11 - Performance Optimization and Profiling

CS 3160 - Game Programming

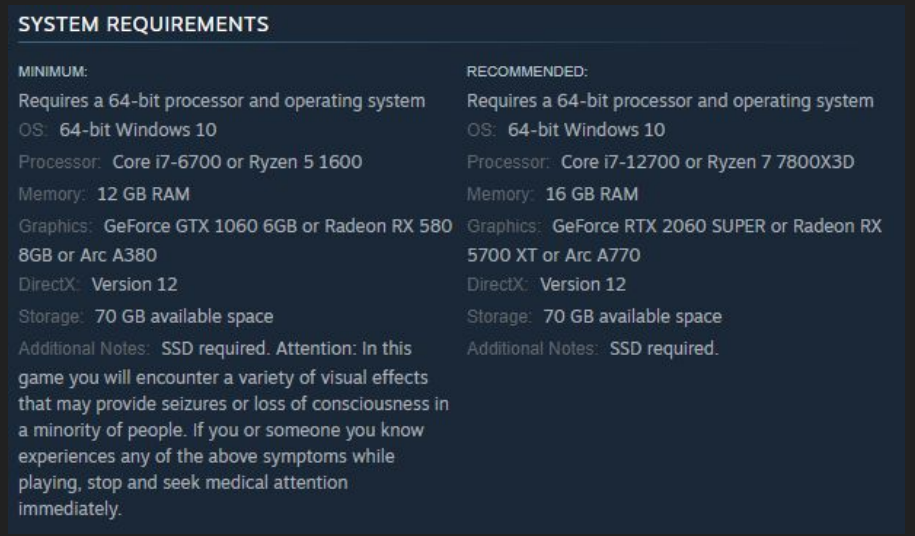
Max Gilson

Performance is Important

- The hardware should not get in the way of the player's experience
  - Low framerate
  - Stuttering gameplay
  - Input delay
  - Long loading times
- The player is also responsible for using "reasonable" hardware
  - You, the programmer, must determine what is "reasonable"
  - i.e. minimum system requirements
- If the performance is poor on standard hardware it's usually the developer's fault

Minimum System Requirements

- Usually minimum system requires is indeed the “minimum”
  - What is required to achieve something barely playable
  - Usually defined by lowest supported graphics API
    - Ex: DirectX 12
  - You should verify that your game is indeed completable at minimum requirements
- The recommended system requirements is very subjective
  - What does recommended mean?
  - 60 FPS? 30 FPS? 1 FPS?
  - Potato graphics settings?



SYSTEM REQUIREMENTS

MINIMUM:	RECOMMENDED:
Requires a 64-bit processor and operating system	Requires a 64-bit processor and operating system
OS: 64-bit Windows 10	OS: 64-bit Windows 10
Processor: Core i7-6700 or Ryzen 5 1600	Processor: Core i7-12700 or Ryzen 7 7800X3D
Memory: 12 GB RAM	Memory: 16 GB RAM
Graphics: GeForce GTX 1060 6GB or Radeon RX 580 8GB or Arc A380	Graphics: GeForce RTX 2060 SUPER or Radeon RX 5700 XT or Arc A770
DirectX: Version 12	DirectX: Version 12
Storage: 70 GB available space	Storage: 70 GB available space
Additional Notes: SSD required. Attention: In this game you will encounter a variety of visual effects that may provide seizures or loss of consciousness in a minority of people. If you or someone you know experiences any of the above symptoms while playing, stop and seek medical attention immediately.	Additional Notes: SSD required.

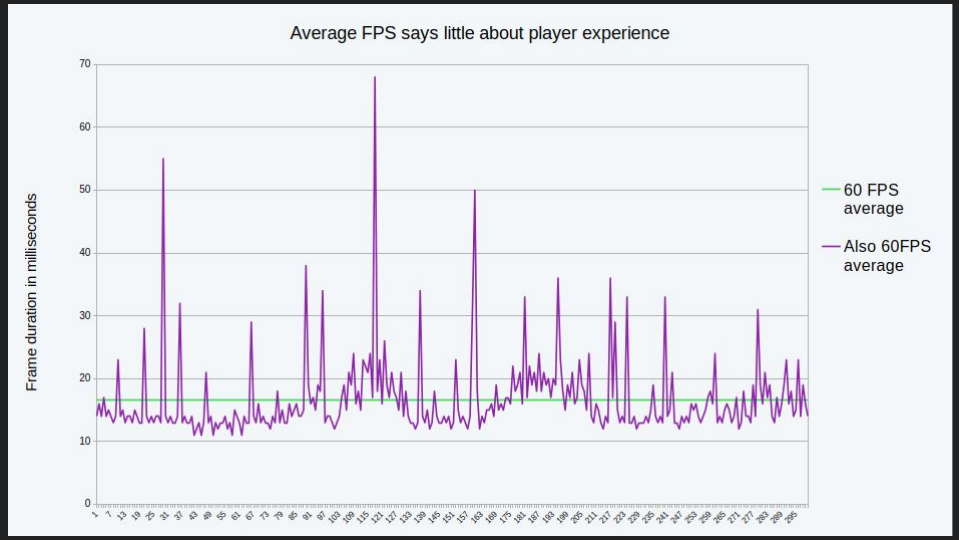
What is Performance?

