Performance Tuning

The Need for Tuning (1 of 2)

- You don’t need to tune your code!
- Most important → Code that works
- Most important → Code that is clear, readable
  - It will be re-factored
  - It will be modified by others (even you!)
- Less important → Code that is efficient, fast
  - Is performance really the issue?
  - Can a hardware upgrade fix performance problems?
  - Can game design fix performance problems?
- Ok, so you do really need to improve performance
  - All good game programmers should know how to …

The Need for Tuning (2 of 2)

- In most large games, typically small amount of code uses most CPU time (or memory)
  - Good game programmer knows how to identify such code
  - Good game programmer knows techniques to improve performance
- Questions you (as a good programmer) may want answered:
  - How slow is my game?
  - Where is my game slow?
  - Why is my game slow?
  - How can I make my game run faster?

Steps for Tuning Performance

- Measure performance
  - Timing and profiling
- Identify “hot spots”
  - Where code spends the most time/resources
- Apply techniques to improve performance
  - Tune
- Re-evaluate

Outline

- Introduction (done)
- Timing (next)
- Benchmarks
- Profiling
- Tuning
- Summary

Time Your Game

- /usr/bin/time (Windows has timeit.exe)

<table>
<thead>
<tr>
<th>Elapsed</th>
<th>User</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:24:04</td>
<td>15.26 sec</td>
<td>2.74 sec</td>
</tr>
<tr>
<td>CPU</td>
<td>11%</td>
<td></td>
</tr>
</tbody>
</table>

Time Your Game

- Elapsed: Wall-clock time from start to finish
- User: CPU time spent executing game
- System: CPU time spent within OS game’s behalf
- CPU: Percent time processing vs blocked for I/O
- Useful, since provides a guideline for user-code (that can be optimized) and general processing/waiting
  - However, I/O accounting isn’t always accurate
- But … which parts are most time consuming?
Time Parts of Your Game

- Call before and after
  ```
  start = getTime()
  // do stuff
  stop = getTime()
  elapsed = stop - start
  ```
- (Where did we do this before?)
- Use Dragonfly Clock
  - Remember, this is not a singleton
- E.g.
  ```
  clock.delta() // start timer
  Pathfind() // do stuff
  elapsed = clock.delta() // compute elapsed
  ```

Outline

- Introduction (done)
- Timing (done)
- Benchmarks (next)
- Profiling
- Tuning
- Summary

Benchmark

- Benchmark – a program to assess relative performance
  - e.g. Compare ATI and NVIDIA video cards
  - e.g. Compare Google Chrome to Mozilla Firefox
- A “good” benchmark will assess performance using typical workload
  - Getting “typical” workload often difficult part
- Use benchmark to compare performance before and after performance. e.g.
  - Run benchmark on Dragonfly → old
  - Tune performance
  - Run benchmark on Dragonfly → new
  - Is new better than old?
- What is a good benchmark for Dragonfly? What should it do?

Bounce – What is it?

- A benchmark designed to estimate Dragonfly performance
  - Primarily dependent upon number of objects can support at target frame rate
  - Assumes “standard” game creates many objects that move and interact
  - Bounce stresses Dragonfly by creating many objects that move and collide
  - When Dragonfly can’t keep up, has reached limit
  - Record value – provides basis for comparison
    - Compare other systems (e.g. faster processor), engine improvements (e.g. scene graph)

Screenshot/Demo

Steps to use
1. Download from Web page
2. Compile
   (Modify Makefile to point to Dragonfly)
3. Run

http://www.youtube.com/watch?v=82G5fzyJYrY&feature=youtu.be

Bounce Details

- Balls random speed (0.1 to 1 spaces/step) and direction
- Balls solid, so collide with other objects and screen edge
- Start → 0 Balls
- Each step → Create one ball
  - So, about 30/second
- Record frame time for latest 30 steps
  - So, about 1 second of time
- Compute median
- If median 10% over target frame time (33 ms) , stop iteration
- Record number of Balls created
- After three iterations → average Balls/iteration is max objects (bounce-mark)
  - (Show code: Ball, Bouncer, bounce)
Bounce Data (1 of 2)

