Lecture 10

Memory Management
Gaming Memory (Current Generation)

- **Playstation 4**
  - 8 GB RAM (unified)

- **X-Box One (X)**
  - 12 GB RAM (unified)
  - 9 GB for games

- **Nintendo Switch**
  - 3 GB RAM (unified)
  - 1 GB only for OS

- **iPhone/iPad**
  - 2 GB RAM (unified)
  - Better than an XBox 360
Memory Usage: Images

- Pixel color is 4 bytes
  - 1 byte each for r, b, g, alpha
  - More if using HDR color

- Image a **2D array** of pixels
  - 1280x1024 monitor size
  - 5,242,880 bytes ~ 5 MB

- More if using **mipmaps**
  - Graphic card texture feature
  - Smaller versions of image
  - Cached for performance
  - But can double memory use
But My JPEG is only 8 KB!

- Formats often **compressed**
  - JPEG, PNG, GIF
  - But not always TIFF
- Must **uncompress** to show
  - Need space to uncompress
  - In RAM or graphics card
- Only load when needed
  - Loading is primary I/O operation in AAA games
  - Causes “texture popping”
But My JPEG is only 8 KB!

- Formats often **compressed**
  - JPEG, PNG, GIF
  - But not always TIFF
- Must **uncompress** to show
  - Need space to uncompress
  - In RAM
- Only load when needed
  - Loading is primary I/O operation in AAA games
  - Causes “texture popping”

*Sounds have a similar problem*
Loading Screens

Rebel Mages

WHEREAS THE CIRCLE WAS ESTABLISHED NOT MERELY TO PROTECT THE WORLD FROM MAGES, BUT ALSO TO ALLOW MAGES TO PRACTICE THEIR ART SAFELY AND WITHOUT FEAR, AND,

WHEREAS UNDER LORD SEEKER LAMBERT'S COMMAND, THE TEMPLARS SWORN TO PROTECT ALL PEOPLE INCLUDING MAGES FROM THE HARMFUL EFFECTS OF MAGIC, HAVE INSTEAD PERSECUTED MAGES WITH SUCH BIASED JUDGMENT AS TO WORSEN THE PROBLEMS THEY WERE MEANT TO MITIGATE, AND,

WHEREAS THE RITE OF TRANQUILITY, INTENDED AS A TOOL OF LAST RESORT TO STOP UNCONTROLLED MAGES FROM HURTING THEMSELVES OR OTHERS, HAS INSTEAD BEEN USED FOR PUNITIVE AND POLITICAL PURPOSES TO SILENCE DISSERT AND INHIBIT CIVILIZED DISCOURSE, AND,

WHEREAS ANDRASTE HERSELF INTENDED THE RELATIONSHIP BETWEEN MAGE AND TEMPLAR TO BE ONE OF PRACTITIONER AND PROTECTOR, NOT PRISONER AND
Problems with Asset Loading

• How to load assets?
  • May have a lot of assets
  • May have large assets

• Loading is **blocking**
  • Game stops until done
  • Cannot draw or animate

• May need to **unload**
  • Running out of memory
  • Free something first

Memory Management
Problems with Asset Loading

- **How to load assets?**
  - May have a lot of assets
  - May have large assets

- **Loading is blocking**
  - Game stops until done
  - Cannot draw or animate

- **May need to unload**
  - Running out of memory
  - Free something first
Loading Screens

Minimal animation/feedback while loading assets
Solution: Asynchronous Loader

Game Thread

- Update

Second Thread

- Asset Loader

Specify Asset

Notify done

Update and draw simple animations until assets loaded
Solution: Asynchronous Loader

Game Thread

- Update
- Draw

Second Thread

- Asset Loader
- Specify Asset
- Notify done

- Also an asset **manager**
  - Each asset given a **key**
  - Can access asset by key
  - Works like hash table
**Solution: Asynchronous Loader**

- **Game Thread**
  - Update
  - Draw

- **Second Thread**
  - Asset Loader
    - Specify Asset
    - Notify done

- **Not always a good idea**
  - May need OpenGL utils
  - **Example**: Textures
  - Limited to main thread
Alternative: Iterative Loader

Game Thread

Update

Draw

Asset Manager

Initialize

Update

Access
Alternative: Iterative Loader

- Uses a time budget
  - Give set amount of time
  - Do as much as possible
  - Stop until next update

- Better for OpenGL
  - Give time to manager
  - Animate with remainder
  - No resource contention

- LibGDX approach
  - CUGL is asynchronous

Asset Manager

- Initialize
- Update
- Access
Alternative: Iterative Loader

- Uses a time budget
  - Give set amount of time
  - Do as much as possible
  - Stop until next update
- Better for OpenGL
  - Give time to manager
  - Animate with remainder
  - No resource contention
- LibGDX approach
  - CUGL is asynchronous
Aside: When Do We Load Assets?

