Dierickx Mathias

Multithreaded game engine design

Graduation work 2018-19

Digital Arts and Entertainment

Howest.be

TABLE OF CONTENTS

ABSTRACT

We will explore multithreaded game engines. We will start off by looking at the features a single threaded game engine has and find out why we should spend the time and effort required to turn these into multithreaded engines. Many different approaches will be researched and discussed, with the goal of creating our own 2D multithreaded game engine. Some of the most common issues that arise during the creation of such an engine will be discussed and multiple solutions will be presented.

The created engine runs a lot faster than its single threaded counterpart, but also raises some questions concerning how much faster it is. Using a benchmarking scene with over 20,000 dynamic and 13,000 static colliders, our multithreaded engine runs over 4 times faster on a quad core CPU with 8 physical threads and over 5 times faster on a hexa-core CPU with 12 physical threads.

INTRODUCTION

Multithreading in software development has been around for well over half a century, first appearing in the late 1950's [1]. We will explore how game engines can benefit from running on multiple cores, looking at the challenges posed by the nature of multithreading and discussing both the advantages and disadvantages of multithreaded game engines.

We will start of by researching engines in general and then move on to specific implementation methods for multithreaded game engines. Using all information gathered in the research phase, we will design our own engine for a 2D environment that scales with the number of available physical CPU cores.

During the research phase, we will ask the following questions; What are the different parts in a typical single threaded game engine and how can we split up these serial tasks, so we can perform them simultaneously on multiple threads. By looking at modern AAA game engines we will gain knowledge on the general approach taken in the industry, so we can base our decisions on methods that have been proven to work. The use of thread pools, worker threads and task managers will be explored in depth and. Along with these concepts, we will look at how cooperative multithreading could benefit us in creating the optimal design.

During the implementation phase, we will look at what scenarios lend themselves best to multithreading and decide what kind of games should use multithreading and what games are better off with using either just a single thread or some very basic version of multithreading.

RESEARCH

1. GENERAL

Before we can start looking at how to multithread a game engine, we must know what a game engine in general is. We will research all the different parts a game engine is responsible for, and determine what parts are eligible for multithreading. In general, we can split up the tasks of a game engine into two distinct categories. Engine tasks and graphics tasks [2]. The following part will discuss some of the main parts a game engine is responsible for. This list is not complete and the way the concepts are implemented depends on the chosen platform and design choices made by the developers.

1.1. ENGINE TASKS

1.1.1. INPUT

Handling input is one of the main tasks of any game engine. Depending on the targeted system, many different input devices must be supported. This could be controllers when targeting consoles or pc, or keyboard and mouse when targeting pc. In a single threaded engine this task usually happens at the start of the game loop, this way all objects will be updated with the latest player input and latency will be as low as possible.

1.1.2. LOGIC

Updating all the objects in the current scene is considered the game logic. This includes running AI, executing player commands, moving objects etc. When working in a component-based system, this comes down to calling the update function on all components of every object. In most single threaded engines, the game logic is handled right after the input handling stage.

1.1.3. PHYSICS

Physics is usually a separate system that controls collisions and gravity in a game scene. This could be implemented using external libraries like PhysX and Havoc, or the engine can implement its own system to handle collisions. When working in a component-based system, the physics calls will be handled by a general physics component. This component is the link between the objects in the game scene and the colliders in the physics scene and has methods to move the objects with physics simulation. This way possible collisions are considered, and objects will not be able to pass through each other.

1.1.4. ANIMATION

Every frame, the animations of all models must be updated as well. In a 2D engine, this is usually done by updating the source rectangle in a sprite sheet. When working in a 3D engine, this gets a bit more complicated. In this case, all skinning matrices of all objects with animations must be updated. This is done by interpolating between the animation's keyframes. This data is then used in in the graphics stages to correctly display all objects on screen.

1.2. GRAPHICS TASKS

1.2.1. DETERMINE VISIBILITY

After all engine tasks, the renderer must start drawing the updated state of the scene to the screen. The first step in this process is determining what objects are visible. This depends on the type of camera being used and how many dimensions we are working in. This step is usually implemented using a graphics API like DirectX or OpenGL.

1.2.2. ISSUE DRAW CALLS

Once we have determined what objects are currently seen by the camera, we issue the draw calls of all visible objects to the graphics API. This will then draw the current scene on the back buffer. After all objects have been drawn to the back buffer, the front and back buffer get swapped. Now the frame is completed, and we can restart the cycle for the next frame.

1.3. GAME LOOP

The game loop is the core of every game engine. It's the piece of code that is executed every frame and handles everything from calling the update and draw functions on a scene to managing the timing of the frames. It usually includes some options like target fps, which sleeps the system for a while if it was running too fast, and an option to toggle waiting for the screen's vertical synchronization.

1.4. ENTITY COMPONENT SYSTEM

This is a feature that is not present in older game engines but is a trend that is popping up in newer game engines. In an entity component-based game engine, an object consists of a group of components instead of making heavy use of polymorphism to create different types of game objects. One of the most common components in such a system is a transformation component, which holds the object's current position and is usually the only compulsory component every game object must have. Another component that is always present in a game engine is a graphical component that is responsible for drawing an object to the screen. This could be a simple 2D texture, or a component that holds a 3D model, animations etc.

For every piece of logic required by a certain game object, a new component is written. In general, a component should only do one thing, but it should do that one thing very well.

1.5. OBJECT POOLS

An object pool is a construct that is often used to reduce the number of memory allocations during the hot code path. Instead of creating new object or components by using the new operator, we use a component pool that has several uninitialized components which we can request and use without having to dynamically allocate memory using the new operator. Once a pool doesn’t have enough objects ready to give out, it creates many new objects at once.

2. WHY MULTI-THREADING

Anyone familiar with the basics of multithreading knows that doing even the most basic multithreaded task requires knowledge of various synchronization features, locking mutexes, avoiding race conditions and so on [3]. So why go through all this trouble?

The power of most modern-day computing systems doesn’t lie in the extreme speed of a single core, but rather in having multiple very fast cores that can execute tasks in parallel. Most laptops have at least 4 logical cores, while consumer desktop computers can go up to 12 or even more cores. The same goes for modern day consoles, with the PS4 and PS4 pro coming in at 4 cores and 8 threads [4] and the Xbox one having 8 cores [5].

But the most telltale sign that all modern devices have multiple CPU cores are smartphones. Qualcomm has mobile chips with up to 8 physical cores [6], which are used in phones from all major android phone makers [7]. Apple has similar offerings with their lineup of A-series chips with their flagship phones using the A12 chip with 6 physical cores [8] [9].

If we would keep working in single threaded environments for our game engines, we would not be using all the available hardware to its fullest potential, resulting in games of poorer visual quality or games with poor performance overall.

