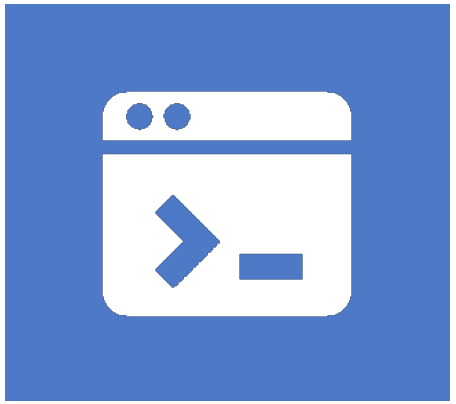
Advanced Game Programming



Week 7

Homework Review

Networking and Deferred Actions

Recursion

Basics and Practices

What is Recursion?

- A method is recursive if it calls itself.
- What happens here?

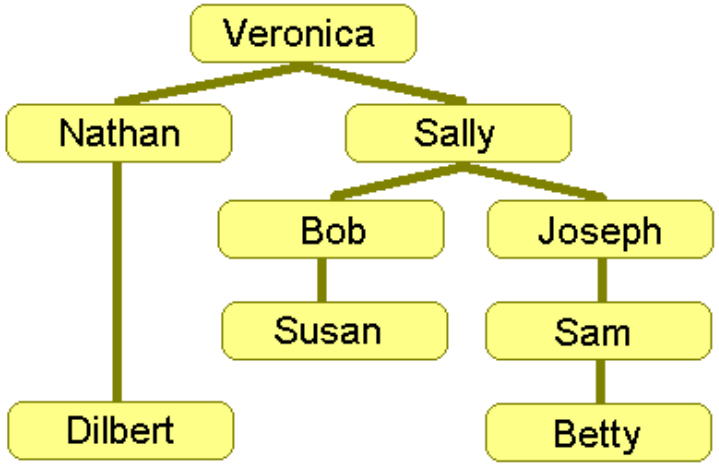
```
string PrintString(string toPrint, int index) {
    if (index >= toPrint.Length)
    {
        return "";
    }

    return toPrint[index] + "!\n" + PrintString(toPrint, index + 1);
}

void Start() {
    PrintString("Hello World");
}
```

Important Considerations

- Handle a simple “base case” without using recursion.
- Avoid cycles.
- Each call of the function represents a complete handling of the given task.

Why use Recursion?

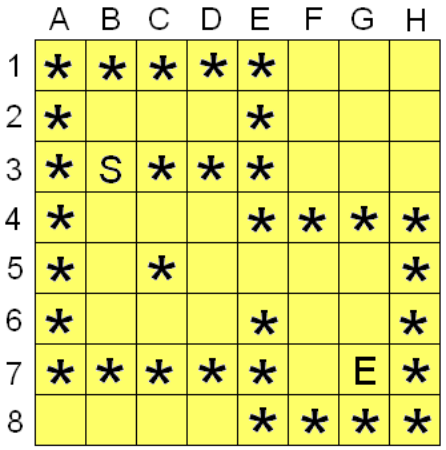
- Hierarchies
- Networks
- Graphs




```
int EmployeesUnderCount(string employeeName, Tree orgChart) {
    return EmployeesUnderCountRecursive(employeeName, orgChart);
}

void EmployeesUnderCountRecursive(string employeeName, Tree subTree) {
    if (root.childrenCount == 0) {
        return (root.manager == employeeName) ? 1 : 0;;
    }

    foreach (var child of subTree.root) {
        if (child.manager == employeeName) {
            counter += 1;
            return 1 + EmployeesUnderCountRecursive(child.name, child.subTree);
        }
        else
        {
            return EmployeesUnderCountRecursive(employeeName, child.subTree);
        }
    }
}
```

Additional Uses of Recursion

- Navigating space




```
private bool IsMazeSolvable(string[,] maze, int startX, int startY)
{
    return ExploreMaze(maze, startX, startY);
}

private bool ExploreMaze(string[,] maze, int x, int y)
{
    if (x < 0 || y < 0 || x >= maze.GetLength(0) || y >= maze.GetLength(1))
        return false;

    if (maze[x, y] == "*") return false;
    if (maze[x, y] == "E") return true;

    if (ExploreMaze(maze, x + 1, y)) return true;
    if (ExploreMaze(maze, x - 1, y)) return true;
    if (ExploreMaze(maze, x, y + 1)) return true;
    if (ExploreMaze(maze, x, y - 1)) return true;

    return false;
}
```

Avoiding Infinite Loops

```
private bool IsMazeSolvable(string[,] maze, int startX, int startY)
{
    var searched = new bool[maze.GetLength(0), maze.GetLength(1)];
    return ExploreMaze(maze, searched, startX, startY);
}

private bool ExploreMaze(string[,] maze, bool[,] searched, int x, int y)
{
    if (searched[x,y])
        return false;

    if (x < 0 || y < 0 || x >= maze.GetLength(0) || y >= maze.GetLength(1))
        return false;

    searched[x, y] = true;

    if (maze[x, y] == "*") return false;
    if (maze[x, y] == "E") return true;

    if (ExploreMaze(maze, searched, x + 1, y)) return true;
    if (ExploreMaze(maze, searched, x - 1, y)) return true;
    if (ExploreMaze(maze, searched, x, y + 1)) return true;
    if (ExploreMaze(maze, searched, x, y - 1)) return true;

    return false;
}
```

Avoiding Repetitious Calculation

The Fibonacci Sequence

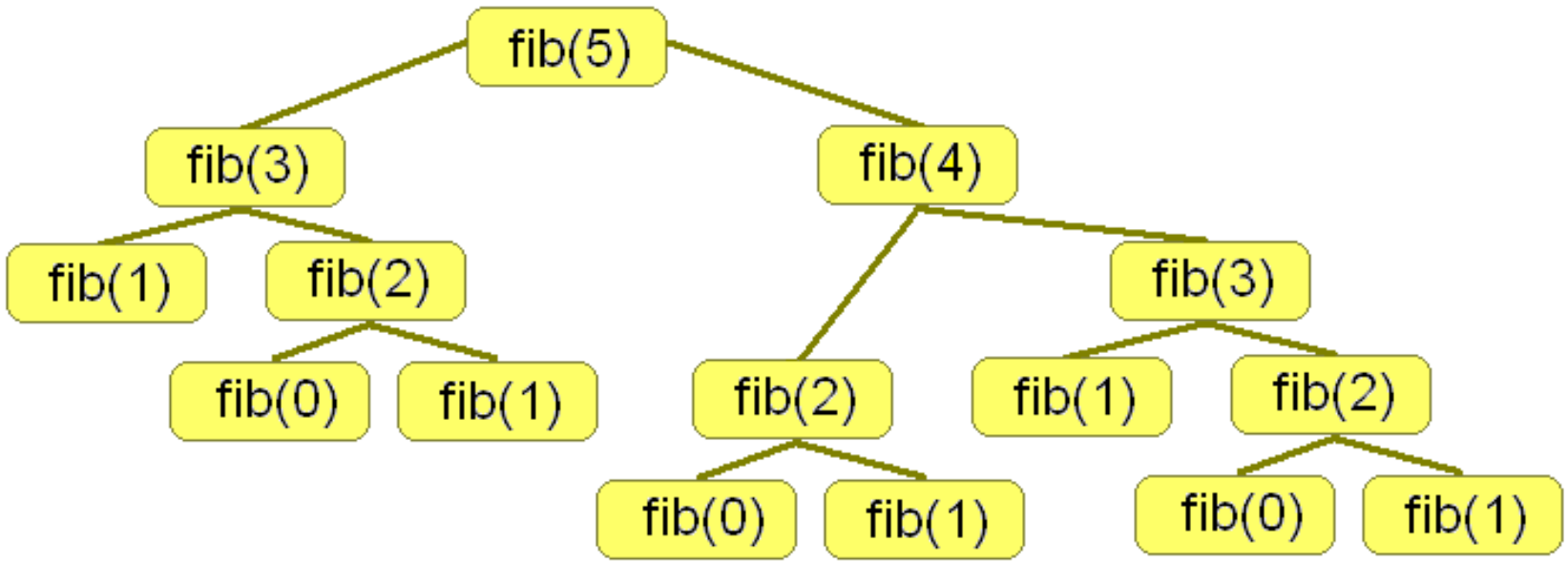
1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21
13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

