CPU & Memory Optimization

Session 1

Agenda

Unity

Agenda

- Unity Profiling Tools
  - Profiler
    - CPU Usage
    - Memory
  - Memory Profiler
  - Code Coverage
  - Others

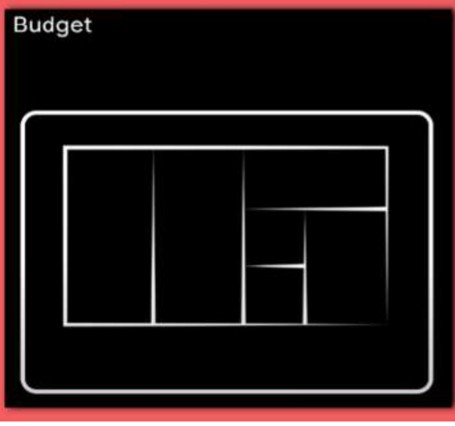
- Optimization Techniques
  - Flyweight Pattern
  - Scriptable Objects
  - Object Pooling
  - Others

Agenda

- Script Optimizations
  - Choosing the right data structure
  - Remove Empty Unity Events
  - Hash the value instead
  - GameObject.AddComponent(...)
  - Cache the References
  - Coroutines
  - Extern Call Caching
  - Find Objects(Type, Tag)
  - Linq Usages
  - String Builder
  - Others
- Unity Optimization Tips
  - Reduce Hierarchy Complexity
  - The Resources Folder
  - Others

Introduction to Optimization

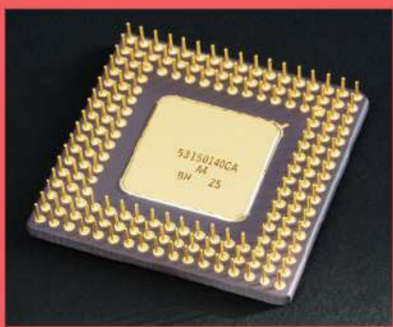
- We can summarize optimization in two words; **budget** and **profiling**.
- Budget, meaning the resources and constraints, such as **memory, resolution, time, even money.**
- Profiling, meaning the process of **measuring the usage of that budget**.

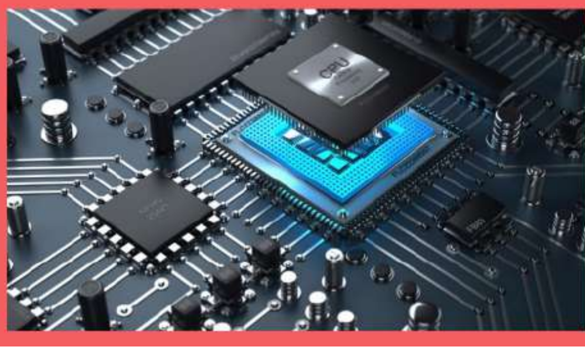


What's CPU?

(Central Processing Unit)

- The CPU performs **basic arithmetic, logic, controlling, and input/output (I/O) operations** specified by the instructions in the program.
- It can execute **million of instructions per second**, but can carry only **one instruction at a time.**

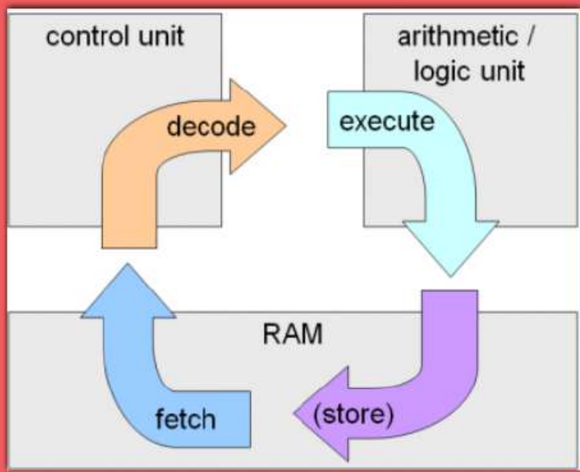




CPU Instruction Cycle

(Fetch-Execute cycle)

1. **Fetching** instructions from memory, in order to know how to handle the input and know the corresponding instructions for that particular input data it received. Specifically, it looks for the address of the corresponding instruction and forwards the request to the RAM. The CPU and RAM constantly work together. This is also called *reading from* memory.
2. **Decoding** or translating the instructions into a form the CPU can understand, which is machine language (binary).
3. **Executing** and carrying out the given instructions.
4. **Storing** the result of the execution back to memory for later retrieval if and when requested. This is also called *writing to* memory.



Examples of Common CPU Instructions

- **MOV (Move):** This instruction copies data from one location to another. It's used for assignments and data transfers.

```
MOV eax, ebx   ; Copy the value in EBX register to EAX register
```

- **ADD (Addition):** This instruction adds two values and stores the result in a destination.

```
ADD eax, 5     ; Add 5 to the value in EAX register
```

- **SUB (Subtraction):** This instruction subtracts one value from another and stores the result in a destination.

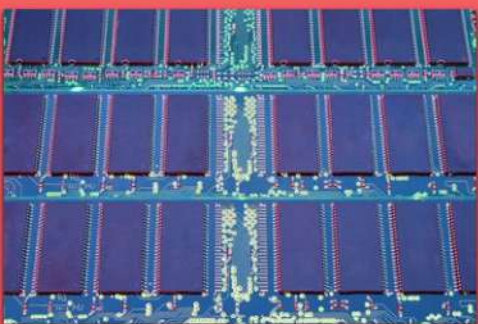
```
SUB ecx, edx   ; Subtract the value in EDX register from ECX register
```

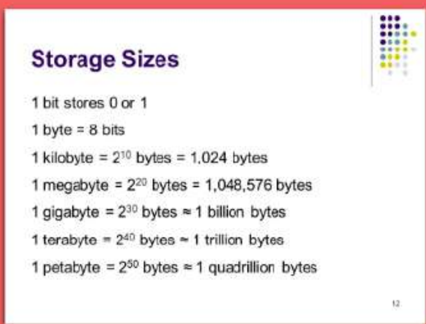
- **RET (Return):** This instruction is used at the end of a subroutine to return control to the calling code.

