the gamedesigninitiative at cornell university

Lecture 15

Memory Management

Take-Aways for Today

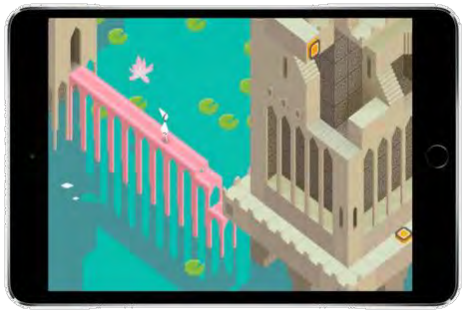
- Why does memory in games matter?
  - Is there a difference between PC and mobile?
  - Where do consoles fit in all this?
- Do we need to worry about it in Java?
  - Java has garbage collection
  - Handles the difficult bits for us, right?
- What can we do in LibGDX?

Gaming Memory (Generation 7)

- **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 (Generation 8)

- **Playstation 4**
  - 8 GB RAM (unified)
- **X-Box One**
  - 12 GB RAM (unified)
  - 9 GB for games
- **Nintendo Wii-U**
  - 2 GB RAM (unified)
  - 1 GB only for OS
- **iPhone/iPad**
  - 2 GB RAM (unified)



Gaming Memory (Current Generation)

- **Playstation 5**
  - 16 GB RAM (unified)
  - **Speed** 448GB/s
- **X-Box Series X**
  - 16 GB RAM (unified)
  - **Speed** 560-336GB/s
- **Nintendo Switch**
  - 3 GB RAM (unified)
  - **Speed** 25.6 GB/s
- **iPhone/iPad**
  - 6 GB RAM (unified)
  - **Speed** 42.7 GB/s

Aside: Memory Affects Games

- **Generation 7**
  - Modern(ish) GPUs
  - Horrible memory
  - **Pretty, but short games**
- **Generation 8**
  - Minor GPU increases
  - Massive memory increases
  - **Open world games**
- **Generation 9**
  - Minor GPU, memory boosts
  - Massive bandwidth boosts
  - **Shorter loading time = ???**