In the following sections of this paper we will discuss some approaches of multithreading a game engine environment, starting with basic principles all the way to a specific implementation.

3. THREADING PATTERNS

Now that we have an idea of the tasks game engines must execute, we can start thinking about how to parallelize these serial tasks. Some of the tasks can easily be parallelized, but others pose real problems. The following part will discuss three different threading patterns used in modern multithreaded game engines [10]. Most game engines use all these patterns together to a certain degree.

3.1. PIPELINING WORK

With this pattern we try to find big chunks of code that can be run in parallel. These chunks are usually parts of the code that work on separate data and therefore don’t interfere with each other’s execution.

The first, and most obvious way to introduce multithreading into an engine is by splitting up the engine tasks and render tasks to different threads. While one thread executes all the engine tasks, the other can process the data from the previous frame to be drawn to the screen at the end of this cycle. This seems like a simple thing, but even this introduces some problems already.

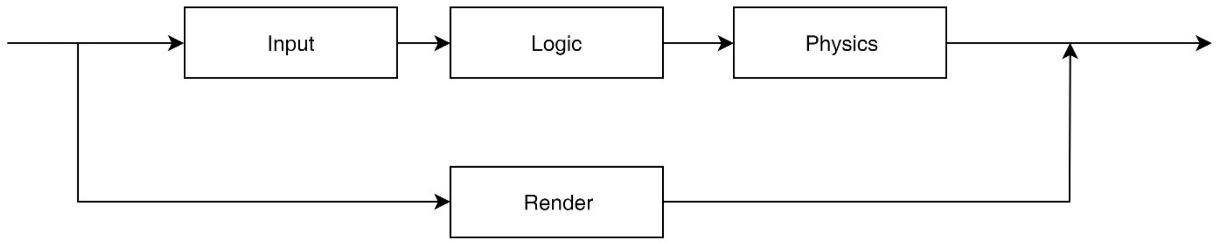


Figure 1: Schematic representation of pipelining work

The first problem that occurs in this pattern is the synchronization of these two working threads. We must make sure that the engine thread doesn't process 10 frames before the render thread can render out the first frame. However, this problem can easily be fixed by using a mutex to synchronize execution.

The second problem this pattern faces is concurrent reads and writes to the same object. The render thread needs to read the data output by the engine thread. This could lead to problems where the engine thread changes data for the current frame before the data of the previous frame has been drawn to the back buffer by the render thread. This problem has a lot of solutions, but the simplest ones will be outlined here.

The first solution to the concurrent data problem is to double buffer the data needed by the render stage. Some examples of data the render state needs are the position and the skinning matrices/ source rectangles of all objects. This can be extended to include data used by certain shaders, like speed of the object used to apply motion blur etc.

Another approach to fixing this problem would be to have 2 separate objects, one engine object and one graphics object. Every frame these objects would be swapped to reuse the memory space. The advantage of this approach is that both threads have all the information they want, at the cost of memory space as it doubles the used memory.

3.2. DEDICATED THREADS

Some tasks can be run on dedicated threads, these threads only execute one kind of specific task. When there are no tasks to perform for these threads, they go to sleep until some more work is submitted for them.

An example of a job that can be offloaded to a dedicated thread is level and asset loading. These are slow operations that must wait on file I/O devices, this would hold back the engine and render tasks if they would just have to wait for all assets to be loaded in from disk. Using a loading thread also enables things like loading screens to be implemented. If loading would not be offloaded to a different thread, the whole game would freeze until the file I/O operations are completed.

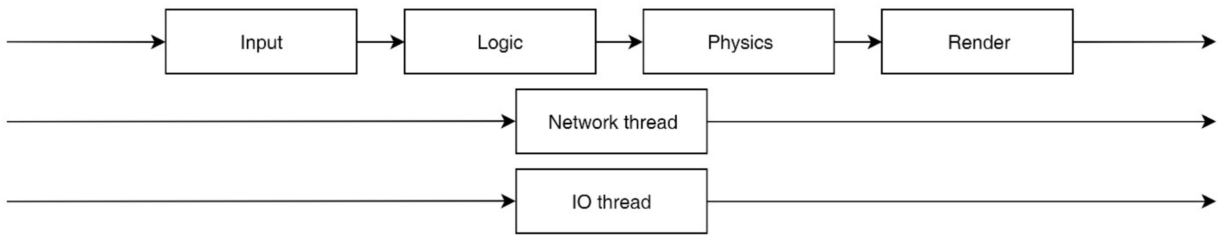


Figure 2: Example of dedicated thread setup

The loading requests would be passed to this thread by a job queue. When a player would walk into a certain area, a task could be scheduled to load in the next part of the level before the player reaches it. This would be submitted to the queue of the loader thread. If the thread would be sleeping because there was no loading to be done, the thread must be notified of this new job. A queue would do the trick for simple engines and simple games, but when more performance is needed a custom solution should be implemented to schedule loading work. Some of the core features this implementation should have would include: canceling tasks, interrupting requests and prioritizing loading requests. This would be useful if a loading request is made when a player nears an unloaded portion of the map, but then quickly turns around making the request obsolete. There would be no reason to load that new part of the level, so we shouldn't waste CPU cycles on loading it.

3.3. TASK SCHEDULERS

The third pattern used in modern multithreaded game engines is the biggest and most important one. Task schedulers are used to distribute work across several worker threads. A worker thread is a thread that constantly checks the task scheduler for new work, if a job is ready it will execute it and repeat the process. This pattern is used in cases where tasks can be cut up in many smaller tasks that can be executed in parallel. We could use a concurrent queue all worker threads could pull jobs from. It doesn't matter which thread executes what job. Once the job queue is empty, we can put the worker threads to sleep.

If we want to be able to put dependencies between tasks, we must implement our own solution instead of using a simple concurrent queue. One way to do this is to create task groups, possible groups would include input tasks, logic tasks, physics tasks etc. Using this system, we can for example make sure no physics tasks are executed before all logic tasks have finished processing.

There are many ways to expand on this solution with task groups. We could introduce priorities so that important tasks get executed first, or per thread task lists where we use an algorithm that would distribute the tasks in a way that they should take about the same amount of time to execute. Threads could be locked to cores to reduce pre-emption by the operating system to ripple through all cores. This last option is not applicable to all devices as the operating system they run on must support locking a software thread to a specific core. On windows you can only tell the OS that you would like a certain software thread to run on a specific core, but the OS scheduler still has the final say.

4. COMMON ISSUES AND SOLUTIONS

When creating a multithreaded game engine design, many factors must be taken into consideration and many questions arise. In the following section we will look at some of these questions and give multiple possible solutions to them. Which solution should be used depends on the specific needs of the engine, a 2D engine will most likely look a lot different than a 3D engine.

