PROGRAMMING FOR PERFORMANCE
A Real-time System responds in a (timely) predictable way to unpredictable external stimuli arrivals.

A system is a real-time system when it can support the execution of applications with time constraints on that execution.

- Dedicated Systems Encyclopedia
Hard vs. Soft Real-time

- Two classifications:
  - “Hard”
    - any lateness of results unacceptable
      - Aircraft fly-by-wire
      - Automotive ABS
      - UPS
  - “Soft”
    - rising cost of lateness
      - ATM
      - Google
Video Games

- Video games have a variety of real-time systems
- Sound has hard constraints
  - Hardware consumes data at 44 KHz (stereo)
  - any amount of dropout is catastrophic
    - Can’t extrapolate to fill in the missing sound data
- Sound also has soft constraints
  - Sound must correlate with visual or input events
    - Latency > 40ms = dissociation
Video Games cont’d

• Rendering is a soft real-time system
  – 60 fps (frames per second) is ideal
  – 20 fps is okay
  – 5 fps is no fun at all
  – Input to visual latency should be < 50 ms
  – Some games are more sensitive (FPS, fighters)

• Training simulators are usually hard real-time
Measures - Latency

• Total time for an operation to take place
• Example:
  – time from initiation of CD seek to time head is placed over correct track: 1-200ms
• Typical latency upper bounds:
  – Display: 50ms
  – Sound: 40ms
  – Input: 30ms
  – Network: 300ms
Measures - Throughput

- Amount of operations that can be completed in a given time
- Example:
  - amount of data that can be read from a CD in one second: 2 MB
- Throughput bounds are application specific
Latency and Throughput Together

- Latency and throughput must be considered together when measuring performance
- Often one can be traded for another
  - CPU example: deep pipelines to increase clock rate
  - GPU example: triangle throughput vs. state change latency
  - Don’t concentrate solely on one to the detriment of another
    - e.g. adding display latency can increase the frame rate of the render, but it may make the controls feel sluggish
- Ideal: 0 latency, infinite throughput
What Should You Measure?

- **Best case**
  - Good for selling things, but not useful by itself

- **Worst case**
  - Must use this to ensure application always performs better than lower-bounds

- **Average**
  - Good indicator, but can be misleading if the performance can spike

- **Overall**
  - Record per frame rate over many frames, plot the results in a spreadsheet to look for trouble areas or areas of high visibility
    - Helps if gameplay session can be repeatable (journaling)

- **Easiest situation: Best=Worst=Average**
  - Achievable, but requires discipline
  - Makes everything hard real-time
Balanced Performance

• You want to achieve balanced performance
• Player experience is smooth, responsive, consistent
• Smooth – throughput handles workload
• Responsive – always achieve better than maximum allowable latency
• Consistent – no peaks or valleys
  – a solid 30 fps is more playable than 5-to-60
Optimisation

• We want to maximize throughput and minimize latency
• We have determined minimum performance bounds for each system in the game
• But:
  – It isn’t always easy to hit the targets on the first try
  – The maximum performance is unknown and very worthwhile goal
• It’s time to optimise
Why Optimise?

• Games have stringent performance constraints
  – display rate
  – sound latency
  – controller response
  – load time
  – network latency

• A laggy, slow, choppy game is not fun
  – online Quake with a 1000 ms ping
Why Optimise?

• Content demands outstrip capabilities of code
  – designers always want more than you can provide
  – puts positive pressure on programmer to improve system

• Hardware remains fixed, quality bar is rising
  – must out-do previous title, competition
Why Optimise?

• Appeal to a wider spectrum of hardware
  – A game that only works on today’s state-of-the-art hardware may shut out a large portion of your audience (and sales)

• Facilitates greater gameplay experience
  – richer content
  – faster, tighter controls

• Fun & challenging
  – optimising promotes understanding
When not to Optimise

• Optimised code has drawbacks
  – takes more time to develop
    • > 10X for assembly vs. C
      – compilers can and will beat you some (most?) of the time
  – maintainability / readability suffers
  – portability sacrificed
  – hard to debug
  – easy to be fooled
    • wild goose chases
    • lots of effort for small gain
  – lost opportunity