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 is 1/64 of total RAM
  - Max is 1/4 of total RAM
  - 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
```

Memory Usage: Images

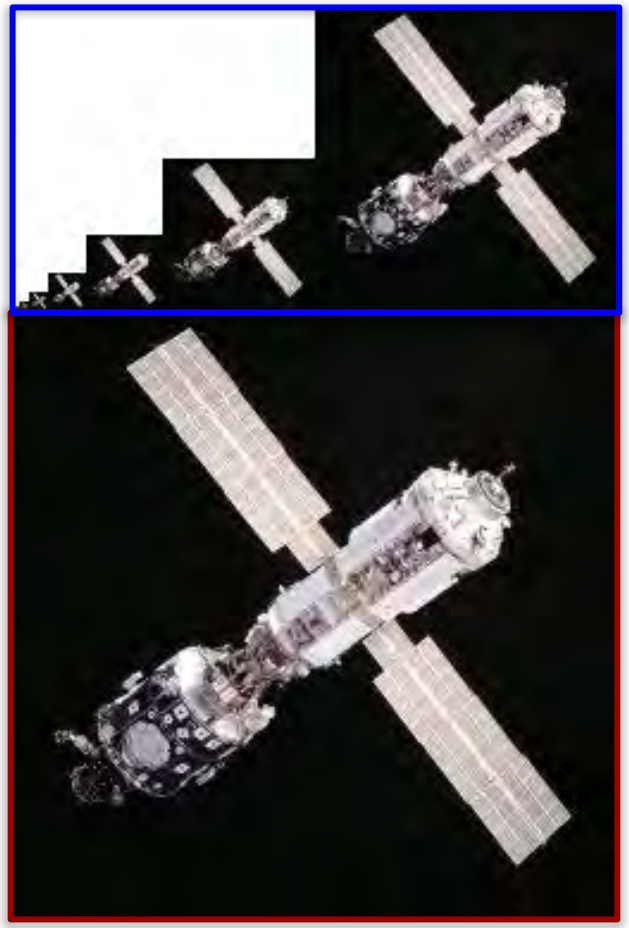
- Pixel color is 4 bytes
  - 1 byte each for r, b, g, alpha
  - More if using HDR color
- Image a **2D array** of pixels
  - 1280x1024 monitor size
  - 5,242,880 bytes ~ 5 MB
- More if using **mipmaps**
  - Graphic card texture feature
  - Smaller versions of image
  - Cached for performance
  - But 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

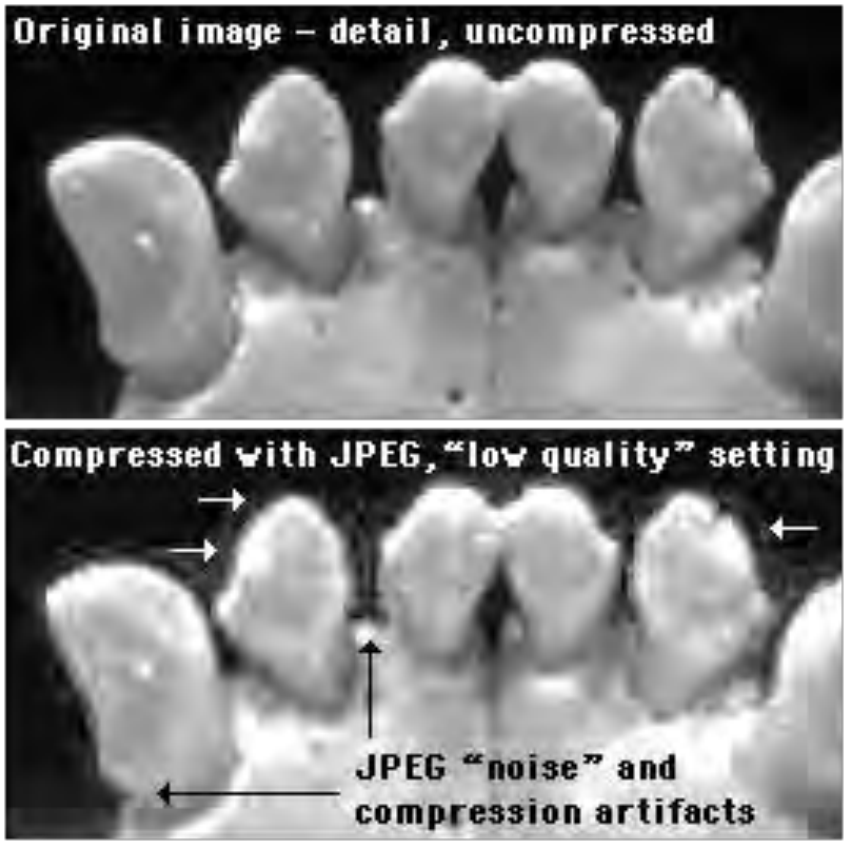