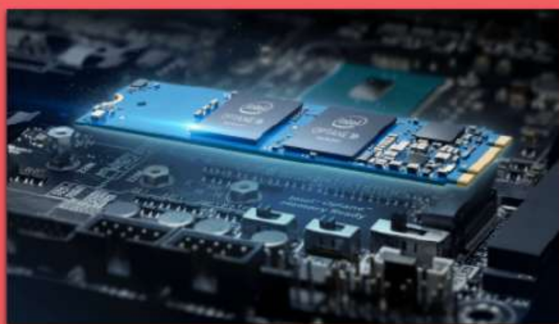
```
RET             ; Return from the current subroutine
```

Memory

- Memory is the **storage space in the computer**, where **data** is to be processed and **instructions** required for processing are stored.
- The memory is divided into large number of small parts called **cells**.
- Each location or cell has a **unique address**, which varies from zero to memory size minus one.

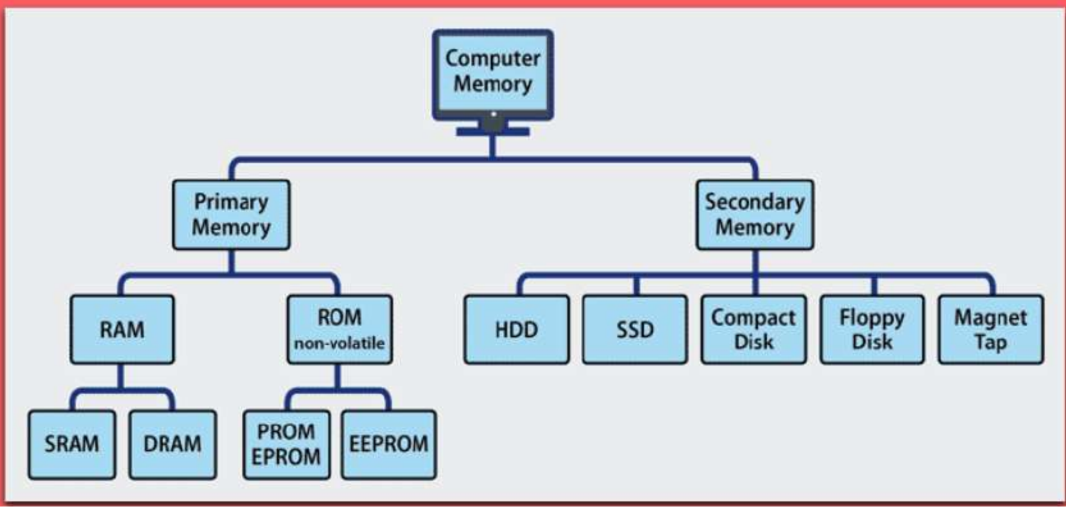




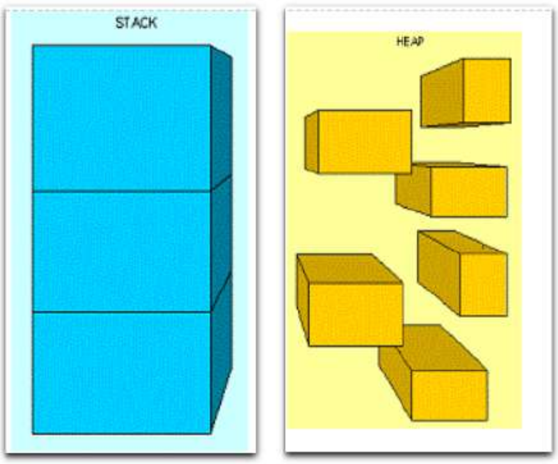


Memory Types

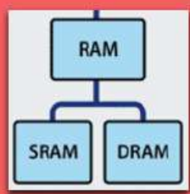
- **RAM:** Random Access Memory
- **ROM:** Read Only Memory
- **SRAM:** Static RAM - Stack
- **DRAM:** Dynamic RAM - Heap



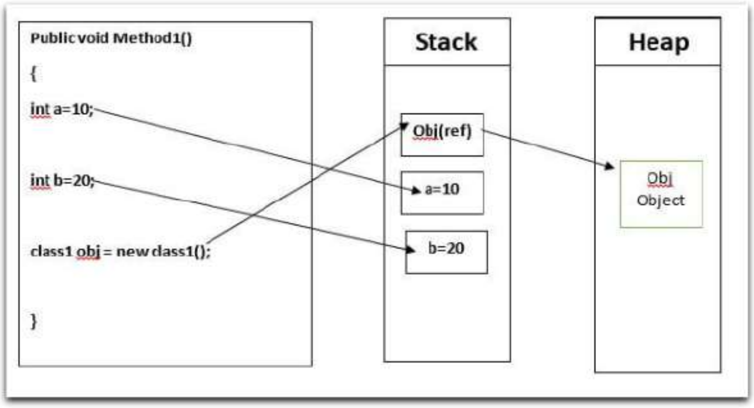
Runtime Memory

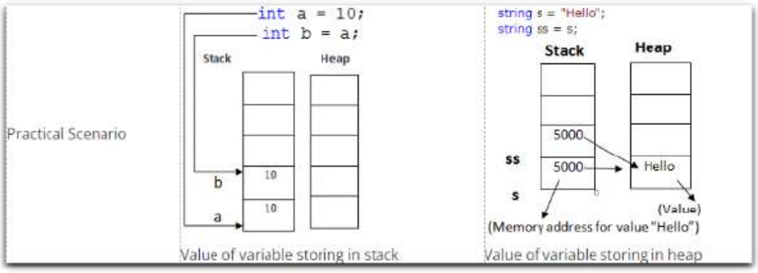


- **RAM** is divided into two different memory spaces; **DRAM** and **SRAM**.
- **DRAM** refers **Heap Memory** space.
- **SRAM** refers **Stack Memory** space.