4.1. SPLITTING UP LINEAR TASKS

In single threaded engines, there is a main game loop that handles input, updates al game objects, calculates physics and draws the current state of the game to the screen.

When converting this scheme to a multithreaded environment, we should find ways to split up all these stages into smaller tasks that can be run simultaneously on multiple threads. Some stages are not eligible for this scheme due to API's that require to be called from the main thread only [11].

In an entity component system, the update and possible late update calls could be split up by updating components per type instead of updating all Entities directly. Doing this does not only allow us to update components on different threads, it will also increase the chance for cache hits as components of the same type are usually created in the same part of memory. It does however introduce several challenges like dependencies and race conditions. The same reasoning can be used when an object-oriented approach is used in the engine, but instead of updating the different component types, we update the different objects on separate threads.

Multithreading the physics stage depends on the used API, but when a custom physics stage is written work could be distributed on multiple threads by using quad trees (or oct trees in 3D). When checking collision, all leaf nodes can be processed on different threads.

4.2. PROCESSOR AFFINITY

"Processor affinity, or CPU pinning, enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU. This can be viewed as a modification of the native central queue scheduling algorithm in a symmetric multiprocessing operating system. Each item in the queue has a tag indicating its kin processor. At the time of resource allocation, each task is allocated to its kin processor in preference to others." Quoted from Wikipedia. [12]

On windows systems a thread can be locked to a physical core, which can result in less actual processor time as the scheduler is restricted from running that thread on any other core [13]. But it can prevent a ripple effect where one thread being pre-empted by the OS triggers another to be pre-empted because it has been running so long that it offers its spot to the first pre-empted thread. This effect can ripple through all threads, which results in a lot of overhead from the OS and a possible increase in cache misses.

Whether or not processor affinity should be used depends greatly on the underlying operating system and should be tested per specific implementation. In theory it can increase the chance for cache hits when the same tasks are ran on the same thread which is locked to the same physical core, but most modern schedulers might do that job better.

4.3. RACE CONDITIONS

“Race conditions arise in software when an application depends on the sequence or timing of processes or threads for it to operate properly. As with electronics, there are critical race conditions that result in invalid execution and bugs. Critical race conditions often happen when the processes or threads depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive. Failure to obey this rule opens up the possibility of corrupting the shared state.” Quoted from Wikipedia [14]

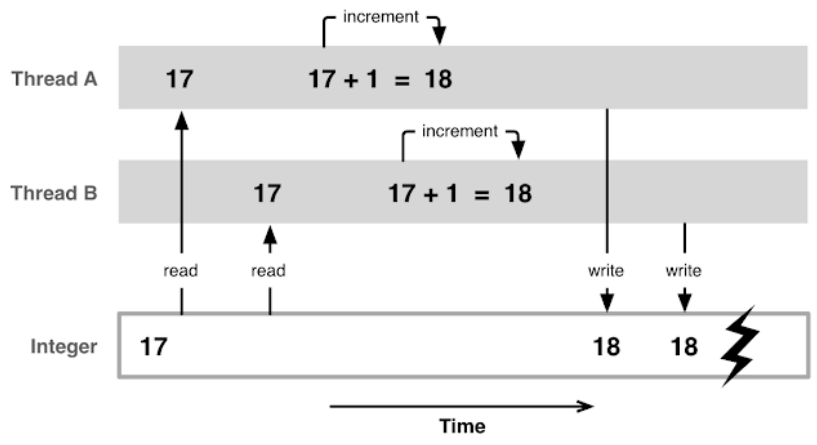


Figure 3: Schematic representation of race conditions [15]

Take two threads that want to increment the same integer. Incrementing an integer is done is three steps, read the current value of the integer, increment it by one and write it to memory. When two threads want to increment the same integer at the same time, it is possible only one of the increments get registered as shown in the figure above.

Any data member of a class that has a public or protected access specifier or getter/setter function should be considered a shared variable, and therefore requires extra caution.

In a game engine race conditions can occur when updating components or objects that rely on each other’s state at the same time. This issue can be solved in many ways. The first is acquiring a lock before any change on shared data. But taking locks is an expensive operation, if possible we want a solution that requires as little locking as possible.

If the shared data are primitive types, atomic variables can be used [16]. These variables have hardware support to atomically store or fetch values from memory. An atomic operation is an operation that can be executed in one hardware instruction and is supported for basic data types like integers, floating points, pointers etc. in C++. Using atomic variables removes the need for locks but storing and fetching instructions are still more expensive than their non-atomic operations.

All shared data can be double buffered, one of the copies will be used for reading and the other copy will be used to write changes to. This way the reads will always be non-blocking and only write operations require a lock to be acquired or must be atomic variables. Using this scheme, every frame the data from the write object must be copied to the read object, so the data should be kept small. This copy can either be done with a full copy using the copy constructor or by calling a copy function that only gets called when the data in the write object was changed.

4.4. HANDLING MULTIPLE FRAMES

Depending on the workload, there can be periods in a frame that all threads must wait for another thread to finish processing before new tasks can be made available. This could happen when there is a component with dependencies on all other components, forcing it to be updated last. While this might be necessary for the game, it is bad for CPU utilization. If the game has many of these idle points for most worker threads, it might be useful to process multiple frames at the same time to fill up these gaps in execution time.

There are different ways to handle multiple frames at the same time. One method is running logic of the current frame while rendering the previous frame. Taking this a step further, we can split up a frame in the logic phase, the physics phase and the rendering phase, where the logic of the current frame is being handled while the physics of the previous frame is being handled while the rendering of the frame before that is being drawn to the screen. If the engine would have more stages, like a late update stage, these could also be handled in a similar way.

The main advantage of this scheme is that all CPU’s will be used to their fullest potential more than without handling multiple frames at the same time. But for every stage, a set of relevant data must be held that represents this state, if this data becomes too big the memory overhead might be too high compared to the gain in processor activity. Another disadvantage arises when there is not a lot of CPU downtime, effectively making this scheme useless. When this happens, memory will be wasted, and input latency will increase for no real benefit in framerate. In general, a latency increase will occur but because the framerate increases with a good implementation, there will be no perceived difference in input latency.

