Game performance sucks, and one attempt at fixing it

- Master Thesis Project Report -

pt101f19

Aalborg University
Software

Copyright © Aalborg University 2016

Department of Computer Science
Selma Lagerløfs Vej 300
9220 Aalborg Øst
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Game performance sucks, and one attempt at fixing it

Theme:
Programming Technology

Project Period:
Spring 2019

Project Group:
pt101f19

Participant(s):
Carsten Schroll Ibsen
Frederik Palmelund Voldby

Supervisor(s):
Lone Leth Thomsen

Copies: 1

Page Numbers: 84

Date of Completion:
06/06/2019

Abstract:

In this thesis we present the `aggregator`, a memory isolation model inspired by software transactional memory, and implement it into a game engine, to test if it is a viable alternative to lock-based parallelism in game engines. We design and implement three game engines: a sequential, parallel with locks, and parallel with the aggregator, and test and compare them over three tests to determine if the aggregator game engine can outperform traditional parallelism with locks. The implementation is done in C++ with SFML as graphics library. The tests involve single-core overhead, work load, and handling of critical regions. The tests are performed on four different test systems, to gain better understanding of how the game engines perform across different hardware configurations. Test results show that the aggregator game engine has potential as it was 4.6 times faster then sequential but still slower then parallel game engine that was 7 times faster. There are potential for the aggregator to perform better with changes to the implementation.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Summary

Formålet med dette projekt er at undersøge en ide, som vi har valg at kalde `aggregator`. Ideen blev udarbejdet i vores forspeciale [5] med henblik på at gøre det nemmere at lave `game engines` som bedre kan gøre brug af parallelitet. Ideen er inspireret af `software transactional memory`.

Vi starter med at beskrive teorien bag ved `aggregatoren`, som omhandler hvordan `aggregatoren` håndterer hukommelse over flere tråde uden behov for at blokere mellem trådene, og en sammenligning med `revisions`, da både revisions og aggregator er lignende ideer som er inspireret af `software transactional memory`.

Efterfølgende designer vi tre game engines; en som kører på én kerne, en som kører parallelt med traditionelle låse, og en som benytter aggregatoren. Derudover designer vi også et lille spil, som skal bruges til at vise at det er muligt at lave det samme spil i alle tre game engines. Vi implementerer alle tre game engines i C++, med SFML som grafikbibliotek til at rendere. Alle tre game engines implementeres med et fælles interface, for at begrænse interfacet mellem game engines og spillet til de nødvendige implementeringsforskelle i de forskellige game engines.

Vi tester alle tre game engines over tre forskellige tests: en test for performance overhead ved én logisk kerne, en test for arbejsmængde, og en test for hvor godt de håndterer arbejde i `critical regions`. Alle test køres på fire forskellige testsystemer for at få en forståelse for, hvordan hver engine klarer sig under forskellige forhold. Resultaterne viser, at aggregator game engine mod vores forventninger klarer sig bedre på én kerne end den game engine, som vi har lavet til at køre på en kerne. På flere kerner klarer aggregatoren sig 4.6 gange så godt som den sekventielle game engine, men den parallelle game engine med låse klarer sig stadig 7.5 gange bedre. I testen med `critical regions` klarer den parallelle game engine sig bedst, mens at aggregator game engine ikke formår at klare sig bedre end den sekventielle game engine.

De tre game engines vi designer i rapporten er simple, og udelader mange funktionaliteter som ellers er at finde i de game engines, som findes på markedet, og vi giver derfor en oversigt over nogen af funktionaliterne, og hvordan de kan påvirke aggregatoren.

Til sidst diskuterer vi aggregatoren og testresultaterne, før vi konkluderer på projektet. Vi kommer frem til, at der er potentiale for aggregatoren, men at den med vores nuværende implementering har plads til forbedringer. Vi giver også et bud fremtidige muligheder for at arbejde med aggregatoren.

Contents

Preface

This report was written by Carsten Schroll Ibsen and Frederik Palmelund Voldby from project group pt101f19 at Aalborg University. The project was made in the period of February 1st to June 6th as our master thesis. We would like to thank our supervisor Lone Leth Thomsen for her feedback and guidance. We would also like to thank Bent Thomsen for inspiration and input. Finally, we want to thank our group-room mates from pt102f19 and pt103f19 for being good sports and camaraderie with room for ideas and help.

The code developed in this project is freely available under MIT license at `https://github.com/palmelund/Master-Engine`.

Aalborg University, 06/06/2019

Aalborg University, 06/06/2019

Frederik Palmelund Voldby
fvoldb14@student.aau.dk

Carsten Schroll Ibsen
cibsen13@student.aau.dk

Reading Guide

This report documents the design, implementation, and test of three game engines, with focus on the aggregator game engine, which is an implementation of our idea for concurrency based on software transactional memory.

We assume the reader has prior knowledge with concurrency, parallelism, and software transactional memory. A prior understanding of game engines, specifically Unity, will also be beneficial. We discuss concurrency, parallelism, and game engines in [5].

In this report we cover the following chapters:

Chapter 1: Introduction We present our motivation for the aggregator, and how it is a continuation of prior work of ours, followed by a presentation of the problem definition.

Chapter 2: Memory Isolation We present the aggregator, and how it can integrate with a game engine to create long-term, non-aborting, non-blocking parallelism based on software transactional memory. We also compare the aggregator with revisions, which are also based on software transactional memory.

Chapter 3: Design A design is presented for three different game engines: a sequential game engine, a parallel game engine with locks, and a game engine implementing the aggregator. We also design a small game, which will be implemented in all three game engines.

Chapter 4: Implementation We limit our choice of technology for programming language and rendering framework. Implementation details are then presented for all three game engines, and the implementation of the game in each of the game engines.

Chapter 5: Test We perform three different tests on each of the game engines over four different test systems. The tests involve single-core overhead, work-load, and performance

in critical regions.

Chapter 6: Excluded Engine Features Several features which are present in commercial game engines were excluded from the game engines we develop. In this chapter we discuss the features, and present how they would affect the aggregator game engine.

Chapter 7: Discussion We discuss our test results and test approach, design and technology decisions, and how we perceive the real-world application of the aggregator.

Chapter 8: Conclusion In the conclusion we present our work and findings, and present future work involving the aggregator.

Appendix A: Test results Test results for all three tests on all four test systems.

Glossary In the glossary we define terms which are necessary to know to understand the report, or to understand our intentions when using the term.

Glossary	
Term	Definition
Game developer	A person involved in game development.
Game designer	A person involved in game development, but with no involvement in programming.
Game programmer	A person involved in game development, with involvement in programming.
Iterative system	A program with a main loop which iterates over the same methods for an unspecified duration, and where the output state of one iteration is the input state of the next iteration. Game engines are the iterative system we focus on in this project.
GameObject	Base object for objects which directly interface with the game engine.
Frame rate	Number of rendered frames per second
Sequential game engine	Game engine which only makes use of one CPU core
Parallel game engine	Game engine which makes use of multiple cores, and uses locks for mutual exclusion
Aggregator game engine	Game engine which makes use of multiple cores, and uses the aggregator for mutual exclusion (see chapter 2)
Multithreaded game engines	Used when referring to both the parallel game engine and the aggregator game engine

1 Introduction

Computer games are resource demanding programs, which pushes the computer hardware to the limit. But one part of the computer hardware which is not always fully utilized is the CPU, which in modern CPU's consist of many cores [12].

In [5], we investigated the architecture of a game engine, and how functional programming and parallelism could be used to improve the CPU utilization of games. The research resulted in the design of a pattern for iterative systems, and targeted towards game engines, we named `aggregator`, based on `software transactional memory`. The aggregator is described more in detail in chapter 2 in this report, but a short explanation follows below.

The aggregator is inspired by immutability in functional programming, but altered such that the game state does not change for the duration of an iteration in a game engine, but keeps track of assignments to variables internally, and merges the interval assignments with its variable at the end of each iteration in the game engine. Immutability allows games to trivially run in parallel, as there is no need for locking between threads. The game programmer is able to define how conflicts are handled when merging changes to a variable. In a game engine, the aggregator allows the three phases `input`, `update`, and `render` to be performed in parallel, where they traditionally are handled sequentially, even in concurrent game engines.

We concluded [5, p27-p42] with a design for a basic game engine using the aggregator, and preliminary test specification, but without any implementation and test results to determine if the idea works and is viable in game engines and games.

In this project we execute parts of the test plan in [5, p39-p42] where we design, implement, and test three game engines. The design involve a `sequential game engine`, a `parallel game engine with locking`, and a `parallel game engine with the aggregator`. We also implement an identical game in each of the three game engines. The first game engine is a sequential game engine, which is our base engine implementation, acting as a base benchmark to measure the multithreaded game engines against. The parallel game engine is an adaption of the first engine modified to be

multithreaded, using traditional locking, to avoid race conditions. The aggregator game engine is implemented using the aggregator, which allows for a parallel implementation which better utilizes resources at the cost of the immutability of the game state. We include all three game engines to measure how the aggregator game engine performs compared to a sequential game engine, as an indication if we actually benefit from multiple cores, and a parallel game engine, to determine how well we could benefit from multiple threads.

In [5, p1-p5] we used benchmarks from PCGamer [32] to determine that computer games do not properly utilize computer hardware, as the base for our motivation. From the results of our research we designed the aggregator, but never got a change to test it. We find it interesting and essential to implement and test the aggregator, such that we can conclude if the idea is a viable alternative for concurrency in game engines.

1.1 Problem definition

In this section, we propose the problem definition and sub-questions which we investigate in this report. The problem definition is based on what we learned in [5] and working to increase the knowledge and understanding of the aggregator.

We define the problem definition as:

How would a game engine implemented using the aggregator compare to game engines using a sequential implementation or a lock-based parallel implementation.

1.2 Delimitation

In this chapter we have presented our problem, and motivation for working with the problem. We also mentioned that the game engine designed in [5] was basic, as a lot of features which are present in game engines implicitly have been excluded from the engine. Such features include, but are not limited to, many resources, a full physics engine except for collision detection, full 2D/3D support, components, and fixed updates. We explain each of these features, and how their inclusion could affect the aggregator in chapter 6. The exclusions allows theory, design, and implementation to be more focused on the core of the aggregator in a game engine.

In the next chapter we describe the theory the aggregator, and draw a comparison with software transactional memory, and related works.

2 | Memory Isolation

In this chapter we describe the `aggregator`, including clarification of many of the shortcomings and uncertainties which were present in in the original explanation in [5, p.27-p.30]. We then present `revisions` in related work, which like the aggregator is also based on `software transactional memory`, and compare similarities and differences between those ideas and the aggregator.

2.1 Aggregator

In this section, we describe the theory of the aggregator. The aggregator is a pattern which allows programmers to trivially parallel code in iterative systems. The aggregator is inspired by functional programming and software transactional memory. The game state is observably immutable during an iteration, hiding all updates and assignments on affected variables internally, and merge them into the variables at the end of an iteration. In a concurrent context, the aggregator allows code to be executed on multiple threads, with zero locking between them. Forking and joining are handled internally, and the programmer only has to provide what should be executed using the aggregator. The aggregator requires no use of locks by programmers, and allows for non-blocking access to critical regions.

Throughout the rest of the report, we discuss the aggregator in the context of game engines. We design a game engine with the aggregator in section 3.4. We also describe the implementation of a game engine with the aggregator in section 4.6 and game using the aggregator engine in section 4.7.

We originally defined the aggregator in [5, p27-p37] but found that the descriptions were confusing and lackluster. In this report, we expand upon the definitions and clarify where necessary.

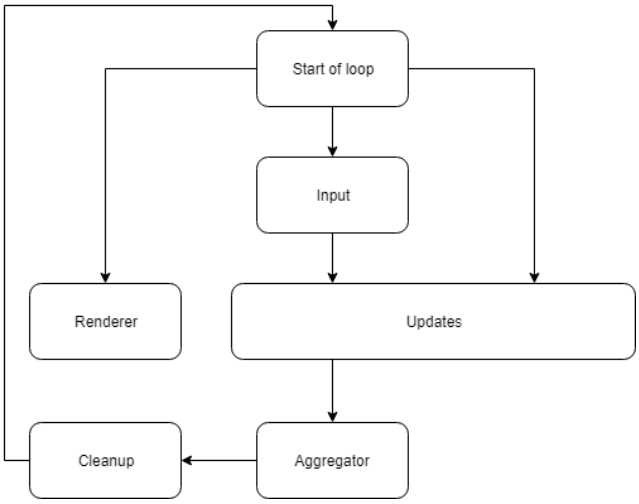


Figure 2.1: Iteration of a potential game loop using the aggregator.

2.1.1 Game loop integrating the aggregator

In this subsection, we describe the aggregator with a focus on how it can be integrated with a game engine. As part of the description, we also summarize the core parts of the game loop in a game engine, while we have a general description in [5, p7-p16]. The game loop of a simple game engine [1] has three main phases: `input`, `update`, and `render`. In a conventional game engine, the three phases are executed sequentially. With the aggregator, the three phases can be executed in parallel.

Figure 2.1 shows a game loop with the integration of the aggregator. At the start of each iteration each of the `input` phase, `update` phase, and `render` phase are started in parallel, but with some `updates` depending on the `input` phase to finish. `Input` is allowed to block some `updates`, as some `updates` depend on the `input` state for the current iteration for correct behavior, such as player movement.

The `input` phase is responsible for handling all player input. Input devices include mouse and keyboard, but can in some games also extend to joy-sticks or wheels. In games with clickable user interfaces, information about which buttons the user clicks are also handled in the `input` phase. Mouse interaction with user interfaces goes beyond the goals of this project, and will not be covered further in this report, limiting the input state to keyboard only.

The `update` phase is responsible for running all updates on game objects. The update phase executes the `update` and `collision` functions for each `GameObject` in the game. There are no requirements for whether `update` or `collision` is processed first

[1] The full game loop for Unity can be seen in [18].

in a game engine, but the order does not matter in the aggregator as the results from either wont affect the results of the other.

The `render` phase is responsible for rendering the frame to display at the end of the iteration. The frame is rendered using the game state from the previous iteration. The consequence is that the game always is one frame behind, compared to game engines where the `render` phase is processed after the `update` phase in the same frame. With a sufficiently high frame rate, the difference will be unnoticeable to the player.

By implementing the aggregator into the game engine, updates and assignments are hidden away during the `update` phase, which trivializes parallelism, as there is no blocking when updating variables in a concurrent context. One thread updating a variable does not affect another thread updating the same variable.

The `input` phase remains the same, except that it only blocks execution of the parts of the `update` phase which depends on input for the current frame. Due to `input` being run at the same time as updates, the input state is mutable in the game engine during the `update` phase.

The `render` is executed alongside the rest of the `phases`, as the game state does not change during an iteration until after the `renderer` is done.

The `aggregate` phase is a new phase introduced into the game engine, which allows the game engine to aggregate all changes made to affected variables, and make the changes visible in the game state. When aggregated, variables are merged using merge rules explained in section 2.1.2.

Even though the aggregator makes the game state immutable, only types which utilize the aggregator are affected. Local variables are always excluded, and class members have to be defined using types integrating with the aggregator to obtain the behavior. In subsection 2.1.2, we define behaviors of types which are used with the aggregator. The aggregator also introduces some overhead, which can be avoided for class-member types which are only used internally in a type, where there is no risk of a race-condition between threads.

2.1.2 Behavior

In this subsection we describe the different behaviors that types can have, depending on how they are modified. When the aggregator was first described in [5, p27-p37], we presented some rules for how types in the aggregator behave but found them to be insufficient. We expand upon them, drawing inspiration from section 2.2, with the introduction of priority assignment.

Types that utilize the aggregator implements three different behaviors: `deletion`, `as-`

signment, and relative operations. The operations are prioritized, and mutually exclusive. If a deletion behavior is present, the aggregator ignores all assignment and relative operations behaviors, while the presence of assignment behavior excludes relative operations behavior. If none of the behaviors are present for a variable, no aggregation is necessary for the variable.

Deletion behavior

The first behavior is deletion, which indicates if a type has been marked for deleting by the game. If a type is marked for deletion, it is an indication that a type will not exist beyond the end of the current iteration, and all assignment and relative operations behaviors will be ignored when aggregated, as the object no longer exists.

Assignment

The second behavior is assignment, which indicates that a value has been assigned to the variable. To give the game programmer more freedom beyond that the first or last assignment of an iteration is the final value, priority assignment is introduced. If an assignment is made for a value with a higher priority, it stores the value and priority until the aggregation step at the end of the iteration.

Relative operations

The third behavior is relative operations, which indicates that a value has been modified by operations other than an assignment. Relative operations include all assignment operations other than = by itself. When aggregating the results of relative operations, the results are applied by their mathematical order, where all multiplications are applied to the resulting value before additions. Examples of relative operations are assignment operations such as $+=$ and $*=$.

2.1.3 Example with the aggregator

In this subsection, we give a short example explaining the behavior of the aggregator on two variables across two threads.

Figure 2.2 shows how two variables x and y behaves across two threads over an iteration. At the start of the iteration, both variables are instantiated to 0. At some point, the first thread increments x by 1. Later in the same iteration, the second thread also increments x

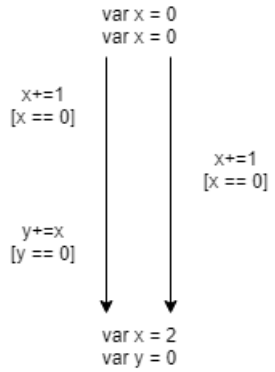


Figure 2.2: Two threads modify shared variables. `[]` shows that the state value of the variables remains the initial value, while the updates are visible after the threads.

by 1. After the second increment, the first threads increments `y` by the value of `x`. After both threads have joined, it can be observed that `x` has the value 2, while `y` has the value 0.

The resulting value of `x` is the sum of the increments from both threads. `y` remains 0 despite the same thread incremented `x` earlier in the same iteration. The lack of visible updates to the values is caused by the immutability of the game state during the `update` phase. If the same threads were to run for a second iteration with the updated state, the resulting values of `x` is 4 while `y` is 2, as the read-value of `x` is the resulting value of the previous iteration.

2.2 Related work

In this section we describe a similar idea called revisions, based on software transactional memory. We use the related works to draw an comparison with the aggregator in section 2.3. We base the description on three papers with shared authors.

The article *Lightweight Software Transactions for Games (Baldassin, Burckhardt)*[2] investigates the use of software transactional memory and revisions in a small game. *Concurrent Programming with Revisions and Isolation Types (Burckhardt, Baldassin, Leijen)* [4] covers further research with revisions using the same game. A third article *Semantics of Concurrent Revisions (Burckhardt, Leijen)*[3] focuses on the semantic parts of revisions.

The basic unit of concurrency in the articles is revisions, which are tasks that can be run on separate threads. Revisions can spawn new revisions, and are later required to be rejoined into a revision with the handle created when it was forked. Revisions are based on software transactional memory, but are long-running, non-aborting and non-reverting.

Figure 2.3 shows a simple program with two versioned variables `x` and `y`. Changes made by revisions to versioned variables are locally visible on the revision, but are only visible

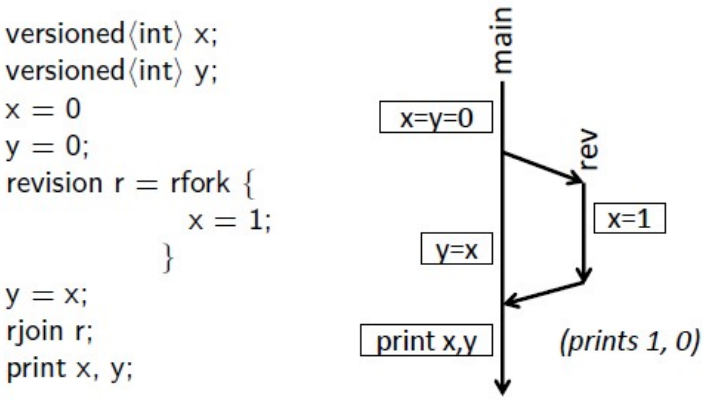


Figure 2.3: Revision diagram showing isolation between revisions. Figure 2 in paper [4].

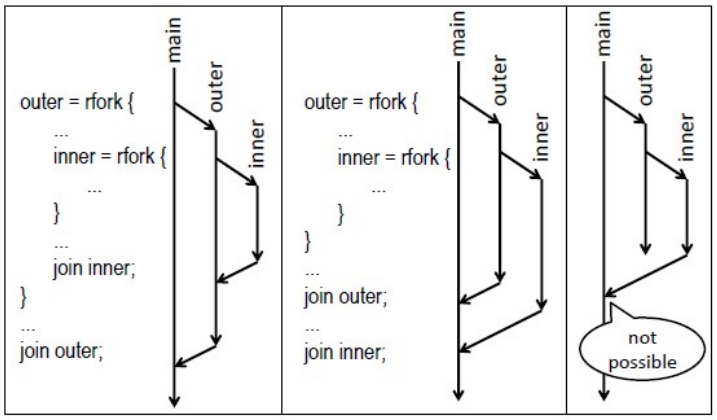


Figure 2.4: Revision diagram showing nesting and joining of revisions. Figure 3 in paper [4].

to other revisions after joining. The main revision instantiates both variables to `0`, before spawning a new revision. The spawned revision assigns `x = 1`, while the main revision assigns `y = x`. On join, the versioned value of `x` becomes visible to the main revision. The assignment by the main revision was executed before the join, leaving the value of `y = 0`, while `x = 1`. The result is always the same, independent of the order the two assignments are executed between fork and join.

