Optimizing Mobile Applications
Introduction
Us

Mark

Ian
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:

### Instruments CPU Profiler: Startup Time

<table>
<thead>
<tr>
<th>Running Time</th>
<th>Self (ms)</th>
<th>Symbol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>617.0ms</td>
<td>64.0%</td>
<td>▼Main Thread 0x13124a</td>
</tr>
<tr>
<td>615.0ms</td>
<td>63.7%</td>
<td>▼main ProductName</td>
</tr>
<tr>
<td>241.0ms</td>
<td>25.0%</td>
<td>▼[UnityAppController(Rendering) repaintDisplayLink] ProductName</td>
</tr>
<tr>
<td>238.0ms</td>
<td>24.6%</td>
<td>▼UnityRepaint ProductName</td>
</tr>
<tr>
<td>3.0ms</td>
<td>0.3%</td>
<td>▼-[UnityAppController(Rendering) repaint] ProductName</td>
</tr>
<tr>
<td>197.0ms</td>
<td>20.4%</td>
<td>▼[UnityAppController startUnity:] ProductName</td>
</tr>
<tr>
<td>174.0ms</td>
<td>18.0%</td>
<td>▼UnityInitApplicationGraphics ProductName</td>
</tr>
<tr>
<td>168.0ms</td>
<td>17.4%</td>
<td>▼PlayerInitEngineGraphics(bool) ProductName</td>
</tr>
<tr>
<td>161.0ms</td>
<td>16.7%</td>
<td>▼PlayerLoadGlobalManagers(char const*, char const*, unsigned int) ProductName</td>
</tr>
<tr>
<td>7.0ms</td>
<td>0.7%</td>
<td>▼InitializeEngineGraphics(bool) ProductName</td>
</tr>
<tr>
<td>4.0ms</td>
<td>0.4%</td>
<td>UnityCoreMotionStart ProductName</td>
</tr>
<tr>
<td>2.0ms</td>
<td>0.2%</td>
<td>▼InitializeGfxDevice(unsigned int) ProductName</td>
</tr>
<tr>
<td>14.0ms</td>
<td>1.4%</td>
<td>▼-[UnityAppController(ViewHandling) showGameUI] ProductName</td>
</tr>
<tr>
<td>9.0ms</td>
<td>0.9%</td>
<td>▼UnityLoadApplication ProductName</td>
</tr>
<tr>
<td>104.0ms</td>
<td>10.7%</td>
<td>▼-[UnityAppController application:didFinishLaunchingWithOptions:] ProductName</td>
</tr>
<tr>
<td>2.0ms</td>
<td>0.2%</td>
<td>▼UnityParseCommandLine ProductName</td>
</tr>
<tr>
<td>2.0ms</td>
<td>0.2%</td>
<td>▼GLOBAL__sub_I_Notifications.mm ProductName</td>
</tr>
</tbody>
</table>
Instruments CPU Profiler: Runtime

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Time (ms)</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Thread 0x13124a</td>
<td>119.0</td>
<td>47.0</td>
</tr>
<tr>
<td>main</td>
<td>119.0</td>
<td>47.0</td>
</tr>
<tr>
<td>-UnityAppController(Rendering) repaaintDisplayLink</td>
<td>106.0</td>
<td>41.8</td>
</tr>
<tr>
<td>UnityRepaint</td>
<td>106.0</td>
<td>41.8</td>
</tr>
<tr>
<td>UnityPlayerLoopImpl(bool)</td>
<td>98.0</td>
<td>38.7</td>
</tr>
<tr>
<td>PlayerLoop(bool, bool, IHookEvent*)</td>
<td>98.0</td>
<td>38.7</td>
</tr>
<tr>
<td>PlayerRender(bool)</td>
<td>60.0</td>
<td>23.7</td>
</tr>
<tr>
<td>PhysicsManager::FixedUpdate()</td>
<td>23.0</td>
<td>9.0</td>
</tr>
<tr>
<td>PlayerConnection::PollListenModel()</td>
<td>8.0</td>
<td>3.1</td>
</tr>
<tr>
<td>EnlightenRuntimeManager::Update()</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td>AudioModule::Update()</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td>PreloadManager::UpdatePreloading()</td>
<td>1.0</td>
<td>0.3</td>
</tr>
<tr>
<td>PlayerSendFrameComplete()</td>
<td>1.0</td>
<td>0.3</td>
</tr>
<tr>
<td>GetDelayedCallManager()</td>
<td>1.0</td>
<td>0.3</td>
</tr>
<tr>
<td>UnityUpdateJoystickData</td>
<td>4.0</td>
<td>1.5</td>
</tr>
<tr>
<td>metal::StartFrame(RenderSurfaceBase*, RenderSurfaceBase*)</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td>UnityEndFrame</td>
<td>2.0</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Instruments: Reading the PlayerLoop

