Lecture 13

Concurrency & Multithreading
Games are Naturally Multithreaded

- The core game loop is **time constrained**
  - Frame rate sets a budget of how much you can do
  - Exceeding that budget causes frame rate drops

- Sometimes we need an extra thread to …
  - Offload tasks that *block* drawing (**asset loading**)
  - Offload tasks that *slow* drawing (**pathfinding**)
  - Execute tasks *decoupled* from drawing (**audio**)

- Part of architecture spec: **computation model**
Multithreading in CUGL

- CUGL has **three** primary threads
  - The *Application*, or main graphics thread
  - The *AssetManager* thread, for loading assets
  - The *AudioEngine* thread, for audio playback
  - Note that only *Application* is required

- Also has tools for making your own threads
  - Most are built on top of C++ and `std::thread`
  - But there are some unique features too
Multithreading in CUGL

- CUGL has **three** primary threads
  - The **Application**, or main graphics thread
  - The **AssetManager** thread, for loading assets
  - The **AudioEngine** thread, for audio playback
- Note that only **Application** is required
- Also has tools for making your own threads
- Most are built on top of C++ and std::thread
- But there are some unique features too
Recall: The Application Thread

60 times/s
= 16.7 ms

Update
- Receive player input
- Process player actions
- Process NPC actions
- Interactions (e.g. physics)

Draw
- Cull non-visible objects
- Transform visible objects
- Draw to backing buffer
- Display backing buffer

Multithreading
Recall: The AssetManager Thread

- Works as a dictionary
  - Each asset given a key
  - Can access asset by key
  - But templated by type
Recall: The AssetManager Thread

Game Thread

Asset Thread

Specify Asset

Update

Asset Manager

Exactly how does this work?

Works as a dictionary

• Each asset given a key
• Can access asset by key
• But templated by type
Asset Loading Consists of Tasks

Task 1  Task 2  Task 3  Task 4

Load Font "Times.ttf"
Load Image "smile.png"
Load Sound "music.ogg"
Load Widget "menu.json"
Ideally, Each One is a Thread

Task 1  Task 2  Task 3  Task 4

Load Font "Times.ttf"  Load Image "smile.png"  Load Sound "music.ogg"  Load Widget "menu.json"

Thread 1  Thread 2  Thread 3  Thread 4
Ideally, Each One is a Thread

But We Cannot Do This
What is the Problem?

- Some tasks have **shared resources**
  - **Example:** Fonts all use same engine to make atlases
  - Cannot execute without protecting critical section
  - Typically easier to just **not** do them concurrently

- Some tasks have **dependencies**
  - **Example:** Widgets must come after images, fonts
  - Forces an order on the asset loading

- What we want is a task **service manager**
  - Executes given tasks in a **partial order**
Solution: Thread Pool

- Threads + scheduler
- Scheduler puts tasks thread
- Uses first available thread
- Holds tasks if all busy
Solution: Thread Pool

Scheduler

Task 1
Task 2
Task 3
Task 4
Solution: Thread Pool
Solution: Thread Pool

- Task 1
- Task 2
- Task 3
- Task 4

Scheduler
Solution: Thread Pool
Solution: Thread Pool

Scheduler

Task 1

Task 2

Task 3

Task 4
Solution: Thread Pool
Solution: Thread Pool

Task 1

Task 3

Task 4

Task 5

Scheduler

Multithreading
Solution: Thread Pool

Scheduler

Task 1

Task 5

Task 3

Task 4

Multithreading
CUGL Support: ThreadPool

• /**
  * Returns a thread pool with the given number of threads.
  *
  * @param threads the number of threads in this pool
  *
  * @return a thread pool with the given number of threads.
  */
  
  static std::shared_ptr<ThreadPool> alloc(int threads = 4)

• /**
  * Adds a task to the thread pool.
  *
  * @param task the function to add to the thread pool
  */
  
  void addTask(const std::function<void()>& task)
CUGL Support: ThreadPool

- /**
  * Returns a thread pool with the given number of threads.
  *
  * @param threads the number of threads in this pool
  *
  * @return a thread pool with the given number of threads.
  */

  static std::shared_ptr<ThreadPool> alloc(int threads = 4)