Naïve Solution

```
int Fibonacci(int index) {
    if (index < 1) return 0;
    if (index == 1) return 1;
    return Fibonacci(index - 2) + Fibonacci(index - 1);
}
```

Issue w/Solution



Solution using Memo

```
int Fibonacci(int index) {
    var memo = new int[index];

    for (var i = 0; i < index; i++) {
        memo[i] = -1;
    }

    memo[0] = 0;
    memo[1] = 1;

    return FibMemo(index, memo);
}

int FibMemo(n, memo) {
    if (memo[n] != -1) return memo[n];

    memo[n] = FibMemo(n - 2, memo) + FibMemo(n - 1, memo);

    return memo[n];
}
```

Steps for Creating a Recursive Algorithm

1. Make simple test cases.
2. Identify the base cases.
3. Identify boundary conditions (how deep to search, etc.)
4. Write algorithm w/base cases first.
5. Identify what needs to be returned.
6. Run on test cases, adjust as necessary.
7. Identify inefficiencies (duplication of effort) and add memo to be passed down.

Optimization

Basics and Practices

Profiling

Profiling

- Three focuses:
  - General Startup Traces
  - General Runtime Traces
  - Spikes

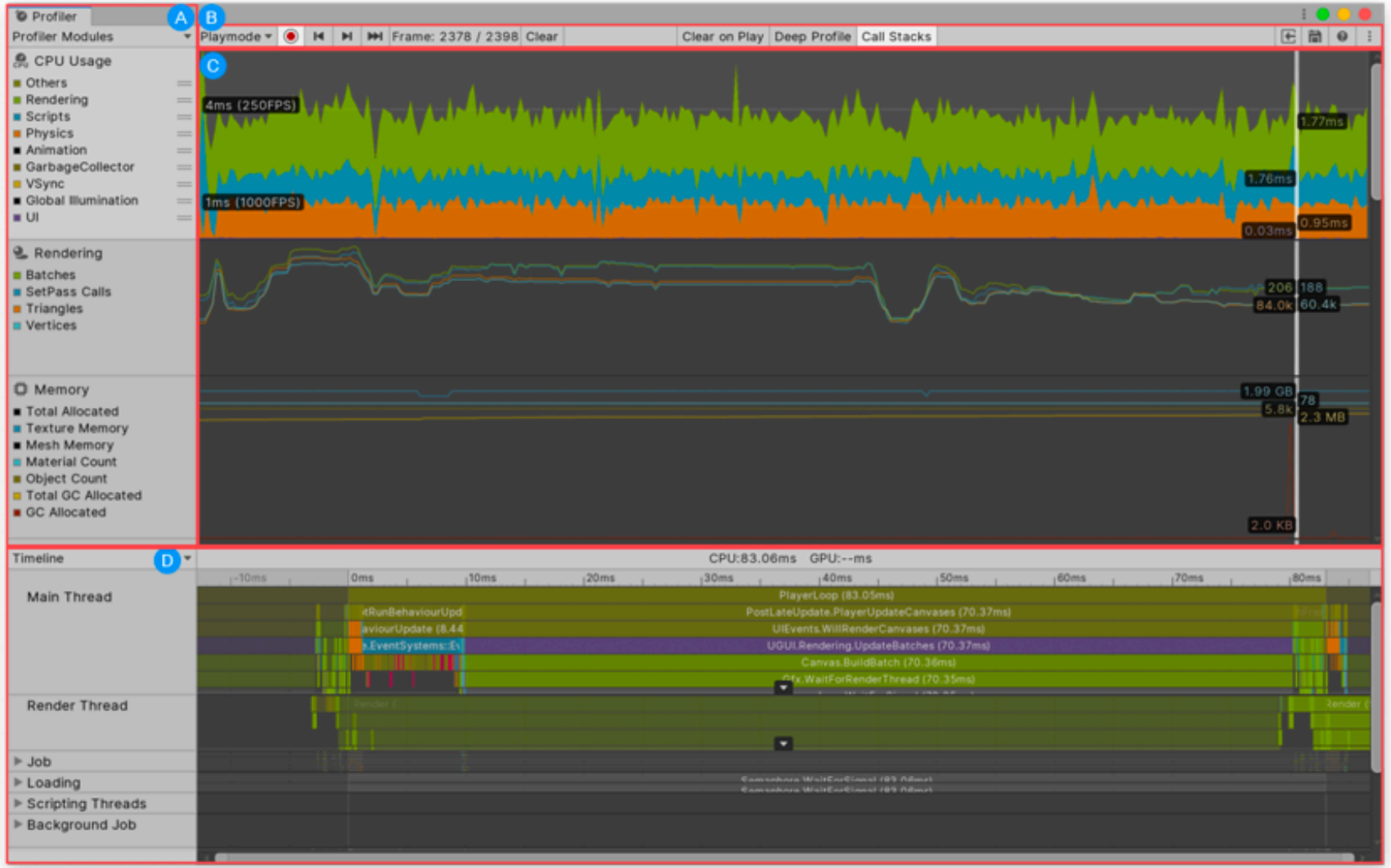
Profiler Window

A: Modules

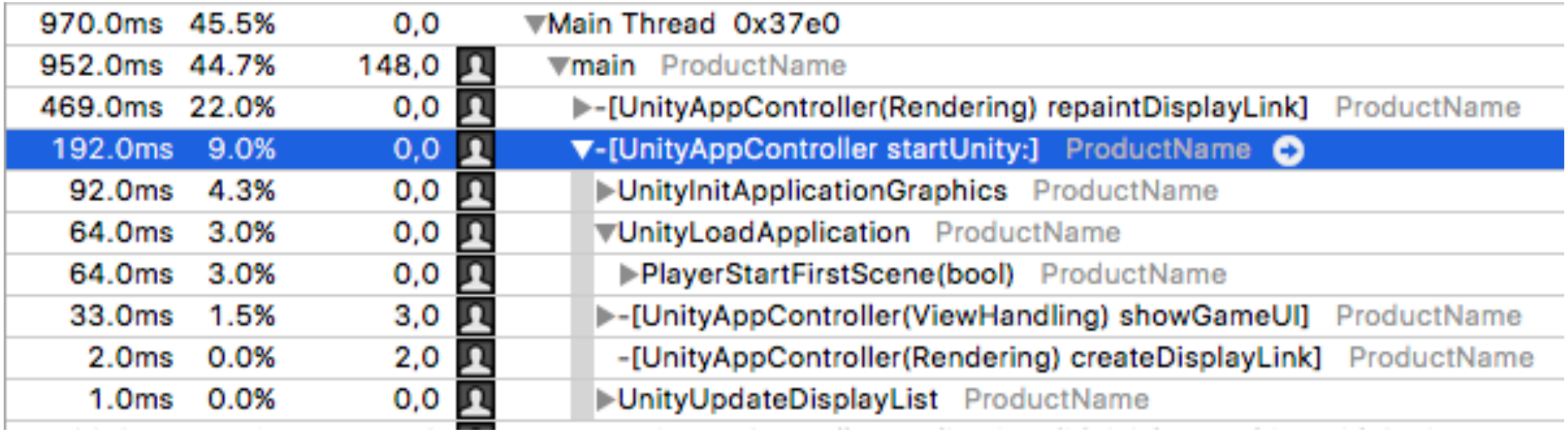
B: Controls

C: Frame Charts

D: Module Details



Profiling: Startup



Profiling: Startup

- startUnity method includes the following:
- UnityInitApplicationGraphics
  - setting up the graphics device
  - initializing many of Unity’s internal systems
  - initializes the Resources system.
- UnityLoadApplication
  - contains methods that load and initialize the first Scene in project
  - Compiling shaders
  - Uploading textures
  - Instantiating Game Objects
  - All Monobehaviors in first scene have Awake() callbacks executed

