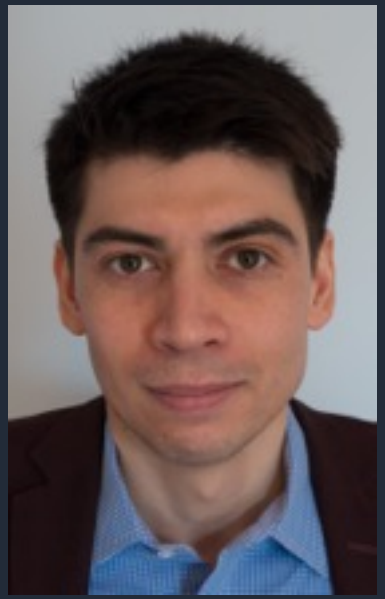
unity

Optimizing Mobile Applications

Us

Mark



Ian

unity

About This Talk

- Getting Good Data
- General Best Practices
- Common Problems & Solutions
  - Memory Usage
  - CPU Performance

Profiling

Use The Best Tools

- iOS: Instruments
- Android: VTune, Snapdragon Profiler
- Unity Editor
  - Timeline
- 5.3: Memory Profiler

Instruments!

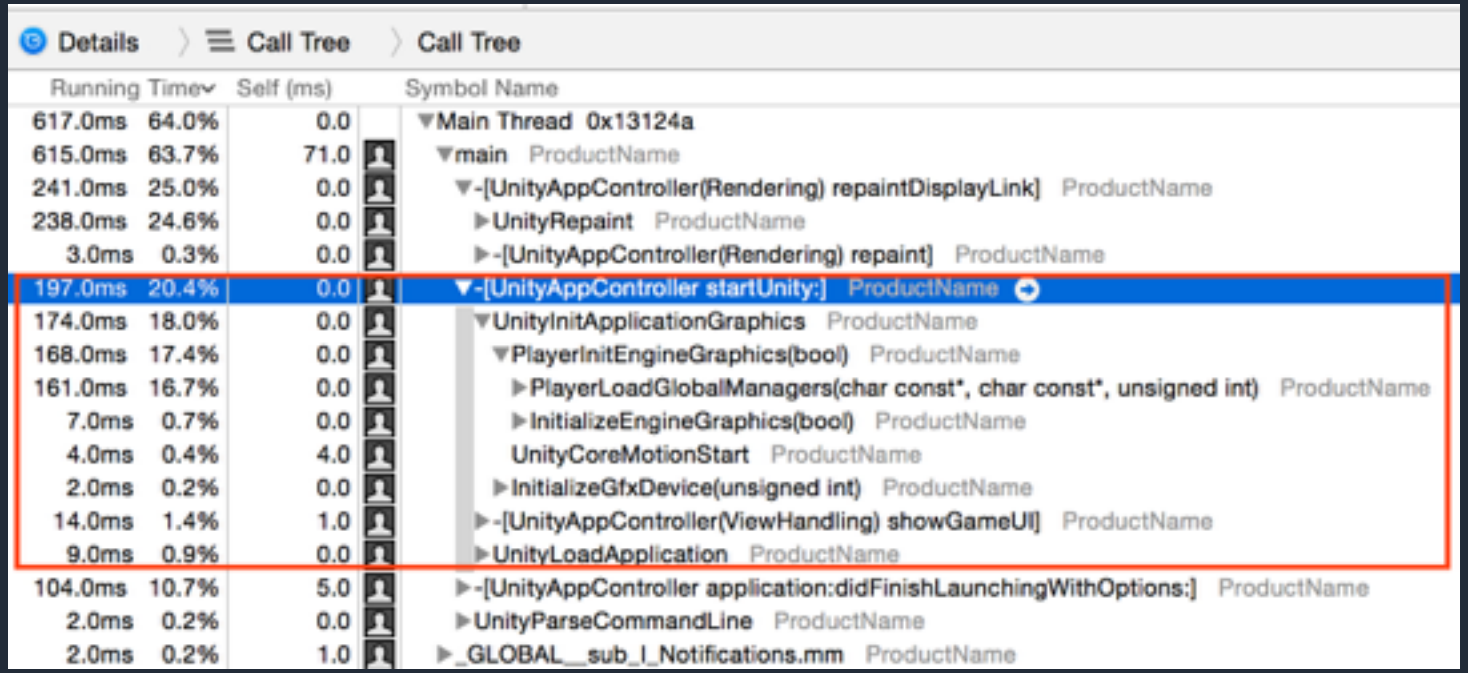
- Free, included with XCode
- Works perfectly with Unity IL2CPP builds
- Best tool for mobile CPU profiling
- Best tool for startup time profiling

Instruments! (2)

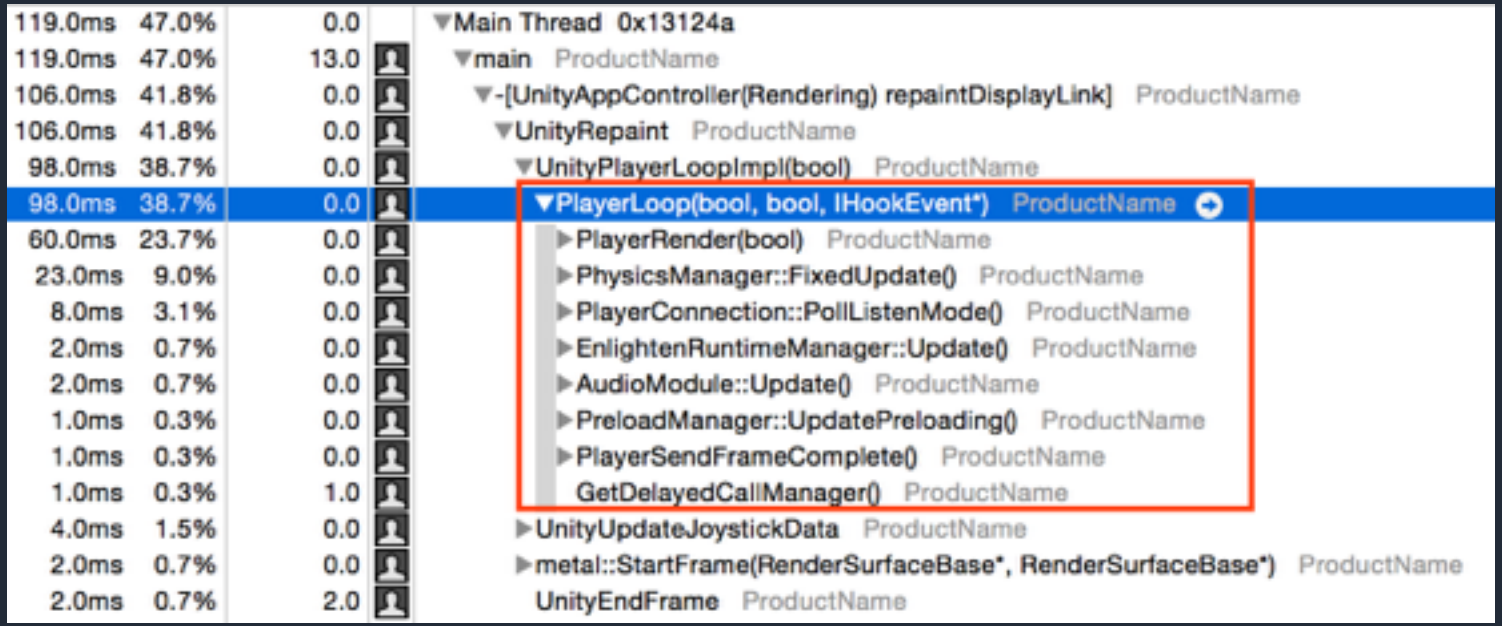
- Instructions on how to run it:

- http://blogs.unity3d.com/2016/02/01/profiling-with-instruments/

Instruments CPU Profiler: Startup Time



Instruments CPU Profiler: Runtime



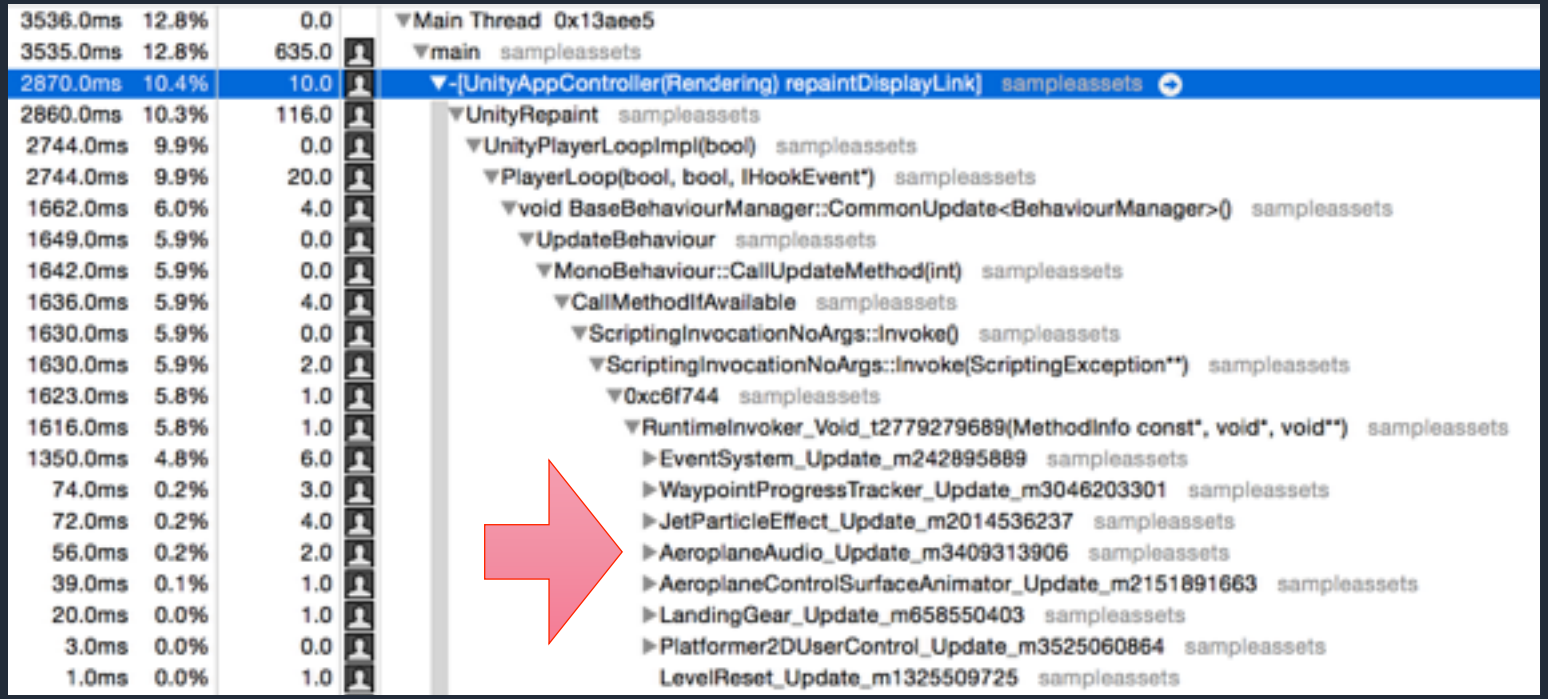
Instruments: Reading the PlayerLoop

- *PlayerRender*
  - Draw calls, batching, OnWillRender & image effect callbacks
- *BaseBehaviourManager::CommonUpdate*
  - Update, FixedUpdate and LateUpdate callbacks

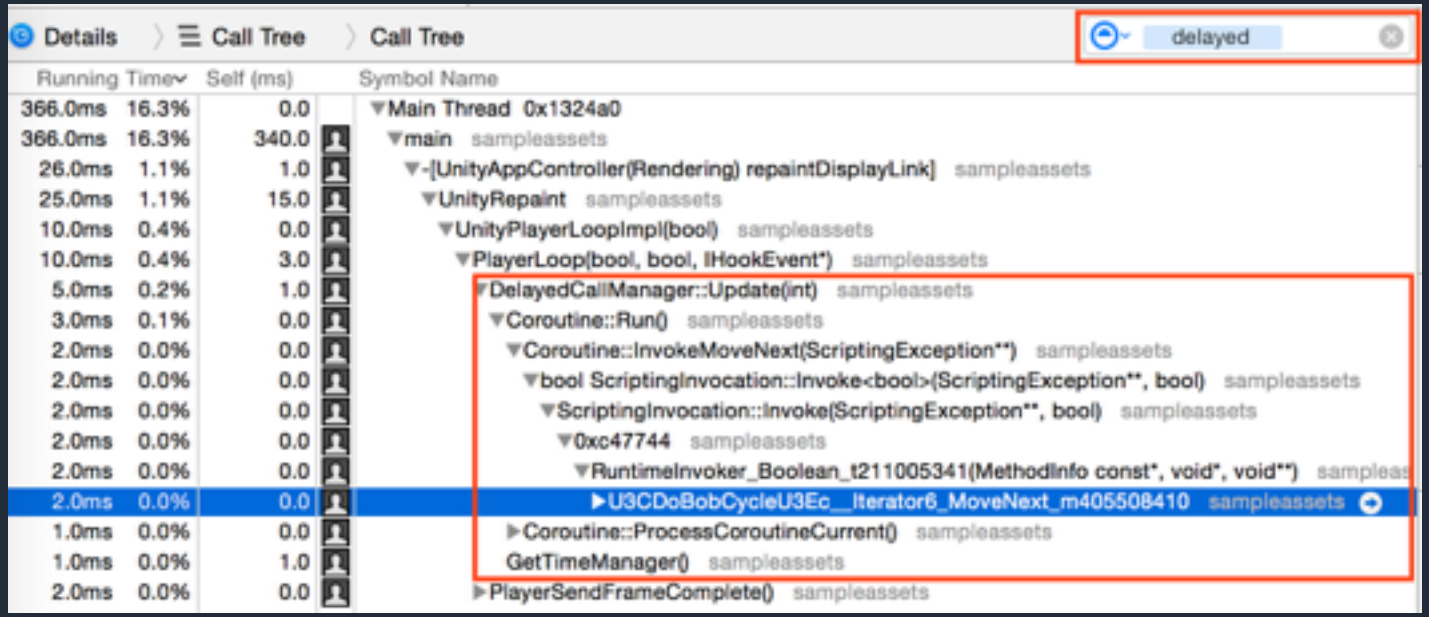
Instruments: Reading the PlayerLoop (2)

- *UI::CanvasManager (several methods)*
  - UI canvas rebatching, text mesh generation, etc.
- *DelayedCallManager::Update*
  - Resumed coroutines
- *PhysicsManager::FixedUpdate*
  - PhysX simulation, OnCollision* and OnTrigger* callbacks