MipMaps

Original Image



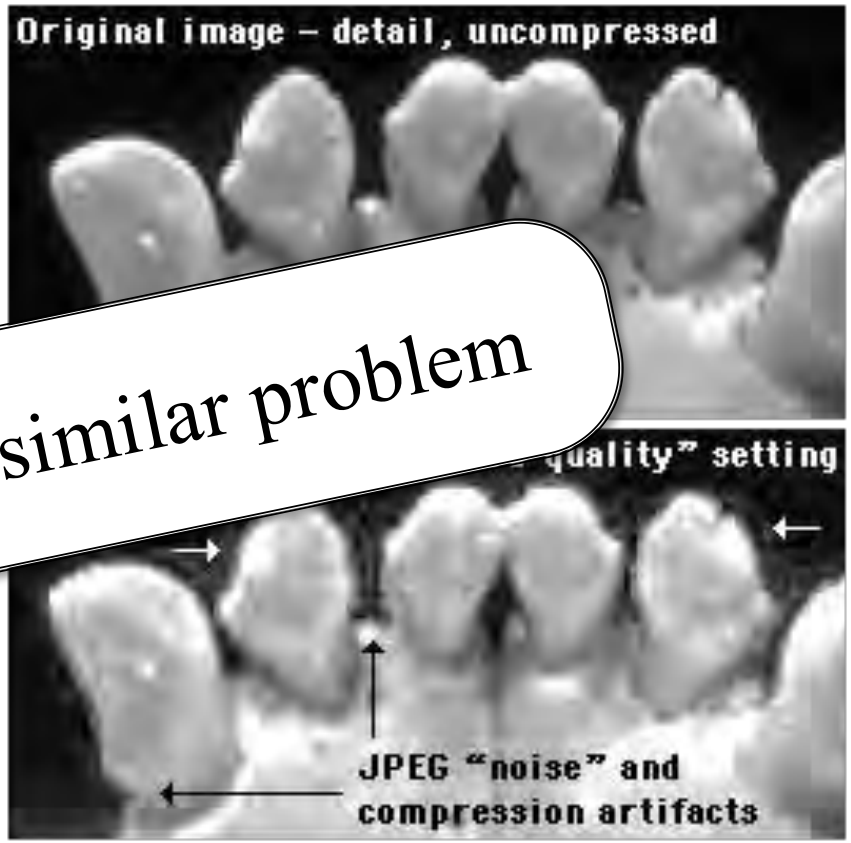
But My JPEG is only 8 KB!

- 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 uncom
  - In RA
- Only lo
  - Loading is primary I/O operation in AAA games
  - Causes "texture popping"

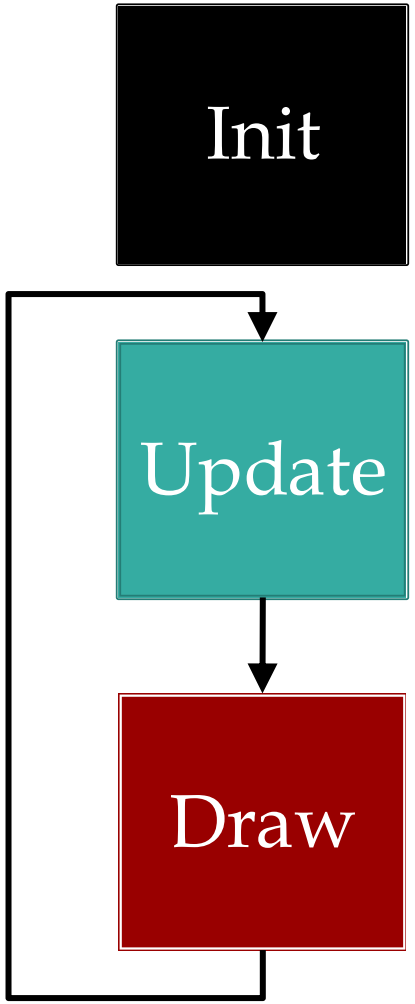


Sounds have a similar problem

Loading Screens

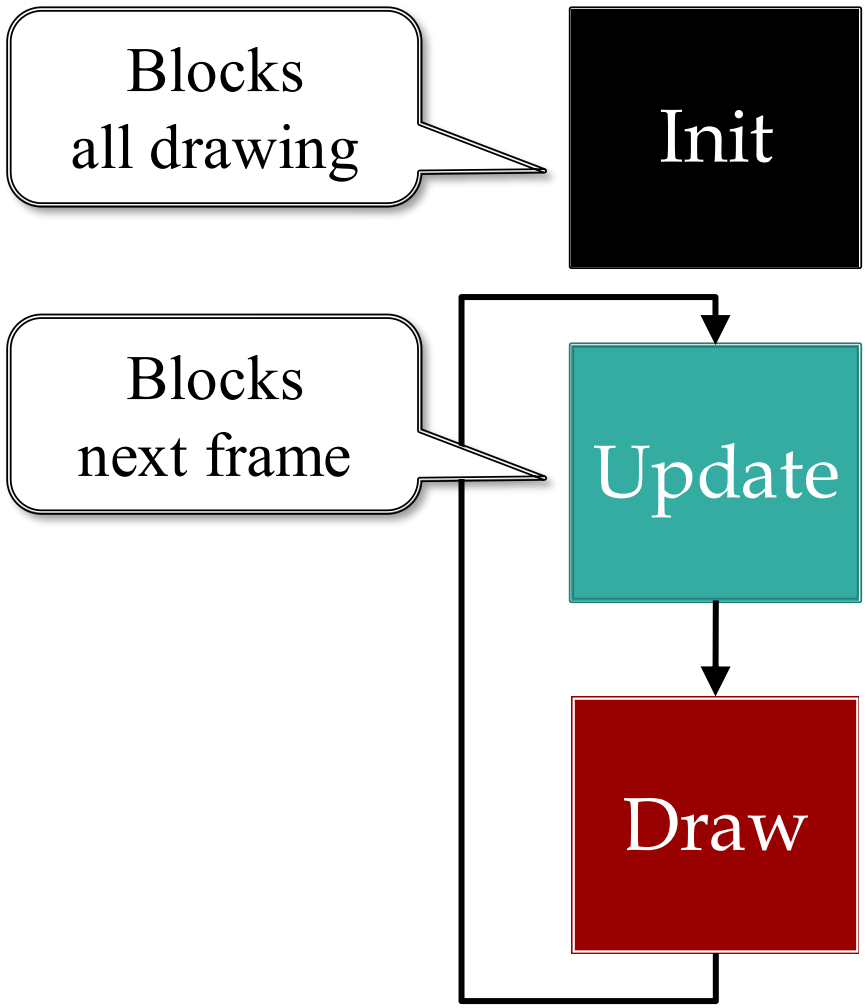
Problems with Asset Loading

- 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



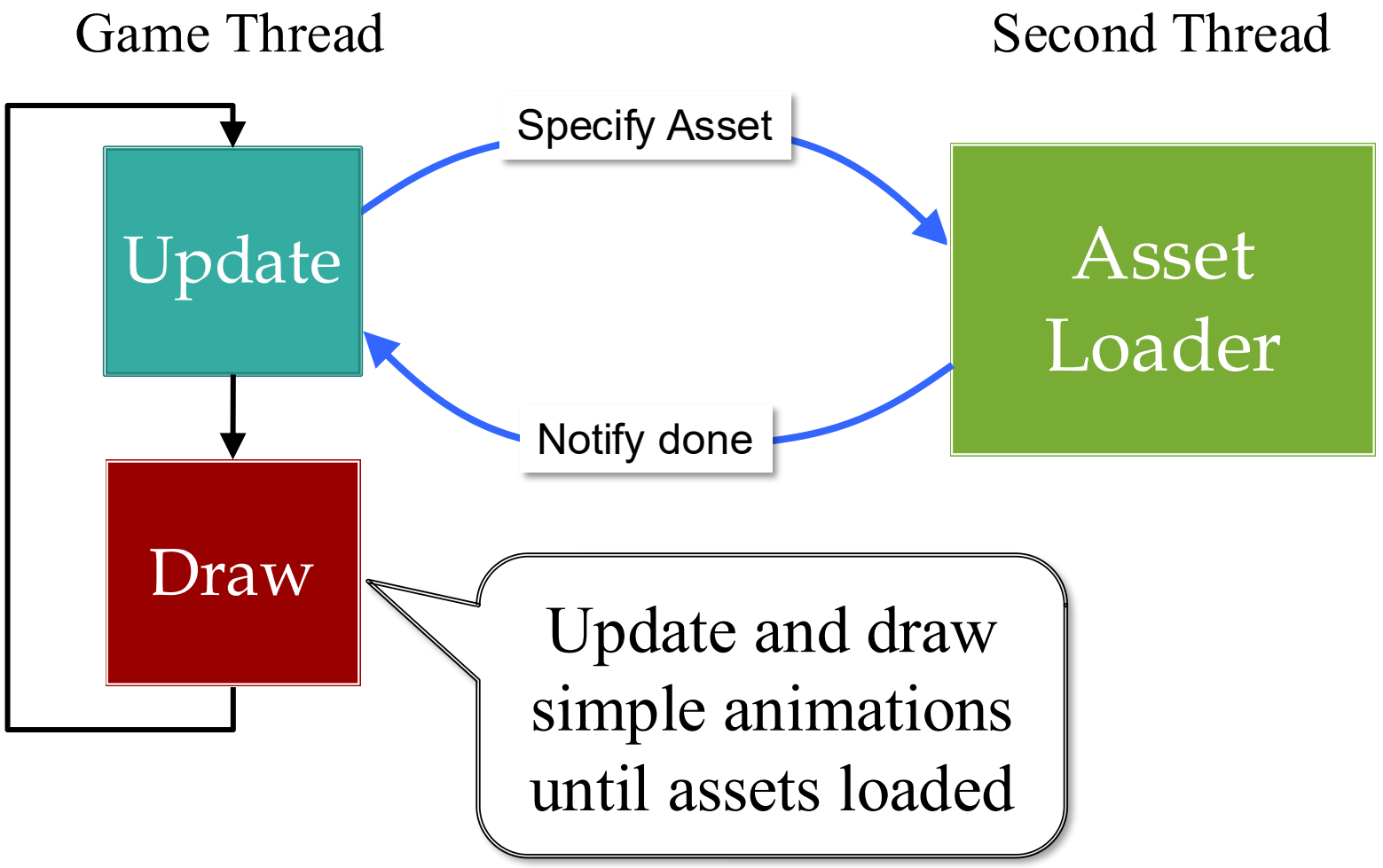
Problems with Asset Loading

- 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

