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 programmer knows how to identify such code
  - Good 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`)

```
claypool 54 fulham% /usr/bin/time saucer-shoot
2:24.04  elapsed (minutes:seconds)
13.26    user (seconds)
2.74     system (seconds)
11%      CPU
```

- 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?

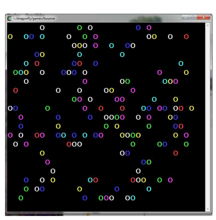
- A benchmark designed to estimate Dragonfly performance
  - Primarily dependent upon *number of objects* can support at target frame rate
- Assumes "standard" game creates many objects that move and interact
  - Bounce stresses Dragonfly by creating many objects that move and collide
- When Dragonfly 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=82GGLjyz3lY&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)

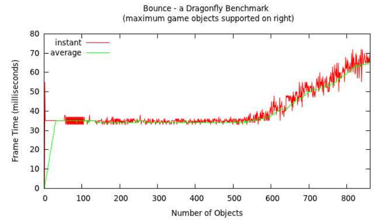
Bounce Data (1 of 2)

```
Bounce - a Dragonfly Benchmark (v1.0)
** Average maximum number of objects (bounce-mark): 1803 **
```

- 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 )
…
05:30:30 BOUNCE: Frame 1634 - 34 of 33 msec ( median is 33 )
05:30:30 BOUNCE: Frame 1635 - 34 of 33 msec ( median is 34 )
05:30:30 BOUNCE: Frame 1636 - 37 of 33 msec ( median is 34 )
05:30:30 BOUNCE: Frame 1637 - 33 of 33 msec ( median is 33 )
…
05:32:34 BOUNCE: Frame 1772 - 38 of 33 msec ( median is 36 )
05:32:34 BOUNCE: Frame 1773 - 39 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)



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 = estimated maximum reported by Bounce
  - M = number of moving objects
  - S = number of static (non-moving) objects
  - Need → M * (M + S) <= 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
  - 100x100 walls
  - Hero and up to 10 bad guys
  - Can Dragonfly support?
  - M = 11, S = 10000
    - → 11 * (11 + 10000) <= 500*500 ?
    - → 110,121 <= 250,000 (**yes**)

M * (M + S) <= B^2

- Say 10x bigger world. And bullets, up to 50 "in flight" during firefight
  - Can Dragonfly support?
  - M = 61, S = 100000
  - → 61 * (61 + 100000) <= 250000
  - → 6,103,721 <= 250,000 (**no**)
- What to do?
  - Tune code (more later)
  - Design differently
    - Don't spawn bad guys until Hero can see them
    - Make levels smaller (but have more of them)
    - Make sections of walls combined → multiple objects to one
    - 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

gprof

- 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)

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 84.54      2.27     2.27  6665307    0.00     0.00  partition
  9.33      2.53     0.25 54328749    0.00     0.00  swap
  2.99      2.61     0.08        1    0.08     2.61  quicksort
  2.61      2.68     0.07        1    0.07     0.07  fillArray
```

Explanations

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

Observations

- `swap()` called many times, but each fast
  - ✓ consumes only 9% of overall time
- `partition()` called many times, fast
  - ✓ consumes 85% of overall time

Conclusions

- Improve performance →make `partition()` faster
- Don't try to make fillArray() or `quicksort()` faster

Gprof – Call Graph Profile

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    2.68                 main [1]
                0.08    2.53       1/1           quicksort [2]
                0.07    0.00       1/1           fillArray [5]
-----------------------------------------------
                             13330614             quicksort [2]
                0.08    2.53       1/1           main [1]
[2]     97.4    0.08    2.53       1+13330614 quicksort [2]
                2.27    0.25 6665307/6665307     partition [3]
                             13330614             quicksort [2]
-----------------------------------------------
```