Instruments: Examining a Callback



Instruments: Examining a Coroutine

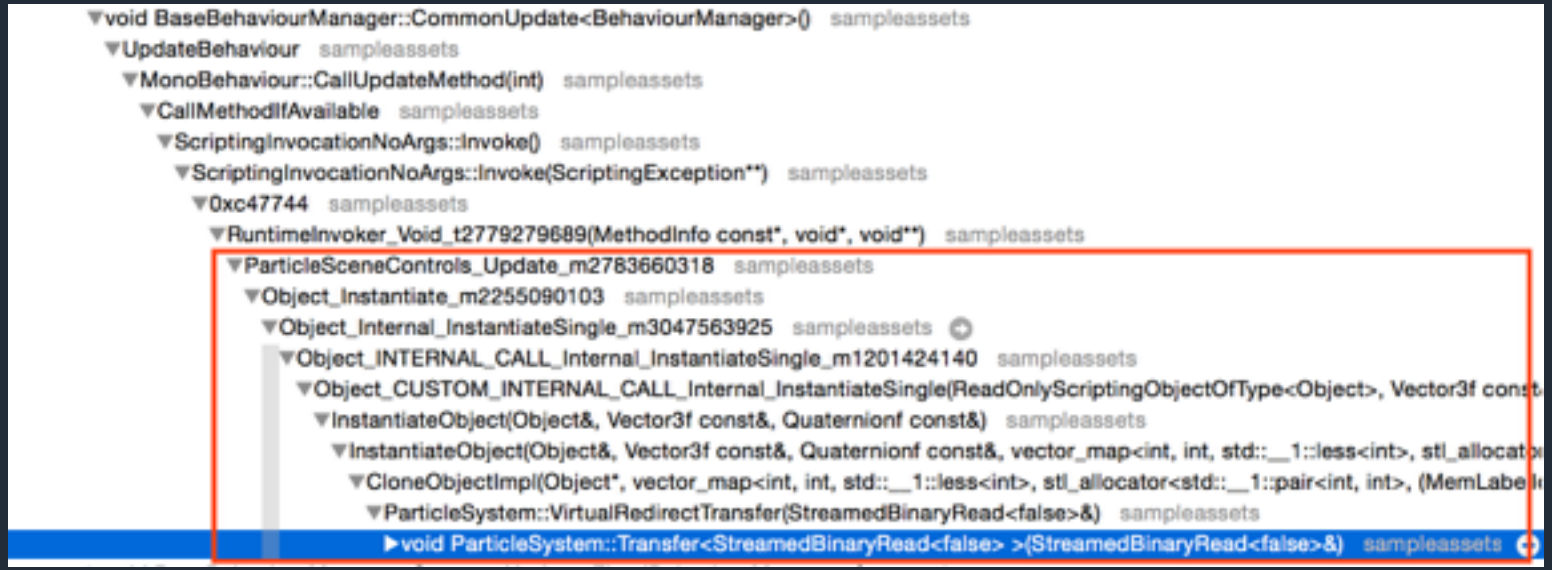


unity

Instruments: Coroutines (2)

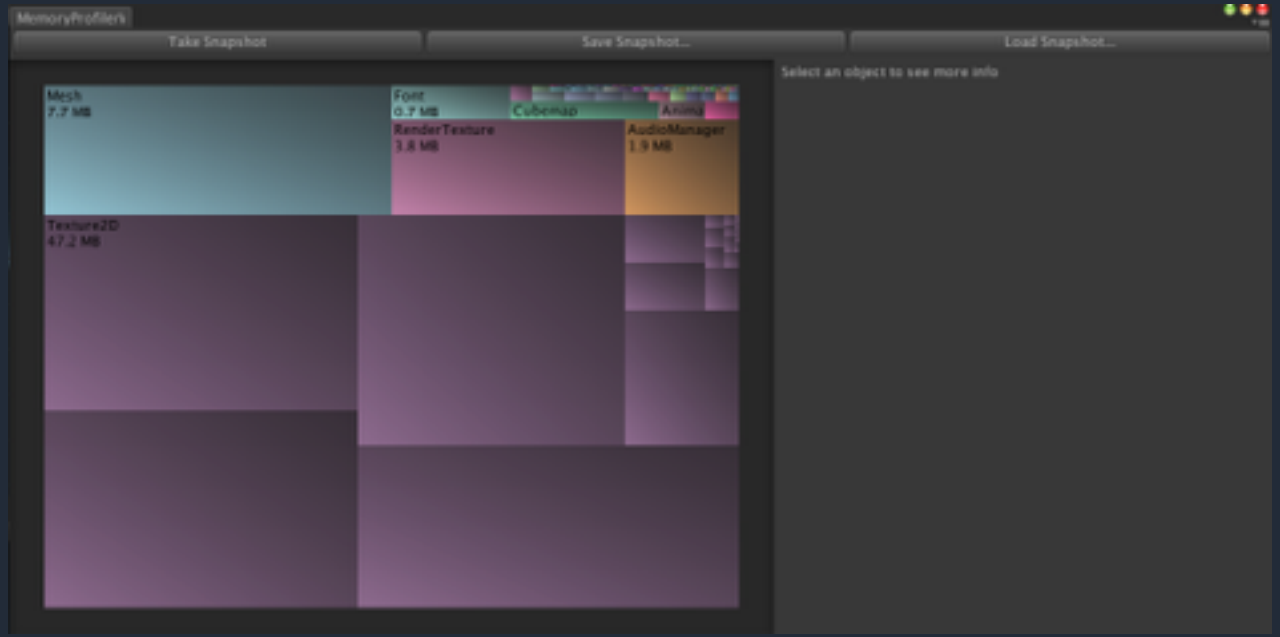
- Coroutine execution is split between two places:
  - The method where the coroutine was started.
    - i.e. where StartCoroutine() was called
  - DelayedCallManager
- StartCoroutine runs all code until the first yield
- DelayedCalledManager runs the rest