Figure 2.4 shows the flow of forking and nesting revisions, and which operations are allowed for joining revisions. Any revision can fork a new revision, but not all revisions can join back into other revisions. A revision can always be joined with its parent, as the parent holds the handle to the revision. If a parent has already joined with its own parent, the handle to a child revision is passed to the new parent, which then can join with the revision. It is not possible for a revision to join with a grandparent, if the parent has not yet joined the grandparent first.

The article defines two memory types for handling shared memory in isolation: `versioned` and `cumulative`.

Versioned Versioned memory, such as the variables used in figure 2.3, are used where there is no risk of concurrent modifications. When versioned types are joined, the value of the forked revision is checked to determine if the value has changed. If the value is identical to the parent revision, nothing is done, otherwise the value is overwritten by the forked revision on join. Versioned types are also used where there is a priority between tasks, such that results of higher priority revisions override the results of lower priority revisions. In a multiplayer game, priority is used to indicate that network packages are more important than local simulation [2] of the values in the package.

Cumulative Cumulative memory is used when a write conflict can occur between two revisions. The cumulative type takes a merge function, which is given the current value of the parent and fork revision, as well as the original value from when the revision was forked. The programmer can then specify how the three arguments should be combined into a resulting value. The code in listing 2.1 shows how a simple merge function could look for integers, where they take the sum of the two revisions on join, and subtract the original value at the point of forking.

```
1 int merge(int original, int master, int revised) {
2     return master + revised - original;
3 }
```

Listing 2.1: Merge function for integers.

Revisions and isolation is tested through an implementation of a game called SpaceWars3D [4]. SpaceWars3D is used for teaching DirectX [2], and the game is not originally designed

[2] https://docs.microsoft.com/en-us/windows/desktop/direct3d

with intentions for multithreading, but the authors found potential for multithreading the collision detection in the game. The authors also found that there are restrictions to rendering in the framework, limiting parallelism. Another opportunity for parallelism is the auto-save system, which otherwise causes the game to freeze for a short duration.

A total of 22 types in the game were replaced with isolation types, being one of four types: `VersionedValue`, `VersionedObject`, `CumulativeValue`, and `CumulativeList`.

They describe that the implementation is not always deterministic, as they cannot guarantee that auto-save finishes in a timely manner, and have to check if it is done before joining. User input, network packet timing, and random numbers all lead to non-determinism. They redefine their goal from `deterministic` to `deterministic record & play`, as it is easier to debug.

In a sequential implementation they found that revisions introduce a `5%` overhead over a normal sequential implementation. On four cores they got a speed-up of `2.6x`. The limiting factor for the game is the render task, which accounts for `99.5%` of the frame time that cannot be parallelized.

2.3 Comparison of models

In this section we compare the aggregator with revisions, which we cover in section 2.2. We also compare the aggregator to software transactional memory, which both the aggregator and revisions are inspired by.

Both the aggregator and revisions are long-running and non-reverting transactions. Both transaction types run for a long time: until the user joins them back into the system for revisions, or the end of an iteration for the aggregator. Both are also non-aborting, meaning that they continue execution after the values they are bound to are changed, and do not roll back. Instead, when joining, they merge their value based on merge rules, which for for revisions and the aggregator can be defined by the programmer.

The aggregator differentiates itself from the revision and transactional memory in that it is immutable during an iteration. When assignments or relative operations are performed on a variable affected by the aggregator, the updated value is stored internally by the aggregator until the end of the current iteration, at which point it merges any update that might have happened to a variable, and makes it the new value if the variable for the next frame. For revisions, the updated value is visible locally to the thread.

Figure 2.5 shows how the three models `sequential consistency`, `transactional memory`, and `concurrent revisions` each handle updates to variables over two different threads. In the same scenario, where two variables are incremented if the other is not,

(sequential consistency)	(transactional memory)	(concurrent revisions)

x = 0 ; y = 0 ;
t = fork {
if (x = 0) y++ ;
}
if (y = 0) x++ ;
join t ;
	x = 0 ; y = 0 ;
	t = fork { atomic {
	if (x = 0) y++ ;
	}}
	atomic { if (y = 0) x++ ; }
	join t ;
		x = 0 ; y = 0 ;
		r = rfork {
		if (x = 0) y++ ;
		}
		if (y = 0) x++ ;
		rjoin r ;

assert((x = 0 ∧ y = 1) ∨ (x = 1 ∧ y = 0) ∨ (x = 1 ∧ y = 1)) ;	assert((x = 0 ∧ y = 1) ∨ (x = 1 ∧ y = 0)) ; ;	assert(x = 1 ∧ y = 1) ; ; ;

Figure 2.5: Outcomes under different programming models. Figure 2 in [3].

the aggregator assert identically to `concurrent revisions`, as updates to variables are invisible until the threads are joined and merged.

2.4 Determinism

In this section, we describe determinism in relation to games and the aggregator. We argue when behavior should and should not be deterministic.

Physics in a game should always produce similar results when observed by the player. Internally physics can have small deviations, as long as the results are close to the same. Physics engines in games prioritize speed over precision [33]. Side effects caused by physics, such as collisions or triggers, might have more strict requirements for precision.

Random numbers are commonly used in games for a variety of things, including AI. Random numbers are explicitly introduced by game programmers, and as such do not affect the determinism of the aggregator itself.

When aggregating at the end of an iteration, everything is applied following the order of behaviors defined in subsection 2.1.2. As long as the game is developed with the restrictions of behaviors in mind, the game behaves deterministically. Assignment with the same priority is non-deterministic, as only the first result is kept, which depends on the `update` which is run first, which the engine again schedules first-in first-out using the internal order.

When describing behaviors in subsection 2.1.2, we describe that all multiplications are applied to a value before additions in the aggregate step. We make a distinction between the two instead of just keeping an updated value in the background which is affected by both multiplication and addition, as it makes the results more deterministic, and thus more pre-

dictable. If the two were treated as a single value, it would also only leave a single value to reduce between multiple threads, which is harder to reason about, as it is the same base number which has been added and multiplied to with different numbers on different threads. Should the result then be the new average, adding the difference from the base together, or something else? While the behavior can be defined, it makes the results less deterministic, and the solution seems more arbitrary.

2.5 Delimitation

In this chapter we have presented the theory of the aggregator, and how it can work in a game engine, by changing the flow of the main loop. We have also compared the aggregator with revisions, as they both are similar ideas inspired by software transactional memory. We also investigated the importance of determinism in games, and if it is possible to implement the aggregator with deterministic execution. This chapter gives background knowledge for the aggregator necessary to follow the decisions behind the design for the three game engines including the game engine implementing the aggregator that follows in the next chapter

3 Design

In this chapter, we describe the design of the game engines we use for testing the aggregator in chapter 5. We first present the design for the single-threaded game engine, followed by the design changes necessary for making the game engine parallel with locks, and finally the changes to design necessary for integrating the aggregator, explained in section 2.1, into the parallel game engine. At the end of this chapter, we design a game which we use to demonstrate that it is possible to create a playable game using all three game engines. Parts of the game is also used in testing, which we explain in section 5.1. The requirements for the game engine are explained in section 3.1, excluding the choices we already made in section 1.2. The design for the sequential game engine is described in section 3.2, and acts as the base on which both multithreaded game engines are built on top. Design details for the multithreaded game engines are explained in section 3.3 and section 3.4. Design for the game is described in section 3.5.

We are inspired by our previous experience with the Unity game engine, which influences our design. We keep the game engines simple, compared to commercial game engines such as Unity, to limit necessary design and implementation while still making a testable game engine.

3.1 Engine Requirements

In this section we list the requirements for the game engine, which we base the game engine design on. The requirements also ensure that we have the features necessary for designing the game in section 3.5. The requirements are based on the original game engine requirements in [5].

The game engines follows the game loop: `frame start`, `input`, `update`, `collision`, `delete`, and `render`.

In the `input` phase the engine polls for input, and determines the current state of all relevant keys. Relevant keys are specified by the game programmer, and added to input as

part of startup of the game engine. The `update` phase calls the update function of all `GameObjects`, the `collision` phase checks for collision on all `GameObjects`, calling `OnCollision` on all objects that are involved in a collision. Deleted `GameObjects` are deleted in the `delete` phase, after the `update` and `collision` phases are done to avoid invalidating iterators. `GameObjects` are rendered on the screen in the `render` phase.

The game engines will implement the following features, which are required for the game designed in section 3.5, and for the tests in chapter 5:

GameObject GameObjects are the master object which other game elements inherit from.

Input The game engine must be able to accept user input for any specified key on the keyboard.

Instantiate GameObjects can be created and instantiated. Part of instantiation includes allowing GameObjects to to be rendered or check for collision.

Render GameObjects with sprites are rendered onto the screen. Sprites have a size and position, and can be transparent. Sprites will be batched to reduce render time and GPU time.

Collision GameObjects can have rectangular colliders attached to them, a function to call on collision, and can be toggled to actively check for collisions, or passively be checked against.

Time Frame rates are variable, and delta times should be available to account for the length of the frame.

Resources Resources can be loaded into memory from disk. The operation performs disk I/O, and should be possible at all times in the engine.

Audio was part of the original design in [5, p13-p16], but its inclusion is not essential for a game engine, and would not contribute to the tests in chapter 5, because audio always should be put on its own thread [28]. The `input` phase is not used in any of the tests performed in chapter 5, but is still used in game we design in section 3.5.

3.2 Sequential game engine

In this section, we describe the design for a single-threaded implementation of the game engine, based on the requirements in section 3.1. The design in this section is the foundation for all three game engines we design, as both multithreaded game engines are built on top of this design.

Figure 3.1 shows the UML design for the sequential engine, and the classes involved. Most classes in the engine are static. Each static class is accessed multiple locations in the code, and only one instance is ever required. The static classes are: `Game Engine`, `Time`, `Input`, `Renderer`, and `Resource Manager`. We make the classes static as passing

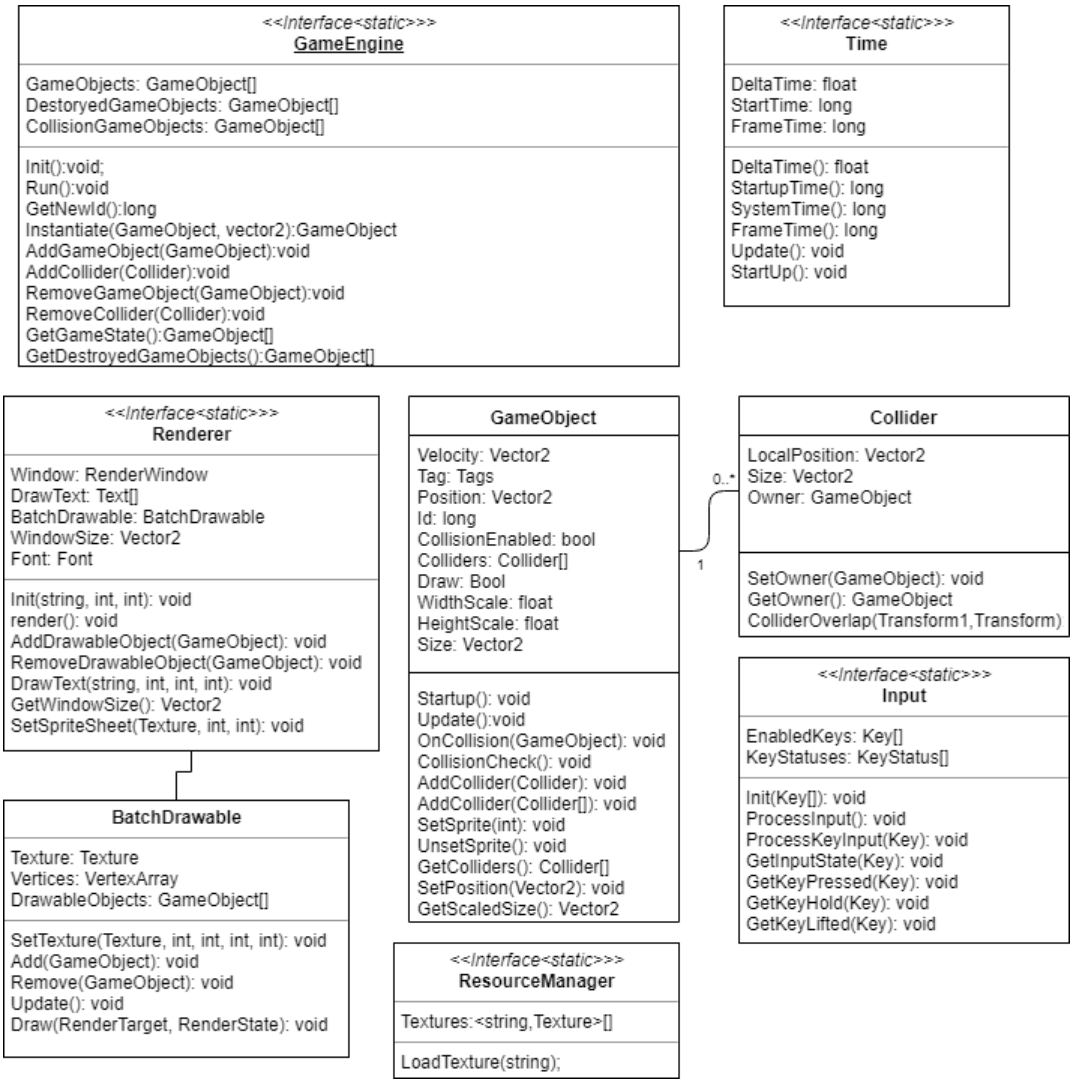


Figure 3.1: UML diagram for the sequential game engine.

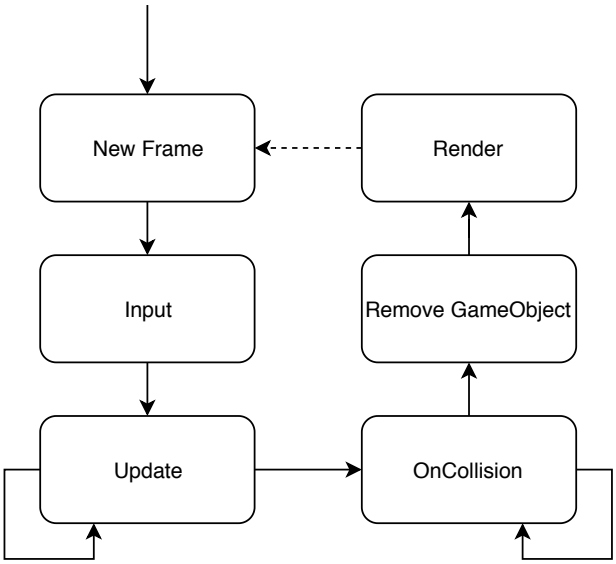


Figure 3.2: Flow of the sequential game engine.

them around gets cumbersome, and often would require the game programmer to pass them around, even without using them directly. Most of the classes could be made non-static, but would require the game engine to be a singleton instead. In the rest of this section we describe each of the classes in the diagram, and what role they serve in the game engine.

Figure 3.2 shows the flow of the sequential game engine, based on the game loop described in section 3.1. A frame starts in `New Frame`, where frame setup is handled. Time and delta time is updated at the start of the frame, and when the game engine is under test, logging is also performed as part of `frame start`. Input reads the current state of all tracked keys, and updates the internal key state. Input keeps track of the keys for both the current and previous frame, to tell the difference between a key which is pressed down in the current frame or previous frame, and if the key was released in the current frame. The `update` phase iterates over the update functions for all game objects. Unlike for collision checking, updates will always be executed for `GameObjects`. In the `collision` phase, all `GameObjects` which have collision checking enabled checks against all objects with colliders, to determine if they collide with anything, and on collision calls the `OnCollision` function on the `GameObject`. The game engine then deletes all `GameObjects` which have been marked for deletion. This operation is performed separately from the other operations to avoid iterator invalidation. `GameObjects` removes themselves from rendering and collision checking, and the game programmer is responsible for ensuring that the objects are also removed from all other collections where they can have pointers. The `renderer` renders all `GameObjects` which have a sprite on the screen, and the game engine then moves back to `new frame`, where it can start the next iteration in the game engine.

3.2.1 GameObject

In this section, we describe the design for the GameObject.

The GameObject is the core of any object defined by a game programmer in the game engine. The GameObject is inspired by Unity's GameObject [19] and MonoBehaviors [20]. When an object inherits from the GameObject it can override the `start`, `update`, and `OnCollision` functions to add logic. The `start` function is called once by the engine when the GameObject is created and instantiated, `update` is called once every frame during the `update` phase, and `OnCollision` is called when the game object collides with anything during the `collision` phase.

The `Startup`, `Update`, and `OnCollision` functions are reserved for execution by the game engine.

3.2.2 Game Engine

The `game engine` object is the core of the game engine, handling initial setup, and running the main loop of the game engine. The `init` function sets up the system. The `run` function starts the game loop. The loop first updates `time` for the frame, then processes `input`, `updates`, and `collision`, and finally `destroy` objects and then render the current frame. If time logging is enabled, such as for testing, it is also handled at at the start and end of each iteration in the `run` function.

The `game engine` class also contains global collections of all instantiated `GameObjects`, and all game objects that have the flag `enable_collision` set to true. The collections are mostly used by the game engine, but are also available to the game programmer if they need to search for specific `GameObjects` which are currently instantiated in the game.

The `Init`, `Run`, and `Instantiate` functions are required to be called by the game programmer when integrating the engine with the game. All other functions are reserved by the game engine.

3.2.3 Time

The `time` class provides the game with functions related to time. The most important function in `time` is `DeltaTime`, which returns the time passed since last frame. `DeltaTime` is used by everything in a game which depends on the frame rate, such as movement and countdowns.

The `StartUp` and `update` functions are reserved by the engine and are used for initializing time values for the first frame and updating time information for each frame.

3.2.4 Input

The `input` class provides functions for accessing user input, and keeps track of all keys defined in the game engine, and their state. A state is defined by the physical keys pressed for the current frame and previous frame. A key has one of the four states: `unpressed`, `pressed`, `hold`, or `lifted`. The key states are inspired by input in the Unity Engine [25].

3.2.5 Renderer

The renderer is responsible for all rendering to the screen. Rendering is limited to sprites and text to keep the design simple. More complex rendering could include user interface, sprites of arbitrary size in the spritesheet, and meshes. The renderer generates a single image to draw on the screen using `BatchDrawable`. The renderer also keeps a list of all text which is to be drawn on the next frame. The rendering pipeline in the game engine is kept simple as a custom graphics pipeline goes beyond the scope of the project.

3.2.6 BatchDrawable

`BatchDrawable` contains an internal collection of `GameObjects` to draw. Each `GameObject` contains an index for the sprite which the `BatchDrawable` should draw for the `GameObject`. The `BatchDrawable` contains a sprite sheet, which contains all sprites. The `BatchDrawable` can produce a single image, which can be drawn by the `renderer`. The `BatchDrawable` could be optimized by keeping track of which elements move on the screen, or keeping separate `BatchDrawables` for moving and static `GameObjects`, but as all elements in the games and tests are moving on the screeen, such optimizations would be unsuitable for this project. Because objects move on the screen it is not possible to cache anything, and the full image has to be recalculated during every frame. The `BatchDrawable` uses a spritesheet for rendering, where each `GameObject` has an index for where in the map the texture is located.

3.2.7 Resource manager

The resource manager is the interface between the game and resources for the game, loading resources such as sprites from the disk. Due to the small size of the game engine, and the

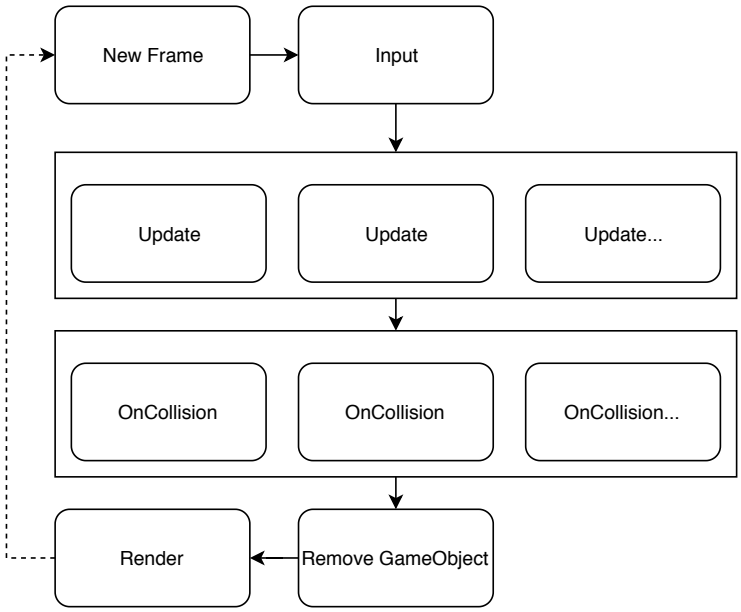


Figure 3.3: Flow of the parallel game engine.

features excluded in section 1.2, the only resource used in the game engine is sprites. Sprites are created from textures.

