AC}}{2A} = \frac{-(\vec{q} \cdot \vec{v}) \pm \sqrt{(\vec{q} \cdot \vec{v})^2 - AC}}{A}$

Observe

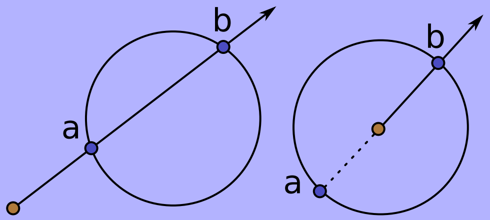
- We defined A=$\vec{v} \cdot \vec{v}$
  - If the vector is unit length, we know A=1
  - So: Normalize the ray direction and then simplify further:
  - $t = -(\vec{q} \cdot \vec{v}) \pm \sqrt{(\vec{q} \cdot \vec{v})^2 - C}$

Branches

- We always take the smallest (nonnegative) t value
- Right now, this involves branchy code
  - We saw how that can result in slower CPU performance due to branch misprediction
- We'd like to reduce or eliminate branches

Branches

- Let a and b represent the two values of t where the ray intersects the sphere
- Let a=$-(\vec{q}\cdot\vec{v}) - \sqrt{(\vec{q}\cdot\vec{v})^2 - C}$, b=$-(\vec{q}\cdot\vec{v}) + \sqrt{(\vec{q}\cdot\vec{v})^2 - C}$
- Clearly, $b \geq a$
- We can have two possibilities:



Upshot

- If we know that the ray always starts outside the sphere, we can just use $t = -(\vec{q} \cdot \vec{v}) - \sqrt{(\vec{q} \cdot \vec{v})^2 - C}$
- Note that we must still check for $t \geq 0$

Code

```
unsigned ci=(unsigned)-1;
v = normalize(v);
for(unsigned i=0;i<scene.spheres.size();++i){
    auto q = s - scene.spheres[i].c;
    auto C = dot(q,q)-scene.spheres[i].r2;
    auto qDotv=dot(q,v);
    auto disc = qDotv*qDotv-C;
    if( disc < 0 )
        continue;
    float t = -qDotv-sqrt(disc);
    if( t>0 && t < closestT ){
        closestT = t;
        ci = i;
    }
}
```

Results

- Does this give us a.significant improvement?
- I tested on a scene with 1000 spheres, 512x512 resolution
  - Original code: 1.145 sec
  - New code: 0.841 sec
  - Speedup: 1.36

Triangles

- What about the triangle code?
  - If we know that our platform implements IEEE arithmetic: Anything divided by zero is ∞, so we need not check for division by zero

Triangles

- Our intersection test involves a call to length(), which means taking a square root
  - Square root is typically pretty expensive
- We can use an alternate formulation: The *scalar triple product*: $(\vec{u} \times \vec{v}) \cdot \vec{w}$
  - Suppose we have a tetrahedron with vertices a,b,c,d
  - We can compute the volume of this tetrahedron by computing $((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) \cdot (\vec{d} - \vec{a})$

Point In Triangle

- Given a triangle with vertices p_0, p_1, p_2 and potential intersection point $\vec{ip} = \vec{s} + t\vec{v}$:
  - Compute volume of tetrahedron with vertices $\vec{s}, \vec{ip}, p_0, p_1$
  - Compute volume of tetrahedron with vertices $\vec{s}, \vec{ip}, p_0, p_2$
  - Compute volume of tetrahedron with vertices $\vec{s}, \vec{ip}, p_1, p_2$
- If any volume is negative, intersection point is outside triangle; else, it's inside

Implementation

- Helper function:

```
bool scalarTripleIsNegative(const vec3& a, const vec3& b, const
    vec3& c){
     return dot(cross(a,b),c)<0.0f;
}
```

Intersection

- The intersection testing code:

```
for(auto& M : scene.meshes ){
    for(unsigned i=0;i<M.triangles.size();++i){
        Triangle& T = M.triangles[i];
        float denom = dot(T.N,v);   //if zero: t=infinity
        float numer = -(T.D + dot(T.N,s) );
        float t = numer/denom;
        if( t < 0  || t >= closestT )
            continue;
        auto vv = t*v; //(s + t*v) - s
        auto v0 = T.p[0]-s;
        auto v1 = T.p[1]-s;
        auto v2 = T.p[2]-s;
        if( scalarTripleIsNegative( vv, v0,v2 ) ||
            scalarTripleIsNegative( vv, v1,v0 ) ||
            scalarTripleIsNegative( vv, v2,v1 )){
            continue;
        }
        auto ip = s + vv;
        ipC = ip;
        N = T.N;
        color = M.color;
        closestT = t;
        inter = true;
    }
}
```