Instruments: Identifying Asset Loads



unity

5.3 Memory Profiler



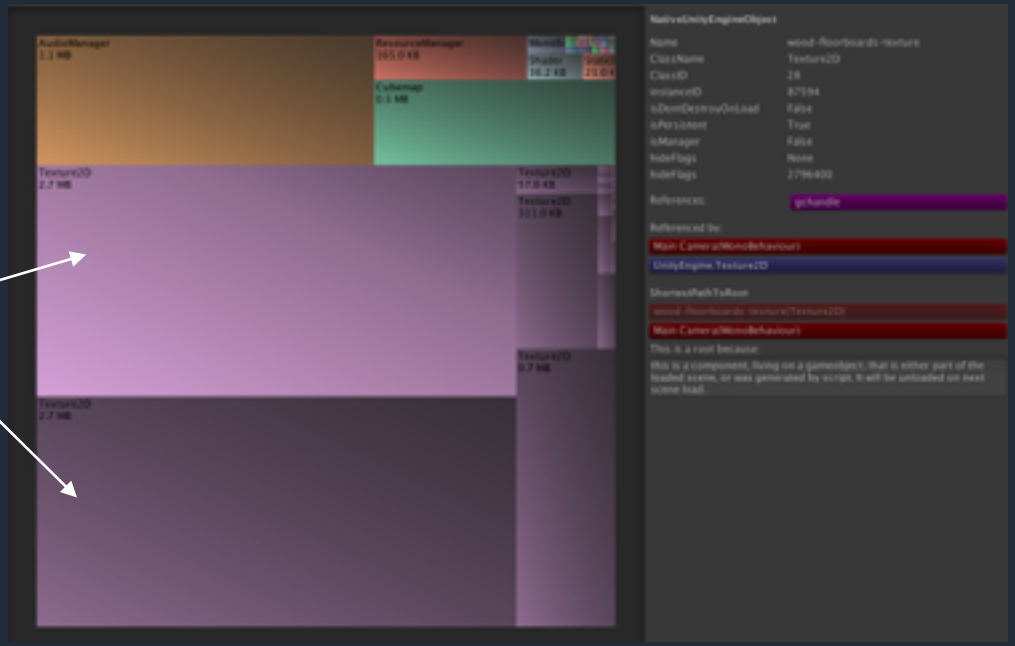
unity

5.3 Memory Profiler

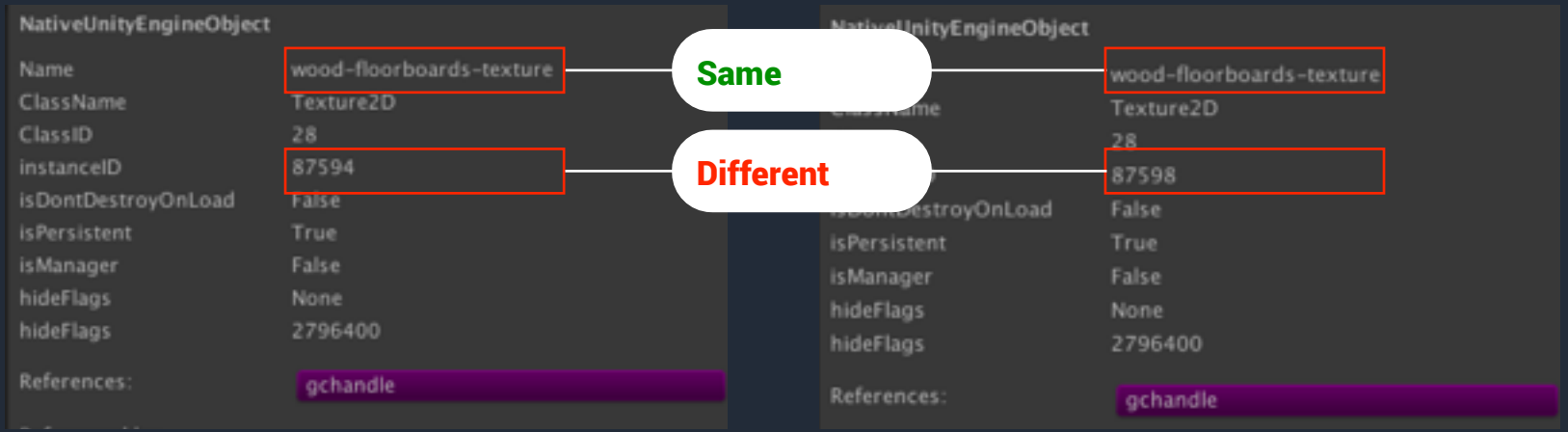
- Download code from Bitbucket
  - http://bitbucket.org/Unity-Technologies/MemoryProfiler/
- Drop into an *Editor* folder inside *Assets*
- In Unity Editor: Window > MemoryProfilerWindow
- Connect Unity Profiler via Profiler Window
- Click “Take Snapshot”

5.3 Memory Profiler: Duplicated Textures

Examine these

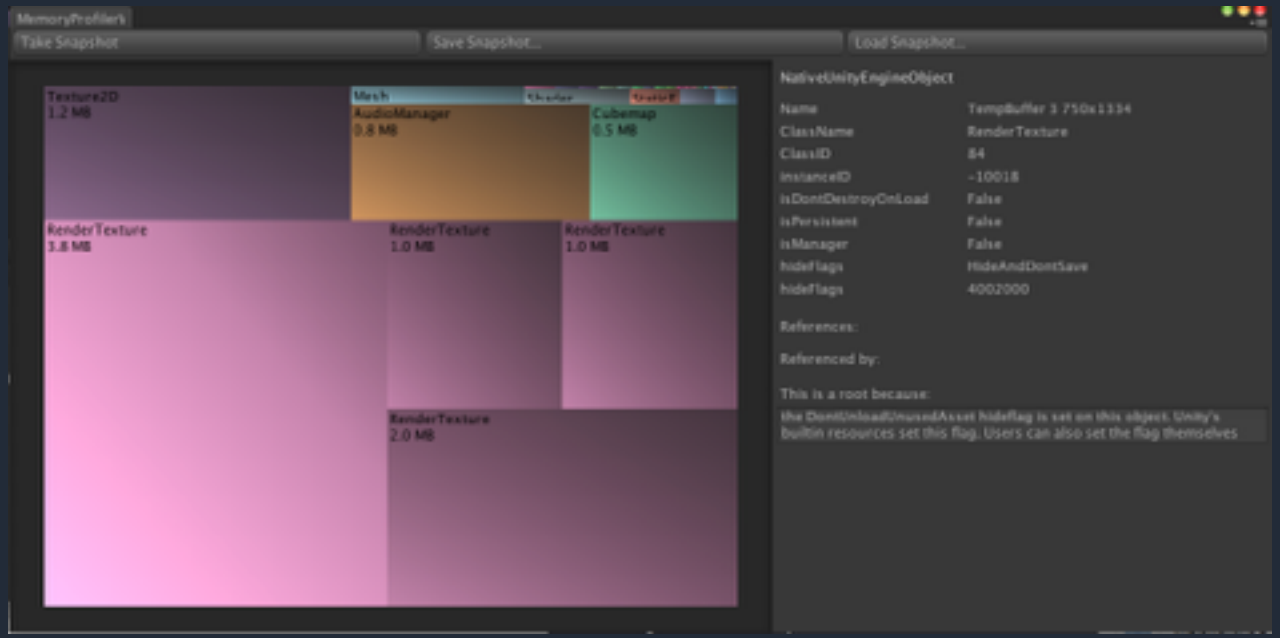


5.3 Memory Profiler: Duplicated Textures



Same Texture, Different Instances

5.3 Memory Profiler



Assets

Asset Auditing: Preventing Mistakes

- Developers are people (arguably)
- People make mistakes
- Mistakes cost dev time

- Write tools to prevent common, costly errors

Asset Auditing: Common Errors

- Insane texture sizes
- Asset compression
- Improper Avatar/Rig settings

- Different rules for different parts of project

Asset Auditing: HOWTO

```
public class AssetAuditExample : AssetPostprocessor {

   public void OnPreprocessTexture() {
      // …
   }

   public void OnPreprocessModel() {
      // …
   }
}
```

Asset Auditing: HOWTO (2)

- AssetPostprocessor classes receive callbacks on import
- Implement *OnPreprocess** methods
- Apply your project's rules to *assetImporter* instance

Asset Auditing: HOWTO (3)

```
public class ReadOnlyModelPostprocessor : AssetPostprocessor {

  public void OnPreprocessModel() {
    ModelImporter modelImporter = (ModelImporter)assetImporter;
    if(modelImporter.isReadable) {
      modelImporter.isReadable = false;
      modelImporter.SaveAndReimport();
    }
  }
}
```