The resource manager offers a single function for getting textures: `LoadTexture`. When a texture name is specified, the resource manager first checks if the texture is already loaded into cache, and returns it. Otherwise it loads it from the file system, and caches it for later look-ups.

3.3 Parallel game engine

In this section, we describe the necessary design changes for the game engine to be ideal for parallel game development. Other than the introduction of locks into certain parts of the game logic, there is no change compared to the design of the sequential game engine in section 3.2. `ThreadPool` is a new class which is added to the UML, and which is used for scheduling `updates`, and `collisions` on multiple threads.

Figure 3.3 shows the flow of an iteration in the parallel game engine. The flow is similar to the sequential game engine on figure 3.2, except for the `update` and `collision` phases. Both phases are parallel utilizing the `ThreadPool`, and can process multiple updates or collision checks at the same time. The engine waits for all threads to complete the workload of a phase before starting the next phase.

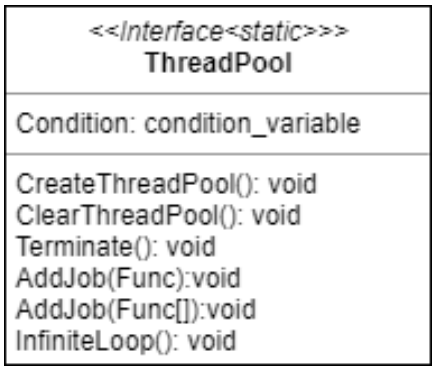


Figure 3.4: ThreadPool for parallelizing game logic.

3.3.1 Thread pool

The `threadpool` takes one or more functions as input, and schedules them to be run on separate threads. A number of threads are created when the threadpool is instantiated, and a thread always start working on scheduled functions if they are not currently busy working on other scheduled functions.

The UML class for the `ThreadPool` can be seen on figure 3.4. The `CreateThreadpool` function is used for instantiating the pool, and creating worker threads. The number of threads created is equal to the number of threads available on the operating system, unless the engine is started with a flag limiting it to one thread, which is used for testing as explained in section 5.1.

The function `AddJob` is used for scheduling a function in the `ThreadPool`. Functions which are intended for scheduling are `updates` and `collision` checks. It is possible for game programmers to schedule other functions in the `ThreadPool`.

3.4 Parallel game engine with the aggregator

In this section we describe the design of the game engine implementing the aggregator. The aggregator is described in section 2.1, and the engine builds upon the engine designed in section 3.3. The design covers how the aggregator game engine handles updates and assignment to variables, and how it handles reducing and merging the updates at the end of each iteration in the game engine.

In [5, p27-p30] we presented a possible solution for keeping track of updates to a variable during a iteration in the game engine. A global collection would keep a record for each variable, which contains all updates for the variable. At the end of the iteration, the aggregator would apply all updates in the record to the variables. The idea was simple, and did not

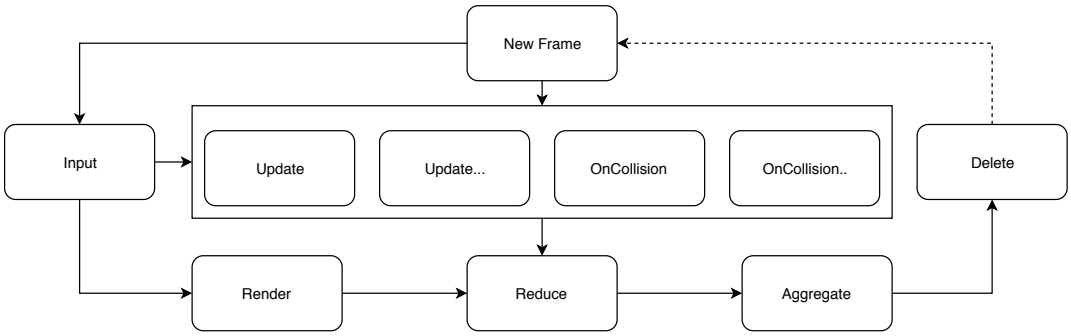


Figure 3.5: Flow of parallel game engine utilizing the aggregator.

consider thread-safety. In this project we improve on using records for storing changes, and do it in a thread-safe, non-blocking manner. The collection of records is changed to contain a list of records for each variable for each thread that has had an update for the current thread. Updates from each thread are then tracked separately, and can then be combined before aggregating with the introduction of a reduce phase in the game engine, taking place just before the aggregate phase. In the reduce phase, records from each thread are combined into a single collection of records, adding together the lists of changes from each thread into one. Reduction was originally presented in [5, p29-p30], but as a low-priority background process, which would reduce changes, limiting the workload when aggregating.

Figure 3.5 shows the flow for the aggregator. Variables in the aggregator game engine stores updates to variables, but does not expose changes until the aggregate phase, and as such the order in which many of the phases are performed becomes irrelevant. All `updates` and `collision checks` which do not depend on user input are schedules as part of `new frame`. The `input` phase is then processed on the main thread, after which all `updates` and `collision checks` which do depend on input are scheduled. The current frame is then rendered on the main thread, after which it waits for all worker threads in the `ThreadPool` to finish. The main thread then starts the reduce step, after which the aggregator applies the changes to the game state, before deleting game objects which are marked for deletion.

The aggregator uses the same scheduler and thread pool we design in section 3.3, but the `reduction` phase and `aggregator` phase also utilizes the thread pool, as reduction can be performed with a divide-and-conquer approach, and reduced records for variables should have no risk of race conditions in the aggregator step.

3.4.1 Wrapper types and deltas

The records stored in each thread collection follows the form of a `delta` type. A delta type defines what operations have been performed on a variable during the current iteration.

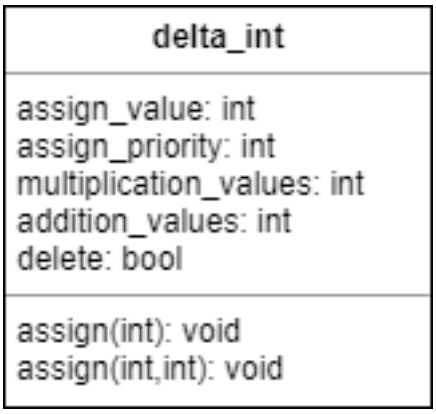


Figure 3.6: Delta value for an int.

Common operations defined in section 2.1 and subsection 2.1.2 are `deletion`, `assignment`, and `relative operations`.

Figure 3.6 shows the UML class for the structure of a basic delta value for integers. The variables `assign_value` and `assign_priority` are a pair, which we combined refer to as an `assignment pair`. The `assignment pair` contains information about assignments to the variable the delta relates to. `assignment_priority` contains the priority of the current assigned value, and any new assignment to the variable is checked against the priority first. If the priority of the new assignment is higher than the value of `assignment_priority`, then `assignment_priority` is set to the new priority, and the value of `assignment_value` is set to the new assigned value. At the start of a new iteration, the value of `assignment_priority` is the minimum possible value of the type, with the value of `assignment_value` being the current real value of the variable.

The variables `multiplication_values` and `addition_values` are used in relative operations. When a number is either added or subtracted through a relative operator, such as +=, the value of `addition_values` is changes accordingly. Same applied for multiplication and division for `multiplication_values`.

The variable `delete` is used to determine if a variable is marked for deletion at the end of the iteration. A deleted value can be a pointer which is deleted or to indicate that the entire `GameObject` is marked for deletion, as the variable the delta belongs to may no longer exist in memory.

Wrapper types are introduced into the game engine and games, as an abstraction over variables and the deltas tied to them. Figure 3.7 shows a wrapper for an integer. The wrapper contains the current value which it wraps, and which always is used on read operations for the duration of the frame. The wrapper implements overloading for assignment operators, which modifies the delta value for the wrapped value.

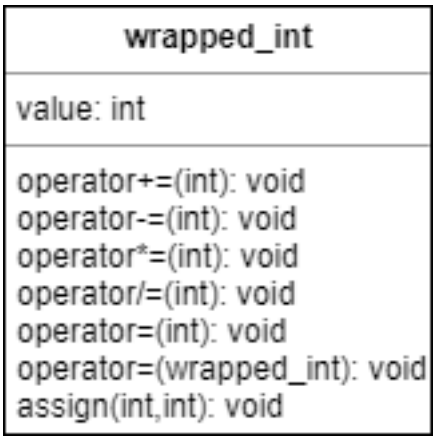


Figure 3.7: A wrapper for int values, with functions for modifying the value.

If a value is not wrapped, it is not affected by the aggregator, and instead modified as normally in an imperative system.

3.5 Game design

In this section we design a small game which we implement in each of the three game engines to demonstrate that it is possible to create a playable game with the aggregator, and which to a player is the same between the engines. Some of the features we design for the game, `background elements` and `gravity wells`, are also used in testing in chapter 5.

In [5, p59-p70] we designed a small game called `Captain Functional` and implemented it in three game engines, Unity Engine (C#), Unreal Engine (C++), and Amethyst (Rust), to experiment with and learn the game engines. In this section, we expand upon the design to create elements relevant for testing.

Specific numbers for features are left out of the specification as some of them will be altered as part of the tests we perform in chapter 5. Other numbers are non-essential for the chapter, and are either used for fine-tuning or visual presentation such as screen size [1].

The game is a side-scroller, where the player moves around within the bounds of the screen using the `WASD` keys, and firing bullets with `Space`. Enemies enter the screen from the right and leave to the left. Enemies can fire projectiles left towards the player, with the projectiles moving at a faster speed than the enemies. Both enemies and bullets can be destroyed by getting hit by the player. A counter keeps track on the number of enemies the player has

[1] For numbers we use in the game engine we refer to the constants file located in MasterEngine/CaptainEverythingShared/Constants.h in the source code.

killed, and another counter keeps track on the number of times the player has collided with enemies or enemy bullets. The value of both counters are printed to the screen.

After a specified number of enemies have spawned on the screen, a boss is spawned, who also enters from the right. The boss is surrounded by a number of shields, which rotates around the boss with a constant speed. All shields have to be destroyed before the boss can be hit directly with a bullet and killed.

Background elements are introduced as objects moving around in the background of the game, colliding with each other. The background elements serve no real purpose other than introducing work on the game engine, bringing down the frame rate. Background elements are spawned into the game from the top, and slowly expands to fill the entire screen as they move around.

Gravity wells are another introduction to the game. They are objects in the game which affects all elements on the screen with colliders, applying a directional force to the objects.

3.6 Delimitation

In this chapter we have described the design of three game engines: a sequential game engine, a parallel game engine, a and aggregator game engine, and described the design for a game which we implement in all three game engines. The parallel game engine builds on top of the design for the sequential game engine, and the aggregator game engine builds on top of the parallel game engine.

In the design we have also described how the aggregator breaks the normal flow of a game engine. Traditional game engines follows the flow of `input`, `update`, and `render` in a sequential order. The aggregator game engine is able to handle `input` and `render` in parallel with `update` and `collision`.

In the next chapter we describe the implementation of each of the three game engines, and details about the implementation of the game in each of the three game engines.

4 Implementation

In this chapter, we describe the implementations of the three game engines and games, and argue for our choice of technologies.

The sequential game engine in section 4.2 and sequential game in section 4.3 describes the sequential implementations, which are the foundation for the other game engine and game implementations. The sections for the multithreaded games and engines describe the interesting changes or additions made to the sequential implementation of the game engine and game.

The code solution is structured in eight projects. Three projects are the game engine libraries. One project is the shared engine functionality, which is used by all three game engines. One project is code which is shared between the games. The last three solutions are game implementations. On the test branch, the game implementations are altered for testing purposes. A separate project with the test runner is also present on the test branch.

4.1 Choise of technology

In this section we discuss the technologies we choose for creating the game engines, and why. We discuss why C++ is our language of choice, and why we use an OpenGL wrapper for graphics and input.

4.1.1 Choice of language

When working on [5] we discussed using C# for implementing the game engines and games, but never included that discussion in the report, as we considered it implementation details, that we had chosen to exclude from the report. C# was attractive to us, as we both have prior experience with creating games in it using the Unity Engine. C# has `Tasks`, which are attractive as they are a built-in feature for writing asynchronous code, that can run on either the same or a different thread[13]. Another language we consider is C++, which is

commonly used for both game development and game engines [6]. We have prior knowledge with using C++ in game development from the work on Unreal Engine in [5, p65-p70]. In [2] they discuss that the rendering library they chose, DirectX, was the cause of a substantial part of their workload, and for the concurrent implementation was the limiting factor for the frame rate. To avoid such problems we wanted to use a smaller graphics library, and decided to choose a language based on said library, where we chose OpenGL, but found that we needed to use C++ to find bindings with proper documentation.

4.1.2 Choice of libraries

We chose to use OpenGL as opposed to DirectX or Vulkan, as we wanted a graphics library with a small CPU and development overhead. DirectX and Vulkan also require a lot more setup from the programmer to be used compared to some OpenGL libraries, as we discuss in [5, p10-p11], and we want to avoid spending too much development time on the graphics library compared to the rest of the game engine, so we chose the simple solution. We originally planned to use GLEW and GLFW as wrappers for OpenGL, but they only provided a simple C interface, while SFML had a full C++ interface, which reduced our graphics implementation size to a minimum, with the exception of `BatchDrawable`, which was introduced to limit the work of both CPU and GPU. SFML is a higher level implementation, but the overhead is small compared to DirectX in [2]. Late in development we came accross a C# library called `MonoGame`, which feature-wise is similar to SFML[17], but have not tested the performance overhead of the library.

To simplify dependency management we use VCPKG, a C++ package manager developed by Microsoft, that integrates into Visual Studio without any additional interaction with a build system.

4.2 Sequential game engine

In this section we describe interesting and central parts of the implementation of the sequential game engine. We will look at the main loop of the game, `GameObject` which is the main object for elements which interface with the game engine, and resource manager which are responsible for fetching resource.

4.2.1 Main Loop

Listing 4.1 shows the main loop of the game engine. The game loop is started by a game programmer with a call to `Run`, after which control is moved to the game engine, until the game window is closed, or the game is otherwise terminated. If the window is closed by

the player, the game receives the `Closed` event, which causes the renderer to clean up, and `is_open()` returns false at the start of the next frame.

```
while (Renderer::is_open()) {
        Time::tick();
        ...
        sf::Event event;
        while(Renderer::poll_event(event)){
            if (event.type == sf::Event::Closed) {
                Renderer::close();
            }
        }
        Input::process_input();
        for (GameObject* object : get_gamestate()) {
            object->update();
        }
        for (GameObject* object : get_gamestate()) {
            object->collision_check();
        }
        for (GameObject* game_object : get_destroyed_game_objects())
        {
            delete game_object;
        }
        get_destroyed_game_object().clear();
        Renderer::render();
    }
```

Listing 4.1: Main loop of sequential game engine.

At the start of each frame, `time` is updated to calculate `delta time` for the current frame.

When the game engine makes a call to `poll_event(event)` is also receives information about a variety of other events, such as window resizing, or input, but we do not use the input events directly from polling. The events are fired for any key which is either `keyPressed` or `keyReleased`, but as mentioned for input in subsection 3.2.4, input in the game engines have four states, and depends on a key state the frame after either of the frames have been fired. We poll the current state for all keys each frame, to determine the current state based on the previous state, as shows in the code in listing 4.2. If a key was pressed down in the previous frame, and continues to be pressed down, its state is changed to being held down, and remains in that state until the key is released. A key is only released for one frame, after which it is unpressed.

```
void Input::process_key_input(sf::Keyboard::Key key) {
    if (sf::Keyboard::isKeyPressed(key)) {
        if (key_statuses_[static_cast<uint64_t>(key)]
            == KeyStatus::unpressed
        || key_statuses_[static_cast<uint64_t>(key)]
            == KeyStatus::lifted) {
            key_statuses_[static_cast<uint64_t>(key)]=KeyStatus::pressed;
        } else {
            key_statuses_[static_cast<uint64_t>(key)] = KeyStatus::hold;
```

```
10          }
11      }
12      else if (!sf::Keyboard::isKeyPressed(key)
13          && (key_statuses_[static_cast<uint64_t>(key)]
14              == KeyStatus::pressed
15          || key_statuses_[static_cast<uint64_t>(key)]
16              == KeyStatus::hold)) {
17          key_statuses_[static_cast<uint64_t>(key)] = KeyStatus::lifted;
18      } else {
19          key_statuses_[static_cast<uint64_t>(key)] = KeyStatus::unpressed;
20      }
21  }
```

Listing 4.2: Process current state of key.

After input has been processed, the game engines processed the `update` function of all `GameObjects` which are currently instantiated. Update functions are used in game code, and an example can be seen in listing 4.7 in section 4.3.

The game engine then performs collision checking for all game objects which have collision checking enabled, and checks the colliders of the `GameObject` against all colliders for all `GameObjects` which are currently instantiated.

After collision, all `GameObjects` marked for deletion are deleted. Deletion is performed after updates and collision to avoid invalidating iterators during updates and collision checking.

At the end of the loop all `GameObjects` are rendered on the screen. Listing 4.3 shows the steps taken to render the the next frame. The window is cleaned, otherwise new rendering would be drawn on top of the current frame. A call to `BatchDrawable.Update()` is used to update the image which should be drawn when the subsequent call to `Draw`. All text is then drawn on the screen after the batch. Elements are drawn in the order in which they are instantiated, so elements instantiated early are drawn behind elements which are instantiated later on. Game engines such as Unity uses layers for controlling which objects to draw on top of others.

```
1  void Renderer::render() {
2      window_.clear(sf::Color::Black);
3      batch_drawable_.update();
4      window_.draw(batch_drawable_);
5      for (auto& text : text_vector_) {
6          window_.draw(text);
7      }
8      text_vector_.clear();
9      window_.display();
10 }
```

Listing 4.3: Code for rendering current frame.

The BatchDrawable was introduced to reduce the time spent drawing a frame. At 300 GameObjects on the screen, the draw call accounted for 12% of all CPU time during a profiling test. To reduce time spent rendering, we implemented a BatchDrawable type, which takes a sprite sheet, and all GameObjects which can be rendered to the screen in a list. During the update of the BatchDrawable, the geometry and sprite of each game object is calculated, before being rendered in a single draw call. At 300 GameObjects the CPU time is reduced to 4%. This result was achieved on a dual core system with hyper-threading for 4 logical cores. In comparison the tests described in section 2.2 used 24% of CPU time on rendering on a 4 core system. There are no images of the game used in the test, making it impossible to compare the graphical similarities between that game, and the game we have created and render.

4.2.2 GameObject

When GameObjects are instatiated, they register themselves in the renderer such that it knows the GameObject can be rendered. It also adds itself to the list of objects which perform collision checks if it is instantiated with the flag set to true. Listing 4.4 and listing 4.5 shows the constructor and destructor for GameObjects. The constructor is responsible for adding the game object to the global list of game objects, and add it to the list of colliders if the collision flag is set. The destructor is responsible for removing the game object from all game engine collections. Multiple collections are used to reduce the need for filtering the global collection of GameObjects multiple times each frame, and the collections are accessed more often than GameObjects are instantiated or destroyed.

```
1  GameObject::GameObject(const bool collision_code)
2              : sprite_pos_(1),
3              collision_code_(collision_code),
4              id_(GameEngine::get_new_id()),
5              velocity_(sf::Vector2f{ 0, 0 }),
6              tag_(Tags::Default),
7              size_(Renderer::get_sprite_size())
8  {
9      GameEngine::add_game_object(this);
10     if (collision_code)
11     {
12         GameEngine::add_collider(this);
13     }
14 }
```

Listing 4.4: Sequential game object constructor.

```
GameObject::~GameObject()
{
    GameEngine::get_gamestate().erase(this);
    unset_sprite();
    for (Collider* collider : colliders_)
    {
        delete collider;
    }
    if(has_collider_)
    {
        GameEngine::remove_collider(this);
    }
}
```

Listing 4.5: Sequential game object destructor.

Setting sprites for a game object are unrelated to the cosntructor, but the destructor still has to clean the renderer for the object, if a sprite is set for the game object. Objects are only drawn if a sprite index is passed to the game object. As rendering of a `GameObject` depends on whatever it has a sprite or not, `unset_sprites()` also checks if a sprite is set first, as it is expected that programmers also has an interest in removing an object from rendering without deleting it.