**Main**

**Application**

- **Application Start-up**

**Scene**

- **Level Load**

**Models**

**Node**

**Choice affects design & ownership of the asset manager**
Assets Beyond Images

- AAA games have a lot of 3D geometry
  - Vertices for model polygons
  - Physics bodies **per polygon**
  - Scene graphs for organizing this data

- When are all these objects created?
  - At load time (filling up memory)?
  - Or only when they are needed?

- We need to understand memory better
Traditional Memory Organization

Dedicated to process.
Consists of machine addressable space.

Leverages Virtual Memory

Function parameters
Local variables
Return values

Objects created via new
Allocations with malloc

Program Code
Static Variables

Memory Management
Mobile Memory Organization

Device Memory

Heap

Stack
Program Data

Stack
Program Data

Stack
Program Data

Stack
Program Data
How Do Apps Compete for Memory?

- Active app takes what it can
  - Cannot steal from OS
  - OS may suspend apps

- App Suspension
  - App quits; memory freed
  - Done only as needed

- Suspend apps can recover
  - OS allows limited paging
  - Page out on suspension
  - Page back in on restart
How Do Apps Compete for Memory?

- Active app takes what it can
  - Cannot steal from OS
  - OS may suspend apps

- **App Suspension**
  - App quits; memory freed
  - Done only as needed

- Suspend apps can **recover**
  - OS allows limited paging
  - Page out on suspension
  - Page back in on restart

You must code this! Otherwise, data is **lost**.
State Management in iOS 7+

- **Active**
  - Running & getting input

- **Inactive**
  - Running, but no input
  - Transition to suspended

- **Background**
  - Same as inactive
  - But apps can stay here
  - **Example**: Music

- **Suspended**
  - Stopped & Memory freed
State Management in iOS 7+

- **Active**
  - Running & getting input
- **Inactive**
  - Running, but no input
  - Transition to suspended
- **Background**
  - Same as inactive
  - But apps can stay here
  - **Example**: Music
- **Suspended**
  - Stopped & Memory freed

Write handlers to process entering, leaving each state
iOS State Handling

- **applicationDidBecomeActive:**
  - Your app became (resumed as) the foreground app.
  - Use this to recover memory state.

- **applicationWillResignActive:**
  - Your app will switch to inactive or background.
  - Stop the game loop and page out memory.

- **applicationDidEnterBackground:**
  - Your app is in the background and may be suspended.

- **applicationWillEnterForeground:**
  - Your app is leaving the background, but is not yet active.
Android State Handling

All methods in **Application** class
Android State Handling

All methods in Application class

Reload memory
Android State Handling

All methods in Application class
CUGL is Simplified Android Model

- **onStartup()**
  - Initialized and now active

- **onSuspend()**
  - Sent to background
  - Gives you chance to save
  - Also time to pause music

- **onResume()**
  - Returns to app to active
  - Allows you to restore state

- **onShutdown()**
  - Stopped & memory freed
CUGL is Simplified Android Model

- **onStartup()**
  - Initialized and now active

- **onSuspend()**
  - Sent to background
  - Gives you chance to save
  - Also time to pause music

- **onResume()**
  - Returns to app to active
  - Allows you to restore state

- **onLowMemory()**
  - Warning memory is low
  - Gives you chance to unload
  - Else app will shut down

- **onShutdown()**
  - Stopped & memory freed
Memory Organization and Games

- **Inter-Frame Memory**: Carries over across frame boundaries

- **Update**

- **Intra-Frame Memory**: Recovered each frame

- **Draw**
Memory Organization and Games

Inter-Frame Memory
Carries over across frame boundaries

Heap or Stack? Does it matter?

Intra-Frame Memory
Recovered each frame

Draw
## Distinguishing Data Types

### Intra-Frame
- **Local computation**
  - Local variables
    (managed by compiler)
  - Temporary objects
    (not necessarily managed)
- **Transient data structures**
  - Built at the start of update
  - Used to process update
  - Can be deleted at end

### Inter-Frame
- **Game state**
  - Model instances
  - Controller state
  - View state and caches
- **Long-term data structures**
  - Built at start/during frame
  - Lasts for multiple frames
  - May adjust to data changes
## Distinguishing Data Types

### Intra-Frame
- **Local computation**
  - Local variables
    - Managed by the compiler
  - Local objects
    - Not necessarily managed
- **Transient data structures**
  - Built at the start of update
  - Used to process update
  - Can be deleted at end

