Memory & Game Content

Memory is precious

- Memory is precious, especially on simpler devices.
  - Even on PCs need to be cautious, using too much memory tends to drive you off a performance cliff
- For consoles, memory fragmentation can be a problem as well as memory exhaustion
- As with performance, memory optimization gives you space to improve your game with more content

Memory analysis

- We should account for three types of memory:
  - Code
    - Not going to cover in detail
    - Only thing you sometimes need to think about: statics and globals go in code memory
  - Stack
  - Heap (global/shared/dynamic memory)
- Let’s look at the latter two types in detail

Stack Memory

- Usually a specific portion of memory earmarked by the system
  - Per thread
  - The programmer might have a say in this. For example, set during thread allocation
- Things that go on stack
  - Per function overhead
  - Local variables in function
- Multiple stack frames on stack as functions call each other

```
void some function()
{
    int i[16]; //64 bytes on stack
}
```

Stack Memory

- Don’t blow the stack!

```
// Dangerous linked-list deletion!
void DeleteNode( Node* n )
{
    if ( n == NULL ) return;
    DeleteNode( n->mNext );
    delete n;
}
```

- This was an actual bug encountered in Prototype
- Replacing the above with a non-recursive while loop fixed the problem.
- Another example of risky stack usage:

```
SomeBigStruct temp[65536]; // !!??
```

Heap Memory

- Refers to dynamic memory available.
  - Allocated/freed via new/delete and malloc/free
- May be organized into a hierarchy of heaps, for budgeting purposes.
  - E.g. World art should not take up more than 1 GB.
  - Budget usually determined, enforced and tweaked by senior programmers on the game team.
  - If so, need to overload new/delete (or malloc/free)
- Sometimes use custom allocators for specific heaps

Alternate allocation strategies

- Pools
  - Create/delete fixed-size objects, up to a maximum.
  - Very efficient, but can only typically create a single type of object.
- Linear allocation:
  - Advances pointers as we allocate.
  - Doesn’t keep track of each allocation – frees everything together.
- Recycling
  - Don’t allocate, figure out ways to reuse objects

Loading Content

- Alright, what do we fill memory with?
  - Art assets
    - Typically pre-processed offline, optimized for both memory and performance.
    - Meshes, textures, lighting, animation, audio
  - Design assets
    - Behaviour trees, mission scripts, physics tuning data, prop definitions, spawn data, etc.
  - Game entities
    - Characters, props, vehicles, etc.
    - Often allocated dynamically, strong candidates for pooling or recycling
- Need to define data formats for loading these

Parsing Data

- Data formats can be categorized into:
  - Object serialization style formats: Simple structures with properties like floats, ints, strings, etc.
    - Usually somewhat generic, easy to add new types
    - Possibly mapped to game objects via some binding system
    - Can often exist in text or binary forms
  - Binary-only custom format: E.g. DDS textures, WAV audio files, etc.
  - Memory images
    - Intended to be used as is directly in memory, no parsing
    - Often somewhat platform-dependent

Text or Binary?

- For data exported from third-party software, binary might be the only option (e.g. DDS textures, WAV files)
- If exporting is done by our own software, text files are certainly easier to:
  - Look at, and
  - Check for differences between versions
- However, it's faster to load binary files, so you might want to convert the text files to binary through the asset pipeline
- For you: Use a text file format with an existing parser
  - JSON, YAML, XML (if you are a horrible monster)

Finding objects

- Once objects are loaded, we typically need to find them by name later.
- One Approach:
  - Store name in object data during load
  - Put it in a hash table
  - Look up by name

```
// Find an animation!
Name animName = “walk”;
Animation* anim = animationsInventory->Find<Animation>(animName);
```

- You: can load objects directly from disk by (file) name

Other Loading Considerations

- Media
  - DVD / Blue-rays are quite slow. Need lots of tricks to work around load contention
  - HDD are much faster but still too slow to fill gigabytes of RAM in time
  - Latest platforms use SSD (Flash memory)
- Concurrency
  - I/O should generally be asynchronous (can be done with async API to avoid you writing multithreaded code)
  - Can often make parsing asynchronous as well (that usually does mean your code is multithreaded)

Other Loading Considerations

- Pack files
  - Often want to have multiple files concatenated into a single large file
  - Simplifies installation
  - May speed up or simplify disk reads
  - Could use standard format (like zip)
  - Could compress as well.
- Memory mapping
  - Although consoles don’t generally support virtual memory, can still use MMU to memory map files from disk
  - May be faster, MMU has tricks not available to regular programmers

Hard-code or script?

- Sometimes need to decide whether to put something in code or in data. For example:
  - How should a monster react when you attack it?
  - What's the maximum speed for a car when you drive it?
- Hard-coding is sometimes quicker to "get it done"
- But, it's usually slower to iterate with code changes
- Remember: iteration is king!
  - Best solution: reload settings without restarting the game

Summary

- Memory allocation and data loading strategy are critical aspects of game performance
- For your project, focus on ease of iteration
  - Prefer text over binary assets
  - Prefer configuration files over hard-coding
  - Prefer hot-reloading of data