Profiling: Runtime

970.0ms	45.5%	0,0	▼Main Thread 0x37e0
952.0ms	44.7%	148,0	▼main ProductName
469.0ms	22.0%	0,0	▼-[UnityAppController(Rendering) repaintDisplayLink] ProductName
464.0ms	21.8%	1,0	▼UnityRepaint ProductName
457.0ms	21.4%	0,0	▼UnityPlayerLoopImpl(bool) ProductName
457.0ms	21.4%	1,0	▼PlayerLoop(bool, bool, IHookEvent*) ProductName
212.0ms	9.9%	0,0	▼PostLateUpdate_FinishFrameRendering ProductName
210.0ms	9.8%	0,0	▶PlayerRender(bool) ProductName
1.0ms	0.0%	1,0	IsBatchmode() ProductName
1.0ms	0.0%	0,0	▶PresentAfterDraw(GfxDevice::PresentMode) ProductName
102.0ms	4.7%	2,0	▶PhysicsManager::FixedUpdate() ProductName
30.0ms	1.4%	1,0	▶DirectorManager::ExecuteStage(DirectorStage) ProductName
18.0ms	0.8%	0,0	▶NavMeshManager::Update() ProductName
16.0ms	0.7%	0,0	▶PostLateUpdate_UpdateAudio ProductName
11.0ms	0.5%	0,0	▶PostLateUpdate_UpdateAllSkinnedMeshes ProductName
10.0ms	0.4%	0,0	▶Script_RunDelayedFixedFrameRate ProductName
8.0ms	0.3%	0,0	▼Script_RunBehaviourUpdate ProductName
8.0ms	0.3%	0,0	▶void BaseBehaviourManager::CommonUpdate<BehaviourManager>() ProductName
6.0ms	0.2%	0,0	▼Script_RunBehaviourFixedUpdate ProductName
6.0ms	0.2%	0,0	▶void BaseBehaviourManager::CommonUpdate<FixedBehaviourManager>() ProductName
6.0ms	0.2%	0,0	▶FixedUpdate_AudioFixedUpdate ProductName
5.0ms	0.2%	0,0	▶PostLateUpdate_PlayerUpdateCanvases ProductName

Profiling: Runtime

- PlayerLoop
  - Unity's main loop
  - Runs once per frame
- PlayerRender is the method that runs Unity's rendering system
  - Culling objects
  - Calculating dynamic batches
  - Submitting drawing instructions to the GPU
  - Image Effects
  - Rendering-based script callbacks (OnWillRenderObject)
  - Ideally, *this is your most expensive CPU method while game is interactive.*
    - Anything slower than PlayerRender is *often* an issue

Profiling: Runtime (cont.)

- BaseBehaviourManager
  - calls three templated versions of CommonUpdate.
  - These invoke certain callbacks within MonoBehaviours attached to active GameObjects in the current Scene.
    - `CommonUpdate<UpdateManager>` calls Update callbacks
    - `CommonUpdate<LateUpdateManager>` calls LateUpdate callbacks
    - `CommonUpdate<FixedUpdateManager>` calls FixedUpdate if the physics system has ticked
  - Generally, *this is the entry point for most script code in a Unity project.*

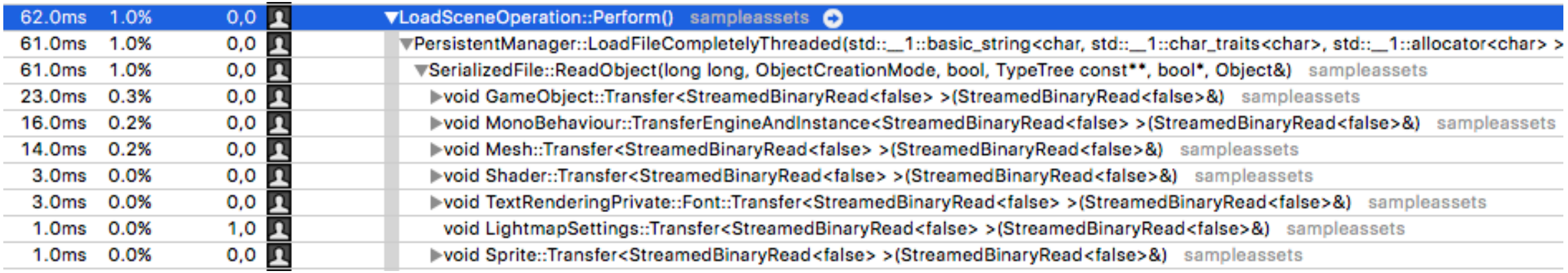
Profiling: Additional Runtime

- UI::CanvasManager invokes several different callbacks if a project uses Unity UI
  - UI__'s batch computation
  - layout updates
- DelayedCallManager::Update runs coroutines
- PhysicsManager::FixedUpdate runs the PhysX physics system
  - Physics2DManager::FixedUpdate if 2D physics
  - Influenced by the number of physics objects in the current scene (Rigidbodies and colliders)
  - OnTriggerStay and OnCollisionStay

Script Methods in Trace

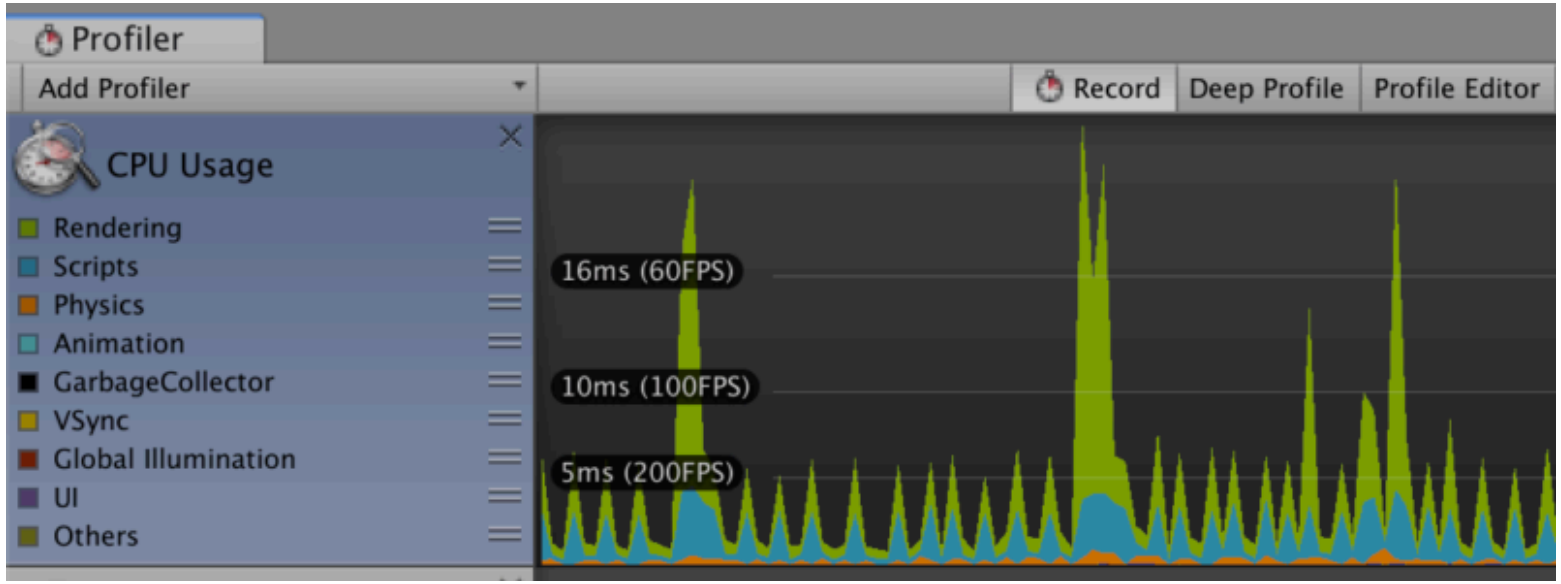
67.0ms	2.1%	0,0		▼Script_RunBehaviourUpdate ProductName
67.0ms	2.1%	0,0		▼void BaseBehaviourManager::CommonUpdate<BehaviourManager>() ProductName
66.0ms	2.1%	1,0		▼UpdateBehaviour ProductName
64.0ms	2.0%	0,0		▼MonoBehaviour::CallUpdateMethod(int) ProductName
63.0ms	2.0%	0,0		▼CallMethodIfAvailable ProductName
63.0ms	2.0%	0,0		▼ScriptingInvocationNoArgs::Invoke() ProductName
63.0ms	2.0%	0,0		▼ScriptingInvocationNoArgs::Invoke(ScriptingException**) ProductName
61.0ms	1.9%	0,0		▼0x100d3badc ProductName
61.0ms	1.9%	0,0		▼RuntimeInvoker_Void_t2863195528(MethodInfo const*, void*, void**) ProductName
24.0ms	0.7%	0,0		▶StandaloneInputModule_Process_m3720469665 ProductName
17.0ms	0.5%	0,0		▶EventSystem_Update_m242895889 ProductName
13.0ms	0.4%	0,0		▶PlayerShooting_Update_m2128394500 ProductName
3.0ms	0.0%	0,0		▶ScoreManager_Update_m318092571 ProductName
3.0ms	0.0%	0,0		▶PlayerShooting_DisableEffects_m2460645109 ProductName
1.0ms	0.0%	0,0		▶Transform_Translate_m1056984957 ProductName