4.2.3 Resource manager

The resource manager is the interface between `GameObjects` and files on the disk. It handles loading the resources, and present them to the game in a more manageable format. The resource manager also caches all resources it operates on, such that it only has to perform I/O for an resource once. It is not possible to remove objects from the cache, but the cache is not a problem due to the limited use of resources in the game.

```
sf::Texture& ResourceManager::load_texture(const std::string& texture_name
    ){
    auto finding = textures_.find(texture_name);
    if(finding != textures_.end())    {
        return finding->second;
    }
    auto pair = textures_.try_emplace(texture_name, sf::Texture{});
    auto& texture = pair.first->second;
    if(!texture.loadFromFile(texture_name))    {
        throw std::exception();
    }
    return texture;
}
```

Listing 4.6: Sequential resource manager.

Figure 4.6 shows the load function for textures, which is the only resource which we use

in the game engine and games. When a resource is requested, the resource manager first checks the cache for the resource, before loading the file from disk. The resource manager was originally intended to be used for several textures, but with the introduction of `BatchDrawable`, only a single sprite is loaded into the game, as it is used as a sprite map.

4.3 Sequential game

In this section, we explain the implementation of the sequential game, which is built on top of the sequential game engine described in section 4.2. The game is an implementation of the design in section 3.5.

4.3.1 Player

The player object was originally created to be controlled by a player, but with an early focus on determinism in the tests, it was changed to be controlled by a AI without random elements. Test plans were later changed to excluded the player object from tests entirely, and as result changed back to be controllable by a player.

```
1  void Player::update() {
2    fire_rate_cooldown_ += Time::delta_time();
3    if (fire_rate_cooldown_ > fire_rate_
4        && Input::get_key_hold(sf::Keyboard::Key::Space)) {
5      fire_rate_cooldown_ = 0;
6      GameEngine::instantiate(new PlayerBullet());
7    }
8    sf::Vector2f velocity = get_velocity();
9    if (Input::get_key_hold(sf::Keyboard::Key::W)) {
10     velocity.y  = speed * Time::delta_time();
11   }
12   else if (Input::get_key_hold(sf::Keyboard::Key::S)) {
13     velocity.y += speed * Time::delta_time();
14   }
15   if (Input::get_key_hold(sf::Keyboard::Key::A)) {
16     velocity.x  = speed * Time::delta_time();
17   }
18   else if (Input::get_key_hold(sf::Keyboard::Key::D)) {
19     velocity.x += speed * Time::delta_time();
20   }
21   sf::Vector2f position = get_position();
22   position = position + velocity * Time::delta_time();
23     /* Clamping omitted */
24   GameObject::set_position(position);
25   GameObject::set_velocity(velocity);
26 }
```

Listing 4.7: Code snippet of player update.

Listing 4.7 shows how a player `GameObject` is controlled in the `update` function. The player can fire a bullet by pressing `space` if the cooldown has passed the fire rate. When a player fires a bullet, a new `GameObject` is instantiated, which is a bullet that spawns at the players current position, and moves right until it hits something or leaves the screen.

The game then gets the current velocity of the player, and checks each directional key (WASD) for the current state. If the state is `hold`, the vertical or horizontal velocity updated based on the player speed and delta time.

The new position is then calculated based on the current position, new velocity, and current delta time, and then clamped to the screen to prevent the player from moving outside the screen.

Collision checking on a player is used to check if the player collides with enemies or enemy bullets. Listing 4.8 shows how collisions for a player is handled when the game engine has checked that the player is colliding with something.

```
1 void Player::OnCollision(GameObject * collider)
2 {
3     if (collider->get_tag() == Tags::Enemy || collider->get_tag() == Tags
      ::EnemyBullet)
4     {
5         Spawner::get_singleton()->add_player_hit();
6         GameEngine::remove_game_object(collider);
7     }
8 }
```

Listing 4.8: Code collision code for player.

If a player collides with something which have the `enemy` or `enemybullet` tag, the internal counter for player hits is incremented, and the object the player collided with is removed from the game.

4.3.2 Spawner

The spawner `GameObject` is a singleton which keeps track on how many enemies have been spawned, and how many times enemies have hit the player.

```
1 void Spawner::start_up() {
2     for (int i = 0; i < Constants::background_elements(); i++) {
3       GameEngine::instantiate(new BackgroundElement(), sf::Vector2f{
      static_cast<float>((Constants::background_element_size * i) % Renderer
      ::get_window_size()->x),(float)((Constants::background_element_size *
      i) / Renderer::get_window_size()->x) });
4     }
5
6     GameEngine::instantiate(new Player(), sf::Vector2f{ 50.0f, Renderer::
```

```
    get_window_size() >y / 2.0f });
7
8     for(int i = 0; i < Constants::gravity_wells(); i++) {
9       GameEngine::instantiate(new GravityWell(), sf::Vector2f{ Renderer::
    get_window_size() >x / 2.0f, Renderer::get_window_size() >y / 2.0f });
10    }
11  }
```

Listing 4.9: Startup code for spawner which instantiates background elements and gravity wells.

Listing 4.9 shows the startup code for the spawner. When started, the spawner instantiates a number of background elements and gravity wells equal to the number defined in the constants file. All objects spawned as part of `start_up` are spawned on top of each other, which causes some slowdown at the start of the game.

`Update` is called each frame for the spawner, and handles spawning enemies and bosses into the game. Listing 4.10 shows the code for spawning enemies. An internal timer is incremented each frame, and compared against a constant to determine when the next enemy should be spawned. The spawner instantiates new enemies in a deterministic position based on the position of the last enemy and screen size, modified by a constant. If a number of enemies have been instantiated, a boss is instantiated outside the screen, and moves into view of the player. The spawner keeps track on number of enemies killed, and number of hits to the player, and prints the numbers to the screen.

```
1   void Spawner::update() {
2     internal_timer += Time::delta_time();
3     if (internal_timer > spawn_time) {
4       internal_timer  = spawn_time;
5       spawn_position_ += 300.0f;
6       spawn_position_ = static_cast<float>(int(spawn_position_) % (
    Renderer::get_window_size() >y    30));
7       GameEngine::instantiate(new Enemy(), sf::Vector2f{ Renderer::
    get_window_size() >x + 100.0f, spawn_position_ });
8       spawn_count_++;
9       if (spawn_count_ > Constants::boss_spawn_threshold) {
10        spawn_count_ = 0;
11        GameEngine::instantiate(new Boss(), sf::Vector2f{ Renderer::
    get_window_size() >x + 100.0f, Renderer::get_window_size() >y / 2.0f
    });
12      }
13    }
14    Renderer::draw_text("Kills: " + std::to_string(enemy_kills_), 10, 10,
    20);
15    Renderer::draw_text("Damage: " + std::to_string(player_hits_), 10, 30,
     20);
16  }
```

Listing 4.10: Update code for the spawner which spawns enemies and bosses.

4.3.3 Background element

Background elements are game objects which moves around in the background of the game. They move with a velocity, and are reflected when colliding with another background element or the edge of the screen.

```
1  void BackgroundElement::on_collision(GameObject * collider)
2  {
3    if (collider->get_tag() == Tags::Background)
4    {
5      sf::Vector2f position = GameObject::get_position();
6      sf::Vector2f col_position = collider->get_position();
7      sf::Vector2f velocity = GameObject::get_velocity();
8      sf::Vector2f force = { 0,0 };
9      sf::Vector2f relative_position = sf::Vector2f{ (position.x + (size_ /
       2))   (col_position.x + (collider->get_width_size() / 2)),(position.y
       + (size_ / 2))   (col_position.y + (collider->get_height_size() / 2))
       };
10
11     if (std::abs(relative_position.x) > std::abs(relative_position.y))
12     {
13       if (relative_position.x > 0)
14       {
15         velocity.x = std::abs(velocity.x);
16       }
17       else
18       {
19         velocity.x = -1 * std::abs(velocity.x);
20       }
21     }
22     else
23     {
24       if (relative_position.y > 0)
25       {
26         velocity.y = std::abs(velocity.y);
27       }
28       else
29       {
30         velocity.y = -1* std::abs(velocity.y);
31       }
32     }
33     GameObject::set_velocity(velocity);
34   }
35 }
```

Listing 4.11: OnCollision for background element.

Listing 4.11 shows Collision code for background elements. When colliding with another background element, the background element determines on which side it collides with the other game object, and inverts its velocity for one of its axis.

4.3.4 Gravity Well

Gravity wells are stationary objects, whose collider covers an area around their position. Listing 4.12 shows the code for collision handling when an object moves into the collider of the gravity well. When inside the collider, a pull force is calculated, and applied to the colliding `GameObject`. The force is then applied to the velocity of the colliding object, which reacts to the new force at the next update call.

```
1       void GravityWell::on_collision(GameObject* collider)
2   {
3     float side_a = (get_position().x + (Constants::gravity_well_size / 2.0
      f))   (collider->get_position().x + (collider->get_width_size() / 2.0f
      ));
4     float side_b = (get_position().y + (Constants::gravity_well_size / 2.0
      f))   (collider->get_position().y + (collider->get_height_size() / 2.0
      f));
5     float side_c = std::sqrt(std::pow(side_a, 2) + std::pow(side_b, 2));
6     float cal_pullforce = (Constants::gravity_well_size   side_c) *
      Constants::gravity_well_pull_force;
7     sf::Vector2f direction_vector{ side_a / side_c , side_b / side_c };
8
9     collider->add_velocity(sf::Vector2f{ direction_vector.x *
      cal_pullforce * Time::delta_time() ,direction_vector.x * cal_pullforce
       * Time::delta_time() });
10  }
```

Listing 4.12: OnCollision for gravity wells.

4.4 Parallel engine

In this section, we describe the implementation of the parallel game engine. The parallel game engine is almost identical to the sequential game engine with the exception of the introduction of the `ThreadPool`, and introduction of locking in critical regions. We also discuss some of the problems encountered with multithreading the game engine.

4.4.1 ThreadPool

Spawning threads can be expensive when done often. A cheap alternative is to spawn a fixed number of threads at once, and then reuse them [27], which is what the `ThreadPool` does. The `ThreadPool` contains a queue of functions which is executed on an idle thread.

```
void ThreadPool::CreateThreadPool() {
    thread_count_ = CaptainEverythingShared::Constants::thread_count();
    for (int ii = 0; ii < thread_count_; ii++) {
        Pool.emplace_back(std::bind(&ThreadPool::worker_thread_loop, this)
    );
    }
}
```

Listing 4.13: Creation of threads.

Listing 4.13 shows the startup code for the threadpool. The threadpool gets the number of logical cores from the operating system, and spawns an equal number of threads. The code is slightly modified for testing, where it also checks if the test is limited to a single core before instantiating and thread.

Each thread runs the function `worker_thread_loop`, which can be seen in listing 4.14. The loop runs forever until the thread is terminated with a call to `ThreadPool::terminate()`. The thread locks a mutex, and then checks if there is any work. If no work is currently present, it notifies that it is done, and waits for new work. When new work is scheduled, or or the queue is not empty, it first checks if is should return, allowing it to join with the main thread. Otherwise it pops a function from the queue, and executes it, before restarting the loop.

```
void ThreadPool::worker_thread_loop() {
    while (true) {
        std::function<void()> Job;
        {
            std::unique_lock<std::mutex> lock(Queue_Mutex);
            if (working_threads_ == 0) {
                condition_done.notify_one();
            }
            condition.wait(lock, [this] {return !JobQueue.empty() ||
    terminate_; });
            if (terminate_) {
                return;
            }
            Job = JobQueue.front();
            JobQueue.pop();
        }
        Job(); // function<void()> type
          working_threads_;
    }
}
```

Listing 4.14: worker_thread_loop() code.

The variable `working_threads` is used to keep track of the amount of work in the queue which still is not dot done. When the value is 0 the threads signal to the main thread that it can continue to the next phase in the main loop.

If `terminate_` is set to true, the thread returns, stopping execution. This is not necessary for running the game, but windows opens an error box when stopping the program, if the threads are allowed to continue when closing the game.

4.4.2 Main loop

The main loop described in section 4.2.1 is modified to account for the `ThreadPool`.

```
auto& game_state = get_game_state();
for (auto i = 0; i < game_state.size(); i++) {
  auto* object = game_state[i];
  thread_pool_.add_job(std::bind(&GameObject::update, object));
}
{
  std::unique_lock<std::mutex> lock(thread_pool_.queue_mutex);
  thread_pool_.condition_done.wait(lock, [] {return thread_pool_.job_queue
    .empty() && thread_pool_.working_threads_ == 0; });
}
```

Listing 4.15: Updates main loop using the threadpool.

Listing 4.15 shows how instead of iterating over update functions on `GameObjects`, all update functions are scheduled in the threadpool. The game engine then waits until the queue is empty and all tasks are done.

4.4.3 Problems with making the game engine parallel

A problem encountered while developing the parallel version of the game engine was several exceptions thrown far down in the C++ libraries. Sometimes the error would be thrown from the main thread, other times a different error would be thrown from a thread in the thread pool. Exceptions include `access violation exception`, `the executable as encountered a breakpoint`, and `illegal address`. The exceptions were only thrown while running the game in debug mode. The problem was revealed to be caused by iterator invalidation, and the code was changed to account for modifying collections, to resolve the errors.

Another problem encountered with the introduction of the `ThreadPool` was synchronization with the main thread. The first implementations only checked if the queue was empty before continuing to the next step, but it resulted in the game engine starting to delete game objects, while some threads were still not done, causing iterator invalidation or null pointer exceptions.

4.5 Parallel game

In this section we discuss how the game was changes to implement the `ThreadPool` and parallelism. At this point in development we also performed some small tests to determine if the game was suitable for parallelism, or if we needed to go back and redesign the game.

The only necessary change to the game is the introduction of mutexes and locks in critical regions, and to introduce the thread pool to thread the game.

```
1 class BckgroundElement {
2     public:
3         std::mutex velocity_modify_mutex;
4         ...
5     void update() {
6         std::unique_lock<std::mutex> lock(GameObject::
    velocity_modify_mutex);
7         // Update logic
8     }
9 }
```

Listing 4.16: Locking of background element in parallel game implementation.

Listing 4.16 shows an example of how mutexes are used to ensure mutual exclusion for the critical regions in the code. Mutexes are introduced to the player, boss, enemies, spawner, bullets, and background elements.

4.5.1 Performance comparison

In this subsection, we describe the performance difference between the sequential and parallel implementations. We do this to argue that the game is usable in a multithreaded context and that the game engines, and sees a speedup from it, as we otherwise would need to redesign the game. We consider it a valid approach to change the game if the results do not meet our expectations since it is known that commercial games benefit from multiple cores [1].

Small 30-60 second runs of the game with logging to the console show that the parallel implementation of the game performs 1.4 times better compared to the sequential implementation. By modifying the number of background elements, an even better performance can be achieved. From the performance improvements in the parallel version, we are confident that features in the game can be used for testing, and that the game can work on both the sequential game engine, and both multithreaded game engines.

We perform full tests and comparisons of the engines in chapter 5, while this only is used as an indication for our work.

4.6 Aggregator Engine

In this section we discuss the changes and additions made to the game engine to include the aggregator. Focus is on the implementation of the wrapper and delta classes, which are used for connection the game with the features of the aggregator.

4.6.1 Delta

Deltas are types which contains the updates for variables for the duration of an iteration. Delta types contains internal variables for all the different operations which can be performed as explained in section 3.4.1.

```
1  void IntDelta::reduce(void* pointer)
2  {
3    const auto target = static_cast<IntDelta*>(pointer);
4    additions_ += target->additions_;
5    multiplications_ *= target->multiplications_;
6    assign(target->assign_, target->assign_priority_);
7  }
8
9  void IntDelta::merge()
10 {
11   if (assign_priority_ > 0)
12   {
13     original_value_->set_value(assign_);
14     return;
15   }
16   original_value_->set_value(original_value_->get_value()*multiplications_
       + additions_);
17
18 }
```

Listing 4.17: IntDelta reduce and merge function.

Listing 4.17 shows the delta functions for reducing and merging the delta into the variable it is related to. Reduction is done using a divide and conquer strategy, combining deltas from each thread into a single collection. Merge then applies the delta to the related variable, either making an assignment if a priority has been set, or modifying it based on relative operations otherwise.

4.6.2 Wrapper

Wrappers are types which are used in code instead of normal values when the aggregator should affect them. The wrapper contains the original value which it affects, and functions and operator overloads which modifies the delta value.

```
void IntWrapper::operator+=(const int & rhs)
{
  auto* pointer_deltas = &MasterEngine::LibAggregator::ThreadPool::
                         deltas[std::this_thread::get_id()];
  if (pointer_deltas->find(this) != pointer_deltas->end())
  {
    auto delta = static_cast<IntDelta*>(pointer_deltas->at(this));
    delta->addition(rhs);
  }
  else
  {
    auto new_delta = new IntDelta{ this };
    new_delta->addition(rhs);
    pointer_deltas->insert(std::pair<void*, BaseDelta*>(this, new_delta));
  }
}
```

Listing 4.18: Example of a wrapper operator overload.

Listing 4.18 shows how the overload of += modifies the delta of the wrapper. First the wrapper looks through the list of deltas for the current thread. If it finds the delta, it adds the new value to the current value in the delta. If it does not find the delta, it creates a new delta, and adds it to the list of deltas for the thread.

4.6.3 Main loop

The main loop is modified to account for the introduction of reduction and the aggregator, as shown in figure 3.5.

```
auto& game_state = get_game_state();
for (auto i = 0; i < game_state.size(); i++) {
  auto* object = game_state[i];
  thread_pool_.add_job(std::bind(&GameObject::update, object));
}

for (auto i = 0; i < collision_game_objects_.get_value().size(); i++) {
  auto object = collision_game_objects_.get_value()[i];
  thread_pool_.add_job(std::bind(&GameObject::collision_check, object));
}

Input::process_input();
Renderer::render();

//Wait for Threadpool to finish all jobs
...

///Aggregator step start
auto reduction_steps = thread_pool_.thread_count;
while (reduction_steps != 1)
```

```
{
  const auto floor = reduction_steps / 2;
  const auto ceil = (reduction_steps + 1) / 2;

  for (unsigned i = 0; i < floor; i++) {
    const int index = i + ceil;
    thread_pool_.add_job(std::bind(&GameEngine::merge_list,
    ThreadPool::threads_ids[i], ThreadPool::threads_ids[index]));
  }

  //Wait for Threadpool to finish all jobs
    ...

  reduction_steps = ceil;
}
for (const auto delta : ThreadPool::deltas[ThreadPool::threads_ids[0]]) {
  auto object = delta.second;
  thread_pool_.add_job(std::bind(&BaseDelta::merge, object));
}
//Wait for Threadpool to finish all jobs
...
```

Listing 4.19: Snippet of the game engine for the aggregator.

Listing 4.19 shows the updated game look, with some details omitted. Functions are added over two rounds depending on if they require input or not, on either side of the call to `Input::process_input()`. After the frame has been rendered the loop waits for synchronization similar to how it is done in the parallel game engine. The aggregator performs a reduction of deltas on all threads by using divide and conquer, scheduling each iteration in the `ThreadPool`, until a single collection contains all the deltas. After reduction, the aggregator schedules merge for all deltas in the `ThreadPool`. The loop waits for synchronization after merging.

4.7 Aggregator Game

In this section, we describe the changes to the game to utilize the aggregator game engine. The only classes which require changes to use the aggregator are the player and background elements, which both are changed such that velocity and position is wrapped instead of used directly.

```
sf::Vector2f position = GameObject::get_position();
sf::Vector2f velocity = GameObject::get_velocity();
sf::Vector2f force = {0,0};
if (position.x < 0 && velocity.x < 0)
{
  force.x =2*  std::abs(velocity.x);
}
```

```
8  else if (position.x > Renderer::get_window_size().x    size_
9          && velocity.x > 0)
10 {
11   force.x  =2*   std::abs(velocity.x);
12 }
13 if (position.y < 0 && velocity.y < 0)
14 {
15   force.y = 2* std::abs(velocity.y);
16 }
17 else if (position.y > Renderer::get_window_size().y    size_
18         && velocity.y > 0)
19 {
20   force.y  = 2* std::abs(velocity.y);
21 }
22 GameObject::velocity_ += (force);
```

Listing 4.20: Code for the backgroundelement update.

Listing 4.20 shows the updated logic for updates in background elements. Instead of assigning the force, it is done as a relative operation with +=. This change is important, as the test in section 5.5 is designed around this behavior.

4.8 Delimitation

In this chapter we have described the implementation of each of the three game engines: sequential game engine, parallel game engine, and aggregator game engine, and the game which is implemented in each of the three game engines.

Both the parallel game engine and aggregator game engine are implemented by modifying the sequential game engine, keeping the three game engines very similar to each other.

In the next chapter we test the three game engines, by using two of the features designed for the game: background elements, and gravity wells.