** grep BOUNCE dragonfly.log
- 05:29:36 BOUNCE: Frame 1 - 33 of 33 msec ( median is 0 )
- 05:29:36 BOUNCE: Frame 2 - 33 of 33 msec ( median is 0 )
- 05:29:36 BOUNCE: Frame 3 - 33 of 33 msec ( median is 0 )
- ...
- 05:32:34 BOUNCE: Frame 1772 - 38 of 33 msec ( median is 36 )
- 05:32:34 BOUNCE: Frame 1773 - 38 of 33 msec ( median is 37 )
- 05:32:34 BOUNCE: Iteration 3 - max objects: 1773
- 05:32:34 BOUNCE: Done. Average max objects: 1780

Bounce Data (2 of 2)

** Average maximum number of objects (bounce-mark): 1803 **

System
- Intel I5-2500, 3.30 GHz
- 8GB RAM
- Windows 7 64-bit, Service Pack 1
- Cygwin

Bounce Results
- 61x20 squares. Dependent upon resolution?
  - 2400x1250 pixels → 360 objects
  - 500x300 pixels → 353 objects
- 290x100 squares. Dependent upon squares?
  - ~2400x1250 pixels → 331
  - ~500x300 pixels → 350
- May want to take minimum bounce-mark. Or, may want take “typical” setup. Or, may want your setup.
  - Will definitely want setup that meets target specifications!

Bounce – What Does it Mean?
- Provides target maximum number of moving objects Engine can support
- Note, game-code computations “cost”, too, so will decrease max
- Note, if single moving object, can support about \( n^2 \) as many objects (e.g. Walls)
- In general:
  - \( B = \text{estimated maximum reported by Bounce} \)
  - \( M = \text{number of moving objects} \)
  - \( S = \text{number of static (non-moving) objects} \)
  - \( \text{Need } M * (M + S) \leq B^2 \)
- Note, this could be refined with “velocity” for more accuracy (and more complications)

How to Use for Planning
- Say Bounce reports 500 objects for target setup (\( B = 500 \))
  - Making game, say a maze runner
    - 10x10 walls
    - Hero and up to 10 bad guys
    - Can Dragonfly support?
      - \( M = 11, S = 10000 \)
      - \( 11 * (11 + 10000) \leq 500*500 ? \)
      - \( 111,121 \leq 250,000 \) (yes)
  - Say 10x bigger world. And bullets, up to 50 “in flight” during firefight
    - Can Dragonfly support?
      - \( M = 61, S = 100000 \)
      - \( 61 * (61 + 100000) \leq 250000 \)
      - \( 9,601,721 \leq 250,000 \) (no)
- What to do?
  - Tune code (more later)
  - Design differently
    - Don’t spawn bad guys until Hero can see them
    - Make levels simpler (but have more of them)
    - Make sections of walls combined into objects
    - Reduce movement speed / fire rate

Outline
- Introduction (done)
- Timing (done)
- Benchmarks (done)
- Profiling (next)
- Tuning
- Summary
Profiling

- Why?
  - Learn where program spent time executing
  - Which functions called
  - Can help understand where complex program spends its time
  - Can help find bugs
- How?
  - Re-compile so every function call records some info
  - After running, profiler figures out what called, how many times
  - Also, takes samples to see where program is (about 100/sec)
  - Keeps histogram

GNU profiler
- Linux, and can install with cygwin, too
- Works for any language GNU compiler supports: C, C++, Objective-C, Java, Ada, Fortran, Pascal...
- For us -> g++
- Broadly, after profiling, outputs: flat profile and call graph
- Flat profile provides overall "burn" perspective
  - How much time program spent in each function
  - How many times function was called
- Call graph shows individual execution profile for each function
  - Which functions called it
  - Which other functions it called
  - How many times
  - Estimate how much time in subroutines of each function

http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf

Running gprof

1) Compile with -pg flag
   - Need for creating all .o files
   - And need when linking!
2) Run program normally
   - Produces file "gmon.out" (overwritten if there)
   - Note, program must exit normally! (e.g. via exit() or return from main())
3) Run gprof on program
   - Uses data from gmon.out
   - Often, redirect to file via ’>
4) Analyze output