- `BaseBehaviourManager::CommonUpdate`
  - `Update`, `FixedUpdate` and `LateUpdate` callbacks
- `PhysicsManager::FixedUpdate`
  - PhysX simulation, `OnCollision*` and `OnTrigger*` callbacks
  - `Physics2DManager::FixedUpdate` if using 2D physics
- `DelayedCallManager::Update`
  - Resumed coroutines
Instruments: Reading the PlayerLoop (2)

- **PlayerRender**
  - Draw calls, batching, `OnWillRender` & image effect callbacks
- **UI::CanvasManager::WillRenderCanvases**
  - UI canvas rebatching, text mesh generation, etc.
- **EnlightenRuntimeManager::Update**
  - Enlighten, precomputed realtime GI, reflection probes
**Instruments: Examining a Callback**

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Percent</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Thread 0x13aee5</td>
<td>3536.0ms</td>
<td>12.8%</td>
<td>3535.0ms</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>3535.0ms</td>
<td>12.8%</td>
<td>635.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>2870.0ms</td>
<td>10.4%</td>
<td>2860.0ms</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>2860.0ms</td>
<td>10.3%</td>
<td>2744.0ms</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>2744.0ms</td>
<td>9.9%</td>
<td>2744.0ms</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1662.0ms</td>
<td>6.0%</td>
<td>4.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1649.0ms</td>
<td>5.9%</td>
<td>4.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1642.0ms</td>
<td>5.9%</td>
<td>2.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1636.0ms</td>
<td>5.9%</td>
<td>2.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1630.0ms</td>
<td>5.9%</td>
<td>2.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1630.0ms</td>
<td>5.9%</td>
<td>2.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1623.0ms</td>
<td>5.8%</td>
<td>2.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1616.0ms</td>
<td>5.8%</td>
<td>1.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1350.0ms</td>
<td>4.8%</td>
<td>1.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>74.0ms</td>
<td>0.2%</td>
<td>4.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>72.0ms</td>
<td>0.2%</td>
<td>4.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>56.0ms</td>
<td>0.2%</td>
<td>2.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>39.0ms</td>
<td>0.1%</td>
<td>1.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>20.0ms</td>
<td>0.0%</td>
<td>1.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>3.0ms</td>
<td>0.0%</td>
<td>1.0</td>
</tr>
<tr>
<td>Main Thread 0x13aee5</td>
<td>1.0ms</td>
<td>0.0%</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Instruments: Examining a Coroutine
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: Summarizing Distributed Costs

- Enter method name into "Search" box

- Suggested searches:
  - "::Box", "Box(" and ")Box"
  - "String_"
Instruments: Identifying Asset Loads
5.3 Memory Profiler
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

![Memory Profiler Interface](image)

- **Texture2D**: 1.2 MB
- **AudioManager**: 0.8 MB
- **Cubemap**: 0.5 MB
- **RenderTexture**: 3.8 MB
  - 1.0 MB
  - 2.0 MB