- **Stack Memory**
  - It is an array of memory.
  - LIFO - Last In First Out - Data Structure
  - PUSH, POP and, PEEK
  - Value Types
- **Heap Memory**
  - It is an area of memory where chunks are allocated to store certain kinds of data objects.
  - In it data can be stored and removed in any order.
  - Reference Types





Value Types ~ *Stack Memory*

- bool
- byte
- char
- decimal
- double
- enum
- uint
- ushort
- float
- int
- long
- sbyte
- short
- struct
- ulong

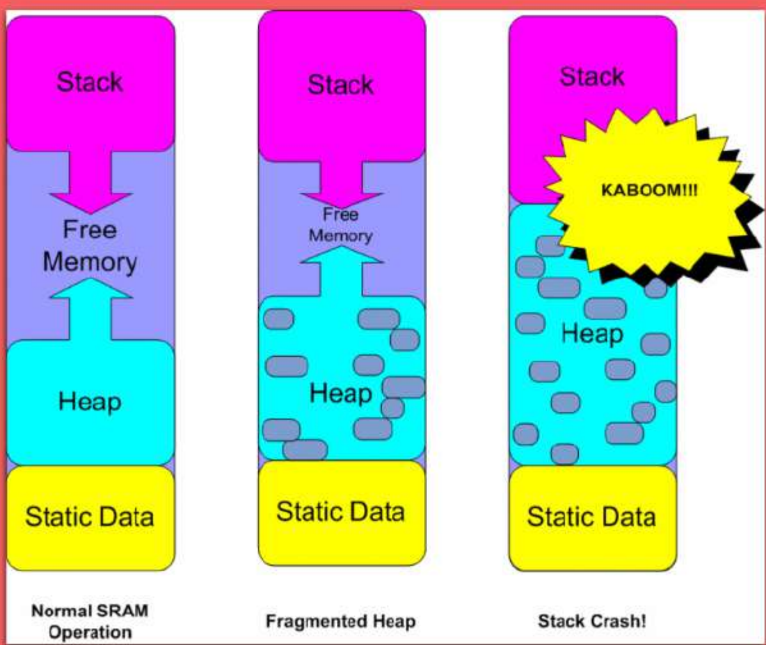
Reference Types ~ *Heap Memory*

- class
- interface
- delegate
- object
- string

1. **Boolean:** Typically 1 byte.
2. **Integer (int):** 4 bytes (32-bit) or 8 bytes (64-bit).
3. **Floating-Point (float/double):** 4 bytes (float) or 8 bytes (double).
4. **Character (char):** 1 byte.
5. **String:** Varies based on length and encoding (1-4 bytes per character in UTF-8).
6. **Arrays:** Depends on element data type and array size.
7. **Structures/Classes:** Varies based on members.
8. **Pointers:** 4 bytes (32-bit) or 8 bytes (64-bit).

Difference Between Stack and Heap Memory

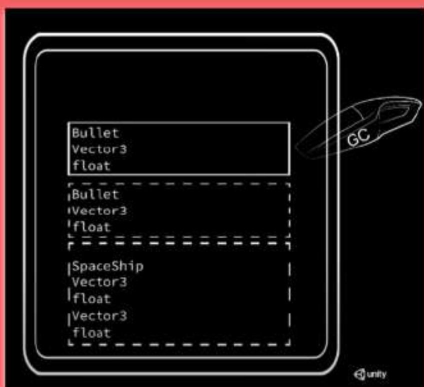
	Stack	Heap
Memory Allocation	Memory allocation is Static	Memory allocation is Dynamic
How is it Stored?	It is stored Directly	It is stored indirectly
Is Variable Resized?	Variables can't be Resized	Variables can be Resized
Access Speed	Its access is fast	Its access is Slow
How is Block Allocated?	Its block allocation is reserved in LIFO. Most recently reserved block is always the next block to be freed.	Its block allocation is free and done at any time
Visibility or Accessibility	It can be visible/accessible only to the Owner Thread	It can be visible/accessible to all the threads
In Recursion Calls?	In recursion calls memory filled up quickly	In recursion calls memory filled up slowly
Used By?	It can be used by one thread of execution	It can be used by all the parts of the application
StackOverflowException	.NET Runtime throws exception "StackOverflowException" when stack space is exhausted	-
When wiped off?	Local variables get wiped off once they lose the scope	-
Contains	It contains values for Integral Types, Primitive Types and References to the Objects	-
Garbage Collector	-	It is a special thread created by .NET runtime to monitor allocations of heap space. It only collects heap memory since objects are only created in heap



Allocation & Garbage Collection

- **Garbage Collector** is a mechanism that cleans up the memory that we previously allocated.
- Even though, we destroy the object, the **object still remain in the memory space**, but it's **marked as unused.**

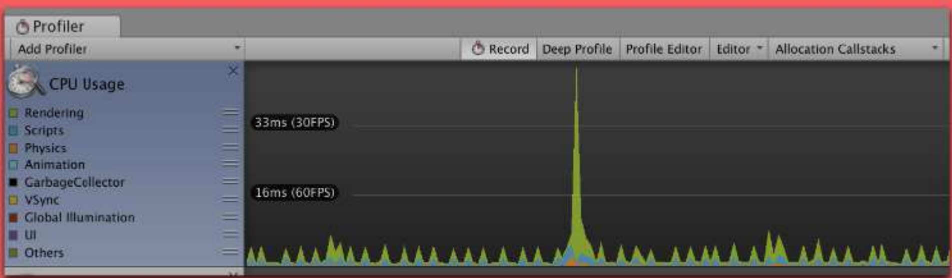
```
void Update()
{
    new SpaceShip();
    new Bullet();
    new Bullet();

    Destroy(SpaceShip);
    Destroy(Bullet);
}

// Garbage Collector executes
```



