Let's run some benchmarks!

In this chapter I am executing scripts in C# and U#, and compare execution times

U# vs C#

U# is noticeably slower than regular C#, to benchmark it I decided to execute two scripts in U# and C#, one calculates the n-th Fibonacci number, the second one generates a maze.

To make those tests a bit fairer, I disable the C# compiler optimization, since the U# compiler does not optimize your scripts during compilation.

Recursive functions

For this test I executed a script that calculates the n-th Fibonacci number, the Fibonacci method looks like this

```
[RecursiveMethod]
public int FibonacciRecursive(int n)
{
    if (n <= 1)
        return n;
    else
        return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);
}
```

Since `FibonacciRecursive` is a recursive function, the `[RecursiveMethod]` attribute need to be added.

This is really a horrible way to calculates the n-th Fibonacci number because certain method calls get called multiple times, and the number of method calls rise very quickly :

- `Fibonacci(1)` calls the Fibonacci method 1 time
- `Fibonacci(2)` calls the Fibonacci method 3 times
- `Fibonacci(10)` calls the Fibonacci method 177 times
- `Fibonacci(22)` calls the Fibonacci method 57313 time

So I felt like this test is a great way to determine how well U# performs this task, and I executed `Fibonacci(22)` in C# and U#

> C# time : 0.189 ms
> U# time : 684.577 ms
> U# was 3629 times slower!

Maze generator

In my "Circuit Master" world, I wrote a custom maze generator algorithm, which I unfortunately cannot share because the algorithm is about 1600 lines long.

I decided to compare the execution time of that algorithm because that would be a more realistic test, it uses pretty much everything (Unity functions, custom list implementations, for-loops, bit manipulations etc.).

The algorithm does not call any recursive functions, but implements a custom stack made out of a ring buffer (ttps://en.wikipedia.org/wiki/Circular_buffer.) which is a very fast container to build and read a stack

> C# time : 1.609 ms
> U# time : 972.712 ms
> U# was 604 times slower!

Based on the previous results, we can see that U# is much slower than regular C#, so keeping your code optimized is even more important in U#!

For loop

For this test I was curious to see how well Udon executes for-loops, and the results I got were really unexpected.

I executed two methods `Benchmark1` and `Benchmark2` and compared their execution time.

```
public override void Benchmark1()
{
    int number = 0;
    for (int i = 0; i < 50000; i++)
    {
        number++;
    }
}

public override void Benchmark2()
{
    int number = 0;
    for (int i = 0; i < 50000; i+=5)
    {
        number++;
        number++;
        number++;
        number++;
        number++;
    }
}
```

Both methods execute a for-loop, and both of them go from 0 to 50000, except that the second method has the step set to 5 and each loop executes 5 instructions instead of 1. So both methods do the exact same thing, except that the second one only loops 10000 times instead of 50000.

> B1 : 79.903 ms (2.2 times slower)
> B2 : 35.388 ms

This benchmark surprised me the most, it seems like the for-loop has a pretty noticeable overhead

Recursive vs iterative

For this test I was curious to see how well Udon executes recursive methods, in the previous test I already compared the execution time of recursive functions in U# and C#, but here I wanted to compare the execution time between a recursive function and an iterative function

I executed two methods `Benchmark1` and `Benchmark2` and compared their execution time.

```
[RecursiveMethod]
private int Recursive(int n)
{
    if (n <= 0)
        return 0;
    else
        return 1 + Recursive(n - 1);
}

private int NotRecursive(int n)
{
    int ret = 0;
    while (n > 0)
    {
        ret++;
        n--;
    }
    return ret;
}

public override void Benchmark1()
{
    Recursive(50000);
}

public override void Benchmark2()
{
    NotRecursive(50000);
}
```

`Benchmark1` calculates a value recursively, `Benchmark2` calculates it iteratively.

B1 : 532.552 ms (6.6 times slower)
B2 : 79.8591 ms

Recursive functions are much slower, they should be avoided!
The reason is that Udon builds a custom stack to save the variables of each function call, otherwise the next function call would override the variables from the previous function call. Implementing such a stack is performance heavy.

Builtin functions vs calculating something manually

Let's say you want to calculate the distance between two vectors.