- Frame Rate = 1 / Frame Time
- Blink of an eye - 150ms
- Your game must deliver a new image to the screen every 33ms or faster
- Many things must happen:
  - CPU calculates game logic and provides data to GPU
  - GPU calculates visuals
  - Memory provides data to CPU
  - HDD/SSD provides assets to memory and/or graphics card
- https://fabiensanglard.net/doomIphone/doomClassicRenderer.php
- Fun fact: in 1993 with a gaming PC worth 2000+ ($4500 today) Doom would run at ~20 FPS

Frame Rate (frames per second)	Frame Time (milliseconds)
30	33.33
60	16.67
120	8.33
240	4.17

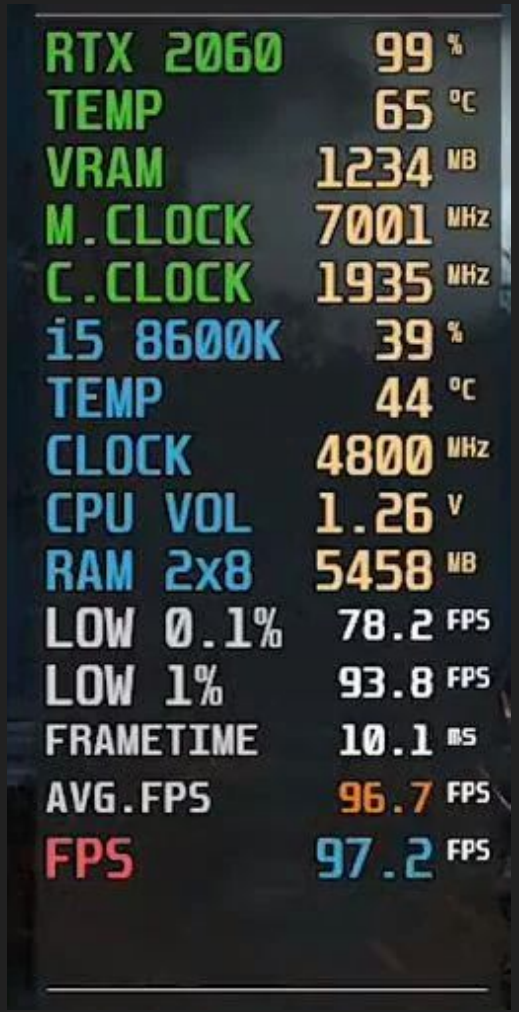
What is Performance? (cont.)

- Performance is frame times and consistency of frame times
  - Average framerate is misleading
- 1% low and 0.1% low framerate are a little better
  - Get 1000 frame rates or frame times
  - Sorted highest to lowest FPS
  - Calculate the average of the lowest 1% (10 frames) and 0.1% (1 frame) values
- A metric for stable framerate is even better
  - https://blog.supertuxkart.net/2024/07/why-average-fps-and-1-low-fps-are.html
- https://youtu.be/-FjdOQmAHpk?si=fCD3jA0e_kX_y3B-&t=33



Performance Tools

- Measuring performance should come in two stages:
  - 1. Measure (profile) the performance of the actual build on actual hardware
  - 2. If there are performance issues, profile the game in the editor if possible
    - Once the performance issue is identified, fix it (if possible)
- MSI Afterburner is a great tool for profiling your build
  - Much better than task manager



Performance Bottlenecks

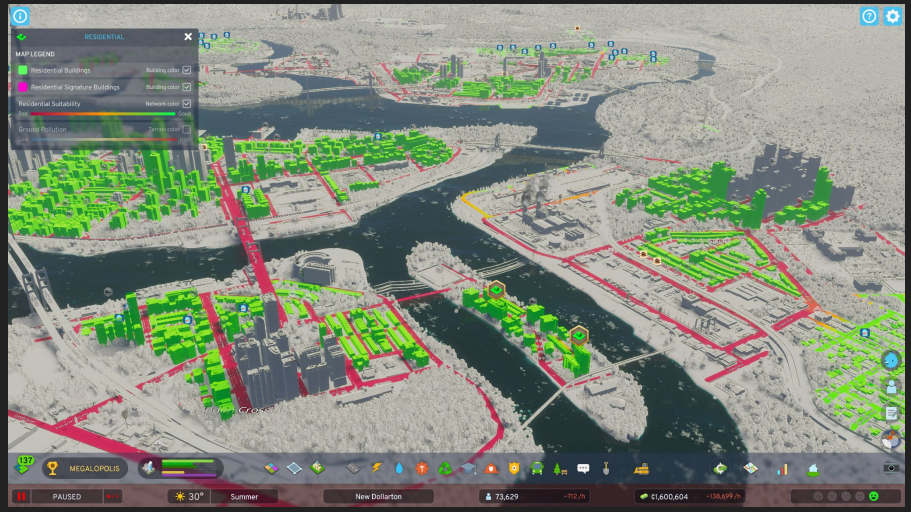
- There are types of performance bottlenecks (one thing that holds everything else back)
  - CPU Bound - performance limited by processor
  - GPU Bound - performance limited by graphics card
  - Memory Bound - performance limited by memory
  - IO Bound - performance limited by IO (usually disk)

