Chapter 3.4
Game Architecture

Overall Architecture

- The code for modern games is highly complex
- With code bases exceeding a million lines of code, a well-defined architecture is essential for:
  - Adhering to deadlines
  - Managing personal

Overall Architecture

- Main structure
  - Game-specific code
  - Game-engine code
  - Both types of code are often split into modules, which can be static libraries, dynamic link libraries (DLLs), or just subdirectories

Overall Architecture

- Coupling of code is concern
- Architecture types
  - Ad-hoc (everything accesses everything)
  - Modular
  - DAG (directed acyclic graph)
  - Layered

Overall Architecture

- Options for integrating tools into the architecture
  - Separate code bases (if there's no need to share functionality)
    - XML data sheets (SVG, X3D)
  - Partial use of game-engine functionality
    - Level Editors
  - Full integration
    - Play, pause, edit, resume

Overview: Initialization/Shutdown

- The initialization step prepares everything that is necessary to start a part of the game

- The shutdown step undoes everything the initialization step did, but in reverse order

FrontEnd Initialization/Shutdown

```
{
        FrontEnd frontEnd; // Facade
        frontEnd.Initialize();
        frontEnd.loop();
        frontEnd.shutdown();
}
```

Memory Leaks (I)

```
char *a = malloc(128*sizeof(char));
char *b = malloc(128*sizeof(char));

// -------------------------------------------
b = a;   // oops!
// ----------------------------------------------------------

free(a);
free(b); // won't work
```

RIAA

- Resource Acquisition Is Initialization
- A useful rule to minimalize mismatch errors in the initialization and shutdown steps
- Means that creating an object acquires and initializes all the necessary resources, and destroying it destroys and shuts down all those resources
- RIAA is helpful for managing memory but can still result in leaks.

FrontEnd Initialization/Shutdown

```
try {
        FrontEnd frontEnd; // Facade
        frontEnd.loop();
}
catch (…) {
   // handle problems here
}

// Destructor ~FrontEnd called before main() ends
```

Memory Leaks (II)

```
BaseClass* obj_ptr = new DerivedClass;
// Allowed due to polymorphism.
...
delete obj_ptr;
// calls ~Parent() destructor NOT ~Child()
```

Optimizations

- Fast shutdown (level change)
  - Creating and destroying objects is costly
  - Using “memory pool” is faster
- Warm reboot
  - Some resources can't be recovered until main() ends.
  - Restarting machine reinitializes stack
  - Hand-Held gaming devices.

Overview:
Main Game Loop

- Games are driven by a game loop that performs a series of tasks every frame
- Game may consist of single main loop
- Some games have separate loops for the front and and the game itself
    - Multi-threading may also entail multiple loops

Frames

- Don't think of a frame as a picture
- A “frame” is a logical unit
  - Game frame (time step)
  - Graphics or rendering frame (screen)
- 30 fps (30hz) flicker test
  - Also frame rate of Halo 3
- Some LCD max of 60fps

Overview:
Main Game Loop

- Tasks
  - Handling time (time stamp, time elapsed)
  - Gathering player input
  - Networking
  - Simulation
  - Collision detection and response
  - Object updates
  - Rendering
  - Other miscellaneous tasks

Overview:
Main Game Loop

- Structure
  - Hard-coded loops
  - Multiple game loops
    - For each major game state
    - Front-End, Main, etc.
    - For major threads
  - Consider steps as tasks to be iterated through

```
while ( !IsDone() ) {
    UpdateTime();
    GetInput();
    GetNetworkMessages();
    SimulateWorld();
    CollisionStep();
    UpdateObjects();
    RenderWorld(); // the 'graphics' part
    MiscTasks();
}
```

Overview:
Main Game Loop

- Execution order
  - Most of the time it doesn't matter
  - In some situations, execution order is important
  - Can help keep player interaction seamless
  - Can maximize parallelism
  - Exact ordering depends on hardware

```
while ( !IsDone() ) {
        Tasks::iterator it = m_tasks.begin();
        for (; it != m_tasks.end(); it ++)
        {
                Task* task = *it;
                it -> update();
        }
}
```

Decoupling Rendering

- Can decouple the rendering step from simulation and update steps
  - 30fps game loop
  - 100fps graphics capability
- Results in higher frame rate, smoother animation, and greater responsiveness
- Implementation is tricky and can be error-prone

```
while ( !IsDone() ) {

    UpdateTime();

    if(TimetoRunSimulation())
        RunSimulation();

    if(SimulationNotRun())
        InterpolateState();

    RenderWorld();

}
```

Multi-threading

- Allowing multiple operations to happen in parallel. (requires sharing of data code)
- Consider that a GPU is a separate processor.
- Potential 'threads'/'concurrent processes'
  - Physics
  - Collision calculations
  - Animation processing
  - Agent updates
  - AI pathfinding

Game Entities

- What are game entities?
  - Basically anything in a game world that can be interacted with
  - More precisely, a self-contained piece of logical interactive content
    - Enemy, bullet, fire, menu button
  - Only things we will interact with should become game entities

Game Entities - Organization

- Basic Choices
  - Simple list
  - Multiple databases
  - Logical tree
  - Spatial database
- Multiple options are often necessary
  - Single Master List
  - Other data structures use pointers to objects in master list
  - Observer model maintains consistency

Game Entities - Updating

- Updating each entity once per frame can be too expensive
- Can use a tree structure to impose a hierarchy for updating
- Can use a priority queue to decide which entities to update every frame
- Different operations can use different data structures.

Game Entities - Creation

- Basic object factories
  - Enum list, switch statement
  - Returns point to object
- Extensible object factories
  - Allows registration of new types of objects
  - Using automatic registration
  - Using explicit registration

Game Entities – Level Instantiation

- Loading a level involves loading both assets and the game state
- It is necessary to create the game entities and set the correct state for them
- Level/State often stored in file
- Using instance data vs. template data
  - Not all object data will be different
  - “Flyweight” concept: one copy of duplicate data

Game Entities - Identification

- Strings
- Pointers
- Unique IDs or handles (UID)
  - Maps a short unique name to a pointer
    - Hashmap, Key → bucket
  - Hashmap usually managed by an object

Game Entities - Communication

- Simplest method is function calls
  - How to know what functions supported?
- Many games use a full messaging system
  - Usually a dedicated object (singleton)
- Need to be careful about passing and allocating messages
  - Messages need to be small, patterned
  - Classes (and union) are useful for establishing message types
  - 1000's of messages per frame, new(), delete() overhead
  - Memory pools key; overriding new/delete to refer to fixed set of locations: "post office boxes"

Exercises

- Highly recommend you look at questions 2,3 and 4 in 2nd edition book.
    - 2. C++ memory leak utilities
    - 3. Simple game loop, separate rendering loop (print messages for each part).
    - 4. Task loop that allows tasks to register.

The End