• Choose your battles carefully!
Common Wisdom: The 90/10 Rule

- 10% of the code takes 90% of the time
  - When you find the 10% you can dramatically increase your speed just by fixing it
  - The speed of most of the code doesn't matter, so you don't need to worry about it
    - And can waste a lot of time optimizing things that don't matter
    - You need to make sure that you find the right 10%
      - This is where good profiling techniques are essential

- But...
Death of a Thousand Cuts

- Sometimes the 90/10 rule doesn't hold
- Pervasive architectural problems and inefficient techniques can hide performance issues where you can't find them
  - Language features and hardware quirks are common culprits here, since they are resistant to many profiling techniques
  - So are over-designed and needlessly abstract systems
- The only way to fight against this is to be aware of the costs of design choices up front
  - You can't generally find and fix these problems once things are nearing completion
Break
How to Optimise

• Three steps:
  1. Find performance bottlenecks
  2. Fix them
  3. Repeat
How to Optimise

• Good optimisation is a combination of knowledge, intuition and measurement
• From Michael Abrash's, "Zen of Code Optimization":

1. Have an overall understanding of the problem to be solved.
2. Carefully consider algorithms and data structures.
3. Understand how the compiler translates your code, and how the computer executes it.
4. Identify performance bottlenecks.
5. Eliminate them using the appropriate level of optimisation.
Understanding the Problem

• Some questions to ask:
  – How long do I have to work on this?
  – Has this been solved before? (yes!)
    • What are the differences?
  – What are the characteristics of the data?
    • Are there special cases?
    • Where is the coherency?
  – What can be computed offline?
  – Is there a simpler problem lurking within?
  – Can the hardware help me?

• Discuss the problem with your colleagues
• Don’t start coding yet
Algorithms and Data Structures

- The most important aspect of fast code
  - a bubble-sort in hand-tweaked assembly is still slow
  - have a toolkit of good general purpose algorithms developed by smart people
    - Quicksort, A*, hashing, etc.
- “Big Oh” analysis is useful
  - in practice, we are less formal about it
  - Remember that ‘n’ and ‘c’ matter in real code!
  - we care more about the particularities of compilers and hardware
Finding Bottlenecks

• Intuition (guessing)
  – helps if you are familiar with the algorithm/code
  – don’t trust it alone though!
    • can be misleading, or just plain wrong

• Profiling
  – measure performance to find hot spots
  – many tools available:
    • algorithm analysis
    • counters
    • timers
    • profiler programs
Profiling: Instrumentation

- Instrumentation should be built into the game:
  - frame rate counter
  - rendering statistics
    - triangle count, textures used, etc.
  - network ping time
  - collision tests per frame
  - anything else that is interesting
Profiling: Isolation

- Isolate components in a running game to determine their contribution to the frame rate:
  - disable parts of the renderer
    - world
    - characters
    - special effects
  - turn off sound
  - turn off collision
- May be misleading if components interact
Profiling: Third-party Tools

- Instrumentation profiler
  - e.g. Metrowerks Codewarrior CATS
  - places code at the beginning and end of every function to record timings
  - gives accurate tallies of function frequency, total function time, etc.
  - records call graph
  - Intrusive since code is changed
    - can affect accuracy of timings
Profiling: Third-party Tools

• Sampling profiler
  – e.g. Intel’s Vtune
  – at regular intervals (e.g. 1 ms), the current PC is recorded
  – later, the samples can be tallied and cross-referenced with the source code
  – fast, non-obtrusive
  – works on non-instrumented code
    • Operating system, video drivers
  – less accurate: events can be missed
  – no call graph available
Compilers

• Speed is lost in the translation of C/C++ to machine code
  – compilers are sophisticated but dumb
  – narrow view of program at any given time
  – no concept of “the problem”

• Don’t waste time try to beat the compiler on its strengths
Compilers

• Be aware of the costs of language features
  – Don't get paranoid though, C++ is plenty fast enough for games, as long as you are aware of its limitations

• Cost of C++ features
  – We'll talk about this in the next lecture
Hardware

• Modern computers are characterised by:
  – fast CPU
  – deep pipelines
  – slow memory

• Some good algorithms perform poorly in practice
  – poor cache locality
  – unpredictable branching
  – high memory usage

• Don’t be afraid to try brute-force solutions
  – Can make the code more transparent to CPU and compilers
Caches

- Cache friendly algorithms are incredibly important
- Try computing information instead of storing it
- Consider this:
  - Xbox360 CPU @ 3.2 GHz
  - L2 miss on Xbox360 is ~500 cycles (give or take)
  - If you can compute the value in 100 cycles that would have been read from main memory, you've gained 5X
  - There are many opportunities to do this
    - e.g. Store a transform as a translation & quaternion (7 floats) instead of a matrix (16 floats). Generate the full matrix on demand.