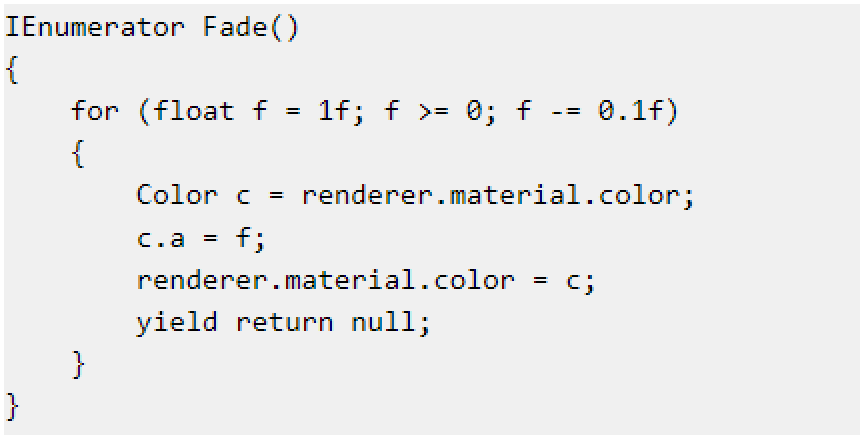
5. COOPERATIVE MULTITASKING

5.1. GENERAL

C++ offers a lot of support for multithreaded applications in the std::thread library [17], however these threads use preemptive multitasking. This means that the OS oversees scheduling the threads, it can choose to sleep a certain thread to make a physical core available for another thread to continue executing. Most OS schedulers are really good at scheduling these threads, but switching threads is expensive [18]. It requires a context switch from user mode to OS mode. Even though the OS will try to minimize the amount of preemption, we would still like to not have to do this. This is where cooperative multitasking comes in, it can be implemented in many ways but the most popular are coroutines and fibers.

Both mechanisms rely on the same base concept. Instead of having the OS switch between threads, we implement a system where functions can cease operation mid execution and pick up where they left off when they last ceased execution [10]. This is done by saving the call stack, stack pointer, all CPU registers and program pointer to memory (yielding) and switching them out with the data of another function (resuming).

In game development, coroutines are mostly used for functions that must execute over the span of multiple frames. Some example uses are fading in or out a texture by gradually changing the alpha layer or general gameplay scripting. All these things can be done using normal functions but would require a lot more code. Using coroutines simplifies the code to a simple for loop and some yield statements.



```
IEnumerator Fade()
{
    for (float f = 1f; f >= 0; f -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return null;
    }
}
```

Figure 4: Example of using co-routines in Unity [26]

The image on the left shows a possible implementation of a co-routine in Unity. This function will update the opacity of a texture over a couple frames, without having to check if you are fading in the update loop of a certain object. All you have to do to make a texture fade out it's opacity is call the function StartCoroutine("Fade"); And the co-routine manager in unity will make sure it gets executed once per frame by default.

As stated before, fibers use the same core principles to achieve cooperative multitasking. Fibers are used for a more general purpose than coroutines, mainly in task based multithreading systems [10]. A fiber provides a task with the stack space used for execution, this fiber is then executed on an actual thread and a scheduler is used to pause a fibers' execution in a cooperative way, without the interference of the operating system. Fibers can also expose one or more functions to the programmer for synchronization, this way a task can be paused until all dependent tasks are completed. Which functionality a fiber has depends completely on the implementation of the general concept. Some even go as far as saying that coroutines and fibers are one and the same thing, which they kind of are under the hood. The only real difference is the use of a scheduler with fibers.

5.2. COOPERATIVE MULTITASKING IN C++

Many languages support some kind of cooperative multitasking in the form of coroutines or fibers. However, C++ does not yet have this capability as of C++11, it is planned to arrive in C++20 [19] but if you want to have cooperative multitasking today you will have to implement this yourself. Platform specific libraries are available for use, like windows fibers or the boost libraries for both fibers and coroutines. All these solutions must be built per system as they require system specific assembly code for saving and restoring the CPU registers and so on. One reason to implement these yourself is control. Using a library gives you little to no control over the features the fibers and coroutines have, which is not that useful for a specific scenario like building a multithreaded game engine. If you don't want to get into assembly code, you can always use the boost context library [20]. This library includes the functionality to do the context switches on user level and it provides an easy tool that builds the assembly code per system, so you don't have to worry about that. Boost also provides their own implementation of fibers and coroutines, built on the context library. These include all functionality for coroutines [21] and gives the end user the ability to create custom schedulers for their fibers [22].

CASE STUDY

1. INTRODUCTION

The following section will outline a theoretical design, based on all previous research. Note that this design is the final design after many iterations and testing.

The engine will build in a 2D environment. We will use the SDL library for window management, loading textures, rendering frames and handling user input. This paper will use SDL version 2.0.4. For some core parts of the game engine, I will take inspiration from my own implementation of a 2D game engine created during Programming4.

2. MULTITHREADED GAME ENGINE DESIGN

The following section will outline the theoretical design that will be used to create a prototype multithreaded 2D game engine. These include all wishes for the engine, but in no way guarantee that they will be included in the prototype. The engine will have an option to build in multithreaded mode or in single threaded mode by toggling a define in the pre-compiled header file.

2.1. CORE ENGINE

The core engine class will encapsulate everything else. It will mainly be responsible for initializing the engine on startup and cleaning up resources when the engine gets shut down. During the initialization phase, all other manager classes get created and initialized.

It will also be the only class that can access the update methods for the game time class. This way, only the core engine can set the delta time value, programmers will only be able to read the value from the time manager. The game loop will be a part of this class.

This class will also hold a set of public member functions that can be called by the game programmers to change some engine settings on the fly. These functions should include: Setting frame rate limit, enable/disable frame rate limits, enable/disable multi frame processing, pausing the game loop and enabling/disabling waiting for vertical synchronization.

2.1.1. GAME LOOP

The engine will feature two distinct game loops. The main game loop will run on the main thread inside the Engine class and will be responsible for handling input, calculating the delta time, rendering the scene and cleaning SDL resources. The secondary game loop will run on all worker threads and will constantly poll the task manager for tasks to complete, until the engine shuts down.

2.1.2. SCENES

The scene class will hold a list containing all Entities and it will hold a component map. This component map holds pointers to all components of all entities in the current scene by component index. The map will be used in the task manager to create tasks for every frame that update every component type separately.

Functions to add and remove Entities from the scene are available to the users, these functions don’t just add or remove the entity to/from the list of entities but is also adds/removes all its components to/from the component map.

For the single threaded version of the engine, functions to update, late update and render all entities in the current scene are also available. When the engine is running in single threaded mode, there is no need to update the component map as it will not be used by anything other than the task manager.

2.1.3. GAME

The engine must be initialized with a game object. The game class will hold all the game's scenes stored in a map with unique names. It offers functions to load a certain scene given its unique name. Changing a scene will first change to a lightweight loading scene, before loading in the actual scene we want to change to. The loading scene can also be changed by the game developers by creating a scene and calling the SetLoadingScene function.

Changing a scene will update the component map in the task manager to use the newly loaded scene.

