COMP 8551 Advanced Games Programming Techniques

Borna Noureddin, Ph.D.

British Columbia Institute of Technology

Software Optimization

Overview

- Optimization:
  - Overview
  - Design techniques
- Parallelization:
  - Partitioning
  - Profiling
  - General techniques

Memory optimization

Motivation

- Hero casts a spell: shimmer of sparkles bursts across screen. This calls for a particle system: to animate little sparkly graphics until they wink out of existence
- Single wave of wand could cause hundreds of particles to be spawned: system needs to create them very quickly. More importantly, need to make sure creating/destroying particles does not cause memory fragmentation

Memory Fragmentation

- Game consoles and mobile devices are types of embedded systems.

- Games must run continuously for a long time without crashing or leaking memory – good memory managers not usually available). Memory fragmentation is deadly!

Memory Fragmentation

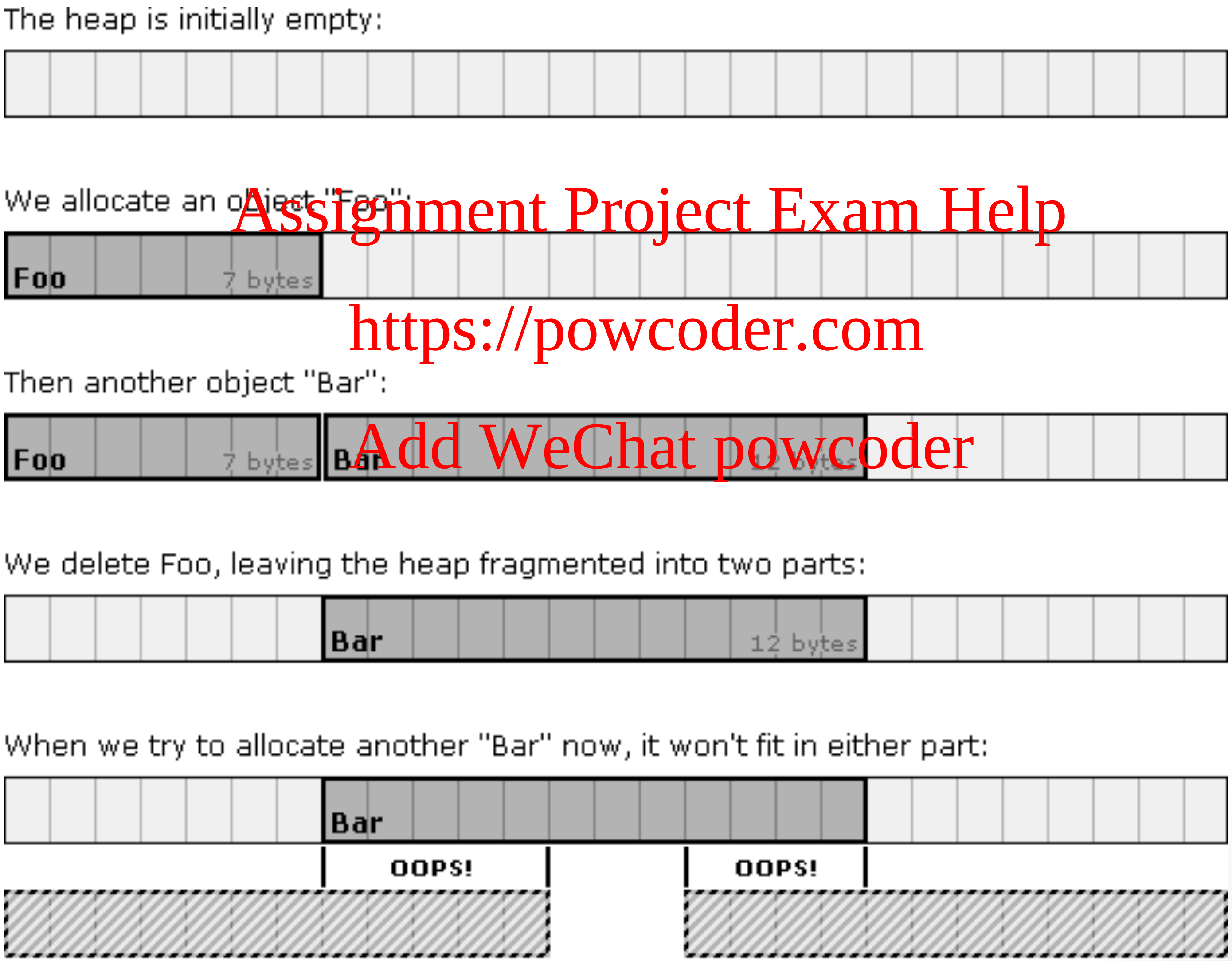
- Free space in heap broken into smaller regions instead of one large open block.

