Lecture 12

Memory Management
Take-Aways for Today

• Why does memory in games matter?
  • Is there a difference between PC and mobile?
  • Where do consoles fit in all this?

• Do we need to worry about it in Java?
  • Java has garbage collection
  • Handles the difficult bits for us, right?

• What can we do in LibGDX?
Gaming Memory (Last Generation)

- **Playstation 3**
  - 256 MB RAM for system
  - 256 MB for graphics card

- **X-Box 360**
  - 512 MB RAM (unified)

- **Nintendo Wii**
  - 88 MB RAM (unified)
  - 24 MB for graphics card

- **iPhone/iPad**
  - 1 GB RAM (unified)
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)
Why Not Virtual Memory?

- **Secondary storage** exists
  - Consoles have 500 GB HD
  - iDevices have 64 GB Flash
- But **access time** is slow
  - HDs transfer at ~160 MB/s
  - Best SSD is ~500 MB/s
- **Recall 16 ms** per frame
  - At best, can access 8 MB
  - Yields uneven performance
Aside: Java Memory

- Initial heap size
  - Memory app starts with
  - Can get more, but stalls app
  - Set with \(-Xms\) flag

- Maximum heap size
  - OutOfMemory if exceed
  - Set with \(-Xmx\) flag

- Defaults by RAM installed
  - Initial 25% RAM (<16 MB)
  - Max is 75% RAM (<2 GB)
  - Need more, then set it

```
> java -cp game.jar GameMain
> java -cp game.jar -Xms:64m GameMain
> java -cp game.jar -Xmx:4g GameMain
> java -cp game.jar -Xms:64m -Xms:64m GameMain
```
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
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

- **Uncompress** to display
  - 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

- Uncompress to display
  - Need space to uncompress
  - In RAM or on video card

- 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 DISSENT 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

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
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
  - Draw

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

- Not always a good idea
  - Only one thread can I/O
  - May need OpenGL utils
  - …so will block drawing
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
  - Re-examine game labs
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
  - Re-examine game labs

```
+-------------------+    +-----------------+
|       Load        |    |          Update  |
+-------------------+    +-----------------+
                     +-------------------+    +-----------------+
                     |          Draw     |    |                |
                     +-------------------+    +-----------------+
```
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

- Program Code
- Static Variables

- Program Data
- Heap
- Free Space
- Stack

High Address

Low Address
Traditional Memory Organization

- Stack
  - Function parameters
  - Local variables
  - Return values

- Heap
  - Program Code
  - Static Variables

- Program Data

- Free Space

High Address to Low Address
Traditional Memory Organization

- Stack:
  - Function parameters
  - Local variables
  - Return values

- Heap:
  - Objects created via new (e.g., Every object in Java)

- Program Data:
  - Program Code
  - Static Variables

Memory Management
Traditional Memory Organization

### Program Data
- Program Code
- Static Variables

### Heap
- Objects created via `new`
  - (e.g., Every object in Java)

### Stack
- Function parameters
- Local variables
- Return values

### Free Space

High Address

Low Address

Easy to Handle

Memory Management
Traditional Memory Organization

High Address

Stack

Function parameters
Local variables
Return values

Free Space

Heap

Objects created via new
(e.g. Every object in Java)

Program Data

Program Code
Static Variables

Low Address

Easy to Handle

Easy to Handle

Problems!
Problem with Heap Allocation

- It can be slower to access
- Not always contiguous
- Stacks are nicer for caches
- Garbage collection is brutal
- Old collectors would block
- New collectors are better…
- …but slower than manual
- Very bad if high churn
- Rapid creation/deletion
- **Example**: Particle systems

```java
private void handleCollision(Shell s1, Shell s2) {
    Vector2 axis = new Vector2(s1.getPosition());
    axis.sub(s2.getPosition());
    ...

    Vector2 temp1 = new Vector2(s2.getPosition());
    temp1.sub(s1.getPosition()).nor();
    Vector2 temp2 = new Vector2(s1.getPosition());
    temp2.sub(s2.getPosition()).nor();

    Vector2 temp1 = new Vector2(s2.getPosition());
    temp1.scl(temp1.dot(s1.getVelocity()));
    temp2.scl(temp2.dot(s2.getVelocity()));

    s1.getVelocity().sub(temp1).add(temp2);
    s2.getVelocity().sub(temp2).add(temp1);
}
```
**Problem with Heap Allocation**

- It can be slower to access
  - Not always contiguous
  - Stacks are nicer for caches

- Garbage collection is brutal
  - Old collectors would block
  - New collectors are better…
  - …but slower than manual