2.2. ENTITY COMPONENT SYSTEM

The engine will implement an entity component system. A general Entity class will hold a vector of classes derived from the base Component class. This class is responsible for updating all components every frame, it does this by iterating over all components and updating them in a serial way. The components won't get updated in parallel as this will cause concurrent reads and writes.

The entity class will have function to Update, Late Update and Render components. These are the functions that will be called every frame on every entity in the current scene. The base component class will implement these functions as virtual functions, where the Update will be pure virtual and the Late Update and Render functions will have an empty implementation.

A component will hold 2 sets of its own data, these two buffers will get swapped at the start of every frame. One of them will hold read only data of the previous frame, the past object. The other copy will be used to write new data to in the current frame, the future object.

2.3. DOUBLE BUFFERING

As mentioned before, every component will hold two copies of its own data. One of the copy is the past object [23], representing the state the component was in at the end of the previous frame and will be used for any read requests to the component. The other copy is the future object, this object will be used to write changes in the component's data to. At the end of every frame, these two objects are swapped. The future object will become the past object and vice versa.

There are two main benefits to using this scheme for component data. The first advantage is the lockless reading of data by any thread. As the past object is read only, no locks are needed to read from it as data races can never occur. The second advantage is that writes are non-blocking for reads. This means that while a write request to the future object is being processed, reads can go on as if nothing is being changed. These advantages come at the cost of memory, as all data is kept twice in memory.

While all reads are lockless, write requests still need to lock the future object using a mutex. This also means we must know which write is the correct write. Consider the transform component with a function SetPosition, which sets the position in the future object to the passed value. If this function gets called twice in one frame by two different objects, how do we know which write is the correct write?

One solution to this question is to use a priority writing mechanism, where the write with the highest priority will persist over the ones with a lower priority. This would however introduce a lot more overhead as priorities must be passed and compared and it would make programming within the engine more cluttered as the game programmers must keep track of these priorities while writing data to future objects.

This design will not use priority writes, instead it will consider the last performed write the correct write and all previous writes are dismissed. This way the game programmers won't have to worry about setting priorities.

When the logic and rendering stages are decoupled, another form of double buffering will have to be utilized. At the start of drawing a new frame, all positional and texture data from the current state of the logic phase must be copied over to the rendering stage. If we wouldn't do this, it would be possible that a part of the frame is rendered with the data from logic phase x, while another part is rendered using the data from logic phase x + 1.

2.4. PHYSICS SYSTEM

The engine will feature a basic physics system with over-lappers and colliders. Every object that wishes to receive information about other over-lappers and colliders in the current scene will have to register with the scene's physics scene using a collider component.

The physics scene itself will keep track of all registered objects using a quad tree [24] structure. Every frame the physics scene will update the structure with the new positions of all objects and check for collisions between all objects within the same node of the quad tree.

Objects that reside in two or more nodes of the quad tree at the same time will require special attention, collision checks will be done between the edge case object and all other objects of every node it coincides with. It is obvious that we want to keep the amount of edge cases to a minimum as checking these will be much more expensive than checking objects that only reside in one node at a time.

2.5. RENDERING SYSTEM

As the focus of this paper lies in the multithreading part of the game engine, the rendering system will not be that vast. SDL will be used for window management and basic texture drawing functionality. All SDL functions should only be called from the main thread that created the window, this means we can only call the draw functions from the main thread. The main thread will be treated as a dedicated render thread, that will call the render function on all render components in the current scene. It will also be responsible for calling all other SDL functionality like handling input, cleaning up SDL textures etc.

The render component has functionality to set the depth of the texture, this will make the render thread render all textures with a lower depth first. Depth 0 is defined as furthest away from the camera and the max integer is the closest to the camera.

2.6. SOUND SYSTEM

A sound manager singleton will be used that has functions to load sounds and play, pause and stop a certain loaded sound.

2.7. TASK SYSTEM

The task manager will be the core of the game engine, responsible for passing the tasks to the worker threads.

2.7.1. TASK

Before we can define the task system's features, we first must define what a task is. This design offers support for three different kinds of tasks.

The first task type is an engine task; these tasks are reserved for the internal engine code and will only be used by the engine programmers. These tasks are a combination of a component ID, a stage ID and a range defined by two integers. Depending on the stage, the worker threads will execute different functions on the components corresponding to the given component ID at the indexes defined by the range. For example, a task with the transform component ID, stage Update and range 0 – 200 will call the update function on the first 200 transform components in the current scene's transform component vector.

The second task type is a command task. A command task consists of a pointer to a command object. A command object is an abstract type that has an execute function and is possibly bound to an entity. When a command task is pushed to the task manager, it gets added in the update stage, so all commands get executed during the update stage. When a command tasks is received by a worker thread, it will simply call the execute function on the given command object.

The last task type is the load task. This is simply an indication that the Game requires a thread to start changing the current scene. This is done by first setting a simple lightweight loading screen that is always loaded in memory as the current scene. Once the loading scene is active, the scene that was active before the loading scene gets unloaded from memory. All the entities and components get returned to their corresponding object pools. After unloading the previous scene, the initialize of the desired scene gets called. After initialization is done, that scene gets set as the active scene and the load task is completed. Swapping scenes must be done by the main engine loop as it needs to happen between frames.

2.7.2. TASK STAGES

Task stages define which function must be called by the worker threads on the components with given component ID. The design has the following 5 engine stages and a stage to define command tasks.

- Update Stage: During this stage all components that have the update stage enabled will have their update function called.
- Physics Initialization Stage: During this stage, all collider components are inserted into the quad tree.
- Physics Process Stage: This stage processes all nodes of the quad tree that are not empty.
- Late Update Stage: During this stage all components that have the late update stage enabled will have their late update function called.

- Swap Buffer Stage: During this stage all components that have the Swap buffer stage enabled will have their swap buffer function called.

Task stages are also used to define command and loading tasks. These are tasks with their stage set to Command or Loading and will be handled by the worker threads as described in the previous section.

2.7.3. SYNCHRONIZATION

In order to synchronize the execution of all these stages, a vector of integers will be used. Each counter corresponds with a component type. At the start of each stage the value of the counters is set to the number of components that should get updated during that stage for each component type. When a task is completed, the counter corresponding with the component type of the task will be reduced by the number of components it handled in that task. Once all counters hit zero, the current stage is completed, and the next stage can begin executing.

2.7.4. TASK MANAGER OVERVIEW

Now that we know the base components of the task manager, we can define how the task manager will behave.

At the start of a frame, all tasks for the remainder of the frame will be added to the task queue. This includes all tasks for every stage of the frame, except for the tasks of the Physics Process Stage. These tasks depend on the result of the Physics Initialization Stage, so they will be created after the Physics Initialization Stage has completed.