5 | Test

In this chapter we present the test strategy for the game engines we have implemented in chapter 4, and the test results for three different test types performed on four different computers. In the test strategy in section 5.1 we present what we test and why, and how we perform the tests.

5.1 Test strategy

In this section, we describe how we test the different game engines, and why we perform the tests that we do. We also describe how we perform the different tests. The goal of the tests is to generate results which can help determine if the aggregator is a viable alternative to lock-based parallelism. We also describe why we include the three different game engines we have implemented in the tests.

In this project we have designed three different game engines that are described in chapter 3, and with implementation described in chapter 4. We perform three different tests on each game engine, which cover single-core overhead, work-load, and handling of critical regions, all of which we explain later in this section.

The three game engines are:

- Sequential game engine, operating on only a single thread.
- Parallel game engine, operating on multiple threads using a locking model.
- Aggregator game engine, operating on multiple threads using the aggregator.

The sequential game engine is included in the tests as a baseline to measure performance against for the other game engines. The game engine runs sequentially with no use of locking or concurrency, and can be used to show the overhead introduced with the multithreaded game engines.

The parallel game engine is introduced as a traditional example of how to use concurrency and mutual exclusion by using mutex locks. In section 2.1 we mention that the aggregator

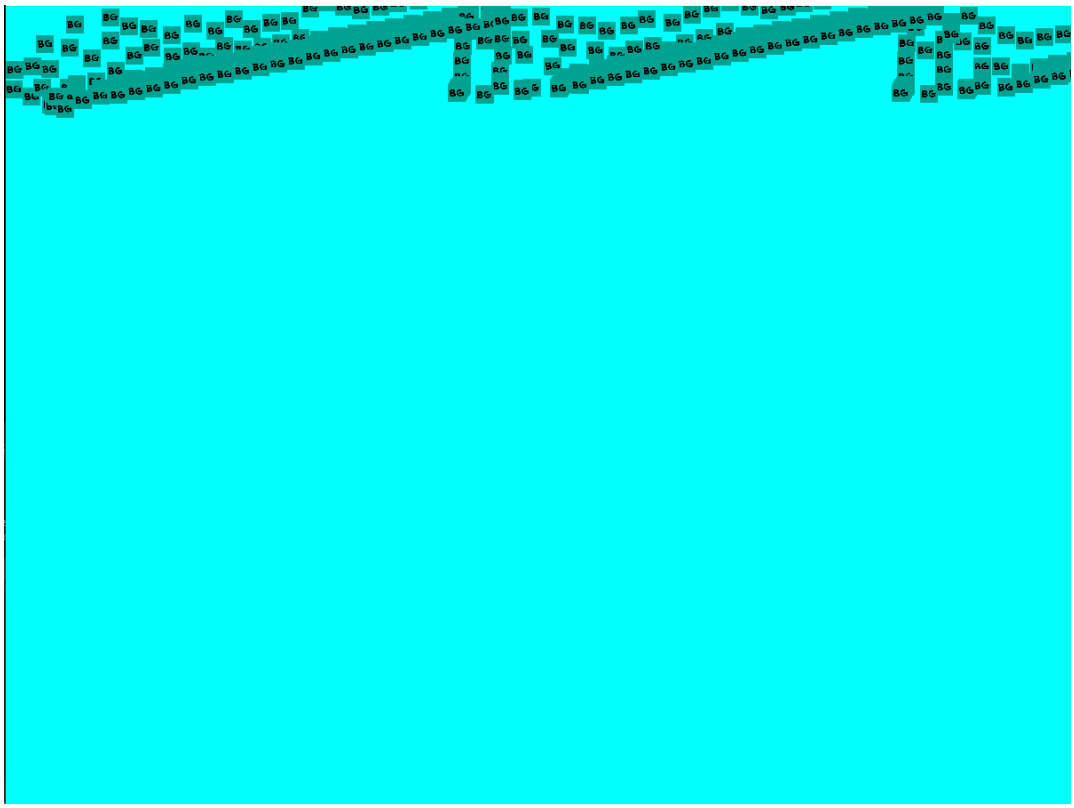


Figure 5.1: Background elements moving down from the top during game startup.

does not use locks to prevent concurrent access to critical regions, but instead keeps track of updates to variables internally. Through the tests we can determine if our lock-free approach with the aggregator can outperform the traditional approach with locks.

Multiple tests will be run on all three game engines, on four computers, to gain a better understanding of how the engines perform on different hardware configurations, with different test sizes.

We conduct three different tests to determine how the different game engines perform compared to each other. The first test is a single-core test to determine the overhead of the multithreaded game engines compared to the sequential game engine. The second test tests how the engine performs on different sizes of workload. The third test tests how well the engines perform in cases that make excessive use of critical regions.

The y-axis for all figures of test results, such as figure 5.2, is the frame rate, which we use for all comparisons. The x-axis for the single-core test and background elements tests are the number of background elements used in the test. The x-axis for the gravity wells test is the number of gravity wells used in the test. All figures present results where the average frame rate on the y-axis is Log_{10} values. We have chosen to run each test for 60 seconds,

as each test is a full system test which runs long enough that small background processes can cause temporary changes in frame rate, even when most processes have been killed. A test has a startup of five seconds before collection of test results starts, where the frame rate of the game is allowed to stabilize from spawning background elements. Background elements are spawned at the top of the screen, and take a few seconds to spread out on the screen, during which the frame rate is lower. The lower frame rate at the start of a test is caused by background elements overlapping when first instantiated, causing additional collision checking. Figure 5.1 shows how background elements move in from the top, and many elements still overlap each other.

In section 3.5 we designed a game called CaptainEverything, with a variation for each game engine, based on a similar game we designed in [5, p59-p60], and called them `CaptainEverythingSequnetial`, `CaptainEverythingParallel`, and `CaptainEverythingAggregator`. The implementation of the games is explained in chapter 4. The game implementation shows that it is possible to create a simple and playable game which is identical in all three game engines. The game has gameplay features which makes it unsuitable for the tests that we perform in this chapter. Everything in the foreground of the game is removed: the player, enemies, and the boss. While the removed elements all are logic, it is hard to modify and scale for testing purposes, while background elements, and gravity wells are more suited for the scales we need for the tests, as many different test sizes are required. The game is also altered such that the number of background elements and gravity wells can be specified with optional command line arguments. The game is also provided with the text to use as the window name, as it makes it easy to keep track of the current test number, without the need of interaction with the test system while it is running. When first running the tests, we observed strange behavior in some of the results. After an analysis we found that the problem was caused by the GPU being at 100% load when rendering several gravity wells. Gravity wells are transparent sprites, which cause the cost of rendering multiple of then on top of each other to explode. Rendering of the gravity wells was disabled to prevent the GPU from throttling those tests. The rest of the screen is rendered as normal, as no other features cause problems which result in the GPU throttling the system. Results from each test are logged in a file, which logs total time, frames, average frame rate, and startup time.

For the first test we expect the sequential game engine to perform best of the three game engines, as it has no overhead introduced for concurrency, while both multithreaded game engines are expected to perform similarly, but worse than the sequential game engine. From the test we are interested in finding out how much worse they perform. The second test is about workload, and we expect the parallel engine to perform the best, but for the aggregator to be close, with the sequential engine lacking behind depending on the number of threads. The naive estimation would be a performance increase equal to the number of cores on the computer, but due to the overhead, we estimate the numbers to be a little lower in performance increase, due to Amdahl's Law [8]. The third test makes heavy use of critical regions, so we expect the aggregator engine to perform at least equal to the parallel game engine, while the sequential game engine lacks behind. We expect the aggregator to perform based on the number of threads, with the parallel game engine a little slower due to locking.

System	Spec
Laptop #1	i7-6500U (2.5GHz/3.1GHz), Integrated Graphics, 8GB DDR3
Laptop #2	i7-7500u (2.70GHz/3.50GHz), Nvidia GeForce 940MX, 16GB DDR4
Desktop #1	i7-7700k (4.2GHz, 4.5GHz), NVidia GeForce GTX 970, 16GB DDR4
Desktop #2	i7-3770 (3.40 GHz/3.90GHz), NVidia GeForce GTX 970, 24GB DDR3

Table 5.1: Test systems.

We have chosen the three tests for the game engines as they test the core features relevant to the aggregator: The overhead, the ability to handle workloads, and the ability to handle critical regions. Other tests we could perform could be mixed load with background elements and gravity wells, or small variations to the aggregator, where we change parts such as reduction or the scheduler, but those had to be omitted due to development time. The runtime for each full test is also so long that testing time could become a problem, but we discuss how testing times can be reduced in section 7.2.

5.2 Test systems

In this section we describe the specifications of each of the four computers which we test on. Different hardware can have vastly different performance in computer games [32], and as such, we test on 4 different computers: two laptops, and two desktops. The tests conducted in [4] where done on only a single system, but as mentioned in [32], it does not represent all computers. The CPU and GPU used in the test is not consumer hardware [10, 14], as the hardware is geared more towards enterprise usage.

The four hardware configurations presented in figure 5.1 are all very different configurations, and with CPUs from different generations where performance increase from generation to generation [31]. In section 5.1 we described how we removed rendering of gravity wells to avoid the GPU bottle-necking the rest of the system. As the GPU is only a very small part of the final system after the optimization, the CPU is the critical factor in the tests that we perform.

Elements	Parallel	Aggregator
30	8,6% - 65,8%	2,3% - 27,0%
300	85,4% - 92,3%	5,1% - 29,1%
1000	118,0% - 128,5%	12,1% - 83,7%

Table 5.2: Percentage-wise performance range compared to sequential implementations for single-core test.

5.3 Single-core overhead

In this section, we describe the single-core test and the results of the test. The purpose of the single-core test is to measure the overhead of the parallel and aggregator game engines compared to the sequential game engine when limited to one logical core.

In the overhead test performed in [4], a single test for overhead with a single core was performed, using a test size of 800 elements. In section 5.2 we mentioned how computer games perform different on different hardware configurations. To gain an understanding of the single-core overhead of the parallel and aggregator engine compared to the sequential game engine, we perform a test with three different test sizes to gain a better idea of the overhead, and how it changes over different work loads.

We perform three tests with 30, 300, and 1000 background elements. A test with 0 background elements in unsuitable for a test comparison, as the overhead introduced for thread safety only is present with work load.

```
1 start \wait \affinity 4 CaptainEverythingX.exe Y 0 1 <title>
```

Listing 5.1: Command for running game under single-core test.

The tests were run with the command in listing 5.1 where `affinity` specifies which logical core the operating system should limit the program to, by using a bit-mask, which for the value 4 indicates that the program is limited to logical core #3. `Wait` indicates that `start` waits for termination before returning control to the test runner. The `X` in `CaptainEverythingX.exe` is replaced with the engine which should be used for the test; for example `CaptainEverythingAggregator.exe`. `Y` is the number of background elements to spawn for the test. The 0 and 1 arguments in the command tells the game to spawn 0 gravity wells, and only allocate for one logical core in the threadpool.

Each of figure 5.2, figure 5.3, figure 5.4 and figure 5.5 shows that with a sufficiently high number of background elements, the parallel game engine performs better than the sequential game engine. For all four test systems it is apparent that the aggregator performs the worst.

Table 5.2 shows the range in performance difference for the different engines when working on different sizes of data. If an game engine has a performance of 100 % it indicates that it

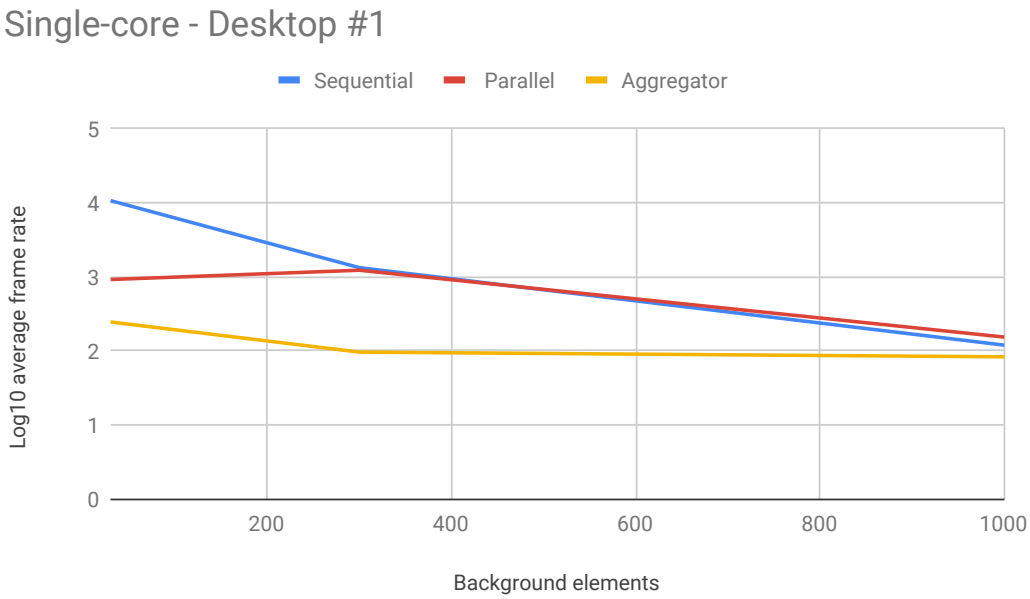


Figure 5.2: Single-core performance for Desktop #1.

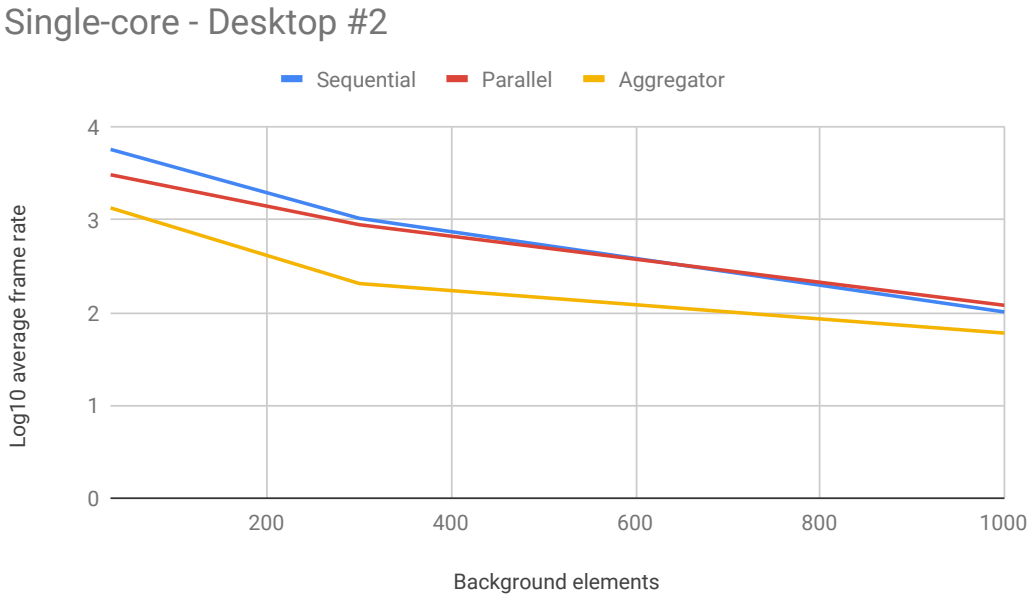


Figure 5.3: Single-core performance for Desktop #2.

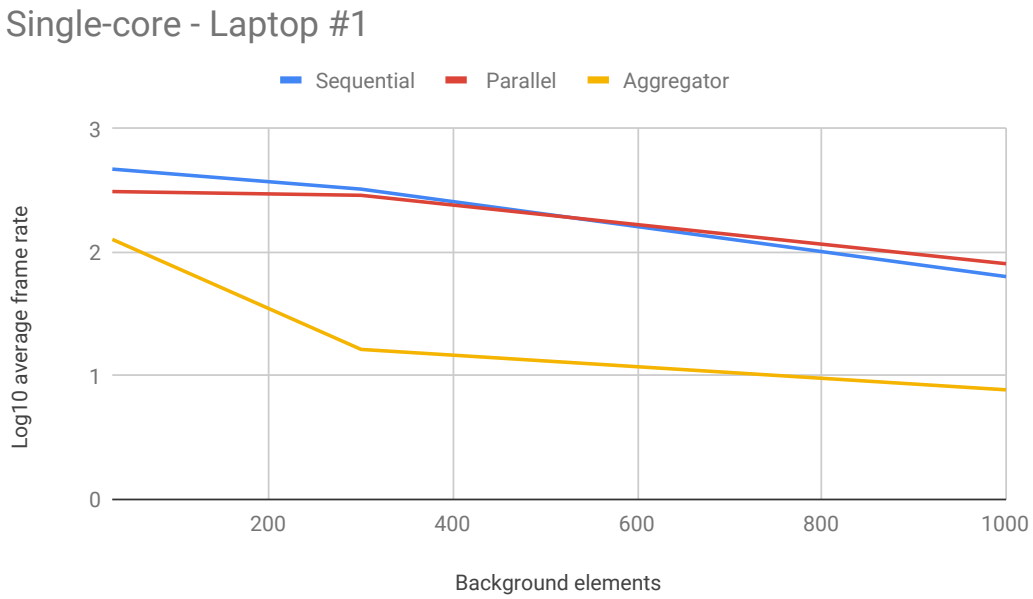


Figure 5.4: Single-core performance for Laptop #1.

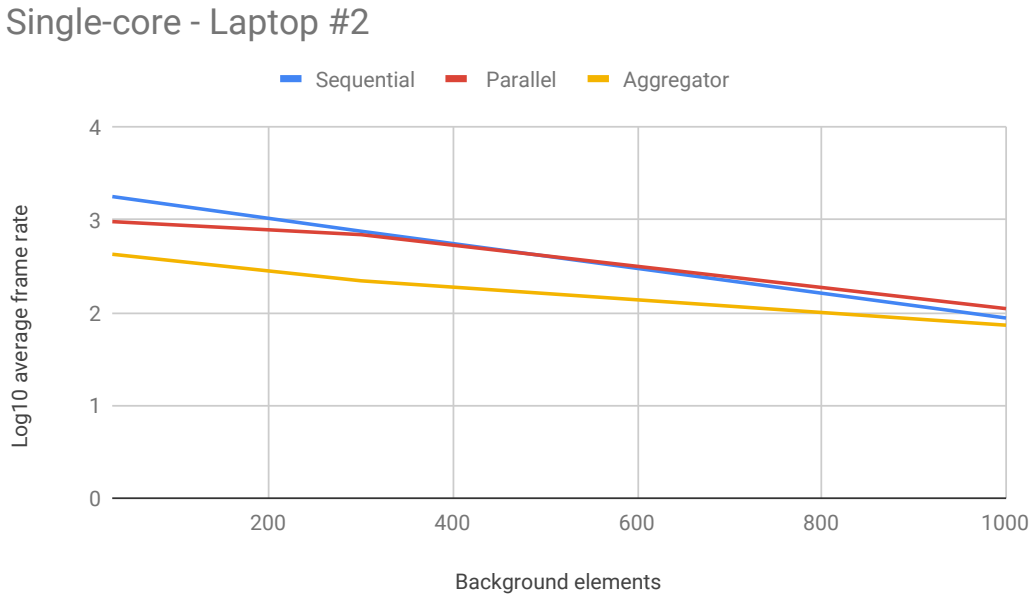


Figure 5.5: Single-core performance for Laptop #2.

performs equally to the sequential game engine for a value. The values shown in the table is the range in which the engines performs. We present the numbers as such, as it shows how big the difference can be for the same games and sizes on different systems. We can see that while the parallel game engine worst case performs poorly on a small number of background elements, out performs better than the sequential game engine when processing a sufficiently high number of background elements. The differences between the two engines are locking in the parallel game engine, and the thread pool where work is scheduled from. We assume that CPU pipe-lining and certain thread-specific optimizations are the cause of the speedup, as the sequential game engine otherwise should be the faster engine.

Another observation is that Desktop #1 sees in increase in frame rate from 30 to 300 background elements for the parallel game engine. We are unsure what could be the cause of the increase in frame rate, as the increased work load should lead to a lower frame rate. Worst case performance for the aggregator game engine is low for all test sizes. It performs closer to the sequential game engine with more background elements, but unlike the parallel game engine it does not out-perform the sequential game engine within the work loads of the tests.

As described in section 5.1 we expected the sequential game engine to perform better than the parallel game engines. From the test results we determined that the parallel game engine out-performs the sequential game engine with a high enough number of background elements. The aggregator performs worse, but following the curve, it is possible that it would pass the sequential engine with enough background elements.

5.4 Background elements

In this section, we describe the background elements test and the results of the test. The purpose of the background elements test is to measure how the different game engines perform under different sizes of workload, when not limited to a single logical core. Two of the three game engines we test are multithreaded, but we want to investigate how well they handle a work load, and if the aggregator game engine or parallel game engine performs better under different workloads.