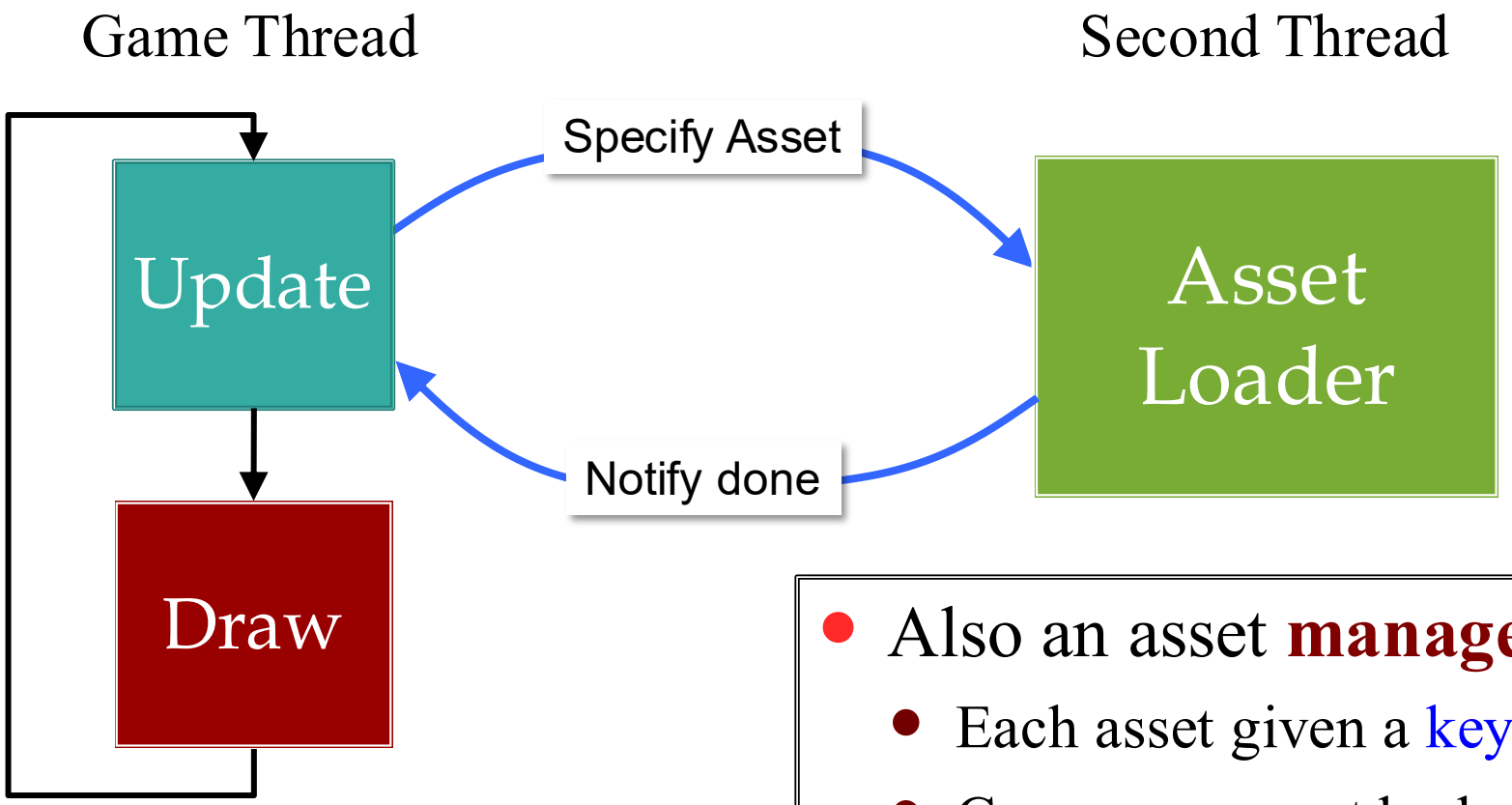


Loading Screens

Solution: Asynchronous Loader

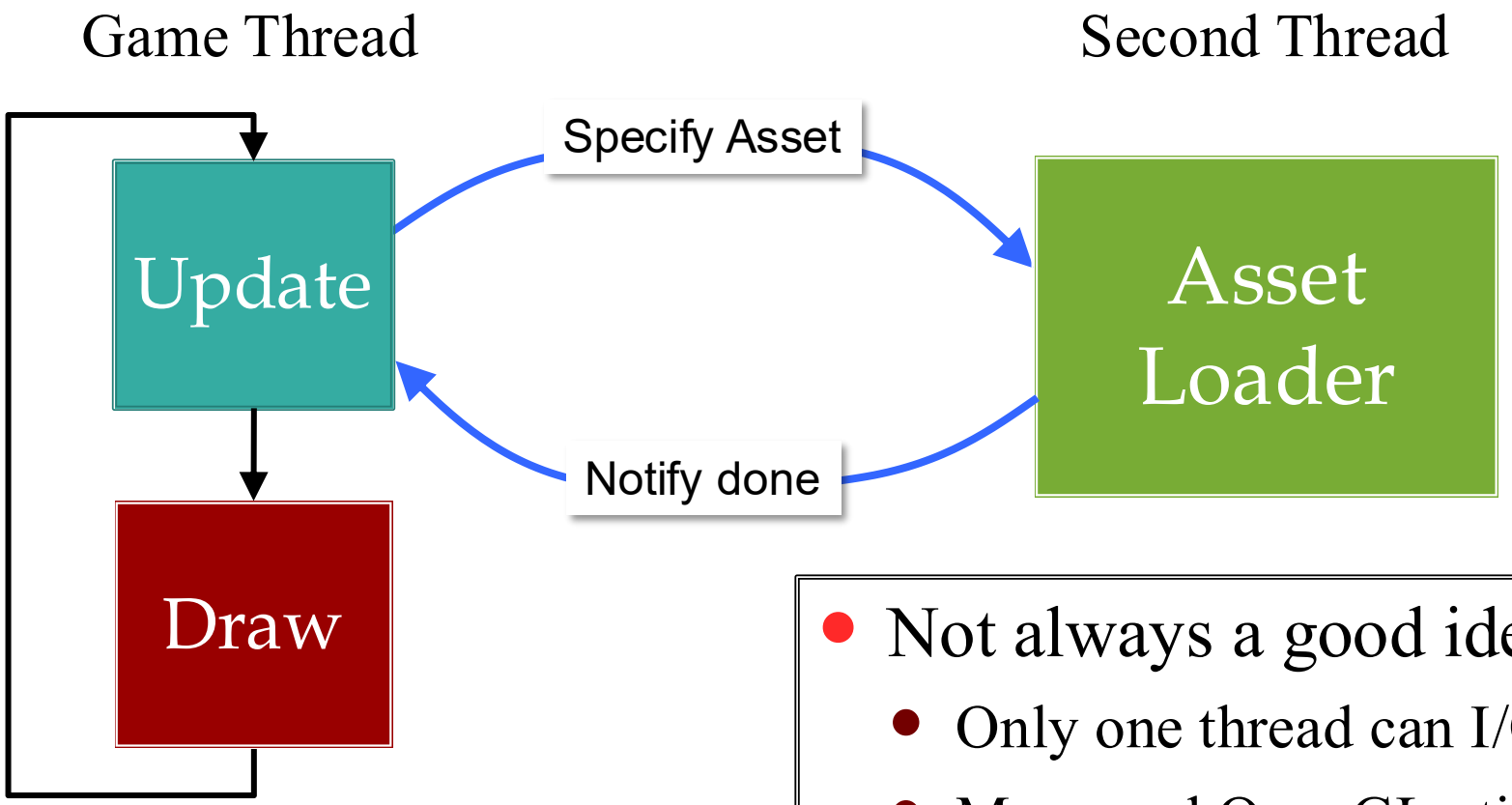


Solution: Asynchronous Loader



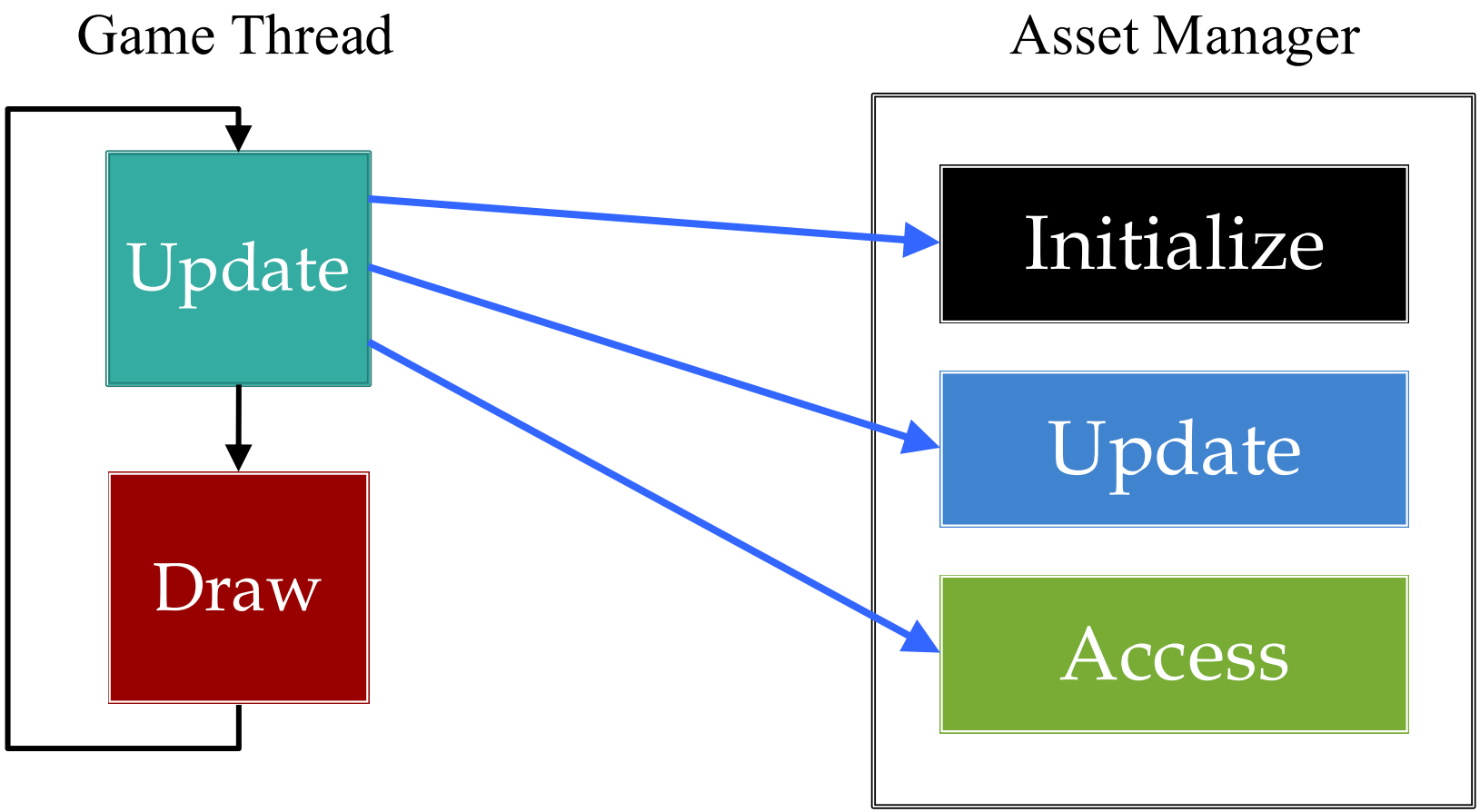
- Also an asset **manager**
  - Each asset given a key
  - Can access asset by key
  - Works like Java Map

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

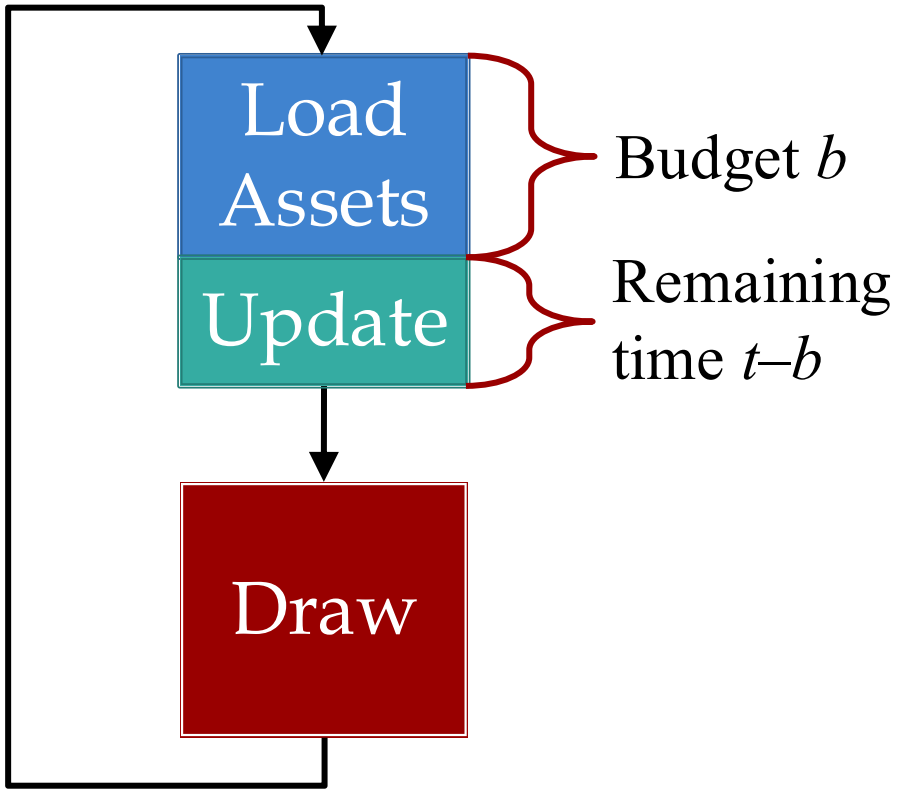


Alternative: Iterative Loader

- 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



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
- **How do we load these things?**
  - Managers handle built-in asset types
  - What if we need to make a custom data type?
- And exactly when do we load these?

Custom Loaders in LibGDX

- The LibGDX asset system is modular
  - Use an asset manager to load/store assets
  - But each asset type has an associate **loader**
- A loader class has the following
  - Inner subclass of `AssetLoaderParameters`
  - Method `loadSync` for loading in main thread
  - Method `loadAsync` safe for separate threads
- GDIAC extensions have associated **parsers**
  - Reads asset json and sends information to loaders
  - Primarily an iterator for `AssetLoaderParameters`

The LoadingScene in LibGDX