Common Rules: Textures

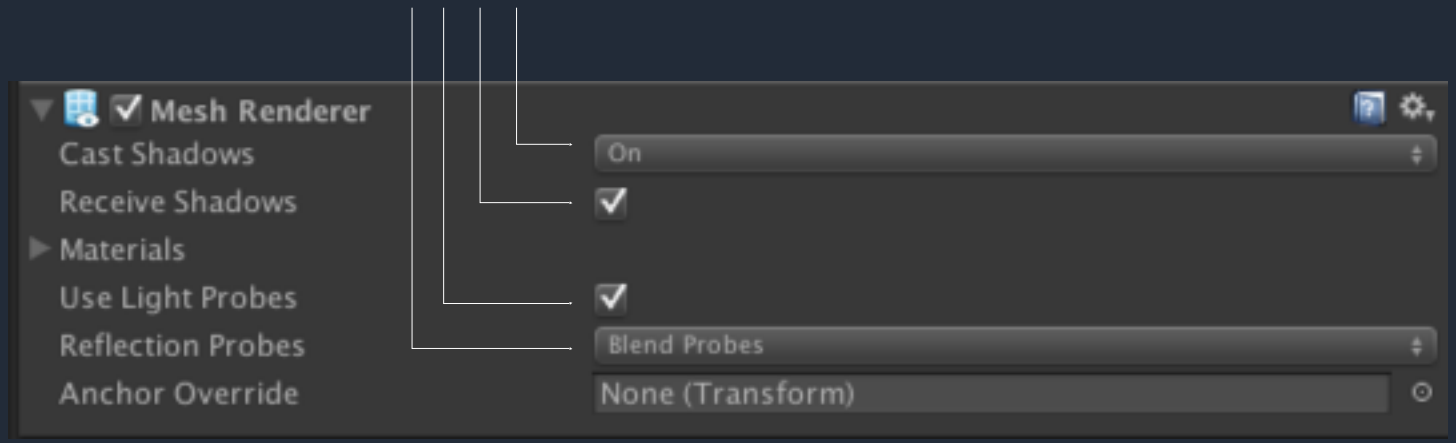
- Make sure Read/Write is disabled
- Disable mipmaps if possible
- Make sure textures are Compressed
- Ensure sizes aren’t too large
  - 2048x2048 or 1024x1024 for UI atlases
  - 512x512 or smaller for model textures

Common Rules: Models

- Make sure Read/Write is disabled
- Disable rig on non-character models
- Copy avatars for characters with shared rigs
- Enable mesh compression

Side Note: Mesh Renderer

Do you need these on?



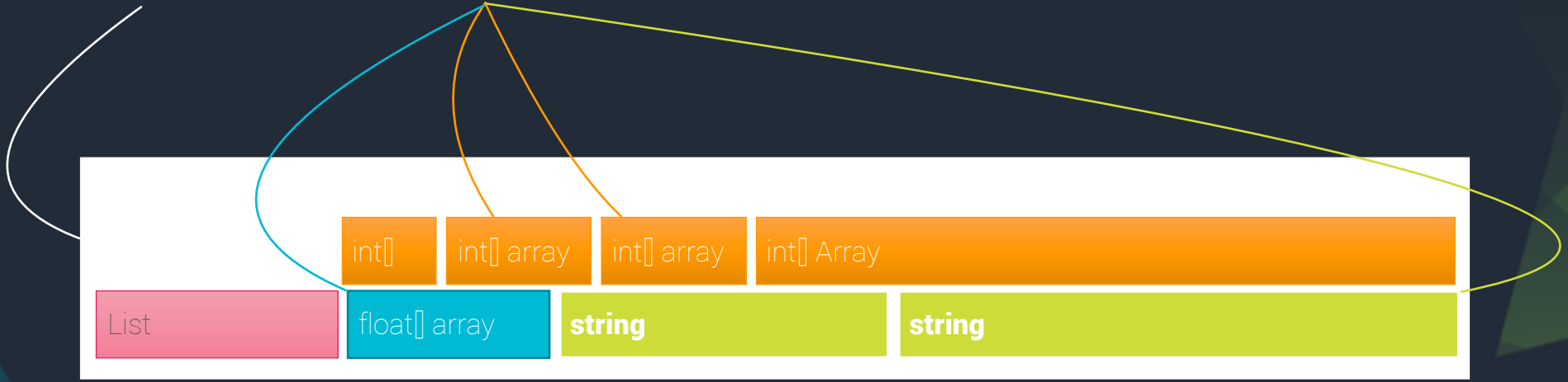
Common Rules: Audio

- MP3 compression on iOS
- Vorbis compression on Android
- “Force Mono” for mobile games
- Set bitrate as low as possible

Memory in Unity

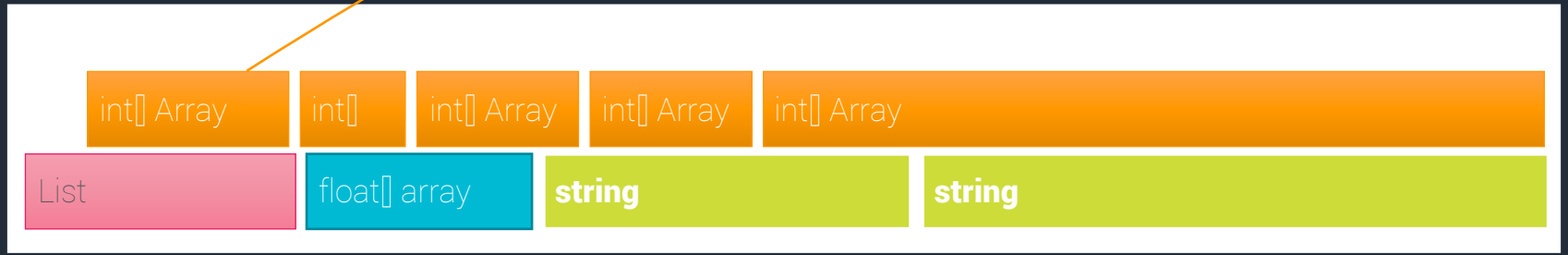
Managed Memory: How It Works

Heap contains objects allocated for Assets and Scripts



Managed Memory: How It Works

More memory is allocated when requested by code.
int[] someNumbers = new int[2048];

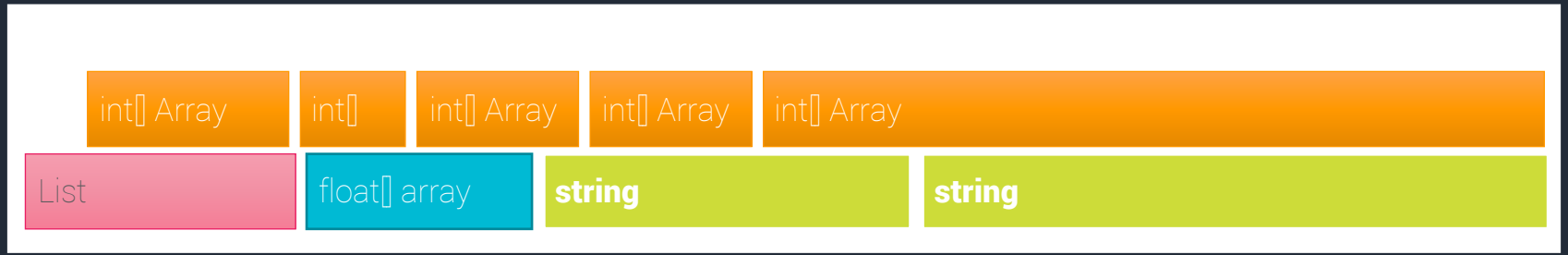


Managed Memory: How It Works

Garbage collector runs periodically, looks for unused objects.