Example - Bounce

- Compile
  - g++  -c -pg  -I../../dragonfly Ball.cpp -o Ball.o
  - g++  -c -pg  -I../../dragonfly Bouncer.cpp -o Bouncer.o
  - g++ bounce.cpp Ball.o Bouncer.o libdragonfly.a -pg -o bounce -lncurses -lrt
- Run
  - ./bounce
- Profile
  - gprof bounce > out
- Analyze
  - (emacs or vi or pico or less) out

Gprof – Flat Profile (e.g. QuickSort)

<table>
<thead>
<tr>
<th>index</th>
<th>time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.00</td>
<td>2.68</td>
<td>main [1]</td>
<td>quicksort [2]</td>
</tr>
<tr>
<td></td>
<td>0.07</td>
<td>0.00</td>
<td>1/1</td>
<td>quicksort [2]</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>97.4</td>
<td>0.08</td>
<td>2.53</td>
<td>1/13330614</td>
<td>quicksort [2]</td>
</tr>
<tr>
<td></td>
<td>2.27</td>
<td>0.25</td>
<td>6665307</td>
<td>6465307</td>
<td>partition [3]</td>
</tr>
<tr>
<td></td>
<td>13330614</td>
<td>quicksort [2]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanations
- Each line describes one function
- name: name of function
- time: percentage of time spent executing
- cumulative time: total time spent
- self time: time spent executing
- calls: number of times function called (excluding recursive)
- self time per call: avg time per exec (excluding descendents)
- total self time per exec: total time per exec (including descendents)

Observations
- swap() called many times, but each fast
- consumes only 1% of overall time
- partition() called many times, fast
- consumes 8% of overall time

Conclusions
- Improve performance
- Don’t try to make fillArray() or quicksort() faster

Gprof – Call Graph Profile

- Each section describes one function
- Which functions called it, and how much time was consumed
- Which functions it calls, how many times, and for how long
- Usually overkill -> we won’t look at it in too much detail
Example – Bounce

Using Profiling (1 of 2)

- Determine where to optimize
  - Pick the bottleneck and make more efficient
    - This provides most "bang for the buck" (bucks = time, often!)
- E.g.
  - Program takes 10 seconds to execute
    - Function A() takes 10% of the time
      - Make A() more efficient!
    - How long does program take? \( \Rightarrow 9.1 \) seconds
    - Function B() takes 90% of the time
      - Instead of working on A(), make B() 50% more efficient
      - How long does program take? \( \Rightarrow 5.5 \) seconds
- Bottleneck will then move to this ok and expected
  - Repeat, as needed

Using Profiling (2 of 2)

- Warning! Just because bottleneck moves does not mean performance is improving!
- E.g. Say boxIntersectsBox() is bottleneck
  - Could alleviate by checking distance between objects before doing boxIntersectsBox()
  - Then boxIntersectsBox() called less often would be small
  - But, distanceObjects() now huge!
- Is this better? Could be \( \Rightarrow \) but only if distance test "cheap" than intersection test
- Can't make code more efficient (e.g. library?) \( \Rightarrow \) may be able to redesign game
  - Q: Consider Mario-type platformer that "can't keep up". How to redesign to improve performance?
    - A: make levels smaller
    - A: spawn/move objects only when Hero is near
    - A: perhaps new type of object – "platform" for movement?
Statistical Inaccuracies (1 of 3)

- Count of function calls is accurate
- Time/percent for function calls may not be → they sampled
- Samples only during run-time
  - So, if game waiting on I/O (say, file or input) won’t show up even if it caused big I/O
- Beware that periodic samples may exactly miss some routines
- Observer effect – by observing behavior of program, we change it
  - This is true for almost any measurements
  - Certainly true for profiling