```
// The loading scene needs some assets
itself
internal = new AssetDirectory(
"loading/boot.json" );
internal.loadAssets();
internal.finishLoading();

// Access those assets and build loading
scene
…

// Now load the real assets
assets = new AssetDirectory( file );

setLoader(MyAsset.class, new MyAssetLoader()
);
addParser(new MyAssetParser());
assets.loadAssets();
```

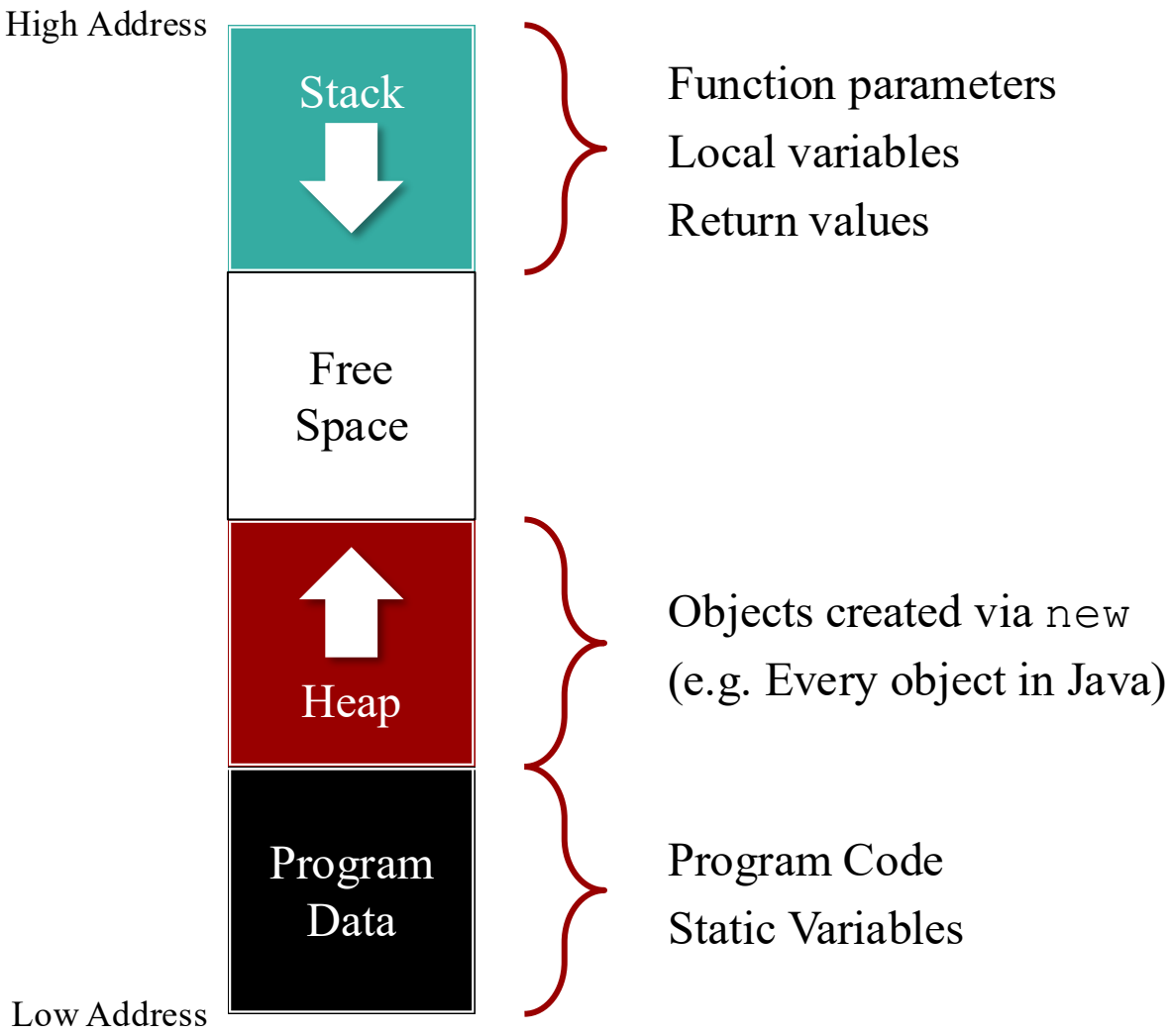
The LoadingScene in LibGDX

```
// The loading scene needs some assets
itself
internal = new AssetDirectory(
"loading/boot.json" );
internal.loadAssets();
internal.finishLoading();

// Access those assets a                ing
scene
…

// Now load the real assets
assets = new AssetDirectory( file );

assets.setLoader                        s, new
MyAssetLoader()
assets.addParser                        arser());

assets.loadAssets();
```

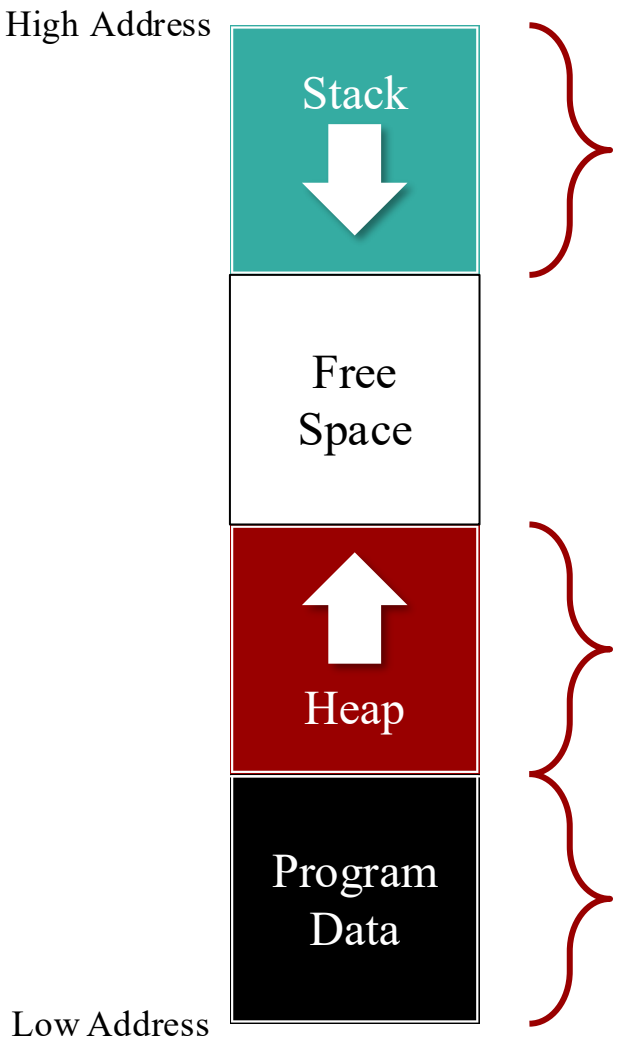
Defines how to load asset

Defines how to read the JSON

Traditional Memory Organization



Traditional Memory Organization



Function parameters
Local variables
Return values

Easy to Handle

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

Problems!

Program Code
Static Variables

Easy to Handle

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

```
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
```

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

```
private void
handleCollision(Shell s1, Shell
s2) {
    // Find the axis of
"coll
    V
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
```

Created/deleted every frame

Aside: Stack Based Allocation

- C++ can put objs on stack
  - Object deleted at end of call
  - No GC computation at all
  - Good for short-life objects
- Java can *approximate* this
  - Checks if local to function
  - If so, will delete it
- But not a perfect solution
  - Can never **return** object
  - Init has hidden costs

```
void getCollides(Shell
s1, Shell s2) {
    // Find collision
axis
    Vector2 axis = new

Vector2(s1.getPosition(
));

axis.sub(s2.getPosition
());
    axis.nor();

axis.scale(s1.getRadius
());
    // Find collision
```

Aside: Stack Based Allocation

- C++ can put objs on stack
  - Object deleted at end of call
  - No GC computation at all
  - Good for short-life objects
- Java can *approximate* this
  - Checks if local to function
  - If so, will delete it
- But not a perfect solution
  - Can never **return** object
  - Init has hidden costs

