Programming for Performance

Textbook Definition of Real-time

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

Real time systems

- Games are mostly real time systems, though with lower costs of failure than most (aircraft fly-by-wire, pacemakers, etc.)

- “Hard”
  - Any lateness of results unacceptable
- “Firm”
  - Occasional lateness is not a total system failure
    - Could be significant quality degradation
    - Results cannot be used past deadline
- “Soft”
  - Rising cost of lateness
    - Quality degrades the later you get

Real Time Systems in Video Games

- Video games have a variety of real-time systems
  - No system in video games are hard real time
  - Failures obviously aren't as bad as in many real-time systems
- Sound has firm constraints
  - Hardware consumes data at 44 KHz (stereo)
  - Any amount of dropout is very bad
  - Can't extrapolate to fill in the missing sound data
- Sound also has soft constraints
  - Sound must correlate with visual or input events

Real Time Systems in Video Games

- Rendering is a soft real-time system
  - 60 fps (frames per second) or higher is ideal
  - 20 fps is okay sometimes
  - 5 fps is no fun at all
  - Some games are more sensitive (FPS, fighters, VR)

Characterizing performance

- Four important measures
  - Latency (individual operation)
  - Throughput (individual operation)
  - Framerate
  - CPU/GPU utilization

Latency

- Total time for an operation to take place
- Example:
  - Time from initiation of Blu-ray read to time the head is placed over the correct track: ~130 ms
- When latency is high, systems need to be asynchronous
- Operations off-CPU often have very high latency:
  - Display, sound, input 10-50ms
  - Disc storage: 20-150 ms
  - Network: 100+ms
- Some latency elements are outside our control
  - Wireless controllers, wireless headphones, motion smoothing on TVs
  - Dissociation if latency get’s too high

Throughput

- Amount of operations that can be completed in a given time
- Example:
  - Most standard computing performance measures (TFLOPS, etc)
  - Amount of data that can be read from an Blu-ray in one second: ~50 MB
  - Vertex or pixel processing rate

Latency and Throughput Together

- Latency and throughput must be considered together when measuring performance
  - Is a hard drive 2x faster than a Blu-ray, or 10x?
- Often one can be traded for another
  - CPU example: deep pipelines to increase throughput
  - 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 feels sluggish
- Danger: throughput is more important than latency for most non game applications, so hardware is optimized heavily for that
  - When trading higher latency for improved throughput, there is usually a lurking catastrophic failure case
    - Branch misprediction, cache stall, pipeline flush

Framerate

- Total time from completion of one frame to completion of the next
- Good general measure of performance
- Often expressed as frames-per-second (60 fps) or as milliseconds per frame (i.e. 16 ms)

Utilization

- Because systems are asynchronous, and may have external constraints (e.g. vsync) different systems may be running for different portions of frame
- Game where CPU is running flat out for 30ms but GPU is only running for 10ms has ‘worse’ performance than one where both are running for 30 ms
  - You are leaving quality on the table, could get either better performance or more stuff by balancing better
- Also applies to multi-core
  - Want to balance utilization of cores as well as possible

What Should You Measure?

- Best case
  - Good for selling things, but not useful for optimisation
- 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)
  - Achievable, but requires discipline
  - Makes everything hard real-time

Balanced Performance

- Player experience is balanced when it is:
  - 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 Criteria

- Games have stringent performance constraints
  - Display rate
  - Sound latency
  - Controller response
  - Load time
  - Network latency
- A laggy, slow, choppy game is not fun
  - Online FPS with a 1000 ms ping
- Hardware constraints
  - Memory optimisation

Optimisation Pressures

- Content demands outstrip capabilities of code
  - Designers and artists 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
- Games have much stronger optimisation pressure then most software
  - More “real-time” systems often have more constrained scope
  - More complex software often has softer constraints, or ability to solve the problem by running it on better hardware

Why Optimise?

- Appeal to a wider spectrum of hardware (PC)
  - 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 better gameplay experience
  - Richer content
  - Faster, tighter controls
  - Higher game reviews
- Fun & challenging
  - Optimising promotes understanding

When not to Optimise

- Optimised code has drawbacks
  - Takes more time to develop
    - Assembly takes more than 10 times as long as C++, but isn't 10x faster
  - 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
  - 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 tools and techniques are essential
- But...

Death by 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

How to Optimise

- Three steps:
  - Find performance bottlenecks
  - Fix them
  - Repeat

How to Optimise

- Good optimisation is a combination of knowledge, intuition and measurement
- From Michael Abrash's, “Zen of Code Optimization”:
  - Have an overall understanding of the problem to be solved
  - Carefully consider algorithms and data structures
  - Understand how the compiler translates your code, and how the computer executes it
  - Identify performance bottlenecks
  - 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 O” 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 exhibits some quantum uncertainty. Can’t always observe without affecting performance.

