Start-Up and Shut-Down

hannes@ru.is
Subsystem Start-up and Shut-Down

- Interdependence of subsystems
  → required order for start-up and shut-down
- Often subsystems are singleton classes (managers)
Static Initialization

class RenderManager {

public:

    RenderManager() {
        // Start up the manager
    }

    ~RenderManager() {
        // Shut down the manager
    }

};

static RenderManager gRenderManager;
Static Initialization

- Global and static objects are constructed before main() is called and destroyed after main() returns
- **BUT:** In unpredictable order!
Initialization

• Typical solution
  – The global singleton is a static variable inside a "get()" function, so initialized when retrieved for the first time

• Problems
  – "get" is not an obvious initialization function
  – Destruction is still arbitrary
Initialization

• Better Solution
  – EXPLICIT startup/shutdown
Explicit Initialization

class RenderManager {
    public:
        RenderManager() { // nothing }
        ~RenderManager() { // nothing }
        void startUp() { // initialize here }
        void shutDown() { // clean up here }
    
};

RenderManager gRenderManager;
int main(int argc, const char* argv) {
    gRenderManager.startUp(); ... }

Initialization

• Better Solution
  – EXPLICIT startup/shutdown

• Benefits
  – Simple
  – Explicit
  – Facilitates debugging
Memory Management

hannes@ru.is
RAM and Performance

• RAM use affects performance
  1. Dynamic allocation is slow
  2. Memory access patterns are important
Dynamic Allocation

• OS Heap Allocator is complex and deals with things like contention between threads. Context switching often required.
• Avoid as much as possible and never use this allocator in a tight loop!
• Games typically implement their own memory allocators.
Custom Memory Allocators

• STACK-BASED Allocators
  – Allocates a contiguous block that can grow
  – Only able to free by popping back to a marker

• DOUBLED-ENDED STACK Allocator
  – Two stacks, one from each end
  – One stack for “slowly” changing data (e.g. levels) and one for “faster” changing data (e.g. temps)
Custom Memory Allocators

• **POOL Allocators**
  – Only allocate same sized blocks
  – Free blocks kept in a single linked list
  – Fast manipulation

• **ALIGNED Allocators**
  – When allocators make sure that returned addresses are properly data aligned for the CPU
  – Typically allocate additional size of alignment and then adjust address upwards until it fits
Custom Memory Allocation

• SINGLE-FRAME Allocators
  – A single stack allocator that gets cleared at beginning of every render loop

• SOUBLE-BUFFERED Allocators
  – Two stack allocators that get swapped at beginning of every render loop, and then the new current one gets cleared
  – Data from previous frame still available

• Danger: Memory „destroyed“ instead of freed
Memory Fragmentation

hannes@ru.is
Memory Blocks

- Allocated memory blocks must be contiguous
- Problem is that after many allocations and freeing of memory, there will be free areas of various sizes, but perhaps none big enough to hold a single large memory block
- This is the problem of fragmentation
Avoiding Fragmentation

- Use
  - Stack allocator (always contiguous)
  - Pool allocator (always same block size)
Defragmentation

- Combine all the free memory into a contiguous block
- Could shift allocated blocks to lower addresses, letting the free memory „bubble up“
Defragmentation: Relocation

• Shifting allocated memory requires the relocation of pointers!
  – Need to find those pointers and change them
  – This can be hard

• Possible to use
  – Smart pointers: Classes that register themselves
  – Handles: Indecies into pointer tables
Defragmentation: Reduce cost

- When blocks are relatively small, it is easy to spread their defragmentation across many frames
- Possible to break up larger blocks
Cache Coherency

hannes@ru.is
Cache Levels

- CPU
- L1 Cache
- L2 Cache
- RAM

Speeds:
- Fast
- Slower
- Slowest
Types of Cache

- **Data Cache (D-Cache)**
  - Cache lines contain data that hopefully gets used next

- **Instruction Cache (I-Cache)**
  - Cache lines contain machine code that hopefully gets executed next
Avoid Misses

• Avoid D-Misses
  – Organize data contiguously
  – Keep it in small chunks (to fit in cache line)
• Avoid I-Misses
  – Keep high-performance machine code as small as possible
  – Avoid calling functions from high-performance code
  – Place other functions close by (translation units stay together in memory)