**NativeUnityEngineObject**
- **Name**: TempBuffer 3 750x1334
- **ClassName**: RenderTexture
- **ClassID**: 84
- **instanceID**: -10018
- **isDontDestroyOnLoad**: False
- **isPersistent**: False
- **isManager**: False
- **hideFlags**: HideAndDontSave
- **hideFlags**: 4002000

**References**

- **Referenced by**:
  - This is a root because: the DontUnloadUnusedAsset hideflag is set on this object. Unity’s built-in resources set this flag. Users can also set the flag themselves.
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
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
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.

```csharp
int[] someNumbers = new int[2048];
```
Managed Memory: How It Works

Garbage collector runs periodically, looks for unused objects. Unused objects are deleted.

```csharp
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...

int[] Array

TOO SMALL

Mesh Audio Clip Texture #2 Texture #1
Managed Memory: How It Works

The heap expands.

- int[] Array
- string
- int[] Array
- Mesh
- Audio Clip
- Texture #2
- Texture #1
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:
```csharp
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.

```csharp
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.

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

- All GameObjects & Components in a prefab are serialized into the prefab’s data file.
- Includes all settings on all Components.

- 2 identical GameObjects in a prefab = 2 copies of data
Large Prefabs (2)

- For very large prefabs, split into smaller parts

- Use `Awake` callbacks to instantiate duplicated parts
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.

```csharp
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);
```
These are so expensive we had to mention them twice.
Slow: RegExps, String.StartsWith, String.EndsWith

Instruments:
- Search for "::Box" and "_Box"
- Search for "String_"
Instruments: Identifying Boxing
Canvases, Draw Calls and Batching

- Canvases “rebuild” their batches to reduce draw calls
  - Rebuilds are very expensive.

- A canvas rebuilds if any Drawable component changes.
  - Any visual change will force a canvas to rebuild.
  - Drawable = anything visible on a canvas.
Canvases, Draw Calls and Batching

• Cost of a rebuild based on number of elements.
• Includes children.
• Frequent changes + lots of UI = lots of CPU usage!
Reducing Batching Cost

- Reduce number of Drawables.
  - Merge sprites, merge text objects
- Split up Canvases
- Balance cost of draw calls and cost of batching.
Splitting Canvases

- Can nest a canvas within another canvas
- Nested canvases isolate their children from rebuilds.
- Guideline: Move elements that change every frame onto separate canvases from static elements.
Splitting Canvases: Example

Stamina

00:30s
Splitting Canvases: Example

Stamina

00:30s
Splitting Canvases: Example

Stamina

00:30s

Change Every Frame

Never Change
Splitting Canvases: Example

Stamina

00:30s

Dynamic Canvas

Background Canvas
Trampolines
Remember...

- **Profile before optimizing.**
- Apply these techniques only when needed.
How Unity Invokes Callbacks

- Internally, C++ Linked List of Components with callbacks
  - Update, LateUpdate, etc.
- Iterate over the Linked List and invoke each callback.
- Small overhead when invoking scripts
How Unity Invokes Callbacks

• If number of callbacks becomes very large, overhead can become significant.

• Thousands of callbacks: 10-20% of Update CPU time
Replacing Callbacks

- Remove `Update`, `LateUpdate`, etc.
- Make a GameObject with a Mono Behaviour
  - Implement `Update`, `LateUpdate`, and other callbacks on this object
- All other code: Subscribe to needed callbacks
Replacing Callbacks: UpdateManager object

```csharp
public class UpdateManager : MonoBehaviour {

    public static UpdateManager Instance { get; set; }
    void Awake() { Instance = this; }

    public UnityEvent OnUpdate = new UnityEvent();

    void Update() {
        OnUpdate.Invoke();
    }
}
```
Replacing Callbacks: Advantages

• Eliminates native-to-managed trampoline overhead.

• Objects can intelligently unsubscribe.
  • Don’t need to return out of Update callbacks!
  • Works well with pooled objects.
Thank you!