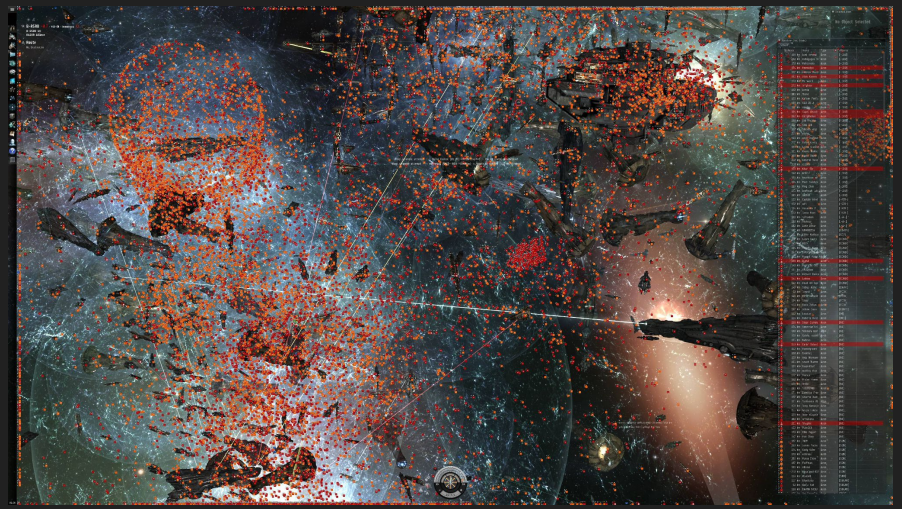
Baseline Single Core Performance + Some Perspective

```
#include <stdio.h>
#include <windows.h>
#define ARRAY_SIZE 8000000
unsigned long long volatile fib[ARRAY_SIZE];

void memoryOperations();
void instructionOperations();

int main() {
    LARGE_INTEGER frequency;
    LARGE_INTEGER start, end;
    double elapsedMilliseconds;
    // Setup timer
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);

    // Do a bunch of memory + math operations
    memoryOperations();
    //instructionOperations();

    // Finish the timer
    QueryPerformanceCounter(&end);

    // Calculate the time
    elapsedMilliseconds = ((double)(end.QuadPart - start.QuadPart) / frequency.QuadPart) * 1000.0f;

    printf("Elapsed time for %d memory + math operations: %.7f ms\n", ARRAY_SIZE, elapsedMilliseconds);

    return 0;
}

void memoryOperations()
{
    fib[0] = 0.0f;
    fib[1] = 1.0f;

    // Calculate the fibonacci sequence numbers
    for (unsigned long long i = 2; i < ARRAY_SIZE; i++)
    {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
}

void instructionOperations()
{
    // Do some basic integer math
    for (unsigned long long i = 2; i < ARRAY_SIZE; i++)
    {
        __asm__ __volatile__ ("addl $1, %%eax":::"%eax");
    }
}
```

CPU Bound Games

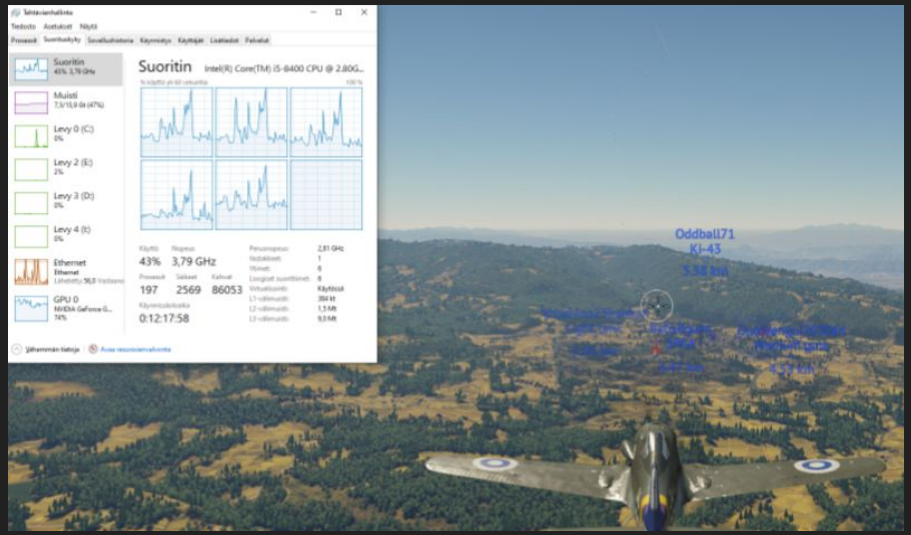
- Games that use the CPU heavily can become CPU bound
  - Simulation games
  - Strategy games
  - Competitive games
- CPU bound means that the limiting factor of performance is the CPU
  - i.e. CPU is at 100% utilization GPU is at low utilization





Improving CPU Bound Games

- What you can do as the programmer:
  - Execute tasks over multiple frames rather than 1
  - Rewrite and simplify or optimize your code
  - Rewrite your game to utilize the CPU cache better
  - Rewrite your game to use multithreading better
  - Reduce the amount of data sent from the CPU to the GPU
    - Hint: use GPU instancing
- What your players can do:
  - Get a better CPU:
    - Faster clock
    - More cores
    - More instructions per cycle
    - More cache
  - Run fewer processes while the game is running