Results

- For scene2.txt
- Original code:
  - 15.60 seconds (Linux, Core i7-5500, 2.4GHz, gcc 9.2.1, 64 bit, -O3 -march=native)
  - 24.88 seconds (Windows 10, Core i7-6500U, 2.5GHz, VS 2019, 64 bit, Release, Favor speed, use AVX instruction set)
  - 23.53 seconds (Windows, no debugger active)
- New code:
  - 8.08 seconds (Linux),
  - 10.89 seconds (Windows, no debugger active)
- Speedup: 1.93 (Linux), 2.16 (Windows) (Twice as fast!)

Principle

- Understanding the underlying mathematics/algorithms can enable us to obtain significant benefits

Observe

- The values v0, v1, v2 are constant for a particular ray
- Maybe we'd see a speedup if we precomputed and saved these values for the triangles...

Code

```
for(auto& M : scene.meshes ){
    for(unsigned i=0;i<M.triangles.size();++i){
        Triangle& T = M.triangles[i];
        float denom = dot(T.N,v);   //if zero: t=infinity
        float numer = -(T.D + dot(T.N,s) );
        float t = numer/denom;
        if( t < 0  || t >= closestT )
            continue;
        auto vv = t*v; //(s + t*v) - s
        if( scalarTripleIsNegative( vv, T.pMinusS[0],T.pMinusS[2]) ||
            scalarTripleIsNegative( vv, T.pMinusS[1],T.pMinusS[0]) ||
            scalarTripleIsNegative( vv, T.pMinusS[2],T.pMinusS[1]) ){
            continue;
        }
        auto ip = s + vv;
        ipC = ip;
        N = T.N;
        color = M.color;
        closestT = t;
        inter = true;
    }
}
```

Note

- We also must tweak traceRay:

```
bool traceRay(Scene& scene, vec3& s, vec3& v, vec3& ip, vec3& N, vec3& color)
{
    float closestT = 1E99;
    bool inter;
    //----------new code
    static bool didpMinusS=false;
    if(!didpMinusS ){
        didpMinusS=true;
        for(auto& M : scene.meshes ){
            for(auto& T : M.triangles ){
                for(int i=0;i<3;++i){
                    T.pMinusS[i] = T.p[i] - s;
                }
            }
        }
    }
    //----------new code
    traceSpheres(scene,s,v,ip,N,color,closestT,inter);
    traceTriangles(scene,s,v,ip,N,color,closestT,inter);
    return inter;
}
```

Results

- Original code: 15.60 seconds
- Point-in-triangle using tetrahedron instead of barycentric: 8.08 seconds
- Tetrahedron + precomputation:7.56 seconds

Notice

- We have this code at the top of our function:

```
for(unsigned i=0;i<M.triangles.size();++i){
    Triangle& T = M.triangles[i];
    ...
}
```

- Question: How much time does this take?

Size

- I looked up the implementation of size() in my compiler (g++, libstdc++ 9.1.1)

```
size_type
size() const _GLIBCXX_NOEXCEPT
{ return size_type(this->_M_impl._M_finish - this->_M_impl.
    _M_start); }
```

- That's ugly!
- But strip out the extraneous stuff and we see: It does an integer subtraction when called

[ ]

- What about the [ ] operator?

```
reference
operator[](size_type __n) _GLIBCXX_NOEXCEPT
{
__glibcxx_requires_subscript(__n);
return *(this->_M_impl._M_start + __n);
}
```

- This involves an addition (and a hidden multiplication, since _M_start is a pointer)

Question

- What if we used the C++ 11 for-each iterator?

```
for(auto& T : M.triangles ){
        ...
}
```

Results

- Original code: 15.60 seconds
- Point-in-triangle using tetrahedron instead of barycentric: 8.08 seconds
- Tetrahedron + precomputation:7.56 seconds
- Tetrahedron + precomputation + for-each: 7.35 seconds

Sources

- Kurt Guntheroth. Optimized C++. O'Reilly Media.
- Christer Ericson. Real Time Collision Detection. CRC Press.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org