```
void getCollides(Shell
s1, Shell s
    // Find
axis
    Vector2 axis = new

Vector2(s1.getPosition(
));

axis.sub(s2.getPosition
());
    axis.nor();

axis.scale(s1.getRadius
());
    // Find collision
```

Deleted

Not Deleted

Aside: Java Garbage Collection

- **Parallel Garbage Collector** (The Default)
  - **Freezes your application** when it collects
- **Serial Garbage Collector** (`-XX:+UseSerialGC`)
  - Like PGC but better for simple programs
- **CMS Garbage Collector** (`-XX:+UseParNewGC`)
  - Concurrent mark-and-sweep rarely freezes app
- **G1 Garbage Collector** (`-XX:+UseG1GC`)
  - Even less app freezing at cost of large heap size

Aside: Java Garbage Collection

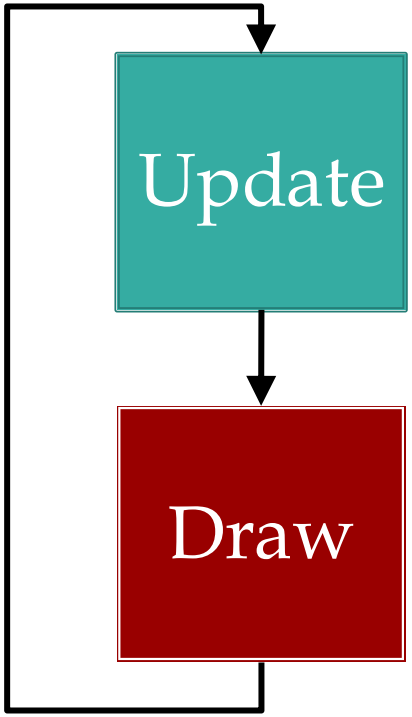
- **Parallel Garbage Collector** (The Default)
  - **Freezes your application** when it collects
- **Serial Garbage Collector** (`-XX:+UseSerialGC`)
  - Like PG
- **CMS Ga** `rNewGC`)
  - Concurre app
- **G1 Garbage Collector** (`-XX:+UseG1GC`)
  - Even less app freezing at cost of large heap size

The preferred GC for high performance Java

Memory Organization and Games

Inter-Frame Memory

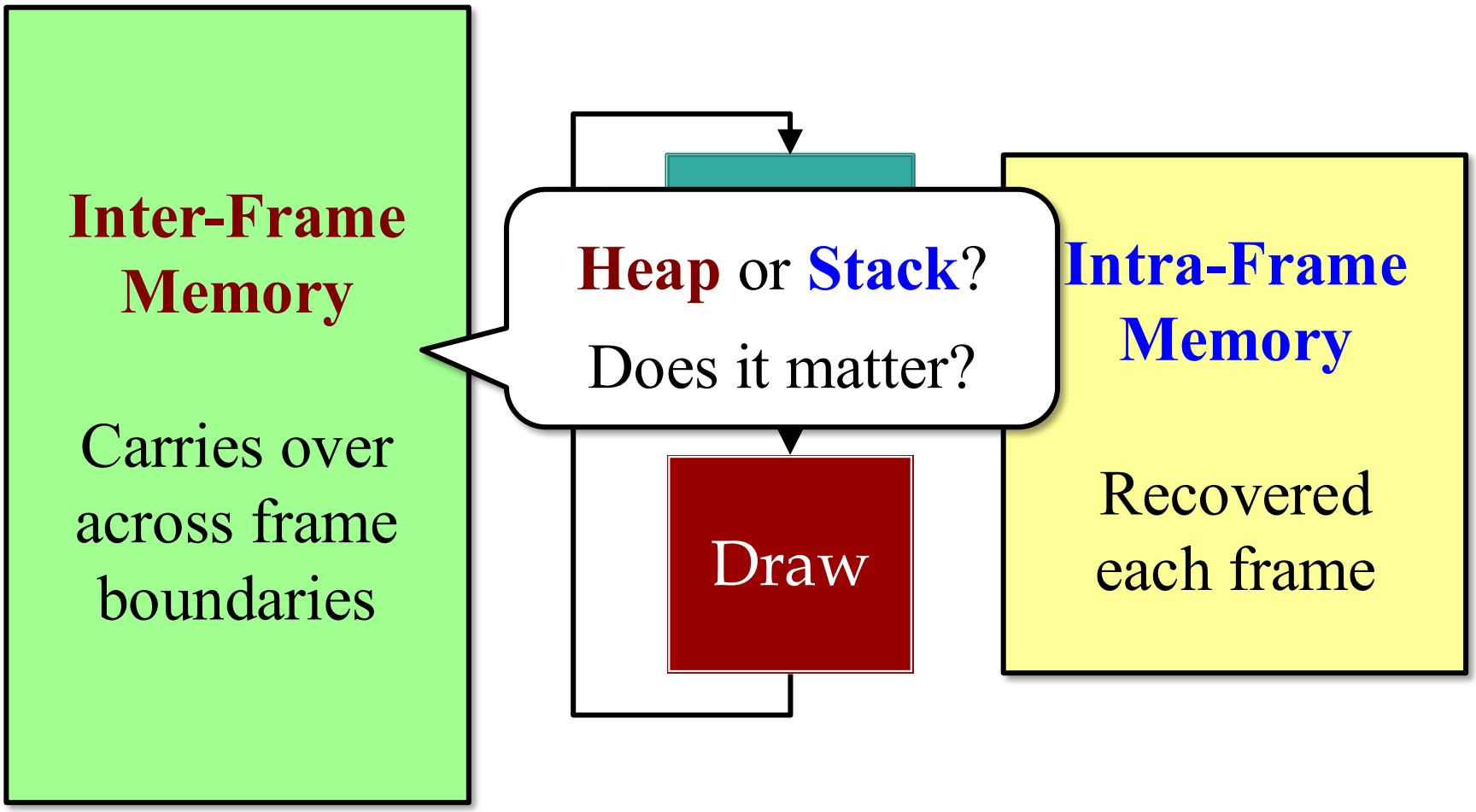
Carries over across frame boundaries



Intra-Frame Memory

Recovered each frame

Memory Organization and Games



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
  - (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
  - and caches
- **Long-term data structures**
  - Built at start/during frame
  - Lasts for multiple frames
  - May adjust to data changes

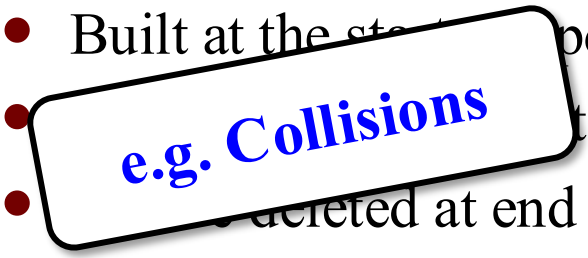
Distinguishing Data Types

Intra-Frame

- **Local computation**
  - Local variables
  - (not necessarily managed)
- **Transient data structures**
  - Built at the st pdate
  - te
  - deleted at end

Inter-Frame

- **Game state**
  - Model instances
  - and caches
- **Long-term data structures**
  - Built at start/d rame
  - mes
  - djust to data changes