We perform four tests with 30, 300, 1000, and 2000 background elements. A test with 0 background elements in unsuitable for a test comparison, as the overhead introduced for thread safety only is present with a work load. We added a test for 2000 background elements in addition to those in the single-core test, as the multithreaded game engine implementations still has a frame rate in the 100's at 1000 background elements. While we could push the tests until the frame rate goes into single-digits, it would result in a loss in precision, and lose real-world relevance as the frame rate goes further beneath 60 fps.

```
1 start \wait CaptainEverythingX.exe Y 0 0 <title>
```

Listing 5.2: Command for running game under background elements test.

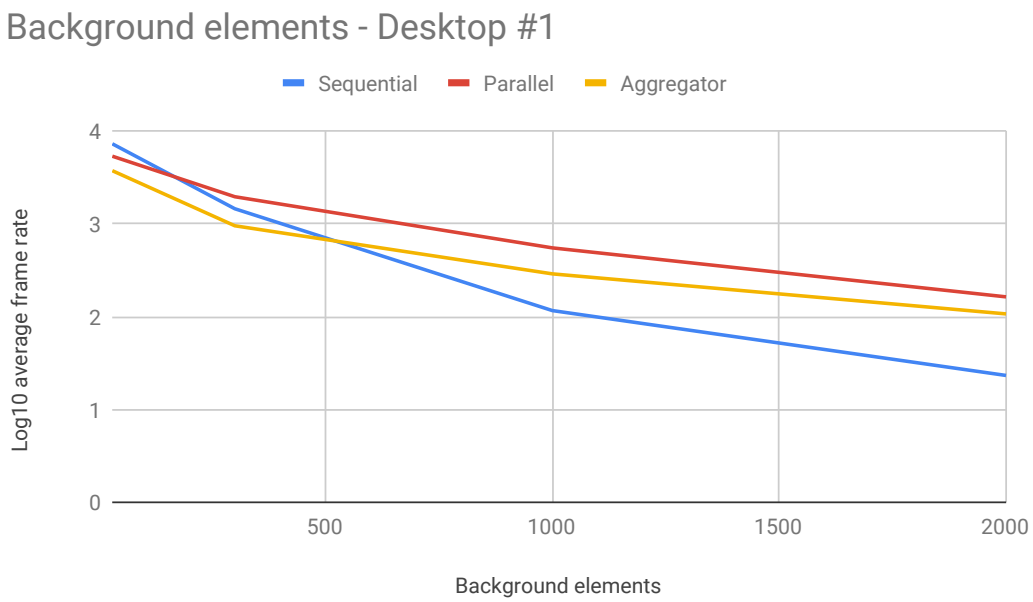


Figure 5.6: Background elements performance for Desktop #1.

Each test is run with the command in listing 5.2, which is mostly similar to the command for single-core test explained in section 5.3, except that the number of threads the threadpool should allocate for is set to zero, which allows the game to use the number specified by the operating system though a call to `std::thread::hardware_concurrency()`.

Each of figure 5.6, figure 5.7, figure 5.8, and figure 5.9 shows that the sequential game engine performs the best on few background elements, but is overtaken by the parallel game engine as the number of background elements increase. The aggregator never performs better than the parallel game engine, but passes the sequential game engine between 300 and 1000 background elements. On laptop #1, the aggregator just about passes the sequential game engine at 300 elements with a 99% performance compared to the sequential game engine.

Similar to the parallel game engine for Desktop #1 in the sequential test in section 5.3, both multithreaded game engines increase in frame rate from 30 to 300 background elements for Laptop #2. We have no real assumption as to why this could be the case, as we would have expected the frame rate to always decrease as the work load increases.

Table 5.3 shows the range in performance difference for different game engines as also explained in section 5.3. From the performance, we can observe that the sequential game engine performs the best at 30 background elements. The aggregator game engine's best performance is close to the sequential game engine at 99,2% for 300 background elements. The aggregator game engine outperforms the sequential game engine by 1000 background elements, and ends with performing 4 times better than the sequential game engine at 2000

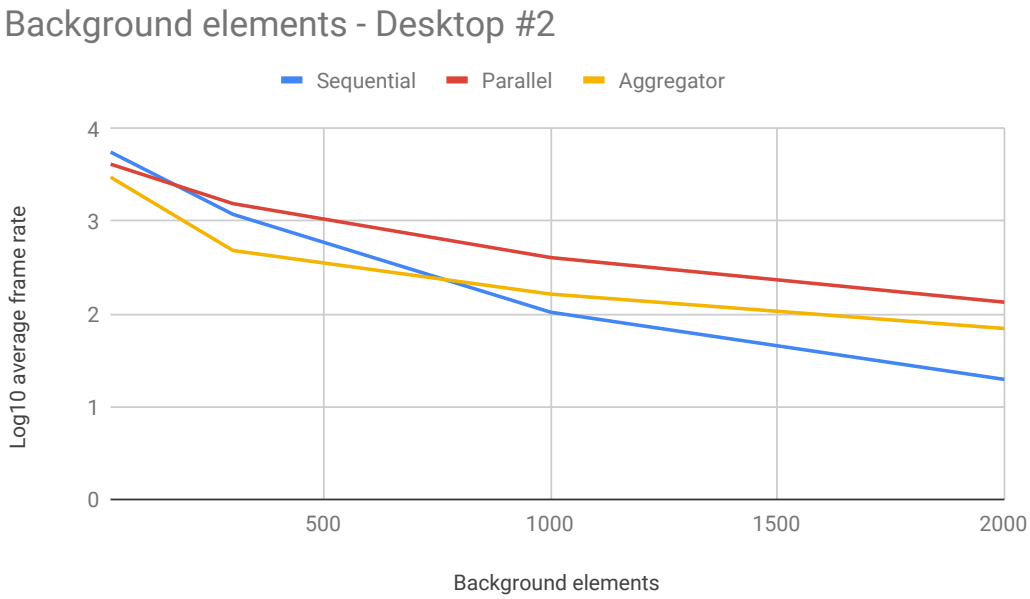


Figure 5.7: Background elements performance for Desktop #2.

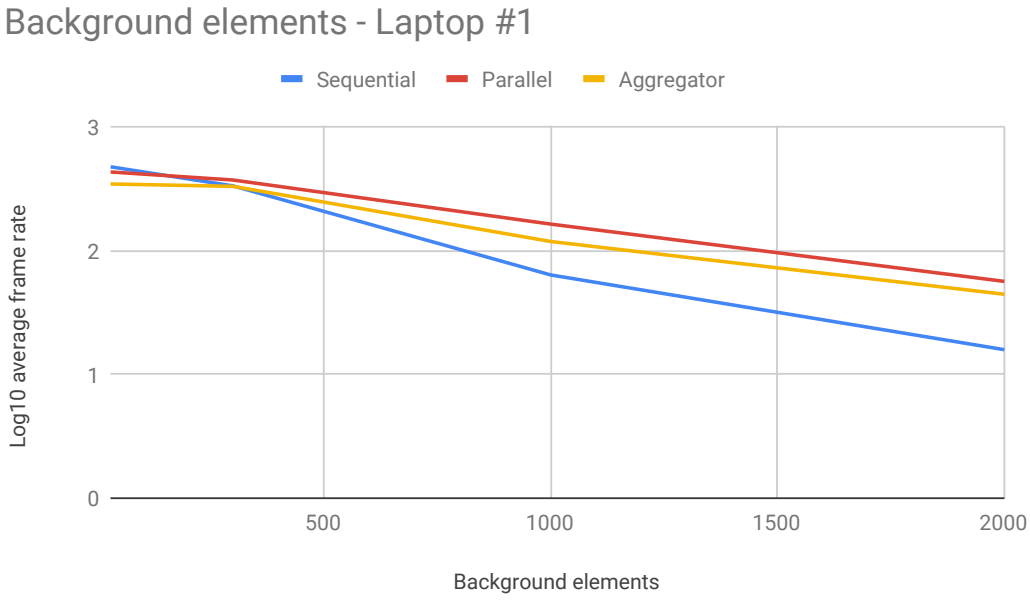


Figure 5.8: Background elements performance for Laptop #1.

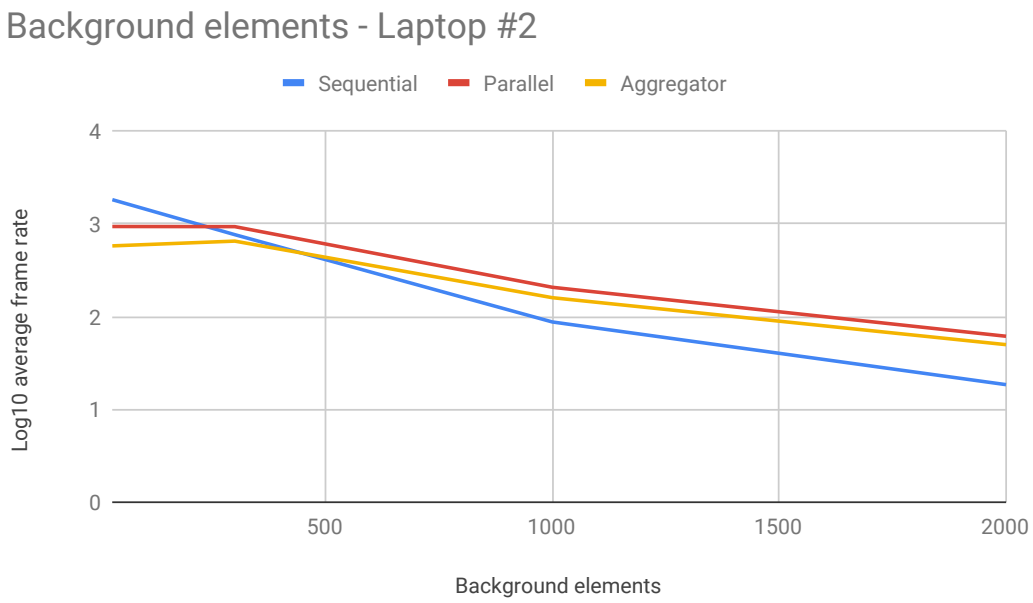


Figure 5.9: Background elements performance for Laptop #2.

Elements	Parallel	Aggregator
30	51,4% - 90,8%	31,7% - 72,7%
300	111,6% - 134,8%	40,7% - 99,2%
1000	236,6% - 474,6%	156,9% - 248,9%
2000	332,6% - 704,2%	270,0% - 464,1%

Table 5.3: Percentage-wise performance range compared to sequential implementations for background elements test.

background elements, while the parallel game engine performs just over 7 times better than the sequential game engine at the best test system.

As described in section 5.1 we expected the parallel game engine to perform the best, but for the aggregator to perform close to it. From the numbers in table 5.3 we can determine that as the number of background elements increase, the advantage to the parallel game engine increases further, with the worst increase being $332,6\%$ for a laptop with a dual-core and hyper-threading, and up to $704,2\%$ for a quad-core with hyper-threading. Meanwhile we observe that the aggregator lacks behind compared to the parallel game engine, where the performance difference between them can go up over 100%. While we assume that the graphs cross at some point, it will be at frame rates so small that it would be useless for actual games.

5.5 Gravity wells

In this section, we describe the gravity wells test and the results of the tests. The purpose of the gravity wells test is to generate a lot of updates in critical regions.

We perform a total of 14 tests using gravity wells. In the initial plan we planned to have tests for 0, 1, 5, 10, 25, and 100 gravity wells, but found that we still had high enough frame rates that we could increase the number of gravity wells to also include 250, 300, 350, 400, 450, 500, 550, and 600. We initially assumed that 100 would be sufficient as the gravity wells are set up to fill the entire screen, and as such every gravity well affects all background elements in the test. Each test is limited to 100 background elements, as the test needs an object which can be influenced in sufficiently large quantity that the difference is visible, and small enough that background objects does not become the primary factor to the performance of the test.

```
1 start \wait CaptainEverythingX.exe 100 Y 0 <title>
```

Listing 5.3: Command for running game under gravity wells test.

The tests were run with the command in listing 5.3, as explained in section 5.3. The difference is that all commands are run with 100 for background elements, where `Y` is changed based on the number of gravity wells to spawn.

Each of figure 5.10, figure 5.11, figure 5.12, and figure 5.13 shows that the sequential game engine performs the best with few gravity wells, but eventually gets passed by the parallel game engine for both Desktop #1 and #2. For laptop #1 the parallel game engine reaches performance similar to the sequential game engine until 300 gravity wells, after which it moves down towards the performance of the aggregator game engine. Laptop #2 sees both multithreaded game engines increasing in performance with a few elements, but none of the graphs ever cross. For both desktops we observe that the sequential game engine and

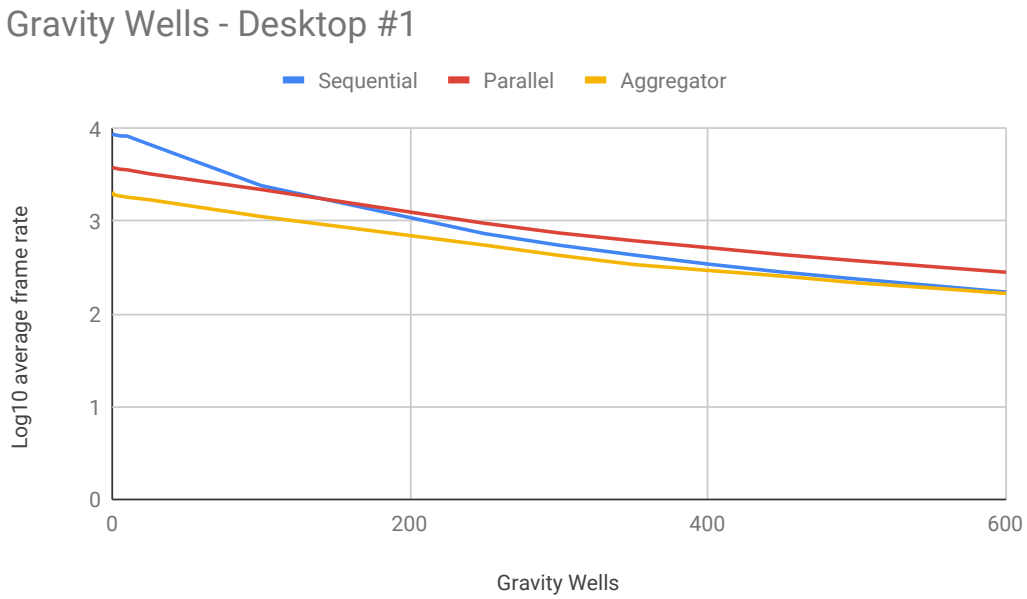


Figure 5.10: Gravity wells performance for Desktop #1.

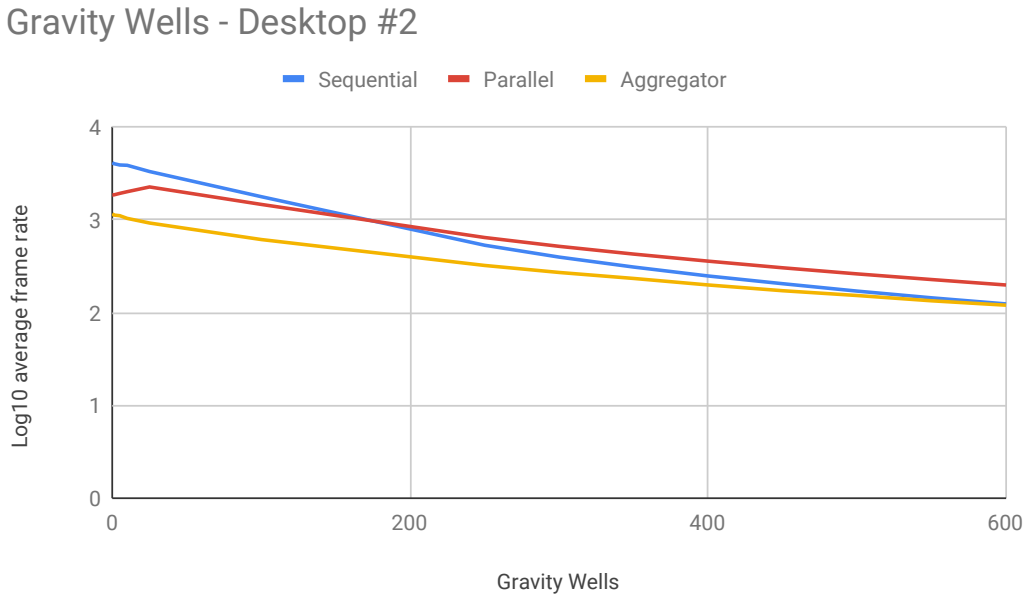


Figure 5.11: Gravity wells performance for Desktop #2.

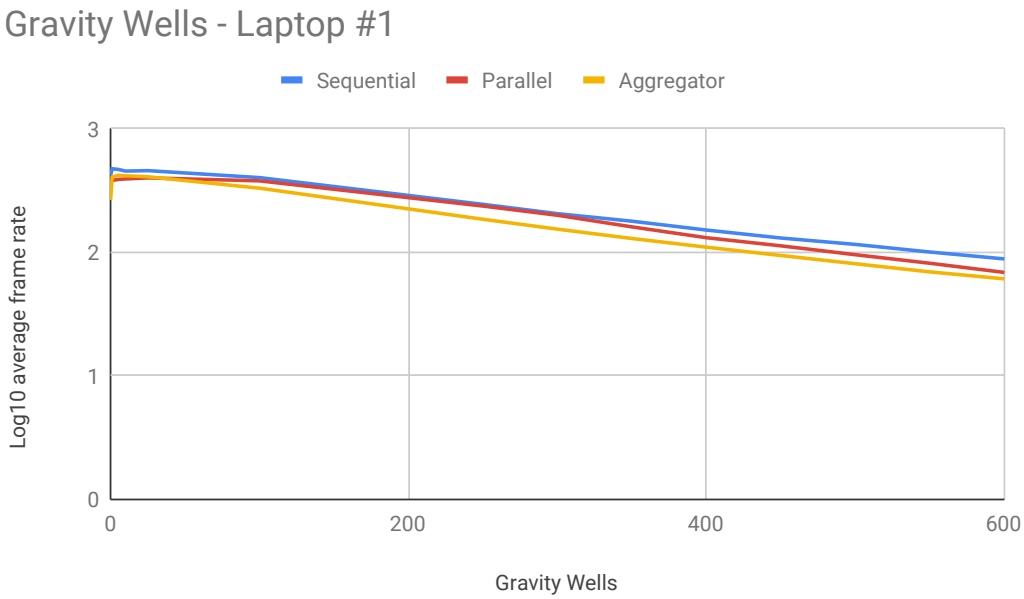


Figure 5.12: Gravity wells performance for Laptop #1.

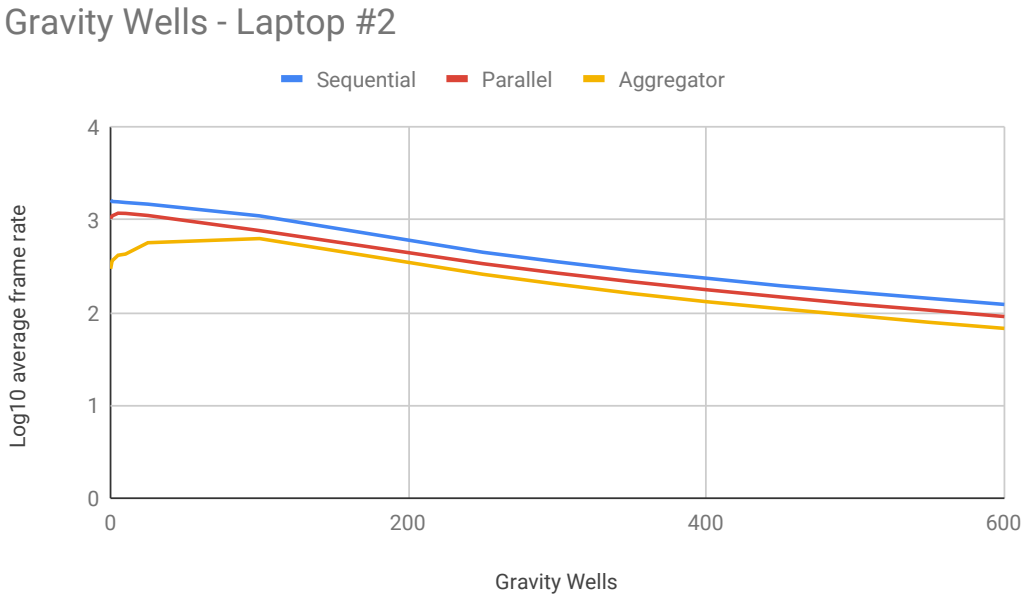


Figure 5.13: Gravity wells performance for Laptop #2.