### Inter-Frame
- **Game state**
  - Model instances
  - Controllers
  - View state and caches
- **Long-term data structures**
  - Built at start/during frame
  - Lasts for multiple frames
  - May adjust to data changes
Distinguishing Data Types

**Intra-Frame**
- Local computation
  - Local variables
    - Managed by compiler
  - Temporary objects
    - Not necessarily managed
- Transient data structures
  - Built at the start of update
  - Used to process update
  - Can be deleted at end
  - e.g. Collisions

**Inter-Frame**
- Game state
  - Model instances
  - Controller state
  - View state and caches
- Long-term data structures
  - Built at start/during frame
  - Lasts for multiple frames
  - May adjust to data changes
  - e.g. Pathfinding
## Handling Game Memory

### Intra-Frame
- Does not need to be paged
  - Drop the latest frame
  - Restart on frame boundary
- Want size reasonably **fixed**
  - Local variables always are
  - Limited # of allocations
  - Limit new inside loops
- Often use **custom allocator**
  - GC at frame boundaries

### Inter-Frame
- Potential to be paged
  - Defines current game state
  - May just want level start
- Size is more **flexible**
  - No. of objects is variable
  - Subsystems may turn on/off
  - User settings may affect
- **OS allocator** okay, but…
  - Recycle with **free lists**
Two Main Concerns with Memory

- Allocating Memory
  - With OS support: standard allocation
  - Reserved memory: memory pools

- Getting rid of memory you no longer want
  - Doing it yourself: deallocation
  - Runtime support: garbage collection
C/C++: Allocation Process

**malloc**
- Based on memory size
  - Give it number of bytes
  - Typecast result to assign it
  - No initialization at all
- **Example:**
  ```c
  char* p = (char*)malloc(4)
  ```

**new**
- Based on data type
  - Give it a data type
  - If a class, calls constructor
  - Else no default initialization
- **Example:**
  ```c
  Point* p = new Point();
  ```

---

Stack | Heap
-----|-----
[Diagram showing allocation]

Memory Management
C/C++: Allocation Process

**malloc**
- Based on memory size
  - Give it number of bytes
  - Typecast result and use it
- Example:
  ```c
  char* p = (char*)malloc(4)
  ```

**new**
- Based on data type
- Give it a data type
  - If a class, calls constructor
  - Else no default initialization
- Example:
  ```c
  Point* p = new Point();
  ```

Stack
- Preferred in C
Heap
- Preferred in C++

Stack
- Preferred in C
Heap
- Preferred in C++
Custom Allocators

Pre-allocated Array

<table>
<thead>
<tr>
<th></th>
<th>(called Object Pool)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Free</td>
</tr>
<tr>
<td></td>
<td>End</td>
</tr>
</tbody>
</table>

- **Idea**: Instead of `new`, get object from array
  - Just reassign all of the fields
  - Use **Factory pattern** for constructor
  - See `alloc()` method in CUGL objects

- **Problem**: Running out of objects
  - We want to reuse the older objects
  - Easy if deletion is FIFO, but often isn’t
Free Lists

- Create an object **queue**
  - Separate from preallocation
  - Stores objects when “freed”

- To allocate an object...
  - Look at front of free list
  - If object there take it
  - Otherwise make new object

- Preallocation unnecessary
  - Queue wins in long term
  - Main performance hit is deletion/fragmentation

```cpp
// Free the new particle
def freelist.push_back(p);

...

// Allocate a new particle
Particle* q;
if (!freelist.isEmpty()) {
    q = freelist.pop();
} else {
    q = new Particle();
}
q.set(...)```
Particle Pool Example

GL verts: 178
GL calls: 3
60.3 / 0.015

Allocated: 16
Particle Pool Example

class ParticlePool {
public:
  /** Creates a ParticlePool with the given capacity. */
  ParticlePool(int capacity);

  /** Returns a new OR reused object from this pool. */
  Particle* obtain();

  /** Marks object as eligible for reuse. */
  void free (Particle* object);

private:
  /** Allocates a new object from the pool. */
  Particle* alloc();
};

Memory Management
Particle Pool Example

class ParticlePool {
public:
    /** Creates a ParticlePool with the given capacity. */
    ParticlePool(int capacity);
    /** Returns a new OR reused object from this pool. */
    Particle* obtain();
    /** Marks object as eligible for reuse. */
    void free(Particle* object);

private:
    /** Allocates a new object from the pool. */
    Particle* alloc();
};

Use instead of `new`

Use instead of `delete`
Particle Pool Example

class ParticlePool {
public:
    /** Creates a ParticlePool with the given capacity. */
    ParticlePool(int capacity);
    /** Returns a new OR reused object from this pool. */
    Particle* obtain();
    /** Marks object as eligible for reuse. */
    void free (Particle* object);
private:
    /** Allocates a new object from the pool. */
    Particle* alloc();
};

Use instead of new

Use instead of delete

What to do if nothing free
Two Main Concerns with Memory

• Allocating Memory
  • With OS support: standard allocation
  • Reserved memory: memory pools

• Getting rid of memory you no longer want
  • Doing it yourself: deallocation
  • Runtime support: garbage collection
Manual Deletion in C/C++

- Depends on allocation
  - malloc: free
  - new: delete

- What does deletion do?
  - Marks memory as available
  - Does not erase contents
  - Does not reset pointer

- Only crashes if pointer bad
  - Pointer is currently NULL
  - Pointer is illegal address