Handling Game Memory

Intra-Frame

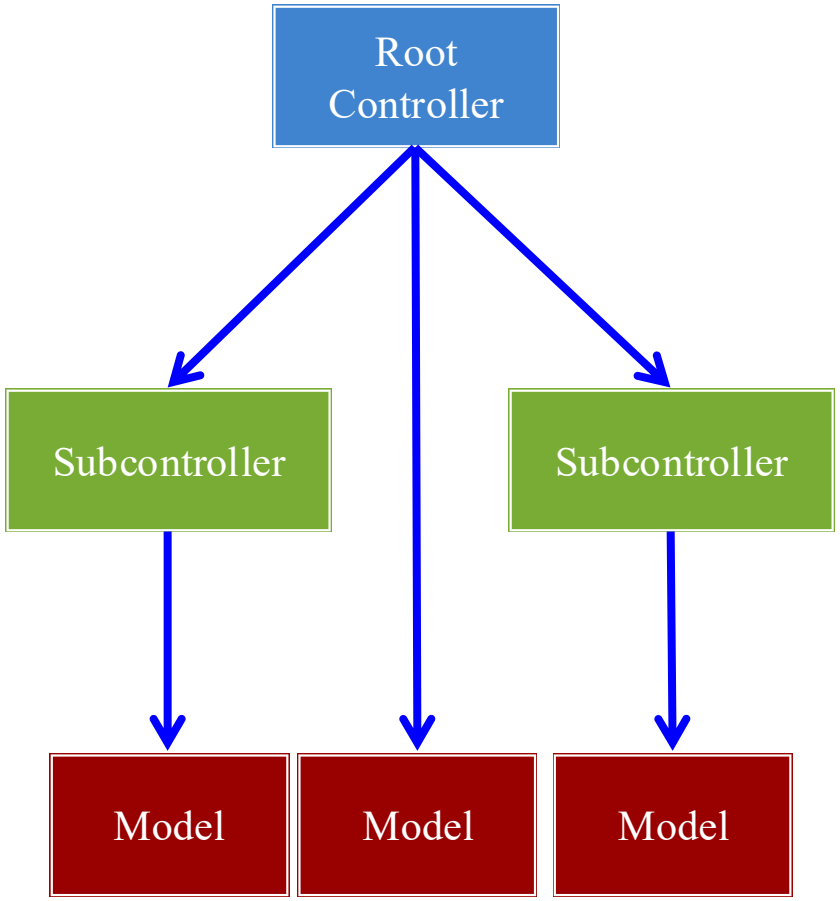
- Does not need to be paged
  - Drop the latest frame
  - Restart on frame boundary
- Want size reasonably **fixed**
  - Local variables always are
  - Limited # of allocations
  - Limit `new` inside loops
- Make use of **cached objects**
  - Requires careful planning

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**

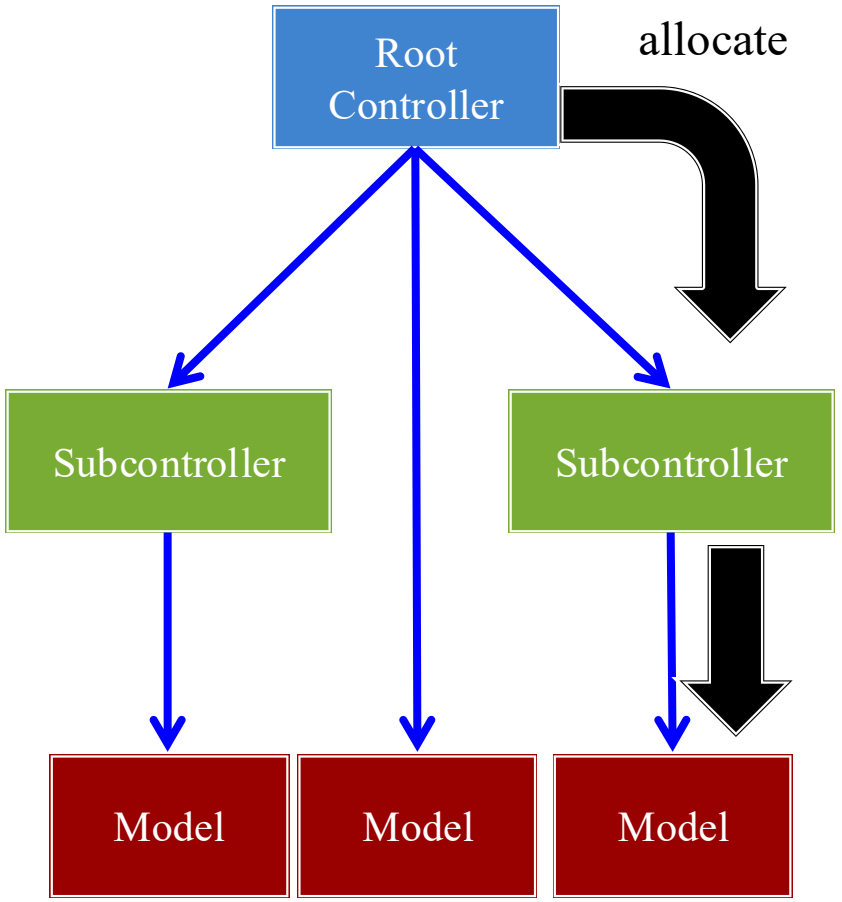
Rule of Thumb: Limiting `new`

- 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



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

```
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(s2.getVeloci
ty()));
```

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

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

    …

    // Compute the projections
    temp1.set(s2.getPosition());

temp1.sub(s1.getPosition()).nor(
);
    temp2.set(s1.getPosition());

temp2.sub(s2.getPosition()).nor(
);

    // Compute new velocities

temp1.scl(temp1.dot(s1.getVeloci
ty()));

temp2.scl(temp2.dot(s2.getVeloci
ty()));
```

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**

```
// 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"
```

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

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

…

// Allocate a new
particle
Particle q;

if
(!freelist.isEmpty())
{
    q =
freelist.pop();
} else {
    q = new
```

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

LibGDX Support: `Pool`

`Pool<T>`

- `public void free(T obj);`
  - Add an object to free list
- `public`
  - Use this
  - If object
  - 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
  - object freed
  - emented by `T`
  - ee constructors
  - Set contents with initializers
- See MemoryPool demo
  - Also `PooledList` in Lab 4

But *deprecated* in latest release of LibGDX

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**