Asset Loading



Profiling: Spikes

- Spike is a sudden drop in the frame rate of a game
- This is noticed when a game suddenly stops and doesn’t move for a noticeable time
- Spikes can be seen as a high points on Profiler Graph



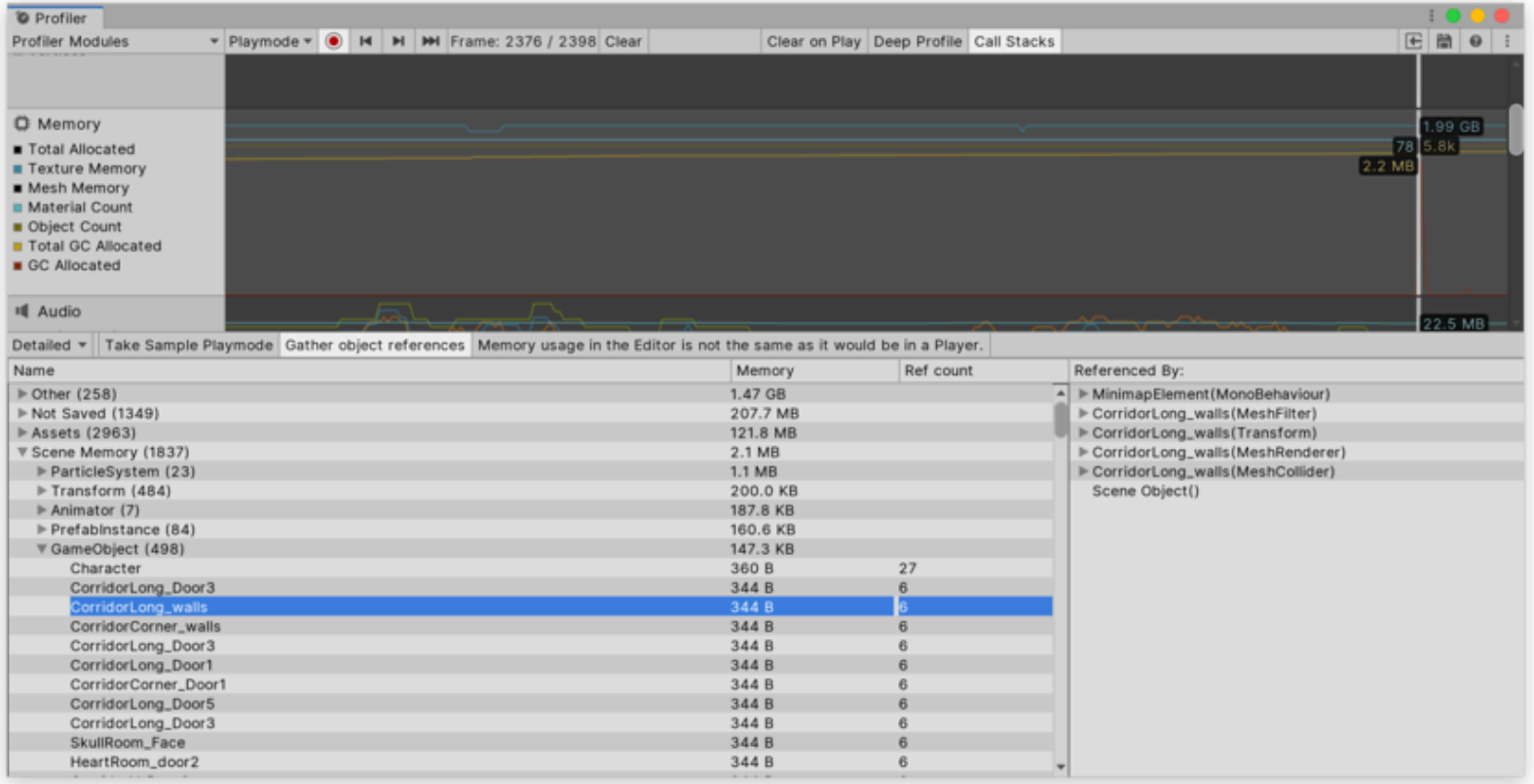
Memory

Memory

- Split levels into additively loaded scenes
- **Pool frequently instantiated objects**
  - Instantiating objects is slow.
  - Create pools of objects at the start of the game/content.
  - Reuse objects instead of creating new ones.
- **Profile memory consumption**
  - Unity has a Memory visualization tool (link)