Unity Garbage Collector

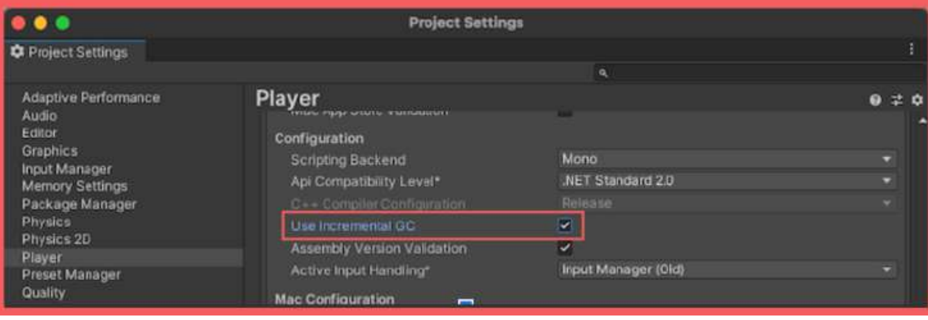
- **Unity** uses **Garbage Collector** to **reclaim memory** from object that the application and Unity are no longer using.
- When a script tries to make an allocation on the **managed heap** but there isn't enough free **heap memory** to accommodate the allocation, **Unity runs the garbage collector.**
- When the **garbage collector** runs, **it examines all objects in the heap**, and marks for deletion any objects that your application no longer references. Unity then deletes the unreferenced objects, which **frees up memory.**

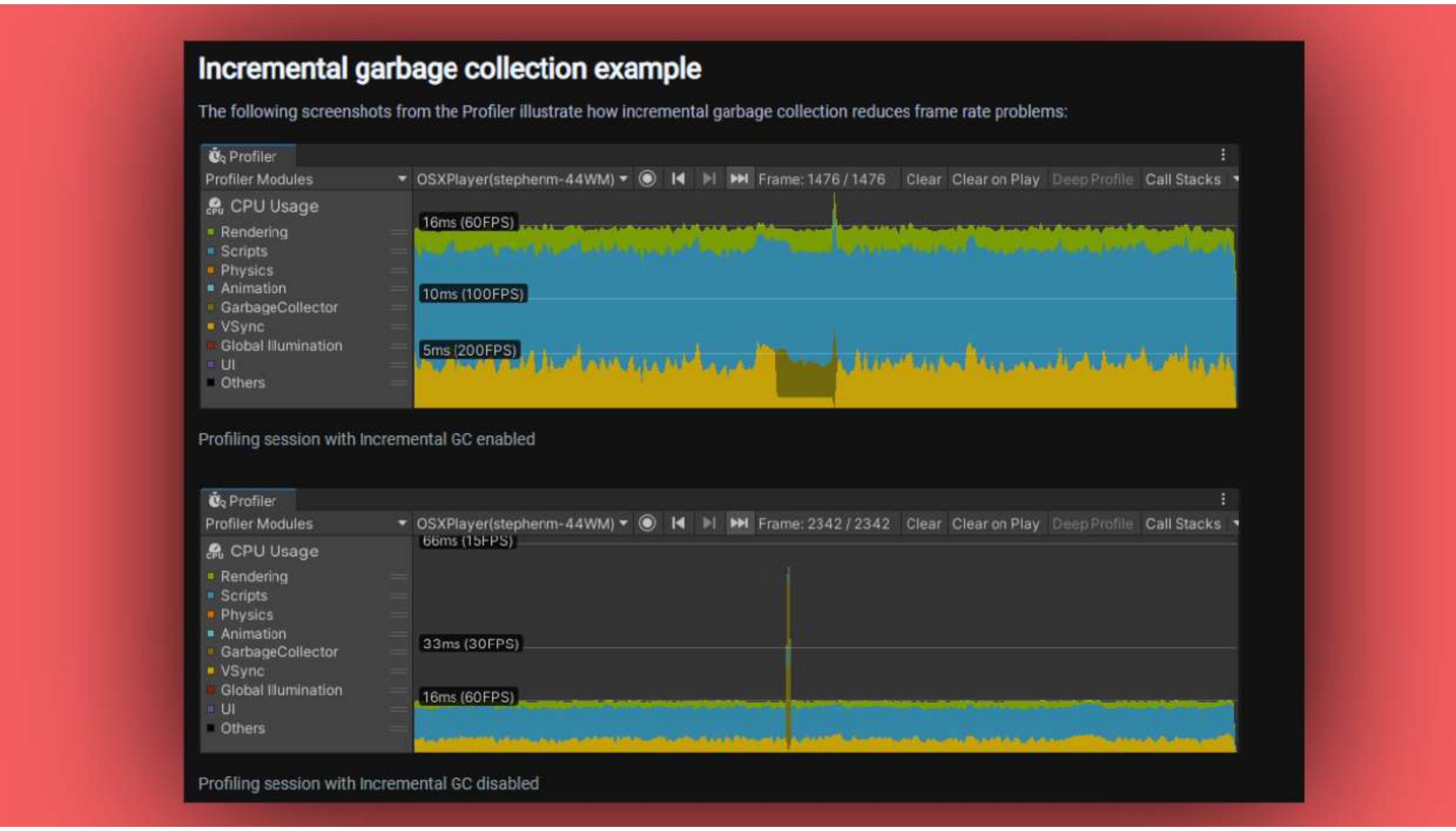
❖ If all memory spaces are allocated and they are in use. What does happen, when garbage collector run in this case? 👀