```cpp
int main() {
    cout << "Program started" << endl;
    int* a = new int[LENGTH];
    delete a;
    for(int ii = 0; ii < LENGTH; ii++) {
        cout << "a[" << ii << "]=" << a[ii] << endl;
    }
    cout << "Program done" << endl;
}
```
Memory Leaks

- **Leak**: Cannot release memory
  - Object allocated on heap
  - Only reference is moved
- Consumes memory fast!
- Can even happen in Java
  - JNI supports native libraries
  - Method may allocate memory
  - Need another method to free
  - **Example**: dispose() in JOGL

```java
memoryArea = newArea;
```
void foo() {
    MyObject* o =
        new MyObject();
    o.doSomething();
    o = null;
    return;
}

void foo(int key) {
    MyObject* o =
        table.get(key);
    o.doSomething();
    o = null;
    return;
}
void foo() {
    MyObject* o = table.get(key);
    table.remove(key);
    o = null;
    return;
}

void foo(int key) {
    MyObject* o = table.get(key);
    table.remove(key);
    table.put(key, o);
    o = null;
    return;
}

Memory Management
A Question of Ownership

Thread 1

void run() {
    o.doSomething1();
}

Thread 2

void run() {
    o.doSomething2();
}

“Owners” of obj

Who deletes obj?
Reference Strength

### Strong Reference

- Reference asserts ownership
  - Cannot delete referred object
  - Assign to NULL to release
  - Else assign to another object
- Can use reference **directly**
  - No need to copy reference
  - Treat like a normal object
- Standard type of reference

### Weak Reference

- Reference != ownership
  - Object can be deleted anytime
  - Often for *performance caching*
- Only use **indirect** references
  - Copy to local variable first
  - Compute on local variable
- Be prepared for NULL
  - Reconstruct the object?
  - Abort the computation?
C++11 Support: Shared Pointers

- C++ can override **anything**
  - Assignment operator =
  - Dereference operator ->
- Use special object as pointer
  - Has field to reference object
  - Tracks ownership of object
  - Uses *reference counting*
- What about deletion?
  - Smart pointer is on *stack*
  - Stack releases ownership

```cpp
Foo object = new Class();
shared_ptr<Foo> handle(object);
...
handle->foo();    // object->foo()
```
C++11 Support: Shared Pointers

```cpp
void foo() {
    shared_ptr<Thing> p1(new Thing);    // Allocate new object
    shared_ptr<Thing> p2=p1;            // p1 and p2 share ownership
    shared_ptr<Thing> p3(new Thing);   // Allocate another Thing

    ...
    p1 = find_some_thing();            // p1 might be new thing
    p3->defragulate();                 // call a member function
    cout <<*p2 << endl;                // dereference pointer

    ...

    // "Free" the memory for pointer
    p1.reset();                        // decrement reference, delete if last
    p2 = nullptr;                      // empty pointer and decrement
}
```

Memory Management
C++11 Support: **Weak Pointers**

```cpp
void foo() {
    shared_ptr<Thing> p1(new Thing); // Allocate new object
    weak_ptr<Thing> p2 = p1; // p2 is a weak reference

    ...

    p1 = find_some_thing(); // p1 might be new thing
    auto p3 = p2.lock(); // Must lock p2 to dereference
    cout << *p3 << endl; // dereference pointer

    ...

    // "Free" the memory for pointer
    p1.reset(); // decrement reference, delete if last
    p2 = nullptr; // empty pointer (but does not decrement)
}
```
Summary

• Memory usage is always an issue in games
  • Uncompressed images are quite large
  • Particularly a problem on mobile devices

• Limit allocations in your animation frames
  • Intra-frame objects: cached objects
  • Inter-frame objects: free lists

• Must track ownership of allocated objects
  • The owner is responsible for deletion
  • C++11 smart pointers can manage this for us