



8: Performance and Optimisation

Assignment Roadmap

- **Assignment 1**
 - Week 9 – Peer review of game and controller
- **Assignment 2**
 - Week 8 – Draft Poster presentation
 - Week 10 – Report Peer Review
- **Next up: WEEK 8 – Draft Poster Presentation**

Learning outcomes

- **Understand** rationale behind UML
- **Understand** a subset of UML Diagrams useful for game development
- **Develop** some UML Diagrams

Introduction

- One of the important aspect of Game Programming is optimising for performance
- We need to understand the hardware our games will be deployed onto
- We need to understand the programming languages we use
- We need to understand the Game Engine we develop on
- And finally we need to understand the tools we can use to tune performance



MEMORY

Introduction

- Memory in most modern programming languages are allocated in two spaces
 - Dynamic Memory (allocated with **new**) is allocated on the **Heap** and will **grow** in size
 - Stack memory (everything that doesn't use **new**) is allocated on the **Stack** and is **fixed** size

Stack Memory

- When you allocated values types (int, float, bool, short, char etc), these allocated on the stack
- Values allocated on the stack are local, these are deallocated when they drop out of scope
- Values passed into functions are copied onto the stack
- The stack is of fixed size
 - 1MB for C#

Stack Memory Example

```
void Update()  
{  
    int x=10;  
    int y=10;  
    Vector2 pos=Vector2(x, y);  
} //<-- x, y and pos drop out of scope here
```


Heap Memory

- Heap memory is allocated dynamically
- Any type allocated using the **new** keyword are allocated on the heap
- We as programmers have responsibility for allocating on the heap
- But ... in C# the Heap Memory is managed by the Garbage Collector
 - In C++ we have to allocate and deallocate on the Heap!

Stack Memory Example

```
public class MonsterStats
{
    private int health ;
    private int strength ;

    public MonsterStats ( )
    {
        health=100;
        strength =10;
    }
    public void ChangeHealth (int h)
    {
        health+=h ;
    } //<- h drops out of scope here

    void ChangeStrength(int s )
    {
        strength+=s ;
    } //<- s drops out of scope here
}

void Start( )
{
    //Create an instance of the class on the Heap
    MonsterStats new stats=MonsterStats ( ) ;
    stats.ChangeHealth(10) ;
    stats.ChangeStrength(-2) ;
}
```

Data Types and Memory in C#

- **Values types** such as int, float, etc are allocated on the stack
- **struct's** are custom **values types** so are allocated on the stack (except on a few cases)
- Reference Types are allocated on the Heap and include **class**, **interface** and **delegate** types



STRINGS

Introduction

- Strings act and look like value types are actually reference types
- This means we need to be careful in allocating new strings
- **And** each time we create a new string using concatenation (+)
- If we are creating lots of new strings we should use the **StringBuilder** class

String Builder Examples

```
//We need to use the namespace - System.Text  
using namespace System.Text
```

```
//Create the string builder with a capacity of -  
1024 and max capacity of 1024  
StringBuilder sb=new StringBuilder(1024,1024);
```

```
//Append some text  
sb.Append("Name: ");  
sb.Append("Brian");  
sb.Append(" Health: ");  
sb.Append(100);
```

```
//Get the String from the String Builder  
string s=sb.ToString();
```



MEMORY MANAGEMENT

Garbage Collection

- C# uses garbage collection to clean up deallocated objects that have been allocated on the heap
- This is an automatic process and has been tuned for maximum performance
- **However** you should understand how this process works and create code which ensures that garbage collection only runs when needed

Garbage Collection Tips – Cache

- Cache, if you call functions which allocated memory on the heap (**Find**, **GetComponent** etc)
- Consider moving these out of **Update** functions and retrieve in the **Start** function

Caching Example

```
void Update( )  
{  
    //Get Health Component and check health  
    Health health=GetComponent<Health>();  
  
    If (health.IsDead())  
    {  
        //Do Something  
    }  
}
```

- The above code allocates on the heap and gets deallocated every update
- Casing not only unnecessary allocation but deallocation via the Garbage Collector

Caching Example - Fixed

```
private Health health;
```

```
void Update( )  
{  
    health=GetComponent<Health>();  
}
```

```
void Update( )  
{  
    If (health.IsDead())  
    {  
        //Do Something  
    }  
}
```

Garbage Collection Tips – Allocation

- Don't allocate on the heap in Update functions (use **caching**)
- Also consider calling function on a timer if you need to allocate frequently, this will reduce the amount of allocations in update

Garbage Collection Tips – Reuse Collections

- Don't initialise collections using the **new** keyword in the **Update** function
- Initialise on the **Start** function and call the **Clear** function of the collection if you need to fill with new data
- This all holds true for some Unity functions that return arrays such as **FindGameObjectsWithTag**

More Garbage Collection Tips

- <https://learn.unity.com/tutorial/fixing-performance-problems#>

UNITY PERFORMANCE TIPS

Unity Optimisation Tips

- Optimisation in Unity - <https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html>
- UI - <https://create.unity3d.com/Unity-UI-optimization-tips>
- Optimising Graphics – <https://create.unity3d.com/Unity-UI-optimization-tips>

PROFILER LIVE DEMO

Profiler

- Test Project - <https://github.com/Falmouth-Games-Academy/COMP140-Profiler-Test>

Profiler – Additional Reading

- Profiling in Unity -
<https://learn.unity.com/tutorial/profiling-applications-made-with-unity>
- Practical Guide to Profiling in Unity -
<https://www.youtube.com/watch?v=OSlOwJP8Z14>