Counters and Metrics

- Various counters and metrics should be built into the game:
  - Frame rate counter
  - Rendering statistics
    - Triangle count, textures used, etc.
  - Memory used per pool
  - Network ping time
  - Collision tests per frame
  - Anything else that is interesting

Isolation Profiling

- 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
- Being able to do this easily is an example of good architecture paying off

Instrumented Profiling

- Instrumentation profiler
  - 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

Sampled Profiling

- Sampling profiler
  - At regular intervals (e.g. 1 ms), the current program counter is recorded
  - Later, the samples can be tallied and cross-referenced with the source code
  - Fast, non-obtrusive
  - Works on non-instrumented code
- Downsides
  - No call graph available
  - Less accurate: events can be missed
  - Operating system, video drivers are not visible
- One Weird Trick: The single sample profiler

System Trace

- Specialized tool for catching certain types of things that other forms of profiling can’t
- Capture various kinds of system events
  - System library calls
  - Context switches or other thread events
  - OpenGL calls
- Generate some visualization of the trace
  - Thread map
  - Replay graphics driver calls to generate detailed profiling info

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 trying to beat the compiler on its strengths
  - Compiler will optimize how you are doing something, you need to optimize WHAT you are doing
- Understand the optimization options of the compiler
- Be aware of the costs of language features
  - Don't get paranoid though, a virtual function call per vertex in your mesh would be bad, a virtual function call per object even in a 1000 object world is nothing.

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:
  - Most modern CPUs have L2 cache miss latencies in the range of 100-300 cycles.
  - If you can compute the value in 50 cycles that would have been read from main memory, you've gained 2-6X performance
  - 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 Mantra

- 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

Multithreading

- All hardware we care about is multithreaded
  - No one is even shipping single core phones any more
- Multithreading is probably the most important optimization technique right now
  - If you properly multithread a game, and you have 4 cores, you could quadruple the performance
- A couple of different approaches
  - Heavyweight threads
  - Job based
  - Local optimization (OpenMP, OpenCL)
- Lots of new bugs though
  - Deadlocks, race conditions, memory stompage, etc
  - Need a strong safety and debug harness

Pre-computation

- Do the tough calculations offline
  - Static lighting (light maps, ambient occlusion maps, spherical harmonics)
  - Potentially visible set (PVS) calculation
- This is the classic speed/memory trade-off

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
    - CPU caches
    - HTTP caching in web browsers
- Radical example: one element inventory cache

Lazy Evaluation

- Defer expensive calculation until result is required
  - If you are lucky, the result isn't needed at all
- Examples
  - Store dirty flags
  - Copy-on-write
    - Instances of a process share the same physical memory until one modifies a given page

Data Organization

- Cache friendly data structures
  - Small == fast
  - Fit into cache line width
  - Walk linearly
- Array of structures vs structure of arrays

```
struct {
    float x. y, z;
    float dx, dy, dz;
    float age;
} particles [10];
```

```
struct {
    float x[10],  y[10],  z[10];
    float dx[10], dy[10], dz[10]
    float age[10];
} particles;
```

  - Better cache utilization if only touching certain fields
  - SoA is better for SIMD
- Separating hot and cold fields

Early Out

- Perform a simple test to avoid a costly operation

```
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

Approximation

- Trade accuracy for speed
  - Simulate gravity, but not collisions, for particles
  - Render at a lower resolution and scale up
  - Use Taylor Series or other mathematical approximations
    - Be aware of error bounds
  - Look-up tables
- Restrict the range of inputs
  - Often opens the way for pre-computation
- Interpolate
  - Calculate properties in vertex shader and interpolate, rather than calculating in pixel shader
  - Store animation keyframes and linearly interpolate, rather than calculating an animation curve at each point

Divide and Conquer

- Break a problem into smaller sub-problems and solve each independently
  - Binary search
  - Quicksort
  - BSP trees or other spatial hierarchy
- Particularly effective if computation of the pieces can be parallelized

Time-boxing

- Amortize expensive operations over multiple frames
  - Partially processed job queues
  - Work to fixed time budget
- Doesn’t necessarily improve average performance, but can improves worst case

Strength Reduction

- Replace costly operations with equivalent cheap operations

```
a = a / 16; // divide (≈40 cycles)
a = a >> 4; // shift (≈1 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

SIMD

- Operate on multiple (usually floating point) values in the same operation
- Useful for many graphics and audio-related operations
  - Vector and matrix operations
  - Particle systems
  - Skinning
  - SFX

Assembly

- The last word in (micro-)optimisation
  - Unless you know how to build your own chips
- Use instructions the compiler doesn't, or things that can't be expressed in C++
  - Conditional writes, bit rotates, cache prefetches, etc.
  - Different register preservation semantics
  - Jump tables
- Hardly ever used in practice any more

Other Low-level Techniques

- Inlining
- Pipelining
- Loop unrolling
- Coiled loops
- Code generation

GPU acceleration

- Operations that are SIMD friendly can often be moved to the GPU on modern hardware
  - GPU's are significantly faster at this sort of thing
- Lots of applications
  - Image processing
  - Portions of physics
  - Particle system updating
  - Nvidia DLSS
- Compute shaders

Summary

- Practice makes perfect
  - Understand the fundamental performance characteristics of the systems you are implementing
  - Develop a repertoire of performance friendly techniques
  - Profile relentlessly
  - Become familiar with your compiler and hardware
- Speculation can be dangerous
- Choose efficient, transparent algorithms
  - But remember that brute force can also work well
- 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