Unity Incremental Garbage Collector

- **Incremental garbage collection (GC)** spreads out the process of garbage collection **over multiple frames. This is the default garbage collection behavior in Unity.**
- This means that Unity makes **shorter interruptions** to your application's execution, **instead of one long interruption** to let the garbage collector process the objects **on the managed heap.**
- **Incremental mode doesn't make garbage collection faster overall**, but because it distributes the workload over multiple frames, **GC-related performance spikes are reduced.**




Incremental garbage collection example
The following screenshots from the Profiler illustrate how incremental garbage collection reduces frame rate problems:
Profiler
Profiler Modules
OSXPlayer(stephenm-44WM)
Frame: 1476 / 1476
Clear
Clear on Play
Deep Profile
Call Stacks
CPU Usage
Rendering
Scripts
Physics
Animation
GarbageCollector
VSync
Global Illumination
UI
Others
16ms (60FPS)
10ms (100FPS)
5ms (200FPS)
Profiling session with Incremental GC enabled
Profiler
Profiler Modules
OSXPlayer(stephenm-44WM)
Frame: 2342 / 2342
Clear
Clear on Play
Deep Profile
Call Stacks
66ms (15FPS)
CPU Usage
Rendering
Scripts
Physics
Animation
GarbageCollector
VSync
Global Illumination
UI
Others
33ms (30FPS)
16ms (60FPS)
Profiling session with Incremental GC disabled

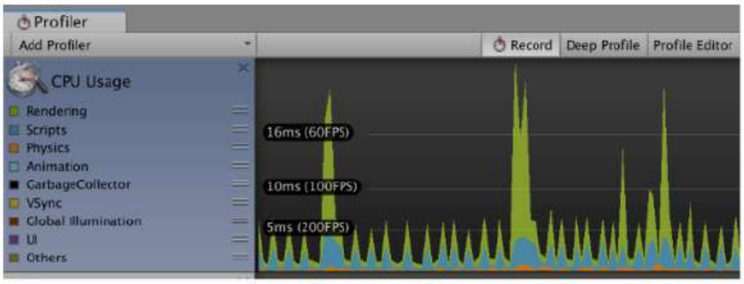
Unity Manual Garbage Collection

Unity

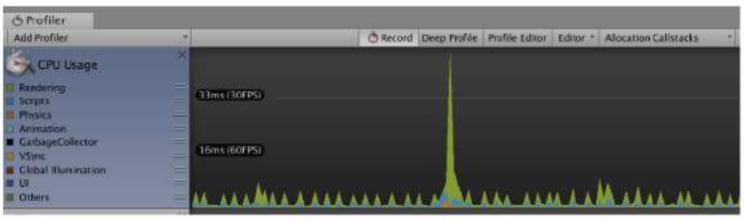
```csharp
public class GarbageCollectionManager : MonoBehaviour
{
  [SerializeField] private float maxTimeBetweenGarbageCollections = 60f;
  private float _timeSinceLastGarbageCollection;

  private void Start()
  {
#if !UNITY_EDITOR
    GarbageCollector.GCMode = GarbageCollector.Mode.Disabled;
#endif
    // You might want to run this during loading times, screen fades and such.
    // Events.OnScreenFade += CollectGarbage;
  }

  private void Update()
  {
    _timeSinceLastGarbageCollection += Time.unscaledDeltaTime;
    if (_timeSinceLastGarbageCollection > maxTimeBetweenGarbageCollections)
    {
      CollectGarbage();
    }
  }

  private void CollectGarbage()
  {
    _timeSinceLastGarbageCollection = 0f;
    Debug.Log("Collecting garbage"); // talking about garbage...
#if !UNITY_EDITOR
    // Not supported on the editor
    GarbageCollector.GCMode = GarbageCollector.Mode.Enabled;
    GC.Collect();
    GarbageCollector.GCMode = GarbageCollector.Mode.Disabled;
#endif
  }
}
```

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. This can break the immersion of the player or cause him to make a mistake he wouldn't have otherwise made. Spikes can be seen as a high points on Profiler Graph.



Spikes are mainly caused by complex calculations or difficult operations performed during a single frame. Annoying Spikes are a problem in high-intensity games which need a stable frame rate and high control over the game to feel good, such as driving or shooting games (FPS).



Unity

Every Frame Costs and Loading Time

Unity

Every-frame costs

Every-frame costs are the calculations and operations that are run every single frame. These can be, for example, physics calculations, running AI behavior or handling animations of characters.

Every-frame costs slow down the general frame rate of the game. They are the little things that slow the game down and make it feel less fluid. If a game just generally runs poorly, this is the area that needs work.

Loading Time

Loading time refers to how long the game takes to load. This includes the first load when the game is opened, and loading that happens during runtime, for example between scenes.

While not usually a major issue, having extremely long loading times or having loading screens appear far too often can negatively affect the user experience.

To reduce the length of loading screens, consider splitting up the work done during them. This can mean preloading assets beforehand to reduce the number of objects that need to be loaded during loading screen, or reducing the complexity of loaded scenes.

Unity Profiling Tools

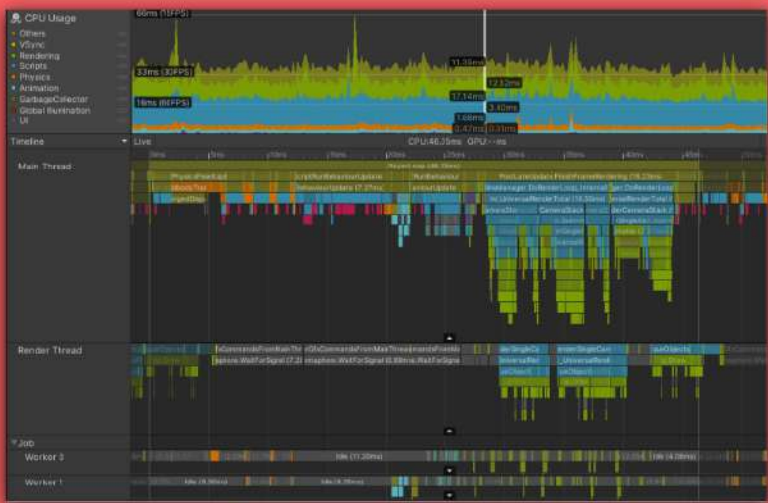
- Profiler
  - CPU Usage
  - Memory
- Memory Profiler
- Code Coverage
- Others