Worker threads will constantly try to get a new task from the task manager, but there will be moments there are no tasks available to process. When this happens, the worker threads will wait for the task condition variable to be notified. This makes the thread sleep until another process notifies that condition variable.

When a new task is pushed to the task queue, the task condition variable [25] gets notified to signal a waiting worker thread that there is a new task available. When all threads are already working, this call has no effect. By notifying this condition variable we prevent the whole system from idling during the creation of the frame tasks.

There are two points where the worker threads must synchronize with the main render thread, these are right before and right after the buffer swap stage. The sync point before the buffer swap stage is needed to make sure that the main thread has completed rendering the current frame before swapping out buffers as the rendering system uses data from the past object of the render and transform components. When the worker threads are swapping buffers, the main thread will process input and update the time manager. This means we must sync with the worker threads again after updating the buffers, so the rendering of the new frame starts with the correct data.

2.8. MANAGING TIME

As mentioned in the previous part, the main thread will update the game time while the worker threads are swapping the component buffers.

The calculate delta time function will be called, this function makes the main thread sleep if the framerate is locked and the engine is running too fast. Only the Engine class in which the main thread runs has access to this function, other classes only have access to the functions to get the delta time and the frame rate. This way the time value can never be changed at the wrong time.

2.9. OVERVIEW

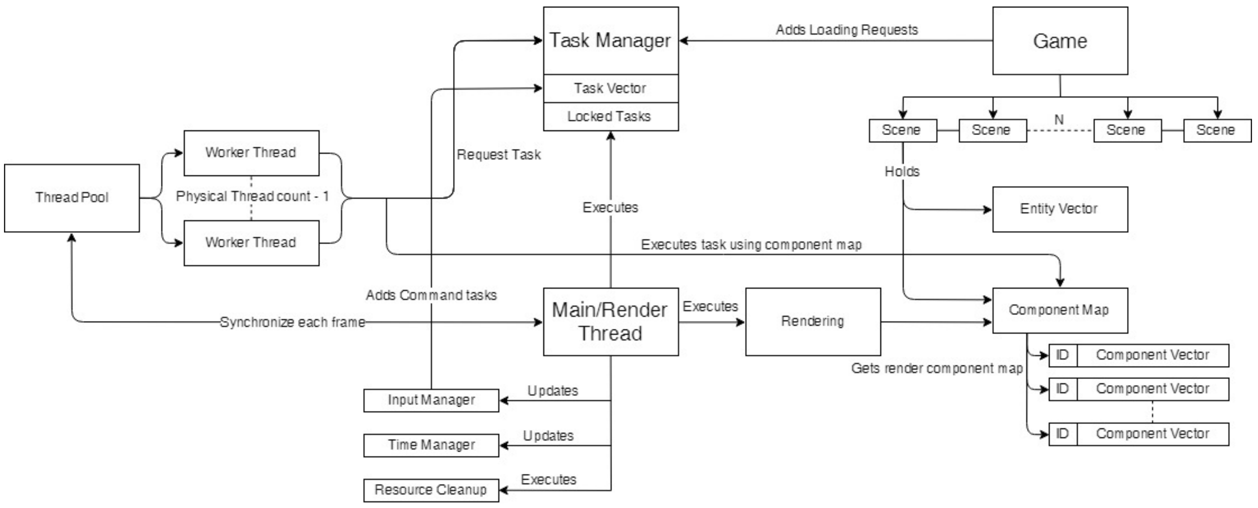


Figure 5: Visual representation of engine flow

3. BENCHMARKS

Now that the we have an engine, we will test the engine's capabilities. This will be done by running a benchmarking scene with 20,736 animated Pacman ghosts and 13,158 area objects that have a green texture when no ghosts are on top of it, and a red texture when there are ghosts overlapping with the object. The ghosts are dynamic colliders and will block all other objects. The area objects are static overlapping colliders, these will register all other blocking and overlapping colliders but will not block movement themselves. The scene will be run on two different machines, one with an AMD CPU and another with an Intel CPU.

Intel machine specifications:

- Intel core i7 7600HQ Quad core, Eight thread CPU
- Nvidia Quadro M2000M GPU
- 16GB RAM
- Windows 10 Education

AMD machine specifications

- AMD Ryzen 5 2600x Hexa core, Twelve thread CPU
- Nvidia GeForce GTX 1080 GPU
- 16GB RAM
- Windows 10 Education

3.1. RESULTS

3.1.1. INTEL RESULTS

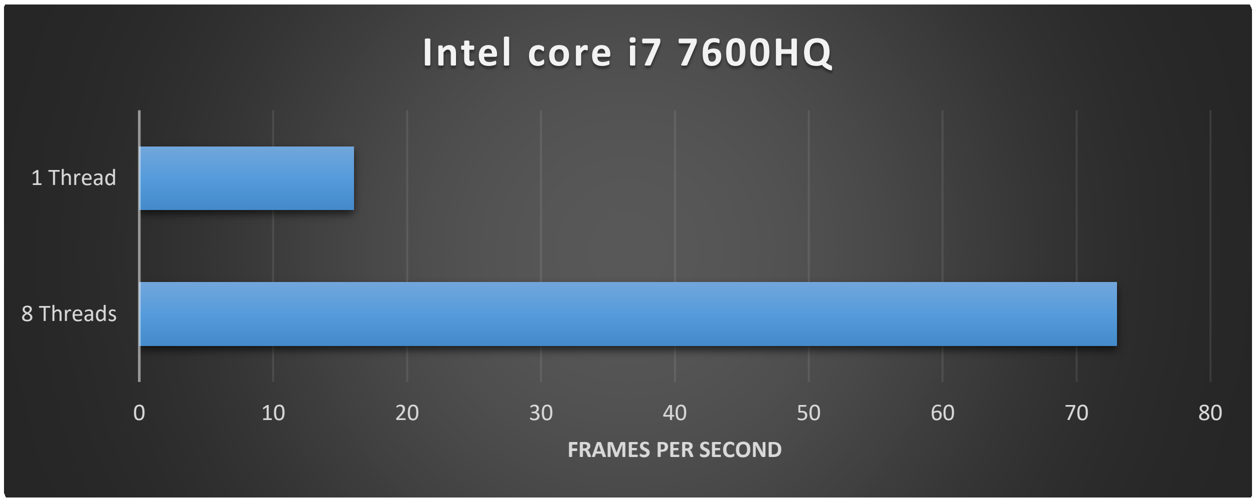


Figure 6: Benchmarking results on Intel CPU

The intel machine has a quad core CPU with 8 physical threads. Running the test scene in single core mode yields an average frames per second of 16. When enabling multithreaded mode, an average of 73 frames per second is reached. This results in an improvement of about 4.5 times the single threaded performance.