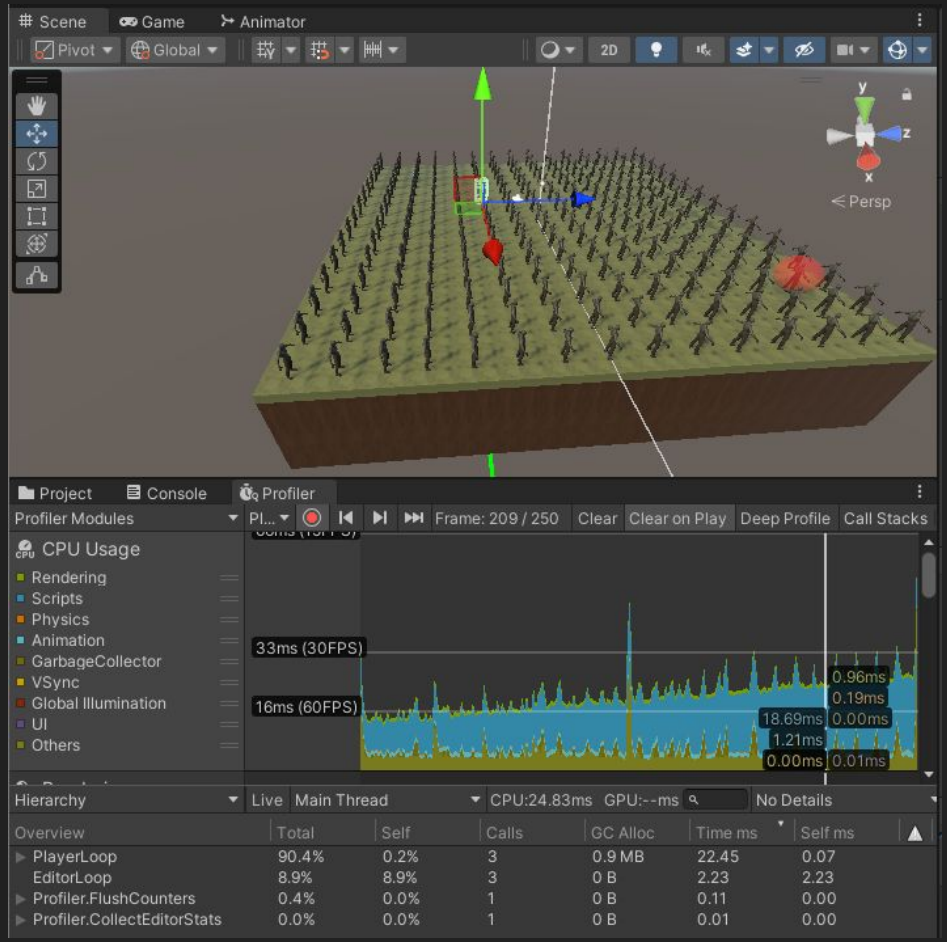


Improving CPU Performance

- Some CPU performance is out of your control:
  - Clock speed
  - Instructions per cycle
  - Core count
  - Cache size
- Some CPU performance IS in your control:
  - Efficient algorithms
  - Efficient multithreading
  - Efficient culling (stop the computation of stuff the player cannot see)
  - Efficient use of CPU cache
  - Minimizing memory allocations (arrays, strings, lists, etc.)
  - Efficient use of IO (loading assets from disk)
  - Instruction level optimizations (SIMD, avoiding division/sin/cos/modulo)

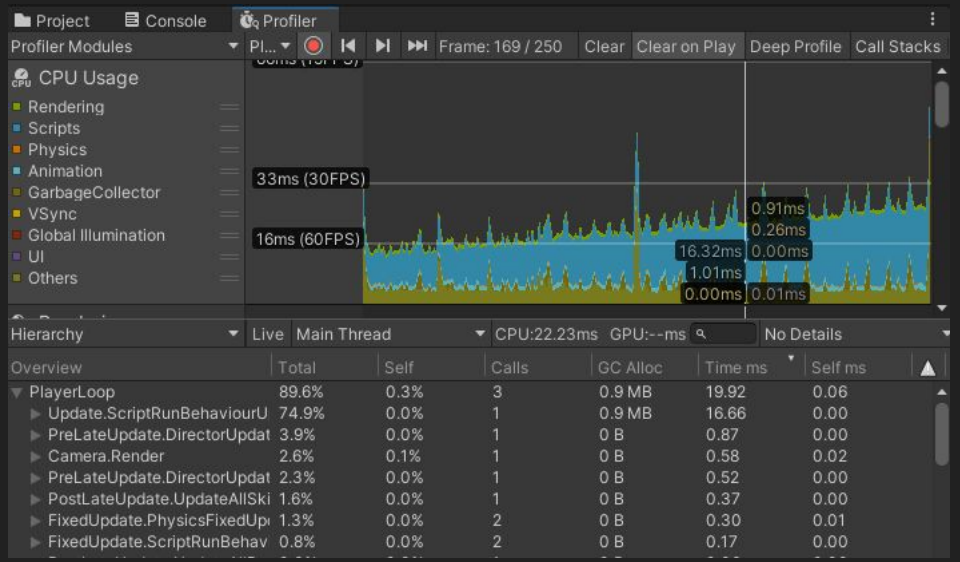
Analyzing CPU Performance

- In Unity, there is a “Profiler” that will allow you to see what your game’s processing is spending the most time on
- PlayerLoop: all of the code running in your game
- EditorLoop: all of the code running in the Unity editor to allow you to edit your game while it is running
    - Note: when make a build of your game, the editor loop will not be running and will not have a performance impact