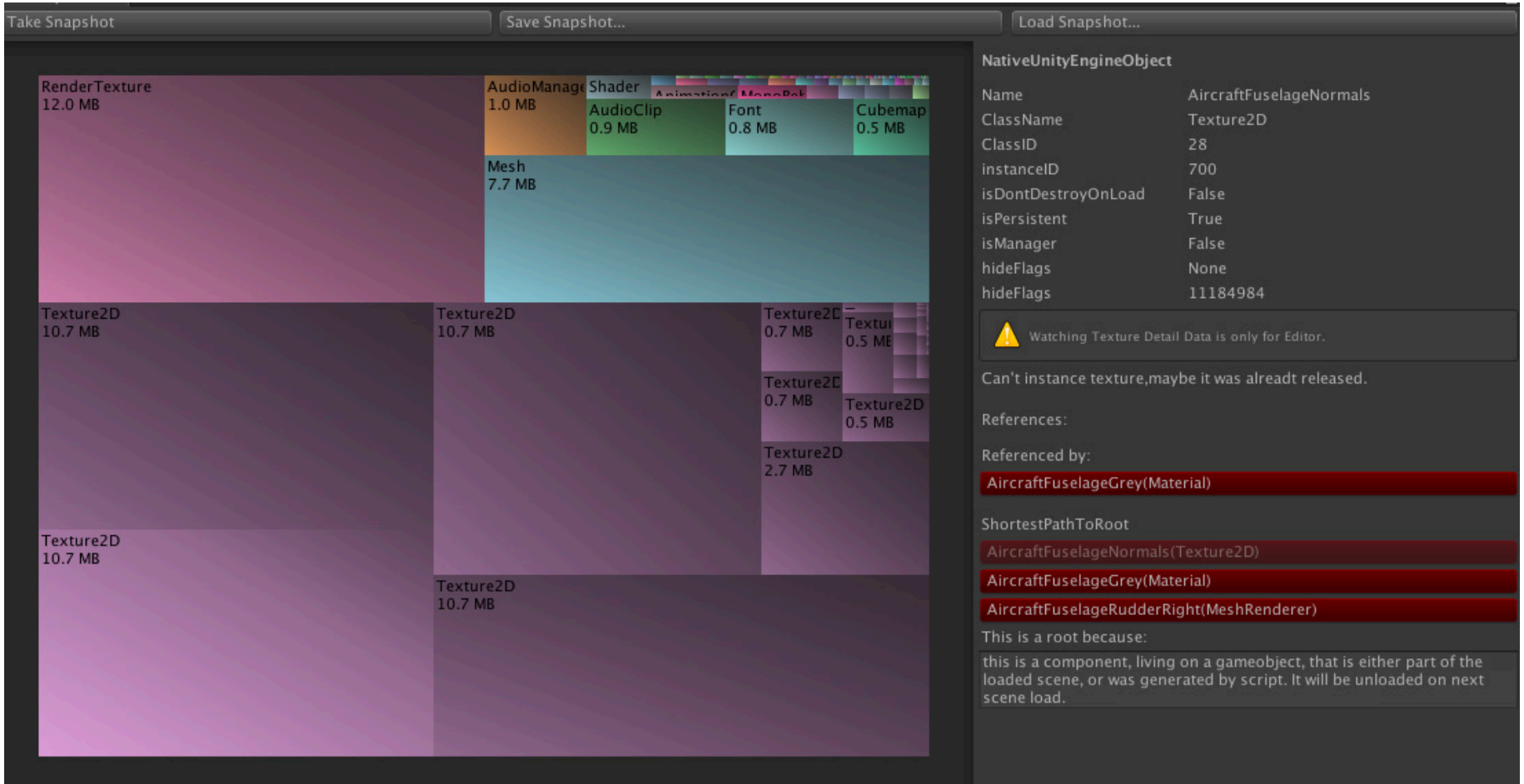
Memory Profiler

- The Profiler Window includes a Memory Profiler
- Includes detailed view

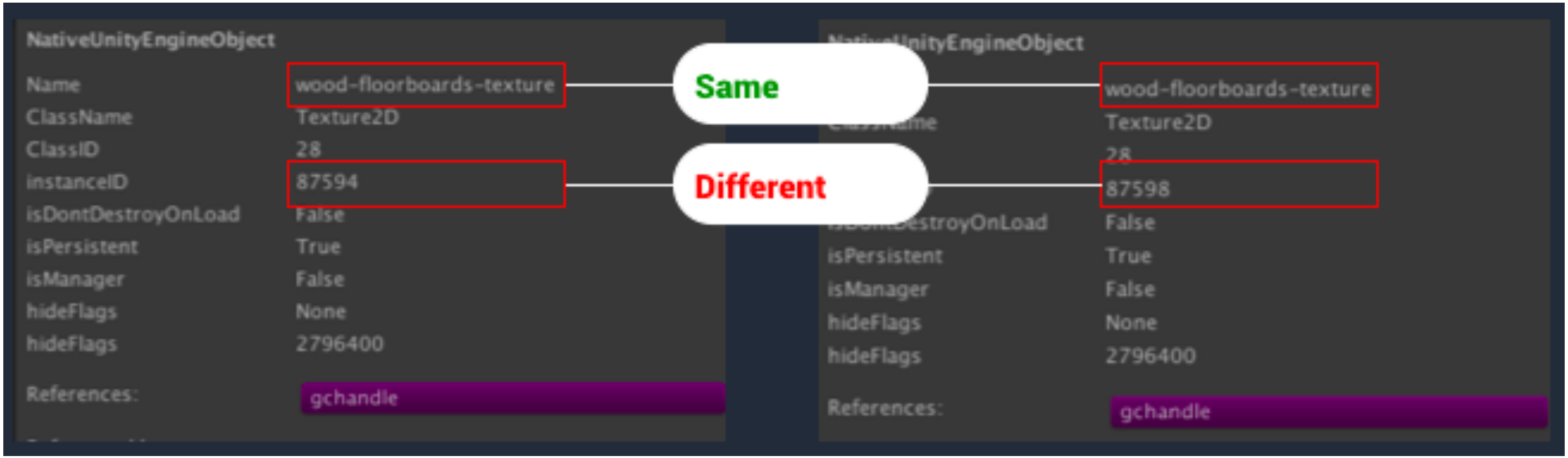


Profile Memory Consumption

- Unity has a Memory visualization tool (link)



Identify Duplicated Textures



Garbage Collection

- When enough memory has been allocated, the garbage collector automatically clears unused objects from the memory for single frame.
- This is referred to as running the garbage collector, and always causes a frame rate drop.

- Solutions:
  - Unity 2019.1 with .NET 4x Equivalent introduced Incremental Garbage Collection
    - Splits GC operation over multiple frames
    - Player Settings > Other > Use Incremental GC
  - Manually garbage collect on loading screens / moments when frame drops won't be noticed
    - System.GC.Collect();

Techniques:

- Caching
  - Instead of looking for an object multiple times, cache it on initialization.
- Object Pooling
  - Instead of instantiating and destroying objects, turn them on and off and track them.

Assets

Textures

- Disable read-write flag
- Disable mipmaps if possible
  - If a texture never changes scale/size
  - is on an object that always in the same relative z-depth to a camera
- Enforce sensible Texture size limits

Models

- Disable the Read/Write enabled flag
  - If not modifying a Mesh at runtime via script
  - If the mesh is not used for a MeshCollider
- Disable rigs on non-character models
  - Animator component is automatically added to them.
- Enable “Optimize Game Objects” option on animated models
- Check MeshRenderer settings
  - By default, Unity enables:
    - Shadow casting
    - Shadow receiving
    - Light Probe sampling
    - Reflection Probe sampling
    - Motion Vector calculation.

Audio

- Force audio clips to mono if targeting mobile
- Reduce audio bitrate where possible
- Import uncompressed audio into Unity
  - Unity will compress audio for you.
- Use identical bit rates and sample rates on your audio.

Resources

- Every Asset file within every folder named “Resources” is included in the build.
  - This includes everything in any subfolder(s)
  - the time required to initialize the Resources system increases at least linearly in correlation with the number of files within “Resources” folders.
- When game starts, a balanced search tree is created to find things in resources
  - Construction time grows at O(n log(n)) on number of objects
- Solution: Use Resources folders while prototyping, then switch to async scene loading or Scriptable objects or AssetBundles.

Resources Folder Replacements

- AssetBundles
- Scriptable Objects

XML, JSON, Other long-form text parsing

- Parsing text is one of the heaviest operations at load time.
- Parsing text can outweigh time spent loading and instantiating Assets.
- C#’s XML parser is flexible, but not optimizable.
- Often third-party parsers are built on reflection
  - Reflection is very slow.

- Solutions:
  - Parse at build time
  - Split and lazy load

Data Structures

Use Correct Data Structures

- Indexing into an **array or list** is constant time - O(1)
  - Randomly indexing into a collection
  - Iterating through all elements
  - INSERTION IS SLOW.
  - SEARCHING IS SLOW
- Adding or removing items from a dictionary or hashset is constant time - O(1)
  - Inserting
  - Removing
  - Searching
  - ITERATING IS SLOW
- Data is related – Dictionary
  - If it's related in a *one-way manner.*

Overuse of Dictionaries

- If you want to iterate over pairs of data every frame, do not use a dictionary.
  - Have to iterate over every *possible* value that *could* be in the dictionary.
  - Even if that key does not have a value associated with it.
- Instead, create a struct and store it in a list.