Unity

Unity Profiler

Unity

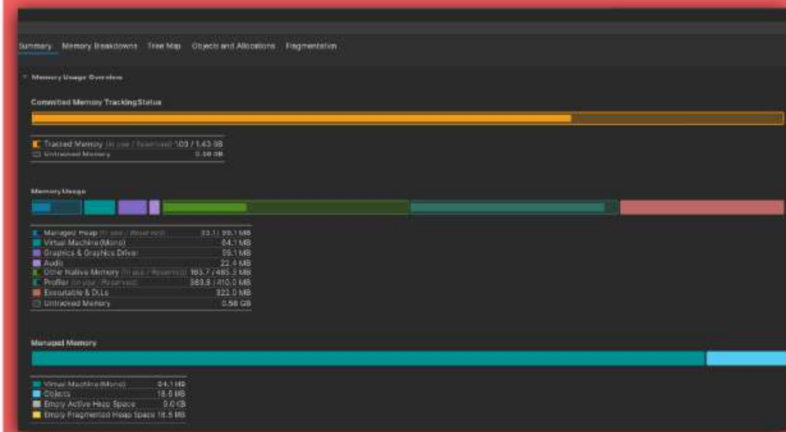
- **The Unity Profiler** is a tool that you can use to get **performance information** about your application.
- **The Profiler gathers and displays data** on the performance of your application in areas such as the **CPU, memory, renderer, and audio.**



Memory Profiler

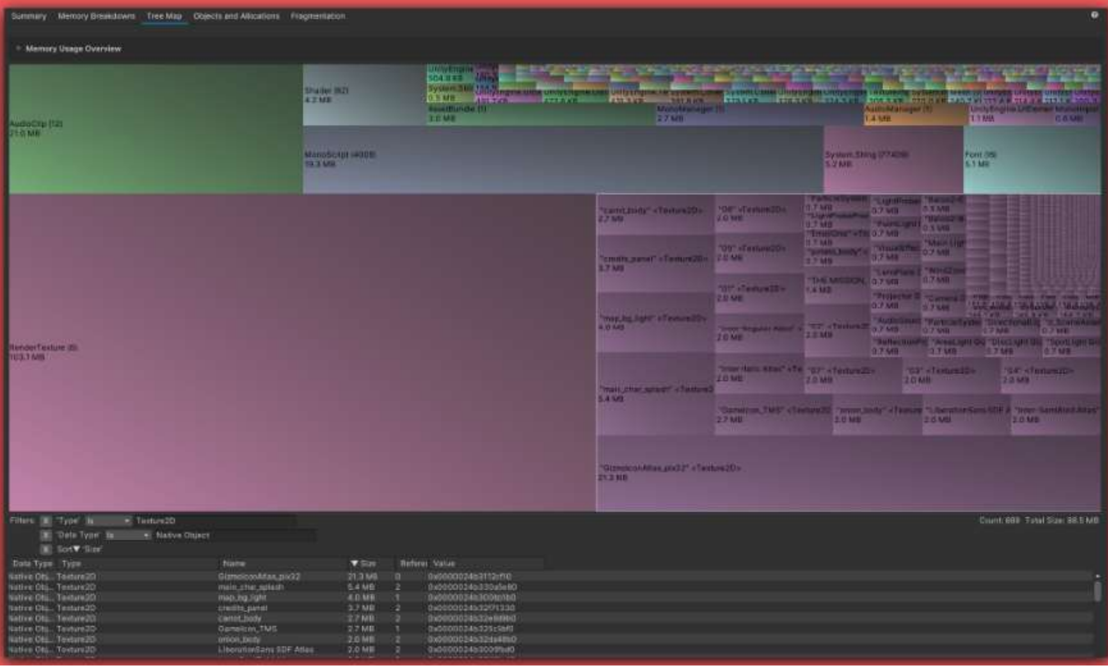
Unity

- **The Memory Profiler** is a tool you can use to **inspect the memory usage** of your **Unity application** and **the Unity Editor.**
- **The profiler** also **provides** an **overview of native and managed memory allocations,** to assess your application's memory use and **identify potential issues such as memory leaks.**



Memory Profiler Tree Map

The profiler allows you to visualize which of the assets used in the scene consumes the most memory.



Code Coverage

- **Code Coverage** is a **measure** of how much of your code has been **executed**.
- The tool **generates an HTML coverage report** showing **which lines of your code are covered by tests.**

Coverage

Collapse all | Expand all

Grouping: By assembly

Filter:

Name	Covered	Uncovered	Coverable	Total	Line coverage	Covered	Total	Branch coverage
– Assembly-CSharp	**782**	**271**	**1053**	**2412**	**74.2%**	**0**	**0**	
AlgebraExtension	3	6	9	18	33.3%	0	0	
Animations.DieAnimation	17	0	17	41	100%	0	0	
Animations.EnemyAttackTestAnimation	0	6	6	18	0%	0	0	
Animations.EnemyBlood	17	0	17	39	100%	0	0	
Animations.EnemyDieTestAnimation	0	9	9	22	0%	0	0	
Animations.SimpleFlash	17	0	17	80	100%	0	0	
Animations.TakeHitAnimation	49	0	49	96	100%	0	0	
Animations.TakeHitAnimationTest	0	6	6	19	0%	0	0	
Cursor.CursorHandler	24	0	24	62	100%	0	0	
Enemy.EnemyController	84	4	88	154	95.4%	0	0	
EnemyAttackAnimation	32	0	32	65	100%	0	0	
Gameplay.AnimationEventTransmitter	3	0	3	16	100%	0	0	
Gameplay.AudioManager	13	24	37	79	35.1%	0	0	
Gameplay.EnemySpawner	44	12	56	110	78.5%	0	0	
Gameplay.GameManager	4	7	11	30	36.3%	0	0	
Gameplay.Sound	0	4	4	79	0%	0	0	
Gameplay.SpawnData	7	0	7	110	100%	0	0	
Gameplay.TimerManager	30	10	40	72	75%	0	0	
Gameplay.TriggerTransmitter	0	4	4	15	0%	0	0	
Main_Character.CharacterDirectionController	48	0	48	90	100%	0	0	
Main_Character.CharacterHealthTestAnimation	0	12	12	26	0%	0	0	
Main_Character.CharacterSpriteSymmetry	31	9	40	74	77.5%	0	0	
Main_Character.CharacterWeaponController	36	7	43	95	83.7%	0	0	
Main_Character.MainCharacterController	87	18	105	190	82.8%	0	0	
Samples.Basic.Scripts.Singleton[T]	0	7	7	23	0%	0	0	
UI.DamageIndicator	26	0	26	61	100%	0	0	
UI.DamageIndicatorFactory	21	3	24	62	87.5%	0	0	
UI.EyesLookToInput	11	4	15	38	73.3%	0	0	
UI.GameOverCanvas	7	71	78	150	8.9%	0	0	
UI.GameplayTimerScore	17	4	21	46	80.9%	0	0	
UI.HealthBar	21	3	24	57	87.5%	0	0	
UI.ScoreIndicator	23	0	23	52	100%	0	0	
UI.ScoreIndicatorFactory	23	3	26	64	88.4%	0	0	
UI.ScreenManager	15	18	33	71	45.4%	0	0	
UI.SpecialButton	15	2	17	43	88.2%	0	0	

Break Section

15 mins.

Optimization Techniques

- Flyweight Pattern
- Scriptable Objects
- Object Pooling
- Others

Unity

Flyweight Pattern

Memory Savings:

- **Flyweight pattern optimizes memory usage** by sharing common data among objects;
  - **Reducing memory overhead**
  - Promoting efficiency, especially in cases where **multiple instances** have similar attributes.

Example in Unity:

- Consider a game with numerous identical trees. Instead of creating a separate object for each tree's **static attributes (intrinsic state)**, the **Flyweight pattern** can be used to share these attributes among all trees, **saving memory.**

Combined with Other Patterns:

- **The Flyweight pattern** can be combined with other design patterns, such as the **Object Pool pattern**, to further optimize **resource usage and performance.**

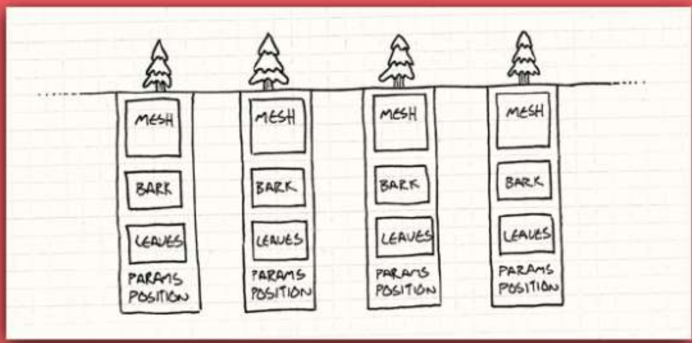
LET'S TAKE A LOOK TO SAMPLE 👀

Unity

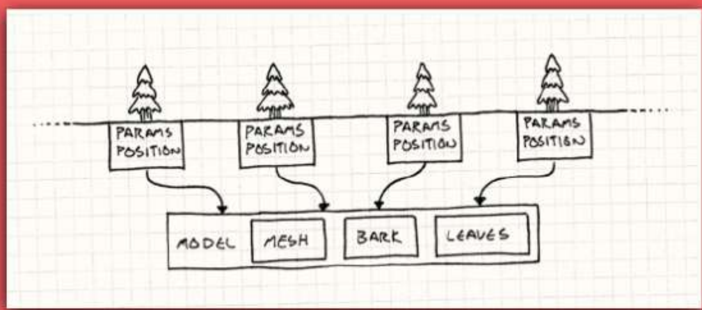
Sample Overview

Unity

Non Optimized



Optimized



Scriptable Objects

Shared Data:

- They allow **data to be shared among multiple GameObjects**, minimizing duplicated data and **conserving memory resources.**

Memory Efficiency:

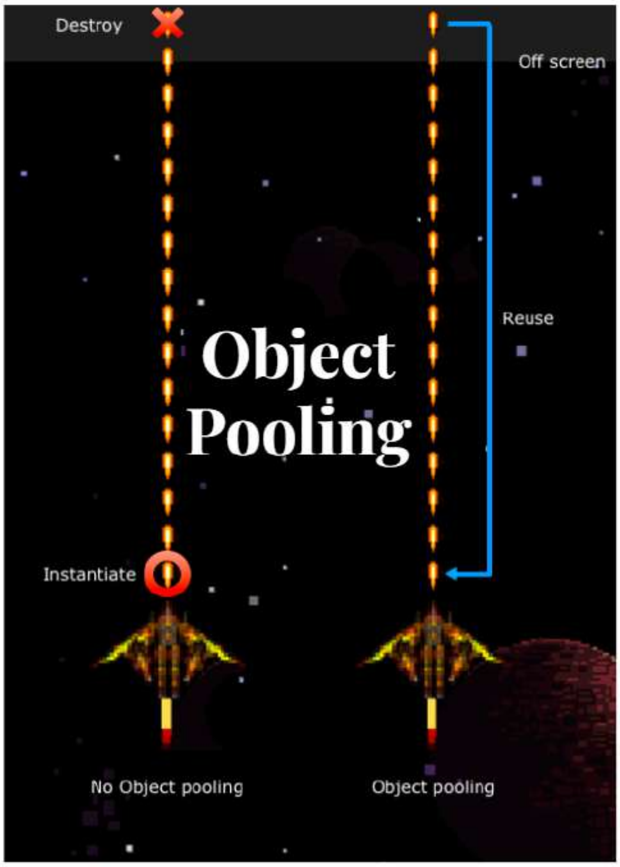
- They consume **less memory** compared to traditional **MonoBehaviour components**, as they're not tied to GameObject instances.