- 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

```
  %   cumulative   self
 time   seconds   seconds   calls    name
29.89      0.52     0.52  15986054  boxIntersectsBox(Box, Box)
16.09      0.80     0.28 303902763  Position::~Position()
12.07      1.01     0.21 206448124  Box::getCorner()
 6.90      1.13     0.12  15986054  getWorldBox(GameObject*)
 6.32      1.24     0.11     73253  WorldManager::isCollision(GameObject *, Pos)
 5.17      1.33     0.09 127780828  Position::getY()
 4.02      1.40     0.07  31676507  Box::getVertical()
 2.87      1.45     0.05  16176361  ObjectListIterator::next()
 2.87      1.50     0.05     79953  ObjectList::ObjectList()
 1.72      1.53     0.03 127781828  Position::getX()
 1.72      1.56     0.03  48402042  Box::~Box()
 1.72      1.59     0.03  16059309  Box::setCorner(Position)
 1.72      1.62     0.03  16059307  Object::getBox()
 1.15      1.64     0.02  32122197  Object::getPosition()
 1.15      1.66     0.02  31674366  Box::getHorizontal()
 1.15      1.68     0.02  16064131  Position::setX(int)
 1.15      1.70     0.02  16060277  Position::setY(int)
 0.57      1.71     0.01  16266646  ObjectListIterator::isDone()
 0.57      1.72     0.01  16176361  ObjectListIterator:currentObject()
 0.57      1.73     0.01    186921  WorldManager::getInstance()
 0.57      1.74     0.01     73253  SceneGraph::solidGameObjects()
 0.00      1.74     0.00    361538  WorldManager::getView()
 0.00      1.74     0.00    312676  getVelocityStep(float&, float&)
 0.00      1.74     0.00    280748  Position::Position(int, int)
 0.00      1.74     0.00    179804  worldToView(Position)
 0.00      1.74     0.00    179804  GraphicsManager::drawCh()
```

Each is a *sample* taken every 0.01 seconds → 1319 samples (more later)

Example – Saucer Shoot

```
  %   cumulative   self
 time   seconds   seconds    calls   name
20.00      0.06     0.06 29179995  Position::~Position()
20.00      0.12     0.06  1265573  boxIntersectsBox(Box, Box)
16.67      0.17     0.05    29255  ObjectList::ObjectList()
10.00      0.20     0.03  1275432  getWorldBox(GameObject*)
 6.67      0.22     0.02 16743168  Box::getCorner()
 3.33      0.23     0.01 12840584  Position::getX()
 3.33      0.24     0.01  2247840  Box::getVertical()
 3.33      0.25     0.01  1297989  Object::getBox()
 3.33      0.26     0.01   755020  worldToView(Position)
 3.33      0.27     0.01    58662  ObjectListIterator::~ObjectListIterator()
 3.33      0.28     0.01    27206  WorldManager::moveGameObject()
 3.33      0.29     0.01    22555  WorldManager::isCollision()
 3.33      0.30     0.01     2025  WorldManager::draw()
 0.00      0.30     0.00 12376988  Position::getY()
```

Example – Bounce (call graph)

Total time in function or children (percent)

Total time in function or children (percent)

Time in function

Function name

Number of times called

```
[1]    100.0    0.00    2.12                main [1]
                0.00    2.12       1/1          GameManager::run() [3]
                0.00    0.00       1/1          GameManager::startUp() [40]
                0.00    0.00       1/1          Bouncer::Bouncer() [41]
                0.00    0.00       1/1          GameManager::shutDown() [46]
                0.00    0.00       1/2          GameManager::getInstance() [107]
-----------------------------------------------
                0.00    2.12       1/1          GameManager::run() [3]
[2]    100.0    0.00    2.12       1            GameManager::run(int) [2]
                0.00    2.08     975/975        WorldManager::update() [4]
                0.01    0.03     976/976        WorldManager::draw() [18]
                0.00    0.00       1/162708     WorldManager::getInstance() [42]
                0.00    0.00    1950/2925       Clock::delta() [74]
                0.00    0.00     976/976        GraphicsManager::swapBuffers() [88]
                0.00    0.00     975/975        InputManager::getInput() [91]
                0.00    0.00     138/1132       LogManager::writeLog(char const*, ...) [80]
                0.00    0.00       1/159811     GraphicsManager::getInstance() [56]
                0.00    0.00       1/3          InputManager::getInstance() [106]
                0.00    0.00       1/1610       LogManager::getInstance() [76]
                0.00    0.00       1/2          Clock::Clock() [110]
```

Additional Options

- '-A' to annotate code

```
366 -> int Sprite::getHeight() {
          return height;
       }

  6 -> void Sprite::setHeight(int new_height) {
          height = new_height;
       }

5300 -> int Sprite::getFrameCount() {
          return frame_count;
       }
```

- '-l' to profile by lines, not functions

Using Profiling (1 of 2)

- Determine where to optimize
  - Pick the bottleneck and make more efficient
  - This provides most "bang for the buck" (buck = time, often!)
- E.g.
  - Program takes 10 seconds to execute
  - Function `A()` takes 10% of the time
  - Make `A()` 90% more efficient!
  - How long does program take? → 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? → 5.5 seconds
- Bottleneck will then move → this is 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 `boxesIntersect()`
  - Then `boxIntersectsBox()` called less often would be small
  - But, `distanceObjects()` now huge!
  - Is this better? Could be → but only if distance test "cheaper" than intersection test
- Can't make code more efficient (e.g. library)? → 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 `GameObjectListItrtr::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

```
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 the 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