The Real World is Complicated

- Remember, Lists, Dictionaries, HashSets, of *reference types* only hold references.
- Use multiple data structures.
  - List for iteration and Hashset to determine whether item is in list
  - Two dictionaries – one for keys to values, another for values to keys.

Graphics

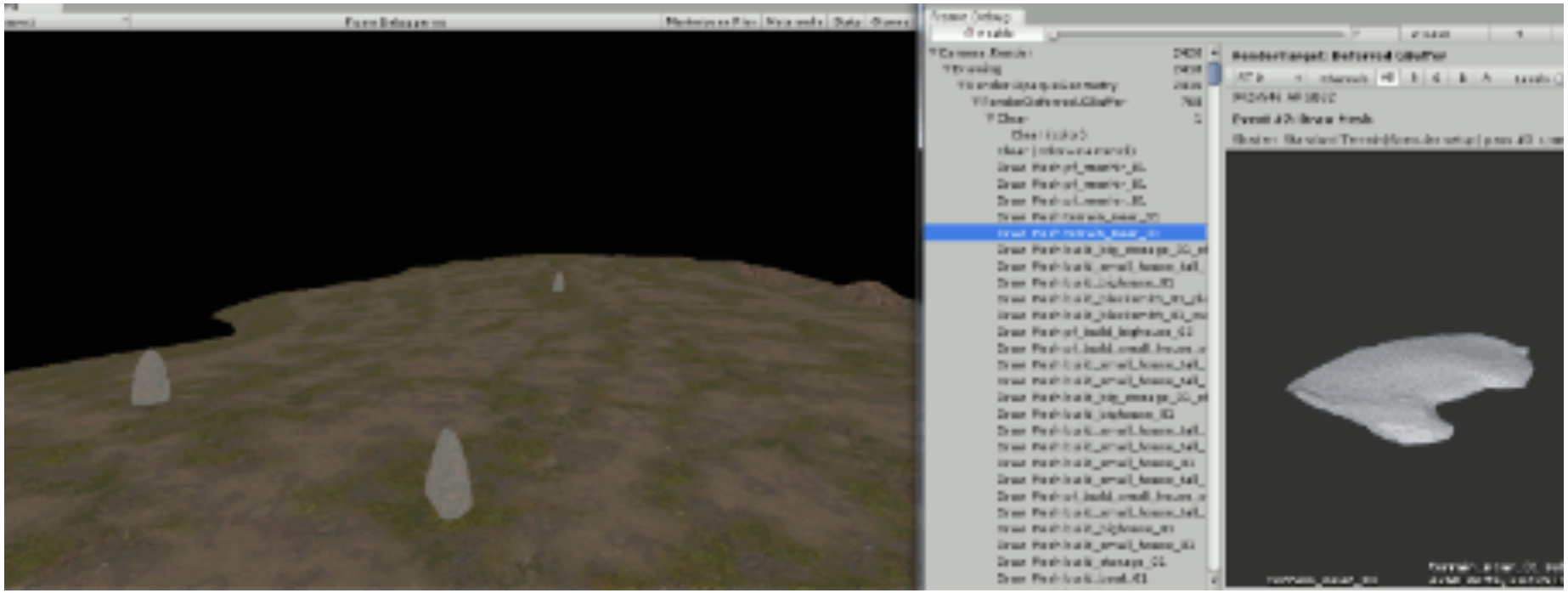
Graphics are Expensive

This is the domain of Tech Art, so we're not going to go into too much detail except to say:

- Transparency is expensive
- Lighting is expensive – and the more lighting, the more expensive
- In shaders, use correct precision (use a bit, byte, or short instead of a long where possible)
- Set up LOD's correctly
- Use Frame Debugger

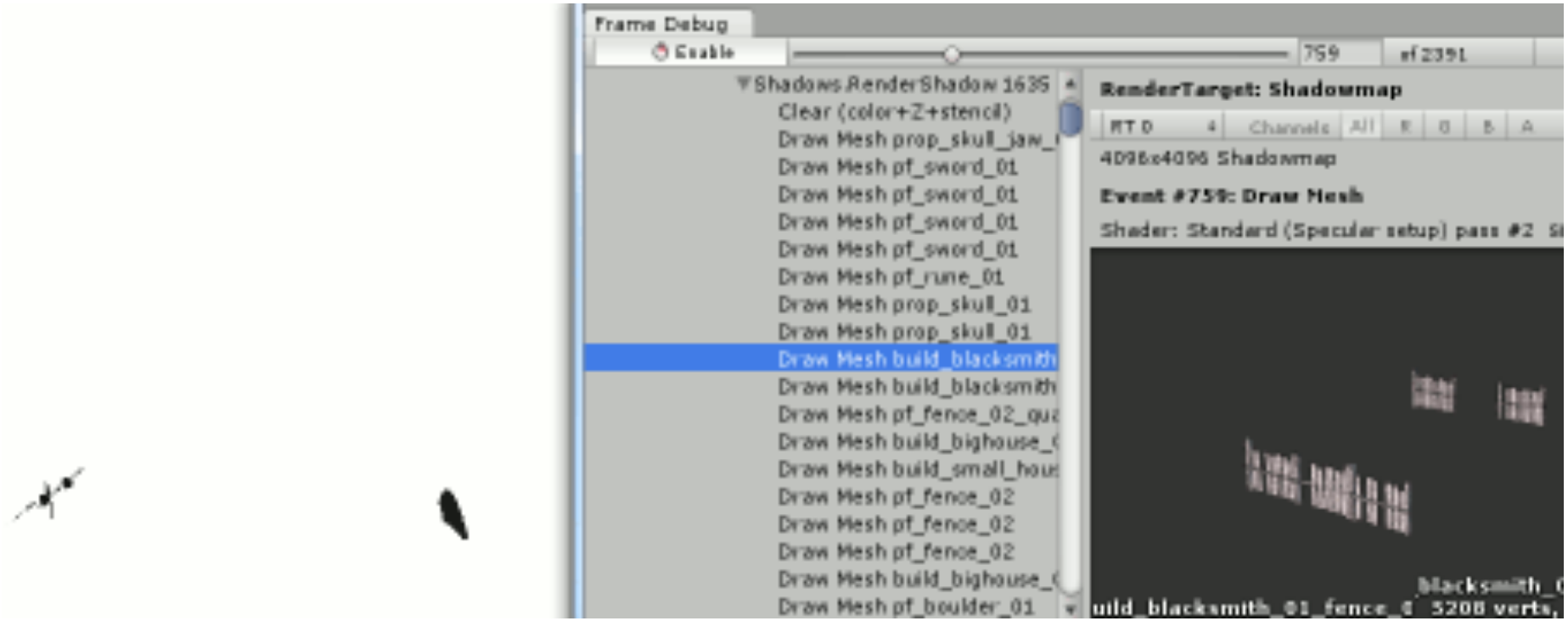
Using the Frame Debugger (link)

- Window > Analysis > Frame Debugger
- Can help identify if a lot of draw calls to objects not in scene /view are occuring



Using the Frame Debugger (cont.)