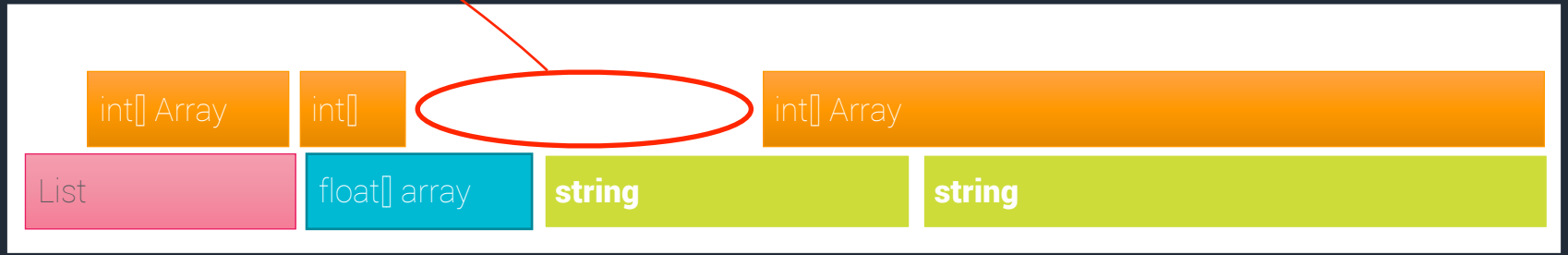
Unused objects are deleted.

GC.Collect();



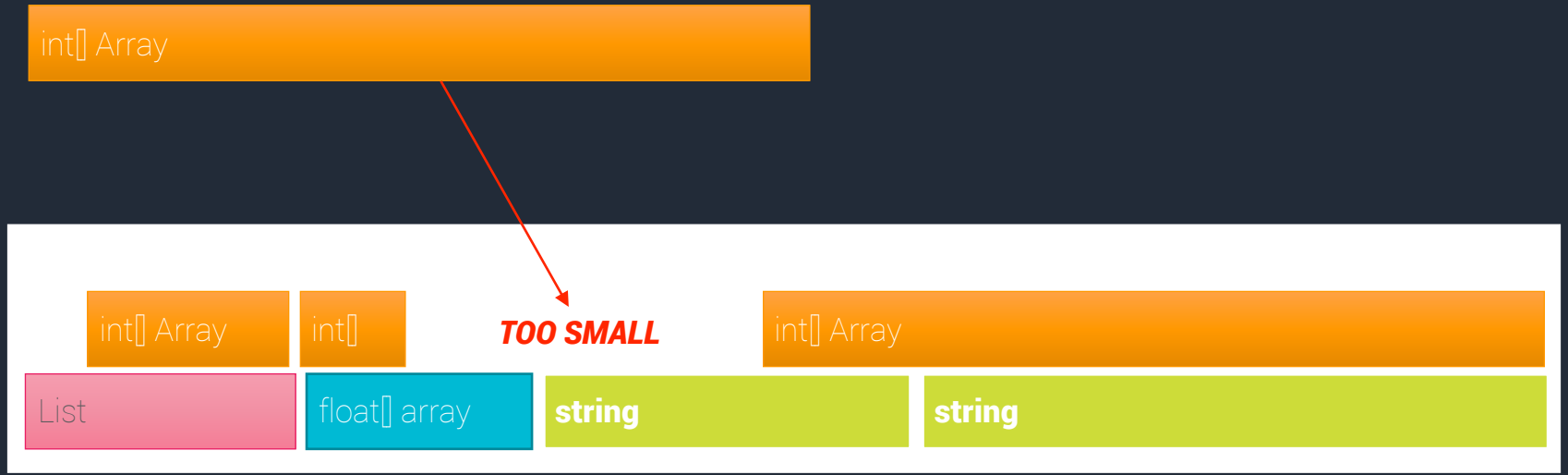
Managed Memory: How It Works

Holes are not filled. This is Memory Fragmentation.



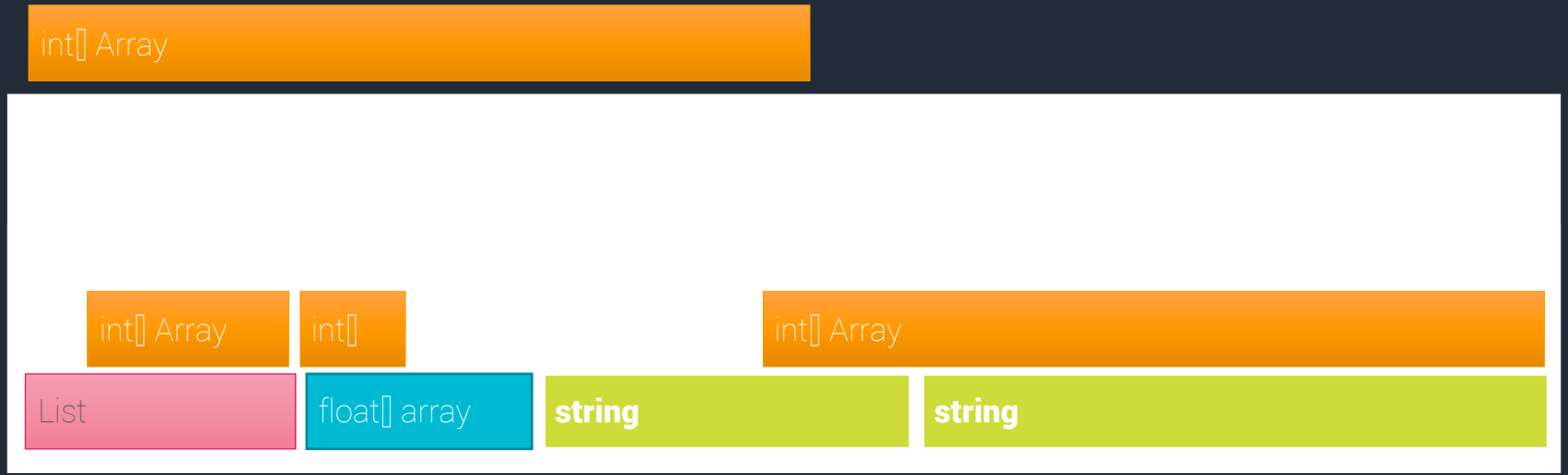
Managed Memory: How It Works

When there isn't enough space for new objects...



Managed Memory: How It Works

The heap expands.



Managed Memory: Problems

- In Unity, the heap only expands. It never shrinks.
- iOS & Android still care about reserved pages.
- Detail: Unused blocks of the heap remain reserved, but are paged out of the working set.

Managed Memory: Problems (2)

- Temporary memory allocations are really bad.
- 1 kilobyte of allocation per frame, 60 FPS
  - = 60 kilobytes per second of allocation
- If GC runs once per minute (BAD for framerate)...
- 3600 kilobytes of memory needed!

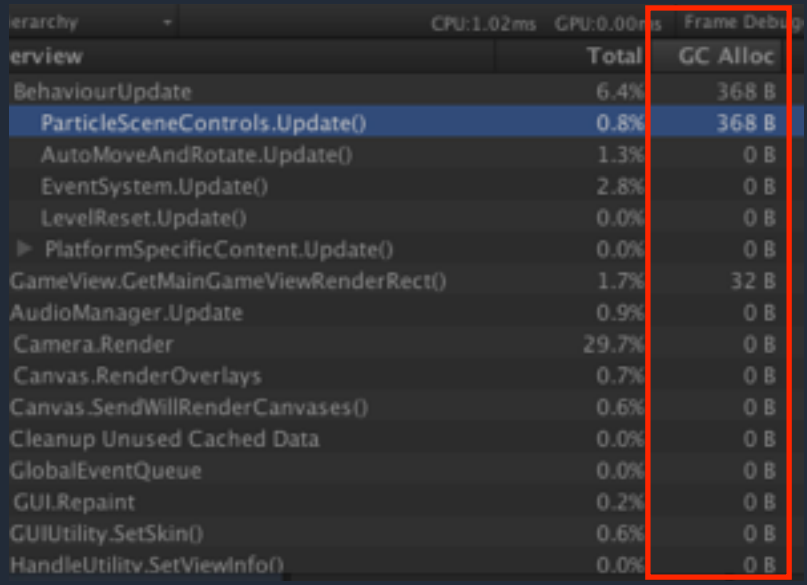
Tracking Managed Memory Allocations

Use Unity Profiler.