Gravity wells	Parallel	Aggregator
0	43,7% - 89,1%	18,1% - 63,1%
1	43,6% - 80,4%	22,4% - 85,7%
5	43,7% - 83,0%	22,5% - 89,6%
10	43,3% - 86,4%	22,9% - 91,6%
25	48,2% - 87,4%	25,3% - 89,2%
100	69,4% - 94,1%	34,5% - 82,1%
250	75,3% - 129,3%	57,9% - 75,7%
300	75,5% - 135,5%	57,3% - 77,7%
350	75,9% - 142,4%	56,8% - 78,9%
400	75,2% - 150,5%	55,8% - 85,4%
450	75,7% - 154,4%	56,4% - 90,6%
500	74,3% - 157,4%	56,1% - 90,9%
550	74,5% - 160,0%	55,2% - 94,1%
600	74,0% - 163,6%	55,2% - 96,9%

Table 5.4: Percentage-wise performance range compared to sequential implementations for gravity wells test.

aggregator game engine meet at 600 gravity wells, and as such we can assume that the aggregator game engine will pass the sequential game engine if we continue with a higher number of gravity wells, but the parallel game engine performs better.

Table 5.4 shows the range in performance difference for the different game engines as also explained in section 5.3. From the performance difference we can see that the parallel game engine passes the sequential game engine between 100 and 250 gravity wells. For the worst test system, the parallel game engine never outperforms the sequential game engine. The aggregator game engine never passes the sequential game engine, but comes close with 96,9% for 600 gravity wells for the best test system, but the worst test system still only performs 55,2% compared to the sequential game engine.'

As described in section 5.1 we expected the aggregator to perform the best of the three game engine, but the tests have shown that it performs the worst of the three engines for all test sizes.

5.6 Problems encountered during testing

Many of the tests have given one or more problems that we need to check and verify to make sure that there was not made an error in the implementation. An example is for the single-core test where the multithreaded game engines end up being faster than the sequential game engine. We are unsure why this is the case, and have been unable to identify the cause of the performance. We assume pipelining and compiler optimizations affect the performance

with the inclusion of threads when limited to a single logical thread. We also changed the number of background elements used in the gravity wells tests. It was originally planned for the test to use 1000 background elements, but too much time was spent on collision checking between background elements, so the number was reduced to 100 background elements. In the first run of the single-core test we encountered stuttering in both multithreaded game engines. The stuttering was caused by context switching in the thread pool, as the game kept allocating threads equal to the number provided by the operating system, even though we only allowed to run on one logical thread. We removed the problem with context switching by adding a flag specifying if the game should be run with only a single core. The addition was made to both multithreaded game engines.

5.7 Delimitation

In this chapter we have tested the game engines over three different tests on four different systems. In the single-core test we found that when limited to one logical core, the multithreaded game engines were able to outperform the sequential game engine with high enough work load. In the background elements test, where the multithreaded engines have access to all cores on the system, we found that both multithreaded game engiens outperform the sequential game engine. The parallel game engine performs the best, but the aggregator also outperforms the sequential game engine with with higher work loads. In the gravity wells test, in which we perform many operations in critical regions, the parallel game engine outperforms the sequential game engine. The aggregator game engine never outperforms the sequential game engine, but on both desktop test systems the aggregator game engine performs equally to the sequential game engine on the highest work load.

In the next chapter we will discuss some of the game engine features which we excluded in chapter 1, and how they could affect the aggregator.

6 | Excluded engine functionality

In this chapter we discuss the engine features which we chose to exclude in section 1.2, as we decided in [5] that we can design a testable game engine with the aggregator without the features. All omitted features are part of commercial game engines such as Unity.

First we will discuss hierarchies and components, which are used to structure games and objects in a game world. Then we will discuss resources, and how they are used in game engines. After that we explain physics and fixed updates, and how it is used in game engines, before finally covering 2D and 3D game engines. For all features we also mention how the aggregator affects or is affected by each feature.

The game engines created in this project uses a flat hierarchy; Game objects all exist in a flat list with no further relation to each other. Game engines such as Unity have a hierarchy of GameObjects, where some GameObjetcs are child objects to other GameObjects. When a parent GameObject is moved, all child objects move accordingly. The introduction of a hierarchy has no effect on the behaviors of the aggregator.

Components are sub-objects which can be attached to objects at runtime to give them new functionality [29]. In the game engines we implemented in chapter 4, we added position, rendering and collision into the GameObject. In a game engine using components, position, renderer, and collision would be added as components. In unity, position is part of a transform, which is a required component for GameObjects, as it manages the hierarchy for a GameObject [21]. Objects such as `player` and `enemy` would also be components of a GameObject, instead of inheriting from it. With the aggregator, it would just be more types to keep track of, but additional merge rules are required for them.

The game engines include a simple resource manager, which is used to load sprites into the game at runtime. Other resources that games can need to load can include audio, shaders, and prefabs. Prefabs are objects which have been constructed and saved as a resource. An example of a prefab could be the enemies in `Captain Everything`, which we designed in section 3.5. Enemies are currently simple objects, but with the introduction of components, objects can be more complex than what is viable to manually construct each time the object is instantiated. With a prefab, an enemy is constructed and then saved as a resource.

The resource manager can then be used to instantiate the enemy prefab in a single command, only requiring the position to spawn the enemy at. Resources are loaded from the desk when needed, but can be cached in memory for faster access. Caching an entire game at startup will not always be possible, as some games can have game sizes over 60GB, and the size increase for newer games [15]. Unity also allows resources to be assigned to variables through its editor, removing explicit access to the resource manager, and optimizing resource usage at compile time [26].

In section 1.2 we limited the physics in the game engine to collision detection on rectangular colliders. Objects in real games can also be circles or polygons [22]. In a 3D setting, collision also affects cubes, spheres, capsules, and 3D polygons. Such shapes can have more complex collision detection then rectangles, which can be more computationally expensive[30]. Collision checking in the game engine currently has time complexity $O(n^2)$, where `n` is the number of game objects affected by physics collision. The complexity is so big it would be unsuitable to use in real games. It is possible to reduce the time complexity using a multi-phase approach [11], but we estimate that the time required to implement a better detection algorithm would out-weight the benefits, when limited to the scope of this project, based on prior experience with collision detection. Other features that we could include includes rigid bodies, resistance, gravity, ray tracing, and forces. We use a simple force implementation in the games developed in this project, but it is implemented in the game layer and not the engine layer. More complex features could include soft body simulation of clothes, destructible objects, and fluid simulation. Many of these features are implemented through a third-party physics engine, or by using technology directly from GPU companies [24, 7]. Physics would have no real influence on the functionality on the aggregator, but unlike with other game engines [18], everything can be executed in parallel with other `update` logic.

Game engines such as Unity also implements `fixed updates` in addition to `updates`[18, 23]. Fixed updates are executed a fixed number of times each second, in-between normal `updates`. `Fixed updates` are used when interacting with the physics system, such as rigid bodies, or when interacting with features which can be processed independently from normal `updates`. `Fixed updates` can be included in a game engine with the aggregator, but it requires a small redefinition of the aggregator game engine to work. Because of the possibility that multiple `fixed updates` are executed at one iteration if the fixed update rate is high enough. Instead the aggregator game engine reduces and aggregates at the end of each fixed update, in addition to the reduction and aggregation which is done after `input`, `update`, and `rendering`.

The game engines are created to be simple 2D game engines, but are limited to only sprite rendering. Commercial game engines offer more rendering options for 2D and/or full 3D support, and a full rendering pipeline. Even simple 2D game engines can put stress on hardware, as we also mentioned in section 5.6, where the GPU under full load for simpler systems, such as those using integrated graphics. The problem was caused by multiple layers of images with transparency, as the cost of rendering the sprites exploded as more transparent

sprites were drawn on top of each other. In section 4.1 we mentioned that we chose to use SFML as graphics library, but it is likely that even with the optimizations that we made, it does not handle a lot of work that well. A move to Vulkan would be the best alternative, as it is a new graphics API, which is set to replace OpenGL. We already discussed the use of Vulkan in [5, p10-p11].

6.1 Delimitation

In this chapter we have discussed the engine features which we chose to exclude in chapter 1. The inclusion of the features would show how the aggregator game engine would perform in a more real-world scenario. An option for testing an engine with many of the features covered in this chapter would be to include the aggregator in an existing game engine, as we further discuss in section 8.1.

In the next chapter we will discuss test results for the engine we did test, as well as the viability of using the aggregator in its current state.

7 | Discussion

In this chapter, we discuss the project and reflect over the choices we have made, and the consequences of those choices.

7.1 Design

In this section we discuss the design we have made in chapter 3, and how the decisions can have affected development and tests.

In section 2.1 we define the aggregator as immutable for the duration of an iteration, but in section 3.4 we allow input to block execution of some update functions, because they can depend on the current state of keys. The alternative would be to save the new state of each key in a delta, and then merge it at the end of an iteration like the rest of the game state. The reason we do not do this is because it would delay the effects of keyboard input until the next frame. Because the results of updates using input is pushed back one frame, so is rendering of the results of the keyboard input. In traditional game engines with the flow input => update => render, the new frame shows the interaction immediately. With the aggregator under the current implementation, displaying input on the frame is delayed by one frame. With the change proposed in this section, it is delayed by another frame, such that input now is off by two frames. The change will be unobservable at a very high frame rate, but might become visible at a lower frame rate. Testing would need to be performed to determine if it can be a cause of problems for players.

We also designed the `ThreadPool` in section 3.3 under the assumption that the language we chose would not offer similar functionality out of the box. In C# `Tasks` and `Parallel` could be alternatives for `non-blocking` and `blocking` concurrency.

7.2 Test and test results

In this section we discuss our test approach and test results. The discussion is based on our work in chapter 5, which is based on the game engine design in chapter 3 and implementation in chapter 4.

From the tests we found that the results did not match our expectations which we presented in section 5.1. For the singe-core test we predicted that the sequential game engine would perform the best, while the multithreaded game engines would perform worse, but not a big difference between the multitreaded game engines. From the test results we found the parallel game engine outperforms the sequential game engine, while the results indicate that the aggregator game engine eventually will overtake the sequential game engine as well. Both the parallel game engine and the aggregator game engine performs better than the sequential game engine even though they are implemented with threads and thread syncronization, which should result in slower execution. We speculate that some compiler optimizations and pipelining can be the cause of the performance increase for both multithreaded applications. The tests run on a single core, leading to no context switching for the parallel game engine, and thus one part of the engines without overhead. We predicted that the parallel game engine would perform the best in the background elements test, with the aggregator game engine following closely. The parallel game engine and aggregator game engine both outperformed the sequential game engine, but with different magnitudes. The parallel game engine performs better than the aggregator game engine by over 250% when compared to single-threaded performance. The aggregator was expected to perform the best in the gravity wells test, but performed the worst of the three game engines. From the results in table 5.4 we can see that the best performance is $96,9\%$, which is almost equal to the sequential game engine, and the growth points towards passing the sequential game engine. The results for few gravity wells shows that it has a quick positive and negative growth, which is caused by the difference in change of frame rate for the game engines at few gravity wells. The sequential game engine has no overhead, and the parallel game engine has locking, but can update the values directly. The aggregator has to look up delta values and update them first, which profiling have shown is a huge contributor to the slowdown for the aggregator, and which leads us to believe that the implementation of deltas are the cause.

When preparing for the tests we followed the test hints in [16], which recommends closing most external programs, and watching out for windows update. All test systems were updated and rebooted before the tests and rechecked for updates, leaving no risk for windows updating in the background. All IO is performed by the game engine either before test measurement starts, when loading the binary and resources, or after the test is done, when results are written to disk. All windows apps were shut down, and some background processes such as Microsoft OneDrive. Some restrictions were also put on Windows defender, as the test binary used for most test systems were built on a single system and then redistricuted, and as such an unknown and unsigned binary downloaded from the internet.

When performing the tests we chose to run each test for a full minute as it was a full system test when testing the game, and we wanted enough time to account for small deviations caused by windows in the background, which is apparent as there still was differences between each of the five runs for each test. As tests take one minute per run, and five seconds of startup, a full test run lasts close to six hours. The system is based on an iterative system and as such needs to run for a number of iterations for a test to cover usable results. An options would be to run a number of tests for a specific duration, and then perform the same tests with a longer duration, until we find the minimum duration which consistently produce the same results. Another option would be to count the duration which it takes to make a number of iterations, but as some of results in chapter 5 show, the results range from over 10.000 to under 10 frames per second, and and can still vastly affect the results depending on the number of iterations. We believe that either are possible options for reducing testing times, which makes it easier to run more different tests, or as we discuss below, more of the same tests.

During data processing, we attempted to create tendency lines for the results to better describe the flow and development, and to predict when lines would cross for the tests where we estimate that they will cross, but do not have test data for it. We used exponential tendency lines, and judged the usefulness of the lines by using the r^2 value, where a value closer to 1 is better. For some tests, a r^2 over 0.95 were possible, which makes for a decent line, but other results had tendency lines in the area of 0.80 which is too low for a line to be usable. A great line would need a r^2 of 0.99. If we had made more tests in-between the current test-sizes, and beyond the upper limits, more numbers would have been available, which would help providing a better tendency line.

All three game engines used in the tests are written in C++. The authors of the project have prior experience with C++ through university courses and working with Unreal Engine in [5], but no other real experience. A consequence of this is that the game engine implementations may be sub-optimal, and some implementations while similar, favor one of the three engines more than others. More experience with C++, or implementations in another language the authors have more experience with could lead to different results. During implementation we also decided to complete the implementation for one engine before starting the next, and then copy the code over to the new engine, and then modify it. It is possible that separate implementations from scratch could lead to engines which better utilize the restrictions based on which of the three game engines it is.

7.3 Choice of technologies

In this section we discuss and reflect over the choices we made when deciding which technologies to use for the game engines in section 4.1.

We chose C++ as programming language as a result of originally deciding on OpenGL as

the rendering framework of choice, because the documentation of all wrappers for C# ports of OpenGL were poor or non-existing, and creating our own wrappers would take away from the focus of the project. After choosing C++ we also tried multiple frameworks before deciding on SFML. It was the only framework with a C++ interface, which made it a better working experience for us, than writing GLEW and GLFW, which only provided a C interface. Vulkan and DirectX 12 were never really considered due to their size, and because we had a fear that they would steal too much development time, which otherwise should be used on implementing the rest of the game engines.

When we worked with C++ in [5] we often encountered problems where C++ would take more than 10 minutes each time we compiled the game, which is similar in complexity to the game we designed in this project in section 3.5. We did not encounter the same problems with C++ in this project, but instead encountered other problems with MSBuild, which were the build system we used for the game. Early in the project we decided to split the code project into multiple libraries, such that we could have one library for engine features, a library for each game engine, and a library for each game. When running the game, Visual Studio would only compile the project marked as `startup project`, omitting changes in the other libraries, leading to much frustration and debugging time. Line numbers in error messages would also be incorrect due to the mistake, causing much frustration before we became aware of the cause of the problem.

Another problem we had with C++, and which we were aware of when choosing the language, was the error messages. They would often appear 10 layers deep in the standard library, and not say anything helpful about why something in the code could not compile, or on runtime where a simple mistake such as invalidating an iterator would have to be discovered independently of the error message.

As far back while working on [5] we considered C# as the language of choice due to the authors better familiarity to it. Well into the project we became aware of a C# library called MonoGame[1], which feature-wise is equivalent with SFML.

The best part of working with C++ was the package manager we chose, VCPKG, which integrated with Visual Studio, such that we never had to touch the build system directly to add new libraries to the project. It also made it easy for us to try the different libraries before deciding on SFML, as it took a single line in a terminal, and VCPKG would compile and configure everything for us.

7.4 Immutability and determinism

One of the strengths of the aggregator is how it hides assignments to an variable, and allow for easy parallelism, but hiding assignments can also cause problems for some games. If

[1] http://www.monogame.net/

a game have two AIs where the choice of one AI can affect another AIs choice, such as invalidating it, the AIs work for the frame would be invalidated. The two ways to currently avoid the problem is to either process all related AI logic in one `update` function, but risk losing all benefit to having multiple threads, or move parts of the AI logic outside the aggregator.

One way to avoid the problems is to redesign the AI such that the outcome of one AI cannot affect the decision of another AI in the same frame, but might require changes to the functionality of game mechanics of the game the AI is part of. An alternative is to extend the existing wrappers, adding the option to peek the current value if merged.

When coming up with the aggregator, we chose to value determinism due to the complexity in many games. Testing games is a big task, and also involves many test disciplines other than just those in traditional software testing [9], and determinism can make testing for many of them harder, when a problem is difficult to reproduce, or only occur in some customer's computer configuration.

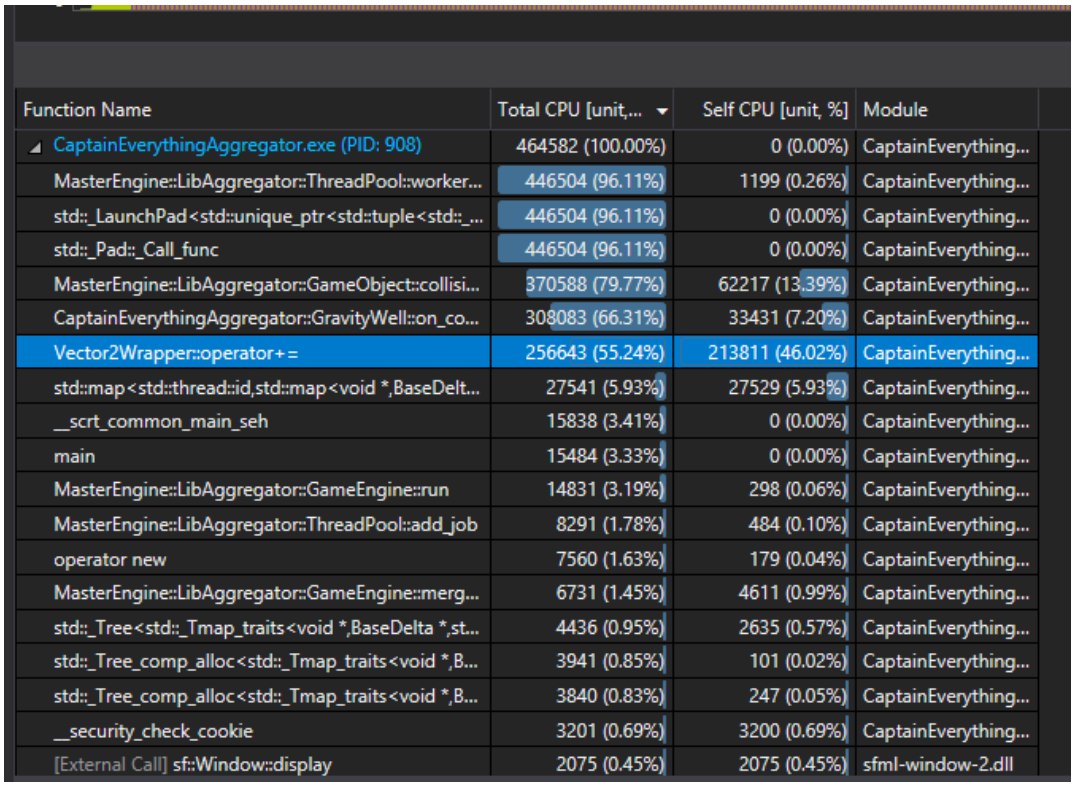
The aggregator game engine can become non-deterministic with priority assignment, if multiple assignments are made with the same priority, as the game engine only keeps the results of the first assignment, which can happen from any thread.

7.5 Improvements to implementation

This section is about the improvements that can be made to the aggregator game engine, and in extension all three game engines. The tests in chapter 5 which we discussed in section 7.2 shows that the aggregator does not perform as well as we expected. We have run the aggregator game engine under the Visual Studio Performance Profiler, where we look at CPU usage. Figure 7.1 shows the CPU usage different functions in the aggregator game engine. We can observe that worker threads, collision detection, and collision handling account for the majority of the total CPU time, but the += operator also accounts for 55% of all CPU usage.

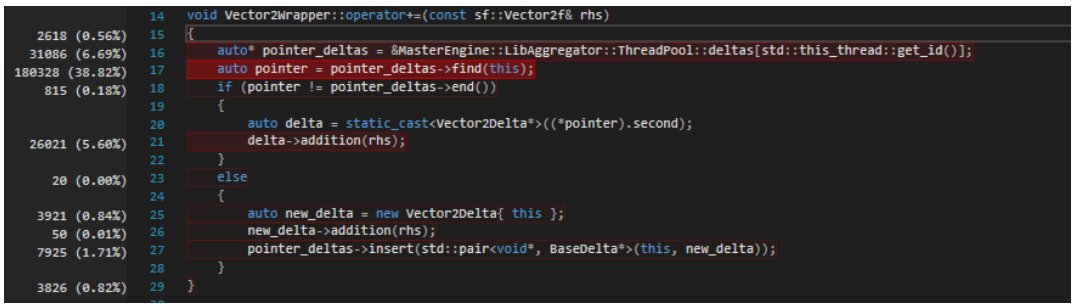
If we investigate the function, we can see that 38% of all CPU time is spend finding the relevant delta for the wrapper for the current thread. We iterate over all deltas to determine if an delta already exist for the wrapper, and either get it, or create a new delta and add to the list. We store all deltas in a global collection such that it is simple to iterate over in the reduction phase and aggregate phase.