Statistical Inaccuracies (2 of 3)

- Actual error larger than one sampling period
- The more samples, the larger the cumulative error
- Guideline: value n times sampling period → expected error is square-root of n sampling periods
  - Say 0.5 seconds for GameObjectList::isDone()
  - Sample period is 0.01 seconds, so 50 times as large
  - So, average error is sqrt(50) = ~7 sample periods → 0.07 seconds (maybe more)
- Note, small run-time (less than sample period) could still be useful
  - E.g. Program’s total run-time large, then small run-time for one function says that function used little of whole → not worth optimizing

Statistical Inaccuracies (3 of 3)

- To get more accuracy, run program longer
- Or, combine data from several runs
  1. Run program once (e.g. a.out)
  2. Move “gmon.out” to “gmon.sum”
  3. Run program again
  4. Merge:
     gprof -s a.out gmon.out gmon.sum
  • Repeat steps 3 and 4, as needed
  • Combine the cumulative data then analyze:
     gprof a.out gmon.sum > output-file

Outline

- Introduction (done)
- Timing (done)
- Benchmarks (done)
- Profiling (done)
- Tuning (next)
- Summary

Tuning (1 of 4)

- Can choose better algorithms or data structures
  - Mergesort instead of Quicksort?
  - Linked List instead of Array?
- Compiler optimizations
  - gcc -Ox
  - x from 1 to 3, with some to more optimizations
  - man gcc, for details
- Unroll loops (compiler optimizations sometimes do this automatically)
- Re-write in assembly (but many compilers excellent)
- Inline function calls

Tuning (2 of 4)

- Better memory efficiency
  - Memory is cheap, so not reduce memory for cost
  - Rather, reduce use for performance → less access often means keeping CPU busier
  - Keep locality of reference to improve performance
    • Pointers tend to scatter locality
    • Arrays preserve locality
  - Use smaller data structures if possible
    • E.g. short instead of int
    • E.g. smaller max size on arrays
  - Compiler option -Os (for size optimization)
Tuning (3 of 4) – Multi-threading

• Many modern CPU’s have multiple cores
  — Can think of each as a separate CPU
• Great if doing 2 independent tasks at once
  — E.g. surfing web while playing music
• Potential speedup is enormous (e.g. 4 core CPU may run up to 4 times faster or support 4 times as many objects)
• How to take advantage of for single application (e.g. game)?
  — Concurrency through multi-threading
• How to this?
  — Easy on the surface (see right)
• So, what’s the problem?
  — Need to share data
  — Thread execution order not deterministic
  — Threads need to synchronize

```c
int a[max];

void DoStuff() {
    for (int i=0; i<max; i++)
        a[i] = i;
}

main() {
    beginThread(DoStuff);
    for (int i=0; i<max; i++)
        a[i] = max - i;
}
```

Tuning (4 of 4) – Multi-threading

• Could partition tasks
  — e.g. Half of array for each thread
• Could “lock” data when using
  — But wastes CPU time when other thread waiting
• Threading best speedup for independent tasks that minimize thread synchronization
• In Dragonfly, would multithreading help? How would you implement it?

Final Notes

• Improving performance is not the first task of a programmer. Nor the second. Nor the third. In fact, it might never be a task!
• Correctly working code is more important than performance
• Code clarity is more important than performance
• Don’t improve performance unless you have to!
• Improving performance is not the last task of a programmer
  — You must test thoroughly after tuning may introduce bugs!
• However, when performance becomes the last obstacle between a working, playable, fun game — you better know how
  — Requires “deep” technical knowledge

Summary

• Tune performance when necessary
  — (Are there easier solutions to the problem?)
• Need measures of performance to gauge potential improvements
  — Timing
  — Benchmarks
  — Profile sections of code
• Identify bottlenecks where most time spent
  — That is where improvements should be targeted
• Apply techniques to improve performance
  — Data structures, algorithms, compiler optimizations, multithreading…
  — Pick the right tool for the job!
• Re-test when done