Sort by “GC Alloc” column.

When user can interact with app, stay as close to zero as possible.

(During loading, allocations aren’t as bad.)



Memory Conservation

- Reuse Collections (Lists, HashSets, etc.)
- Avoid string concatenation
  - Reuse StringBuilders to compose strings
- Avoid closures & anonymous methods

Memory Conservation: Boxing

- Happens when passing a value type as a reference type.
- Value is temporarily allocated on the heap.

- Example:

```
int x = 1;
object y = new object();
y.Equals(x);                    // Boxes “x” onto the heap
```

Memory Conservation: Boxing (2)

- Also happens when using enums as Dictionary keys
- Example:

```
enum MyEnum { a, b, c };
var myDictionary = new Dictionary<MyEnum, object>();

myDictionary.Add(MyEnum.a, new object()); // Boxes value “MyEnum.a”
```

- Workaround: Implement IEqualityComparer class

Memory Conservation: Foreach

- Allocates a Enumerator when loop begins
- Specific to Unity's version of Mono

- Just don't use it.

Memory Conservation: Unity APIs

- If a Unity API returns an array, it allocates a new copy.
- Every time it is accessed.
- Even if the values do not change.

Memory Conservation: Unity APIs (2)

- This code allocates many Touch[] arrays.

```
for ( int i = 0; i < Input.touches.Length; i++ )
{
   Touch touch = Input.touches[i];
    // ...
}
```

Memory Conservation: Unity APIs (3)

- This code allocates only one copy of the Touch[] array.

```
Touch[] touches = Input.touches;
for ( int i = 0; i < touches.Length; i++ )
{
   Touch touch = touches[i];
   // ...
}
```

Memory Conservation: Unity APIs (4)

- Some APIs have allocationless versions.

```
int touchCount = Input.touchCount;
for ( int i = 0; i < touchCount; i++ )
{
   Touch touch = Input.GetTouch(i);
   // …
}
```

CPU Performance Tips: Loading

XML, JSON & other text formats

- Parsing text is very slow.
- Avoid parsers built on Reflection — extremely slow.
  - In 5.3: Use Unity's JsonUtility class!

- Three strategies to speed up data parsing.

XML/JSON: Reduce Workload

- Strategy 1: Don’t parse text.
- Bake text data to binary
  - Use ScriptableObject

- Useful for data that does not change often.
  - e.g. Game design parameters

XML/JSON: Reduce Workload (2)

- Strategy 2: Do less work.

- Split data into smaller chunks.
- Parse only the parts that are needed.
- Cache parsing results for later reuse.

XML/JSON: Reduce Workload (3)

- Strategy 3: Threads.

- Pure C# types only.
- No Unity Assets (ScriptableObjects, Textures, etc.)
- Be VERY careful.

The Resources Folder

- An index of Resources is loaded at startup.
- Cannot be avoided or deferred.

- Solution: Move assets from Resources to Asset Bundles.

CPU Performance Tips: Runtime

Easy: Material/Animator/Shader Properties

- Never address Material, Shader, or Animator properties by name.
- Internally, hashes the property name into an integer.
- Don't do this:

```
material.SetColor("_Color", Color.white);
animator.SetTrigger("attack");
```

Cached Material/Animator/Shader Properties

- Do hashing at startup, cache results and reuse them.

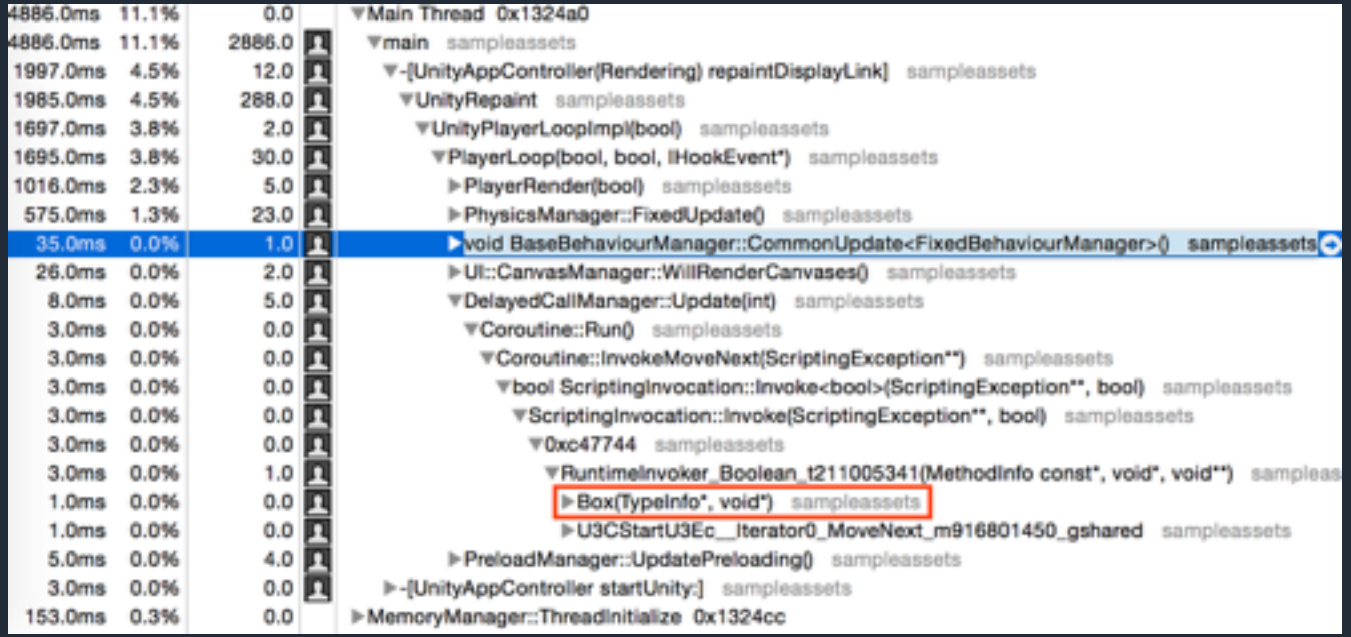
```
static readonly int material_Color = Shader.PropertyToID("_Color");
static readonly int anim_Attack = Animator.StringToHash("attack");

material.SetColor(material_Color, Color.white);
animator.SetTrigger(anim_Attack);
```

Boxing

- So irritating we had to mention it twice.
- Instruments: Search for “::Box”

Instruments: Identifying Boxing



Strings

Some people, when confronted with a problem, think "I know, I'll use regular expressions."

Now they have two problems.