Some of you would probably write something like this :

```
distance = Vector3.Distance(a, b);
```

Others might write something like this (which is how Unity implemented the `Distance` method) :

```
float num = a.x - b.x;
float num2 = a.y - b.y;
float num3 = a.z - b.z;
distance = (float)System.Math.Sqrt(num * num
    + num2 * num2
    + num3 * num3);
```

In regular C#, that wouldn't make a difference, because C# code is compiled and both examples above would probably get compiled into something similar.

That's not the case in Udon, each line of code has an execution time because Udon is an interpreted programming language, so the first example would call a compiled C# function (fast), the second example would calculate the distance in Udon (slow)

This benchmark is mostly to show you that is is better to use builtin function instead of calculating something manually. Before implementing a function, make sure that the function hasn't already been implemented yet, the `Mathf` class implements many math functions!

I executed two methods `Benchmark1` and `Benchmark2` and compared their execution time.

```
private int RandomVal()
{
    return Random.Range(0, 100);
}
```

```
public override void Benchmark1()
{
    float distance;
    for (int i = 0; i < 10000; i++)
    {
        Vector3 a = new Vector3(RandomVal(), RandomVal(), RandomVal());
        Vector3 b = new Vector3(RandomVal(), RandomVal(), RandomVal());
        float num = a.x - b.x;
        float num2 = a.y - b.y;
        float num3 = a.z - b.z;
        distance = (float)System.Math.Sqrt(num * num
            + num2 * num2
            + num3 * num3);
    }
}

public override void Benchmark2()
{
    float distance;
    for (int i = 0; i < 10000; i++)
    {
        Vector3 a = new Vector3(RandomVal(), RandomVal(), RandomVal());
        Vector3 b = new Vector3(RandomVal(), RandomVal(), RandomVal());
        distance = Vector3.Distance(a,b);
    }
}
```

Both methods calculate distances, but the second method calls the builtin Distance method.

> B1 : 166.699 ms (1.79 times slower)
> B2 : 92.678 ms

> Use builtin functions as much as you can!

Function overhead test

For this test I was curious to see the overhead of a function call

I executed two methods `Benchmark1` and `Benchmark2` and compared their execution time.

```
private void Func()
{
    int j = 0;
}

public override void Benchmark1()
{
    for (int i = 0; i < 50000; i++)
    {
        int j = 0;
    }
}

public override void Benchmark2()
{
    for (int i = 0; i < 50000; i++)
    {
        Func();
    }
}
```

Both methods do the same thing, setting a variable j to 0, except that `Benchmark2` calls a function.

> B1 : 57.036 ms
> B2 : 66.207 ms (1.16 times slower)

> There's a little difference, calling a function has a little overhead, but nothing too bad.
> So in theory, putting everything into a single function is more performant (But obviously that would be a bad programming advice)

GetComponent<>()

Many of you may already know that calling `GetComponent` is pretty expensive in Unity.

But how expensive is it in Udon? Let's see!

I executed two methods `Benchmark1` and `Benchmark2` and compared their execution time.

```
public override void Benchmark1()
{
    Labyrinth labyrinth = GetComponent<Labyrinth>();
    for (int i = 0; i < 1000; i++)
    {
        labyrinth.InitGrid(2, 2);
    }
}

public override void Benchmark2()
{
    for (int i = 0; i < 1000; i++)
    {
        Labyrinth labyrinth = GetComponent<Labyrinth>();
        labyrinth.InitGrid(2, 2);
    }
}
```

Both methods do the same thing, except that `Benchmark2` calls `GetComponent` inside a for-loop

> B1 : 60.0576 ms
> B2 : 111.367 ms (1.85 times slower)

> The difference is very noticeable, I'ld hightly recommend to call GetComponent<>() only once, for instance in Start()

Calling methods from a separate script

Let's say you have script A that accesses a method from script B.

Would it be more performant to merge script A and B together? Let's see!

```
public Fibonacci FibonacciInstance;

[RecursiveMethod]
private int FibonacciRecursive(int n)
{
    if (n <= 0)
        return 0;
    else
        return 1 + FibonacciRecursive(n - 1);
}

public override void Benchmark1()
{
    for (int i = 0; i < 10000; i++)
    {
        FibonacciRecursive(2);
    }
}

public override void Benchmark2()
{
    for (int i = 0; i < 10000; i++)
    {
        FibonacciInstance.FibonacciRecursive(2);
    }
}
```

Both methods do the same thing :

- `Benchmark1` calls `FibonacciRecursive` in the same script
- `Benchmark2` calls `FibonacciRecursive` from a separate script

> B1 : 236.120 ms
> B2 : 305.513ms (1.29 times slower)

So yes, calling a method from a separate script affects the performance.