- Very bad if high churn
  - Rapid creation/deletion
  
  **Example**: Particle systems

```java
private void handleCollision(Shell s1, Shell s2) {
    // Find the axis of "collision"
    Vector2 axis = new Vector2(s1.getPosition());
    axis.sub(s2.getPosition());

    // Compute the projections
    Vector2 temp1 = new Vector2(s2.getPosition());
    temp1.sub(s1.getPosition()).nor();
    Vector2 temp2 = new Vector2(s1.getPosition());
    temp2.sub(s2.getPosition()).nor();

    // Compute new velocities
    temp1.scl(temp1.dot(s1.getVelocity()));
    temp2.scl(temp2.dot(s2.getVelocity()));

    // Apply to the objects
    s1.getVelocity().sub(temp1).add(temp2);
    s2.getVelocity().sub(temp2).add(temp1);
}
```

Created/deleted every frame
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

<table>
<thead>
<tr>
<th>Intra-Frame</th>
<th>Inter-Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Local computation</td>
<td>• Game state</td>
</tr>
<tr>
<td>• Local variables (managed by compiler)</td>
<td>• Model instances</td>
</tr>
<tr>
<td>• Temporary objects (not necessarily managed)</td>
<td>• Controller state</td>
</tr>
<tr>
<td>• Transient data structures</td>
<td>• View state and caches</td>
</tr>
<tr>
<td>• Built at the start of update</td>
<td>• Long-term data structures</td>
</tr>
<tr>
<td>• Used to process update</td>
<td>• Built at start/during frame</td>
</tr>
<tr>
<td>• Can be deleted at end</td>
<td>• Lasts for multiple frames</td>
</tr>
<tr>
<td></td>
<td>• May adjust to data changes</td>
</tr>
</tbody>
</table>
# Distinguishing Data Types

## Intra-Frame

- **Local computation**
  - Local variables
  - Objects
    - 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 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
  - Controller state and caches
- Long-term data structures
  - Built at start/end of frame
  - Lasts for multiple frames
  - May adjust to data changes

**e.g. Collisions**
**Local Variables**
**Object Fields**
**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
- Make use of **cached objects**
  - Requires careful planning

### 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
- **Preallocate** as possible
  - Recycle with **free lists**
Rule of Thumb: Limiting new

- Limit new to constructors
  - Identify the object owner
  - Allocate in owner constructor

- **Example**: cached objects
  - Look at what algorithm needs
  - Allocate all necessary objects
  - Algorithm just sets the cache

- **Problem**: readability
  - Naming is key to readability
  - But new names = new objects
  - Make good use of comments
Rule of Thumb: Limiting `new`

- Limit new to constructors
  - Identify the object owner
  - Allocate in owner constructor

- **Example**: cached objects
  - Look at what algorithm needs
  - Allocate all necessary objects
  - Algorithm just sets the cache

- **Problem**: readability
  - Naming is key to readability
  - But new names = new objects
  - Make good use of comments
Object Preallocation

- **Idea**: Allocate before need
  - Compute maximum needed
  - Create a list of objects
  - Allocate contents at start
  - Pull from list when needed

- **Problem**: Running out
  - Eventually at end of list
  - Want to reuse older objects
  - Easy if deletion is FIFO
  - But what if it isn’t?

- Motivation for **free list**

```java
// Allocate all of the particles
Particle[] list = new Particle[CAP];
for(int ii = 0; ii < CAP; ii++) {
    list[ii] = new Particle();
}

// Keep track of next particle
int next = 0;

... // Need to “allocate” particle
Particle p = list[next++];
p.set(...);
```
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 garbage collector

```java
// Free the new particle
freelist.push(p);
...

// Allocate a new particle
Particle q;
if (!freelist.isEmpty()) {
    q = freelist.pop();
} else {
    q = new Particle();
}
q.set(…)
```
LibGDX Support: Pool

Pool<T>

- **public void free(T obj);**
  - Add an object to free list
- **public T obtain();**
  - Use this in place of new
  - If object on free list, use it
  - Otherwise make new object
- **public T newObject();**
  - Rule to create a new object
  - Could be preallocated

Pool.Poolable

- **public void reset();**
  - Erases the object contents
  - Used when object freed
  - Must be implemented by T
  - Parameter free constructors
  - Set contents with initializers
- **See MemoryPool demo**
  - Also PooledList in Lab 4
Summary

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

- Asset loading must be balanced with animation
  - LibGDX uses an incremental approach

- Limit calls to `new` in your animation frames
  - **Intra-frame** objects: cached objects
  - **Inter-frame** objects: free lists