- /**
  * Adds a task to the thread pool.
  *
  * @param task the function to add to the thread pool
  *
  * @param thread pool
  */

  void addTask(const std::function<void()> &task)

AssetManager is a one thread pool
Recall: Custom Loaders

- **void read**(key, src, cb, async)
  - Reads asset from file src
  - async indicates if in sep thread
  - Callback cb executed when done

- **void read**(json, cb, async)
  - Values key and src now in json
  - As are other special properties

- **void materialize**(key, asset, cb)
  - Code to “finish” asset
  - Always in the **main thread**
Recall: Custom Loaders

- `void read(key, src, cb, async)`
  - Reads asset from file `src`
  - `async` indicates if in sep thread
  - Callback `cb` executed when done

- `void read(json, cb, async)`
  - Values `key` and `src` now in `json`
  - As are other special properties

- `void materialize(key, asset, cb)`
  - Code to “finish” asset
  - Always in the main thread
Executing Tasks on the Main Thread

- Any other thread can access the `Application`
  - Use the static method `Application::get()`
  - This class is essentially a singleton

- That object has a `schedule` method
  - Works much like `addTask` in thread pool
  - But executes that task on the main thread
  - Executed just before the call to your `update`

- **Scheduling** this task is thread safe
The Schedule Method

/**
 * Schedules a task function on the main thread.
 *
 * @param cb The task callback function
 * @param ms The number of milliseconds to delay
 *
 * @return a unique identifier for the task
 */

Uint32 schedule(std::function<bool()> cb, Uint32 ms)
/**
 * Schedules a task function on the main thread.
 *
 * @param cb The task callback function
 * @param ms The number of milliseconds to delay
 *
 * @return a unique identifier for the task
 */

Uint32 schedule(std::function<bool()> cb, Uint32 ms)
Putting it All Together

Game Thread

Update

Draw

Asset Thread

Asset Manager

addTask(...)
schedule(...) Schedules materialize

Application

ThreadPool

Multithreading
Aside: Schedule is Useful in General

• Can specify an event to **run in the future**
  • This is the purpose of the milliseconds
  • May be easier than tracking a timer yourself

• Can specify a task to **run periodically**
  • **Example:** Spawning enemies
  • The task returns `true` if it wants to run again
  • Same delay is applied as the first time
  • Alternate schedule separates `delay` and `period`
Recall: Playing Sound Directly

Write PCM chunk to buffer

PCM data buffer

Game Loop

Sound Card

Missing a write causes pops/clicks
The CUGL Approach

Game Thread

Update

Draw

DSP Graph

Audio Thread

Update

Application

Thread

Multithreading
The CUGL Approach

Game Thread
- Update
  - Draw

DSP Graph
- modifies

Audio Thread
- Update
  - reads

Application

Thread

Multithreading
The CUGL Approach

Game Thread  DSP Graph  Audio Thread

This is a very complex Producer/Consumer

Application  Thread

Multithreading
Aside: Audio is Not a ThreadPool

- Audio is a dedicated `std::thread`
  - Because it needs to run as long as the game does
  - Started when you initialized `AudioEngine`

- But process is similar to `ThreadPool`
  - Package your task as a `std::function<void()>`
  - Pass this when you create the thread object

- Difference is that task is in a loop
  - Has an attribute called `running` to manage loop
  - When you set to `false`, the thread is done
The CUGL Approach

Game Thread
- Update
- Draw

DSP Graph
- modifies

Audio Thread
- Update
- reads

How do we protect the critical section?

Multithreading
The Java Approach: Synchronized

public class CriticalSection {

    synchronized void method1() {...}

    synchronized void method2() {...}

    synchronized void method3() {...}

}
The Java Approach: Synchronized

```java
public class CriticalSection {

    synchronized void method1() {...}

    synchronized void method2() {...}

    synchronized void method3() {...}

}
```