Caching Networking.LocalPlayer

Some programmers like to cache the local player for later use, for instance by adding a private member `private VRCPlayerAPI _localPLayer;` then setting the local player `_localPlayer = Networking.LocalPlayer;` in Start()

Let's see how it affects the performance :

```
public override void Benchmark1()
{
    VRCPlayerApi localPlayer = Networking.LocalPlayer;
    for (int i = 0; i < 10000; i++)
    {
        string name = localPlayer.displayName;
    }
}

public override void Benchmark2()
{
    for (int i = 0; i < 10000; i++)
    {
        string name = Networking.LocalPlayer.displayName;
    }
}
```

> B1 : 15.097 ms
> B2 : 18.251 ms (1.20 times slower)

> Caching the local player can improve the performance if the local player is used multiple times in the script.

This does not only apply to Networking.LocalPlayer! for instance if you need to access the Transform of a particular GameObject it might be interesting to cache the Transform for later use:

```
Transform myTransform;

void Start()
{
  myTransform = myGameObject.transform;
}

void Update()
{
  //this is more performant than "myGameObject.transform.position"
  //especially if the transform is used multiple times, like in this Update()
  myTransform.position = newPosition;
}
```

The "ref" keyword

U# now supports the "ref" keyword, which is really cool! For those who don't know what the "ref" keyword does, I'll link the C# documentation here : https://learn.microsoft.com/en-US/dotnet/csharp/language-reference/keywords/ref

But does it affect the performance in U#?

```
private void FunctionRef(ref int a)
{
    a = 1;
}

private int FunctionRet(ref int a)
{
    return 1;
}

public override void Benchmark1()
{
    for (int i = 0; i < 50000; i++)
    {
        int a = 0;
        FunctionRef(ref a);
    }
}

public override void Benchmark2()
{
    for (int i = 0; i < 50000; i++)
    {
        int a = 0;
        a = FunctionRet();
    }
}
```

Both methods do the same thing, setting a variable `a` to 1, but the first script passes a reference.

B1 : 14.859 ms (1.04 times slower)
B2 : 14.171 ms

Good news! The difference is really negligible, you can safely use "ref" without worrying about performance impacts.

400 Update() calls vs one Update() call iterating 400 times

What would be more performant in Udon?

- 400 GameObjects executing some code every frame with an Update event :

```
public class EveryFrame : UdonSharpBehaviour
{
    void Update()
    {
        transform.position = Vector3.zero;
    }
}
```

- One GameObject with one Update event, but that Update event iterates though 400 GameObjects

```
public class EveryFrameHandler : UdonSharpBehaviour
{
    public EveryFrameCustomUpdate[] ArrayElements; //this array contains 400 elements

    void Update()
    {
        foreach(var el in ArrayElements)
        {
            el.CustomUpdate();
        }
    }
}

public class EveryFrameCustomUpdate : UdonSharpBehaviour
{
```

```
    public void CustomUpdate()
    {
        transform.position = Vector3.zero;
    }
}
```

I benchmarked it using the Udon Profiler by Merlin :
https://gist.github.com/MerlinVR/2da80b29361588ddb556fd8d3f3f47b5

> First example : 0.90ms per frame in average
> Second example : 2.06ms per frame in average

There are mostly two reasons explaining this difference :

- For-loops have a noticeable overhead, which I benchmarked here : https://vrclibrary.com/wiki/books/udon-benchmarking-and-performance-tests/page/for-loop
- Executing methods from a separate script also have a noticeable overhead : https://vrclibrary.com/wiki/books/udon-benchmarking-and-performance-tests/page/calling-methods-from-a-separate-script

Out of curiosity, I also replaced the for-loop with 400 lines of code I generated :

```
// ...
// ArrayElements[0~114].CustomUpdate();
ArrayElements[115].CustomUpdate();
ArrayElements[116].CustomUpdate();
ArrayElements[117].CustomUpdate();
ArrayElements[118].CustomUpdate();
ArrayElements[119].CustomUpdate();
ArrayElements[120].CustomUpdate();
// ArrayElements[120~399].CustomUpdate();
// ...
```

And I got 1.65ms per frame in average.