- Favour small data
- Favour coherent memory access patterns
- Avoid cache pollution
Optimisation

• Constantly challenge assumptions
  – profile, profile, profile

• Be creative and a little bit crazy
  – optimisation is very non-linear
  – it takes a big bag-of-tricks to be effective
  – practice!

• Know how deep to go
  – hand-tweaked assembly can beat the compiler by a factor of 100 in some cases
  – this takes a clear understanding of all factors to succeed
  – spending days only to have the compiler beat you is no fun
Techniques
Technique: Precomputation

- Do the tough calculations offline
  - static lighting (light maps)
  - potentially visible set (PVS) calculation
  - display lists
  - just-in-time compilation

- This is the classic speed/memory trade-off
Technique: Caching

"All programming is an exercise in caching."

- Terje Mathisen

- Take advantage of coherency by storing frequently used results for quick retrieval
- This technique pops up everywhere
- Enormous amounts of literature out there
Technique: Lazy Evaluation

- Defer expensive calculation until result is required
  - if you are lucky, the result isn’t needed at all
- e.g. Copy-on-write
  - operating systems
    - instances of a process share the same physical memory until one modifies a given page
Technique: Early Out

• Perform a simple test to avoid a costly operation
  
  ```cpp
  if( OnScreen(object.BoundingSphere()) )
  {
    object.Draw();
  }
  ```

• Make sure the extra test saves time!
  - If the early out test fails most of the time, then it’s just overhead
Technique: Approximation

- Trade speed for accuracy
  - e.g. Gouraud shading
  - be aware of error bounds
- Restrict the range of inputs
  - often opens the way for precomputation
- Interpolate
  - e.g. Gouraud shading
Technique: Divide and Conquer

• Break a problem into smaller sub-problems and solve each independently
  – binary search
  – quicksort
  – BSP trees
Technique: Strength Reduction

- Replace a costly operations with equivalent cheap operations
  
  \[
  a = a / 16; \quad // \text{divide} \ (\approx 40 \ \text{cycles})
  \]
  
  \[
  a = a >> 4; \quad // \text{shift} \ (\approx 1 \ \text{cycle})
  \]

- Compilers are very good at this
  - all modern compilers will perform instruction level strength reduction
  - when using assembly, you have to do it yourself
To the Metal: Assembly

• The last word in optimisation
  – unless you know how to build your own chips
• Use instructions the compiler doesn’t
  – SIMD, conditional writes, bit rotates, etc.
• Use Concepts that can’t be efficiently expressed in C/C++
• Not needed as much any more on the PC
  – still alive on the consoles
  – many instruction sets to learn
    • X86, MMX, SSE (PC/Xbox)
    • MIPS, VU (Playstation2)
    • PowerPC (GameCube)
Other Low-level Techniques

- Parallelism
- Inlining
- Pipelining
- Loop unrolling
- Coiled loops
- Jump tables
- Prefetching
- Code generation
- SIMD
- Cache locking
- DMA
- Write gathering
- Self-modifying code
- Floating point bit manipulation
Summary

• Practice makes perfect
• Understand the fundamental performance characteristics of the systems you are implementing
• Develop a repertoire of performance friendly techniques
• Profile relentlessly
  – Speculation can be dangerous
• Choose efficient, transparent algorithms
  – But remember that brute force can work well
• Become familiar with your compiler and hardware
• Know when to pull out all the stops
• Games have no bounds when it comes to desired performance
Quotes

Rules of Optimisation

Rule 1: Don’t do it.

Rule 2 (experts only): Don’t do it yet.

- M.A Jackson

“…premature optimisation is the root of all evil.”

- Donald Knuth