Locked to one thread at a time

Lock applies to all of the methods
# C++ Actually Has Two Tools

<table>
<thead>
<tr>
<th>std::mutex</th>
<th>std::atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Used to protect a code block</strong></td>
<td><strong>Used to protect a variable</strong></td>
</tr>
<tr>
<td>• Places lock on code block</td>
<td>• Prevents data races</td>
</tr>
<tr>
<td>• Only one thread can access</td>
<td>• Useful for shared setters</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td>• Can replicate synchronized</td>
<td>• 10x faster than std::mutex</td>
</tr>
<tr>
<td>• Relatively easy to use</td>
<td>• Sometimes easy to use</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td>• Locking has some cost</td>
<td>• Extremely limited in use</td>
</tr>
<tr>
<td>• Deadlocks easy if careless</td>
<td>• Advanced use is advanced</td>
</tr>
</tbody>
</table>
C++ Actually Has Two Tools

<table>
<thead>
<tr>
<th>std::mutex</th>
<th>std::atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td>- Used to protect a code block</td>
<td>- Prevents data races</td>
</tr>
<tr>
<td>- Places lock on code block</td>
<td>- Useful for shared setters</td>
</tr>
<tr>
<td>- Only one thread can access</td>
<td></td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td>- Locking has some cost</td>
<td>- Extremely limited in use</td>
</tr>
<tr>
<td>- Deadlocks easy if careless</td>
<td>- Advanced use is advanced</td>
</tr>
</tbody>
</table>

- Audio thread uses only when it must do so
- Audio thread uses whenever it is possible
class CriticalSection {
private:
    /** Mutex to synchronize methods */
    std::mutex _mutex;

public:
    void method() {
        _mutex.lock();        // Lock method code
        ...
        _mutex.unlock();    // Release when done
    }
}
Observations About `std::mutex`

- **It is not a reentrant lock** (unlike `synchronized`)  
  - Locking it again inside same class will deadlock  
  - This matters when you have locks on helpers  
  - Must use `std::recursive_mutex` for reentrant lock

- **Manual lock/unlock calls are frowned upon**  
  - To easy to forget to unlock and deadlock  
  - Preferred way is to attach a `locking object`  
  - When locking object is deleted, so is lock
Using a Locking Object

class CriticalSection {
private:
    // ** Mutex to synchronize methods */
    std::mutex _mutex;

public:
    void method() {
        std::lock_guard<std::mutex> lock(_mutex);
        ...
        // Mutex unlocked once lock variable deleted
    }
}
What If Critical Section is a Variable?

- **Example**: *running* attribute controlling thread
  - Audio thread loops so long as it is *true*
  - Setting it to *false* stops the audio

- Mutexes exist to prevent *inconsistent states*
  - Either all code is executed, or none is
  - Cannot happen to variable assignment, right?

- C++ is not *assembly code*!
  - A single assignment is multiple lines of assembly
  - This is not thread safe (*especially* on Windows)
What If Critical Section is a Variable?

- **Example**: `running` attribute controlling thread
  - Audio thread loops so long as it is `true`
  - Setting it to `false` stops the audio

- Mutexes exist to prevent inconsistent states
  - Either all code is executed, or none is
  - Cannot happen to variable assignment, right?

- C++ is not *assembly code*!
  - A single assignment is multiple lines of assembly
  - This is not thread safe (*especially* on Windows)
std::atomic Protects Assignment

- **Template** around a type: `std::atomic<int>`
  - Supports all primitive C++ types
  - Cannot apply to objects in general, but …
  - Is possible to make `std::shared_ptr` atomic

- Supported by two methods
  - `load()`: An atomic **getter** for the value
  - `store(value)`: An atomic **setter** for the value
  - Shared pointers are slightly more complicated
std::atomic Protects Assignment

- **Template** around a type: `std::atomic<int>`
  - Supports all primitive C++ types
  - Cannot apply to objects in general, but...
  - Is possible to make `std::shared_ptr` atomic

- Supported by two methods:
  - `load()`: An atomic **getter**
  - `store(value)`: An atomic **setter**

- Shared pointers are slightly more complicated

**Means assignment is atomic, not methods**
class WithAtomics {
private:
    std::atomic<int> _xvar;  // Atomic integer
public:
    /** Change the value of X */
    void writeX(int val) { _xvar.store(val); }