Serialization Optimization:

- Scriptable Objects **serialize efficiently**, **leading to faster scene loading times** and **less impact on overall game performance.**

Leveraging Scriptable Objects for optimization empowers you to create a **more memory-efficient**, **performance-optimized**, and **modular Unity project**, resulting in smoother gameplay and a better user experience.

Are There Similarities Between Scriptable Objects and the Flyweight Pattern? 🤔



Memory Efficiency:

- Reduces **memory fragmentation.**
- Eliminates frequent memory allocation.

Garbage Collection Reduction:

- Minimizes **performance-hindering garbage collection.**

Performance Boost:

- **Faster object usage** compared to creation.
- **Maintains consistent performance.**

Lower CPU Load:

- **Decreases CPU overhead** from allocation and deallocation.
- Optimizes core gameplay functions.

Scalability and Mobile Benefits:

- **Crucial for mobile games** with resource constraints.
- Supports handling more objects on-screen.

Object pooling stands as a key technique **for optimizing Unity games,** leading to **efficient memory usage, reduced CPU strain,** and **an overall smoother** and **more responsive player experience.**

What's memory fragmentation? 🤔

Memory Fragmentation

Unity

Memory fragmentation in Unity occurs when memory is allocated and deallocated in a way that leaves small, **scattered memory blocks.** This can lead to problems like:

- Failed **larger memory requests** due to lack of contiguous space.
- **Slower allocation and deallocation**, hampering performance.
- More frequent **garbage collection** and **performance spikes**.
- **Inefficient memory use** and wasted resources.

Object pooling and **efficient memory management** help **reduce fragmentation** and its **negative impact on Unity game performance.**

Is it possible to track memory fragmentations in Unity? 🤔

Are you curious for more?

Learn more about these concepts:

- Asset Bundles
- Addresabbles
- Multithreading
    - Job System
    - Burst Compiler

Agenda

- Script Optimizations
  - Choosing the right data structure
  - Remove Empty Unity Events
  - Hash the value instead
  - GameObject.AddComponent(...)
  - Cache the References
  - Coroutines
  - Extern Call Caching
  - Find Objects(Type, Tag)
  - Linq Usages
  - String Builder
  - Others

Unity

Choosing the right data structure

Unity

- ❑ GameObject[ ] enemies;

or

- ❑ Dictionary<int, GameObject> enemies;

or

- ❑ List<GameObject>() enemies;

....

....

....

Choosing the right data structure

Lists:

- **Dynamic size.**
- Flexible **insertion/removal.**
- Slightly **higher memory overhead.**
- **Slower** random **access.**

Arrays:

- **Compact memory** layout.
- **Fast** random **access.**
- **Fixed size.**
- **Resize** operations can be costly.

Dictionaries:

- **Fast** key-based **access.**
- Ideal for associations.
- Can **consume** more **memory.**
- Hash function quality **impacts performance.**

Considerations:

- **Choose lists for dynamic sizing and flexibility.**
- **Use arrays for predictable memory access and fixed sizes.**
- **Opt for dictionaries for efficient key-based lookups.**
- **Always consider the specific needs of your program.**

Unity

Are you curious for more?

Learn more about these concepts:

- Linked Lists
- Stacks
- Heaps
- Queues
- Hash Tables
- Trees (Binary, AVL, Red-Black)

Remove Empty Unity Events

Unity

- // Start is called before the first frame update

```
void Start( )
{
}
```

- // Update is called once per frame

```
void Update( )
{
}
```

Hash to Value Instead

```
animator.SetTrigger("Jump");

material.SetTexture("_MainColor", _color);
```

```
// Getting the hashed value and caching
int parameterId = Animation.StringToHash("Jump");
animator.SetTrigger(parameterId);

// For materials, use the Shader class
int propertyId = Shader.PropertyToID("_MainColor");
material.SetTexture(propertyId, _color);
```

Unity

GameObject.Add Component(...)

```
GameObject newBarrel = Instantiate(template);
newBarrel.AddComponent(typeof(CharacterData));
newBarrel.AddComponent(typeof(BrekableObject));
newBarrel.AddComponent(typeof(LootSpawner));
newBarrel.AddComponent(typeof(NavMeshObstacle));
```

Garbage Collection:

- Frequent use can trigger **memory allocation** and **garbage collection,** causing performance spikes.

Component Initialization:

- **Adding** components **dynamically** can **impact frame times** due to **initialization**.

Dynamic Memory:

- Can **lead** to **fragmented memory allocation** over time.

Dependencies:

- Components might require specific setups, leading to **unexpected behaviour.**

Unity

Agenda

- ....
- Cache the References
- Avoid new keyword in Coroutine
- Extern Call Caching (transform etch.)
- Avoid to use Find Objects(Type, Tag)
- Reduce LINQ Usages
  - *Just How Much Garbage Does LINQ Create?*
- Use String Builder for String manipulations to reduce memory allocations.
- ....

Unity

Agenda

Unity

- Unity Optimization Tips
  - Reduce Hierarchy Complexity
  - Be Careful about Resources Folder
  - ...

Any Others? 🤔

THANKS!

Any Feedbacks? 🤔