

Udon Benchmarking and performance tests

A few weeks ago I benchmarked Udon, and I decided to share the results here.

So if you're curious to know how to make your U# code more performant, you can check it out here!

The introduction just explains how I benchmarked Udon.

In the second chapter, I share all my benchmarking results.

I also give a few optimization tips in the last chapter.

For questions, you can contact me on Twitter (<https://twitter.com/MyroDev>) or post comments.

- [Introduction](#)
- [Let's run some benchmarks!](#)
 - [U# vs C#](#)
 - [For loop](#)
 - [Recursive vs iterative](#)
 - [Builtin functions vs calculating something manually](#)
 - [Function overhead test](#)
 - [GetComponent<>\(\)](#)
 - [Calling methods from a separate script](#)
 - [Caching Networking.LocalPlayer](#)
 - [The "ref" keyword](#)
 - [400 Update\(\) calls vs one Update\(\) call iterating 400 times](#)
- [Other Udon optimization tips](#)

Introduction

Last update : February 2023

As many of you may already know, Udon can be pretty slow, but I was curious to know what really slows it down, so I made a world called "Udon Benchmark" to benchmark it, the world can be found here https://vrchat.com/home/world/wrld_174475ad-6f8e-444d-8b02-67cd13e13b74
In that world I benchmarked :

- The execution time between C# and U#
- As many C# features I could possibly think off, for instance for-loops, function calls, recursive functions etc. and compare their execution times.

Each script got executed 50 times, and the results got averaged.

All results I'll share are the results I got from my world, you'll probably get different results on your hardware.

Since VRChat keeps improving Udon, many results I'll share will probably be out of date in the future, and I cannot promise that I'll keep those values up-to-date.

The values bellow shows a summary of all execution times, I'll explain each line in the next chapter, if you want you can paste those values into a .CSV file, or generate your own .CSV file in my "Udon Benchmark" world that features a CSV exporter.

- B1 : execution time of the Benchmark1 method
- B2 : execution time of the Benchmark2 method
- SD : Standard deviation, which shows the amount of variation or dispersion of a set of values https://en.wikipedia.org/wiki/Standard_deviation

Version 1.1 - Number o iterations : 50	B1 (ms)	B1 min (ms)	B1 max (ms)	B1 SD	B2 (ms)	B2 min (ms)	B2 max (ms)	B2 SD	B1/B2
C# vs U# (Part 1)	684.576 946	661.567	804.889 1	21.4634 2736309 57					

C# vs U# (Part 2)	972.71161	928.6402	1119.3484	35.2456449139138					
For-loop overhead test	79.903544	76.9296	94.951	3.70946646218348	35.388596	33.3341	53.8828	3.81011410065158	2.25788963201592
Recursive vs iterative	532.552632	519.5953	577.7497	12.9280788058929	79.859174	77.0833	93.3914	3.69170508577324	6.6686468858293
Builtin functions vs calculating something manually (part 1)	166.699154	161.6328	189.9542	5.54937905157721	92.678434	88.3964	125.4238	6.55232813505581	1.79868332691077
Builtin functions vs calculating something manually (part 2)	60.58319	57.8675	78.6651	3.79625209517229	63.034632	60.8127	76.2538	2.8186812640978	0.961109600830223
Function overhead test	57.03638	54.4904	65.0939	2.92755877119487	66.20722	63.0205	81.5224	3.63724205056524	0.861482780880997
GetComponent<>()	60.0576	58.0447	69.7074	2.32634479559673	111.367394	106.5221	122.6288	3.81778258895972	0.539274538470389
Calling methods from a separate script	236.120418	228.2506	261.7778	7.89126462852919	305.513994	297.6651	333.578	7.45977174212214	0.772862856161017
Caching Networking.Local Player	15.097448	14.5741	18.5321	0.783616316634614	18.251728	17.7745	26.3932	1.23896447132918	0.827179103260798
The "ref" keyword	14.859624	13.9105	22.6906	1.79010585715594	14.171158	13.4693	18.1117	1.23886350436035	1.04858219772865

Let's run some benchmarks!

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

Let's run some benchmarks!

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 (https://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#!

Let's run some benchmarks!

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

Let's run some benchmarks!

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.

Let's run some benchmarks!

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 it 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!

Let's run some benchmarks!

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)

Let's run some benchmarks!

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'd highly recommend to call `GetComponent<>()` only once, for instance in `Start()`

Let's run some benchmarks!

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.

Let's run some benchmarks!

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;
}
```

Let's run some benchmarks!

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.

Let's run some benchmarks!

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 through 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.

Other Udon optimization tips

Update events, like Update() or FixedUpdate()

It is generally not recommended to execute code every frame, the performance impact can be quite noticeable.

`FixedUpdate()` is even worse because the refresh rate is generally higher. It is actually locked to your current screen's or headset's refresh rate (this behaviour is not documented in the VRChat documentation so it could change in the future...), you can get the fixed delta time with `Time.fixedDeltaTime`.

In Unity, the default fixed update rate is 50 updates per second, that value can be changed in the project settings, but VRChat overrides that value and it is not possible to change the update rate.

So if you're playing VRChat on a 200 Hz monitor, the `FixedUpdate` event will be executed up to 200 times per second!

Benchmarking your Update events

Merlin once published on Gist a Udon Profiler that you can use to benchmark your Updates, it can be downloaded here : <https://gist.github.com/MerlinVR/2da80b29361588ddb556fd8d3f3f47b5>

The profiler shows the execution time of your Udon code for each frame, it works for `Update()`, `FixedUpdate()`, `PostLateUpdate()`, and `LateUpdate()`, it does not benchmark `SendCustomEventDelayedSeconds/Frames()`

Alternatives to Update() #1 : SendCustomEventDelayed

A small Update event loop is generally fine, but if you want to execute an Update loop for each player in the instance, ask yourself the question if you really need that.

Also ask yourself the question if you really need to update something every frame. If you don't need to, you could use `SendCustomEventDelayedSeconds()` or `SendCustomEventDelayedFrames()` and build your own custom Update loop. Those methods take as parameter the event name (you need to use your custom event method), and the delay.