3.1.2. AMD RESULTS

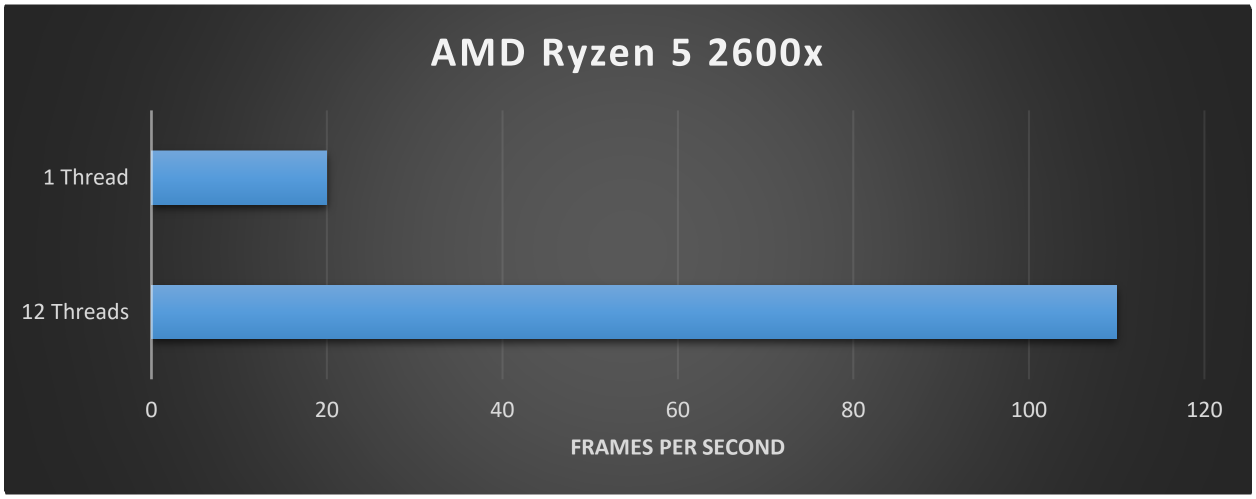


Figure 7: Benchmarking results on AMD CPU

The AMD machine has a hexa-core CPU with 12 physical threads. Running the test scene in single core mode yields an average frames per second of 20. When enabling multithreaded mode, an average of 110 frames per second is reached. This results in an improvement of about 5.5 times the single threaded performance.

3.2. REMARKS

These results show that the engine does perform better in multithreaded mode than they do in single threaded mode, but they also raise some questions and offer some answers.

3.2.1. PERFORMANCE

The first is the performance gain versus the number of physical cores. Why is it that a processor with 8 physical cores only yields an improvement of 4.5 times, instead of the theoretical maximum of 8?

A multithreaded engine will never be able to reach its theoretical improvement factor, due to the nature of multithreaded applications. There will always be some overhead due to synchronization. In the case of this engine, there is also overhead for creating the tasks to be executed every frame. This engine also works better the more logic heavy a scene is. Scenes with little to no logic computations, will not gain much from the multithreading as there will most likely be more overhead for creating the tasks than actual computations to be done. But scenes with many objects that require a lot of logic computations (e.g. Pathfinding, heavy calculations…) may have higher improvement rates than the benchmarking scene.

3.2.2. SCALABILITY

The second question that arises is the one of scalability. When comparing the improvements gained from the 8 thread CPU versus the 12 thread CPU, the engine does improve but has a decreasing improvement rate compared to the number of available logical cores.

The biggest factor in this case is the implementation of the task manager. Currently, it is a big bottleneck in the system that only gets worse when adding more and more cores to the worker thread pool. As there is only one task queue, all threads need to acquire the same lock before they can receive their next task. This can lead to lots of waiting overhead when the tasks take little to no time to complete.

A possible solution that would fix this problem would be to have a dedicated task manager thread that will fill a task queue on a per thread basis, this way the threads don't have to acquire a lock to get a value from one single queue but will receive tasks while they are processing their current task.

It also has to be noted that both scenarios were tested on different platforms and therefore might not be directly comparable as the Intel platform is a lot more mature than the AMD ryzen family of processors.

3.2.3. PROCESSOR AFFINITY

During the research phase, the question arose if processor affinity should or shouldn't be used. Testing this resulted in a quite clear answer for applications running on Windows 10. On the Intel platform, setting the processor affinity had little to no effect on the framerate, but enabling processor affinity on the AMD system resulted in periodic frame freezes.

Due to there being either no improvement or a big disadvantage for using processor affinity, we decided to disable it for this engine and let the OS schedule our threads for us.

3.2.4. CACHE COHERENCE

After implementing scene changes, where the current scene returns all its entities and components to the correct Objectpool, big frame drops could be seen when reloading the same scene for the second time around. This is due to the components in the scenes component map no longer being guaranteed to be continuous in memory. Solving this issue would require returning the components in the opposite order as they were received so that they remain in continuous memory.

CONCLUSION

We started by researching what a game engine does in general. We took a brief look at some of the components a general engine could have. The main ones are the logic tasks; handling input, logic tasks, physics tasks, updating animations and the rendering tasks. Most modern game engines implement an entity component system, but an inheritance-based model can also be used. We discussed the general layout of a classic game loop and looked at what object pools can be used for.

Secondly, we discussed the need for multithreading game engines in general. We should multithread game engines because all modern hardware, ranging from desktop computers to mobile phones, have at least and often more than two physical cores. Not utilizing this would leave a lot of potential performance unused.

A lot of research was done on many different techniques that can be utilized to effectively multithread a game engine. We looked at how we can parallelize the serial tasks from a single threaded game engine, we looked at different patterns used by many different AAA game engines like dedicated threads, task managers, worker pools and more. Some engines utilize cooperative multithreading to be able to create tasks and wait for then to complete.

Finally, we created our own design based on the research and implemented this design using SDL and C++. This resulted in a 2D game engine that can handle many more entities per scene than a single threaded variant but left us asking some big questions.

The acquired result showed us a good improvement over single threaded performance but showed room for improvement. A big bottleneck in the task manager made the engine scale rather poorly with more threads. Overall, this project gave a lot of insight into the possible structures for game engine and uncovered some issues we did not find during our research.

REFERENCES

[1] M. Smotherman, "History of multithreading," 1 4 2005. [Online]. Available: https://people.cs.clemson.edu/~mark/multithreading.html. [Accessed 4 11 2018].

[2] "How Ubisoft Develops Games for Multicore," 23 10 2014. [Online]. Available: https://www.youtube.com/watch?v=X1T3IQ4N-3g. [Accessed 27 9 2018].