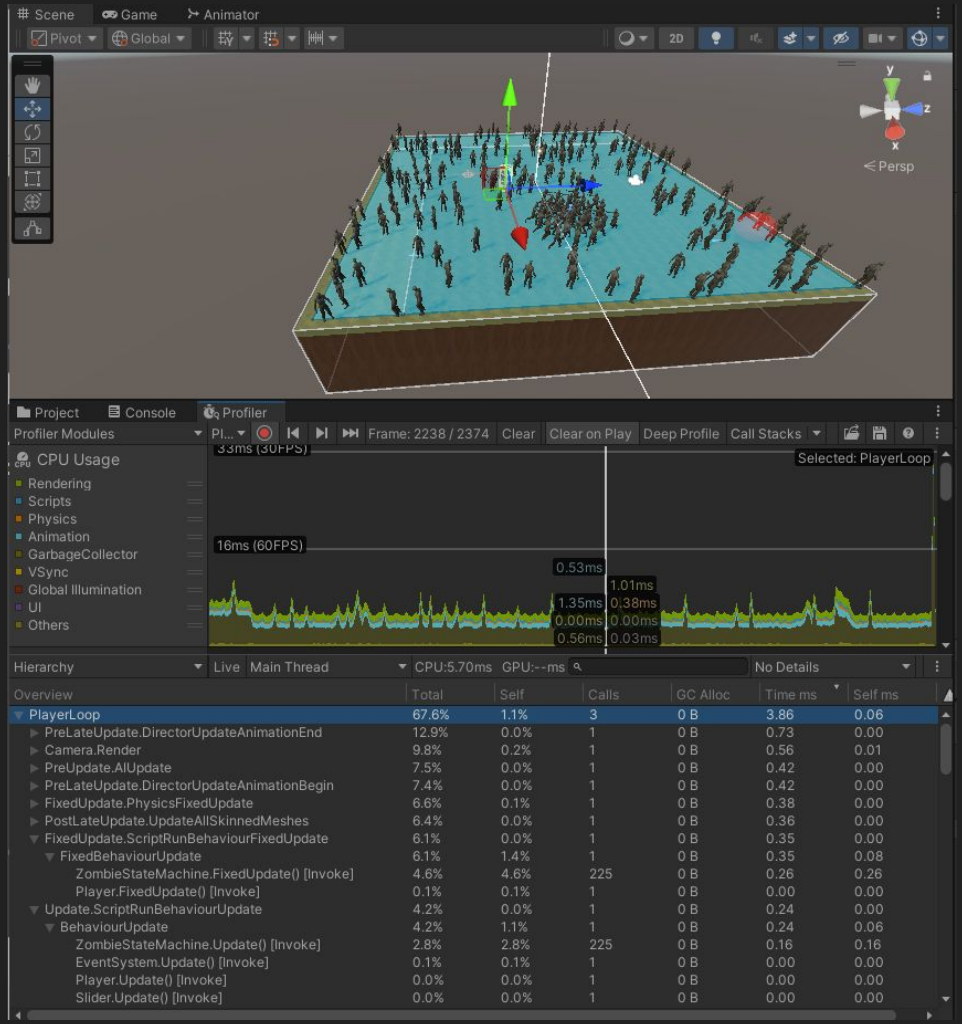
Analyzing CPU Performance (cont.)

- Total - total time spent in a function as a percentage
- Self - total time spent in a function as a percentage, excluding sub-functions
- Calls - number of calls made to this function in this frame
- GC Alloc - heap memory Unity has allocated in the current frame
- Time ms - total time spent on a function, in milliseconds
- Self ms - total time spent in a function in milliseconds, excluding sub-functions



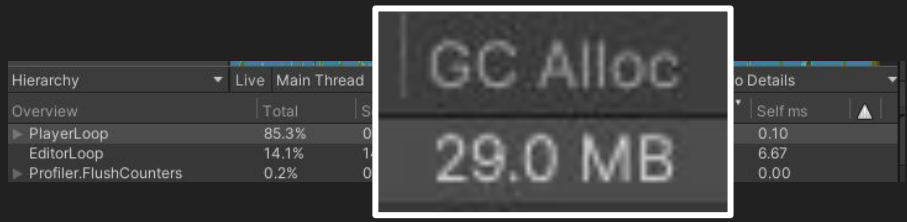
Analyzing CPU Performance (cont.)

- Most of the code that you write will probably be in:
  - Update.ScriptRunBehaviorUpdate
  - FixedUpdate.ScriptRunBehaviourFixedUpdate
- It's very likely that there will other bottlenecks though
- Notice in the profiler:
  - Zombie logic in fixed update and update are taking a combined 0.42 ms out of the total PlayerLoop's 3.86 ms
    - Can you tell how many zombies are in the scene?
  - Animation related methods are taking nearly 4 times longer (1.51 ms) to compute compared to the zombie logic
  - If you were tasked with optimizing the zombie horde, improving the animations would be step 1!



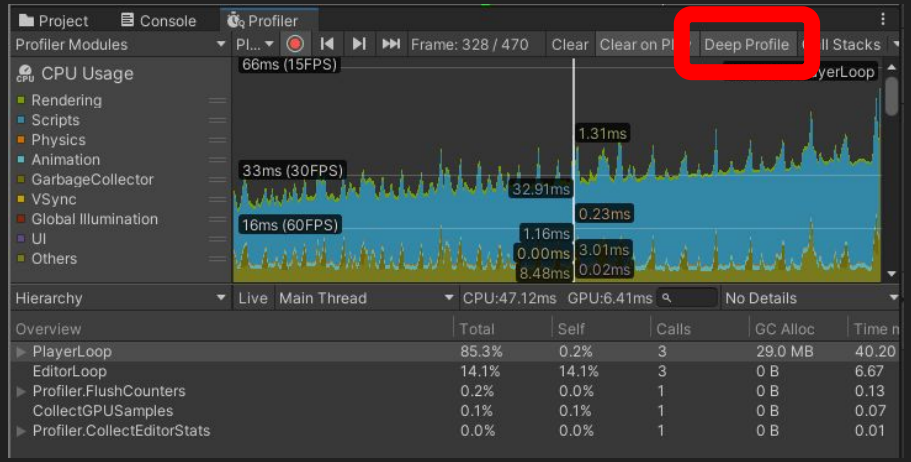
Analyzing CPU Performance (cont.)