- JWZ

Regular Expressions

- Avoid RegExps whenever possible
- RegExps generate *tons* of temporary garbage
- `Regex.Match("foo", "bar")` = 5 kilobytes

Precompile the RegExps

- Don't: Regex.Match(myString, "foo");
  - 5 kilobyte allocation
- Do:
  - var myRegExp = new Regex("foo"); myRegExp.Match(myString);
  - 320 byte allocation

String Comparisons

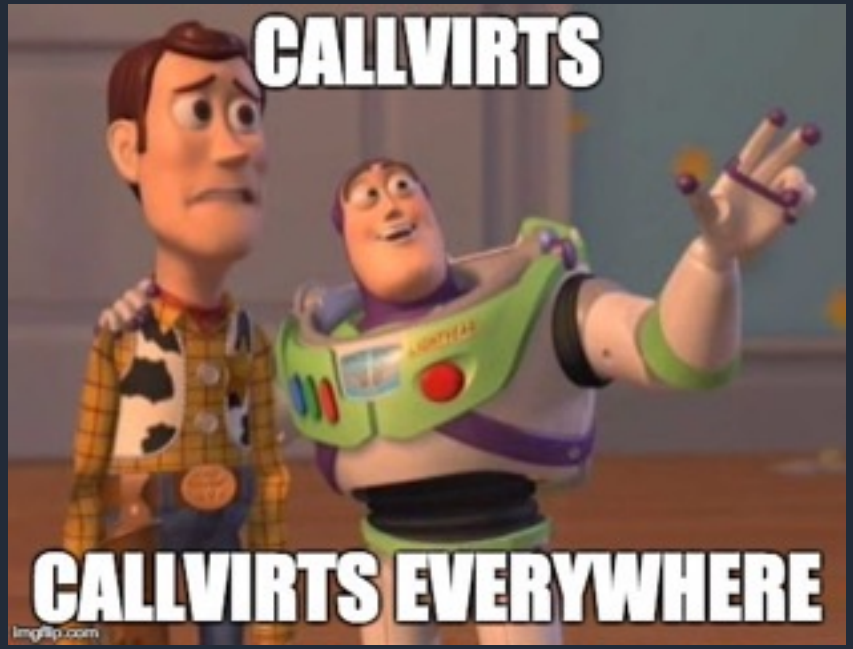
- Locale coercion is slow
  - ‘e’ matches the ‘æ’ ligature in the en-US culture
  - *String.Equals(“encyclopedia”, “encyclopædia”)*

- Ordinal comparison compares bytes
  - *myString.Equals(otherString, StringComparison.Ordinal);*

The String Library is Just Slow

- Do not use String.StartsWith, String.EndsWith
- 10-100x slower than a hand-coded bytewise replacement

- Why?


CALLVIRTS
CALLVIRTS EVERYWHERE
imgflip.com

A Tale of Two Calls

```
int Accum { get; set; }
Accum = 0;

for(int i = 0;
    i < myList.Count;
    i++) {
  Accum += myList[i];
}
```

```
int accum = 0;
int len = myList.Count;

for(int i = 0;
    i < len;
    i++) {
  accum += myList[i];
}
```

unity

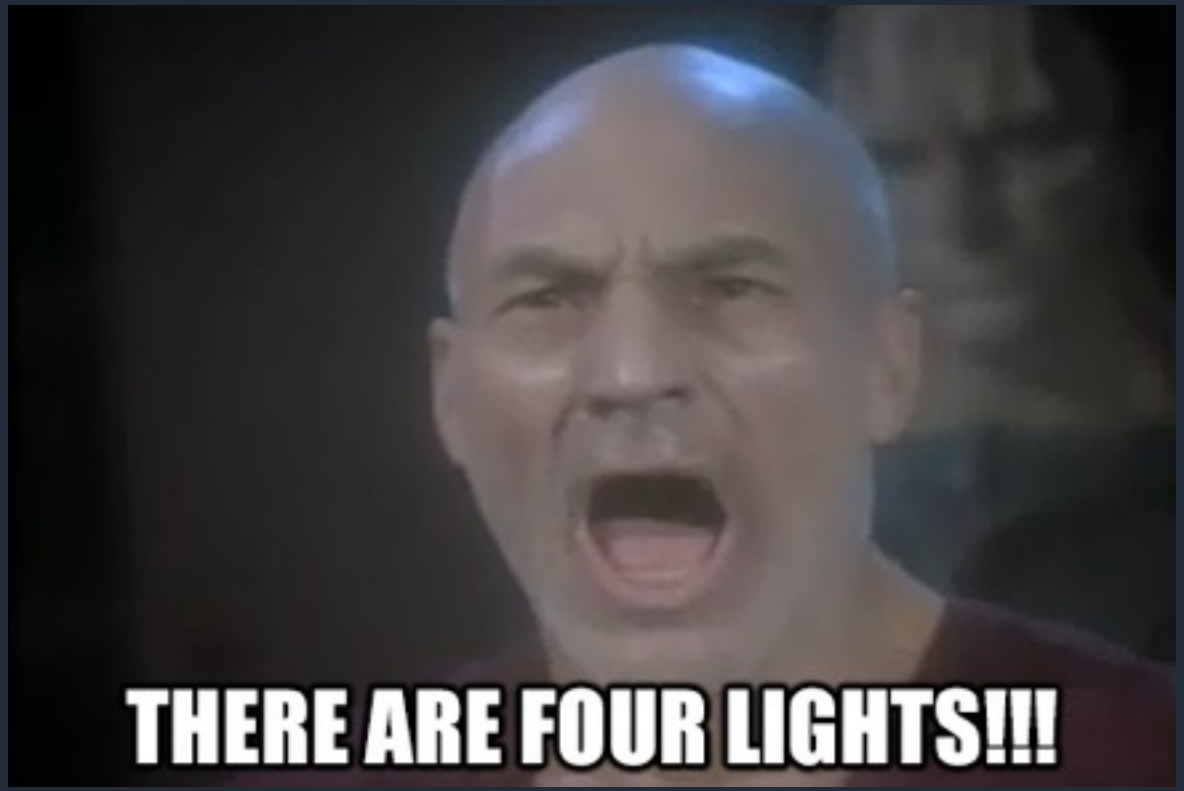
```
int Accum { get; set; }
Accum = 0;

for(int i = 0;
    i < myList.Count;
    i++) {
  Accum += myList[i];
}
```

List::get_Count

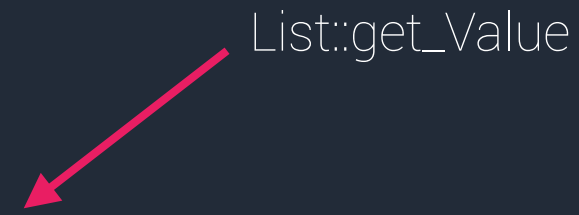
List::get_Value

get_Accum set_Accum


THERE ARE FOUR LIGHTS!!!

```
int accum = 0;
int len = myList.Count;

for(int i = 0;
    i < len;
    i++) {
  accum += myList[i];
}
```



Timings for 100,000 Iterations

```
int Accum { get; set; }
Accum = 0;

for(int i = 0;
    i < myList.Count;
    i++) {
  Accum += myList[i];
}
```

324 milliseconds

```
int accum = 0;
int len = myList.Count;
for(int i = 0;
    i < len;
    i++) {
  accum += myList[i];
}
```

128 milliseconds

Remember...

- Unity does minimal, if any, inlining.

- A call in the source is a callvirt in the IL.
- A callvirt in the IL is probably a call in the binary.

- Property accessors are always method calls.

Trivial properties are bad news

- Vector3.zero
  - get { return new Vector3(0,0,0); }
- Quaternion.identity
  - get { return new Quaternion(0, 0, 0, 1); }

- For simple types: make a `const`
- For complex types: make a `static readonly`

So are trivial methods.

- Quaternion.Set, Vector3.Scale
  - Do the math and assign the variables.

- Transform.Translate, Transform.Rotate?
  - Assign the position/rotation.

Remember...

- Profile before optimizing.
- Apply these techniques only when needed.

Thanks for listening!

unity