- Total memory available may be large, but largest contiguous memory might be small.

- E.g., 100 bytes free, but fragmented into 2x50-byte pieces with a little chunk of in-use memory between them.

  - If we try to allocate a 60-byte object, we will fail. No more sparklies onscreen!
- Think parallel parking on busy street!

Memory Fragmentation



The heap is initially empty:

We allocate an object "Foo":

Then another object "Bar":

We delete Foo, leaving the heap fragmented into two parts:

When we try to allocate another "Bar" now, it won't fit in either part:

Memory Fragmentation

- Even if fragmentation is infrequent, can still gradually reduce heap to an unusable foam of open holes and filled-in crevices, causing system (game) to crash.
- Console makers require games to pass “soak tests”: leave game running in demo mode for several days. Will not ship until no crashes (sometimes fail because of a rarely-occurring bug, but it’s usually creeping fragmentation or memory leakage.

Memory Fragmentation

- Because of fragmentation and because allocation may be slow, games must be very careful about when/how they manage memory.
- Simple solution: grab a big chunk of memory when the game starts and don't free it until the game ends.
  - Pain for systems where we need to create and destroy things while game is running.

Object Pools

- Object Pool gives us best of both worlds:
  - As far as memory manager is concerned, we are just allocating one big chunk of memory up front and not freeing it while the game is playing.
  - For users of the pool, we can freely allocate and deallocate objects to our heart’s content.
- More next term…

Optimization Techniques

- Trade-off between memory optimization (which we touched on previously) and speed optimization (which we discuss now)

- Used to be simple: unroll loops (loop code has a lot of overhead), lookup tables for trig functions – in some cases, these are still good, simple techniques (e.g., for small embedded systems)

- But now, for example, CPUs are good at handling loops, compilers often unroll them for you, and use tricks for fast computation of math functions (or use intrinsic instructions)

Optimization Techniques

- Two major overarching concepts: load balancing (e.g., no good having an idle GPU while pinning the CPU, or vice versa, or putting most of the load on one core) and dependency minimization (one processor should not be waiting for or be dependent on the output of another if at all possible)
- Now, more gains by looking at algorithms

Best (and most important) first step: profile your code!

Optimization Techniques

General ways to deal with bottlenecks:

- Avoid or remove the code altogether
- Reduce the number of times the code is called
- Change the algorithm, if possible
- Optimize that code manually (line by line) or rewrite code if needed: avoid/reduce I/O, memory allocations, loops
- Put the code in its own thread and/or run it on a separate core (but this is not always the best solution, surprisingly)

Optimization Techniques

Other techniques for incremental gains:

- Use basic type matched to CPU integer size
  - E.g., on 32-bit system, use [un]signed int's
  - bool, BYTE, char too small (inefficient memory access)
  - float slow computation, and double is even slower (although some instruction set extensions deal well with fixed floating point operations)
- Align structures and classes to align with 4 byte (8 byte on 64-bit systems) boundaries (e.g., rearrange member variables)
- Absolutely minimize memory allocation (e.g., object pools)

Optimization Techniques

Other techniques for incremental gains:

- Do not pass structures/objects to functions by value – use pointers!
    - By value means passing data on call stack instead of on heap (slower)
    - Passes copy of data, and then copies it back after returning from function: may even call constructor/destructor of class
    - Can always pass const reference if worried about inadvertently changing the value