The simple solution to the problem would be to keep a local reference to the delta in the wrapper, such that it is faster to loop up at runtime. We did not implement this fix for retesting due to the time required for running tests, as discussed in section 7.2.



Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
CaptainEverythingAggregator.exe (PID: 908)	464582 (100.00%)	0 (0.00%)	CaptainEverything...
MasterEngine::LibAggregator::ThreadPool::worker...	446504 (96.11%)	1199 (0.26%)	CaptainEverything...
std::_LaunchPad<std::unique_ptr<std::tuple<std::_...	446504 (96.11%)	0 (0.00%)	CaptainEverything...
std::_Pad::_Call_func	446504 (96.11%)	0 (0.00%)	CaptainEverything...
MasterEngine::LibAggregator::GameObject::collisi...	370588 (79.77%)	62217 (13.39%)	CaptainEverything...
CaptainEverythingAggregator::GravityWell::on_co...	308083 (66.31%)	33431 (7.20%)	CaptainEverything...
Vector2Wrapper::operator+=	256643 (55.24%)	213811 (46.02%)	CaptainEverything...
std::map<std::thread::id,std::map<void *,BaseDelt...	27541 (5.93%)	27529 (5.93%)	CaptainEverything...
__scrt_common_main_seh	15838 (3.41%)	0 (0.00%)	CaptainEverything...
main	15484 (3.33%)	0 (0.00%)	CaptainEverything...
MasterEngine::LibAggregator::GameEngine::run	14831 (3.19%)	298 (0.06%)	CaptainEverything...
MasterEngine::LibAggregator::ThreadPool::add_job	8291 (1.78%)	484 (0.10%)	CaptainEverything...
operator new	7560 (1.63%)	179 (0.04%)	CaptainEverything...
MasterEngine::LibAggregator::GameEngine::merg...	6731 (1.45%)	4611 (0.99%)	CaptainEverything...
std::_Tree<std::_Tmap_traits<void *,BaseDelta *,st...	4436 (0.95%)	2635 (0.57%)	CaptainEverything...
std::_Tree_comp_alloc<std::_Tmap_traits<void *,B...	3941 (0.85%)	101 (0.02%)	CaptainEverything...
std::_Tree_comp_alloc<std::_Tmap_traits<void *,B...	3840 (0.83%)	247 (0.05%)	CaptainEverything...
__security_check_cookie	3201 (0.69%)	3200 (0.69%)	CaptainEverything...
[External Call] sf::Window::display	2075 (0.45%)	2075 (0.45%)	sfml-window-2.dll

Figure 7.1: profiled result of gravity well test with 600 gravity well.



```
                14  void Vector2Wrapper::operator+=(const sf::Vector2f& rhs)
                15  {
  2618 (0.56%)  16      auto* pointer_deltas = &MasterEngine::LibAggregator::ThreadPool::deltas[std::this_thread::get_id()];
 31086 (6.69%)  17      auto pointer = pointer_deltas->find(this);
180328 (38.82%) 18      if (pointer != pointer_deltas->end())
   815 (0.18%)  19      {
                20          auto delta = static_cast<Vector2Delta*>((*pointer).second);
 26021 (5.60%)  21          delta->addition(rhs);
                22      }
    20 (0.00%)  23      else
                24      {
  3921 (0.84%)  25          auto new_delta = new Vector2Delta{ this };
    50 (0.01%)  26          new_delta->addition(rhs);
  7925 (1.71%)  27          pointer_deltas->insert(std::pair<void*, BaseDelta*>(this, new_delta));
                28      }
  3826 (0.82%)  29  }
```

Figure 7.2: operator+= profiled result of gravity well test with 600 gravity well.

7.6 Real world application

In this section we will discuss if it makes sense to use the aggregator to create games and game engines outside academia. Two factors are important to consider: the usability of the aggregator, and performance of the aggregator compared to alternatives.

In this project we have only focused on whether the aggregator game engine works, and how well it performs compared to a sequential game engine and parallel game engine. In [5, p.49-p.50] we discussed options for testing how easy the aggregator is to use compared to other game engines, but based most of it on speculation as we only had a design to base it on. In this project we have shown that it is possible to create small games using the aggregator game engine. It is essential that using the aggregator is not harder than software transactional memory or revisions, if it is going to be adapted. Many games use forces when interacting with the position of objects, and requires a minimum of changes to use the aggregator instead.

In section 7.5 we discussed options for improving the performance of the aggregator. It is important that the aggregator performs on parallel systems to be usable for adaptation, and in the gravity wells test in section 5.5 we found that the aggregator only performed equally to the sequential game engine. Further improvements have to be made, such that it is a viable choice when programmers and game developers choose their tool for concurrent programming.

7.7 Game

In this section we discuss the game which we designed in section 3.5 and implemented in chapter 4. The game itself was not the focus in the project, but allowed us to show that the same game can be implemented in all three game engines, and be playable. If the game had played or behaved differently in the aggregator game engine compared to the other game engines, it would be a negative to the aggregator, as it would limit the ability to create games, or create game which are not enjoyable. If the aggregator caused stuttering or other problems visible to the player due to the design of the aggregator game engine, it would be unusable for game development. From our personal experience with the game, and also playing the game on the different game engines, we can not feel any difference on how the player or enemies move and behave on the screen. The difference can sometimes be observed in the behavior of background elements, but can be fixed with minor changes to some assignments in the code, which can be changed to relative operations.

8 | Conclusion

In this project we have worked with the idea we call the aggregator, which is based on software transactional memory and first described in [5]. The aggregator has been designed for iterative systems such as game engines, which are used for making computer games. In [5] we ended our work with a design for the aggregator and game engines. In this report we have further described the theory of the aggregator, and compared it to revisions which have many similarities. We have designed three game engines: a sequential game engine, a parallel game engine with locks, and a game engine using the aggregator. We have also designed a game with the intent of showing that all three game engines can be used to implement the same game. All three game engines have been implemented, and the same game implemented for each of the game engines. We have tested the game engines with the goal of comparing the aggregator game engine to the parallel game engine over three tests on four different test systems

Through the tests we have shown that the aggregator game engine is capable of out-performing a sequential game engine by 4.6 times in the best case, but performs worse than a similar parallel game engine using locks. In other tests it performs worse than the single-threaded game engine which we also implemented to be as similar as possible, but we have assumed that is a problem with the implementation which causes that problem.

We have implemented a game for all three game engines, to demonstrate that all three games can be used for creating the same, simple game. The engines have been designed to be as similar as possible, and the game engine interface for the games was identical, except for the features unique for each game engine.

The game engines have been implemented using C++, with the OpenGL wrapper SFML as graphics library. When investigating related works we came across [2], where the rendering library was a limiting factor for the frame rate, and we have chosen a low-level graphics back-end to avoid the same problem.

In the start of the project we asked the question:

How would a game engine implemented using the aggregator compare to game engines using a sequential implementation or a lock-based parallel implementation.

From the tests we have concluded that in some tests the aggregator game engine performs better than the sequential game engine, and in other tests it performs worse than the sequential game engine. It does not at any point in any test perform better than a parallel game engine utilizing locks.

Although not ready yet, our initial tests have shown potential for the aggregator, and with a little more work it should be possible to improve the aggregator game engine to out-perform the sequential game engine, and have a performance close to the parallel game engine.

8.1 Future work

In this section we propose options for future work based on this project. We look at usability testing the aggregator, and implementing the aggregator in an existing game engine. Both topics in this section are also briefly described in [5, p41, p49-p50].

8.1.1 Usability testing

In [5, p41] we included usability testing as a possible option for testing, including a list of possible areas to test. While the tasks listed are few, they cover features beyond what is currently supported in the game engines. For any meaningful usability test with the aggregator in a gaming context, a more mature game engine is needed, which implements many of the features discussed in chapter 6.

The purpose of the usability test will be to document the users ability to create games under the restrictions of the aggregator, to determine if and how the aggregator affects a persons ability to make concurrent computer games. To limit variables in the usability tests, we use both multithreaded game engines for the tests due to their similarity.

We perform two tests with test users: one where we test the aggregator game engine by itself, and one where we compare the aggregator game engine with the parallel game engine. In the test focusing on the aggregator game engine, we will perform an in-depth usability test of the aggregator game engine, testing how the user interacts with many of the different game engine features. In the comparison test we ask the test person to implement a small concurrent game in both game engines, and then comment on which engine they found the easiest to work with and why.

8.1.2 Implement the aggregator in an existing game engine

In this project we have implemented a minimal game engine with the aggregator, but omitted several engine features which can affect the aggregator as explained in chapter 6. An endeaver is to implement the aggregator into another, existing game engine with a more complete feature set.

Starting with a smaller game engine with the features we have omitted, the porting to using the game engine would give insight into several areas:

- Porting the game engine to use the aggregator would give measurements on the time and changes necessary to port the entire game engine, as well as the problems that will be encountered in the process. The numbers can be used to approximate the cost of later porting another game engine to use the aggregator, and determine if the same problems are encountered in ports for different game engines.
- It gives access to larger, complete games implemented in the game engine. Parting the games gives an indication of the cost associated with converting the games, and metrics on the performance difference between the game on the normal game engine and the aggregator game engine.
- It helps finding limitations in the aggregator, as situations will be encountered which are currently not considered in the design or implementation of the aggregator.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Hin Y. Lee Abu Asaduzzaman and Deepthi Gummadi. Impact of thread synchronization and data parallelism on multicore game programming. https://pdfs.semanticscholar.org/533c/18be4bcd951b336d8efcee6d0b8615bdaf2c.pdf, 2014. Online; accessed 2019-03-26.

[2] Alexandro Baldassin and Sebastian Burckhardt. Lightweight software transactions for games. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, 2009.

[3] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *European Symposium on Programming*, pages 116–135. Springer, 2011.

[4] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. *SIGPLAN Not.*, 45(10):691–707, October 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869515. URL http://doi.acm.org/10.1145/1932682.1869515.

[5] Frederik Palmelund Voldby Carsten Schroll Ibsen. Designing a tool for multithreading game engines in a multicore environment, 2018.

[6] Wikipedia contributors. List of game engines. https://en.wikipedia.org/wiki/List%5fof%5fgame%5fengines, 2018. online; accessed 2018-09-07.

[7] Epic Games. Physics simulation. https://docs.unrealengine.com/en-us/Engine/Physics, 2019. online; accessed 2019-05-24.

[8] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7): 33–38, July 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.209.

[9] Johan Hoberg. Differences between software testing and game testing. https://www.gamasutra.com/blogs/JohanHoberg/20140721/221444/Differences%5fbetween%5fSoftware%5fTesting%5fand%5fGame%5fTesting.php, 2014. online; accessed 2019-06-05.

[10] Intel. Intel xeon processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html, 2018. online; accessed 2019-05-16.

[11] Santosh Pande Jaswanth Sreeram. Parallelizing a real-time physics engine using transactional memory. In Jean Roman Emmanuel Jeannot, Raymond Namyst, editor, *Euro-Par 2011 Parallel Processing*, pages 206–223, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23397-5.

[12] Jeremy Laird. Why you don't need more than four cpu cores. https://www.rockpapershotgun.com/2015/03/05/quadcore-gaming/, 2015. online; accessed 2018-12-15.

[13] microsoft. Task-based asynchronous programming. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming, 2019. online; accessed 2019-06-03.

[14] Nvidia. Quadro. https://www.nvidia.com/en-us/design-visualization/quadro/, 2019. online; accessed 2019-05-16.

[15] PCGamer. The problem with growing download sizes. https://www.pcgamer.com/the-problem-with-growing-download-sizes/, 2015.

[16] Peter Sestoft. Microbenchmarks in java and c. *http://academia.edu*, 2015.

[17] Slant. Monogame vs sfml. https://www.slant.co/versus/1069/1100/~monogame%5fvs%5fsfml, 2019. online; accessed 2019-06-03.

[18] Unity Technologies. Execution order of event functions. https://docs.unity3d.com/Manual/ExecutionOrder.html, 2018. online; accessed 2018-12-06.

[19] Unity Technologies. Gameobject. https://docs.unity3d.com/ScriptReference/GameObject.html, 2018. online; accessed 2018-10-30.

[20] Unity Technologies. Monobehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html, 2018. online; accessed 2018-12-19.

[21] Unity Technologies. Transform. https://docs.unity3d.com/Manual/class-Transform.html, 2018. online; accessed 2018-10-30.

[22] Unity Technologies. Scripting api: Collider. https://docs.unity3d.com/ScriptReference/Collider.html, 2019. online; accessed 2019-05-21.

[23] Unity Technologies. Scripting api: Monobehaviour.fixedupdate(). https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html, 2019. online; accessed 2019-05-21.

[24] Unity Technologies. Announcing unity and havok physics for dots. https://blogs.unity3d.com/2019/03/19/announcing-unity-and-havok-physics-for-dots/, 2019. online; accessed 2019-05-24.

[25] Unity Technologies. Scripting api: Input. https://docs.unity3d.com/ScriptReference/Input.html, 2019. online; accessed 2019-05-29.

[26] Unity Technologies. Scripting api: Resources. https://docs.unity3d.com/ScriptReference/Resources.html, 2019. online; accessed 2019-05-21.

[27] theburningmonk. Threading – using the threadpool vs. creating your own threads. https://theburningmonk.com/2010/03/threading-using-the-threadpool-vs-creating-your-own-threads/, 2010.

[28] Real time audio programming 101: time waits for nothing. Ross bencina. http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing, 2019. online; accessed 2019-06-03.

[29] Unity. Introduction to components. https://docs.unity3d.com/Manual/Components.html, 2019.

[30] Unity. Mesh collider. https://docs.unity3d.com/Manual/class-MeshCollider.html, 2019. online; accessed 2019-05-17.

[31] Nail Valimatov. What is the difference between intel core generations? https://www.quora.com/What-is-the-difference-between-Intel-core-generations, 2018. online; accessed 2019-05-16.

[32] Jarred Walton. The best cpu for gaming. https://www.pcgamer.com/best-cpu-for-gaming/, 2018. online; accessed 2018-10-04.

[33] Andrew Wesson. Do video game engines use actual physics equations in their algorithms? https://www.quora.com/Do-video-game-engines-use-actual-physics-equations-in-their-algorithms, 2016. Online; accessed 2019-03-11.

A | Test results

Single Core					
Desktop #2	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	5769,574	3075,17	1349,516	53,29977569	23,3902191
300	1041,724	889,2608	206,3056	85,36433835	19,80424757
1000	101,8888	120,2096	60,27936	117,9811716	59,16190985
Desktop #1	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	10664,34	921,938	246,721	8,645054453	2,31351401
300	1331,736	1229,416	96,75044	92,31679552	7,264986454
1000	120,0816	154,3384	83,62468	128,5279343	69,63987822
Laptop #1	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	469,7316	309,2642	126,8676	65,8384916	27,00852998
300	323,5036	288,421	16,34408	89,15542207	5,05220962
1000	63,44476	80,61966	7,698303	127,0706359	12,13386732
Laptop #2					
Laptop #2	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	1789,112	960,236	426,9854	53,67109493	23,86577252
300	756,4234	694,846	220,814	91,85940044	29,19185208
1000	87,6383	110,6408	73,33512	126,247086	83,6793046
Background elements					
Desktop #2	Sequential	Parallel	Aggregator	Parallel %	Aggregator %
30	5575,384	4131,818	2989,09	74,10822286	53,61227137
300	1186,064	1548,796	483,3124	130,5828353	40,74926817
1000	104,461	405,9888	163,8692	388,6510755	156,8711768
2000	19,68834	133,9934	69,71346	680,5723591	354,0850067
Desktop #1	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	7347,316	5400,236	3768,184	73,49943843	51,28653783
300	1464,652	1973,97	959,543	134,7739941	65,51337792
1000	116,7522	554,0824	290,6334	474,5798366	248,9318403
2000	23,313	164,161	107,4906	704,1607687	461,0757946
Laptop #1	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	478,7054	434,9032	347,8908	90,84986298	72,67325583
300	335,456	374,4402	332,6872	111,6212558	99,17461605
1000	63,90102	164,7342	119,1386	257,7958849	186,4424073
2000	15,87852	56,65912	44,59214	356,8287221	280,8331003
Laptop #2	Sequential	Parallel	Aggregator	Parallel %	Aggregator%
30	1830,076	940,0054	580,8516	51,36428214	31,73920646
300	767,558	936,8056	653,9026	122,0501382	85,19259782
1000	87,79652	207,7072	160,4056	236,5779418	182,7015467
2000	18,50752	61,56184	49,96824	332,6314925	269,9888478

Gravity Wells					
Desktop #2	Sequential	Parallel	Aggregator	Parallel %	Aggregator %
0	4161,486	1869,226	1172,384	44,91727234	28,17224424
1	4052,5	1865,076	1130,976	46,02285009	27,90810611
5	3920,126	1936,432	1112,194	49,39718774	28,37138398
10	3889,294	2019,976	1040,58	51,93682967	26,75498432
25	3331,914	2271,684	927,5694	68,17955085	27,83893582
100	1784,288	1472,436	615,3088	82,52232823	34,48483653
250	534,04	646,9034	323,2978	121,1338851	60,53812449
300	397,8212	519,7	271,6538	130,6365774	68,28540058
350	310,7518	428,7548	234,127	137,9733923	75,34212191
400	248,9426	359,7888	199,1536	144,5268106	79,99980718
450	205,5234	305,039	172,6232	148,4205691	83,99199313
500	170,97	262,1798	153,2696	153,3484237	89,64707259
550	144,7014	227,8546	134,3766	157,4653735	92,86475459
600	124,3686	198,9354	120,4568	159,9562912	96,85467232
Desktop #1	Sequential	Parallel	Aggregator	Parallel %	Aggregator %
0	8764,834	3830,66	2085,316	43,70487792	23,7918482
1	8614,236	3759,312	1929,406	43,64068967	22,39787719
5	8344,502	3645,37	1874,828	43,68589042	22,46782372
10	8286,856	3585,634	1817,704	43,26893094	21,93478443
25	6736,298	3247,674	1705,55	48,21155477	25,3188027
100	2423,98	2197,64	1124,128	90,66246421	46,37530013
250	735,269	950,7076	552,7952	129,3006505	75,18271544
300	551,4168	747,3652	428,509	135,5354425	77,71054491
350	433,0666	616,8166	341,5716	142,4299634	78,8727646
400	345,1898	519,4074	294,9314	150,4700892	85,44035774
450	282,5538	436,2644	256,0904	154,4004717	90,63420842
500	238,0358	374,5812	216,4718	157,3633882	90,94085848
550	202,71	324,4342	190,7982	160,0484436	94,12372355
600	172,269	281,8884	166,6164	163,6326907	96,71873639
Laptop #1	Sequential	Parallel	Aggregator	Parallel %	Aggregator %
0	418,9596	373,3662	264,3376	89,11747099	63,09381621
1	472,8486	380,2944	405,2612	80,4262506	85,70633391
5	467,4008	387,7488	418,7118	82,95852296	89,58303024
10	452,7318	391,187	414,821	86,40590301	91,62621225
25	456,6296	398,9518	407,4126	87,36879957	89,22167989
100	401,025	377,387	329,2564	94,10560439	82,10370925
250	243,694	236,0262	184,5674	96,85351301	75,73735915
300	204,8834	198,8854	153,5164	97,07248123	74,92866674
350	177,9926	160,1368	128,9766	89,96823463	72,4617765
400	150,995	130,8904	109,7264	86,68525448	72,66889632
450	130,1502	112,5008	93,97992	86,43920639	72,2088172
500	115,5462	95,35674	80,62034	82,52693728	69,77325087

550	100,26748	81,3411	69,24654	81,12410923	69,06181346
600	88,20348	68,55578	60,84316	77,72457504	68,98045293
Laptop #2	Sequential	Parallel	Aggregator	Parallel %	Aggregator %
0	1638,708	1037,7232	297,3548	63,32569317	18,1456855
1	1583,198	1106,592	362,8568	69,89599532	22,91923057
5	1570,054	1183,666	416,5776	75,39014582	26,53269251
10	1543,302	1178,064	428,954	76,33399037	27,79455998
25	1482,036	1118,538	568,6104	75,47306543	38,36684129
100	1107,486	768,2782	630,1186	69,37136903	56,89630388
250	448,4456	337,7054	259,5244	75,3057673	57,87199161
300	354,0894	267,1814	202,794	75,45591594	57,27197708
350	283,8884	215,5722	161,1154	75,93554369	56,75307621
400	235,504	177,036	131,4308	75,17324547	55,80830899
450	195,0412	147,604	110,0644	75,67836949	56,43135912
500	166,2982	123,5582	93,22114	74,29918063	56,05661396
550	142,633	106,2574	78,66852	74,4970659	55,15450141
600	122,7606	90,88034	67,7138	74,030544	55,15922861