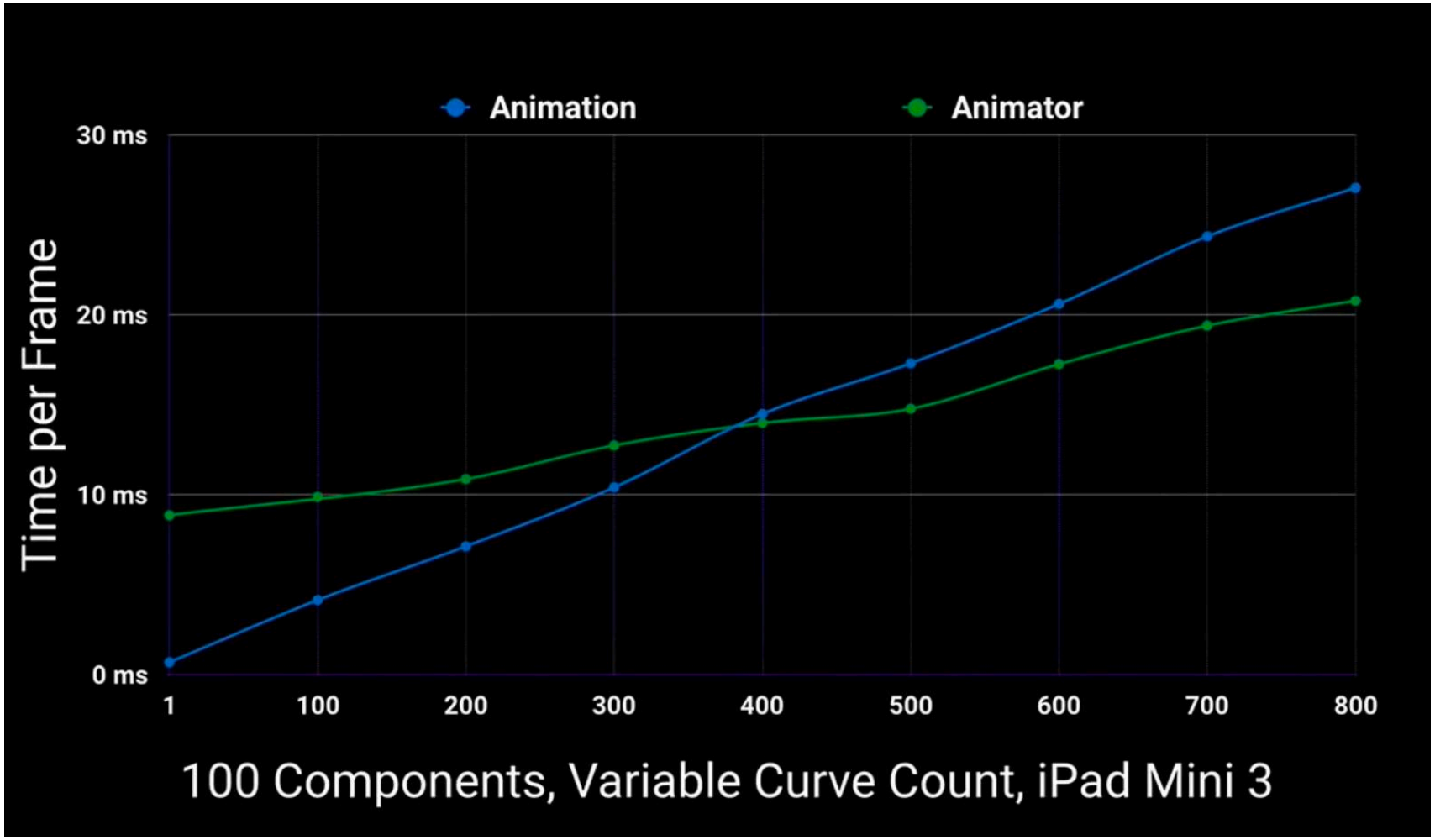
- Can also help diagnose when too many shadows are rendering



Animator System vs. Animation System

- "Animator System”: Animator component, attached to GameObjects, and the AnimatorController asset, which is referenced by one or more Animators.
  - Previously Mecanim, and is very feature-rich.
  - Animator Controller, defines states (Animation Clip or Blend Tree)
  - States organized into Layers*
  - heavily multithreaded*
- “Animation System”: The Animation component
  - it is very simple.
  - Each frame, each active Animation component linearly iterates through all the curves in its attached Animation Clip, evaluates those curves, and applies the results.
  - Almost zero overhead

Animator System vs. Animation System



Canvases

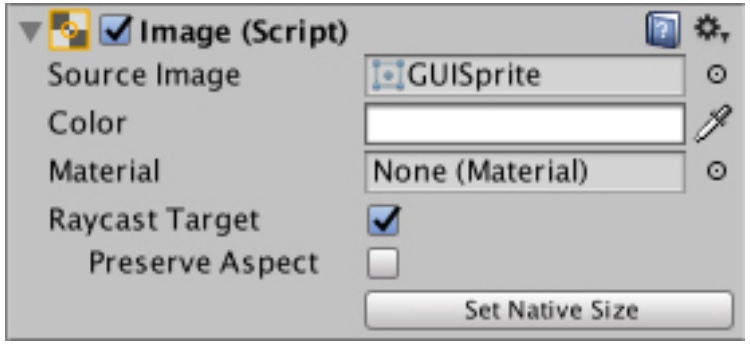
Dirty Canvases

- Canvas generates meshes that represent UI elements placed on it.
- Also *regenerates* meshes when elements change
- Issues draw calls to the GPU so they are displayed.
- Generating meshes is expensive
  - Whenever one element changes, everything on the canvas has to be re-analyzed
  - If you have one canvas w/ dozens or hundreds of elements, this can cause long spikes whenever any element changes

- Solution:
  - Divide Up Your Canvases
    - Nested canvases isolate content, maintain their own geometry, and perform their own batching
  - Group canvases by when they get updated
  - Separate static from dynamic
  - Don’t use canvases

Graphic Raycasting

- Graphic raycaster is the component that translates input into UI Events for UI elements
- You need a graphic raycaster on every Canvas that requires input
  - Performs intersection checks on every UI element

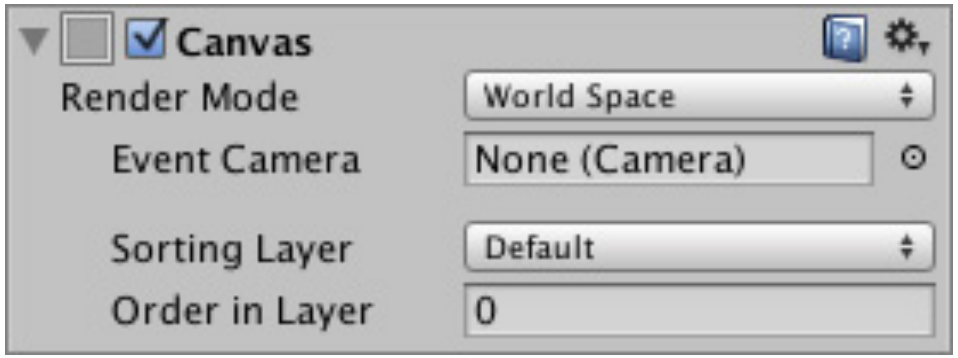


- Solution:
  - Turn off the Raycast Target for static or non-interactive elements.
  - Remove Graphic Raycasters from non-interactive UI Canvases

World Space Canvases

- **Problem: World Space canvases need to know which camera their interaction events should come from.**
- Depending on the code path Unity takes, it will access Camera.main between 7-10 times per frame, per Graphic Raycaster, per World Space Canvas.

- Solution:
  - DO NOT LEAVE THIS SETTING EMPTY.



Other Canvas Techniques

- Avoid Layout Groups where possible
  - Every UI element that tries to dirty its Layout will perform at least one GetComponents call.
  - Solution: Use anchors or write your own code.
- Pool UI Objects carefully
  - Disable then reparent, instead of reparenting then disabling
    - Dirties the old hierarchy only once.
- Disable the Canvas Component, not the Canvas GameObject
  - Doesn’t trigger expensive OnDisable/OnEnable callbacks on the Canvas’ hierarchy
  - Canvas keeps it’s vertex buffer, retains all meshes and vertices
  - Re-enabling doesn’t trigger a rebuild.
- Animators *always* dirty every frame.
  - Only put animators on dynamic elements that always change
  - For elements that rarely change, create your own code.