[3] "Multithreading in C++," 19 10 2016. [Online]. Available: https://www.geeksforgeeks.org/multithreading-in-cpp/. [Accessed 19 10 2018].

[4] "Playstation Specifications," 19 10 2018. [Online]. Available: https://en.wikipedia.org/wiki/PlayStation_4_technical_specifications#Versions. [Accessed 19 10 2018].

[5] "Xbox One Hardware Specs," 4 11 2016. [Online]. Available: https://www.ign.com/wikis/xbox-one/Xbox_One_Hardware_Specs. [Accessed 19 10 2018].

[6] Qualcomm, "qualcomm," 1 1 2017. [Online]. Available: https://www.qualcomm.com/products/snapdragon-845-mobile-platform. [Accessed 4 11 2018].

[7] GSMArena, "GSMArena," Samsung, 9 8 2018. [Online]. Available: https://www.gsmarena.com/samsung_galaxy_note9-9163.php. [Accessed 4 11 2018].

[8] Wikipedia, "Apple A12," 30 10 2018. [Online]. Available: https://en.wikipedia.org/wiki/Apple_A12. [Accessed 4 11 2018].

[9] P. Holland, "iPhone XS specs vs. XS Max, XR, X: What's new and different," Cnet, 28 10 2018. [Online]. Available: https://www.cnet.com/news/iphone-xs-specs-max-xr-compared-x-what-is-new-and-different-features-2018-available-now/. [Accessed 4 11 2018].

[10] "Multithreading the Entire Destiny engine," 2015. [Online]. Available: https://www.gdcvault.com/play/1022164/Multithreading-the-Entire-Destiny. [Accessed 21 10 2018].

[11] "SDL threading," [Online]. Available: https://wiki.libsdl.org/CategoryThread. [Accessed 30 10 2018].

[12] "Processor affinity," 1 4 2018. [Online]. Available: https://en.wikipedia.org/wiki/Processor_affinity. [Accessed 30 10 2018].

[13] Microsoft, "SetThreadAffinityMask," 19 10 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-setthreadaffinitymask. [Accessed 30 10 2018].

[14] Wikipedia, "Race conditions," 29 10 2018. [Online]. Available: https://en.wikipedia.org/wiki/Race_condition. [Accessed 30 10 2018].

[15] opensourceforgeeks, "Race Condition, Synchronization, atomic operations and Volatile keyword.," opensourceforgeeks, 14 1 2014. [Online]. Available: http://opensourceforgeeks.blogspot.com/2014/01/race-condition-synchronization-atomic.html. [Accessed 4 11 2018].

[16] C. Reference, "std::atomic," 18 6 2018. [Online]. Available: https://en.cppreference.com/w/cpp/atomic/atomic. [Accessed 30 10 2018].

[17] "std::thread Libary reference," 21 10 2018. [Online]. Available: https://en.cppreference.com/w/cpp/thread/thread. [Accessed 19 10 2018].

[18] "Cost of context switching," 28 12 2012. [Online]. Available: https://blogs.msdn.microsoft.com/andrewarnottms/2012/12/28/the-cost-of-context-switches/. [Accessed 21 10 2018].

[19] "C++ timeline," 17 4 2017. [Online]. Available: https://isocpp.org/files/img/wg21-timeline-2017-03.png. [Accessed 21 10 2018].

[20] "Boost Context Library," 1 8 2018. [Online]. Available: https://www.boost.org/doc/libs/1_68_0/libs/context/doc/html/index.html. [Accessed 15 10 2018].

[21] "Boost Coroutines," 1 8 2018. [Online]. Available: https://www.boost.org/doc/libs/1_68_0/libs/coroutine/doc/html/index.html. [Accessed 15 10 2018].

[22] "Boost Fiber Customization," 1 8 2018. [Online]. Available: https://www.boost.org/doc/libs/1_68_0/libs/fiber/doc/html/fiber/custom.html#custom. [Accessed 16 10 2018].

[23] F. Majerech, "A Concurrent Component-Based Entity Architexture For Game Development," P. J. Safarik University, 2015.

[24] "Quad trees in game programming," 3 09 2012. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374. [Accessed 21 10 2018].

[25] "Condition variable," 28 6 2018. [Online]. Available: https://en.cppreference.com/w/cpp/thread/condition_variable. [Accessed 10 30 2018].

[26] U. Documentation, "Coroutines," Unity, 10 10 2018. [Online]. Available: https://docs.unity3d.com/Manual/Coroutines.html. [Accessed 4 11 2018].

APPENDICES

PREPARATION INTERVIEW LARIAN STUDIOS (DUTCH)

1. Memory management
    a. Custom allocator?
        i. Allocator per type of global
        ii. Pooling of allocators voor alles
    b. Wordt er rekening gehouden met cache coherence? Hoeveel impact heeft dit op performance? -> Naughty dog Engine houdt hier geen rekening mee, in hun geval was het efficienter van alle cores zoveel mogelijk te laten werken ipv bepaalde objecten te locken aan een logical core.
    c. Race conditions: Welke systemen zijn er in plaats om race conditions tegen te gaan dmv zo weinig mogelijk locks te gebruiken?
2. Task system?
    a. Als er een task system gebruikt wordt:
        i. Task manager: Welke features heeft de task manager?
            1. Priorities, centralized/per task list…
        ii. Worker threads
            1. Hoeveel threads worden er gemaakt? (max aantal logical cores, LC – 1, meer dan het aantal logical cores?...)
            2. Hoe worden tasks doorgegeven aan een thread?
                a. Fibers, function pointer...
            3. Indien fibers, hoe zijn deze implemented
                a. Platform specific (Windows fibers, PS4 fiber library)
                b. Custom fiber system?
3. Dedicated threads
    a. Zijn er dedicated threads voor specifieke tasks
        i. File IO, networking requests
4. Object dependencies
    a. Hoe kunnen objecten garanderen dat een ander object eerder geupdate wordt in de frame; bijvoorbeeld een character dat een zwaard heeft, voor het zwaard geupdate kan worden met de character geupdate zijn
        i. Task groups in task based system?
            1. Dynamic defined groups/predefined?
        ii. Krijgt character ownership over het zwaard en staat de character in om het zwaard te updaten na het updaten van zichzelf -> hierachy tree
5. Frame interleaving/ render decoupling
    a. Worden frames frame by frame behandeld? Of wordt de vorige frame geupdate terwijl de volgende frame logic uigevoerd wordt?
    b. Of logic gaat continu en renderer double buffert nodige data op een set interval (16ms/33ms) -> meer nauwkeurigheid van input maar niet elke state wordt gerendert.
6. Testing thread safety
    a. Hoe worden multithreaded systemen gevalideerd; unit testing, isolated test programmas…

PAPERS

See the papers folder for all papers read during research and used as reference for this project.