- GC Alloc is the “Garbage Collector Allocation” used every single frame
- Unity uses a garbage collector so that you don’t have to manage your own memory
    - If you create a string, list, array, etc. you don’t have to worry about requesting or releasing the memory
- Unity requesting, freeing, and managing this memory uses a lot of resources
- Avoid GC Alloc at all costs
- If you have to, try to put all GC Alloc inside of Start() method of GameObjects



Deep Profiling

- The Profiler allows you to use “Deep Profile” which will have every function call profiled
- Without “Deep Profile”, only a few sub function calls are recorded in the profiler
- NOTE: Using the “Deep Profile” option can make the profiler run very slowly but gives you much more information



Graphics Cards are Basically Another Computer

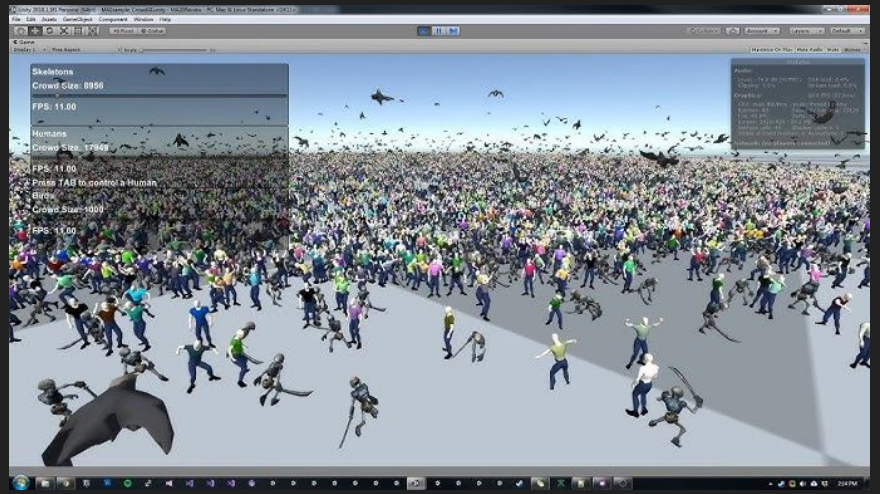
- GPUs are kind of like an additional computer inside of your computer
  - They have their own RAM (called VRAM)
  - They have 1000's of their own cores (called CUDA cores)
  - They have their own instructions and programming languages
- They are specifically designed to process calculations in parallel
- Before GPUs were popular, the CPU had to do all the work (called software rendering)
  - https://fabiensanglard.net/doomIphone/doomClassicRenderer.php
- Now, CPU hands over bundles of data to the GPU over the PCI bus to do the rendering (called hardware rendering)

GPU Bound Games

- Games that use the GPU heavily can become GPU bound
  - Any graphically impressive game
  - Ray tracing
  - Photorealism
- Pushing the graphics card harder also requires more work from the CPU
  - Remember: the CPU has to tell the GPU what to do!

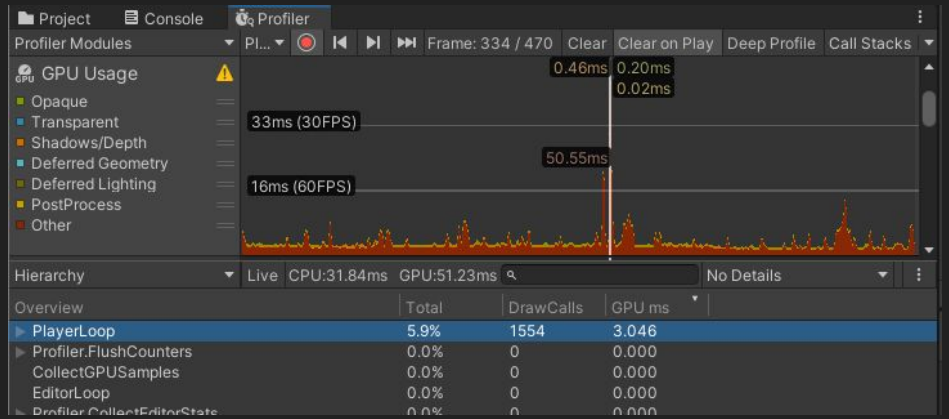
Improving GPU Bound Games

- What you can do as the programmer:
  - Reduce number of shaders
  - Rewrite shaders
  - Reduce texture resolution and filtering
  - Reduce ray tracing or real time lighting
  - Reduce the number of graphical elements on the screen
  - Use GPU instancing
    - Good for grass or crowds
- What your players can do:
  - Get a better GPU:
    - More VRAM
    - More cores
    - More features (i.e. RT)
  - Reduce graphics settings in game (if possible)
    - Use frame generation