General Optimization Techniques

Address by ID, not String Literals

- Do not use string literals for Animator, Material, or Shader properties
  - Unity doesn’t use strings internally
  - All property names are hashed into ID’s
  - Whenever using Set or Get on an Animator, Material, or Shader, use the integer valued method instead of string valued method.

```
Material mat;

void Update() {
  mat.SetFloat("_MyFloat", 1.0f);
}
```

```
int _MyFloatID;
Material mat;

private void Awake() {
  _MyFloatID = Shader.PropertyToID("_MyFloat");
}

void Update() {
  mat.SetFloat(_MyFloatID, 1.0f);
}
```

Use non-allocating APIs (Physics)

- ~~RaycastAll~~ | RaycastNonAlloc
- ~~SphereCastAll~~ | SphereCastNonAlloc
- ~~OverlapSphere~~ | OverlapSphereNonAlloc
- Each one finds all colliders that meet a specification and puts them in an array
  - NonAlloc methods you pass in an array, it fills it and returns how many colliders it found.
  - Regular methods create an array each time, and return the array

Avoid allocating API’s (everywhere)

- If a function returns a collection (arrays in particular), it’s likely expensive.
- Mesh.vertices
- GetComponents

Check Null Comparisons

- ***No Null comparisons against UnityEngine.Object subclasses***
  - ***myGameObject == null ☹***
  - ***myComponent == null ☹***
  - *myClassThatDoesNOTInheritFromMonobehavior == null ☺*
- Invoking methods on instances of UnityEngine.Object subclasses calls engine code.
  - Must perform lookups and validations to convert script references into native references.
- Replace w/ ReferenceEquals(someObject, null)

Math

- For any math that is done in tight loops:
  - Integer math is faster than floating-point math
  - floating-point math is faster than vector, matrix or quaternion math

- Which is faster?

```
Vector3 x; int a, b;
Vector3 answer1 = a * x * b;
Vector3 answer2 = a * b * x;
```

```
position += new Vector3(5f, 5f, 5f);
position.x += 5f;
position.y += 5f;
position.z += 5f;
```

Find and FindObjectOfType

- Object.Find and Object.FindObjectOfType
- Anything that searches through a scene will be, at best, O(nlog(n)), and likely will be 0(n)
- Camera.main calls Object.FindObjectWithTag

- Solutions:
  - Static references to important variables or data structures w/important variables
  - Manager class that tracks variables of type
  - Access in Start() or OnEnable(), cache the reference

Debug Code

- Anything that calls UnityEngine.Debug is slow
  - This is Debug.Log()!!
- It’s also not removed from builds.

- Solution:
  - You can wrap each with preprocessor blocks (#if … #endif)
  - You can make a helper class:

```
public static class Logger {
   [Conditional("ENABLE_LOGS")]
   public static void Debug(string logMsg) {
        UnityEngine.Debug.Log(logMsg);
   }
}
```

Lots of Monobehaviors

- Unity tracks lists of objects interested in its callbacks, such as Update, FixedUpdate and LateUpdate.
  - MonoBehaviours are added to/removed from these lists when they are Enabled or Disabled
- It’s convenient to put things into little Monobehaviors w/ their own Update functions, but it’s very inefficient
- Instantiating Prefabs with many monobehaviors attached is also slow

- Takeaways:
  - Have as few monobehaviors as possible.
  - Have as few w/built in Unity callbacks as possible.
    - If they need none, they do not need to be monobehaviors.

Lots Of Monobehaviors (example)

```
class PodMovement :Monobehavior {
    void Update() {
        if (Vector3.Distance(transform.position, Services.Player.transform.position) < 1)
            Run();
    }
}

class PodFlash : Monobehavior {
    bool flashing;

    void Update() {
        if (flashing)
            Flash();
    }
}

class PodTwist() {
    void Update() {
        transform.rotation = Time.deltaTime * Quaternion.Euler(10, 10, 10) * transform.rotation;
    }
}
```

First Fix

- More expressive as one method
- Also easy memory and CPU timesave
- Fewer monobehaviors to track and fewer method calls.

```
class Pod : Monobehavior {
    void Update() {
        Rotate();

        if (Vector3.Distance(transform.position, Services.Player.transform.position) < 1)
            Run();
        if (flashing)
            Flash();

    }
}
```

Second Fix

- Have manager classes determine whether to call an Update, rather than calling update every frame.

```
private class PodManager : Monobehavior {
    List<Pod> pods;
    private class Pod {
        GameObject podObject;
        bool flashing;

        public void Rotate() { /* ... */ }
        public void Run() { /* ... */  }
        public void Flash() { /* ... */ }
    }

    void Update() {
        foreach (var pod in pods) {
            pod.Rotate;
            if (pod.flashing) Flash();
            ... etc.
        }
    }
}
```

Cost of Monobehaviors

Operation	Time (ms)	Time (%)
Iterate over all Behaviours	1517 ms	15.2%
Check if the call is valid	2188 ms	21.9%
Prepare to invoke the method	2061 ms	20.6%
Check if arguments are valid	900 ms	9%
(IL2CPP) Check if method exists raising an exception otherwise	1018 ms	10.2%
(IL2CPP) Invoke the method	1803 ms	18%
Do important things in Update method	42 ms	0.4%
Overhead	450 ms	4.5%
Total	**9979 ms**	**100%**

https://blogs.unity3d.com/2015/12/23/1k-update-calls/

Reducing Method Call Overhead

- Or – why do I make you write things that are already written?
- Did you know there's a built-in string function called String.StartsWith?
  - This is convenient:

```
if (exampleString.StartsWith("cat"))
    count++;
```

  - This is 100x faster

```
if (exampleString.Length > 3 &&
    ("" + exampleString[0] + exampleString[1] + exampleString[2]) == "cat")
    count++;
```

Another Method Call Example

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

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

Cost of Expressiveness

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

for(int i = 0;
        i < myList.Count;     // call to List::getCount
        i++) {
    Accum                     // call to set_aggregate
    +=                        // call to get_aggregate
    myList[i];                // call to List::get_Value
}
```

Comparison

- 75% speed increase

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

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

```
int accum = 0;
int len = myList.Count; // 1 call to List::getCount

for(int i = 0; i < len; i++) {
    accum += myList[i]; // call to List::get_Value
}
```

This is due to “Method Inlining”

- When method calls in an application end up producing an additional method call instead of the code being repackaged into your binary.
- Unity performs very little method inlining, if any.
  - many methods do not currently inline properly.
  - This is especially true of properties.
  - *Virtual* and interface methods cannot be inlined at all.
- As a result, method calls to C# source is very likely to end up producing method calls in binary application

Coroutines

- Coroutines include the function they're calling, plus a class to track the state of the coroutine across all the invocations of the coroutine (can be multiple times a frame as it's assessed).
- The memory pressure of starting a coroutine is:
  - the overhead of tracking the coroutine
  - the ongoing state of the coroutine
  - the local-scope variables of the coroutine
- Use fewest coroutines possible:
  - Nested coroutines are excellent for code clarity and maintenance, but bad for efficiency
- If a coroutine runs nearly every frame and does not yield on long-running operations, replace with an Update or LateUpdate callback
  - particularly long-running or infinitely-looping coroutines

Final Notes

- Optimize throughout development:
  - Sometimes the correct answer to solve a performance problem is to change your games design.
  - The longer you wait, the harder it is to make that change.
- If performance issue can’t be measured, it’s not an issue.
- Use reasonable resolution assets
- Only call in update what needs to be updated every frame.

Resources

Articles/Documents

- Unity Scripting API – Optimization (link)
- Makaka Unity Optimization (link)

Videos

- Unite 2015 – Uncover Your Game's Power and Performance Profile (link)
- Unite 2012 – Performance Optimization Tips and Tricks for Unity (link)