```

private void Start()
{
    CustomUpdateSeconds();
    CustomUpdateFrames();
}

public void CustomUpdateSeconds()
{
    SendCustomEventDelayedSeconds(nameof(CustomUpdateSeconds), 1.0f);
}

public void CustomUpdateFrames()
{
    SendCustomEventDelayedFrames(nameof(CustomUpdateFrames), 10);
}

```

Alternatives to Update() #2 : Animators

If you want to move an object smoothly from a point A to a point B, you could use an animator instead.

But also be careful, too many animators can also affect the performance, in certain big VRChat events with 60+ players you may have noticed heavy FPS drops even after hiding all avatars, that's because many avatars have terribly optimized animators, this also applies to worlds : Optimize your animators, but animators are still better than doing animations in Udon.

Alternatives to Update() #2 : Shaders

VRChat is heavily CPU bound, so sometimes it could be interesting to execute some "code" on the GPU somehow...

Well, shaders are executed on the GPU!

For instance here is a very basic vertex Shader function I wrote that scales a mesh based on the distance of the player, without any Udon !

```

v2f vert(appdata_base v)
{
    v2f o;

    #if UNITY_SINGLE_PASS_STEREO
        //The user is in VR
        //To avoid getting a weird 3D effect in VR, since there are two cameras in VR,

```

we'll basically say that the camera is between both eyes...

```
float3 cameraPositionWS = (unity_StereoWorldSpaceCameraPos[0] +
unity_StereoWorldSpaceCameraPos[1]) / 2;

#else

//The user is on Desktop

float3 cameraPositionWS = _WorldSpaceCameraPos;

#endif

float cameraDistance = distance(mul(unity_ObjectToWorld, float4(0.0,0.0,0.0,1.0)),
cameraPositionWS);

o.position = UnityObjectToClipPos(v.vertex * cameraDistance / _SizeReducer);
o.texcoord = TRANSFORM_TEX(v.texcoord, _MainTex);
return o;
}
```

- If you move further away from the mesh, the mesh scales up.
- If you get closer to it, the mesh gets smaller

The code above could probably be optimized even further, I am not a Shader programmer and I don't know any shader best practice...

If you want to know what amazing things the VRC community created with shader, here are a few worlds I highly recommend to check out:

- VRCLinux by _pi_ : https://vrchat.com/home/world/wrld_8126d9ef-eba5-4d49-9867-9e3c4f0b290d an entire operating system coded inside a shader
- pix2pix by SCR_N : https://vrchat.com/home/world/wrld_014d362b-164b-4b91-b628-a68f5ea0d5b7 an algorithm coded inside a shader that can detect what you're drawing.
- ShaderFes : https://vrchat.com/home/world/wrld_b526c3e0-7c32-4774-931e-5dcbbdd125794 A showcase of some really amazing shader creations.

There are many, many other amazing worlds, and it is impossible for me to list them all.

Disabling scripts if they are not used

Let's say you have a fancy clock in your world, you might not need to update the clock if the player doesn't look at it.

Unity has two events OnBecameVisible/OnBecameInvisible, you could use those events to only enable the script if the renderer is visible by the player.

As of January 2023, OnBecameVisible/OnBecameInvisible does not work in Udon, but

`_onBecameVisible()` and `_onBecameInvisible()` works!

```

public Clock ClockInstance;

///  

//It is important to note that those events only works on renderers!  

//So that script needs to be attached to a MeshRenderer
public void _onBecameVisible()
{
    ClockInstance.enabled = true;
}

public void _onBecameInvisible()
{
    ClockInstance.enabled = false;
}

///  

//Once those events get fully exposed in Udon, you can use the lines  

//bellow
void OnBecameVisible()
{
    ClockInstance.enabled = true;
}

void OnBecameInvisible()
{
    ClockInstance.enabled = false;
}

```

Side note concerning triggers

Player triggers could also be used to disable scripts, but as of January 2023, stations can cause issues (PlayerTriggerEnter/Exit events might not get fired if someone sits on a station, even avatar stations). So make sure your triggers won't cause any issues because of a few stations.

- If you're **inside** a trigger, and you sit on a station that is located **inside** a trigger, `OnTriggerEnter` is called twice.
- If you're **inside** a trigger, and you sit on a station that is located **outside** a trigger, `OnTriggerExit` is not called.
- If you're **outside** a trigger, and you sit on a station that is located **inside** a trigger, `OnTriggerEnter` is not called, but gets called one you exit the station.

In certain cases, the safest way would be to use `OnTriggerStay`, but `OnTriggerStay` is called every fixed update rate (so it is again linked to the screen's refresh rate), so I would not recommend to

use it!