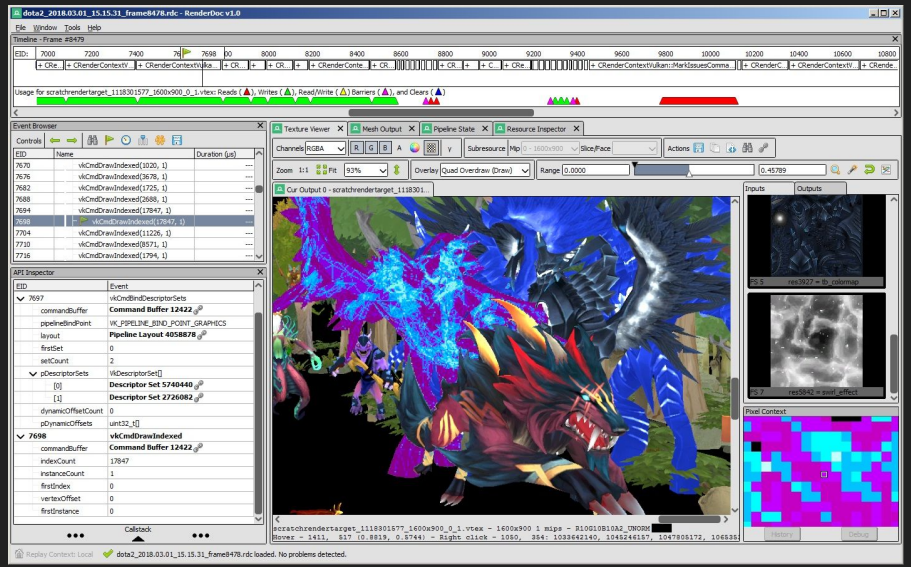
Analyzing GPU Performance

- The Unity Profiler also provides a GPU Usage section
- If you are using advanced graphical techniques, this section may be useful
- If you need to deeply debug the GPU, use RenderDoc



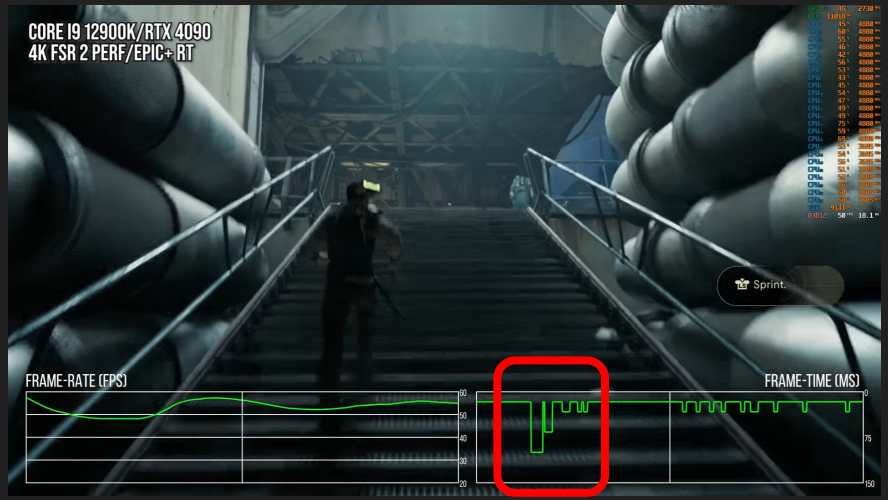
Analyzing GPU Performance (cont.)

- RenderDoc will list out every call made to the GPU
  - A “call” is a command sent from the CPU to the GPU
  - Example: a draw call is a command telling the GPU to display something
- A complex analysis of the GPU requires a deep understanding of the GPU hardware
- Minimize the amount of draw calls necessary
  - GPU instancing can help