    /** Use the value of X to compute something */
    void readX() {
        int x = _xvar.load();  // Copy value to local variable
        // Use x in local computation
    }
}
class WithAtomics {
private:
    std::atomic<int> _xvar;  // Atomic integer
public:
/** Change the value of X */
void writeX(int val) { _xvar.store(val); }

/** Use the value of X to compute something */
void readX() {
    int x = _xvar.load();  // Copy value to local variable
    // Use x in local computation
}
}
This Is Only Scratching the Surface

- C++ supports monitors and semaphores
  - These are used for producer/consumer problem
  - Monitor allows consumer to wait on producer

- C++ supports promises
  - These are threads that return a value
  - Simplify critical section in that case

- Atomics support memory orders
  - These are used to optimize performance
  - Best avoided unless you know what you are doing
This Is Only Scratching the Surface

- C++ supports monitors and semaphores
  - These are used for producer/consumer problem
  - Monitor allows consumer to wait on producer
- C++ supports promises
  - These are threads that return a value
  - Simplify critical section in that case
- Atomics support memory orders
  - These are used to optimize performance
  - Best avoided unless you know what you are doing

See readings if want more
So Why Do We Care?

- All of these threads are made for you!

- But how about making your own threads?
  - Pathfinding is a classic example
  - NPC behavior can also be long-running

- How can extreme can we go?
  - What if all updates are in separate thread?
  - Then the main thread just draws!
  - This can give us potentially very high FPS
This Will Not Quite Work

Frame 1

Frame 2

Frame 3

Without update, redraw same image.

We need animation in the core loop.
## Recall: Two Approaches to Animation

<table>
<thead>
<tr>
<th>Tweening</th>
<th>Physics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Animates</strong> <strong>timed actions</strong></td>
<td><strong>Animates</strong> <strong>physical objects</strong></td>
</tr>
<tr>
<td>• Given a duration and a start</td>
<td>• Bodies with force and mass</td>
</tr>
<tr>
<td>• Interpolates scene over time</td>
<td>• Also shape for collisions</td>
</tr>
<tr>
<td><strong>Render thread simply…</strong></td>
<td><strong>Render thread simply…</strong></td>
</tr>
<tr>
<td>• accesses all active actions</td>
<td>• steps simulation forward</td>
</tr>
<tr>
<td>• moves them forward by $dt$</td>
<td>• renders objects at end</td>
</tr>
<tr>
<td><strong>Gameplay creates actions</strong></td>
<td><strong>Gameplay nudges objects</strong></td>
</tr>
<tr>
<td>• Happens less frequently</td>
<td>• <em>Might</em> be less frequent</td>
</tr>
<tr>
<td>• Decoupled from render</td>
<td>• If so, can also decouple</td>
</tr>
</tbody>
</table>
Recall: Two Approaches to Animation

Tweening

- Animate **timed actions**
  - Given a duration and a start
  - Interpolates scene over time
- Render thread simply...
  - Accesses all active actions
  - Moves them forward by $dt$
- Gameplay **creates** actions
  - Happens less frequently
  - Decoupled from render

Physics

- Animate **physical objects**
  - Bodies with force and mass
  - Also shape for collisions
- Gameplay **nudges** objects
  - *Might* be less frequent
  - If so, can also decouple

Like networking, animation uses **dead reckoning** when missing input
A New Architecture

Animation Thread

- Update Tweening
- Simulate Physics

Gameplay Thread

- Process Input
- Process Player Actions
- Process NPC Actions
- Process Interactions

Game State

- reads
- modifies

Draw
A New Architecture

Animation Thread

- Update Tweening
- Simulate Physics

Gameplay Thread

- Process Input
- Process Player Actions
- Process NPC Actions
- Process Interactions

Game State

Draw

But don’t want this slow either!

modifies
Summary

- Games engines are naturally multithreaded
  - Offload tasks that *block* drawing (*asset loading*)
  - Offload tasks that *slow* drawing (*pathfinding*)
  - Execute tasks *decoupled* from drawing (*audio*)

- CUGL has native *task-based parallelism*
  - ThreadPool for tasks off the main thread
  - Application::schedule for tasks on main thread

- C++ has general-purpose tools for parallelism
  - std::thread class for managing other threads
  - std::mutex and std::atomic for critical sections