Optimization Techniques

Other techniques for incremental gains:

- Inline small functions (not everything!)
- Comparing distances: compare square of distances (square root function is expensive)
- Avoid casting as much as possible
- Cache results (balance between keeping commonly used results around and keeping so many around that the search takes too long)

See especially http://www.gamasutra.com/view/feature/1879/the_top_10_myths_of_video_game_.php *for discussion of gamedev-specific optimization techniques*

Parallelization

Parallelization vs. optimization

- Optimization is a generic term used to describe how to make your code more efficient (faster, use less memory or other resources, etc.)
- Parallelism specifically seeks to perform multiple operations in a single step (e.g., clock cycle, iteration of render loop, etc.)
- In both cases, there is a cost-benefit tradeoff: it's not always worth spending days to get a 2% improvement

Parallelization

- Candidates for parallelization:
  - Rule of thumb: any operation that takes longer than a minute
  - Not good candidates: screen refresh, email, web browsing
  - Good candidates: large image conversions, simulations and analyses, rendering complex 3D graphics, video editing, large scale search/traversal
- Partition size: minimize time spent communicating among partitions relative to time spent in computation

Parallelization

- Word of caution: get the code working well and bug-free before attempting either optimization or parallelization
- Caveat: design your code and algorithms with efficiency in mind (easier to do design optimization than code optimization), including thread-safe code
- Be especially vigilant about code that could cause memory corruption or leaks: these are difficult to hunt down in regular code, but near impossible with optimized or parallelized code

Parallelization

- Profile your code before all else!
  - How much time is spent executing a particular section of code?
  - How many times is a particular section of code executed typically?
  - How many I/O operations does a particular section of code perform?
  - How many memory operations does a particular section of code perform?

Parallelization

- If your code seems to run slow, but the CPU is not pinned, optimization will not be the main issue; if there is more than one core and neither is pinned, then parallelization will not be the main issue either – the problem is more likely I/O or load balancing

- For real optimal performance in games, CPU and GPU should be matched (better to have moderate but balanced CPU-GPU, than high end CPU with low end GPU or vice versa)

Parallelization

Parallelization techniques:

- Loop unrolling

  Instead of

```
for (i=0; i<n; i++) { c[i] = a[i] * b[i]; }
```

  Use

```
// executed over “n” functional units
c[id] = a[id] * b[id];
```

- I/O: read small chunk, spawn thread to start processing it while waiting for next chunk

Parallelization

Parallelization techniques:

- Data decomposition
  - Operations on large chunks of data broken into units
  - Processing of each unit put in separate thread
- Functional decomposition
  - Independent functions (especially slow operations) put in separate threads

- Use packages such as OpenCL (http://www.khronos.org/opencl/), Cascade, CUDA (https://developer.nvidia.com/cuda-zone)

Case Study and Demos

Case study - designing game render software to taking advantage of hardware architecture:

http://software.intel.com/en-us/articles/optimizing-the-rendering-pipeline-of-animated-models-using-the-intel-streaming-simd-extensions/

Also see SMOKE

http://www.drdobbs.com/go-parallel/article/showArticle.jhtml;jsessionid=1JB0DRPY3VC3HQE1GHPSKH4ATMY32JVN?articleID=219400687&pgno=2

Additional Reading

http://www.raywenderlich.com/23037/how-to-use-instruments-in-xcode

http://www.khronos.org/opencl/

http://daugerresearch.com/vault/parallelization.shtml

http://www.drdobbs.com/go-parallel/article/showArticle.jhtml;jsessionid=D5CCBGSICL3J5QE1GHPSKH4ATMY32JVN?articleID=219400687&pgno=2

http://www.gamasutra.com/view/feature/1879/the_top_10_myths_of_video_game_.php

http://www.drdobbs.com/go-parallel/article/showArticle.jhtml;jsessionid=D5CCBGSICL3J5QE1GHPSKH4ATMY32JVN?articleID=227500610

Review

- Optimization:
  - Overview
  - Design techniques

- Parallelization:
  - Partitioning
  - Profiling
  - General techniques

END