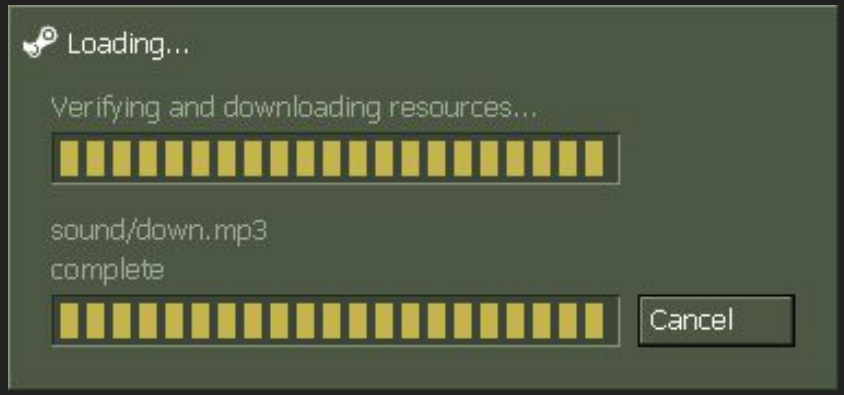
Shader Compilation

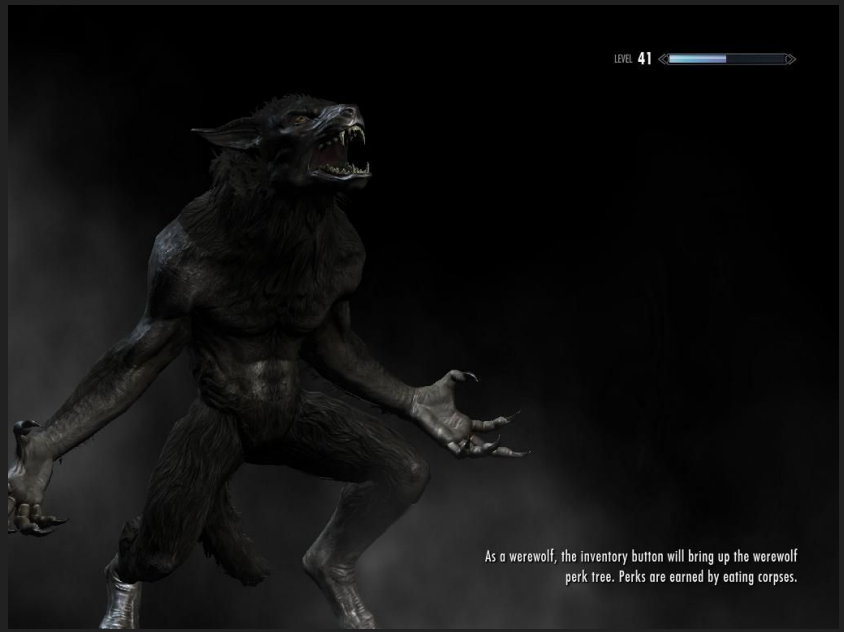
- Recall: a shader is a program that runs on your GPU
  - All GPUs are different
  - All platforms have different GPUs
  - There could be hundreds of variants of a given shader
- Shader compilation is the process compiling these programs to run on a specific GPU
  - This compilation runs on the CPU and is very slow
  - https://youtu.be/uI6eAVvvmg0?si=stiMAdxT_0fS3WFo&t=228
- This is not an issue on consoles because the developer knows what GPU is being used on PS5, Xbox, and Switch
  - The shader compilation is done during the build process and shipped precompiled with the game



IO Performance

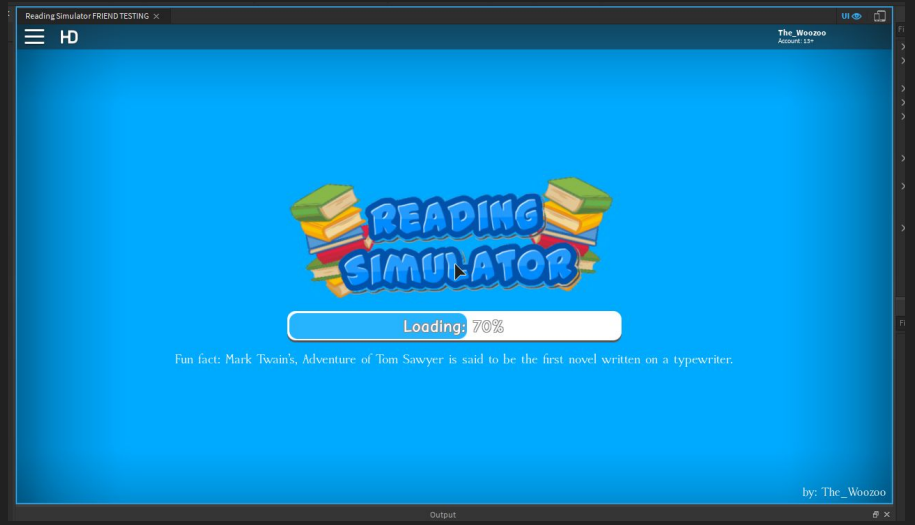
- Your CPU has to constantly work with your storage device (IO) to retrieve assets
  - Models, textures, sounds, scenes, etc.
- This can be accomplished in two ways:
  - Load everything you need early (loading screen)
  - Load small parts as needed while the game is running
    (can result in stuttering)
- Regardless, your IO accesses can have significant performance implications on your game





IO Performance (cont.)

- IO performance may also involve networking
- Some games require downloading content
  - Mobile games require it to keep their install size low
  - Some games require it to allow custom content
- Typically, this downloadable content is handled with a loading screen
  - It is hard to predict when custom content may be accessed, so download all of it



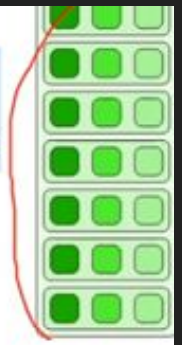
Memory Performance

- Memory has a impact on your performance
- How you choose to organize your code affects how quickly you can access your data
- The negative performance impacts are:
  - Randomly accessing memory locations
  - Poorly clustering of data (bad for cache)
  - Memory leaks
- Avoid chasing pointers (randomly accessing memory)
- Avoid data not being clustered
  - OOP is really bad with both of these
  - Data oriented design (like ECS) is much better
  - Objects are reference types (pointer) that contain more reference types (more pointers!)
- Avoid allocating memory without freeing it
  - Not allocating memory properly is lazy programming
- Use Structure of Arrays rather than Array of Structures (if possible)
  - Unity is almost all OOP so we don’t have much of a choice

Memory Performance

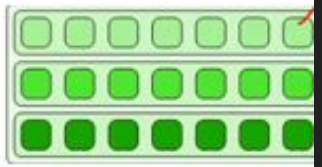
- AoS
  - Most logical layout
  - Difficult for gather reads and scatter writes, difficult for vectorization
  - May lead to better cache utilization if data is accessed randomly

```
struct AOS_Point {
  float x, y, z;
};
struct AOS_Point points[1024];
```



- SoA
  - Your data are organized in a way that if you have a struct with x,y,z, you have all the x all together and the same for the other variables
  - Separate each structure field
  - Keep memory access contiguous if access over structure instances

```
struct SoA_point {
  float x[1024], y[1024], z[]1024;
};
struct SoA_point Points;
```



Don’t Optimize Unless You Have To

- First, set performance targets for your game
  - What is your audience’s hardware?
  - What will your audience be satisfied with?
  - What FPS?
  - What 1% low FPS?
  - Depends entirely on the game and the audience
- Next, profile your game
- Lastly, optimize only if you are below your performance target
  - If you spend 10 years hyper optimizing a game that never gets released… what’s the point?